Arithmetic Data Value Speculation

A thesis submitted to

the School of Electrical and Electronic Engineering

of the University of Adelaide

by

Daniel R. Kelly

B.E. (Computer Systems)(Hons),

B.Sc. (Maths & Computer Science)

in fulfillment of the requirements for

the degree of Doctor of Philosophy (Microelectronics)

Commenced, April 2005

Thesis Submitted, August 2010

Declaration of Originality

This work contains no material that has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of the thesis, when deposited in the University Library, being available for loan, photocopying, and dissemination through the library digital thesis collection, subject to the provisions of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library catalogue, the Australasian Digital Thesis Program (ADTP) and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Daniel R. Kelly
School of Electrical and Electronic Engineering
The University of Adelaide
September 13, 2010

Acknowledgements

I wish to acknowledge my principal supervisor, Dr. Braden J. Phillips, for all of his help and guidance in all the things that mattered, including: identifying a stimulating research project; establishing my scholarship that made the project possible; being flexible enough to allow my sometimes backwards approach but firm enough to reign me in where needed; and an attention to detail and conviction in the importance of good, clear writing (if in coming pages you disagree, I am happy to award him blame or credit ;P).

My co-supervisor Dr. Said Al-Sarawi has been invaluable in providing a very focused, objective critique—with an attention to detail and response time that was amazing. I am grateful for the time and resources that Said has committed to me when needed, above and beyond the call of duty.

Dr. Andrew Beaumont-Smith has been extremely generous in providing an industry perspective and finding opportunities to work in the job of a lifetime. Andrew's influence on me has been to contextualise my research and to change my perspective in a way that no one else has. *"Just get your PhD ASAP"*.

Fiona Tselentis has been behind the scenes gently pushing me along and helping in so many ways that she doesn't even know about.

Finally I would like to thank my parents for their support and encouragement. This thesis would not have been possible without them.

This project has been a rewarding and challenging undertaking. I will never, ever do one again. Without further ado, I present my thesis, warts and all.

Daniel R. Kelly
September 13, 2010

Abstract

Arithmetic approximation is used to decrease the latency of an arithmetic circuit by shortening the critical path delay or the sampling period so that result is not guaranteed to be correct for every input combination. Thus, an acceptable compromise between the circuit latency and the average probability of correctness drives the circuit design. Two methods of arithmetic approximation are:

temporally incompleteness where circuits quote the result before the critical path delay (overclocking); and

logically incompleteness where circuits use simplified logic, so that most input cases are calculated correctly, but the slowest cases are calculated incorrectly.

Arithmetic data value speculation (*ADVS*) is a speculation scheme based on arithmetic approximation, and is used to increase the throughput of a general purpose processor. *ADVS* is similar to branch prediction, an arithmetic instruction is issued to an exact arithmetic unit and an approximate arithmetic unit which provides an approximate result faster than the exact counterpart. The approximate result is forwarded to dependent operations so they may be speculatively issued. When the exact result is eventually known, it is compared to the approximate result, and the pipeline is flushed if they differ.

This thesis, *"Arithmetic Data Value Speculation"*, presents work in the field of digital arithmetic and computer architecture. A summary of current probabilistic arithmetic methods from the literature is provided, and novel designs of approximate integer arithmetic units are presented, including results from logical synthesis. A case study demonstrates approximate arithmetic units used to increase the average throughput of benchmark programs by speculatively issuing dependent operations in a RISC processor.

The average correctness of the approximate arithmetic units are shown to be highly data dependent, results vary depending on the benchmarks being run. In addition, the average correctness when running benchmarks is consistently higher than for random inputs. Simulations show that many arithmetic operations are often repeated in the same benchmark, leading to a high variation in correctness. Speculative gains from one operation can be offset by speculation losses due repeated incorrect approximation of another approximate unit, so typical throughput gains through speculation in a general purpose processor pipeline are low. The minimum threshold correctness of an approximate arithmetic unit used for speculation is shown to be approximately 95%. Logic synthesis is used to determine power, area and timing information for approximate units implemented from novel algorithms, and show a reduction in arithmetic cycle latency for integer operations, and the expense of 50 % leakage and area, and 90 % dynamic power.

Value speculation can be complemented by result caching; repeated pipeline flushes can be avoided if the correct result is know before speculation, the average operation latency can be reduced, and caching can be used for operations that are difficult to approximate.

Publications

The following is a list of publications published or submitted during the Ph.D. candidature by the author.

Published book chapters

1. Daniel R. Kelly and Braden J. Phillips, "Arithmetic data value speculation", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3740 NCS, pp. 353–366, 2005.

2. Daniel R. Kelly, Braden J. Phillips and Said Al-Sarawi, *Algorithm-ArchitectureMatching for Signal and Image Processing*, volume 73 of *Lecture Notes in Electrical Engineering*, chapter 4 "Operators for embedded systems: Approximate Multiplication and Division for Arithmetic Data Value Speculation in a RISC Processor". Springer, 1st edition, December 2010. ISBN: 978-90-481-9964-8.

Published conference papers

3. Daniel R. Kelly, Braden J. Phillips and Said Al-Sarawi, "An open source synthesisable model in VHDL of a 64-bit MIPS-based processor", in *Proceedings of SPIE—The International Society for Optical Engineering*, vol. 6414, (Adelaide, Australia), 2007.

4. Braden J. Phillips, Daniel R. Kelly and Brian W. Ng, "Estimating adders for a low density parity check decoder", in *Proceedings of SPIE—The International Society for Optical Engineering*, vol. 6313, (San Diego, CA, United States), 2006.

5. Braden J. Phillips, Cain D. Schmidt and Daniel R. Kelly, "Recovering data from USB flash memory sticks that have been damaged or electronically erased", in *e-Forensics '08: Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop*, (ICST, Brussels, Belgium, Belgium), pp. 1–6, ICST (Institute for Computer Sciences, Social Informatics and Telecommunications Engineering), 2008.

6. Daniel R. Kelly, Braden J. Phillips and Said Al-Sarawi, "Approximate unsigned binary integer dividers for arithmetic data value speculation". In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Sophia Antipolis, France, September 22–24, 2009. `http://www.ecsi-association.org/ecsi/dasip/dasip09`.

7. Daniel R. Kelly, Braden J. Phillips and Said Al-Sarawi, "Approximate signed binary integer multipliers for arithmetic data value speculation". In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Sophia Antipolis, France, September 22–24, 2009. `http://www.ecsi-association.org/ecsi/dasip/dasip09`.

8. Daniel R. Kelly, Braden J. Phillips and Said Al-Sarawi, "Increasing throughput of a RISC system using arithmetic data value speculation". In *Conference Record of the Forty-Third Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, USA, November 1–4, 2009.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ADPCM	adaptive differential pulse code modulation
AWCCL	average worst-case carry length, in bits
AWGN	additive white gaussian noise
CCITT	International Telegraph and Telephone Consultative Committee
CMOS	complementary metal oxide semiconductor
DFT	discrete fourier transform
FP	floating point
EPIC	Efficient Pyramid Image Coder
FA	full adder
FFT	fast fourier transform—an efficient algorithm to compute the discrete fourier transform (DFT) and its inverse.
GSM	Global System for Mobile communications: originally from Groupe Spécial Mobile
HA	half adder
IEEE	Institute of Electrical and Electronic Engineers
INT	integer
ISA	instruction set architecture
JPEG	Joint Photographic Experts Group

List of Algorithms

LSB	least significant bit
MPEG	Moving Picture Experts Group
MSB	most significant bit
MUX	Multiplexor
SNR	signal-to-noise ratio
QPSK	quadrature phase-shift keying
PGP	Pretty Good Privacy
SPEC	Standard Performance Evaluation Corporation
SPHERE	SPeech HEader REsources
ULP	Unit of least precision

Nomenclature

Mathematical notation

$f(n) \in O(n)$	$f(n) \leq g(n) \cdot k$, f is bounded above by g (up to constant factor) asymptotically
$f(n) \in \Omega(n)$	$f(n) \geq g(n) \cdot k$, f is bounded below by g (up to constant factor) asymptotically
$f(n) \in \Omega(g(n))$	$g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$, f is bounded both above and below by g asymptotically
$\lfloor x \rfloor$	floor(x), greatest integer $\leq x$
$\lceil x \rceil$	ceiling(x), least integer $\geq x$
P_M	probability of correctness for a multiplier
P_C	probability of correctness for a counter

Standard notation

N	Number of bits in an operand; word length
a_N	Average worst-case carry length for N bit addition
l	Length in bits of a carry-chain in an adder
PC	Program Counter register
SP	Stack Pointer register

Multiplication

n	number of input bits to a counter

m	number of output bits from a counter

Division

z	dividend
d	divisor
q	exact quotient after division $q = z/d$
d_H	least binary power greater than or equal to d
d_L	greatest binary power less than or equal to d
$\tilde{d}$	approximate divisor
$\tilde{D}$	$\log_2(d)$, or the number of zeros after the MSB of d
q_i	quotient after round i
r_i	remainder after round i
e_i	error in q_i after round i
n	number of bits in the unsigned binary representation of z and d
k	number of bits in the quotient, q
t	maximum number of division rounds
f	number of fractional bits maintained in q_i
m	constant multiplicand in the division round: $m = d - \tilde{d}$
p_1	most significant non-zero partial product in the binary multiplication $r_i \times m$
p_2	after p_1, the next most significant partial product in the binary multiplication $r_i \times m$

Chapter 1

Introduction

"Any effort that has self-glorification as its final endpoint is bound to end in disaster."

Robert M. Pirsig (1928 —), in
Zen and the Art of Motorcycle Maintenance:
An Inquiry Into Values

This chapter introduces approximate arithmetic and arithmetic data value speculation (*ADVS*), and defines the goals of the research project: to design approximate arithmetic units, profile their design tradeoffs, and investigate their implementation in a speculative scheme to increase the throughput of a processor pipeline. An overview of the remaining chapters is presented, and the main contributions of this thesis are briefly summarised.

Modern processors are complex machines. Transistor counts on processors have increased exponentially from 2300 transistors on the Intel 4004 in 1971, to over two billion on the latest upcoming Intel Itanium processors in 2009 [Intel Corp., 2009]. Over the past 38 years many technologies and design paradigms have been tested, with CMOS emerging as the dominant technology and high performance processors having moved towards multiple cores, very long instruction word (VLIW) instructions and hardware support for multiple threading [NVIDIA Corp., 2009]. Speculation techniques such as branch prediction have matured and become pervasive, although many other forms of speculation have not.

Power dissipation has also increased, and become a first order design constraint for many designs. Coupled with the ubiquity of mobile computing and “good-enough” computing, power consumption has forced engineers to maximise efficiency and carefully consider all aspects of their designs to achieve their design goals. Customer expectations can be markedly different in various markets, and it is often price, not performance that is the primary concern of consumers.

Researchers and manufacturers have begun to consider approximate and probabilistic computing to extend battery life (or cost for the same performance) to satisfy the *good-enough*, low performance market. Simultaneously, value speculation has remained largely uncommon in high performance designs to increase performance by breaking the data flow limit.

Approximate hardware provides a result faster than an exact unit, and has applications to designs at both ends of the performance spectrum. A low performance design might accept the potential errors, or be tolerant of them to some threshold; a high performance design can use the approximate result to speculatively execute instead of idly waiting for the exact result to become available.

1.1 Research objective

Approximate hardware is characterised by its probability of correctness—the chance that the approximate hardware will provide the correct result—and the delay required for the output bits to be asserted, from the time the inputs are provided. The delay of the approximate circuit should be less than the delay of an exact arithmetic circuit.

The research objectives of this thesis are:

- to investigate the trade-off between arithmetic latency and probability of correctness for a number of fundamental modules such as integer multipliers and dividers;
- to adapt these modules to IEEE floating point units and investigate the effect on probability of correctness and delay;
- to demonstrate the feasibility of these approximate modules in a speculative execution scheme in a RISC processor pipeline, and the effect on the retired instructions per clock for benchmarks programs.

For this thesis approximate arithmetic hardware for each of the fundamental integer operations of addition, subtraction, multiplication and division was designed. The new hardware units were profiled in terms of their delay and probability of correctness for traced benchmark operands. Each feasible approximate integer unit was also integrated into an IEEE floating point design. This research seeks to determine if approximate arithmetic hardware can be utilised in a processor pipeline in a speculative execution scheme to increase throughput.

1.2 Research outcome

This thesis demonstrates that ADVS can be successfully employed in a processor pipeline to increase throughput. As a caveat, it is noted that the proposed system modifications are expensive in terms of area and power. The area, power, and correctness of each component is reported individually, and on a per-benchmark basis so that an implementor can balance the potential gains versus the cost.

The designs for approximate arithmetic hardware units arising from this thesis have been carefully characterised, and can be adapted to use in error tolerant systems where additional hardware is not required to correct errors. In fact, it is shown that the approximate units designed are often smaller, faster and consume less power.

1.3 Thesis outline

This thesis explores the use of approximate arithmetic for speculative execution in high performance computing, and is presented in 4 parts.

Preliminary matter is covered in Part I *Background*.

Chapter 2 *A Brief Review of Computer Architecture and Digital Arithmetic* contains an overview of digital arithmetic and computer architecture. The fundamental integer operations of addition, subtraction, multiplication and division are discussed, along with their application in floating point calculations. The platform and tools used for the main investigation of this thesis are discussed: the *PISA* instruction set architecture (ISA) determines the types and format of arithmetic operations; the *SimpleScalar* tool set was used to model and trace execution; and a variety of benchmarks are selected.

Chapter 3 *Theory and Applications of Arithmetic Approximation* reports results and designs of approximate arithmetic units from the literature that are used in current methods of probabilistic computing and arithmetic approximation. Some preliminary simulations were conducted with random and benchmark data, to assess the probability of correctness of these units.

Chapter 4 *Can ADVS Improve the Performance of a Generic RISC Processor?* defines the scope of the research project, establishing that this thesis sets out to determine the feasibility of using arithmetic data value speculation to improve the throughput of a modified RISC processor.

Chapter 5 *Preliminary Experiments* presents the results of simulations where simple arithmetic operations were modified to produce a result quickly, but with a probability of error. It is shown that it is difficult to yield a high probability of correctness when the critical path is shortened, because of the high degree of dependence that intermediate results have on input operands in arithmetic operations.

Part II *Approximate Arithmetic* presents designs for approximate arithmetic units.

Chapter 6 *Approximate Integer Multiplication* presents a design for an unsigned approximate integer multiplier, which was extended to accommodate signed operands. The new approximate multipliers were based on modern high performance tree multipliers built from counter units. Logically incomplete counters were used as the fundamental unit for a family of multipliers that had a reduced latency and probability of correctness compared to a baseline Wallace tree multiplier constructed from exact full-adder cells.

Chapter 7 *Approximate Integer Division* presents an iterative division algorithm in which the quotient was calculated by convergence, with a variable number of quotient digits calculated per iteration. The latency of the divider was reduced by reducing the precision of the intermediate calculations, and restricting the number of division rounds. The new division algorithm was sufficient to calculate the integer (but not necessarily fractional) bits of the quotient result for operands typically observed in benchmark programs, with a lower latency than a similar variable-cycle radix-4 SRT divider.

Chapter 8 *Approximate Floating Point Arithmetic* adapts the approximate integer adder, multiplier and divider units developed previously to IEEE floating point designs. The delay, area, power and probability of correctness of the floating point units are compared to a baseline implementations, and their suitability for *ADVS* is assessed.

Chapter 9 *Result Caching* investigates the caching of arithmetic results to further reduce the average effective latency of arithmetic operations, including operations where no feasible approximate hardware exists. Indexing schemes were developed to better distribute entries within the cache. Replacement policies from the literature were used to introduce set associativity, and improved the hit rate over direct mapped result caches.

Part III *Application* focuses on the application and implementation of approximate arithmetic.

In Chapter 10 approximate adders are demonstrated in an error tolerant application—low density parity check decoding, where it is found that approximate compressors can yield a saving to area, power and delay, as well as decreasing the frame error rate and average number of decoding iterations.

Chapter 11 adapts the hardware developed in Chapters 6, 7, 8 and 9 for implementation in a processor pipeline. The latencies of each approximate arithmetic unit were compared to each other and their baseline exact systems to determine the operation latency in cycles. A range of arithmetic benchmarks were simulated running on a modified RISC processor with result caching and arithmetic data value speculation. The cost of ADVS was assessed in terms of the increased area and power due to the exact systems used for result checking.

In Chapter 12 *Conclusions*, a brief summary of the designs and simulation outcomes in this thesis are presented and discussed, with an emphasis on the engineering trade offs necessary to implement ADVS, and uses of approximate arithmetic in general. Future research avenues are identified.

Appendices are provided in Part IV, and provide excerpts from documentation used throughout the project, samples of source code written by the author to generate results, and extended tables of results summarised in earlier chapters.

1.4 Research Contributions

This thesis makes the following contributions:

1. an analysis of typical arithmetic operands in benchmark software;
2. an extended investigation of existing designs for approximate integer adders and subtractors;
3. designs and synthesis results of approximate arithmetic hardware for signed and unsigned integer multiplication and division;
4. an analysis of approximate arithmetic adapted for IEEE floating point multipliers and dividers;

5. an in depth study of caching arithmetic results to improve performance;
6. the use of inexact arithmetic to reduce the latency of a low density parity check decoder; and
7. simulation results that define upper and lower bounds for performance improvement by employing ADVS in a RISC processor, and a simulated implementation incorporating the approximate arithmetic designs proposed.

PART I

BACKGROUND

Chapter 2

A Brief Review of Computer Architecture and Digital Arithmetic

“Speed isn’t everything, it’s the only thing.”

Seymour Cray (1925–1966)

This chapter contains an overview of digital arithmetic and computer architecture. The fundamental integer operations of addition, subtraction, multiplication and division, and their application in floating point calculations are summarised. The tool sets, libraries and benchmarks used for the project are introduced, and a summary of the literature is presented.

The aim of this thesis is to test the feasibility of increasing of the throughput of a RISC processor by speculating on the result of arithmetic instructions that are calculated quickly, but with a probability that the result is incorrect. This arithmetic data value speculation (*ADVS*) scheme requires the design of approximate arithmetic units to provide results faster than their exact counterparts.

In the following sections a summary of tools used consistently throughout the thesis is presented. Section 2.1 presents a brief review of digital arithmetic, focusing on integer and floating point representations and the fundamental operations of addition, subtraction, multiplication and division. The tool set *SimpleScalar* is presented in Section 2.2. These tools are used to trace arithmetic operations from benchmark programs, and simulate the operation of a regular RISC processor, and one that uses *ADVS*. In Section 2.3 a collection of benchmarks are presented that are used to measure the average correctness of arithmetic units, and throughput gains when used in a pipeline using *ADVS*. Finally, Section 2.4 summarises the tools used to synthesise the digital arithmetic circuit designs used in this project.

2.1 Digital arithmetic

This section provides an overview of digital arithmetic, focusing on number representations typically found in modern computers, including signed and unsigned integer arithmetic, and *IEEE-754* floating point arithmetic.

The hardware designs in this thesis were developed for a 32 bit RISC processor with a *MIPS* ISA. Hence the designs and results are based almost exclusively on 32 bit operands. Most arithmetic operations are binary, and calculate one 32 bit result from two inputs. Square root is a unary operation, but is not addressed in this project. Integer multiplication produces a 64 bit result, and integer division produces a 32 bit quotient and remainder. Both operations are defined to write to special architectural registers, instead of the normal integer register file. Later in this thesis, an approximate divider is developed, but is only capable of generating the quotient result. The remainder was found to be used rarely after a division, and even less frequently was both the quotient and remainder used.

However, the approximate divider can still be used in this processor, despite the absence of the remainder—in this pipeline the latency required to calculate and move the remainder to the main register file is the same as if only the quotient was calculated and written directly to the register file, and the remainder calculated by a subtraction.

Signed and unsigned versions of all integer arithmetic operations are available in the ISA, and approximating hardware was developed for both. Signed integers use a twos complement representation.

Although single and double precision *IEEE-754* floating point is a part of the ISA, only 32 bit formats were addressed. In most cases, benchmarks could be compiled using a fixed precision, but in cases where they could not, the less frequent double precision operations were not approximated. Single precision floating point numbers consist of a 1 bit sign, an 8 bit exponent, and a 23 bit significand that is generally extended to include a hidden bit and possible additional bits to maintain precision for rounding. Approximate floating point hardware was devised by scaling approximate 32 bit integer hardware down to 24 or more bits for the significand.

2.2 SimpleScalar

SimpleScalar is a tool set to perform simulations of processors that implement the *SimpleScalar* architecture (a close derivative of the *MIPS III* architecture). It includes a modified version of *gcc* version 2.6.3 to cross compile *C* programs for a *SimpleScalar* target [Burger and Austin, 1997]. *SimpleScalar* was used as the target platform for all benchmarks in this thesis.

The *SimpleScalar* tool set contains several special purpose simulators. Each simulator and its function is listed below.

`sim-bpred` Implements a branch predictor analyser.

`sim-cache` Simulates cache access patterns for various replacement schemes.

`sim-eio` Generates an external event trace (EIO trace) for later re-execution. Includes support for checkpointing.

`sim-fast` Executes instructions serially. Does no time accounting and assumes no cache.

`sim-safe` Same as *`sim-fast`*, but also checks alignment and access permissions for each memory reference.

`sim-profile` Implements a functional simulator, with profiling support.

`sim-outorder` Implements a very detailed out-of-order issue superscalar processor with a two level memory system and speculative execution support. This simulator is a performance simulator, tracking the latency of all pipeline operations.

2.2.1 System components

This section describes some of the system level components implemented in the *SimpleScalar* simulator. The ISA used is a *MIPS*-like hypothetical architecture, including floating point and atomic memory synchronisation primitives.

2.2.1.1 Registers

The processor has 32 general purpose registers, and a floating point coprocessor for *IEEE-754* floating point arithmetic that has 32×32 bit registers. 64 bit double precision floating point operands are held in adjacent even-odd indexed registers, reducing the effective number of registers available.

Two special registers called *HI* and *LO* store the result of arithmetic operations that require a double precision result, or two result fields. The `intMult` instruction calculates the 64 bit product of two 32 bit operands, and the `intDiv` instruction calculates the 32 bit quotient and 32 bit remainder from two 32 bit operands.

2.2.1.2 Branch predictor

The branch predictor in the *SimpleScalar* simulator can be set to one of several different types, including predict-taken, predict-not-taken, perfect, bimodal, two-level and combination of bimodal and two-level predictors. The branch predictor history table defaults to 1024 entries, despite the configuration.

The return address stack defaults to 8 entries. The branch target buffer has 512 entries and 4 way set associativity. The branch predictor history table is not updated with the result of an instruction executed on the speculative path. The default 1024 entry *bimodal* predictor was used in simulations presented in this thesis.

2.2.1.3 Cache

SimpleScalar uses a Harvard architecture, with separate instruction (`iCache`) and data caches (`dCache`). The 16 kB level 1 `dCache` is a 128 set, 32 block, 4 way associative, least recently used (*LRU*) replacement cache. The `dCache` has a 1 cycle hit latency. The 16 kB level 1 `iCache` is a 512 set, 32 block, 1 way associative (direct mapped), *LRU* replacement cache. The `iCache` and `dCache` have a 1 cycle hit latency.

The 256 kB level 2 cache is unified instruction and data cache, with 1024 sets, 64 blocks, 4 way set associativity, and an *LRU* replacement policy. The level 2 cache has a 6 cycle hit latency.

2.2.1.4 Memory

A translation look-aside buffer (*TLB*) is maintained for the `iCache` and `dCache` to cache physical memory addresses for the memory paging system. The instruction translation look-aside buffer (`iTLB`) is a 16 set, 4096 block, 4 way associative, *LRU* replacement buffer. The data *TLB* (`dTLB`) is a 32 set, 4096 block, 4 way associative, *LRU* replacement buffer. Both *TLB*s have a miss latency of 30 cycles.

The memory uses an 8 byte (64 bit) bus. The initial access latency is 18 cycles, with 2 cycles for each additional 8 byte chunk.

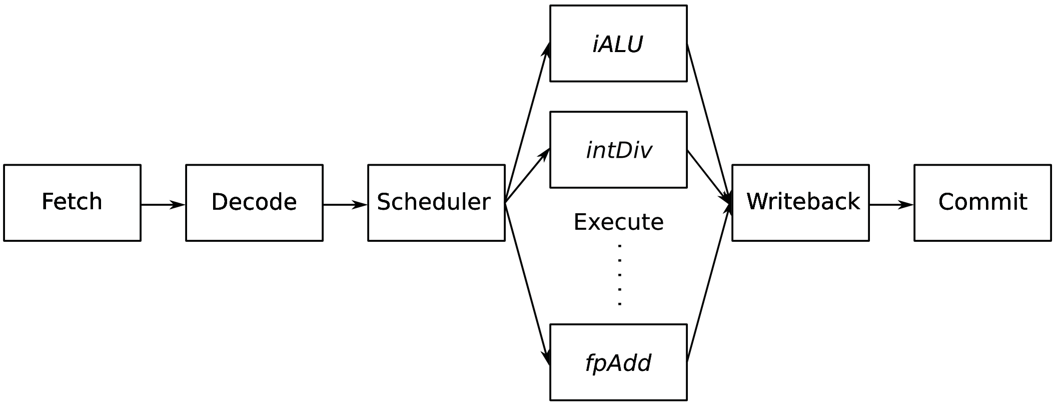


Figure 2.1: Overview the of *SimpleScalar PISA* pipeline. In the execution stage, instructions are executed in individual functional units that might have a multi-cycle latency, before continuing to the write-back stage.

2.2.2 Pipeline overview

The *SimpleScalar PISA* pipeline is similar to the classic *MIPS I* architecture, but the instruction set architecture (ISA) is closer to the *MIPS III* ISA, including exclusive memory operations, and floating point support. A list of instructions and opcodes for the *PISA* architecture is provided in Appendix F.

The *SimpleScalar* pipeline has 6 stages. Every instruction is either a memory operation, or register-register operation. There are no instructions in the architecture that compound both. These operations are distinct, on the premise that they are fast. Figure 2.1 shows the basic instruction flow in the processor pipeline, from fetch to commit. Instructions issue to dedicated functional units for the execution stage, such as one of the *ALU*s or `intDiv` units shown.

In the *PISA* architecture, the pipeline stages are:

Fetch Up to 4 instructions are fetched from one *iCache* line and are inserted into the dispatch queue.

Decode Instructions on the dispatch queue are decoded and register renaming is performed. Decoded instructions are inserted on the scheduler queue.

Scheduler Instructions on the scheduler queue that have all source operations available are issued to functional units, including arithmetic units, and inserted in the Register Update Unit (*RUU*), a queue that maintains program order until commit. Memory operations are also added to the Load Store Queue (*LSQ*) for in-order memory access.

Execute The functional units execute ready instructions issued to them. Load operations access the *dCache*.

Write-back Results from completed instructions in the *RUU* are written back to memory, and the dependents of completed instructions are marked as ready.

Commit Completed instructions are committed in-order in the *RUU* to maintain program semantics, updating the *dCache* and memory.

SimpleScalar is configurable to allow testing with a number of different hardware settings, including cache size, instruction latency, branch prediction algorithm, etc. The *SimpleScalar* configuration used in this thesis is shown in Table 2.1.

The register update unit (*RUU*) maintains program order for out-of-order execution and holds up to 16 instructions. The load-store queue (*LSQ*) maintains memory ordering and holds up to 8 instructions. The front end can fetch up to 4 instructions in each stage, and up to 4 instructions can be issued and committed per cycle, depending on availability of functional units and memory alignment. The memory subsystem has 1 read port and 1 write port.

The size of the *RUU* and the *LSQ* are important to overall system throughput, and must be balanced with the processor frequency, pipeline depth and operation latency. Longer queues allow more instructions "in flight" at any time, and in an out-of-order processor more younger independent operations may be issued and executed if an older, high latency operation is causing a stall.

2.2.3 Arithmetic operation latencies

In *SimpleScalar* all *ALU* operations are single cycle, except for some arithmetic instructions. Additions, subtractions, logic and load/store operations are performed in *ALU*s in the Execute stage (*iALU*) and floating point coprocessor (*fpALU*). There are dedicated units for integer multiplication (intMult), integer division (intDiv), and floating point multiplication (fpMult). There is also a combined unit for floating point division (fpDiv) and square root (fpSqrt). There are 4 *iALU*s and *fpALU*s, and one each of the other arithmetic units.

The latency and repeat-rate of the arithmetic operations in *SimpleScalar* is listed in Table 2.2. In the original version of *SimpleScalar*, the integer multiplier and divider are combined into a single unit,

Table 2.1: Configuration used in *SimpleScalar*.

Characteristic	Description
Instruction fetch	Up to 4 instructions per cycle, across multiple branches and cache line boundaries, stopping on an *iCache* miss
Instruction cache (*iCache*)	16 kB direct mapped 32 B lines, 6 cycle latency
Branch predictor	2048 entry, 4 way set associative bimodal predictor
Speculative execution	Out-of-order issue up to 4 operations per cycle, 16 entry reorder buffer, 8 entry load-store queue with store/load bypassing
Architectural registers	32 general purpose and 32 floating point
Functional unit latency	*iALU* 1/1, `intMult` 3/1, `intDiv` 20/19, `fpAdd` 2/1, `fpMult` 4/1, `fpDiv` 12/12 `fpSqrt` 24/24
Arithmetic	32 bit word and 32 bit address, little endian
Data cache (*dCache*)	16 kB 2 way set associative 32 B lines, 6 cycle miss latency, read port and write port, one outstanding miss per register

Table 2.2: Arithmetic operation latencies and repeat-rates simulated in *SimpleScalar*.

Operation	Units	Latency (cycles)	Repeat-rate (cycles)
`iALU`	4	1	1
`intMult`	1	3	1
`intDiv`	1	20	19
`fpALU`	4	2	1
`fpMult`	1	4	1
`fpDiv`	1	12	9
`fpSqrt`	1	24	18

however the operation latency and repeat-rate for integer multiplication and division are different. The same is true for the combined floating point multiplier/divider/square-rooter.

For this thesis *SimpleScalar* was modified so that each arithmetic operation could be studied independently. The integer multiplier/divider unit and floating point multiplier/divider unit were separated into distinct units, but maintained their original latency and repeat-rate shown in Table 2.2.

The `iALU`s perform logical operations and unsigned and signed integer additions for jumps, branches, loads, stores and integer addition instructions. The `fpALU` performs floating point addition, integer conversion and comparison operations. The `fpDiv` and `fpSqrt` are combined into a single unit.

2.3 Benchmarks

This section introduces the benchmarks used throughout the research project. The benchmarks were compiled for the *SimpleScalar PISA* platform using *gcc* version 2.6.3 supplied with *SimpleScalar*. Four benchmark sets were used: arithmetic, *Mediabench*, *SPEC*, and a *test* set that was not used in the design stage. They are described in more detail below.

Each benchmark was compiled at various compilation levels: *-O0*, *-O1*, *-O2*, *-O3*, *-O3 -funroll-loops* and *-O3 -finline-functions*. Increasing the compiler level often has the effect of increasing the code density; ideally the same work is done in fewer machine operations, improving the speed of execution. High density code generally has a higher number of dependencies per instruction than low density code.

Consider two versions of the same program, with high and low code densities. A program with high density could gain a greater relative throughput increase for highly dependent serialised code. However, parallelism and long latency stalls can mask some of the speculative gains. On a relative scale, low density code is naturally more insulated from these stalls due to its inefficiency. Conversely, speculation of serial code might be more detrimental to high density programs as speculation losses represents more of the overall execution.

Execution traces were captured with the in-order *SimpleScalar* tool *sim-profile*. The arithmetic operations were extracted and used to measure the average probability of correctness of the approximate arithmetic units in the design stage. The same benchmarks were later used in out-of-order simulations using the *sim-outorder* tool, with the addition of the *test* benchmark set.

2.3.1 Arithmetic benchmarks

The *arithmetic* set consists of benchmarks with a high arithmetic density. They are synthetic benchmarks that generally suffer from over-simplification and test only a small fraction of a CPU's capabilities. For instance, some benchmarks might intensively use integer, but not floating point instructions. Most have such a small memory footprint they fit entirely in the caches, removing the memory access latency from the overall system performance.

Most of the *arithmetic* benchmarks iterate through a main loop, repeating few operations many times. The total execution time, or execution rate is often quoted as a performance metric.

Many of the synthetic benchmarks used in the *arithmetic* set are highly repetitive. In each loop iteration the input operands might not change, or have many of the same values in common. This repetition can bias the evaluated correctness. For repeatability and automated testing some of the benchmarks were modified to remove human input.

The benchmarks in the *arithmetic* benchmark set are:

calc_pi calculates the first 1,000 digits of pi using an arithmetic identity [Author unknown, 2008].

dhrystone is a synthetic benchmark that tests integer performance [Weicker, 1984]. It was composed to be representative of typical programs. Version 2.1 was used. It is written in *C*.

whetstone is a synthetic benchmark that measures floating point performance [Curnow and Wichmann, 1976]. The version used was written in *C*.

linpack is a collection of subroutines written in *Fortran* (and translated to *C*) that analyse and solve linear equations and linear least-squares problems [Dongarra et al., 1986]. The systems are represented as different types of matrices including general, banded, triangular, symmetric positive definite. *linpack* uses column operations to increase pipelined hardware utilisation by exploiting spatial data locality in memory.

livermore is a benchmark developed at the Lawrence Livermore National Laboratories in 1970, later expanded and released in 1986 as *Fortran* source code. The benchmark includes 24 numerical computation kernels, including calculation of a hydrodynamics fragment, Monte Carlo search and a Planckian distribution [McMahon, 1986]. The version used was written in *C*.

2.3.2 Mediabench

Mediabench is a suite of applications and inputs assembled by Saint Louis University to be representative of communications and multimedia applications. Each application in the *Mediabench* suite is freely available [Lee et al., 1997]. A digest of applications in the *Mediabench* suite is provided below. More detailed descriptions are provided in Appendix D. *Mediabench* applications exhibit statistically different characteristics to *SPEC INT* benchmarks in four metrics: achieved instructions-per-clock, instruction cache hit rate, data cache read hit rate, and memory bus utilisation [Lee et al., 1997].

Mediabench was selected over the newer *Mediabench 2.0* benchmark suite to limit size and number of tests performed, reducing simulation time in later experiments. Each compiled benchmark was run unmodified, with standard inputs. Many of the benchmarks in the *Mediabench* suite are programs to encode and decode pictures, sound and video to different formats. Results reported in this thesis are averaged for encoding and decoding, where applicable. Not all of the *Mediabench* benchmarks could be compiled to *SimpleScalar* because of the availability of libraries, or syntax not supported by the included compiler. The following benchmarks were used in this thesis:

JPEG A standardised compression method for full-colour and gray-scale images. The benchmark implements JPEG image compression and decompression. JPEG is a lossy compression algorithm.

mpeg2play A player for MPEG-1 and MPEG-2 video bit streams. It is based on mpeg2decode by the MPEG Software Simulation Group. This version is optimised for high execution speed at the cost of a less straightforward implementation and slightly non-compliant decoding.

ADPCM (Adaptive Differential Pulse Code Modulation). A family of speech compression and decompression algorithms. The Intel/DVI ADPCM code was used, and is recommended by the IMA Digital Audio Technical Working Group.

G.721 The CCITT (International Telegraph and Telephone Consultative Committee) implementation of G.721 voice compression. This source code is released by Sun Microsystems, Inc. to the public domain.

PEGWIT A program for performing public key encryption and authentication. It uses an elliptic curve over GF(2^{255}), SHA1 for hashing, and the symmetric block cipher square.

ghostscript A set of software that provides:

1. An interpreter for the PostScript™ language.
2. A set of *C* procedures (the Ghostscript library) that implement the graphics capabilities that appear as primitive operations in the PostScript language.
3. An interpreter for Portable Document Format (PDF) files.

Mesa A 3D graphics library with an API which is very similar to that of OpenGL.

RASTA A program for speech recognition that supports the following front-end techniques: PLP, RASTA, and Jah-RASTA with fixed Jah-value. The Jah-Rasta technique simultaneously handles additive noise and spectral distortion.

EPIC (Efficient Pyramid Image Coder) is an experimental image data compression utility. The filters have been designed to allow extremely fast decoding on non-floating point hardware, at the expense of slower encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition).

2.3.3 SPEC

SPEC CPU2000 is a benchmark suite published by Standard Performance Evaluation Corporation (SPEC), and is divided into two subsets: the *SPEC CINT2000* benchmarks profile the integer arithmetic performance of CPUs, and the *SPEC CFP2000* benchmarks are floating point intensive. Ideally each benchmark is portable to many platforms, uses significant hardware resources, and has execution that is not dominated by I/O [Henning, 2000]. *SPEC CPU2000* was chosen over the newer *SPEC CPU2006* suite to reduce simulation time and memory requirements [Standard Performance Evaluation Corporation, 2000; Henning, 2007]. Each benchmark is written in *C*, *C++*, *Fortran-77* or *Fortran-90*. A short description of each benchmark in the *SPEC* suite shown in Table 2.3.

SPEC CPU2000 benchmarks are designed to thoroughly test a wide range of attributes of a computer system, hence the execution times for full tests are very long, and infeasible for an execution driven simulation. The *SPEC* benchmark tool set includes three input sets, called *test*, *train* and *ref*.

The *test* input set operates on a small representative input set to test the likelihood that the compilation of each *SPEC* binary was successful. The *ref* input set is the largest set, requiring more system memory and execution time, and is used for standardised reporting of the benchmark results. The *train* input set was designed for a total running time in between the *train* and *ref* sets. It is used to train a compiler in execution-directed compilation, generating better optimised code, at the expense of an additional compilation pass.

To reduce the execution time of execution driven simulation with *sim-outorder*, the smallest input set, *test*, was used in all reported instances of *SPEC* benchmark results throughout this thesis.

Due to difficulties and omissions with the compilers, libraries and implementations of certain system calls in *SimpleScalar*, not all *SPEC CPU2000* benchmarks could be compiled for use. Table 2.4 lists the benchmarks implemented.

2.3.4 Test benchmarks

The *arithmetic*, *Mediabench* and *SimpleScalar* benchmarks were used extensively to evaluate the correctness of approximate arithmetic units; the distribution of operands in these benchmarks affected their design. The *test* benchmark set was not used in the design stage so that it could be used as an unbiased test.

The *test* benchmarks are written in portable *C*, and are:

Table 2.3: Brief description of the benchmarks in the *SPEC CINT2000* suite.

Set	Benchmark	Language	Category
SPEC INT	*164.gzip*	*C*	Compression
	175.vpr	*C*	FPGA circuit placement and routing
	gcc	*C*	C programming language compiler
	181.mcf	*C*	Combinatorial optimisation
	186.crafty	*C*	Game playing: chess
	197.parser	*C*	Word processing
	252.eon	*C++*	Computer visualisation
	253.perlbmk	*C*	PERL programming language
	254.gap	*C*	Group theory, interpreter
	255.vortex	*C*	Object-oriented database
	256.bzip2	*C*	Compression
	300.twolf	*C*	Place and route simulator
SPEC FP	*168.wupwise*	*Fortran-77*	Physics/quantum chromodynamics
	171.swim	*Fortran-77*	Shallow water modelling
	172.mgrid	*Fortran-77*	Multi-grid solver: 3D potential field
	173.applu	*Fortran-77*	Parabolic/elliptic partial differential equations
	177.mesa	*C*	3D graphics library
	178.galgel	*Fortran-90*	Computational fluid dynamics
	179.art	*C*	Image recognition/neural networks
	183.equake	*C*	Seismic wave propagation simulation
	187.facerec	*Fortran-90*	Image processing: face recognition
	188.ammp	*C*	Computational chemistry
	189.lucas	*Fortran-90*	Number theory/primality testing
	191.fma3d	*Fortran-90*	Finite-element crash simulation
	200.sixtrack	*Fortran-77*	High energy nuclear physics accelerator design
	301.apsi	*Fortran-77*	Meteorology: pollutant distribution

Table 2.4: *SPEC CPU2000* benchmarks compiled for use with *SimpleScalar*.

Type	Benchmark	Compiles?	Runs?
SPEC CFP2000	*168.wupwise*	Yes	Yes
	171.swim	Yes	Yes
	172.mgrid	Yes	Yes
	173.applu	Yes	Yes
	177.mesa	Yes	Yes
	178.galgel	No[1]	No[8]
	179.art	Yes	Yes
	183.equake	Yes	Yes
	187.facerec	No[1]	No[8]
	188.ammp	Yes	Yes
	189.lucas	No[1]	No[8]
	*191.fma3d3*d	No[1]	No[8]
	200.sixtrack	Yes	No[7]
	301.apsi	Yes	Yes
SPEC CINT2000	*164.gzip*	Yes	Yes
	175.vpr	Yes	Yes
	gcc	Yes	Yes
	181.mcf	Yes	Yes
	186.crafty	No[2]	No[8]
	197.parser	Yes	Yes
	252.eon	No[3]	No[8]
	253.perlbmk	Yes	No[4]
	254.gap	No[5]	No[8]
	255.vortex	Yes	Yes
	256.bzip2	Yes	Yes
	300.twolf	Yes	No[4]

[*] There is no *Fortran-90* compiler for SimpleScalar (*PISA*).

[†] *gcc* compiler uses unsupported opcodes.

[‡] *gcc* requires unavailable library `iostream`.

[§] Unimplemented system call in *SimpleScalar*.

[¶] System file `termio.h` is not available for *SimpleScalar*.

[**] Runtime error.

[††] Unable to compile—*gcc* and *glibc* versions are too old.

arith-throughput A simple benchmark, similar to *whetstone*, that records the time taken to perform a number of loop iterations, where each loop performs integer, floating point, trigonometric or I/O operations [Cowell-Shah, 2004]. Each operation is performed on the loop index, so that the operands are guaranteed to be different in each iteration, and cannot be optimised out by a compiler.

fbench A small ray tracing benchmark to test floating point performance [Walker, 1980]. The benchmark models optical ray-tracing using a coeffients taken from a textbook that describes a basic lens. The benchmark is floating point intensive, and contains many repeated operations.

ffbench A small *FFT* benchmark that performs a fourier transform, followed by an inverse fourier transform on a 256×256 double precision complex matrix [Walker, 1989].

miller-rabin A simple, portable program that implements the Miller-Rabin primality test [Author unknown, 2009]. A given number can be tested, non-deterministically, if it is a prime number with k tests. The probability of error of the outcome is 4^{-k}. This program was modified to test the primality of 10 known primes and 10 known composites in the range $0 \cdots 2^{32}$. Faster implementations exist, but require arbitrary precision number libraries that were difficult to port to *SimpleScalar*.

All of the benchmarks have configurable inputs for extended testing, but for simulation purposes in this thesis the defaults are used. A more complete description is included in Section D.3.

2.3.5 Benchmark sizes

The total memory footprint of a program affects the overall performance of the system. Small binary files that fit entirely in the level 1 `iCache` or level 2 cache are not subjected to much memory access latency. The compiled binary sizes for a selection of benchmarks is shown in Table 2.5, and can be compared to the cache sizes of Table 2.1.

The size of the benchmark binaries are larger than the level 2 unified cache, so it is likely that the memory hierarchy will be exercised just by fetching instructions, and probably more when data accesses are accounted for. Most benchmark binaries reduce in size when the compiler optimisation level is increased. Additional optimisations such as code inlining (*-O3 -funroll-loops*) and loop unrolling (*-O3 -finline-functions*) tend to increase the binary sizes with compared to *-O3* alone.

2.3.6 Arithmetic operations in benchmark programs

Benchmarks were traced with *sim-profile*. Tables C.1, C.2, C.3 on pages 319–321 show the number of integer operations for *arithmetic*, *Mediabench* and *SPEC* benchmarks. Tables C.5, C.6, C.7 on pages 323–325 express the number of integer arithmetic instructions as a percentage of total instructions retired.

Similar tables are shown for single and double precision floating point operations are in Tables C.9–C.15 on pages 327–333.

2.4 Synthesis

Throughout this thesis synthesis results are presented to provide delay, area and power estimates of digital circuits. Most digital circuits are described in synthesisable VHDL. The primary tools used were *Synopsys VHDL Analyser* (version Y-2006.06-SP2) and *Synopsys Design Compiler* (DC) (version B-2008.09-SP2). Each individual arithmetic unit tested was driven and loaded by a unit-sized `D flip-flop`(*DFF*), and synthesised with high compiler effort and typical-conditions wire load model.

A single target library was consistently used for the project to provide a fair comparison between circuits. The synthesis library used was the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library, originally released in 2000. It provides basic logic cells, arithmetic cells, and register file cells in a number of sizes and drive strengths, including low-power variants [Art, 2002].

The synthesis and simulation tools discussed in this chapter were used in the next chapter to derive and verify fundamental arithmetic results, used later to design approximate arithmetic units.

Table 2.5: File sizes of the *Arithmetic*, *Mediabench* and *SPEC* benchmarks, in kB.

	Benchmark	*-O0*	*-O1*	*-O2*	*-O3*	*-O3*	*-O3*
						-funroll	*-finline*
arithmetic	*calc_pi*	204	204	204	204	204	204
	dhrystone	228	224	224	224	224	224
	linpack	232	220	220	220	224	232
	livermore	560	524	524	524	524	524
	whetstone	244	240	240	240	240	240
Mediabench	*ADPCM*	216	216	216	216	216	216
		216	216	216	216	216	216
	EPIC	356	336	336	336	336	364
		352	320	320	320	320	344
	G.721	244	240	240	240	240	244
		240	240	240	240	240	244
	ghostscript	4088	2928	2920	3080	3080	3620
	JPEG	528	452	452	476	476	544
		604	484	488	496	496	560
	Mesa	2012	1316	1300	1372	1372	1664
		2088	1368	1352	1428	1428	1724
		2008	1316	1296	1372	1372	1656
	mpeg2play	508	436	432	436	436	460
		404	360	360	360	360	384
	PEGWIT	408	348	348	364	364	376
	RASTA	956	908	908	904	904	910
SPEC	*255.vortex*	1860	1564	1548	1576	1580	1636
	256.bzip2	336	296	296	320	324	388
	300.twolf	1052	724	712	712	716	916
	197.parser	644	540	536	604	608	800
	188.ammp	704	704	704	704	548	704
	181.mcf	308	292	292	292	296	312
	183.equake	332	332	332	332	304	332
	gcc	4836	3436	3392	3680	3684	4484
	179.art	316	316	316	316	308	316
	175.vpr	744	592	584	592	596	676
	164.gzip	500	460	460	468	472	536

Chapter 3

Theory and Applications of Arithmetic Approximation

"Errors using inadequate data are much less than those using no data at all."

Charles Babbage (1791–1871)

This chapter reviews digital arithmetic analyses and techniques found in the literature, with examples of probabilistic computing and arithmetic approximation. Also included are studies validating results found for approximate arithmetic units, and basic properties of commonly executed arithmetic instructions.

Fundamental limits restrict the maximum instruction execution throughput for computer processors. Firstly, there is a *control limit*, arising from instructions such as branches that direct the execution of the program. Secondly, the availability of physical hardware to execute instructions in the program flow impose a *structural limit*. Thirdly, data dependencies between a data producing and a data consuming instruction impose a *data flow limit* [Lipasti and Shen, 1998].

In a superscalar processor, multiple independent instructions can be executed simultaneously. Adding extra hardware can ease the structural limit, but this does not reduce the number of dependencies. In this case the control and data flow limits are exacerbated with increasing parallelism.

Approximation and prediction are used to break the data and control flow limits by providing a speculative result before it would normally be available, so that dependent instructions can begin execution earlier. The dependent instructions execute in a speculative mode until the exact result is known, and checked against the speculative result. In the case where the speculative result is correct, the speculative execution contributes to an increase in throughput. In the case where the exact result is incorrect, all of the dependent operations must be executed again. This is likely to be costly because time is taken to reset the internal machine state, and potentially re-execute instructions that might have already been ready to retire. Hence, the required probability of correctness for a unit in a speculative scheme is usually higher than 50 %. Throughout this thesis the probability of correctness of a unit shall be referred to as the unit's *correctness*.

Branch prediction and other speculation schemes have been common in high performance microprocessor designs for many years [Com, 1998; SPA, 1992; IBM, 2007; Price, 1995]. Branch prediction is a much researched method of increasing the overall throughput of general purpose processors by speculating on the predicted outcome of branch instructions [Yeh and Patt, 1991, 1992, 1993; Chang et al., 1997]. It can require a prediction accuracy greater than 95 % [Chang et al., 1994]. Multi-valued schemes predict the outcome of a data value such a load target address rather than a binary outcome such as branch-taken or branch-not-taken, and were common in the 1980s. More elaborate schemes involving state machines and branch history buffers were introduced in early PowerPC and Intel Pentium Pro architectures in the early 1990s [Lipasti and Shen, 1996]. The branch history buffer is an example of a multi-valued control speculation. Arithmetic data value speculation (ADVS), the topic of this thesis, is also a multi-valued speculation scheme. The results of arithmetic operations are approximated using specialised hardware, and are available for speculation before the exact arithmetic result is known.

In the late 1990s a primary focus of computer architecture research was on reducing the latency of specific types of instructions (usually loads from memory) by rearranging pipeline stages, initiating memory accesses earlier, or speculating that dependencies to older stores do not exist [Lipasti and Shen, 1996; Moshovos et al., 1998]. As pipelines became increasingly complex, design issues such as buffer bandwidth and simultaneous memory access became prevalent.

This chapter presents results from the literature in the subjects of digital arithmetic and computer architecture. Section 3.1 introduces logic function approximation, and establishes a system of classification for functional approximation. Section 3.2 presents theoretical results that can be used to construct or analyse the approximate arithmetic hardware for an ADVS system. A summary of existing value prediction and approximation schemes and hardware is presented in Section 3.3.

3.1 Methods of approximation

This section discusses two methods of logic approximation that can be employed at the fine grained logic level or on large mega-cells. Approximation techniques modify the properties of circuits more than worst-case design allows so that the critical path delay is reduced, and the output is incorrect in as few cases as possible. Ideally approximation methods should be systematic so that they can easily be used on large structures with minimal intervention. It is also desirable that changes to the circuit result in predictable changes to power, area, delay, and correctness.

Two methods of logic approximation are [Kelly et al., 2009]:

temporal incompleteness The output is read before the result is guaranteed to be correct. This is called *over-clocking* in synchronous systems.

logical incompleteness The logic implementing a function is simplified so that the effect on correctness is predictable, and the critical path delay is reduced. Signal wires, or logic gates for corner cases can be removed, so that the critical path delay is shortened.

Sections 3.1.1 and 3.1.2 discuss these techniques in more detail.

3.1.1 Logical incompleteness

A circuit is logically incomplete if the output cannot be evaluated correctly for every input combination. Logical incompleteness can be used on small or large structures. A simple method is to remove wires and gates on data paths that are particularly slow, or that impose additional overhead for corner cases. After the omission, the circuit is *logically incomplete*, hence the output will not be correctly asserted in some cases.

Alternatively, the function output could be made to be asserted when it would otherwise not be. This would correspond to adding additional terms when the logic function is represented in it's canonical form, as a sum of minterms.

Adding or removing logic can decrease the critical path delay, however, in most cases the most significant delay reduction is realised when the modified logic is re-factored; often a faster implementation of the structurally incomplete circuit can be found. In this project many arithmetic units were designed this way.

Experiments were performed to identify which input combinations occurred frequently, and which occurred rarely. This information was used with different circuit topologies to find the least used static data paths. Other simplifications can be made at the conceptual level. For example in adder circuits carries are unlikely to propagate through many bits, but they could be generated from any of the input bits. In this case the adder can be divided into smaller sections that allow carries to propagate only a short distance. The approach is discussed in Section 3.2.1.6.

3.1.1.1 Logic function approximation

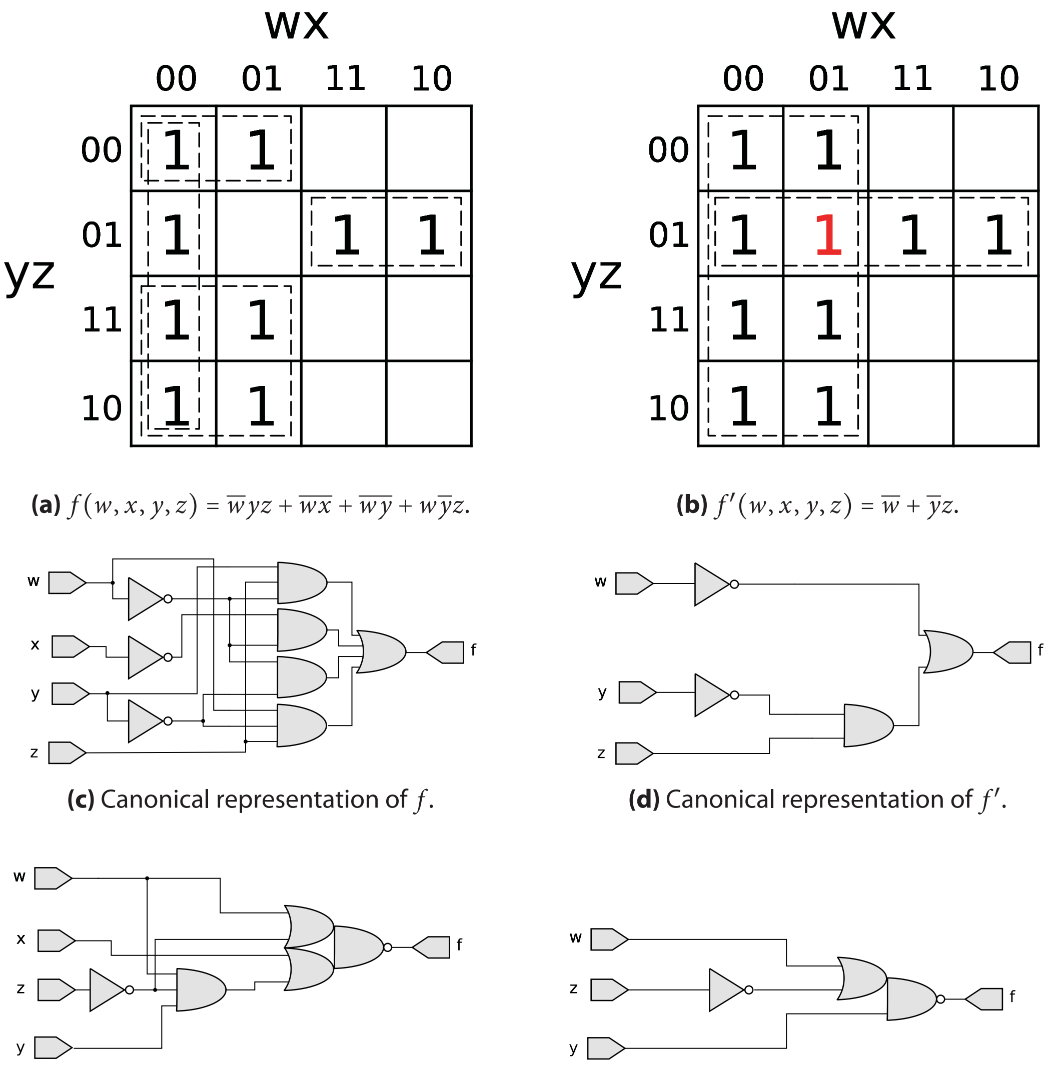
A systematic method of functional approximation was developed, anticipating that smaller functions or control blocks would often lie on the critical path, or that most larger structures would be implemented by approximating the constituent functional blocks.

Logic functions are deterministic, and have a finite set of input and output variables. It is therefore possible to systematically approximate logic functions with specific correctness targets. A program called *logicApprox* was developed to transform an arbitrary logic function f to an approximate function f', for a given correctness target.

logicApprox changes the output of the approximated function for selected input combinations so that the number of minterms in the canonical representation is maximally reduced. Source code for *logicApprox* is shown in Section B.2, and an algorithm is presented in Algorithm 3.1 on the facing page.

Algorithm 3.1 *logicApprox*---An algorithm to introduce errors to a logic function up to a maximum error count. The induced errors are intended to simplify the implementation and reduce circuit delay at the expense of correctness. The function $blocks\_of\_size$ below finds all of the partitions of all of the possible input combinations---these are visualised as the groupings on a Kanaugh Map. The function $bits\_to\_assert$ inspects all of bits in a partition and determines how many bits need to be flipped so that all of the bits in the partition are the same. Each flipped bit is an inserted error.

```
$bits\_asserted \leftarrow 0$ {Each asserted bit is an error that has been introduced.}
{Start with large blocks and progressively use smaller and smaller contiguous blocks.}
for $i = 0 \cdots inputs$ do
  {Find the blocks where asserting as few bits as possible will force every bit in the block
  to be the same.}
  {Try to assert as few ones as possible.}
  for all $block_i \in$ blocks_of_size($2^{inputs-i}$) do
    ($block$, $assertions$) ← min_bits_to_assert($block_i$, 1)
    if $bits\_asserted + assertions < max\_errors$) then
      assert_block($block$, 1)
      $bits\_asserted \leftarrow bits\_asserted + assertions$
    end if
    {Try to assert as few zeroes as possible.}
    ($block$, $assertions$) ← min_bits_to_assert($block_i$, 0)
    if $bits\_asserted + assertions < max\_errors$) then
      assert_block($block$, 0)
      $bits\_asserted \leftarrow bits\_asserted + assertions$
    else
      goto $done$
    end if
  end for
end for
```



(a) $f(w,x,y,z) = \overline{w}yz + \overline{w}\,\overline{x} + \overline{w}\,\overline{y} + w\overline{y}z.$

(b) $f'(w,x,y,z) = \overline{w} + \overline{y}z.$

(c) Canonical representation of f.

(d) Canonical representation of f'.

(e) Synthesised logic for $f(w,x,y,z)$.

(f) Synthesised logic for $f'(w,x,y,z)$.

Figure 3.1: An example of logic approximation using a Kanaugh map. An exact logic function f is shown in 3.1a. The approximated function f' is derived by modifying the function in a limited number of cases, shown in 3.1b. The canonical implementation with *AND*, *OR* and *NOT* gates is shown in 3.1c. Synthesised implementations of f and f' are shown in 3.1e and 3.1e, respectively.

An example of using *logicApprox* on a simple 4 input function is shown in Figure 3.1. In this case a single error, shown in red, was introduced by asserting f' when $(w,x,y,z) = (0,1,0,1)$. Figures 3.1c and 3.1d show the reduction of logic, when implemented directly from the canonical representation. However, when the f and f' are synthesised with modern synthesis tools, the cells used to implement the function are often changed, depending on the synthesis constraints, and properties of the target library. Direct implementations using the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library both have a critical path delay of 294 ps, so the loss in correctness does not improve the latency. Figures 3.1e and 3.1f show synthesised versions using the *Synopsys Design Compiler*. The critical path delay was reduced to 239 ps and 114 ps respectively for the exact and approximate functions. The error of $^1/_{2^4}$=6.25 % results in a 52 % reduction in latency in this case.

An n input binary logic function has 2^n distinct inputs. By specifying the maximum number of erroneous input cases k, the maximum probability of error is thus $^k/_{2^n}$, assuming equally likely inputs. In the example in Figure 3.1 a logical *1* was introduced to reduce the number of minterms. It is also possible that the output from certain an input combination could be forced to *0* to minimise the number of minterms. The user can specify the total number of assertions, either *1* or *0*, and optionally force the direction of the changed output, called a positive or negative assertion.

The *logicApprox* program:

- can automatically decide to make positive or negative assertions to reduce the number of minterms;
- operates on functions with many inputs;
- will only assert output bits up to the maximum specified if required, so error is not unnecessarily introduced;
- includes an arbitrary tie breaker if two or more input combinations result in the same probability of correctness (i.e., the input is 'symmetric'); and
- weights each input combination equally

For real data, some input combinations occur much more frequently that others. *logicApprox* does not consider input weightings, but it is trivial to modify the algorithm to account for this.

logicApprox does not perform logic minimisation, and so produces verbose output for non-trivial circuits. Instead, the output of *logicApprox* is formatted to be read by the program *Espresso* [University of California, Berkeley, 1994], a logic minimisation tool developed at University of California, Berkeley. *Espresso* implements the ESPRESSO heuristic for logic minimisation. It is not guaranteed to determine the global minimum, but it will produce a result that is free from redundancy [Brayton et al., 1984]. The minimised function f' can then be efficiently represented and easily ported to hardware description languages for implementation.

Logic function approximation can be used effectively at a fine grained level as shown here, but does not necessarily extend to larger composite units such as arithmetic units. In large circuits that are composed of regularly repeated cells, the probability of correctness is strongly influenced by the probability distribution of the input operands. When compounded thorough many cells, the probability of error can accumulate, and the assumption of uniform random inputs can become less valid.

3.1.2 Temporal incompleteness

A circuit is temporally incomplete when its output is sampled before it is guaranteed to be correctly asserted. In a synchronous system this is called over-clocking. The output is sampled from logic between two timing elements at a regular interval, determined by the clock signal. The correctness of the circuit is determined by the sampled value that might or might not need to change in order to assume the correct value. Hence, the probability of correctness depends on the current correct value, and the previous value of the circuit. In addition the sampled value might not have assumed a distinct voltage representing a `1` or `0`. This introduces a state called metastability, discussed in Section 3.1.2.

Temporal incompleteness can also be applied to asynchronous circuits. The evaluation time of an asynchronous circuit is not known ahead of time. The next functional unit cannot start execution until a ‘completed’ signal is provided by the prior circuits. The evaluation of the ‘complete’ signals is often burdensome to the circuit, as it imposes additional area and capacitance for wiring and logic. Simplifying the completion logic could have the effect of signalling completion earlier than the output might be correctly asserted, and potentially speeding up the asynchronous circuit by removing circuit load on the critical path.

Asynchronous circuits were not investigated as part of this research, which was dedicated to modifying a common RISC processor.

Metastability

Sequential logic components (latches or flip-flops—`DFF`) have setup and hold times during which the inputs must be stable. If these conditions are met, then the correct output is present at the timing element output after the appropriate delay for the circuit. If the timing element input is not guaranteed to be at a valid logic level during the setup and hold time there is a chance that the circuit will become *metastable*. This can occur if the driving circuit is still changing the voltage at the clock edge, or in the presence of noise on the input node.

Metastable behaviour includes latch output voltage being held between logic thresholds, or toggling. A metastable circuit can hold is output at a voltage between logic levels, or it can cause the output to oscillate. The feedback mechanisms in timing elements can propagate the metastable state for in indeterminate time. Typically changes in the voltage levels will eventually force the output back into a valid state.

The probability of metastability can be reduced by using synchroniser circuits. Synchronisers effectively work by inserting additional timing elements after a possibly metastable latch, to provide the output time to settle to the desired logic level. The second latch can operate on the same clock as the metastable latch, or on a phase-shifted clock. The probability of sampling a metastable input at the synchroniser decreases exponentially with the amount of buffering time [Weste and Harris, 2004].

Temporally incomplete circuits must consider the effects of metastability when setting the over-clocked frequency. The acceptability of passing metastable output to later logic stages, and the additional latencies due to synchronisation offsetting the gains of over-clocking should be balanced for the application and technology.

A simple investigation of a temporally incomplete multiplier is shown in Section 5.2.6.1.

3.2 Fundamental arithmetic results

This section summarises observations, properties and theorems presented in the literature that are relevant to the latency or accuracy of basic digital arithmetic operations.

3.2.1 Addition and subtraction

Addition is conceptually the easiest arithmetic operation to understand and implement. Simple adders can be implemented serially using very few components but have high latency. Parallel adders use more components, but reduce the total calculation time by spreading the work among parallel data paths. The worst-case latency for addition, even in parallel prefix networks, is dependent on the time required to propagate a carry from the least significant bit (LSB) to the most significant bit (MSB). The latency of approximate arithmetic units is less than the worst-case latency, introducing a probability that the result is incorrect. In the following sections the expected carry-length of an addition is shown to be much less than the worst-case carry length, exposing a method to reduce the latency without significantly reducing the probability of correctness.

3.2.1.1 Average worst-case carry length

Addition is performed by summing digits at each position of significance for the input, and producing a sum and carry digit for each. Each carry is propagated to the next highest digit and summed until there are no further digits to sum. The worst-case time is thus the time required to propagate a carry digit from the least significant digit to the most significant digit. Modern adder circuits use innovative implementations to operate on several digits at a time, and thus reduce the worst-case delay. However, the worst-case delay is very unlikely, as only few input combinations will produce this pathological case.

N digit adder circuits can be constructed of single-digit `full-adder`(*FA*) cells, that each only operate on a single digit. Considering a long-hand addition, in each position of significance in the result, the carry may be generated, propagated, or killed. If there are no carries in the addition, each bit can be independently added in parallel to form the correct sum. If the operands cause a carry to be generated in the position of least significance, and propagated though each digit, then the delay will be the worst-case delay.

The above cases are rare compared to all the possible input combinations. With many input digits, there can be several cases where a carry is generated, propagate over a few digits, and then killed. The worst-case carry length is the longest number of digits that a carry is propagated over. Thus, the delay will be proportional to the *average worst-case carry length* (AWCCL) of all possible input combinations.

The AWCCL is much smaller that the worst-case carry length, but it is notoriously difficult to calculate exactly. Even determining the exact number for a large number of digits via simulation is difficult, due to the enormous number of input combinations. Instead, mathematical approaches to find the range of the AWCCL have been undertaken. The original upper bound on the AWCCL was produced by Burks et. al. before the first electronic computer had even been built [Burks et al., 1946; Goldstine and von Neumann, 1963]!

Given the following:

$$P_N(l) \leq \min\left[1, \frac{N-l+1}{2^{l+1}}\right],$$

where $P_N(l)$ is the probability of there existing a carry-chain of length l or greater in an N bit addition, the average worst-case carry length a_N is

$$a_{N_{\text{Burks}}} \leq \log_2(N). \tag{3.1}$$

This upper limit was reduced by Briley to [Briley, 1973]

$$a_{N_{\text{Briley}}} \leq \log_2(N) - \frac{1}{2}. \tag{3.2}$$

Knuth formulated an expression for the average worst-case carry length using rigorous asymptotic analysis, and gave the result in terms of the word length N, in base b. The base-2 result is:

$$a_{N_{\text{Knuth}}} = \log_2(N) + \frac{\gamma}{\ln(2)} + {}^1\!/_2 + \log_2\left({}^1\!/_2\right) - \delta(N) + O(n^{-1}), \tag{3.3}$$

where

$$\delta(N) = \frac{2}{\ln(2)} \sum_{k\geq 1} \mathbb{R}\left(\Gamma\left(\frac{-2\pi ik}{\ln 2}\right) \exp\left(2\pi ik \log_2\left(\frac{N}{2}\right)\right)\right)$$

and γ is the Euler-Mascheroni constant, the limiting difference between the harmonic series and the natural logarithm:

$$\begin{aligned} \gamma &= \lim_{n\to\infty}\left[\left(\sum_{k=1}^{n}\frac{1}{k}\right) - \ln(n)\right] = \int_1^\infty \left(\frac{1}{\lfloor x \rfloor} - \frac{1}{x}\right) dx \\ &= 0.5772156649\ldots \end{aligned}$$

$\delta(N)$ is bounded by

$$\frac{2}{\ln(2)} \sum_{k\geq 1} \left|\Gamma\left(\frac{-2\pi ik}{\ln(2)}\right)\right| = \frac{2}{\ln(2)} \sum_{k\geq 1} \left(\frac{\ln(2)}{2k\sinh(2\pi^2 k/\ln(2))}\right)^{\frac{1}{2}},$$

hence

$$|\delta(N)| \lessapprox 0.000001574.$$

This unbounded summation is impractical for an approximate evaluation; in practice Briley's upper bound is sufficient to estimate the AWCCL for N bit addition.

3.2.1.2 AWCCL in asynchronous designs

A qualitative method of evaluating the average worst-case carry length in N bit *dual-rail*, ripple carry addition was presented when it was anticipated that *"Future designs of parallel digital computers will be concerned with increased accuracy* (precision)*...one basic speed limitation to these operations is the time required to propagate carries in addition or borrows in subtraction"* [Hendrickson, 1960]. The authors championed asynchronous designs as a solution. An example dual-rail full adder cell from [Murakami et al., 1996] is shown in Figure 3.2, and can be daisy chained to from an N bit ripple carry adder. Each adder cell i for bits $0 \cdots N$ provides true and complementary outputs S_i, $\overline{S_i}$, $C_{\text{out}i}$ and $\overline{C_{\text{out}i}}$.

The equations for the adder output bits are:

$$\begin{aligned}
C_i &= \left(A_iB_i + (A_i + B_i)C_{i-1}\right).O \\
\overline{C_i} &= \left(\overline{A_i}\overline{B_i} + (olnA_i + \overline{B_i})\overline{C_{i-1}}\right).O \\
S_i &= \left(A_iB_i + \overline{A_i}\overline{B_i}\right)C_{i-1} + \left(\overline{A_i}B_i + A_i\overline{B_i}\right)\overline{C_{i-1}} \\
\overline{S_i} &= \left(\overline{A_i}B_i + A_i\overline{B_i}\right)C_{i-1} + \left(A_iB_i + \overline{A_i}\overline{B_i}\right)\overline{C_{i-1}}
\end{aligned}$$

The true and complementary carry signals are gated by an operate signal O to regulate the operation of the adder unit. Completion of the adder can be detected by the complete signal:

$$\text{complete} = (S_0 + \overline{S_0})(S_1 + \overline{S_1})\cdots(S_{N-1}\overline{S_{N-1}}) \tag{3.4}$$

Hendrickson also provides an approximate formula for the average worst-case carry length:

$$a_{N_{\text{Hendrickson}}} \approx \log_2({}^{5N}\!/_4). \tag{3.5}$$

Garside performed a study of worst-case carry length for the purpose of optimising adders for asynchronous operation [Garside, 1993]. A trace of the synthetic benchmark *dhrystone* was captured on an ARM platform, and the worst-case carry length for address and data calculations was recorded. It was found that 32 bit data operations had an average carry length of around 18 bits, and address calculations had an average carry length of about 9 bits. The combined average was less than 13 bits. This was significantly higher than other empirical evidence.

Figure 3.2: An example dual-rail full-adder cell, from [Murakami et al., 1996].

Another study compared the average calculation time of different adders to a ripple carry adder, including *serial adders* (conditional sum adder and completion-detection conditional sum adder), *tree adders* (carry-lookahead adder), and *hybrid adders* (carry-skip adder and carry-select adder) [Franklin and Pan, 1994]. It was found that the average latency of the serial, tree and hybrid adders was 20–40 % faster than the ripple carry adder, and instruction throughput could be increased by up to 15 % in an asynchronous *DLX* design executing randomly generated instructions.

The performance gains from asynchronous parallel additions exploiting the short AWCCL are criticised due to the overhead imposed by the completion circuit required to detect when an adder has finished calculation. The very wide *AND* (or *NOR*) structures required by (3.4) impose an area cost, and also a delay cost especially due to logical fan-out [Kinniment, 1996]. An asynchronous parallel adder is shown to increase performance only over simple adder designs such as a conditional sum adder, and only in limited circumstances where area budgets and regularity limit the choice of available adders. In particular, where speed is important, asynchronous designs can perform worse than high-performance adders and consume more power.

A summary of the predicted average worst-case carry length, in bits, is shown in Table 3.1.

3.2.1.3 Empirical studies of AWCCL

A comprehensive study of worst-case carry length was performed by Li [Li, 2002]. The simulator *SimpleScalar* was used to trace the execution of several *SPEC CPU2000* benchmarks compiled for the *PISA* target. The AWCCL was recorded for every addition or subtraction observed in the simulated

Table 3.1: Upper bounds on the AWCCL, calculated using different mathematical models and expressed in bits.

Adder width (bits)	AWCCL Model		
	$a_{N_{\text{Burks}}}$	$a_{N_{\text{Briley}}}$	$a_{N_{\text{Hendrickson}}}$
4	2.000	1.500	2.322
8	3.000	2.500	3.322
16	4.000	3.500	4.322
32	5.000	4.500	5.322
64	6.000	5.500	6.322
128	7.000	6.500	7.322

execution of the benchmarks. The addition and subtraction operands were recorded in categories, derived from the calculations of the program counter (PC), addition and subtraction instructions, load and store memory targets, branch targets, and the stack pointer.

The overall AWCCL was found to be 1.7 bits, but the results were extremely biased by the inclusion of the PC address calculation in the results. In each normal clock cycle, the PC is incremented to form the next PC, but can be set to other values when a jump or branch instruction is executed. The average worst case carry length for PC adds in all *SPEC CPU2000* benchmarks was 0.9 bits. The results of Li's experiments are reproduced in Table 3.2.

Register-register instructions source their operands from a register and write their result to a register. The *PISA* ISA is a load-store architecture, so no data operands are sourced or written directly to memory in a single instruction. Memory operations calculate a memory address for loads or stores by adding a constant displacement to a base address. Branch instructions calculate a branch target as a constant displacement added to next PC. Stack operations are register or memory operations where one operand is the stack pointer (SP). Stack pointer operations manipulate the SP, and stack memory operations manipulate data on the stack.

Li notices a large proportion of register additions that contain a worst-case zero length carry. Li attributed this to the `add` instruction being used to move the contents of one register to another, such as the assembler example provided below:

```
004002a8 <main+b8> addu $v1[3],$zero[0],$v0[2] # $v1[3]<-$v0[2]
```

Table 3.2: Summary of Average Worst-Case Carry Length for *SPEC CINT2000* benchmarks.

Category	Normalised (%)	AWCCL (bits)	Sum (instructions)
PC	53.5	0.976	140,790,091,129
Register	15.6	2.688	41,916,975,483
Memory	22.7	2.583	63,637,439,363
Branch Taken	2.7	2.014	6,637,326,120
Branch Not Taken	1.4	1.755	2,932,744,517
Stack Pointer	1.4	1.482	3,328,690,511
Stack Memory	2.8	1.069	5,326,388,846
Overall	**100.0 %**	**1.655**	**264,569,655,969**

3.2.1.4 AWCCL for signed arithmetic

[Yuen, 1974] analysed the average carry length of two's complement numbers for the cases when both operands are positive, opposite, or both negative. Yuen's analysis restricts the range of possible operands to -2^M to $2^M - 1$. Hence, the upper $N - M$ bits are sign extended. Each of the following cases is considered:

1. Both addends positive, with probability p^2.
2. One addend positive, one negative, and the result positive, with probability $p(1-p)$.
3. One addend positive, one negative, and the result negative, with probability $p(1-p)$.
4. Both addends negative, with probability $(1-p)^2$.

The AWCCL, a_N, for signed twos complement numbers is shown in (3.6).

$$a_{N_{\text{Yuen}}} \approx -p/2 + p.\left(\log_2(M)\right) + (1-p)(N-M) \tag{3.6}$$

Table 3.3 shows the effect of increasing the probability of negative operands on the average carry length. Operands with a small magnitude and low probability of being positive have the highest AWCCL, because the long chain of asserted leading sign bits will propagate or generate carries, but cannot kill them.

Table 3.3: The AWCCL calculated for signed (two's complement) 32 bit integers, where p is the probability that both operands are positive, and the magnitude of each operand is restricted to the lower M bits.

p (%)	M (bits)						
	4	**8**	**16**	**20**	**24**	**28**	**32**
0	28.000	24.000	16.000	12.000	8.000	4.000	0.000
10	25.350	21.850	14.750	11.182	7.608	4.031	0.450
20	22.700	19.700	13.500	10.364	7.217	4.061	0.900
30	20.050	17.550	12.250	9.547	6.825	4.092	1.350
40	17.400	15.400	11.000	8.729	6.434	4.123	1.800
50	14.750	13.250	9.750	7.911	6.042	4.154	2.250
60	12.100	11.100	8.500	7.093	5.651	4.184	2.700
70	9.450	8.950	7.250	6.275	5.259	4.215	3.150
80	6.800	6.800	6.000	5.458	4.868	4.246	3.600
90	4.150	4.650	4.750	4.640	4.476	4.277	4.050
100	1.500	2.500	3.500	3.822	4.085	4.307	4.500

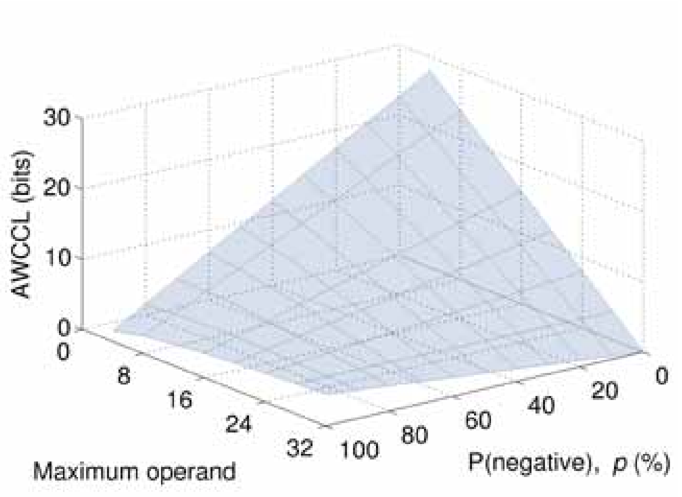


Figure 3.3: Average worst-case carry length for signed two's complement addition, with each operand magnitude constrained to M bits and probability p that each operand is negative.

3.2.1.5 Mathematical analysis of circuit depth and carry length

Ladner and Fischer published a rigorous mathematical approach to parallel prefix computation, including addition. They considered the size S, depth D of overall circuits, and fan-in f as variables of circuit elements to predict delay [Ladner and Fischer, 1980]. The circuit size represents the total number of circuit elements required to construct the circuit, where each circuit element may be a simple cell such as a *`full-adder`*. The size S directly affects the circuit area. The depth of the circuit D is the length of the longest path in the circuit, and directly affects the circuit latency. Fan-in f is the number of inputs for each circuit element, and can affect the overall delay and area of a circuit.

Addition circuits have been proposed with constant fan-in and $\Omega(\log N)$ depth [Reif, 1993], as well as near-linear size, constant depth, but having $\Omega(N)$ elements of unbounded fan-in, making them impractical for implementation [Chandra et al., 1985] (recall that $f(n) \in \Omega(n)$ is bounded below asymptotically). It has previously been shown that the lower bound on the depth required to calculate an N bit binary addition with constant fan-in f is $\Omega(\log_f N)$.

Reif extended this analysis to parallel prefix circuits. An upper bound was derived for the probability of correctness given the delay and depth, where the delay is the number of parallel stops required for evaluation, and depth is the length of the longest path in the circuit. It is shown that [Reif, 1993]:

> There are Boolean circuits for addition and subtraction of random N bit binary numbers with:
>
> 1. constant fan-in f, linear size, and depth $O\left(\log_2((\alpha+1)\log_2(N))\right)$, with error probability *at most* $N^{-\alpha}$; and
> 2. errorless Boolean circuits with constant fan-in except for a single node, linear size, depth $O(\log_2 n)$, and delay at most $O\left(\log_2((\alpha+1)\log_2(n))\right)$, with error probability *at least* $1 - n^{-\alpha}$.

3.2.1.6 Carry length versus probability of correctness

The average worst-case carry length measures the number of carry bits that are needed to correctly calculate 50 % of all additions. A more useful result for the design of approximate arithmetic relates the probability of correctness to the worst case carry length. This relationship can be used to determine both the delay of the adder and the probability of correctness. Using the AWCCL will result in a correctness of approximately 50 %. To design for a specific correctness, a more general result is needed. This section investigates approximate distributions of maximum carry length

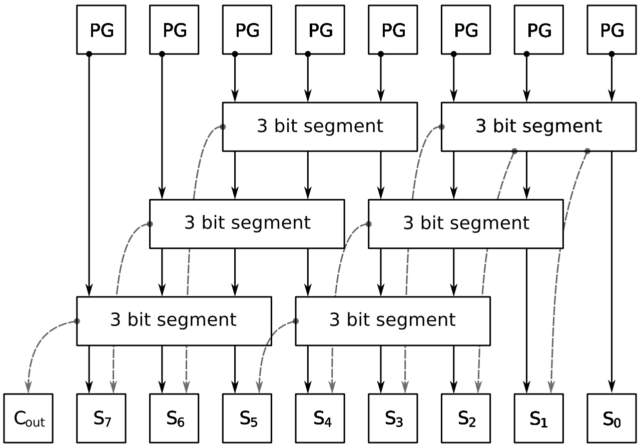


Figure 3.4: An 8 bit approximate Liu and Lu adder, with maximum carry length of 3 bits ($N = 8, l = 3$).

vs.čorrectness.

An approximate distribution for the maximum carry length is shown in (3.7), and has been proven to be a lower bound on correctness [Pippenger, 2002]. It is used in this thesis as a pessimistic approximation to the performance of an adder with a maximum carry length of l bits.

$$P_{\text{Pippenger}}(N, l) = e^{-N/2^{l+1}} \tag{3.7}$$

An analysis of the probability of correctness in an adder for uniform random inputs is shown in (3.8). A corresponding adder was published based on this analysis [Liu and Lu, 2000], where all carries propagated though any l bits segment were separated. In this thesis the adder is referred to as *Liu and Lu's adder*. An 8 bit adder with 3 bit carry segments is shown in Figure 3.4.

$$P_{\text{Liu and Lu}}(N, l) = \left(1 - \frac{1}{2^{(l+2)}}\right)^{(N-l-1)} \tag{3.8}$$

3.2.1.7 Analysis of reported probability of correctness

Results from the literature providing a simple equation to calculate the probability of correctness for a N bit Liu and Lu adder with a maximum carry length of l bits do not yield the exact probability of correctness for uniform random inputs. (3.7) is merely a lower bound, and (3.8) is incorrect, as explained below.

In the derivation of (3.8) it is stated that *"if we only consider l previous bits to generate the carry, the result will be wrong if the carry propagation chain is greater than $(l+1)$"* and *"... moreover, the previous bit must be in the carry generate condition"* [Lu, 2004]. Both statements are incorrect [Kelly and Phillips, 2005].

In analysis of adder circuits, it is useful to define the result of an N bit addition as the product of generate g_i, propagate p_i, and annihilate a_i signals for each digit $i = 0 \ldots (N\text{-}1)$ in the addition (where $i = 0$ for the least significant digit) [Parhami, 2000].

If we consider any l bit segment in an N bit addition, in the Liu and Lu adder a carry will not be propagated from the l-th bit to any other bit. Hence the result in the $(l+1)$-th bit will be wrong. Therefore the Liu and Lu adder can only provide correct answers for a carry length *less than l* bits. The approximate result will be wrong if any carry propagation chain is greater than or equal to l bits.

Now, consider a very long carry string of length $2l$, $(g_i p_{i+1} \ldots p_{i+2l-1})$. As demonstrated above, the most significant l bits in the $2l$ bit segment will be incorrect, as they will not have a carry propagated to them. Thus, it is possible that an incorrect result can be produced without requiring that the previous bit to the most significant l bit segment is a carry *generate*—it might be a carry *propagate*. Also, two *or more* disjoint carry lengths in N bits can produce a spurious result if $l \leq N/2$.

The probability of any input being a generate signal is $P(g_i) = 1/4 = 1/2^2$, and for a propagate signal $P(p_i) = 1/2$, because it can occur in two distinct ways. Hence the probability of each l bit segment producing an erroneous result is actually $1/2^{l+1}$. There are also $(N-l+1)$ overlapping l bit segments required to construct the (N,l) bit Liu and Lu adder.

The result in (3.8) is arrived at by Liu and Lu's assumption *"the probability of (each l bit segment) being correct is one minus the probability of being wrong ... we multiply all the probabilities to produce the final product"*. As discussed above, the Liu and Lu adder will produce spurious results if there exists any carry lengths $\geq l$ bits in the N bit addition. Hence, there are many probability cross terms not represented in (3.8). The probabilities of each l bit segment producing a carry out is fiendishly difficult to calculate as each l bit segment overlaps with $(l-1)$ to $2(l-1)$ other such segments.

3.2.1.8 *BackCount* algorithm

In order to analyse Liu and Lu's quoted probability of correctness it is necessary to know the actual probabilities of success $P(N,l)$ for each word length N and carry segment l. Exhaustive calculation is not feasible for large word lengths, as there are 4^N distinct input combinations for a two operand N bit adder. Set theory and probabilistic calculation are difficult due to the overlapping nature of the l bit segments. For these reasons, a counting algorithm was devised to quickly count all the patterns of carry segments that would produce an incorrect result in the Liu and Lu adder.

For any N and l the number of carry strings which cause the failure of the Liu-Lu adder out of 4^N possible inputs is counted. A carry string of length l bits or more will cause the adder to fail. The algorithm works efficiently if the result for $P(N, l+1)$ is already known, as these combinations can be discounted when counting the violations in $P(N, l)$. For this reason we refer to the algorithm as the *BackCount* algorithm.

Consider a carry string of exactly length l. The string consists of a generate signal g_i and $(l-1)$ propagate signals $p_{i+1} \dots p_{i+l-1}$. Furthermore, the next bit, if it exists, must be an annihilate signal a_{i+l}, or the start of another carry string g_{i+l}. There are two possible ways in which a propagate signal can occur, but only one way in which a generate or annihilate signal can occur in position i. For an arbitrary l bit segment there are r input signals (bits) to the left and s input signals to the right. So, $2^{l-1}\dot{4}^{r+s}$ possible offending combinations are counted. However, from this we must subtract all combinations containing carry lengths longer than l bits to avoid double counting. This is achieved by recursively calling the *BackCount* algorithm on the s and r bits on either side of the l bit segment being considered, until r and s are too small.

To avoid double counting all the combinations involving multiple carry strings of length l in the N bit addition, we must consider each case individually. This is the most time consuming part of the algorithm, as it is computationally equivalent to generating a subset of the partitions of N. The number of partitions of N increases exponentially, and so the process of counting many small carry chains for $l \ll N$ is very time consuming. However inefficient this may be, it has been verified to produce correct results for 8 bit addition against exhaustive calculation. The region of interest is generally $l > log_2(N)$, the expected-worst-case-carry length (see (3.1)). The *BackCount* algorithm is inefficient for $l \leq log_2(N)$, but the calculation is reduced greatly from considering all 4^N input combinations. MATLAB code for the *BackCount* algorithm is provided in Section B.1. The algorithm is shown in Algorithm 3.2.

Table 3.4shows the number of incorrect input cases for 8 bit addition with a maximum carry length of l bits, using the *BackCount* algorithm compared to (3.8) and (3.7).

A simple case exists when $l = 0$, and is included to highlight the difference in prediction against other methods. A zero-length-maximum-carry cannot exist if there are any generate signals. There are four distinct input combinations per bit and three that are not a generate signal. Hence the proportion of maximum-zero-length-carries is given below as

Algorithm 3.2 *backCount*---An algorithm that computes the probability of a k bit carry in N bit addition. It is simpler to exhaustively calculate all the instances where there are carry chains of length $k+1$ bits and subtract them from all possible combinations. A divide and conquer approach is used to partition sections to the left and right of any $k+i$ bit segment being considered. Note that the segments are asymmetrical, because the carry chain must start with a *generate* signal, and can *propagate* indefinitely or be *killed*. The implementation shown in the appendices uses tables to record temporary results from repeated recursion to reduce the execution time.

```
$total\_cnt \leftarrow 0$
for all $carry\_chain_i \in \text{length}(k+1 \cdots N)$ do
    for all $placement_j \in \text{position\_of}(carry\_chain_i, N)$ do
        $bits_{left} \leftarrow \text{left\_of}(placement_j)$
        $bits_{right} \leftarrow \text{right\_of}(placement_j)$
        $cnt_j \leftarrow \text{combinations\_of}(\text{backCount}(bits_{left}, i-1), \text{backCount}(bits_{right}, i-1)) * \text{num\_permutations}(carry\_chain_i)$
        $total\_cnt \leftarrow total\_cnt + cnt_j$
    end for
end for
```

Table 3.4: Predicted number of errors for an N=8 bit logically incomplete approximate adder, with worst case carry length of l bits.

Maximum carry length l (bits)	Exact	*BackCount*	$P_{\text{Liu and Lu}}$	$P_{\text{Pippenger}}$
0	58975	58975	65600	64519
1	43248	43248	65536	63520
2	23040	23040	65280	61565
3	10176	10176	64516	57835
4	4096	4096	62511	51040
5	1536	1536	57720	39750
6	512	512	47460	24109
7	128	128	29412	8869
8	0	0	8748	1200

$$P(N,0) = \frac{4^N - 3^N}{4^N} \,. \tag{3.9}$$

3.2.1.9 Comparison of models of probability of correctness

Although the distribution given by (3.8) and (3.7) are not exact, they provide a sufficiently close approximation for word lengths of 32, 64, and 128 bits. The distribution given by (3.8) approaches the exact distribution for large N because the proportion of long carry chains to all the possible input combinations is smaller for long word lengths. However, Lu's distribution is optimistic because it does not consider all the ways in which the adder can fail. For instance, the predicted accuracy of a 64 bit adder with an 8 bit carry segment, is calculated as $P_{Liu\text{-}Lu}(64,8) = 0.9477$. Results indicate that the correct value is $P(64,8) = 0.9465$, to 4 decimal places.

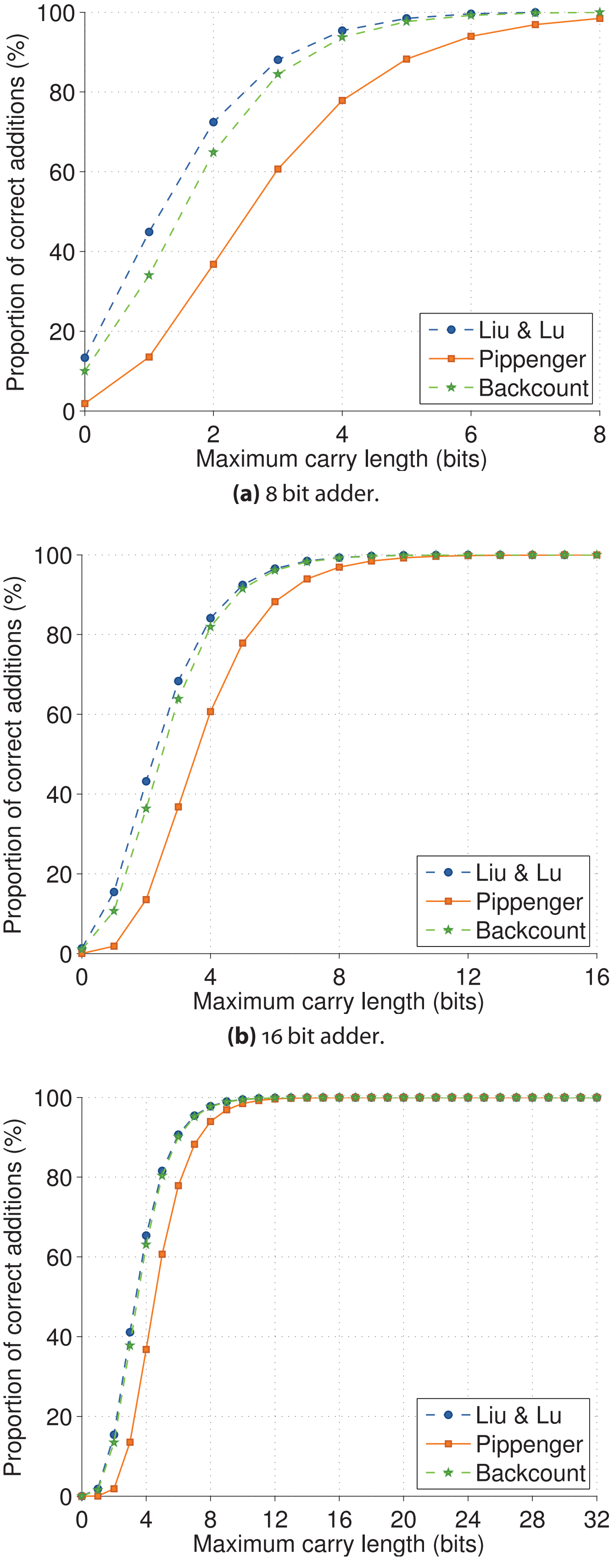
Figures 3.5a, 3.5b, and 3.5c show the predicted accuracy of the various methods for calculating the proportion of correctly speculated results vs. the longest carry chain in the addition. Note that to achieve the accuracies shown with a worst-case carry length of l bits will require the designed adder to use $(l+1)$ bit segments.

3.2.1.10 Synthesis of Liu and Lu's adder

32 bit Liu and Lu adders with increasing carry lengths were compared to a high performance Sklansky adder. Sklansky adders and other high performance designs are discussed in Section 3.2.2. They were synthesised using *Synopsys Design Compiler* and the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library. Both adders include a C_{IN} and C_{OUT} signal. The worst case carry length of the Liu and Lu adder was synthesised from $0 \cdots 32$ bits.

Table 3.5 shows synthesis results for the adders. A graphical representation is shown in Figure 3.6. Both designs were synthesised assuming they are driven by and loaded with a unit sized `DFF`. No other registers were connected to the design because the adder were assumed to complete in one clock. The output `flip-flop` was counted towards the total area, and the output latch setup time was added to the total latency.

The overall latency of the Liu and Lu adder increased with l bit carry length. The exceptions for $l = 5$ and $l = 25$ were due to gate resizing on the critical path, because increasing l added more capacitive load at each stage. Although the latency of Liu and Lu's adder was favourable compared to a ripple carry adder the Sklansky adder was much faster, even when the Liu and Lu adder had a short maximum carry length. The Liu and Lu adder necessarily had a very high fanout at each cell input.



(a) 8 bit adder.

(b) 16 bit adder.

(c) 32 bit adder.

Figure 3.5: Probability of correctness of N bit adders with a maximum carry length of l bits.

Table 3.5: Synthesis results of 32 bit Liu and Lu adder with various maximum carry lengths, compared to a 32 bit Sklansky adder.

Adder	Carry Length	Latency	Area	Power	
	(bits)	(ns)	(μm^2)	Dyn. (μW)	Leak. (nW)
Liu and Lu	1	2.12	0.014	31.47	19.54
	2	2.96	0.032	43.64	29.74
	3	4.52	0.066	51.01	31.25
	4	4.87	0.067	53.25	34.71
	5	4.64	0.076	54.70	40.33
	6	5.37	0.092	56.02	43.21
	7	6.14	0.107	57.08	45.99
	8	6.65	0.122	57.82	48.91
	9	6.92	0.136	58.53	51.56
	10	7.56	0.149	59.21	54.04
	11	8.09	0.162	59.84	56.40
	12	8.74	0.174	60.44	58.65
	13	9.39	0.186	61.35	60.78
	14	10.03	0.197	62.01	62.81
	15	10.67	0.208	62.64	64.74
	16	11.30	0.218	63.07	66.55
	17	11.92	0.227	63.55	68.22
	18	12.57	0.236	64.12	69.79
	19	13.22	0.244	64.52	71.28
	20	13.87	0.251	64.93	72.66
	21	14.52	0.258	65.33	73.94
	22	15.17	0.265	65.67	75.11
	23	15.68	0.271	65.88	76.23
	24	16.24	0.276	66.21	77.20
	25	15.89	0.280	66.34	78.05
	26	16.49	0.284	66.62	78.78
	27	17.08	0.288	66.84	79.41
	28	17.67	0.291	66.98	79.94
	29	18.26	0.293	67.13	80.36
	30	18.82	0.295	67.19	80.70
	31	19.40	0.296	67.23	80.94
	32	19.58	0.314	68.05	84.97
Sklansky	—	4.14	0.018	906.91	5.76

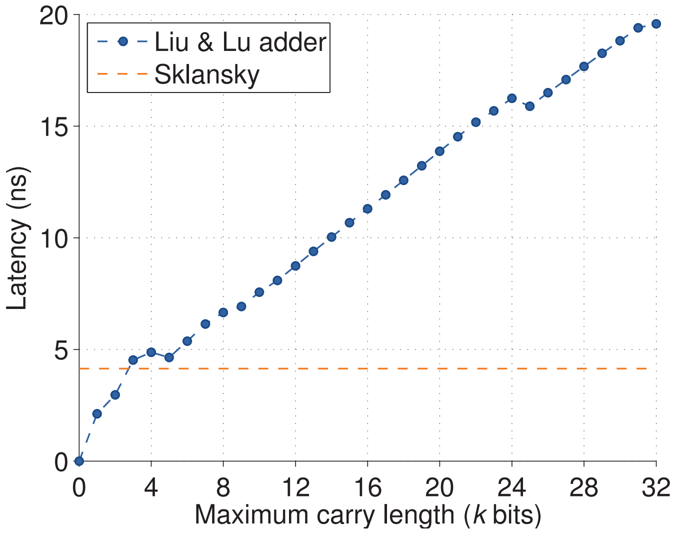


Figure 3.6: Latency of a 32 bit Liu and Lu adders with k carry bits, compared to a 32 bit Sklansky adder.

The latency of Liu and Lu's adder was approximately equal to the Sklansky adder for $l = 3$ bits, corresponding to a probability of correctness of less than 20 % for 32 bit operands (see Figure 3.5c). For a desired correctness of over 95 %, using Liu and Lu's approximation of the expected correctness from (3.8), a carry segment of at least 7 bits is required. As seen in Figure 3.6 and Table 3.5, the latency of a 7 bit Liu and Lu adder with 7 bit carry segments is 6.14 ns, slower than the 4.14 ns exact Sklansky adder.

As shown in Figure 4.2, the distribution of operand bit assertions is different for random numbers compared to operands observed in benchmarks. Hence, the length of the carry segments required might be different for the same expected correctness. In the case of integer addition, *longer* carry segments are required for benchmark programs for a correctness above 90 %, shown in Figure 5.2. Because a 95 % correctness is required for a net throughput increase in *ADVS*, the length of the carry segments would need to be grater than 7 bits.

Hence Liu and Lu's adder does not appear to be feasible for any application of approximate arithmetic requiring high probability of correctness and low latency. Alternative approximate prefix adders are proposed in Section 3.2.2, but were not examined further in this project.

3.2.2 Approximate parallel prefix adders

Parallel prefix adders use a carry recurrence relationship to propagate the carries through each bit of the sum, and can be implemented with different topologies that trade fan-out, logic depth and interconnect [Weste and Harris, 2004]. Some parallel prefix adders are:

Sklansky The prefix carry fan-out is doubled at each stage so that an N bit adder is formed in $log_2(N)$ levels. The logic depth is minimal, but fan-out is high in the later stages.

Kogge-Stone Fan-out is minimised by distributing the carry calculation using many wiring tracks. This adder suffers from high interconnect area and delay.

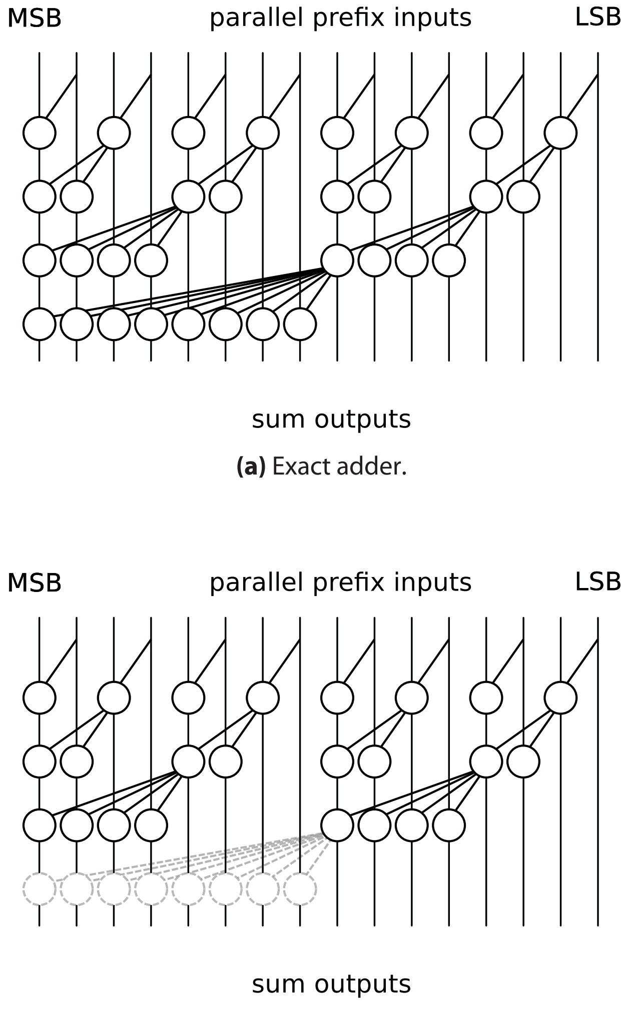
Brent-Kung Additional logic levels are used to gradually generate carries, preventing high fan-out in a single logic level, and reducing interconnect. The initial and later levels are dedicated to generating many short-length carries, and the middle stages to a few long-length carries.

Figures 3.7–3.9 show the parallel prefix adder topologies of the designs above, including possible approximate adders. Each circle represents a prefix cell that calculate the carry propagate and generate signals. The adder LSB is on the right hand side, and the MSB on the left hand side.

The approximate adder units can be formed by removing prefix cells that sum carries greater than a threshold length, or that have a high fan-out. If all the prefix cells in one logic level can be removed the logic depth of the approximate adder unit can be reduced. Each of the exact adder units shown are asymmetrical, so the probability correctness of each unit can be different. For example, bit 15 in the approximate Brent-Kung adder will always be correct. This is not the case for the approximate Sklansky adder. Other approximate designs are possible. Approximate parallel prefix adders were not simulated or synthesised.

Liu and Lu's adder provides a *l* bit carry path for every possible input bit, resulting in a high probability of correctness with relatively short carry chains. The approximate parallel prefix adders shown above do not necessarily provide a minimum *l* bit path for all input bits, so the correctness distribution can be different to Liu and Lu's adder. Nonetheless, it is possible that the correctness is not adversely affected for benchmark inputs, because benchmark operands can be of a smaller magnitude than the pruned adder cells.

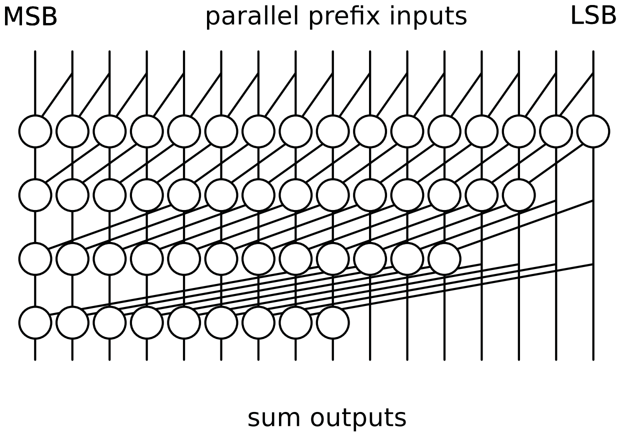
Using the technique above, the approximate Kogge-Stone adder will have the same correctness as Liu and Lu's adder with the same worst-case carry length, l. However, in the case of the Kogge-Stone



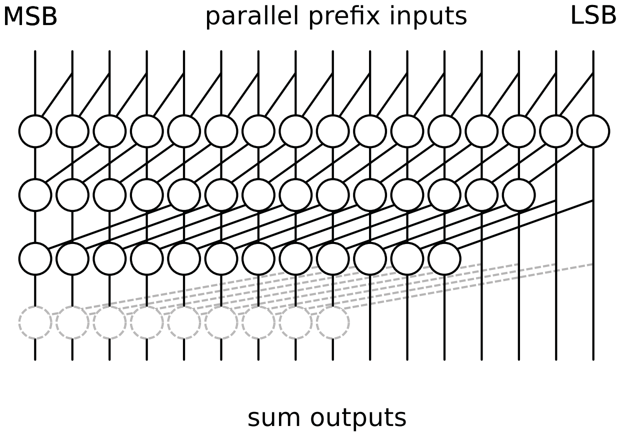
(a) Exact adder.

(b) Approximate adder.

Figure 3.7: An exact and approximate 16 bit Sklansky adder.

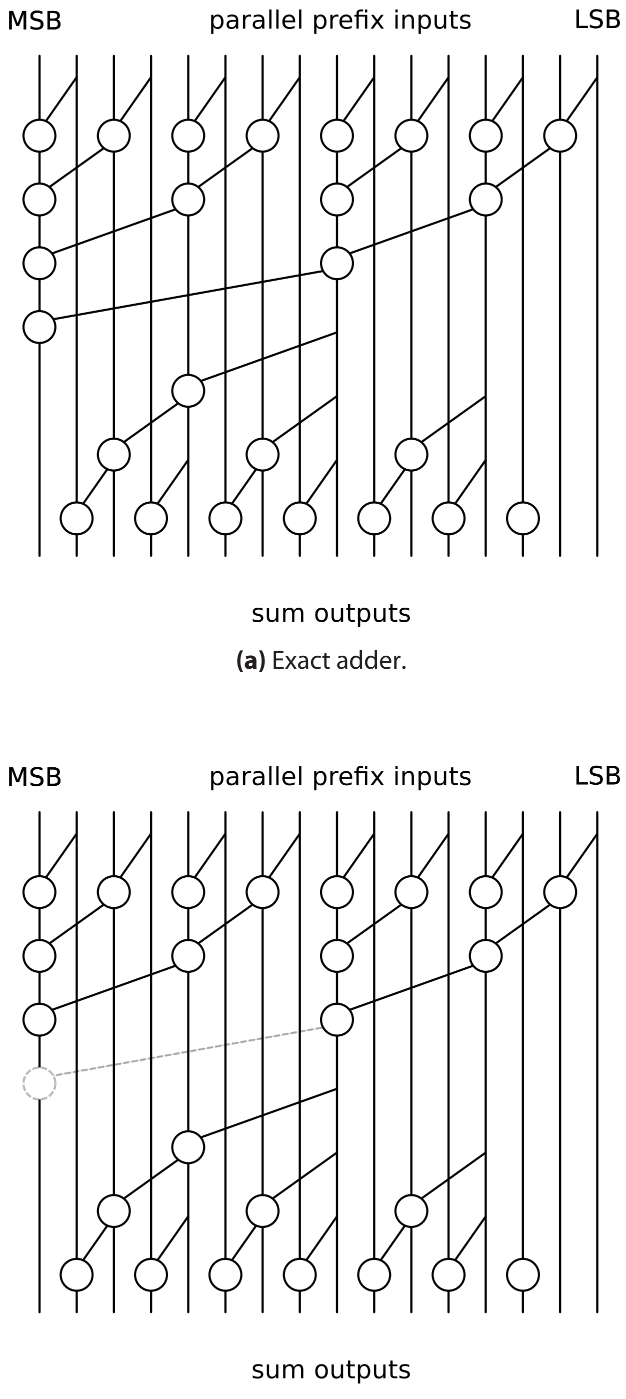


(a) Exact adder.



(b) Approximate adder..

Figure 3.8: An exact and approximate 16 bit Kogge-Stone adder.



(a) Exact adder.

(b) Approximate adder.

Figure 3.9: An exact and approximate 16 bit Brent-Kung adder.

adder, l is restricted to a power of 2.

3.2.3 Multiplication and Division

There is little research in the literature specifically about the theoretical probability of correctness of approximate multipliers and dividers operating on general random inputs. However, in Sections 3.3.2 and 3.3.3 examples of approximate and probabilistic multipliers are provided.

Multiplication and division by constants

Reif characterises multiplier or divider circuits when operating with constant values. For the multiplication of a uniformly distributed random number x by an integer constant $C \geq 2$ [Reif, 1993]:

> There are Boolean circuits for multiplication and division of a random $\lceil \log_2(C) \rceil N$ bit binary number by integer y, with
>
> 1. constant fan-in, linear size, depth $O\left(\log_2(\beta \log_2(N))\right)$ and error probability at most $N^{-\alpha}$; and
> 2. errorless circuits with constant fan-in except at a single node, size $O(N)$, depth $O(\log_2(N))$, but delay at most $O\left(\log(\beta)\log_2(N)\right)$, with probability at least $1 - N^{-\alpha}$,

where

$$\begin{array}{rcl} b & = & \lceil \log_2(C) \rceil \\ \alpha & > & 0 \\ C & \geq & 2 \\ x & \geq & 0 \\ \beta & = & \dfrac{-(\alpha+1)}{\log_2(1-2^{-b})}. \end{array}$$

Comparatively, the delay of a Wallace multiplier is $O\left(\log_2(N)\right)$ [Wallace, 1964]. However this result cannot be directly applied in a microprocessor data path, because the multiplier and multiplicand are variable. This result might assist in a design where a multiplication or division operand is constant, but cannot be directly used for variable inputs.

3.3 Applications of approximate arithmetic

This section covers the applications of approximate arithmetic and speculation techniques. One of the most common techniques for speculation of multi-valued results is prediction, rather than approximation, where a data value is anticipated based on past values. Other approaches to approximation found in the literature include *probabilistic computing* where error is potentially introduced at the transistor level [Chakrapani et al., 2006]. The following subsections address the prediction and approximation of data values based on the arithmetic operation.

3.3.1 Addition and subtraction

Integer addition and subtraction are necessary operations for many instructions, including `addiu` and `sub`, but are also used to increment the PC, adjust the stack pointer and calculate a load/store target, etc. Below is a summary of schemes used for the prediction, approximation or probabilistic calculation of sum or difference results in synchronous systems, and methods to reduce the addition latency in asynchronous systems. Asynchronous systems are of interest because they highlight the dependence of latency on carry length in addition.

Instead of predicting a load address, it is instead possible to simply *predict* the load target contents. Load *target*-prediction and load *value*-prediction methods have been studied and compared. Stride-based predictors are often used for prediction of multi-valued load/store targets [Gonzalez and Gonzalez, 1998; Marcuello et al., 1999]. A stride predictor uses a constant stride-length as an offset from a base address for accesses to regular data structures.

It was concluded that *target*-prediction was more accurate than *value*-prediction when using stride based predictors, however load value prediction was generally faster due to the required memory access times. It was also concluded that the hardware cost associated with value prediction is not negligible, but value prediction might yet be included in for future architectures.

Adders for *asynchronous* systems have been developed to exploit the fact that common arithmetic operands do not have a uniform random distribution, and exhibit different carry length distributions depending on the data (see Table 3.2). For instance, operands for loop counter in programs are usually small and positive, and address offsets to access the memory stack can produce quite long carry strings. An asynchronous dynamic Brent-Kung adder [Nowick et al., 1997] was extended with early termination logic and application specific optimisations to reduce the addition latency [Koes et al., 2005].

An adder that reduces the addition latency by *approximating* the exact result and reducing the probability of correctness was proposed by Liu and Lu. A modification to a ripple carry adder restricts the maximum length that a carry can propagate [Liu and Lu, 2000; Lu, 2004], shortening the critical path compared to a ripple-carry adder. The probability of correctness is dependent on the maximum carry length l bits, and can be estimated with $P_{\text{Liu and Lu}}$ in (3.8). Each output bit is calculated in a ripple-carry fashion from the previous l input bits. Hence, the delay is linear, circuit area (excluding wiring) is geometric, but the probability of correctness can be > 50 % if $l > \log_2 N$. An example Liu and Lu-adder is shown in Figure 3.4. Alternatively, the least significant carries could be generated using shorter segments in the lower-order gaps, sparing load on the first carry segment.

An example of a *probabilistic* adder is used in a special multimedia processor, to improve the power efficiency of an MPEG-4 encoder, at the expense of accuracy [Varatkar and Shanbhag, 2006]. Much of the processing effort for MPEG-4 encoding is concentrated in the motion estimation routine that performs a sum of absolute differences for a 256 element array of numbers, essentially an addition operation. In the probabilistic adders timing errors are introduced by voltage overscaling, with the aim of increasing the power effiency. Errors are corrected using algorithmic noise tolerance (ANT) in which a decision block selects between output from the approximated block and a processed subsampled stream. This is shown to be robust to timing errors. This signal processing technique has also been applied to a general DSP correcting soft errors, because the ANT system is agnostic to the error source [Hegde and Shanbhag, 1999; Shim and Shanbhag, 2006].

3.3.2 Multiplication

Probabilistic CMOS (PCMOS) is a technology used to implement *probabilistic computing*; transistors are not guaranteed to output the correct logic level, but consume much less energy in operation [Chakrapani et al., 2006]. Probabilistic computing is suited to specific '*probabilistic applications*' that can tolerate computational error; in this case the arithmetic error is not corrected.

Arithmetic circuits built in PCMOS use voltage scaling to increase the probability of correctness in the most significant bits, thus reducing the magnitude of the error. Low performance PCMOS ripple-carry adders and array multipliers were used in the simulated operation of a FFT with *HSPICE* [George et al., 2006]. The energy-probability relationship of the PCMOS transistors was tuned to yield a 5.6X power improvement for a radar imaging application compared with a regular CMOS implementation. Errors in operation manifest as degraded image quality, mitigated by the low error magnitude.

Another example of an error-tolerant application is a low density parity check (LDPC) decoder, that operates iteratively on the coded data and parity bits of a message to decode it. The Belief Propagation algorithm employed can be implemented with multioperand adders, similar to multipliers [Phillips et al., 2006]. LDPC decoders using approximate arithmetic reduced the frame error rate for a given noise level, and converged in fewer iterations than a decoder with exact multioperand adders [Phillips et al., 2006]. This is discussed in more detail in Chapter 10.

3.3.3 Division

High speed division is generally performed in one of two ways: the quotient is generated by multiplication by a reciprocal; or by an iterative algorithm requiring a quotient digit selection [Burgess, 2005]. There are examples in the literature of division units that employ temporary approximation to hasten the availability of results, although the quotient is corrected before being output.

In some iterative designs this is referred to as 'internal speculation'. Such designs reduce the latency of the quotient digit selection stage via speculation of the quotient digit, [Pan et al., 1995; Wong and Flynn, 1992; Srivastava, 2007]. Using speculative quotient digits can reduce the size of the lookup table for quotient digit selection, however multiple possible quotient digits are generated. The calculation of the partial remainder can occur earlier with the 'predicted' quotient digits, and the correct partial remainder is selected later. Reducing the lookup table size is attractive for high radix division because the size of the lookup table required increases exponentially with radix. These dividers correct any error in the quotient digits before the result is output.

The dividers above employ internal speculation, but may incur a delay penalty to output an exact result. An example of a general purpose approximate divider that generates inexact quotients was not found.

Some asynchronous designs improve the average case latency by simplifying the quotient calculation, possibly generating an error in the quotient, requiring additional cycles to correct [Fenwick, 1995]. Similar designs have been extended to high radix division and square root [Cortadella and Lang, 1994], and variable latency double precision floating point [Cornetta and Cortadella, 1999].

An example of a divider that could be modified to output an approximate quotient is a reciprocal multiplier which normalises the operands and inspects the upper bits of the divisor d and dividend z to lookup an approximation of the reciprocal of d in a table, using interpolation to reduce the quantization error introduced [Nakano, 1987].

3.4 Conclusion

This chapter has identified techniques for approximation, and presented observations of arithmetic phenomena that establish average case correctness for arithmetic operations. Examples from the literature were shown where system performance was improved using *probabilistic* adders and multipliers, or long latency dividers that use internal speculation or another method to output common-case results faster than normal.

The advantage of *logical incompleteness* is that the resulting approximate circuit will often be smaller than the exact circuit from which it was derived, saving power and area. Furthermore, the result is deterministic, so the effects of approximation are known at design time if the inputs are well characterised. The disadvantages of logical incompleteness include granular control of the correctness; in circuits with high internal fan-out the output might be critically dependent on many or all of the logic in the data path. Hence, modifying or removing any part of the circuit might change many output cases.

The advantage of *temporally incomplete* circuits are that the average correctness is determined predominantly by the clock period. Where removing logic from the data path will cause discrete changes in the critical path delay and the correctness, adjusting the clock period is more like a continuum when there are many inputs. The disadvantages include additional area required for synchronisers, and dependence on the previous state of the output registers.

Addition is a frequent, low latency operation. Although the dependencies between the input and output bits are conceptually easy to understand, the relationships between maximum carry length and correctness are non-linear. The results in this chapter relating probability of correctness to maximum carry length are later used to determine probability of correctness to latency. Operations like multiplication and division are more complex, because more intermediate dependencies exist. In Chapter 5, basic multipliers and divides are built around adders and subtractors.

Chapter 4

Can ADVS Improve the Performance of a Generic RISC Processor?

"All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value."

Carl Sagan (1934–1996)

This chapter defines the scope of the research project undertaken: to determine the feasibility of using arithmetic value speculation to improve the throughput of a RISC processor.

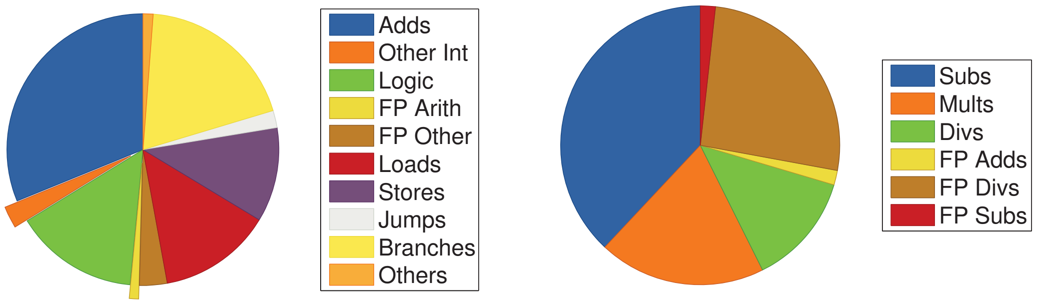
SPECULATION is a well known technique to increase the throughput of processor pipelines. The predictability of branch outcomes and serially accessed memory locations can be successfully exploited so that the overall benefit of increased throughput outweighs the risk of speculative parallel execution. The most likely candidates for improvement are common or slow instructions. Branch prediction and speculative memory accessing are mature fields; modern schemes yield diminishing, incremental relative improvements. Arithmetic instructions are long latency *ALU* instructions, and are used in few speculative schemes because the outcome depends on a calculated result that could have many values, making prediction difficult. To realise a net increase in throughput, the execution of arithmetic instructions must be on the critical path, which is more likely if each arithmetic instruction occurs frequently.

In this chapter the arithmetic components of benchmark programs are analysed to determine the feasibility of arithmetic data value speculation as a scheme for increasing throughput of a general purpose RISC processor. A detailed analysis of the execution of benchmark programs shows the frequency and characteristics of typical arithmetic instructions. Finally, a high level probabilistic simulation of a processor executing benchmark programs with arithmetic units with a variable probability of correctness and execution latency shows the effect on system throughput measured as instruction retirement rate. From this a minimum target for the correctness/delay tradeoff is established for the design of approximate arithmetic units.

4.1 Program execution

In this section benchmark programs are analysed by recording statistics on executed and retired instructions.

The benchmarks from the *arithmetic*, *Mediabench*, and *SPEC* suites were executed in *sim-profile*, the in-order *SimpleScalar* simulator, and the number of occurrences of each instruction type was recorded. Figure 4.1a shows the relative proportion of the different types of instruction. A list of instruction opcodes is shown in Appendix F. Instructions are grouped on the fundamental operation type, so that immediate and register instructions are counted together, and the single and double precision floating point operations are counted together. The extended segments represent the average proportion of integer and floating point arithmetic instructions, excluding all addition in-



(a) Average proportion of instruction types for all benchmarks.

(b) Integer and floating point arithmetic operations, excluding integer add operations.

Figure 4.1: Average relative proportions of each executed instruction type.

structions (`add`, `addiu`, etc). The relative frequency of each instruction type is shown numerically in Table 4.1.

Load and store operations account for $^1/_4$ of the total instructions executed. Almost $^1/_2$ of all instructions are single cycle *iALU* instructions, including `add` and logic operations. A further $^1/_5$ of all operations are control operations, including a large proportion of branch instructions, and some jump instructions. Multi-cycle arithmetic operations account for less that 6 % of the instructions executed.

The relative proportions of the non-`add` arithmetic instructions is shown in Figure 4.1b. The most frequently occurring of these instructions are integer subtractions. Integer multiplication and division operations are represented roughly in the same quantity, but both consist of less than 1 % of all instructions executed. The most common floating point arithmetic instructions are divisions (`div.s`/`div.d`). The most uncommon instructions are floating point multiplications and square-roots.

4.2 Integer arithmetic

In this section, statistics and properties of integer arithmetic operations and traced operands from benchmark programs are presented, including a summary of the types of integer instructions in the *PISA* architecture, a comparison of signed and unsigned integer arithmetic, a comparison of

Table 4.1: Average proportions of benchmark instructions executed in an in-order *SimpleScalar* simulator.

Type	Proportion (%)
Adds	31.1675
Branches	19.1535
Logic	14.7082
Loads	13.4723
Stores	11.2455
FP Other	3.2929
Jumps	2.0363
Subs	1.4123
Others	1.2089
FP Divs	0.9714
Mults	0.7193
Divs	0.4845
FP Subs	0.0650
FP Adds	0.0622
FP Mults	0.0002
FP SQRT	0.0000

the magnitudes of integer arithmetic operands, an analysis of the distribution of operands for each instruction, and a summary of the number of repeated operands in benchmark programs.

4.2.1 Types of integer arithmetic

Modern hardware typically provides the four fundamental arithmetic operations, addition, subtraction, multiplication and division. Occasionally other hardware units such as square, square-root and fused multiply add are available. The integer addition and subtraction unit in the *ALU* is often not necessarily dedicated to addition operations, but may also compute branch, jump, load and store targets.

Arithmetic instructions are also available with an operand supplied in an immediate field, sparing the use of a register or memory location for operand storage. Immediate fields are typically one byte wide, hence they have different carry distribution arise from immediate instructions compared to 32 bit and 64 bit register instructions. Multiplication and division hardware are usually provided in signed and unsigned variants, and may be combined at the expense of throughput for a saving in circuit area.

4.2.2 Signed and unsigned arithmetic

The *PISA* architecture defines signed and unsigned versions of each integer arithmetic instruction, but the compiler supplied with *SimpleScalar* does not use all of them. Tables C.1–C.3 in Chapter 2 show that signed integer `add`, `addi` and `sub` instructions are not used by the compiler in the *arithmetic*, *Mediabench* and *SPEC* benchmarks. The *gcc* compiler does use signed and unsigned multiplication and division instructions, however signed instructions are much more common. Table 4.2 shows the ratio of signed to unsigned multiplication and division operations in benchmarks.

4.2.3 Operand magnitude

The structure of most programs is systematic; various runs of programs differ according to the input that they receive [Sodani and Sohi, 1998]. Most programs spend most if their execution time in a few blocks of code. Some of the executed arithmetic instructions are determined by the control flow of the program (loop variables, etc.) and others from the data flow. Neither the data or control is random, hence the operands observed in program execution are not random.

Table 4.2: Ratio of signed to unsigned retired multiplication and division instructions in benchmarks.

Set	Benchmark	`mult/multu`	`div/divu`
Arithmetic	*calc_pi*	—	90.1
	livermore	—	0.0
	dhrystone	—	476.2
	helloworld	—	—
	linpack	—	0.0
	matrix_mult	—	—
	whetstone	—	0.0
Mediabench	*ADPCM*	—	—
	EPIC	511.8	113.1
	G.721	—	—
	ghostscript	8845.9	0.0
	JPEG	—	—
	Mesa	—	—
	mpeg2play	—	—
	RASTA	0.5	0.3
SPEC	*164.gzip*	—	0.0
	168.wupwise	820345.7	3914.7
	171.swim	0.5	0.0
	172.mgrid	2342.1	0.0
	173.applu	120.4	0.1
	175.vpr	41.4	1.3
	gcc	4.1	0.1
	177.mesa	—	0.4
	179.art	—	0.0
	181.mcf	195.5	0.0
	183.equake	663.7	0.0
	188.ammp	3432.2	0.0
	197.parser	—	174.7
	200.sixtrack	—	—
	253.perlbmk	—	—
	255.vortex	7.1	0.0
	256.bzip22	—	1.0
	300.twolf	—	—
	301.apsi	3287.7	33.7

Table 4.3: Proportion of unsigned integer arithmetic operands that are zero.

Type	Zero (%)	Positive (%)
`add`	23.99	76.01
`sub`	18.67	81.33
`mult`	1.80	98.20
`div`	8.66	91.34

Table 4.4: Proportion of signed integer arithmetic operands that are negative or zero.

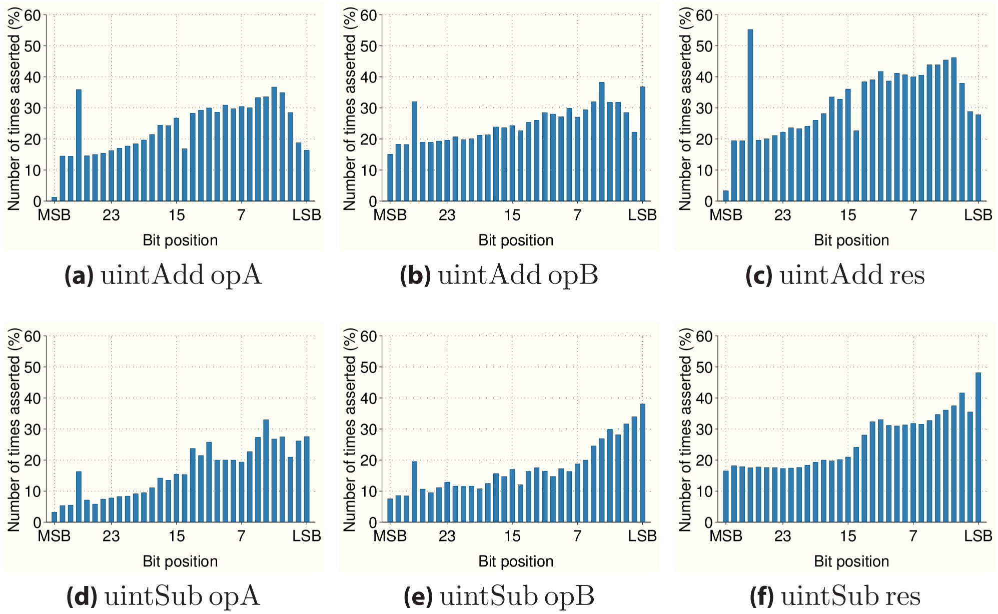
Type	Negative (%)	Zero (%)	Positive (%)
`add`	—	—	—
`sub`	—	—	—
`mult`	7.52	33.64	58.84
`div`	17.30	2.16	80.54

Figures 4.2–4.4 show the distribution of integer operands and their result fields. The distributions were formed by averaging the observed asserted bits for arithmetic instruction operands in each benchmark.

Integer addition and subtraction operands in Figure 4.2 show that the higher order bits are asserted less than the lower order bits, except for bit 29. This outlier is not as obvious when each benchmark distribution is printed individually; it is an artifact of averaging. The result field distributions retain a similar shape to the operands, but are taller than for the input operands. There is also little difference in the distribution of asserted bits for the `opA` and `opB` operands. There are no obvious trends to exploit from examining the input operands, such as a stronger bias towards less significant bits.

The signed and unsigned `intMult` and `uintMult` operands in Figure 4.3 are on average almost twice as likely contain asserted bits in the least significant bits than the most significant bits. As a result, the 64 bit product of the 32 bit operands is more 'spread', but retains the same basic shape. The signed `opB` operand contains characteristic comb-like peaks, formed by averaging many repeated operands in the few benchmarks that contain unsigned multiplications.

The distribution of division operations is shown in Figure 4.4. The *PISA* architecture does not specify that the `opB` operand must be a 16 bit operand, however many of benchmarks contain `uintDiv` `opB`



(a) uintAdd opA

(b) uintAdd opB

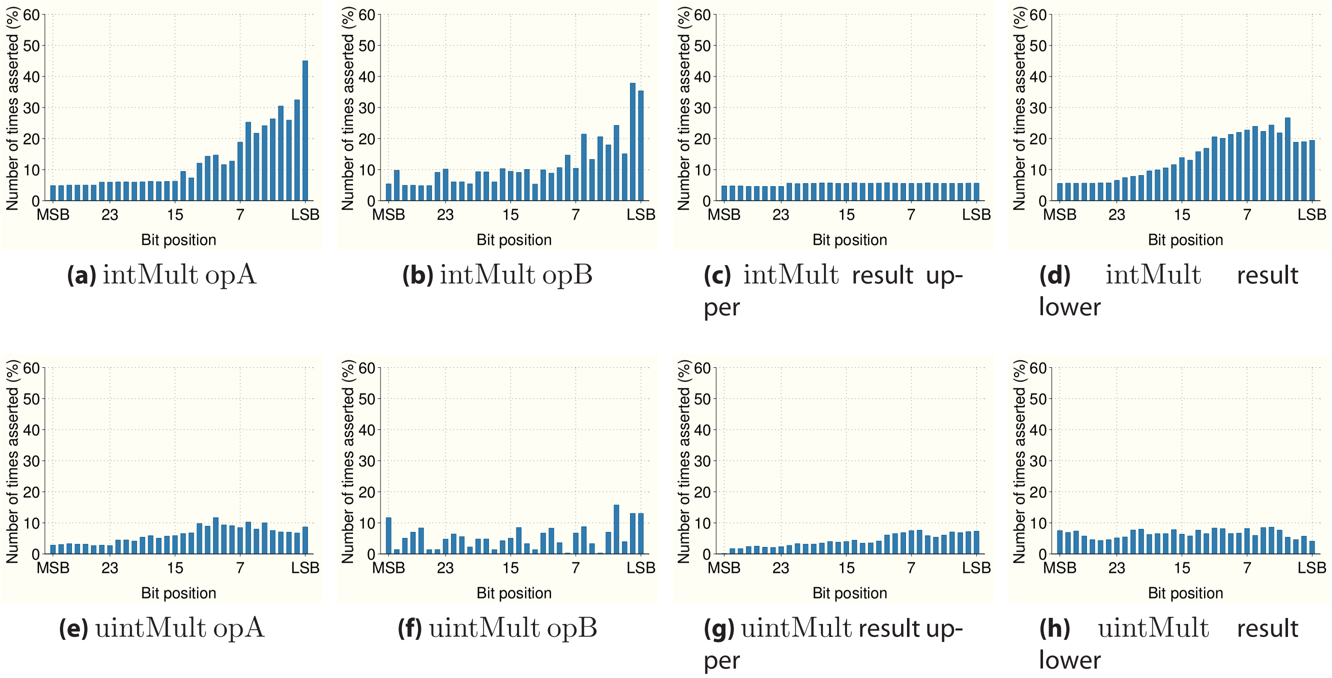
(c) uintAdd res

(d) uintSub opA

(e) uintSub opB

(f) uintSub res

Figure 4.2: Average distribution of input and output bits for unsigned integer addition and subtraction operations. The compiler did not generate signed addition or subtraction instructions.



(a) intMult opA

(b) intMult opB

(c) intMult result upper

(d) intMult result lower

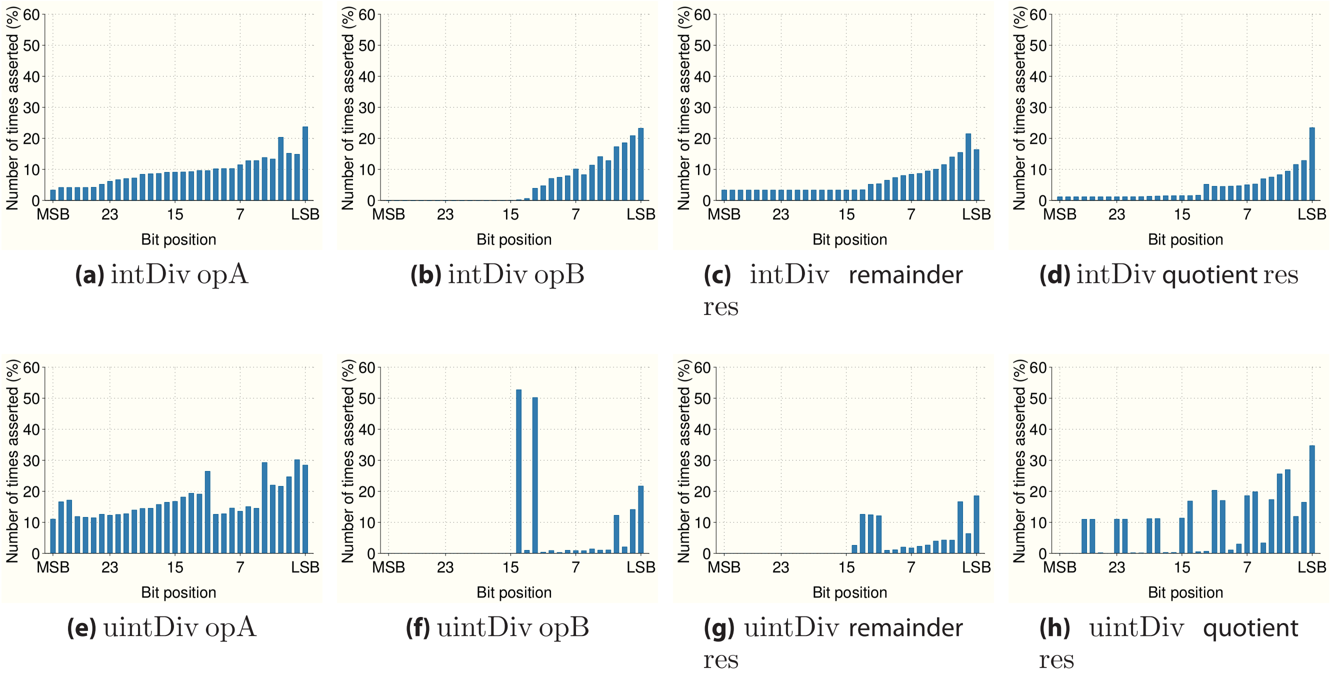
(e) uintMult opA

(f) uintMult opB

(g) uintMult result upper

(h) uintMult result lower

Figure 4.3: Average distribution of integer operands for signed and unsigned multiplication.



(a) intDiv opA

(b) intDiv opB

(c) intDiv remainder res

(d) intDiv quotient res

(e) $\mathrm{uintDiv}$ opA

(f) $\mathrm{uintDiv}$ opB

(g) $\mathrm{uintDiv}$ remainder res

(h) $\mathrm{uintDiv}$ quotient res

Figure 4.4: Average distribution of input and output integer operands for signed and unsigned division.

operands that all fit into a 16 bit field. The only exception is *ghostscript*.

The relative shapes of the quotient and `opA` distributions are similar, implying that many of the `opB` operands are small in magnitude, otherwise the upper bits in the quotient would be asserted less frequently. The distribution of the signed and unsigned quotient and remainder fields are similar, except that the signed distributions have a longer upper tail due to the sign bits, and the signed result distributions maintain the jagged shape of the `uintDiv` `opB` operand distribution.

The distributions of the signed multiplication and division operands are similar to the unsigned multiplication and division operands because a small proportion of the total operands (< 20 %) are signed (see Table 4.4), and hence the upper sign bits are not strongly biased.

4.3 Floating point arithmetic

This section presents an analysis of floating point instructions traces from benchmark programs, including a summary of the types of floating point instructions, a comparison of signed and unsigned operands, a comparison of the magnitudes of floating point operands, and a summary of the number of repeated operands in benchmark programs.

4.3.1 Types of floating point arithmetic

The floating point operations available in the *PISA* architecture are standard *IEEE-754* 32 and 64 bit varieties. The *PISA* architecture implements the minimal set of features to be compliant with the standard, with the addition of a floating point square root (`fpSqrt`) instruction. This calculation is performed on hardware shared with the `fpDiv` instructions. The study of the `fpSqrt` operation is out of the scope of this thesis.

The *IEEE-754* standard defines four rounding modes for the result of arithmetic operations, but unless otherwise noted the ‘round to nearest even’ (Banker’s rounding) is used throughout this thesis.

All floating point operations are performed on a separated floating point coprocessor, with 32 single-precision floating point registers. Double-precision operands are stored in adjacent registers, reducing the overall storage. The *`fpALU`* performs conversions between integer and floating point numbers, compares floating point numbers for conditional instructions, calculates the absolute value of

Table 4.5: Single precision floating point operand values.

Type	NaN	+denorm	−denorm	+Inf	−Inf	-0	+0	Negative	Positive
add	0.00	0.00	0.00	0.00	0.00	0.00	5.40	22.57	72.03
sub	0.00	0.00	0.00	0.00	0.00	0.00	12.93	9.36	77.71
mult	0.00	0.00	0.00	0.00	0.00	0.00	4.27	23.13	72.60
div	0.00	0.00	0.00	0.00	0.00	0.00	0.71	7.80	91.49

a floating point number, and performs floating point addition and subtraction.

4.3.2 Signed and unsigned arithmetic

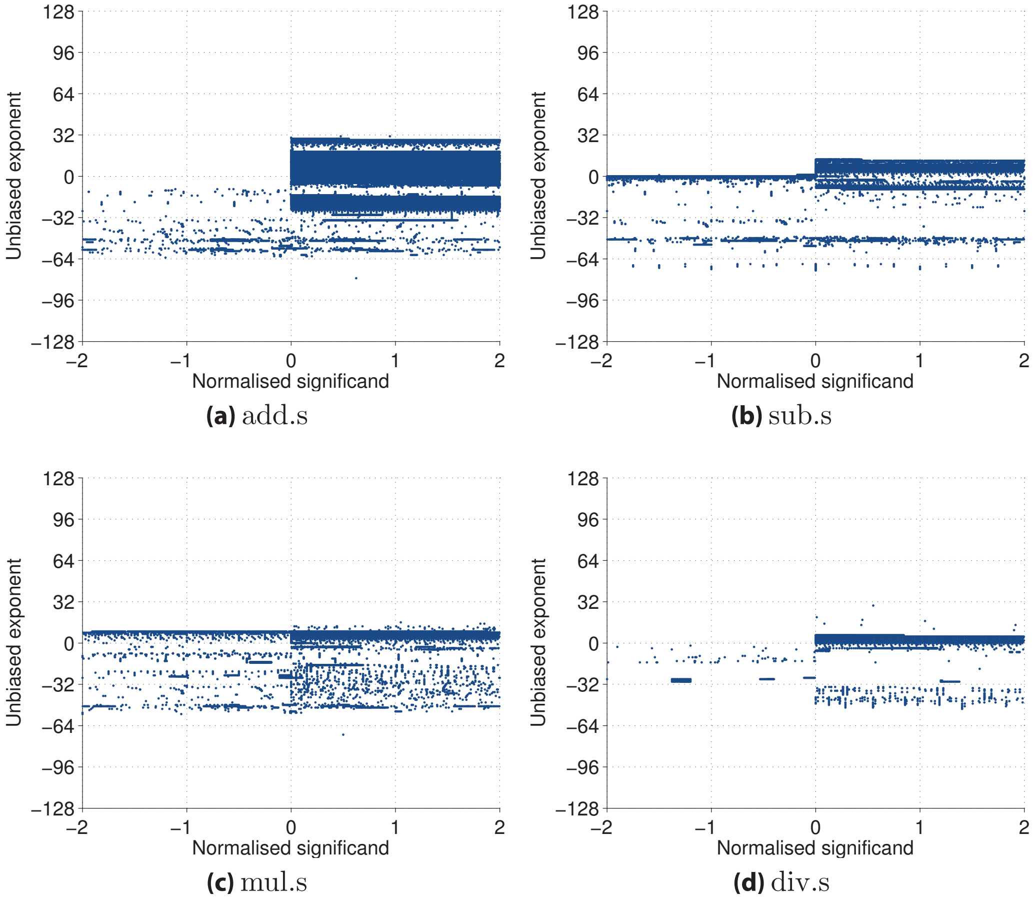
The *IEEE-754* floating point standard defines the number format, including a dedicated sign bit, so all regular numbers are signed. The sign bit is also applied to 0 and ∞, so there are representations of $+0$, -0, $+\infty$ and $-\infty$. The sign bit is ignored for representations that are not-a-number (NaN).

Figure 4.5 shows a scatter plot of addition, subtraction, multiplication and division operands traced from benchmark programs. The y-axis shows the unbiased 32 bit floating point exponent, and the x-axis shows the normalised significand. *IEEE-754* exponents are biased and represented as ≥ 0, but each exponent shown is unbiased. The exponent e is valid in the range $-126 \leq x \leq 127$, but few observed exponents are outside of the range $-64 \leq x \leq 64$. *IEEE-754* floating point significands are normalised, and so the maximum value is < 2. The signed significands shown indicate the polarity of the sign bit.

The plots are more dense on the right-hand side, indicating that positive numbers occur more frequently than negative numbers. The range of negative exponents is greater than positive exponents, indicating the use of small fractional numbers, but the occurrence of numbers greater than unity is higher due to the density of numbers with an exponent ≥ 1.

4.3.3 Operand magnitude

Average bit assertion histograms for single precision floating point operands from benchmark programs are shown in figures 4.6–4.7, highlighting the sign, exponent and significand fields. For each operation, the exponent bits were asserted on average more frequently than the significand bits, and



(a) add.s

(b) sub.s

(c) mul.s

(d) div.s

Figure 4.5: Scatter plot of 32 bit floating point operands in benchmark programs.

Table 4.6: Double precision floating point operand values.

Type	NaN	+denorm	−denorm	+Inf	−Inf	-0	+0	Negative	Positive
add	0.00	0.00	0.00	0.00	0.00	0.00	25.94	36.98	37.08
sub	0.00	0.00	0.00	0.00	0.00	0.00	39.13	28.98	31.89
mult	0.00	0.00	0.00	0.00	0.00	0.00	40.41	25.24	34.35
div	0.00	0.00	0.00	0.00	0.00	0.00	78.09	12.37	9.54

the significand bits are asserted nearly uniformly.

4.4 Performance limits

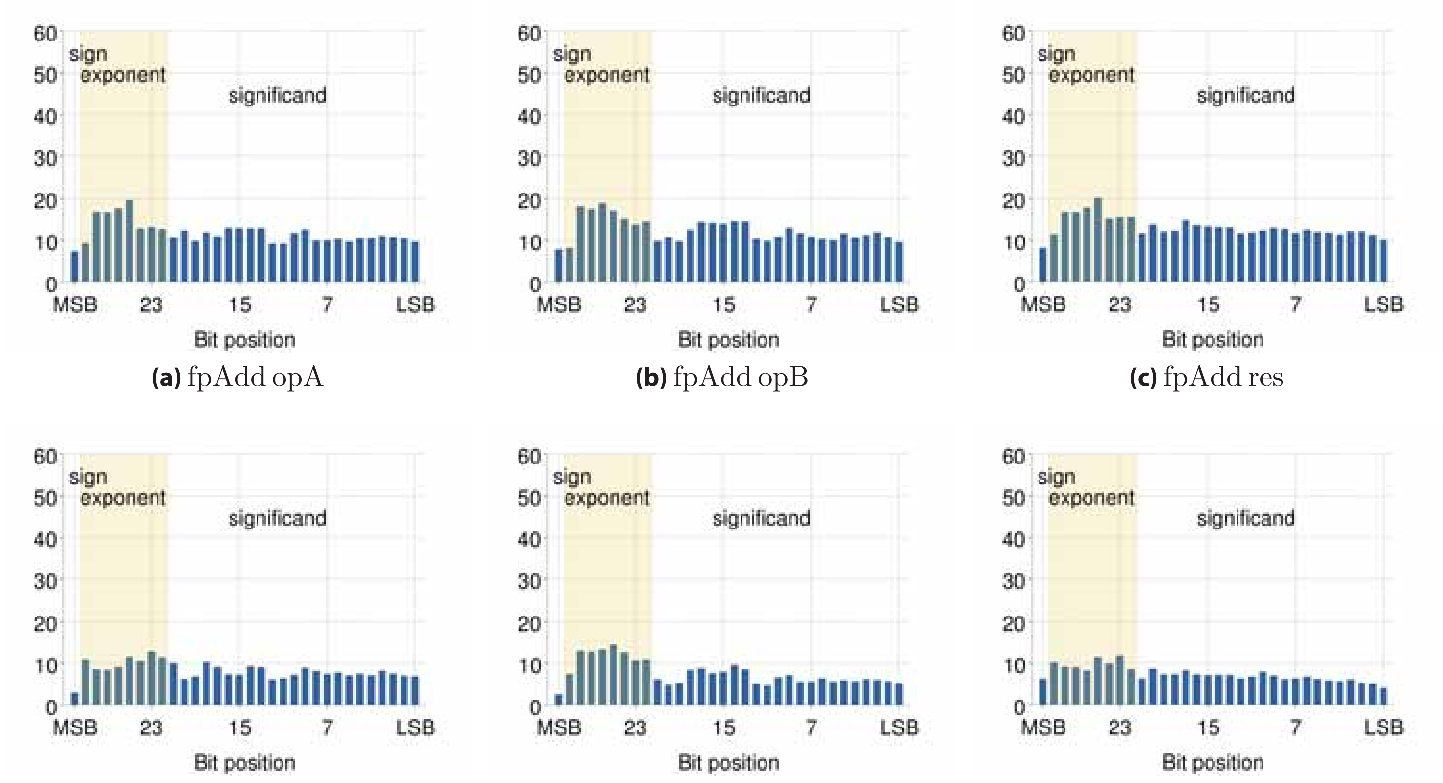
This section presents an initial study of arithmetic data value speculation in a RISC processor pipeline to establish limits of performance, delay and accuracy targets for approximate arithmetic units. The study uses the default *SimpleScalar* configuration described in Section 2.2.2, with the addition of a mechanism to issue dependent instructions of arithmetic operations after the approximate latency.

Two schemes are employed when an erroneous approximation is detected; they are listed below in increasing order of complexity:

arithmetic speculation All instructions younger than the erroneous arithmetic instruction are flushed from the pipeline, and PC is set to the address of the next instruction after the erroneous operation. In the next cycle the front end re-fetches the next youngest instruction.

no front end resteering All instructions younger than an arithmetic instruction are marked as speculative in the decode stage, and remain speculative until the approximate instruction is marked as completed and correct. If the approximate instruction is incorrect, the issue pointer is set to the first arithmetic-speculative instruction and the front end ceases fetching. From the next cycle the younger speculative operations are reissued for each younger operation in the RUU. Fetching is re-enabled in the front end at the previous PC, unless updated by a reissued jump/branch operation.

The *C* source code for *sim-outorder*, the out-of-order superscalar simulator for *PISA*, was modified to determine the correctness of each integer and floating point operation based on a fixed probability, using random numbers. The latency of the exact arithmetic units were maintained at their default values, but approximate arithmetic units were assigned a smaller cycle latency. Each benchmark was run with various settings for probability of correctness and approximate arithmetic unit latency, and the average IPC gain was recorded.



(a) fpAdd opA

(b) fpAdd opB

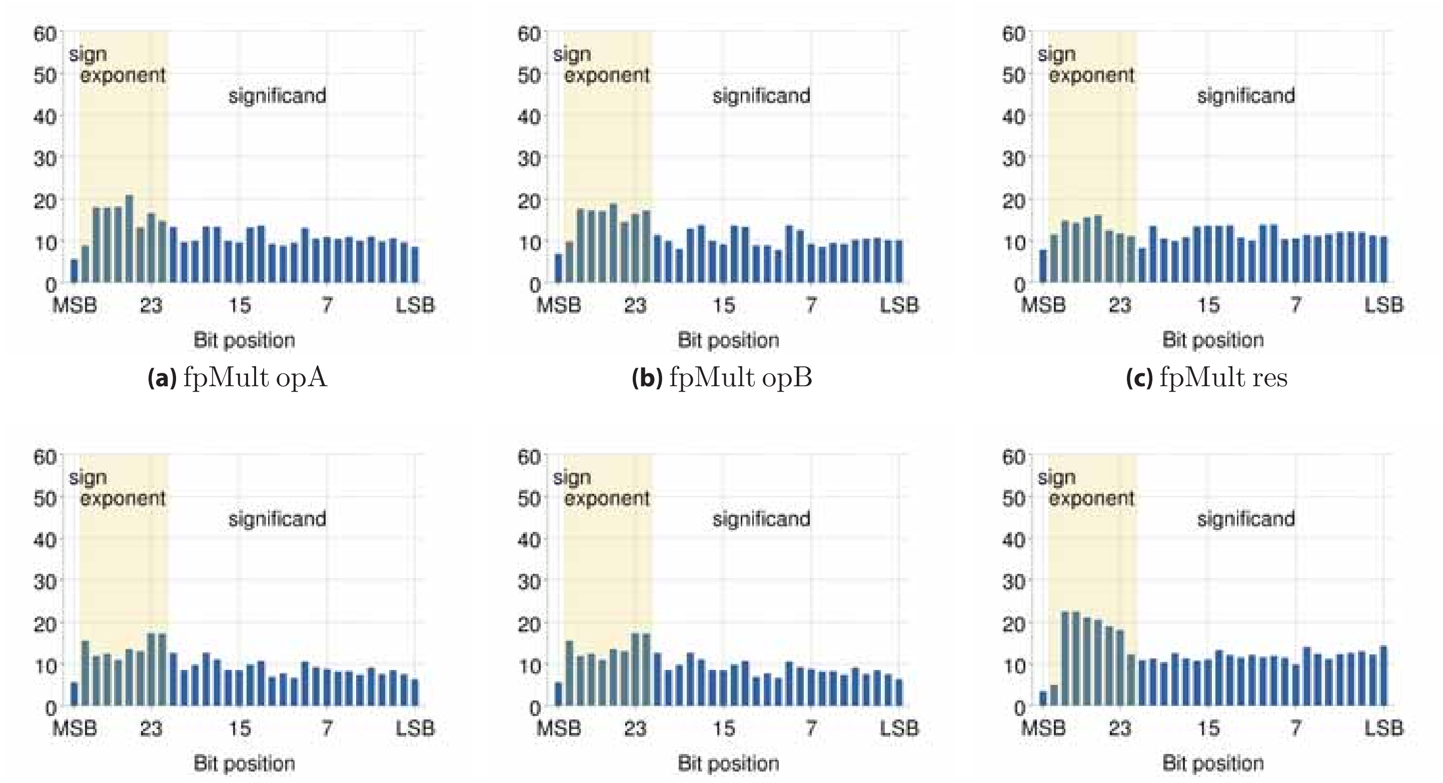
(c) fpAdd res

(d) fpSub opA

(e) fpSub opB

(f) fpSub res

Figure 4.6: Bit assertion distribution of 32 bit floating point addition and subtraction operands from benchmark programs.



(a) fpMult opA

(b) fpMult opB

(c) fpMult res

(d) fpDiv opA

(e) fpDiv opB

(f) fpDiv res

Figure 4.7: Bit assertion distribution of 32 bit floating point multiplication and division operands from benchmark programs.

Table 4.7: Simulated cycle latencies for approximate arithmetic units in *sim-outorder*, when executing the *arithmetic* and *Mediabench* benchmarks.

Arithmetic	Integer		Floating point			
Latency (%)	`mult`	`div`	`add.s`	`mul.s`	`div.s`	`sqrt.s`
10	1	2	1	1	1	2
20	1	4	1	1	2	4
30	1	6	1	1	3	7
40	1	8	1	1	4	9
50	1	10	1	2	6	12
60	1	12	1	2	7	14
70	2	14	1	2	8	16
80	2	16	1	3	9	19
90	2	18	1	3	10	21
100	3	20	2	4	12	24

4.4.1 Simulation parameters

The simulations presented in this chapter have used a discrete scale for the simulated correctness and latency of the approximate arithmetic units. Probability of correctness is defined as a percentage, and the outcome of each simulated calculation is determined by comparison with a random number. Correctness was selected in small granular intervals from 50 % to 100 %. Arithmetic latency was also specified in the simulations, at coarser intervals. Most arithmetic operations only require a few cycles each for the exact calculation, so in practice a significant reduction in latency is required to reduce the approximate arithmetic latency by one cycle.

For the purposes of simulation, Table 4.7 and Table 4.8 show the conversion from arithmetic latency expressed as a percentage to an integer number of machine cycles. The longer *SPEC* benchmarks were simulated at fewer operating points to reduce the number of simulations.

4.4.2 Throughput upper bound

Using the simulator options in *SimpleScalar* it is possible to determine the upper bound of performance gain due to arithmetic speedup. In Tables 4.9–4.11, the effects on execution time of setting all long latency operations to a single cycle is shown. This is the maximum performance increase

Table 4.8: Simulated cycle latencies for approximate arithmetic units in *sim-outorder*, when executing the *SPEC* benchmarks.

Arithmetic	Integer		Floating point			
Latency (%)	`mult`	`div`	`add.s`	`mul.s`	`div.s`	`sqrt.s`
20	1	4	1	1	2	4
40	1	8	1	1	4	9
60	1	12	1	2	7	14
80	2	16	1	3	9	19
100	3	20	2	4	12	24

Table 4.9: Maximum throughput increase of *arithmetic* benchmarks with ADVS when each approximate arithmetic unit operates in a single cycle. The throughput increase is shown as a percentage.

Benchmark	*-O0*	*-O1*	*-O2*	*-O3*	*-O3 -finline*	*-O3 -funroll*
dhrystone	0.42	0.80	0.82	0.82	0.82	0.82
whetstone	12.50	30.95	31.70	31.70	31.70	32.13
calc_pi	7.14	0.22	0.23	0.23	0.23	3.03
matrix_mult	11.90	0.41	0.10	0.10	0.10	0.22
linpack	6.29	12.67	13.42	13.42	5.77	4.48
Average (%)	**9.79**	**5.98**	**4.94**	**4.94**	**4.12**	**3.91**

due to arithmetic speculation possible for each benchmark.

Figure 4.8 shows the increase in throughput when all arithmetic operations are approximated correctly in single cycle. The approximate results are checked with normal arithmetic units at full latency. Tables C.1–C.3 and C.9–C.15 show the number of occurrences of the integer and floating point operations. The maximal speed up possible is the product of the number of occurrences of each operation and the number of cycles saved. Comparing the number of cycles saved to this number shows the proportion of operations that lie on the critical path for each benchmark. The remaining proportion of cycles represents the lost gains due to structural and data hazards.

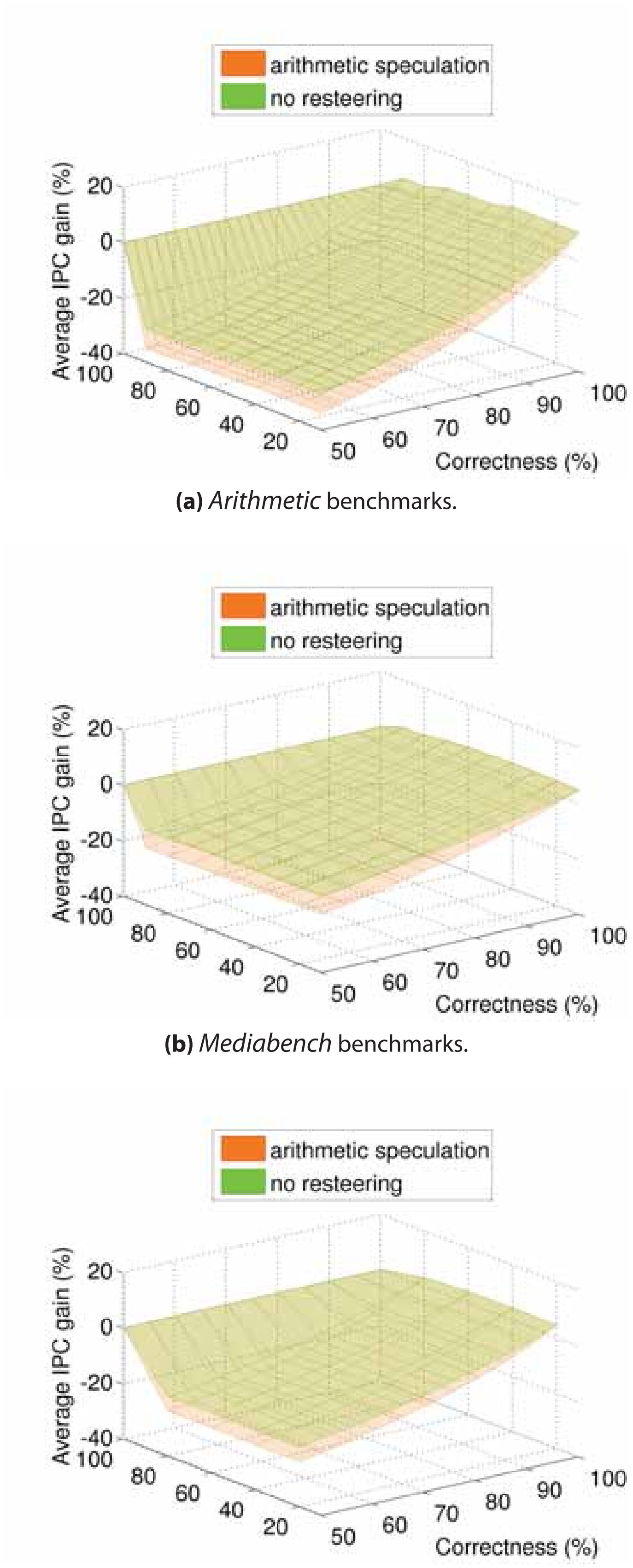
Figures 4.9 and 4.10 show a slice of the delay-correctness profile of Figure 4.8. Figure 4.9 shows the maximum possible IPC gain if each approximate arithmetic unit produces an output in a single

Table 4.10: Maximum throughput increase of *Mediabench* benchmarks with ADVS when each approximate arithmetic unit operates in a single cycle. The performance increase is shown as a percentage relative to the baseline.

Benchmark	*-O0*	*-O1*	*-O2*	*-O3*	*-O3 -finline*	*-O3 -funroll*
ADPCM encode	0.00	0.00	0.00	0.00	0.00	0.00
ADPCM decode	0.00	0.00	0.00	0.00	0.00	0.00
Mesa texgen	4.57	7.30	5.75	5.73	5.73	5.37
Mesa mipmap	5.87	14.50	20.60	15.47	15.47	20.58
Mesa demo	1.99	3.35	3.21	2.92	3.01	3.13
EPIC encode	8.36	17.28	18.48	18.47	18.47	19.82
EPIC decode	5.57	3.91	5.48	4.97	5.49	4.52
G.721 encode	0.60	1.56	1.05	0.63	0.63	0.40
G.721 decode	0.58	0.87	0.59	0.45	0.43	0.51
RASTA	4.91	7.17	6.93	7.05	7.05	6.96
JPEG encode	0.03	0.50	0.42	0.41	0.41	0.45
JPEG decode	0.34	0.39	0.43	0.42	0.42	0.36
ghostscript	8.91	9.59	7.99	9.95	9.95	8.89
mpeg2play encode	0.92	1.84	1.85	1.85	1.85	1.27
mpeg2play decode	3.83	9.16	7.34	7.36	7.36	3.28
Average (%)	**2.74**	**4.56**	**4.73**	**4.48**	**4.51**	**3.99**

Table 4.11: Maximum throughput increase of *SPEC* benchmarks with ADVS when each approximate arithmetic unit operates in a single cycle. Performance gain is shown as a percentage relative to the baseline.

Benchmark	*-O0*	*-O1*	*-O2*	*-O3*	*-O3* *-finline*	*-O3* *-funroll*
172.mgrid	19.30	16.49	7.53	7.53	7.53	7.53
175.vpr	0.86	1.11	1.75	1.25	1.25	1.25
177.mesa	4.43	6.66	7.83	6.38	6.38	6.38
197.parser	0.30	0.83	0.85	0.77	0.78	0.78
173.applu	3.23	10.76	8.53	8.53	8.53	8.53
200.sixtrack	0.00	0.00	0.00	0.00	0.00	0.00
188.ammp	3.92	6.61	4.17	4.21	4.21	4.21
164.gzip	0.01	0.01	0.01	0.00	0.00	0.00
171.swim	4.67	11.07	9.35	9.35	9.35	9.35
179.art	1.39	0.90	1.51	2.74	2.74	2.74
300.twolf	4.79	5.02	6.21	6.24	6.24	6.24
301.apsi	8.77	11.73	8.96	8.83	8.83	8.83
168.wupwise	4.29	5.88	6.16	6.16	6.16	6.16
256.bzip2	0.00	0.00	0.00	0.00	0.00	0.00
gcc	0.07	0.13	0.12	0.13	0.14	0.14
183.equake	7.28	5.53	7.02	7.24	7.24	7.24
Average (%)	**3.30**	**4.32**	**3.65**	**3.61**	**3.62**	**3.61**



(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

Figure 4.8: Average IPC gain for benchmark programs running in an out-of-order RISC pipeline, with arithmetic data value speculation (correctness vs. latency).

machine cycle. Figure 4.10 shows the maximum possible IPC gain if each approximate arithmetic unit operates with 100 % correctness.

4.4.2.1 Use of hardware resources

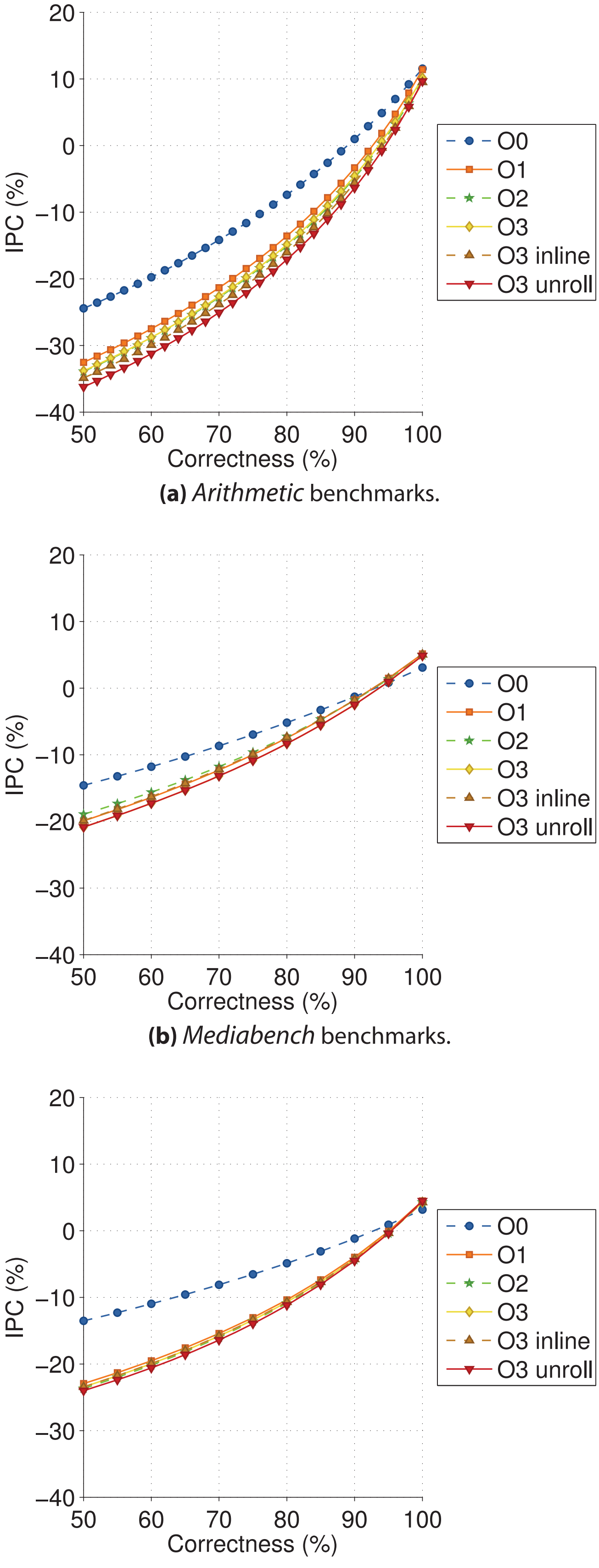
ADVS requires additional hardware to calculate approximate and exact results, and perform the speculation. In this section simulations are performed to determine if simply providing more parallel exact units can result in increased throughput. The advantage of more exact units is that the additional design effort is lower, and there is no performance loss due to speculation. Conversely, *ADVS* introduces a possible throughput penalty if approximations are too frequently incorrect.

In these simulations the approximate unit from the *ADVS* model were substituted for an extra exact unit, but no additional ports were introduced to the register file.

The simulations were conducted with the same parameters as in Section 4.4.2, and approximate instruction latencies increasing in 10 % increments. In the previous simulations the number of approximate arithmetic units was such that no stalls would be caused by the unavailability of an exact unit to check the result of an approximate unit. The number required is a function of the latency and repeat rate of the approximate and exact units. In this model, all of the approximate units were substituted for exact units. This is often only one additional unit, because the units are mostly pipelined and can issue in back to back cycles.

All of the benchmarks in the *arithmetic*, *Mediabench* and *SPEC* sets were executed, and the average throughput gain was calculated compared to the default configuration. A negligible throughput gain ($< 0.01\,\%$) was observed over all of the benchmark sets. Despite the increased available bandwidth, it was very rare that two of the same operation would be ready to issue in the same cycle. In order for a significant throughput gain to be observed, a higher density of identical, independent arithmetic operations was required. Dependent younger operations do not benefit from the additional parallel hardware.

Increasing the number of arithmetic units did not significantly increase throughput because most arithmetic operations do not occur very frequently. It is therefore unlikely that two independent instructions will be ready for issue at the same time, causing a stall due to unavailable resources. Because of the long latency of the arithmetic operations, it is better to allocate similar hardware resources to reducing the average latency through speculation.

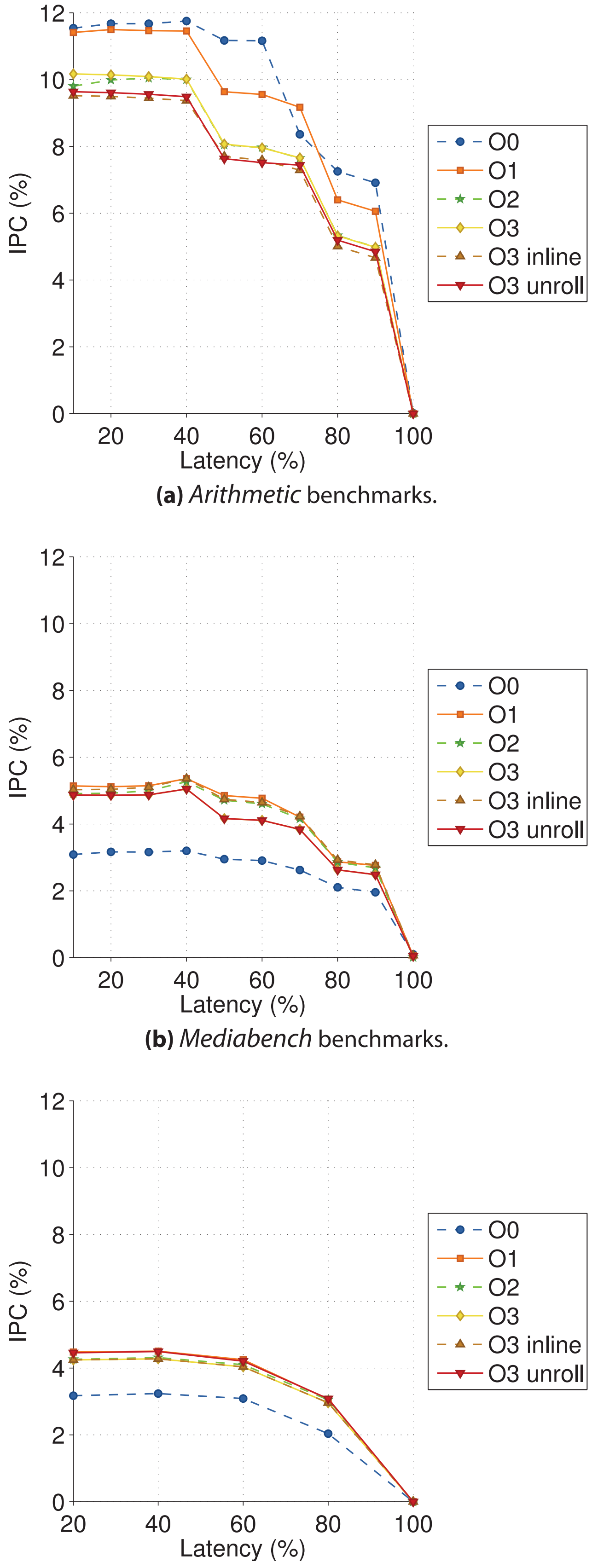


(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

Figure 4.9: Maximum possible throughput gain using arithmetic data value speculation. Each approximate arithmetic unit has a latency of 1 cycle.



(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

Figure 4.10: Maximum possible IPC gain using arithmetic data value speculation. Each approximate arithmetic unit has 100 % correctness.

Table 4.12: Average percentage reduction in execution cycles of the benchmarks running in *sim-outorder* without ADVS.

Benchmark set	*-O1*	*-O2*	*-O3*	*-O3* *-finline*	*-O3* *-funroll*
arithmetic	54.14	56.35	56.59	57.71	59.29
Mediabench	42.22	44.06	44.04	43.94	44.04
SPEC	53.98	55.92	56.60	56.60	56.60

4.4.3 Compiler optimisation effects

Source code compilers typically come with many options to direct the compilation and linking of computer programs. The *SimpleScalar* tool set uses a modified version of *gcc* to compile *C* programs.

gcc has many options that can be enabled in addition to four optimisation levels (*-O0*··· *-O3*). Each optimisation level introduces additional strategies to attempt to improve performance over lower optimisation levels, balanced against increases in the binary size.

Table 4.12 shows the average reduction in execution time, measured in machine cycles, for the benchmarks executed at different compiler optimisation levels in the out-of-order simulator, *sim-outorder*, without ADVS enhancements. Performance is measured relative to a benchmark binary compiled without any compiler optimisations, optimisation level zero (*-O0*).

Each benchmark exhibits a large performance gain when level one optimisation (*-O1*) is enabled, and incremental performance improvements for level two (*-O2*), level three (*-O3*), and level three optimisation with inlining (*-O3 -funroll-loops*) and loop unrolling (*-O3 -finline-functions*). Code inlining and loop unrolling do not guarantee a performance improvement when enabled—both optimisations have positive and negative impacts on average when applied to different benchmark data sets. A detailed description of *gcc*'s compiler optimisation levels is provided in Appendix A. Increasing the optimisation level, particularly for optimisations such as inlining and loop unrolling can increase the binary size. Table 2.5 shows the effect of optimisation level on binary size.

Table 4.12 shows that increasing the compiler optimisation level is detrimental to the *relative* performance gain in the *ADVS* system, however savings to number of cycles (in *absolute* terms), increase with optimisation level. Figure 4.11 shows the number of cycles required to execute the *172.mgrid* benchmark from the *SPEC* set with ADVS enabled, at different optimisation levels.

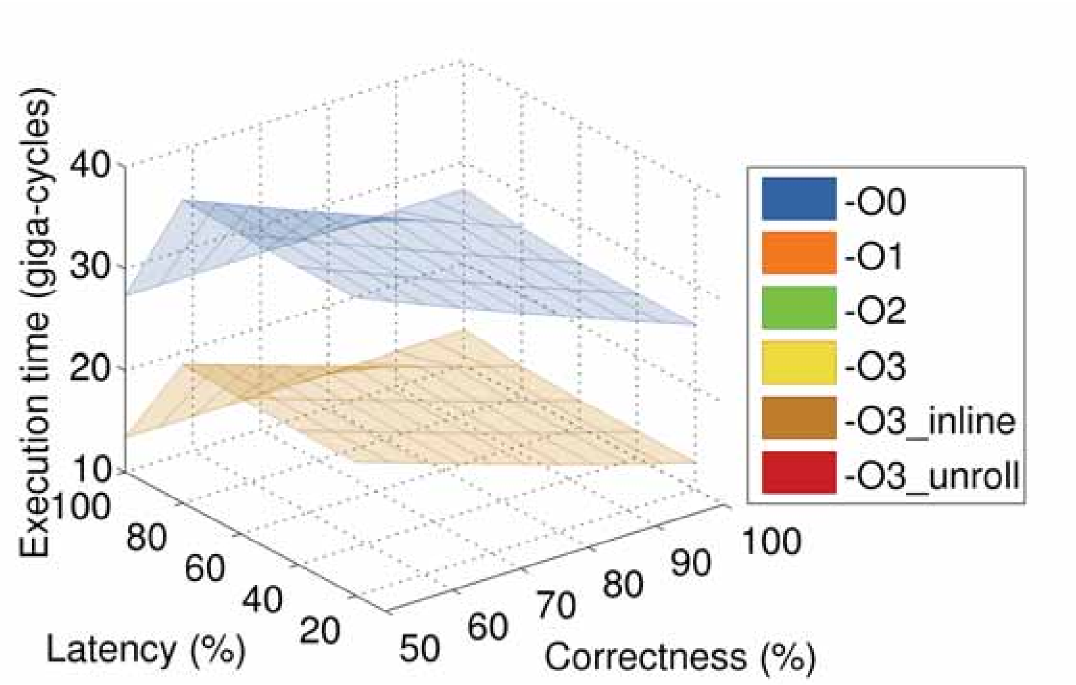


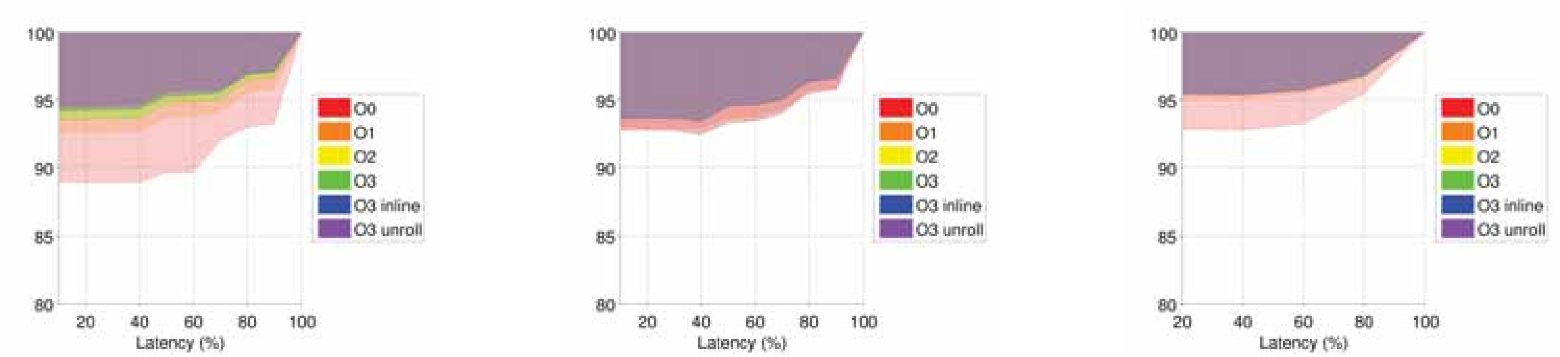
Figure 4.11: Number of execution cycles for *SPEC 172.mgrid* at different optimisation levels. All of the optimised benchmarks except '-O0' are superimposed.

Optimised binaries, with optimisation level at -*O1* or greater require significantly less execution time than unoptimised binaries performing the same tasks. Increasing the optimisation level above -*O1* yields incremental performance improvements in a regular pipeline, but in an ADVS enabled pipeline, the returns are diminishing for increased optimisation level. In terms of executed cycles, there is little benefit for increasing the optimisation level above -*O1* using *gcc* version 2.6.3.

4.4.4 Delay-correctness lower bound

Figure 4.12 shows the feasible delay-correctness region for approximate hardware used in the arithmetic speculation scheme. The region was constructed by linearly interpolating the delay vs. correctness plots shown in Figure 4.8 to intersect the baseline throughput. Figure 4.13 shows the feasible delay-correctness region when using the basic 'arithmetic speculation' flushing scheme, where all younger operations are flushed.

The throughput is much more sensitive to the probability of correctness than the latency of the approximate hardware—feasible approximate designs to be used in ADVS will require a high probability of correctness to avoid being detrimental to performance. Additionally, the feasibility region is smaller with greater compiler optimisation; as the code density increases so does the exposure to

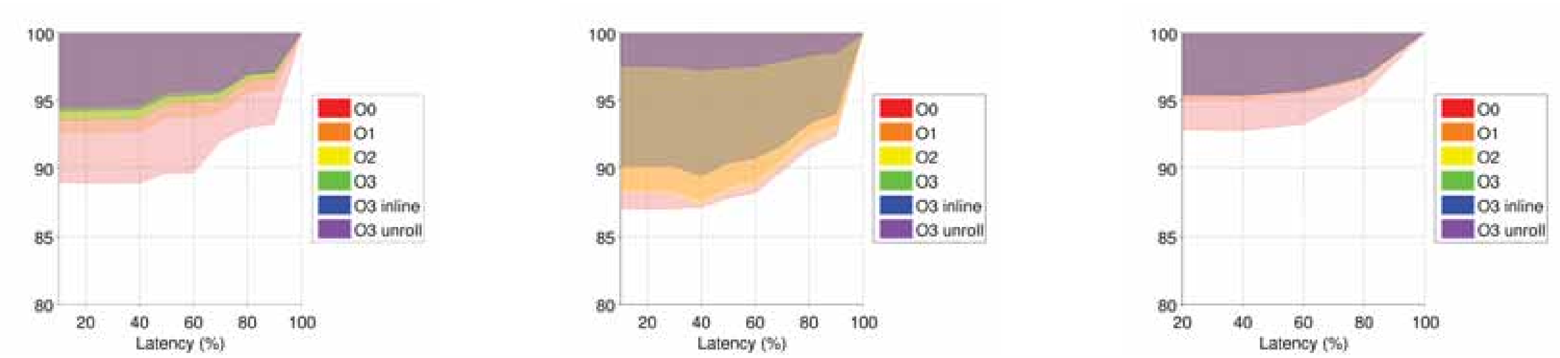


(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

Figure 4.12: Delay-correctness feasibility region using the basic arithmetic flushing scheme.



(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

Figure 4.13: Delay-correctness feasibility region using the no-resteering flushing scheme.

an incorrect approximation flush, because long memory access times due to jumps and branches are masked.

4.5 Conclusion

This chapter has presented a study of benchmark programs including an analysis of the frequency of occurrence of each machine instruction, and summaries of the arithmetic hardware and instructions available in the *PISA* architecture.

Arithmetic operands were traced from the simulated execution of the benchmarks, and used to profile typical operands. It was found that most arithmetic operands in signed arithmetic are positive, and that most floating point operands are used to represent positive fractional numbers less than one.

Finally, the benchmarks were simulated in an ADVS enabled pipeline for a range of correctness and approximate unit latencies, and the average improvement in IPC was shown. Two different flushing schemes were employed when an approximated arithmetic result is detected to be incorrect. By reissuing flushed instructions from the front end, avoiding a resteer of the fetch unit, a consistent favourable difference in IPC is obtained.

Measuring the average IPC gain with a minimal single-cycle latency for approximate arithmetic units yields the maximum possible increase in throughput with respect to probability of correctness. Similarly, the maximum increase in throughput was shown for the case of perfect approximation, with 100 % correctness.

Can *ADVS* improve the performance of a generic RISC processor? Yes, it can shown that the behaviour of typical applications is understood, and the characteristics of typical program data can be exploited to minimise the probability of pipeline flushes. This chapter showed that the delay-correctness break-even point requires a high probability of correctness of the arithmetic units to make *ADVS* feasible. In general, a probability of correctness of over 95 % is required for an approximate arithmetic unit that operates with a latency 80 % of an exact arithmetic unit.

This chapter has shown that *ADVS* is feasible, assuming probabilistic outcomes for the correctness of the approximate arithmetic units. In later chapters approximate arithmetic units are developed and modelled so that the data dependence is captured.

Chapter 5

Preliminary Experiments

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Turing (1912–1954)

This chapter presents the results of preliminary experiments in which basic arithmetic units are made approximating through simple modifications that shorten the critical path by selectively removing circuit elements. It is difficult to yield a high probability of correctness by omitting logic because of the many dependencies that intermediate results have on input operands.

Circuit optimisations for arithmetic units are well understood, and techniques such as exploiting parallelism and recoding have been applied so that incremental reductions in latency are extremely difficult, and often costly. Arithmetic approximation can be used to reduce the latency of the fastest arithmetic circuits, *without maintaining the functional correctness of the circuit*. A good approximate circuit:

- has a lower latency than the exact circuit from which it was derived; and
- *P*, the probability of correctness of the circuit, is high. In this thesis, the average probability of correctness is often referred to as the *correctness* of the circuit.

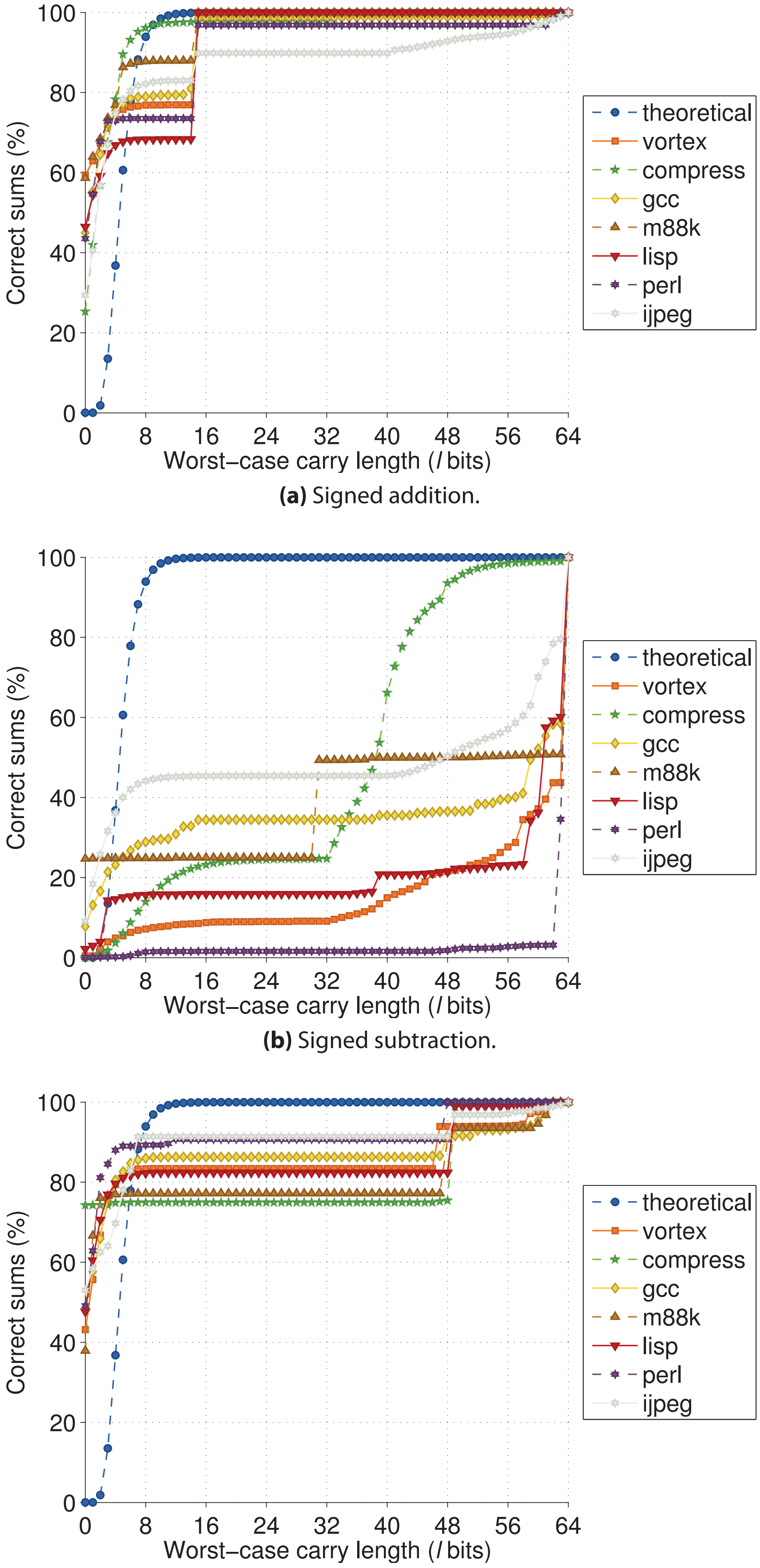
The appropriate value of *P* is subjective; it is up to the designer to balance the correctness/delay trade-off for a particular application. The target correctness for feasibility in an *ADVS* enabled system was shown to be approximately 95 % in Chapter 4.

The experiments in this chapter demonstrate the difficulty in achieving a high correctness with simple modifications to basic arithmetic circuits. Basic arithmetic circuits are conceptually easier to understand, so approximating them can give insights, that are valuable for attempting to approximate faster, more complex circuits. In Chapter 6 and Chapter 7 approximate integer multipliers and dividers are presented, capable of 90 % correctness for benchmark inputs.

5.1 Validation of average worst-case carry length

In this section some results from the literature are verified by simulation, including the empirical results for AWCCL in adders (see Section 3.2). The simulations are performed using *SimpleScalar* with *Mediabench* and *SPEC* benchmarks. Other simulations are conducted to determine typical properties of commonly executed arithmetic instructions.

64 bit addition and subtraction operands were traced from the *SPEC CPU95* benchmarks and used to determine *P* for Liu and Lu's adder and the AWCCL [Kelly and Phillips, 2005]. Figure 5.1 shows the correctness of an 64 bit adder for signed addition (`add`, `addi`, `addiu`), subtraction (`sub` and `subi`) and addressing calculations for load and store operations.



(a) Signed addition.

(b) Signed subtraction.

(c) Memory addressing additions.

Figure 5.1: Proportions of correct simulated *SPEC CPU95* additions with a maximum carry length of *l* bits.

The theoretical probability of correctness shown in the graphs in Figure 5.1 was calculated with the *BackCount* algorithm, assuming uniform random inputs. The correctness of Liu and Lu's adder for add instructions was higher than expected when the worst case carry length l was small. However, some benchmarks repeat many operations with identical operands. In some benchmarks the carry length l must be very high to achieve an average correctness greater than 90 %. This is because most benchmarks repeat many operations, and some benchmarks repeat additions with a long AWCCL.

When an adder is used to perform an unsigned subtraction on two binary inputs, the subtrahend is inverted, and the carry-in bit C_{IN} is set to one. If both inputs are zero, then one input operand would consist of N ones. Due to the C_{IN} bit, the carry chain for the addition (subtraction) would propagate the entire length of the input. Likewise, subtraction operations involving small subtrahends produce large carry chains.

For subtraction operations, the Liu and Lu adder performs much less well than the theoretical correctness. It is possible to approximate subtraction results by performing the full operation on l bits out of N, and then sign-extending the result. This has the effect of reducing the calculation time for additions which result in very long carry chains. However, by sign extending past l bits, the range of possible accurate results is reduced, as the bits of greater significance will be ignored.

Subtraction operations formed less than 2.5 % of all integer arithmetic operations in the *SPEC INT* benchmarks. This is an important design consideration for the implementation of approximate arithmetic units. If in practice subtraction operations were common and not easily approximated, it would be best not to use them in a speculative execution scheme.

Figure 5.2 shows the average correctness of unsigned addition operations from the standard benchmarks. For a maximum carry length of 8 bits or more, all benchmark average over 90 % correctness.

5.2 Simple modifications to basic arithmetic circuits

In this section the result of simple experiments with basic multipliers and dividers are presented. Each arithmetic unit is modified to yield an arithmetic result faster, and the effect on P is shown. It is established that simple omissions of logic to reduce the critical path delay have a significant adverse effect on the correctness. The simple arithmetic units used are representative of the complex relationships and dependencies between the input and outputs of arithmetic operations.

In multipliers, the multioperand addition of partial products can generate many carries over the

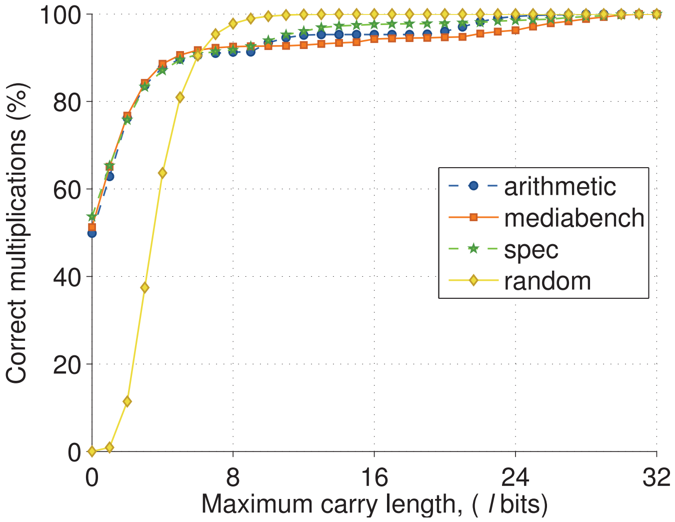


Figure 5.2: Proportion of correct sums in Liu and Lu with a worst-case carry length of l bits. The results were aggregated over each benchmark set used in this thesis.

entire circuit that all contribute to the correctness and latency. The overall delay of a multiplier can be considered as the sum of the delay of partial product generation, partial product addition, and final carry propagate adder (CPA). The partial product addition can occur in a carry save adder (CSA) like in a tree multiplier, or using regular full adders.

In dividers, the necessity of calculating the partial remainder (or part thereof) imposes a similar problem.

This section presents analyses of basic arithmetic circuits, and investigates simple modifications and their affects on correctness with benchmark inputs.

5.2.1 Carry propagation in array multipliers

Figure 5.3 shows a simple array multiplier where one partial product is generated per row. The partial products are then cumulatively summed until the final product result is output. The worst case delay is determined by a carry rippling through the each row in series.

Let us consider carry propagation through 32×32 bit array multipliers. Figure 5.4 shows the probability that each *full-adder* asserts its C_{OUT} bit. Each *full-adder* in an array multiplier outputs the cumulative sum of partial products, generated from the *AND* of input operand bits. Uniform random input bits have a 50 % chance of being asserted, hence the *AND*-ed partial product bits have

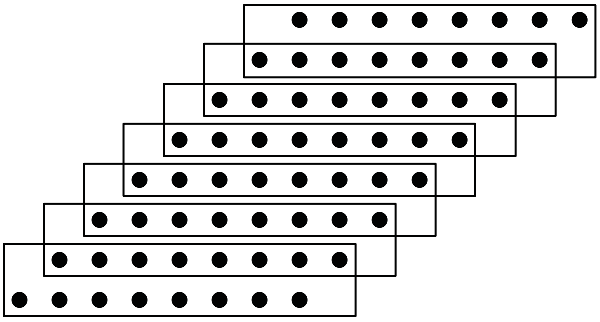


Figure 5.3: The first stage of partial product accumulation in an array multiplier. Each dot represents a bit and each row is a partial product. The multiplier calculates the sum of the partial products by adding the partial product bits in columns, and carries overflow into the group below. The rows cannot be added independently because they require the carry from the row above, hence they are shown to overlap.

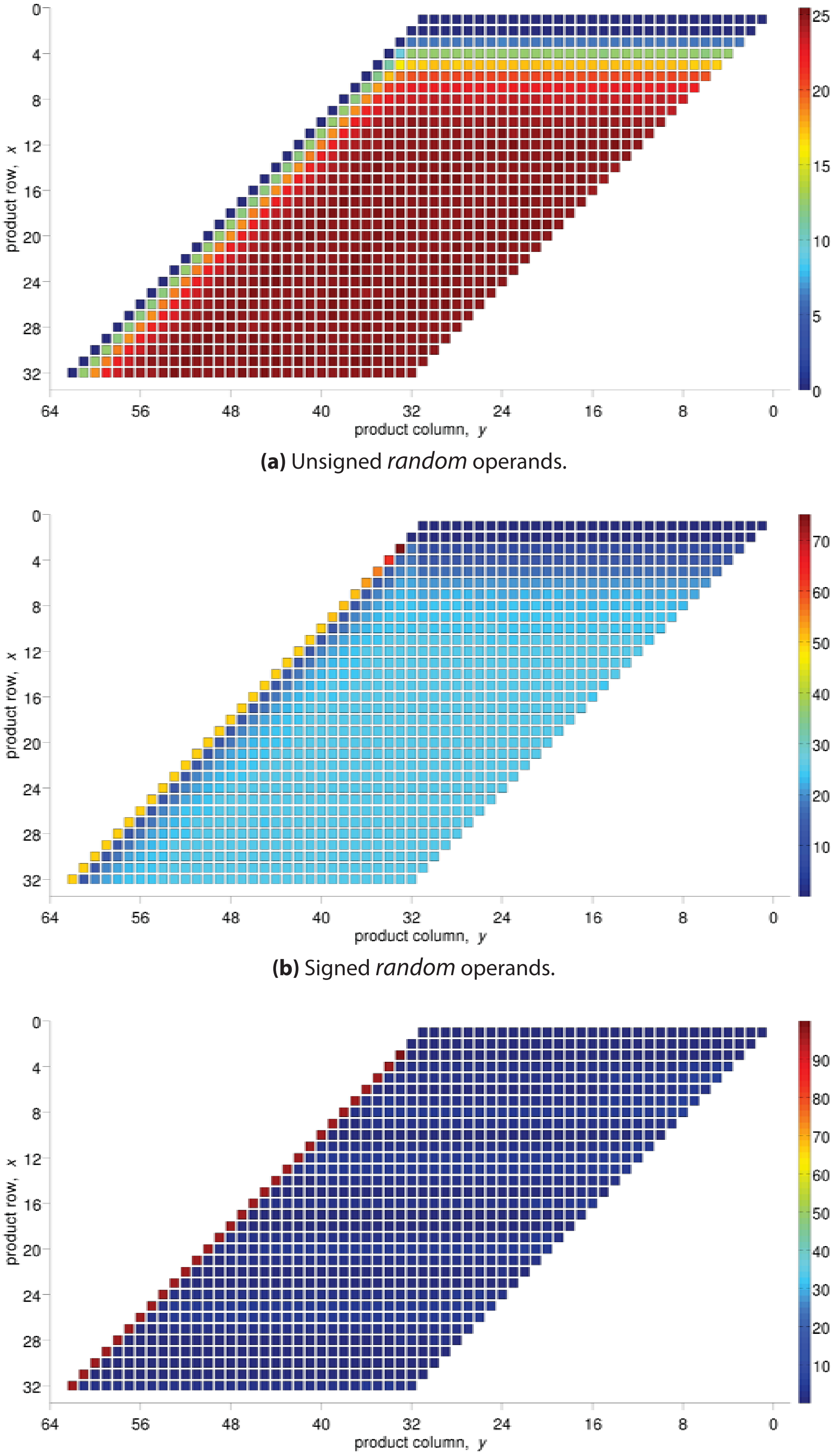
a 25 % change of being asserted. Each *full-adder* outputs a C_{OUT} with roughly 25 % probability (see Figure 5.4a). The probability of assertion of most *full-adder*s is similar for signed operands, but the sign bit in Figure 5.4b affects the probability of assertion of the leading bits of all the partial products. This is discussed in more detail in Chapter 6.

The operand distribution of benchmark operands is different to uniform random numbers, so many of the *full-adder*s of lower significance in Figure 5.4c for *SPEC* operands are asserted the most frequently, however the average is less than 25 %. As carry outputs occur nearly uniformly on average, and have a non-negligible probability of assertion, preventing any particular carry from propagating is likely render the output product incorrect.

It is difficult to calculate exact probability of a carries propagating through a long subsection of the critical path. However, severing the critical path to reduce delay will also affect many paths required for correct results, so the probability of correctness would diminish significantly. Hence logical incompleteness in an array multiplier is unlikely to achieve high correctness.

5.2.2 Tree multipliers without a CPA

Tree multipliers, such as Wallace and Dadda multipliers, are fast because they sum partial products column-wise in a carry-save adder tree, so columns can be summed independently. In the final stage, the saved sum and carry bits of the adder trees form two input operands to a carry-propagate adder. The worst case delay of the CPA is determined by the propagation of the carry through the



(a) Unsigned *random* operands.

(b) Signed *random* operands.

(c) Signed *SPEC* operands.

Figure 5.4: Observed average assertion, as a percentage, for C_{OUT} bits in 32×32 bit array multipliers.

Table 5.1: Proportion of correct multiplications in a 32 bit Wallace tree multiplier without a CPA.

Benchmark	Unsigned (%)	Signed (%)
arithmetic	0.00	0.00
Mediabench	14.83	0.00
SPEC	8.15	0.04
random	0.55	0.00

entire operand width.

Trivial multiplications such as multiplication by zero, one or powers of two might not generate a stored carry, and hence will not require the final CPA. If the CPA can be abandoned or shortened, a significant delay saving can be made.

The number of input bits to each exact multioperand CSA tree, and the final CPA are roughly the same, and both the CSA tree and the CPA can operate at best in $O(\log_2(N))$ time [Parhami, 2000].

Figure 5.5 shows an 8×8 bit Wallace multiplier, constructed with `full-adder`s. The partial products in the first level are normally reduced in a tree structure until they form two input rows to the final CPA. In this case, the CPA is discarded. The lower product bits in blue are determined before the CPA stage. The upper product bits in green are the output sum bits of the final `full-adder` cells in the tree, and the C_{OUT} bits in orange are discarded. The output product is the concatenation of the upper sum bits and lower product bits.

Table 5.1 shows the average correctness of unsigned multiplications calculated in a 32 bit Wallace tree multiplier without a CPA. The least significant bits that can be exactly calculated without requiring the CPA are used, and the upper bits of the product were taken from the A operand to the CPA. The B operand bits were discarded. The *SPEC* and *Mediabench* benchmarks yielded higher average multiplier correctness, P_M, than the *arithmetic* and *random* operands. In both cases a single benchmark containing a high proportion of trivial operations, such a multiply by zero or one, biased the results. Most benchmarks yield few multiplications with many correct operations.

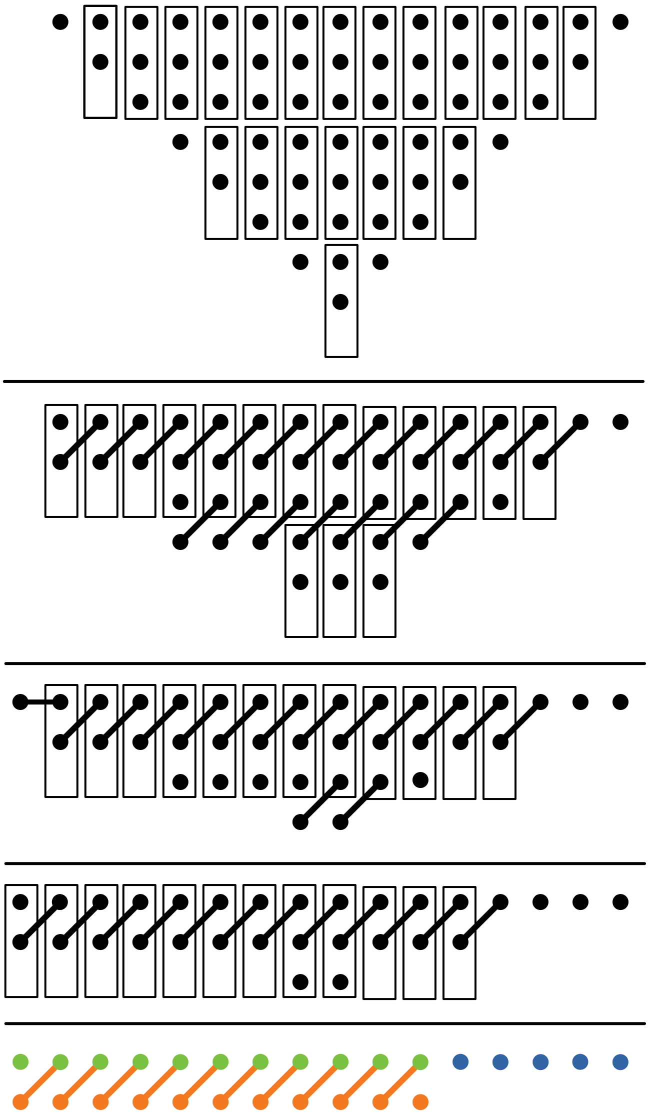


Figure 5.5: An 8×8 bit Wallace tree multiplier, with no final carry propagate adder. Lower product bits generated before the last level in the tree are exact. Counters in the final level of the tree generate sum and carry bits, however the carry bits are discarded.

5.2.3 Array multiplication with an l bit CPA

The final CPA in an array or tree multiplier is a two operand addition. Instead of removing the CPA entirely, a small gain to delay might be made by restricting the length of the maximum carry length. The carry length can be restricted by using an l bit ripple carry adder, or by using an N bit Liu and Lu adder with l bit maximum carry length. Figure 5.6b shows an array multiplier with a short l bit ripple carry adder. The bits of the product more significant than the l bits from the CPA are taken to be the sum output of the final row in the array multiplier.

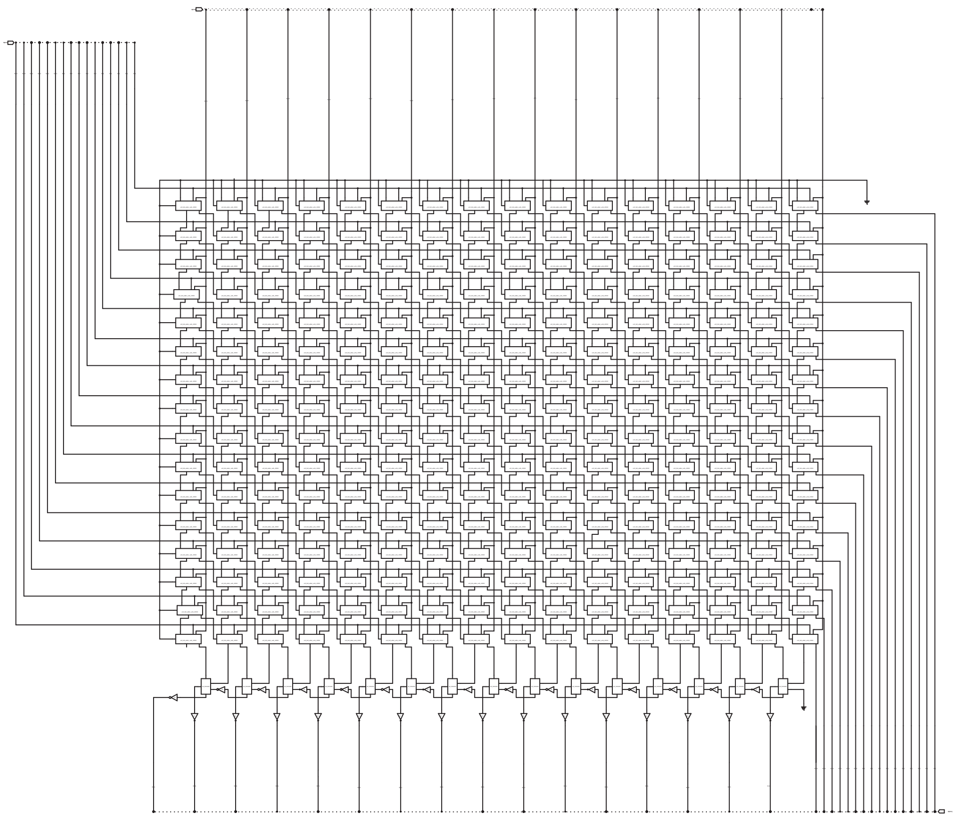
Figure 5.7 shows P_M of a 32 bit array multiplier, where the final CPA adder is a l bit ripple carry adder. The upper 32-l bits are discarded. There are no unsigned integer multiplications in the *arithmetic* benchmarks.

The product is correct when the l bit ripple carry adder is long enough to capture the MSB of the partial products. Uniform random numbers are likely to have asserted bits in the upper operands, so many bits in the CPA are required for high correctness. Benchmark operands are typically smaller in magnitude, so only a short ripple carry CPA is needed to correctly calculate the product. Figure 5.7 shows that the longest carry lengths are mostly greater than 27 bits.

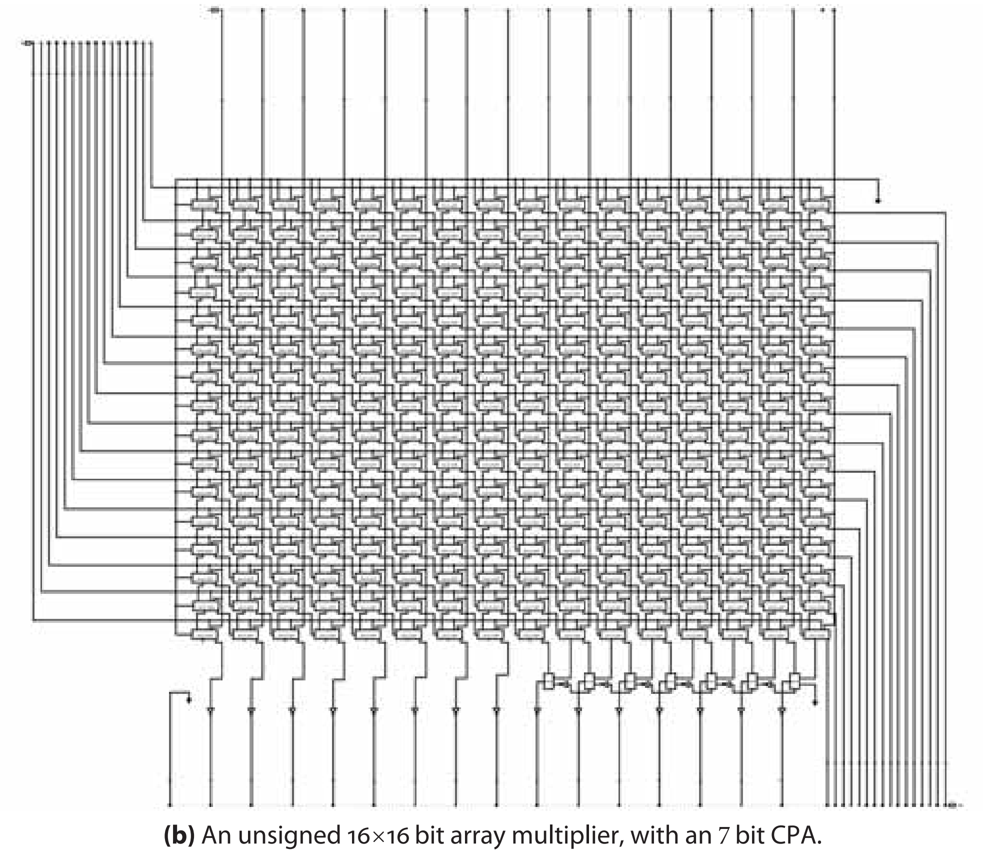
Figure 5.8 shows the effect on P_M of using Liu and Lu's adder with l bit carry segment for the 32 bit CPA in a 32 bit array multiplier.

A 32 bit multiplier multiplies two 32 bit numbers to form a 64 bit product. An array of 32 rows of 32 bit partial products is generated from the input operands. The logic depth for the least significant 32 bits of the product is shorter than the upper 32 bits, so they are determined exactly. The upper 32 bits are summed in CPA to remove the redundancy of the carry-save form of the upper 32 bits.

Benchmark operands are typically less than 16 bits wide, so the result usually fits within the lower 32 bits of the 64 bit product. Correctness is high because no bits in the upper 32 bits are discarded. Random inputs are typically large in magnitude, so the products require the correct calculation of the upper 32 bits in the CPA. Compared with the ripple carry CPA in Figure 5.7, the Liu and Lu CPA adder will correctly add the upper product bits, provided the maximum carry length is less than l bits. Correctness is much higher when the maximum carry length is allowed to be any carry chains of length l bits, instead of just the lower product carry chains.



(a) An array multiplier with a ripple carry adder for the carry propagate adder.



(b) An unsigned 16×16 bit array multiplier, with an 7 bit CPA.

Figure 5.6: Unsigned 16×16 bit array multiplier with modified CPA adders.

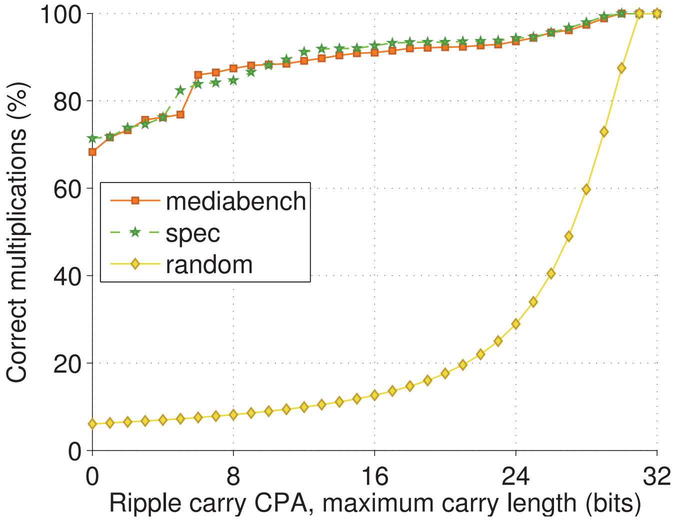


Figure 5.7: Correctness of an unsigned array multiplier using a l bit ripple carry adder as the CPA. The lower l bits of the product can be calculated without affecting the critical path, but the upper $32 - l$ bits of the partial product carries are discarded.

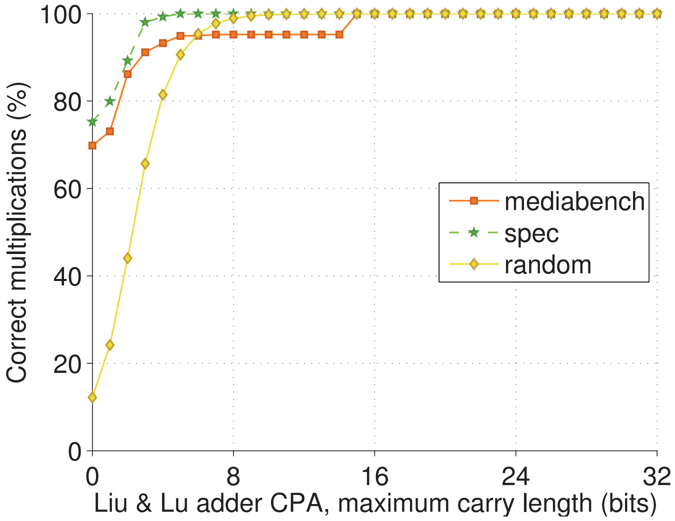


Figure 5.8: Proportion of correct multiplications when using a l bit Liu and Lu adder as a CPA in an unsigned array multiplier. The upper $32 - l$ bits are discarded.

5.2.4 Restoring division with *t* cycles

Restoring division is a very simple division method. In each division round a full subtraction of the shifted partial remainder is performed to determine the quotient digit. If the trial difference is negative, the partial remainder must be restored to it's previous positive value. Restoring division operates only on unsigned operands.

One radix *r* quotient digit is calculated per round in restoring dividers; usually one division round occurs per clock cycle so that the partial remainder can be latched between rounds. The sign of the final remainder is defined be the same as the sign of the dividend, and the magnitude of the remainder must be smaller than the magnitude of the divisor. The partial remainder is also maintained as a positive number so that it is guaranteed to have the correct sign when the algorithm terminates.

Integer division is guaranteed to produce a quotient that has a magnitude less than or equal to the magnitude of the dividend if the dividend is non-zero. Restricting the number of division rounds can reduce the latency of the division operation from the uncommon worst case. Figure 5.9 shows the proportion of correct quotient calculations with a maximum number of division rounds, when a restoring divider is used. The most significant quotient bits are calculated first, and the remainder is ignored.

Few benchmark quotients can be correctly calculated without all of the 16 division rounds, because the LSB is not calculated until the final round. Some unsigned division quotients are correctly calculated with fewer than 16 rounds because the dividend is zero. A common case in signed benchmark division occurred when the product was a power of two—the lower bits are zero, hence the approximation is correct.

5.2.5 Non-restoring division with *t* cycles

Non-restoring division is similar to restoring division, but the requirement to maintain a positive partial remainder is removed. Additional bits are required to store the sign of the non-restored partial remainder, allowing operation on signed two's complement operands.

The simulations of Section 5.2.4 were repeated with a non-restoring divider. Figure 5.10 shows P_D of correctly calculating a correct 16 bit quotient in 16 cycles by radix 2 division of a 32 bit dividend by a 16 bit divisor. The lower 16 bits of the divisor d of operands traced from the benchmarks were used in the simulation.

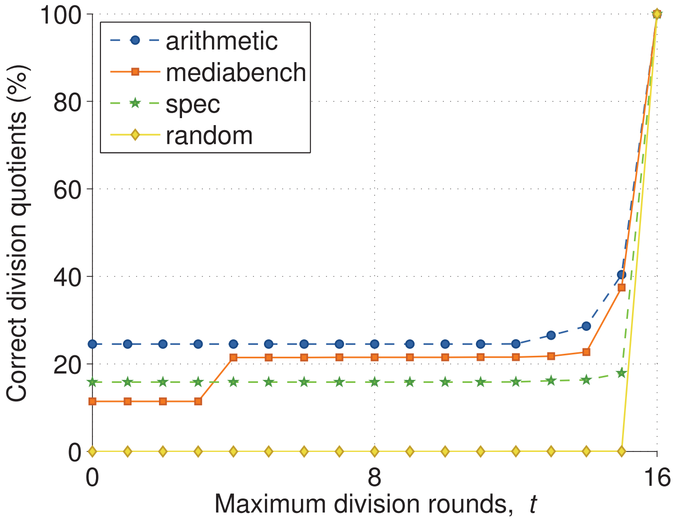


Figure 5.9: Proportion of correct quotients for a restoring divider operating in *t* division rounds.

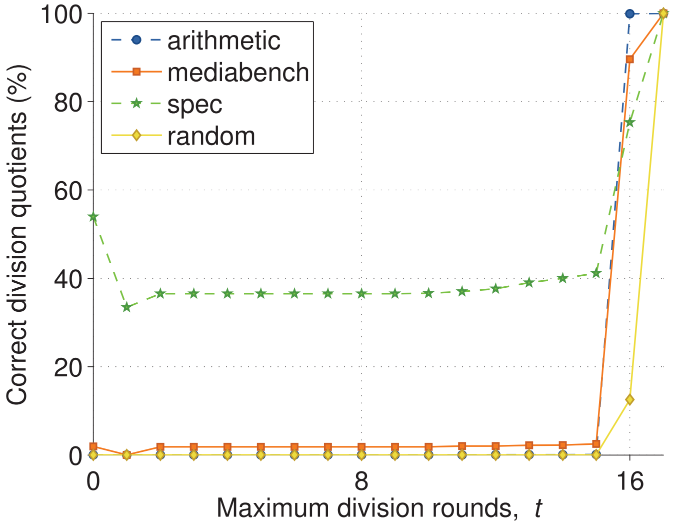


Figure 5.10: Proportion of correct quotients for a non-restoring divider operating in *t* division rounds.

The simulated P_D for *SPEC* benchmarks in Figure 5.10 is not monotonically increasing because some quotients are correctly represented with zero division rounds. In the case where the dividend z is greater than the divisor d, the quotient $q = z/d$ is zero. The quotient is initially zero, so the quotient is correct after zero division rounds. However, after one or more division rounds, the partial remainder that is eventually output as the quotient, might be temporarily incorrect at the end of each round, because it is not restored. In this case the partial remainder is not correctly represented as zero until the final restoring step after all division rounds. This pathological case is repeated many times in the *SPEC* benchmarks.

The correctness of the non-restoring approximating divider is less than the restoring divider for the same number of division cycles because of the cases where the partial remainder would equal the quotient. In restoring division the result would be corrected in the same cycle, but another clock is required for non-restoring division. However, the minimum clock period is shorter for a non-restoring divider.

5.2.6 Floating point rounding

Rounding modes are a part of the *IEEE-754* standard, and necessary for standard compliance. After any floating point operation, the possible non-representable significant bits in the significand are rounded (possibly changing the exponent) by one of four rounding modes. The four defined rounding direction modes are:

Nearest (even) The infinitely precise result is rounded to the nearest value, or if both values are equally near, to the value with the least significant bit zero. This is sometimes called *'Banker's rounding'*, and is the default rounding mode.

Positive infinity The representable bits are adjusted towards $+\infty$. This is sometimes called *'rounding up'*.

Negative infinity The representable bits are adjusted towards $-\infty$. This is sometimes called *'rounding down'*.

Zero The representable bits are adjusted towards zero. This is achieved by truncating the infinitely precise significand.

Round to nearest requires that some additional bits of the significand are calculated. The exact representation of a recurring binary division quotient is infinitely long. The first few bits longer than the floating point significand and the sign bit are used to determine the rounding direction. Rounding to ±∞ also requires the sign bit and additional significand bits to determine if the significand should be adjusted by one unit is least precision (ULP). Round to zero does not require an adjustment to the significand because it is implemented by a truncation.

The default rounding mode is round to nearest, and can be changed by software, although this is rarely done. If it can be assumed that round towards zero is sufficient, then hardware to implement significand rounding and adjustment of the exponent can be removed from the critical path.

A simulation was performed where the rounding mode was fixed to one of three rounding modes. The rounded result was counted as correct if it matched the result by using *round to nearest*. All operations were repeated with rounding modes to $+\infty$, to $-\infty$ and to zero. Table 5.2 shows the proportion of results for each rounding scheme that are identical to the 'round to nearest' result.

The correctness when using any rounding mode in the place of 'round to nearest' is high. The correctness of using '$+\infty$' and '$-\infty$' are both >50%, showing that rounding is not always required for benchmark operations. In most cases the correctness of single precision results was within 15 % of double precision results for each approximated rounding scheme.

From an implementation standpoint, using 'round to zero' (truncate) is attractive because the additional rounding bits can simply be discarded. The high correctness of the 'round to zero' rounding shows that most floating point results are positive. A high degree of variation in correctness when using round to zero exists, between $P_{min} = 48.5\,\%$ and $P_{max} = 98.0\,\%$, so rounding cannot be discarded for general purpose applications.

5.2.6.1 Temporally incomplete multiplication

An alternative method of approximation is *temporal incompleteness* as discussed in Section 3.1.2, where a logic circuit is overclocked. An approximate unsigned multiplier was simulated to determine the correctness profile of a circuit with many interconnected logic paths.

The 16×16 bit multiplier was created as a schematic in *Electric* [Static Free Software, 2005] using `full-adder` cells. Timing characteristic were derived from hand placed layouts of the `full-adder` cells in a 0.3μm *MOCMOS* process for *MOSIS* [MOSIS Integrated Circuit Fabrication Service, 2009].

The probability of correctness of the circuit was determined by the simulation of the multiplier using *IRSIM* [MultiGiG, Inc., 2006] with timing information extracted from the `full-adder` cells. The number of correct multiplications in a set of 100,000 uniform random 16 bit numbers was counted for circuits with a clock period of 1 to 40 ns in 1 ns increments. The correctness of the temporally

Table 5.2: Average number of floating point results that are identical to the result calculated using the 'round to nearest' rounding mode.

Benchmarks	Operation	to $+\infty$ (%)	to $-\infty$ (%)	to zero (%)	Average (%)
Arithmetic	`fpAdd`	60.951	65.647	65.243	**63.947**
	`fpSub`	86.814	91.105	90.699	**89.539**
	`fpMult`	60.731	48.683	48.541	**52.651**
	`fpDiv`	74.571	64.518	65.767	**68.286**
	`dblAdd`	61.565	64.726	65.012	**63.768**
	`dblSub`	70.194	84.103	83.191	**79.163**
	`dblMult`	59.678	59.276	59.226	**59.393**
	`dblDiv`	48.899	56.621	56.452	**53.991**
	Average	**65.425**	**66.835**	**66.766**	
Mediabench	`fpAdd`	77.650	76.301	76.347	**76.766**
	`fpSub`	94.340	94.371	94.306	**94.339**
	`fpMult`	67.442	66.746	66.631	**66.940**
	`fpDiv`	71.134	66.859	66.882	**68.292**
	`dblAdd`	87.404	88.711	88.759	**88.291**
	`dblSub`	98.058	98.127	98.005	**98.063**
	`dblMult`	79.095	79.016	79.173	**79.095**
	`dblDiv`	69.862	71.295	71.274	**70.810**
	Average	**80.623**	**80.178**	**80.172**	
SPEC	`fpAdd`	94.146	81.524	81.528	**85.733**
	`fpSub`	99.620	99.636	99.815	**99.690**
	`fpMult`	80.202	80.148	80.148	**80.166**
	`fpDiv`	95.476	95.345	95.305	**95.375**
	`dblAdd`	77.456	78.404	77.836	**77.899**
	`dblSub`	83.446	86.028	83.970	**84.482**
	`dblMult`	71.456	70.521	70.926	**70.967**
	`dblDiv`	72.709	68.140	68.133	**69.661**
	Average	**84.314**	**82.468**	**82.208**	

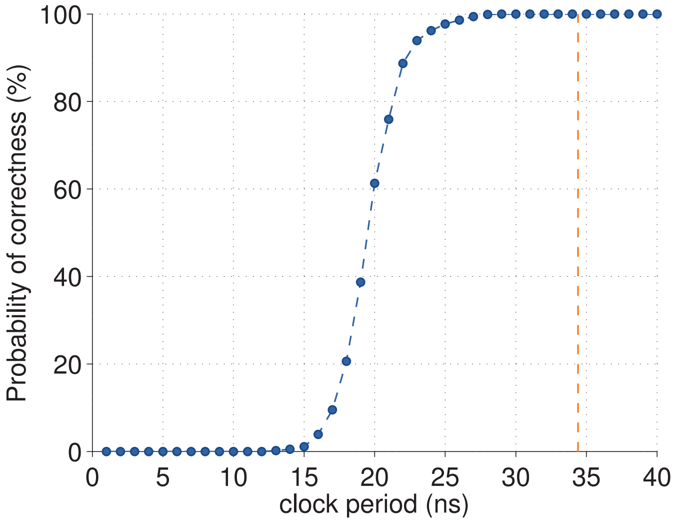


Figure 5.11: Correctness of a temporally incomplete 16×16 bit array multiplier. The critical path delay is shown as an orange dashed line.

incomplete multiplier is shown in Figure 5.11.

The overclocked multiplier maintains a 95 % correctness at 26 ns, 8.4 ns (24.4 %) faster than the critical path delay of the circuit, but the correctness of the circuit decreases rapidly below this point. Small changes in the sampling period could make large changes to the expected correctness at this point.

Temporally incomplete circuits were not pursued further in the course of the thesis. The simulation of the temporally incomplete multiplier lead to the conclusions that:

- Fast arithmetic such as the type used in high performance arithmetic circuits are likely to be pipelined. Partitioning long latency circuits into sections with timing elements introduces sources of metastability at each overclocked latch, and reduces the proportion of cycle time available for execution by setup time for synchronisation latches.
- The use of synchronisers to mitigate metastability requires additional clock signals, with a carefully controlled phase and period to control the overall circuit latency and probability of metastability.
- Calculating the probability of metastability occurring depends on many factors, including the process technology, layout, electrical properties of the interconnect, clock jitter, data inputs, and noise.

A sensible implementation of *ADVS* requires that the probability of correctness is well characterised for the approximate circuits used. Metastability introduces indeterminism, and an additional level of complexity in system design. Temporal incompleteness was abandoned to focus on deterministic logically incomplete circuits. The possibility of maintaining multiple clocks for several approximate arithmetic units in an *ADVS*-enabled system was undesirable. Techniques for local clock generation exist, but are outside the scope of this thesis.

5.3 Conclusion

The preliminary experiments in this chapter investigated simple techniques to reduce the latency of arithmetic operations, at the expense of correctness. Arithmetic operations output result words from a functional mapping between the input and output bits, where many dependencies exist between the input and output. It is difficult to modify high performance arithmetic units to sacrifice correctness for a proportionally higher gain to latency, especially for a target correctness of over 95 %, as required for *ADVS*.

PART II

APPROXIMATE ARITHMETIC

Chapter 6

Approximate Integer Multiplication

“What? You search? You would multiply yourself by ten, by a hundred? You seek followers? Seek zeros!”

Friedrich Nietzsche (1844–1900)

This chapter presents designs for unsigned approximate integer multipliers, which are extended to accommodate signed operands. The new approximate multipliers are based on modern high performance tree multipliers built from counter units. A family of logically incomplete counters are used as the fundamental unit to construct a family of multipliers, which have a reduced latency and probability of correctness due to the approximate counters. The corresponding latency and correctness trade-off is investigated, and compared to a Wallace multiplier constructed from full-adders cells.

Binary integer multiplication is essentially a multioperand addition problem. Integer multiplication operations are used to compute array indexes, or for fixed point arithmetic in processors that lack a floating point unit (FPU), for statistics and other numerical programs, and for data processing applications.

Multiplication is performed by generating partial products from the multiplier and multiplicand operands. The partial products are then summed to form the final product. A simple, high latency implementation can sum partial products in series. The worst-case delay for a multiplier is determined by the delay of the carries that arise in partial product accumulation. Deferring the accumulation of carries using carry-save methods gives rise to complex, low latency designs. Tree multipliers such as Wallace and Dadda multipliers [Wallace, 1964; Dadda, 1965] divide the multioperand addition into parallel additions, and accumulate the stored carries in the final phase. Further reductions in latency are achieved by operating at higher radices, so that each digit operation accounts for a greater proportion of the original operand width.

This chapter introduces a modification to binary tree multipliers to produce an approximate product result with a high probability of correctness. Similar to Liu and Lu's approach to addition, the data path is shortened so that the worst-case delay is reduced, but certain input combinations yield an incorrect result. Like the Liu and Lu adder, the method of approximation is parameterisable, so that a delay-correctness profile is established.

Few other examples of approximate multipliers exist in the literature. George et. al. describe a modified array multiplier constructed in PCMOS for a '*probabilistic computing*' application where the occasional error is tolerated. The error is not corrected and is therefore desired to be small; the PCMOS multiplier described is used in non-critical applications. An example is provided of a PCMOS array multiplier used to perform an *FFT* for radar image processing, where errors manifest as a subjective degradation in the image quality [George et al., 2006].

Another example of an error-tolerant application is the density parity check (*LDPC*) decoder described in Chapter 10 *Approximate Adders in LDPC*). Multioperand adders, being similar to multipliers, can be made approximating using the techniques described in this chapter.

In the following sections the design for a family of unsigned approximate multipliers is presented, and is later extended to signed approximate multipliers. Section 6.1 discusses counter cells used in arithmetic circuits, and introduces approximate counters; Section 6.2 shows how approximate counters can be used to construct unsigned approximate multipliers, and the probability of correctness is determined for operation on uniform random inputs; in Section 6.3 the unsigned approximate multipliers are simulated operating on benchmark data to derive the probability of correctness; an extension is presented in Section 6.4 to enable operation on signed operands, and the probability of correctness is determined for signed approximate multipliers.

Synthesis results for signed and unsigned approximate multipliers are presented in Section 6.5, and the synthesis delay is used to determine the delay-correctness trade-off. Finally, two potential signed and unsigned multipliers are highlighted as candidates for *ADVS* and discussed in Section 6.6. A brief summary is provided in Section 6.7.

6.1 Counters

Counters are common arithmetic units used in multioperand addition and multiplication. An $(n;m)$ counter takes n input bits and computes their m bit sum. Common counters are the $(3;2)$ counter (full-adder—*FA*) and the $(2;2)$ counter (half-adder—*HA*).

For this work I have introduced the idea of exact and approximate counters. A counter is called exact if the sum of its n input bits can be correctly represented in m bits; $m \geq \lceil \log_2(n) \rceil$. A counter is called approximate if $m < \lceil \log_2(n) \rceil$. Approximate counters are faster than exact counters, but their output is not correct for all input combinations. The output sum is truncated to m output bits; the upper sum bits are discarded because they are the least likely to be asserted. Table 6.1 shows the probability of $(n;m)$ counters producing a correct sum.

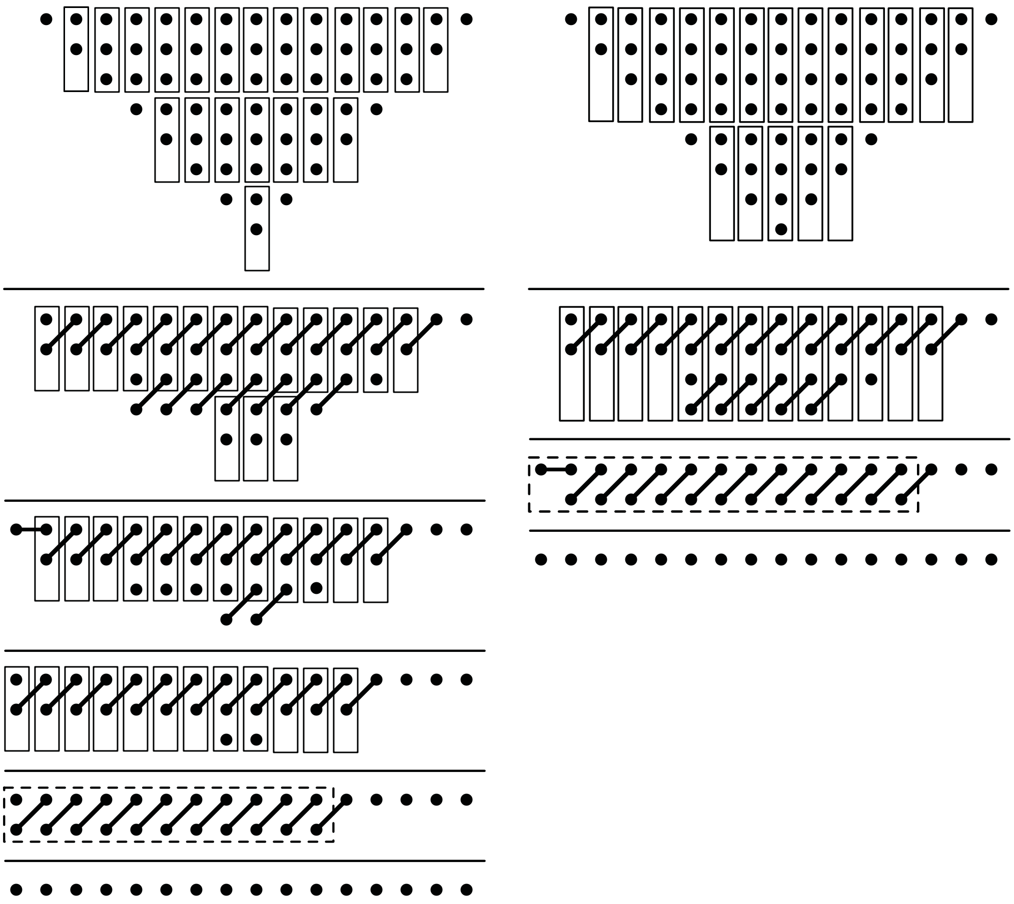
6.2 Multipliers

The remainder of this section discusses multiplier design, and modifications that were made to produce approximate outputs. The multipliers in this chapter use a Wallace tree structure, where counters sum the partial products in parallel in stored carry form [Parhami, 2000].

Table 6.1: Probability of correctness (%) for a counter with n input bits and m output bits. The input bits are assumed be asserted with equal probability.

				m		
		1	**2**	**3**	**4**	**5**
	2	75.00	100.00			
	3	50.00	100.00			
	4	31.25	93.75	100.00		
	5	18.75	81.25	100.00		
	6	10.94	65.62	100.00		
	7	6.25	50.00	100.00		
	8	3.52	36.33	99.61	100.00	
n	**9**	1.95	25.39	98.05	100.00	
	10	1.07	17.19	94.53	100.00	
	11	0.59	11.33	88.67	100.00	
	12	0.32	7.30	80.62	100.00	
	13	0.17	4.61	70.95	100.00	
	14	0.09	2.87	60.47	100.00	
	15	0.05	1.76	50.00	100.00	
	16	0.03	1.06	40.18	100.00*	100.00

*Result rounded to 100.00 %.



(a) Exact $(3;2)$ multiplier.

(b) Approximate $(4;2)$ multiplier.

Figure 6.1: Exact and approximate unsigned 8×8 bit tree multipliers.

6.2.1 Multiplier topology

Partial products were generated from the input operands. They are represented here in standard dot notation (Figure 6.1a) where each dot represents a logical *AND* of input bits from the multiplicand and multiplier [Parhami, 2000]. Each dot was allocated to a column, representing its numeric significance.

Partial products of the same significance (in the same column) were grouped and form inputs to $(n;m)$ counters. This was repeated until there were one or zero remaining partial products for each level of significance. Each counter's m output bits were distributed to the appropriate column and formed the partial products for the next level.

The construction of an 8×8 bit tree multiplier is shown in Figure 6.1a. 61 $(3;2)$ counters are used in

4 levels. This arrangement requires an 11 bit CPA, shown as a dashed line. This multiplier is exact: its probability of correctness is 100 %.

An approximate $(4;2)$ 8×8 bit multiplier is shown in Figure 6.1b. This tree multiplier is constructed with approximate $(4;2)$ counters instead of exact $(3;2)$ counters. In this case only 31 $(4;2)$ counters are needed in 2 levels, requiring a 13 bit CPA. The multiplier product will be incorrect when one or more approximate counters discard an asserted sum bit. Simulation of all input combinations to the $(4;2)$ 8×8 bit unsigned multiplier showed the probability of correctness is 93.55 %.

6.2.2 Allocation schemes

Fast multipliers employ a tree structure to quickly sum partial products in parallel. Two strategies can be used where either the length of the final CPA can be minimised in a *Wallace tree*, or by using as few *FA*s and *HA*s as possible, called a *Dadda tree*. Both strategies (*earliest-possible* and *just-in-time*) yield the same product in exact multipliers. The approximate multipliers described here trade probability of correctness for delay by allocating partial products on the critical path to approximate counters. In approximate multipliers, the allocation scheme affects both the delay and the correctness.

For the approximate multipliers described below, the longest column of partial products is aggressively allocated the minimum number of counters possible, and no other column can use more counters. The partial products in shorter columns can be allocated less aggressively; the lower 'input density' results in fewer opportunities for the approximate counters to overflow.

Three schemes are outlined below to allocate partial products to counters. Each scheme affects the overall delay and probability of correctness of an approximate multiplier. Designs used in the project used one scheme to uniformly allocate a single type of counter. Nonetheless, some counters may be substituted by design or synthesis optimisation for other elements like *HA*s or *FA*s without affecting the probability of correctness.

If longest column of partial products contains W bits, every bit in the column is allocated to one of $\lceil W/n \rceil$ counters, so that the minimum number of $(n;m)$ counters are used. The bits in the remaining columns are allocated to counters by one of the following allocation schemes:

greedy In this scheme the partial products are distributed evenly to as many counters as possible. Up to $\lceil W/n \rceil$ can be used in each column.

sparse This scheme uses fewer counters, but as many counters as required may be used per column (up to the maximum $\lceil W/n \rceil$ counters) to minimise the probability that any one counter overflows. Counters are first allocated up to $2^m - 1$ bits, and remaining bits are then distributed evenly in the column.

minimal This scheme uses the minimum number of counters possible. A column with Z bits only requires $\lceil Z/n \rceil$ counters, but the packing density increases the probability that any one counter will overflow.

Consider, for example, an approximate multiplier constructed from $(4; 2)$ counters. The longest column contains 12 bits, so no column may have more than $\lceil 12/n \rceil = \lceil 12/4 \rceil = 3$ counters to satisfy the delay constraint. Let us now consider the 3 schemes to allocate the bits in another column containing 5 bits (see Figure 6.2).

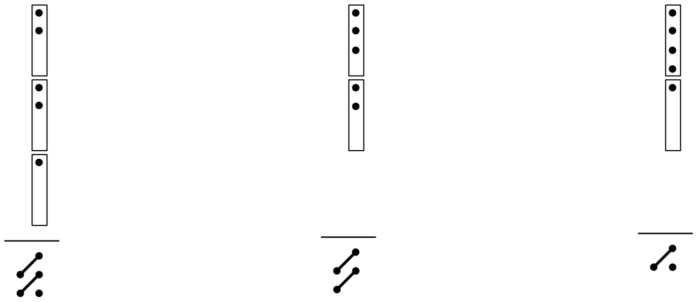
Using the *greedy* scheme, the 5 column bits are allocated as evenly as possible to the maximum 3 counters. Each counter is therefore allocated $\lfloor 5/3 \rfloor = 1$ or $\lceil 5/3 \rceil = 2$ bits (see Figure 6.2a). The *greedy* scheme produces 3 sum bits and 2 carry bits.

A $(4; 2)$ counter is guaranteed correct with up to $2^m - 1 = 2^2 - 1 = 3$ input bits. Using the *sparse* scheme, 3 bits are allocated to the first counter, and the remaining $5 - 3 = 2$ bits are allocated to another counter (see Figure 6.2b). For this case, the *sparse* scheme produces 2 sum bits and 2 carry bits.

Using *minimal* allocation, 4 bits are allocated to the first counter, and the remaining bit is allocated to another counter (see Figure 6.2c). In practise, a counter is not required to sum a single bit. The *minimal* scheme produces 2 sum bits and 1 carry bit.

For the example above the *greedy* scheme produced the greatest number of partial products. An approximate multiplier constructed with the *greedy* scheme has the highest probability of correctness because each counter has the fewest number of input bits, and is therefore the least likely to discard sum bits. The *minimal* scheme generated an approximate multiplier with the least correctness; fewer counters were used, but they were all the most likely to discard sum bits. Simulation of $(n; m)$ 32 bit multipliers showed the *sparse* scheme generally generated multipliers with correctness in between the *greedy* and *minimal* schemes.

The *greedy* and *sparse* schemes produced more bits in each column than the *minimal* scheme. When either was applied to all columns in a multiplier, the greater number of output bits from the additional counters caused more levels in the tree to be required. Further simulations showed that, in general, only the *minimal* scheme generated approximate multipliers with the same or fewer tree levels than an identically sized exact multiplier. For the remainder of this chapter, only the *minimal*



(a) greedy **(b)** sparse **(c)** minimal

Figure 6.2: Partial product allocation for approximate integer multiplication schemes using $(4;2)$ counters.

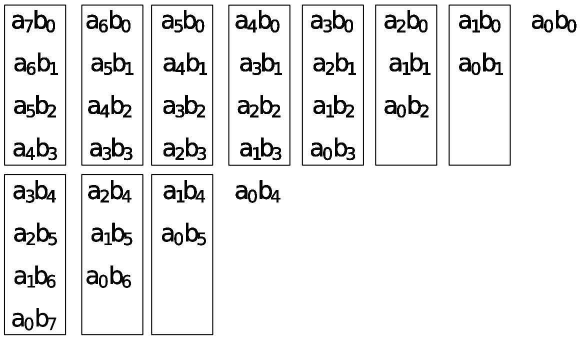


Figure 6.3: Correlated partial products allocated to $(4;2)$ counters.

allocation scheme is used, such as in Figure 6.1b.

6.2.3 Uniform random inputs

The probability of correctness of the approximate multipliers, P_M, is difficult to calculate. They generate partial products by *AND*-ing bits of the multiplicand with bits of the multiplier, therefore partial products in adjacent columns are not independent. Common modern multipliers typically have operands of size 32 or 64 bits. Automated generation of tree multipliers showed that 32×32 bit multipliers require in the order of 1,000 *FA*s. The large number of counters in each multiplier, and the complex relationship between the input and sum bits make calculating many cross-correlations necessary to exactly calculate P_M. A simpler, simulation-based approach was employed to determine P_M.

The correctness of the multipliers was influenced by the assignment of partial products to counters, and could be maximised by selectively grouping probabilistically weighted partial products in each column. The output bits of a counter have different probabilities of being asserted—the least significant bits have a higher probability of being asserted than the most significant bits. Hence, the allocation of bits in each column affects P_M.

Partial product bits all have different probabilities of being asserted. Selectively allocating individual partial product bits to specific counters to maximise P_M is difficult because increasing the probability of correctness of one counter in a column, P_C, invariably reduces the correctness of another. Simulation showed that the effect on overall P_M by redistributing partial product bits in every column was marginal. In the remainder of this chapter, partial products are assigned to counters in the order that they appeared in a column.

Table 6.2 shows the simulated correctness of multiplying 100,000 unsigned 32 bit uniform random number pairs with all feasible $(n;m)$ approximate multipliers, $n\in(2\ldots16)$. An $(n;m)$ multiplier is constructed entirely from $(n;m)$ counters, and a counter is considered feasible if $m<n$, and the sum of n input bits was representable in m bits or less, $m\leq\lfloor\log_2(n)\rfloor+1$.

Table 6.2 shows the correctness of $(n;m)$ multipliers with random inputs. Interestingly, the correctness of the entire multiplier $P_M(n;3)$ can be greater than the correctness of an individual counter $P_C(n;3)$ (see Table 6.1 on page 140). This is because random input bits to a $(n;m)$ counter are asserted with probability $^1/_2$. However, input operands to a multiplier are *AND*-ed to form the partial products; the assertion probability of most bits in the first level is $^1/_4$.

The overall delay of the multiplier is a complex function of the number of input bits, the size of the counters used and the allocation scheme used. Using counters with a high degree of truncation yields a fast but inaccurate multiplier. Using approximate counters with a low degree of truncation has a greater delay, but also greater probability of correctness.

6.3 Unsigned multiplier results

In this section the correctness of $(n;m)$ 32×32 bit multipliers is reported for multipliers operating on data from benchmark programs. The measured probability of correctness is compared with the observed probability of correctness for uniform *random* inputs.

Benchmark data was traced from the benchmarks listed in Sections 2.3.1—2.3.3. The total data set consisted of over 1 million unsigned multiplications and over 10 million signed integer multiplications.

Table 6.2: Measured probability of correctness P_M (%) for multiplication of uniform random numbers by unsigned $(n;m)$ 32×32 bit multipliers. An $(n;m)$ multiplier is constructed from counters with n input bits and m output bits.

				m		
		1	**2**	**3**	**4**	**5**
		1	2	3	4	5
	2	0.00				
	3	0.00	100.00			
	4	0.00	3.10	100.00		
	5	0.00	0.91	100.00		
	6	0.00	0.50	100.00		
	7	0.00	0.25	100.00		
n	**8**	0.00	0.06	99.87	100.00	
	9	0.00	0.06	98.08	100.00	
	10	0.00	0.07	94.30	100.00	
	11	0.00	0.07	94.12	100.00	
	12	0.00	0.08	91.49	100.00	
	13	0.00	0.08	87.72	100.00	
	14	0.00	0.09	82.74	100.00	
	15	0.00	0.09	77.21	100.00	
	16	0.00	0.08	71.66	100.00*	100.00

*Result rounded to 100.00 %.

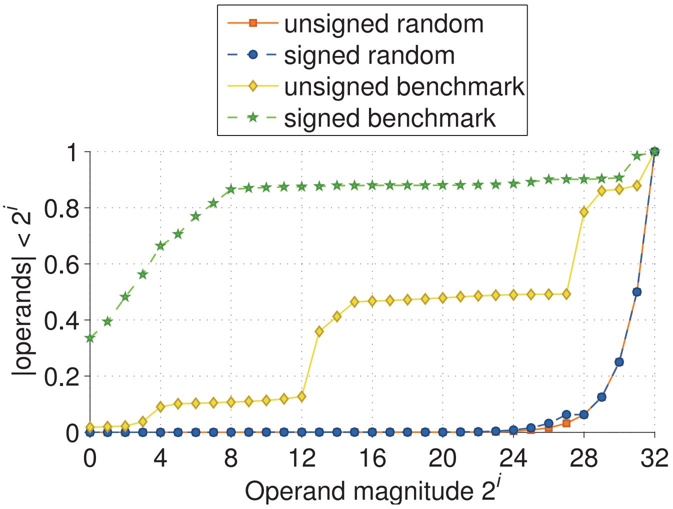


Figure 6.4: Cumulative distribution function of the absolute magnitude of signed and unsigned multiplication operands in benchmark programs.

6.3.1 Benchmark data inputs

Arithmetic data in real programs was observed to be significantly different from *random* data. Figure 6.4 shows a cumulative distribution plot (CDF) of the absolute magnitude of multiplication operands in benchmark programs. The benchmarks measured contained a significant proportion of signed multiplications where at least one operand had a value of one or zero. The average absolute magnitude of multiplication operands was much lower than uniform *random* numbers (see Figure 6.4). Operands with small magnitude had long strings of identical leading bits, affecting the probability that the multiplier produced an incorrect output. In the benchmark programs only 1.8 % of unsigned integer operands were zero, and 98.2 % were positive.

Unsigned $(n; m)$ 32×32 bit multipliers were simulated operating on benchmark data. Each product was recorded as correct or incorrect, and correctness of each multiplier was calculated. The results are shown in Table 6.3.

The approximate multipliers showed a higher correctness for benchmark data compared to *random* data due to the distribution of operands: benchmark operands were much smaller in magnitude and contained a higher proportion of zeroes. In general, Table 6.5 shows that the higher the degree of truncation in the counter's m sum bits, the lower overall P_M of the multiplier. Two exceptions were the $(10; 3)$ and $(11; 3)$ multipliers, when compared to the $(12; 3)$ multiplier. It is possible that in later levels in the tree, accumulated carries and sums in a particular column were susceptible to a few pathological cases, where a smaller counter was allocated more asserted bits, and hence more likely to truncate the sum output.

Table 6.3: Measured correctness (%) of unsigned $(n;m)$ 32×32 bit multipliers operating on benchmark data. An $(n;m)$ multiplier is constructed of counters with n input bits and m output bits.

		m				
		1	**2**	**3**	**4**	**5**
	2	33.23				
	3	32.84	100.00			
	4	32.93	93.87	100.00		
	5	33.04	92.96	100.00		
	6	32.76	85.14	100.00		
	7	32.76	86.48	100.00		
	8	32.76	86.43	100.00	100.00	
n	**9**	32.76	85.59	100.00	100.00	
	10	32.76	86.46	99.91	100.00	
	11	32.76	85.56	99.94	100.00	
	12	32.76	85.56	100.00	100.00	
	13	32.76	84.32	100.00	100.00	
	14	32.76	84.33	99.71	100.00	
	15	32.76	84.35	98.86	100.00	
	16	32.83	84.32	99.06	100.00*	100.00

*Result rounded to 100.00 %.

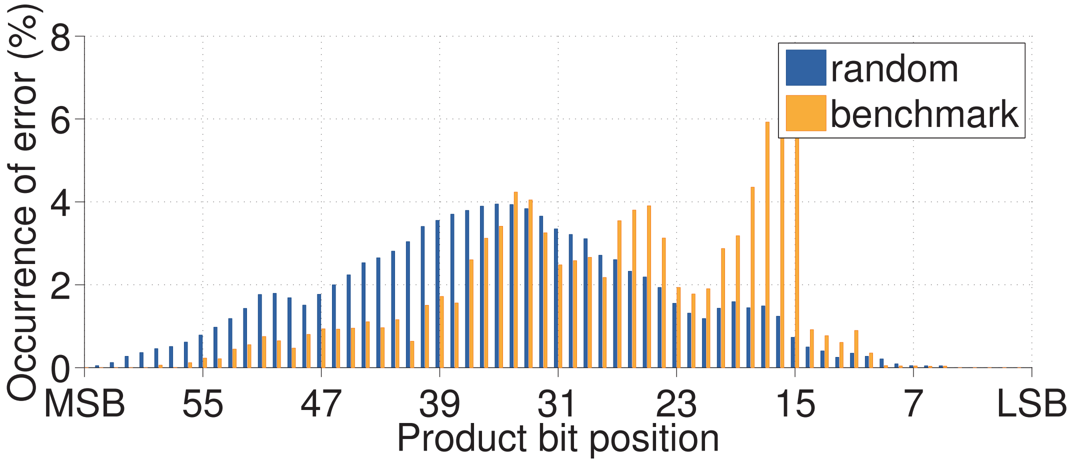


Figure 6.5: Bit error histograms for a (4; 2) unsigned 32×32 bit multiplier.

In Section 6.5, the multipliers were synthesised to determine their delay. Combining these results, a multiplier suitable for *ADVS* was selected based on the delay vs. correctness trade off.

6.3.2 Multiplier error

The approximate multipliers presented were designed to increase the speed of multiplication at the expense of the probability of correctness. In *ADVS*, the correctness is more important than the magnitude of error, but a relative error analysis is provided for completeness.

An error occurs in approximate unsigned multiplication when a counter discards a sum bit. As each partial product is accumulated in the final approximate product, erroneous products are less than or equal to exact products. The relative error e is therefore bounded $0 \leq e \leq 1$ for approximate unsigned multiplication.

6.3.2.1 Absolute error

The probability of absolute error was inferred by analysing the bit-error positions in the approximated product. Figure 6.5 shows a histogram of the occurrence of errors in each product bit of an unsigned $(4; 2)$ 32×32 bit multiplier when operating on *random* and benchmark data. As the average magnitude of the benchmark operands was much lower than the *random* data, the bit errors were more apparent in the lower bits. The correctness peak for benchmark data near bits 15–18 was due to the large number of repeated operands in the *255.vortex* and *RASTA* benchmarks.

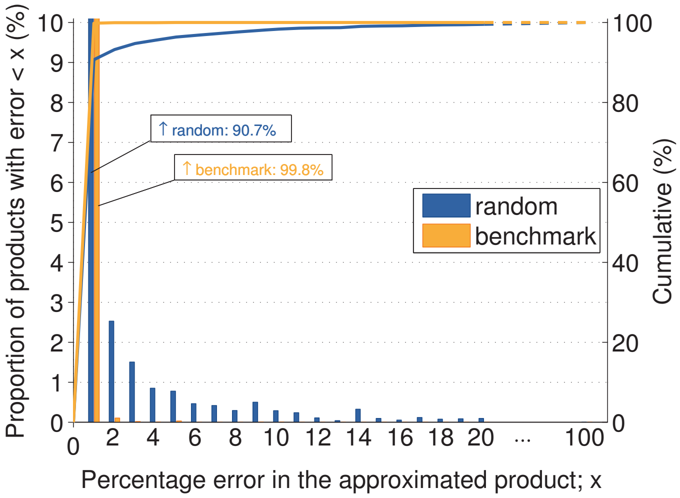


Figure 6.6: Relative error histogram of an unsigned (4; 2) 32×32 bit multiplier. Relative error is measured in 1 % bins up to 20 %. The left y-axis displays the scale for the histograms, and the right y-axis displays the cumulative line plot.

6.3.2.2 Relative error

The error relative to the exact product was determined by simulated multiplication of *random* and benchmark data for a (4; 2) 32×32 bit multiplier (see Figure 6.6). As shown, 90.7 % of all multiplications of *random* inputs yielded an error ≤1 % of the magnitude of the exact product; for benchmark data 99.8 % of all multiplications yield an error of ≤1 %.

6.4 Signed approximate multiplication

This section discusses a modification to approximate multipliers that enables them to handle signed twos complement operands, and an analysis of the impact on correctness and delay for *random* and benchmark inputs.

6.4.1 Baugh-Wooley signed multiplication

Unsigned multipliers can be modified to handle signed operands using the Baugh-Wooley method [Baugh and Wooley, 1973]. Figure 6.7 shows the changes applied to the partial products of a signed 8×8 bit

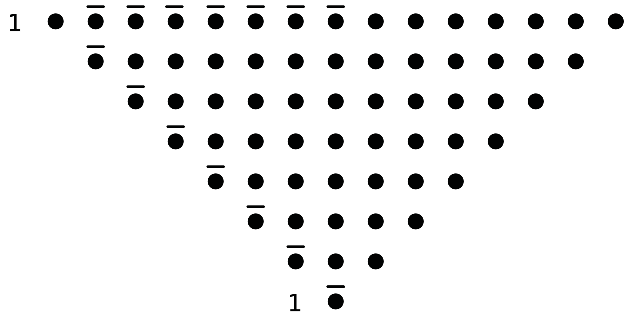


Figure 6.7: Partial products using dot notation for a signed 8×8 bit Baugh-Wooley multiplier.

multiplier. The negative weight of the twos complement sign bits is handled by inverting the sign bit and borrowing from the column of next greatest significance, rather than subtracting each bit in the full partial product. After the complement/borrowing is applied to all bits generated with a sign bit, most of the borrows cancel, but a few logical '1' bits (shown as 1) are required for balancing. Partial products are shown as dots (•) in the column representing their numeric significance. Complemented bits are represented with a bar above them ($\bar{\bullet}$).

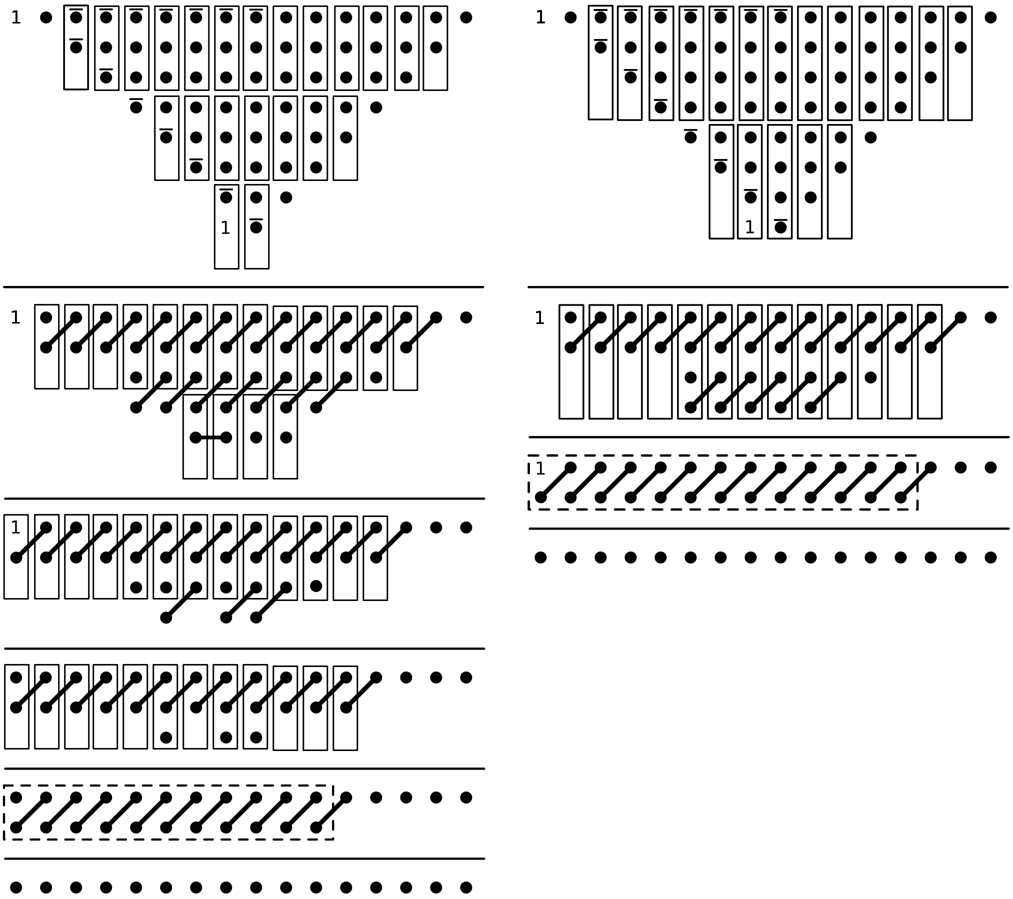
In Section 6.2.3 it was noted that the effective probability for each bit in the unsigned partial products (•) was $^1/_4$. The complemented bits in the Baugh-Wooley scheme effectively increase the probability of assertion of some bits ($\bar{\bullet}$) in the first tree level to $^3/_4$, reducing the overall P_M.

However, in the benchmark programs 33.6 % of signed operands were zero and only 7.6 % were negative, so it can be expected that few approximate errors should occur due to counter truncation or Baugh-Wooley complementation.

The example 8×8 bit tree multiplier in Figure 6.1 is repeated for a signed 8×8 bit $(4;2)$ approximate multiplier, and compared to a signed exact $(3;2)$ multiplier in Figure 6.8. In the case of the approximate multiplier in Figure 6.8b, the additional logical 1 bit bits do not affect the number of counters or levels required, but could for other $N{\times}N$ $(n;m)$ signed multipliers.

Simulation of all input combinations to the $(4;2)$ 8×8 bit signed multiplier in Figure 6.8b showed the correctness had decreased to 71.52 % from 93.55 % for the unsigned $(4;2)$ 8×8 bit multiplier in Figure 6.1b.

The probability of correctness P_M of feasible approximate signed 32×32 $(n;m)$ multipliers was determined by simulation of *random* (see Table 6.4) and benchmark data (see Table 6.5). In both cases the correctness of the signed multiplier was less than the unsigned multiplier, due to the assertion probability of the initial partial products. Again, the correctness of the benchmark data set



(a) Exact $(3;2)$ multiplier.

(b) Approximate $(4;2)$ multiplier.

Figure 6.8: Exact and approximate signed 8×8 bit tree multipliers.

Table 6.4: Measured correctness (%) of signed 32×32 bit multipliers constructed from counters with n input bits and m output bits, operating on uniform random inputs.

		m				
		1	**2**	**3**	**4**	**5**
	2	0.00				
	3	0.00	100.00			
	4	0.00	0.84	100.00		
	5	0.00	0.13	100.00		
	6	0.00	0.05	100.00		
	7	0.00	0.02	100.00		
	8	0.00	0.00	99.74	100.00	
n	**9**	0.00	0.00	96.81	100.00	
	10	0.00	0.00	89.98	100.00	
	11	0.00	0.00	90.13	100.00	
	12	0.00	0.00	86.55	100.00	
	13	0.00	0.00	81.17	100.00	
	14	0.00	0.00	74.56	100.00	
	15	0.00	0.00	67.64	100.00	
	16	0.00	0.00	57.98	100.00*	100.00

*Result rounded to 100.00 %.

was much higher than for *random* data, because of the absolute magnitude of the operands (see Figure 6.4).

6.4.2 Signed approximate multiplication error

In approximate signed multiplication, a counter that discards one of the more significant bits of the product could change the sign and magnitude of the product. In twos complement representation, identical leading bits reveal the sign of the operand, and the remaining lower order bits determine the magnitude. If a counter discards a bit that affects the accumulation of the leading sign bits, the sign or magnitude or both, could change. The error of signed approximate multiplier described here is therefore unbounded.

Table 6.5: Probability of correctness (%) for signed 32×32 bit multipliers constructed from counters with n input bits and m output bits, operating on benchmark data.

		m				
		1	**2**	**3**	**4**	**5**
n	**2**	0.00				
	3	0.00	100.00			
	4	0.00	88.80	100.00		
	5	0.00	87.97	100.00		
	6	0.00	87.38	100.00		
	7	0.00	87.05	100.00		
	8	0.00	86.31	98.53	100.00	
	9	0.00	85.97	98.38	100.00	
	10	0.00	85.28	98.04	100.00	
	11	0.00	85.43	98.14	100.00	
	12	0.00	85.36	97.98	100.00	
	13	0.00	85.35	97.97	100.00	
	14	0.00	85.28	97.84	100.00	
	15	0.00	84.95	97.08	100.00	
	16	0.00	85.38	96.58	98.54	100.00

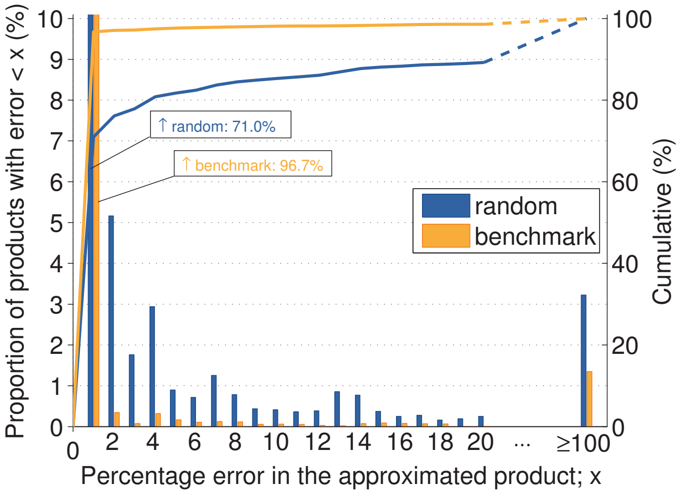


Figure 6.9: Relative error histogram for $(4;2)$ 32×32 bit multipliers. Relative error was measured in 1 % bins up to 20 %, and a final bin for error >100 % of the exact product. The left y-axis displays the scale for the histograms, and the right y-axis displays the cumulative line plot.

6.4.2.1 Signed approximate absolute error

The bit-error distributions of signed multipliers show the proportion of bit errors was roughly proportional to the number of counters in each column. The distribution is slightly skewed to the upper bits, where the input partial products were complemented. There were proportionally more bit errors in product bits 59–63. This was attributed to the sudden increase in operands with magnitude near 2^{30} and 2^{31} (see Figure 6.4).

6.4.2.2 Signed multiplier relative error

The error relative to the exact signed product was also determined by simulation for the case of a $(4;2)$ approximate signed multiplier (see Figure 6.9). The Baugh-Wooley scheme changed the initial assertion probabilities of the partial products, reducing the number of operands that yielded a relative error ≤1 %. An unsigned $(4;2)$ multiplier produced an error of ≤1 % for 90.7 % of *random* multiplications, but the signed version attained only 70.1 %. The decrease was less for benchmark data, from 99.8 % to 96.7 %, because the average magnitude of signed benchmark operands was less than the unsigned benchmark operands, and few signed operands were negative.

Although the error was unbounded, only 3.2 % of *random* data multiplications and 1.3 % of benchmark data multiplications yielded errors greater than 100 % product magnitude, because there were few errors that occurred in the leading sign bits of the exact product. When the exact product was positive and the leading sign bits of the product were zero, the partial products in that column

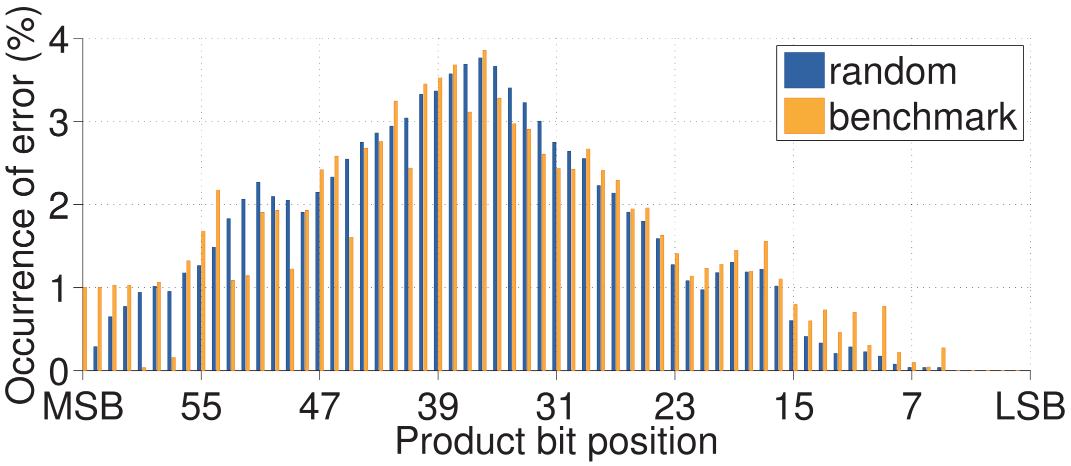


Figure 6.10: Bit error histograms for a (4; 2) signed 32×32 bit multiplier.

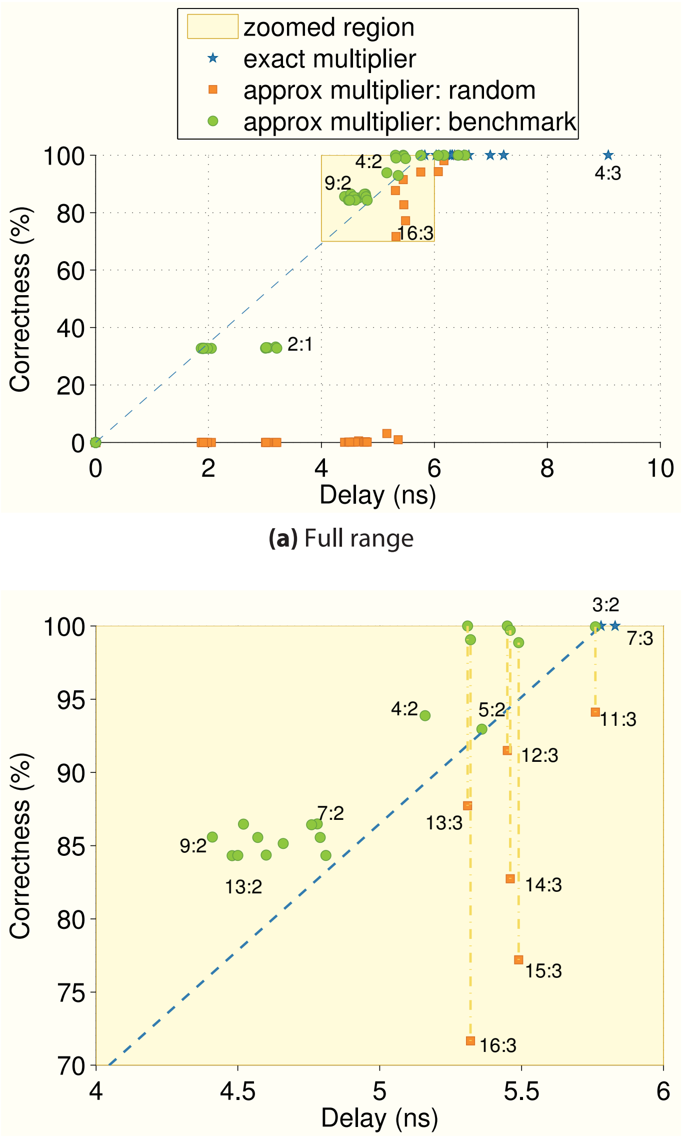
were zero, and therefore all counters were correct. When the exact product was negative and the leading sign bits of the product were one, accumulated partial products typically generated carries that rippled through the leading ones in the most significant bits until they overflowed and were discarded. Hence, multiple counter failures were required to cause one of the leading sign bits of the approximate product to be incorrect.

6.5 Synthesis

Table 6.7 shows the delay, area and power of signed $(n; m)$ 32×32 bit multipliers synthesised with *Synopsys Design Compiler*, using the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library. Delay was measured as the time a signal took to propagate along the critical path from the assertion of the operands until all product bits were asserted. Area was the total combinatorial and interconnect area, and power was measured as the sum of the synthesised dynamic and leakage power of the multipliers.

Figure 6.12 shows the delay versus correctness of signed $(n; m)$ 32×32 bit multipliers. Some interesting labelled cases are discussed below. The dashed line indicates linear trade-off with respect to the exact $(3; 2)$ (full-adder) multiplier. Figure 6.12a shows all $(n; m)$ multipliers, and the shaded region of interest is shown in Figure 6.12b. Multipliers in this region have a high probability of correctness.

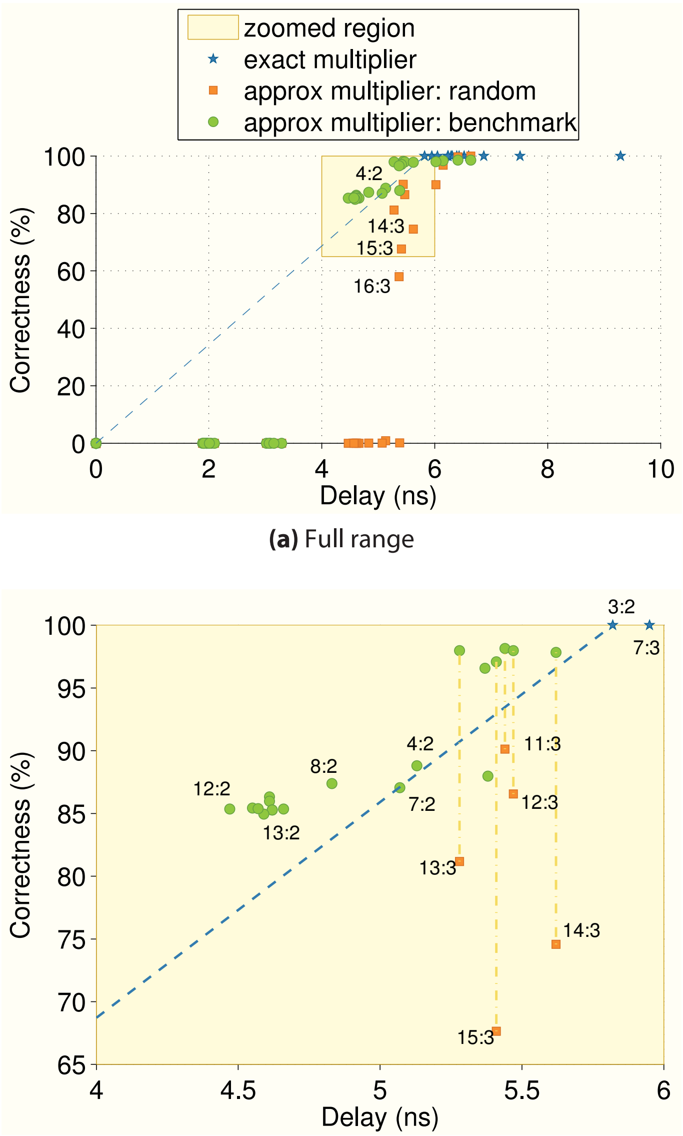
Multiplier area was determined by the number of logic cells used in each design, corresponding to the degree of truncation of the counters used. As m increased, each counter outputted more bits,



(a) Full range

(b) Zoomed region

Figure 6.11: Full scatter plot and zoomed view of unsigned $(n; m)$ 32×32 bit multipliers, showing multiplier delay vs. correctness.



(a) Full range

(b) Zoomed region

Figure 6.12: Full scatter plot and zoomed view of signed $(n; m)$ 32×32 bit multipliers, showing multiplier delay vs. correctness.

Table 6.6: Synthesis results for unsigned $(n; m)$ 32×32 bit multipliers.

					m		
			1	2	3	4	5
n	2	(ns)	3.190				
		(μm^2)	0.579				
		(W)	0.211				
	3	(ns)	3.010	5.780			
		(μm^2)	0.534	1.586			
		(W)	0.188	0.515			
	4	(ns)	3.050	5.160	9.080		
		(μm^2)	0.585	1.259	2.320		
		(W)	0.214	0.399	0.558		
	5	(ns)	3.020	5.360	7.230		
		(μm^2)	0.537	1.182	1.768		
		(W)	0.191	0.399	0.537		
	6	(ns)	1.940	4.660	6.280		
		(μm^2)	0.479	1.221	1.699		
		(W)	0.165	0.388	0.466		
	7	(ns)	1.900	4.780	5.830		
		(μm^2)	0.483	1.147	1.515		
		(W)	0.165	0.371	0.490		
	8	(ns)	1.910	4.760	6.540	6.990	
		(μm^2)	0.480	1.112	1.597	1.814	
		(W)	0.165	0.358	0.469	0.503	
	9	(ns)	1.880	4.410	6.170	6.320	
		(μm^2)	0.482	1.117	1.570	1.760	
		(W)	0.165	0.352	0.441	0.475	
	10	(ns)	1.900	4.520	6.070	6.610	
		(μm^2)	0.477	1.123	1.551	1.704	
		(W)	0.164	0.360	0.450	0.510	
	11	(ns)	1.910	4.790	5.760	6.520	
		(μm^2)	0.480	1.080	1.517	1.720	
		(W)	0.164	0.334	0.450	0.490	
	12	(ns)	2.050	4.570	5.450	6.080	
		(μm^2)	0.472	1.136	1.539	1.767	
		(W)	0.163	0.349	0.458	0.486	
	13	(ns)	1.870	4.480	5.310	6.280	
		(μm^2)	0.484	1.115	1.515	1.688	
		(W)	0.165	0.352	0.449	0.492	
	14	(ns)	1.980	4.810	5.460	6.360	
		(μm^2)	0.484	1.071	1.504	1.731	
		(W)	0.166	0.340	0.446	0.488	
	15	(ns)	1.910	4.600	5.490	6.310	
		(μm^2)	0.475	1.113	1.481	1.743	
		(W)	0.163	0.349	0.433	0.478	
	16	(ns)	3.210	4.500	5.320	6.420	6.030
		(μm^2)	0.584	1.127	1.500	1.711	1.169
		(W)	0.211	0.351	0.438	0.484	0.360

Table 6.7: Synthesis results for signed $(n; m)$ 32×32 bit multipliers.

			m				
			1	2	3	4	5
n	2	(ns)	3.090				
		(μm^2)	0.586				
		(W)	0.214				
	3	(ns)	3.020	5.820			
		(μm^2)	0.527	1.553			
		(W)	0.185	0.516			
	4	(ns)	3.290	5.130	9.290		
		(μm^2)	0.580	1.275	2.279		
		(W)	0.208	0.408	0.547		
	5	(ns)	3.070	5.380	7.510		
		(μm^2)	0.542	1.210	1.785		
		(W)	0.191	0.406	0.545		
	6	(ns)	2.040	4.830	6.230		
		(μm^2)	0.477	1.215	1.733		
		(W)	0.164	0.379	0.485		
	7	(ns)	2.100	5.070	5.950		
		(μm^2)	0.479	1.143	1.527		
		(W)	0.164	0.375	0.494		
	8	(ns)	1.920	4.610	6.410	6.870	
		(μm^2)	0.475	1.123	1.609	1.835	
		(W)	0.163	0.358	0.481	0.519	
	9	(ns)	1.890	4.610	6.150	6.290	
		(μm^2)	0.490	1.134	1.583	1.798	
		(W)	0.166	0.353	0.447	0.496	
	10	(ns)	1.960	4.620	6.020	6.600	
		(μm^2)	0.489	1.102	1.528	1.711	
		(W)	0.167	0.362	0.441	0.515	
	11	(ns)	1.910	4.550	5.440	6.390	
		(μm^2)	0.487	1.115	1.530	1.783	
		(W)	0.165	0.348	0.454	0.516	
	12	(ns)	1.970	4.470	5.470	6.310	
		(μm^2)	0.485	1.115	1.552	1.745	
		(W)	0.166	0.350	0.468	0.487	
	13	(ns)	1.960	4.660	5.280	6.240	
		(μm^2)	0.486	1.121	1.508	1.700	
		(W)	0.165	0.354	0.449	0.504	
	14	(ns)	1.910	4.620	5.620	6.520	
		(μm^2)	0.483	1.119	1.488	1.732	
		(W)	0.165	0.348	0.444	0.490	
	15	(ns)	2.010	4.590	5.410	6.430	
		(μm^2)	0.477	1.118	1.486	1.712	
		(W)	0.163	0.352	0.435	0.474	
	16	(ns)	3.150	4.570	5.370	6.640	6.050
		(μm^2)	0.585	1.098	1.483	1.697	1.183
		(W)	0.212	0.342	0.431	0.481	0.369

so more counters were required, and the area increased rapidly. As n increased, the logic required for each counter increased, however fewer counters were used so the area decreased slowly.

The trends in power were similar to area. The behaviour of each multiplier was similar in terms of logic switching activity, so power consumption was closely related to the logic and interconnect area.

Many factors affected the delay of the multiplier circuits. Delay was comprised of delay of the counters, the number of levels in the tree, interconnect delay and delay through the CPA. As m increased, the additional counters that were required dominated, and delay increased. As n increased, the delay of each counter increased, but fewer counters were required in the tree. Therefore, the number of levels in the tree decreased, which reduced delay, but also increased the width of the CPA required. Small changes to the width of the CPA had a marginal effect on CPA delay when the CPA was a fast parallel prefix adder.

P_M was calculated from operation on *random* and benchmark data. As discussed in Section 6.3.1, the multipliers tended to have a higher correctness when they operated on benchmark data due to the distribution of input operands (see Figure 6.12b). The increased P_M is shown by a dotted vertical line in Figure 6.11b and Figure 6.12b, showing the change in correctness of the same multiplier operating on *random* (×) and benchmark (•) data.

Many of the multipliers shown in Figure 6.12b exhibited a better than linear trade-off of P_M for delay, compared to the signed $(3;2)$ multiplier. Multipliers such as the unsigned $(4;2)$ and $(5;2)$, and the signed $(4;2)$ and $(7;2)$ multipliers yielded a small reduction in delay and a small penalty to P_M. Another cluster, including the unsigned $(7;2)$, $(9;2)$, $(13;2)$, and signed $(8;2)$, $(12;2)$ and $(13;2)$ multipliers are slightly faster again, with further degraded probability of correctness.

P_M was highly data dependent for multiplier constructed of counters with a high degree of truncation, such as the $(13;2)$.

6.6 Approximate integer multipliers for ADVS

Two interesting signed and unsigned approximating multipliers potentially suitable for *ADVS* are discussed below, and compared to an exact Wallace $(3;2)$ signed tree multiplier. Multipliers based on counters that truncate two or more bits were excluded, due to their measured correctness being highly data dependent.

The selected unsigned 32×32 multipliers were the $(4;2)$ and $(7;2)$. The $(4;2)$ unsigned multiplier was 10.7 % faster than the equivalent exact $(3;2)$, while P_M decreased 6.13 % for benchmarked multiplications. The approximate $(7;2)$ was 17.3 % faster and incorrect in 13.52 % of benchmark multiplications. Both multipliers yielded an area saving in excess of 35 %, and power savings (combined dynamic and leakage) in excess of 20 %, compared to the $(3;2)$ exact multiplier.

The selected approximate signed multipliers were the signed $(4;2)$ and $(8;2)$. The signed $(4;2)$ multiplier has small delay gain and small penalty to correctness, and the $(8;2)$ has a larger delay gain and larger penalty. The signed $(4;2)$ was 11.8 % faster than the exact $(3;2)$, with 11.2 % probability of error for benchmarked multiplications. The signed $(8;2)$ multiplier was 20.8 % faster, and incorrect in only 13.7 % of benchmark multiplications. Both multipliers yielded an area saving in excess of 17 %, and again power savings (combined dynamic and leakage) are in excess of 20 %.

An in-depth study of the errors generated by the approximate $(4;2)$ unsigned and signed 32×32 bit multipliers was shown in Sections 6.3.2 and 6.4.2 respectively.

6.7 Conclusion

This chapter introduced the concept of approximate counters, which yielded the truncated sum of n input bits in m output bits. Designs for a family of signed and unsigned approximate multipliers using approximate counters were given, with signed multipliers based on the Baugh-Wooley method. An analysis of operands from benchmark programs was performed, showing that the expected operand magnitude was much lower than *random* numbers. Furthermore, multiplication with such operands increased the expected probability of correctness for approximate multipliers. The distribution of product bit errors and distribution of the magnitude of relative errors was shown for a signed and unsigned $(4;2)$ 32×32 bit multiplier.

The probability of correctness and delay trade-off for approximate multipliers depend on the operand distribution of the target application, the degree of truncation of the counters used, the allocation of partial products, and the technology process in which the multiplier was fabricated. This chapter has demonstrated the feasibility of constructing approximate multipliers that can operate faster than exact multipliers with small probability of error, and small relative error when they occur. Selected candidate multipliers are simulated in an *ADVS*-enabled system in Chapter 11 *ADVS Simulation*.

Chapter 7

Approximate Integer Division

"You've got many refinements. I don't think you need to worry about your failure at long division. I mean, after all, you got through short division, and short division is all that a lady ought to be called on to cope with."

Tennessee Williams (1911–1983)

This chapter presents designs for approximate integer dividers. The well known iterative division algorithm is modified so that an approximation to the exact quotient converges in each round. A variable number of quotient digits are calculated per iteration, at the expense of correctness (due to loss of precision), and a variable number of division rounds. It is demonstrated that the new division algorithm is sufficient to calculate the integer (but not necessarily fractional) component of the quotient for operands typically observed in benchmark programs, with a lower latency than a similar variable latency radix-4 SRT divider.

Division is a time consuming and infrequent arithmetic operation. This chapter presents an algorithm for approximate division of unsigned integers, and three designs of approximating dividers based on this algorithm. The dividers produce an approximate quotient, but not a remainder, and implemented with simple, fast operations. The unsigned approximate dividers are then modified to operate on signed inputs.

This chapter is organised as follows: Section 7.1 defines an algorithm for the exact division of two unsigned binary operands, and modifications made to the exact algorithm to produce an approximate quotient; Section 7.2 presents three implementations of the unsigned approximate division algorithm; in Section 7.3 the probability of correctness of each implementation is measured using random inputs and benchmarks; the unsigned approximate hardware developed is modified to handle signed inputs in Section 7.4; and results of synthesis of the approximating dividers are presented in Section 7.5. A brief summary is provided in Section 7.7.

7.1 Division algorithms

In this section an algorithm to compute the exact quotient $q = z/d$ is presented. Section 7.1.2 contains simplifications that are progressively applied to make the algorithm more efficient in hardware, at the cost of occasionally producing an erroneous result. The error in the approximate quotient is bounded, however the important result in this chapter is the probability of correctness, and not the magnitude of the error.

The latency, correctness and hardware cost for the approximating dividers are compared against a baseline divider in Section 7.5.

To produce an approximating algorithm, a natural approach is to use an algorithm that converges towards an exact result, but terminates before an exact result is guaranteed. This approach is used initially, and later further approximations are introduced by simplifying complex steps in the algorithm.

7.1.1 Exact division algorithm

The exact division algorithm divides the quotient q by the divisor d by first dividing by a nearby binary power $\tilde{d}$, and then iteratively correcting the result. Division by a power of two can be implemented as a fast shift operation. Alternative approximate divisors are possible; d_H is the power of two greater than d, and d_L is the power of two less than or equal to d. Take

$$\tilde{d} = \begin{cases} d_H & \text{if } (d_H - d) < (d - d_L) \\ d_L & \text{otherwise} \end{cases} \tag{7.1}$$

and consider an algorithm in which

$$\frac{z}{d} = q_i + \frac{r_i}{d}$$

is true at each round, and in which the q_i terms converge towards the exact quotient $q = z/d$.

To find q_1 divide z by $\tilde{d}$. As $\tilde{d}$ is a binary power, this division can be performed with a binary shift to the right. Thus

$$\begin{aligned} \frac{z}{d} &= q_1 + \frac{r_1}{d} \\ &= \frac{z}{\tilde{d}} + \frac{r_1}{d} \end{aligned} \tag{7.2}$$

for some value of r_i. The error e_1 in the quotient approximation q_1 is

$$\begin{aligned} e_1 = \frac{r_1}{d} &= \frac{z}{d} - \frac{z}{\tilde{d}} \\ &= \frac{z}{d} - \frac{z}{\tilde{d}} \\ &= -\frac{z(d - \tilde{d})}{d.\tilde{d}} \\ &= -\frac{z(d - \tilde{d})\,{}^{1}\!/_{\tilde{d}}}{d} \\ &= -\frac{z(d - \tilde{d})\,{}^{1}\!/_{\tilde{d}}}{d} \end{aligned} \tag{7.3}$$

To generate q_2, a refined approximation of the quotient, undertake the division $e_1 = r_1/d$ by dividing by $\tilde{d}$. From (7.2):

$$\begin{aligned}
\frac{z}{d} &= \frac{z}{\tilde{d}} + \left(\frac{r_1}{d}\right) \\
&= \frac{z}{\tilde{d}} + \left(\frac{r_1}{\tilde{d}} + \frac{r_2}{d}\right) \\
&= q_2 + \frac{r_2}{d} && (7.4)
\end{aligned}$$

Similar to the first round find

$$\begin{aligned}
e_2 = \frac{r_2}{d} &= \frac{z}{d} - \frac{r_1}{\tilde{d}} \\
&= -\frac{r_2(d - \tilde{d})}{d \times \tilde{d}}
\end{aligned}$$

Thus the following iteration is observed:

$$\begin{aligned}
q_0 &= 0 \\
r_0 &= z \\
e_0 &= z/d \\
q_{i+1} &= q_i + \frac{r_i}{\tilde{d}} && (7.5) \\
r_{i+1} &= \frac{-r_i(d - \tilde{d})}{\tilde{d}} && (7.6) \\
e_i &= \frac{r_i}{d}
\end{aligned}$$

An error bound for e_i follows:

$$\begin{aligned}
|e_i| &= \left|\frac{r_i}{d}\right| \\
&= \left|\frac{1}{d} \times \frac{r_{i-1}(d - \tilde{d})}{\tilde{d}} \times \ldots \times r_0\right| \\
&= \left|\frac{z}{d} \times \left(\frac{d - \tilde{d}}{\tilde{d}}\right)^i\right| \\
&\leq q \times \left(\frac{1}{2}\right)^i && (7.7)
\end{aligned}$$

Note that $|r_{i+1}| < |r_i|$, provided $|d - \tilde{d}| < |d|$. This is guaranteed as $\tilde{d}$ is the nearest binary power to d. Hence the remainders, r_i, tend to zero and the quotients, q_i, converge on the exact quotient z/d. Observe that many rounds could be required to obtain the exact quotient.

7.1.2 Approximating division algorithm

The following subsections show how the exact division algorithm from the previous section can be simplified to make the hardware implementation more efficient by introducing a bounded error to the quotient. The sources of error are: limiting the number of fractional bits to store q_i; limiting the number of rounds to perform the calculation of q_i; and approximating the multiplication of $r_i \times (d - \tilde{d})$. These error sources are discussed in detail below.

7.1.2.1 Fractional bits

According to (7.2), the first round quotient, q_1, is found by dividing by the approximate divisor $\tilde{d}$. This can be implemented by shifting z by $\tilde{D}$ bits to the right. Thus, q_1 contains both integer and fractional bits. In subsequent rounds the quotient q_i is refined by adding $r_i/\tilde{d}$, a right shift of r_i by $\tilde{D}$ bits. For practical purposes, a finite number of fractional bits is stored. Limiting the number of fractional bits in q_i to f introduces a quantisation error. As examined in Section 7.3.1.1 and 7.3.1.2, the choice of f affects the correctness of the divider, as well as its physical characteristics, including delay.

7.1.2.2 Fixed number of rounds

The calculation of the exact quotient by successive right shifts can require many rounds. In each round the quotient is adjusted by a diminishing factor, but only the integer part of the quotient is retained. An upper limit can be set on the number of rounds, t. This fixes the circuit delay, but can cause an error by terminating the algorithm before the integer part of the approximate quotient has converged.

7.1.2.3 Approximate multiplication

In (7.6), each round required a multiplication $r_i \times (d - \tilde{d})$. A full n-by-n multiplication in each round was expensive in circuit delay and area. Furthermore the full precision of the multiplication was not be needed, as the lower bits of the product were right shifted by $\tilde{D}$, and only affected the fractional bits of q_i and r_i.

Multiplication is typically performed using a sum of shifted partial products [Ercegovac and Lang, 2003]. Each partial product is the product of the multiplicand and a digit of the multiplier. A simple, fast method of approximation is to sum only the most significant few partial products.

In the following sections the multiplicand is denoted $m = d - \tilde{d}$; note that this is constant for a particular division. The most significant non-zero partial product was used to approximate the binary product $r_i \times m$, and denoted p_1. The next most significant partial product was also used, denoted p_2. Observe that p_2 will be equal to a left shift of the multiplicand, or zero if the next most significant bit of the multiplier is zero. In the approximating division algorithm the term $(r_i \times m)$ is approximated as $p_1 + p_2$. When this approximation is used, the approximating division algorithm can no longer be guaranteed to converge on the exact quotient.

7.2 Hardware design

In this section designs for implementations of the approximating algorithm described above are presented. The divider is considered in three distinct stages: *initialisation*, *division* and finally *accumulation*.

7.2.1 Initialisation stage

The initialisation stage computed constant values used in the subsequent division stage. This included the right shift amount $\tilde{D}$ and the multiplicand $m = d - \tilde{d}$ used throughout the division.

7.2.1.1 Initial shift amount

The initial approximation q_o was the right shift of z by $\tilde{D}$. The terms d_U and d_L in (7.1) depended on the most significant non-zero bit in d. This was found using a leading zeroes detector (LZD) circuit. The output, $\mathit{LZD}(d)$ was a hot-one encoded bit vector showing the most significant non-zero bit. d_L was found by binary encoding the output of the LZD. The output of the LZD was also used to indicate division by zero.

Selection of $\tilde{d}$ as either d_U or d_L in (7.1) depended on the bit to the right of the most significant bit in d. To inspect this, the output of the LZD was shifted right one bit and AND-ed with d. When the result was zero $\tilde{d} = d_L$ was selected; otherwise $\tilde{d} = d_U = d_L + 1$.

$\tilde{D}$ was thus the initial shift amount to implement the division by $\tilde{d}$ in the initial calculation of q_0. This is called the *initialShamnt*.

7.2.1.2 Constant multiplicand

The multiplicand $m = d - \tilde{d}$ required for (7.6) was simple to calculate when $\tilde{d} = d_L$. In this case it was sufficient to mask out the most significant bit of d using the output of the *LZD*:

$$m = d - \tilde{d} = d \text{ AND NOT}(\text{LZD}\,(d))$$

When $\tilde{d} = d_U$, m must be determined by the subtraction:

$$m = d - \tilde{d} = -\left((\text{LZD}(d) \ll 1) - d\right)$$

The selection of d_H or d_L thus changed the sign of r_i, and had to be accounted for in the divider implementation.

7.2.1.3 Terms for approximate multiplication

As discussed in Section 7.1.2.3 *Approximate multiplication*, the multiplication to find $r_i \times m$ (see (7.6)) was approximated using the most significant non-zero partial product in the binary multiplication, p_1, and the next most significant partial product, p_2.

The value of p_1 was found by shifting r_i left. The shift amount was determined by the position of the most significant non-zero bit of m, and is constant for a division. The value of p_2 was either 0 or $p_1 \gg 1$, depending on the next most significant bit in m. The most-significant non-zero bit and its less significant neighbour were determined with a *LZD* and mask of m, as was performed on d in Section 7.2.1.3.

7.2.1.4 Constant round shift amount

Throughout the exact division algorithm of Section 7.1.1 *Exact division algorithm*, the divisor $\tilde{d} = \tilde{d}$ remained constant as given in (7.5) and 7.6. Likewise, p_1 and p_2 were calculated using the constant multiplicand (see Section 7.2.1.3). Hence, r_{i+1} in (7.6) was calculated by a constant left shift for the multiplication by m, and a constant right shift for the division by $\tilde{d}$. Taking the difference in the shift amounts yielded a constant right shift, as $|r_{i+1}| < |r_i|$. The constant right shift was called the *roundShamnt*.

7.2.2 Division stage

The outputs of the division stage were latched and iterated upon in the next division round. The division stage operated for a fixed number of clock cycles, as indicated in Section 7.1.2.2. In each clock cycle q_i and r_i were calculated using (7.5) and (7.6). The division operation shown in (7.6) was implemented a right shift by *roundShamnt*.

$$\begin{aligned} r_{i+1} &\leftarrow r_i \gg \mathit{roundShamnt} \\ q_{i+1} &\leftarrow q_i + r_{i+1} \end{aligned}$$

To increase the speed of the addition of the partial remainder, carry-save adders (CSA) were used. The intermediate result q_{i+1} was produced in carry save form, $q_{s_{i+1}}$ and $q_{c_{i+1}}$, such that $q_{i+1} = q_{s_{i+1}} + q_{c_{i+1}}$. Similarly r_{i+1} was stored in this redundant format as r_{c_i} and r_{s_i}.

7.2.3 Accumulation stage

The final accumulation stage summed the carry and sum terms of q_i in a carry-propagate adder to produce a result in non-redundant form.

7.2.4 Implementation notes

The exact division algorithm was carefully factored to fix many of the terms in each round as a constant, and to reduce the latency of the repeated division stage. Constant terms were factored out, but the synthesised initialisation stage was expensive in terms of area and delay. However, the division stage where the division iteration was performed had a small area and has low latency.

As the division stage had a lower latency than the initialisation and accumulation stages, the division stage iteration was unrolled to reduce number of clock cycles required to achieve t iterations. Other arrangements of timing elements are possible to balance the total number of cycles and cycles latency.

More familiar division algorithms such as SRT achieve speedup through use of higher radices than base 2. Unlike SRT, the divider presented here cannot benefit from higher radices. High radix dividers generate multiple quotient bits per round by inspecting the residual and divisor with higher precision. There is no comparable operation in the approximating divider.

Pipelining registers for the approximating dividers were inserted between each stage. As discussed above, additional pipelining registers were inserted in the initialisation stage, requiring two initialisation cycles.

7.2.5 Variations

In this section, variants of the divider architecture are proposed to shorten the critical path and hence reduce the minimum clock period, at the expense of correctness.

Direct Implementation (DI) Divider: An implementation based on a direct implementation of the description above is shown in Figure 7.1a. This design contains two terms in the multiplication approximation, $p_1 + p_2$.

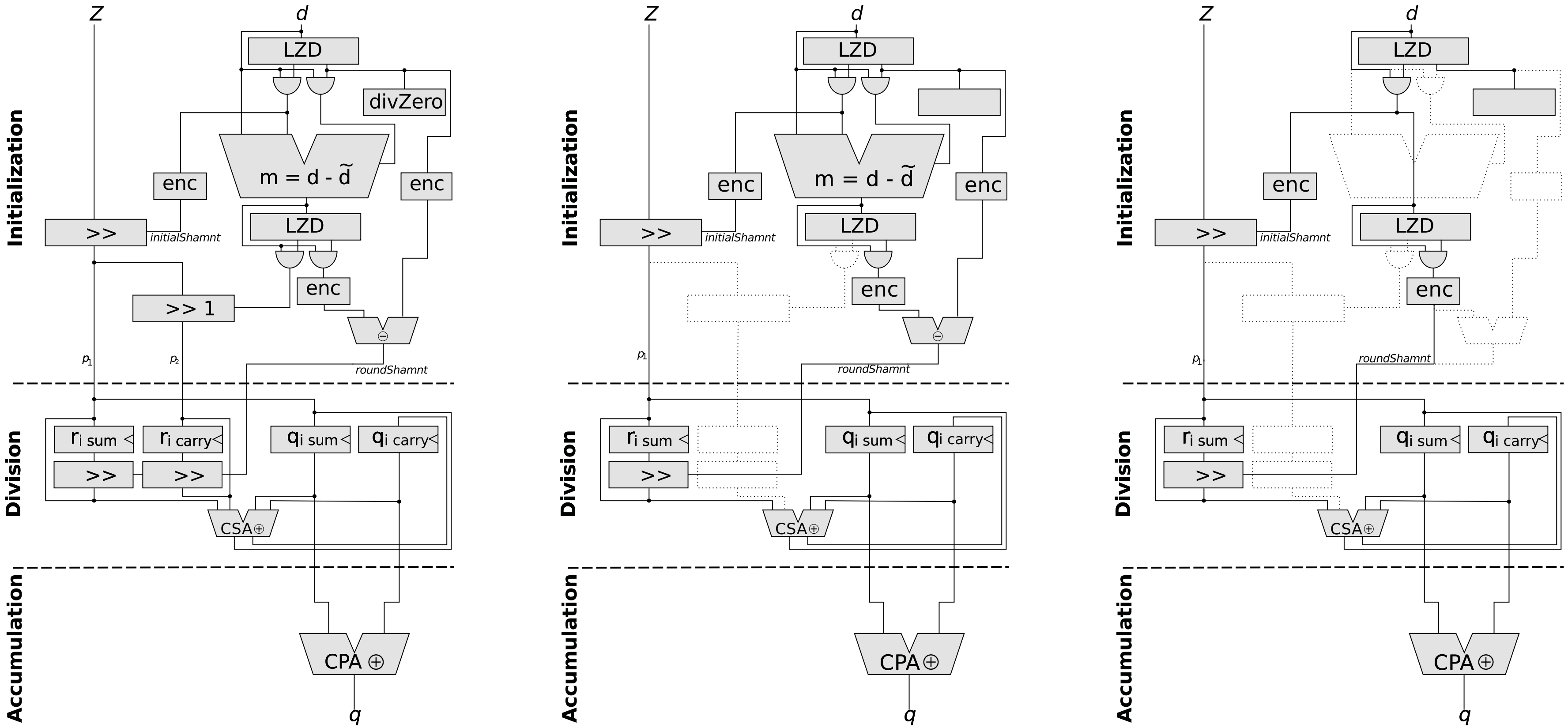
Single Multiplication Term (SMT) Divider: A further approximation to the numerator multiplication (see section 7.1.2.3) was made by discarding p_2. In the initialisation stage a right shift was no longer required. In the division stage the CSA tree was simplified to a simple CSA, and the r_{c_i} latch was removed. Changes are shown in Figure 7.1b.

Greatest Binary Power (GBP) Divider: The delay of the initialisation stage was reduced by not handling both of the $\tilde{d} = d_H$ and $\tilde{d} = d_L$ cases. Instead, d_L term was generated from the output of the first *LZD*. Calculating the d_H term requires a full propagation subtraction, as discussed in Section 7.2.1.2. Fixing $\tilde{d} = d_L$ removed the need for the subtractor. The resulting latency was reduced to latency of the *SHIFT* and *AND* in the calculation of d_L.

Forcing the approximate divisor to d_L forced the error term e_i to be positive. The correction to the quotient q_i in each round was thus negated from the previous round—hence the approximate quotient converged like a dampened oscillation. The *roundShamnt* also altered, causing slower convergence. The GBP divider is shown in Figure 7.1c.

The GBP incorporated all the simplifications of the SMT divider.

A graphical representation of the convergence of the quotient for a sample division is shown in Figure 7.2. The exact quotient convergence was calculated using the GNU Multi-Precision library



(a) *Direct Implementation (DI) Divider*. Control elements such as multiplexors, timing elements and clock signals and latches are not shown.

(b) *Single Multiplication Term (SMT) Divider*. Latency in the initialisation stage was reduced by including only one term in the approximation of $r_i \times m$. Logic removed by this simplification is shown by the dotted outline.

(c) *Greatest Binary Power (GBP) Divider*. An n bit subtraction was removed from the critical path by fixing $\tilde{d} = d_L$. Redundant logic that was removed is shown by the dotted outline.

Figure 7.1: Designs for approximating unsigned integer dividers.

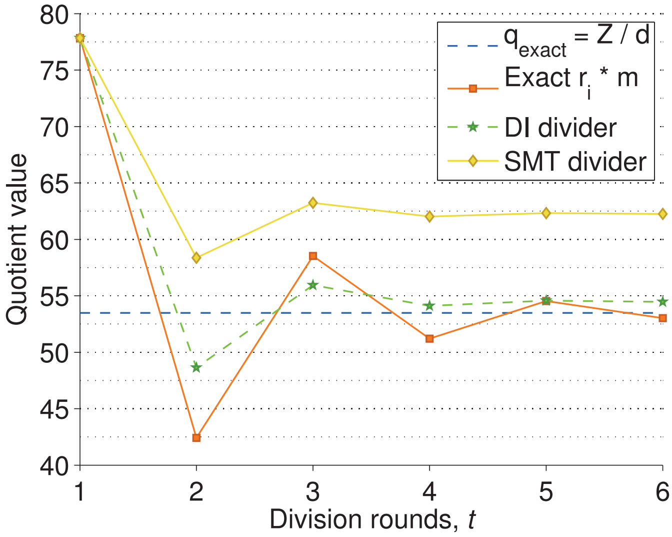


Figure 7.2: Convergence for the division of z/d; z=0x26EA316F and d=0xBA3F7F. The DI divider used two multiplication terms, but the SMT divider used only one.

(GMP) to prevent quantisation error [Free Software Foundation, Inc. , 2008]. In this example, the DI divider with 2 multiplication terms successfully converged, but the SMT divider with 1 multiplication term did not. Other observable effects include quantisation error, and loss of precision due to the lack of a p_2 term.

7.3 Probability of correctness

In this section the correctness of each approximating divider was measured as the proportion of correct quotients calculated for a set of divisions, given a number of division rounds t and fractional bits f maintained. A comparison was made with a variable latency SRT divider, by investigating the number of SRT division rounds required to achieve the same correctness as the approximating dividers.

7.3.1 Approximating dividers

Operation of the approximating dividers was simulated by a program written in C, and used to profile the correctness of the approximating dividers operating on random numbers and benchmark data.

7.3.1.1 Random input probability of correctness

A set of 10 million random numbers was generated using the GNU `srandom()` function, however, uniform random numbers do not mimic typical divisions observed in program execution very well. When $d > z$, which was often for uniform random numbers, the quotient was correctly set to zero, and the approximating dividers did not require any cycles in the division stage, inflating the correctness of the dividers. To mitigate this effect and handicap the approximating dividers, the uniform random numbers were biased such that $d < z$ by

$$d \leftarrow d_{\text{biased}} = d_{\text{uniform}} \bmod (z+1)$$

Biased random numbers were used to test the effect on correctness of using one or two multiplication terms in the approximate division. Figure 7.3a shows the correctness of the DI divider (2 multiplication terms), and Figure 7.3b shows the correctness of the SMT divider (1 multiplication term) respectively. Peak probability of correctness of 83 % occurred with f=8 and t=3 in the DI divider for random inputs. The difference in correctness of the dividers is shown in Figure 7.3c. The maximum difference in correctness when using a single multiplication term in the SMT divider was 6.24 %, when f=8 and t=3.

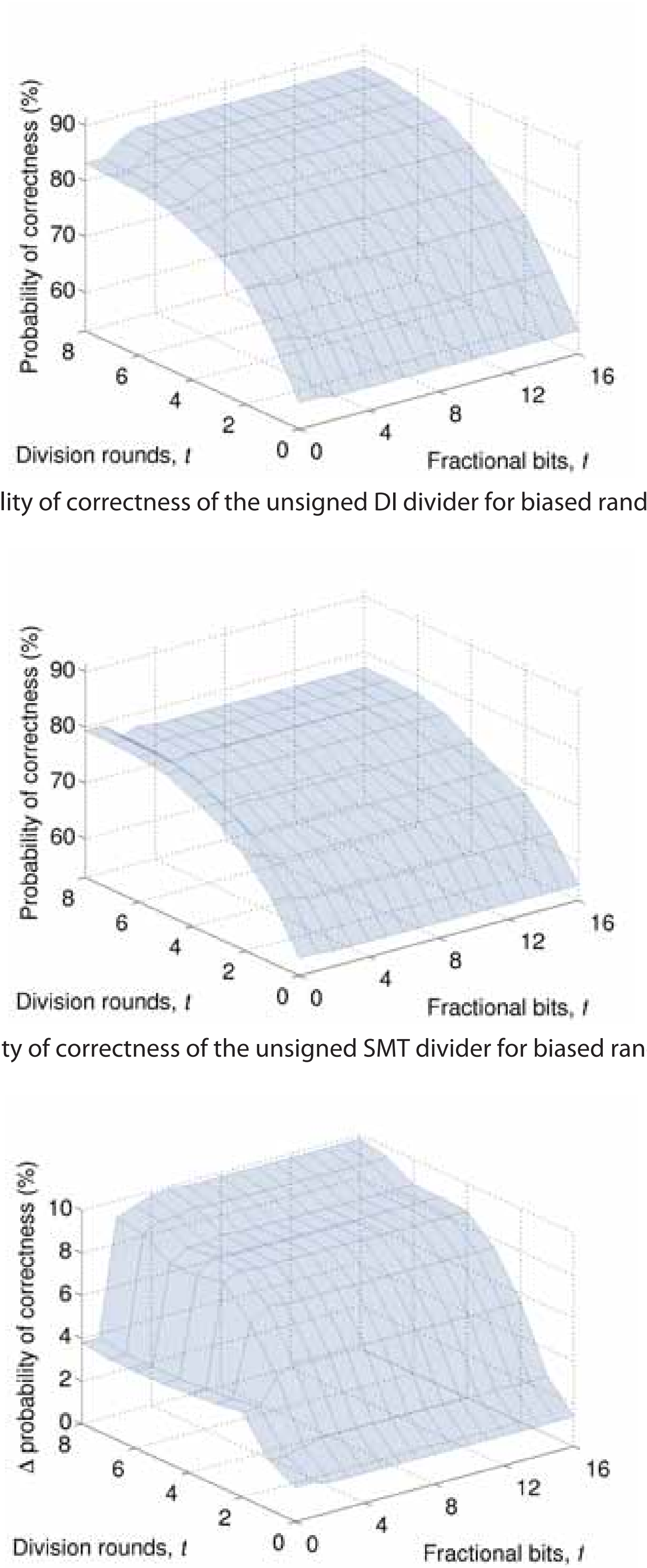
7.3.1.2 Benchmark probability of correctness

Tables C.1, C.2 and C.3 on pages 319–321 show the occurrence of division instructions in each benchmark. Tables C.1, C.2 and C.3 show that on average the number of unsigned division operations for *arithmetic*, *Mediabench* and *SPEC* benchmarks ranged from 0.05 % to 0.6 % of all instructions retired. A division by zero did not occur in any benchmark, but on average 6.9 % of unsigned dividends were zero.

Figure 7.4 shows average correctness for *arithmetic*, *Mediabench*, *SPEC* and random data. A peak probability of correctness of 99.4 % occurred for f=2 and t=7 in the *Mediabench* data. All benchmarks achieved a maximum correctness of at least 75 %, except the *253.perlbmk*, *181.mcf* and *197.parser* integer benchmarks in the *SPEC CINT2000* set, which exhibited a peak correctness of under 43 %.

The correctness profiles of the average *arithmetic*, *Mediabench*, and *SPEC* benchmarks were similar in shape, but had different peak and average values. In later synthesis results and simulations, the dividers with f=2 and t=7 were used as the approximate dividers in the processor pipeline.

The correctness profile of the approximating dividers with benchmark inputs differed to uniform

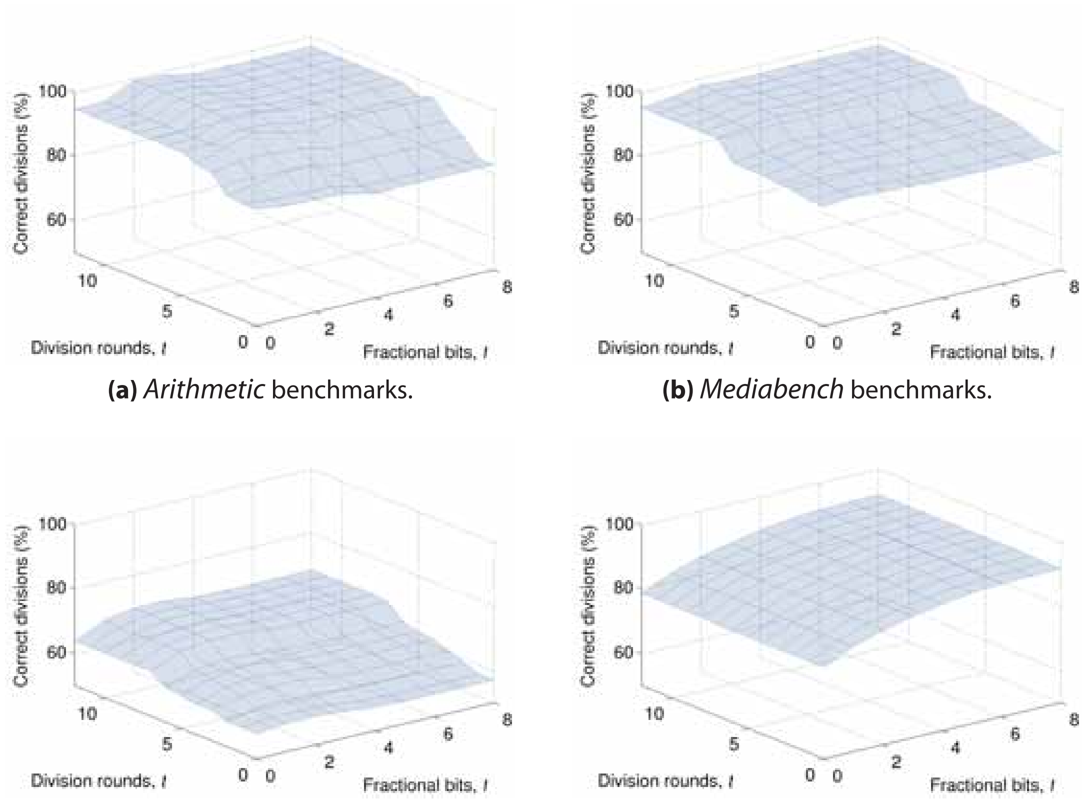


(a) Probability of correctness of the unsigned DI divider for biased random inputs.

(b) Probability of correctness of the unsigned SMT divider for biased random inputs.

(c) Difference in probability of correctness for DI and SMT divider, for uniform random inputs.

Figure 7.3: Correctness of DI and SMT dividers for biased random inputs.



(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

(d) *random* data.

Figure 7.4: Average probability of correctness for approximating dividers operating on benchmark data. The average maximum difference in correctness of the DI, SMT and GBP dividers was within 0.3 %.

random input profiles in their dependence on the division rounds t rather than the number of fractional bits f. An unusual peak for f=2 is apparent in the *Mediabench* output, and is attributed to quantisation error and the distribution of input operands. The average correctness for the *Mediabench* benchmarks of the DI divider and SMT divider were negligibly different. The maximum difference in average correctness did not exceed 0.01 %. The average correctness of the further simplified GBP divider was within 0.3 % of the other approximating dividers. Hence, the best design is the unsigned GBP divider, as it has the least overhead in the initialisation stage.

The approximate DI, SMT and GBP dividers are synthesised in Section 7.5, to investigate the delay, area and power of the approximate arithmetic units. Despite a peak average correctness of 99.3 % for *arithmetic* benchmarks, and 97.8 % for *Mediabench*, the average correctness quoted is only 89.5 %, due to the low correctness obtained in *SPEC* benchmarks.

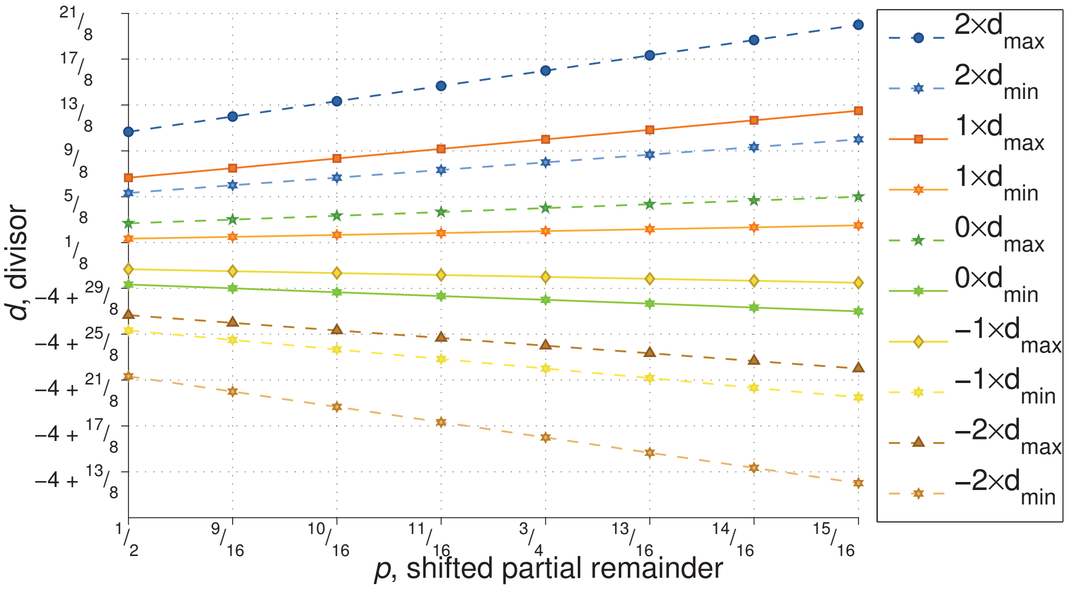


Figure 7.5: P-D diagram showing the selection intervals for the quotient digit selection stage of the baseline SRT divider. The quotient digit is selected based on the divisor and shifted partial remainder in each division round. The range of each possible quotient digit are colour coded.

7.3.2 Baseline SRT divider

A 32/32 bit variable latency radix-4 SRT divider was selected as a baseline, because SRT is widely implemented and radix-4 designs do not suffer the exponential area increase of higher radix designs [Harris et al., 1997; Oberman and Flynn, 1997]. The baseline SRT divider was written in synthesisable VHDL. Like the approximating dividers, it did not output a remainder, saving a carry propagate adder. The divider stored the residual in redundant carry-save format, and generated the quotient digit combinatorially. The baseline divider operated in 3 stages, like the approximate divider.

A P-D diagram of the quotient digit selector used in the baseline SRT divider is shown in Figure 7.5. The P-D diagram shows the possible minimum and maximum values for each quotient digit, so that the partial remainder result is in the range $[-\frac{1}{2}, \frac{1}{2})$.

In the initialisation stage, SRT operands could not be pre-normalised (shifted) to avoid overflow. The pre-normalisation shift amounts were used to calculate the number of division rounds required, and the number of bits to shift the quotient to the correct numeric significance. The initialisation stage was initially set a latency of 1 cycle.

In the division stage the residual was saved in stored carry form to reduce latency. The leading bits

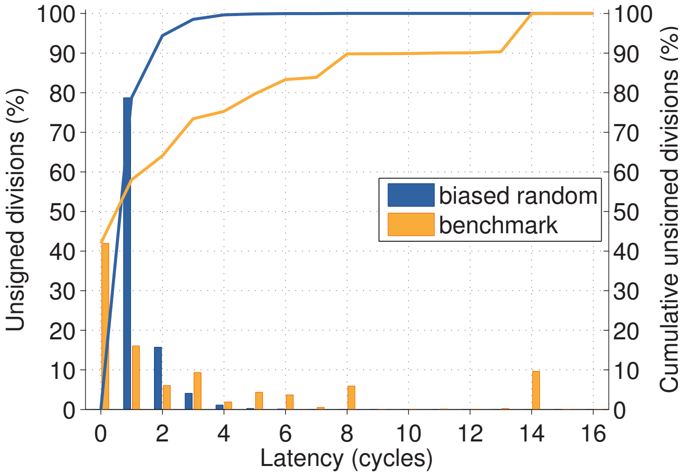


Figure 7.6: Histogram of the number of cycles required by the baseline radix-4 SRT divider in the division stage. The distribution bars are shown on the left y-axis, and the cumulative line plot on the right y-axis.

of the residual and divisor were inspected to determine the quotient digits combinatorially, from the redundant digit set $[-2, 2]$. An on-the-fly quotient conversion was employed to convert the redundant digits to non-redundant binary [Ercegovac and Lang, 2003]. The selected quotient digit was used to generate the appropriate multiple of the divisor and subtract it from the residual, for the next division round.

In the final stage, the quotient was shifted to the required significance, and possibly decremented if the residual (remainder) was negative. The final stage required 1 cycle.

Pipelining registers for the baseline SRT divider were inserted in between each stage. The delay of the quotient digit selection could be further reduced by using a fast lookup table.

In the worst case, a 32/32 bit radix-4 divider required up to 16 division cycles, because one radix-4 quotient digit was generated per division round. The baseline SRT divider could generate an approximate result by limiting the number of division cycles. Figure 7.6 shows the number of cycles required by the SRT divider in the division stage.

For biased random inputs, 0.0 % of divisions were correctly calculated with 0 division cycles. No exact quotient result was 0 because $d < z$, and at least one division round was required to calculate a non-zero quotient. With 3 division cycles the SRT divider could correctly calculate 99.5 % all random input divisions. The average number of division cycles required was small because both z and d were likely to be very large in magnitude.

The average operand distribution for benchmark operands was significantly different from random inputs. For benchmark inputs, 0 division cycles were required to correctly calculate 41.8 % of divisions, but 8 cycles were required to correctly calculate 89.9 % of divisions. In Section 7.5 the 8 cycle SRT divider is used to compare against synthesised approximate dividers in terms of latency, delay, area and power.

In Section 7.3.1.2 *Benchmark probability of correctness*, the GBP divider was determined to require 7 division cycles to correctly calculate 99.4 % of all *Mediabench* divisions. For the radix-4 SRT divider to have similar probability of correctness to the GBP divider, 7 division cycles, 1 pre-normalisation cycle and 1 post-shift cycle were required, for a total latency of 9 cycles.

7.4 Signed approximate integer division

This section discusses how a modification to the approximate dividers proposed was retrofitted allow the divider to operate on signed operands. It was desirable that a modification impose a low delay and area overhead, sacrifice little or no correctness, and work in the DI, SMT and GBP dividers.

The initialisation stage set the worst-case delay for the unsigned approximate divider—it was already long and complex. Much of the calculation effort was spent inspecting the divisor d using two *LZD* units, which assume that the divisor is unsigned (or positive). To handle negative signed operands, a leading ones detector (*LOD*) could have been used, but nearly all of the logic in the initialisation stage would have needed to be duplicated, and neither would the outputs have been amenable to the encoders that determine the shift amounts. Additionally, each shifter would have needed to be replaced with a conditional arithmetic shifter, to preserve the sign bits.

A simpler solution was to negate z and d when they were negative before operation in the initialisation stage. The quotient was conditionally negated after the accumulation stage, as determined by the initial sign bits of z and d. The number 2^{N-1} is not representable N bit twos complement, so the approximate divider was wrong for the case when $d = -1 \times 2^{N-1}$.

The simplest method of negating a signed number was to complement the number, and add 1 to the LSB. It was trivial to negate the quotient, because complementation is simple, and the final accumulation stage already employed a full propagate adder. In the initialisation stage z could be easily negated because the data path for z was much shorter than the data path for d, which also involves a full propagate adder/subtractor. Both z and q were negated with a conditional inverter/adder.

Table 7.1: An example demonstrating the approximation of $-d$. A twos complement number can be negated by inverting and incrementing. Each column shows a separate example.

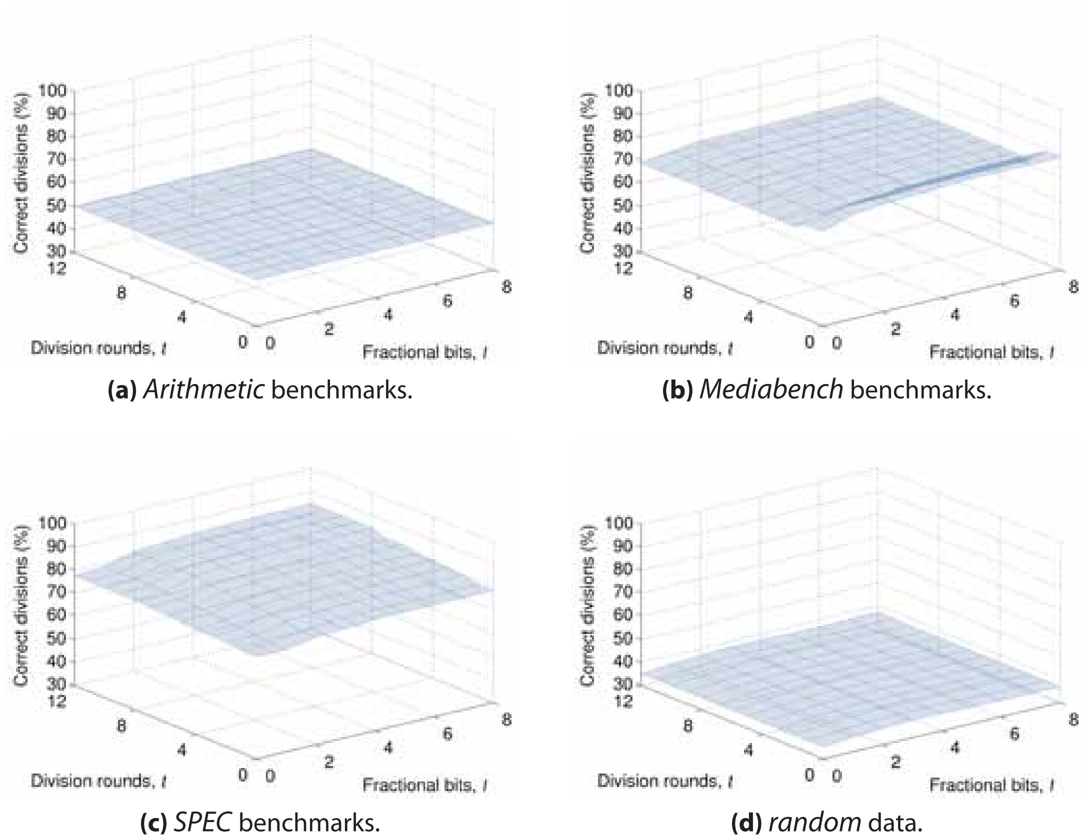
Operation	+ve d	−ve d (exact)	−ve d (approx.)	−ve d (approx.)
d	$\texttt{0}\underline{\texttt{11}}\texttt{00101}$	11010000	11010000	11111000
invert		00101111	00101111	00000111
increment		$\texttt{00}\underline{\texttt{11}}\texttt{0000}$	$\texttt{00}\underline{\texttt{11}}\texttt{0000}$	$\texttt{0000}\underline{\texttt{1}}\texttt{000}$
initialShamnt	6 ✓	5 ✓	5 ✓	2 ✗
roundShamnt	5 ✓	4 ✓	3 ✗	1 ✗

It was more difficult to negate d, because extra hardware on the data path added to the total delay. Also, the negation couldn't be incorporated into the existing adder/subtractor for m; the negation had to be performed before the first *LZD* so the *initialShamnt* could be encoded (see Figure 7.1b). It was undesirable from a delay perspective to introduce another full propagate adder at the d input. Instead, the divisor d was complemented (if negative), *but not incremented*.

If the most significant bit of the negated d was incorrect, then the *initialShamnt* was incorrect. If the next most significant bit was incorrect, then the *roundShamnt* was incorrect. Table 7.1 shows an example of approximating the negation of d.

The baseline unsigned radix-4 SRT divider could handle signed inputs with simple modifications. Because the unsigned baseline divider selected quotient digits from the set $[-2, 2]$, the adder for the partial remainder was widened to operate on signed digits. The additional bits were used to store the extended sign bits for the divisor and shifted partial remainder. Additional logic was required for the signed SRT divider to inspect the input operands and perform the post-division shift. The signed and unsigned baseline exact integer dividers had similar delay, area, and power characteristics due to the small differences.

The baseline radix-4 SRT divider was set to operate on signed operands from the benchmark data set, and the number of division cycles was recorded. Figure 7.8 shows that the average number of division round cycles required for a target correctness of ≥90 % reduced to 7 cycles for signed benchmark operands. Thus, the total division latency for the signed SRT divider was 1 initialisation cycle, 7 division cycles and 1 accumulation cycle—a total of 9 cycles. The SRT divider attained 94.2 % correctness with 7 division cycles.



(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

(d) *random* data.

Figure 7.7: Average probability of correctness for signed approximating dividers operating on benchmark data. The maximum difference in correctness of the SMT divider compared to the DI divider was as much as 22 % less, and the GBP was as much as 21 % less for *SPEC* benchmarks.

Table 7.2 shows the peak average correctness when the approximating dividers operated on each data set, and the overall average of all benchmarks. The number of fractional bits f and number of division rounds t for the divider with the peak correctness is shown. The correctness profiles of the signed dividers in Figure 7.7 are much flatter than the unsigned profiles. It was determined that only 4 signed division rounds was required to maximise the peak average correctness over all benchmarks.

The correctness penalty was determined by simulation of benchmark data. Table 7.3 shows the average difference in correctness the negation of d in the initialisation stage was approximated. The dividers were compared when d was properly negated, and approximated by just complementing. The signed DI divider suffers the greatest correctness penalty in all benchmarks compared to the unsigned dividers. While the correctness of the signed SMT dividers were negligibly affected, the GBP dividers improved very slightly, due to a much repeated pathological case in the *253.perlbmk* and *181.mcf* benchmarks in the *SPEC* suite.

Table 7.2: Peak probability of correctness for signed and unsigned DI dividers for benchmark and random data.

Benchmark	**unsigned**			**signed exact −*d***			**signed approx. −*d***		
	f	t	P_D (%)	f	t	P_D (%)	f	t	P_D (%)
random	8	3	92.06	4	4	37.59	9	4	92.07
Arithmetic	2	6	99.38	9	8	50.67	9	8	50.67
Mediabench	2	9	97.80	9	1	78.81	9	1	80.70
SPEC	3	10	70.72	9	13	84.66	9	13	88.33
Benchmark average	**2**	**7**	**89.52**	**9**	**4**	**74.96**	**9**	**4**	**76.74**

Table 7.3: Difference in correctness for signed dividers approximating $-d$ in the initialisation stage. A signed divider approximating $-d$ is compared to a signed divider that calculates $-d$ exactly.

Benchmark	**DI**	**SMT**	**GBP**
random	-54.47	0.00	0.00
Arithmetic	0.00	0.00	0.00
Mediabench	-5.65	0.00	0.00
SPEC	-13.90	0.00	4.25
Benchmark average	**-7.80**	**0.00**	**2.03**

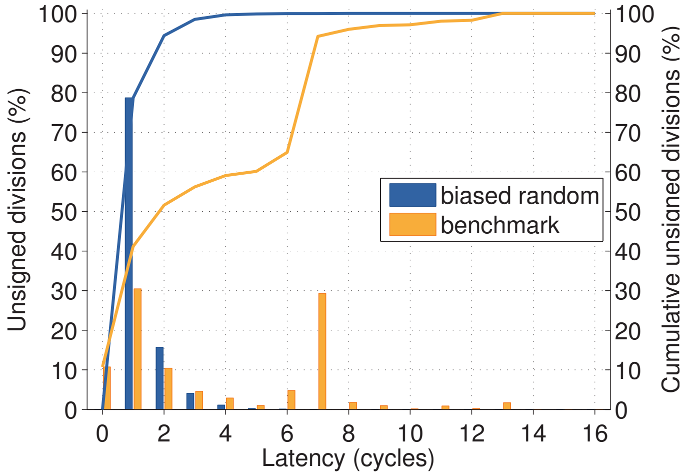


Figure 7.8: Histogram of the number of cycles required by the baseline radix-4 SRT divider in the division stage, when operating on signed inputs. The distribution bars are measured on the left y-axis, and the cumulative line plot on the right y-axis.

In Section 7.3.1.2 *Benchmark probability of correctness* a comparison was made between the peak average correctness of the unsigned DI, SMT and GBP dividers when operating on sets of benchmark data. Table 7.4 shows the difference in correctness for signed and unsigned SMT and GBP dividers, compared to the DI divider. The SMT and GBP divider suffer a small correctness reduction compared to the DI divider for unsigned random inputs, and negligible difference for benchmark inputs.

The differences in correctness for signed dividers are quoted from dividers that calculated the exact value of $-d$ in the initialisation stage. A further correctness penalty must be added from Table 7.3 when $-d$ is approximated. The difference in correctness for the signed SMT and GBP dividers compared to the signed DI divider for signed inputs was much greater than for unsigned inputs.

The correctness profiles of the signed divider in Figure 7.7 were different to the profiles of the unsigned divider in Figure 7.4. The number of division rounds and fractional bits in the signed divider do not have a strong individual effect on the correctness, which remained at a near-constant level. As seen in Table 4.4, most of the signed benchmark division operands were positive, so the divider did not suffer errors from negating the leading sign bits of z when negative. Figure 4.4 showed that the magnitude of the mostly positive signed benchmark divisors were likely to be lower than the magnitude of unsigned benchmark divisors. The fewest number of division rounds were required when the difference in the magnitudes of the divisor and dividend are very small (e.g. $z/1$), or very large (e.g. z/z). The most division rounds are required when $d \approx {}^{z}/_{2}$.

Table 7.4: Maximum difference in peak average correctness for the SMT and GBP dividers compared to the DI divider.

Benchmark	unsigned		signed	
	SMT (%)	GBP (%)	SMT (%)	GBP (%)
random	-2.46	-3.58	-20.76	-19.08
Arithmetic	0.00	0.00	0.08	0.15
Mediabench	0.00	-0.39	-11.26	-8.48
SPEC	0.03	0.27	-22.20	-21.51
Benchmark average	**0.01**	**-0.07**	**-11.76**	**-10.56**

The sources of error were limiting f, limiting the number of rounds t and approximating the multiplication of $r_i \times (d - \tilde{d})$. The probability of correctness of the signed divider was much lower than the approximate unsigned divider. This is because the same operands were frequently repeated, and because of the effect of complementation. Operations that included a negative divisor would need to be complemented before and after the operation. After the approximate division, a result would be calculated with fractional bits. In the unsigned case, the fractional bits would be truncated, yielding the integer quotient. In the signed case, the result may need to be complemented by inversion and incrementing. Frequently, the loss of precision in the fractional bits caused the least significant integer bits to be incremented when it shouldn't, or not incremented when it should. It was observed that the average error in the signed integer quotient was small.

7.5 Synthesis

Each approximating divider design and SRT baseline was synthesised from a VHDL description with the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library, under typical conditions. The synthesiser used was *Synopsys Design Compiler*. Each approximating divider was synthesised with f and t from Table 7.2 corresponding to the peak average probability of correctness across all benchmark applications. Table 7.5 shows the results from synthesis. The unsigned and signed exact SRT dividers required 1 pre-normalisation cycle, 8 or 9 division cycles for $\gtrapprox 90\,\%$ correctness, and 1 post-shift and quotient adjustment cycle. The approximate dividers required 2 initialisation cycles, 7 or 4 division cycles and 1 accumulation cycle.

Table 7.5: Results from synthesis of 32/32 bit dividers. Each design was optimised for delay. The minimum achieved clock period is shown for typical operating conditions and wire load. Dynamic and leakage power consumption estimates from synthesis are also shown.

Type	Design	Clock	f	t	Correctness	Cycles	Latency		Area	Power	
		(ns)			(%)		(ns)	(%)	(μm^2)	Dyn. (mW)	Leak. (nW)
Unsigned Exact	SRT radix-4	4.45	—	—	—	18	71.2		0.717	19.78	569.5
Exact (under-cycled)	SRT radix-4	4.45	—	—	89.5[*]	10	44.5	100	0.717	19.78	569.5
Approximating (unsigned)	*DI*	3.60	2	7	89.5[‡]	10	36.0	80.9 (-19.1)	0.802	10.71	563.5
	SMT	3.55	2	7	89.5[¶]	10	35.5	79.8 (-20.2)	0.675	9.21	483.6
	GBP	3.45	2	7	89.5[¶]	10	34.5	77.5 (-22.5)	0.539	10.09	408.4
Signed Exact	SRT radix-4	4.45	—	—	—	18	71.2		0.717	19.78	569.5
Exact (under-cycled)	SRT radix-4	4.45	—	—	94.2[†]	9	44.5	100	0.717	19.78	569.5
Approximating (signed exact $-d$)	*DI*	4.75	9	4	75.0[§]	7	33.3	74.7 (-25.3)	0.839	15.73	536.7
	SMT	4.29	9	4	63.2[§]	7	30.0	67.7 (-32.5)	0.799	18.04	595.6
	GBP	4.29	9	4	64.4[§]	7	30.0	67.5 (-32.5)	0.647	17.45	512.7
Approximating (signed approx $-d$)	*DI*	3.62	9	4	76.7[‡]	7	25.3	56.9 (-43.1)	0.605	18.50	524.4
	SMT	3.56	9	4	63.2[¶]	7	24.9	56.0 (-44.0)	0.492	16.98	458.1
	GBP	3.49	9	4	62.4[¶]	7	24.4	54.9 (-45.1)	0.621	21.82	544.2

[*] Histogram of correctness for the unsigned SRT divider in Figure 7.6.

[†] Histogram of correctness for the signed SRT divider in Figure 7.8.

[‡] 'Benchmark average' correctness in Table 7.2.

[§] Correctness penalty for approximation of $-d$ in Table 7.3.

[¶] Maximum difference in correctness in Table 7.4.

The synthesis results in Table 7.5 show the simplifications of the approximating signed and unsigned SMT divider and GBP divider do yield a lower minimum clock period, and thus lower latency, than the DI under typical conditions. As shown in Sections 7.3.1.1 and 7.3.1.2, the approximating dividers have a high probability of correctness. The hardware cost, as determined by area and power requirements of each approximating divider, are comparable to the baseline 32 bit SRT divider.

7.6 Approximate integer dividers for ADVS

The dividers in this chapter were designed for use in an *ADVS*-enabled processor pipeline. In Chapter 4 it was determined that an approximate arithmetic unit should have a correctness of 95 % when operating at approximately 80 % of the exact unit, so that the overall system performance gain is positive.

Several unsigned dividers were synthesised with similar characteristics to the target minimum. In particular, under typical operating conditions, the unsigned GBP divider offered a 22.5 % reduction in latency compared to the 32 bit radix-4 SRT divider operating with 89.5 % probability of correctness, and 51.5 % faster than an exact radix-4 SRT divider that always operates in 16 cycles.

Likewise, signed approximate dividers were synthesised under typical conditions, but were found to have a lower average correctness than the unsigned dividers (see Table 7.2). To handle signed operands quickly, a further source of error was introduced in the initialisation stage to approximate the negation of the divisor d when negative. Although this introduced a small correctness penalty (see Table 7.3), the average benchmark probability of correctness of was significantly lower than the target 95 %. One reason for the lower average probability of correctness in the signed dividers is the number of repeated operations in the benchmarks. Additionally, the number of benchmarks containing signed division operands is less than benchmarks with unsigned divisions (see Tables C.1–C.3).

The delay-correctness feasibility regions from Chapter 4 provided an indication of the minimum characteristics required of an arithmetic unit to be used in *ADVS*. However, the simulations performed assumed default values for the arithmetic latencies of each unit. The delay of the baseline SRT dividers developed in this chapter will also be incorporated into the *ADVS*-enabled model in future chapters. Although under the correctness target, the unsigned approximate GBP divider was selected to be employed in the system simulation. The peak performance of the signed divider was much lower, and was excluded from the system. Tables C.5–C.7 show that the average proportion of signed integer divisions is similar to unsigned division, except for the *arithmetic* benchmarks, due to the high number of signed divisions in *calc_pi*.

7.7 Conclusion

This chapter has presented three novel approximating divider architectures for unsigned binary integers. An implementation through synthesis has been presented with 3 variants that represent different correctness and delay tradeoffs. The dividers were characterised in terms of the number of division rounds t, number of fractional bits f used to calculate the quotient, and number of multiplication approximation terms. It was shown through simulation that the unsigned dividers performed similarly on division operations from benchmark applications.

The unsigned dividers presented in this chapter were suitable for *probabilistic* computing or speculation using *ADVS*, when adjusted to obtain the peak correctness through the selection of f and t.

A modification was proposed to handle to signed operands, and despite introducing another source of error, the additional correctness penalty was shown to be low. However, the correctness of the signed approximate dividers was too far below the target minimum to be considered for an *ADVS*-enabled system.

Chapter 8

Approximate Floating Point Arithmetic

"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years."

John Von Neumann (1903–1957)

In this chapter approximate integer arithmetic units from Chapters 5--7 are adapted for IEEE floating point units. Investigations of the latency and correctness of the proposed units are insufficient for use in an *ADVS* enabled system.

Long latency operations present the highest potential benefit in data value speculation schemes. The longer the latency of an operation, the higher the probability that a dependent operation will be blocked from issue while it is waiting for the source. Floating point operations are longer latency than integer operations because of the additional complexity imposed by the *IEEE-754* standard, such as handling special values, operand alignment, and result rounding. In particular, although floating point division is a low-frequency and high-latency operation, slow implementations degrade system performance in many applications. This is further exacerbated in wide issue superscalar processors [Oberman and Flynn, 1997].

This chapter discusses the design of approximate floating point units, and assesses their suitability for use in an *ADVS*-enabled system based on their latency and probability of correctness.

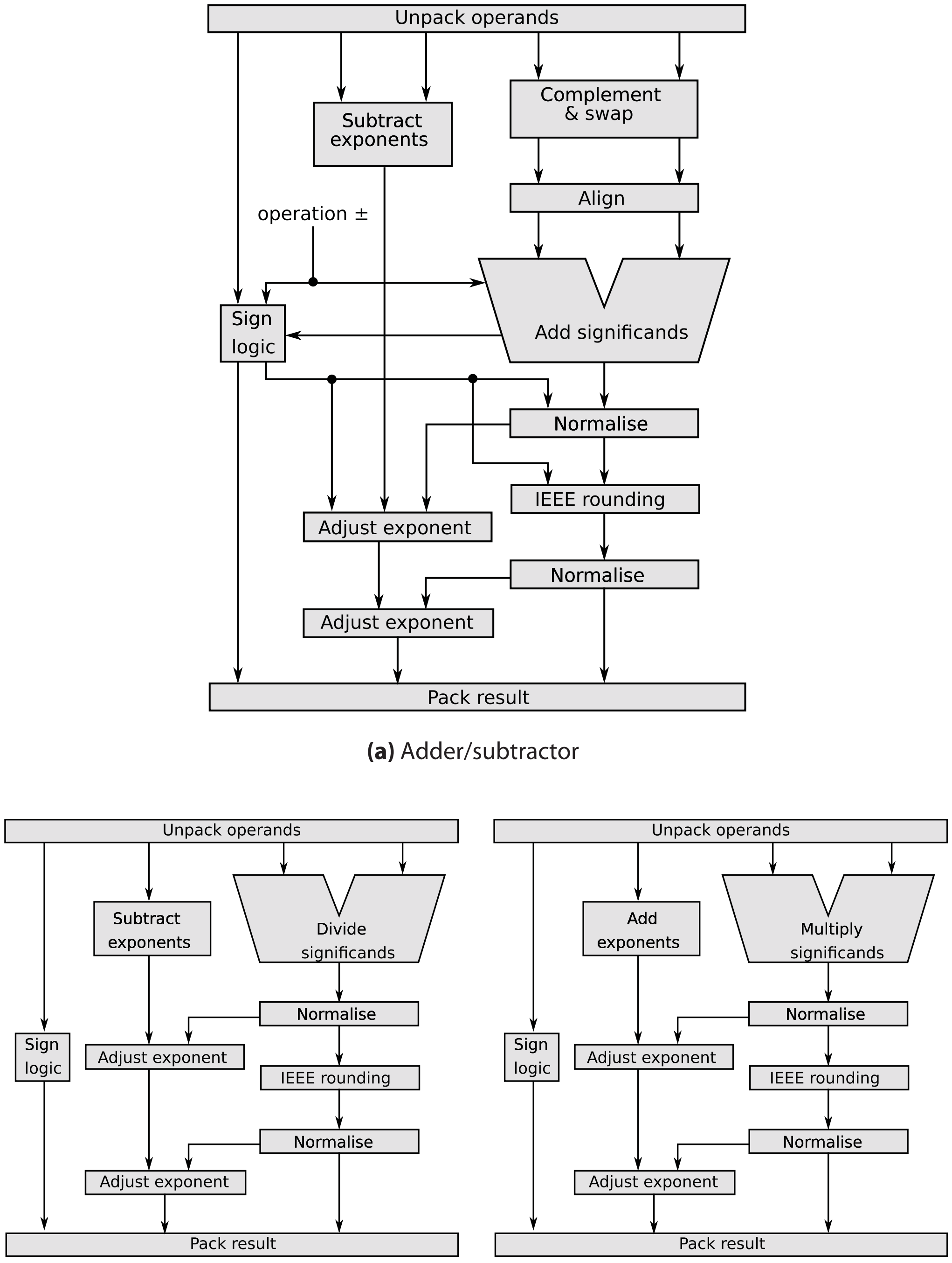
8.1 Approximating floating point units

This section briefly discusses the structure of floating point units, and possible modifications to yield approximate units with a high probability of correctness, and low latency.

IEEE-754 compliant floating point operations can considered in three basic stages under normal conditions. Special cases are ignored in this chapter, normally they can be handled in either hardware or software or a combination of both. In the first stage, the operands are unpacked to their sign, exponent and significand fields, and are aligned if necessary. In the second stage, input fields are operated on to calculate the result field, and in the third stage the data fields are updated if rounding or overflow occurs. Basic block diagrams of *IEEE-754* floating point units are shown in Figure 8.1. Logic for handling special cases is not shown. The basic designs of the different arithmetic units are similar—the input operands are first split into sign, exponent and significand fields, and operated on as independently as possible.

The sign, exponent and significand fields are partly on independent data paths, but later merge so that rounding and overflow detection can be performed.

Floating point operands are represented in sign-magnitude form, hence calculations for the exponent and significant can use unsigned arithmetic units. This presents an opportunity to substitute arithmetic components with approximate units that could have a high probability of correctness.



(a) Adder/subtractor

(b) Divider

(c) Multiplier

Figure 8.1: Block diagrams of IEEE Std. 754 floating point units.

The sign and exponent fields are handled with simple logic and adders; most of the complexity is in the significand calculation, rounding, and special case logic.

All floating point results must be rounded using the active rounding mode, which can induce an additional adjustment to the exponent, depending on the operand values. Floating point addition also requires an alignment shift after the input operands are unpacked (see Figure 8.1a).

Each stage is considered for approximation:

unpacking/alignment Unpacking is a short operation, and alignment is only required for floating point addition. Alignment is a highly serialised operation, involving inspecting the exponent bits and variable shifting the significand. Alignment shifts are common, and difficult to approximate because the shift amount is encoded to save interconnect. Logical incompleteness introduces a high probability of error.

field operation The data operations (sign, exponent, and significand) are integer operations, simple logic, or selections, at the appropriate bit width. The exponent and significand are both unsigned fields. The integer operations can be approximated using the same methods applied to regular integer arithmetic instructions.

adjustment/rounding The exponent overflow adjustment and significant rounding can both be performed by adding or subtracting a small constant offset. This logic can be simplified from the general case of adding and subtracting a variable. It can be omitted entirely, if all cases are generated, and then selected when the desired outcome is known. Furthermore, removing the rounding step entirely is not feasible; the effect is the same as forcing the rounding mode to zero—shown in Table 5.2 to have an average correctness of approximately 80 %, less than required for *ADVS*.

The most feasible method of approximation reduces the latency of the integer operation in the significand calculation. The significand is the widest field and hence more likely to be on the critical path. Simultaneously approximating the exponent is therefore not likely to shorten the critical path, but would introduce more sources of error in the calculation.

In this chapter approximate integer arithmetic units developed in previous chapters are modified for use in *IEEE-754* floating point units.

8.1.1 Approximation techniques

The main components in floating point calculations are discussed below, with an emphasis on their overall contribution to the critical path delay, and their potential for approximation with a high probability of correctness.

8.1.1.1 Exponent addition and subtraction

The addition or subtraction of the exponents is performed in parallel with the significand calculation. In the first exponent calculation, both of the exponent fields could be anywhere within the full representable range, but later adjustments only increment or decrement the exponent with a small correction, usually ±1. The exponent operation is a lower latency operation than the significand operation because the exponent is narrower, and addition and subtraction are simpler to implement that other significand operations. The exponent latency is not on the critical path, hence approximating the exponent is pointless.

8.1.1.2 Significand operations

The significands are the widest fields in floating point numbers, and the significand results are required for rounding and exponent adjustment. The operations are therefore unavoidably on the critical path.

Significand addition

The significands are added in a floating point adder (see Figure 8.1a). Attempting to approximate the significand adder is worthwhile because it is on the critical path. Liu and Lu's design can be implemented, however to be feasible the average worst-case carry length must be very short in typical floating point operands, so that the probability of correctness is high. Figure 3.6 shows that as the carry segments increase in length, the fan-in load increases rapidly until the adder latency is greater than a high performance exact adder.

Significand multiplication

The multiplication of the significand fields in an *IEEE* floating point multiplier was on the critical path, as determined by analysing timing data in synthesis log files. The significand multiplier is a good candidate for approximation. The most promising approach to designing an approximate multiplier with a favourable latency/correctness tradeoff is by utilising the approximate compressor multipliers from Chapter 6 *Approximate Integer Multiplication*. Section 8.2.2 discusses the correctness of approximate 24 bit significand multipliers, and Section 8.3 investigates their latency through synthesis.

Significand division

The division of the significand field is iterative. If an approximating divider could be made to do more work in one cycle, the exact divider probably could too, unless the approximate divider is structurally incomplete. If so, any errors introduced in the division iteration are not likely to be corrected, and may compound in magnitude. The output will be incorrect if an error occurs in any iteration.

Like the baseline `fpDiv` unit, the approximate unsigned integer divider from Chapter 7 was adapted for the approximate `fpDiv`. The correctness of the divider was measured for floating point operands using benchmark data in Section 8.2.3.

8.1.1.3 Normalisation

The normalisation stage includes detecting the significand MSB; shifting the significand; performing rounding; and possibly performing a full propagate addition of the significand result and two full propagate additions of the exponent result. The location of the MSB must be detected in case of denormal inputs, which is a time consuming step. As all of the operations are serialised and affect the both significand and exponent, it was decided not to approximate any of the normalisation steps. Additionally, the adders are narrower than the 32 bit adders investigated in Section 3.2.1.10. Due to logarithmic relationship between the AWCCL and the operand width (see (3.1)), the potential latency savings are lower for narrower operands.

An alternative for approximating a floating point unit is to omit the rounding and normalisation stage. This was considered in Section 5.2.6, but the highest average correctness in the *SPEC* benchmark suites was less than 81 % for `fpMult` and less than 75 % for `fpDiv`. The probability of correctness was considerably lower in other benchmark sets.

8.2 Approximate FP unit correctness

Approximate floating point units were constructed by replacing the significand operation with an approximate unit. This approach was selected because the significand lies on the critical path, the approximate logic is contained to a single unit, and there are opportunities to vary latency or correctness as desired. The probability of correctness of the approximate floating point units was de-

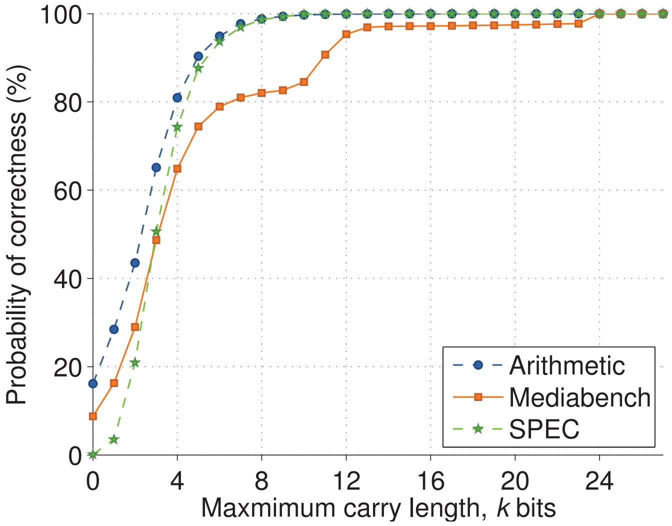


Figure 8.2: Probability of correctness of a Liu and Lu adder used for the significand field in a floating point adder.

termined by simulating the operation of just the significands with approximate hardware.

Operands were traced from the benchmarks using *SimpleScalar*. Extracting the significand fields was straightforward because none of the floating point operands were denormal.

8.2.1 Approximate FP adder correctness

The average correctness of benchmark data was profiled using a Liu and Lu adder with increasing maximum carry lengths. Figure 4.6 on page 99 shows that the significand bits are asserted less frequently than in the integer operands in Figure 4.2. Furthermore, when the significand alignment shift in the `fpAdd` is large, there are fewer asserted bits and less opportunity for carry propagation. Figure 8.2 shows the probability of correctness of an approximate 27 bit Liu and Lu adder. The 27 operand bits were comprised of the 23 significand bits, the hidden bit, and the guard, round and sticky bits used for *IEEE-754* rounding.

Compared to adding integer operands (shown in Figure 5.2), there is more of a difference in the AWCCL between benchmark sets for the floating point data. The *Mediabench* benchmarks in particular exhibit a small plateau between bits 6–10, and a rapid increase between bits 10–12Ṫhis characteristic was introduced by a few of the audio decoding applications, where operands were frequently accumulated resulting in the addition of large and small numbers.

Worst case carry chains of 7 bits are required for 95 % correctness. Recall that with a 7 bit carry

Table 8.1: Probability of correctness (%) of 24 bit significand multipliers operating on benchmark data, using approximate counters. The counters were arranged in a tree structure for multioperand addition and have n input bits and m output bits.

				m		
		1	2	3	4	5
	2	29.97				
	3	29.94	100.00			
	4	29.96	57.44	100.00		
	5	29.94	51.57	100.00		
	6	29.94	46.70	100.00		
	7	29.94	44.76	100.00		
	8	29.94	43.68	94.01	100.00	
n	9	29.94	43.15	93.37	100.00	
	10	29.94	43.21	92.60	100.00	
	11	29.94	42.58	91.83	100.00	
	12	30.00	42.17	91.32	100.00	
	13	29.99	42.76	90.63	100.00	
	14	29.97	42.22	89.61	100.00	
	15	29.97	41.81	87.68	100.00	
	16	29.96	41.50	86.35	100.00*	100.00

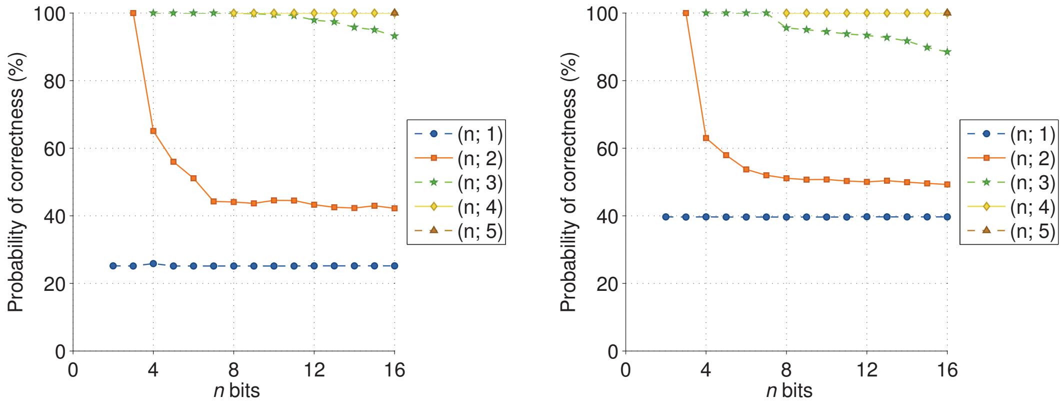
*Result rounded to 100.00 %.

chain, a Liu and Lu adder is slower that a Sklansky adder. Hence, using Liu and Lu's adder for a high performance approximate floating point adder is infeasible.

8.2.2 Approximate FP multiplier correctness

The approximate multiplier design in Chapter 6 were adapted for use as a significand multiplier with narrower operands. The correctness simulations were repeated with benchmark data derived from floating point operands. Table 8.1 shows the average correctness of $(n; m)$ 24 bit significand multipliers. Few of the multipliers meet the required 95 % correctness threshold for *ADVS*. However, the suitability of the multipliers also depends on the circuit latency.

Figure 8.3 shows the average correctness when the number of input bits to each counter is varied,



(a) Random inputs.

(b) Benchmark inputs.

Figure 8.3: Approximate counter input bits n vs. correctness of an *fpMult* significand multiplier. The tree multiplier was constructed entirely from one type of approximate counter.

but the number of output bits is fixed. Only counters where $n>m$ are shown.

Benchmark operands tend to contain repeated operations, and the magnitude of both operands is usually similar. The probability of correctness of floating point products is sensitive to the density of ones in the most significant bits of the significand. This is because the result N bit significand is taken from the upper bits of the $2N$ bit product, and the upper bits accumulate more carries than the lower bits. All broken carry paths are a potential source of error when the maximum carry length is limited.

Figure 8.3 shows that the correctness of the random data is often higher than the benchmark data in the same multiplier. This effect is probably exacerbated by repeated pathological cases in the benchmark data, and a relatively small sample size. There were only 8 benchmarks that contained `fpMult` operations, including all of the *SPEC CFP2000* benchmarks. Interestingly, the average correctness for the *177.mesa* benchmark in the *SPEC* benchmarks, and the *Mesa* benchmark in the *Mediabench* set were quite different. Both benchmarks share a similar code base forked from Brian E. Paul's clone of the OpenGL library, Mesa version 2.0. The primary difference is the input data and tasks performed. The *Mediabench* version primarily performs mipmapped texture generation and rendering, while the *SPEC* version maps contour lines onto a 3D surface. This case highlights the strong dependence of correctness on input data.

8.2.3 Approximate FP divider correctness

In this section the DI, SMT and GBP dividers from Chapter 7 are resized and adapted for the simulation of a floating point significand divider.

The significands can be treated as large unsigned integers. The important difference is that the integer quotient is shifted so that only the upper integer bits are retained, and the fractional bits are discarded. All of the desired bits in a floating point significand are fractional, so the significand quotient bits are shifted so that the result contains a fixed number of bits. Thus, the approximate dividers cannot 'short cut' and avoid computation of bits that would otherwise be discarded because they are now retained. The higher precision of the significands compared to typical integer operands increases the probability that some bits are approximated incorrectly.

Figure 8.4 shows the average error of the incorrectly approximated significands, using dividers with different numbers of multiplication terms. Recall that in the approximate division algorithm, a multiplication was required for the term $r_i \times (d - \tilde{d})$. To reduce the delay of each iteration, it is necessary to approximate this by including fewer partial products. The figure shows an infeasible hypothetical divider using an exact multiplication, and two approximate dividers with 2 and 1 terms (the approximate DI and SMT dividers respectively).

The hypothetical divider with the exact multiplication term shows that the magnitude of the error (the difference between the exact result and the approximate result) decreased as the number of division rounds increased. This is expected because as the number of division rounds increases, the more iterations that the result has to converge on the exact result. The number of additional fractional bits in the intermediate results had a small effect of the error.

The correctness of the hypothetical divider with an exact multiplication term (shown as a blue plane) is erratic. A higher error rate is noticeable when the number of division rounds is an odd number. This effect was previously illustrated in Figure 7.2 on page 173, where the correction to the intermediate is added then subtracted alternatively in successive rounds. The chance of this type of correction is roughly 50 %, to the effect is apparent in the average.

Approximate dividers require more division rounds to converge when the distance in bits between the MSB and the next most significant bit is short, or when the magnitude of the operands is large. The correctness of the quotient bits can oscillate each round when the next most significand bit in the divisor is asserted. The number of additional bits maintained in the intermediate calculations (corresponding to 'fractional' bits in the integer divider) has minimal effect on the error because errors are more likely to be introduced by the high loss of precision in the approximate multiplication

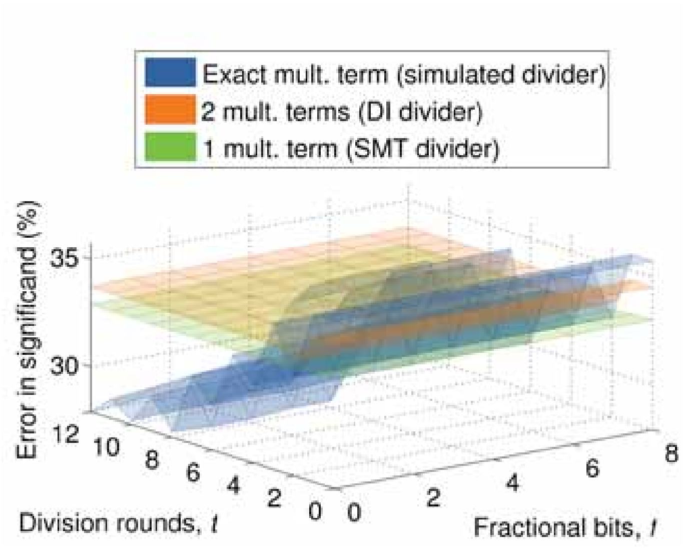


Figure 8.4: The average percentage error in the *fpDiv* approximate significand relative to the exact result for benchmark data.

term.

The average error of the DI and SMT dividers was nearly constant despite varying the number of division rounds and number of maintained fractional bits. This suggests that the correctness of these dividers is low, and that the number of partial products maintained in the approximated multiplication term are critical to the correctness of the result. The integer dividers were able to get away with fewer multiplication terms, because the precision of the integer quotients was low on average. For floating point numbers, the precision is fixed at the operand width, and the MSB is guaranteed to be asserted due to the 'hidden one'.

Many division rounds were required to reduce the average error, compared to the implementation of the integer divider in Section 7.6, however, the average magnitude of the error does not matter for application in *ADVS*.

Figure 8.5 shows the average correctness of the approximated significands for floating point division, using appropriately sized DI, SMT and GBP dividers (see Section 7.2.5). The average correctness of the operations in the *arithmetic*, *Mediabench* and *SPEC* benchmark sets were similar, and far lower than the threshold required for *ADVS*. The low correctness is due to the high precision required in the intermediate calculations of the approximated result.

Similar to the results of approximate integer division, the differences in correctness between all dividers was at most 0.06 %. The GBP divider is not shown, but the results were similar. The correctness increased minutely when the number of division rounds was increased. However, due to

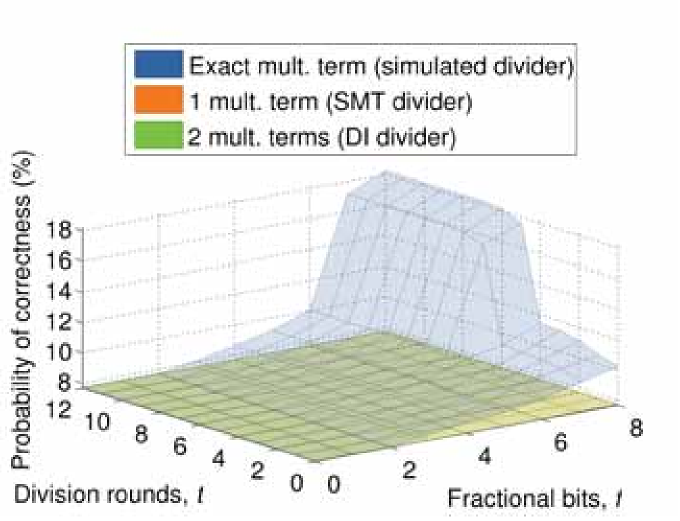


Figure 8.5: Average correct unsigned significand divisions when the approximate divider uses an exact multiplication term, or an approximation with 1 or 2 partial products. The SMT and DI dividers are very similar and appear superimposed.

the small number of maintained partial products for the multiplication term, the full precision of the significands could not be often accurately calculated.

Only the DI divider showed a significant increase in correctness when the number of fractional bits and division rounds were both high. This increase did not occur in the SMT and GBP dividers, showing that the additional term in the approximation of $r_i \times m$ (see Section 7.1.2.3) is important for correctness. (Recall that the DI divider uses two partial products, and the SMT and GBP dividers use only one). To increase the average correctness further, more multiplication terms should be used $p_1 + p_2 + p_3 + \cdots$.

Dividers with higher correctness due to more division rounds and fractional bits also have a longer latency and do not offer an advantage over exact floating point units. Summing more partial products $p_1 \cdots p_N$ in the multiplication term further increases the latency. The correctness of the dividers simulated above was so low that approximate floating point division was also abandoned for this project.

8.3 Synthesis

A set of significand multipliers were synthesised to extract the circuit delay, so that the delay vs. correctness profile could be established. The same method as Section 6.5 was followed, and all feasible tree multipliers using approximate counters up to $(16;5)$ were synthesised with the TSMC Artisan 0.18 μm process. The approximate multipliers were automatically generated in a Wallace tree structure using *multgen* (see Section B.4). The baseline multiplier was the exact $(3;2)$ multiplier, using *FA*s as each counter cell.

8.3.1 Baseline

Synthesisable *VHDL* models of the exact and approximate floating point multipliers were written to benchmark the latency, area and power of approximate floating point units. Both share common components like the prenormalisation logic and exponent rounding. The most significant differences were the implementations of the significant operations.

The baseline exact significand multiplier component was adapted from the exact $(3;2)$ tree multiplier in Section 6.5, but narrowed for the width of the significands. The *VHDL* was automatically generated with a modified version of *multgen* (see Section B.4).

The binary product of an integer multiplication is normally as wide as the concatenation of both input operands, but this much precision is not required in floating point. The hidden one is inserted into the MSB of the significand for regular numbers, hence only the upper bits need to be maintained. The lower bits can be discarded, except for 3 bits called guard, round and sticky that are kept for rounding. The guard and round bits are regular bits from the result, but the sticky bit accumulates any discarded bits of lower significance by an *OR* operation.

All of the components of the exact and approximate floating point units were described in structural *VHDL*, with the exception of the exponent adder. This was written using behavioural *VHDL* knowing that the *Synopsys Design Compiler* synthesiser implements a carry-lookahead adder. The exact baseline floating point models were kept for a comparison of total system area and power in Chapter 11.

8.3.2 Approximate units

It was assumed that the approximate multipliers would occupy their own pipeline stage(s), and so they were synthesised in isolation for simplicity. The input drive and output capacitance was set to match a unit sized *DFF*, with no intermediate pipelining registers. The circuit delay and correctness of the multipliers is shown in a scatter plot in Figure 8.6.

The delay/correctness characteristic for the floating point significand multipliers was different to that observed for integer multipliers in Chapter 6. Vertical dashed lines show the drop in correctness for the same approximate multipliers that operated on benchmark data and random data. Figure 8.6a shows that the multipliers are clustered into tight groups forming an upward curved trend in correctness as delay increases. Of all the exact multipliers marked with a ($\star$), the $(3;2)$ multiplier is the fastest.

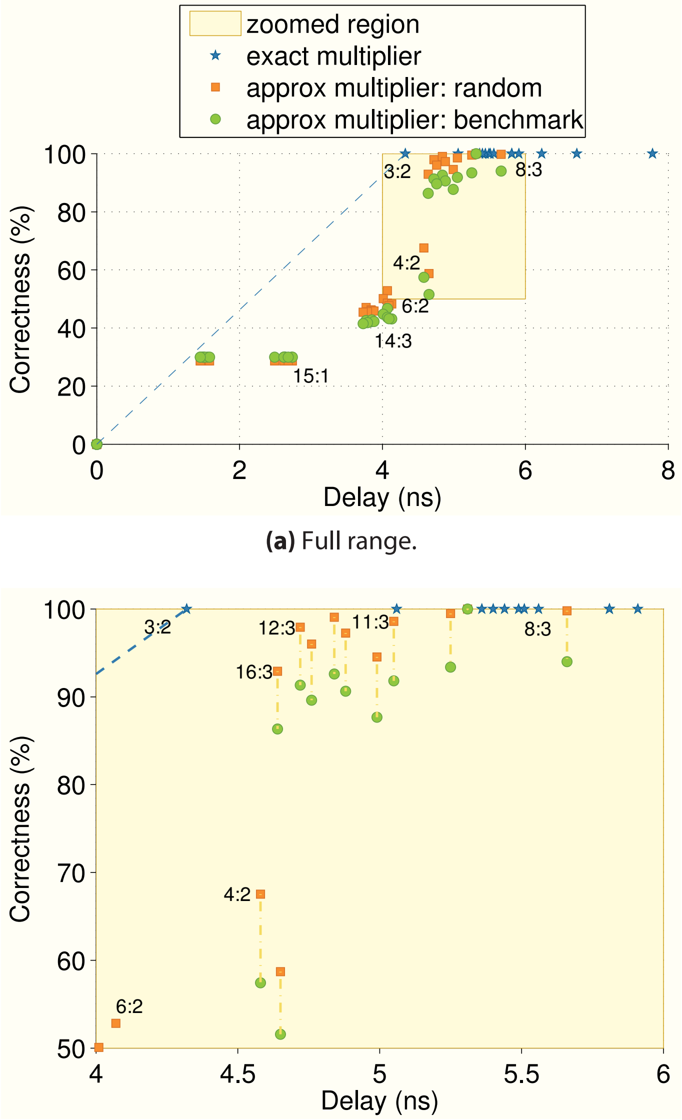
The dashed blue line shows linear trade-off of delay for correctness compared to the exact $(3;2)$ multiplier. The scatter plot shows that there are no approximate multipliers that have a better than linear improvement to correctness, and that the only approximate multipliers faster than the exact $(3;2)$ multiplier have an unsatisfactory correctness for *ADVS*. A cluster near 50,% correctness is much lower than the required target of 95 %.

It was intended that a set of suitable multipliers would be selected from the candidates above, and synthesised in floating point units with pipelining registers. This did not eventuate because the latency of the approximate multipliers was too high compared to the exact multiplier. Although these multipliers were synthesised in isolation, it is unlikely that full synthesis of the approximate multipliers would be faster than the $(3;2)$ multiplier by at least one clock with variations in drive strength, capacitive load, and pipelining depth.

Approximate floating point multipliers were not used in simulations of an *ADVS*-enabled system.

8.4 Conclusion

In this chapter the feasibility of using approximate floating point units in *ADVS* was investigated. The significand units lie on the critical path and are more amenable to design trade-offs that affect both latency and correctness. Previous designs of approximate integer units were used as components of floating point adders, multipliers and dividers.



(a) Full range.

(b) Zoomed region.

Figure 8.6: Latency vs. correctness for the multiplication of significands from floating point multiplication operations in benchmarks.

Simulations using benchmark data showed that the average probability of correctness of the divider was too low for use in an *ADVS* enabled system. The average worst case carry length of the approximate adder was shown to be too long for the required correctness target, resulting in a long latency design. Several configurations of approximate tree multipliers had an appropriate correctness, but their latency was too high for use in value speculation.

Based on these results approximate floating point units were not used in a system simulation of *ADVS*, presented in Chapter 11.

PART III

APPLICATION

Chapter 9

Result Caching

"What you save is, later, like something found."

Yiddish Proverb

This chapter investigates the caching of arithmetic results to further reduce the average effective latency of arithmetic operations, including operations where no feasible approximate hardware exists. Indexing schemes are developed to better distribute entries within the cache. Replacement policies from the literature are used to introduce set associativity, and are shown to improve the hit rates over direct mapped result caches.

Chapter 9: Result Caching

Result caching was proposed by Richardson to decrease the latency of multi-cycle arithmetic operations by storing previously calculated arithmetic results in a direct-mapped result cache [Richardson, 1992]. Result caches with hit rates of over 50 % were demonstrated, due to the value locality of many programs.

Despite their apparent promise, result caches remain uncommon in modern processors. Branch predictors, on the other hand, have been successful in improving throughput by speculating on the control path of programs, and have increased in sophistication from static prediction to schemes addressing indirect branches.

This chapter investigates several result caching schemes, and their effect on throughput. Result caching, rather than prediction, was investigated because result caching does not incur a misprediction penalty. Caching was restricted to long latency arithmetic operations; single cycle operations are inexpensive in terms of delay, and load/store operations can cause cache thrashing. Indexing schemes based on operand values, rather than PC or architectural register names were used because it was assumed that result caches indexed by value would be more likely to be hit by another process after a context switch.

Simple arithmetic result caches were extended in three areas: benchmark operands were inspected to find a cache indexing scheme that could improve utilisation; cache set associativity was introduced to improve hit rate; and several replacement algorithms were implemented for direct mapped and associative arithmetic result caches. Operand caches were successively refined to maximise the average hit rates, and were then used in simulations of benchmark programs to determine the effect on throughput. Finally, timing information was obtained for small operand caches, and matched to the clock period of arithmetic units developed for an *ADVS*-enabled system.

Other than improving the latency of multi-cycle arithmetic instructions, operand caching provides a mechanism in *ADVS* to store the correct outcome of operations that are incorrectly approximated and repeated, saving possible frequent pipeline flushes. It is shown that increasing cache associativity improves hit rates, and improves the throughput of benchmark programs.

9.1 Caching techniques

This section introduces the basic properties of caches and summarises alternative caching schemes, and alternative methods of value prediction. Latter sections of this chapter extend Richardson's research to improve cache hit rate and demonstrate performance improvement in terms of retired instructions-per-clock (IPC) for benchmark programs.

9.1.1 Result caching schemes

Result caches were originally described as single-entity, and were direct-mapped [Richardson, 1992]. They were later extended to include distinct caches for individual operations [Oberman and Flynn, 1995, 1996]. For example, separate caches could be maintained for division and square root results. Caches for intermediate results were also described. For example, a divider that calculates ${}^{a}/_{b}$ as $a \times ({}^{1}/_{b})$ could benefit by caching the intermediate reciprocal ${}^{1}/_{b}$ in the case when the divisor b is frequently reused. A benefit of caching unary operations such as reciprocal and square root is that they require less storage.

Caches for the results of function calls appeared in [Huang and Lilja, 2000]. The inputs and outputs of functions were stored in memory, in structures called 'memos'. This caching technique was called 'memoisation'. An analysis of function calls showed the average resources required by 85 % of functions was four input registers, four output registers, three memory inputs and two memory outputs. The potential gain in throughput by bypassing entire function calls is large, but storage and memory accesses are required for each function input and output. Furthermore, result caching of arbitrary sections of code requires profiling to identify repeated sections of code, and value tracing to verify that significant value recurrence does occur [Richardson, 1992]. Functions must be necessarily called by value, because the target of pointers are indeterminate.

The approach in [Cheng and Hsiao, 2005] did not limit memoisation to functions, but instead to repeated small sections of code. Furthermore, the result cache was treated as hit if the input values were merely close to the cached values within a certain delta threshold. A proof-of-concept demonstrated a region-level approximate computation buffer that cached the results of a Inverse Discrete Cosine Transform (ICDT) used for MPEG2 decoding. A 70 % reduction in execution time of the IDCT code was observed.

9.1.1.1 Extended result caching

Caching schemes indexed by operand values, architectural register names, and architectural registers with dependence chains were shown to reuse over 20 % of instructions with a 1024 entry buffer [Sodani and Sohi, 1998]. Instruction caches can be located early in processor pipelines, such as the instruction decode stage, if the indexing scheme depends only on information such as the PC and architectural register names, and not on operand values. Hence, even single cycle operations can be bypassed, and pipeline resources are not occupied if the instruction does not issue.

Instruction reuse schemes were shown to perform as well as value prediction schemes because reuse schemes do not suffer misprediction penalties. Furthermore, overall throughput for value prediction schemes were sensitive to the branch resolution scheme. Branch mispredictions could be reduced by not resolving branches until after the branch operands were non-speculative [Sodani and Sohi, 1998].

Caches indexed by PC, called reuse buffers, were used to read their operands earlier than value-indexed caches. Multiple instructions could be reused when fetched simultaneously. The main drawback of the reuse buffer was that an identical operation at other addresses could not benefit from the cached results [Molina et al., 1999].

9.1.2 Result prediction schemes

Value prediction schemes attempt to predict the data value of *dynamic instructions*. These are value producing instructions and include arithmetic and logic operations, conditional results and load targets. Speculative load targets (in result caches indexed by PC) are desirable to mask load latency, particularly when a traditional cache miss occurs, because the load can be initiated early [Lipasti and Shen, 1998; Lipasti, 1998].

It was shown that an average accuracy of over 50 % was attainable by simple *last-value* and *constant-stride* value predictor schemes [Gabbay, 1996]. The most common correctly predicted instructions were integer additions and load instructions.

Dynamic instructions were also found to be predictable on different execution paths. After a branch, instructions on the incorrect speculative path could be identical, and yield the same values as instructions on the correct path. Additionally, repeated function calls with different inputs share many common control-invariant instructions [Sodani and Sohi, 1997].

Context-based predictors predict the next value in a series, and were shown to be capable of higher

Table 9.1: An example history table for a context-based predictor, after a result stream '$a\,c\,a\,a\,a\,a\,a\,b\,a\,a$?'. The pattern '$aa$' before the unknown next value '?' determines that the first row of the history table is used. The element with the highest counter is column 'a', which is selected as the predicted value.

Pattern	Next Value		
	a	***b***	***c***
aa	3	1	0
ab	1	0	0
ac	1	0	0
ba	1	0	0
⋮			
cc	0	0	0

prediction accuracy than *last value*- and *stride*- based predictors in *SPEC* benchmarks [Sazeides and Smith, 1997, 1999]. Context based predictors track the occurrence of certain value patterns, where the length of the pattern is termed the *order* of the pattern. For example an instruction could be observed to output values a, b, and$c$. A complete 2^{nd} order model would record the observed next value in a history table for all patterns aa, ab, ac, $\ldots cc$. A counter for each next value records the occurrence of each of a, b and c as the next value in the sequence. To make a prediction, the history table is consulted for the next value that was observed in the past. An example is shown in Table 9.1.

A context based predictor requires cyclic value patterns to make correct value predictions. It also requires a training period to observe the initial patterns. Context based predictors work poorly on regularly incremented data, such as loop variables. Hybrid predictor models were proposed in [Sazeides and Smith, 1999] to obviate the vulnerability to either cyclic or incremental patterns.

Common arithmetic instructions in *SPEC* and *Mediabench* typically operate on narrow operands, often less than a 64 bits. As a power saving mechanism, it was proposed that a processor use clock gating to turn off portions of the ALU that are unused by small operands [Brooks and Martonosi, 2000]. Additionally, a dynamic scheme to vectorise a series of instructions could yield performance gains without the need to recompile code.

9.2 Results

Caches can be hit more frequently when they are fully utilised. This requires that the stored data is indexed so that there is the least conflict amongst entries. In the following section an analysis of the asserted bits in operands traced from benchmark programs is used to find an indexing scheme. The arithmetic operands do not have a uniform random distribution. Instead, the bit assertion pattern is used to determine a hashing scheme to index an operand cache. Later, a comparison of indexing schemes and cache replacement policies is made. Finally some result caches are used in an out-of-order simulation to measure the impact on throughput.

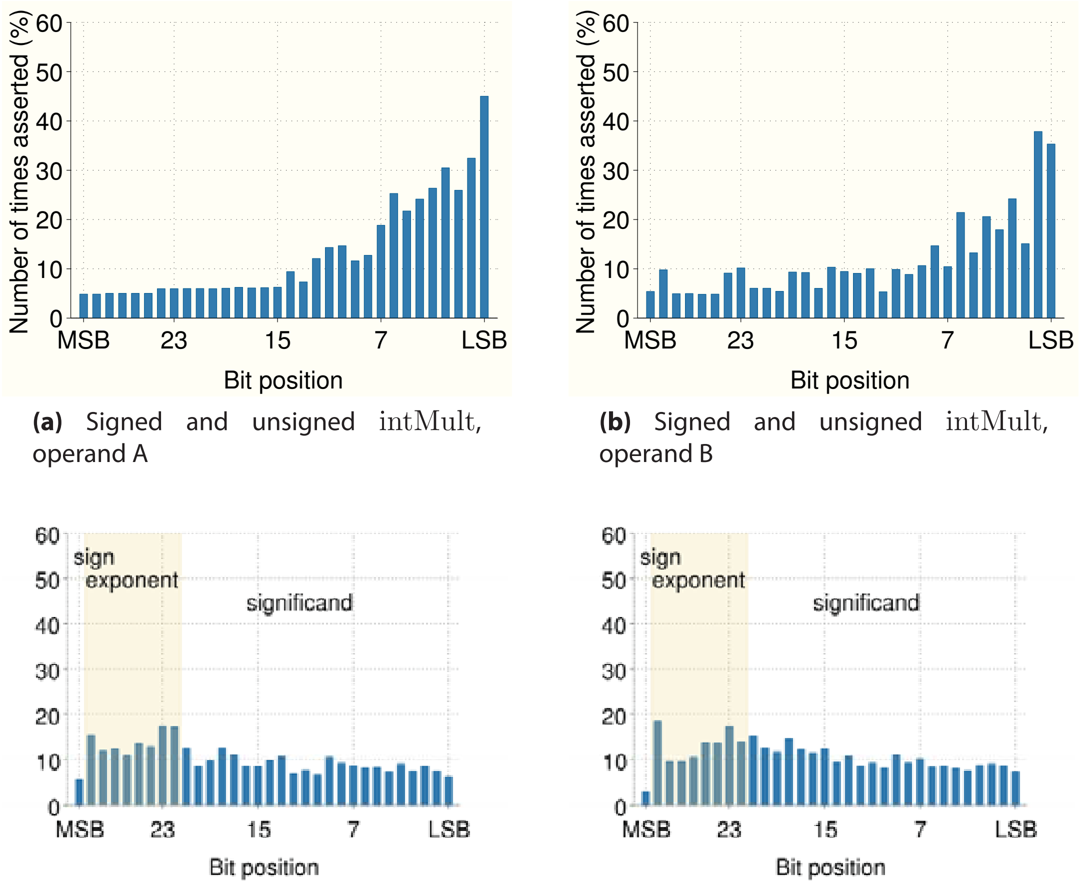
9.2.1 Operand Bit Assertion

Figure 9.1 shows the average frequency of assertion of the operand bits in `intMult` and `fpDiv` operations. The following indexing schemes were considered to determine which line entries were mapped to:

- Schemes labelled with a suffix `-A` were indexed with the least significant bits of operand A. Those with `-B` were indexed with the lest significant bits of operand B. Schemes labelled with suffix`-C` were hybrid schemes using a concatenation of the lower bits of operand A and B.
- Labels `-XA` and `-XB` indicate that the indexing scheme used the operand bits in the order found in Table 9.2— the bits most likely to be asserted near 50 % of the time.
- The cache schemes without a suffix did not use additional information to determine the cache index—the entire cache was treated as a circular buffer.

In the simulations performed, signed and unsigned integer operands were assigned to the same operation cache to save hardware. The number of signed operations usually outnumbered unsigned integer operations. Also, single and double precision floating point operands of the same operation were assigned to the same cache. Caches were maintained for combined signed and unsigned `intMult` and `intDiv`, and combined single- and double-precision `fpMult`, `fpDiv` and `fpSqrt`.

Bit assertion histograms are shown for `intMult` and `fpDiv` operations in Figure 9.1, and both are typical of observed integer and floating point operands, respectively. In general, the lower bits of integer operands were more commonly asserted due to the average magnitude of the integer operands, and because there were few negative signed operands. For floating point, the exponent



(a) Signed and unsigned intMult, operand A

(b) Signed and unsigned intMult, operand B

(c) fpDiv, operand A

(d) fpDiv, operand B

Figure 9.1: Bit assertion histograms for operands A and B for intmult and fpDiv.

bits were more commonly asserted than the significand bits. Table 9.2 shows the 16 bit positions that were asserted closest to 50 % of the time. The A operand tended to be asserted more frequently then the B operand in both integer and floating point arithmetic.

9.2.2 Cache replacement policies

When a cache is misses and is filled, the new data is stored in the cache for future re-use. When the cache is full a replacement policy is used to determine which old data is evicted to store the new data. Three well known policies were examined:

FIFO (first in, first out) Each element was stored in a circular buffer, and the oldest element was always evicted.

LRU (least recently used) The element that was accessed the longest time ago was

Table 9.2: An ordered list of bit assertion frequencies of arithmetic operands. The 16 bits asserted closest to 50 % of the time are shown. The sampled integers were represented in little endian format.

Operation	Operand	Operand bit (little endian)															
`intMult`	A	0	1	3	2	4	7	5	6	8	11	9	12	13	10	15	14
`intMult`	B	1	0	3	7	5	4	2	9	6	12	8	17	24	10	20	16
`intDiv`	A	0	1	5	3	2	12	4	7	14	13	15	6	16	8	9	17
`intDiv`	B	0	13	15	1	3	2	4	5	6	8	7	9	11	10	12	14
`fpAdd`	A	26	29	27	28	24	11	19	23	25	16	10	21	22	17	14	60
`fpAdd`	B	60	61	59	58	57	26	28	24	56	14	18	29	55	27	16	54
`fpSub`	A	53	55	57	56	52	54	59	60	61	58	62	26	24	23	22	28
`fpSub`	B	53	57	55	54	56	60	61	59	58	52	27	62	26	28	18	29
`fpMult`	A	26	29	27	28	24	23	55	25	18	10	15	19	60	61	59	58
`fpMult`	B	29	28	26	59	60	61	58	27	23	57	56	15	10	24	25	19
`fpDiv`	A	52	55	54	53	62	56	48	57	58	60	61	45	59	40	39	51
`fpDiv`	B	62	54	52	55	53	50	29	48	26	51	49	39	45	24	44	43

evicted.

pLRU (pseudo-*LRU*) Each element was evicted based on the state of a decision tree, and every access changed the state. This is simpler to implement than *LRU*, although pathological cases can significantly degrade performance compared to *LRU*.

9.2.3 Simulation Results

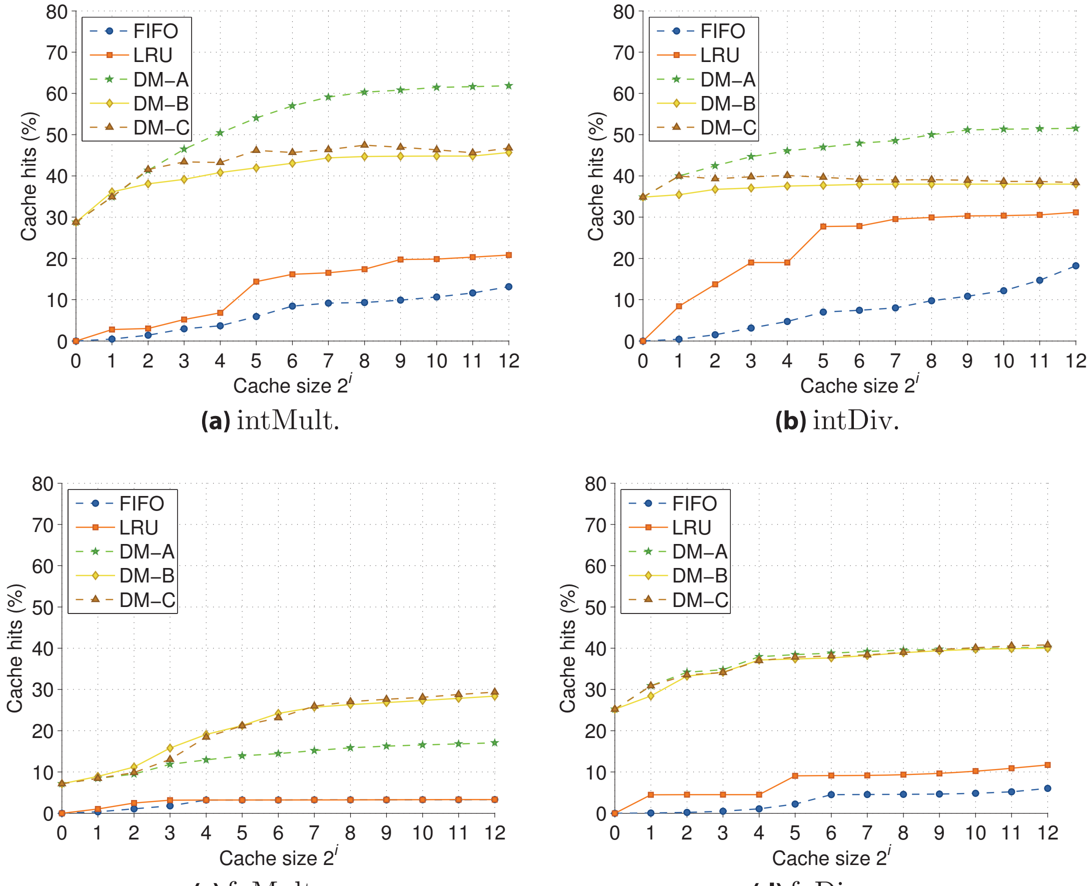
Figure 9.2 shows the hit rate of simple direct mapped caches for tested arithmetic operations. Each operation was indexed into the result cache with different schemes. The arithmetic operations all had similar hit-rate profiles, so only `intMult` and `intDiv` are shown. As expected from the bit-error histograms in Figure 9.1, indexing schemes based on operand A (`-A`) performed slightly better than schemes indexed on operand B (`-B`). Also, the dominance of the direct mapped schemes over the linear replacement schemes (*LRU* and *FIFO*) show the advantage of exploiting value locality rather than temporal locality in result caches.

Figures 9.3 and 9.4 show result caches with increased associativity. *LRU* and *FIFO* are used as replacement algorithms. The A operand was shown to be more frequently asserted, and `-A` indexing yielded a higher hit rate for simple caches. The `-A` and `-XA` indexing schemes were used in the higher associativity caches. As expected, higher cache associativity increased the hit rates in both cases, but performance gain diminished with associativity.

Figure 9.5 shows the hit rates using various associativities, compared to direct mapped caches. Each associative cache used 4 ways as a tradeoff between complexity and performance. Both *LRU* and *FIFO* replacement schemes showed an increased hit rate over the direct mapped caches. The `-XA` and `-XB` indexing schemes did not yield a significant improvement in hit rates, because `-XA` and `-XB` hashed similar operand bits to the `-A` and `-B` indexing schemes. The `-XA` and `-XB` schemes were based on average bit assertions; performance was not improved using these schemes because the assertion averages failed to capture short-term trends in value patterns.

9.2.4 Implementation

Using the results of Section 9.2.3 operand caching was implemented in simulations of an out-of-order *MIPS* pipeline using *SimpleScalar*. The arithmetic operations `intMult`, `intDiv`, `fpMult`



(a) intMult.

(b) intDiv.

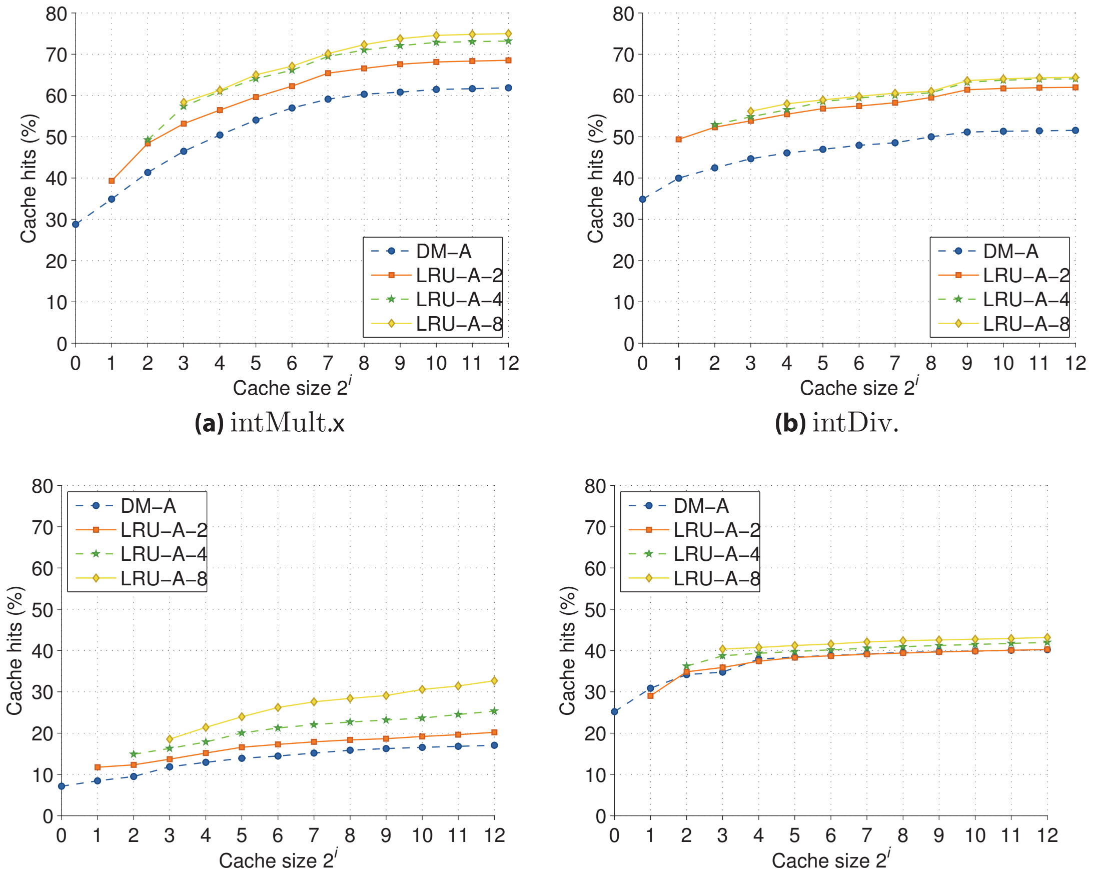
(c) fpMult.

(d) fpDiv.

Figure 9.2: Hit rate of direct-mapped result caches. Each operation was tested with a private cache sized from 1 to 4 k entries.

and `fpDiv` were assigned private caches, each with 64 entries, 4 way set associative, `-A` indexing, and various replacement algorithms. Gains to IPC are shown in Figure 9.6.

Similar trends to hit rates and throughput were observed compared to simple direct mapped result caches [Richardson, 1992]. Increasing the total size of the result caches yielded a higher but diminishing increase to hit rates. Richardson measured the total number of cycles eliminated through caching from three hypothetical processors. The processors were only defined in terms of their arithmetic latencies, with short, medium and very long (> 100) latencies. In this chapter performance was shown in terms of IPC gain for a simulated *MIPS*-like processor. The throughput trends are similar—throughput increased almost continually, and did not develop a knee. The maximum throughput increase was over 7 % using the 4 way *FIFO* cache with caches indexed with the `-A` scheme. Figure 9.6 shows that the set associative implementations maintain an almost constant 1.5 % IPC improvement over the direct mapped cache.



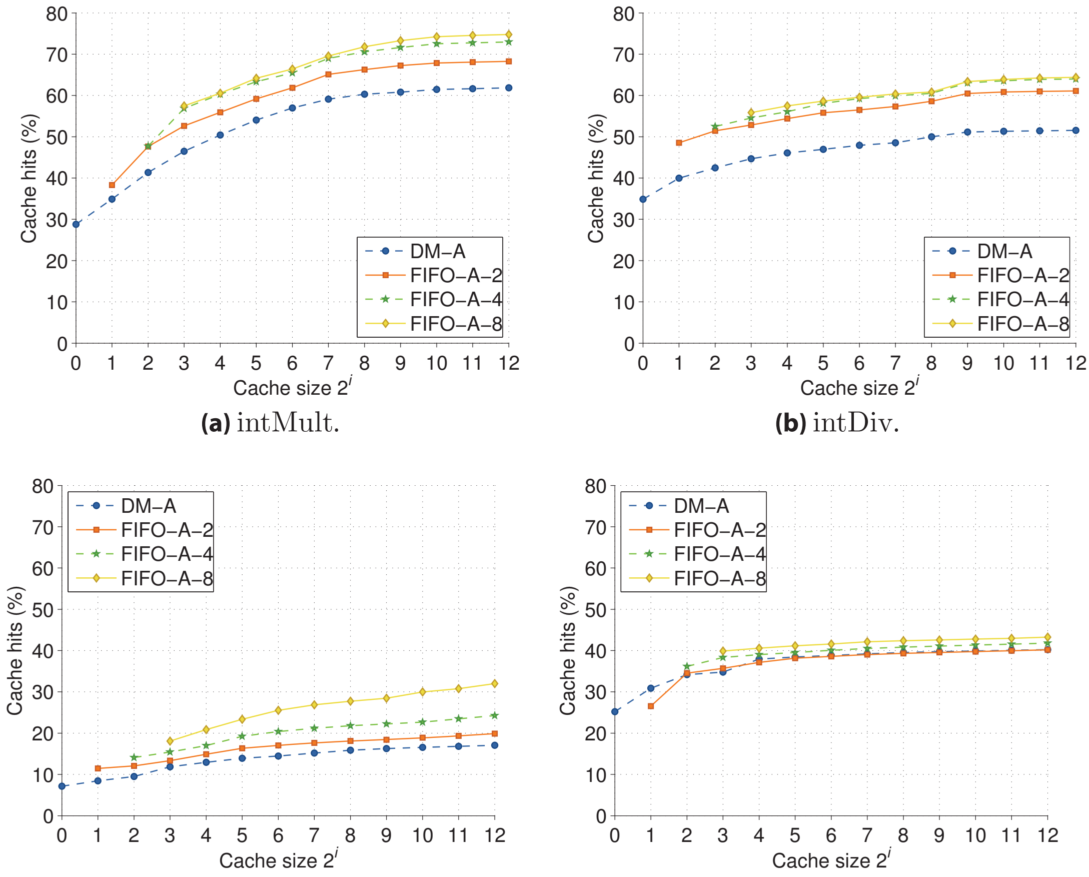
(a) intMult.x **(b)** intDiv. **(c)** fpMult. **(d)** fpDiv.

Figure 9.3: Hit rate of *LRU* caches. Each operation was tested with a private cache from 1 entry to 4 k entries, with different levels of associativity.

Although the average hit rates for the operand caches in Figure 9.5 started to level out at 4 k entries, the trend in IPC continued to increase in Figure 9.6. This effect was observed because distinct caches were maintained for each operation. Tables 2.2, C.1–C.3 and C.9–C.11 show that the more frequent operations like `intMult` and `fpAdd` have lower latency, while infrequent operations like `fpDiv` have higher latency. If all arithmetic operations shared the same cache, it is likely that the long latency infrequent operations would be evicted before being reused.

9.3 Result caches for ADVS

In this section two sets of caches are simulated with the Cache Access and Cycle Time Information (*CACTI*) simulators [HP labs, 2008].



(a) intMult.

(b) intDiv.

(c) fpMult.

(d) fpDiv.

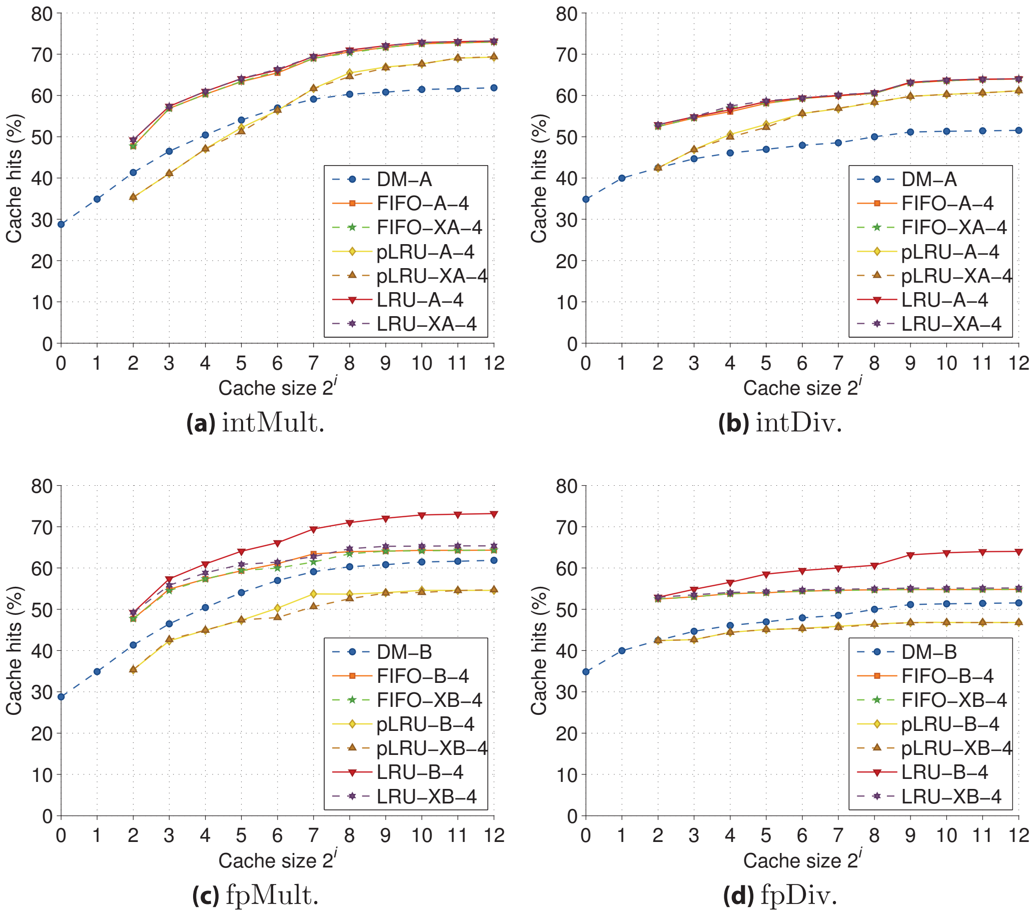
Figure 9.4: Hit rate of *FIFO* caches. Each operation was tested with a private cache from 1 entry to 4 k entries, with different levels of associativity.

9.3.1 65 nm process

A hypothetical example cache is simulated here in a modern 65 nm process, assuming aggressive parameters. The purpose of this section is to investigate the scalability of result caches to newer systems. *CACTI* 5.3 was required for this feature size. A small access time is desirable because the cache tags are directed from registers, and are forwarded to execution units in the next stage.

In this implementation the individual caches were implemented separately, allowing the possibility of independent access. Each individual cache is sized at 4 k entries, yielding a total of 368 kB. The cache sizing is shown in Table 9.3.

The total access time is shown in Table 9.4. In a short pipeline with few pipeline stages, such as used in *SimpleScalar*, we can assume that each stage will operate in approximately 20 FO4 delays. Also, assuming a 45 ps FO4 delay in a 65 nm process, the clock is set at 20×45 ps=0.9 ns, or 1.1 GHz.



(a) intMult. **(b)** intDiv. **(c)** fpMult. **(d)** fpDiv.

Figure 9.5: Hit rate for caches with different cache replacement schemes. Most caches are 4-way associative.

Comparing with Table 9.4, the proposed caches can operate in a single cycle, as required to reduce the cycle latency of arithmetic operations.

9.3.2 180 nm process

In this section the operand caches are simulated in a 180 nm technology, corresponding to the feature size of the arithmetic units synthesised in previous chapters. The total size of all caches in bytes was chosen to be approximately the same as the level 1 `iCache` and `dCache` (16 kB). This corresponds to 2^8=256 entries per operand cache, and an average IPC increase of approximately 6 %.

CACTI 4.1 was used to operate with the older 180 nm feature size. All caches were configured with one dedicated read and write port.

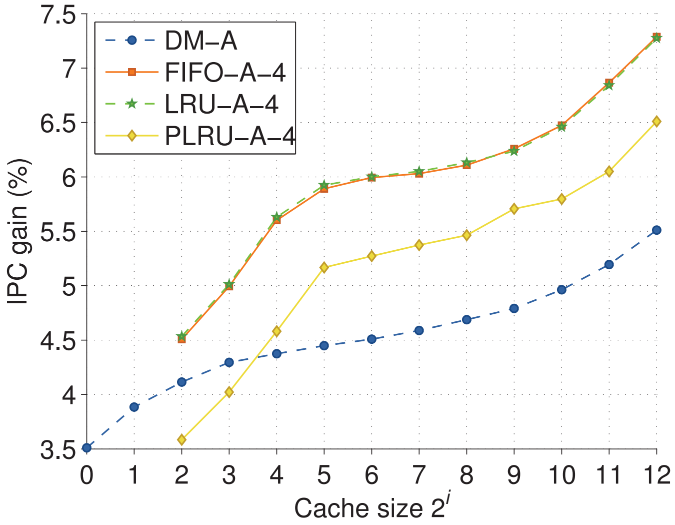


Figure 9.6: Average IPC gain for benchmarks run in *SimpleScalar* with various cache replacement schemes. Long latency integer and floating point multiplication, division, and square root were all cached separately and indexed with the -A scheme.

Table 9.3: Modelled cache access times in a 65 nm process using *CACTI* 5.3. Each cache has 4 k entries

Cache	Operand (B)	Result (B)	Size (kB)
`uintMult/intMult`	2×4	2×4	64
`uintDiv/intDiv`	2×4	1×4	48
`fpMult`	2×8	1×8	96
`fpDiv`	2×8	1×8	96
`fpSqrt`	1×8	1×8	64

Table 9.5 shows the results of simulation of the operand caches. All of the caches can be accessed in approximately 1 ns. Assuming that the 32 bit Sklansky adder in Section 3.2.1.10 sets the clock at 5 ns, it is possible that a result cache lookup could be performed in one machine cycle. To avoid issuing an instruction to an arithmetic unit, the operand cache lookup must occur in the same cycle as the register access, before execution. If timing is an issue, lookup can occur in parallel with execution, sacrificing part of the latency saving.

The operand caches simulated in this section have an appropriate data access time to be integrated into an *ADVS*-enabled system as a further enhancement, to mitigate the cases where repeated incorrectly approximated instructions would cause the system to degrade performance due to flushing.

Table 9.4: Modelled cache access times in a 65 nm process using *CACTI 5.3*.

Cache	Access time (ns)
64 kB	0.837
92 kB	0.887
48 kB	0.829

Table 9.5: 180 nm result caches process using *CACTI 4.1*.

Operation	Entry (B)	Ways	Lines	Total size (kB)	Access (ns)	Read Power (W)	Area (mm^2)
`intMult`	12	4	64	3	1.402	0.213	0.726
`intDiv`	12	4	64	3	1.402	0.213	0.726
`fpMult`	24	4	64	6	1.393	0.532	1.026
`fpDiv`	24	4	64	6	1.393	0.532	1.026
`fpSqrt`	16	4	64	2	1.401	0.215	0.649

9.4 Conclusion

This chapter confirmed that arithmetic result caching can yield significant gains to performance, as previously proposed in the literature. The experiments in this chapter incorporated associativity and different replacement strategies to simple result caches, and maintained hit rates of over 50 % due the repetitiveness of arithmetic operations. Average throughput gains from result caching were in excess of 5 %. It was also found that increasing set associativity and using an appropriate replacement algorithm improved throughput up to 1.5 % over direct mapped caches; and exploiting value locality can yield higher re-use over temporal locality for arithmetic results. The hardware required to implement associativity in addition to a that required for result caching is likely to be appealing for a 1.5 % throughput improvement.

Cache area and timing information was derived for sample operand caches in a modern technology, and in the same technology as synthesised arithmetic units. The data access time of the caches were found to be reasonable to be used in an *ADVS*-enabled system.

Further research could investigate the impact of context switches on cache hit rates. Entries indexed by value could contain operands and results used by other processes, where caches indexed by PC or architectural registers are less likely to.

When simulating the processor throughput using separate operand caches, they were all configured homogeneously, being indexed with the same schemes, employing the same replacement policies and maintaining with the same number of entries. The total efficiency of the caches could be improved by heterogeneous caching, where each cache is configured discretely.

Chapter 10

Approximate Adders in LDPC

"Parity is for farmers."

Seymour Cray (1925–1966),
when asked why he did not include memory parity in the CDC 6600.

"Farmers buy a lot of computers."

Seymour Cray (1925–1966),
when asked why he agreed to put parity error-correction in the Cray-1.

This chapter demonstrates approximate adders in an error tolerant application---low density parity check decoding, where it is found that approximate compressors can yield a saving to area, power and delay, as well as decreasing the frame error rate and average number of decoding iterations, compared to an implementation found in the literature.

APPROXIMATE arithmetic can be used in designs other than value speculation. With *ADVS* the system is critically dependent on the probability of correctness of the approximate result. However, approximate arithmetic can be used in other applications where low latency is important, and small errors can be tolerated. This chapter investigates low density parity check (*LDPC*) codes as a case study of an approximation error-tolerant design.

LDPC codes are used to encode and decode data for transmission through a noisy medium. The sender encodes the signal, and the receiver decodes the signal, with the intent of correcting transmission errors. *LDPC* correcting codes are robust against occasional approximation errors in their intermediate calculations, and occasional uncorrected errors are expected and tolerated.

The chapter investigates the use of approximate multioperand adders in the computation nodes of an *LDPC* decoder to reduce decoder latency.

10.1 Background

Digital processing techniques balance the precision maintained in number representations against the implementation performance and cost. This quantisation error can be considered a type of approximation, and other types of approximation are possible. For example, elementary functions can be approximated using addition and table look-up [Hassler and Takagi, 1995]. The goal of techniques such as this is to efficiently compute an approximate value with a well-behaved, bounded error.

The logical incompleteness technique used to develop approximate arithmetic hardware in Chapters 6–8 typically don't have a bounded error is because the intermediate calculations can drop bits at any significance.

Section 10.2 discusses the iterative algorithms used to in *LDPC* to converge upon the desired result, and their tolerance to occasional errors. Arithmetic approximations are commonly used in *LDPC* decoders, but using approximate multioperand adders in *LDPC* check nodes is a new idea. Approximate multioperand adders, such as first discussed in Section 6.1, are modified for use in *LDPC* check nodes in Section 10.3. The simulated operation of *LDPC* decoders using approximate check nodes is shown in Section 10.4. It was found that it is possible to reduce decoder latency, consumed power

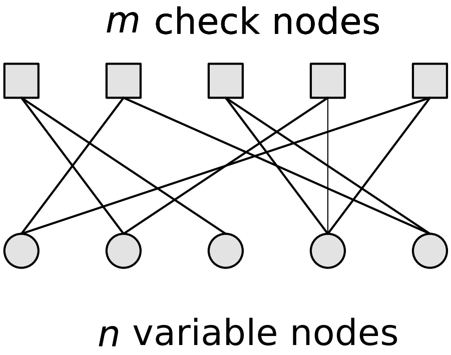


Figure 10.1: *LDPC* codes can be represented as a bipartite graph of check and variable nodes.

and silicon area with little increase in the decoder frame error rate.

10.2 Low density parity check codes

This section discusses *LDPC* decoding, and introduces an implementation from the literature that is used as a baseline model.

LDPC codes [Gallager, 1962] are linear block codes that are asymptotically superior to turbo codes with respect to coding gain [Richardson et al., 2001]. Each *LDPC* code is associated with an $m \times n$ sparse binary parity check matrix $\mathbb{H}$. An *LDPC* encoder concatenates m parity bits with a k bit information word to form an n bit codeword x such that $\mathbb{H} \times \mathrm{x}^{\mathrm{T}} = 0$.

An *LDPC* decoder can be represented by a bipartite graph as shown in Figure 10.1. Each row in $\mathbb{H}$ corresponds to a parity check node in the graph and each column corresponds to a variable node. Variable node j is associated with the j^{th} data symbol. A non-zero element $h_{i,j}$ in $\mathbb{H}$ indicates that data symbol j participates in parity check i—in the graph there is a connection between variable node j and check node i.

LDPC decoding is performed using an algorithm known as *belief propagation*. The variable nodes are initialised according to the received data symbols. At the start of an iteration, each variable node v_j sends a message $Q_{i,j}$ to each of its connected check nodes c_j. $Q_{i,j}$ represents v_j's 'belief' of being in a particular state given inputs from all connected check nodes except c_j. Each of the check nodes c_i then computes messages $R_{i,j}$ which are sent to the connected variable nodes v_j. $R_{i,j}$

is a 'reliability message' that indicates the probability of parity check i being satisfied given that v_j is in a particular state and taking into account the beliefs of the connected variable nodes other than v_j [E. Zimmermann and Fettweis, 2004].

There are many variations to this scheme: soft- and hard-decision versions of the decoder exist; the check matrix can be regular (constant column and row sums) or irregular; the nodes can be updated in the lock-step fashion indicated above (called flooding) or a different schedule can be used [Lestable and Zimmermann, 2005]. Messages can be represented in the log-likelihood domain to replace multiplications in the computation nodes with additions, but the check nodes then require the nonlinear *'box plus'* function. This can be approximated using the *'MinSum'* algorithm [Lestable and Zimmermann, 2005].

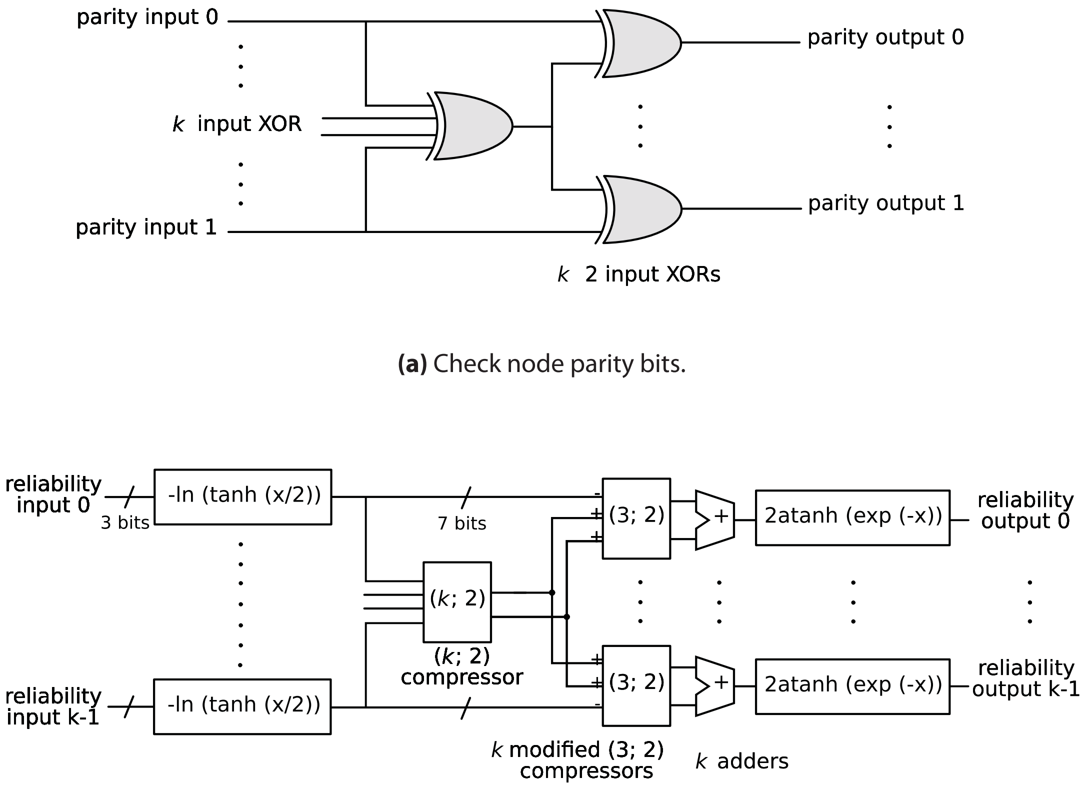
A benchmark implementation based on the *LDPC* code from *IEEE-802.16e* [Institute of Electrical and Electronics Engineers, 2006] and the decoder architecture of Blanksby & Howland [Howland and Blanksby, 2001] was adopted. This was a parallel, flooding, soft-decision decoder with 4 bit sign-magnitude reliability and belief messages in the log-likelihood domain. Decoding was terminated after 64 iterations at most. The 1056 bit, rate $^1/_2$ irregular code from *802.16e* was also used. All check nodes in this code have either 6 or 7 inputs.

Figure 10.2 shows the architecture of a check node [Howland and Blanksby, 2001]. Each 4 bit reliability message was split into a 3 bit magnitude and a 1 bit parity (sign) that were handled separately. Given the very short word-lengths, the hyperbolic trigonometric functions were merged with adjacent logarithmic and exponentiation functions and implemented efficiently using random logic. The $(3;2)$ compressors were modified to subtract one of the inputs using 2's complement arithmetic. This was achieved by inverting one of the inputs and injecting +1 into one of the otherwise vacant positions at the least significant end of the compressor. The result must therefore be greater than or equal to zero, simplifying the sign logic.

10.3 Approximate multioperand adders

This section revisits multioperand adders, and introduces new logically incomplete counters and compressors that are used in the check nodes of the *LDPC* decoders.

Multioperand adders are conventionally constructed using trees of binary counters, the most fundamental of which are the $(2;2)$ counter (half adder) and the $(3;2)$ counter (full adder) [Parhami,



(a) Check node parity bits.

(b) Check node reliability bits.

Figure 10.2: Architecture of a k input check node.

Table 10.1: Truth tables for saturating and reflecting $(4;2)$ counters, calculating the carry and sum bits cs for the sum $w + x + y + z$.

		wx			
		00	**01**	**11**	**10**
yz	**00**	00	01	10	01
	01	01	10	11	10
	11	10	11	11	11
	10	01	10	11	10

(a) Saturating (4; 2) counters.

		wx			
		00	**01**	**11**	**10**
yz	**00**	00	01	10	01
	01	01	10	11	10
	11	10	11	10	11
	10	01	10	11	10

(b) Reflecting (4; 2) counters.

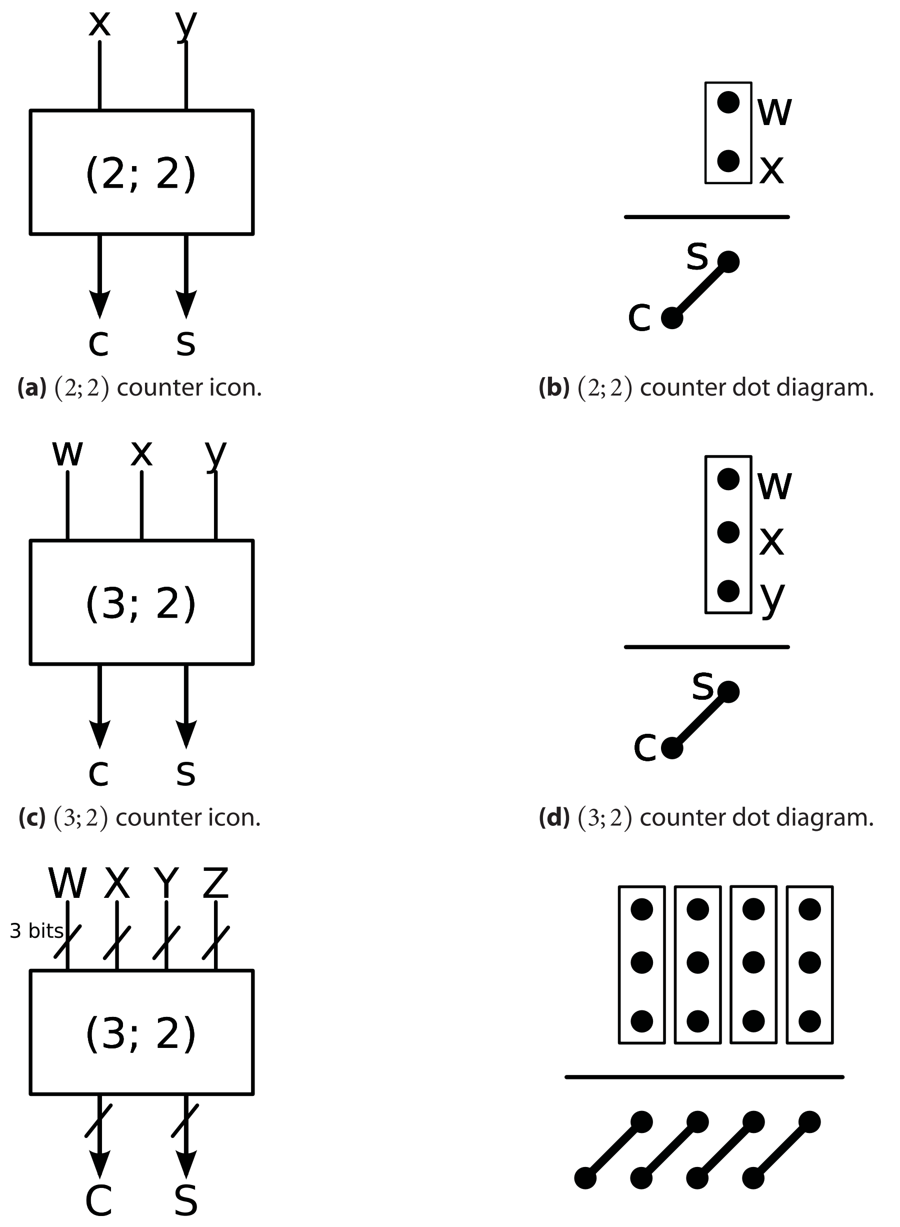
2000]. These are shown as schematic icons and as dot diagrams in Figure 10.3. They take 2 or 3 input bits respectively and computed their 2 bit sum. As shown in Figure 10.3, a row of $(3;2)$ counters can be used to make a $(3;2)$ compressor (carry save adder), a circuit that takes 3 input words and produces 2 output words with the same sum as the inputs. Larger compressors can be built from trees of $(3;2)$ compressors. For example, 4 bit $(6;2)$ and $(7;2)$ compressors are shown in Figures 10.4 and 10.5. Given an $(n;2)$ compressor, an n input multioperand adder can be built by including a carry propagate adder to find the sum of the outputs of the compressor.

The logic for two alternative $(4;2)$ approximate counters is shown in Table 10.1. Both counters produce exact outputs for every input case except $wxyz = 1111$. The first counter, a saturating $(4;2)$ counter, gives the sum $1 + 1 + 1 + 1 = 11_2 = 3_{10}$. The second counter, a reflecting $(4;2)$ counter, approximates this sum more crudely as $1 + 1 + 1 + 1 = 10_2 = 2_{10}$.

These $(4;2)$ approximate counters can be used to build approximate compressors with fewer stages of counters than their exact counterparts. To investigate the relative performance of $(3;2)$ and approximate $(4;2)$ counters, logic circuits were designed and simulated.

Logic synthesis using *Synopsys Design Compiler* and targeting the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library was used to derive schematic circuits for a fast $(3;2)$ counter and a fast $(4;2)$ saturating counter. These are shown in Figures 10.6 and 10.7.

These circuits were then simulated using *HSPICE* using transistor level netlists carefully designed to match the *Artisan* gates as closely as possible. Exhaustive simulations were performed to determine the worst case propagation delays. The gates under test were driven through 2 serial input-shaping inverters and loaded by 2 serial instances of the gate under test. The results are shown in Table 10.2.



(a) $(2;2)$ counter icon.

(b) $(2;2)$ counter dot diagram.

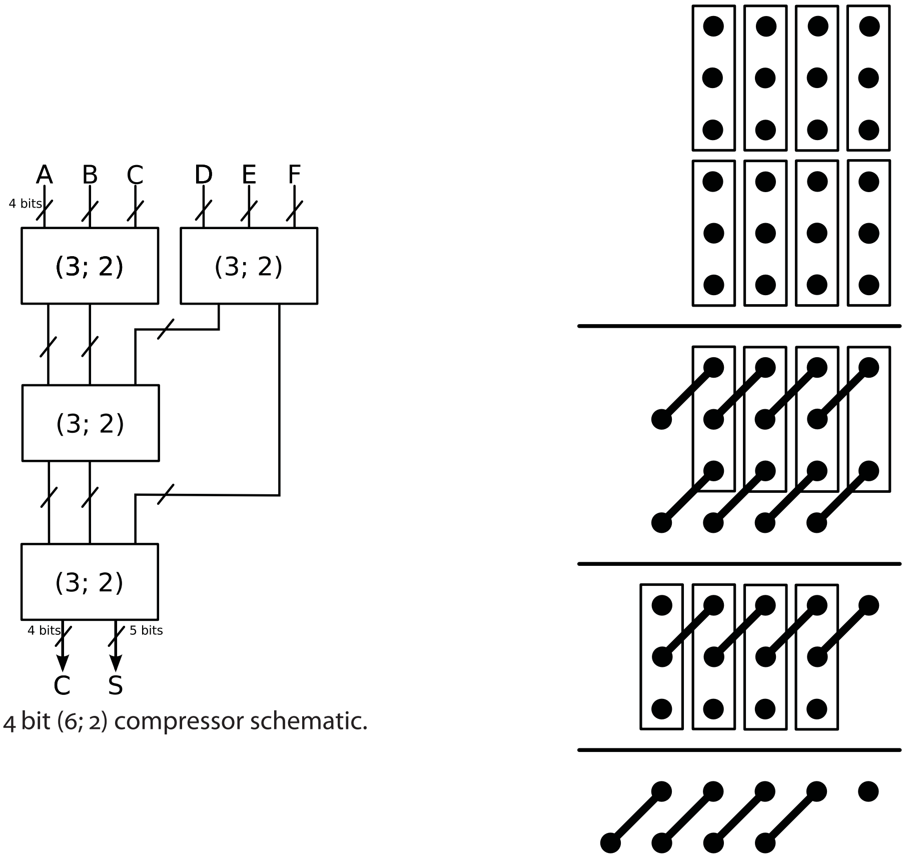
(c) $(3;2)$ counter icon.

(d) $(3;2)$ counter dot diagram.

(e) (4; 2) compressor icon.

(f) $(4;2)$ compressor dot diagram.

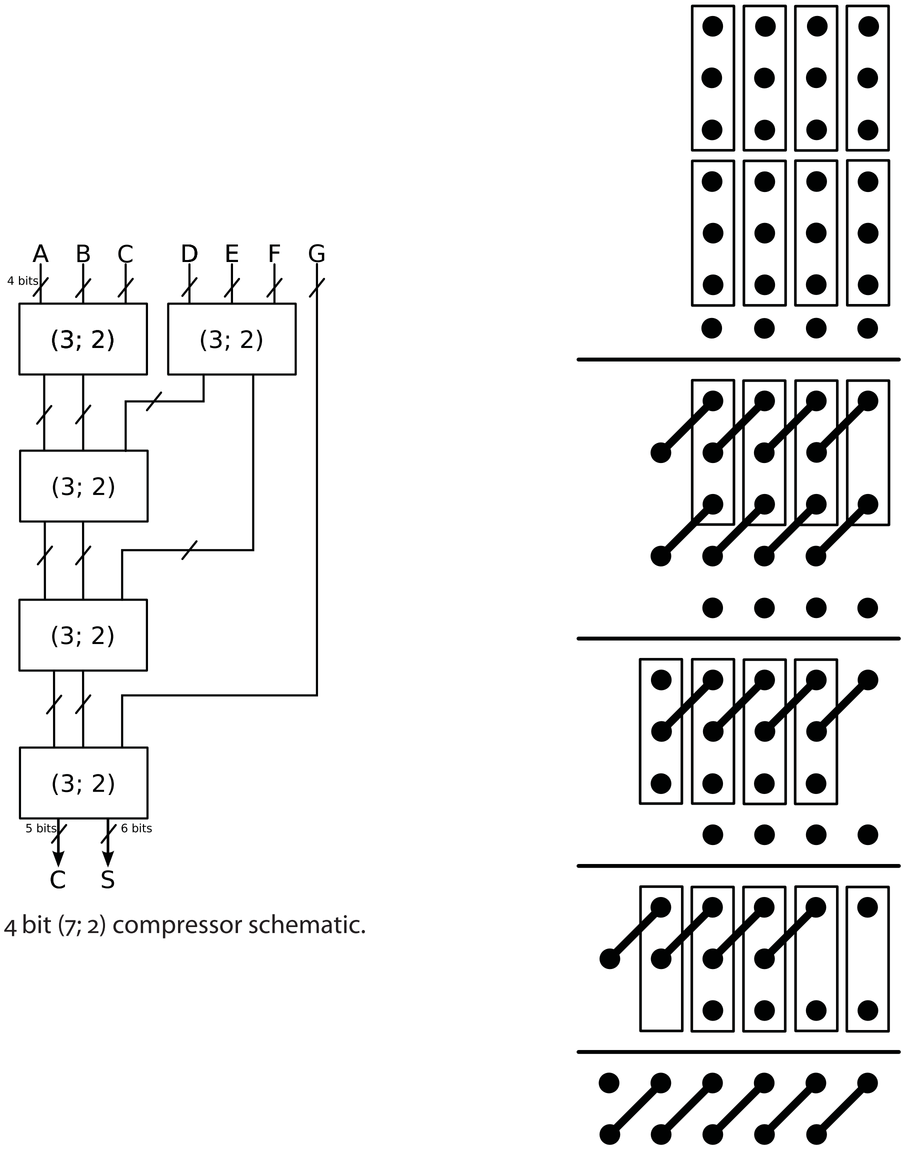
Figure 10.3: (2; 2) and (3; 2) counters and a (3; 2) compressor.



(a) 4 bit (6; 2) compressor schematic.

(b) 4 bit (6; 2) compressor dot diagram.

Figure 10.4: Exact $(6;2)$ compressors using (3; 2) counters.



(a) 4 bit $(7;2)$ compressor schematic.

(b) 4 bit $(7;2)$ compressor dot diagram.

Figure 10.5: Exact $(7;2)$ compressors using $(3;2)$ counters.

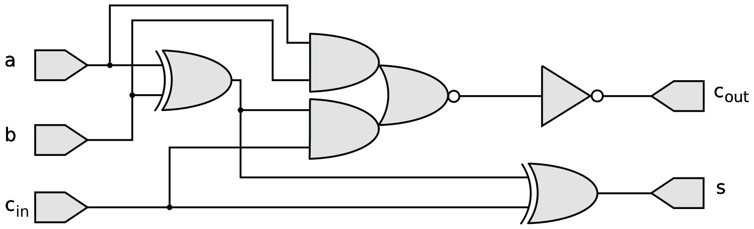


Figure 10.6: Schematic view of a $(3;2)$ counter.

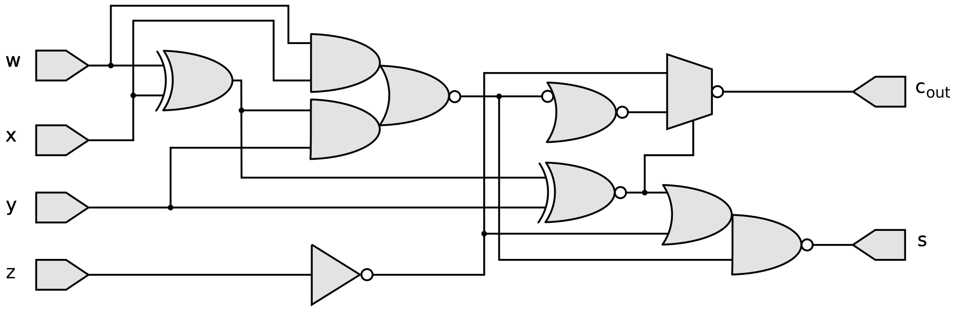


Figure 10.7: Schematic view of a $(4;2)$ saturating counter.

Table 10.2: HSPICE timing simulation results for an exact $(3;2)$ counter and $(4;2)$ saturating counter.

Input		Delay (ps)	
a	→	$sum\uparrow$	256.9
a	→	$sum\downarrow$	243.6
b	→	$sum\uparrow$	239.0
b	→	$sum\downarrow$	233.9
c_{in}	→	$sum\uparrow$	135.9
c_{in}	→	$sum\downarrow$	121.6
$\boldsymbol{a}$	**→**	$\boldsymbol{c_{out}\uparrow}$	**270.0**
a	→	$c_{out}\downarrow$	245.6
b	→	$c_{out}\uparrow$	263.5
b	→	$c_{out}\downarrow$	235.6
c_{in}	→	$c_{out}\uparrow$	126.9
c_{in}	→	$c_{out}\downarrow$	157.0

(a) Exact $(3;2)$ counter.

Input		Delay (ps)	
w	→	$sum\uparrow$	364.4
w	→	$sum\downarrow$	373.3
$\boldsymbol{x}$	**→**	$\boldsymbol{sum\uparrow}$	**374.4**
x	→	$sum\downarrow$	356.6
y	→	$sum\uparrow$	230.4
y	→	$sum\downarrow$	200.0
z	→	$sum\uparrow$	130.3
z	→	$sum\downarrow$	103.3
w	→	$c_{\text{out}}\uparrow$	341.4
w	→	$c_{\text{out}}\downarrow$	288.4
x	→	$c_{\text{out}}\uparrow$	286.1
x	→	$c_{\text{out}}\downarrow$	309.6
y	→	$c_{\text{out}}\uparrow$	178.8
y	→	$c_{\text{out}}\downarrow$	177.1
z	→	$c_{\text{out}}\uparrow$	119.4
z	→	$c_{\text{out}}\downarrow$	94.0

(b) Approximate $(4;2)$ saturating counter.

Figure 10.8 shows 4 bit $(6; 2)$ and $(7; 2)$ approximate compressors based on the $(4; 2)$ approximate counters. As expected, they require fewer stages of counters than the exact compressors of Figure 10.4. For example, the $(6; 2)$ approximate compressor require a row of $(3; 2)$ counters and a row of $(4; 2)$ approximate counters; a $(6; 2)$ exact compressor require 3 rows of $(3; 2)$ counters. The worst-case delays from Table 10.2 give delays of $3 \times 270.0 = 810$ ps for the exact compressor and $270.0 + 374.4 = 644.4$ ps for the approximate compressor. Circuit synthesis, discussed in the next section, confirms that the approximate compressors can be made faster than their exact counterparts.

The correctness of the approximate compressors was determined by simulation using *VHDL* models. The frequency of errors for 7 bit versions of the $(6; 2)$ and $(7; 2)$ approximate compressors was calculated empirically using 1,000,000 uniformly distributed random inputs in the range $[0, 27)$ for each compressor. The results are shown in Figure 10.9.

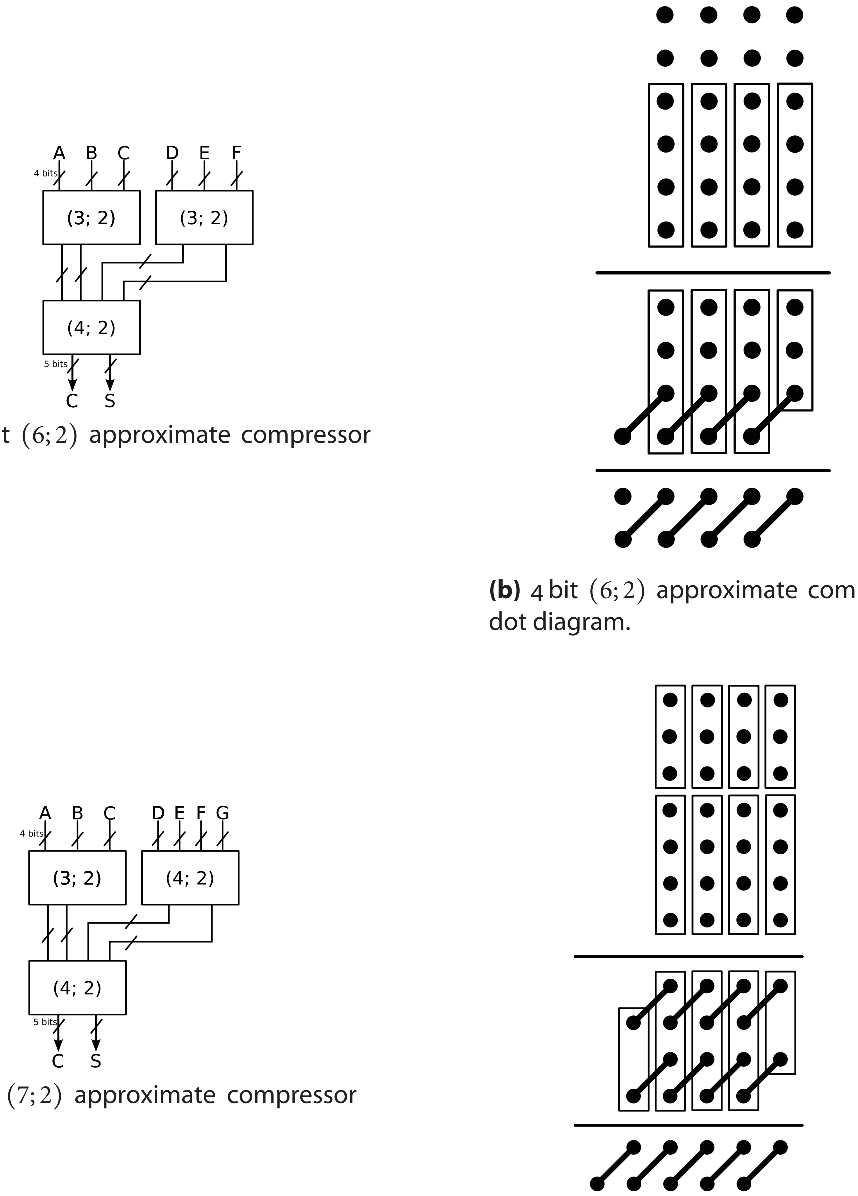
10.4 LDPC using approximate multioperand adders

In this section the effect of replacing the 7 bit $(6; 2)$ and $(7; 2)$ compressors in the check node architecture of Figure 10.2 with the approximate compressors of Figure 10.8 is presented. The resulting implementations are compared on the basis of latency, power, and area.

10.4.1 LDPC check node synthesis

Logic synthesis was used to compare the different check node architectures in hardware. *Synopsys Design Compiler* was used with the TSMC Artisan 0.18 μm process 1.8 V SAGE-X™ standard cell library to synthesise logic for the check nodes from the input of the $(7; 2)$ (or $(6; 2)$) compressor to the outputs of the $(3; 2)$ compressors. For the experiment the inputs were given the drive of a unit-size *DFF* as if they followed a pipeline stage. Outputs were loaded as if they were driving one input of a fast full adder. The synthesiser was given a timing constraint tighter than it could meet and no area constraint. Typical environmental conditions and wire load models were used.

Results reported by the synthesiser are shown in Table 10.3. The check nodes with approximate compressors achieved lower delay, area and power than their exact counterparts. In the case of the 7 input check node, the sum circuit with an approximate compressor built from reflecting $(4; 2)$ counters improved delay by 23 %, area by 8 %, and power by 11 % compared to the sum circuit using



(a) 4 bit $(6;2)$ approximate compressor icon.

(b) 4 bit $(6;2)$ approximate compressor dot diagram.

(c) 4 bit $(7;2)$ approximate compressor icon.

(d) 4 bit $(7;2)$ approximate compressor dot diagram.

Figure 10.8: 4 bit approximate compressors.

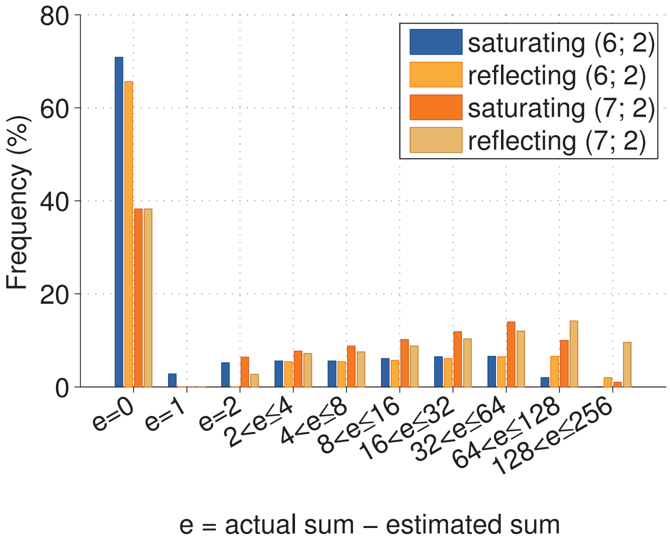


Figure 10.9: Error frequency for 7 bit approximating compressors given uniformly distributed random inputs. Results from 1,000,000 samples for each compressor.

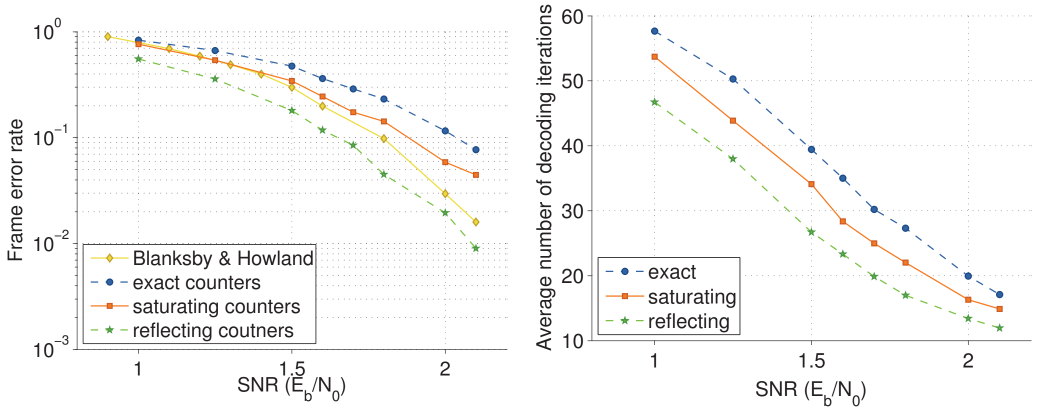
an exact compressor. It now remains to evaluate the effect of this approximate arithmetic on *LDPC* decoder performance.

10.5 LDPC decoding performance

MATLAB simulations were used to measure the performance of the *LDPC* decoder. An additive

Table 10.3: Synthesis results for check node sum logic with and without approximating compressors.

Check node sum circuit	Delay	Area	Power
	(ns)	(μm^2)	(mW)
7 input, exact compressor	2.21	201531	124.9
7 input, approximating compressor, saturating counters	1.95	208371	119.9
7 input, approximating compressor, reflecting counters	1.70	185281	110.8
6 input, exact compressor	1.83	167369	100.9
6 input, approximating compressor, reflecting counters	1.62	162504	97.3



(a) LDPC frame error rate vs. SNR.

(b) LDPC average iterations vs. SNR.

Figure 10.10: Frame error rate (FER) and average decoder iterations versus signal-to-noise ratio (E_b/N_0) for the *LDPC* decoders.

white gaussian noise (AWGN) channel with quadrature phase-shift keying (QPSK) modulation was assumed. Results are shown in Figure 10.10. Recall that a 1056 bit, rate $^1/_2$, soft-decision decoder with 4 bit messages was used. Decoding was terminated after 64 iterations.

The following properties were measured:

Frame error rate The ratio of data frames that are incorrectly decoded.

SNR (E_b/N_0) A dimensionless measure of the signal-to-noise ratio (SNR), normalised per transmitted data bit. $E_b$is the energy per transmitted bit of data, excluding check bits. $N_0$is the noise spectral ratio.

The results from Blanksby & Howland [Howland and Blanksby, 2001] are for a 1024 bit, rate $^1/_2$ regular *LDPC* code with 4 bit messages and exact compressors. Decoding here was also terminated after 64 iterations.

The results are surprising because the decoders with approximate compressors in their check nodes actually exhibit slightly better decoding performance than their exact counterparts. A simulation error in the relative performance of the exact and approximate 1056 bit decoders is unlikely because their source code is identical except for the few lines implementing the check node sums. The hypothesis to explain this behaviour is that the perturbations introduced by the approximate additions help the decoder to converge in a manner analogous to simulated annealing: they bump the decoder from local maxima, allowing it to converge more frequently on the correct result. More experiments are required to test this hypothesis.

The approximate decoders achieved improved frame error rate (FER) at a given SNR level (E_b/N_0), and with fewer iterations on average. Therefore the benefit of reduced hardware latency was not lost by increasing the number of decoding iterations. The FER of the approximate versions with saturating and reflecting counters was very similar, but the version with saturating counters appeared to require fewer iterations on average. Given that the reflecting counters were slightly faster, the best choice for a given implementation depends on the delay of the counters relative to the total time per iteration.

10.6 Conclusions

Approximate multioperand adders were constructed from several approximate $(4;2)$ counters. Synthesis results indicated that approximate multioperand adders exhibited reduced delay, silicon area and power consumption compared with exact multioperand adders constructed using exact $(3;2)$ counters.

Approximate 7 bit 6- and 7-input approximate compressors constructed from approximate counters were then used in the check nodes of an *LDPC* decoder. The approximate decoders achieved a better FER and lower average number of decoding iterations, compared to a baseline decoder using exact compressors.

The rate $^1/_2$ code from *IEEE-802.16e* was used to simulate check nodes with at most 7 inputs. Higher rate codes, such as the rate $^2/_3$, $^3/_4$ and $^5/_6$ variations require check nodes with 10, 11, 15 and 20 inputs respectively. Further research could attempt to implement decoders at these rates with logically incomplete arithmetic units. Furthermore, the implementation in this chapter only considered approximate arithmetic in the *LDPC* check nodes. The effect of using approximate arithmetic in the variable nodes was not considered.

Chapter 11

ADVS Simulation

"They say three weeks in the lab will save you a day in the library, every time."

R. Stanley Williams (1952 —)

This chapter uses the hardware developed in Chapters 6, 7, 8 & 9 in a processor pipeline. The operation latencies of arithmetic units in machine cycles are derived from synthesis results and used in an architectural simulation. A range of arithmetic benchmarks are simulated in a processor with result caching and arithmetic data value speculation enabled to determine the performance gain. The cost of ADVS is assessed in terms of the increased area and power due to the exact arithmetic units used for result checking.

VALUE speculation increases the complexity of processor pipelines, due to the hardware required to predict a variable that could assume many values, and the support hardware necessary to correctly maintain the architectural state if a misspeculation occurs.

The aim of this chapter is to provide a detailed case study of *ADVS* in a realistic system. Approximate hardware units developed in previous chapters are used in a processor pipeline. The necessary pipelining of exact and approximate arithmetic units is discussed in Section 11.1. Changes are made to the baseline *SimpleScalar* model to match the characteristics of the synthesised exact hardware. The cost of the speculation scheme is evaluated in terms of the additional resources dedicated to the approximate arithmetic calculations. The cost of maintaining pipeline state is not addressed, because of it's complexity and dependence on the microarchitecture of the system. The area dedicated to handing incorrect speculations depends on the forwarding paths available in the system before *ADVS* is enabled, the floor-plan and proximity of the front-end to the execution pipelines, and the technology used.

In Section 11.2 the benefit of the *ADVS*-enabled system is evaluated by performing execution driven simulations of an out-of-order, superscalar processor, with approximate integer arithmetic hardware. As a further enhancement, result caching is introduced to mitigate the circumstances where a repeated pathological case for the approximate arithmetic hardware causes excessive flushes due to incorrect approximations. The probability of correctness is analysed per benchmark, and averaged for different benchmark sets, including the *test* set containing benchmarks that have not been used to optimise the approximate hardware units.

11.1 Synthesis

The purpose of this section is to investigate the physical cost of implementing *ADVS*, and to provide realistic parameters that can be used to simulate systems that have been modified to include *ADVS*. First, exact arithmetic units from previous chapters are adopted for use in *SimpleScalar* by adding pipelining registers. Then, the process is repeated for approximate arithmetic units. The results are compared, and the cost of the arithmetic logic, including pipelining overhead is presented as additional power and area expenditure.

11.1.1 Pipelined arithmetic units

In previous chapters, most of the exact and approximate arithmetic cells synthesised were not pipelined. To compare the cycle latency of arithmetic units, pipelining registers were manually inserted, and the units were re-synthesised. To pipeline the designs, the clock period must be defined. In previous chapters, the arithmetic units were synthesised assuming that a unit sized *DFF* was connected as a load. An initial attempt at pipelining was made, where each unit was divided into stages so that the total number of cycles would match the *SimpleScalar* arithmetic cycle latencies as closely as possible. The resulting cycle latencies were then compared, and it was decided that the best match would occur if the clock period was set to 5 ns, corresponding to an 18 % increase above the latency of the integer adder. The extra overhead allows for additional logic so that the adder could be used in an *ALU*. Sequencing overhead was included in the basic adder design.

With the clock period determined, the exact arithmetic units were re-synthesised. In some cases more or fewer cycles were required so the cycle latencies of the units used in the simulations were adjusted accordingly. The cycle latencies of the exact units are used in Section 11.2 as the cycle latencies of the unmodified baseline system, without *ADVS*.

Pipelining registers were then inserted into the approximate arithmetic units, with the same cycle latency of 5 ns. It was confirmed that is was possible to reduce the number of cycles required for the approximate arithmetic units, due to the logic that had been removed from the data path. The difference between the exact and approximate latencies determines the potential gain of *ADVS*. The probability of correctness was seen in Chapter 4 to strongly affect the potential loss in throughput, due to pipeline flushes and instruction replays. The approximate cycle latencies are also used in Section 11.2 to simulate execution with *ADVS*.

The following subsections discuss the pipelining and synthesis of the exact and approximate arithmetic units suitable for *ADVS*. Lastly, a summary is presented showing the additional area and power requirements for the approximate arithmetic units. Integer adders are not shown, because addition was defined to be a single cycle operation, so a lower latency approximate variant was not possible.

11.1.1.1 Pipelined exact integer multiplier

Multiplier cells were synthesised from *VHDL*, and verified with test-benches. The *VHDL* source files were generated using a program written in *C* that could be set to generate Wallace or Dadda tree multipliers. The program was generalised so that the number of input and output bits for all compressors in the multiplier tree can be selected. An optional parameter determined the logic depth between latches, and was used to construct the pipelined version of the multiplier. The depth pa-

Table 11.1: Synthesis results for exact integer multipliers.

Type	Cycles	Clock (ns)	Latency (ns)	Area (μm^2)	Power Dyn. (mW)	Power Leak. (nW)
unsigned						
unpipelined	1	5.78	5.78	1.585	514.8	2,428.3
pipelined	3	3.41	10.23	1.622	92.1	1,363.3
signed						
unpipelined	1	5.82	5.82	1.553	515.7	2,420.0
pipelined	3	3.47	10.41	1.670	94.5	1,410.5

rameter defined the maximum number of compressors that were inserted between each pipelining register. The pipelining registers were inserted at a uniform depth in a cross section of the tree.

The results of pipelining are shown in Table 11.1. The unpipelined version is the exact (3; 2) multiplier used as the baseline multiplier in Chapter 6 *Approximate Integer Multiplication*. The first array of latches was inserted before the inputs to the final carry-save adder. The latency of the tree structure was too high for one clock cycle, so another row of latches were inserted in the tree, dividing it into two parts. A modest area overhead for the timing elements was introduced, but the latency was significantly increased by the synchronisation. The effect on power was less dramatic because the timing paths are less aggressive than minimising the entire tree and CSA from inputs directly to the outputs.

11.1.1.2 Pipelined approximate integer multiplier

The approximate integer multipliers in Section 6.6 were not pipelined, so pipelined versions of the signed (4; 2) and (8; 2), and unsigned (4; 2) and (7; 2) multipliers were synthesised. As shown in Section 6.6, the (8; 2) and the (7; 2) multipliers are faster than the signed and unsigned (4; 2) multipliers, however, when pipelined, these multipliers all provided the result in 2 cycles; at least 1 cycle faster than the exact multiplier. The (4; 2) multipliers were used in *ADVS* simulations because they met the correctness and latency requirements.

Results from synthesis are shown in Table 11.2.

The approximating (7; 2) and (8; 2) multipliers were faster than the (4; 2) multipliers because the higher degree of approximation in each counter resulted in a shorter tree depth and less interconnect.

Table 11.2: Synthesis results for approximate integer multipliers.

Type	Cycles	Clock	Latency	Area	Power	
		(ns)	(ns)	(μm^2)	Dyn. (mW)	Leak. (nW)
			unsigned			
unpipelined (4; 2)	1	5.16	5.16	1.238	398.5	1,903.6
pipelined (4; 2)	2	3.77	7.54	1.703	87.3	2,208.0
unpipelined (7; 2)	1	4.95	4.95	1.123	366.4	1,598.8
pipelined (7; 2)	2	3.11	6.22	2.036	54.5	1,9153
			signed			
unpipelined (4; 2)	1	5.26	5.26	1.236	390.5	1,827.1
pipelined (4; 2)	2	3.77	7.54	1.695	88.7	2,219.4
unpipelined (8; 2)	1	4.61	4.61	1.122	353.3	1,599.7
pipelined (8; 2)	2	3.96	7.92	1.956	73.7	2,863.1

Table 11.3: Synthesis results for approximate pipelined exact integer dividers.

Type	Cycles	Clock	Latency	Area	Power	
		(ns)	(ns)	(μm^2)	Dyn. (mW)	Leak. (nW)
unpipelined	1	11.86	11.86	1.365	303.0	1,488.9
pipelined	5	4.76	23.80	1.241	60.8	678.0

11.1.1.3 Pipelined exact integer divider

The exact integer divider in Chapter 7 was based on iterative SRT division, so already contained pipelining registers, (see Figure 7.1). The exact integer dividers used for the baseline simulations were the same unsigned radix-4 SRT dividers discussed in Section 7.3.2 and Section 7.4, and required 1 initialisation cycle, 16 division cycles, and 1 accumulation cycle.

Synthesis results for the exact and approximate dividers are shown in Tables 11.10, and 11.7–11.9 on pages 248–250. As demonstrated in Chapter 7, the correctness of the unsigned approximate divider was suitable for *ADVS*, but the correctness of the signed integer divider was too low, so only the unsigned approximate divider was used in the *ADVS* simulations in the next section. It is unfortunate that the correctness of the signed integer divider was not high enough, because signed integer division is much more common (see Table 4.2).

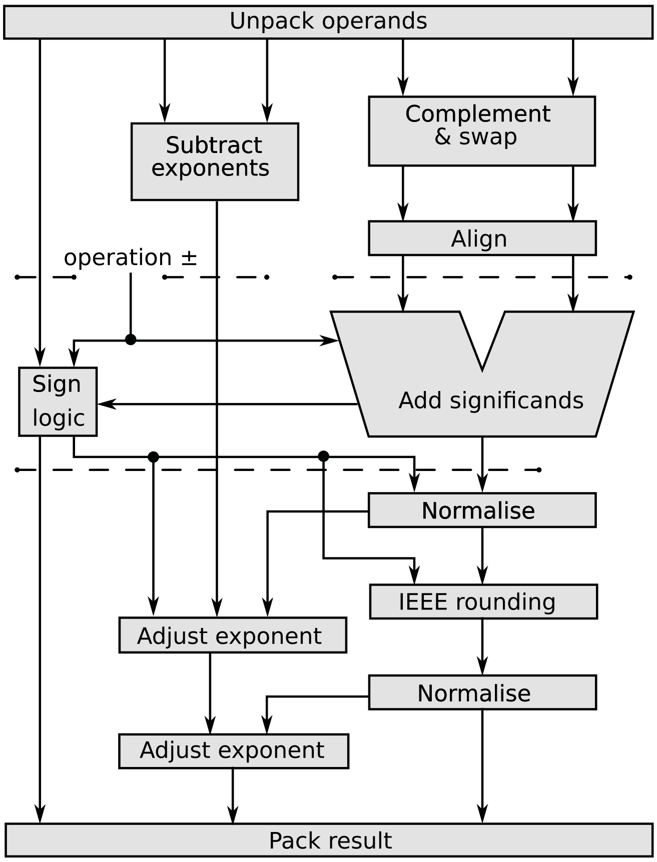


Figure 11.1: Basic design of an IEEE Std. 754 floating point adder/subtractor. Pipelining registers were inserted at the dashed lines.

11.1.1.4 Pipelined approximate integer divider

Similar to the exact integer divider, the approximate integer divider synthesised in Chapter 7 contained pipelining registers (see Figure 7.1). Synthesis results are shown in Table 7.5. The signed and unsigned approximate integer dividers were used without modifications, because the clock period was suitable to use with the other arithmetic units in the system *ADVS* scheme.

11.1.1.5 Pipelined floating point adders

A floating point adder was first shown in Chapter 8. A design with pipelining registers inserted between significand stages is shown in Figure 11.1.

This `fpAdd` was unmodified because the cycle latency of the unit is the same as the *SimpleScalar* latency, and the critical path delay of the longest stage, the normalisation and rounding, is within the target 5 ns goal. Full synthesis results are shown in Table 11.4.

Table 11.4: Synthesis results for the pipelined floating point adder.

Type	Cycles	Clock	Latency	Area	Power	
		(ns)	(ns)	(μm^2)	Dyn. (mW)	Leak. (nW)
pipelined `fpAdd`	3	4.98	14.96	0.844	15.3	522.7

Table 11.5: Synthesis results for the pipelined floating point multiplier.

Type	Cycles	Clock	Latency	Area	Power	
		(ns)	(ns)	(μm^2)	Dyn. (mW)	Leak. (nW)
pipelined `fpMult`	4	19.04	47.08	1.241	60.8	678.0

A general approach for approximating floating point arithmetic was introduced in Section 8.1, where the block operating on the significand is replaced with approximating arithmetic. If this were applied to the floating point adder in Figure 11.1, pipelining registers would isolate the approximate logic from the exact logic. The significand addition occurs in one machine cycle, so approximation by this method will not yield a faster floating point adder in terms of clock cycles. For this reason no approximate pipelined unit was synthesised.

11.1.1.6 Pipelined exact FP multiplier

An approximate `fpMult` was presented in Chapter 8, where the approximate integer multiplier from Chapter 6 was resized to 24 bits and used for the significand, including the 'hidden one' bit. The initial version was not pipelined, so *DFF*s were later inserted, as shown in Figure 11.2. Like the pipelined `fpAdd`, the multiplier clock period was set by the normalisation and rounding stage. The clock period was less than 5 ns, so the pipelined exact multiplier was adopted for simulations later in this chapter.

Synthesis results are shown in Table 11.5. As the latency of the approximate significand multipliers was *higher* than the exact 24 bit significand multiplier, the approximate `fpMult` was not used in the *ADVS*-enabled processor core. Generally, the latency of the approximate counters used in the approximate multipliers is much greater than the exact counters. The synthesiser typically used the library provided *FA* cell, which is hand placed and optimised for speed. To synthesise other counters such as the $(4;2)$, other primitive gates from the library were used. In order for an approximate multiplier to be faster than an exact tree multiplier, the latency saving form reducing the number

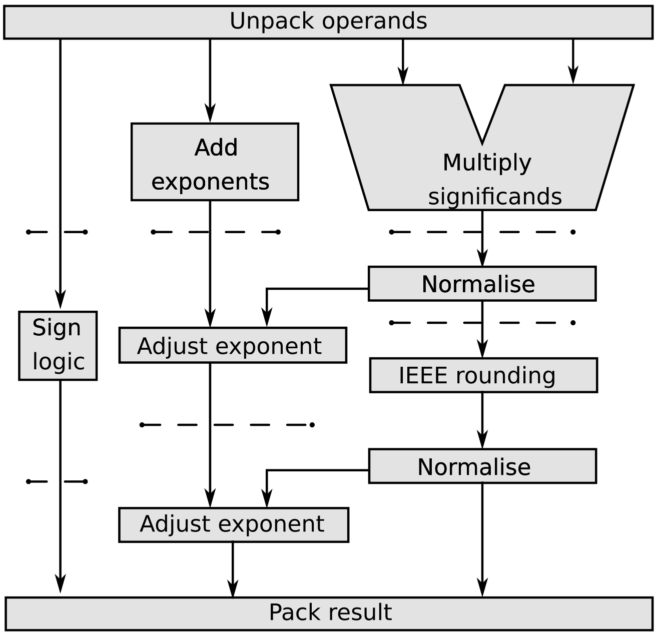


Figure 11.2: Basic design of an IEEE Std. 754 floating point multiplier. Pipelining registers were inserted at the dashed lines.

of levels and fanout at each level must be higher than the cost of the slower gates. This was the case for the wider 32 bit integer multipliers that require more levels in the multiplier tree but not for the narrower 24 bit significand multipliers. This difference in cost could be mitigated with custom approximate counter cells.

11.1.1.7 Pipelined exact FP divider

A floating point divider was synthesised with pipelining registers inserted in between major stages, shown in Figure 11.3. Additional registers were inserted into the significand division stage, for iteration of the partial remainder (not shown).

The floating point divider was synthesised with the same constraints and tools as the other arithmetic units; results are shown in Table 11.6.

For comparison, synthesis results for the exact unpipelined floating point divider from Chapter 8 are shown in Table 11.6. The unpipelined version simply provides a reference for the area and power of the pipelined version when synthesised for minimum latency. The unpipelined version is functionally incorrect because the iterative division algorithm requires the use of the pipelining registers and feedback. The total area and power reduced with the addition of the pipelining registers because the 5 ns constraint on the clock allowed many timing paths to relax in each circuit stage. Likewise

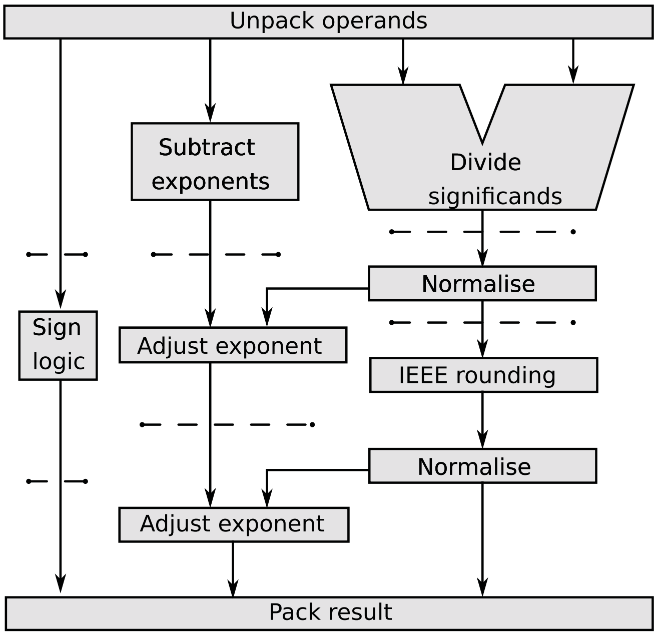


Figure 11.3: Basic design of an IEEE Std. 754 floating point divider. Pipelining registers are inserted at the dashed lines.

Table 11.6: Synthesis results for the pipelined floating point divider.

Type	Cycles	Clock	Latency	Area	Power	
		(ns)	(ns)	(μm^2)	Dyn. (mW)	Leak. (nW)
unpipelined `fpDiv`	1	—	—	0.602	29.9	528.9
pipelined `fpDiv`	11	4.28	47.08	0.551	17.2	238.9

Table 11.7: ADVS arithmetic unit area cost.

Area (μm^2)						
	Baseline			ADVS		
Unit	Qty.	Area	Total	Qty.	Area	Total
`uintAdd/intAdd`	4	0.018	0.092	0		
`uintMult`	1	1.622	1.622	1	1.703	1.703
`intMult`	1	1.670	1.670	1	1.695	1.695
`uintDiv`	1	0.717	0.717	1	0.539	0.539
`intDiv`	1	0.717	0.717	0		
`fpAdd`	1	0.844	0.844	0		
`fpMult`	1	1.241	1.241	0		
`fpDiv`	1	0.551	0.551	0		
Total			**7.454**			**3.937** **+53.82 %**

the dynamic power reduced because the effective toggle rate is determined by the clock.

As shown in Chapter 8, the correctness of the approximate significand divider was too low for *ADVS*, hence no pipelined approximate `fpDiv` was synthesised.

11.1.2 Synthesis summary

This section presents a summary of the pipelined exact and approximated arithmetic units used in the *ADVS* enabled system. Tables 11.7–11.9 show the area, leakage power, and dynamic power of the pipelined units presented. The baseline system contains exact arithmetic units, and the *ADVS*-enabled system contains the exact units in the baseline system, plus additional approximating units for unsigned integer multiplication and division, and signed multiplication. In each case, the percentage increase is shown for the *ADVS*-enabled system.

Table 11.7 shows the total increase in area due to the additional approximate arithmetic units. The largest units were the multipliers; all of the arithmetic units were optimised for speed. The layout of tree multipliers is irregular leading inefficient placement and high interconnect area. Interconnect dominated most of the other designs but to a lesser degree. Furthermore, the pipelining scheme was simple, and performed by inserting registers at regular depths as measured by the number of

Table 11.8: ADVS arithmetic unit dynamic power cost.

Dynamic Power (mW)						
	Baseline			ADVS		
Unit	Qty.	Power	Total	Qty.	Power	Total
`uintAdd/intAdd`	4	0.907	3.628	0		
`uintMult`	1	92.1	92.1	1	87.2	87.2
`intMult`	1	94.5	94.5	1	88.7	88.7
`uintDiv`	1	19.8	19.8	1	10.1	10.1
`intDiv`	1	19.8	19.8	0		
`fpAdd`	1	15.3	15.3	0		
`fpMult`	1	60.8	60.8	0		
`fpDiv`	1	17.2	17.2	0		
Total			**323.1**			**186.0**
						+57.56 %

approximate counters in the data path. Approximate multipliers could be investigated further with different area and timing constraints, to determine if the difference in latency between exact and approximate versions increases or decreases with respect to the fast case presented.

Table 11.8 shows that the approximate and exact multipliers consume the most dynamic power, corresponding to the large area footprint. In the case of the exact `fpMult`, the significand multiplier was synthesised in the same way as the approximating tree multipliers, but exact $(3;2)$ counters were substituted for the approximating counters. The significand multiplier is the dominant component in the `fpMult` unit for area and power due to the aggressive latency optimisations. The power consumed by the `fpMult` is less than the `intMult` because the power saved by scaling down the operand width from 32 to 24 bits is greater than the added power of the additional rounding and pre-shift logic. Dynamic power increased by approximately 58% in the *ADVS*-enabled scheme, similar to the 54% area overhead.

The leakage power increased by approximately 90%, and is due to the approximate integer multipliers. The increase in leakage power of the exact multipliers compared to other exact units is roughly proportional to the relative area of each circuit, however, the leakage power of the approximate multipliers is not representative of the small area increase compared to the exact multipliers.

The approximate multipliers consume approximately 15% more interconnect, 16% more cells, 23%

Table 11.9: ADVS arithmetic unit leakage power cost.

	Leakage Power (nW)					
	Baseline			ADVS		
Unit	Qty.	Power	Total	Qty.	Power	Total
`uintAdd/intAdd`	4	5.755	23.0	0		
`uintMult`	1	1,363.3	1,363.3	1	2,208.4	2,208.4
`intMult`	1	1,410.5	1,410.5	1	2,219.4	2.219.4
`uintDiv`	1	569.5	569.5	1	408.4	408.4
`intDiv`	1	569.5	569.5	0		
`fpAdd`	1	522.7	522.7	0		
`fpMult`	1	678.0	678.0	0		
`fpDiv`	1	238.9	238.9	0		
Total			**5,376.3**			**4,836.2**
						+89.96 %

more cell area, but nearly 90% more leakage power. In both cases, the types of cells used by the synthesiser were similar, but the approximate multiplier used more cells, with a larger drive, capacitance and leakage then the exact multiplier. Although the depth of the multiplier tree is shorter for approximate multiplier in terms of counters, the synthesised multipliers were factored differently by the synthesiser to minimise the critical path delay.

Unlike the approximate multipliers, the approximate divider consumed less area and power than the exact counterpart, so the overall impact of the additional approximate `uintDiv` unit was small compared to the total for all the exact arithmetic units combined.

11.2 ADVS simulation

This section describes simulations to determine throughput of a processor pipeline using *ADVS*. First, the delays of the synthesised approximate arithmetic units are compared, and expressed as machine cycles. Secondly, the machine cycles are used in simulations of an *ADVS* enabled system. The ultimate gain from *ADVS* is determined by the correctness of the arithmetic units, and the latency reduction by approximation.

Table 11.10: Latencies of the pipelined exact and approximate arithmetic units used in the simulation of an ADVS enabled system.

Arithmetic	*SimpleScalar*	Exact		Approx.	
unit	cycles	period (ns)	cycles	period (ns)	cycles
intAdd/uintAdd	1	4.14[1]	1	—[2]	—
uintMult	3	3.41[3]	3	3.77[4]	2
intMult	3	3.46[5]	3	3.77[6]	2
uintDiv	20	4.45[7]	18	3.45[8]	10
intDiv	20	4.45[9]	18	—[10]	—
fpAdd	2	4.92[11]	3	—[12]	—
fpMult	4	4.76[13]	3	—[14]	—
fpDiv	12	4.89[15]	11	—[16]	—

[1] Sklansky adder in Table 3.5.
[2] Approximate adders cannot operate faster than 1 cycle (see Chapter 8).
[3] Unsigned (3; 2) multiplier in Table 11.1.
[4] Unsigned approximate (4; 2) multiplier in Table 11.2
[5] Signed (3; 2) multiplier in Table 11.1.
[6] Signed approximate (4; 2) multiplier in Table 11.2
[7] The baseline unsigned SRT divider was used from Section 7.5 (see Table 7.5).
[8] The correctness and latency of the unsigned divider are shown in Section 7.3 and Table 7.5.
[9] Unsigned exact SRT radix-4 divider in Table 7.5.
[10] The correctness of the signed approximate divider was too low (see Section 8.2.3).
[11] Pipelined exact fpAdd (Table 11.4).
[12] The pipelined significand addition stage in fpAdd could not be shortened below one cycle (see Chapter 8).
[13] Pipelined exact floating point multiplier in Table 11.5.
[14] The latency of the proposed approximate significand multipliers was too high (see Section 8.3).
[15] Pipelined exact fpDiv unit is shown in Table 11.6.
[16] The correctness of approximate fpDiv unit was too low for ADVS, see Chapter 8.

11.2.1 Simulated arithmetic latencies

Table 11.10 shows the latencies of the synthesised approximate arithmetic units. Some of the default latencies used by *SimpleScalar* are different to the synthesised units. The cycle latencies of the synthesised units are dependent on a number of factors, including the target library, and process conditions.

In the following sections simulations are used to determine the effectiveness of an arithmetic data value speculation scheme. Later, result caching is introduced to reduce the penalty of repeated calculations that are incorrectly approximated. In the simulations, a new baseline throughput rate is established for each benchmark, due to the differences when using the latencies of the synthesised approximate units.

11.2.2 Benchmark simulation with ADVS

This section presents the results of simulation of the *ADVS*-enabled system.

The results for the correctness of some operations are different to the values quoted in previous chapters for the individual approximate units. Only the first 10 million instances of any operation were recorded in each instruction trace, but the results shown here include the entire benchmark execution. Also, compiler optimisation was not enabled in the initial traces, and execution was performed in order. The simulations performed in this chapter use *gcc -O2* optimisation.

Level two optimisation enables an additional static code scheduling pass by the compiler. Assuming that the compiler is properly tuned for the *SimpleScalar* defaults arithmetic latencies, this could improve performance, or at worst have no impact because the arithmetic latencies used (shown in Table 11.10) are not longer than the *SimpleScalar* defaults. Additional optimisations are detailed in Section A.1.

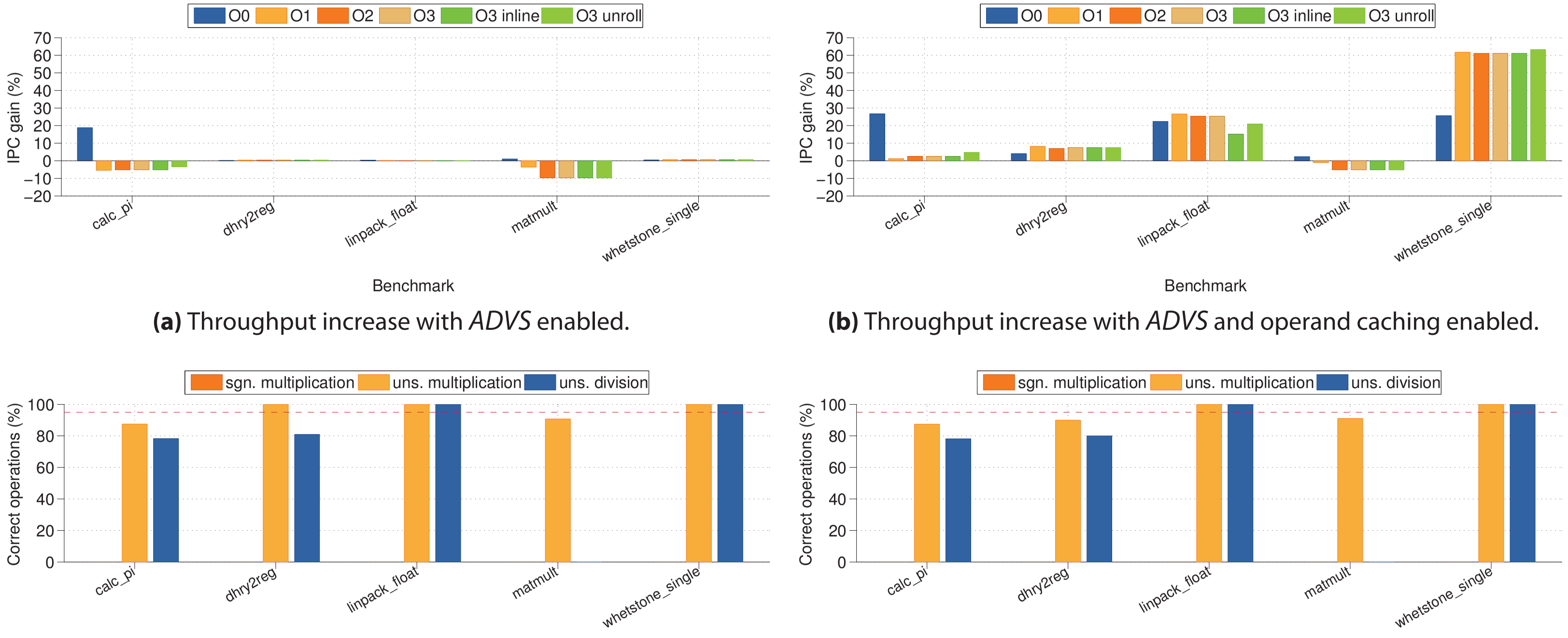
Later, a selection of the benchmarks are analysed in detail, with attention paid to benchmarks with a very high or very low correctness, a significant change in IPC when *ADVS* was enabled, or a significant improvement through result caching.

The *ADVS*-enabled simulation results are presented in sets. Figure 11.4 shows the results for the *arithmetic* benchmarks, Figure 11.5 shows *Mediabench* results, Figure 11.6 shows the *SPEC CINT2000* benchmarks, and finally the *test* benchmarks are shown in Figure 11.7.

All of the figures show the IPC impact for the all benchmarks in the set, at different optimisation levels in subfigure *(a)*. In comparison, the IPC improvement of a system implementing *ADVS* and operand caching is shown alongside in subfigure *(b)*. Beneath the IPC plots are histograms of the achieved correctness of each approximated operation, shown in subfigures *(c)* and *(d)*. The correctness histograms are only shown for binaries compiled with optimisation *-O2*.

The cache configuration is shown in Section 9.3.

The average proportion of correct approximations in each benchmark set is shown in Tables 11.11 and 11.12 for systems with *ADVS*, and with *ADVS* and result caching. As discussed above, some of the average correctness values are slightly different to the values quoted in previous chapters. Note also that the operands from the *test* benchmark set has not been previously simulated.



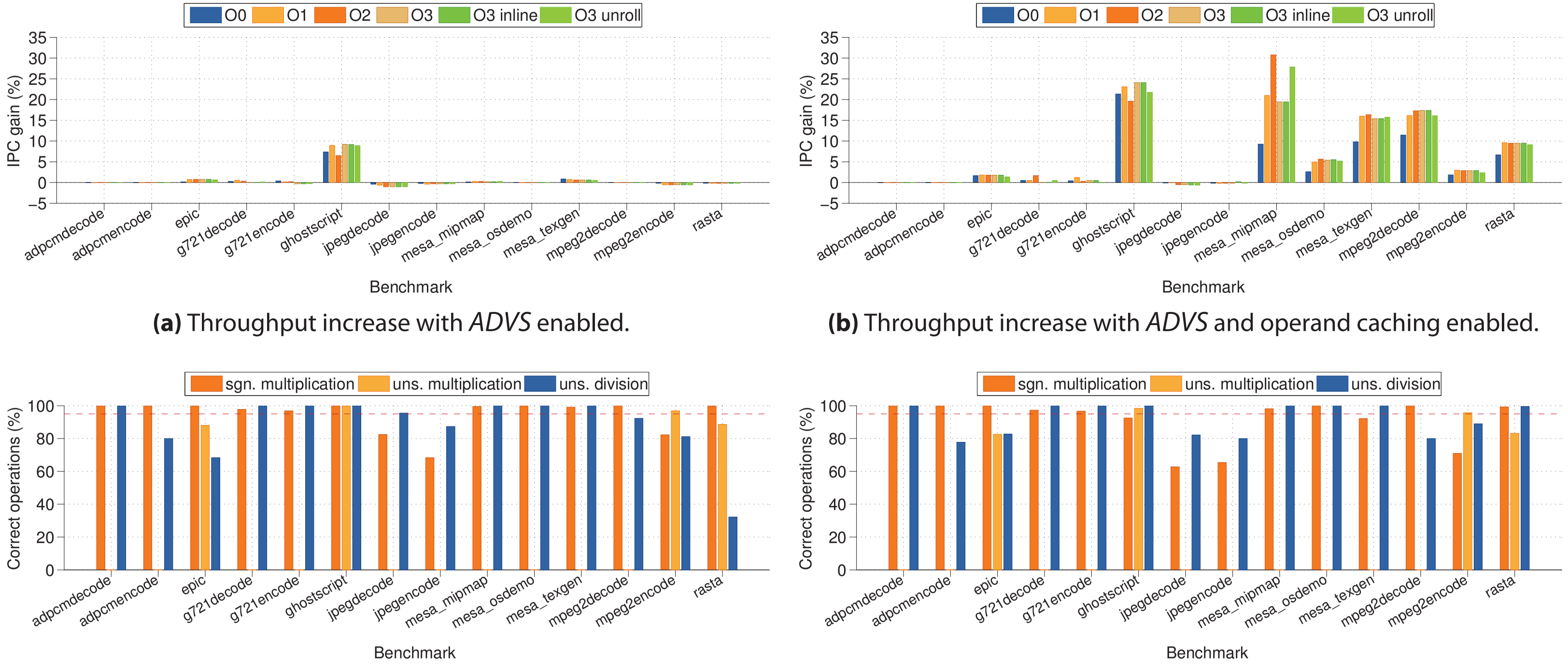
(a) Throughput increase with *ADVS* enabled.

(b) Throughput increase with *ADVS* and operand caching enabled.

(c) Correctness of approximated operations. Target correctness of 95 % is shown a dashed line.

(d) Correctness with operand caching enabled.

Figure 11.4: IPC of *arithmetic* benchmarks in an ADVS-enabled system.



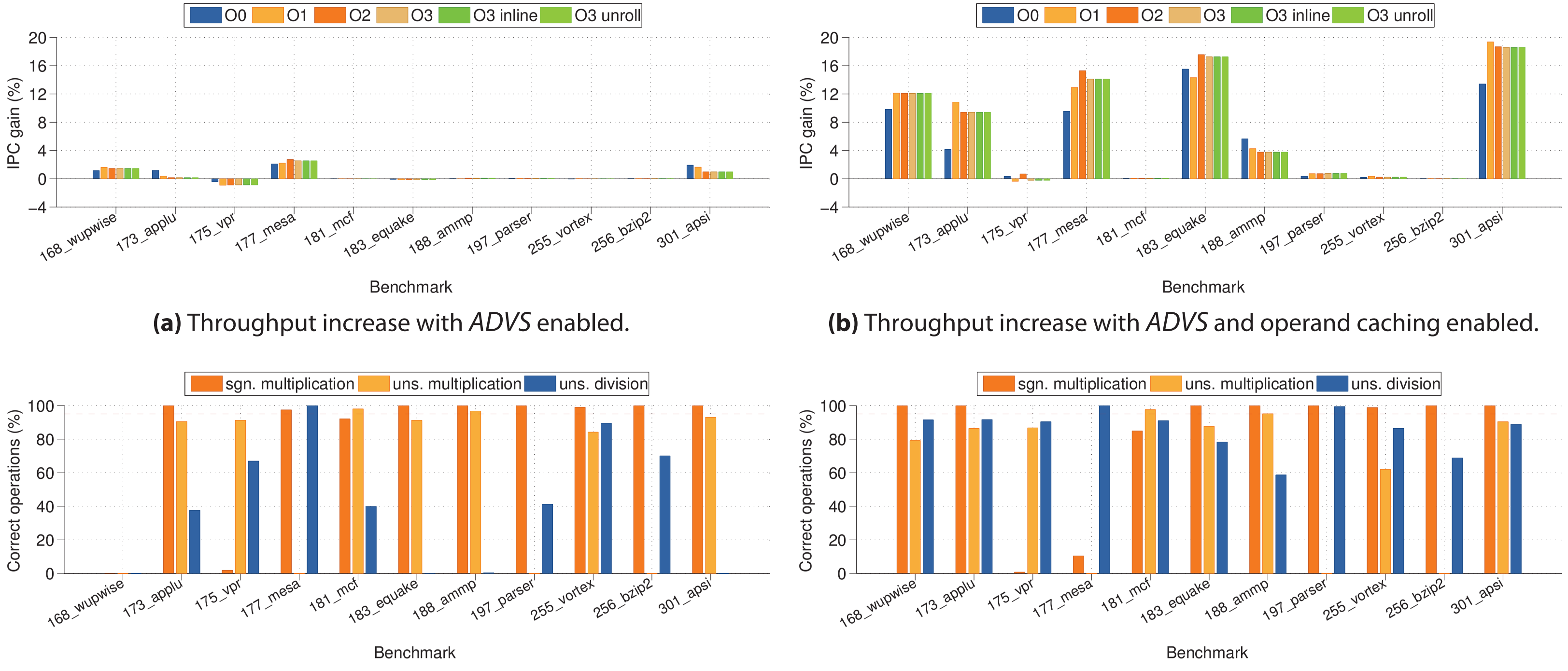
(a) Throughput increase with *ADVS* enabled.

(b) Throughput increase with *ADVS* and operand caching enabled.

(c) Correctness of approximated operations. Target correctness of 95 % is shown a dashed line.

(d) Correctness with operand caching enabled.

Figure 11.5: IPC for *Mediabench* benchmarks in an ADVS-enabled system.



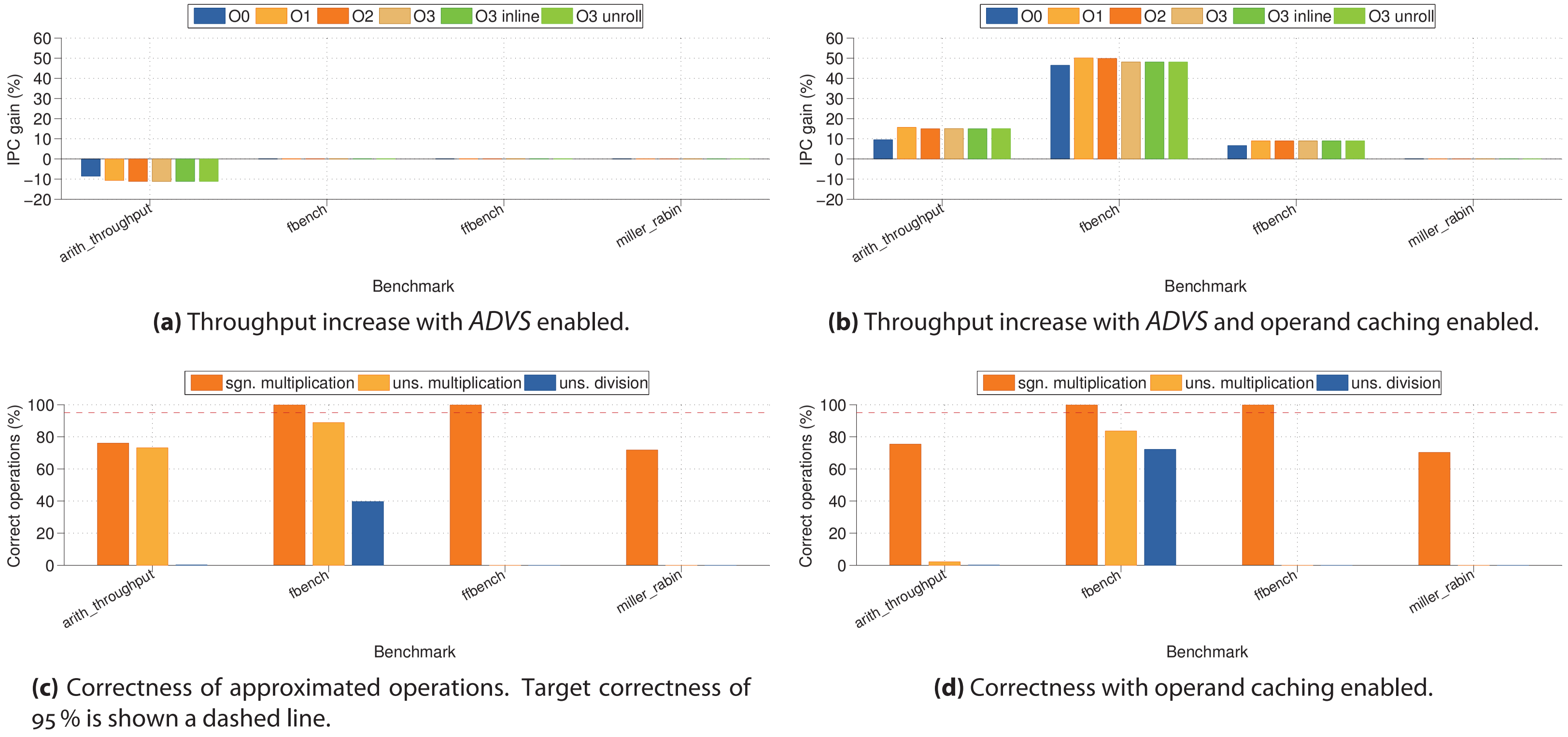
(a) Throughput increase with *ADVS* enabled.

(b) Throughput increase with *ADVS* and operand caching enabled.

(c) Correctness of approximated operations. Target correctness of 95 % is shown a dashed line.

(d) Correctness with operand caching enabled.

Figure 11.6: IPC of *SPEC* benchmarks in an ADVS-enabled system.



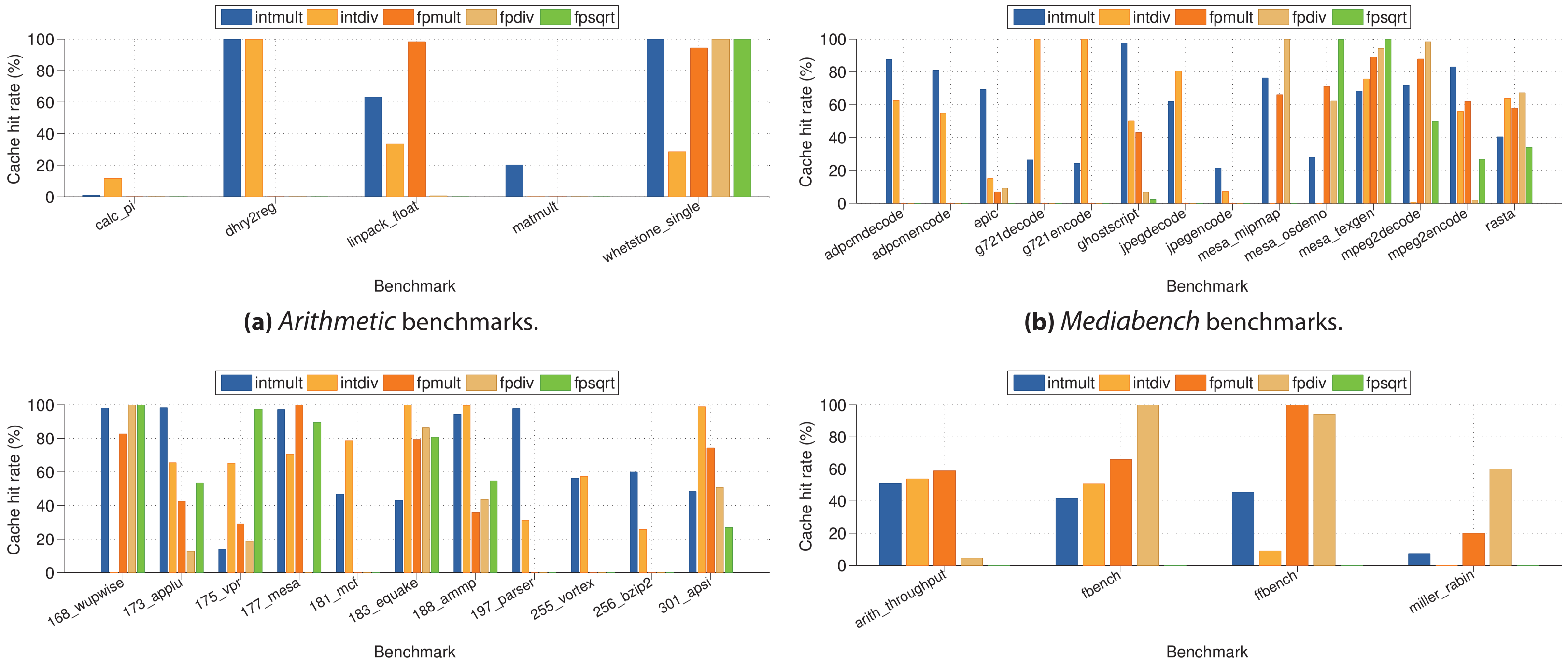
(a) Throughput increase with *ADVS* enabled.

(b) Throughput increase with *ADVS* and operand caching enabled.

(c) Correctness of approximated operations. Target correctness of 95 % is shown a dashed line.

(d) Correctness with operand caching enabled.

Figure 11.7: IPC of *test* benchmarks in an *ADVS*-enabled system.



(a) *Arithmetic* benchmarks.

(b) *Mediabench* benchmarks.

(c) *SPEC* benchmarks.

(d) *Test* benchmarks.

Figure 11.8: Operand cache hit rates in an *ADVS*-enabled system.

Table 11.11: Average proportion of correct approximations of each approximate arithmetic unit with benchmark operands. *SimpleScalar* was modified for with arithmetic latencies from Table 11.10.

Benchmark	`uintMult`	`intMult`	`uintDiv`
Arithmetic	—	95.65	89.81
Mediabench	93.30	93.38	85.16
SPEC	92.15	88.10	43.19
Test	81.02	86.97	19.89

Table 11.12: Average number of correct approximations of each approximate unit in an *ADVS* enabled system, with operand caching.

Benchmark	`uintMult`	`intMult`	`uintDiv`
Arithmetic	—	93.71	89.54
Mediabench	89.97	89.54	90.08
SPEC	85.62	88.44	84.51
Test	42.89	86.45	36.17

Table 11.13 shows the average hit rates of the operand caches in the *ADVS*-enabled system. As shown in Figure 11.7, the *test* benchmarks gained very little IPC, except for the *arith-throughput* benchmark where the low correctness of the approximate `uintMult` and `uintDiv` units caused many instruction replays.

Table 11.14 shows the average IPC change of each of the benchmark sets with *ADVS*, and *ADVS* with result caching. The averages are calculated per benchmark, and are not weighted by the number of retired instructions or executed arithmetic operations. The averages show that the effect on IPC of *ADVS* with the achieved levels of accuracy is low. The IPC rates for the benchmarks with operand caching are high for the *arithmetic* and *test* benchmarks, due to their small memory footprint and repetitive operation. With the IPC change shown per benchmark in Figures 11.4–11.7, and per benchmark set in Table 11.14, it can be observed that the variation in IPC is high for different programs. The average percent IPC change per benchmark in the *arithmetic*, *Mediabench* and *SPEC* sets is 6.37 %, as expected from Figure 9.6, using a caches with 2^8 entries. Including the *test* benchmarks, the average IPC change increases to 9.46 % due to the high cache hit rates of the long latency `fpMult` and `fpDiv` operations.

Table 11.13: Operand cache hit rates in an ADVS-enabled system. Each cache is 4 way set associative with FIFO replacement, indexed by the lower bits in the B operand, and has 64 cache lines.

Benchmark	**`intMult`**	**`intDiv`**	**`fpMult`**	**`fpDiv`**	**`fpSqrt`**
Arithmetic	56.85	43.33	96.34	50.29	99.99
Mediabench	60.40	53.68	51.53	36.70	28.24
SPEC	68.75	62.23	54.13	47.65	72.85
Test	36.35	28.37	61.18	64.61	0.03

Table 11.14: Average percentage change in IPC for each benchmark set, in an *ADVS*-enabled system with and without operand caching.

Benchmark	*ADVS*-enabled	*ADVS* & caching
Arithmetic	4.18	16.25
Mediabench	0.66	3.96
SPEC	0.37	4.94
Test	-2.14	15.67
Average	0.75	9.46

11.2.2.1 Arithmetic benchmarks

Despite the name, the *arithmetic* benchmarks do not contain the highest proportion of arithmetic operations (excluding `intAdd`) of the benchmark sets. The benchmarks are small, perform little input processing, and have a high proportion of loops. There is a low proportion of long latency integer operations; most of the integer operations increment loop counters. Most of the arithmetic operations are floating point, so in the scheme tested, the full benefit of *ADVS* was not realised because many of the arithmetic operations in the benchmarks were not approximated. Despite a high average correctness, the average performance measured as IPC did not improve much. The correctness of all of the approximated operations in *calc_pi* and *matrix_mult* was less that the target, and the IPC suffered due to speculation flushes.

Linpack

The *linpack* benchmark in the *arithmetic* set uses the BLAS (Basic Linear Algebra Subprograms) libraries to test system throughput by performing matrix operations on a 100×100 matrix. First, the matrix $\mathbb{A}$ is generated and reduced by Gaussian elimination, then solved for $\mathbb{A} \times \mathbf{x} = \mathbb{B}$. The main loop is shown below in *C* code. The variable `ntimes` is set to 1, so that the main loop is executed only once.

```c
for (j=1 ; j<6 ; j++)
  {


    for (i = 0; i < ntimes; i++)
      {
        matgen(a,lda,n,b,&norma);

        /* dgefa factors a double precision matrix by gaussian elimination. */
        dgefa(a,lda,n,ipvt,&info );
      }


    for (i = 0; i < ntimes; i++)
      {
        /* dgesl solves the double precision system
         *  a * x = b  or  trans(a) * x = b
         */
        dgesl(a,lda,n,ipvt,b,0);
      }
  }
```

The three functions shown are executed six times each in succession, with the matrix $\mathbb{A}$ in variable `a` initialised to the same values in each iteration. The most common arithmetic operation is floating point addition, accounting for over 8 % of the total number of instructions retired (see Table C.13). It is expected that integer multiplication is also a common operation, used frequently to calculate the array indices when accessing the matrix elements. However, the compiler re-factors the indexing expression so that the index is incremented or decremented in each iteration (see Table C.5).

Despite a high probability of correctness of the approximated integer operations (see Figure 11.4c), the impact to throughput was negligible. With operand caching enabled, the effective latency of the frequent floating point operations is reduced due to the high hit-rate (see Figure 11.8a), causing a significant gain in IPC (see Figure 11.5b).

Whetstone

The *whetstone* benchmark benefits negligibly from *ADVS* (see Figure 11.4a), but the performance significantly improves with result caching (see Figure 11.4b). *Whetstone* iterates thorough a main outer loop containing nested loops that perform test operations including array element access, conditional branches, integer arithmetic, floating point trigonometry and procedure calls. In this small benchmark, the main loop is passed through once, but the nested loops iterate several times.

In the following code fragment, the integer variables are fixed in each loop iteration (`j=1`, `k=2`, `l=3`). The inner loop executes 25,000 times, so the benefit of result caching is realised many times.

```
/* Section 4, Integer arithmetic */
j = 1;
k = 2;
l = 3;
timea = dtime();
{
  for (ix=0; ix<xtra; ix++)
    {
      for(i=0; i<n4; i++)
        {
          j = j *(k-j)*(l-k);
          k = l * k - (l-j) * k;
          l = (l-k) * (k+j);
          e1[l-2] = j + k + l;
          e1[k-2] = j * k * l;
        }
    }
}
```

In the code sequence below, the value of `t` changes in each loop cycle, so `x` and `y` change in each iteration. Some of the inner terms are repeated trigonometric expressions so there is some benefit to result caching, but not as significant as in Section 4 of the code.

```
/* Section 5, Trig functions */
x = 0.5;
y = 0.5;
timea = dtime();
{
  for (ix=0; ix<xtra; ix++)
    {
      for(i=1; i<n5; i++)
        {
          x = t*atan(t2*sin(x)*cos(x)/(cos(x+y)+cos(x-y)-1.0));
          y = t*atan(t2*sin(y)*cos(y)/(cos(x+y)+cos(x-y)-1.0));
        }
      t = 1.0 - t;
    }
  t = t0;
}
```

In the code fragment below, the variable `x` varies in each of the 9,300 iterations, but the cache hit rate remained high in *SimpleScalar* for this section of code. When investigated, a error was found in the output of the log() function in the version of *glibc* library provided with *SimpleScalar* cross-compiler. Table 11.15 shows the manifestation of a cumulative error in the *SimpleScalar whetstone* simulation. The effect of the error is that the value of x in the code fragment repeated periodically when it should be monotonically decreasing, and hence the frequent floating point operations become cacheable, artificially inflating the performance.

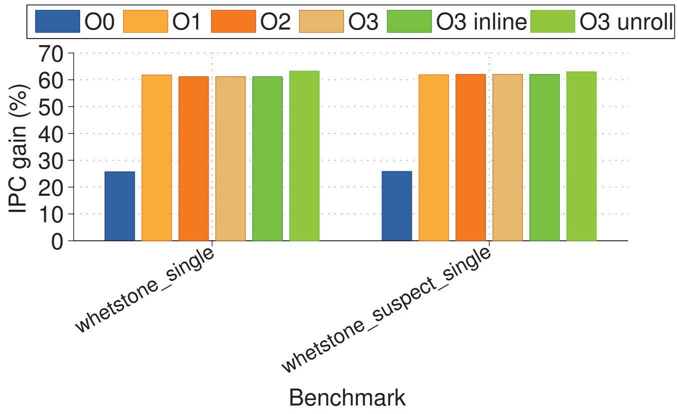
Table 11.15: Differences in *glibc* function values for a PC and *SimpleScalar*.

Iteration	Term	Correct value	*SimpleScalar* value
1	x	`0x3f400000`	`0x3f400000`
	$\log(x)$	`0xbe934b11`	`0xbe934b11`
	$\log(x)/t_1$	`0xbf134b0c`	`0xbf134b0c`
	$\exp(\log(x)/t_1)$	`0x3f100003`	`0x3f100003`
	$\mathrm{sqrt}(\exp(\log(x)/t_1))$	`0x3f400002`	`0x3f400002`
2	x	`0x3f400002`	`0x3f400002`
	$\log(x)$	`0xbe934b0c`	`0xbe934b0b`
	$\log(x)/t_1$	`0xbf134b07`	`0xbf134b07`
	$\exp(\log(x)/t_1)$	`0x3f100005`	`0x3f100006`
	$\mathrm{sqrt}(\exp(\log(x)/t_1))$	`0x3f400004`	`0x3f400004`
3	x	`0x3f400004`	`0x3f400004`
	$\log(x)$	`0xbe934b07`	`0xbe934b06`
	$\log(x)/t_1$	`0xbf134b02`	`0xbf134b01`
	$\exp(\log(x)/t_1)$	`0x3f100008`	`0x3f100009`
	$\mathrm{sqrt}(\exp(\log(x)/t_1))$	`0x3f400005`	`0x3f400006`

```
/* Section 8, Standard functions */
x = 0.75;
timea = dtime();
{
  for (ix=0; ix<xtra; ix++)
    {
      for(i=0; i<n8; i++)
        {
          x = sqrt(exp(log(x)/t1));
        }
    }
}
```

Further investigation found that the source of the error not in the provided library, but in the floating point hardware used on the host system. A floating point error was found using Kahan's *paranoia* program, written in *C* [Kahan, 1986]. Because *SimpleScalar* uses the host machines hardware for many calculations, the bug was exposed on the host Intel® Pentium® 4 CPU, at 3 GHz. The floating point hardware on this platform did not correctly account for guard, round and sticky bits in the intermediate representations of the significand. A sample of the output of paranoia is shown below:

Figure 11.9: Simulated throughput of *whetstone* using *ADVS* on platforms with and without erroneous floating point arithmetic. The simulation marked 'suspect' executed on the erroneous system.



```
 Checking rounding on multiply, divide and add/subtract.
* is neither chopped nor correctly rounded.
/ is neither chopped nor correctly rounded.
Addition/Subtraction neither rounds nor chops.
Sticky bit used incorrectly or not at all.
FLAW: lack(s) of guard digits or failure(s) to correctly round or chop
(noted above) count as one flaw in the final tally below.
```

This problem was addressed by running the *SimpleScalar* simulations on another platform that did not suffer from the same errors. The code labelled 'Section 8' was a small part of the total benchmark, so there was a negligible reduction in IPC when correct floating point arithmetic was used. Table 11.9 shows the total difference in IPC with and without the erroneous floating point arithmetic.

Importantly, the anomaly discovered above does not affect the results of *ADVS* simulation, because there is no approximate floating point hardware used. The results using operand caches are indexed by the lower bits of the operand, and so could be affected by the floating point bug. The caching results were repeated on reliable hardware, and the reliable results were presented in Figures 11.4–11.8.

11.2.2.2 *Mediabench* benchmarks

The *Mediabench* benchmarks are similar to the *arithmetic* benchmarks, but they contain a higher diversity of arithmetic instructions in each benchmark. In most cases, the arithmetic speculation gains due to a high correctness of one operations are offset by a low correctness of another operation. Throughput gains of almost 10 % were achieved by *ghostscript*, due to a very high correctness in the all approximate units, and the highest `uintDiv` proportion of any benchmark. Significant performance gains are made when result caching is introduced, due to effective latency reduction of the other frequent arithmetic operations that were not approximated.

11.2.2.3 *SPEC* benchmarks

In general the *SPEC* benchmarks did not significantly gain in IPC, and only *175.vpr* lost performance at all optimisation levels. In most cases, the correctness of at least one operation was too far below the threshold to gain IPC. Interestingly some benchmarks such as *181.mcf* tolerated very low correctness for `uintDiv` operations, but the number of operations was too small to compromise performance much.

177.mesa and *301.apsi*

The two *SPEC* benchmarks *177.mesa* and *301.apsi* gained the highest percentage in IPC with *ADVS*, and were among the top benchmarks with operand caching enabled.

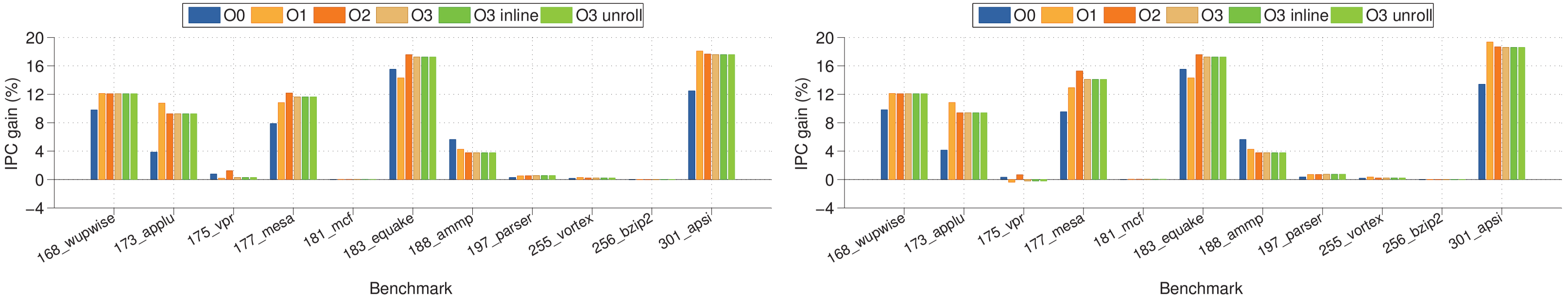
In the case of *mesa*, the correctness of the `intMult` operation fell when caching was introduced. Correctly approximated `intMult` operations that were repeated would eventually be cached, and the correctness of the approximate units would decrease because these operations would not contribute to the correctness average.

The *301.apsi* benchmark increased the correctness of the `uintDiv` operation dramatically with operand caching enabled. In this case an operation would miss in the result cache the first time that is was encountered, and the approximate unit would also approximate incorrectly. On future occasions, the result cache would hit, sparing the repeated loss of correctness to the arithmetic unit.

The effect of the result caching was dominant over the execution cycles saved through approximation. Figure 11.10 shows the throughput increase of *SPEC* benchmarks by using only result caching in Figure 11.10a, isolated from result caching with *ADVS* in Figure 11.10b. The small effects of the *ADVS*-only system in Figure 11.6a can be observed as the small difference between the result caching systems with and without *ADVS*; most benchmarks have incremental gains, and the IPC of *175.vpr* declines slightly.

11.2.2.4 Test benchmarks

Operand data was traced from the *arithmetic*, *Mediabench*, and *SPEC* benchmarks, and was used to determine the correctness of the approximate arithmetic units. These benchmarks form a training set for the approximate arithmetic units. Many benchmarks in different categories were used. They



(a) Operand caching only.

(b) Operand caching and *ADVS*.

Figure 11.10: Effect of operand caching in an *ADVS*-enabled system.

were designed to exhibit a broad range of characteristics, so that they test different aspects of a computer system. However, because the same data was used in the architectural simulations in this chapter, there is a risk that the approximate hardware will be over-optimised for the training set. The *test* set of benchmarks was also simulated to increase the coverage.

A description of the benchmarks in the *test* set is provided in Chapter 2. Most of the arithmetic operations in the *test* benchmarks are floating point operations, especially in the *fbench* and *ffbench* benchmarks, so they benefit greatly from result caching.

arith_throughput

The *arith-throughput* benchmark suffers a approximately a 10 % performance loss with *ADVS* in Figure 11.7a, but this is improved to a >10 % gain with operand caching. Integer multiplication and unsigned division account for approximately 4 % of the total number of instructions retired. In this case the correctness of the approximate divisions was less than 1 % so it is hard to see on the graph. The benchmark tests integer throughput using the code shown below:

```
while (i < intMax)
  {
    intResult -= i++;
    intResult += i++;
    intResult *= i++;
    intResult /= i++;
  }
```

`intResult` is initially zero, and follows the series

`intResult` $= $ `intResult` $+ 1 - 2 * 3/4 + 5 - 6 * 7/8 + 9 - 10 * 11/12$

The operations performed in the first iteration are: $0 \leftarrow 1 - 12 \leftarrow 0 + 26 \leftarrow 2 * 31 \leftarrow 6/4$, and in the second iteration are: $-4 \leftarrow 1 - 52 \leftarrow -4 + 614 \leftarrow 2 * 71 \leftarrow 14/8$

The input operands to the approximate divider in this series tend to yield quotients that are small, and truncated. Often, the approximate algorithm used will terminate after a few correction rounds, and the result would be correctly approximated, particularly when the operands are both positive. In this case the operands were signed but the approximate signed `intDiv` unit was not included in the *ADVS*-enabled system because the average correctness was low in other programs. The poor performance of the `uintDiv` operations outside of this loop were the single biggest contributor to the performance losses in this benchmark.

11.3 Conclusion

The *ADVS*-enabled system presented in this chapter yielded small gains to IPC. As shown in Chapter 4 the correctness requirements of approximate arithmetic hardware are above 90 % on average. Throughput performance was more sensitive to correctness than the speedup in machine cycles of an approximate unit compared to an exact unit. It was demonstrated that the average achieved correctness of the approximate units was near the threshold in most benchmark sets, and the outcome in terms of IPC gain matched predictions made with earlier probabilistic models. It is therefore likely that improving the correctness of the approximate arithmetic units, and extending them to more frequent and longer latency floating point operations will improve the throughput. The average latency improvement using the *ADVS*-scheme described was 1.1 % for the *arithmetic*, *Mediabench* and *SPEC* benchmarks, but this was reduced to 0.75 % with the introduction of the test benchmarks. The overall change in IPC was found to be highly variable between programs. The *177.mesa* and *Mesa* benchmarks share a very similar code base, but the differences in input data yielded highly differences in IPC gain. The variables affecting correctness include compiler optimisation and code structure, input data, and the method of approximation.

The cost of an *ADVS* scheme could be prohibitive unless the application is well known and characterised. Most of the arithmetic units, exact and approximate, were synthesised with speed as the predominant constraint. Hence, the associated costs in area and power are exacerbated, particularly in the case of the integer multipliers. The approximate units increased the arithmetic consumed power and area by approximately 55 % per unit, assuming highly aggressive implementations for the baseline and approximate units.

The introduction of result caching as backup plan for incorrectly approximated values was found to dominate the performance change of the benchmarks. Firstly, operand caching was enabled for operations that could not be approximated with a high enough correctness for *ADVS*, secondly the caches maintained high hit rates, and did not impose a penalty on a miss, and thirdly the operand caches are situated earlier in the pipeline, so the already small effects of *ADVS* are only applicable when the operand caches do miss.

The operand caches selected were deliberately kept as small as possible while maintaining a high positive impact on performance, but they have assumptions on their implementation and use. The total size of the operand caches in bytes is approximately the size of the level 1 `iCache` or `dCache`. The same effective area might be required do to additional wiring for the two operands and result that form each entry. The access time to the caches was also assumed to be 1 cycle, the same as the

SimpleScalar traditional level 1 caches. This could impose a challenge to a floor plan engineer.

This chapter cumulates an investigation of *ADVS* from the system architecture level, down to the level of individual arithmetic units. A systematic approach was adopted to share information between levels of abstraction, so that a realistic *ADVS*-enabled system could be developed as a case study. The throughput results of this chapter have closely matched the outcomes predicted by different models, and validate the correctness targets developed in earlier chapters. The detailed results in this chapter could be used to further develop *ADVS* for use in general purpose processor pipelines by improving the correctness of approximate arithmetic units, or by applying approximate arithmetic to niche applications, such as *LDPC* explored in Chapter 10.

Chapter 12

Conclusions

"Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law."

Douglas Hofstadter (1945 —), in
Godel, Escher, Bach: an Eternal Golden Braid

This chapter contains a brief summary of the designs presented and simulation outcomes presented in previous chapters, with a particular emphasis on the engineering trade offs necessary to implement ADVS, and approximate arithmetic in general. Future research avenues are identified.

Chapter 12: Conclusions

The majority of computing technology has been built around determinism; the system outputs should always be exact and definite for a given set of inputs. Some newer technologies are relaxing this constraint in favour of increased throughput or battery life, or reduced cost. If this trend continues, approximate arithmetic units will be a fundamental component of approximate systems. Even if these technologies are further developed and become commonplace, there will always be a need for high performance exact systems.

Speculation has been employed in many facets of computer technology, throughout the memory hierarchy and within the pipelines to advance the machine state during idle time. This thesis has applied approximate arithmetic to value speculation to leverage further speculative gains and increase throughput.

The top-down approach used in this research determined the design targets for the approximate arithmetic units that were considered suitable for data value speculation. Although arithmetic operations formed a relatively small part of a total program, they were often on the critical path of execution due to their longer latencies. In the case of division and square root this was exacerbated because the operation was not pipelined.

The minimum required correctness for arithmetic speculation was found to be above 95 %. Modern high performance hardware exploits many techniques for latency hiding and parallelism, so the remaining opportunities for improving throughput are harder, and yield only incremental gains.

New approximate arithmetic algorithms and hardware were investigated for each of the fundamental arithmetic units. When attempting to trade latency for correctness in high performance arithmetic units, it was found that the high degree of parallelism inherent in arithmetic circuit designs left correctness susceptible to even small modifications, especially on the critical path that must necessarily be shortened.

A simulation of an *ADVS*-enabled system was conducted with hardware that was shown to have a high correctness and latency less than exact arithmetic units, allowing the processor pipeline to speculate in the approximation window. The goal of this thesis was to investigate the feasibility of using arithmetic value speculation as a mechanism to increase throughput in a processor pipeline. It can be concluded that:

1. Reducing the latency of arithmetic circuits while maintaining high correctness is difficult, due to the logic density and fan out in the data paths of high performance arithmetic circuits.

2. The correctness of approximate arithmetic is sensitive to the input data. Any system that uses approximate arithmetic, for speculation or error-tolerant probabilistic computing, will need to be application specific, and be carefully designed for characteristics of the expected input.

3. The effects of speculation in complex processor pipelines is compounded by multiple levels of speculation, and deeply exploited parallelism. Because approximate arithmetic latency is hard to reduce, potential gains are small, yet potential detriment is high because the pipeline replay caused by an incorrect approximation *must* be on the critical path of the machine, if it were to yield any real time saving.

4. Full implementation of all of the necessary hardware for speculation is expensive, due to the need to replicate hardware. While the considerable costs would have to be carefully balanced against the potential gains, it is unlikely that the expense would be justified without significant improvement to the average correctness of approximate units. The constraints for *ADVS* are imposing, and in measured cases where the achieved correctness was almost 100 % for all of the approximated calculations, the maximum throughput gains were incremental.

5. Although many programs contain repetitive operands, different programs have different behaviours with respect to the predictability of operands. This makes approximation difficult for general purpose computing. Alternative schemes like operand caching are less sensitive to operand values.

6. The performance of *ADVS*-enabled systems is sensitive to many factors. Simulations of benchmarks showed that even where speculative gains were obtained by approximating one operation, they could be offset by another. Because of the variations in performance between programs, but general consistency within programs, arithmetic speculation could be adaptable, and disabled after a repeated hindrance to performance is detected.

7. Approximate arithmetic units can operate over a broad range of delay and correctness. Logical incompleteness typically introduces promising reductions to power, delay and area, so the future applications of approximate arithmetic are likely to be in probabilistic or error tolerant fields, like *LDPC* or multimedia decoding.

8. Programs with a high proportion of arithmetic operations tend to reuse values, therefore caching operand values can improve throughput of long latency operations, but imposes significant area overhead.

12.1 Summary of Contributions

This thesis has made the following contributions:

1. Comprehensive statistics of typical operands observed in a range of benchmark programs was presented in Chapter 2.

2. A survey of approximation methods, including an algorithm for determining the exact probability of correctness for N bit addition with a maximum carry length of l bits was presented in Chapter 3.

3. Performance thresholds for approximate arithmetic units to be used in value speculation in a RISC processor pipeline were established in Chapter 4.

4. Simple characteristics of arithmetic operands in benchmark programs, such as the probable locations of carries in array multiplication, and the likelihood of floating point units requiring rounding of the result were measured in Chapter 5.

5. New algorithms and implementations of approximate integer multiplication and division for signed and unsigned operands were presented in Chapters 6 and 7, including an analysis of their average correctness for benchmark operands.

6. Approximate integer arithmetic techniques were adapted to floating point hardware in Chapter 8.

7. In Chapter 9 basic result caches were extended with increased associativity and replacement policies, and demonstrated an increase in cache read hit rate.

8. Alternative uses for approximate arithmetic units other than value speculation was demonstrated in Chapter 10 by employing approximate multioperand adders in error-tolerant *LDPC* decoders.

9. The thesis culminated in Chapter 11 with a full system simulation using a range of benchmarks to assess the real-world performance of an out-of-order, superscalar RISC processor employing data value speculation.

12.2 Future Research

Liu and Lu's adder was analysed in Chapter 3, comparing it to a high performance Sklansky adder, showing that the restricted carry length is not a good indicator of the physical properties of the synthesised adder, especially delay. Furthermore, addition is typically a single cycle operation in many processor pipelines, so reducing the system addition cycle latency is not possible. Other approximate prefix adder topologies were proposed, and could be investigated by further research.

Approximate counters were introduced in Chapter 6 and used to construct a family of approximate tree multipliers by homogeneously replacing counters in an exact tree. Each approximate counter truncates one or more sum bits from the exact sum, saving critical path delay, area and power, at the expense of correctness. Each asserted sum bit that was truncated will cause an error in the product. The probability of assertion of each approximate counter output bit is function of many highly dependent relationships between bits in the partial products. A rigorous analysis of these probabilities might suggest heterogeneous approximate multipliers and use different approximate or exact counters to improve the probability of correctness for marginally increased delay.

Chapter 7 presented a modified algorithm for integer division that converged on the precise division quotient in a variable, possibly infinite, number of division rounds, and requiring infinite storage. An error bound was found for the division algorithm after n rounds. An implementation of the division algorithm removed the requirements for an exact result, and possibly introduced an error in the quotient result by setting an upper limit on the number of division rounds and fractional bits maintained. The division algorithm presented required a delay-expensive multiplication, and so the multiplication product was approximated using the most significant partial products, introducing a further source of error. A modification was proposed to allow the implemented dividers to handle signed operands, requiring that negative operands and possibly the integer quotient result are negated. To reduce the delay introduced by negating the d operand, the negation was approximated by complementation. The effects of each approximation introduced are complex and not independent, so the probability of correctness of each divider was determined by the simulation of many benchmark inputs. Further research could determine an error bound for each implemented divider or a mathematical analysis of the correctness of each divider after the introduced quantisation error, etc.

An approximate, iterative division algorithm was developed in Chapter 7. The algorithm takes advantage of the distribution of typical integer operands observed in benchmarks so that as little precision is maintained to calculate a correct inter quotient. Because the algorithm is convergent, and much information is regularly discarded, it is not possible to know with certainty *ahead of time* if the quotient produced is incorrect. Other approximation techniques not using feedback, such as the approximate `intMult` unit could detect a dropped carry early, and signal an error before the result is available. For this reason, future research with approximate integer division could be applied to *probabilistic* computing.

In Chapter 8 An approximate `fpMult` unit was built around an approximate significand multiplier, using the same techniques from Chapter 6. Although some of the multiplier trees obtained a correctness above 90 %, the multiplication trees using the larger counters were slower than the case using more faster, exact counters. A similar approach was taken to construct an approximate floating point divider, but the significand divider was replaced with the approximate integer dividers discussed in Chapter 7. The approximate divider performed poorly, due to the very different distribution of floating point significands compared to integer operands.

The basic approach was taken with both approximate floating point units; the significands were approximated; and the exponent and normalisation calculations were exact. Unlike the exponent calculation, the rounding and normalisation stage are on the critical path, and highly serialised. Future research could consider treating the components of this stage separately in order to approximate some of the functions, if the total calculation time can be reduced below a clock cycle threshold. For example, the carry to the exponent update could be omitted, as it is unlikely that a rounding will cause overflow in the significand (see Figure 8.1c).

In Chapter 9 set associativity and line replacement algorithms were introduced to operand value-indexed result caches, and indexing schemes derived from benchmark data were investigated. It was found that set associative caches improved the hit rate of value-indexed result caches, and could yield gains of over 5 % to IPC. Value indexed result caches must be read later in the pipeline than caches indexed by architectural register or PC, but are able to hit when identical data values are produced by an otherwise identical instruction at another address, or referencing other registers. Further research could investigate if this property can yield a higher hit rate than other result caching schemes after a context switch, where the nature of the code executed, and the PC address range is likely to change.

Rate $^1/_2$ LDPC decoders were constructed in Chapter 10 *Approximate Adders in LDPC* with approximate $(4;2)$ counters introduced into the check nodes, and were shown to reduce the power, area and delay of the multioperand adders used in the LDPC check nodes, and unexpectedly with slightly better decoding performance for the same noise level. Further research can be conducted into extending the degree of approximation used in the counters, such as with $(8;3)$ counters, etc. used in

Chapter 6. Other combinations of approximate counters and decoding rates, such as $^2/_3$, $^3/_4$ and $^5/_6$ are possible. Furthermore, approximation could be introduced into the LDPC variable nodes.

Chapter 11 summarised the synthesis results of arithmetic units optimised for speed, and presented thorough simulations of an out-of-order RISC processor using *ADVS* and result caching. Future research could investigate the latency and correctness tradeoffs at area or power efficient design points. Using custom approximate arithmetic cells could improve the latency and power margins compared to exact arithmetic units. Benchmarking was important to determine the correctness of the arithmetic units, and can be performed for other application domains.

Although many benchmarks were analysed on the modified system, little was done to further modify the system to enhance *ADVS*. In particular, although the system had many of the features of a modern high performance system, the size of hardware blocks and microarchitectural techniques were scaled down from the bleeding edge, such as might be used in a high performance server. In particular, *ADVS* could be investigated on systems with deeper pipelines, or vector arithmetic units.

Finally, there are many potential applications of approximate arithmetic that are tolerant of errors, including audio, video and picture decoding, forecasting and heuristic-based computation.

PART IV

APPENDICES

Appendix A

GCC MAN PAGES

"Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer."

Donald Knuth (1938 —)

This appendix highlights some key features of the *SimpleScalar* compiler used throughout the project. The *SimpleScalar* compiler is based on GCC version 2.6.3.

A.1 *gcc* man pages

The following is extracted from the man pages of the cross-compiler used for *SimpleScalar*, *gcc* version 2.6.3.

A.1.1 OPTIMIZATION OPTIONS

These options control various sorts of optimizations:

-O

-O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without '-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without '-O', only variables declared register are allocated in registers. The resulting compiled code is a little worse than produced by PCC without '-O'.

With '-O', the compiler tries to reduce code size and execution time.

When you specify '-O', the two options '-fthread-jumps' and '-fdefer-pop' are turned on. On machines that have delay slots, the '-fdelayed-branch' option is turned on. For those machines that can support debugging even without a frame pointer, the '-fomit-frame-pointer' option is turned on. On some machines other flags may also be turned on.

-O2 Optimize even more. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done, for example. As compared to -O, this option increases both compilation time and the performance of the generated code.

-O3 Optimize yet more. This turns on everything -O2 does, along with also turning on -finline-functions.

-O0 Do not optimize.

If you use multiple -O options, with or without level numbers, the last such option is the one that is effective.

Appendix B

Source Code

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it. "

Brian Kernighan (1942 —)

This appendix provides the source code to some of the tools written by the candidate, and a selection of benchmarks referenced throughout the thesis. Many benchmarks are common, and are easily available online. Other benchmarks not listed here can be found from references in the Bibliography.

B.1 Probability of correctness

The *BackCount* algorithm was developed to accurately count the number of cases in N bit addition that a carry chain of longer that l bits would be generated. The algorithm subtracts the number of cases from the total number of possible 2^N additions, and derives the exact probability of correctness. The carry lengths are progressively shortened until l bits are reached. Every possible placement of an i bit carry chain are calculated using partitions of N, and combinatorics. Hence, the number of placements increases exponentially for small l, and the running time increases dramatically. This is of little consequence, as the expected probability of correctness for $l < \log_2(N)$ is $< 50\,\%$.

The source code for *BackCount* was coded in *MATLAB*, and was written to provide an exact result, rather than the complex equations that yield lower bounds on the correctness.

B.1.1 *backCount.m*

```
% Return the number of incorrect sums for an N-bit adder, with maximum
% carry length of k-bits.
function [count,table] = backCount(N,k,table,s)

s = [s '  '];

if N<=k
    count = 0;
    return;
end

if k==0
    count = 4^N - 3^N;
    return
end

if table(N,k)~=-1
    count = table(N,k);
    return;
end

count = 0;
% consider strings of length N ... (k+1)
for len = N:-1:(k+1)

    if len > N/2
        smallLen = false;
    else
```

```
        smallLen = true;
    end

    % consider each way of having a len-string
    for i = 1:(N-len+1)
        % (a) places on the left of this string
        a = i-1;

        % (b) places right of this string
        b = N-len-i+1;

        % (c) non-propagate after gpp string
        if b>0
            c = 1;
        else
            c = 0;
        end

        % (d) number of places after gpp string terminator (c)
        if b >= 1
            d = b-1;
        else
            d = 0;
        end

        % count the ways of having this as the longest string
        if smallLen
            % leftGEi: errors on the left with errors >= len
            if a<len
                leftErrors = 4^a;
            elseif a>=len
                leftErrors = 4^(a)-backCount(a,len-1,table,s);
            elseif a==0
                leftErrors = 1;
            else
                error(’Impossible case (b)’);
            end

            % errors to the right with length >= len
            if b<len
                rightErrors = 2^(c)*4^(d);
            elseif b>=len
                rightErrors = 2^(c)*4^(d) -fixGenRight(b,len-1,table,s);
            elseif b==0
                rightErrors = 1;
            else
                error(’Impossible case (a)’);
            end

            % all combinaions of errors of length len to the right of i
            precount = count;
            errors = 2^(len-1)*leftErrors*rightErrors;
            count = count + errors;

        else
            % (not smallLen)
            precount = count;
            errors = 2^(len-1)*4^(a+d)*2^(c);
            count = count + errors;
        end

    end

    % consider having multiple length blocks
    if len <= N/2 && len > k
        overlaps = overlapComb(N,k,len,table,s);
        count = count + overlaps;
    end

    % write-back to the table for easy reference
    table(N,len) = count;
end
```

B.1.2 *fixGenRight.m*

```

% a routine for the combinations to the right, with first string a generate
function err = fixGenRight(N,k,table,s)

err = 0;
% trivial case
if k>=N
    return;
end

% consider all gpp strings starting at first bit > k
for i = (k+1):N
    if N-i > 0
```

```
        c = 1;
    else
        c = 0;
    end
    if N-i-1 > 0
        d = N-i-1;
    else
        d = 0;
    end
%     err = err + 2^(i-1)*2^(c)*4^(d);%*backCount(N-i,k,table,s);
    err = err + 2^(i-1)*( 2^(c)*4^(d) - fixGenRight(N-i,k,table,s));%*backCount(N-i,k,table,s);
end
% count the remaining errors on the right, ignoring the first bit
err = err + 2*backCount(N-1,k,table,s); % CHECK THIS... 2 or 4? I'll go with 2, thankyou....
```

B.2 Logic approximation

logicApprox is a program to automate the generation of logically incomplete circuits, by minimising the number of minterms in the canonical representation of the approximate function, with provided error bounds. The user determines the maximum number of input combinations for which the output can be asserted differently, which sets the maximum probability of error. The assertions can be automatic using the **-a** flag, or forced to be positive (forced to 1) or negative (forced to zero) using the **-p** and **-n** flags respectively.

The output is printed in the canonical form, formatted for the *Espresso* logic minimiser to condense to a more efficient representation. *logicApprox* was written in *C*, and is provided below.

B.2.1 *approx.c*

```
#include "approx.h"

int main(int argc, char *argv[])
{

  trace = 0;
  /* trace */
  if (trace) {
    printf("%s\n", "main()");
    printf("%s%s\n", "espresso file: ", argv[argc-1]);
  }

  extern int posAssert;
  extern int negAssert;
  extern int totalAssert;
  extern int fileOutput;
  posAssert = negAssert = totalAssert = 0;
  fileOutput = 0;
  int i = 0;
  if (argc==2 && strncmp(argv[1], "-h", 2)!=0) {
    printf("\nYou must specify a maximum number of assertions\n"
           "or every function will be approximated as true.\n"
           "For help try ‘approx -h’\n\n");
    return 1;
  }
  else if (argc==2 && strncmp(argv[1], "-h", 2)==0) {
    printHelp();
  }

  /* this will need some serious extra error handling */
  for (i=0; i<argc-1; i++) {
    if (strncmp(argv[i], "-h", 2)==0) {
      i = argc-2;
      printHelp();
    }
    else if (strncmp(argv[i], "-p", 2)==0) {
      posAssert = atoi(argv[i+1]);
      i++;;
    }
    else if (strncmp(argv[i], "-n", 2)==0) {
      negAssert = atoi(argv[i+1]);
      i++;;
    }
    else if (strncmp(argv[i], "-a", 2)==0) {
      totalAssert = atoi(argv[i+1]);
      i++;;
    }
    else if (strncmp(argv[i], "-o", 2)==0) {
      extern char* outputFile;
      outputFile = argv[i+1];
      fileOutput = 1;
      i++;;
    }
  }

  /* interpret ‘-p x’ as ’-p x -a x’ */
  if (totalAssert==0) {
    if (posAssert==0 && negAssert!=0) {
      totalAssert = negAssert;
    }
    else if (negAssert==0 && posAssert!=0) {
      totalAssert = posAssert;
    }
  }

  /* interpret ’-a x’ as ’-p x -n x -a x’ */
  if (totalAssert!=0) {
    if (negAssert==0 && posAssert==0) {
      posAssert = totalAssert;
      negAssert = totalAssert;
    }
  }

  /* trace */
  if (trace) {
    printf("%s%i%s%i%s%i\n", "posAssert = ", posAssert,  \
           "\nnegAssert = ", negAssert, "\ntotalAssert = ",       \
           totalAssert);
    if (fileOutput) {
      printf("%s%s\n", "writing output to file: ", outputFile);
    }
    else {
      printf("Not printing to a file.. brace for stdout spam...\n");
    }
  }

  /* populate the terms struct */
  if (readFile(argv[argc-1]) != 0)
  {
    printf("%s%s\n", "Error reading file ", argv[argc-1]);
    printf("%s\n","Usage: approx <options> <espresso_output_file>");
    return 1;
  }

  /* trace */
  if (trace) {
    printf("%s%i\n", "numSignals: ", numSignals);
    printf("%s%i\n", "numFunctions: ", numFunctions);
  }

  /* set up an array, so that all outputs can be reduced equally */
  extern int *pos, *neg, *all;
  pos = malloc(numFunctions*sizeof(int));
  neg = malloc(numFunctions*sizeof(int));
  all = malloc(numFunctions*sizeof(int));


  /* Populate the arrays of reductions */
  for (i=0; i<numFunctions; i++) {
    all[i] = totalAssert;
    neg[i] = negAssert;
    pos[i] = posAssert;
  }

  /* parameters set up, run algorithm */
  int reset = 1;
  do {
    reset = approximate();
  } while (reset!=0);

  /* print the function: stdout or outputFile */
  printOutput();

  return 0;
}
```

B.2.2 *approx.h*

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <gsl/gsl_combination.h>

/* Trace variable */
/* this is poor form, get rid of this before code release */
int trace;

/* Global variables */
int posAssert;
int negAssert;
int totalAssert;
int fileOutput;
char *outputFile;

int *pos;
int *neg;
int *all;

char *inVarCnt;
char *outVarCnt;
char *inVarNames;
char *outVarNames;

int * fnTerms;
int *initFnTerms;
int numSignals;
int numFunctions;

int ***cell;

int *termLog;
int *fnLog;
int totalNumTerms;

int *similar;
int *diff1;
int *diff2;
int *diff3;
int *diff4;

/* Function prototypes */
int readFile(char *);
void printHelp(void);
int approximate(void);
void printOutput(void);
int signalCnt(int *term);
int reduce_a(int fnNum);
int reduce_b(int fnNum);
int reduce_c(int fnNum);
int reduce_d(int fnNum);
int reduce_e(int fnNum);
int reduce_f(int fnNum);
int reduce_g(int fnNum);
void stalemate_heuristic(void);
void compare2Terms(int fnNum, int term_a, int term_b);
void compare3Terms(int fnNum, int term_a, int term_b, int term_c);
void compare4Terms(int fnNum, int term_a, int term_b, int term_c, int term_d);
int findDiffSig(int *term, int index);
int similarCnt(void);
void copyTerm(int *out, int *in);
void copySimilar(int *dest, int *source);
void convEspresso();
int checkEspressoTerm(int *term, int numLines, int funcNum);
char *getEspOutput();
```

B.2.3 *reduce.c*

```c
#include "approx.h"

void printTerms(void);

/* Performs the logic reduction algorithm.
   Data structure is an extern variable.
   In: Nothing
   Out: Nothing
*/
int approximate(void) {

  /* trace */
  if (trace) {
    printf("approximate()\n");
  }
```

```

extern int *similar;
extern int *diff1;
extern int *diff2;
extern int *diff3;
extern int *diff4;

int initialAssert = totalAssert;

similar = NULL;
diff1 = NULL;
diff2 = NULL;
diff3 = NULL;
diff4 = NULL;

int i = 0, reset = 0;

/* trace */
if (trace) {
  printf("\t - start a new round -\n");
  printOutput();
}

for (i=0; i<numFunctions; i++) {
  if (all[i]>0 && pos[i]>0 && !reset) {
    reset = reduce_a(i);

    /* trace */
    if (trace) {
      printOutput();
    }
    if (reset==1)
      return reset;

  }
}

if (!reset) {
  for (i=0; i<numFunctions; i++) {
    if (all[i]>0 && neg[i]>0 && !reset) {
      reset = reduce_b(i);

      /* trace */
      if (trace) {
        printOutput();
      }
      if (reset==1)
        return reset;
    }
  }
}

if (!reset) {
  for (i=0; i<numFunctions; i++) {
    if (all[i]>0 && pos[i]>0 && !reset) {
      reset = reduce_c(i);

      /* trace */
      if (trace) {
        printOutput();
      }
      if (reset==1)
        return reset;
    }
  }
}

if (!reset) {
  for (i=0; i<numFunctions; i++) {
    if (all[i]>0 && neg[i]>0 && !reset) {
      reset = reduce_d(i);

      /* trace */
      if (trace) {
        printOutput();
      }
      if (reset==1)
        return reset;
    }
  }
}

if (!reset) {
  for (i=0; i<numFunctions; i++) {
    if (all[i]>0 && pos[i]>0 && !reset) {
      reset = reduce_e(i);

      /* trace */
      if (trace) {
        printOutput();
      }
    }
  }
}

```

```c
  /* stalemate_heuristic */
  if (totalAssert==initialAssert)
    stalemate_heuristic();


  free(similar);
  free(diff1);
  free(diff2);
  free(diff3);
  free(diff4);

  return 0;
}


/* Compares two terms in the cell for a given function.
   Arrays of structs are written to show differences, similarities.
   In: The function number to compare, and two term numbers.
   Out: Nothing
*/
void compare2Terms(int fnNum, int term_a, int term_b) {

  /* trace */
  if (trace) {
    printf("compare2Terms()\n");
  }

  int i;
  extern int *similar;
  extern int *diff1;
  extern int *diff2;
  extern int numSignals;

  /* trace */
  if (trace) {
    printf("free mem...\n");
  }

  free(similar);
  free(diff1);
  free(diff2);

  /* trace */
  if (trace) {
    printf("mallocs...\n");
  }

  similar = (int *)malloc(numSignals * sizeof(int));
  diff1 = (int *)malloc(numSignals * sizeof(int));
  diff2 = (int *)malloc(numSignals * sizeof(int));

  /* trace */
  if (trace) {
    printf("done mallocs...\n");
  }

  for (i=0; i<numSignals; i++) {
    if (cell[fnNum][term_a][i] == cell[fnNum][term_b][i]) {
      similar[i] = 1;
      diff1[i] = -1;
      diff2[i] = -1;
    }
    else {
      similar[i] = 0;
      diff1[i] = cell[fnNum][term_a][i];
      diff2[i] = cell[fnNum][term_b][i];
    }
  }
}


/* Compares three terms in the cell for a given function.
   Arrays of structs are written to show differences, similarities.
   Comparisons are only made between (a,b) & (a,c), not (b,c).
   In: The function number to compare, and the indices of the three terms.
   Out: Nothing
*/
void compare3Terms(int fnNum, int term_a, int term_b, int term_c) {

  /* trace */
  if (trace) {
    printf("compare3Terms()\n");
  }

  int i;
  extern int *similar;
  extern int *diff1;
  extern int *diff2;
  extern int *diff3;
  extern int numSignals;

  free(similar);
  free(diff1);
  free(diff2);
  free(diff3);

  similar = (int *)malloc(numSignals * sizeof(int));
  diff1 = (int *)malloc(numSignals * sizeof(int));
  diff2 = (int *)malloc(numSignals * sizeof(int));
  diff3 = (int *)malloc(numSignals * sizeof(int));

  for (i=0; i<numSignals; i++) {
     if (cell[fnNum][term_a][i] == cell[fnNum][term_b][i] \
        && cell[fnNum][term_a][i] == cell[fnNum][term_c][i] ) {
      similar[i] = 1;
      diff1[i] = -1;
      diff2[i] = -1;
      diff3[i] = -1;
    }
    else {
      similar[i] = 0;
      diff1[i] = cell[fnNum][term_a][i];
      diff2[i] = cell[fnNum][term_b][i];
      diff3[i] = cell[fnNum][term_c][i];
    }
  }
}


/* Compares four terms in the cell for a given function.
   Arrays of structs are written to show differences, similarities.
   Differences are the terms different in the comparison of (a,b), (a,c) &
   (a,d) i.e., there is no comparison between (b,c), (b,d) & (c,d).
   In: The function number to compare, and index of the four terms.
   Out: Nothing
*/
void compare4Terms(int fnNum, int term_a, int term_b, int term_c, int term_d) {

  /* trace */
  if (trace) {
    printf("compare4Terms()\n");
  }

  int i;
  extern int *similar;
  extern int *diff1;
  extern int *diff2;
  extern int *diff3;
  extern int *diff4;
  extern int numSignals;

  free(similar);
  free(diff1);
  free(diff2);
  free(diff3);
  free(diff4);

  similar = (int *)malloc(numSignals * sizeof(int));
  diff1 = (int *)malloc(numSignals * sizeof(int));
  diff2 = (int *)malloc(numSignals * sizeof(int));
  diff3 = (int *)malloc(numSignals * sizeof(int));
  diff4 = (int *)malloc(numSignals * sizeof(int));

  for (i=0; i<numSignals; i++) {
    if (cell[fnNum][term_a][i] == cell[fnNum][term_b][i] \
        && cell[fnNum][term_a][i] == cell[fnNum][term_c][i] \
        && cell[fnNum][term_a][i] == cell[fnNum][term_d][i] ) {
      similar[i] = 1;
      diff1[i] = -1;
      diff2[i] = -1;
      diff3[i] = -1;
      diff4[i] = -1;
    }
    else {
      similar[i] = 0;
      diff1[i] = cell[fnNum][term_a][i];
      diff2[i] = cell[fnNum][term_b][i];
      diff3[i] = cell[fnNum][term_c][i];
      diff4[i] = cell[fnNum][term_d][i];
    }
  }
}


/*Reduction method (a)
  (positive assertion)
  (w + x + y + z).common -> 1.common
  In: nothing
  Out: nothing
*/
int reduce_a(int fnNum) {

  /* trace */
  if (trace) {
    printf("\t\t-:reduce_a():-\n");
  }

  int term1;
  int term2;
  int term3;
  int term4;

  extern int *fnTerms;
  extern int *diff1;
  extern int *diff2;
  extern int *diff3;
  extern int *diff4;

  int numTerms;
  numTerms = fnTerms[fnNum];

  /* this section can be very expensive, abort if too many combinations... */
  if (numTerms >= 20) {
    return 0;
  }

  if (numTerms >= 4) {
    /* trace */
    if (trace) {
      printf("new combination time...\n");
      printf("new combination time...\n");
    }

    gsl_combination * comb = gsl_combination_calloc(numTerms,4);
    do  {
      term1 = gsl_combination_data(comb)[0];
      term2 = gsl_combination_data(comb)[1];
      term3 = gsl_combination_data(comb)[2];
      term4 = gsl_combination_data(comb)[3];
      compare4Terms(fnNum, term1, term2, term3, term4);
      /* trace */
      if (trace) {
        printf("%s%i", "signalCnt(diff1) = ", signalCnt(diff1));
        printf("%s%i", "signalCnt(diff2) = ", signalCnt(diff2));
        printf("%s%i", "signalCnt(diff3) = ", signalCnt(diff3));
        printf("%s%i", "signalCnt(diff4) = ", signalCnt(diff4));
      }

      if (signalCnt(diff1)==1 && signalCnt(diff2)==1 && signalCnt(diff3)==1 \
          && signalCnt(diff4)==1) {

        /* sufficient conditions to reduce logic */
        /* don't overwrite other terms */
        if (cell[fnNum][fnTerms[fnNum]-1]!=cell[fnNum][term1]) {
          memcpy(cell[fnNum][term1],            \
                 cell[fnNum][fnTerms[fnNum]-1], \
                 sizeof(int *)                  \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-2]!=cell[fnNum][term2]) {
          memcpy(cell[fnNum][term2],            \
                 cell[fnNum][fnTerms[fnNum]-2], \
                 sizeof(int *)                  \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-3]!=cell[fnNum][term3]) {
          memcpy(cell[fnNum][term3],            \
                 cell[fnNum][fnTerms[fnNum]-3], \
                 sizeof(int *)                  \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-4]!=cell[fnNum][term4]) {
          memcpy(cell[fnNum][term4],            \
                 cell[fnNum][fnTerms[fnNum]-4], \
                 sizeof(int *)                  \
                 );
        }
        /* perform the reduction */
        memset(cell[fnNum][fnTerms[fnNum]-4], -1, numSignals*sizeof(int));
        fnTerms[fnNum] -= 3;
        pos[fnNum]--;
        all[fnNum]--;
        /* reset the reduction procedure */
        return 1;
      }
    } while (gsl_combination_next (comb) == GSL_SUCCESS);
  }
  return 0;
}


/*Reduction method (b)
  (negative assertion)
  (wxy + z).common -> z.common
  In: nothing
  Out: nothing
*/
int reduce_b(int fnNum) {

  /* trace */
  if (trace) {
    printf("\t\t-:reduce_b():-\n");
  }

  int term1;
  int term2;

  int numTerms = fnTerms[fnNum];
  /* trace */
  if (trace) {
    printf("%s%i\n", "numTerms: ", numTerms);
  }

  if (numTerms>=2) {
    gsl_combination * comb = gsl_combination_calloc(numTerms,2);
    do {
      term1 = gsl_combination_data(comb)[0];
      term2 = gsl_combination_data(comb)[1];
      compare2Terms(fnNum, term1, term2);

      if ((signalCnt(diff1)==1 && signalCnt(diff2)==3) ||       \
          (signalCnt(diff1)==3 && signalCnt(diff2)==1) ) {
        /* both term have the correct num of signals, and no differences
         sufficeint conditions to reduce
        */
        int *temp = malloc(numSignals*sizeof(int));
        if (signalCnt(diff1)==1)
          memcpy(temp, cell[fnNum][term1], numSignals*sizeof(int));
        else
          memcpy(temp, cell[fnNum][term2], numSignals*sizeof(int));

        if (cell[fnNum][term1]!=cell[fnNum][fnTerms[fnNum]-1]) {
          memcpy(cell[fnNum][fnTerms[fnNum]-1],     \
                 cell[fnNum][term1],                \
                 sizeof(int *)                      \
                 );
        }
        if (cell[fnNum][term2]!=cell[fnNum][fnTerms[fnNum]-2])
          memcpy(cell[fnNum][fnTerms[fnNum]-2],     \
                 cell[fnNum][term2],                \
                 sizeof(int *)                      \
                 );
        memcpy(cell[fnNum][fnTerms[fnNum]-2], temp, numSignals*sizeof(int));
        free(temp);
        fnTerms[fnNum]--;
        neg[fnNum]--;
        all[fnNum]--;
        return 1;
      }
    } while(gsl_combination_next (comb) == GSL_SUCCESS);
  }
  return 0;
}


/*Reduction method (c)
  (positive assertion)
  (wx + wy + wz).common -> w.common
  In: nothing
  Out: nothing
*/
int reduce_c(int fnNum) {

  /* trace */
  if (trace) {
    printf("\t\t-:reduce_c():-\n");
  }

  int numTerms = fnTerms[fnNum];
  int term1;
  int term2;
  int term3;
  int *temp;

  if (numTerms>=3) {
    gsl_combination *comb = gsl_combination_calloc(numTerms,3);
    do {
      term1 = gsl_combination_data(comb)[0];
      term2 = gsl_combination_data(comb)[1];
      term3 = gsl_combination_data(comb)[2];
      compare3Terms(fnNum, term1, term2, term3);
      if (signalCnt(diff1)==1 && signalCnt(diff2)==1    \
          && signalCnt(diff3)==1 ) {
        temp = malloc(numSignals*sizeof(int));
        memcpy(temp, cell[fnNum][term1], numSignals*sizeof(int));

        /* all have exactly one similar term (plus common) */
        if (cell[fnNum][fnTerms[fnNum]-2]!=cell[fnNum][term2]) {
          memcpy(cell[fnNum][term2],                    \
                 cell[fnNum][fnTerms[fnNum-2]],         \
                 numSignals*sizeof(int)                 \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-1]!=cell[fnNum][term1]) {
          memcpy(cell[fnNum][term1],                    \
                 cell[fnNum][fnTerms[fnNum]-1],         \
                 numSignals*sizeof(int)                 \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-3]!=cell[fnNum][term3]) {
          memcpy(cell[fnNum][term3],                    \
                 cell[fnNum][fnTerms[fnNum]-3],         \
                 numSignals*sizeof(int)                 \
                 );
        }
        memcpy(cell[fnNum][fnTerms[fnNum]-3], temp, numSignals*sizeof(int));
        free(temp);

        /* get rid of the differnce */
        cell[fnNum][fnNum[fnTerms]-3][findDiffSig(diff1, 0)] = -1;
        fnTerms[fnNum] -= 2;
        pos[fnNum]--;
        all[fnNum]--;
        return 1;
      }
    } while(gsl_combination_next (comb) == GSL_SUCCESS);
  }

  return 0;
}


/*Reduction method (d)
  (negative assertion)
  (wyz + wx).common -> wx.common
  In: nothing
  Out: nothing
*/
int reduce_d(int fnNum) {

  /* trace */
  if (trace) {
    printf("\t\t-:reduce_d():-\n");
  }

  int term1;
  int term2;
  int *temp = malloc(numSignals*sizeof(int));
  int reduce = 0;

  int numTerms = fnTerms[fnNum];
  if (numTerms>=2) {
    gsl_combination * comb = gsl_combination_calloc(numTerms,2);

    do {
      term1 = gsl_combination_data(comb)[0];
      term2 = gsl_combination_data(comb)[1];
      compare2Terms(fnNum, term1, term2);

      if (signalCnt(diff1)==1 &&                \
          signalCnt(diff2)==2) {
        memcpy(temp, cell[fnNum][term1], numSignals*sizeof(int));
        reduce = 1;
      }
      else if (signalCnt(diff1)==2 &&   \
               signalCnt(diff2)==1 ) {
        memcpy(temp, cell[fnNum][term2], numSignals*sizeof(int));
        reduce = 1;
      }

      /* conditions satisifed to reduce */
      if (reduce!=0 && similarCnt()>0) {
        if (cell[fnNum][fnTerms[fnNum]-1]!=cell[fnNum][fnTerms[fnNum]-1]) {
          memcpy(cell[fnNum][term1],                    \
                 cell[fnNum][fnTerms[fnNum]-1],         \
                 numSignals*sizeof(int)                 \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-2]!=cell[fnNum][fnTerms[fnNum]-2]) {
          memcpy(cell[fnNum][term2],                    \
                 cell[fnNum][fnTerms[fnNum]-2],         \
                 numSignals*sizeof(int)                 \
                 );
        }
        /* reduction operation */
        memcpy(cell[fnNum][fnTerms[fnNum]-2],           \
               temp,                                    \
               numSignals*sizeof(int)                   \
               );
        fnTerms[fnNum]--;
        neg[fnNum]--;
        all[fnNum]--;
        free(temp);
        return 1;
      }
    }  while(gsl_combination_next (comb) == GSL_SUCCESS);
  }
  free(temp);
  return 0;
}


/*Reduction method (e)
  (positive assertion)
  (wyz + wxy).common -> wy.common
  In: nothing
  Out: nothing
*/
int reduce_e(int fnNum) {

  /* trace */
  if (trace) {
    printf("\t\t-:reduce_e():-\n");
  }

  int term1;
  int term2;

  int numTerms = fnTerms[fnNum];
  if (numTerms>=2) {
    gsl_combination * comb = gsl_combination_calloc(numTerms,2);
    do {
      term1 = gsl_combination_data(comb)[0];
      term2 = gsl_combination_data(comb)[1];
      compare2Terms(fnNum, term1, term2);

      if (signalCnt(diff1)==1 && signalCnt(diff2)==1) {
        /* sufficient requirements for reduction */
        int *temp = malloc(numSignals*sizeof(int));
        memcpy(temp, cell[fnNum][term1], numSignals*sizeof(int));

        if (cell[fnNum][fnTerms[fnNum]-1]!=cell[fnNum][term2]) {
          memcpy(cell[fnNum][term2],           \
                 cell[fnNum][fnTerms[fnNum]-1], \
                 numSignals*sizeof(int) \
                 );
        }
        if (cell[fnNum][fnTerms[fnNum]-2]!=cell[fnNum][term2]) {
          memcpy(cell[fnNum][term2],           \
                 cell[fnNum][fnTerms[fnNum]-2], \
                 numSignals*sizeof(int)  \
                 );
        }
        copySimilar(cell[fnNum][fnTerms[fnNum]-2], temp);
        fnTerms[fnNum]--;
        pos[fnNum]--;
        all[fnNum]--;
        return 1;
      }
    } while(gsl_combination_next (comb) == GSL_SUCCESS);
  }

  return 0;
}


/*stalemate_heuristic
  Used to transform large xor structures into other logic if
  many spare assertions exist.
  In: nothing
  Out: nothing
*/
void stalemate_heuristic(void) {

  /* trace */
  if (trace) {
    printf("stalemate_heuristic()\n");
  }

  /* Not currently implemented */
}


/*Count the number of (valid) signals in a term.
  In: Pointer to the array of signals
  Out: Count of signals in the term
*/
int signalCnt(int *term) {

  /* trace */
  if (trace) {
    printf("%s\n", "signalCnt()");
  }

  extern int numSignals;
  int i = 0;
  int cnt = 0;

  for (i=0; i<numSignals; i++) {

    /* trace */
    if (trace) {
      printf("%s%i%s%i\n", "\tsignal[", i, "] = ", term[i]);
    }

    if (term[i]!=-1)
      cnt++;
  }
  return cnt;
}
```

```


/*Look through a diff term to find the first occurance of a signal which is
  not invalid (not -1), starting at 'index'.
  In: Pointer to the diff term (array of signals)
  Out: Index of the first occuring valid signal, return -1 if no such signal
  occurs.
*/
int findDiffSig(int *term, int index) {

  /* trace */
  if (trace) {
    printf("findDiffSig()\n");
  }

  extern int numSignals;
  int i;
  for (i=index; i<numSignals; i++) {
    if (term[i]!=-1)
      return i;
  }
  return -1;
}


/*Look through a diff term to find the first occurance of a signal which is
  not invalid (not 0), starting at 'index'.
  In: Pointer to the diff term (array of signals)
  Out: Index of the first occuring valid signal, return -1 if no such signal
  occurs.
*/
int findSimilarSig(int *term, int index) {

  /* trace */
  if (trace) {
    printf("findSimilarSig()\n");
  }

  extern int numSignals;
  int i;
  for (i=index; i<numSignals; i++) {
    if (term[i]!=-1)
      return i;
  }
  return -1;
}


/*Count the number of similarities between two terms.
  In: Pointer to the similar term
  Out: Count of similarities.
*/
int similarCnt() {

  /* trace */
  if (trace) {
    printf("similarCnt()\n");
  }

  extern int numSignals;
  int i=0, cnt=0;
  for (i=0; i<numSignals; i++) {
    if (similar[i]==1)
      cnt++;
  }
  return cnt;
}


/*Copy the contents of term IN to term OUT.
  In: Pointers to IN and OUT.
  Out: VOID
*/
void copyTerm(int *out, int *in) {

  /* trace */
  if (trace) {
    printf("copyTerm()\n");
  }

  int i;
  for (i=0; i<numSignals; i++)
    out[i] = in[i];

  return;
}


/*Copy only the similarites of a term at index to the destination.
  In: fnNum and termIndex to indentify the term,
  as well as poiner to destination
  Out: VOID
*/
void copySimilar(int *dest, int *source) {

  /* trace */
  if (trace) {
    printf("copySimilar()\n");
  }

  int i;
  for (i=0; i<numSignals; i++) {
    if (similar[i]==1)
      dest[i] = source[i];
    else
      dest[i] = -1;
  }
}





void printTerms(void) {
  int i;
  printf("\t%s", "all = [ ");
  for (i=0; i< numFunctions; i++) {
    printf("%i%s", all[i], " ");
  }
  printf("]\n");
  printf("\t%s", "pos = [ ");
  for (i=0; i< numFunctions; i++) {
    printf("%i%s", pos[i], " ");
  }
  printf("]\n");
  printf("\t%s", "all = [ ");
  for (i=0; i< numFunctions; i++) {
    printf("%i%s", neg[i], " ");
  }
  printf("]\n\n");
}
```

B.2.4 *io.c*

```
#include "approx.h"

/* Prints the help for the command line.
   In: Nothing
   Out: Nothing
*/
void printHelp(void) {

  /* trace */
  if (trace) {
    printf("printHelp()\n");
  }

  printf("\n%s\n\n","APPROX: A logic approximation program");
  printf("%s\n\n", "Approx simplifies the logic for a given function, or logic unit\n"
         "with multiple outputs (functions) by inferring a limited number\n"
         "of errors to a circuit. The repesentation will reduce the number\n"
         "of AND/OR gates in the function representations. The input file\n"
         "suppied must be a BASIC espresso output file, *and is assumed\n"
         "to be minimised*.");
  printf("%s\n","Usage: approx <options> <espresso_file>");
  printf("\n%s\n","\toptions\t\tdescription");
  printf("%s\n","\t-------\t\t-----------");
  printf("%s\n", "help\t-h \t\tturns out you already found it!");
  printf("%s\n", "+ve\t-p <num>\tthe max number of positive assertions");
  printf("%s\n", "-ve\t-n <num>\tthe max number of negative assertions");
  printf("%s\n", "all\t-a <num>\tthe total maximum number of assertions");
  printf("%s\n\n", "output\t-o <filename>\tthe output file, default stdout");
  printf("%s\n\n", "e.g., to have 10 assertions, but no more than 8 of either,\n"
         "then use options ‘-p 9 -n 8 -a 10’\n"
         "\n"
         "It is not necessary to specify a total number of assertions\n"
         "if all the assertions you want are to be of one type or another\n"
         "(just use ‘-p 10’ instead of ‘-p 10 -a 10’)" );
}

/* Reads a string input starting at pos,.
   In: espresso output filename
   Out: boolean (success=>0, failure=>!0)
*/
int readFile(char *espFile) {

  /* trace */
  if (trace) {
    printf("%s\n", "readFile()");
  }

  FILE *fp;
  int lineLength = 128;
```

```c
char line[lineLength];

extern char *inVarCnt;
extern char *outVarCnt;
extern char *inVarNames;
extern char *outVarNames;

fp = fopen(espFile, "r");
/* trace */
if (trace) {
  printf("%s%i\n", "fp = ", (int)fp);
}

if (fp == NULL)
  return 1;

/* determine the number of varibles and outputs */
/* determine thh number of functions in the file */
extern int numSignals;
extern int numFunctions;
char  space[] = " ";
char *token;

/* inputs */
fgets(line, lineLength, fp);
/* trace */
if (trace) {
  printf("%s%s", "read from file: ", line);
}

inVarCnt = malloc(strlen(line)+1);
strcpy(inVarCnt, line);

if (strcmp(token = strtok(line, space),".i")!=0) {
  printf("%s\n", "Error. ‘.i’ required in line 1.");
  return 1;
}
/* trace */
if (trace) {
  printf("%s%s\n%s%s\n","line: ",line,"token: ",token);
}

token = strtok(NULL, space);
/* trace */
if (trace) {
  printf("%s%s","token: ",token);
}

numSignals = atoi(token);
/* trace */
if (trace) {
  printf("%s%i\n","Value read from file: .i ",numSignals);
}

/* Functions */
fgets(line, lineLength, fp);
/* trace */
if (trace) {
  printf("%s%s", "read from file: ", line);
}

outVarCnt = malloc(strlen(line)+1);
strcpy(outVarCnt, line);

if (strcmp(token = strtok(line, space),".o")!=0) {
  printf("%s\n", "Error. ‘.o’ required in line 2.");
  return 1;
}
/* trace */
if (trace) {
  printf("%s%s\n%s%s\n","line: ",line,"token: ",token);
}

token = strtok(NULL, space);
/* trace */
if (trace) {
  printf("%s%s","token: ",token);
}

numFunctions = atoi(token);
/* trace */
if (trace) {
  printf("%s%i\n","Value read from file: ",numFunctions);
}

/* The remaining lines indicates the number of terms for
   each function.
*/
extern int *fnTerms;
fnTerms = calloc(numFunctions, sizeof(int)/4);
char fileTerm[] = ".e";
char tempStr[2];
int i;
int fnTok = 0;

```

```
fgets(line, lineLength, fp);
inVarNames = malloc(strlen(line)+1);
strncpy(inVarNames, line, strlen(line));

fgets(line, lineLength, fp);
outVarNames = malloc(strlen(line)+1);
strncpy(outVarNames, line, strlen(line));

fgets(line, lineLength, fp);
fgets(line, lineLength, fp);

/* zero out the array */
for (i=0; i<numFunctions; i++) {
  fnTerms[i] = 0;
}

while(strncmp(line, fileTerm, 2)!=0) {
  /* the second token per line indicates the functions */
  token = strtok(line, space);
  token = strtok(NULL, space);
  /* trace */
  if (trace) {
    printf("%s%s","token: ", token);
  }

  for (i=0; i<strlen(token)-1; i++) {
    /* trace */
    if (trace) {
      printf("%s%c\n", "  char: ", token[i]);
    }

    strncpy(tempStr,token+i,1);
    /* trace */
    if (trace) {
      printf("%s%s\n", "tempStr: ", tempStr);
    }

    fnTok = atoi(tempStr);
    if (fnTok==1)
      fnTerms[i]++;
    /* trace */
    if (trace) {
      printf("%s%i%s%i\n", "    fnTerms[", i, "] = ", fnTerms[i]);
    }

  }
  if (strncmp(line, fileTerm, 2)!=0)
    fgets(line, lineLength, fp);
  /* trace */
  if (trace) {
    printf("%s%s\n", "line: ", line);
  }
}

/* trace */
if (trace) {
  printf("%s\n", "closing the file...");
}

fclose(fp);

/* create the data structure */
extern int ***cell;
/* a term is an array of signal structs */
int j = 0;
int temp = 0;

/* trace */
if (trace) {
  printf("%s\n", "ready to initalise data structure...");
}

cell = (int ***)calloc(numFunctions, sizeof(int **));

/* trace */
if (trace) {
  printf("%s\n", "cell created!");
}

for (i=0; i<numFunctions; i++) {
  /* trace */
  if (trace) {
    printf("%s%i%s\n", "cell [", i, "] ...");
  }

  temp = (int)calloc(fnTerms[i], sizeof(int *));
  if (temp!=(int)NULL){
    cell[i] = (int **)temp;
    }
  else {
    printf("%s\n\n","Memory allocation failure.");
    return 1;
  }
  /* trace */
  if (trace) {
    printf("%s\n", "    ... created");
  }

  for (j=0; j<fnTerms[i]; j++) {
    /* trace */
    if (trace) {
      printf("%s%i%s%i%s%i%s\n", "cell [", i, "][", j, "]... (there are ", \
             fnTerms[i], " terms in this function...)");
    }

    temp = (int)calloc(numSignals, sizeof(int));
    if (temp!=(int)NULL)
      cell[i][j] = (int *)temp;
    else {
      printf("%s\n\n","Memory allocation failure.");
      return 1;
    }
    /* trace */
    if (trace) {
      printf("%s\n", "    ... created");
    }
  }
}

/* trace */
if (trace) {
  printf("fnTerms = [ ");
  for (i=0; i<numFunctions; i++)
    printf("%i%s", fnTerms[i], " ");
  printf("]\n");
}

/* populate the data structure */
char *fnToken;
char *termToken;
int sigNum = 0;
/* a temporary working copy of fnTerms */
int *terms = malloc(numFunctions*sizeof(int));
memcpy(terms, fnTerms, sizeof(int)*numFunctions);

/* trace */
if (trace) {
  printf("Re-open the file to populate the data.. \n");
}

fp = fopen(espFile, "r");
if (fp == NULL)
  return 1;
for (i=0; i<6; i++)
  fgets(line, lineLength, fp);
/* trace */
if (trace) {
  printf("%s%s\n", "The first function line read is.. ", line);
}

while (strncmp(line, ".e", 2)!=0) {
  termToken = strtok(line, " ");
  fnToken = strtok(NULL, " ");
  /* trace */
  if (trace) {
    printf("%s%s%s%s\n", "termTok: ", termToken, "\nfuncTok: ", fnToken);
  }

  for (i=0; i<numFunctions; i++) {
    if (strncmp(fnToken+i, "1", 1)==0) {
      /* trace */
      if (trace) {
        printf("%s%i\n", "\tfunction: ", i);
      }
      for (j=0; j<numSignals; j++) {
        strncpy(tempStr, termToken+j, 1);
        if (strncmp(tempStr, "1", 1)==0)
          sigNum = 1;
        else if (strncmp(tempStr, "0", 1)==0)
          sigNum = 0;
        else if (strncmp(tempStr, "-", 1)==0)
          sigNum = -1;
        else {
          printf("File does not look like a standard espresso file.\n");
          return 1;
        }
        /* trace */
        if (trace) {
          printf("%s%i\n", "\tsignal: ", sigNum);
        }

        cell[i][fnTerms[i]-terms[i]][j] = sigNum;
        /* trace */
        if (trace) {
          printf("\t%s%i%s%i%s%i%s%i\n", "(", i, ",", fnTerms[i]-terms[i], \
                 ",", j, ") = ", sigNum);
        }
      }
      terms[i]--;
    }
  }
  fgets(line, lineLength, fp);
}

/* save the initial number of terms of each array for mem dealloc later */
extern int *initFnTerms;
initFnTerms = malloc(numFunctions*sizeof(int));
memcpy(initFnTerms, fnTerms, numFunctions*sizeof(int));

return 0;
}


/* Displays the (bitter) fruits of my labour.
 In: Nothing
 Out: Nothing
*/
void printOutput() {

/* trace */
if (trace) {
  printf("printOutput()\n");
}

convEspresso();

/* trace */
if (trace) {
  int i, termCnt = 0;
  for (i=0; i<numFunctions; i++)
    termCnt += fnTerms[i];
}

if (fileOutput) {
  FILE *fp = fopen(outputFile, "w");
  fprintf(fp, "%s", inVarCnt);
  fprintf(fp, "%s", outVarCnt);
  fprintf(fp, "%s", inVarNames);
  fprintf(fp, "%s", outVarNames);
  fprintf(fp, "%s%i\n", ".p ", totalNumTerms);
  fprintf(fp, "%s\n", getEspOutput());
  fprintf(fp, "%s\n", ".e");
}
else {
  printf("%s", inVarCnt);
  printf("%s", outVarCnt);
  printf("%s", inVarNames);
  printf("%s", outVarNames);
  printf("%s%i\n", ".p ", totalNumTerms);
  printf("%s\n", getEspOutput());
  printf("%s\n", ".e");
}
}


/* Compress the data structure into standard espresso form.
 In: Nothing
 Out: Nothing
*/
void convEspresso() {

/* trace */
if (trace) {
  printf("\nconvEspresso()\n");
}

/* trace */
trace = 0;

extern int *fnLog;
extern int *termLog;
extern int numSignals;
extern int totalNumTerms;

int maxSize = 0;
int i, j, dupeNum;
for (i=0; i<numFunctions; i++)
  maxSize += fnTerms[i];
/* trace */
if (trace) {
  printf("%s%i\n", "maxSize = ", maxSize);
}

/* trace */
if (trace) {
  printf("%s\n", "initialised");
}

/* init */



termLog = malloc(maxSize*numSignals*sizeof(int));
fnLog = malloc(maxSize*numFunctions*sizeof(int));
totalNumTerms = 0;

/* populate */
for (i=0; i<numFunctions; i++) {
  /* trace */
  if (trace) {
    printf("\t%s%i", "i=", i);
  }
  for (j=0; j<fnTerms[i]; j++) {
  /* trace */
  if (trace) {
    printf("\t%s%i", "j=", j);
  }
    /* trace */
    if (trace) {
      printf("%s%i\n",                                       \
             "function is logged? ",                          \
             checkEspressoTerm(cell[i][j], totalNumTerms, i)  \
       );
    }

    dupeNum = checkEspressoTerm(cell[i][j], totalNumTerms, i);
    if (dupeNum==-1) {
      memcpy(termLog + totalNumTerms*numSignals,              \
             cell[i][j],                                      \
             numSignals*sizeof(int)                           \
             );
      /* trace */
      if (trace) {
        printf("%s%i%s%i%s\n", "\ttermLog write (",                   \
               totalNumTerms*numSignals*(int)sizeof(int), "/",        \
               maxSize*numSignals*(int)sizeof(int), ")"               \
               );
        printf("\tcell[i][j][k] = [ ");
        int k;
        for (k=0; k<numSignals; k++) {
          printf("%i%s", cell[i][j][k], " ");
        }
        printf(" ]\n");
        printf("\ttermLog =       [ ");
        for (k=0; k<numSignals; k++) {
          printf("%i%s", termLog[totalNumTerms*numSignals + k], " ");
        }
        printf(" ]\n");
        printf("%s%i\n", "\t** fnLog index i: ", i*totalNumTerms+j);
      }

      fnLog[totalNumTerms*numFunctions + i] = 1;
      /* trace */
      if (trace) {
        int k;
        printf("\t%s%i%s", "term(", totalNumTerms, ") = ");
        for (k=0; k<numSignals; k++) {
          printf("%i%s", cell[i][j][k], " ");
        }
        printf("\n");
        printf("\t%s%i%s%i%s\n", "fn(", totalNumTerms, ",", i, ") = 1");
      }

      totalNumTerms++;
    }
    else {
      fnLog[dupeNum*numFunctions + i] = 1;
    }
  }
}
}


/* Check if the standard espresso form contains this term.
 If it does, then associate it with the new function as well.
 In: number of lines in esp std form, function term pointer, function num.
 Out: -1->term does not exist, x->term exists, associted with function.
*/
int checkEspressoTerm(int *term, int numLines, int funcNum) {

/* trace */
if (trace) {
  printf("checkEspressoTerm()\n");
}

int i, j, same;
for (i=0; i<numLines; i++) {
  same = 1;
  for (j=0; j<numSignals; j++) {
    /* trace */
    if (trace) {
      printf("%s%i%s%i%s\n", "(", i, ",", j, ")");
      printf("%s%i%s%i%s%i%s%i\n", "termLog[", j, "]=",
             termLog[i*numSignals + j], ", term[", j, "]=", term[j]);
    }
    if (termLog[i*numSignals + j]!=term[j]) {
      same = 0;
    }
  }
  if (same) {
    return i;
  }
}

return -1;
}


/* Convert the Espresso standard output data structure to a string to display.
 In: Nothing.
 Out: String representation of espresso standard output.
*/
char *getEspOutput() {

/* trace */
if (trace) {
  printf("getEspOutput()\n");
}

/* trace */
if (trace) {
  printf("%s%i\n", "numSignals: ", numSignals);
  printf("%s%i\n", "numFunctions: ", numFunctions);
  printf("%s%i\n", "totalNumTerms: ", totalNumTerms);
}

char *str = malloc((numSignals+numFunctions+2)
                   *totalNumTerms*sizeof(char) + 4*sizeof(char));

int i, j, charNum = 0;
for (i=0; i<totalNumTerms; i++) {

  /* trace */
  if (trace) {
     printf("%s%i\n", "\ti = ", i);
  }

  for (j=0; j<numSignals; j++) {

    /* trace */
    if (trace) {
      printf("%s%i\n", "\t\tj = ", j);
    }

    if (termLog[i*numSignals + j]==0)
      sprintf((char *)((int)str + charNum), "%i", 0);
    else if (termLog[i*numSignals + j]==1)
      sprintf((char *)((int)str + charNum), "%i", 1);
    else if (termLog[i*numSignals + j]==-1)
      sprintf((char *)((int)str + charNum), "%c", '-');
    else
      sprintf((char *)((int)str + charNum), "%c", '?');

    charNum++;
  }

  sprintf((char *)((int)str + charNum), "%c", ' ');
  charNum++;

  for (j=0; j<numFunctions; j++) {

    if (fnLog[i*numFunctions + j]==0)
      sprintf((char *)((int)str + charNum), "%i", 0);
    else if (fnLog[i*numFunctions + j]==1)
      sprintf((char *)((int)str + charNum), "%i", 1);
    else
      sprintf((char *)((int)str + charNum), "%c",
              fnLog[i*numFunctions + j]);

    charNum++;
  }

  sprintf((char *)((int)str + charNum), "%c", '\n');
  charNum++;
}
sprintf((char *)((int)str + charNum-1), "%c", '\0');

return str;
}
```

B.3 Simple arithmetic benchmarks

B.3.1 Dhrystone

The *dhrystone* benchmark is a well known simple intger benchmark. The program attempts to measure the execution rate in *DMIPS*, (millions of instructions per second, in *dhrystone*). Although basic, and lacking code to exercise many components of typicaly modern processors, DMIPS/MHz is still widely used in industry to quote the performance of embedded systems.

B.3.1.1 *dhry.c*

Most of the code in a typical portable dhrystone implementation is dedicated to measuring time in order to calculate the execution rate. This version below was been modified to remove user input, fix the number of loops thorugh the main body, and remove timing code.

```
/*****************************************************************************
 *  The BYTE UNIX Benchmarks - Release 3
 *          Module: dhry_2.c   SID: 3.4 5/15/91 19:30:22
 *
 *****************************************************************************
 * Bug reports, patches, comments, suggestions should be sent to:
 *
 *      Ben Smith, Rick Grehan or Tom Yager
 *      ben@bytepb.byte.com   rick_g@bytepb.byte.com   tyager@bytepb.byte.com
 *
 *****************************************************************************
 *  Modification Log:
 *
 *  Adapted from:
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 * **** WARNING **** See warning in n.dhry_1.c
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_2.c (part 3 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ****************************************************************************/
/* SCCSid is defined in dhry_1.c */

#include "dhry.h"

#ifndef REG
#define REG
        /* REG becomes defined as empty */
        /* i.e. no register variables   */
#endif

extern  int     Int_Glob;
extern  char    Ch_1_Glob;


Proc_6 (Enum_Val_Par, Enum_Ref_Par)
/*********************************/
    /* executed once */
    /* Enum_Val_Par == Ident_3, Enum_Ref_Par becomes Ident_2 */

Enumeration  Enum_Val_Par;
Enumeration *Enum_Ref_Par;
{
  *Enum_Ref_Par = Enum_Val_Par;
  if (! Func_3 (Enum_Val_Par))
    /* then, not executed */
    *Enum_Ref_Par = Ident_4;
  switch (Enum_Val_Par)
  {
    case Ident_1:
      *Enum_Ref_Par = Ident_1;
      break;
    case Ident_2:
      if (Int_Glob > 100)
        /* then */
      *Enum_Ref_Par = Ident_1;
      else *Enum_Ref_Par = Ident_4;
      break;
    case Ident_3: /* executed */
      *Enum_Ref_Par = Ident_2;
      break;
    case Ident_4: break;
    case Ident_5:
      *Enum_Ref_Par = Ident_3;
      break;
  } /* switch */
} /* Proc_6 */


Proc_7 (Int_1_Par_Val, Int_2_Par_Val, Int_Par_Ref)
/**********************************************/
    /* executed three times                                      */
    /* first call:      Int_1_Par_Val == 2, Int_2_Par_Val == 3,  */
    /*                  Int_Par_Ref becomes 7                    */
    /* second call:     Int_1_Par_Val == 10, Int_2_Par_Val == 5, */
    /*                  Int_Par_Ref becomes 17                   */
    /* third call:      Int_1_Par_Val == 6, Int_2_Par_Val == 10, */
    /*                  Int_Par_Ref becomes 18                   */
One_Fifty       Int_1_Par_Val;
One_Fifty       Int_2_Par_Val;
One_Fifty      *Int_Par_Ref;
{
  One_Fifty Int_Loc;

  Int_Loc = Int_1_Par_Val + 2;
  *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
} /* Proc_7 */


Proc_8 (Arr_1_Par_Ref, Arr_2_Par_Ref, Int_1_Par_Val, Int_2_Par_Val)
/*********************************************************************/
    /* executed once      */
    /* Int_Par_Val_1 == 3 */
    /* Int_Par_Val_2 == 7 */
Arr_1_Dim       Arr_1_Par_Ref;
Arr_2_Dim       Arr_2_Par_Ref;
int             Int_1_Par_Val;
int             Int_2_Par_Val;
{
  REG One_Fifty Int_Index;
  REG One_Fifty Int_Loc;

  Int_Loc = Int_1_Par_Val + 5;
  Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
  Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
  Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
  for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)
    Arr_2_Par_Ref [Int_Loc] [Int_Index] = Int_Loc;
  Arr_2_Par_Ref [Int_Loc] [Int_Loc-1] += 1;
  Arr_2_Par_Ref [Int_Loc+20] [Int_Loc] = Arr_1_Par_Ref [Int_Loc];
  Int_Glob = 5;
} /* Proc_8 */


Enumeration Func_1 (Ch_1_Par_Val, Ch_2_Par_Val)
/*************************************************/
    /* executed three times                                         */
    /* first call:      Ch_1_Par_Val == ’H’, Ch_2_Par_Val == ’R’    */
    /* second call:     Ch_1_Par_Val == ’A’, Ch_2_Par_Val == ’C’    */
    /* third call:      Ch_1_Par_Val == ’B’, Ch_2_Par_Val == ’C’    */

Capital_Letter   Ch_1_Par_Val;
Capital_Letter   Ch_2_Par_Val;
{
  Capital_Letter        Ch_1_Loc;
  Capital_Letter        Ch_2_Loc;

  Ch_1_Loc = Ch_1_Par_Val;
  Ch_2_Loc = Ch_1_Loc;
  if (Ch_2_Loc != Ch_2_Par_Val)
    /* then, executed */
    return (Ident_1);
  else  /* not executed */
  {
    Ch_1_Glob = Ch_1_Loc;
    return (Ident_2);
   }
} /* Func_1 */


Boolean Func_2 (Str_1_Par_Ref, Str_2_Par_Ref)
/*************************************************/
    /* executed once */
    /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1’ST STRING" */
    /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */

Str_30  Str_1_Par_Ref;
Str_30  Str_2_Par_Ref;
{
  REG One_Thirty        Int_Loc;
      Capital_Letter    Ch_Loc;

  Int_Loc = 2;
  while (Int_Loc <= 2) /* loop body executed once */
    if (Func_1 (Str_1_Par_Ref[Int_Loc],
                Str_2_Par_Ref[Int_Loc+1]) == Ident_1)
      /* then, executed */
    {
      Ch_Loc = 'A';
      Int_Loc += 1;
    } /* if, while */
  if (Ch_Loc >= 'W' && Ch_Loc < 'Z')
    /* then, not executed */
    Int_Loc = 7;
  if (Ch_Loc == 'R')
    /* then, not executed */
    return (true);
  else /* executed */
  {
    if (strcmp (Str_1_Par_Ref, Str_2_Par_Ref) > 0)
      /* then, not executed */
    {
      Int_Loc += 7;
      Int_Glob = Int_Loc;
      return (true);
    }
    else /* executed */
      return (false);
  } /* if Ch_Loc */
} /* Func_2 */


Boolean Func_3 (Enum_Par_Val)
/***************************/
    /* executed once        */
    /* Enum_Par_Val == Ident_3 */
Enumeration Enum_Par_Val;
{
  Enumeration Enum_Loc;

  Enum_Loc = Enum_Par_Val;
  if (Enum_Loc == Ident_3)
    /* then, executed */
    return (true);
  else /* not executed */
    return (false);
} /* Func_3 */
```

B.3.2 CalcPi

The *calc_pi* program was chosen as a simple, easy to understand version of a program to calculate the value of π, to 1,000 decimal places. The algorithm is short, arithmetically intensive, written in *C*, and does not depend on other libraries, making it suitable to compile for *SimpleScalar*. *calc_pi* was sourced online [Author unknown, 2008, 1938].

B.3.2.1 *calc_pi.c*

```
#include <stdio.h>
#define SCALE 10000
#define MAXARR 2800
#define ARRINIT 2000

int main()
{
    int i, j;
    int carry = 0;
    int arr[MAXARR+1];

    for (i = 0; i <= MAXARR; ++i)
        arr[i] = ARRINIT;
    for (i = MAXARR; i; i -= 14)
    {
        int sum = 0;
        for (j = i; j > 0; --j)
        {
            sum = sum*j + SCALE*arr[j];
            arr[j] = sum % (j*2-1);
            sum /= (j*2-1);
        }
        printf("%04d", carry + sum/SCALE);
        carry = sum % SCALE;
    }
    return 0;
}
```

B.4 VHDL multiplier generator

The multiplier generator *multgen* can generate *VHDL* code for an $N \times N$ bit multiplier. The multiplier is approximate, constructed of $(n;m)$ compressors, which determine the degree of approximation. Pipelining is optional, and the user determines the depth in compressors between flip-flops. The user must provide a *VHDL* description of a $(n;m)$ counter, and a *DFF* is pipelining is used. *multgen* is writen in *C*.

B.4.1 *multgen.c*

```
/*
 * Generate VHDL descriptions of compressor multipliers.
 * -----------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>
#include <math.h>
#include <time.h>
#include <getopt.h>


int checkDefined (char c, int val);
int intArrayMax (int *array, int len);
int checkRange (char c, int x, int low, int high);
int printHeader (FILE* file, char *name, int a, int b, int n, int m, int p,
                 int signMult);
int freeStrPtrMatrix (char *** matrix, int x, int y);
int printPNames (char ***array, int x, int y);
int printPCnt (int *array, int len);
int printUsage (char *name);
int mostSigBit (int num);


#define MULT_MAX               128
#define GREATEST_REQUIRED_M(N) ( (int)( floorf( log2f( (float)N ) ) ) +1 )
#define DECLARE_FILE_NAME      "cmultgen.declare.temp"
#define USE_FILE_NAME          "cmultgen.use.temp"
#define LINE_SIZE              256
#define PARTIAL_NAME_SIZE      64
#define DEBUG                  0


char *entityName;

/*
 * Main program. Acquire inputs, sanity check args, invoke methods to print
 * VHDL output.
 */
int main (int argc, char *argv[]) {

  char *progName;                   /* name of this binary */
```

```
int n, m;                         /* use n:m compressors */
int a, b;           /* multiply two numbers A & B - what are their widths? */
char opt;                         /* command line option */
int *pCnt, *pCntNext, *pCntTemp;  /* number of partials in this column */
char ***pNames, ***pNamesNext;    /* names of the partial products */
int i, j;                         /* loop counters */
int signMult=0;                   /* multipler is signed/unsigned */
int wantSigned=0, wantUnsigned=0; /* user wants a signed/unsigned mult */

/* Signal names will be defined as we need them. VHDL required that they are
 * declared before they are used. Open two files, one for declatations, and
 * one for assignment and variable use. Aftet the multiplier has been
 * generated, print back from the files in order, and remove them.
 */
FILE *DECLARE;
FILE *USE;
FILE *OUTPUT;
char *outputFileName;
int  compNonZeroInputs;  /* how many partials wired into this compressor? */
int  partialCnt;         /* how many partials input in this column? */

char *partialName;                /* create a partial with this name */
char *partialNameLat;             /* create a latched partial */
char *line;                       /* a line read from one of the temp files */
int compCnt=0;                    /* how many compressors instantiated? */
int passThruCnt;                  /* how many partials to pass through? */
int cpaLen;                       /* how long does the CPA ader need to be? */
int pipeDepth=0;                   /* how deep before adding a register ? */

progName = argv[0];
/* program name is after the last trailing slash */
if (strrchr(progName, '/')) {
  progName = strrchr(progName, '/')+1;
}

n = m = a = b = INT_MIN;

/* parse the command line args */
while ((opt = getopt(argc, argv, "a:b:n:m:p:su")) != -1) {
  switch (opt) {
  case 'a':
    a = atoi(optarg);
    break;
  case 'b':
    b = atoi(optarg);
    break;
  case 'n':
    n = atoi(optarg);
    break;
  case 'm':
    m = atoi(optarg);
    break;
  case 'p':
    pipeDepth = atoi(optarg);
    break;
  case 's':
    wantSigned = 1;
    break;
  case 'u':
    wantUnsigned = 1;
    break;
  default:
    fprintf (stderr, "Unrecognized option ‘%c’.\n", (char)optopt);
    printUsage(progName);
    exit(0);
  }
}

/* check the inputs */
if (!checkDefined('a', a)) { printUsage(progName); exit(0); }
if (!checkDefined('b', b)) { printUsage(progName); exit(0); }
if (!checkDefined('n', n)) { printUsage(progName); exit(0); }
if (!checkDefined('m', m)) { printUsage(progName); exit(0); }
if (!checkDefined('p', pipeDepth)) { printUsage(progName); exit(0); }

if (!checkRange('a', a, 1, MULT_MAX)) { printUsage(progName); exit(0); }
if (!checkRange('b', b, 1, MULT_MAX)) { printUsage(progName); exit(0); }
if (!checkRange('n', n, 2, MULT_MAX)) { printUsage(progName); exit(0); }
if (!checkRange('p', pipeDepth, 0, INT_MAX)) {printUsage(progName); exit(0);}
if (!checkRange('m', m, 1, log2f((float)MULT_MAX))) {
  printUsage(progName); exit(0);
}

if (m >= n) {
  fprintf(stderr, "Error: m >= n (%i >= %i).\n", m, n);
  printUsage(progName);
  exit(0);
}

if (m > b) {
  fprintf (stderr, "Error: m > b. You don’t need a compressor with so many"
           "outputs if\n the B operand has %i bits\n", b);
  exit (0);
}

```

```
/* check that the compressors are not greater than required to be exact */
if ( m > GREATEST_REQUIRED_M(n) ) {
  fprintf (stderr, "Error: Redundancy. %i inputs only require %i outputs.\n",
           n, GREATEST_REQUIRED_M(n) );
  exit(0);
}

if (!(wantSigned ^ wantUnsigned)) {
  fprintf (stderr,
           "Error: Multiplier must be one of signed OR unsigned.\n");
  printUsage(progName);
  exit(0);
}
if (wantSigned) {
  signMult = 1;
} else {
  signMult = 0;
}

entityName = malloc (64);
outputFileName = malloc (64);
if (signMult) {
  sprintf (entityName, "signed_compressor_mult_%ib_by_%ib_%i_to_%i",
           a, b, n, m);
} else {
  sprintf (entityName, "unsigned_compressor_mult_%ib_by_%ib_%i_to_%i",
           a, b, n, m);
}
strcpy (outputFileName, entityName);
if (pipeDepth) {
  strcat (outputFileName, "_pipelined");
}
strcat (outputFileName, ".vhd");

/* allocate memory for arrays */
pCnt = (int *)malloc ((a+b)*sizeof(int));
pCntNext = (int *)malloc ((a+b)*sizeof(int));
pCntTemp = (int *)malloc ((a+b)*sizeof(int));
pNames = (char ***)malloc (b*sizeof(char **));
pNamesNext = (char ***)malloc (b*sizeof(char **));
for ( i=0 ; i<b; i++ ) {
  pNames[i] = (char **)malloc ((a+b)*sizeof(char **));
  pNamesNext[i] = (char **)malloc ((a+b)*sizeof(char **));
  for ( j=0 ; j<a+b ; j++ ){
    pNames[i][j] = (char *)malloc (PARTIAL_NAME_SIZE);
    pNamesNext[i][j] = (char *)malloc (PARTIAL_NAME_SIZE);
  }
}

/* generate the VHDL output */
/* ************************ */

/* initial number of partials */
i = 1;
for ( j=0; j<a+b; j++) {
  /* set the number of partials */
  if (i >= 0 ) {
    pCnt[j] = i;
  } else {
    pCnt[j] = 0;
  }
  /* calculate the next number */
  if (j<b-1) {
    i++;
  }
  if (j>=a-1) {
    i--;
  }
  pCntNext[j] = 0;
}

/* add additional bits for Baugh Wooley signed multiplier scheme */
if (signMult) {
  pCnt[a]++;
  pCnt[a+b-1]++;
}

if (DEBUG) { printPCnt(pCnt, a+b); }

/* open temp files for writing signal names */
DECLARE = fopen(DECLARE_FILE_NAME,"w");
USE = fopen(USE_FILE_NAME,"w");

/* create the initial partials */
/* keep a temporary count to check the partials have been
 * created in the correct place */
for ( i=0 ; i<a+b ; i++ ) {
  pCntTemp[i] = 0;
}
partialName = (char *)malloc(PARTIAL_NAME_SIZE);
partialNameLat = (char *)malloc(PARTIAL_NAME_SIZE);
/* insert the additional partials for Baugh Wooley first */
if (signMult) {
  sprintf (partialName, "bw1");
```

```
    pCntTemp[a]++;
    strcpy(pNames[0][a], partialName);
    fprintf (DECLARE, "  signal %14s : std_logic; -- Baugh-Wooley\n", partialName);
    fprintf (USE, "  %s <= ’1’;\n", partialName);
    sprintf (partialName, "bw2");
    pCntTemp[a+b-1]++;
    strcpy(pNames[0][a+b-1], partialName);
    fprintf (DECLARE, "  signal %14s : std_logic; -- Baugh-Wooley\n", partialName);
    fprintf (USE, "  %s <= ’1’;\n", partialName);
  }

  /* generate the initial partial products */
  /* ************************************* */
  for ( j=0; j<b ; j++)  {
    for ( i=0; i<a ; i++)  {
      sprintf (partialName, "a%i_b%i", i, j);
      if (pCntTemp[i+j] < pCnt[i+j]) {
        strcpy (pNames[pCntTemp[i+j]][i+j], partialName);
        fprintf (DECLARE, "  signal %14s : std_logic; -- partial product\n",
                 partialName);
        if ( signMult && ((j==a-1)^(i==b-1)) ) {
          fprintf (USE, "  %s <= NOT ( multA(%i) AND multB(%i) );\n",
                   partialName, i, j);
        } else {
          fprintf (USE, "  %s <= multA(%i) AND multB(%i);\n",
                   partialName, i, j);
        }
      } else {
        fprintf (stderr, "Error: Too many partials (%i > %i) in column %i.\n",
                 pCntTemp[i+j], pCnt[i+j], i+j);
        exit (0);
      }
      pCntTemp[i+j]++;
    }
  }

  if (DEBUG) { printf ("initial:\n"); printPNames(pNames, b, a+b); }

  /* check that we have initialised the correct number of partials */
  for ( i=0 ; i<a+b ; i++ ) {
    if (pCnt[i] != pCntTemp[i]) {
      fprintf (stderr, "Error: not enough partials in column %i.\n", i);
      exit (0);
    }
  }

  fprintf (USE, "\n");

  int depth=0;
  int latchCnt=0;

  /* reduce the partials until they can be added by a CPA */
  while (intArrayMax(pCnt, a+b) > 2) {

    if (pipeDepth) {
      depth = (depth+1) % pipeDepth;
      if (depth==0) {
        latchCnt++;
      }
    }

    if (DEBUG) { printf ("another round:\n"); printPNames(pNames, b, a+b); }

    /* group partials in each column */
    for ( i=0 ; i<a+b ; i++ ) {
      if (DEBUG) { printf ("column %i\n", i); }
      /* pass through the partials that won’t be put into compressors */
      passThruCnt = pCnt[i]%n;
      if (passThruCnt==1) {
        strcpy (pNamesNext[pCntNext[i]][i],pNames[pCnt[i]-1][i]);
        pCntNext[i]++;
        pCnt[i]--;
      }
      if (DEBUG) {
        printf ("after pass through:\n");
        printPNames (pNames, b, a+b);
        printf ("to:\n");
        printPNames (pNamesNext, b, a+b);
      }
    } /* pass though partials i<a+b */

      /* now feed the rest of the partials through compressors */
    for (i=0 ; i<a+b ; i++) {
      /* track which partials have been input into a compressor */
      partialCnt=0;

      while (pCnt[i]) {
        /* count partials that input into this compressor */
        compNonZeroInputs = 0;
        if (DEBUG) {printf ("groups of %i\n", n); printPNames(pNames, b, a+b);}
        /* 3 or more, group them into compressors */
        sprintf (partialName, "c%ii", compCnt);
        fprintf (DECLARE, "  signal %14s : std_logic_vector(%i downto 0); "
                 "-- compressor %i input\n",
                 partialName, n-1, compCnt);
```

```
if (pipeDepth && (depth==pipeDepth-1)) {
  /* pipe register */
  sprintf (partialNameLat, "c%ii_lat", compCnt);
  fprintf (DECLARE, "  signal %14s : "
           "std_logic_vector(%i downto 0); \n",
           partialNameLat, n-1);
}
fprintf (USE, "  c%ii <= ", compCnt);
for ( j=0; j<n; j++ ) {
  if (pCnt[i]>0) {
    fprintf (USE, "%s", pNames[partialCnt][i]);
    compNonZeroInputs++;
    partialCnt++;
    pCnt[i]--;
  } else {
    fprintf (USE, "’0’");
  }
  if (j!=n-1) {
    fprintf (USE, " & ");
  } else {
    fprintf (USE, ";\n\n");
  }
}

/* add latches if needed */
if (pipeDepth && (depth==pipeDepth-1)) {
  fprintf (USE, "  compressor%ii_lat: for i in %i downto 0 generate\n",
           compCnt, n-1);
  fprintf (USE, "    compressor%i_latch_level%i : dFFReset\n",
           compCnt, latchCnt);
  fprintf (USE, "    port map (\n");
  fprintf (USE, "      D   => c%ii(i),\n", compCnt);
  fprintf (USE, "      R   => RESET,\n");
  fprintf (USE, "      CLK => CLK,\n");
  fprintf (USE, "      Q   => c%ii_lat(i));\n", compCnt);
  fprintf (USE, "  end generate compressor%ii_lat;\n\n", compCnt);

  /* latched signal */
  fprintf (USE, "  compressor%i : compressor_%i_to_%i\n",
           compCnt, n, m);
  fprintf (USE, "    port map (\n");
  fprintf (USE, "      X => c%ii_lat,\n", compCnt);

} else {
  /* normal signal */
  fprintf (USE, "  compressor%i : compressor_%i_to_%i\n",
           compCnt, n, m);
  fprintf (USE, "    port map (\n");
  fprintf (USE, "      X => c%ii,\n", compCnt);
}

/* determine the output of the compressors */
sprintf (partialName, "c%i", compCnt);
fprintf (USE, "      Y => %s);\n\n", partialName);
if (m>1) {
  fprintf (DECLARE,
           "  signal %14s : std_logic_vector(%i downto 0); "
           "-- compressor %i output\n",
           partialName, m-1, compCnt);
} else {
  fprintf (DECLARE,
           "  signal %14s : std_logic; "
           "-- compressor %i output\n",
           partialName, compCnt);

}
for ( j=0; j<m; j++) {
  /* don’t connect the output if there is no chance of
   * it being asserted */

  if ( j <= mostSigBit(compNonZeroInputs) ) {
    /* output partials need to go into the appropriate column */
    if (m>1) {
      sprintf (partialName, "c%i(%i)", compCnt, j);
    } else {
      sprintf (partialName, "c%i", compCnt);
    }

    /* don’t include partials that can’t possibly contribute to the
     * product
     * ie (for 32x32 bit multiplication, the result must be in 64 bits)
     */
    if (i+j < a+b) {

      /* check array in bounds */
      if (pCntNext[i+j] >= b) {
        fprintf (stderr,
                 "pNamesNext[%i][%i] out of bounds on line %i\n",
                 pCntNext[i+j], i+j, __LINE__); exit(0);
      }
      if (i+j>=a+b) {
        fprintf (stderr,
                 "pNamesNext[%i][%i] out of bounds on line %i\n",
                 pCntNext[i+j], i+j, __LINE__); exit(0);
      }
```

```
              /************************/

              strcpy (pNamesNext[pCntNext[i+j]][i+j], partialName);
              pCntNext[i+j]++;
            }

          } /* j <= mostSigBit */

        }
        compCnt++;
        if (DEBUG) { printf ("became:\n"); printPNames (pNamesNext, b, a+b); }
      } /* while pCnt */

    }

    /* copy the arrays */
    for ( i=0 ; i<b ; i++ ) {
      for ( j=0 ; j<a+b ; j++ ) {
        strcpy (pNames[i][j], pNamesNext[i][j]);
      }
    }
    for ( i=0 ; i<a+b ; i++ ) {
      pCnt[i] = pCntNext[i];
    }

    /* clear the next arrays */
    for ( i=0 ; i<b ; i++ ) {
      for ( j=0 ; j<a+b ; j++ ) {
        sprintf (pNamesNext[i][j], "%s", "");
      }
    }
    for ( i=0 ; i<a+b ; i++ ) {
      pCntNext[i] = 0;
    }

    if (DEBUG) { printf ("check pCnt: "); printPCnt(pCnt, a+b); }

  } /* while intArrayMax > 2 */


  /* all the partials have been accumulated, determine the length of the
   * required carry propagate adder */
  cpaLen = a+b;
  i = 0;
  while ( (pCnt[i]<2) && (i<a+b)) {
    cpaLen--;
    i++;
  }

  /* accumulate the remaining partials in a carry propagate adder */
  if (cpaLen > 0) {
    fprintf (DECLARE, "  signal %14s : std_logic_vector(%i downto 0);\n",
             "cpaAdd_A", cpaLen-1);
    fprintf (DECLARE, "  signal %14s : std_logic_vector(%i downto 0);\n",
             "cpaAdd_B", cpaLen-1);
    fprintf (DECLARE, "  signal %14s : std_logic_vector(%i downto 0);\n",
             "cpaAdd_SUM", cpaLen-1);
  }
  fprintf (DECLARE, "  signal %14s : std_logic_vector(%i downto 0);\n",
           "product", a+b-1);

  if (cpaLen > 0) {
    fprintf (USE, "  cpaAdd_A <= ");
    for ( i=a+b-1 ; i>=a+b-cpaLen ; i-- ) {
      if (strlen(pNames[0][i])>0) {
        fprintf (USE, "%s", pNames[0][i]);
      } else {
        fprintf (USE, "’0’");
      }
      if (i>a+b-cpaLen) {
        fprintf (USE, " & ");
      }
    }
    fprintf (USE, ";\n");
    fprintf (USE, "  cpaAdd_B <= ");
    for ( i=a+b-1 ; i>=a+b-cpaLen ; i-- ) {
      if (strlen(pNames[1][i])>0) {
        fprintf (USE, "%s", pNames[1][i]);
      } else {
        fprintf (USE, "’0’");
      }
      if (i>a+b-cpaLen) {
        fprintf (USE, " & ");
      }
    }
    fprintf (USE, ";\n\n");

    /* instantiate Zimmerman’s Adder from the arith_lib */
    fprintf (USE, "  cpaAdder : Add\n");
    fprintf (USE, "    generic map (\n");
    fprintf (USE, "      width => %i,\n", cpaLen);
    fprintf (USE, "      speed => 2)\n");
    fprintf (USE, "    port map (\n");
    fprintf (USE, "      A => cpaAdd_A,\n");
    fprintf (USE, "      B => cpaAdd_B,\n");
    fprintf (USE, "      S => cpaAdd_SUM);\n\n");

  }

  /* generate the product from the CPA adder outputs and indivdual partials */
  if (cpaLen!=a+b-1) {
    fprintf (USE, "  product(%i downto 0) <= ", a+b-cpaLen-1);
  } else {
    fprintf (USE, "  product(0) <= ");
  }
  for ( i=a+b-cpaLen-1 ; i>=0 ; i-- ) {
    if (strlen(pNames[0][i]) > 0) {
      fprintf (USE, "%s", pNames[0][i]);
    } else {
      fprintf (USE, "%s", "’0’");
    }
    if (i>0) { fprintf (USE, " & "); }
  }
  fprintf (USE, ";\n");

  if (cpaLen > 0) {
    fprintf (USE,
             "  product(%i downto %i) <= cpaAdd_SUM;\n\n", a+b-1, a+b-cpaLen);
  }

  fprintf (USE, "  PROD <= product;\n\n");

  /* close the temp files */
  fclose (DECLARE);
  fclose (USE);

  /* reopen the temp files for reading */
  DECLARE = fopen (DECLARE_FILE_NAME,"r");
  USE = fopen(USE_FILE_NAME,"r");

  /* print files to OUTPUT */
  OUTPUT = fopen (outputFileName, "w");
  printf ("Writing to %s\n", outputFileName);
  printHeader(OUTPUT, progName, a, b, n, m, pipeDepth, signMult);

  line = (char *)malloc (LINE_SIZE);
  while (fgets(line, LINE_SIZE, DECLARE)!=0) {
    fprintf (OUTPUT, "%s", line);
  }
  fprintf (OUTPUT,"begin\n\n");
  while (fgets(line, LINE_SIZE, USE)!=0) {
    fprintf (OUTPUT, "%s", line);
  }
  fprintf (OUTPUT, "end arch;\n\n");
  if (pipeDepth) {
    fprintf (OUTPUT, "\n-- used %i levels of latches\n\n", latchCnt);
  }

  fprintf (OUTPUT, "------------------------------------------------------------------------------\n");

  /* close the temp files */
  fclose (DECLARE);
  fclose (USE);
  remove (DECLARE_FILE_NAME);
  remove (USE_FILE_NAME);

  return 0;
}


/*
 * Check the option ‘c’. If it is set to INT_MIN, the user didn’t set it.
 */
int checkDefined (char c, int val) {
  if (val==INT_MIN) {
    fprintf (stderr, "Error: Undefined input ‘%c’.\n", c);
    return 0;
  } else {
    return 1;
  }
}


/*
 * Find the maximum element in the one dimensional int array.
 */
int intArrayMax (int *array, int len) {
  int i, max=INT_MIN;
  for ( i=0 ; i<len ; i++) {
    if (max<array[i]) {
      max = array[i];
    }
  }
  return max;
}


/*
 * Check that the integer is low <= x <= high
 */
```

```
int checkRange (char c, int x, int low, int high) {
  if (low > x ) {
    fprintf (stderr, "Error: ‘%c’ < %i.\n", c, low);
    return 0;
  }
  if (high < x) {
    fprintf (stderr, "Error: ‘%c’ > %i.\n", c, high);
    return 0;
  }
  return 1;
}


/*
 * Print the partial names array to STDOUT. A debugging routine.
 */
int printPNames (char ***array, int x, int y) {
  int i, j;
  for ( i=0 ; i<x ; i++ ) {
    for ( j=y-1 ; j>=0 ; j-- ) {
      printf ("%12s ", array[i][j]);
    }
    printf ("\n");
  }
  printf ("\n");
  return 0;
}


/*
 * Print an array of integers to STDOUT. A debugging routine.
 */
int printPCnt (int *array, int len) {
  printf ("pCnt = [ ");
  int i;
  for ( i=len-1; i>=0; i--) {
    printf ("%i ", array[i]);
  }
  printf ("]\n");
  return 0;
}


/*
 * Print the VHDL header for operands of width ‘a’ and ‘b’.
 */
int printHeader (FILE *file, char *name, int a, int b, int n, int m, int p,
                 int signMult) {
  time_t systime;
  fprintf (file, "-------------------------------------------------------------------------------\n");
  fprintf (file, "-- File        : %s.vhd\n", entityName);
  fprintf (file, "-- Author      : This file was generated with %s, by Dan Kelly\n",
           name);
  fprintf (file, "-- Company     : University of Adelaide\n");
  time(&systime); /* get time since epoch */
  fprintf (file, "-- Date        : %s\n", ctime(&systime) /* time to str */ );
  fprintf (file, "-------------------------------------------------------------------------------\n");
  fprintf (file, "-- Copyright (c) 2008 University of Adelaide, AUSTRALIA\n");
  fprintf (file, "-------------------------------------------------------------------------------\n");
  fprintf (file, "-- Description :\n");
  fprintf (file, "-- An ");
  if (signMult) {
    fprintf (file, "signed ");
  } else {
    fprintf (file, "unsigned ");
  }
  fprintf (file, "%i bit by %i bit multiplier, based on %i:%i compressors\n",
           a, b, n, m);
  fprintf (file, "-------------------------------------------------------------------------------\n");
  fprintf (file, "\n");
  fprintf (file, "library IEEE;\n");
  fprintf (file, "use IEEE.std_logic_1164.all;\n");
  fprintf (file, "use IEEE.numeric_std.all;\n");
  fprintf (file, "\n");
  fprintf (file, "library compressorLib;\n");
  fprintf (file, "use compressorLib.compressorLib.all;\n");
  fprintf (file, "\n");
  fprintf (file, "library arith_lib;\n");
  fprintf (file, "use arith_lib.arith_lib.all;\n");
  fprintf (file, "\n");
  if (p>0) {
    fprintf (file, "library dfflib;\n");
    fprintf (file, "use dfflib.dfflib.all;\n");
    fprintf (file, "\n");
  }
  fprintf (file, "-------------------------------------------------------------------------------\n");
  fprintf (file, "\n");
  fprintf (file, "entity mult is\n");
  fprintf (file, "\n");
  fprintf (file, "  port (\n");
  fprintf (file, "    multA  : in  std_logic_vector (%i downto 0);\n", a-1);
  fprintf (file, "    multB  : in  std_logic_vector (%i downto 0);\n", b-1);
  if (p>0) {
    fprintf (file, "    CLK    : in  std_logic;\n");
    fprintf (file, "    RESET  : in  std_logic;\n");
  }
  fprintf (file, "    PROD   : out std_logic_vector (%i downto 0)\n", a+b-1);
  fprintf (file, "    );\n");
  fprintf (file, "\n");
  fprintf (file, "end mult;\n");
  fprintf (file, "\n");
  fprintf (file, "-----------------------------------------------------------------------------\n");
  fprintf (file, "\n");
  fprintf (file, "architecture arch of mult is\n");
  return 0;
}


/*
 * Loop through the srting matrix, and free all char pointers
 */
int freeStrPtrMatrix (char *** matrix, int x, int y) {
  int i, j;
  for ( i=0; i<x ; i++ ) {
    for ( j=0 ; j<y ; j++ ) {
      free(matrix[i][j]);
    }
  }
  return 0;
}


/*
 * Tell the user what the program is for and how to use it.
 */
int printUsage (char *name) {
  fprintf (stderr,
           "usage: %s -a <num> -b <num> -n <num> -m <num> [-u] [-s]\n", name);
  fprintf (stderr,
           "\nPerform multiplication on inputs A, B, with n:m compressors.\n"
           "A and B have width a, b respectively.\n");
  fprintf (stderr, "\ta: width of the operand A\n");
  fprintf (stderr, "\tb: width of the operand B\n");
  fprintf (stderr, "\tn: input bits in the n:m compressors\n");
  fprintf (stderr, "\tm: output bits in the n:m compressors\n");
  fprintf (stderr, "\ts: multiplier is for signed numbers\n");
  fprintf (stderr, "\tu: multiplier is for unsigned numbers\n");
  fprintf (stderr, "\tp: insert pipelining registers every p levels\n");
  return 0;
}


/*
 * Determine the most significant bit asserted in the integer NUM.
 * Returns -1 if num == 0.
 */
int mostSigBit (int num) {
  int i=0;
  int msb=-1;
  for (i=0; i<sizeof(int)*8; i++) {
    if ((1<<i) & num) {
      msb = i;
    }
  }
  return msb;
}
```

Appendix C

Arithmetic Operands

"I wish to God these calculations had been executed by steam."

Charles Babbage (1791–1871)

C.1 Arithmetic operands in benchmark programs

This appendix list the number of arithmetic operations observed in all benchmarks, and shows them as a total fraction of all instructions in the benchmarks.

Table C.1: Number of retired integer instructions in the *arithmetic* benchmark.

Benchmark	add	addi	sub	mult	div	addu	addiu	subu	multu	divu
calc_pi	0	0	0	284,724	281,400	1,168,018	1,157,310	288,868	0	3,124
livermore	0	0	0	1,868,153	0	290,264,232	156,544,596	6,738,482	0	318
dhrystone	0	0	0	10,022	10,000	380,822	1,051,967	30,081	0	21
linpack	0	0	0	3,489	0	1,577,264	12,160,950	178,881	0	6
matrix_mult	0	0	0	2,146,689	0	4,427,468	8,655,518	23	0	0
whetstone	0	0	0	126,008	0	866,636	1,036,924	72,397	0	7

Table C.2: Number of retired integer instructions in the *Mediabench* benchmarks.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
ADPCM	0	0	0	28	0	1,522,696	827,625	463,917	0	28
EPIC	0	0	0	99,798	32,582	7,933,356	9,889,788	40,367	195	288
G.721	0	0	0	2,979,131	0	35,967,358	98,915,432	11,240,141	0	221,283
ghostscript	0	0	0	14,613,369	91,566	212,848,326	147,043,560	35,710,906	1,652	14,535,577
JPEG	0	0	0	53,425	7,171	4,116,435	2,207,898	698,503	0	270
Mesa	0	0	0	1,204,309	59,546	14,613,815	11,250,939	296,889	0	7
mpeg2play	0	0	0	2,698,251	1,081,535	228,033,917	139,707,125	166,612,698	4,050	8,868
PEGWIT	0	0	0	147	0	10,546,142	3,633,409	85,677	0	52
RASTA	0	0	0	60,847	13,914	5,082,805	4,460,391	658,718	128,083	44,811

Table C.3: Number of retired integer instructions in the *SPEC CINT2000* benchmarks.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
164.gzip		0	0	7,417,185	0	10,748,519,291	9,711,875,448	987,692,252	0	590
168.wupwise		0	0	256,768,204	579,377	1,799,819,185	998,328,207	172,161,285	313	148
171.swim		0	0	2,327	76	136,744,720	52,889,958	2,826,741	4,417	1,709
172.mgrid		0	0	361,424,369	1,842	7,246,481,424	751,470,772	401,507	154,317	42,635
173.applu		0	0	4,634,846	752	146,835,280	32,689,745	1,645,207	38,505	14,988
175.vpr		0	0	167,184	138,475	117,657,415	44,198,413	701,587	4,035	104,591
176.gcc		0	0	1,899,689	24,569	176,279,386	255,038,087	7,421,987	461,289	248,384
177.mesa		0	0	49,423,747	3,585,357	358,395,491	469,724,684	37,400,315	0	8,614,864
179.art		0	0	307,222	0	142,393,692	356,944,308	62,046	0	8
181.mcf		0	0	37,929	0	25,805,673	29,985,124	3,849,255	194	17,183
183.equake		0	0	1,128,331	43	128,969,606	136,898,646	1,326,091	1,700	287,461
188.ammp		0	0	7,066,924	0	155,780,186	528,578,795	7,229,742	2,059	119,408
197.parser		0	0	1,178,603	2,699,920	444,957,119	484,900,436	49,146,932	0	15,458
200.sixtrack		0	0	0	0	9,721,040	2,655,607	896,002	0	0
255.vortex		0	0	1,247,849	0	1,338,193,411	1,006,057,525	5,766,285	175,750	4,342,091
256.bzip2		0	0	193	156	1,325,552,598	2,331,943,414	24,308,361	0	157
301.apsi		0	0	484,882,740	1,984,303	1,844,620,395	547,390,682	43,710,538	147,486	58,869

Table C.4: Number of retired integer instructions from each *test* benchmark.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
fbench		0	0	385	0	177,546	719,699	3,352	1,019	385
ffbench		0	0	409	720	101,796,861	4,203,097	2,478,052	0	9
miller-rabin		0	0	7,229	39	4,309,262	2,662,082	198,229	0	0
arith-throughput		0	0	1,251,750	250,000	8,727,467	4,597,358	2,941,881	501,129	501,730

Table C.5: Proportion of integer instructions in the *arithmetic* benchmarks as a percentage of retired instructions.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
calc_pi	0.000	0.000	0.000	4.210	4.160	17.250	17.100	4.270	0.000	0.050
livermore	0.000	0.000	0.000	0.140	0.000	21.510	11.600	0.500	0.000	0.000
dhrystone	0.000	0.000	0.000	0.190	0.190	7.270	20.070	0.570	0.000	0.000
linpack	0.000	0.000	0.000	0.010	0.000	3.460	26.650	0.390	0.000	0.000
matrix_mult	0.000	0.000	0.000	7.070	0.000	14.580	28.510	0.000	0.000	0.000
whetstone	0.000	0.000	0.000	0.730	0.000	5.020	6.010	0.420	0.000	0.000
Average	**0.000**	**0.000**	**0.000**	**2.058**	**2.175**	**10.456**	**18.410**	**1.068**	**0.000**	**0.050**

Table C.6: Proportion integer instructions in the *Mediabench* benchmarks as a percentage of retired instructions.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
ADPCM	0.000	0.000	0.000	0.000	0.000	25.190	13.700	7.230	0.000	0.000
EPIC	0.000	0.000	0.000	0.190	0.060	15.020	18.730	0.080	0.000	0.000
G.721	0.000	0.000	0.000	1.100	0.000	13.230	36.400	4.130	0.000	0.080
ghostscript	0.000	0.000	0.000	1.130	0.010	16.480	11.390	2.770	0.000	1.130
JPEG	0.000	0.000	0.000	0.820	0.050	46.180	21.700	8.120	0.000	0.000
Mesa	0.000	0.000	0.000	1.690	0.080	32.800	25.180	0.890	0.000	0.000
mpeg2play	0.000	0.000	0.000	0.290	0.100	34.010	28.280	14.830	0.000	0.000
PEGWIT	0.000	0.000	0.000	0.000	0.000	41.760	14.400	0.340	0.000	0.000
RASTA	0.000	0.000	0.000	0.150	0.030	12.690	11.140	1.640	0.320	0.110
Average	**0.000**	**0.000**	**0.000**	**0.490**	**0.033**	**14.730**	**13.753**	**1.497**	**0.320**	**0.620**

Table C.7: Proportion integer instructions retired from each *SPEC CPU2000* benchmark as a percentage.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
164.gzip	0.000	0.000	0.000	0.010	0.000	12.720	11.490	1.170	0.000	0.000
168.wupwise	0.000	0.000	0.000	1.420	0.000	9.970	5.530	0.950	0.000	0.000
171.swim	0.000	0.000	0.000	0.000	0.000	14.600	5.650	0.300	0.000	0.000
172.mgrid	0.000	0.000	0.000	1.000	0.000	20.110	2.090	0.000	0.000	0.000
173.applu	0.000	0.000	0.000	0.740	0.000	23.480	5.230	0.260	0.010	0.000
175.vpr	0.000	0.000	0.000	0.020	0.020	16.550	6.220	0.100	0.000	0.010
176.gcc	0.000	0.000	0.000	0.120	0.000	10.750	15.560	0.450	0.030	0.020
177.mesa	0.000	0.000	0.000	1.580	0.110	11.440	14.990	1.190	0.000	0.270
179.art	0.000	0.000	0.000	0.020	0.000	7.010	17.580	0.000	0.000	0.000
181.mcf	0.000	0.000	0.000	0.020	0.000	12.770	14.840	1.900	0.000	0.010
183.equake	0.000	0.000	0.000	0.080	0.000	8.830	9.380	0.090	0.000	0.020
188.ammp	0.000	0.000	0.000	0.130	0.000	2.780	9.430	0.130	0.000	0.000
197.parser	0.000	0.000	0.000	0.030	0.080	12.980	14.140	1.430	0.000	0.000
200.sixtrack	0.000	0.000	0.000	0.000	0.000	27.720	7.570	2.560	0.000	0.000
255.vortex	0.000	0.000	0.000	0.010	0.000	14.230	10.700	0.060	0.000	0.050
256.bzip2	0.000	0.000	0.000	0.000	0.000	14.610	25.700	0.270	0.000	0.000
301.apsi	0.000	0.000	0.000	4.340	0.020	16.510	4.900	0.390	0.000	0.000
Average	**0.000**	**0.000**	**0.000**	**0.680**	**0.058**	**13.945**	**10.647**	**0.750**	**0.020**	**0.063**

Table C.8: Proportion integer instructions retired from each *test* benchmark as a percentage.

Benchmark	`add`	`addi`	`sub`	`mult`	`div`	`addu`	`addiu`	`subu`	`multu`	`divu`
fbench	0.000	0.000	0.000	0.000	0.000	1.770	7.160	0.030	0.010	0.000
ffbench	0.000	0.000	0.000	0.000	0.000	15.340	0.630	0.370	0.000	0.000
miller-rabin	0.000	0.000	0.000	0.020	0.000	14.540	8.980	0.670	0.000	0.000
arith-throughput	0.000	0.000	0.000	2.090	0.420	14.580	7.680	4.910	0.840	0.840
Average	**0.000**	**0.000**	**0.000**	**1.055**	**0.420**	**11.557**	**6.112**	**1.495**	**0.425**	**0.840**

Table C.9: Number of retired floating point instructions from each *arithmetic* benchmark.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
calc_pi	0	0	0	0	0	0	0	0	0	0
livermore	7,050,974	3,817,333	6,465,984	162,337	0	4,026,508	1,590,493	3,116,325	1,654,784	28,455
dhrystone	0	0	0	0	0	0	0	0	0	0
linpack	3,850,750	0	3,785,200	1,100	0	0	120,000	0	121,089	0
matrix_mult	0	0	0	0	0	0	0	0	0	0
whetstone	439,299	87,998	253,000	98,300	0	492,513	159,883	549,497	66,288	9,300

Table C.10: Number of retired floating point instructions from each *Mediabench* benchmark.

Benchmark	add.s	sub.s	mul.s	div.s	sqrt.s	add.d	sub.d	mul.d	div.d	sqrt.d
ADPCM	0	0	0	0	0	0	0	0	0	0
EPIC	9,720	0	2,676,737	2	0	2,644,324	32,427	5	65,540	0
G.721	0	0	0	0	0	0	0	0	0	0
ghostscript	12,621	5,740	18,714	1,058	0	51,177	33,891	82,032	13,111	1,228
JPEG	0	0	0	0	0	0	0	0	0	0
Mesa	3,590,465	1,153,984	6,375,184	239,454	0	1,467,204	682,111	1,737,472	237,218	27,230
mpeg2play	0	0	0	0	0	20,542,454	1,791,071	17,791,849	42,489	1,384
PEGWIT	0	0	0	0	0	0	0	0	0	0
RASTA	546,665	244,373	603,908	23,571	0	498,138	197,384	560,737	50,521	8,018

Table C.11: Number of retired floating point instructions from each *SPEC CPU2000* benchmark.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
164.gzip	0	0	0	0	0	0	0	0	0	0
168.wupwise	0	0	0	0	0	779,008,026	289,536,039	1,155,020,853	13,824,015	1,024,003
171.swim	0	0	0	512	0	74,456,297	41,718,968	66,841,311	4,198,478	0
172.mgrid	0	0	0	0	0	7,410,424,463	727,410,763	1,246,904,688	22	2
173.applu	0	0	0	0	0	18,002,032	19,647,684	76,212,148	1,979,311	510
175.vpr	3,173,224	7	106,389	260	0	27,532	1,412	1,930	552	1
176.gcc	0	0	0	0	0	0	0	11,464	11,464	0
177.mesa	17,990,084	10,761,183	14,350,649	324	0	14,342,602	10,756,514	10,757,472	3,589,553	48
179.art	0	0	0	0	0	140,450,544	10,200,126	131,010,470	30,940,056	45,084
181.mcf	0	0	0	0	0	2	0	2	0	0
183.equake	0	0	0	0	0	65,060,819	7,322,656	79,549,397	12,397,122	5,212
188.ammp	0	0	0	0	0	336,299,727	222,856,358	701,419,972	19,307,562	13,174,626
197.parser	0	0	0	0	0	0	0	0	0	0
200.sixtrack	0	0	0	0	0	14	2	16	2	1
255.vortex	0	0	0	0	0	0	0	0	0	0
256.bzip2	0	0	0	0	0	0	0	0	0	0
301.apsi	0	0	0	0	0	515,027,026	418,164,570	706,601,749	143,407,610	3,045,748

Table C.12: Number of retired floating point instructions from each *test* benchmark.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
fbench	0	0	0	0	0	596,000	506,000	1,004,000	404,000	0
ffbench	0	0	0	0	0	62,983,920	63,001,777	84,045,136	1,921	0
miller-rabin	0	0	0	0	0	30	20	20	20	0
arith-throughput	0	0	0	0	0	1,669,961	546,102	729,953	300,003	10,000

Table C.13: Proportion of floating point instructions retired from each *arithmetic* benchmark as a percentage.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
calc_pi	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
livermore	0.520	0.280	0.480	0.010	0.000	0.300	0.120	0.230	0.120	0.000
dhrystone	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
linpack	8.440	0.000	8.290	0.000	0.000	0.000	0.260	0.000	0.270	0.000
matrix_mult	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
whetstone	2.550	0.510	1.470	0.570	0.000	2.850	0.930	3.190	0.380	0.050
Average	**3.837**	**0.395**	**3.413**	**0.290**	**0.000**	**1.575**	**0.437**	**1.710**	**0.257**	**0.050**

Table C.14: Proportion of floating point instructions retired from each *Mediabench* benchmark as a percentage.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
ADPCM	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
EPIC	0.020	0.000	5.070	0.000	0.000	5.010	0.060	0.000	0.120	0.000
G.721	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
ghostscript	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.010	0.000	0.000
JPEG	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Mesa	6.210	2.120	11.200	0.450	0.000	3.600	1.450	4.370	0.560	0.040
mpeg2play	0.000	0.000	0.000	0.000	0.000	6.370	0.450	5.590	0.000	0.000
PEGWIT	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RASTA	1.360	0.610	1.510	0.060	0.000	1.240	0.490	1.400	0.130	0.020
Average	**0.690**	**0.610**	**3.290**	**0.060**	**0.000**	**3.125**	**0.275**	**0.705**	**0.125**	**0.020**

Table C.15: Proportion of floating point instructions retired from each *SPEC CFP2000* benchmark as a percentage.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
164.gzip	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
168.wupwise	0.000	0.000	0.000	0.000	0.000	4.310	1.600	6.400	0.080	0.010
171.swim	0.000	0.000	0.000	0.000	0.000	7.950	4.450	7.140	0.450	0.000
172.mgrid	0.000	0.000	0.000	0.000	0.000	20.570	2.020	3.460	0.000	0.000
173.applu	0.000	0.000	0.000	0.000	0.000	2.880	3.140	12.190	0.320	0.000
175.vpr	0.450	0.000	0.010	0.000	0.000	0.000	0.000	0.000	0.000	0.000
176.gcc	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
177.mesa	0.570	0.340	0.460	0.000	0.000	0.460	0.340	0.340	0.110	0.000
179.art	0.000	0.000	0.000	0.000	0.000	6.920	0.500	6.450	1.520	0.000
181.mcf	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
183.equake	0.000	0.000	0.000	0.000	0.000	4.460	0.500	5.450	0.850	0.000
188.ammp	0.000	0.000	0.000	0.000	0.000	6.000	3.980	12.520	0.340	0.240
197.parser	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
200.sixtrack	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
255.vortex	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
256.bzip2	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
301.apsi	0.000	0.000	0.000	0.000	0.000	4.610	3.740	6.330	1.280	0.030
Average	**0.510**	**0.340**	**0.235**	**0.000**	**0.000**	**6.462**	**2.252**	**6.698**	**0.619**	**0.093**

Table C.16: Proportion of floating point instructions retired from each *test* benchmark as a percentage.

Benchmark	`add.s`	`sub.s`	`mul.s`	`div.s`	`sqrt.s`	`add.d`	`sub.d`	`mul.d`	`div.d`	`sqrt.d`
fbench	0.000	0.000	0.000	0.000	0.000	5.930	5.040	9.990	4.020	0.000
ffbench	0.000	0.000	0.000	0.000	0.000	9.490	9.500	12.670	0.000	0.000
miller-rabin	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
arith-throughput	0.000	0.000	0.000	0.000	0.000	2.790	0.910	1.220	0.500	0.020
Average	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**6.070**	**5.150**	**7.960**	**2.260**	**0.020**

C.2 Operand bit-assertion tables

This appendix lists the operand bits in descending order from the least frequently asserted to the most frequently asserted. The results are grouped by input and output operands, and separately for each arithmetic operation.

Table C.17: Bit assertion probabilities for signed 32 bit integer operands.

	A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
+	**B**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	Z	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
−	**B**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	Z	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	A	0	1	3	4	2	7	5	6	8	11	12	9	13	10	15	14	16	19	17	18	20	22	23	25	24	21	26	27	29	28	30	31
×	**B**	1	0	3	7	5	4	2	9	6	10	8	17	24	14	12	30	16	20	19	25	15	11	22	23	18	21	31	13	28	29	26	27
	Z	3	5	7	8	6	9	4	10	12	11	0	1	2	13	14	16	15	17	18	19	20	21	22	23	24	26	25	28	30	27	29	31
	ZH	11	15	6	18	19	10	0	23	12	1	14	13	3	8	4	20	2	5	7	17	9	21	16	22	30	31	29	27	28	25	26	24
	A	0	3	2	1	5	4	6	7	8	9	10	11	13	12	14	15	16	17	18	19	20	21	22	23	24	25	26	30	29	27	28	31
÷	**B**	0	1	2	3	5	4	6	8	7	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	Z	0	1	2	3	4	5	6	7	13	8	9	10	12	11	14	15	16	17	18	19	20	21	22	23	24	25	27	26	28	29	30	31
	ZH	1	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	20	19	21	22	23	24	25	26	27	28	29	30	31

Table C.18: Bit assertion probabilities for unsigned 32 bit integer operands.

	A	4	28	3	5	6	10	8	7	12	9	13	11	2	14	16	18	17	19	20	1	21	22	23	15	0	24	25	26	27	30	29	31
+	**B**	5	0	28	6	4	3	9	7	2	12	11	10	8	13	14	16	18	17	15	1	19	20	23	21	22	24	25	26	27	30	29	31
	Z	3	4	28	6	5	12	10	9	7	8	13	11	14	2	16	18	17	1	19	0	20	21	23	22	15	24	25	26	27	30	29	31
	A	5	0	3	6	4	1	12	14	7	13	2	10	9	11	8	28	16	15	18	17	19	20	21	22	23	24	25	27	26	29	30	31
−	**B**	0	1	2	4	3	5	6	7	28	8	13	10	16	12	9	14	18	11	17	24	19	15	23	21	22	25	20	27	26	30	29	31
	Z	0	2	3	4	1	5	12	6	13	8	7	9	11	10	14	15	16	17	19	18	20	21	30	29	27	22	26	25	28	23	24	31
	A	11	7	5	13	10	9	12	0	8	6	4	3	2	14	1	15	16	19	17	20	18	22	23	21	29	27	28	30	31	25	24	26
×	**B**	3	1	0	31	7	15	27	11	4	28	12	8	23	22	16	29	19	20	24	17	2	10	6	14	21	25	30	13	18	26	5	9
	Z	5	6	12	8	11	21	17	22	4	14	31	29	30	9	10	19	13	18	16	20	7	15	28	1	23	3	24	2	27	25	26	0
	ZH	7	8	0	1	3	9	2	10	11	4	6	5	15	12	18	16	17	13	19	14	22	20	21	23	27	28	24	26	25	30	29	31
	A	1	5	0	12	2	4	3	14	13	15	29	16	30	17	18	7	9	6	19	20	21	8	22	10	25	11	23	24	28	27	26	31
÷	**B**	13	15	0	1	3	2	6	4	5	14	9	11	8	7	12	10	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	Z	1	5	0	12	2	4	3	14	13	15	29	16	30	17	18	7	9	6	19	20	21	8	22	10	25	11	23	24	28	27	26	31
	ZH	0	2	14	13	12	1	4	3	5	6	15	7	9	8	10	11	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Table C.19: Bit assertion probabilities for single precision floating point numbers.

	A	26	27	29	28	24	17	14	16	15	25	23	10	21	19	11	3	18	2	22	4	5	1	7	8	9	20	6	0	30	13	12	31
+	**B**	27	29	28	26	25	15	14	23	18	17	16	24	10	19	2	9	5	3	11	8	1	21	4	13	7	6	12	22	20	0	30	31
	Z	26	27	29	28	23	24	25	18	21	17	16	15	14	10	9	7	11	19	2	3	20	6	12	5	8	13	22	30	4	1	0	31
	A	24	26	23	30	25	19	22	15	18	27	14	10	29	28	3	9	7	2	5	8	17	16	11	6	4	1	0	20	12	21	13	31
−	**B**	26	27	29	28	25	23	24	15	18	14	19	16	17	30	10	11	7	3	22	2	5	1	4	6	8	9	20	0	13	21	12	31
	Z	24	26	30	25	29	28	21	23	18	27	11	19	17	20	14	15	16	10	12	7	13	8	22	31	6	9	3	5	4	2	1	0
	A	26	27	29	28	24	23	14	19	18	22	15	25	10	4	6	8	2	9	7	5	20	17	3	21	1	16	11	13	30	12	0	31
×	**B**	26	29	28	23	27	24	25	18	10	15	14	19	9	22	2	3	4	1	0	17	21	30	6	5	8	16	13	12	7	20	11	31
	Z	26	27	29	28	10	11	14	16	15	21	17	25	3	4	2	24	5	30	7	1	6	23	0	18	13	20	8	9	12	19	22	31
	A	24	23	30	26	25	19	22	28	29	18	27	14	10	15	20	9	4	8	17	21	2	16	6	7	12	1	3	5	13	11	0	31
÷	**B**	30	24	22	19	23	26	25	21	16	18	20	17	10	14	27	8	28	29	15	9	12	2	3	1	13	6	7	11	5	4	0	31
	Z	28	29	27	26	25	24	0	7	15	2	3	19	6	4	23	1	12	14	10	11	13	9	18	21	5	16	22	17	20	8	30	31

Table C.20: Bit assertion probabilities for double precision floating point numbers.

	A	54	60	61	59	58	55	53	56	52	48	45	57	39	43	47	62	51	41	44	40	49	46	35	37	36	38	50	33	32	42	34	29	⋯
+	**B**	57	60	61	59	58	56	55	53	54	52	40	48	38	44	46	32	20	49	37	43	35	50	39	36	24	47	45	28	42	30	34	51	⋯
	Z	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	⋯
	A	53	55	57	52	56	54	59	60	61	58	62	40	51	39	48	43	45	49	37	47	42	46	36	32	41	44	50	38	33	31	35	34	⋯
−	**B**	57	53	56	54	55	60	61	59	58	52	62	43	39	45	40	49	30	33	44	34	48	31	37	51	38	50	28	36	35	27	47	32	⋯
	Z	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	⋯
	A	55	60	61	59	58	54	57	56	53	52	51	48	62	47	49	50	46	45	44	43	42	39	38	40	41	37	30	29	33	31	35	36	⋯
×	**B**	59	60	61	58	57	56	55	53	54	52	48	43	32	44	35	29	36	47	30	37	51	39	50	40	28	49	31	34	21	33	23	15	⋯
	Z	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	⋯
	A	52	55	54	53	56	62	57	58	60	61	59	48	45	40	39	51	38	35	33	47	37	49	36	41	43	32	50	44	46	42	30	18	⋯
÷	**B**	62	54	52	55	53	50	48	51	49	44	56	39	43	35	45	32	40	29	31	36	57	58	59	60	61	42	47	46	37	38	33	41	⋯
	Z	41	45	48	51	44	40	32	35	37	30	18	26	28	22	11	10	8	31	13	29	24	25	17	27	21	23	14	15	16	12	20	19	⋯

Table C.21: Bit assertion probabilities for double precision floating point numbers *(continued. . .)*.

	A	···	30	28	26	31	27	24	20	22	11	25	19	13	21	23	18	16	10	17	12	8	4	7	14	6	15	9	3	5	2	1	0	63
+	**B**	···	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	Z	···	29	30	28	22	10	27	11	24	16	26	18	23	8	13	25	14	20	21	4	12	5	19	15	3	1	63	7	17	6	9	2	0
	A	···	33	26	18	31	62	41	22	14	10	16	23	29	19	7	8	27	25	4	12	15	1	21	17	11	3	63	0	5	2	6	9	13
−	**B**	···	18	29	19	46	20	23	26	8	41	42	11	10	24	22	21	14	16	7	4	9	12	25	17	63	13	15	1	3	2	6	5	0
	Z	···	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	A	···	34	25	20	32	23	17	21	26	18	13	11	28	8	15	10	24	19	27	7	6	16	14	22	12	2	1	9	5	4	0	3	63
×	**B**	···	16	1	38	20	10	8	17	41	19	27	46	4	7	42	45	26	24	3	25	12	11	22	6	2	9	13	14	18	5	0	62	63
	Z	···	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	A	···	8	22	26	10	4	3	11	28	13	34	31	29	24	15	25	14	9	16	19	7	17	12	21	0	2	1	23	27	6	5	20	63
÷	**B**	···	34	26	23	25	30	24	22	16	19	27	15	1	13	11	20	12	4	14	28	9	3	21	17	18	2	5	6	10	7	0	8	63
	Z	···	9	7	36	38	39	42	43	0	1	2	3	4	5	6	33	34	46	47	49	50	52	53	54	55	56	57	58	59	60	61	62	63

Appendix D

Detailed benchmark descriptions

"Let's say the docs present a simplified view of reality... :-)"

Larry Wall (1954 —)

D.1 Mediabench benchmarks

The following are extended descriptions of each *Mediabench* benchmark from the website [Fritts, 1997].

JPEG is a standardized compression method for full-color and gray-scale images. The benchmark implements JPEG image compression and decompression. JPEG is lossy. This package contains C software to implement JPEG image compression and decompression. JPEG (pronounced "jay-peg") is a standardized compression method for full-color and gray-scale images. JPEG is intended for compressing "real-world" scenes; line drawings, cartoons and other non-realistic images are not its strong suit. JPEG is lossy, meaning that the output image is not exactly identical to the input image.

mpeg2play mpeg2play is a player for MPEG-1 and MPEG-2 video bitstreams. It is based on mpeg2decode by the MPEG Software Simulation Group. In mpeg2decode the emphasis is on correct implementation of the MPEG standard and comprehensive code structure. The latter is not always easy to combine with high execution speed. Therefore a version has been derived which is optimized for higher decoding and display speed at the cost of a less straightforward implementation and slightly non-compliant decoding. In addition all conformance checks and some fault recovery procedures have been omitted from mpeg2play.

GSM is an implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding which uses RPE/LTP (residual pulse excitation/long term prediction) coding at 13 kbit/s. The quality of the algorithm is good enough for reliable speaker recognition given the bandwidth limitations of 8 kHz sampling rate. As part of this effort we are publishing an implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding, prI-ETS 300 036, which uses RPE/LTP (residual pulse excitation/long term prediction) coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits; for compatibility with typical UNIX applications, our implementation turns frames of 160 16 bit linear samples into 33-byte frames (1650 Bytes/s). The quality of the algorithm is good enough for reliable speaker recognition; even music often survives transcoding in recognizable form (given the bandwidth limitations of 8 kHz sampling rate).

ADPCM (Adaptive Differential Pulse Code Modulation) is a family of speech compression and decompression algorithms. The ADPCM code used is the Intel/DVI ADPCM code which is being recommended by the IMA Digital Audio Technical Working Group. ADPCM stands for Adaptive Differential Pulse Code Modulation. It is a family of speech compression and decompression algorithms. A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The ADPCM code used is the Intel/DVI ADPCM code which is being recommended by the IMA Digital Audio Technical Working Group. Note that this is NOT a CCITT G722 coder. The CCITT ADPCM standard is much more complicated, probably resulting in better quality sound but also in much more computational overhead.

G.721 is the CCITT (International Telegraph and Telephone Consultative Committee) implementation of G.721 voice compression. The files in this package comprise ANSI-C language reference implementations of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions. They have been tested on Sun SPARCstations and passed 82 out of 84 test vectors published by CCITT (Dec. 20, 1988) for G.721 and G.723. [The two remaining test vectors, which the G.721 decoder implementation for u-law samples did not pass, may be in error because they are identical to two other vectors for G.723_40.]

PGP (Pretty Good Privacy) uses “message digests” to form signatures. A message digest is a 128-bit cryptographically strong one-way hash function of the message (MD5). To encrypt data, it uses a block-cipher IDEA, RSA for key management and digital signatures. A session key is generated for an individual message and the message is encrypted by IDEA using the session key and the session key is encrypted using RSA. PGP uses "message digests" to form signatures. A message digest is a 128-bit cryptographically strong one-way hash function of the message (MD5). To encrypt data, it uses a block-cipher IDEA, RSA for key management and digital signatures. A session key is generated for an individual message and the message is encrypted by IDEA using the session key and the session key is encrypted using RSA.

PEGWIT Pegwit is a program for performing public key encryption and authentication. It uses an elliptic curve over GF(2^{255}), SHA1 for hashing, and the symmetric block cipher square.

ghostscript Ghostscript is the name of a set of software that provides:

1. An interpreter for the PostScript (TM) language.
2. A set of C procedures (the Ghostscript library) that implement the graphics capabilities that appear as primitive operations in the PostScript language.
3. An interpreter for Portable Document Format (PDF) files.

Ghostscript is the name of a set of software that provides: (1) An interpreter for the PostScript (TM) language, and (2) A set of C procedures (the Ghostscript library) that implement the graphics capabilities that appear as primitive operations in the PostScript language, and (3) An interpreter for Portable Document Format (PDF) files.

Mesa Mesa is a 3-D graphics library with an API which is very similar to that of OpenGL.

SPHERE (SPeech HEader REsources) is a set of library functions and command-level programs which can be used to read and modify NIST-formatted speech waveform files. SPeech HEader REsources (SPHERE) is a set of library functions and command-level programs which can be used to read and modify NIST-formatted speech waveform files.

RASTA is a program for the rasta-plp processing and it supports the following front-end techniques: PLP, RASTA, and Jah-RASTA with fixed Jah-value. The Jah-Rasta technique simultaneously handles additive noise and spectral distortion. RASTA is a program for the rasta-plp processing and it supports the following front-end techniques: PLP, RASTA, and Jah-RASTA with fixed Jah-value. The Jah-Rasta technique handles two different types of harmful effects for speech recognition systems, namely additive noise and spectral distortion, simultaneously, by bandpass filtering the temporal trajectories of a non-linearly transformed critical band spectrum.

EPIC (Efficient Pyramid Image Coder) is an experimental image data compression utility. The filters have been designed to allow extremely fast decoding on non-floating point hardware, at the expense of slower encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition). EPIC (Efficient Pyramid Image Coder) is an experimental image data compression utility written in the C programming language. The compression algorithms are based on a biorthogonal critically-sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters have been designed to allow extremely fast decoding on conventional (ie, non-floating point) hardware, at the expense of slower encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition).

D.2 SPEC benchmarks

The following are descriptions for each benchmark are modified from descriptions provided on the *SPEC* website [Corporation, 2000b,a].

D.2.1 *SPEC CINT2000*

164.gzip(GNU zip) is a popular data compression program written by Jean-Loup Gailly <gzip@gnu.org> for the GNU project. 'gzip' uses Lempel-Ziv coding (LZ77) as its compression algorithm.

SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to just the CPU and the memory subsystem.

175.vpr VPR is a placement and routing program; it automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs.

Placement consists of determining which logic block and which I/O pad within the FPGA should implement each of the functions required by the circuit. The goal is to place pieces of logic which are connected (i.e. must communicate) close together in order to minimize the amount of wiring required and to maximize the circuit speed. This is basically a slot assignment problem – assign every logic block function required by the circuit and every I/O function required by the circuit to a logic block or I/O pad in the FPGA, such that speed and wire-minimization goals are met. VPR uses simulated annealing to place the circuit. An initial random placement is repeatedly modified through local perturbations in order to increase the quality of the placement, in a method similar to the way metals are slowly cooled to produce strong objects.

Routing (in an FPGA) consists of determining which programmable switches should be turned on in order to connect the pre-fabricated wires in the FPGA to the logic block inputs and outputs, and to other wires, such that all the connections required by the circuit are completed and such that the circuit speed is maximized. The connections required by the circuit are represented as a hypergraph, and the possible connections of wire segments to other wires and to logic block inputs and outputs are represented by (a different) directed graph, which is often called a "routing-resource" graph.

VPR uses a variation of Dijkstra’s algorithm in its innermost routing loop in order to connect the terminals of a net (signal) together. Congestion detection and avoidance features run "on top" of this innermost algorithm to resolve contention between different circuit signals over the limited interconnect resources in the FPGA.

gcc 176.gcc is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled. 176.gcc has had its inlining heuristics altered slightly, so as to inline more code than would be typical on a Unix system in 1997. It is expected that this effect will be more typical of compiler usage in 2002. This was done so that 176.gcc would spend more time analyzing it’s source code inputs, and use more memory. Without this effect, 176.gcc would have done less analysis, and needed more input workloads to achieve the run times required for *SPEC CINT2000*.

181.mcf A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C, the benchmark version uses almost exclusively integer arithmetic.

The program is designed for the solution of single-depot vehicle scheduling (sub-)problems occurring in the planning process of public transportation companies. It considers one single depot and a homogeneous vehicle fleet. Based on a line plan and service frequenciesd, so-called timetabled trips with fixed departure/arrival locations and times are derived. Each of this timetabled trip has to be serviced by exactly one vehicle. The links between these trips are so-called dead-head trips. In addition, there are pull-out and pull-in trips for leaving and entering the depot.

Cost coefficients are given for all dead-head, pull-out, and pull-in trips. It is the task to schedule all timetabled trips to so-called blocks such that the number of necessary vehicles is as small as possible and, subordinate, the operational costs among all minimal fleet solutions are minimized.

For simplification in the benchmark test, we assume that each pull-out and pull-in trip is defined implicitly with a duration of 15 minutes and a cost coefficient of 15.

For the considered single-depot case, the problem can be formulated as a large-scale minimum-cost flow problem that we solve with a network simplex algorithm accelerated with a column generation. The core of the benchmark 181.mcf is the network simplex code "MCF Version 1.2 – A network simplex implementation", For this benchmark, MCF is embedded in the column generation process.

The network simplex algorithm is a specialized version of the well known simplex algorithm for network flow problems. The linear algebra of the general algorithm is replaced by simple network operations such as finding cycles or modifying spanning trees that can be performed very quickly. The main work of our network simplex implementation is pointer and integer arithmetic.

186.crafty Crafty is a high-performance Computer Chess program that is designed around a 64-bit word. It runs on 32 bit machines using the “long long” (or similar, as _int64 in Microsoft C) data type. It is primarily an integer code, with a significant number of logical operations such as and, or, exclusive or and shift. It can be configured to run a reproducible set of searches to compare the integer/branch prediction/pipe-lining facilities of a processor.

197.parser The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words.

The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. The parser is robust; it is able to skip over portions of the sentence that it cannot understand, and assign some structure to the rest of the sentence. It is able to handle unknown vocabulary, and make intelligent guesses from context about the syntactic categories of unknown words.

252.eon Eon is a probabilistic ray tracer based on Kajiya's 1986 ACM SIGGRAPH conference paper. It sends a number of 3D lines (rays) into a 3D polygonal model. Intersections between the lines and the polygons are computed, and new lines are generated to compute light incident at these intersection points. The final result of the computation is an image as seen by camera. The computational demands of the program are much like a traditional deterministic ray tracer as described in basic computer graphics texts, but it has less memory coherence because many of the random rays generated in the same part of the code traverse very different parts of 3D space.

253.perlbmk 253.perlbmk is a cut-down version of Perl v5.005_03, the popular scripting language. *SPEC*'s version of Perl has had most of OS-specific features removed. In addition to the core Perl interpreter, several third-party modules are used: MD5 v1.7, MHonArc v2.3.3, IO-stringy v1.205, MailTools v1.11, TimeDate v1.08.

254.gap It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).

255.vortex VORTEx is a single-user object-oriented database transaction benchmark which exercises a system kernel coded in integer C. The VORTEx benchmark is a derivative of a full OODBMS that has been customized to conform to *SPEC CINT2000* (component measurement) guidelines.

The benchmark 255.vortex is a subset of a full object oriented database program called VORTEx. (VORTEx stands for "Virtual Object Runtime EXpository.")

Transactions to and from the database are translated though a schema. (A schema provides the necessary information to generate the mapping of the internally stored data block to a model viewable in the context of the application.)

256.bzip2

256.bzip2 is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and 256.bzip2 is that *SPEC*'s version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.

twolf

The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors (known as standard cells) which constitute the microchip. The placement problem is a permutation. Therefore, a simple or brute force exploration of the state space would take an execution time proportional to the factorial of the input size. For problems as small as 70 cells, a brute force algorithm would take longer than the age of the universe on the world's fastest computer. Instead, the TimberWolfSC program uses simulated annealing as a heuristic to find very good solutions for the row-based standard cell design style. In this design style, transistors are grouped together to form standard cells. These standard cells are placed in rows so that all cells of a row may share power and ground connections by abutment. The simulated annealing algorithm has found the best known solutions to a large group of placement problems. The global router which follows the placement step interconnects the microchip design. It utilizes a constructive algorithm followed by iterative improvement.

The basic simulated annealing algorithm has not changed since its inception in 1983. The version in the *SPEC* suite is the most numerically intensive version. Recent versions have reduced runtimes by clever reductions in the search space. However, the move strategy and cost functions are identical to later versions.

The version of TimberWolfSC that has been submitted to *SPEC* has been customized for *SPEC*. It has been modified specifically for the benchmark suite so that it would have a behavior that captures the flavor of many implementations of simulated annealing. The submitted program spends most of its time in the inner loop calculations. Therefore this version traverses memory often creating cache misses. In fact, this version running small jobs looks like later simulated annealing versions running on large jobs. This was to insure that the benchmark would be applicable to future versions of the program running large instances. The submitted version should be a computers worst nightmare, yet realistic for future problems.

D.2.1.1 *SPEC CFP2000*

168.wupwise “wupwise” is an acronym for “Wuppertal Wilson Fermion Solver”, a program in the area of lattice gauge theory (quantum chromodynamics).

Lattice gauge theory is a discretization of quantum chromodynamics which is generally accepted to be the fundamental physical theory of strong interactions among the quarks as constituents of matter. The most time-consuming part of a numerical simulation in lattice gauge theory with Wilson fermions on the lattice is the computation of quark propagators within a chromodynamic background gauge field. These computations use up a major part of the world’s high performance computing power.

Quark propagators are obtained by solving the inhomogeneous lattice-Dirac equation. The Wuppertal Wilson Fermion Solver (wupwise) solves the inhomogeneous lattice-Dirac equation via the BiCGStab iterative method which has established itself as a method of choice.

171.swim Benchmark weather prediction program for comparing the performance of current supercomputers. The model is based on the paper, "The Dynamics of Finite-Difference Models of the Shallow-Water Equations", by Robert Sadourny, J. ATM. SCIENCES, VOL 32, NO 4, APRIL 1975.

Adapted by *SPEC* for use in the *SPEC CPU2000* Suites as an example of a compute intensive floating point program that was once relegated only to "supercomputers" but can now be done on current computer systems.

172.mgrid 172.mgrid demonstrates the capabilities of a very simple multigrid solver in computing a three dimensional potential field.

Adapted by *SPEC* from the NAS Parallel Benchmarks with modifications for portability and a different workload.

173.applu Solution of five coupled nonlinear PDE’s, on a 3-dimensional logically structured grid, using an implicit psuedo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. This scheme is functionally equivalent to a nonlinear block SSOR iterative scheme with lexicographic ordering. Spatial discretization of the differential operators are based on second-order accurate finite volume scheme. Insists on the strict lexicographic ordering during the solution of the regular sparse lower and upper triangular matrices. As a result, the degree of exploitable parallelism during this phase is limited to $O(N^2)$ as opposed to $O(N^3)$ in other phases and it's spatial distribution is non-homogenous. This fact also creates challenges during the loop re-ordering to enhance the cache locality.

177.mesa Mesa is a free OpenGL work-alike library. Since it supports a generic frame buffer it can be configured to have no OS or window system dependencies. Any number of client programs can be written to stress FP, scalar or memory performance (or a mix). Output can be written to image files for verification.

178.galgel This problem is a particular case of the GAMM (Gesellschaft fuer Angewandte Mathematik und Mechanik) benchmark devoted to numerical analysis of oscillatory instability of convection in low-P_{randtl}-number fluids.

The physical problem is the following. There is a rectangular box filled by a liquid whose P_{randtl} number is $P_r = 0.015$. The aspect ratio of the cavity length-/height is 4. The left and right vertical walls are maintained at higher and lower temperatures respectively. This causes a convective motion in the liquid. When the temperature difference is relatively small the convective flow is steady. The flow looses its stability and become oscillatory when the temperature difference exceeds a certain value.

The buoyancy force, which causes the convective flow, is characterized by a parameter called Grashof number. Besides all, the Grashof number (Gr) is proportional to the characteristic temperature difference (difference of the temperatures at the vertical walls in this case).

The task of the GAMM benchmark is to calculate the critical value of the Grashof number which corresponds to a bifurcation from steady to oscillatory state of the flow. Together with the critical Gr it is necessary to calculate the critical frequency (the frequency of the resulting oscillations when Gr is equal to its critical value).

The critical values (critical Grashof number and critical frequency) depend on all parameters of the problem and the boundary conditions. The GAMM benchmark considers fixed values of the P_{randtl} number and the aspect ratio (0.015 and 4 respectively), and varies the boundary conditions. The boundary conditions used here correspond to the Rigid/adiabatic–Free/adiabatic case.

The numerical method used here is the spectral Galerkin method with the basis functions defined globally in the whole region of the flow.

The Galerkin method requires large computer memory required to keep all coefficients of the resulting dynamic system. To avoid this some coefficients are recalculated each time when a calculation of rhs of the dynamic system is necessary, leading to a rapid increase of the required memory and cpu time when the number of the Galerkin basis functions is increased.

A relatively small number of degrees of freedom makes it possible to study linear stability of steady solutions, requiring solution of an eigenvalue problem, which is usually impossible for an arbitrary CFD code. It becomes possible with the use of the global Galerkin method, and it was successfully done for convective flows and for swirling flows in a closed cylindrical container.

After linear stability analysis is completed and the bifurcation point is calculated, we calculate an asymptotic approximation of the supercritical flow.

179.art The Adaptive Resonance Theory 2 (ART 2) neural network is used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first trained on the objects. After training is complete, the learned images are found in the scanfield image. A window corresponding to the size of the learned objects is scanned across the scanfield image and serves as input for the neural network. The neural network attempts to match the windowed image with one of the images it has learned.

The ART 2 neural network models several characteristics of organic neural processing that is not modelled in more traditional Feed Forward Neural Networks(FFNN). In brief, ART 2 neural networks offer the following advantages over traditional FFNN:

- Expectation influences inputs—The past learnings of an ART 2 neural network influence the matching process.
- Creates own classifications—During training, the ART 2 neural network does not need explicit output information; it creates its own classification groups.
- Learns on-the-fly—ART 2 neural networks are capable of learning and classifying at the same time. The benchmark does not use this feature of ART 2 neural networks.
- Contrast enhancement—ART 2 neural networks perform constrast enhancement through a series of normalizations in the dynamical system.

183.equake The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin. The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method.

187.facerec This is an implementation of the face recognition system described by M. Lades et al.

In this application, an object—here, faces photographed frontally —are represented as labeled graphs. In the simplest case, used here, the graph is a regular grid. To each vertex of the grid graph a set of features are attached; they are computed from the Gabor wavelet transform of the image and represent it in the surroundings of a vertex. An edge of the graph is labeled with the vector connecting its two vertices and represents the topographical relationship of those vertices.

An object represented in this way can now be compared to a new image in a process called elastic graph matching. This is done by first determining the Gabor wavelet transform for the new image. Then, for a given correspondance between the graph's vertices and a set of image points, a function taking into account both the similarity of the feature vectors at every vertex and its corresponding image point, and the distortion of the graph generated by the set of image points, measured as the change in the edge labels, can be computed. This graph similarity function is then the objective function of an optimization process that varies the set of corresponding points in the image. This optimization process is implemented in two steps: The global move step keeps the graph rigid and moves it systematically over all of the image, resulting in a placement that has the highest similarity to the graph. This step can be considered as finding the object (face) in the image. The local move step then takes this placement as the starting position, and visits every vertex in random order. At each vertex, the similarity function is evaluated on a small subgrid surrounding the current position. (This is a small change from the algorithm as originally published, where the trial moves at each node were random as well.) If the similarity function's value is improved at one of those positions, the change is made permanent; such a move is called a hop. One round visiting each vertex position is called a sweep. The local move step terminates when a sweep is completed without a hop having been performed.

The benchmark consists of the following main phases:

- Face Learning: The system has no prior knowledge of the class of object it is supposed to recognize. It "learns" this by extracting a canonic graph from one so-called canonic image; that image and the position at which the graph is to be extracted are specified by the user.
- Graph Generation: For each of the images in the album gallery (see Input Description, below), the Gabor wavelet transform is computed, and the global move step is performed using the canonic graph. The resultant graph is extracted from the transform and stored.
- Recognition: For each of the images in the probe gallery (see Input Description, below), the Gabor wavelet transform is computed, and the global move step is performed using the canonic graph. Then, a local move step is performed using each of the stored graphs. The resultant vector of similarity values is searched for the maximal value; the associated graph (and image) indicate the person recognized.

The parts that take the most computational time have the following characteristics:

- Gabor Wavelet Transform: The transform is performed by computing the forward fast Fourier transform (FFT) of the image, multiplying it with a number (here, 40) of kernels, computing the backward FFT for each of the results, and inserting the absolute value for each pixel into a two-dimensional array of feature vectors. This last step is similar to performing a transpose of a large matrix, and stresses a processor's memory subsystem. Finally, each feature vector is normalized. Run time is proportional to the sum of the number of entries in the album and probe galleries.
- Global Move: This takes only a smallish part of the total run time. It is dominated by the computation of the feature similarity function, which basically is the scalar product of the two feature vectors (one from the graph, one from the image transform). Again, run time is proportional to the sum of the number of entries in the album and probe galleries.

- Local Move: The local move step is dominated by the computation of the similarity function. In addition to the feature similarity function described above, now also the distortion of the grid introduced by the hops has to be taken into account. This is done incrementally, i.e., the contribution of the vertex currently being considered to the distortion is computed for both its old and its new position, which entails handling the nine different positions a vertex can be in the graph. The run time of this phase is proportional to the product of the number of entries in the album and probe galleries.

The program allocates its memory on reading the run parameters (see below), and makes use of it while generating the graphs. During each recognition step, practically all of the code is exercised.

188.ammp The benchmark runs molecular dynamics (i.e. solves the ODE defined by Newton's equations for the motions of the atoms in the system) on a protein-inhibitor complex which is embedded in water. The energy is approximated by a classical potential or "force field". The protein is HIV protease complexed with the inhibitor indinavir. There are 9,582 atoms in the water and protein making this representative of a typical large simulation. This benchmark is derived from published work on understanding drug resistance in HIV.

189.lucas Performs the Lucas-Lehmer test to check primality of Mersenne numbers $2^p - 1$, using arbitrary-precision (array-integer) arithmetic. Accomplishes the Mersenne-mod squaring via the discrete weighted transform technique of Crandall and Fagin. Uses a data-local, cache-friendly FFT to efficiently perform the large-integer squaring of the Lucas-Lehmer iterations.

*191.fma3d*3d (FMA-3D) FMA-3D is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads. As an explicit code, the program is appropriate for problems where high rate dynamics or stress wave propagation effects are important. In contrast to programs using implicit time integration algorithms, the program uses a large number of relatively small time steps, with the solution for the next configuration of the body being explicit (and inexpensive) at each step. To further reduce the computational effort, the program has a complete implementation of Courant subcycling in which each element is integrated with the maximum time step permitted by local stability criteria. For simulations that have large differences in element critical time steps over the mesh, very significant savings in execution time are achieved. There are no inherent limits on the size of an analysis model, and storage allocation is dynamic within the code.

The program may be applied to static and quasi-static problems either by using the dynamic relaxation option or by simply applying the external loads slowly and integrating the dynamics equations until all significant transients have died out.

The algorithms and architecture of the program are designed for accuracy and robustness. The solution portion of the program is in a form suitable for cache-based computer hardware architectures. The program is written in Fortran-90 and consists of over 50,000 lines of code and comments. The program's key features include:

- Innovative techniques for managing large model databases
- A complete library of finite elements including rigid bodies
- An extensive assortment of constraint and loading options
- Practical sliding interfaces for multi-component analyses
- Extensive options for model modification during a simulation
- Transient dynamic analysis using Courant subcycling
- One-, two- and three-dimensional strain gauges.

200.sixtrack The function of the program is to track a variable number of particles for a variable number of turns round a model of a particle accelerator such as the Large Hadron Collider (LHC) to check the Dynamic Aperture (DA) i.e. the long term stability of the beam.

301.apsi Program to solve for the mesoscale and synoptic variations of potential temperature, U AND V wind components, and the mesoscale vertical velocity W pressure and distribution of pollutants C having sources Q. The synoptic scale components are in quassi-steady state balance, while the mesoscale pressure and velocity W are found diagnostically.

The solution of the complete system is performed by using the splitting-up method. In specific the horizontal advection is carried out by an explicit leapfrog scheme, the horizontal diffusion is performed by the method of the eigenvalues, or equvalently by vertical multiplying the fourier coefficients by appropriate exponentials. The vertical diffusion is treated with a semi-implicit pade-cranknickolson, as well as the vertical advection. The pressure derivative terms are treated with the so-called pressure averaging technique. Finally other terms will be dubbed in the advection part (coriolis). The model calculates prognostically the potential temperature, U,V wind components and concentrations of pollutants C (POTT,UX,VY,C). The pressure and the vertical velocity will be calculated diagnostically (PRES,WZ). The diffusivities are also calculated diagnostically using information on UX,VY, POTT.

D.3 Test benchmarks

The *test* benchmark suite includes two floating-point intensive benchmarks written in *C* by John Walker, used to evaluate an ADVS enabled system. Both benchmarks *fbench* and *ffbench* are run with default parameters.

D.3.1 fbench

fbench is a small benchmark written in *C* by John Walker, implementing raytracing [Walker, 1980]. The benchmark is described by the author:

> *fbench* is a complete optical design raytracing algorithm, shorn of its user interface and recast into portable *C*. It not only determines execution speed on an extremely floating point (including trigonometric functions) intensive real-world application, it checks accuracy on an algorithm that is exquisitely sensitive to errors. The performance of this program is typically far more sensitive to changes in the efficiency of the trigonometric library routines than the average floating point program.
>
> The benchmark may be compiled in two modes. If the symbol `INTRIG` is defined, built-in trigonometric and square root routines will be used for all calculations. Timings made with `INTRIG` defined reflect the machine's basic floating point performance for the arithmetic operators. If `INTRIG` is not defined, the system library *math.h*

functions are used. Results with `INTRIG` not defined reflect the system's library performance and/or floating point hardware support for trig functions and square root. Results with `INTRIG` defined are a good guide to general floating point performance, while results with `INTRIG` undefined indicate the performance of an application which is math function intensive.

D.3.2 ffbench

ffbench is a small benchmark that performs an *FFT* [Walker, 1989].

Ffbench executes a specified number of passes (default 20) through a loop in which each iteration performs a fast Fourier transform of a square matrix (default size 256 × 256) of complex numbers (default precision `double`), followed by the inverse transform. After all loop iterations are performed the results are checked against known correct values.

This benchmark is intended for use on *C* implementations which define `int` as 32 bits or longer and permit allocation and direct addressing of arrays larger than one megabyte.

Appendix E

Arithmetic VHDL source code

"It ended up being so slow it could do hardly anything, and we had to abandon it. But at least it was an aggressive shot—one that we just didn't target correctly."

Gordon Moore (1929 —)

E.1 32 bit (7; 2) unsigned multiplier

```
--------------------------------------------------------------------------------
-- File        : unsigned_compressor_mult_32b_by_32b_7_to_2.vhd
-- Author      : This file was generated with cmultgen, by Dan Kelly
-- Company     : University of Adelaide
-- Date        : Fri Feb 20 14:12:43 2009

--------------------------------------------------------------------------------
-- Copyright (c) 2008 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- An unsigned 32 bit by 32 bit multiplier, based on 7:2 compessors
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library compressorLib;
use compressorLib.compressorLib.all;

library arith_lib;
use arith_lib.arith_lib.all;

--------------------------------------------------------------------------------

entity mult is

  port (
    multA  : in  std_logic_vector (31 downto 0);
    multB  : in  std_logic_vector (31 downto 0);
    PROD   : out std_logic_vector (63 downto 0)
    );

end mult;

--------------------------------------------------------------------------------

architecture arch of mult is
  signal          a0_b0 : std_logic; -- partial product
  signal          a1_b0 : std_logic; -- partial product
  signal          a2_b0 : std_logic; -- partial product
  signal          a3_b0 : std_logic; -- partial product
  signal          a4_b0 : std_logic; -- partial product
  -- ...
  -- <etc>
  -- ...
  signal        a30_b31 : std_logic; -- partial product
  signal        a31_b31 : std_logic; -- partial product

  signal            c0i : std_logic_vector(6 downto 0); -- compressor 0 input
  signal             c0 : std_logic_vector(1 downto 0); -- compressor 0 output
  signal            c1i : std_logic_vector(6 downto 0); -- compressor 1 input
  -- ...
  -- <etc>
  -- ...
  signal           c291 : std_logic_vector(1 downto 0); -- compressor 291 output
  signal          c292i : std_logic_vector(6 downto 0); -- compressor 292 input
  signal           c292 : std_logic_vector(1 downto 0); -- compressor 292 output
  signal       cpaAdd_A : std_logic_vector(59 downto 0);
  signal       cpaAdd_B : std_logic_vector(59 downto 0);
  signal     cpaAdd_SUM : std_logic_vector(59 downto 0);
  signal        product : std_logic_vector(63 downto 0);
begin

  a0_b0 <= multA(0) AND multB(0);
  a1_b0 <= multA(1) AND multB(0);
  a2_b0 <= multA(2) AND multB(0);
  -- ...
  -- <etc>
  -- ...
  a30_b31 <= multA(30) AND multB(31);
  a31_b31 <= multA(31) AND multB(31);

  c0i <= a1_b0 & a0_b1 & ’0’ & ’0’ & ’0’ & ’0’ & ’0’;

  compressor0 : compressor_7_to_2
    port map (
      X => c0i,
      Y => c0);

  c1i <= a2_b0 & a1_b1 & a0_b2 & ’0’ & ’0’ & ’0’ & ’0’;

  -- ...
  -- <etc>
  -- ...

  compressor291 : compressor_7_to_2
    port map (
      X => c291i,
      Y => c291);

  c292i <= c231(1) & c232(0) & '0' & '0' & '0' & '0' & '0';

  compressor292 : compressor_7_to_2
    port map (
      X => c292i,
      Y => c292);

  cpaAdd_A <= c232(1) & c291(1) & c290(1) & c289(1) & c288(1) & c287(1) & c286(1) & c285(1) & c284(1) & c283(1) &
        c282(1) & c281(1) & c280(1) & c279(1) & c278(1) & c277(1) & c276(1) & c275(1) & c274(1) & c273(1) & c272
        (1) & c271(1) & c270(1) & c269(1) & c268(1) & c267(1) & c266(1) & c265(1) & c264(1) & c263(1) & c262(1) &
         c261(1) & c260(1) & c259(1) & c258(1) & c257(1) & c256(1) & c255(1) & c254(1) & c253(1) & c252(1) & c251
        (1) & c250(1) & c249(1) & c248(1) & c247(1) & c246(1) & c245(1) & c244(1) & c243(1) & c242(1) & c241(1) &
         c240(1) & c239(1) & c238(1) & c237(1) & c236(1) & c235(1) & c234(1) & c233(1);
  cpaAdd_B <= c292(1) & c292(0) & c291(0) & c290(0) & c289(0) & c288(0) & c287(0) & c286(0) & c285(0) & c284(0) &
        c283(0) & c282(0) & c281(0) & c280(0) & c279(0) & c278(0) & c277(0) & c276(0) & c275(0) & c274(0) & c273
        (0) & c272(0) & c271(0) & c270(0) & c269(0) & c268(0) & c267(0) & c266(0) & c265(0) & c264(0) & c263(0) &
         c262(0) & c261(0) & c260(0) & c259(0) & c258(0) & c257(0) & c256(0) & c255(0) & c254(0) & c253(0) & c252
        (0) & c251(0) & c250(0) & c249(0) & c248(0) & c247(0) & c246(0) & c245(0) & c244(0) & c243(0) & c242(0) &
         c241(0) & c240(0) & c239(0) & c238(0) & c237(0) & c236(0) & c235(0) & c234(0);

  cpaAdder : Add
    generic map (
      width => 60,
      speed => 2)
    port map (
      A => cpaAdd_A,
      B => cpaAdd_B,
      S => cpaAdd_SUM);

  product(3 downto 0) <= c233(0) & c165(0) & c0(0) & a0_b0;
  product(63 downto 4) <= cpaAdd_SUM;

  PROD <= product;

end arch;

--------------------------------------------------------------------------------
```

E.2 32 bit approximate DI divider

```
------------------------------------------------------------------
-- File        : dan_div_full
-- Author      : Dan Kelly  <dankelly@ieee.org>
-- Company     : University of Adelaide
-- Date        : Aug 26 2008
--------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- Perform the approximating division on operands Z and D.
-- Produce the 32-bit quotient Q. Does not produce a remainder.
-- Signals a division by zero with the divZero line
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library arith_lib;
use arith_lib.arith_lib.all;

use work.divlib.all;

--------------------------------------------------------------------------------

entity dan_div_full is

  port (
    Z       : in std_logic_vector(31 downto 0);    -- dividend
    D       : in std_logic_vector(31 downto 0);    -- divisor
    CLK     : in std_logic;                        -- clock signal
    RESET   : in std_logic;                        -- reset signal
    Q       : out std_logic_vector(31 downto 0);   -- quotient
```

```
                                                  -- no remainder
    divZero : out std_logic                       -- d == 0 -> ’1’
    );

end dan_div_full;

--------------------------------------------------------------------------------

architecture structure of dan_div_full is
  signal initApprox     : std_logic_vector(31 downto 0);
  signal multTerm2      : std_logic_vector(31 downto 0);
  signal rndShamnt      : std_logic_vector( 4 downto 0);
  signal quo_sum        : std_logic_vector(31 downto 0);
  signal quo_carry      : std_logic_vector(31 downto 0);
begin

  ------------------------------------------------------------------------------
  -- INITIALIZATION
  ------------------------------------------------------------------------------

  init_phase: dan_div_full_init
    port map (
      Z          => Z,
      D          => D,
      CLK        => CLK,
      RESET      => RESET,
      initApprox => initApprox,
      multTerm2  => multTerm2,
      rndShamnt  => rndShamnt,
      divZero    => divZero);

  ------------------------------------------------------------------------------
  -- DIVISION
  ------------------------------------------------------------------------------

  div_phase: dan_div_full_div
    port map (
      initApprox => initApprox,
      multTerm2  => multTerm2,
      rndShamnt  => rndShamnt,
      CLK        => CLK,
      RESET      => RESET,
      quo_sum    => quo_sum,
      quo_carry  => quo_carry);

  ------------------------------------------------------------------------------
  -- ACCUMULATION
  ------------------------------------------------------------------------------

  accum_phase: dan_div_full_acc
    port map (
      quo_sum   => quo_sum,
      quo_carry => quo_carry,
      CLK       => CLK,
      RESET     => RESET,
      quo       => Q);

end structure;
--------------------------------------------------------------------------------
```

E.2.1 Initialisation stage

```
-------------------------------------------------------------------
-- File         : dan_div_full_init
-- Author       : Dan Kelly  <dankelly@ieee.org>
-- Company      : University of Adelaide
-- Date         : Aug 26 2008
--------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- Perform the approximating division on operands Z and D.
-- Produce the 32-bit quotient Q. Does not produce a remainder.
-- Signals a division by zero with the divZero line
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library arith_lib;
use arith_lib.arith_lib.all;

use work.divlib.all;

--------------------------------------------------------------------------------

entity dan_div_full_init is
```

```
  port (
    Z          : in  std_logic_vector(31 downto 0);  -- dividend
    D          : in  std_logic_vector(31 downto 0);  -- divisor
    CLK        : in  std_logic;                      -- clock signal
    RESET      : in  std_logic;                      -- reset signal
    initApprox : out std_logic_vector(31 downto 0);  -- quotient
    multTerm2  : out std_logic_vector(31 downto 0);
    rndShamnt  : out std_logic_vector( 4 downto 0);
    divZero    : out std_logic);

end dan_div_full_init;

--------------------------------------------------------------------------------

architecture structure of dan_div_full_init is
  -- initialization
  signal lzd_d           : std_logic_vector(31 downto 0);
  signal lzd_dmr1        : std_logic_vector(31 downto 0);  -- mask r-shift 1
  signal lzd_dmr1_and_d  : std_logic_vector(31 downto 0);
  signal initShamnt      : std_logic_vector( 4 downto 0);
  signal initApprox_sig  : std_logic_vector(31 downto 0);
  signal initApprox_sig_lat : std_logic_vector(31 downto 0);
  signal multTerm2_en    : std_logic;                      -- enable multTerm2
  signal numTerm_A       : std_logic_vector(31 downto 0);
  signal numTerm_B       : std_logic_vector(31 downto 0);
  signal numTerm_CI      : std_logic;
  signal numTerm_S       : std_logic_vector(31 downto 0);
  signal numTerm_S_lat   : std_logic_vector(31 downto 0);
  signal numTerm_CO      : std_logic;
  signal lzd_numTerm     : std_logic_vector(31 downto 0);
  signal lzd_numTerm_mr1 : std_logic_vector(31 downto 0);
  signal lzd_nT_mr1_and  : std_logic_vector(31 downto 0);
  signal d_masked        : std_logic_vector(31 downto 0);
  signal d_masked_enc    : std_logic_vector( 4 downto 0);
  signal d_masked_enc_lat: std_logic_vector( 4 downto 0);
  signal nextLead        : std_logic_vector( 4 downto 0);
  signal rndShamntOut    : std_logic_vector( 4 downto 0);

begin

  -- div by zero ?
  divZero <= ’1’ when D="00000000000000000000000000000000"
             else ’0’;

  ------------------------------------------------------------------------------
  -- INITIALIZATION
  ------------------------------------------------------------------------------

  lzd_of_d: LeadZeroDetMask
    generic map (
      width => 32,
      speed => 2)
    port map (
      A     => D,
      Z     => lzd_d,
      Z_mr1 => lzd_dmr1);

  lzd_dmr1_and_d <= lzd_dmr1 and D;

  numTerm_A <=  (lzd_dmr1_and_d(30 downto 0) & ’0’)  --(2^B - d)
              when lzd_dmr1_and_d=(31 downto 0 => ’0’) else
               D;                                    --(d - 2^A)

  numTerm_B <= not(D)                                --(2^B - d)
             when lzd_dmr1_and_d=(31 downto 0 => ’0’) else
               lzd_d;                                --(d - 2^A)

  numTerm_CI <= ’1’;                    -- always a subtraction

  numTerm_add: AddCfast                 -- or AddC?
    generic map (
      width => 32,
      speed => 2)
    port map (
      A  => numTerm_A,
      B  => numTerm_B,
      CI => numTerm_CI,
      S  => numTerm_S,
      CO => numTerm_CO);

  numTerm_S_latch: for i in 31 downto 0 generate
    numTerm_S_latches: dFFReset
      port map (
        D   => numTerm_S(i),
        R   => RESET,
        CLK => CLK,
        Q   => numTerm_S_lat(i));
  end generate numTerm_S_latch;

  lzd_of_numTerm: LeadZeroDetMask
    generic map (
      width => 32,
      speed => 2)
    port map (
      A     => numTerm_S_lat,
      Z     => lzd_numTerm,
      Z_mr1 => lzd_numTerm_mr1);

  lzd_nT_mr1_and <= lzd_numTerm_mr1 and numTerm_S_lat;

  multTerm2_en <= '0' when lzd_nT_mr1_and=(31 downto 0 => '0')
                  else '1';

  enc_initShamnt: Encode
    generic map (
      width => 32)
    port map (
      A => lzd_d,
      Z => initShamnt);

  initApprox_sig <= std_logic_vector(shift_right(unsigned(Z),
                                           to_integer(unsigned(initShamnt))));

  initApprox_sig_latch: for i in 31 downto 0 generate
    initApprox_sig_latches: dFFReset
      port map (
        D   => initApprox_sig(i),
        R   => RESET,
        CLK => CLK,
        Q   => initApprox_sig_lat(i));
  end generate initApprox_sig_latch;

  multTerm2 <= ('0' & initApprox_sig_lat(30 downto 0)) when multTerm2_en='1'
               else
               (31 downto 0 => '0');

  enc_nextLead: Encode
    generic map (
      width => 32)
    port map (
      A => lzd_numTerm,
      Z => nextLead);

  d_masked <= lzd_dmr1;

  enc_d_masked: Encode
    generic map (
      width => 32)
    port map (
      A => d_masked,
      Z => d_masked_enc);

  d_masked_enc_latch: for i in 4 downto 0 generate
    d_masked_enc_latches: dFFReset
      port map (
        D   => d_masked_enc(i),
        R   => RESET,
        CLK => CLK,
        Q   => d_masked_enc_lat(i));
  end generate d_masked_enc_latch;

  rndShamnt_sub: Add
    generic map (
      width => 5,
      speed => 2)
    port map (
      A => d_masked_enc_lat,
      B => nextLead,
      S => rndShamntOut);

  initApprox <= initApprox_sig_lat;

  rndShamnt  <= rndShamntOut;

end structure;
--------------------------------------------------------------------------------
```

E.2.2 Division stage

```
------------------------------------------------------------------
-- File        : dan_div_full_div
-- Author      : Dan Kelly  <dankelly@ieee.org>
-- Company     : University of Adelaide
-- Date        : Aug 26 2008
--------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- Perform the approximating division on operands Z and D.
-- Produce the 32-bit quotient Q. Does not produce a remainder.
-- Signals a division by zero with the divZero line
--------------------------------------------------------------------------------
```

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library arith_lib;
use arith_lib.arith_lib.all;

use work.divlib.all;

-------------------------------------------------------------------------------

entity dan_div_full_div is

  port (
    initApprox : in  std_logic_vector(31 downto 0);
    multTerm2  : in  std_logic_vector(31 downto 0);
    rndShamnt  : in  std_logic_vector( 4 downto 0);
    CLK        : in  std_logic;
    RESET      : in  std_logic;
    quo_sum    : out std_logic_vector(31 downto 0);
    quo_carry  : out std_logic_vector(31 downto 0));

end dan_div_full_div;

-------------------------------------------------------------------------------

architecture structure of dan_div_full_div is
  signal x_sum          : std_logic_vector(31 downto 0);
  signal x_carry        : std_logic_vector(31 downto 0);
  signal x_sum_rs       : std_logic_vector(31 downto 0);
  signal x_carry_rs     : std_logic_vector(31 downto 0);
  signal quo_sum_sig    : std_logic_vector(31 downto 0);
  signal quo_carry_sig  : std_logic_vector(31 downto 0);
  signal csa_tree_in    : std_logic_vector(127 downto 0);
  signal csa_sum        : std_logic_vector(31 downto 0);
  signal csa_carry      : std_logic_vector(31 downto 0);
  signal x_sum_latch_in : std_logic_vector(31 downto 0);
  signal x_car_latch_in : std_logic_vector(31 downto 0);
  signal q_sum_latch_in : std_logic_vector(31 downto 0);
  signal q_car_latch_in : std_logic_vector(31 downto 0);
  signal rndShamnt_lat  : std_logic_vector( 4 downto 0);

begin

  -----------------------------------------------------------------------------
  -- DIVISION
  -----------------------------------------------------------------------------

  x_sum_latch_in <= initApprox when RESET=’1’
                    else x_sum_rs;

  -- x_sum
  x_sum_latch: for i in 31 downto 0 generate
    x_sum_latch_i: dFFReset
      port map (
        D   => x_sum_latch_in(i),
        R   => RESET,
        CLK => CLK,
        Q   => x_sum(i));
  end generate x_sum_latch;

  x_sum_rs <= std_logic_vector(shift_right(unsigned(x_sum),
                                           to_integer(unsigned(rndShamnt))));

  x_car_latch_in <= multTerm2 when RESET=’1’
                    else x_carry_rs;
  -- x_carry
  x_carry_latch: for i in 31 downto 0 generate
    x_carry_latch_i: dFFReset
      port map (
        D   => x_car_latch_in(i),
        R   => RESET,
        CLK => CLK,
        Q   => x_carry(i));
  end generate x_carry_latch;

  -- latch the rndShamnt
  rndShamnt_latch: for i in 4 downto 0 generate
    rndShamnt_latch_i: dFFReset
      port map (
        D   => rndShamnt(i),
        R   => RESET,
        CLK => CLK,
        Q   => rndShamnt_lat(i));
  end generate rndShamnt_latch;

  x_carry_rs <= std_logic_vector(shift_right(unsigned(x_carry),
                                           to_integer(unsigned(rndShamnt_lat))));

  q_sum_latch_in <= initApprox when RESET=’1’
                    else csa_sum;

  -- quo_sum
  quo_sum_latch: for i in 31 downto 0 generate
```

```
    quo_sum_latch_i: dFFReset
      port map (
        D   => q_sum_latch_in(i),
        R   => RESET,
        CLK => CLK,
        Q   => quo_sum_sig(i));
  end generate quo_sum_latch;

  q_car_latch_in <= (31 downto 0 => ’0’) when RESET=’1’
                    else csa_carry;

  -- quo_carry
  quo_carry_latch: for i in 31 downto 0 generate
    quo_carry_latch_i: dFFReset
      port map (
        D   => q_car_latch_in(i),
        R   => RESET,
        CLK => CLK,
        Q   => quo_carry_sig(i));
  end generate quo_carry_latch;

  -- concatenate all inputs
  csa_tree_in <= x_sum_rs & x_carry_rs & quo_sum_sig & quo_carry_sig;

  -- CSA
  csa_tree: AddMopCsv
    generic map (
      width => 32,
      depth => 4,
      speed => 2)
    port map (
      A => csa_tree_in,
      S => csa_sum,
      C => csa_carry);

  quo_sum   <= quo_sum_sig;
  quo_carry <= quo_carry_sig;

end structure;

-------------------------------------------------------------------------------
```

E.2.3 Accumulation stage

```
-------------------------------------------------------------------
-- File        : dan_div_full_acc
-- Author      : Dan Kelly  <dankelly@ieee.org>
-- Company     : University of Adelaide
-- Date        : Aug 26 2008
-------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
-------------------------------------------------------------------------------
-- Description :
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library arith_lib;
use arith_lib.arith_lib.all;

use work.divlib.all;

-------------------------------------------------------------------------------

entity dan_div_full_acc is

  port (
    quo_sum   : in  std_logic_vector(31 downto 0);
    quo_carry : in  std_logic_vector(31 downto 0);
    CLK       : in  std_logic;
    RESET     : in  std_logic;
    quo       : out std_logic_vector(31 downto 0));

end dan_div_full_acc;

-------------------------------------------------------------------------------

architecture structure of dan_div_full_acc is
  signal cpa_q          : std_logic_vector(31 downto 0);
begin

  -----------------------------------------------------------------------------
  -- ACCUMULATION
  -----------------------------------------------------------------------------

  cpa_add: Add
```

```
    generic map (
      width => 32,
      speed => 2)
    port map (
      A => quo_sum,
      B => quo_carry,
      S => cpa_q);

  quo <= cpa_q;

end structure;
--------------------------------------------------------------------------------
```

E.3 32 bit floating point adder/subtractor

```
--------------------------------------------------------------------------------
-- Title       : 32-bit IEEE floating point adder
-- Project     : MIPS R4400
--------------------------------------------------------------------------------
-- File        : fpAdderSng.vhd
-- Author      : Dan Kelly  <dankelly@eleceng.adelaide.edu.au>
-- Company     : University of Adelaide
-- Date        : 01/11/2005
--------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- An IEEE floating point adder
-- Stages are together in one unit, will be pipelined later
-- Control bits
--      subtract is like Cin bit: 0->add, 1->subtract
-- Round bits
--      "00"->round towards nearest (default)
--      "01"->round towards +infinity
--      "10"->round towards -infinity
--      "11"->round towards 0
-- EOP (effective operation)
--      1->add, 0->subtract
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

library mips_lib;
use mips_lib.EX_comp.all;
use mips_lib.GEN_comp.all;
use mips_lib.FPU_comp.all;
--use mips_lib.mips_utils.all;
--use mips_lib.image_pkg.all;

--------------------------------------------------------------------------------

entity fpAdderSng is

  port (
    A         : in  std_logic_vector(31 downto 0);
    B         : in  std_logic_vector(31 downto 0);
    subtract  : in  std_logic;
    round     : in  std_logic_vector(1 downto 0);
    Z         : out std_logic_vector(31 downto 0);
    overflow  : out std_logic;
    underflow : out std_logic;
    -- divZero   : out std_logic;
    inexact   : out std_logic;
    invalid   : out std_logic);

end fpAdderSng;

--------------------------------------------------------------------------------

architecture structure of fpAdderSng is

  signal sgnA        : std_logic;
  signal sgnB        : std_logic;
  signal expA        : std_logic_vector( 7 downto 0);
  signal expB        : std_logic_vector( 7 downto 0);
  signal manA        : std_logic_vector(23 downto 0);  -- inc hidden bit
  signal manB        : std_logic_vector(23 downto 0);  -- inc hidden bit
  signal muxExp      : std_logic_vector( 7 downto 0);
  signal expCmp      : std_logic;
  signal expCmpBar   : std_logic;
  signal zeroD       : std_logic;
  signal ZisSpec     : std_logic;
  signal specZ       : std_logic_vector(31 downto 0);
  signal denormA     : std_logic;
  signal denormB     : std_logic;
  signal zeroA       : std_logic;
  signal zeroB       : std_logic;
  signal zeroes      : std_logic;
  signal invalidOp   : std_logic;
  signal noSft       : std_logic;
  signal expDiffOut  : std_logic_vector( 7 downto 0);
  signal swpAOut     : std_logic_vector(26 downto 0);
  signal swpBOut     : std_logic_vector(23 downto 0);
  signal rShiftOut   : std_logic_vector(26 downto 0);
  signal AInvOut     : std_logic_vector(26 downto 0);
  signal BInvOut     : std_logic_vector(26 downto 0);
  signal eop         : std_logic;
  signal invB        : std_logic;
  signal invA        : std_logic;
  signal Cin         : std_logic;
  signal manGT       : std_logic;
  signal manEq       : std_logic;
  signal manCmp      : std_logic;
  signal manSum      : std_logic_vector(26 downto 0);
  signal manOvf      : std_logic;
  signal normShamnt  : std_logic_vector( 4 downto 0);
  signal manNorm     : std_logic_vector(26 downto 0);
  signal rndOvf      : std_logic;
  signal sgnZ        : std_logic;
  signal rndLoss     : std_logic;
  signal rndOut      : std_logic_vector(22 downto 0);
  signal preSftLoss  : std_logic;
  signal postSftLoss : std_logic;
  signal result      : std_logic_vector(31 downto 0);
  signal expUpdOut   : std_logic_vector( 7 downto 0);
  signal specOvf     : std_logic;
  signal expOvf      : std_logic;

begin

  inspect : fpSpecCaseDetect
    port map (
      subtract  => subtract,
      A         => A,
      B         => B,
      ZisSpec   => ZisSpec,
      invalidOp => invalidOp,
      specZ     => specZ,
      denormA   => denormA,
      denormB   => denormB,
      zeroA     => zeroA,
      zeroB     => zeroB);

  zeroes <= zeroA and zeroB;

  -- reinstate the hidden bit (if not a denormal)
  manA(23) <= ’1’ when (denormA=’0’ and zeroA=’0’) else
              ’0’;
  manB(23) <= ’1’ when (denormB=’0’ and zeroB=’0’) else
              ’0’;

  -- sign, exp, mantissas
  manA(22 downto 0) <= A(22 downto 0);
  manB(22 downto 0) <= B(22 downto 0);
  expA <= A(30 downto 23);
  expB <= B(30 downto 23);
  sgnA <= A(31);
  sgnB <= B(31);

  swap : fpAddSwap
    port map (
      Ain  => manA,
      Bin  => manB,
      swp  => expCmpBar,
      Aout => swpAOut(26 downto 3),
      Bout => swpBOut);

  manCompare : genGECmp2
    generic map (
      n  => 24)
    port map (
      A  => manA,
      B  => manB,
      GT => manGT,
      ET => manEq);

  manCmp <= manGT or manEq;

  bitInvCtrl : fpAddBitInvCtrl
    port map (
      eop    => eop,
      cmp    => manCmp,
      zero_d => zeroD,
      inv_a  => invA,
      inv_b  => invB,
      sub    => Cin);

```

```
expDiff : fpAddExpDiff
  port map (
    Ex  => expA,
    Ey  => expB,
    cmp => expCmp,
    d   => expDiffOut);

zeroDDetect : genEqCmp
   generic map (
     n => 8)
   port map (
     A => expDiffOut,
     B => "00000000",
     Z => zeroD);

expSelect : fpAddMux
  port map (
    A   => expA,
    B   => expB,
    sel => expCmpBar,
    Z   => muxExp);

expCmpBar <= not(expCmp);

rShift : fpAddRShift
  port map (
    A       => swpBOut,
    shamnt  => expDiffOut,
    Z       => rShiftOut,
    sftLoss => preSftLoss);

swpAOut(2 downto 0) <= "000";

AInv : genCondInv
  generic map (
    n => 27)
  port map (
    A   => swpAOut,
    inv => invA,
    Z   => AInvOut);

BInv : genCondInv
  generic map (
    n   => 27)
  port map (
    A   => rShiftOut,
    inv => invB,
    Z   => BInvOut);

eopLogic : fpAddEOPLogic
  port map (
    op     => subtract,
    sign_A => sgnA,
    sign_B => sgnB,
    EOP    => eop);

manAdd : fpAddManAdd_zim
  port map (
    A   => AInvOut,
    B   => BInvOut,
    Cin => Cin,
    Z   => manSum,
    ovf => manOvf);

LR1Shift : fpAddLR1Shift
  port map (
    A           => manSum,
    rs          => manOvf,
    num         => normShamnt,
    Z           => manNorm,
    postSftLoss => postSftLoss);

noSft <= (zeroA or denormA) and (zeroB or denormB);

LOD : fpAddLOD
  port map (
    A     => manSum,
    noSft => noSft,
    Z     => normShamnt);

rounding : fpManRnd
  port map (
    A       => manNorm,
    sgn     => sgnZ,
    rnd     => round,
    Z       => rndOut,
    expOvf  => rndOvf,
    rndLoss => rndLoss);

result(22 downto 0) <= rndOut;

expUpd : fpAddExpUpd
  port map (
    E       => muxExp,
    man_ovf => manOvf,
      shamnt  => normShamnt,
      rnd_ovf => rndOvf,
      Z       => expUpdOut,
      ovf     => expOvf,
      undf    => underflow);

  result(30 downto 23) <= expUpdOut;

  sign : fpAddSignLogic
    port map(
      sgnX   => sgnA,
      sgnY   => sgnB,
      expCmp => expCmp,
      idExp  => zeroD,
      sub    => subtract,
      manCmp => manCmp,
      sgnZ   => sgnZ);

  -- sign(Z) is needed for rounding module and special case detection
  result(31) <= sgnZ;

  specialCase : fpAddSpecCase
    port map (
      ZCalc     => result,
      ZisSpec   => ZisSpec,
      specZ     => specZ,
      ovf       => expOvf,
      manEq     => manEq,
      expEq     => zeroD,
      eop       => eop,
      round     => round,
      zeroes    => zeroes,
      Zout      => Z);

  overflow <= expOvf;
  inexact  <= rndLoss or preSftLoss or postSftLoss;
  invalid  <= invalidOp;

end structure;

--------------------------------------------------------------------------------
```

E.4 32 bit FP multiplier

```
--------------------------------------------------------------------------------
-- Title       : 32-bit IEEE floating point multiplier
-- Project     : MIPS R4400
--------------------------------------------------------------------------------
-- File        : fpMult.vhd
-- Author      : Dan Kelly  <dankelly@eleceng.adelaide.edu.au>
-- Company     : University of Adelaide
-- Date        : 01/04/2009
--------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- An IEEE floating point adder
-- Stages are together in one unit, will be pipelined later
-- Control bits
--      subtract is like Cin bit: 0->add, 1->subtract
-- Round bits
--      "00"->round towards nearest (default)
--      "01"->round towards +infinity
--      "10"->round towards -infinity
--      "11"->round towards 0
-- EOP (effective operation)
--      1->add, 0->subtract
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library mips_lib;
use mips_lib.EX_comp.all;
use mips_lib.GEN_comp.all;
use mips_lib.FPU_comp.all;

library arith_lib;
use arith_lib.arith_lib.all;

--------------------------------------------------------------------------------

```

```
entity fpMult is

  port (
    A         : in  std_logic_vector(31 downto 0);
    B         : in  std_logic_vector(31 downto 0);
    round     : in  std_logic_vector(1 downto 0);
    Z         : out std_logic_vector(31 downto 0);
    overflow  : out std_logic;
    underflow : out std_logic;
    inexact   : out std_logic;
    invalid   : out std_logic);

end fpMult;

--------------------------------------------------------------------------------

architecture structure of fpMult is

  signal sgnA            : std_logic;
  signal sgnB            : std_logic;
  signal expA            : std_logic_vector( 7 downto 0);
  signal expB            : std_logic_vector( 7 downto 0);
  signal manA            : std_logic_vector(23 downto 0);  -- inc hidden bit
  signal manB            : std_logic_vector(23 downto 0);  -- inc hidden bit
  signal muxExp          : std_logic_vector( 7 downto 0);
  signal expCmp          : std_logic;
  signal expCmpBar       : std_logic;
  signal zeroD           : std_logic;
  signal ZisSpec         : std_logic;
  signal specZ           : std_logic_vector(31 downto 0);
  signal denormA         : std_logic;
  signal denormB         : std_logic;
  signal zeroA           : std_logic;
  signal zeroB           : std_logic;
  signal zeroes          : std_logic;
  signal invalidOp       : std_logic;
  signal noSft           : std_logic;
  signal expDiffOut      : std_logic_vector( 7 downto 0);
  signal swpAOut         : std_logic_vector(26 downto 0);
  signal swpBOut         : std_logic_vector(23 downto 0);
  signal rShiftOut       : std_logic_vector(26 downto 0);
  signal AInvOut         : std_logic_vector(26 downto 0);
  signal BInvOut         : std_logic_vector(26 downto 0);
  signal eop             : std_logic;
  signal invB            : std_logic;
  signal invA            : std_logic;
  signal Cin             : std_logic;
  signal manGT           : std_logic;
  signal manEq           : std_logic;
  signal manCmp          : std_logic;
  signal manSum          : std_logic_vector(26 downto 0);
  signal manOvf          : std_logic;
  signal normShamnt      : std_logic_vector( 5 downto 0);
  signal manNorm         : std_logic_vector(26 downto 0);
  signal rndOvf          : std_logic;
  signal sgnZ            : std_logic;
  signal rndLoss         : std_logic;
  signal rndOut          : std_logic_vector(22 downto 0);
  signal preSftLoss      : std_logic;
  signal postSftLoss     : std_logic;
  signal result          : std_logic_vector(31 downto 0);
  signal expUpdOut       : std_logic_vector( 7 downto 0);
  signal specOvf         : std_logic;
  signal manProd         : std_logic_vector(47 downto 0);
  signal manProdNorm     : std_logic_vector(47 downto 0);
  signal prodLeadOne     : std_logic_vector(47 downto 0);
  signal prodLeadOneRev  : std_logic_vector(47 downto 0);
  signal prodLeadOneEnc  : std_logic_vector( 5 downto 0);
  signal expSum          : std_logic_vector( 7 downto 0);
  signal updExp          : std_logic_vector( 7 downto 0);
  signal manZ            : std_logic_vector(22 downto 0);
  signal undf            : std_logic;
  signal ovf             : std_logic;
  signal expOvf          : std_logic;
  signal expUpdOvf       : std_logic;
  signal expUnd          : std_logic;
  signal normLoss        : std_logic;

begin

  inspect : fpMultSpecCaseDetect
    port map (
      fpA       => A,
      fpB       => B,
      ZisSpec   => ZisSpec,
      invalidOp => invalidOp,
      specZ     => specZ,
      denormA   => denormA,
      denormB   => denormB,
      zeroA     => zeroA,
      zeroB     => zeroB);

  zeroes <= zeroA and zeroB;

  sgnZ <= sgnA xnor sgnB;
```

```

-- reinstate the hidden bit (if not a denormal, or a zero)
manA(23) <= ’1’ when (denormA=’0’ and zeroA=’0’) else
            ’0’;
manB(23) <= ’1’ when (denormB=’0’ and zeroB=’0’) else
            ’0’;

-- sign, exp, mantissas
manA(22 downto 0) <= A(22 downto 0);
manB(22 downto 0) <= B(22 downto 0);
expA <= A(30 downto 23);
expB <= B(30 downto 23);
sgnA <= A(31);
sgnB <= B(31);

expAdd : fpMultExp
  port map (
    Ea => expA,
    Eb => expB,
    ovf => expOvf,
    und => expUnd,
    Exp => expSum);

-- multiplier from Zimmerman’s lib
manMul : MulUns
  generic map (
    widthX => 24,
    widthY => 24,
    speed => 2)
  port map (
    X => manA,
    Y => manB,
    P => manProd);

-- prods of denormals will need to be shifted left a variable amount
prodLeadOneDet : LeadZeroDet
  generic map (
    width => 48,
    speed => 2)
  port map (
    A => manProd,
    Z => prodLeadOne);

-- reverse the bits to make a L-shamnt
prodLeadOneBitRev : for i in 47 downto 0 generate
  prodLeadOneRev(i) <= prodLeadOne(47-i);
end generate prodLeadOneBitRev;

-- encode the L-shamnt
prodLShamntEnc : Encode
  generic map (
    width => 48)
  port map (
    A => prodLeadOneRev,
    Z => prodLeadOneEnc);

-- left-shift mantissa where necessary
manProdNorm <= std_logic_vector(shift_left(unsigned(manProd),
                          to_integer(unsigned(prodLeadOneEnc))));

-- sticky bit formulation, and rounding
manRnd : fpMultManRnd
port map (
  manProd => manProdNorm,
  sgn     => sgnZ,
  rnd     => round,
  Z       => manZ,
  expOvf  => rndOvf,
  rndLoss => normLoss);

-- update the exponent from shifts and rounding
expUpd : fpMultExpUpd
port map (
  exp    => expSum,
  lShamnt=> prodLeadOneEnc,
  rndOvf => rndOvf,
  Z      => updExp,
  ovf    => expUpdOvf);

-- sign(Z) is needed for rounding module and special case detection
result(31)           <= sgnZ;
result(30 downto 23) <= updExp;
result(22 downto  0) <= manZ;

-- overflow and underflow conditions
 ovf <= expOvf or expUpdOvf;
 undf <= ’1’ when updExp="00000000" and expUnd=’1’
         else
         ’0’;

specialCase : fpMultSpecCase
  port map (
    ZCalc     => result,
    ZisSpec   => ZisSpec,
    specZ     => specZ,
      ovf       => ovf,
      undf      => undf,
      round     => round,
      Zout      => Z);

  overflow  <= ovf;
  inexact   <= normLoss;
  invalid   <= invalidOp;
  underflow <= undf;

end structure;

--------------------------------------------------------------------------------
```

E.5 32 bit FP divider

```
--------------------------------------------------------------------------------
-- Title       : 32-bit IEEE floating point adder
-- Project     : MIPS R4400
--------------------------------------------------------------------------------
-- File        : fpDivSng.vhd
-- Author      : Dan Kelly  <dankelly@eleceng.adelaide.edu.au>
-- Company     : University of Adelaide
-- Date        : 01/11/2005
--------------------------------------------------------------------------------
-- Copyright (c) 2005 University of Adelaide, AUSTRALIA
--------------------------------------------------------------------------------
-- Description :
-- An IEEE floating point adder
-- Stages are together in one unit, will be pipelined later
-- Control bits
--      subtract is like Cin bit: 0->add, 1->subtract
-- Round bits
--      "00"->round towards nearest (default)
--      "01"->round towards +infinity
--      "10"->round towards -infinity
--      "11"->round towards 0
-- EOP (effective operation)
--      1->add, 0->subtract
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library mips_lib;
use mips_lib.EX_comp.all;
use mips_lib.GEN_comp.all;
use mips_lib.FPU_comp.all;

library arith_lib;
use arith_lib.arith_lib.all;

library divgenlib;
use divgenlib.divgenlib.all;

--------------------------------------------------------------------------------

entity fpDivSng is

  port (
    A         : in std_logic_vector(31 downto 0);
    B         : in std_logic_vector(31 downto 0);
    round     : in std_logic_vector( 1 downto 0);
    START     : in std_logic;
    CLK       : in std_logic;
    RESET     : in std_logic;
    Z         : out std_logic_vector(31 downto 0);
    overflow  : out std_logic;
    underflow : out std_logic;
    divZero   : out std_logic;
    inexact   : out std_logic;
    invalid   : out std_logic);

end fpDivSng;

--------------------------------------------------------------------------------

architecture structure of fpDivSng is

  signal sgnA             : std_logic;
  signal sgnB             : std_logic;
  signal expA             : std_logic_vector(7 downto 0);
  signal expB             : std_logic_vector(7 downto 0);
  signal manA             : std_logic_vector(23 downto 0);  -- inc hidden bit
  signal manB             : std_logic_vector(23 downto 0);  -- inc hidden bit
  signal muxExp           : std_logic_vector(7 downto 0);
  signal expCmp           : std_logic;
  signal expCmpBar        : std_logic;
  signal zeroD            : std_logic;
  signal ZisSpec          : std_logic;
  signal specZ            : std_logic_vector(31 downto 0);
  signal denormA          : std_logic;
  signal denormB          : std_logic;
  signal zeroA            : std_logic;
  signal zeroB            : std_logic;
  signal zeroes           : std_logic;
  signal invalidOp        : std_logic;
  signal noSft            : std_logic;
  signal expDiffOut       : std_logic_vector(7 downto 0);
  signal swpAOut          : std_logic_vector(26 downto 0);
  signal swpBOut          : std_logic_vector(23 downto 0);
  signal rShiftOut        : std_logic_vector(26 downto 0);
  signal AInvOut          : std_logic_vector(26 downto 0);
  signal BInvOut          : std_logic_vector(26 downto 0);
  signal eop              : std_logic;
  signal invB             : std_logic;
  signal invA             : std_logic;
  signal Cin              : std_logic;
  signal manGT            : std_logic;
  signal manEq            : std_logic;
  signal manCmp           : std_logic;
  signal manSum           : std_logic_vector(26 downto 0);
  signal manOvf           : std_logic;
  signal normShamnt       : std_logic_vector(5 downto 0);
  signal manNorm          : std_logic_vector(26 downto 0);
  signal rndOvf           : std_logic;
  signal sgnZ             : std_logic;
  signal rndLoss          : std_logic;
  signal rndOut           : std_logic_vector(22 downto 0);
  signal preSftLoss       : std_logic;
  signal postSftLoss      : std_logic;
  signal result           : std_logic_vector(31 downto 0);
  signal expUpdOut        : std_logic_vector(7 downto 0);
  signal specOvf          : std_logic;
  signal expUpdOvf        : std_logic;
  signal normManA         : std_logic_vector(23 downto 0);
  signal normManB         : std_logic_vector(23 downto 0);
  signal denormLeadOne    : std_logic_vector(23 downto 0);
  signal denormLeadOneRev : std_logic_vector(23 downto 0);
  signal denormShamnt5    : std_logic_vector(4 downto 0);
  signal denormShamnt8    : std_logic_vector(7 downto 0);
  signal divisor          : std_logic_vector(26 downto 0);
  signal dividend         : std_logic_vector(26 downto 0);
  signal quotient         : std_logic_vector(26 downto 0);
  signal quotientRnd      : std_logic_vector(22 downto 0);
  signal quoLeadOne       : std_logic_vector(25 downto 0);
  signal quoLeadOneRev    : std_logic_vector(25 downto 0);
  signal quoLShamnt       : std_logic_vector(4 downto 0);
  signal quoNorm          : std_logic_vector(26 downto 0);
  signal expOvf           : std_logic;
  signal expUnd           : std_logic;
  signal expUpdUnd        : std_logic;
  signal expQuoRndOvf     : std_logic;
  signal expOvfSpecCase   : std_logic;
  signal manUnd           : std_logic;
  signal divEnd           : std_logic;

begin

  inspect : fpDivSpecCaseDetect
    port map (
      sngA      => A,
      sngB      => B,
      ZisSpec   => ZisSpec,
      invalidOp => invalidOp,
      specZ     => specZ,
      denormA   => denormA,
      denormB   => denormB,
      zeroA     => zeroA,
      zeroB     => zeroB);

  zeroes <= zeroA and zeroB;

  sgnZ <= sgnA xor sgnB;

  -- reinstate the hidden bit (if not a denormal)
  manA(23) <= ’1’ when (denormA=’0’ and zeroA=’0’) else
              ’0’;
  manB(23) <= ’1’ when (denormB=’0’ and zeroB=’0’) else
              ’0’;

  -- sign, exp, mantissas
  manA(22 downto 0) <= A(22 downto 0);
  manB(22 downto 0) <= B(22 downto 0);
  expA <= A(30 downto 23);
  expB <= B(30 downto 23);
  sgnA <= A(31);
```

```
sgnB <= B(31);

-- both operands are normalised, unless they are denormals
-- shift the (denormal) mantissa B to have a lead one

denormOneDetect : LeadZeroDet
  generic map (
    width => 24,
    speed => 2)
  port map (
    A => manB,
    Z => denormLeadOne);

-- swap the bit order of the denormLeadOneRev
bitSwap: for i in 22 downto 0 generate
  denormLeadOneRev(i) <= denormLeadOne(22-i);
end generate bitSwap;

-- encode to 5-bits, then shift to 8-bits to match exponent
denormShamntEnc : Encode
  generic map (
    width => 24)
  port map (
    A => denormLeadOneRev,
    Z => denormShamnt5);

denormShamnt8 <= std_logic_vector(resize(unsigned(denormShamnt5),8));

-- normalisation shift of the (denomal) manB
normManB <= std_logic_vector(shift_left(unsigned(manB),to_integer(unsigned(denormShamnt8))));

-- select the necesary divisor, concat the rounding bits (GRS)
divisor <= normManB & "000" when denormB=’1’
           else
           manB & "000";

-- the dividend does not need to be normailsed, but concat trailing zeroes
dividend <= manA & "000";

-- parallel prefix adder
divExp : fpDivSngExp
  port map (
    Ea         => expA,
    Eb         => expB,
    manBShamnt => denormShamnt8,
    ovf        => expOvf,
    und        => expUnd,
    Exp        => expDiffOut);

-- division unit, using DIVGEN
manDiv : SRT_divider
  port map (
    dividend => dividend,
    divisor  => divisor,
    clock    => CLK,
    reset    => RESET,
    start    => START,
    quotient => quotient,
    complete => divEnd,
    overflow => manOvf);

-- denormals will need to be shifted left a var amount
-- division can produce a result in [0,2), so may need to right shift 1
quoLeadOneDet : LeadZeroDet
  generic map (
    width => 26,
    speed => 2)
  port map (
    A => quotient(25 downto 0),
    Z => quoLeadOne);

quoLeadOneBitRev : for i in 25 downto 0 generate
  quoLeadOneRev(i) <= quoLeadOne(25-i);
end generate quoLeadOneBitRev;

quoLShamntEnc : Encode
  generic map (
    width => 26)
  port map (
    A => quoLeadOneRev,
    Z => quoLShamnt);

-- correctly normalise
LR1Shift : fpAddLR1Shift
  port map (
    A => quotient,
    rs => quotient(26),
    num => quoLShamnt,
    Z => quoNorm,
    postSftLoss => postSftLoss);

QuoRnd: fpDivSngQuoRnd
port map (
  manQuo  => quotient,
  sgn     => sgnZ,
    rnd     => round,
    Z       => quotientRnd,
    manUndf => manUnd,
    expOvf  => expQuoRndOvf,
    rndLoss => rndLoss);

  expUpd: fpDivExpUpd
    port map (
      E       => expDiffOut,
      man_ovf => '0',
      shamnt  => quoLShamnt,
      rnd_ovf => rndOvf,
      Z       => expUpdOut,
      ovf     => expUpdOvf,
      undf    => expUpdUnd);

  -- sign(Z) is needed for rounding module and special case detection
  result(31) <= sgnZ;
  result(30 downto 23) <= expUpdOut;
  result(22 downto 0) <= quotientRnd;

  expOvfSpecCase <= expOvf or expQuoRndOvf;

  specialCase : fpDivSngSpecCase
    port map (
      ZCalc     => result,
      ZisSpec   => ZisSpec,
      specZ     => specZ,
      ovf       => expOvfSpecCase,
      undf      => expUnd,
      round     => round,
      Zout      => Z);

  overflow  <= expOvf or expQuoRndOvf or expUpdOvf or rndOvf;
  inexact   <= rndLoss or preSftLoss or postSftLoss;
  invalid   <= invalidOp;
  underflow <= expUnd or manUnd or expUpdUnd;
  divZero   <= zeroB;

end structure;

--------------------------------------------------------------------------------
```

Appendix F

SimpleScalar machine instructions

"Low-level programming is good for the programmer's soul."

John Carmack (1970 —)

F.1 SimpleScalar machine instructions

Table F.1: Machine instructions provided in the *PISA* architecture.

Opcode FIRST	Name	Type
abs.d	absolute value FP doubleword	*fpALU* arithmetic
abs.s	absolute value FP	*fpALU* arithmetic
add	add	*iALU* arithmetic
add.d	add FP doubleword	*fpALU* arithmetic
add.s	add FP	*fpALU* arithmetic
addi	add immediate	*iALU* arithmetic
addiu	add immediate unsigned	*iALU* arithmetic
addu	add unsigned	*iALU* arithmetic
and	logic *AND*	*iALU* logic
andi	logic *AND* immediate	*iALU* logic
bc1f	branch on FP false	*iALU*
bc1t	branch on FP true	*iALU*
beq	branch equal	*iALU*
bgez	branch greater than or equal to zero	*iALU*
bgtz	branch greater than zero	*iALU*
blez	branch less than or equal to zero	*iALU*
bltz	branch less than zero	*iALU*
bne	branch not equal	*iALU*
break	unconditional breakpoint exception	*iALU*
c.eq.d	compare and set if equal FP doubleword	*iALU*
c.eq.s	compare and set if equal FP single	*iALU*
c.le.d	compare and set if less than or equal FP doubleword	*iALU*
c.le.s	compare and set if less then or equal FP single	*iALU*
c.lt.d	compare and set if less than FP doubleword	*iALU*
c.lt.s	compare and set if less than FP single	*iALU*
cfc1	copy from FP coprocessor	*fpALU*
ctc1	copy to FP coprocessor	*fpALU*
cvt.d.s	convert FP doubleword to FP single	*fpALU*
cvt.d.w	convert FP doubleword to integer	*fpALU*
cvt.s.d	convert FP single to FP doubleword	*fpALU*
cvt.s.w	convert FP single to integer	*fpALU*
cvt.w.d	convert integer to FP doubleword	*fpALU*
cvt.w.s	convert integer to FP single	*fpALU*
div	divide	*iALU* arithmetic
div.d	divide FP doubleword	*fpALU* arithmetic
div.s	divide FP	*fpALU* arithmetic
divu	divide unsigned	*iALU* arithmetic
dlw		
dmfc1	move from FP coprocessor doubleword	*fpALU*

Table F.1: *continued. . .)*

Opcode	Name	Type
`dmtc1`	move to FP coprocessor doubleword	*fpALU*
`dsw`	store doubleword	memory
`dsw`	store doubleword	memory
`dsz`		
`j`	jump	*iALU*
`jal`	jump and link	*iALU*
`jalr`	jump and link (register)	*iALU*
`jr`	jump (register)	*iALU*
`l.d`	load DP doubleword	memory
`l.s`	load FP	memory
`l.s.r2`		
`lb`	load byte	memory
`lbu`	load byte unsigned	memory
`lh`	load halfword	memory
`lhu`	load halfword unsigned	memory
`lui`	load unsigned immediate	memory
`lw`	load word	memory
`lw.r2`		
`lwl`	load word left	memory
`lwr`	load word right	memory
`mfc1`	move from FP coprocessor	*fpALU*
`mfhi`	move from *HI*	*iALU*
`mflo`	move from *LO*	*iALU*
`mov.d`	move FP doubleword	*fpALU*
`mov.s`	move FP	*fpALU*
`mtc1`	move to FP coprocessor	*fpALU*
`mthi`	move to *HI*	*iALU*
`mtlo`	move to *LO*	*iALU*
`mul.d`	multiply FP doubleword	*fpALU* arithmetic
`mul.s`	multiply FP	*fpALU* arithmetic
`mult`	multiply	*iALU* arithmetic
`multu`	multiply unsigned	*iALU* arithmetic
`neg.d`	negate FP doubleword	*fpALU* arithmetic
`neg.s`	negate FP	*fpALU* arithmetic
`nop`	no operation	*iALU*
`nor`	logic *NOR*	*iALU* logic
`or`	logic *OR*	*iALU* logic
`ori`	logic *OR* immediate	*iALU* logic
`s.d`	store doubleword	memory
`s.s`	store FP	*fpALU* arithmetic
`s.s.r2`		
`sb`	store byte	memory
`sh`	store halfword	memory
`sll`	shift left logical	*iALU*
`sllv`	shift left logical variable	*iALU*
`slt`	set if less than	*iALU*
`slti`	set if less than immediate	*iALU*

Table F.1: *continued. . .)*

Opcode	Name	Type
`sltiu`	set if less the immediate unsigned	*iALU*
`sltu`	set if les than unsigned	*iALU*
`sqrt.d`	square root FP doubleword	*fpALU* arithmetic
`sqrt.s`	square root FP single	*fpALU* arithmetic
`sra`	shift right arithmetic	*iALU* logic
`srav`	shift right arithmetic variable	*iALU* logic
`srl`	shift right logical	*iALU* logic
`srlv`	shift right logical variable	*iALU* logic
`sub`	subtract	*iALU* arithmetic
`sub.d`	subtract FP doubleword	*fpALU* arithmetic
`sub.s`	subtract FP	*iALU* arithmetic
`subu`	subtract unsigned	*iALU* arithmetic
`sw`	store word	memory
`sw.r2`		
`swl`	store word left	memory
`swr`	store word right	memory
`syscall`	system call	
`xor`	logic *XOR*	*iALU* logic
`xori`	logic *XOR* immediate	*iALU* logic

Bibliography

TSMC 0.18µm Process 1.8 Volt SAGE-X™ Standard Cell Library Databook. Artisan Components, Inc., Sunnyvale, CA, USA, February 2002.

Author unknown. Calculation of the digits of pi. *American Mathematical Monthly*, 45, 1938.

Author unknown. Digits of pi calculation. Online: `http://www.codecodex.com/wiki/index.php?title=Digits_of_pi_calculation`, 2008.

Author unknown. Miller-rabin primality test (c). Online: `http://en.literateprograms.org/Special:Downloadcode/Miller-Rabin_primality_test_(C)`, 2009.

C. R. Baugh and B. A. Wooley. A two's complement parallel array multiplication algorithm. *IEEE Transactions on Computers*, C-22:1045–1047, 1973.

R. K. Brayton, G. D. Hachtel, C. T. McCullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer, 1984.

B. E. Briley. Some new results on average worst case carry. *IEEE Transactions On Computers*, C–22 (5):459–463, 1973.

D. Brooks and M. Martonosi. Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance. *ACM Trans. Comput. Syst.*, 18(2):89–126, 2000. ISSN 0734-2071. `http://doi.acm.org/10.1145/350853.350856`.

D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997. ISSN 0163-5964.

N. Burgess. Prenormalization rounding in IEEE floating-point operations using a flagged prefix adder. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(2):266–277, 2005. ISSN 1063-8210.

Bibliography

A. W. Burks, H. H. Goldstein, and J. von Neumann. Preliminiary discussion of the design of an electronic computing instrument. Inst. Advanced Study, Princeton, N. J., June 1946.

L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1110–1115, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. ISBN 3-9810801-0-6.

A. K. Chandra, S. Fortune, and R. Lipton. Unbounded fan-in circuits and associative functions. *JCSS: Journal of Computer and System Sciences*, 30, 1985.

P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 22–31, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. http://doi.acm.org/10.1145/192724.192727.

P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. *SIGARCH Comput. Archit. News*, 25(2):274–283, 1997. ISSN 0163-5964. http://doi.acm.org/10.1145/384286.264209.

X. Cheng and M. S. Hsiao. Region-level approximate computation reuse for power reduction in multimedia applications. In K. Roy and V. Tiwari, editors, *Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005, San Diego, California, USA, August 8-10, 2005*, pages 119–122. ACM, 2005. ISBN 1-59593-137-6.

Alpha Architecure Handbook. Compaq Computer Corporation, version 4 edition, October 1998.

G. Cornetta and J. Cortadella. Radix-16 SRT division unit with speculation of the quotient digits. *Proceedings of the IEEE Great Lakes Symposium on VLSI*, pages 74–77, 1999. ISSN 1066-1395.

S. P. E. Corporation. CFP2000 (floating point component of SPEC CPU2000). Online: http://www.spec.org/cpu2000/CFP2000/, 2000a.

S. P. E. Corporation. CINT2000 (integer component of SPEC CPU2000). Online: http://www.spec.org/cpu2000/CINT2000/, 2000b.

J. Cortadella and T. Lang. High-radix division and square-root with speculation. *IEEE Transactions on Computers*, 43(8):919–931, 1994. ISSN 0018-9340.

C. W. Cowell-Shah. Arithmetic throughput benchmark. Online: http://www.ocf.berkeley.edu/~cowell/research/benchmark/code/Benchmark.c, 2004.

H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, Feb. 1976.

L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356, 1965.

M. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *Linpack*. Philadelphia, 1986.

P. K. B. E. Zimmermann, P. Pattisapu and G. Fettweis. Reduced complexity LDPC decoding using forced convergence. In *Proceedings of the 7th International Symposium on Wireless Personal Multimedia Communications (WPMC'04)*, Abano Terme, Italy, September 2004.

M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, New York, 2003.

P. Fenwick. High-radix division with approximate quotient-digit estimation. *Journal of Universal Computer Science*, 1(1):2–22, January 1995. http://www.jucs.org/jucs_1_1/high_radix_division_with.

M. A. Franklin and T. Pan. Performance comparison of asynchronous adders. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 117–125, Salt Lake City, UT, USA, 1994.

Free Software Foundation, Inc. . Gnu multiple precision arithmetic library. Online: http://gmplib.org/manual/, July 2008.

J. Fritts. Mediabench applications. Online: http://euler.slu.edu/~fritts/mediabench/, 1997.

F. Gabbay. Speculative execution based on value prediction. Technical report, EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

R. G. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1): 21–28, Jan. 1962.

J. Garside. A CMOS VLSI implementation of an asynchronous ALU. *IFIP Transactions A (Computer Science and Technology)*, A-28:181–92, 1993. ISSN 0926-5473.

J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In S. Hong, W. Wolf, K. Flautner, and T. Kim, editors, *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea, October 22-25, 2006*, pages 158–168. ACM, 2006. ISBN 1-59593-543-6.

H. Goldstine and J. von Neumann. On the principles of large-scale computing machines. In *Collected Works of John von Neumann, Vol. 5*, pages 45–46. Pergamon, 1963.

J. Gonzalez and A. Gonzalez. Data value speculation in superscalar processors. *Microprocessors-and-Microsystems*, 22(6):293–301, November 1998.

D. L. Harris, S. F. Oberman, and M. A. Horowitz. SRT division architectures and implementations. *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, pages 18–25, Jul 1997.

H. Hassler and N. Takagi. Function evaluation by table look-up and addition. In *Proc. 12th IEEE Symposium on Computer Arithmetic*, pages 10–16. IEEE, 1995.

R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In F. N. Najm, J. Cong, and D. Blaauw, editors, *ISLPED Low Power Electronics and Design, 1999, San Diego, California, USA, August 16-17, 1999*, pages 30–35. ACM, 1999. ISBN 1-58113-133-X.

H. Hendrickson. Fast high-accuracy binary parallel addition. *IRE – Transactions on Electronic Computers*, EC-9(4):465–469, 1960.

J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000. ISSN 0018-9162.

J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Comput. Archit. News*, 35(1):84–89, 2007. ISSN 0163-5964.

C. Howland and A. Blanksby. A 220 mW 1 gb/s 1024-bit rate-$^1/_2$ low density parity check code decoder. Online `http://gladstone.systems.caltech.edu/~jeremy/other_papers/1GbpsLDPC.pdf`, December 2001.

HP labs. Hp labs: cacti. Online: `http://www.hpl.hp.com/research/cacti/`, 2008.

J. Huang and D. J. Lilja. Extending value reuse to basic blocks with compiler support. *IEEE Trans. Computers*, 49(4):331–347, 2000.

Power ISA™. IBM Corporation, version 2.05 edition, October 2007.

Institute of Electrical and Electronics Engineers. IEEE standard for local and metropolitan area networks — part 16: Air interface for fixed and mobile broadband wireless access systems. IEEE Std 802.16e-2005, Feb. 2006. `http://standards.ieee.org/getieee802/download/802.16e-2005.pdf`.

Intel Corp. World's first 2-billion transistor microprocessor. Online: `http://www.intel.com/technology/architecture-silicon/2billion.htm`, 2009.

W. M. Kahan. Paranoia. online: `http://www.netlib.org/paranoia/paranoia.c`, January 1986. Translated by D. M. Gay and T. Sumner.

D. R. Kelly and B. J. Phillips. Arithmetic data value speculation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3740 LNCS, pages 353–366, Singapore, Singapore, 2005.

D. R. Kelly, B. J. Phillips, and S. F. Al-Sarawi. Increasing throughput of a RISC system using arithmetic data value speculation. In *Conference Record of the Forty-Third Asilomar Conference on Signals, Systems, and Computers, 2009*, Pacific Grove, California, USA, November 2009.

D. J. Kinniment. An evaluation of asynchronous addition. *IEEE Trans. Very Large Scale Integr. Syst.*, 4(1):137–140, 1996. ISSN 1063-8210. `http://dx.doi.org/10.1109/92.486088`.

D. Koes, T. Chelcea, C. Oneyama, and S. C. Goldstein. Adding faster with application specific early termination. School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, January 2005.

R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 330–335, Triangle Park, NC, USA, 1997.

T. Lestable and E. Zimmermann. LDPC options for next generation wireless systems. In *Proceedings of the 14th Wireless World Research Forum (WWRF'05)*, San Diego, USA, July 2005.

A. Li. An empirical study of the longest carry length in real programs. Master's thesis, Department of Computer Science, Princeton University, May 2002.

M. H. Lipasti. *Value locality and speculative execution*. PhD thesis, Pittsburgh, PA, USA, 1998.

M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM Int. Symposium on Microarchitecture*, pages 226–237. IEEE, 1996.

M. H. Lipasti and J. P. Shen. Exploiting value locality to exceed the dataflow limit. *International Journal of Parallel Programming (IJPP)*, 26(4):505–538, Aug. 1998.

T. Liu and S.-L. Lu. Performance improvement with circuit level speculation. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 348–355. IEEE, 2000.

S.-L. Lu. Speeding up processing with approximation circuits. *IEEE Computer Magazine*, 37(3): 67–73, March 2004.

P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. *Proceedings of the Annual International Symposium on Microarchitecture*, pages 230–236, 1999. ISSN 1072-4451. http://dx.doi.org/10.1109/MICRO.1999.809461.

F. McMahon. The Livermore fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, Dec. 1986.

C. Molina, A. Gonzalez, and J. Tubella. Dynamic removal of redundant computations. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 474–481, N.Y., June 1999. ACM Press.

A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Table-based data speculation circuit for parallel processing computer. U. S. Patent 5,781,752, July 1998.

MOSIS Integrated Circuit Fabrication Service. Mosis. Online: http://www.mosis.com, 2009.

MultiGiG, Inc. IRSIM™. Online: http://opencircuitdesign.com/irsim/, 2006.

H. Murakami, N. Yano, Y. Ootaguro, Y. Sugeno, M. Ueno, Y. Muroya, and T. Aramaki. A multiplier-accumulator macro for a 45 MIPS embedded RISC processor. *IEEE Journal of Solid-state Circuits*, 31(7):1067–1071, JULY 1996.

H. Nakano. Method and apparatus for division using interpolation. U. SṖatent 4,707,798, Nov. 1987.

S. M. Nowick, K. Y. Yun, P. A. Beerel, and A. E. Dooply. Speculative completion for the design of high performance asynchronous dynamic adders. In *International Symposium on Advance Research in Asynchronous Circuits and Systems*, pages 210–223, Eindhoven, The Netherlands, 1997. IEEE Comput. Soc. Press.

NVIDIA Corp. TESLA c1060 datasheet. Online: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jun08_FINAL_LowRes.pdf, 2009.

S. F. Oberman and M. J. Flynn. On division and reciprocal caches. Technical report, Stanford, CA, USA, 1995. http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Astan%3ASTAN%2F%2FCSL-TR-95-666.

S. F. Oberman and M. J. Flynn. Reducing division latency with reciprocal caches. *Reliable Computing*, 2(2):147–154, 1996.

S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEETC: IEEE Transactions on Computers*, 46, 1997.

T.-H. Pan, H.-S. Kay, Y. Chun, and C.-L. Wey. High-radix SRT division with speculation of quotient digits. In *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 479–484, Austin, TX, USA, 1995.

B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2000.

B. J. Phillips, D. R. Kelly, and B. W. Ng. Estimating adders for a low density parity check decoder. In *Proceedings of SPIE—The International Society for Optical Engineering*, volume 6313, San Diego, CA, United States, 2006.

N. Pippenger. Analysis of carry propagation in addition: an elementary approach. *Journal of Algorithms*, 42(2):317–333, 2002.

C. Price. *MIPS IV Instruction Set*. MIPS Technologies, Mountain View, California, USA, revision 3.2 edition, September 1995.

J. H. Reif. Probabilistic parallel prefix computation. In *Computers & Mathematics with Applications*, volume 26, pages 101–110, 1993.

S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, Mountain View, CA, USA, 1992.

T. Richardson, A. Shokrollahi, and R. Urbanke. Design of capacity-approaching irregular low-density parity-check codes. *IEEETIT: IEEE Transactions on Information Theory*, 47, 2001.

Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8.

Y. Sazeides and J. E. Smith. Limits of data value predictability. *Int. J. Parallel Programning*, 27(4): 229–256, 1999. ISSN 0885-7458. `http://dx.doi.org/10.1023/A:1018789613517`.

B. Shim and N. R. Shanbhag. Energy-efficient soft error-tolerant digital signal processing. *IEEE Trans. VLSI Syst*, 14(4):336–348, 2006.

A. Sodani and G. S. Sohi. Dynamic instruction reuse. *SIGARCH Comput. Archit. News*, 25(2): 194–205, 1997. ISSN 0163-5964.

A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on*

Microarchitecture, pages 205–215, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3.

The SPARC Architecture Manual. SPARC International, Inc., Menlo Park, California, USA, version 8, revision sav080si9308 edition, 1992.

N. R. Srivastava. Radix 4 SRT division with quotient prediction and operand scaling. In *Proceedings Design, Automation and Test in Europe, DATE*, pages 195–200, Nice Acropolis, France, 2007.

Standard Performance Evaluation Corporation. SPEC CPU2000 memory footprint. Online: `http://www.spec.org/cpu2000/analysis/memory/`, 2000.

Static Free Software. Electric™. Online: `http://www.staticfreesoft.com/electric.html`, 2005.

University of California, Berkeley. Espresso. Online: `http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm`, 1994.

G. Varatkar and N. R. Shanbhag. Energy-efficient motion estimation using error-tolerance. In W. Nebel, M. R. Stan, A. Raghunathan, J. Henkel, and D. Marculescu, editors, *Proceedings of the 2006 International Symposium on Low Power Electronics and Design (ISLPED), 2006, Tegernsee, Bavaria, Germany, October 4-6, 2006*, pages 113–118. ACM, 2006. ISBN 1-59593-462-6.

J. Walker. fbench: Trigonometry intense floating point benchmark. Online: `http://www.fourmilab.ch/fbench/fbench.html`, December 1980.

J. Walker. ffbench: Fast fourier transform benchmark. Online: `http://www.fourmilab.ch/fbench/ffbench.html`, April 1989.

C. Wallace. A suggestion for a fast multiplier. In *IEEE Transactions on Electronic Computers*, volume EC-13, pages 14–17, Feb. 1964.

R. P. Weicker. DHRYSTONE : A synthetic systems programming benchmark. *Comm. ACM*, 27 (10):1013–1030, 1984.

N. H. E. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition)*. Addison-Wesley, May 2004.

D. Wong and M. Flynn. Fast division using accurate quotient approximations to reduce the number of iterations. *IEEE Transactions on Computers*, 41(8):981–995, 1992. ISSN 0018-9340.

T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, New York, NY, USA, 1991. ACM. ISBN 0-89791-460-0. `http://doi.acm.org/10.1145/123465.123475`.

T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. *SIGARCH Comput. Archit. News*, 20(2):124–134, 1992. ISSN 0163-5964. http://doi.acm.org/10.1145/146628.139709.

T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 257–266, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. http://doi.acm.org/10.1145/165123.165161.

C. K. Yuen. Comment on 'Some new results on average worst case carry'. *IEEE Transactions On Computers*, C–23(3):333, 1974.

INDEX

Index