The Dragon Processor

Russell R. Atkinson, Edward M. McCreight
Xerox Palo Alto Research Center

Abstract: The Xerox PARC Dragon is a VLSI research computer that uses several techniques to achieve dense code and fast procedure calls in a system that can support multiple processors on a central high bandwidth memory bus.

1. Introduction

From the outset, Dragon was intended as a synthesis of the best modern ideas in microprocessor and shared-memory multiprocessor design. The IBM 801 [Radin 82] made us aware that for the same number of integrated circuits, a processor executing a simple instruction set designed with pipelining in mind could execute considerably faster on real benchmarks than a processor with a complex instruction set. The Berkeley RISC [Patterson 85] project introduced us to the notion of register windows. The Berkeley RISC and the Stanford MIPS [Hennessy 82] showed us that the 801 principles could be applied even to a single chip processor. Subsequent commercial announcements of the Fairchild Clipper, the HP Spectrum family, the IBM RT, the AMD 29000, and others tend to confirm the wisdom of simple instruction sets.

Our own experience at Xerox led us to value compact binary instruction encodings [Sweet 82] and to seek a fast procedure call [Lampson 82]. Many of the design decisions for Dragon were based on our experience with the Dorado processor [Lampson 81] and the Cedar programming environment [Swinehart 86]. Earlier overviews of the Dragon system appeared in [McCreight 85] and [Monier 85].

The processor design interacts with the multiprocessor system design primarily in two ways: the consistency mechanism and the synchronization primitive. C. Thacker and S. Dashiell of Xerox invented a snoopy cache consistency protocol. J. Goodman [Goodman 83] concurrently invented a similar protocol. We implement a protocol derived from Thacker's in caches attached to the processors. Our synchronization primitive, Conditional Store, is similar to that used in the IBM 370 and Motorola 68020, although its implementation is novel.

2. Overview

A Dragon processor has two VLSI chips: an Instruction Fetch Unit (IFU) and an Execution Unit (EU). Each chip has an associated off-chip cache. The IFU decodes instruction bytes and produces control signals for the EU and the caches. The EU has a 3-port register file with 128 stack registers, 16 auxiliary registers, and 12 constant registers. The IFU also has a 15-element stack of program counters and context pointers, where each context pointer determines the base of a register window in the EU stack registers. The EU has a 32 bit ALU and a 64 to 32 bit funnel shifter. The average execution rate is between 1 and 2 cycles per instruction.

We did not include internal support for floating point operations, fixed point multiplication, or fixed point division. This was due in part to a concern about silicon area, and in part because we thought it best to provide these functions through a coprocessor. The processor sees a coprocessor as a cache with a separate set of commands, so coprocessor control adds no logic beyond that required for the cache. Further details on floating point support are not yet final.

Fetches and stores are pipelined so that a new memory operation can be initiated every cycle except when the cache rejects the request or the results of a fetch are required in the immediately following cycle. Also, fetch and store instructions allow an offset to be added to the address in the same instruction, a feature that decreases the instruction count.

Although it has only a small effect on the instruction set, the Dragon design includes snoopy caches that provide the appearance of a uniform memory space to multiple processors. Each cache listens to two busses: the processor bus and the

memory bus. The caches are also responsible for address translation, so the processor bus carries virtual addresses and the memory bus carries physical addresses.

The central memory bus has a sufficiently high bandwidth (designed for a usable 200 Mbytes/sec) to service an estimated 4 to 8 Dragon processors. For increased bandwidth, the bus is a split-cycle bus: each bus operation has a request and a reply packet, so that packets from separate operations can be interleaved. Multiple memory controllers further increase concurrency and throughput on the central bus. The bus architecture is similar to that of the Sequent Spectrum [Manuel 87].

The multiprocessor system we are designing for individual researchers includes 4 processors, 32 Mbytes of memory, a color display of roughly 1 million 8-bit pixels, and peripherals such as disk and Ethernet. To facilitate connecting a variety of devices, a commercial bus (with microprocessor) is provided via a bus adapter. We expect to see the first instances of these machines in 1988.

3. The instruction set

The instructions have lengths of 1, 2, 3, or 5 bytes. The 3 byte instructions are used for arithmetic, logical, fetch, store, or field unit operations, as well as for conditional branches. The 5 byte instructions are used for calls or jumps to arbitrary locations, and for operations on 32 bit literals. The 1 and 2 byte operations are primarily for code density – most of them have 3 or 5 byte equivalents.

We believe that nearly all programs can be more compactly compiled into a variable width instruction set than a fixed width 32 bit instruction set. Usually this compactness leads to lower paging and cache miss rates, and therefore better performance. With lower cache miss rates there is more bus bandwidth available. Consequently, more processors can share a given bus without saturation, again leading to better overall performance.

Dynamically, the average instruction is about 18 bits long. A 16 byte span of memory can contain 7 average Dragon instructions, or 4 instructions of 32 bits each. Static measures of instruction size are close to the dynamic measures.

Jumps cause some of the bytes fetched to be unused. Although it is frequently possible to word align some jump destinations (like loop tops and procedure entries), some of the bytes fetched remain unused. Prefetching instructions can also cause instructions to be fetched and not used, but shorter instructions also permit fewer words to be fetched for the same benefit. Not all of these unused bytes contribute to a larger effective instruction size, since the unused bytes may belong to other instructions in the working set. An effective instruction size of 20 bits is consistent with our simulations.

Other reasons to prefer word length instructions include the reduced complexity, shorter pipeline (in Dragon one pipeline stage would be removed), increased branch distance, and potentially faster execution.

One benefit of providing 5 byte instructions is the ability to put full 32 bit addresses directly in the code stream in a single instruction (and single cycle). We estimate that 5% of instructions use full 32 bit literals (primarily as addresses), although the figure is known to be highly application dependent.

4. Fast procedure call

Procedure call is the most important abstraction mechanism in languages like Pascal, C, and Mesa, yielding both information hiding and (usually) code density. However, expensive procedure calls lead programmers to avoid using procedures. Our belief in the value of abstraction led us to design a fast procedure call mechanism.

Dragon has variable size register windows. The registers for a local frame of a procedure are bounded below by the L register, and above by the S register. Arguments are passed by pushing them onto the stack (incrementing S), and results are passed back on the stack. Not only are most local variables accessed at register speeds, but register saving is eliminated from normal execution. Arguments become local variables at procedure entry with a single instruction that sets L from S and an offset. The fastest procedures are the most common ones – those with a fixed, small number of argument and result words.

Dragon actually has two register stacks: one for data and one for control. Having separate stacks lets us avoid manipulating the EU registers when performing a call. The IFU has a control stack of program counter (PC) and status register pairs. Calling a procedure (or taking a trap) saves the active PC and status (including the active L register); returning restores the most recent pair. There are separate traps for IFU and EU frame overflow. IFU underflow is handled by software, and EU stack underflow is prohibited by convention.

As an example of procedure call, consider the following fragment of Mesa:

```
AddProc: PROC [u: INT, v: INT] RETURNS [INT] = {
    RETURN [u + v + 17];
    };
z ← AddProc[x, y + 1];
```

Assume that x, y, and z are in local registers 3, 4, and 5. Then, the resulting code for the call is

```
LR3                     -- push x
RADD  [S + 1] +, r4, c1 -- push y + 1
DFC   AddProc           -- call AddProc
SR5                     -- pop the result into z
-- 4 instructions, 10 bytes
```

Simply compiled code for the AddProc procedure is

```
ALS   -1                -- L ← S - 1
RADD  [S + 1] +, r0, r1 -- temp ← u + v
ADDB  17                -- temp ← temp + 17
SR0                     -- r0 ← temp; pop temp
RET   0                 -- return; S ← L
-- 5 instructions, 10 bytes
```

Optimized code for the AddProc procedure is

```
ADD                     -- x ← x + y; pop y
ADDB  17                -- x ← x + 17
RETN                    -- return x, no change to S
-- 3 instructions, 4 bytes
```

Process switch is much more expensive than procedure call, since there is more state to save and restore. A Dorado procedure call takes 6 μsec and a process switch takes 20 μsec. A Dragon procedure call takes under 1 μsec and process switch takes 200 μsec, depending on the average stack depth. For processor intensive Cedar programs on Dragons we project from 50K to 100K procedure calls per second per processor (5-10% of all cycles), and less than 500 process switches per second for an N-processor machine (10%/N of all cycles).

One alternative to register windows is global optimization of register allocation [Radin 82]; another is link time allocation [Wall 86]. We chose a more dynamic course, since we make significant use of object-style programming, where many procedures are bound at runtime.

5. Benchmarks and comparisons

This paper presents numbers from running three benchmarks. Puzzle, written by Forrest Baskett, is a small puzzle solving program that has been run on a variety of machines. Richards, written by Martin Richards, is a small program that simulates event scheduling in an operating system. Dhrystone [Weicker 84] is a Mesa version of the popular synthetic benchmark. We are limited by the lack of globally accepted benchmarks, and by the early stage of software development for Dragon. We do not present these numbers as being definitive.

Some other programs we have measured either fall into the range of these benchmarks or are clearly dominated by some special effect. For example, the naive recursive program for Ackerman's function has extreme stack depth variation. Such a program is almost entirely dominated by register stack overflow and underflow, yet few practical programs exhibit such swings. Floating point operations (and intensive fixed point multiplication and division) were also avoided, since the external support for these operations is not defined yet.

5.1. Some observations

This section gives some numbers obtained from detailed simulation of the benchmarks.

Cycles/inst gives the average number of cycles per instruction. This number reflects the amount of delay encountered from all sources. This figure must be used with caution, since large numbers of single cycle instructions (like the null operation) can improve this figure while decreasing the effective speed of the machine.

Bytes/inst gives the number of bytes per instruction actually fetched and used. It does not include the number of bytes fetched and not used.

1st word not ready gives the percentage of instructions that could not start execution due to the first word being not ready for instruction decode. This combines two effects: the dead cycle after control transfer, and the effects of mispredicted branches.

2nd word not ready gives the percentage of instructions that could not start execution due to the second word being not ready for instruction decode. This is one penalty for having variable size instructions.

Inst wait cycles give the percentage of cycles spent waiting for instruction bytes to arrive in the IFU for all causes.

	Puzzle	Richards	Dhrystone
Cycles/inst	1.62	1.65	1.39
Bytes/inst	2.24	2.24	2.20
1st word not ready	24%	17%	16%
2nd word not ready	10%	2%	3%
Inst wait cycles	21%	12%	14%

The IFU cache miss rate was negligible for the three programs. The EU cache miss rate was negligible for Richards and Dhrystone, and was only 5% for Puzzle.

5.2. Comparisons with the Dorado

To get the following numbers, we measured minimum elapsed times for the Dorado and obtained the Dragon cycle counts with a simulator. For both machines the instruction counts were determined using byte code interpretation. The Dorado cycle is 65 nsec. The Dragon cycle was assumed to be at its design speed of 100 nsec. Cycle counts of the Dragon hardware are consistent with the Dragon simulator, but the first version of the processor did not execute at the intended design rate due to an error in clock distribution. An improved version is being fabricated.

Puzzle	seconds	inst 10^6	μsec/inst	cycles/inst
Dorado	3.66	15.4	0.237	3.66
Dragon	1.19	7.3	0.162	1.62
Richards	seconds	inst 10^6	μsec/inst	cycles/inst
Dorado	2.58	6.29	0.410	6.31
Dragon	0.95	5.75	0.165	1.65
Dhrystone	seconds	inst 10^6	μsec/inst	cycles/inst
Dorado	0.336	1.08	0.315	4.83
Dragon	0.089	0.64	0.139	1.39

The three benchmarks given above show the Dragon as having between 2.7 and 3.8 times the performance of a Dorado. Of course, the above examples may not be typical: cache misses, process switches, and other effects characteristic of large programs will tend to narrow the differences. Still, we consider a doubling in performance to be a conservative estimate even for large programs.

The Dorado rates as a 3000 Dhrystone machine for an unoptimized Mesa version of the familiar benchmark (1000 iterations). Simple optimizations such as disabling bounds checking and inline substitution of procedures push the raw Dorado rate to near 6000. A Dragon running the unoptimized program is a 11000 Dhrystone machine (although this rate is very sensitive to the cost of string comparison). We have not tried any optimizations on the Dragon version.

6. Other features

6.1. Jump prediction

The Dragon uses jump prediction to reduce the cost of jumps. A correctly predicted jump takes 1 cycle to fall through and 2 cycles to branch. An incorrectly predicted jump takes 5 cycles. The static prediction produced by compiler heuristics ranged from 93% correct for Puzzle down to 65% correct for Dhrystone.

How much can static predictions improve with additional knowledge about the program? A simple tool was written to monitor conditional jumps, and the results were used to change the predictions, then the test program was rerun. For Dhrystone, the correct prediction percentage improved from 65% to 84%, and the cycle count improved about 6%, although only 10 predictions were inverted. For the Richards benchmark, however, correct predictions improved from 71% to 80%, and the cycle count improved by only 2%. No significant improvement in the cycle count resulted for Puzzle, where only one prediction was changed. It remains to be seen whether this range is typical of larger programs. We intend to experiment further with runtime adjustment of jump prediction.

6.2. Precise traps

The Dragon processor delays committing the effects of an instruction until all previous instructions have committed their effects. If an instruction traps, the state of the processor when the trap is taken is as if all previous instructions had completed, and the trap instruction has had no effect on the registers. This execution model, an extreme case of "precise traps," is nearly ideal for the trap handler programmer. We chose it because we worried that trap handling would be dynamically significant, especially since register window overflow causes a trap. As a result of this model, trap handling is fast, but at a noticeable cost in silicon area and a small cost in cycle time for all instructions. Further, this model prohibits out-of-order finish, which might be desirable for a floating point coprocessor, and it contributed to our decision to omit delayed branches because they involve two program counters. This latter decision we now regard as a mistake. We now favor a more pragmatic approach to traps that provides hardware assistance for rapid handling of those traps that will be dynamically significant, and provides the bare hardware essentials necessary to infer and to restore the state of the machine after other traps.

6.3. Field unit

We added a field unit to support field extract and insert, as well as bitmap operations. In one cycle this field unit can extract a 1 to 32 bit arbitrary field from a pair of words, perform logical shifts or rotates of a word, or insert a 1 to 32 bit field into a word. Such a field unit promotes data density by allowing efficient packing. Also, long strings of arbitrarily aligned data can be moved or compared rapidly.

Even with this field unit, byte fetching using a pointer and offset takes 6 instructions open-coded, and byte storing takes 9 instructions open-coded. This is more expensive than we would like, yet statistics for Cedar do not indicate that it is important to add additional byte support.

6.4. CST

The Conditional Store (CST) instruction was chosen as the atomic update primitive for Dragon. CST can be described as having the effect of atomically executing:

```
sample ← MemFetch[ptr];
IF sample = old THEN MemStore[ptr, new];
```

CST allows the atomic computation of X ← F[X], where X is an arbitrary word in memory, and F is an arbitrary function (which always returns the same value for a given value of X). This is useful not only

for the implementation of semaphores and monitor locks, but also for operations like reference counting.

The implementation of CST is primarily in the cache; the IFU presents the cache with the ptr, old, and new words, issues a CST command, and receives the sample result word. Synchronization with other processors is not necessary unless the addressed cache line is shared and old equals sample.

The advantages of this implementation are speed and simplicity for the processor and reduced central bus traffic. The disadvantage is a modest increase in complexity for the cache, although the extra area required is quite small.

Extrapolating from rates of primitives in Cedar, in a four-processor Dragon system there could be over 150,000 CST instructions per second due to reference counting, locking, scheduling and other atomic operations referencing memory. If the bus were held for 1 μsec for each operation, then 15% of the bus load would be due to CST instructions. By avoiding the bus traffic for non-shared words, and by using the split cycle bus, we reduce the bus load due to CST instructions to well under 5% (the exact amount is application dependent).

7. Conclusion

How successful has the Dragon processor effort been? It has taken a long time to finish the logic design and layout. We have had few experienced people, ambitious performance goals, a need to build tools, and early staff turnover. Commercial processors satisfying nearly all of our requirements are nearly available now, so the processor effort must be regarded as a limited success.

In retrospect, we should have forced the design into a single chip. The design would be simpler, and we the combined chip would be only slightly larger than the current IFU using the same design rules.

The most important change we would make to the instruction set would be to incorporate delayed branches. For our benchmarks the performance gain would range from 10% to 20%. The complexity penalty would be small.

Our decision to use variable length instructions is questionable. In most respects a 32 bit fixed size instruction would have been better. Only code density and the ability to express full 32 bit constants in a single instruction argue for variable size instructions. We still do not have enough statistics to make this decision clear, although we lean towards fixed length instructions.

The use of variable size register windows has strong advantages over our previous architecture, and has advantages over fixed register set machines for unoptimized or object-style code. Variable size register windows also more fully use registers than fixed size register windows.

Word addressing gives us a larger virtual memory than byte addressing. A requirement to run existing C code might influence us to choose byte addressing instead of word addressing.

Chuck Thacker and Butler Lampson provided the original impetus for Dragon, and Phil Petit worked on the first IFU design. Don Curry and Louis Monier performed wonders in designing the IFU and EU. Ed Fiala contributed to the instruction set and diagnostics. Many other members of PARC's Computer Science Laboratory also deserve credit for other aspects of the Dragon system.

References

[Goodman 83] Goodman, James R. Using Cache Memory to Reduce Processor-Memory Traffic. *Computer Architecture Symposium Proceedings*: 124-131, IEEE, 1983.

[Hennessy 82] Hennessy, John, N. Jouppi, F. Baskett, T. Gross, and J. Gill. Hardware/Software Tradeoffs for Increased Performance. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*: 2-11, Mar. 1982. Published as SIGPLAN Notices 17 (4), Apr. 1982.

[Lampson 81] Lampson, Butler W., et al. *The Dorado: A High-Performance Personal Computer*. Technical Report CSL-81-1, Xerox Palo Alto Research Center, Jan. 1981.

[Lampson 82] Lampson, Butler W. Fast Procedure Calls. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*: 66-76, Mar. 1982. Published as SIGPLAN Notices 17 (4), Apr. 1982.

[Manuel 87] Manuel, Tom. How Sequent's new model outruns most mainframes. *Electronics* 60 (11): 76-78, May 28, 1987.

[McCreight 85] McCreight, Edward M. The Dragon Computer System. *Proceedings of the NATO Advanced Science Institute on Microarchitecture of VLSI Computers*: 83-101, Martinus Nijhoff Publishers, Dordrecht, 1985.

[Monier 85] Monier, Louis and Pradeep Sindhu. The Architecture of the Dragon. *Proceedings of the Thirtieth IEEE International Conference*: 118-121, Feb. 1985.

[Patterson 85] Patterson, David A. Reduced instruction set computers, *CACM* 28 (1): 8-21, Jan. 1985.

[Radin 82] Radin, George. The 801 minicomputer. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*: 39-47, Mar. 1982. Published as SIGPLAN Notices 17 (4), Apr. 1982.

[Sweet 82] Sweet, Richard E., and James G. Sandman. Empirical Analysis of the Mesa Instruction Set. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*: 158-166, Mar. 1982. Published as SIGPLAN Notices 17 (4), Apr. 1982.

[Swinehart 86] Swinehart, Daniel C., Polle T. Zellweger, Richard J. Beach, and Robert B Hagmann. *A Structural View of the Cedar Programming Environment*. Technical Report CSL-86-1, Xerox Palo Alto Research Center, Jun. 1986. Also published in ACM TOPLAS, October 1986.

[Wall 86] Wall, David W. *Global Register Allocation at Link Time*, WRL Report 86/3, Digital Equipment Corporation Western Research Laboratory, Palo Alto, Oct. 1986.

[Weicker 84] Weicker, Reinhold P. Dhrystone: A synthetic systems programming benchmark, *CACM* 27 (10): 1013-1030, Oct. 1984.