A MULTI-MICROPROCESSOR IMPLEMENTATION OF A GENERAL PURPOSE PIPELINED CPU

by

Richard R. Ramseyer
Raytheon Submarine Signal Division
Portsmouth, Rhode Island

and

Andries van Dam
Program in Computer Science
Brown University
Providence, Rhode Island

Abstract

This paper discusses and shows by example the potential of a network of microprogrammable microprocessors as a cost-effective alternative to traditional hardwired medium- and large-scale mainframes. While biased towards vector processing, this system is not intended to compete with multi-million dollar supercomputers such as the 360/195, CDC STAR, Illiac IV, CRAY-1, TI ASC, etc., which use special algorithms and the fastest circuitry available.

The architecture incorporates pipelining, multiprocessing and distributed processing techniques with bipolar microprocessor technology. The result should be a machine which will equal or outperform most traditional third- and fourth-generation mainframes at a fraction of the CPU cost. This should be the case even for scalar, general purpose computation.

Modularity in the hardware is a further feature. This system can be implemented using a small library of components (e.g., the AM 2900 bipolar microprocessor family) and relatively little random logic.

The paper presents an overview of the proposed target machine, with emphasis on a simple scheme for detecting and resolving dependencies among instructions which must run sequentially.

Keywords: microprocessors, pipeline, distributed processing, vector machine, array processing.

1. Introduction

The proposed multi-microprocessor architecture is not intended as a panacea for the EDP industry's current or future processing requirements. Rather, it is one example of a low cost approach to large-volume number-crunching (e.g., large matrix multiplication) found in many scientific applications such as signal processing. It is not intended to be all things to all users, although we feel it can be competitive even in areas other than number-crunching.

Traditionally, CPU speed and power have been gained by improvements in circuitry. Logic chips have become significantly more sophisticated and faster every few years. This trend will decelerate in the not-too-distant future, however, due to basic constraints of physics and electronics. Perhaps the Josephson Effect circuitry[1] or optical processing[2] will provide order(s) of magnitude speed increases, but these techniques will not be routinely available in the near future. Significant performance increases over traditional systems, therefore, are implemented today with architectural as well as technological solutions.

Currently, bipolar microprocessors have the best price/performance ratio, as building blocks requiring minimal design time. Our proposed machine, then, will use the best aspects of architectural, microprocessor and integraged circuitry technologies to achieve the design goals.

Our use of architectural techniques such as pipelining and multi-processing is not novel. There are a number of machines in use today which employ many of these same techniques. The CDC STAR[3] and especially the CRAY-1[4] are fine examples of pipelined vector machines. The Illiac IV is the archetypal array processor[5] and the Carnegie-Mellon C. mmp network is a good example of a tightly coupled mini-network with a price/performance ratio superior to that of traditional mainframes[6].

The architectural tools, then, are available and to some extent have been used in combination before. What we propose is using microprogrammable microprocessors such as the AM 2900 series as building blocks with which to implement a low cost CPU that uses the best architectural features known to us. Thus, hardwired stations in a traditional pipeline will be replaced by microprogrammed microprocessors.

The design outlined is limited to internal computation. We do not address such problems as I/O and interrupt handling here. We feel that these are amenable to a similar treatment. Also, the exact sizes of instructions, memories or buffers, and numbers of microprocessor stations, etc. are by no means definitive or even optimal—we only claim that this first-cut configuration should work quite well, at the very attractive component cost of less than $10,000 for some 30 microprocessors constituting the CPU (exclusive of Instruction and Data memory).

2. User's View of The Machine

Figure 1 shows a programmer's view of the proposed target machine. It stores 84-bit instructions in an instruction memory and 32-bit data in a data memory. The instruction memory cannot be addressed for writing purposes with normal instructions: program loading is done with privileged instructions. The three-address format (20 bits absolute

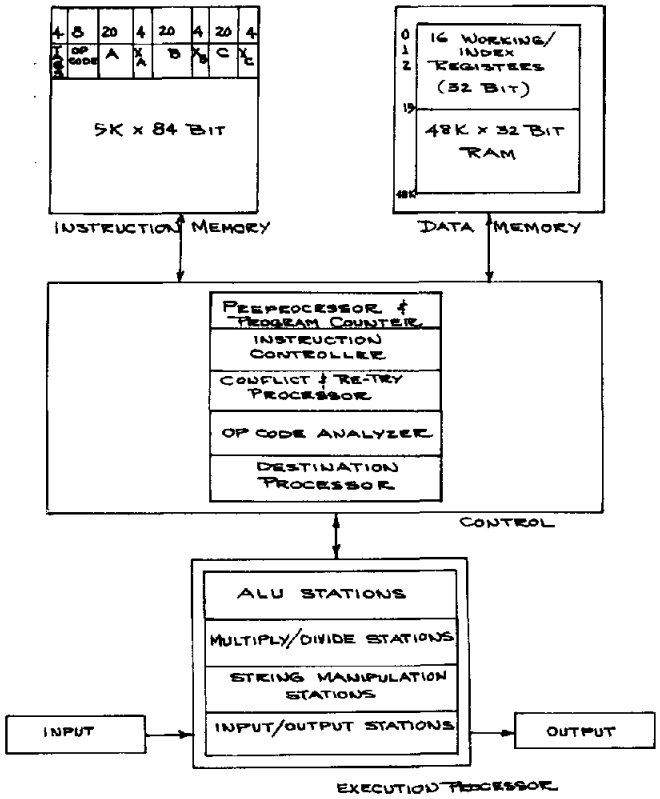


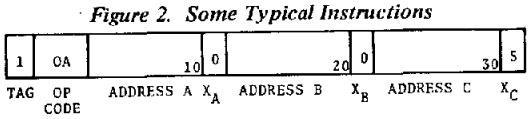
Figure 1. Architectural Block Diagram

address plus 4 bits index register) speeds up computation by eliminating redundant loads and stores from registers; it is also natural for many vector processes. The first 16 locations of memory can be addressed either as ordinary memory locations or as general-purpose working/index registers. A typical set of instructions for such a three-address machine is shown in Figure 2. Vector instructions are encoded in the normal three address instruction format and use one or more auxiliary vector control blocks. Figure 3 shows a typical vector instruction which requires a sequence of two operations: a pairwise product and a running sum of products. The former can be done by running the multiplications on the 16 microprocessors which constitute the Execution Processor in parallel, the other as a form of postprocessing by the Destination Processor prior to storing the result. Figure 4 shows an appropriate Vector Control Block.

TYPICAL SCALAR OR VECTOR INSTRUCTIONS (Tag field has vector bit)

MNEMONIC	MEANING	NOTE
ADD	C← A + B	
SUB	C← A - B	
MULT	C← A * B	Tag specifies Store MSW,LSW or both
DIV/MODULO	C← A ÷ B	Tag specifies Store Q,R, or both
MIN/MAX	C← A ∵ B	Tag specifies Min,Max, or Min,Max
CON	C← A Convolution B	Vector operation only
INP	C← A Dot B	Inner Product
CONDITIONALS	C← A Relation B	If (A Relation B) = True then jump to C: REL: =,≠,>,<,≥,≤
SWAP	C↔A	
MOVE	C← A	
PACK	C← B	Remove zeros from B and pack ones right
	SCALAR INSTRUCTIONS	
CALL		Save return address and jump to C
JUMP	Jump to C	C is literal
RETURN	Jump (C)	C contains return address

Figure 2. Some Typical Instructions



Field	Interpretation
Tag	LSB = 1 implies vector mode
OP CODE	OA is OPCODE for INP
Address A/X_A	First word for Vector A control block is at location 16_{10} in data memory; X_A = 0 = no indexing
Address B/X_B	First word for Vector B control block is at location 32_{10} in data memory; X_B = 0 = no indexing
Address C/X_C	The scalar result is to be placed at location 48_{10} of data memory indexed by X_5.

Figure 3. Vector INP Instruction Word Format

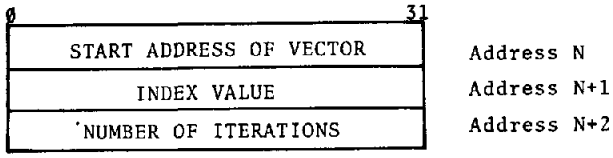


Figure 4. Vector Control Block

3. Internals

3.1 Overview

Figure 5 shows a machine consisting of a pipeline of four major stations, each of which will be treated in more detail later. The Preprocessor network of four microprocessors basically is used to fetch the four instruction fields in parallel from the Instruction memory and, in the case of vector mode, Vector Control Blocks from Data memory as well; it also performs very menial tasks in order to queue up jobs for the next station in the pipeline. The program counter in the Preprocessor may be incremented, loaded under program control (branching, subroutining), or loaded by an interrupt mechanism to effect a context switch. As all other microprogrammed bipolar microprocessors in the machine, these are driven from (writable) control stores for their special function.

The Instruction Controller network of four microprocessors is fed preconditioned instructions via the FIFO queue between it and the Preprocessor. It, in turn, checks with the Opcode Analyzer to see if a suitable processing station for an opcode is available in the Execution Processor, while simultaneously fetching the source operands and sending the destination address to the Conflict Controller. The Execution stations are typically divided into ALU, multiply/divide, string, and I/O groups. However, since the microprocessors have dynamic control store, they may be reconfigured: for example, they could all be reloaded to run a large arithmetic vector operation running 16 multiplies at once for maximum throughput. The internal timing of the Instruction Processor causes each station to be activated about 400 nanoseconds behind its predecessor. Since a

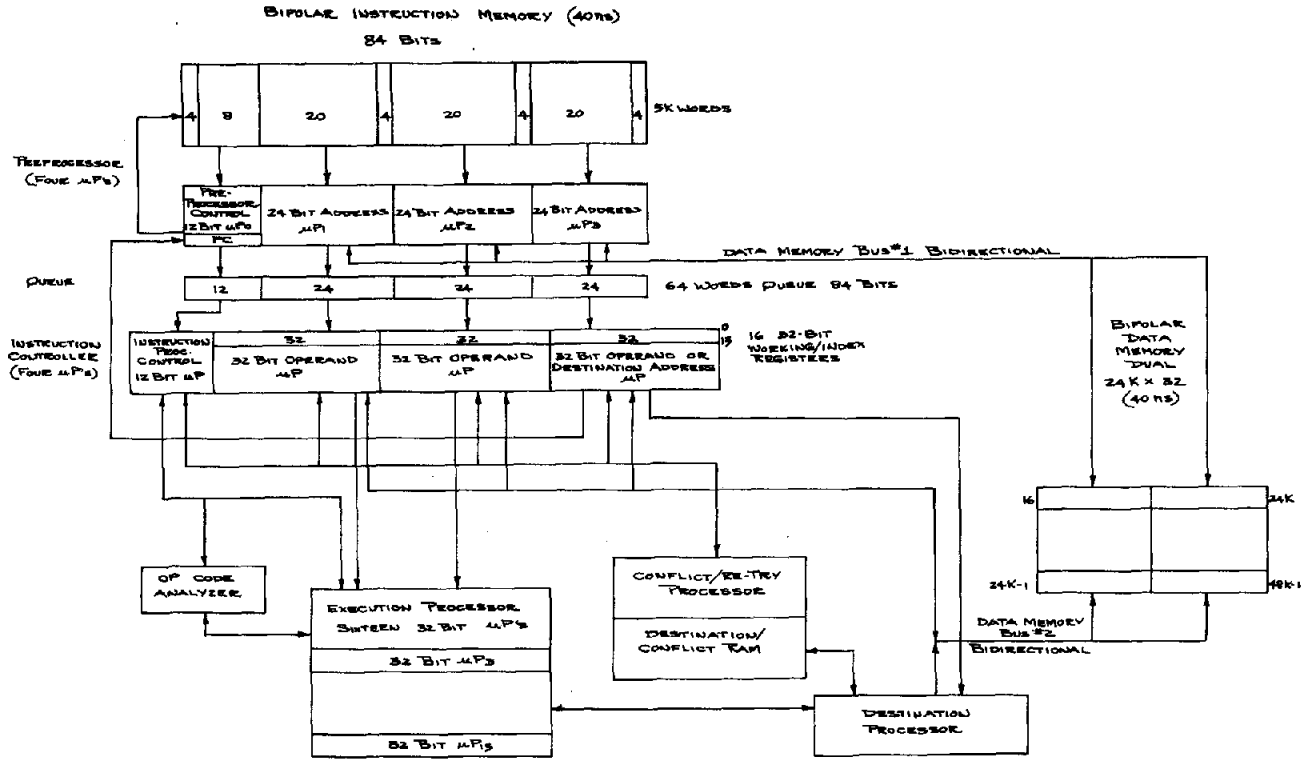


Figure 5. Lower Level Architectural Block Diagram

fixed point multiply takes approximately 16 x 400 ns, by the time station 15 has been scheduled, station 0 is available for rescheduling. The Instruction Processor then has just enough time to get 0 started again when 1 terminates, and so on. This will give an effective rate of about 400 ns per pair of points for an inner product, once the process is started.

The term "preconditioned" above refers to the fact that whenever possible the Preprocessor has relieved the Instruction Processor of the burden of certain basic tasks. For instance, in vector mode the Preprocessor need not dump raw (20 + 4 bit) addresses in the FIFO queue, but can do all effective address calculations; once started on a vector instruction, there can be no interference with the index register and memory locations by other instructions. The Preprocessor uses the data in the Vector Control Blocks to decompose the vector instruction into a sequence of scalar instructions for the FIFO queue. Further instruction prefetching may be inhibited until the vector sequence is done. Also, the Preprocessor will not pass on an instruction (such as unconditional branch) which it is capable of executing. In case the branch is taken, unneeded instructions past the branch are flushed from the queue simply by resetting the queue's pointers.

The Conflict/Retry Processor ascertains whether instructions conflict. A conflict is said to occur whenever the instruction trying to start has as one of its operands the result of an earlier instruction which has not yet completed execution, or its destination is the same as one in progress. The two conflicting instructions are interdependent, as described in data flow formulations[7].

If a conflict is detected and/or the Instruction Processor cannot find a station to execute the current instruction, the Instruction Processor will store the instruction to the 16-word Retry Buffer in the Conflict/Retry Processor. Valid operands are stored as 32-bit data; ones in conflict, by their 20-bit effective address.

If there is no conflict and an available Execution Processor station exists, the Instruction Processor puts the two source operands on the bus to the Execution Processor along with the opcode via the opcode bus. The selected 32-bit microprocessor in the Execution Processor gates the opcode and operands from the bus into its internal registers and emulates the designated target operation under direction of its control store.

Meanwhile, the destination effective address is saved in a location in a destination/conflict RAM which corresponds to the number of the Execution Processor station which was activated for this opcode.

When a station completes, the Destination Processor fetches the destination address from the destination RAM's location with the corresponding address and stores the results to Data memory. It also notifies the Conflict/Retry Processor that conflicting instructions waiting on that just-computed operand may be retried.

Note that nonconflicting instructions can run in arbitrary order and often in parallel by taking advantage of the multiple execution stations. Simple arithmetic and logical instructions may vary from 400 ns to 10 microseconds; with the possible parallelism, the effective throughput can be a fraction of these individual instruction rates, as in the inner product example above. Furthermore, there are unusually few register load instructions since most indexed loops can be handled by vector instructions, and all memory locations can be used for operations without explicit loading and storing of accumulator registers (and associated execution overhead) which takes such a large fraction of the programs for ordinary,

non three address architectures. Thus, typical instruction streams will execute considerably faster on this architecture than on traditional ones. Also, it can be made even faster by the usual technique of adding special purpose function boxes, e.g., hardwired multiplication, floating point, trigonometrics, etc.

3.2 Conflict Resolution

3.2.1 Overview. As an instruction is peeled off the FIFO queue by the Instruction Processor, it stores it in a buffer register and tries to schedule it by invoking three processes in parallel:

1) it asks the opcode analyzer if there is a free Execution Processor station;
2) it starts a fetch from data memory on the assumption that neither the source operands nor the destination operand are in conflict with a destination operand currently being computed in the Execution Processor or waiting to be scheduled for computation in the Retry Buffer;
3) it verifies this assumption simultaneously through the Conflict Analyzer which will signal any conflicts.

When the fetches are completed (less than 75 ns later, a valid source operand (one not currently being computed and therefore not in conflict) is placed in the appropriate 32-bit field of the partially decoded instruction in the buffer register, while an invalid one is kept as a 20-bit effective address, with an associated invalid operand bit. A completely valid instruction for which the Opcode Analyzer found a station is then sent to the Execution Processor. Conversely, if no station is available, or one or both source operands or the destination address are still being computed, the partially decoded instruction (opcode, effective addresses for invalid operands and fetched data for valid operands) is considered blocked. It is stored in its partially decoded form in a slot in the Retry Buffer of the Conflict Processor for subsequent rescheduling when an execution station frees up and both source operands are available and/or the destination address is available for a new computation of the same variable*.

When all operands become valid, the Retry Processor is notified that it ought to present the ready-to-run instruction to the Instruction Processor, as soon as the Instruction Processor completes handling a prior instruction from either the FIFO queue or the Retry Buffer. If a station is not available, the instruction is flagged as ready to run and remains in the Retry Buffer for periodic sampling by the Retry Processor. Note that the Instruction Processor need not do a memory fetch or a conflict check for instructions presented to it by the Retry Processor, as it does for instructions from the FIFO queue.

3.2.2 The Conflict Processing and Retry Algorithms. The conflict processing described below is functionally not significantly different from that of the 360/91[8] and the CDC 6600[9]. The implementation described here differs in that the control of the conflict processing is kept simple and is done principally by a microprocessor network. Figure 6 shows a somewhat more detailed block diagram of the Retry Buffer, Destination RAM, a number of status bits, and some queues built up for an atypical instruction mix example with an abundance of conflicts.

The diagram can be divided into two parts. The upper half consists primarily of a 16-element vector of 20-bit destination addresses. The i'th element contains the destination address for the result being computed in the i'th Execution station. (Status bits and queues associated with the element will be explained below.) The diagram shows variables X, Y, B, and Z being computed.

While the upper half thus pertains to instructions already being executed, the lower half, i.e., the Retry Buffer, contains blocked instructions awaiting execution. For example, X op Y P is the first blocked instruction. Since the Retry Buffer was empty initially, the first few instructions were placed in the Retry Buffer in chronological order. 20-bit destination addresses of these blocked instructions (e.g., P for the first one) are stored in an identical format to those of executing instructions in the top half.

The combined vector of up to 16 executing and 16 blocked destination addresses is referred to as the Destination RAM. Its principal purpose is to allow the Conflict Processor to determine by direct comparison with the entries in this RAM whether there is a conflict. Each of the three addresses of an instruction just taken from the preprocessor's FIFO Queue is compared against all destination operands being computed (X, Y, B, Z) or about to be computed (P, C, Q, etc.). In reality there are three identical copies of the entire 32 entry vector of destination addresses, which allows the comparison to take place in less than 200 ns for all three addresses simultaneously.

The format of a blocked instruction in the Retry Buffer is opcode, A (source) operand, conflict tag A, B (source) operand, conflict tag E, and C (destination) operand, conflict tag C. The conflict tags distinguish a valid 32 bit operand from an invalid 20 bit effective address.

Thus in the first instruction X and Y are currently being computed (by execution stations 0 and 1, respectively), as shown in the upper half of the Destination RAM. When the result of the source operand computation on which a blocked instruction is waiting becomes available, effective addresses are replaced by the computed value in the Retry Buffer and the appropriate conflict bits are cleared. When X and Y become available, therefore, the first blocked instruction may be scheduled, if there is an available station.

If a destination address of a new instruction conflicts with a prior computation of that operand, the C operand conflict tag is used to indicate that the new instruction should not be started until completion of the prior instruction which computes the same operand. As an example, locations 1 and 4 contain computations of C; the conflict bit is 0 for the first but 1 for the second computation. In this manner, instructions chronologically between the first and second computation will get the right version of the twice-computed operand. This avoids a critical race in which the second instruction, if it were scheduled and completed before the first one was,

---

* Note that we only compute one instance of a variable at a time to simplify bookkeeping and space requirements.

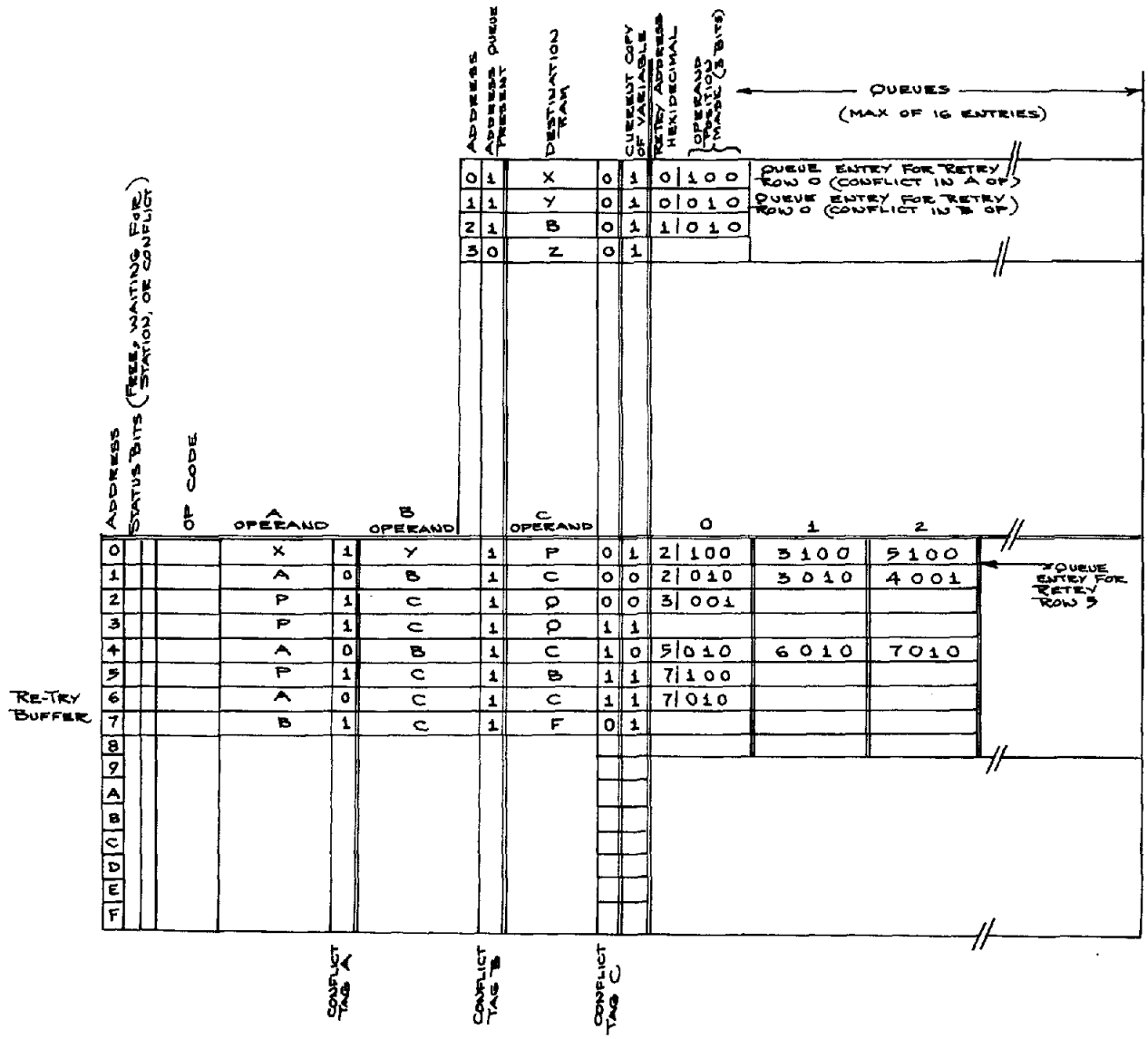


Figure 6. Retry Buffer and Destination RAM

would store its result in Data memory, only to be erroneously overwritten when the first completed**. When the first version of such an operand is available, all intermediate instructions which were waiting on it as a source are cleared as before. Similarly, the conflict tag C on the second version's instruction is cleared, allowing it to be retried if its sources are available.

The Retry Processor is notified when a blocked instruction has all three conflict tags zeroed. Additional buffer status bits (shown to the left of the Retry Buffer) distinguish between slots with blocked instructions, slots waiting for an Execution Processor, and free slots. Since the order in which instructions are scheduled is not known beforehand and is almost certainly not chronological, the Retry Buffer is filled at random using the free location bits.

As noted, the computation of an operand should result in the replacement of the address of that operand by its value in all dependent instructions which use the result as a source, as well as in a data memory store operation. Data flow dependencies in an instruction stream are typically modeled as a directed graph which could be encoded in a linked list structure. Instead the convention here is to store in a FIFO queue the Retry Buffer addresses of all instructions fetched which use a given destination operand. Each destination in the Destination RAM has such a queue, with between 0 and 16 entries. (The queue bit in the upper half of the Destination RAM facilitates checking for empty queues.) Each element in the queue consists of a 4-bit Retry Buffer address (shown as a single hex digit) and a three-bit operand position mask which is used to identify which of the three operands in the dependent instruction is in conflict. For example, the third queue element for result P computed by the first instruction in the Retry Buffer contains a 5, indicating that Retry Buffer address 5 contains a dependent instruction, and a bit pattern of 100 indicating that the computed value should replace the first (A) operand address.

** Alternatively, the second instruction could be started as soon as its operands were available and its result treated as if it were an independent entity. The redundant first store could then be obviated.

Similarly, the second queue element of C at location 2 indicates that the second (B) operand address at location 3 is to be replaced.

A destination address of a blocked instruction in the Retry Buffer will be moved to a slot in the upper 16 words of the Destination RAM when its instruction is scheduled, and the queue of addresses will simply be copied to the new location. (Also, since this slot in the Retry Buffer is now available, the status bits are reset to 00.) For example, when the first instruction, X op Y P, is scheduled, P is put in an Execution station slot in the Destination RAM, its status bits are cleared, and its queue is copied. When P is available, the first queue element indicates that retry slot two is to have its A operand assigned the value P; likewise, P will be put in the A operand of the third and fifth instructions. In case of a 1 bit in the third position, no value need be substituted but the conflict tag is cleared to allow the instruction to proceed. For example, when C in slot 1 is computed, its third queue element will cause the conflict tag C of the instruction at 4 to be cleared.

To facilitate building distinct queues for multiply-computed variables, the C (current) bit next to a destination address is set to 1 for the current (latest) copy of a given variable/destination address. A new entry in the Retry Buffer can then have its address and field bits put into the queue of the current destination variable.

It should again be noted that the scheduling and conflict/retry algorithms are conservative in order to keep them simple, cheap to implement, and adequately fast. Special casing could be used to do more optimal scheduling, and keep the execution stations busier. Since they are cheap, however, idleness is not a cardinal sin. Also, an optimizing compiler[10] can help produce instruction streams which reduce instruction conflicts and therefore allow more execution concurrency.

4. Summary

We have presented a high-level description of a cheap CPU architecture which uses networks of microprocessors to effect a pipelined architecture well suited to number-crunching, e.g., large vector and matrix manipulations. Many details have been left unspecified even in the much more detailed design presented in[11] and need to be resolved through detailed simulation. Also, such facilities as string handling, I/O, virtual memory, and operating system support need special treatment and some additional hardware. However, preliminary calculations show that this type of architecture can make very good use of dynamically microprogrammable microprocessors to achieve parallelism and therefore high throughput. The expected performance is very competitive with all but the biggest existing processors, at greatly reduced cost and greater modularity and expandability. For example, memory and buffers can be enlarged, more execution stations can be added, and slower microprocessors can be replaced, where needed, by faster ones or hardwired logic.

In many cases the use of a microprocessor in our system may be "overkill". Our goal was to develop a machine as free of hardwired logic as possible, so microprocessors were used for nearly every feasible station in the pipeline. In addition, we feel that the current technological trend indicates that overkill may be economically justificable before very long, i.e., the cost of hardware will outstrip the cost of a microprocessor with firmware for even trivial implementations.

5. Bibliography

1. Yu, M.L. and A.M. Saxema, "Coherent A.C. Josephson Effect in a Bulk Granular Superconducting System", Brookhaven National Laboratory, New York, Sept., 1974.
2. Stotts, L.B., "High Speed Optical Matrix Multiplier System", Department of the Navy, Washington, D.C., May 1975.
3. Hintz, R.G. and D.P. Tate, "Control Data STAR-100 Processor Design", Proceedings Sixth Annual IEEE Computer Society International Conference, San Francisco, California, September 1972.
4. Cray, Seymour, "An Introduction to the CRAY-1 Computer", Cray Research, Inc., Chippewa Falls, 1975.
5. Barnes, G.H. et al., "The Illiac IV Computer", IEEE Transactions on Computers, Vol. C-17, No. 8, August 1968, pp. 746-757.
6. Fuller, Samuel H., "Price/Performance Comparison of C. mmp and the PDP-10", ACM/IEEE Symposium on Computer Architecture, January 1976.
7. Dennis, J.B. and D.P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing", Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974.
8. Tomasulo, R.M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM J R & D, 11, 1967 pp. 25-33.
9. Thornton, J.E., Design of a Computer, the Control Data 6600, Scott, Foresman, and Co., Glenview, Ill., 1970
10. Wedel, D. "FORTRAN for Texas Instruments ASC System", SIGPLAN Notices, Vol. 10, No. 3, March 1975.
11. Ramseyer, R.R., "Multi-microprocessor Implementation of General Purpose Mainframe CPU Systems", Masters Thesis, University of Pennsylvania, August 1976.