

# Machine organization of the IBM RISC System/6000 processor

by G. F. Grohoski

**The IBM RISC System/6000\* processor is a second-generation RISC processor which reduces the execution pipeline penalties caused by branch instructions and also provides high floating-point performance. It employs multiple functional units which operate concurrently to maximize the instruction execution rate. By employing these advanced machine-organization techniques, it can execute up to four instructions simultaneously. Approximately 11 MFLOPS are achieved on the LINPACK benchmarks.**

## Introduction

This paper describes the machine organization of the IBM RISC System/6000\* (RS/6000) processor. Companion papers in this issue describe the instruction-

set architecture [1] and the organization of the floating-point dataflow [2]. The next section describes the motivation for the original design work. The third section describes the problems inherent in a highly overlapped multiple-execution-unit design, and the solutions which were developed for them. The fourth section describes modifications to the original design point introduced during the implementation.

## Motivation for the design

### • *Evolution of 801-based machine organizations*

In the early 1980s various projects at the IBM Thomas J. Watson Research Center examined aspects of high-performance Reduced Instruction-Set Computer (RISC) designs. From earlier work [3] based on the experimental 801 computer, it was clear that RISC processors offered many advantages over conventional CISC (Complex Instruction-Set Computer) designs such as the IBM System/370. First, the amount of logic required to implement the architecture naturally led to a compact, efficient design which could potentially be brought to market in a short period of time. A fast cycle time could be supported, since control could be hard-wired, and a simple dataflow effectively supported the instruction set. Ignoring finite cache effects, the 801 inherently executed nearly one instruction per clock cycle.

John Cocke believed that a suitably augmented scalar RISC processor could effectively compete with larger and more expensive vector processors by using multiple

\* RISC System/6000 is a trademark of International Business Machines Corporation.

©Copyright 1990 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

execution units and by dispatching several instructions per cycle.

The notion of using multiple functional units which operate concurrently to improve performance was examined in early computer designs, notably the IBM System/360 Model 91 [4] and the CDC 6600 [5]. One basic question was how far a RISC machine organization could be pushed: Could a sustained rate of less than one cycle per instruction be achieved? How much hardware would be required? Would the cycle time be lengthened enough to offset any gain in cycles per instruction? What architecture changes would be required to effectively support a multiple-execution-unit 801 design?

This so-called superscalar approach was studied in the Cheetah project at the Watson Research Center in 1982–1983.<sup>†</sup> The Cheetah machine organization used separate branch, fixed-point, and floating-point execution units to speed instruction processing. Significant changes were made to the 801 architecture to facilitate the implementation of a multiple-execution-unit design and to expose this design to the compiler. The RS/6000 machine organization owes much to that of the Cheetah machine; important differences between the two will be discussed where appropriate.

The target technology of the superscalar studies was bipolar ECL. By 1984 it became clear that CMOS was achieving a level of integration, chip size, and circuit performance which allowed high-performance RISC processors to be packaged on a few chips. The resulting cost and cost/performance advantages of this design point were dramatic.

The AMERICA project was undertaken to study further the implementation of a multiple-execution-unit 801 design in CMOS. The author, working with John Cocke and Gregory Chaitin, wrote a cycle-by-cycle simulator of the machine organization (called a “timer” in IBM parlance) to demonstrate clearly the processing power of the machine organization, to validate the organizational concepts, and possibly to be used as the initial logic specification for a prototype. Some areas of the machine organization, such as the interrupt synchronization mechanism, were developed during the following year.

The end result of this work was a combination of machine organization, instruction-set architecture, and compiler techniques which allowed a VLSI CMOS processing unit to perform at a level comparable to those of ECL vector processors such as the Cray-1.

#### • AMERICA machine organization

Figure 1 depicts the organization of the AMERICA processor; the organization of the RS/6000 processor is

<sup>†</sup>T. K. M. Agerwala and D. Prener, “Cheetah Principles of Operation,” IBM internal document, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 1982.

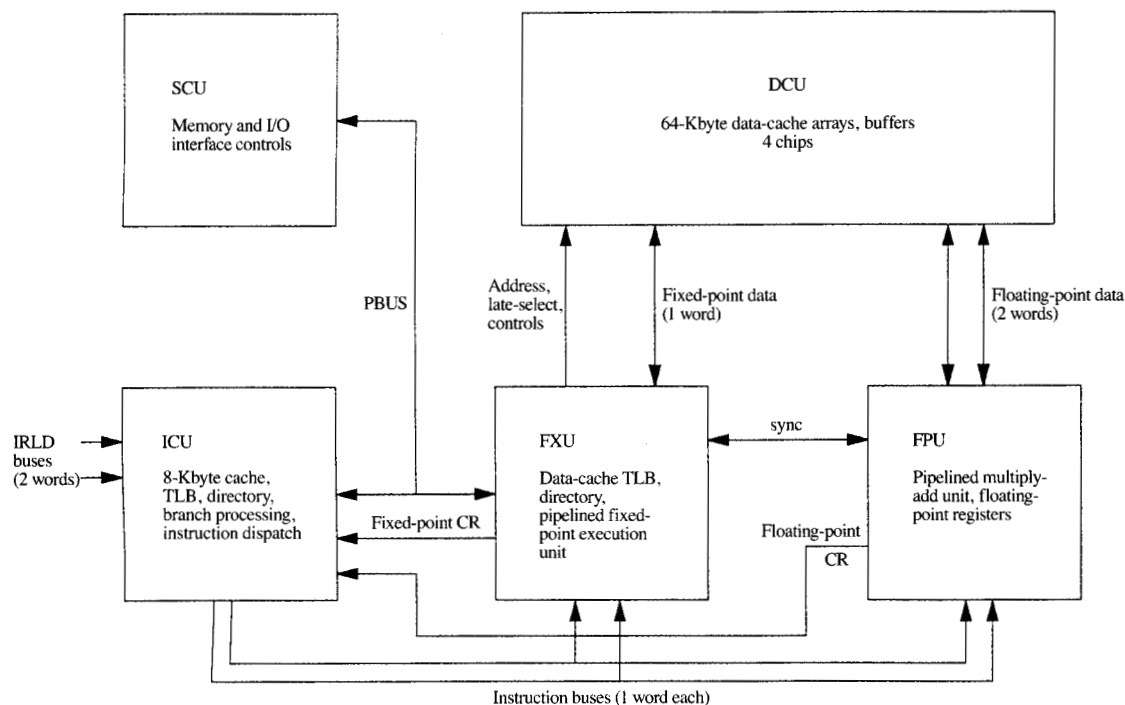
identical. It consists of several functional units, each partitioned onto one chip (except for the data cache). The instruction cache unit (ICU) fetches instructions and executes branch and LCR (Condition Register Logic) instructions. It dispatches two instructions per cycle to the fixed-point unit and floating-point unit, and receives condition-code information from each unit over dedicated buses. A two-word instruction-reload bus refills an instruction cache line when a miss occurs. The fixed-point unit (FXU) executes fixed-point instructions, performs address calculations for floating-point loads and stores, and contains the address translation, directories, and controls for the data cache. It controls the PBUS, an internal processor bus used to communicate cache-miss and store-back information to the memory interface, and to transfer architected registers between the FXU and ICU. The floating-point unit (FPU) is a high-speed chip which is capable of executing floating-point loads in parallel with arithmetic instructions. One fixed-point, one floating-point, one branch, and one LCR instruction can be executed simultaneously. The system control unit (SCU) contains the memory and I/O interface and controls. The data-cache unit (DCU) contains 64-Kbyte data-cache arrays and data-cache buffers. More details on the actual implementation can be found in [6]. This paper is concerned primarily with the ICU, the FXU, and the control interface to the FPU.

To understand the operation of the AMERICA processor, consider the following 2D graphics transform. The RS/6000 pipeline is nearly identical, except that floating-point loads and stores work differently. It rotates a list of points  $(x_i, y_i)$  through an angle  $\theta$  and displaces them by an amount  $(x_{dis}, y_{dis})$  to produce a new set of points  $(x'_i, y'_i)$ , stored in the same locations as the original set. An RS/6000 pseudoassembly code excerpt for this routine is given below:

```

FL FR0, sin theta           ;load rotation matrix
FL FR1, -sin theta         ;constants
FL FR2, cos theta
FL FR3, xdis                ;load x and y
FL FR4, ydis                ;displacements
MTCTR i                     ;load Count register
                             ;with loop count
LOOP UFL FR8, x(i)          ;load x(i)
FMA FR10, FR8, FR2, FR3    ;form x(i)*cos + xdis
UFL FR9, y(i)              ;load y(i)
FMA FR11, FR9, FR2, FR4    ;form y(i)*cos + ydis
FMA FR12, FR9, FR1, FR10   ;form -y(i)*sin + FR10
FST FR12, x(i)'            ;store x(i)'
FMA FR13, FR8, FR0, FR11   ;form x(i)*sin + FR11
FST FR13, y(i)'            ;store y(i)'
BCT LOOP                   ;continue for all points

```



**Figure 1**

General organization of the AMERICA and RISC System/6000 processors.

UFL is an update-form floating-point load that auto-increments the address to use for the next point. FMA is a multiply-add instruction that accepts four register operands. The second and third operands are multiplied together, added to the fourth operand, and stored in the first. FST stores a floating-point result in memory. BCT is a special loop-closing branch instruction which examines a value in the Count register; if it is greater than zero, the branch is taken. The Count register is auto-decremented.

Following is a description of the cycle-by-cycle execution of the inner loop in the AMERICA machine organization; a diagram is shown in **Figure 2**. (The actual RS/6000 pipeline is described later in more detail.)

- IF* The instruction-fetch cycle of the processor. The instruction cache is accessed and four instructions are fetched from the cache arrays and placed into instruction buffers.
- Disp/BRE* During this cycle, up to four instructions are

examined for dispatching. Branch and LCR instructions are executed, if they can be removed from the buffer. The target addresses for branch instructions are generated. Fixed- and floating-point instructions are transmitted to the fixed- and floating-point units.

- FXD* During this cycle the fixed-point unit decodes fixed-point instructions and accesses the register file for operands.
- FXE* The fixed-point unit executes instructions during this cycle. For load and store instructions, the address is generated, and the data-cache translation look-aside buffers (TLBs) and directories are searched. The array address is transmitted to the data-cache arrays and latched.
- C* During this cycle the data-cache arrays are accessed. Based upon a late-select signal from the fixed-point unit, which chooses data from one of the four sets of the data cache, data is returned to either the fixed- or floating-point units.

	1	2	3	4	5	6	7	8	9			
IF	UFL8 FMA	UFL9 FMA	FMA FST	FMA FST	BCT			UFL8 FMA	UFL9 FMA	FMA FST	FMA FST	BCT
Disp/BRE		UFL8 FMA	UFL9 FMA	FMA (BCT) FST	FMA FST	BCT	UFL8 FMA	UFL9 FMA				
FXD			UFL8 FMA	UFL9 FMA	FMA FST	FMA FST	UFL8 FMA	UFL9 FMA	FMA FST			
FXE				UFL8	UFL9	FST	FST	UFL8	UFL9			
C					UFL8	UFL9						UFL8
WB												
PD			UFL8 FMA	UFL9 FMA	FMA FST	FMA FST	UFL8 FMA	UFL9 FMA	FMA FST			
Remap				UFL8 FMA	UFL9 FMA	FMA FST	FMA FST	UFL8 FMA	UFL9 FMA			
FPD					FMA	FMA	FMA	FMA	FMA			
FPE1						FMA	FMA	FMA	FMA			
FPE2							FMA	FMA	FMA			
FPWB								FMA	FMA			

Figure 2

AMERICA processor pipeline showing cycle-by-cycle execution of the inner loop (see pp. 39-40 for definitions and discussion of the terms at the left).

**WB** During this cycle the fixed-point unit writes the results of instructions to the register file. For RR instructions, this cycle is in parallel with the cache access cycle. Data for loads is written into the register file during this cycle.

**PD** This is the floating-point pre-decode cycle. It is at the same pipeline level as FXD. During this cycle instructions are pre-decoded in preparation for renaming.

**Remap** During this cycle the registers of floating-point instructions are mapped to physical registers.

**FPD** This is the floating-point decode cycle. The registers are read out for floating-point arithmetic instructions.

**FPE1** The first cycle of the multiply-add pipeline.

**FPE2** The second and final cycle of the multiply-add pipeline.

**FPWB** During this cycle the results of floating-point arithmetic instructions are written to the floating-point register file.

During cycle 1 the first four instructions starting at LOOP are fetched. During cycle 2 the first load and multiply are dispatched to the floating-point unit. The next four instructions are also fetched.

During cycle 3 the second instruction pair is dispatched to the fixed- and floating-point units. The first pair is in fixed-point decode and floating-point pre-decode. The fixed-point unit will execute the floating-point load and discard the multiply-add. The floating-point unit will send both instructions to the rename stage. The loop-closing BCT instruction, along with three

subsequent instructions (not shown), is being fetched from the instruction cache.

During cycle 4 the fixed-point unit generates the address for the first floating-point load. The floating-point unit renames the floating-point load and the multiply-add. The second instruction pair is in fixed-point decode and floating-point pre-decode. The instruction cache dispatches the third instruction pair, and branch-scanning logic looks five instructions deep in the instruction buffer to generate the target address of the BCT.

During cycle 5, the instruction cache fetches the top of the loop. The fourth instruction pair is dispatched to the fixed- and floating-point units, and the BCT is executed. The first FMA is in floating-point decode; the first floating-point load is accessing the data cache. At the end of this cycle, the data will return and the FMA will enter the floating-point execution pipeline, since all of its registers are free. The fixed-point unit is generating the address for the second floating-point load.

During cycle 6 the second floating-point load is accessing the data cache. The second floating-point multiply-add will decode, since all required registers are available. The address of the first floating-point store is being generated; it will be placed in a store data address buffer at the end of the cycle. When the data is produced in cycle 10 (not shown), the store will be written to the data cache at the first free cache cycle.

Several points are notable. The BCT causes no pipeline delays, and as far as the fixed- and floating-point units are concerned, no branch ever occurs. The floating-point pipeline is kept 100% busy, and produces two floating-point results each cycle (one multiply and one add). Ignoring finite cache effects, this computation proceeds at 50 MFLOPS in the inner loop on AMERICA at a 40-ns clock cycle. Due to problems encountered during the implementation of the floating-point unit, the RS/6000 processor executes this code at a 28-MFLOP rate. The remainder of this paper describes how this processing rate was achieved.

### Problems of a multiple-execution-unit design approach

A RISC design which uses multiple functional units simultaneously executes several instructions per cycle; therefore, several instructions must be fetched each cycle. The effect of branch instructions on the pipeline must be reduced, because it is relatively greater than in a machine which executes only a single instruction per cycle. The execution units must be synchronized when interrupts occur, to maintain sequential program consistency and to ensure that arithmetic operations are performed using the correct data in the correct order.

If the effect of branch instructions can be mitigated and the floating-point and fixed-point units can be

supplied with instructions and data at a high rate, a large increase in processor performance is possible. The central requirements which needed to be addressed were the following:

1. Design a low-latency, high-bandwidth instruction-fetching mechanism.
2. Overlap the execution of branch instructions with fixed-point and floating-point instructions.
3. Overlap the fixed-point and floating-point units in order to keep the floating-point unit supplied with data.
4. Maintain the effects of sequential program execution while executing several instructions in parallel.
5. Design a high-performance floating-point execution unit.

The solutions to the first four requirements, developed during the AMERICA project, are discussed here. The design of the floating-point dataflow is discussed in [2].

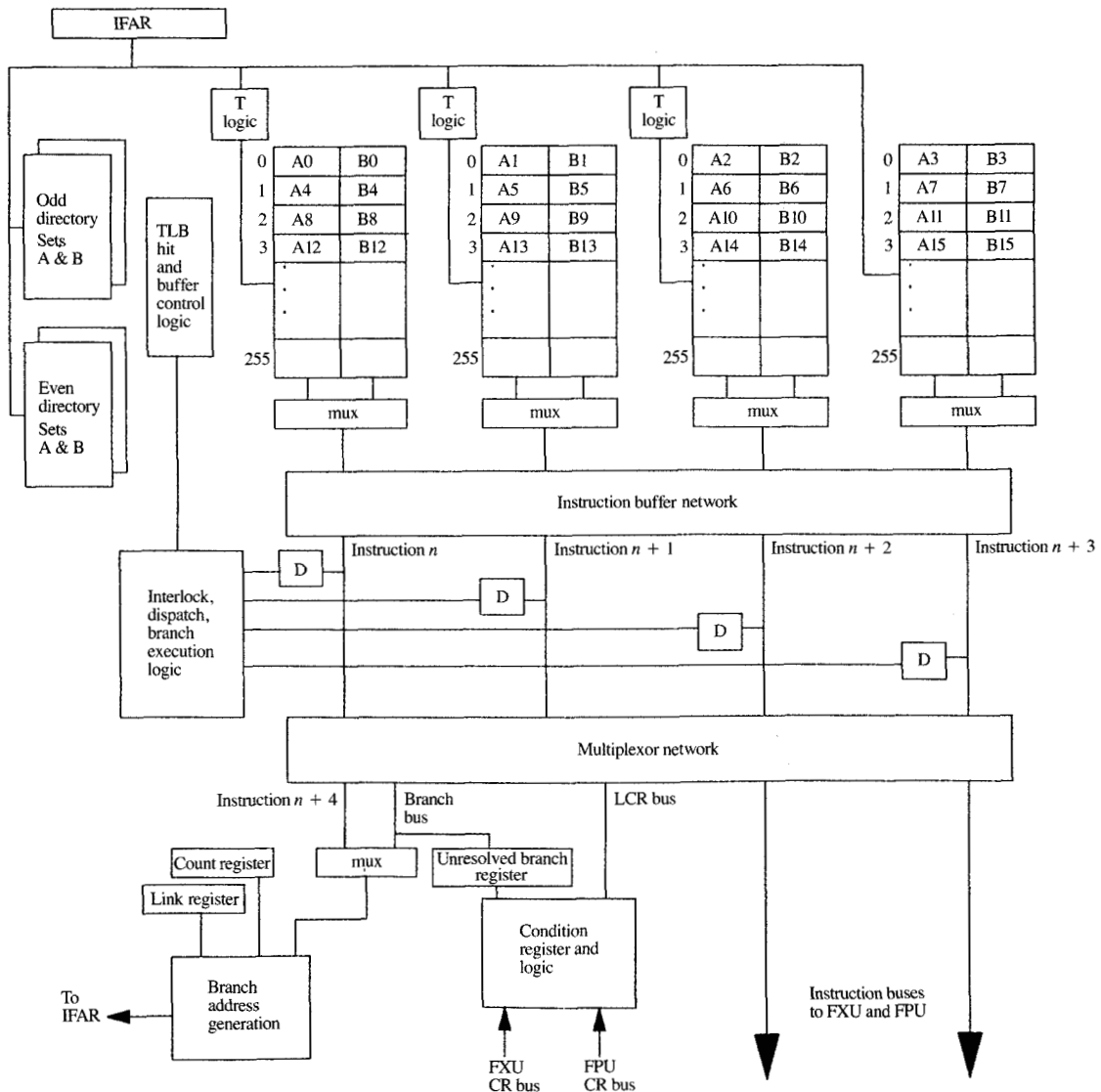
#### • *Instruction fetching*

The instruction-fetching mechanism must have a low latency so that the execution units remain busy when the target of a taken branch is being fetched. This argues for a cache which can be accessed in one machine cycle.

While the processor can execute four instructions per machine cycle, it more commonly executes three instructions (a branch, a fixed-point, and a floating-point instruction) per machine cycle in heavy floating-point code. The cache must at least match this rate. In order to help overlap the execution of branch instructions with fixed- and floating-point instructions, the branch-scanning logic, as it looks through the instruction buffers, must detect a branch somewhat in advance of its execution. This means that the instruction-cache bandwidth must be greater than the raw bandwidth required by the execution pipelines.

In view of the high bandwidth required, an on-chip dedicated instruction cache was designed which could be accessed in one cycle. In order to fetch multiple instructions per cycle, a new cache organization was developed.

All instructions are four bytes (one word) in length. The first design choice was to build an instruction cache which, given an arbitrary byte address, truncated the four low-order bits and returned the resulting quadword (QW)-aligned set of four instructions. However, this did not supply the processor with the required number of instructions if, for example, an instruction branched to the last word in a QW. In this case only one instruction would be supplied to the execution units, and this would seriously degrade loop performance. Possibly the



**Figure 3**

General organization of the instruction cache and branch-processing unit.

compiler and loader could be directed to QW-align all branch targets for loops, but this would increase code size and could lead to additional complications.

Consider the instruction-cache arrays to be composed of four smaller, independent arrays, each fetching one instruction per cycle. By controlling the address presented to each array and interleaving the instructions among the cache arrays, four instructions can always be fetched, as long as they reside in the same cache line. **Figure 3** diagrams the organization of the cache arrays for a two-

way set-associative cache with a line size of 16 instructions (64 bytes).

Each subsequent instruction is placed into a different cache array, computed modulo 4. If the actual word linewidth of each array is two instructions, instruction 0 of associativity sets A and B occupies row I of cache array 0. Row I of cache array 1 contains instruction 1 of a given cache line, and so on. In this case one cache line is split into four rows of a cache array. Consider how four sequential instructions in a cache line, regardless of the

address of the first instruction of the group, can be fetched.

In order to fetch instructions 0, 1, 2, and 3 of a given cache line, the same row address can be presented to all cache arrays. This is precisely the QW-aligned case mentioned above. In order to fetch instructions 1, 2, 3, and 4 of a cache line, the row address for cache array 0 must be incremented, since instruction 4 resides in the next row. This is determined by address bit 28 (bit 0 is the high-order bit of a 32-bit address).

By considering all 16 possibilities of the starting address of a word in a cache line, it is seen that cache arrays 0, 1, and 2 need to have their row addresses incremented, while array 3 does not. The T logic of Figure 3 provides the row incrementation and selection functions.

By interleaving the cache arrays and providing the necessary row incrementation, row-selection multiplexors, and row-selection logic, four instructions can be fetched each cycle as long as they are contained in the same cache line. If the group of four is within the last three instructions of a cache line, it spans two cache lines. In this case, only 3, 2, or 1 instructions can be fetched. On the average, this organization allows  $(13/16) \times 4 + (1/16) \times 3 + (1/16) \times 2 + (1/16) \times 1 = 3.625$  instructions to be fetched each machine cycle.

- *Branch delays*

Reducing the delays caused by branches in a pipelined machine has been one of the classical challenges of computer design. Consequently, many approaches have been developed. The objective, of course, is to have branches take zero execution cycles. Branches reduce the effective throughput of the pipeline by causing several types of delays; following is a description of these delays and the state of the art in reducing them in 1984:

- It takes time to fetch the target of a successful (taken) branch. During this time the execution pipeline may be starved of instructions.

There are several approaches which reduce the target-fetch delay. First, the branch-target address can be calculated while the branch is in the early stages of the execution pipeline. This generally requires a separate branch-target address adder. Then, the address can be provided to the instruction-fetching mechanism to fetch the branch target before the outcome is known. If the branch is not taken, the branch-target instructions can be discarded. If the target is fetched, a pipeline delay can be introduced if too few sequential instructions are available to the execution pipeline to cover the delay of re-fetching the sequential path. Generally, then, the target address is fetched based upon some prediction of the branch being taken.

Another technique, typically useful for IBM System/370 processors [7], is to store the calculated branch-target address in a table which is indexed using the address of the branch instruction. When the branch instruction is fetched, its address is used to access the table, which provides the branch-target address. Logic is provided which ensures that the table contains the proper branch-target address by invalidating the table entry for the branch if the register which the branch uses for its target address is changed.

To further reduce the target-fetch delay, a branch-target buffer [8] can be provided which stores the target address of a branch and the first several instructions from a branch target. Once the branch-target address has been computed, it can be presented to the branch-target buffer, where the target address is compared with the addresses of branch targets contained in the buffer. If a match is found, no request need be made to the instruction-fetching mechanism until the outcome of the branch is known. If taken, the instruction-fetching mechanism is given the address of the instruction just beyond the last instruction stored in the matching entry in the branch-target buffer. If no match is found, the branch target can be fetched. It is added to the buffer by replacing a buffer entry which has not been referenced recently. The effectiveness of a branch-target buffer depends on the fraction of taken branches whose targets are found in the buffer.

Yet another technique common to RISC machines is to use some variant of the so-called branch-with-execute instruction [3, 8]. An instruction which originally preceded a branch is moved behind the branch by the compiler. This subject instruction is executed whether or not the branch is taken. If the branch is not taken, no penalty is incurred. If the branch is taken, at least one instruction is available to the execution pipeline while the branch target is being fetched. Variations on this technique utilize more subject instructions, or can choose instructions from the target of the branch to be used as subject instructions. With this form, termed branch-or-skip, if the branch is not taken the execute instructions must be skipped, possibly introducing some delay. This form is used when the branch is unconditional or has a high likelihood of being taken. The Cheetah machine used up to four subject instructions in both branch-with-execute and branch-or-skip forms.

These execute-form branches have several drawbacks. Architectural and implementation complications result if the subject instruction causes an interrupt. If the interrupt handler returns to the subject instruction once its interrupt has been serviced, the branch may be taken or not taken. If it is to be taken, the machine must "remember" the branch target

address. Or the interrupt handler may examine a bit which denotes the interrupting instruction as the subject of a branch-with-execute, reexecute the subject instruction, and then return either to the target of the branch or to the next sequential instruction following the subject, depending upon the interrupt handler's determination of the branch outcome. Alternatively, the interrupt handler may return directly to the branch instruction and reexecute the subject instruction. In this case the branch instruction must not alter any registers (or the interrupt handler must undo the effects of the changes). Whichever course is chosen, the situation becomes even more complicated if multiple subject instructions are used.

Nor can subject instructions always be found. Because of dependencies in basic blocks, subject instructions can be used to fill the execution slot only about 60% of the time [3].

- Conditional branches require an execution unit to set a condition code. There is typically some pipeline latency before the condition code is available and the outcome of the branch can be determined, which stalls the execution of the branch instruction.

In order to reduce the delays caused by waiting for the condition code to become available, several techniques can be employed. Branch-prediction techniques can be combined with branch-target buffers or decode history tables [8, 9] to reduce branch delays. Smith [10] examined several branch-prediction strategies. A branch can be guessed taken, or not taken, as a function of history (the branch history table [9] or its variants) or of branch type, or based on a bit placed in the instruction and set by the compiler. Prediction simply allows the machine to proceed down either the sequential or the target path. Since the outcome of the branch is uncertain, the pipeline must treat the instructions in a conditional fashion and be able to undo any changes to the architected machine state if a wrong prediction has been made. Alternatively, machines have been proposed which proceed down both paths [7], although this requires the duplication of hardware and in general has been too costly to implement.

- Branches also typically proceed through the execution pipeline, thereby consuming at least one pipeline slot and delaying subsequent fixed-point instructions. This is the case with most current RISC machines [11-14].

In Cheetah, a separate branch-execution unit was provided to eliminate this pipeline delay. In order for this to be most effective, architecture changes were made to decouple the branch and fixed-point execution units (these are described in [1]).

In a machine which can execute several instructions per cycle, the effects of these branch delays are magnified.

The approach chosen for AMERICA was the following. First, a separate branch-execution unit was provided. This allowed for the possibility of zero-cycle branches. Second, logic was provided to scan through the instruction buffers for branches, to generate the branch-target address, and to determine the branch outcome if possible. If the branch outcome was undefined, instructions would be dispatched from the sequential path to the fixed- and floating-point execution units in a conditional fashion. When the branch outcome was determined, these instructions would either be executed, and the branch-target instructions discarded, or canceled, and the branch-target instructions transmitted to the execution units. (The notion of combining branch-address generation logic and instruction-cache accessing had also occurred to other researchers [15], but they did not consider fully integrating a separate branch-processing unit and an instruction cache.)

The justification for this simplistic strategy was the following. Gross branch statistics available from 801 instruction studies indicated that branches comprised approximately 20% of all instructions (in fixed-point code). Approximately one third of the branches were unconditional; another third were used to terminate do-loops of the form  $do\ i = 1, n;$  and the final third were conditional. If a separate branch-execution unit is used, with proper scanning ahead to overlap branch execution with the execution of fixed-point instructions, unconditional branches should cause no pipeline delay provided the branch target is in the cache. Using the loop-closing branch instruction, which is basically an unconditional branch for the first  $n - 1$  iterations, should also cause no delay. Of the remaining conditional branches, about half are taken, and half are not taken. The branches not taken should cause no delay, since they would be predicted not taken. The branches taken would cause some delay, estimated to be two pipeline cycles. Thus, branches, instead of requiring one cycle each to execute, would require approximately  $(5/6) \times 0 + (1/6) \times 2$ , or about 0.33 cycles on the average.

Some form of branch prediction for conditional branches could further reduce the delay. One strategy would be to have a branch history table for conditional branches whose outcome is unknown when they are first encountered. However, the published effectiveness of most branch-prediction strategies is skewed because unconditional branches are included in the prediction mechanism. This raises their apparent effectiveness substantially. Our feeling was that the remaining conditional branches were essentially random in nature



and that typical branch-prediction techniques would not be very effective. A branch-prediction mechanism would require some significant space to implement.

Furthermore, in order to decode down the target path, and dispatch target instructions to the fixed- and floating-point units, additional logic would have to be added to the instruction-cache unit. This logic would at most eliminate the 0.33-cycle delay entirely, if both streams were decoded; otherwise, it might perform worse if a branch were predicted to be taken but was not taken. For these reasons, sophisticated branch prediction, such as a branch history table, was not implemented.

Branch-with-execute and branch-or-skip were not utilized. Assuming that the branch-processing unit and branch-scanning logic run far enough ahead of the fixed- and floating-point execution units, the branch target can be fetched in time to avoid pipeline execution delays for most taken branches. Not-taken branches do not benefit from branch-with-execute. Certain implementation difficulties could be avoided (801 implementations were notorious for having problems with bugs in branch-with-execute), and the architecture could be simplified, if these branches were not architected. Branch-with-execute can provide one advantage: The branch is effectively moved forward in the instruction stream, allowing the branch-scanning logic to detect it earlier. (Referring to Figure 2, if the BCT were an execute-form branch, it would be detected one cycle earlier, and the target could be fetched one cycle earlier. The branch-scanning logic would only have to look four instructions deep to detect the branch.) This potential advantage was offset by simply looking further ahead in the instruction buffers for a branch.

To illustrate the design, several branch-execution examples are depicted in **Figure 4**, which illustrates the RS/6000 pipeline delays. Figure 4(a) shows an unconditional branch, and 4(b) its associated pipeline behavior. The pipeline cycle names are the same as in Figure 2. At the end of the fixed-point execution cycle (FXE), condition-code results are transmitted to the branch unit so that conditional branches can be resolved in the following cycle.

Figures 4(c) and 4(d) depict a conditional branch which is not taken. Figures 4(e) and 4(f) depict a taken conditional branch whose condition is set two fixed-point instructions before the branch, causing a one-cycle pipeline delay. Figures 4(g) and 4(h) depict a taken conditional branch which causes no pipeline delay.

Note that the only branches which typically cause any pipeline delay are taken conditional branches that depend upon a fixed-point compare which cannot be scheduled with three or more instructions between it and the branch. Thus, the AMERICA branch-processing approach is robust and simple. Although sophisticated branch-prediction techniques are not used, branch

instructions cause a fraction of the pipeline delay of most other RISC machines.

- *Overlap of fixed-point and floating-point units*

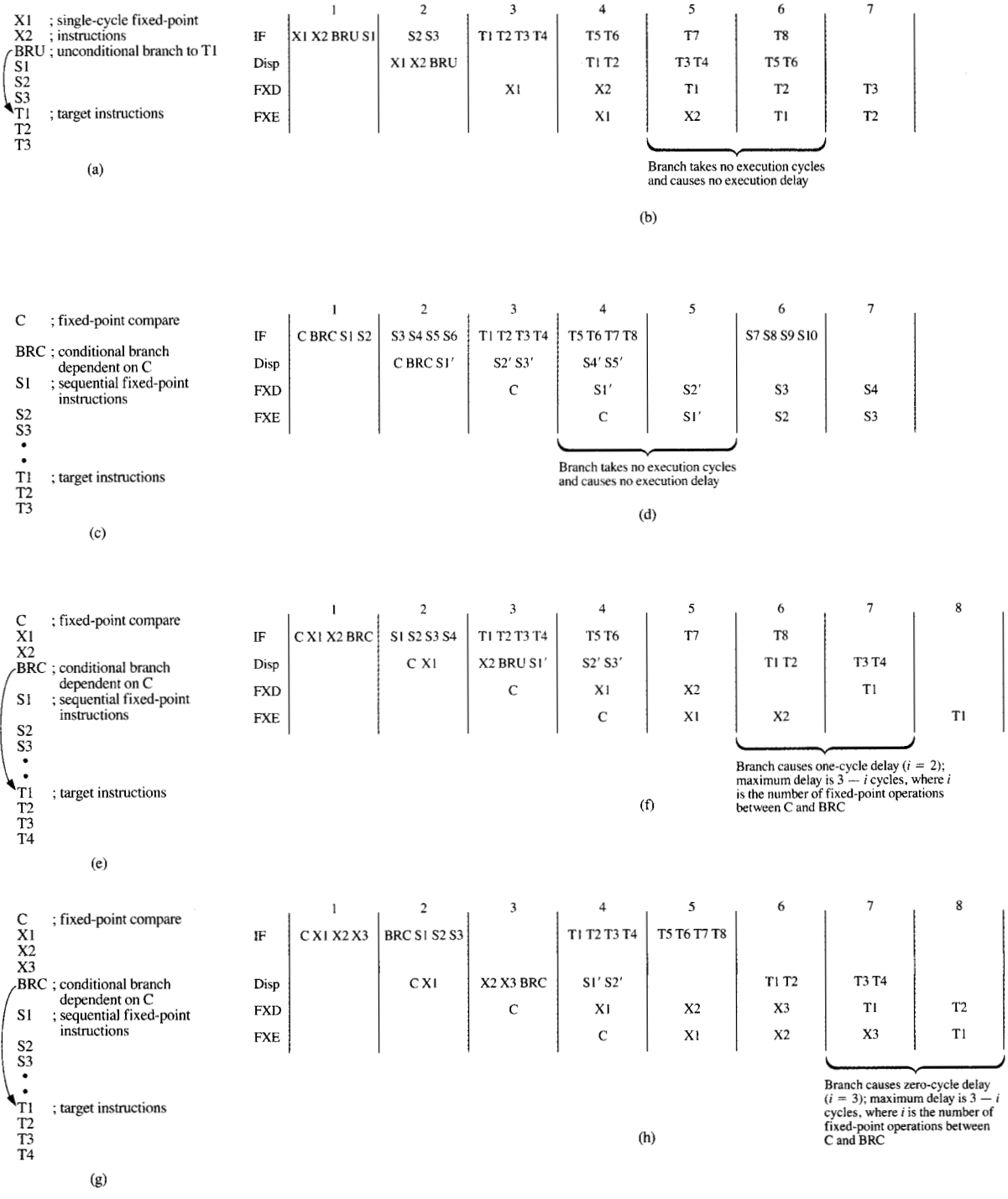
Several problems needed to be solved. One was how to synchronize the fixed- and floating-point units to maintain precise interrupts and still allow a high rate of instruction processing. The second was how to allow floating-point loads to proceed when the state of the floating-point register file was unknown to the fixed-point unit. The third was how, with the fixed-point unit performing address calculations for floating-point loads and stores, to ensure that the correct data was loaded into or stored from the floating-point register file.

#### *Synchronization*

One design goal was to keep the fixed- and floating-point units overlapped sufficiently that the execution rate of floating-point code would depend only upon either 1) the rate at which data could be fetched from the data cache or 2) the rate at which arithmetic operations could be performed, considering the effects of dependencies, by the floating-point arithmetic dataflow. That is, the MFLOP rate of the processor should not be limited by the instruction-processing characteristics and synchronization requirements of the pipeline. The synchronization scheme must allow the simultaneous processing of fixed- and floating-point instructions. It must also maintain precise interrupts for loads, stores, and trap instructions. When an interrupt for one of these instructions occurs during the execution phase of the fixed-point pipeline, all prior instructions must complete, and no subsequent instructions may alter the machine state. A final objective of the synchronization scheme was that it be simple to debug, preferably by inspection.

The synchronization scheme which was used in the AMERICA timer is diagrammed in **Figure 5**. The major pipeline stages are depicted. In the fixed-point unit, a set of instruction-prefetch buffers (IPB0-IPB3) feeds two decode registers, D0 and D1. The IPBs allow the branch unit to get ahead of the fixed- and floating-point units. A mux (multiplexer, not shown) between D0 and D1 feeds the selected fixed-point instruction to the register file and pipeline controls. This instruction is then logically fed to the execute cycle, where ALU, shift, address translation, and cache-directory operations are completed.

On the floating-point side, a mirror image of the instruction buffers and decode registers is provided. Decode registers PD0 and PD1 feed the rename registers (register renaming is discussed shortly) R0 and R1. These registers feed floating-point instruction-decode buffers (IDB) which in turn feed the floating-point decode register. The IDBs are provided so that the fixed-point unit is not held up waiting for floating-point arithmetic instructions to complete.



**Figure 4**  
 Examples of branch execution: (a) An unconditional branch. (b) Pipeline behavior for the branch shown in (a). (c), (d) A conditional branch that is not taken. (e), (f) A conditional branch that is taken and causes a one-cycle pipeline delay. (g), (h) A conditional branch that is taken and causes no pipeline delay.

The fixed-point unit and the early stages of the floating-point unit are kept in lock step by synchronization signals which are passed between the two units. During fixed-point decode, pipeline-hold conditions produce a signal which prevents the floating-point unit from pre-decoding. Thus, registers D0 and PD0 and D1 and PD1 always contain the same instruction. Similarly, during the fixed-point execute phase, signals are generated and passed to the floating-point unit which tell whether the instruction in execution completed, or caused an interrupt. The instruction in fixed-point execution is always in either register R0 or R1. Only floating-point loads, stores, and arithmetic instructions pass from registers R0 and R1 to the IDB and decode stages. Fixed-point instructions that enter R0 and R1 are discarded as soon as they are executed by the fixed-point unit (or are interrupted). If an instruction in fixed-point execution causes an interrupt, the contents of registers R1, D0, D1, PD0, PD1, and both sets of instruction buffers are purged. Additionally, the contents of R0 may or may not be purged, depending upon whether the content of R0 is a floating-point load, store, or other fixed-point instruction. If so, it is purged, since it is precisely the instruction causing the interrupt. If not, it must be a prior floating-point arithmetic instruction, and is allowed to proceed.

Similarly, a hold signal can be produced by logic in the floating-point rename stage. If the IDB becomes full, or the renaming mechanism runs out of rename registers, the floating-point unit tells the fixed-point unit to hold execution.

Figures 5(a-f) illustrate the operation of this synchronization scheme on one iteration of the loop of Example 1. In Figure 5(a), the first two instructions have been dispatched to the fixed- and floating-point units. The fixed-point unit decodes the floating-point load and discards the FMA. The floating-point unit pre-decodes both instructions and transfers them to the rename stage.

In Figure 5(b), the next two instructions enter D0, PD0, D1, and PD1. The fixed-point unit is performing the address generation and translation for UFL FR8. Similarly, the floating-point unit is remapping UFL FR8 and FMA FR10. If the UFL should cause an interrupt, the fixed-point unit informs the floating-point unit via the cancel line to cancel all instructions in rename and above. Any instructions in the IDBs or floating-point decode are not affected. Similarly, if the floating-point unit runs out of IDB space or rename registers, it informs the fixed-point unit to hold in the execution stage.

In Figure 5(c), the second UFL is in fixed-point execution, and the first FMA has entered floating-point decode. At the end of this cycle, the floating-point data returns from the data cache, and execution of the FMA starts as data is bypassed to the execution pipeline.

Figure 5(d) continues the sequence. In Figures 5(e) and 5(f), the instructions from the next iteration of the loop have entered decode and pre-decode and the process repeats.

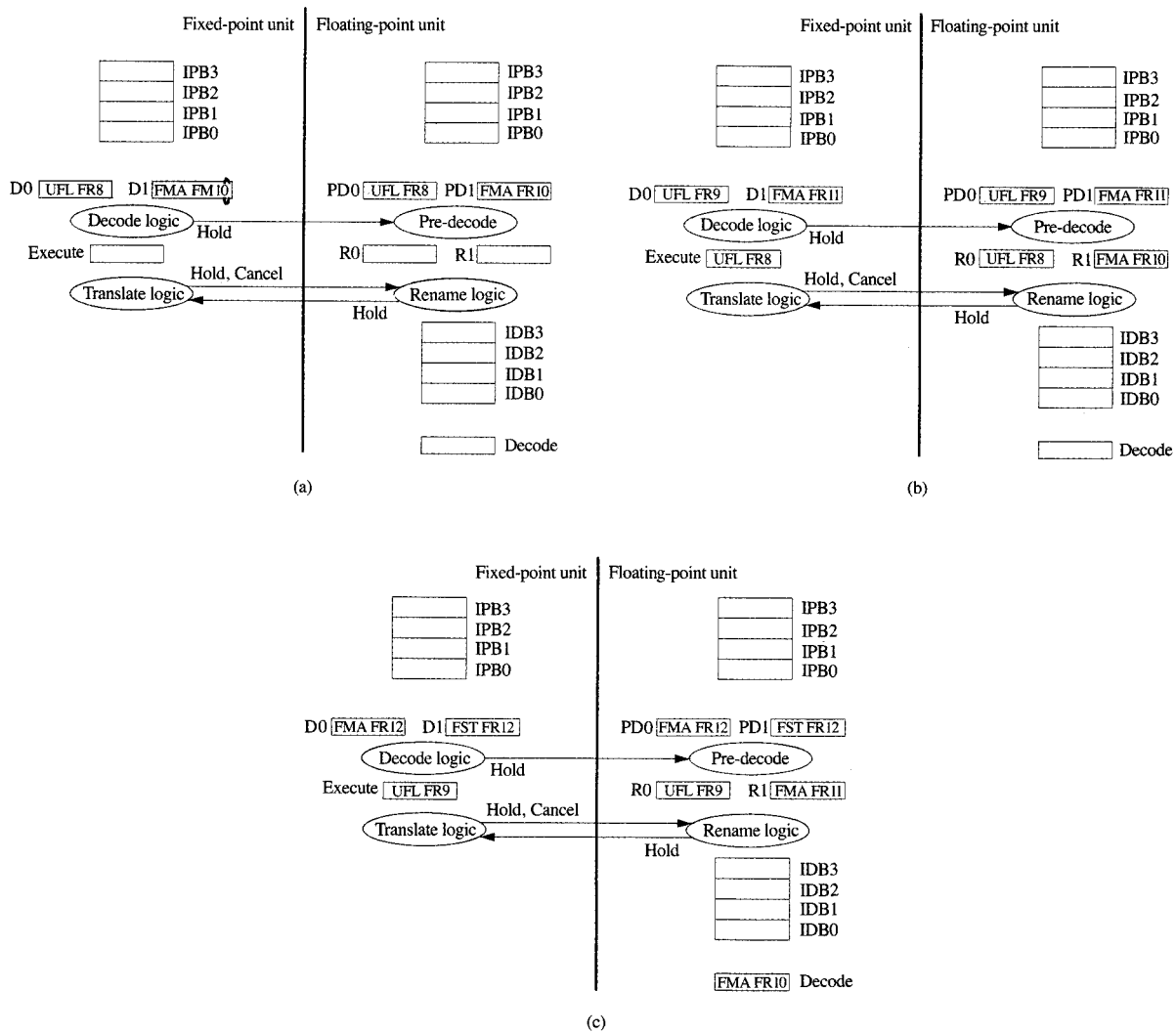
This synchronization scheme allows the floating-point and fixed-point units to operate in an overlapped manner without inhibiting the processing rate of the pipeline, and it maintains precise interrupts. It is easy to debug, since the contents of registers must correspond to one another. Although it adds two stages to the floating-point pipeline, this does not affect the processing of floating-point arithmetic instructions that need data from the data cache, since the data does not return until the floating-point decode cycle. It does, however, mean that the branch unit must wait an additional two cycles before resolving a conditional branch that depends on a floating-point compare. In the code sequences which were studied, this did not cause any great delay, since the compare typically depended upon a floating-point load (directly or indirectly) and thus could not have been executed any sooner.

#### *Register renaming*

When the fixed-point unit performs address generation and initiates the data-cache request for a floating-point load, the floating-point register denoted as the target of the load is overwritten with new data. The floating-point load can be considered to define a new value of the floating-point register (FPR). The FPR cannot be overwritten until all prior floating-point instructions which reference the old value of the register have accessed that value.

In the simplest implementations, the fixed-point unit is prevented from initiating a floating-point load until the floating-point unit signals that all previous floating-point operations are complete. In a coprocessor arrangement, this is acceptable; however, it severely limits fixed-floating-point overlap, which is required for high floating-point performance. With respect to Figure 5, the question is, when a floating-point load is in fixed-point execution, how is the fixed-point unit to know that the request can be sent to the cache? If the data returns too early, it will overwrite a value in the register file which may still be needed.

There are several solutions to this problem. One of the most elegant was invented by Tomasulo for the IBM System/360 Model 91 [16]. Floating-point data buffers are provided in the floating-point unit. When the instruction unit executes a floating-point load, it reserves one of these buffers. The instruction unit can proceed as long as the data buffers are not full. The load instruction is sent to the floating-point unit, so that the data can be placed in the floating-point register file at the appropriate point in the program sequence. When the load is decoded



**Figure 5**

Fixed-point–floating-point synchronization scheme used in the AMERICA timer: (a) First two instructions dispatched to fixed- and floating-point units. (b) Next two instructions enter D0, PD0, D1, and PD1. (c), (d) Second UFL in fixed-point execution, and first FMA enters floating-point decode; sequence continues. (e), (f) First instructions from next iteration of the loop enter decode and pre-decode; process repeats.

by the floating-point unit, the data is transferred to the register file from the buffer, once it is available.

A second approach was examined in the Cheetah machine. Two copies of the floating-point registers were architected. The primary floating-point registers, 32 in number, were used by the arithmetic unit. The backup registers, also 32 in number, were used by the fixed-point unit to load data from the data cache. Receive (RCV) instructions moved data from the backup register to the

primary register. The RCV operation could be coded as a bit in a floating-point arithmetic instruction.

This procedure worked in the following manner. A floating-point load would load the backup register with data. The first floating-point arithmetic instruction to use the data would have its RCV bit set for that register field. The data would be transferred from the backup register to the primary register. A valid bit would denote whether or not the backup register was in use. If it was, a

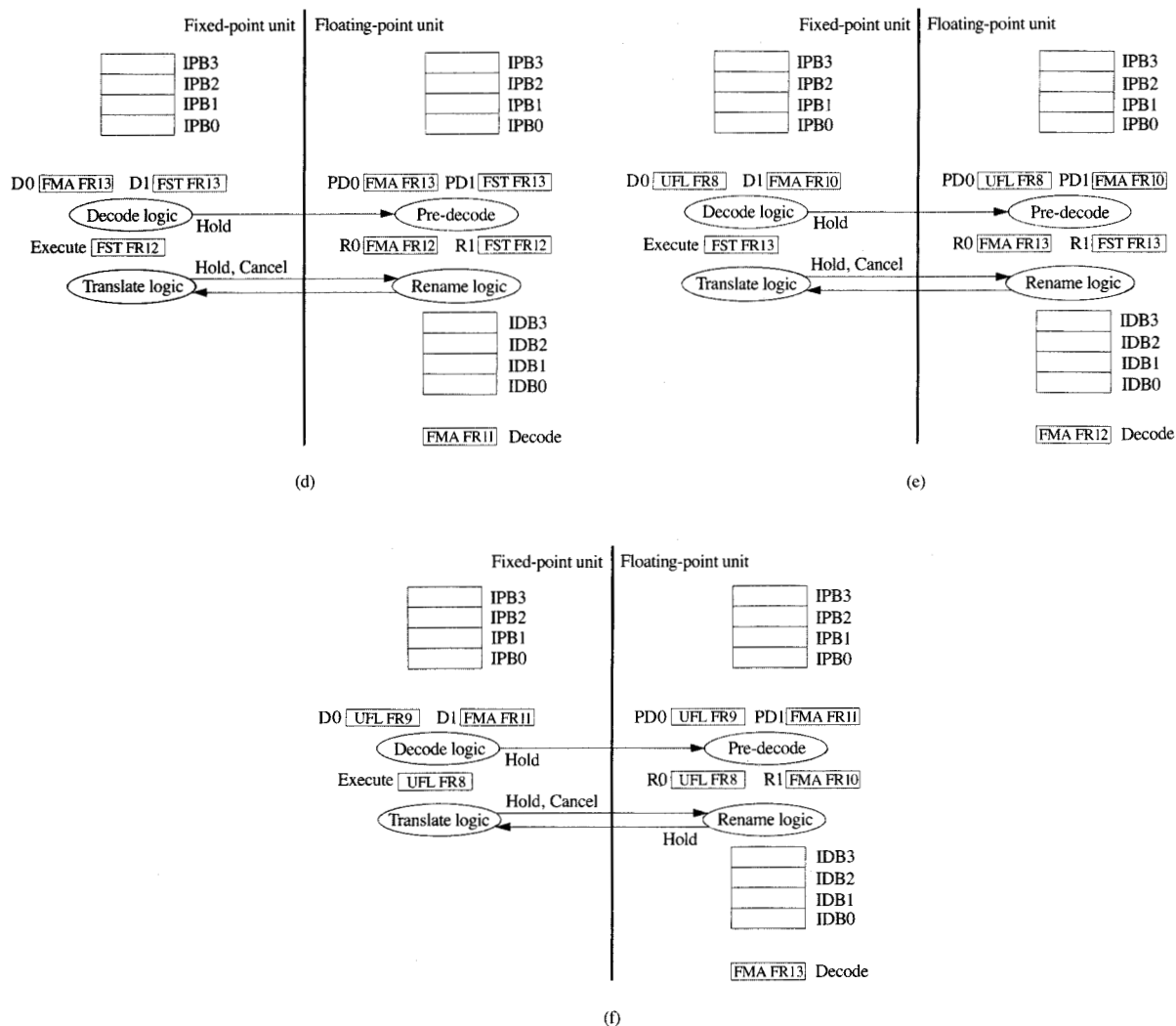


Figure 5 Continued

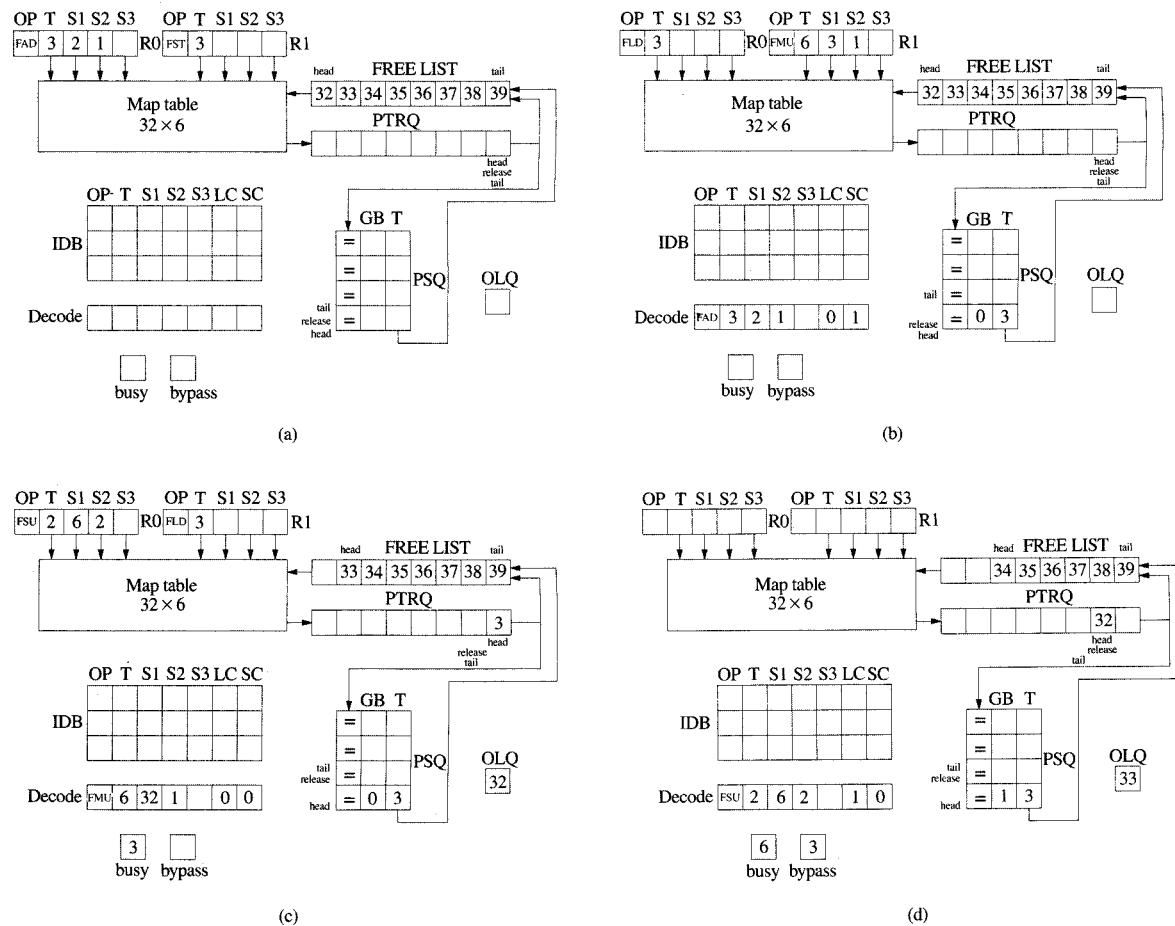
subsequent floating-point load to that register could not proceed and would be held up by the fixed-point unit. Otherwise, the fixed-point unit could effectively pre-load the next value of the FPR. This was particularly useful in floating-point loops, where the fixed-point unit would typically reach the top of the loop before the floating-point unit. Instead of waiting for the floating-point unit to catch up, it could proceed with loads and overlap them with the execution of prior floating-point code.

However, the backup registers doubled the size of the register file, and might not be utilized uniformly by the compiler. Receive and transmit operations were another chore for the compiler to get right. In a straightforward

implementation, 32 valid wires must be exposed to the fixed-point unit.

Simple interlocking, as in a typical coprocessor implementation, was too slow. The Cheetah approach was cumbersome. Instead, a variant of the Tomasulo approach, called register renaming, was developed.

A load of a floating-point register creates a new semantic value for that register. If there were a pool of physical registers, greater than the number of architected registers, the extra registers could serve as a dynamic buffer to hold data for floating-point loads executed by the fixed-point unit but not yet encountered by the floating-point unit. As long as there were some free



**Figure 6**

Register-renaming structure: (a) Map table initialized to identity. (b) Floating-point add passes from rename into decode. (c) Floating-point add in execution. (d) Store removed and tag returned to free list. (e) Subtract can decode, releasing register tag 32, which is returned during next cycle. (f), (g) Subtract completes execution.

physical registers, floating-point loads could be processed by the fixed-point unit without regard to the actual state of particular floating-point registers. Internal floating-point control logic would determine when to use the value in the buffer pool based on the decoding of floating-point loads and the registers being used by instructions in execution.

The organization that was adopted is illustrated in **Figure 6**. R0 and R1 are the rename registers. They contain an opcode field, a target-register field, and three source-register fields.

The map table is a 32-entry, 6-bit-wide table which maintains the correspondence of an architected register

to a physical register. For instance, if the entry for register 12 is 38, then physical register 38 currently contains the contents of architected register 12.

The free list (FL) contains a list of currently unassigned physical registers. In the initial state, the map table is initialized to identity and the remaining registers are placed on the free list. Since there are 40 physical registers, the FL can contain a maximum of eight entries. The FL is maintained as a circular queue and uses a head pointer and a tail pointer.

The pending-target return queue (PTRQ) contains those physical registers which are being used by instructions in the IDB or decode phases, and will

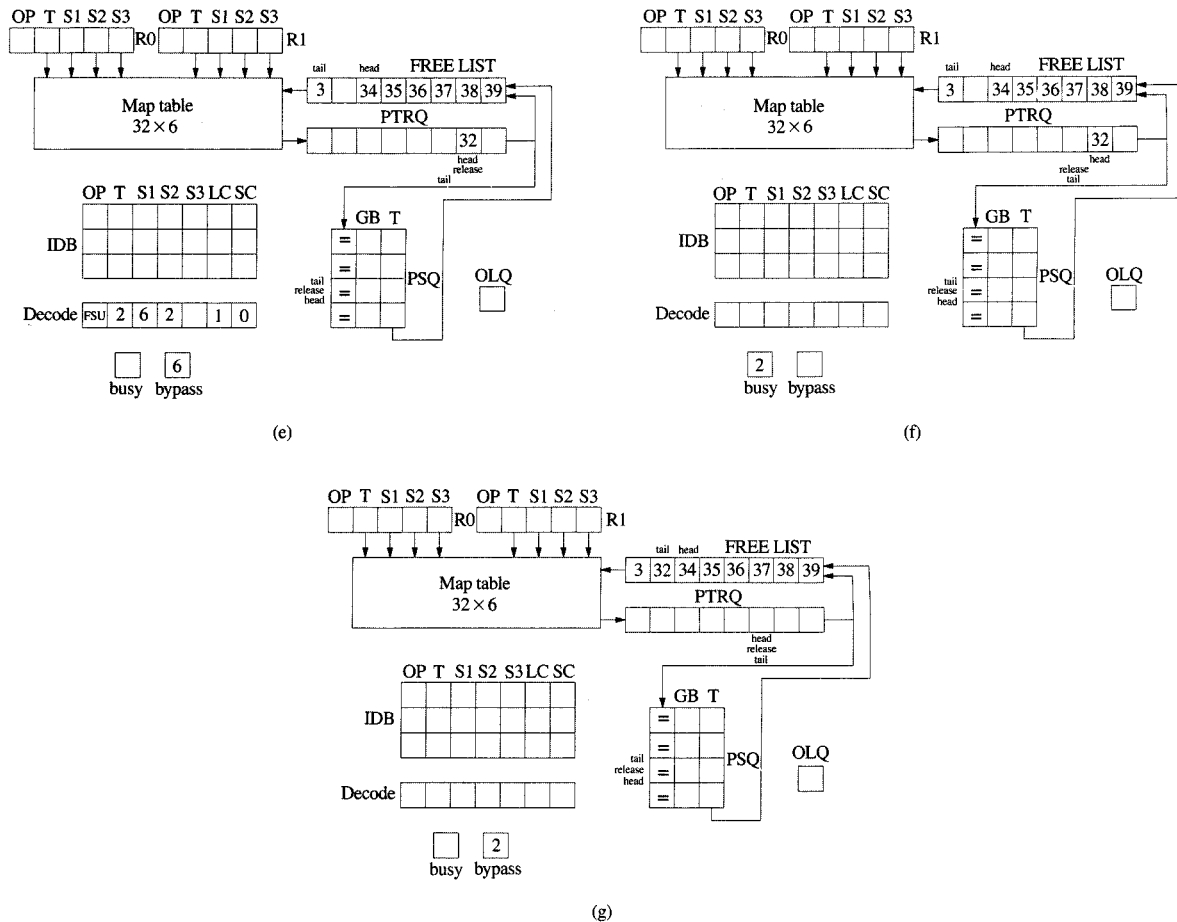


Figure 6 Continued

become free as soon as these instructions pass decode. It also has a maximum size of 8. Like the FL, it is maintained as a circular queue with head and tail pointers. It also has an additional pointer. The release pointer keeps register tags on the PTRQ until all prior arithmetic instructions which could have required the data in the corresponding physical register have decoded.

The decode stage contains floating-point arithmetic instructions. The instruction-decode buffers (IDB) buffer instructions which have been renamed but which cannot enter decode. They allow the fixed-point unit to run ahead of the floating-point arithmetic pipeline. The decode and IDB registers each contain load-count (LC) and store-count (SC) fields. When an instruction decodes, the LC field is used to increment the release pointer for the PTRQ, in order to release physical registers to the

free list. In a similar way, the SC field increments the release pointer of the store queue to allow floating-point stores to be performed.

The BUSY and BYPASS registers contain the physical register number of the floating-point instruction currently in the first and second execution stages. If any register field of an instruction in decode compares with the BUSY register, it is prevented from decoding. If a source field compares with the BYPASS register, the data is read from the execution pipeline and not from the register file.

The outstanding load queue (OLQ) contains the physical register number of the next floating-point load whose data will return from the cache. It stops instructions from decoding if they require data which has not returned from the data cache.

When floating-point stores are renamed, they are placed on the pending-store queue (PSQ). This eliminates the need for them to go through the floating-point decode phase. They remain on the queue until they are released by an arithmetic instruction decoding. Once released, they may be performed if their data is available.

When an instruction enters the rename phase (registers R0 and R1), what happens next depends upon the instruction type.

*Floating-point arithmetic instructions* When an arithmetic instruction is renamed, the contents of the map table are not altered. Each 5-bit architected register number is replaced by a 6-bit physical register tag. The instruction proceeds to the decoder, or, if the decoder is full, to an IDB position. If the arithmetic instruction is in R1, its LC and SC fields are set to 0. If it is in R0, these fields are set to 1 if there is a floating-point load or store in R1.

Once renamed, the arithmetic instruction enters the decode stage, if the decode stage is empty or is becoming empty; otherwise it is placed into an IDB. If the IDB is full, the pipeline backs up, and the fixed-point unit is told to stop executing instructions.

Once in decode, the arithmetic instruction reads out the contents of its physical registers. If the contents of any of its physical registers are not valid, because they are being loaded from memory or are being computed in the pipeline, the instruction remains in decode. These conditions are checked by comparing each physical register field with the OLQ and BUSY registers. When the instruction can successfully decode, it enters the arithmetic pipeline. The store-count field increments the release pointer of the PSQ to allow subsequent floating-point stores to be performed. The load-count field increments the release pointer of the PTRQ to release unneeded physical registers. These registers are then placed on the free list, as long as there are no stores on the store queue using this register which have not been done.

*Floating-point stores* When a floating-point store enters the rename stage, the target register is renamed to a physical register. The store could have been placed in the IDB or Decode stages, like arithmetic instructions. However, since the fixed-point unit executes fixed-point instructions and throws away floating-point arithmetic instructions, the floating-point unit must process loads or stores and arithmetic instructions in parallel. Otherwise, it will slow down the fixed-point unit. Thus, if the store were placed into the decoder, the decoder would have to inspect two instructions per cycle.

Instead, it is placed on the pending-store queue (PSQ). It remains there until the value of the physical

register it is trying to store is available. Before leaving rename, the store causes the store-count field of the most recent prior arithmetic instruction to be incremented. This is because the last instruction which could have produced the result is that arithmetic instruction (or a load). The store count keeps the store from being performed until all prior arithmetic instructions have decoded.

Before the store is actually performed (before the fixed-point unit is notified that data is available), several conditions must be met. First, the store must be at the head of the PSQ, since stores are done in order. The physical register tag of the store at the head of the queue must not be coming from the pipeline, or be busy from memory. These conditions are checked by comparing the physical register tag with the contents of the BUSY tag and the OLQ. Once the data is valid, the store is performed. If the give-back bit is set (see below), the tag is returned to the free list.

A few more words about floating-point stores are in order. While the fixed-point unit generates the address for a floating-point store, it must know when the data will be available. Again, in many coprocessor schemes, it would simply wait for the floating-point unit to produce the data and stop executing subsequent instructions. The RISC System/6000 uses a store-data buffer similar to that of the System/360 Model 91 [16] to allow the fixed-point unit to proceed to execute subsequent instructions. It works in the following way.

After generating the address for a floating-point store, the fixed-point unit places the address in a pending-store queue. When the floating-point unit removes the store from the store queue, it places the data in a data buffer on the floating-point chip, and informs the fixed-point unit that the data is available. Now that the data and address are available, the fixed-point unit can perform the store on any subsequent cycle.

*Floating-point loads* A floating-point load, since it defines a new semantic value for the architected register, causes the map table to be updated. When a load enters the rename stage, the map table is accessed for the target register. The tag stored there is placed on the PTRQ. It cannot be returned immediately to the free list since there may be pending floating-point arithmetic and store instructions which still need the value in that physical register. The tag at the head of the free list is written into the map-table entry. If there are no free tags, the fixed-point unit is informed to stop executing instructions.

The new physical register tag is then placed on the OLQ, and the LC field of the most recent prior arithmetic instruction is incremented.

Tags are returned to the free list from the PTRQ in the following way. The contents of a physical register become



unused (free) when the last arithmetic instruction or store referencing that physical register has been performed. For arithmetic instructions, this occurs when they complete decode. For stores, this occurs when they are removed from the store queue. When a load causes a new logical-to-physical mapping, the last instruction which could have used that physical register was the most recent arithmetic instruction, or floating-point store. (It may actually never have been used for many, many instructions.) Thus, when the most recent prior arithmetic instruction has decoded or store has been performed, that physical register can be returned to the free list.

Several mechanisms could have been used to determine this. The most straightforward would have been to compare the old physical register tag to all outstanding register fields of instructions in an IDB or the decoder, and to stores on the store queue. The last instruction to have matched would then be told to return the tag to the free list when it decoded or was removed from the store queue. This would have required up to 20 comparators, and 20 latch bits to ensure that the instruction returned the register.

An alternate method was developed. First, it is sufficient to let only arithmetic instructions return tags to the free list, if, before doing so, they check (via comparators) the contents of the store queue to ensure that there are no pending stores of this physical register. Then, each time a load renames a register, the old register tag is placed on the PTRQ. A counter, the load count (LCT) associated with each arithmetic instruction, is incremented. When the load decodes, the LCT field is used by the PTRQ controls to release that number of tags to the free list. This method eliminates the need for comparators with the IDB and decode stages. However, comparators and a bit are required for each store-queue entry. **Table 1** illustrates the procedure.

The operation of this example is shown in Figure 6. In Figure 6(a), the map table is initialized to identity, and all physical registers are free. The decoder, IDB, arithmetic pipeline, OLQ, and PSQ are empty. A floating-point add and a store are in registers R0 and R1. They are renamed, and pass to the decoder and the PSQ. The store-count field of the add is set to 1.

In Figure 6(b) the floating-point add has passed from rename into decode, since all required physical registers are free. Its store count increments the release pointer of the PSQ to release the store. A floating-point load and a multiply are in rename. The load causes a new mapping for architected register 3. It places the old mapping for register 3 on the PTRQ, and replaces it with register 32 from the free list. The release pointer for the PTRQ will automatically be incremented, since the add in decode (the most recent arithmetic instruction) will read out the

**Table 1** Register renaming for floating-point load instructions.

<i>Original stream</i>	<i>Rename table</i>	<i>Free head</i>	<i>Renamed stream</i>	<i>PTRQ</i>
FADD R3, R2, R1	(1, 1); (2, 2); (3, 3)	32	R3, R2, R1	
FST R3	(3, 3)	32	R3	
FLD R3	(3, 3)	32	PR32	3
FMUL R6, R3, R1	(1, 1); (3, 32); (6, 6)	33	R6, R32, R1	
FSUB R2, R6, R2	(2, 2); (6, 6); (2, 2)	33	R2, R6, R2	
FLD R3	(3, 32)	33	PR33	32

old value. The load will be placed on the OLQ. The multiply will be renamed, after which it passes into decode, with its LC and SC fields set to 0.

During the next cycle [Figure 6(c)], the floating-point add is in execution. Its target register is in the BUSY register. The store of register 3, while released, cannot be performed since the data is being produced by the add and is not yet available. The load is on the OLQ, and the data will return from the data cache during this cycle. The PTRQ controls will try to place physical register 3 on the free list. However, the store still needs the old value. The give-back bit for the store will be set. The multiply is in decode and will decode during this cycle, since there are no register interlocks. The last two instructions, the subtract and the final load, are in rename and will proceed into the decoder and the OLQ during the next cycle. The load count field of the subtract is set to 1.

During the next cycle [Figure 6(d)], the store can be removed, since the data is available from the add. Since its give-back bit is set, it will also return the tag to the free list. The last load is on the OLQ. The subtract must wait for the multiply to execute and will remain in decode.

During the next cycle [Figure 6(e)], the subtract can decode, releasing register tag 32 on the PTRQ. It is returned during the next cycle. One cycle later [Figures 6(f) and 6(g)], the subtract completes execution.

While register renaming appears to be a fairly complex control mechanism, it elegantly allows fixed- and floating-point instructions to be overlapped to take maximum advantage of each execution unit. In conjunction with the fixed-floating-point synchronization scheme, precise interrupts are maintained between fixed- and floating-point operations.

• *Maintaining the consistency of the instruction stream*

In a highly overlapped machine, interrupts may be precise or imprecise. An interrupt is precise if, when it is processed, no subsequent instructions have begun execution and all prior instructions have completed. Precise interrupts force the machine to preserve the view of a machine which executes one instruction at a time,

finishing it before processing the next one. Imprecise interrupts, on the other hand, allow the processor to leave the instruction stream in the neighborhood of the interrupt in a fragmented, but recoverable, state. For instance, all prior instructions may have executed, and some subsequent instructions may have begun execution and updated architected registers. In this case, it may not be possible to re-execute them. Imprecise interrupts therefore require the architecture to provide a means for reconstructing the instruction stream around the point of the interrupt, so that post-interrupt processing software can recreate the sequential state of the machine.

Due to the pipeline complexity of the AMERICA machine organization, it would have been difficult to architect a facility for handling interrupts in an imprecise fashion, which would have accounted for the many possibilities for instruction execution past the point of an interrupt. Therefore, precise interrupts were specified for all program-generated interrupts.

In order to guarantee precise interrupts, each interrupt type was analyzed, and a means of handling each in a precise fashion was developed. External, asynchronous interrupts were handled by stopping instruction dispatch and waiting for the pipeline to drain. If an instruction in the pipeline caused an interrupt, that interrupt was taken. Other interrupt conditions, such as invalid instructions, were detected in the ICU during the dispatch cycle, causing a wait for the pipeline to drain in a similar fashion.

The following mechanism was developed to ensure that interrupts resulting from the execution of load, store, or trap instructions by the fixed-point unit were precise. These three types of instructions are termed interrupt-causing (IC) instructions, although they are not the only instructions which can cause interrupts.

The fixed-floating-point synchronization scheme ensures that precise interrupts are maintained between the fixed- and floating-point units. Since no floating-point instructions could cause interrupts in AMERICA, the remaining need was to synchronize the branch-processing unit with the fixed-point unit.

From the pipeline structure it is apparent that branches and LCR operations executed by the branch unit change the count, link, and condition registers before the branch unit is informed that a prior IC instruction has caused an interrupt. Therefore, any changes to these registers must be undone to reflect their state at the time of the interrupt. In addition, the address of the IC instruction must be saved in SRR0 so that interrupt-handling software can process it.

The program counter stack (PCS) mechanism was developed to handle this. Each cycle, as instructions are dispatched to the execution units, logic-records the relationship of branches and LCR instructions which

modify the count, link, and condition registers to IC instructions. The addresses of IC instructions are also recorded on a stack. If a branch or LCR instruction is executed before a prior IC instruction has been executed, the old value of the count, link, or condition register is saved on a backup stack for that register.

As IC instructions are executed by the fixed-point unit, entries are removed from the PCS; old entries for the count, link, or condition registers are removed from their backup stacks, since no interrupt has occurred. When an IC instruction causes an interrupt, the head entry on the backup stacks is written to that register, and the address of the IC instruction saved on the PCS is saved in SRR0. Consider the following example:

```
1000 L      ;Load which will cause an interrupt
1004 CRAND ;LCR instruction which modifies CR
1008 BL     ;Branch and Link changing LR
```

Assume that these three instructions are dispatched during one cycle. The address of the load, 1000, is recorded on the PCS, and the load is sent to the fixed-point unit. The old value of the condition-register field modified by the CRAND instruction is recorded on the CR backup stack. The branch-and-link updates the value of the link register to 100C hex, and causes the old value of the link register to be placed on the link-register backup queue. Status bits are set which reflect the fact that changes to the CR and LR were made after an interrupt-causing instruction. When the load interrupts several cycles later, the PCS is accessed. The address stored there is placed in SRR0. The status bits cause the values for the link register and condition register to be restored, erasing any changes to the machine state.

### Design of the IBM RISC System/6000 processor

During the implementation of the RS/6000 processor, several notable changes were made to the processor specification. Additionally, several areas of the machine were developed in greater detail.

#### ● *Instruction cache*

As originally specified, the instruction-fetching mechanism could fetch four instructions per cycle as long as they were within the first 13 instructions of a cache line (out of a total of 16). While implementing the array access logic, it was noted that the same interleaving principle could be applied to the cache directories. By splitting the cache directories into even and odd components and supplying a "row-incrementation feature" to the even directory, four instructions could be fetched even if two cache lines were crossed. If the first instruction of the group was in an even cache line, the odd directory could be accessed with the same row address to search for the remaining instructions in the

successor cache line. Similarly, if the first instruction was in an odd cache line, the next cache line would be contained in the next congruence class in the even directory, requiring that the row address be incremented. This feature complicated the hit logic and replicated some comparators, but was introduced at no cycle-time penalty. As a result, assuming that both lines are present in a cache, four instructions can always be fetched as long as the lines are in the same 4-Kbyte virtual page. This case could also have been handled by interleaving the instruction TLB into even and odd pages, but this was not worth the implementation cost.

The instruction cache is organized as an 8-Kbyte, two-way set-associative cache, which has a 64-byte (16-instruction) line size. The size of the instruction cache was limited by that which could fit on the 12.7-mm-square chip used for implementing the RS/6000 processors. With an 8-Kbyte instruction cache, a miss ratio of less than 2% is expected. A 64-byte line size was chosen for three reasons. Simulation of IBM System/370 processors with this line size indicated that larger lines tended to increase miss ratios at this cache size. The package available dictated that the reload bus from memory be two instructions wide. With a 64-byte line size, eight cycles over this bus are required to reload a cache line when a miss occurs. A 128-byte line would require 16 cycles, and would tie up the instruction cache for too long. A cache with 32-byte lines requires twice the directory space of a cache with 64-byte lines. A two-way set-associative cache was used, since the behavior of a direct-mapped cache is worse, and the performance gain of a four-way set-associative cache is marginally superior to that of a two-way set-associative cache. Also, properly designed, the two-way set-associative cache would not lengthen the machine cycle, since there are many other paths in other pipeline cycles which could equally well determine the cycle time.

- *Branch processing*

One path that was known to be critical in AMERICA was the condition-code-setting path from the fixed-point unit to the branch unit. In one cycle the fixed-point unit was to execute a compare-type instruction and transmit the resulting condition code to the ICU chip, where the branch-resolution logic would determine the outcome in time to switch the instruction stream. The time taken to resolve the branch and perform the stream switch took longer than originally anticipated, so it was delayed until the next cycle. As a result, the compare-branch penalty was increased to three cycles from two cycles, as illustrated in Figure 4. Nevertheless, in the Dhrystone 1.1 benchmark [17], of 64 branches in the inner loop, only 20 cause any delay. The total delay is 46 cycles, so that branches are executed at the rate of  $46/64 \approx 0.72$  cycles per instruction. This is a considerable improvement over

the 1+ cycles per branch instruction typical of most current RISC processors.

- *Condition register*

One of the prime examples of how the 801 architecture was changed to support concurrent execution of many instructions is embodied in the condition register (CR) [1]. It contains eight fields, each of which can be designated to hold the results of a compare-type operation. It is located in the instruction cache and branch-processing unit. By having one interlock bit per field, eight outstanding operations can be maintained. When an instruction is dispatched to the fixed- or floating-point units, which sets CR field 0, the corresponding interlock bit is set. Subsequent instructions which try to read or to set that field remain in the instruction buffers. Eventually the fixed-point or floating-point unit executes the instruction, and the ICU is informed via the condition-register bus from that unit. Then, the interlock bit is reset. Since instructions which set the condition register can be dispatched conditionally, and canceled, any corresponding condition-register interlock bits which were set must be reset. A similar reset occurs for compares which are dispatched after IC instructions that interrupt. This is performed and maintained by the PCS.

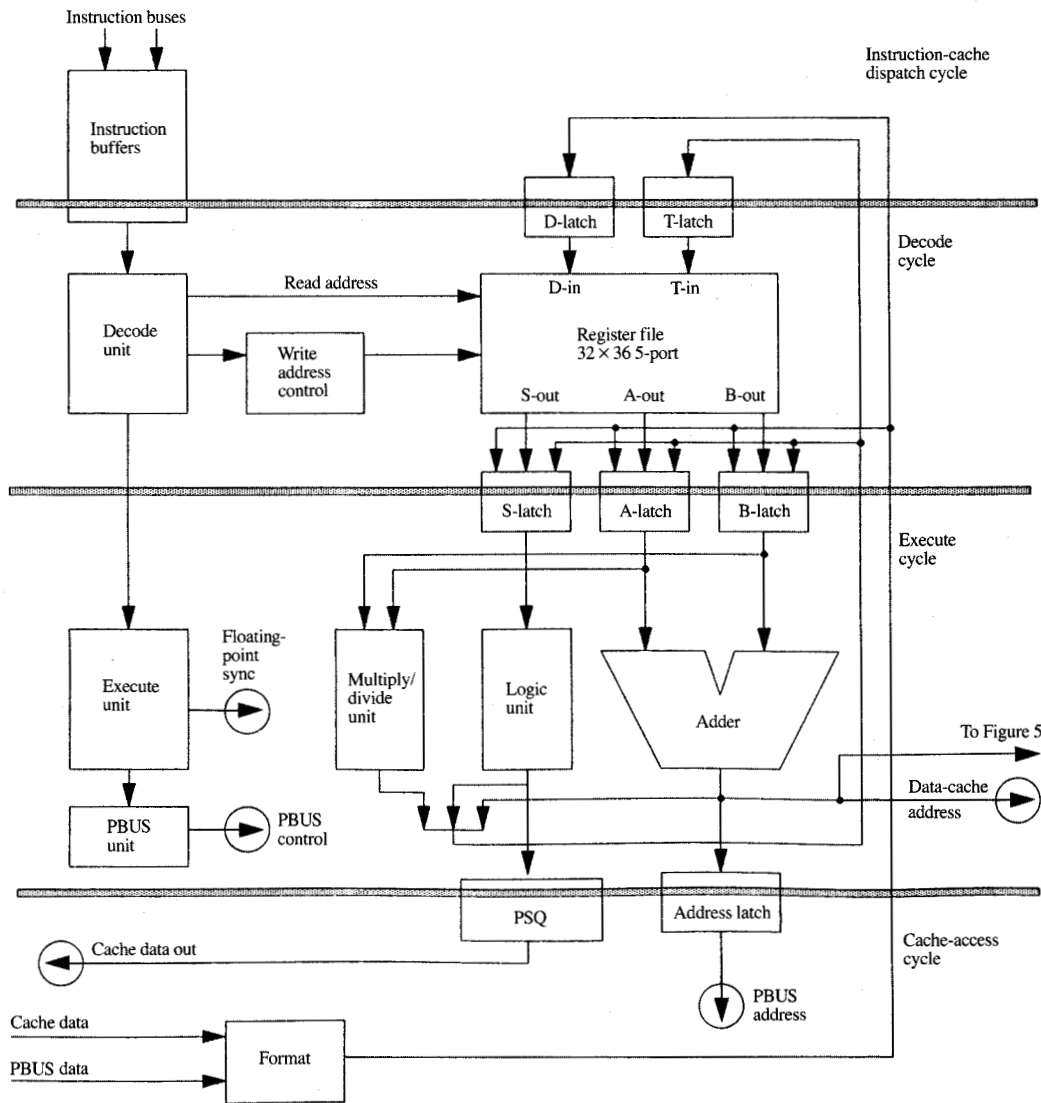
During the implementation, the amount of space dedicated to resolving condition-register interlocks in the dispatch and PCS logic became too large. Four interlock bits were introduced, so that four outstanding operations to the condition register could be maintained.

- *Synchronization of fixed-point and floating-point units*

The design of the fixed-point unit was such that instructions were held off in the execution phase rather than in the decode phase. As a result, the decode hold line shown in Figure 5 was not necessary. Also, the hold/cancel signal generated by the fixed-point unit was too late to make a chip crossing to the floating-point unit in one cycle. Therefore, instructions were permitted to enter floating-point decode, where they would be held off and canceled. This meant that, potentially, one rename cycle would have to be undone. Furthermore, many bugs were found in the synchronization scheme, so a counter was added to the fixed-point unit. Each time the floating-point unit shifted instructions out of rename, the shift amount was transmitted to the fixed-point unit. The fixed-point unit subtracted the number of instructions it shifted; the value of the difference specified the buffer position in the fixed-point unit of the instruction which was in R0 in the floating-point unit.

- *Register renaming and floating-point control*

Originally both floating-point arithmetic instructions and floating-point loads caused new logical-to-physical



**Figure 7**

General organization and pipelining of the fixed-point unit.

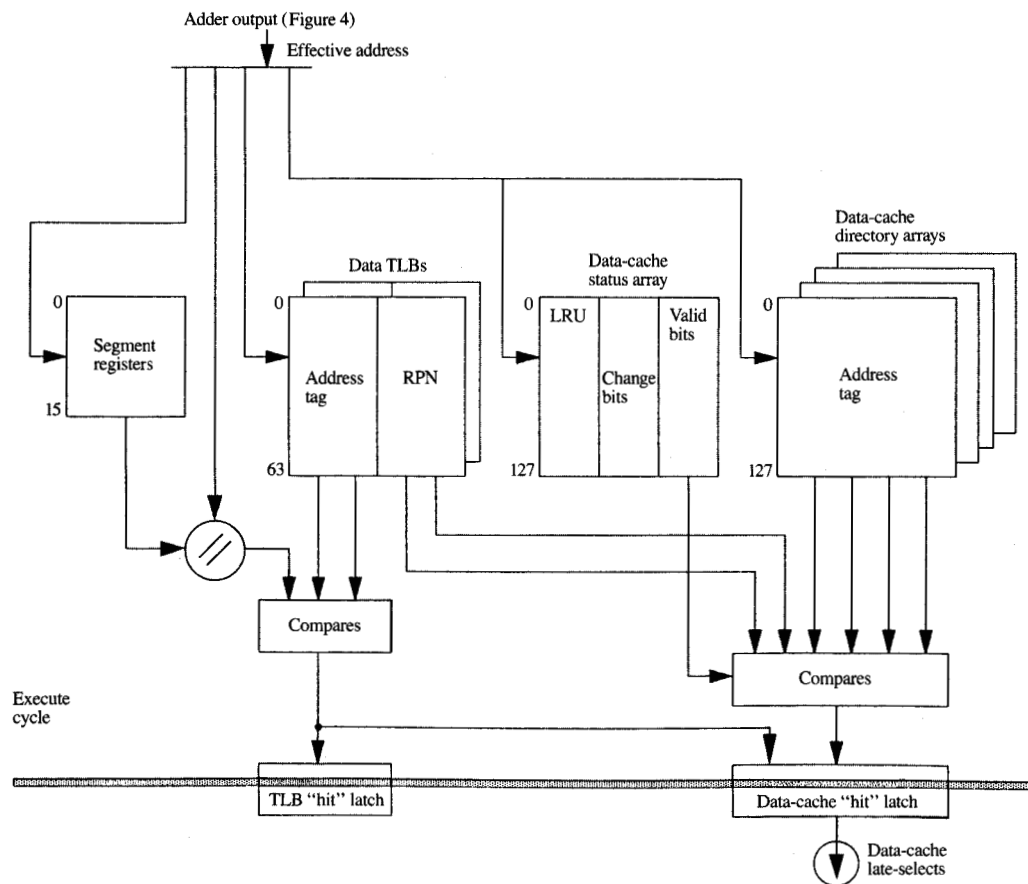
register mappings for their targets. Remapping arithmetic instructions complicates the renaming logic, since more ports are required on the PTRQ and the free list. It is only useful if arithmetic instructions can be executed out of order. In a sequence such as

```
FDIV FR0,    ;FR0 gets divide
FST FR0,     ;store FR0
FADD FR0,    ;FR0 used for add result
```

the floating-point add could proceed. Sequences like this

would not occur frequently, since the RS/6000 processor has 32 floating-point registers. Since the floating-point unit executed floating-point arithmetic instructions in sequence, there was no need to remap floating-point arithmetic instructions. This simplified the remap logic by requiring fewer ports on the AVRQ and PTRQ, as well as reducing the control logic complexity.

Another significant change was that the RISC System/6000 uses the IEEE floating-point arithmetic format, while AMERICA used the IBM System/370 format. This necessitated several changes to the controls.



**Figure 8**

General organization and pipelining of the data TLB and data-cache directory/status arrays.

The most significant was that floating-point stores were required to proceed through the floating-point decoder, so that any normalizations were performed properly. This is described in more detail in [2]. This degraded floating-point performance substantially in peak floating-point loops. For example, using the 2D graphics example described above, the RS/6000 machine takes seven cycles per loop iteration as opposed to four in AMERICA. On balance, however, this degradation is less severe; while the potential AMERICA LINPACK performance was approximately 15 MFLOPS, the RISC System/6000 achieves nearly 11 MFLOPS.

• *Fixed-point execution, data-cache access, and address translation*

The RS/6000 FXU is diagrammed in Figure 7. The basic 801 fixed-point execution-unit organization was sufficient

to provide an instruction execution rate close to one cycle per instruction. Thus, it was not a primary area of focus for improvement. However, some attention was given to making loads and stores operate quickly by placing the data-cache TLBs (translation look-aside buffers) and the data-cache directories on the fixed-point chip. Initial studies indicated that the following pipeline structure could be utilized. Starting at the beginning of the execution cycle, the address is generated by the ALU. This requires approximately one-half cycle. In the second half cycle, the segment registers are accessed, the virtual address generated, and the TLB and directories are searched in parallel. At the end of the cycle, it is known whether or not the access resulted in a hit or miss, and whether or not the access was permitted or caused a data storage interrupt. Also, during the last half cycle, the address is transmitted across a chip boundary to the

data-cache arrays and latched. This is diagrammed in **Figure 8**.

At the beginning of the next cycle, the late-selects are generated to select one of the four sets in the four-way set-associative data cache, and are sent to the data-cache chips. In parallel, the data-cache arrays are accessed to provide one word from each of four sets. The late-selects then select one word, which is transmitted to the fixed-point unit. It is formatted (sign-extended, rotated) as necessary, bypassed to the ALU and shifter, and latched for writing into the register file during the next cycle.

This cache organization therefore provides data in two execution cycles. An instruction using a register being loaded must wait one cycle before being executed. About two thirds of the time, the load can be scheduled back by the compiler, and this delay can be covered.

### Summary

The IBM RISC System/6000 machine organization uses multiple execution units to achieve high performance. A separate instruction cache which fetches four instructions per cycle effectively eliminates pipeline starvation. A robust branch-processing unit removes the execution pipeline penalties of most branch instructions without using a branch history table or elaborate branch-prediction mechanisms. By overlapping the execution of floating-point loads and stores with floating-point arithmetic operations, high floating-point performance is achieved. Precise interrupts are maintained to simplify system-interrupt handlers. The result is a powerful, robust processing platform which gives high system performance across a wide spectrum of application programs.

### Acknowledgments

The author would like to thank the many people of IBM Austin without whom the processor could not have been designed. Although too many contributed to be listed here, the contributions of several people are notable. Chuck Moore, Ed Boufarah, and C. C. Lee helped implement the instruction cache and branch-processing unit. Jim Kahle, Larry Thatcher, Dennis Gregoire, Paul Harvey, and Brian Bakoglu worked on the fixed-point unit. Myhong Nguyenphu, Daniel Cocanougher, Richard Fry, Pat Mills, Oscar Mitchell, and Troy Hicks worked on the floating-point unit. Brett Olsson analyzed the branch performance of the RISC System/6000 on the Dhrystone 1.1 benchmark.

### References

1. R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM J. Res. Develop.* **34**, 23-36 (1990, this issue).
2. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. Develop.* **34**, 59-70 (1990, this issue).

3. G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, in *ACM SIGARCH Computer Architecture News* **10**, No. 2, 39-47 (1982).
4. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM J. Res. Develop.* **11**, 8-24 (1967).
5. J. E. Thornton, *Design of a Computer—The Control Data 6600*, Scott, Foresman, & Co., Glenview, IL, 1970.
6. *IBM RISC System/6000 Technology*, Order Number SA23-2619, 1990; available through IBM branch offices.
7. J. F. Hughes et al., "Decode Branch History Table," *IBM Tech. Disclosure Bull.* **25**, 2396-2398 (1982).
8. J. K. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Designs," *IEEE Computer Magazine* **17**, 6-22 (1984).
9. J. J. Losq, "Generalized History Table for Branch Prediction," *IBM Tech. Disclosure Bull.* **25**, 99-101 (1982).
10. J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Symposium on Computer Architecture*, Institute of Electrical and Electronics Engineers, May 1981, pp. 135-148.
11. R. B. Garner et al., "The Scalable Processor Architecture (SPARC)," *Proceedings of COMPCON '88*, Institute of Electrical and Electronics Engineers, 1988, pp. 278-293.
12. C. Rowen et al., "RISC VLSI Design for System Level Performance," *VLSI Systems Design*, pp. 81-88 (March 1988).
13. M. Johnson, "System Considerations in the Design of the AMD 29000," *IEEE Micro*, pp. 28-41 (August 1987).
14. Tom Manuel, "Taking a Close Look at the Motorola 88000," *Electronics*, pp. 75-78 (April 28, 1988).
15. D. A. Patterson et al., "Architecture of a VLSI Instruction Cache for a RISC," *Proceedings of the 10th Annual Symposium on Computer Architecture*, Institute of Electrical and Electronics Engineers, 1983, pp. 108-116.
16. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Develop.* **11**, 25-33 (1967).
17. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Commun. ACM* **27**, 1013-1030 (October 1984).

Received February 28, 1989; accepted for publication January 17, 1990

**Gregory F. Grohoski** IBM Advanced Workstations Division, 11400 Burnet Road, Austin, Texas 78758. Mr. Grohoski received a B.S. with distinction in electrical engineering from Cornell University in 1980 and an M.S. in electrical engineering from the University of Illinois at Urbana-Champaign in 1981. That same year he joined the IBM Research Division at the Thomas J. Watson Research Center in Yorktown Heights, New York, where he worked on high-performance RISC machine designs. In 1986 he transferred to IBM Austin to work on the RISC System/6000 project. Mr. Grohoski holds two IBM Invention Achievement Awards and an IBM Outstanding Technical Achievement Award; he has applied for five patents. He is currently an Advisory Engineer in the hardware architecture group.