PIPE: A HIGH PERFORMANCE VLSI PROCESSOR IMPLEMENTATION

G. L. Craig
J. R. Goodman
R. H. Katz
A. R. Pleszkun
K. Ramachandran
J. Sayah
J. E. Smith

Computer Sciences Technical Report #513

September 1983

PIPE: A High Performance VLSI Processor Implementation

G. L. Craig, J. R. Goodman, R. H. Katz[1], A. R. Pleszkun,
K. Ramachandran, J. Sayah, J. E. Smith

Computer Sciences Department
Electrical and Computer Engineering Department
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT: PIPE is a high performance computer architecture with several features that make it well-suited for VLSI implementation. A PIPE "machine" consists of a memory controller and two co-processors. An access processor *(A-unit)* computes operand addresses for the execute processor *(E-unit)* that performs the main calculations of a program running on a PIPE machine. The A- and E-units, communicating through *architectural queues,* execute their own decoupled instruction streams. In our initial implementation, the units are identical pipelined processors. Program execution is overlapped within the processors through pipelining and among the processors through decoupling. The processor incorporates an on-chip *instruction cache* for high speed operation despite limited communication bandwidth. We describe the design of the prototype PIPE processor. Portions of the processor have been layed out in nMOS technology and submitted for fabrication.

KEY WORDS AND PHRASES: Decoupled Architectures, Pipelining, VLSI Processor Implementation.

1. Introduction

High performance machine design must exploit parallelism at all system levels, both within processors through pipelining and among processors through multiprocessing. We have been designing a high performance single processor with an aggressive pipelined implementation, as part of a very tightly coupled multiprocessor system. We believe that future high performance systems will be built from many very powerful processors, such as that described in [PATT80], rather than very many very simple processors. The implementation efforts reported here are an adjunct to our architectural studies. Our goal is to discover organizational approaches that are well-suited for VLSI implementation, and to provide useful performance feedback to our design efforts. While we plan on showing feasibility by implementing pieces of our machine using the technology available to us, we do not wish to limit our investigations to what could fit on a university-designed chip today. We feel confident that the processor we have designed could be integrated on a single chip within our five year time frame.

[1]Current Address: Computer Science Division, E.E.C.S. Department, University of California, Berkeley, Berkeley, CA 94720

Advanced research on high performance computer architectures is difficult to pursue within the university environment. Industrial designers have available more advanced processes [MIKK81] and greater knowledge of circuit engineering [BAYL81a, BAYL81b, BUDD81, KAMI81]. The most ambitious university projects exhibit levels of integration an order of magnitude less dense than commercially available components (e.g., the RISC processors [FITZ81, SHER82] were implemented with approximately 45,000 transistors, MIPS [HENN81] with 25,000, yet the HP FOCUS chip [BEYE81] contains 450,000 transistors!). High performance machine design demands a number of talents. While architectural innovations can significantly contribute to overall system performance, aggressive circuit designs, advances in integrated circuit technology, and exotic packaging cannot be neglected. We have focused on architectural innovations well-suited for the VLSI environment.

PIPE, Pipelined Instructions with Parallel Execution, is a decoupled access and execute architecture [COHL81, SMIT82, PLES83]. A program can be viewed as two coupled components, one that computes the addresses of operands and another that performs calculations over these operands. PIPE mirrors the dual nature of programs. A PIPE "machine" consists of a memory controller and one processor that computes addresses and fetches operands for a second processor that executes the main calculations of the program. The processors and the memory controller are linked through hardware queues. In this first implementation, the Access Processor (A-unit) and Execute Processor (E-unit) are identical, although they execute separate instruction streams. This paper describes the implementation of the PIPE processor, which functions as either the A-unit or E-unit of a PIPE machine. Its implementation features include an on-chip instruction cache, hardware queues, and a pipelined organization.

The rest of the paper is organized as follows. In the next section, we give more details on the PIPE architecture, in particular those features that have important effects on the implementation. Section 3 describes the initial implementation of the PIPE processor. The PIPE datapath has load and store queue subsystems, a shifter functional unit, and an ALU functional unit. The latter has two pipeline stages and forwarding logic to route ALU results back as inputs to the ALU. The instruction unit has three stages: (1) instruction fetch, with an on-chip instruction cache, (2) instruction decode, and (3) instruction issue, with pipeline conflict resolution and interlocks. We then discuss the memory-processor interface and some of the design issues of the memory controller. Section 4 describes the lessons learned. Our status and future directions are given in section 5.

2. PIPE Architecture

The PIPE architecture is more fully described in [SMIT83a]. In this section we concentrate on its aspects that most influence its high performance implementation.

PIPE is a decoupled architecture that has a streamlined instruction set suitable for a pipelined implementation. Pipelining is a good match for VLSI. VLSI circuits are characterized by a very large number of transistors to interface pins. While a pipelined processor requires more logic than a serial processor, it does not need additional pins. Performance is increased with a modest additional cost in chip real estate and complexity. This is one way to use the additional transistors available through advances in integrated circuit technology. Another is to place more storage on-chip, in the form of registers and/or caches. PIPE does the latter by incorporating an instruction cache on-chip.

A decoupled architecture [COHL81, SMIT82, PLES83] is one in which two (or more) processors execute a single process divided into parallel instruction streams. The division is functional: operand address computation is separated from the main program calculations. The processors coordinate their parallel execution by communicating through architectural queues. Decoupling supports parallelism within conventional programs, without precluding multiprocessing at the level of the process. The access processor (A-unit) calculates memory addresses and makes memory references for both processors. Architecturally, each processor has two store and one load queue between it and the memory controller (see figure 1).

Data is fetched by the access processor for the execute processor (E-unit) as follows. The A-unit calculates a memory address and issues an *alternate* load request, i.e., load the operand into the "other" processor (the E-unit). The memory controller fetches the operand and places it into the load queue of the E-unit. When the E-unit needs to access data, it reads from its load

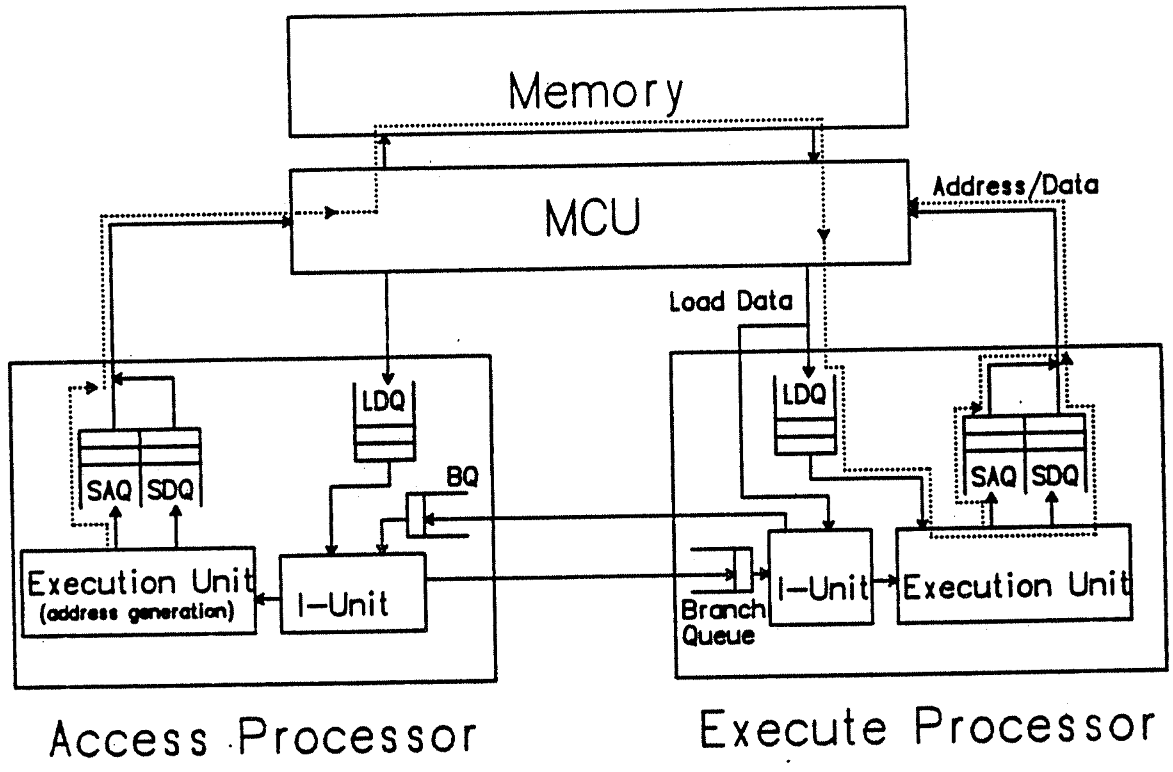

Figure 1 — Decoupled Access/Execute Architecture

queue.

Data is written to memory by placing the address on the store address queue and the data on the store data queue. As above, an alternate store address can be provided to the memory controller. The top store address element and store data element are paired and written to memory. To insure that both processors branch the same way on a conditional test, Boolean results are exchanged through additional branch queues. The intent of a decoupled architecture is that the access processor should run ahead of the execute processor and reduce or eliminate observed memory delays; we have observed that this is the case [SMIT82].

The key for achieving a high performance implementation is an *elemental* instruction set, as has been clearly demonstrated in the CDC 6600, CDC 7600, and CRAY-1 machines. An elemental instruction is one whose resource requirements can be readily determined before instruction execution. An instruction can use any resource available to it during a single flow through the execution pipeline, even if the action it performs is quite complex. Pipeline conflict detection and interlocks can be implemented in a single *instruction issue* stage that precedes the execution pipeline. The "one cycle" constraint makes the issue conditions relatively easy to determine. While simple instructions are usually elemental, the opposite need not be true. Consider a postincrementing load instruction that simultaneously forms a load address from the contents of a register, while replacing the register with its contents plus an immediate field.[2] Although the instruction might be considered complex, it is elemental in that (1) its resource requirements are easy to determine, (2) they are few in number, and (3) the instruction can be executed in one cycle. During the first stage of the execution pipeline, the register is routed along the result bus from the register file to the load address register, as well as to the ALU along the source bus. During the second execution stage, the summation with the immediate value is completed and routed along the result bus back into the register file. The resources are the register, the result bus into the register file, and the pipeline to memory. The use of the result bus during stage 1 and stage 2 can be scheduled at instruction issue time. The only additional resource requirement beyond a conventional load instruction is the additional need for the result bus.

For this reason, register-to-register architectures are better suited for pipelining than memory-oriented architectures. The resource and time requirements of register-to-register operations are known in advance. In addition, register-to-register architectures allow data to be loaded from memory in advance of when it is needed by an instruction. In a storage-to-storage architecture, data is not loaded from memory until the time it is needed, thus adding to the observed instruction latency.

[2]PIPE has such an instruction.

PIPE's repertoire includes three-address register-to-register and queue-to-register arithmetic/logical/shift instructions, queue-to-register load/store instructions, and branch instructions. Instructions are blocked from issue because of (1) read-after-write hazards, (2) load queue empty, (3) store queue full, and (4) result bus conflicts. The instruction set has been selected to simplify the detection of these conditions.

Load and store data pass through PIPE's architectural queues. These are the LOAD DATA QUEUE (LDQ), the STORE DATA QUEUE (SDQ), and the STORE ADDRESS QUEUE (SAQ). The STORE queue tails and LOAD queue head are encoded as one of the processor's registers in the register file (foreground register 7). This eliminates the need for separate queue manipulation instructions. Queues are essential for the decoupled operation described above. A further advantage is that a general purpose register need not be allocated for an operand that is only used once. Queues also provide subtle advantages for pipelining. At instruction issue time, a load instruction does not need to reserve a path to the register file for the load data. Otherwise, interlocks would need to deal with the unpredictable memory system response time. The hazard condition of loaded data not yet available is encoded in the empty/not empty status of the load queue.

Condition codes cause notorious problems for pipelined machines. Their setting or testing causes hazard conditions analogous to those of the registers. If every instruction could potentially change the condition codes, then there is a write-after-write hazard among all non-branch instructions. PIPE does not have condition codes. The computation of the effective target address, the evaluation of the branch condition, and the transfer of control are specified separately. The PIPE architecture has eight branch registers (BRs) that are loaded under program control with branch target addresses. That a branch will be taken is determined by a prepare-to-branch instruction (PBR), which specifies the *branch condition,* the *branch register* containing the target address, and a *branch distance* (up to 7) of instruction parcels (16 bit words) to execute before the transfer of control occurs. The separation of transfer of control from determination of transfer gives the instruction fetch logic advance notice of a branch, thus providing a smoother flow of instructions. The prepare-to-branch concept was first proposed in [SCHO71].

The register file is divided into a foreground and background bank of eight registers each. The dual register banks support fast procedure calls. Background registers are dumped to memory or loaded with parameters as needed before a call. The foreground/background status of the banks are swapped on call and return.

3. PIPE Implementation

A block diagram for the PIPE processor is given in figure 2. The pipelined instruction unit contains three stages: instruction fetch and cache, instruction decode, and instruction issue. The pipelined execution unit contains two functional units, a one stage barrel shifter and a pipelined two stage arithmetic logic unit. The datapath width is 16-bits, with two register banks of eight

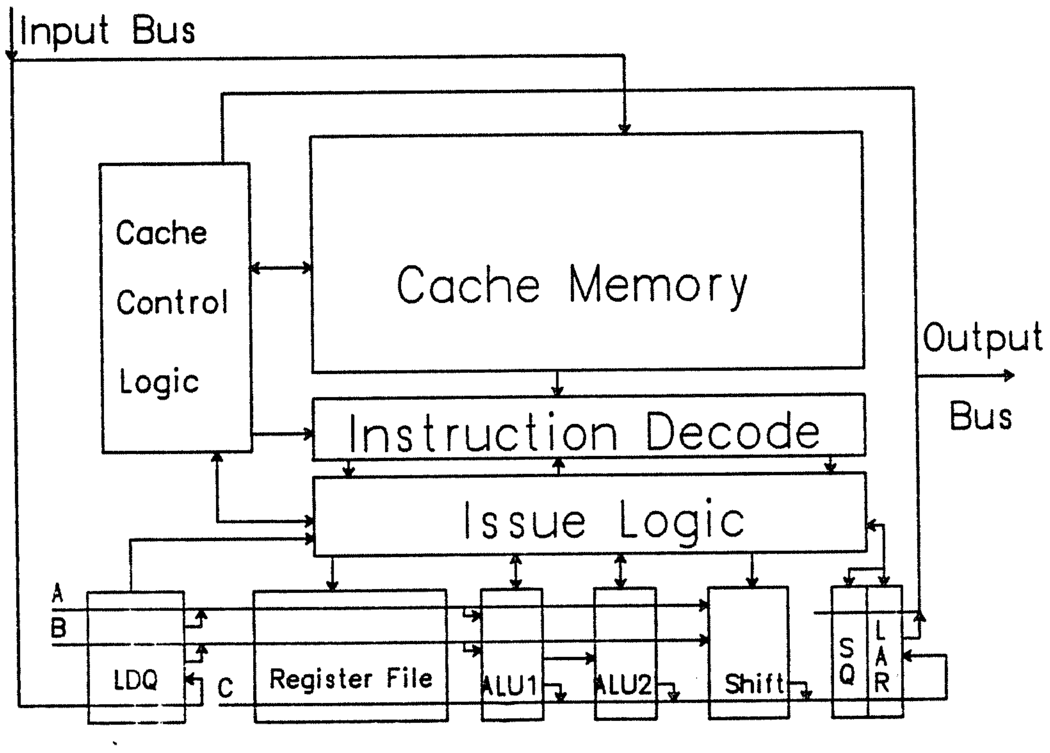

Figure 2 – Block Diagram of PIPE Processor

registers each, the shifter, and the ALU. The LOAD and STORE queues are part of the memory-to-processor interface. Pins to and from memory are unidirectional, rather than multiplexed, to avoid delays induced by switching input/output direction. Traffic between the processor and the memory controller is tagged. On output, the tags distinguish between instruction addresses, load and store addresses (both for the issuing processor and its "alternate"), and store data. On input, they distinguish between instructions and load data.

3.1. Implementation Goals

Our hypothesis is that a well-designed, elemental instruction set can lead to a well-structured pipelined implementation. Our purpose is to demonstrate that the PIPE architecture is suited for VLSI implementation and to identify timing bottlenecks that limit performance. Understanding the effectiveness of the architecture's features is more important than comparing its implementation with commercially available processors. What distinguishes PIPE from other university designed processors is our focus on the implementation of a pipelined instruction unit with hardware interlocks and an integrated instruction cache, and the pipelined execution unit (separate functional units for shift and arithmetic/logical operation, the latter with a two stage pipeline).

Some aspects of the implementation have been simplified to keep it manageable. PIPE is a 16-bit processor with a 16-bit address space (separate instruction and data space). While there is nothing inherently difficult about implementing a 32-bit datapath, this would have limited the available area for the more interesting control unit. PIPE single parcel (16-bit) arithmetic/logical/shift instructions and double parcel (32-bit) load/store/immediate instructions. The on-chip instruction cache was kept small (64 16-bit words), as were the number of registers in the register file (16 x 16-bit words).

In a further simplification, we have ignored the problems of trap and interrupt handling by postulating the existence of an external interrupt handler. Handling interrupts is complicated because of the large amount of state information, kept in the queues, which must be saved on an interrupt. PIPE can only be interrupted when it is convenient to do so, i.e., when the memory and branch queues are empty. This is analogous to the situation in microcoded machines, where the machine cannot be interrupted after an arbitrary microinstruction.

3.2. Datapath Description

The datapath is organized around a conventional three bus architecture: two source buses (A, B) and one result bus (C). The source buses are precharged during clock phase PHI2 and are used during PHI1. The result bus behaves conversely. The datapath contains a source to result bus by-pass, a register file, shifter, ALU, load data queue, load address register, and a single register that implements both the store address and store data queues (see figure 3). The load queue behaves like a read-only register, written by the memory-processor interface. Similarly, the store queue is like a write-only register, read by the memory-processor interface. To facilitate bus routing, these are located at opposite ends of the datapath. Only the source buses pass through the load queue, while the result bus passes through the store queue. The buses are run in metal, with

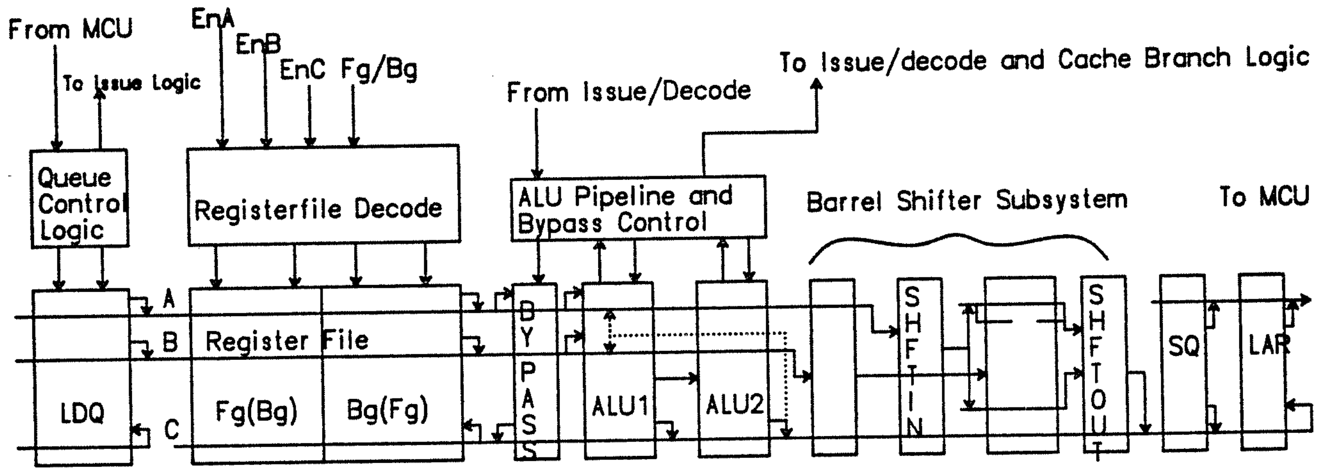

Figure 3 – PIPE Data Path

control in polysilicon running perpendicular to it.

3.2.1. Conventional Data Path Objects

The register file is a tri-ported (two read ports and one write port) array of 16 x 16 bit registers. It is organized into two banks of eight registers each. Arithmetic/shift/logical instructions implicitly refer to the foreground bank. Data is transferred within and between these by a collection of move instructions. The foreground/background status is swapped on procedure call or return.

The barrel shifter is an adaptation of the RISC-II shifter, described in [SHER82]. Since the shift amount is specified in a general purpose register, rather than a literal as in RISC, we use a different layout for the control registers (see figure 4). A 31 bit L bus, a 16 bit R bus, and a 16 bit S bus pass through the barrel shifter. Surrounding the shifter in the datapath are a ShftIn register for the shifter input, a ShftDecode latch to map the low order 4 bits of the B-Bus into the 16 S control lines, and a ShftOut register for the shift result. An additional bus routes the ShftIn

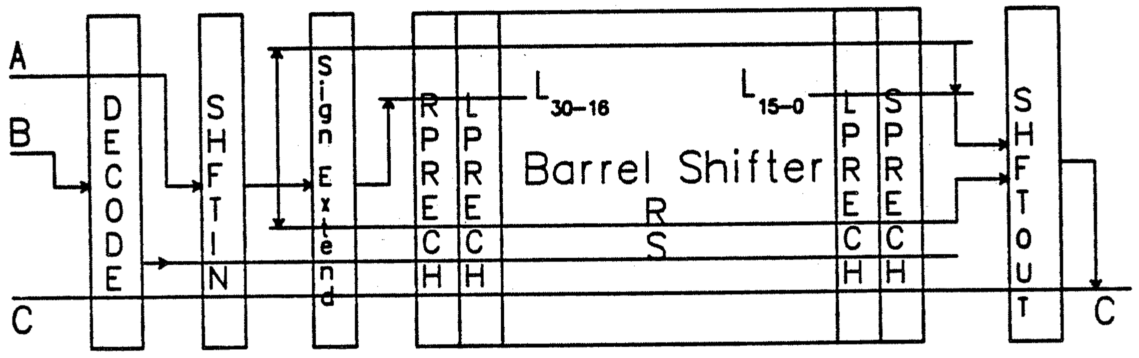

Figure 4 — Barrel Shifter Subsystem

data to the other side of the shifter for rotating shifts. This does not alter the pitch of the data-path, since the source buses do not pass through the shifter subsystem.

Data is placed onto the L and R buses in complemented form. A rotate shift is accomplished by gating the ShftIn register onto both L[30:16] and L[15:0], and gating to ShftOut from R. A logical right shift is performed by placing ShftIn onto L[15:0] and reading R into ShftOut. L[30:16] are precharged high. When these bits are shifted onto R and read into ShftOut, they are complemented and appear as logical 0. A logical left shift is similar, except that ShftIn is gated to R and the L[15:0] is gated to ShftOut. An arithmetic right shift is the same as the logical right shift except that L[30:16] is selectively discharged if ShftIn holds a negative number.

3.2.2. Bus By-Pass

The PIPE instruction set supports general data movements between registers, either within banks or between them. Data could be passed through the ALU on its way to a new register. However, to support post-incrementing loads and stores, we chose to incorporate a by-pass between the A-bus and result bus. During a post-incrementing operation, an operand address is formed from the contents of a register. The register is afterwards incremented by an immediate operand within the instruction. The register contents are latched into the by-pass and the first stage of the ALU during PHI1. During PHI2, the register contents are latched into the load address register and the ALU stage 1 computation completes. The summation is completed by the ALU stage 2 during the next clock cycle. The result is written back to the register file during PHI2.

3.2.3. Two Stage ALU Pipeline

The ALU consists of five functional parts: (1) the function generator, (2) condition flag generation, (3) the carry lookahead, (4) the sum stage, and (5) the overflow signal generation. An ALU result can be routed back to the inputs through an internal forwarding path when it is used as an operand in a subsequent operation.

The function generator forms all bitwise logical functions and creates the generate and propagate signals for addition and subtraction. The condition flag generator is a zero test circuit for the A-Bus plus the sign bit, and is used for conditional branches. The carry lookahead circuitry generates 4-bit, 8-bit, and 12-bit carry signals for use by the four bit sum stages. The sum stages operate on a 4-bit wide slice and generate four sum outputs. The overflow bit is generated from the high order three bits to indicate a two's complement overflow.

The ALU timing is spread across four clock phases: stagel/phil, stagel/phi2, stage2/phil, stage2/phi2. During stagel/phil, the inputs to the first stage become available from either the register file or the queue through the A- and B-Buses, or the second stage of the ALU through the internal forwarding path. Logical operations are performed here. If the operation is arithmetic, the propagate/generate signals are also produced. The zero condition is evaluated.

During stagel/phi2, results from a single stage ALU operation are gated to the C-Bus. The zero flag and sign bit are made available to the control logic. The two level carry-lookahead logic calculates the 4-bit group carry-in signals. The propagate and generate signals are latched for the next stage.

During stage2/phil, the group carry and the propagate and generate signals are examined. The sum is generated, and is immediately gated to either or both of the A- and B-buses if the forwarding path has been enabled by the control logic. The feedback result is latched by stagel.

During stage2/phi2, the sum result is gated to the C-bus. Overflow is determined from the carry in and carry out of the result's sign bit.

The ALU forwarding path takes advantage of the observation that the first stage of the ALU, stagel/phil, must include enough time to read data from the register file. The phil phase during stage 2 is actually much longer than necessary to compute the summations. Thus results can be forwarded back to stage 1 for immediate use during the same clock phase in which they are formed.

3.2.4. Queue Subsystems

PIPE's load and store queues provide a buffered interface to memory. A request for an operand load is separated from its use. Several load addresses can be sent to memory before the first operand is actually used in some instruction. The compiler must schedule its generated code appropriately to take advantage of this feature. Similarly, several store addresses can be placed in the store queue before the first store data is placed there.

Major design decisions related to the queue subsystem include queue lengths, how they are to be partitioned across the processor and the memory controller, and what queues can be combined. We are currently investigating how long these queues should be. On-chip queue lengths are kept small, while the extensions of the queues within the memory controller are significantly longer. Furthermore, there is no advantage in separating the on-chip store address and store data queues, since these must be multiplexed over the single output pins to memory. In the initial PIPE implementation, the store address and store data queues are implemented on-chip by a single element STORE QUEUE (SQ). However, the SAQ and SDQ are implemented as separate queues in the memory controller. Tags distinguish between addresses and data. The memory-processor interface is responsible for forwarding these to the memory controller. A status bit indicates when the on-chip queue is full, thus blocking store address or store data instructions from issue.

The top three elements of the load queue are implemented on-chip. The queue consists of a three element register file and two bit arrays for head and tail pointers. The load queue behaves as a read-only register. The queue stages are implemented with cells identical to those of the register file. Because of the three address instruction format, either one or two elements can be removed from the queue at a time. To simplify the implementation, we restrict generated code to make only one reference to the load queue per instruction. The queue subsystem has two output ports that are connected to the datapath's two source buses. A removed element can be directed to either bus. The queue subsystem input port is connected to the memory input bus. Data can be placed in the queue by the memory-processor interface during PHI1. An element can also be removed during PHI1. The full/empty status bits change on the PHI2 phase of the operation cycle that makes the queue full or empty. A reset input resets the head and tail pointers and sets the queue status to be empty.

Forwarding logic allows data to be passed from the memory input bus to either of the source buses directly. Thus data can be gated onto the buses in a single cycle without first latching it into the queue.

3.3. Instruction Unit

Our philosophy is that the instruction unit should never fetch an instruction from memory that will not be executed. The relatively small number of pins available on an integrated circuit package imposes a significant bottleneck on traffic to and from memory. Thus, the instruction fetch unit will not prefetch instructions from off-chip unless it is known, through the prepare-to-branch mechanism, that they will be executed.

The instruction unit consists of a cache/fetch stage, a decode stage, and an issue stage. The cache responds to requests from the fetch unit, initiating a memory transaction when a desired instruction is not found in the cache. The unit fetches instructions for the decode stage, interpreting branch instructions to continue the orderly flow of instructions through the pipeline. The decode stage interprets the instruction, and sets up the signals to control its execution in the two stage execution pipeline. The issue logic compares the instruction to be issued with those already in the execution pipeline, and determines whether any conflicts could arise. If so, the instruction is blocked awaiting completion of the conflicting instruction in the pipeline. These will be described in more detail in the following subsections.

3.3.1. Instruction Cache/Fetch Unit

An on-chip instruction cache is integrated with the fetch logic of the control unit of the PIPE processor. An on-chip instruction cache is attractive because (1) its control logic is simple to implement and (2) it reduces the traffic from memory. We have chosen a small cache size of sixteen lines of four by sixteen bit words because of our limited chip area and design expertise.

Clearly a much larger cache could be built by experienced designers in state-of-the-art technologies. While the choice of cache size is critical for effective performance, the actual size does not affect the implementation details of controlling it and interfacing it with the fetch logic. Further, we have done cache studies that indicate that the performance of a large cache can be extrapolated from a smaller one's performance [GOOD83].

The cache subsystem is organized as shown in figure 5. For simplicity, the cache is direct mapped. Our studies indicate that more complicated organizations are not significantly more effective [SMIT83b]. A ten bit tag word is associated with each 64 bit line of the cache. To make

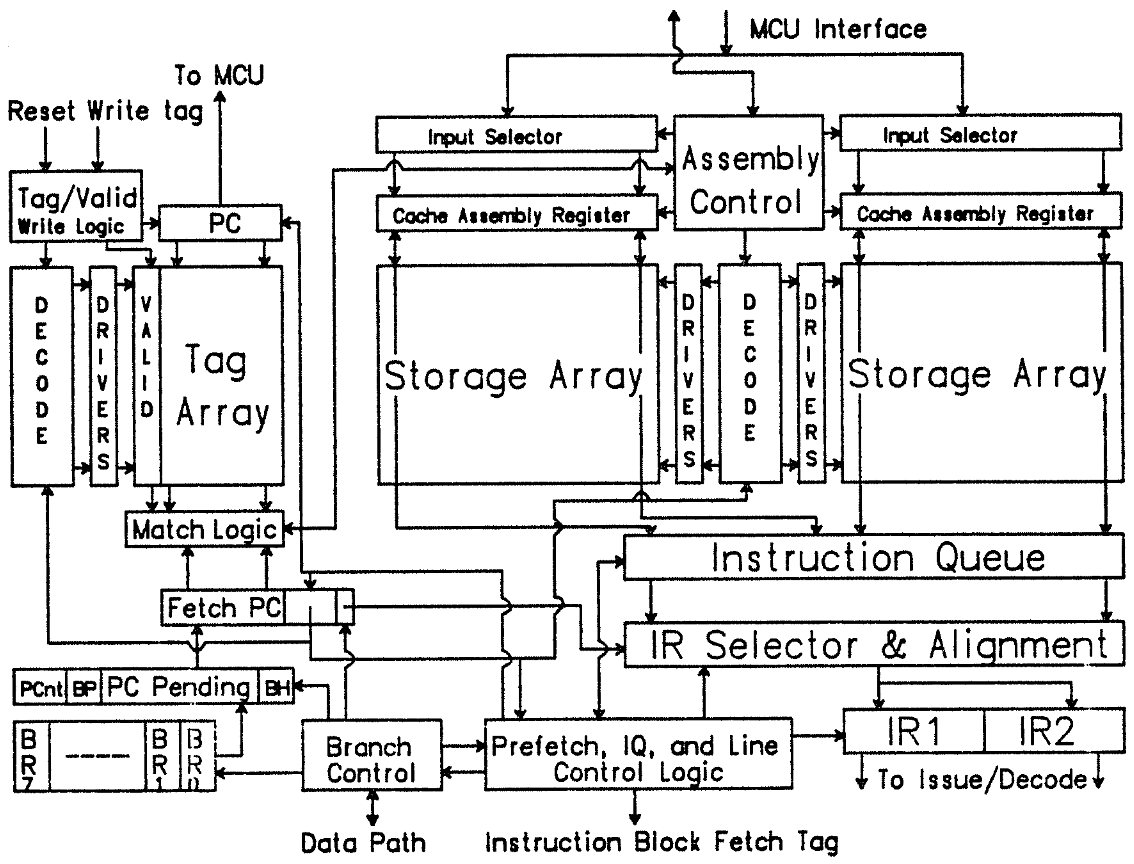

Figure 5 -- Cache Subsystem Block Diagram

cache lookup fast, separate decoders are used for the line and the tag. The line decoder is placed in the middle of the array.

An instruction is accessed as follows. When the instruction decode stage of the control pipeline is ready for another instruction, the fetch program counter is presented to the cache decode. The 64 bit line and 10 bit tag are read from the cache. The tag is compared with the high order bits of the PC, and if a match occurs, the appropriate word of the line is gated to the instruction register. If there is no match, the entire 64 bit line is replaced from memory. An instruction fetch request is made to the memory-processor interface control logic. Eventually instruction words are returned in word order (i.e., line word 0, line word 1, ...) and are assembled in the cache registers. The entire line is rewritten into the cache, and is simultaneously passed through for selection and latching into the instruction register.

The PIPE instruction set has been designed to enable the processor to gracefully change control flow. Branch target addresses are separately loaded into Branch Registers (BRs). A transfer of control is indicated by a Prepare-to-Branch instruction, which specifies (1) a condition to test, (2) a branch register, and (3) a number of instruction parcels to execute before the branch is taken. The parcel count field tells the fetch unit how many parcels can be fetched before the transfer of control.

The fetch unit works as follows. It detects when the decode stage is ready and provides it with another instruction parcel for decode. A fetch is normally initiated by incrementing the Fetch Program Counter (FetchPC), and starting a cache access. When an instruction parcel is received from the cache, possibly after being loaded into it from memory because of a miss, the fetch unit checks to see if it is a PBR instruction. The fetch unit decodes the PBR instruction to determine the maximum number of parcels that can be fetched and allowed to enter instruction decode and issue before the PBR instruction actually executes. Only instructions that will reach execution are fetched into the instruction pipeline. The PBR decode logic in the fetch stage sets a Branch Pending (BP) flag, loads the parcel count field into the Parcel Counter (PCnt), and loads the specified Branch Register into the Pending Program Counter (PendingPC). Before a cache access is actually initiated, the fetch unit checks whether a branch is pending, and if so, whether PCnt is greater than zero. Under these conditions, fetching proceeds as normal. After each fetch, PCnt is decremented by the same signal that increments the FetchPC. Fetching is inhibited whenever BP is asserted and PCnt is zero.

The branch condition is evaluated when the PBR instruction reaches execution. If true, the execution control sets the Branch-to-Happen (BH) status flag. Otherwise the BP flag is reset, permitting sequential fetching to continue. If PCnt is zero and BH is asserted, the fetch unit gates PendingPC to FetchPC. Future instructions will be fetched from the target address, thus control has been effectively transferred. Both BH and BP are reset by the fetch unit.

The fetch unit does not yet incorporate a prefetching strategy, other than that an entire cache line is fetched from memory at a time. The accessed line could be partially decoded to determine whether it would be executed sequentially because (1) it contains no PBR instructions, and (2) no delayed branch could be taken within the line. Bringing the next line into the cache could be overlapped with the execution of the current line. Since this significantly complicates the fetch unit logic, it was omitted in this initial implementation.

3.3.2. Instruction Decode

The decode logic is relatively straightforward. The instruction opcodes have been carefully chosen to make it easy to distinguish among classes of instructions, in particular, one versus two parcel instructions. The decode stage is responsible for generating the control signals for both stages of the execution pipeline.

Unfortunately, the decode logic has been complicated by two design decisions in the choice of instruction formats: (1) in certain contexts, R0 and R7 have special meanings, and (2) some instruction fields are overloaded. R7 indicates that the operand is to be found in either the load queue or the store queue, depending on whether it appears as a source or as a destination. R7, however, is not a phantom register. Return addresses on procedure calls are placed in background register 7. Thus some additional logic is needed to inhibit the register file when R7 is used in arithmetic/logical/shift operations, but not when implicitly used in a procedure call operation. Similarly, R0 means the constant zero in load and store operations, but is a conventional general purpose register in arithmetic/logical/shift operations.

The second problem is due to the overloading of certain instruction fields. A one parcel instruction (RRR format) consists of a seven bit opcode, and three three-bit register fields denoting the destination and two sources of the instruction's operands. A two parcel load/store/immediate instruction (LS format) consists of an opcode, one register field, six bits reserved for memory expansion, and a sixteen bit address/immediate field. Note that the first field in an RRR instruction is a destination, while it is a source in the LS format. A multiplexor selects the appropriate bits to forward to the register file.

While PIPE has only two formats, the assignment of functions to fields is not so straightforward. The move instruction takes two operands, and the third register field is actually used as an extension of the opcode to distinguish between foreground and background register movement. Another example of a non-uniform instruction format is found in the branch instructions. Two of the register fields refer to the parcel count and the target address branch register, rather than general purpose registers. Additional logic is needed to enable/disable the various units that interpret the instruction fields.

3.3.3. Instruction Issue

PIPE and the CRAY-1 have similar philosophies about instruction issue: resources needed by an instruction are reserved at issue time [CRAY76]. If an instruction to be issued would conflict with an already issued instruction in the execution pipeline, the instruction is blocked until the conflict is removed. The advantage of the approach is its simplicity: all pipeline control is resolved at a single point within the control unit. The disadvantage is the communications considerations: all status information about instructions in the pipeline must be made available to the issue logic to permit it to make the issue decision.

For PIPE, with its simple datapath and execution unit, the design of the issue logic is much simpler than in the CRAY. Since the execution unit is only two stages long, instructions are guaranteed to complete in issue order (this need not be the case for longer pipelines). Further, the possible conflicts are small. These center around the result bus into the register file and the status of the queues. If instructions could complete out of sequence, because of a longer execution pipeline, the registers would also have to be reserved at issue time.

A one stage arithmetic operation cannot be issued if the previous instruction issued was a two stage operation. This is because both would need the result bus during the same clock period.

Instructions which refer to queue operands cannot be issued if an inappropriate queue status is detected. For example, an arithmetic operation referring to R7 as a source cannot be issued if the LDQ is empty. Similarly, an instruction with R7 as the destination cannot be issued as long as the SDQ is full.

A subtle interlock involves the SWAP instruction, which changes the status of the foreground and background banks. The three-bit register fields are bound into a four-bit register descriptor at issue time. A register reference instruction is blocked from issue if the currently executing instruction is a SWAP.

The final interlock involves register read after write. Normally, the issue logic cannot issue an instruction if one of its source operand registers is the destination of an instruction already in execution. We have circumvented this problem by incorporating a result forwarding path within the ALU described in that section.

3.4. Memory/Processor Interface and Memory Controller

A unique feature of the PIPE implementation is dedicated input and output pins. Most processors time multiplex their pins, but then they must pay a time penalty when the tri-state pads switch from input to output and vice versa. PIPE's interaction with memory is more like interprocessor communication than a conventional microprocessor bus protocol. Several different kinds of information can be supplied on the input or output pins during each clock period. These are distinguished by associated tags. The To-memory and From-memory controllers operate independently, although one of the input tags is used to inhibit further output requests when the memory controller gets overloaded.

Output Tags:

0 0 0	No Memory Request
0 0 1	Instruction Address
0 1 0	Internal Load Address
0 1 1	Alternative Load Address
1 0 0	Store Data
1 0 1	Block fetch
1 1 0	Internal Store Address
1 1 1	Alternative Store Address

Input Tags:

0 0	Memory Busy (Inhibit To-Memory Requests)
0 1	No Input Data
1 0	Instruction Data
1 1	Data

4. Lessons Learned

Overloading the meaning of R7 was a significant design mistake. What looked like an elegant and natural solution of making the queues appear as general purpose registers significantly complicated the decode logic. The simplicity of the RISC implementations is in part due to the ease with which the instructions could be decoded. Had we a larger instruction size (32 bits), with more room for additional opcodes, we would have included separate queue manipulation operations.

On one occasion we discovered that our "streamlined" instruction set was not as elemental as we had thought. An instruction needed a combination of resources that could not be provided in a single instruction execution time. The problem arose as a side-effect of a proposed Prepare-to-Call instruction. PCall operates analogously to PBR, except that when the transfer of control occurs, the return PC is gated to BR7 and the foreground and background banks are swapped. These actions are done simultaneously with the execution of the last instruction before the branch is taken. A conflict could arise over the use of the result bus to the register file. The solution is to inject the branch actions into the instruction stream following the last instruction. Rather than make this a side-effect, we have replaced PCall by a collection of primitive instructions for

(1) indicating control transfer (the already existing PBR), (2) saving the return address (a MOV instruction is used), and (3) swapping the foreground and background register banks (through a SWAP instruction). Our machine has no special instruction for procedure call or return. A procedure call is performed by following software conventions using combinations of the three instructions above. Since PCall was not an elemental operation, we have replaced it by its elemental components.

5. Status and Directions

Thus far, we have implemented major pieces of the processor and have submitted these for fabrication. These include (1) the datapath, including the pipelined ALU, (2) the instruction cache and fetch unit, and (3) the queue subsystem. The instruction cache test chip is shown in figure 6. Work continues on completing the instruction pipeline. Our main goal has been to obtain performance estimates for the design, particularly timing and area data. We do not expect to be able to produce a working PIPE processor on a single chip in the near future. To do so would require us to be too conservative for our research goals.

The PIPE implementation project is only one of several related to high performance machine design at the University of Wisconsin-Madison. Research groups are focusing on the issues of (1) generating high performance, well scheduled machine code for decoupled architectures from high level languages, (2) designing a specialized access unit based on [PLES83], and (3) designing the Memory Controller. Work continues on extending the PIPE ideas to queue-based supercomputers of the future.

6. Acknowledgements

We wish to acknowledge the contributions of the members of Professors Katz's and Pleszkun's seminar classes in the Spring of 1983. The students helped us to refine the architecture and the implementation in a number of ways. Daniel Schuh helped with design tools support and reverse engineering of the RISC-II and MIPS designs, which were graciously made available to us by Professors David Patterson and John Hennessy and their students. Honesty Young was

responsible for the PIPE Pascal Compiler. Koujuch Liou aided in the analysis of the PIPE Memory Controller.

7. References

[BAYL81a] Bayliss, J. A., et. al., "The Instruction Decode Unit for the VLSI 432 General Data Processor," IEEE J. of Solid-State Circuits, V SC-16, N 5, (October 1981).

[BAYL81b] Bayliss, J. A., et. al., "The Interface Processor for the INTEL 432 32-Bit Computer," IEEE J. of Solid-State Circuits, V SC-16, N 5, (October 1981).

[BEYE81] Beyers, J. W., et. al., "A 32-Bit VLSI CPU Chip," IEEE J. of Solid-State Circuits, V SC-16, N 5, (October 1981).

[BUDD81] Budde, D. L., et. al., "The Execution Unit of the VLSI 432 General Data Processor," IEEE J. of Solid-State Circuits, V SC-16, N 5, (October 1981).

[CRAY76] Cray Research, Inc., Cray-1 S Series Hardware Reference Manual, HR-0808, (1976).

[COHL81] Cohler, E. U., J. E. Storer, "Functional Parallel Architecture for Array Processors," IEEE Computer Magazine, (September 1981).

[FITZ81] Fitzpatrick, D. T., et. al., "VLSI Implementations of a Reduced Instruction Set Computer," CMU Conference on VLSI Systems and Computations, H. T. Kung, B. Sproull, G. Steele, eds., Computer Science Press, Rockville, MD, (October 1981).

[GOOD83] Goodman, J. R., unpublished simulation studies, 1983.

[HENN81] Hennessy, J., et. al., "MIPS: A VLSI Processor Architecture," CMU Conference on VLSI Systems and Computations, H. T. Kung, R. Sproull, G. Steele, eds., Computer Science Press, Rockville, MD, (October 1981).

[HENN82] Hennessy, J., et. al., "Hardware/Software Tradeoffs for Increased Performance," ACM Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA, (March 1982).

[HENN83] Hennessy, J., et. al., "Design of a High Performance VLSI Processor," Third CalTech Conference on Very Large Scale Integration, R. Bryant, ed., Computer Science Press, Rockville, MD, (March 1983).

[KATE83] Katevenis, M., et. al., "The RISC II Micro-Architecture," Proc. VLSI 83, Oslo, Norway, (August 1983).

[KAMI81] Kaminker, A., et. al., "A 32-Bit Microprocessor with Virtual Memory Support," IEEE J. of Solid-State Circuits, V SC-16, N 5, (October 1981).

[MIKK81] Mikkelson, J. M., et. al., "An nMOS VLSI Process for Fabrication of a 32-Bit CPU Chip," IEEE J. of Solid-State Circuits, V SC-16, N 5, (October 1981).

[PATT80] Patterson, D. A., C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," IEEE Trans. on Computers, V C-29, N 2, (February 1980).

[PATT81] Patterson, D. A., C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," Eighth Annual Symposium on Computer Architecture, (1981).

[PLES83] Pleszkun, A. R., E. Davidson, "A Structured Memory Access Architecture," 1983 International Conference on Parallel Processing, Bellaire, MI, (August 1983).

[SCHO71] Schorr, H., "Design Principles for a High-Performance System," Symp. on Computers and Automata, Polytechnic Institute of Brooklyn, (April 1971).

[SHER82] Shernburne, R. W., et. al., "Datapath Design for RISC," Proc. Conference on Advanced Research in VLSI, P. Penfield, ed., Artech House, Dedham, MA, (January 1982).

[SMIT82] Smith, J. E., "Decoupled Access/Execute Computer Architectures," Ninth Annual Symposium on Computer Architecture, (May 1982).

[SMIT83a] Smith, J. E., et. al., "PIPE: A High Performance VLSI Architecture," IEEE Workshop on Computer Systems Organization, New Orleans, LA, (March 1983).

[SMIT83b] Smith, J. E., J. R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," Tenth Annual Symposium on Computer Architecture, (June 1983).