OVERVIEW OF THE PIPE PROCESSOR IMPLEMENTATION

Matthew K. Farrens
Computer Science Division
University of California, Davis
Davis, CA 95616
(farrens@american.ucdavis.edu)

Andrew R. Pleszkun
Department of Electrical and
Computer Engineering
University of Colorado-Boulder
Boulder, CO 80309-0425
(arp@tosca.colorado.edu)

ABSTRACT

The PIPE processor is an outgrowth of the PIPE Project, a research project at the University of Wisconsin-Madison whose goal was to investigate computer architectures that would be well suited to VLSI implementation. The implemented PIPE processor is a 32-bit pipelined single chip processor with a simplified load-store instruction set, a 5 stage pipeline, a two-cycle ALU, and the following unique features:

(1) Architectural I/O queues that lie between the processor internals and the external memory. These queues are used to reduce the impact of memory delays on processor performance.

(2) A delayed branch scheme that allows the compiler to specify the number of instructions after a branch that will be unconditionally executed (between 0-7) based on how well it was able to schedule code.

(3) A sophisticated instruction fetch mechanism featuring a small on-chip instruction cache, an instruction queue and an instruction queue buffer that together perform as well as a much larger conventional instruction cache.

(4) A register file that is divided into foreground and background registers, to improve the performance of subroutine calls.

Extensive simulations of the original design indicated that the features listed above provide significant performance improvements. However, it was felt that this combination of architectural features was sufficiently unique to justify an actual implementation of the processor, to investigate whether they would work as well in practice as in theory.

The processor was fabricated by MOSIS in 1.5 micron nMOS, and is 2-3 times faster than the 1.5 micron nMOS versions of the RISC and MIPS chips. This improved performance is due in large part to the presence of the I/O queues, which allow the processor internals to run at a clock rate completely independent of the external memory speed. Many other valuable lessons were learned from the implementation, and a number of new questions have been generated whose resolution is still under investigation. We feel that the benefits from implementing the processor have definitely been worth the effort.

1. Introduction

In the early 1980's a research project, known as the **PIPE** (**P**arallel **I**nstruction with **P**ipelined **E**xecution) project, was formed to investigate high performance computer architectures that would be well suited to VLSI implementation. One of the primary goals of the project was to devise architectural methods of minimizing the impact of off-chip memory accesses on processor performance. Crossing the boundary between the processor chip and external memory is one of the main impediments to achieving high performance in VLSI processors, and the problem is growing in severity. Because the on-chip clock frequency of a VLSI processor chip is increasing at a much higher rate than external memory speeds, it is projected that it will soon take 10-100 processor clock cycles to perform a single external memory reference.

The PIPE project members decided to define a processor architecture that would contain a number of unique features specifically designed to deal with this problem. For example, the architecture provides support for a *decoupled access/execute* mode of execution. (A decoupled architecture is one in which a program to be executed is divided into two or more instruction streams, and a number of processors cooperate in the execution of the task). The architecture also features I/O queues, visible to the programmer, that provide a means of buffering external memory accesses. In addition, the PIPE processor has a unique way of handling branches, as well as providing support for subroutine calls. Extensive analysis and simulation studies of the architecture [Youn85] have shown that a single PIPE processor, **without** running in a decoupled mode, is still capable of significant performance improvements over more conventional processors.

While the project members were encouraged by the results of these simulation studies, the feeling was that the architecture was enough different from existing architectures that unless the merit of our innovations could *demonstrated*, the simulation results would not be taken seriously. The project members felt that the best way to demonstrate the worth of the ideas incorporated within the PIPE architecture was to actually build a working PIPE processor chip. Therefore, the decision was made to implement the PIPE architecture.

Because of the number of unique features in PIPE and the importance of the problem the PIPE processor was designed to address, the implementation of the PIPE architecture should be of interest not only those exploring decoupled access/execute architectures, but to designers of conventional processors as well. The following sections of this paper will present a basic outline of the machine, followed by a description of the PIPE processor that was actually implemented, and an evaluation of some of the valuable lessons learned through the implementation.

2. The PIPE Machine

It is important to differentiate between the PIPE *machine* and the PIPE *processor*. The PIPE *machine* is composed of an intelligent memory and two identical processors, which provide support for a decoupled access and execute mode of execution [PlDa83, Smit82] (see Figure 1.).

As mentioned in the introduction, a decoupled architecture is one in which a program to be executed is divided into two or more instruction streams, and a number of processors cooperate in the execution of the program. In the PIPE machine, the two processors are referred to as the *Access* and the *Execute* processors. The Access processor is responsible for operand address computation, generating data requests both for itself and the Execute processor. The Execute processor functions essentially as a highly intelligent math co-processor, consuming the data sent to it by the Access processor and performing what the programmer would view as the heart of the required computations. These processors communicate with each other and with memory via hardware queues. The intent of separating a program this way is to allow the Access processor to get ahead of the Execute processor, thereby reducing or eliminating the delays due to accessing external memory. A much more detailed

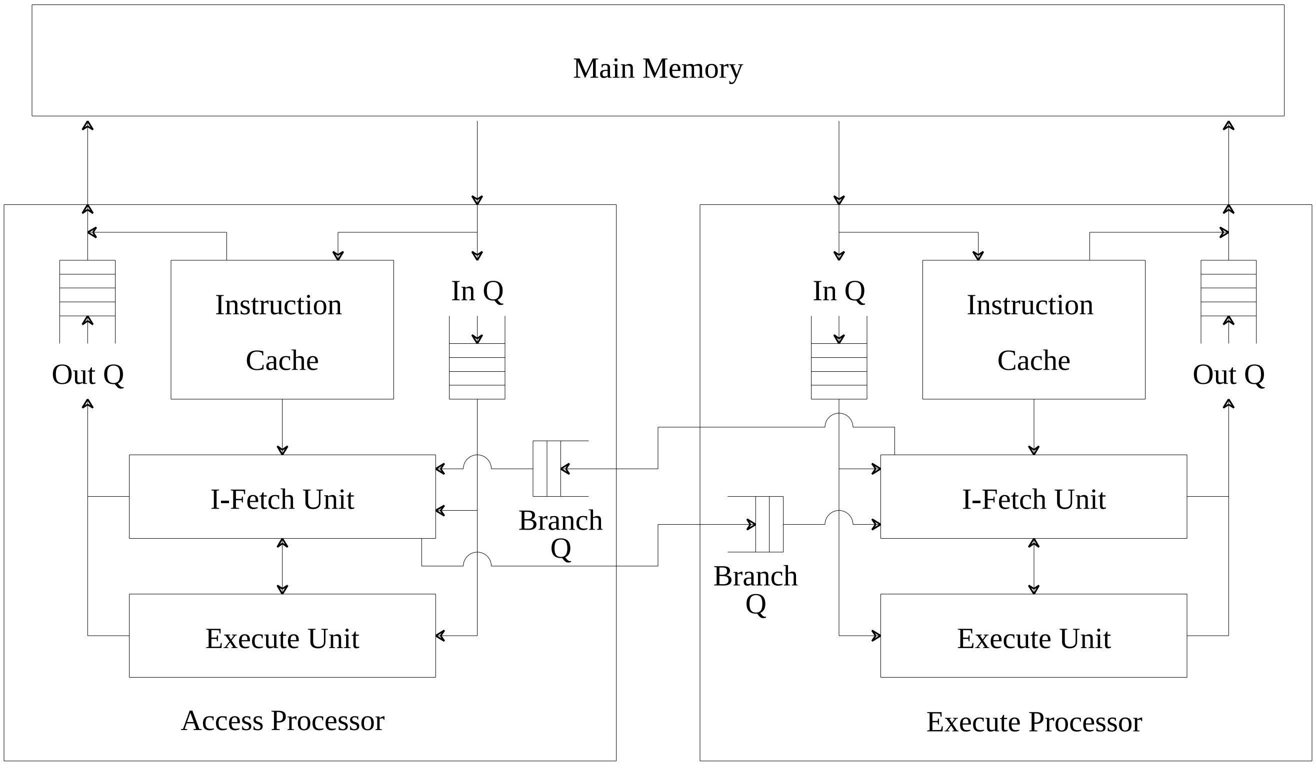

Figure 1. Block Diagram of the PIPE Machine.

description of the PIPE project is available in [Farr89, GHLP85].

3. The PIPE Processor

Since in the PIPE machine both the Access and the Execute processors are identical, throughout the rest of this paper these processors will be referred to as **PIPE** processors. A PIPE processor has much in common with other "RISC" processors. For example, it is a register to register type architecture, similar to Cray and CDC architectures. It is a 32-bit processor with a 32-bit wide internal bus, is 16-bit word-addressable and has separate input and output busses. The PIPE processor uses a barrel shifter for performing shifts and a two-stage ALU for performing adds and subtracts as well as logic functions. The PIPE processor employs an *elemental* instruction set, one whose resource requirements can be easily determined at instruction issue time. This instruction set supports a basic repertoire of 3 operand instructions; addition, subtraction, logical operations, and shifts in their various forms are all provided. Both a multiply and a divide instruction are also specified for future expansion.

Like many other VLSI processors, PIPE is pipelined to increase performance. PIPE has a five-stage pipeline, consisting of Instruction Fetch, Instruction Decode, Instruction Issue, ALU1/Logical, and ALU2. Unlike most other "RISC" processors, however, PIPE instructions come in 2 forms (see Figure 2). Instructions can be either a single parcel or two parcels long, where a parcel is a 16-bit quantity. The impact of having two different instruction sizes will become clear later in this paper. The PIPE instruction set is summarized in Table 1.

3.1. Unique Features

While a PIPE processor has much in common with most "RISC" processors, it also differs significantly in a number of respects. The following sections present a brief description of some of the unique features employed by PIPE to achieve a high level of performance.

3.1.1. Architectural Queues

The PIPE processor provides both input and output queues which act as buffers between external memory and the internal processing elements of the chip. This arrangement allows the on-chip clock rate to be determined solely by the timing delays through the processing elements that comprise the chip. The speed of the external memory has no effect on the processor's internal clock rate.

Since an PIPE processor is a register to register type machine, all memory interactions occur via Load and Store instructions. A Load instruction generates a memory address and places it on the tail of the Load Address Queue (LAQ). Items in the LAQ are then sent to the memory system, which sometime later responds with the requested data item. It is assumed that the compiler, employing well known optimization techniques, will place the load instructions as far ahead of the instruction requiring the data as possible.

When the data item is returned by memory it is not placed directly in a register; instead, it is put in the *load queue* (LDQ) which acts as a buffer for data. By making this queue explicit in the architectural definition, a program can have multiple outstanding memory requests without forcing the issue logic to reserve a path into the register file for each request. The head of this input queue is visible to the programmer as register R_7.

The writing of data items to memory occurs in a similar manner. A store address is generated and placed on the tail of the Store Address Queue (SAQ) using an instruction designed for this purpose. Data items are placed on the tail of the Store Data Queue (SDQ) by specifying register R_7 as the destination operand of most instructions. The items at the top of the SAQ and SDQ are then sent individually to memory, where they are recombined in the proper order.

3.1.2. Prepare to Branch

Branch instructions are notorious for causing performance degradation in heavily pipelined machines. This is due to the difficulty of keeping the pipeline full of useful instructions while the branch condition is being evaluated. This problem has been extensively studied, [DeLe87, McHe86, Smit81], and a number of methods for minimizing the impact of branches have been developed. The method used in the PIPE architecture is a generalized form of the delayed branch [HJBG82].

In the delayed branch scheme, there are a fixed number of *delay slots* following a branch that are filled with instructions that are guaranteed to execute. Ideally the number of delay slots should be as large as possible to guarantee that the branch condition will have been evaluated by the time the instructions complete, thus keeping the pipeline full. However, for many benchmark programs it is difficult to fill more than two delay slots with instructions that will perform useful work, forcing the compiler to place null operations into the slots it is unable to fill.

Our experience with numeric programs is slightly different. We have found that a compiler can easily generate code with an average of four instructions that can be unconditionally executed after a branch [YoGo84]. The PIPE processor uses an instruction called the Prepare-to-Branch (PBR) instruction which allows the compiler to *specify* the number of delay slots after a branch instruction (between 0 and 7). This capability means the compiler is never forced to place null operations after a branch, and if the compiler is able to fill the delay slots

Table 1. The PIPE Instruction Set

Instruction Class	Source1 = Register Source2 = Register	Source1 = Register Source2 = Immediate
Arithmetic	Add, Subtract	Add, Subtract
Logical	AND, OR, XOR	AND, OR, XOR
Shift	Shift Left, Shift Right, Arithmetic Shift Right, Rotate Left	Shift Left, Shift Right Arithmetic Shift Right Rotate Left
Move	Register Copy Register-Branch Register Copy	Load Register Load Branch Register
Change of Control	Jump Prepare to Branch $=, !=, >, \geq, <, \leq$	
Generate Load Address	$R_j+R_k \rightarrow$ (**LAQ** *or* **ALAQ**) $R_j+R_k \rightarrow R_i$ and (**LAQ** *or* **ALAQ**) $R_j \rightarrow$ (**LAQ** *or* **ALAQ**); $R_j+R_k \rightarrow R_i$	$R_j+Immd \rightarrow$ (**LAQ** *or* **ALAQ**) $R_j+Immd \rightarrow R_i$ and (**LAQ** *or* **ALAQ**) $R_j \rightarrow$ (**LAQ** *or* **ALAQ**); $R_j+Immd \rightarrow R_i$
Generate Store Address	$R_j+R_k \rightarrow$ (**SAQ** *or* **ASAQ**) $R_j+R_k \rightarrow R_i$ and (**SAQ** *or* **ASAQ**) $R_j \rightarrow$ (**SAQ** *or* **ASAQ**); $R_j+R_k \rightarrow R_i$	$R_j+Immd \rightarrow$ (**SAQ** *or* **ASAQ**) $R_j+Immd \rightarrow R_i$ and (**SAQ** *or* **ASAQ**) $R_j \rightarrow$ (**SAQ** *or* **ASAQ**); $R_j+Immd \rightarrow R_i$
Zero Operand	Halt, NOP, Switch Register File, Interrupt (Enable, Disable)	

Opcode	R_i	R_j	R_k
15 9	8 6	5 3	2 0

(a) 16-bit format

Opcode	R_i	R_j	OpExt	Immediate
15 9	8 6	5 3	2 0	31 16

(b) 32-bit format

Figure 2. PIPE Instruction Format.

with three or four instructions, it can guarantee that there will be no break in control flow. The PBR instruction allows the PIPE architecture to *always* perform at least as well as the more restrictive delayed branch scheme, and outperform it as pipeline depth increases.

3.1.3. Branch Register File

In order to support the PBR instruction, 8 Branch Registers were added to the architecture. These registers are not part of the set of general purpose registers, but rather a separate set of registers used to store branch target addresses. This allows the PBR instruction to be a single parcel instruction, and allows the compiler to load several branch target addresses at the beginning of a basic block. A separate instruction is provided in the instruction set to enable writing into these registers. Once written, the ALU has no way to access these registers. They can only be read by the instruction fetch logic.

3.1.4. Subroutine Call Support

The PIPE processor supports the passing of data between subroutines by logically partitioning the 16 registers in the register file into two sets of 8 registers. This approach was chosen in order to try to balance the amount of chip area devoted to registers with rapid subroutine call support. These two partitions are referred to as the foreground (FG) register file and the background (BG) register file. The operand fields in a typical instruction always reference the current FG register file. Instructions are also provided which permit movement of data between the foreground and the background register sets, to facilitate the passing of parameters between the calling and called routines. There is also a *switch register file* instruction that swaps the pointers to the FG and the BG register files, effectively swapping the contents of the FG and BG register files without any data movement actually taking place.

In order to perform a subroutine call, a Switch Register File instruction is executed, followed by an unconditional PBR instruction. By providing a BG register file, no saving away of the calling routine's active register file is needed if the called procedure is a leaf procedure (i.e. it does not make any procedure calls). Should the called routine make a subroutine call, it can schedule the saving away of the registers in the background register file (formerly the foreground register file) at its convenience. The only constraint is that this be done sometime before the called procedure performs a procedure call itself. This scheduling of register saves, coupled with the store data queue, make this aspect of register saves non-time-critical.

As explained earlier, register R_7 of the current FG set is the head of the LDQ and the tail of the SDQ. When referencing BG register R_7, however, an actual register is referenced and not the head or tail of a queue. This means there are 15 actual registers and one memory register available to the programmer at any one time.

3.1.5. Branch Queue

When the PIPE machine is operating in the decoupled mode described earlier, some method for synchronization between the two cooperating processors is necessary. The branch queue is used to synchronize branch points between the two processors and allow the Access processor to run ahead of the Execute processor. This synchronization is performed by using the Prepare to Branch on Queue instruction in conjunction with the Internal Prepare to Branch instructions. Executing an Internal Prepare to Branch instruction will place the result of the branch test in the Branch Queue, which the other processor accesses by executing a Prepare to Branch on Queue instruction.

4. The PIPE Implementation

Implementing the processor was a critical test of the PIPE architecture. Our goal was to demonstrate that our various innovative architectural features could be combined into a functioning whole. We wanted to show that the queues and the queuing discipline, interacting with the pipeline control logic, could be made to operate with a reasonable clock frequency and a minimum of chip area. In addition, we also had to insure that the hardware interlocks in the pipelined instruction unit were not prohibitively difficult to implement. The requirement of having an on-chip instruction cache also put a severe restriction on the amount of chip area available for control logic, registers, ALU, queues, etc. Finally, the existence of input and output queues made providing the ability to interrupt the PIPE processor, which was not specified in the original PIPE architecture, a real design challenge. A block diagram of the major components of the PIPE processor is shown in Figure 3.

The fact that PIPE is a university research project placed certain restrictions on available manpower and technology. Given these restrictions, we decided to implement and test *pieces* of the PIPE processor separately before attempting the implementation of the entire processor. We chose this approach because, at the time, no one at the University of Wisconsin had attempted a VLSI project of this magnitude. We believed (and our experience has confirmed) that it was critical to determine early in the implementation phase how accurate our tools were, and to ensure we fully understood the iterative process of designing, submitting, and testing a VLSI chip.

Three different pieces of the processor were submitted to the MOSIS fabrication facility at USC. The first piece was a chip that contained the register file, the two-stage ALU, the branch test logic, and the I/O bypass. The second piece of the processor to be implemented was a chip that contained a single input queue. The third and final piece of the PIPE processor to be implemented separately was the cache and instruction fetch logic. This chip contained over 15,000 transistors, and was by far the most complicated of the chips submitted prior to the final processor chip. Interestingly, it also came the closest to working exactly as specified, with only one small design error detected after extensive testing. This was enormously encouraging, and left us confident that we were capable of actually implementing the entire processor chip on a single die.

A detailed description of each component of the processor chip is beyond the scope of this paper. The next two sections provide a brief description of two of the more interesting elements of the processor.

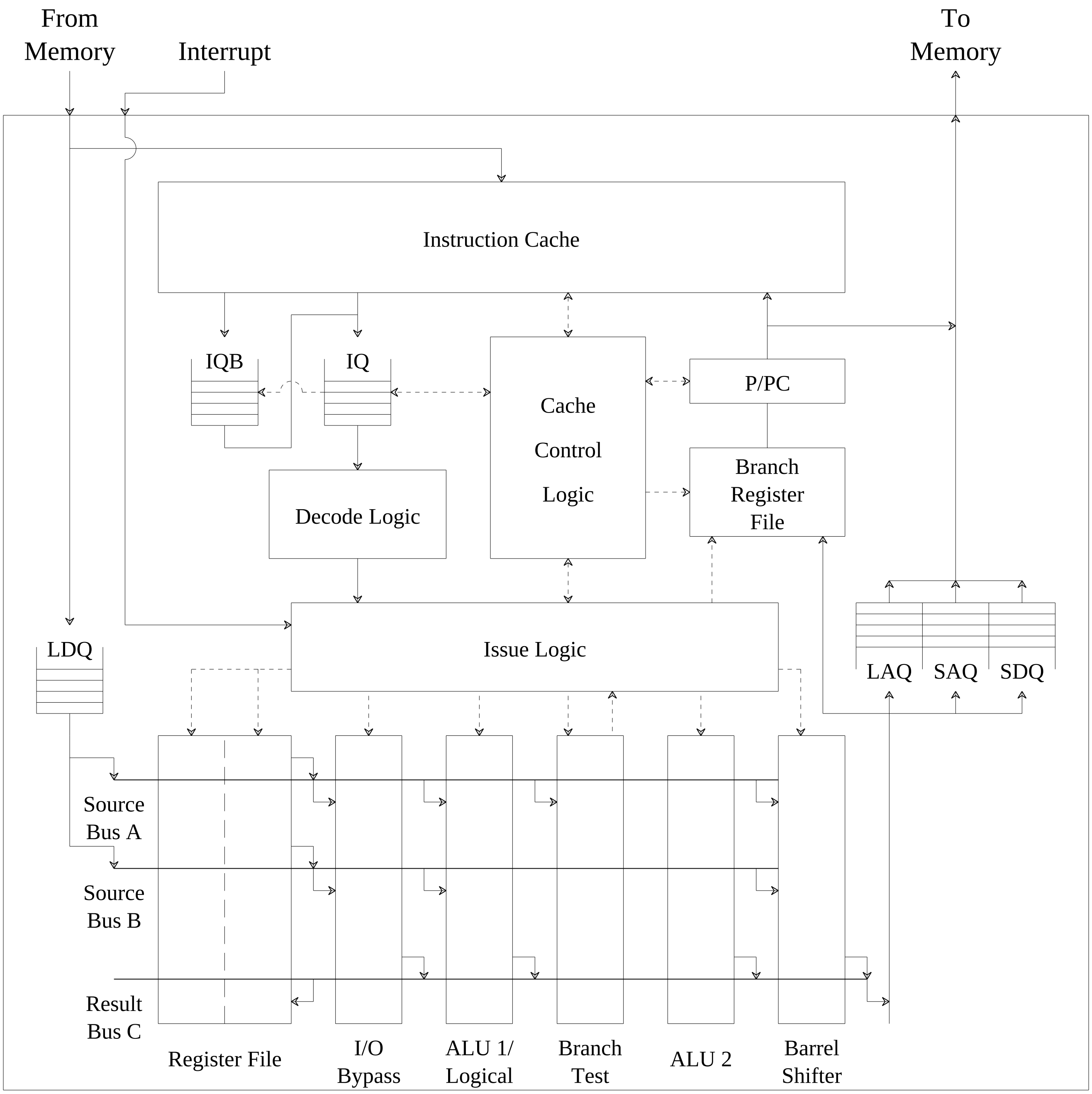

Figure 3. Block Diagram of the PIPE Processor.

4.1. The Instruction Fetch and Cache Logic

The structure of the PIPE instruction cache is intimately connected to the pipelined structure of the PIPE machine and, in particular, to the instruction fetch unit. The PIPE instruction cache is direct mapped and composed of sixteen 4-word lines for a total of 64 words, or 128 bytes. Although a larger cache would be preferable, hard limits are imposed by both available technology and the sharing of chip real estate with the rest of the PIPE processor. This instruction cache, while small, proved sufficient for our purposes. It allowed us to verify the design of the control logic, and several of the Lawrence Livermore Loops fit easily into a cache this large. In addition, special consideration is given to the cache design, in order to take advantage of the PBR instruction.

Directly below the cache cell array is a 64-bit register, the *instruction queue buffer* (IQB), which feeds the 64-bit *instruction queue* (IQ) register. The outputs of the IQ register pass through a MUX into a pair of 16-bit registers that form the *instruction register* (IR). Since instructions can be one or two parcels long, the IR consists of a pair of 16-bit registers, similar to the CIP and LIP registers of the CRAY1-S. The PIPE implementation differs in its ability to place 2-parcel instructions into the IR on every clock cycle. In the CRAY, only a single parcel can be placed in the CIP-LIP registers on each clock. The ability to repeatedly place two parcels into the IR on every clock arises from having an entire cache line held in the IQ and the use of a MUX that permits any sequential pair of parcels in the IQ to be moved into the IR during a single clock cycle.

The IQB acts as a prefetch buffer for the IQ. The cache control logic tries to keep the the IQB full by making a memory request should the needed line be missing from the cache. However, a request will only be made if the control logic can guarantee that the needed line contains at least one instruction that will definitely be executed. As soon as the entire contents of the IQ have been accessed, the IQB is moved into the IQ.

The goal of the PIPE instruction fetch logic is to keep the IQB and IQ full of valid instructions. If the control logic cannot decide the correct values to move into these registers, it guesses that the next sequential line will be needed. The effect of this guessing is limited to on-chip operations. Whenever a guess may force an off-chip operation, the control logic waits until it can ensure that some portion of the requested instructions will be executed. This restriction is mainly due to the limited bandwidth of a single chip processor and our desire to limit memory traffic.

Another ramification of this restriction is that PIPE does not *prefetch* instructions from off-chip in the classical sense. Classical prefetch strategies involve attempts to keep the cache full of useful data by anticipating what will most likely be needed and fetching it in advance of its use. Because these strategies *anticipate,* they are occasionally incorrect and fetch information that is not useful. The PIPE instruction fetch logic prefetches instructions from the on-chip cache, but only fetches from external memory items it *knows* it will need. Therefore, one has to be careful when talking about prefetching in PIPE and keep in mind what is actually occurring.

For straight line code, we check for branch instructions in the IQ and prefetch if none are detected. Due to the encoding of the PIPE instruction set, the existence of a PBR instruction is determined by a single bit of the opcode, making it easy to determine the presence of a branch instruction in the IQ. In the case of branches, the degree of achieved prefetching depends on the number of instructions to be unconditionally executed after the branch and the position of the PBR instruction in the queue. In either case, the degree of effective prefetching depends on the number of pipeline stages and the amount of blocking that occurs due to dependencies between instructions. The instruction fetch and cache logic have been explained in detail elsewhere [FaPl89, Farr89].

4.2. The Issue Logic

The issue logic is responsible for a number of tasks. It schedules the use of the result bus and blocks instruction issue if any hazards are detected. It is also responsible for keeping track of the number of elements in the queues, and tracking the number of outstanding memory requests. In addition, it is responsible for some of the synchronization between the instruction fetch and datapath logic.

The issue logic is also responsible for the execution of an interrupt. One criticism of using input and output queues is that they make it difficult to interrupt the processor. We felt it was important to demonstrate that while it does in fact make the processor somewhat more difficult to interrupt than a conventional machine, it is certainly not an insurmountable problem. The interrupt logic decides when an external interrupt can be recognized. The interrupt supported by PIPE is a non-realtime interrupt since there are a number of conditions that must be resolved before the interrupt can be recognized. For example, the instruction register must contain a valid instruction, and if there is a branch pending then the interrupt cannot be serviced until the branch is resolved. Furthermore, the instruction in the instruction register (IR) must not be a zero-operand instruction, and interrupt recognition must be enabled. All these conditions and certain others must be valid before an interrupt is allowed to occur.

The cache control logic calculates the conditions necessary for an interrupt to be recognized, and informs the issue logic when this occurs. At this point, the issue logic loads the interrupt PC register with the address of the valid instruction in the IR and then invalidates the instruction. The issue logic allows instructions waiting to issue and in the decode logic to finish, and then blocks subsequent instruction issue until there are no outstanding memory requests remaining. (An outstanding memory request is a request for data that has not yet produced an associated write into the input queue.)

As soon as there are no outstanding memory requests remaining, the issue logic places the count of the number of elements in the input queue into the top three bits of the interrupt PC register, and resumes normal issue procedures. The interrupt routine is then expected to read the interrupt PC register, extract the count of the number of data elements in the input queue, and execute the appropriate number of saves to allow the state of the machine to be restored at the end of the interrupt.

4.3. The PIPE Chip

Having successfully fabricated and tested most of the parts of the PIPE processor, we were now ready to incorporate the entire design onto a single piece of silicon. Technology had made significant advances since the original specification of the PIPE architecture, however. Circuit densities had gotten high enough (even in nMOS) that the decision was made to make several modifications to the original architectural specifications before implementing the entire processor on one chip. These modifications were made to make the processor as "real" as possible, given our design constraints.

When be began the implementation, neither our CAD tools nor the MOSIS fabrication facility were adept at dealing with CMOS designs. By this time, however, both had progressed to the point where CMOS was the technology of choice. Unfortunately, time constraints and the fact that this was a one-man project prevented the conversion of all that had been done from nMOS to CMOS. However, we felt that an nMOS implementation (while not ideal) would still accomplish our primary goal of demonstrating the validity of our architectural choices.

After the design was completed and the handshakes between all the functional units had been fully defined and tested individually, the chip as a whole was extensively simulated in an attempt to catch as many logic errors as possible. While the processor chips were being fabricated, a number of test programs were written to exercise various portions of the chip. Upon receipt of the fabricated processor chips, testing began and a number of new errors were detected. (None of them prevented further testing, however.) The errors were primarily in the cache - issue logic interface, and required adding No Operation instructions in certain places to prevent incorrect instruction execution. The fabricated chips have correctly executed a bubble sort program, a hash sort program, and a Booth's multiply program. The average maximum clock speed of the chips tested is 5.5 Megahertz, which corresponds to a peak execution rate of 5.5 MIPS. This is within 20% of the rate predicted by our simulations.

While this number is not as impressive as the performance numbers quoted by several other current processors, it is important to remember that PIPE was fabricated in a very restrictive technology. Therefore, PIPE should be compared to processors designed in the same restrictive technology. PIPE was fabricated in 3μm nMOS with a single level of metal. A more accurate comparison for PIPE would be the 3μm nMOS versions of the RISC-II [Henn84] or MIPS [HJPR83] chips; PIPE has a clock rate that is is 2-3 times higher than either of these machines.

Furthermore, SPICE simulations of the PIPE processor using 2μm nMOS parameters with low resist polysilicon interconnect indicate that if this less restrictive nMOS fabrication process were available to us, the PIPE performance rate rises to over 18 MIPS. The availability of a second level of metallization instead of low resist polysilicon would improve the performance even more. Unfortunately, MOSIS does not support these fabrication processes in nMOS.

5. Lessons Learned from the Implementation

Actually implementing the processor provided a unique opportunity to test the wisdom and effectiveness of a number of design choices. Analysis of the implemented processor indicates that a number of architectural choices made in the design process were good choices, and that some others were not.

5.1. Architectural Queues

One of the most general (and perhaps most important) result that emerges from an analysis of the implementation is that architectural queues do not prove to be difficult to implement. The potential of architectural queues has been shown again and again by simulation results demonstrating the performance improvements that can be achieved through the use of queues [Farr89, YoGo84, Youn85]. The building of the PIPE processor has demonstrated that from an implementation perspective, there is no reason *not* to equip a single-chip processor with I/O queues. By adding an interrupt to the processor, we were also able to demonstrate that the "machines with queues are hard to interrupt" objection to the use of queues can be overcome.

5.2. Issue Logic

Implementing the PIPE processor also demonstrated that the amount of logic necessary to resolve interlocks quickly and easily at issue time is not prohibitive. The benefits of using architectural queues also show up again. Their use actually simplifies the design of the issue logic somewhat, since there is only a single bit that the issue logic must test to determine when memory has responded with a requested item. The issue logic does not have to know anything about the speed of the external memory or its structure. This allows the processor design to move gracefully into different kinds of implementations and technologies, since no redesign is necessary to allow the processor to deal with the varieties of relative memory speed it will be facing.

5.3. Barrel Shifter

Due to the type of technology used to implement PIPE, the barrel shifter could not be made to functional as originally specified and still fit within the desired clock frequency. It could have been made to function correctly, even in the technology used, if it had been pipelined. However, we decided to stick to the original specifications of a single cycle shifter, and left out some of the functionality.

This question of pipelining the barrel shifter led us to a totally unexpected discovery, however. As presented in detail in [Farr89], simulation studies of how pipelining the barrel shifter would affect processor performance led us to the surprising conclusion that when a processor employs I/O queues, the difference between having all functional units of length 1 and all of length 2 causes less than a 2% degradation in performance over the benchmark programs studied. These results imply that there is no reason to spend an inordinate amount of time designing the ALU to do an add as fast as the the rest of the machine cycle. Making the ALU take two pipeline stages, and making all other functional units match it in length, has a minimal impact on performance.

5.4. The Cache Control Logic

The cache control circuitry is undoubtedly the most complex logic on the chip, and provides the PIPE processor with a sophisticated method for making the small on-chip instruction cache appear many times larger than it actually is. It was not clear at the beginning that we would be able to implement the complicated cache control logic and integrate it on chip with the rest of the processor circuitry. However, the actual implementation proves that the scheme is not prohibitively complicated, and that the scheme or a subset thereof can be a powerful tool for a designer.

5.5. The Branch Count

Supporting a branch count of zero in PIPE is not recommended for at least two reasons. First, because of some race conditions, supporting a zero branch count complicates the implementation tremendously. These are not implementational race conditions, but rather logical race conditions where getting the cache fetch logic to function correctly and at its peak requires that information about an upcoming branch instruction be made available to the cache control logic 1/2 clock cycle early.

Second, supporting a zero branch count limits the number of instructions that can appear after a branch instruction. With the current maximum branch count of seven, only three long instructions can be placed into the slots following a branch. This can can cause problems when it takes four clock cycles to get the branch result back from the ALU and begin fetching from the appropriate stream. By having a parcel count of eight, the processor can put four full long instructions after a branch, alleviating this problem.

5.6. Instruction Set Format

It is abundantly clear from the implementation that the decision to support an instruction set with two different instruction sizes and allow consecutive 2-parcel instruction issue has a tremendous impact on the design of the instruction fetch logic. This combination requires the PC to be able to increment by either one or two, and also means that a path from the IQB into the IR must be provided and supported. Providing these functions comprises a major portion of the size of the instruction fetch logic.

By removing the requirements for variable length PC change, the issue logic can be made significantly less complicated. There are two ways to accomplish this: Make all instructions the same size, or only allow a single parcel to move into the IR on each clock. This second approach is the one used in the CRAY-1, which has an instruction set very similar to PIPE's. By restricting the loading of the IR, the PC only needs to increment once per clock, and the desired reduction in complexity of the issue logic is achieved. Studies of the effect of this restriction on the performance of the processor indicate that very little performance improvement would occur if the CRAY instruction fetch logic was modified to allow two parcels to move into the IR each clock [WeSm84].

In order to analyze the affect such a restriction would have on the PIPE processor, a number of simulations were performed on the PIPE simulator in which the loading of the IR was restricted in this manner. The results of these simulations indicate that the ability to issue consecutive 2-parcel instructions is virtually mandatory in a single-chip processor with more than one instruction size. The main reason for this disparity between the CRAY and PIPE simulation results is that single chip processors have a much higher rate of instruction issue than the CRAY. This high issue rate increases the demand on the instruction fetch logic, exposing the delays due to waiting for the second parcel of an instruction.

Removing the requirement for variable length PC incrementation by making all instructions the same size is the approach taken by most current single-chip processors, and the fixed instruction size is generally 32 bits. While this may make the instruction fetch logic simpler, there is a down side. Having two different instruction sizes effectively increases the instruction bus width and improves the performance of the instruction fetch logic. This is especially true when the on-chip instruction cache size is limited; with a fixed 32 bit instruction size, the PIPE instruction cache could only hold 32 instructions, while with two different sizes it can hold up to 64.

The initial design of instruction fetch logic capable of supporting an instruction set with two different instruction sizes and moving more than one parcel into the IR on each clock was extremely difficult, and very challenging to debug and test. However, because of the potential performance improvement mentioned above, and the fact that once the design is completed and verified steps can be taken to remove it from the worst case path of the implementation, it is really not clear whether it is better to have a fixed 32-bit instruction size or an instruction set with two different sizes.

6. Summary and Conclusions

In this paper we have presented and discussed the implementation of the PIPE processor, a 32-bit pipelined single chip processor with a simplified load-store instruction set, a 5 stage pipeline, a two-cycle ALU, and a number of unique architectural features. Extensive simulations of the original design indicated that the architectural features referred to above provide significant performance improvements. However, it was felt that this combination of architectural features was sufficiently unique to justify an actual implementation of the processor, to investigate whether they would work as well in practice as in theory.

Before implementing the entire 32-bit processor, pieces of the processor, including the ALU, queue subsystem, and instruction were implemented separately. This approach permitted us to test and gain confidence in the design tools that we were using.

The total processor chip was submitted to MOSIS for fabrication in 3µm nMOS. The resulting chips were exercised by executing several programs on them (bubble sort, hash sort, Booth's multiply), and have a peak execution rate of 5.5 MIPS. While this number is not as impressive as the performance numbers quoted by several other existing processors, it is important to remember that PIPE was fabricated in a very restrictive technology. A better comparison for PIPE would be the 3µm nMOS versions of the RISC-II [Henn84] or MIPS [HJPR83] chips; PIPE has a clock rate that is is 2-3 times faster than either of these machines. SPICE simulations of the PIPE processor using 2µm nMOS parameters with low resist polysilicon interconnect indicate that with a less restrictive nMOS fabrication process the PIPE performance rate would be over 18 MIPS.

Implementing the processor architecture enabled us to evaluate several of our design decisions in much greater detail. We demonstrated that supporting architectural queues did not prove to be difficult and did not unduly complicate the instruction issue logic, and did free the internal clock rate of the processor from any external memory speed influences. In addition, we were able to implement a sophisticated instruction fetch strategy that allowed us to make highly efficient use of a relatively small on-chip instruction cache. Portions of this strategy could effectively be used by any conventional single-chip processor.

Primarily due to the technology that we were using, we discovered that it was difficult to build a fast shifter. The shifter's design raised questions regarding the amount of effort that should be placed into making even apparently simple operations take only one clock cycle.

We also uncovered a couple of problems with our architecture. The first relates to the Prepare to Branch (PBR) instruction. The PBR instruction can have a variable number of unconditionally instructions associated with it, including zero. Supporting the zero case made the implementation significantly more difficult. It would probably have been wiser to unconditionally execute between 1 and 8 instruction parcels instead of between 0 and 7 instruction parcels. Finally, the implications of having an instruction format that specified two different instruction lengths did not become apparent until the instruction fetch logic was being built. The question of the "correct" instruction format remains unresolved, however. Providing two different instruction lengths allows better utilization of the available off-chip bus bandwidth, and improves the effectiveness of a very small on-chip instruction cache.

We feel that we learned a number of things about our architecture that no amount of simulation could have revealed, and uncovered a number of interesting questions that we continue to pursue today. In our opinion, the benefits of going through the implementation process far outweighed the drawbacks.

7. Acknowledgements

This work was supported by National Science Foundation Grants DCR-8604224 and CCR-8706722. We would also like to extend a special thanks to Jim Goodman, Randy Katz, Jim Smith and Gary Craig for their invaluable assistance at various points along the way.

8. References

[DeLe87] J. A. DeRosa and H. M. Levy, ‘‘An Evaluation of Branch Architectures’’, *Proceedings of the Fourteenth Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania (June 1987), pp. 10-16.

[FaPl89] M. K. Farrens and A. R. Pleszkun, ‘‘Improving the Performance of Small On-Chip Instruction Caches’’, *Proceedings of the Sixteenth Annual International Symposium on Computer Architecture*, vol. 17, no. 3 (June 1989), pp. 234-241.

[Farr89] M. K. Farrens, *The Design and Analysis of a High Performance Single Chip Processor*, Ph.D. Thesis, Department of Electrical and Computer Engineering,, Madison, Wisconsin, (August 1989).

[GHLP85] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter and H. C. Young, ‘‘PIPE: a VLSI Decoupled Architecture’’, *Proceedings of the Twelveth Annual International Symposium on Computer Architecture*(June 1985), pp. 20-27.

[HJBG82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, ‘‘Hardware/Software Tradeoffs for Increased Performance’’, *Proceedings of the First International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California (March 1982), pp. 2-11.

[HJPR83] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen and T. Gross, ‘‘Design of a High Performance VLSI Processor’’, *Proceedings of the Third Caltech Conference on VLSI*(March 1983), pp. 2-11.

[Henn84] J. L. Hennessy, ‘‘VLSI Processor Architecture’’, *IEEE Transactions on Computers,* vol. C-33, no. 12 (December 1984), pp. 1221-1246.

[McHe86] S. McFarling and J. Hennessy, ‘‘Reducing the Cost of Branches’’, *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture,* Tokyo, Japan (June 1986), pp. 396-403.

[PlDa83] A. R. Pleszkun and E. Davidson,, ‘‘A Structured Memory Access Architecture’’, *International Conference on Parallel Processing*, Bellaire, MI (1983).

[Smit81] J. E. Smith, ‘‘A Study of Branch Prediction Strategies’’, *Proceedings of the Eighth Annual International Symposium on Computer Architecture*, Minneapolis, Minnesota (May 1981), pp. 135-148.

[Smit82] J. E. Smith, ‘‘Decoupled Access/Execute Computer Architectures’’, *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, Austin, Texas (April 26-29, 1982), pp. 112-119.

[WeSm84] S. Weiss and J. E. Smith, ‘‘Instruction Issue Logic for Pipelined Supercomputers’’, *Proceedings of the Eleventh Annual Symposium on Computer Architecture,* vol. 12, no. 3 (June 1984), pp. 110-118.

[YoGo84] H. C. Young and J. R. Goodman, ‘‘A Simulation Study of Architectural Data Queues and Prepare-to-Branch Instruction’’, *Proceedings of the IEEE INT Conference on Computer Design: VLSI in Computers,* Port Chester, New York (October 1984), pp. 544-549.

[Youn85] H. C. Young, *Evaluation of a Decoupled Computer Architecture and the Design of a Vector Extension*, Ph.D Thesis, University of Wisconsin-Madison, (July 1985).