PIPE: A HIGH PERFORMANCE VLSI ARCHITECTURE

James E. Smith
Andrew R. Pleszkun
Randy H. Katz
James R. Goodman

Computer Sciences Technical Report #512

September, 1983

PIPE: A High Performance VLSI Architecture

James E. Smith, Andrew R. Pleszkun, Randy H. Katz, James R. Goodman

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

Abstract

The PIPE architecture (Parallel Instructions and Pipelined Execution) is proposed as a research vehicle for studying high performance VLSI architectures and organizations. Principal features are: 1) it is pipelined, 2) it is capable of a decoupled mode of operation where two processors cooperate in executing the same task and communicate via hardware queues, 3) it has an instruction cache, and 4) it has a memory interface that allows overlap of memory transactions. This paper describes the proposed architecture and its planned implementation. Included are a discussion of design considerations for a pipelined VLSI architecture, and a description of the particular architecture we have chosen to study.

1. Introduction

The PIPE architecture (Parallel Instructions and Pipelined Execution) is a VLSI-oriented research project underway at the University of Wisconsin. This paper describes the architecture, the motivation behind it, and some of the features of its planned implementation.

Although we draw a sharp distinction between architecture and organization, a major part of our research is the study of their interaction. The *architecture* is a functional description, most notably the instruction set, and the *organization* is the hardware implementation.

The two main architectural features of PIPE are: 1) it is pipelined, and 2) it can be decoupled [Smit82, Ples82, CoSt81], where two identical processors execute parallel instruction streams, communicating via architectural queues to execute a single process.

We feel that pipelining and VLSI are a good match. A pipelined computer organization requires more logic than a serial one, but it does not necessarily require more interface pins. Furthermore, the additional logic required for pipelining contributes significantly to system performance. Our research is not to demonstrate the viability of a pipelined organization in a VLSI environment. Rather, we will explore architectures that permit pipelined VLSI system organizations that are simple, efficient, and well-structured.

Decoupled architectures are a recent development. They support multiprocessing at a low level, while not precluding multiprocessing the process level. A decoupled architecture processor complements traditional multiprocessing forms. Two processors of a decoupled architecture might be used in an access/execute (AE) mode of operation. The *access* processor (AP) calculates all memory addresses and makes all memory references for both processors. Any data intended for use in the *execute* processor (EP) is placed in a hardware queue held by the execute processor. It performs the data processing calculations, e.g. arithmetic operations on data. When it needs memory data, the EP simply goes to its input queue and takes the first item. Computed results to be stored in memory are placed in a hardware output queue. When the AP has computed a store address, it and the data in the EP's output queue are paired and sent to memory. Conditional branches are also coordinated via queues, and in many cases the access processor runs well ahead of the execute processor. This results in more efficient use of memory, significantly less effective memory delay, and higher performance [Smit82]. The two decoupled processors can be identical. Thus, only one processor need be designed. A wider variety of operational modes besides AE mode mentioned above can be supported.

Two main implementation features are: 1) an instruction cache, and 2) a processor/memory communication interface permitting overlap among consecutive

memory references.

The instruction cache provides a high performance payoff for hardware cost. Although we feel that an on-chip data cache requires too much real estate and design complexity for our system, a pure instruction cache provides a good balance of chip area requirements, design difficulty, and performance.

The second feature is memory reference overlap. Rather than supplying an address and waiting for a memory transaction to complete before initiating the next, we support a CPU/memory communication interface that allows simultaneous memory transactions. This is widely used in pipelined mainframe computers. The interface will still support memory interleaving.

1.1. Design Considerations

There are three considerations that guide the direction of this research. These are the so-called von Neumann bottleneck, the lesser-known Flynn bottleneck, and efficient compiler code generation.

The von Neumann bottleneck [Back78] is the restriction of the communication path between the CPU and main memory to one word per clock cycle. The bottleneck is not that restrictive, since high performance computers have provisions for transferring several words per cycle. Nevertheless, it characterizes the VLSI environment. The features discussed earlier are directed at the von Neumann bottleneck. An instruction cache cuts down on the number of words passed through the bottleneck, and overlapped memory references use the available pins as efficiently as possible. The von Neumann bottleneck also provides part of the motivation for a decoupled architecture. The queues of a decoupled architecture provide buffering in the memory access path so that temporary congestion at the von Neumann bottleneck can be smoothed out.

The severity of the von Neumann bottleneck can be reduced if a high instruction cache ratio is achieved. In high performance systems using caches, the major obstacle to performance is not the von Neumann bottleneck; it is the Flynn bottleneck. Flynn [Flyn66] observes that the real performance constraint is that in the instruction fetch/decode path there is some bottleneck through which instructions pass at the maximum rate of one per clock period. This observation is supported by all the commercially available high performance computer systems.

Decoupled architectures, with their two parallel instruction streams, reduce the constraints of the Flynn bottleneck. Because there are two program counters, decoders, issue units, etc., the capacity of the bottleneck has effectively been doubled. Our load/store instruction set is aimed at raising the relative percentage of memory loads and stores in the instruction mix, with the goal of fully using both bottlenecks simultaneously, yielding a balanced system. This will be made more apparent in later sections.

Software issues are a major consideration in any architectural project, but a pipelined system increases its importance. The overlapped operation provided by pipelining adds another dimension to code optimization: code scheduling. Instructions must be ordered to maximize the amount of overlap. Good scheduling provides significant speedup, but it often takes years between the development of a pipelined architecture and the development of a compiler that generates near optimum code. A decoupled architecture effectively permits some code scheduling to be performed dynamically at runtime, thereby reducing the importance of static compile-time scheduling.

2. Description of the Architecture

2.1. Single Processor Architecture

The decoupled architecture we envision requires two identical co-processors that communicate via hardware queues. However, a single processor could be run by itself, and would provide reasonably high performance. Hence, we begin by defining a single processor architecture. Section 3 describes the few extensions necessary to support decoupled operation.

2.2. Pipelining and Architecture

The success of the Seymour Cray architectures, CDC6600 [Thor70], CDC7600 [Bons69], and CRAY-1 [Russ78], have clearly demonstrated that a computer's architecture is a critical factor in determining efficiency of a high-speed implementation. Although our design environment and technology are considerably different, many of the architectural concepts are still valid. A guiding principle of the Cray architectures is simplicity. Hence, we use a simple instruction set that is in some ways similar to RISC-I [PaSe81], although the motivation and general characteristics are generally closer to those of the CDC6600.

We agree with many of the arguments put forward for simple instruction sets [PaSe81, Radi82]. However, there is another, perhaps stronger, argument when a pipelined implementation is used. Simple instructions have fewer dependencies with other instructions and allow more nearly optimal code scheduling and sequencing. To quote J. Thornton, the CDC6600 design involves the "use of 'micro' instructions which can be arranged with flexibility to provide overlap" [Thor70]. Of course, the use of simple instructions to optimize scheduling can be carried too far: if there are too many instructions, the Flynn bottleneck problem can be made more severe. Hence, a balance must be struck.

Another part of our design philosophy is that pipeline interlocks should be in hardware. This is in contrast to the MIPS project [HJBG82], another pipelined VLSI architecture, where the interlocks are in software. We feel that the MIPS architects overestimate the difficulty of hardware interlocks and underestimate the difficulties of code generation for a pipelined architecture. By using an architecture designed specifically for pipelining, essentially all interlocks involve use of the registers, and can be resolved at one point in the instruction pipeline with relatively simple logic. This is done in the CRAY-1 where interlock resolution is primarily done at only one pipeline stage. In contrast, an architecture not designed for pipelining, e.g. IBM 360/370, can lead to complex hardware interlock problems when a pipelined implementation is used. If software interlocks are used, many features that are ordinarily implementation-dependent become part of the architecture -- for example pipeline lengths. The code generator must be aware of pipeline lengths to provide correct interlocking. A new implementation forces a change in the code generator, and code for the older implementation must be recompiled or rewritten.

An architecture incorporating interlocks in software takes on the appearance of an array processor, and generating good code for an array processor is notoriously difficult [KaCo81]. While it is undoubtedly possible to generate code with software interlocks, we feel that it is not necessary since hardware interlocks in a well-designed architecture are straightforward to implement.

2.3. General Information

The PIPE architecture is still evolving. What follows is an overview of the current architecture with a discussion of the major instructions. This should give a good idea of the approach we are using, though the details of the architecture will no doubt change as our research progresses.

The PIPE word length is currently 32 bits. While the long word length is desirable, particularly for numerical applications, the fabrication technology available to us severely limits the complexity of the chip we can build. We therefore are seriously considering reducing the word length to 16 bits, which would result in minor changes in the instruction set.

There are only two instruction formats, one that is 32 bits; the other is 16 bits. The two formats are shown in Fig. 1.

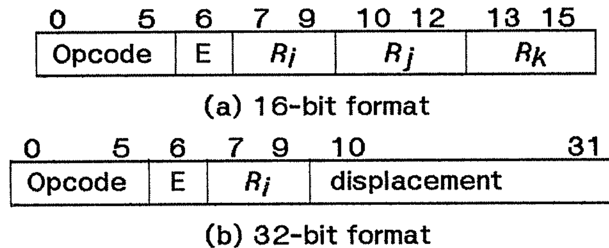

Fig. 1. Instruction formats.

In the 16-bit format, there is a 7-bit opcode, including a branch exit or "E" bit, to be explained in Section 2.6. There are three 3-bit operand fields. These designate either one of 7 general purpose registers or a hardware queue head or tail (see Section 2.4). The three operands are typically two sources and one destination. Occasionally, one of the fields is not used.

In the 32-bit format, there is a 7-bit opcode as before, a single 3-bit operand field, and a 22-bit signed displacement field to hold immediate data and address offsets.

2.4. Load/Store Instructions

All memory load and store data pass through hardware queues. The queues are the LOAD DATA QUEUE (LDQ), the STORE DATA QUEUE (SDQ), and the STORE ADDRESS QUEUE (SAQ). The LDQ holds data after it has been fetched from memory. The SAQ and SDQ hold store addresses and data that are directed toward memory. If the SAQ and SDQ are both nonempty, the elements at their respective heads are paired and sent to memory as a store operation. At the machine level, queue heads and tails often appear to be like registers. For mnemonic purposes, we use the queue names in instructions, but in the machine code, register fields containing a 7 value designate the queue, only one of which may be specified in a given field; the context unambiguously determines which.

Queues are essential to a decoupled mode of operation, to be explained later, where two processors cooperate on executing the same process. They do have some advantages in a single pipelined processor, however. First, they allow use of a relatively small number of general purpose registers even when optimal code scheduling is used. Any data loaded and used only once is never allocated a register of its own; it is loaded into the LDQ and used directly from there. This may reduce the number of registers necessary to give well-scheduled code; we are undertaking a study to see if this is true.

The second advantage is subtle, but important: at issue time loads need not reserve an input path into the register file. Ordinarily at instruction issue time this path is reserved for use during the clock period when the instruction completes. Load instructions pose a problem, however. They may have unpredictable completion times due to variations in memory access delays. Hence, they either need their own path into the register file, which requires the capability to do two writes simultaneously, or the result path must be reserved later in the execution of the load instruction, when the exact time of the data's arrival from memory can be determined. By

using the LDQ, we naturally have a dedicated path for load data; it never goes into the register file. This means that we can resolve all interlocks during the one-clock-period instruction-issue time.

The instruction LDQ ← (Ri,disp) takes the content of Ri added to disp to generate an effective memory address. For all the load and store instructions, a 0 in the Ri field is not interpreted as R0, but as a literal 0. The memory data is loaded and placed in the tail of the LDQ. When the data works its way up to the head of the LDQ, a subsequent instruction can refer to it as a source operand, using LDQ (a 7 value in the source field) as a source designator. The implementation must enforce a discipline on memory fetch requests so that they are returned from memory and placed in the LDQ in program order.

A 32-bit store instruction is written as: SAQ ← (Ri,disp). Here, the effective address is formed as in a load, and is placed in the SAQ. Strictly speaking, this instruction only generates a store address; the store does not actually take place until the data is placed in the SDQ by some other instruction, and both the address and data work their way up to the heads of their respective queues.

There is a set of autoincrementing loads and stores with the 32-bit format. Autoincrementing loads and stores might appear to be in conflict with the philosophy of simple instructions. Because the issue conditions are the same regardless of whether or not autoincrementing is used, no instruction scheduling flexibility is lost. In addition, autoincrementing instructions reduce the number of instructions to pass through the Flynn bottleneck. LDQ ← (Ri,disp)+ is a post-incrementing load. It uses the content of Ri as the effective address of the load, and then increments Ri by the amount given in the displacement field. Similarly, LDQ ← +(Ri,disp) does a pre-incrementing load where the effective address is the content of Ri added to disp, and Ri is loaded with this sum.

There are also 16-bit load and store instructions. These are of the form LDQ ← (Ri,Rj) and SAQ ← (Ri,Rj), respectively. With these instructions, the Rk field is not used. The effective address is formed by adding Ri and Rj. Note that either the Ri or Rj fields could be a 7, denoting the head of the LDQ; such might be the case if indirect addressing through memory is being used. There are also autoincrementing forms of the 16-bit loads and stores. LDQ ← +(Ri,Rj) does a pre-increment, and LDQ ← (Ri,Rj)+ does a post-increment. There are similar autoincrementing stores, SAQ ← +(Ri,Rj), and SAQ ← (Ri,Rj)+.

Closely related to loads is the "enter" instruction that places an immediate value in a register. This differs from a load immediate instruction in that the data is not provided by a queue, as with other load instructions. It is in the 32-bit format, and is of the form: Ri ← disp.

2.5. Arithmetic and Logical Instructions

The arithmetic and logical instructions are all "register-to-register" in the three operand, 16-bit format. The LDQ head can be used as either source operand, and the SDQ tail can be used as the destination. As before, a 7 value designates the appropriate queue. For example, SDQ ← R3+LDQ causes R3 to be added to the head of the LDQ, with the sum being placed in the SDQ. The PIPE architecture contains a simple set of arithmetic and logical operations. As suggested above, the add instruction is of the form Ri ← Rj+Rk, and "subtract" (probably two kinds), "or", "and", "exclusive or", "not" are all similar; "not" does not use the Rk field. There are also 32-bit immediate forms of the arithmetic and logical instructions. For example, Ri ← Ri+disp is an add immediate. In PIPE, a typical shift instruction is Ri << Rj|Rk which performs a left shift of Rj by count in Rk into Ri.

2.6. Branch Instructions

In PIPE, no condition codes are used. In a pipelined architecture with condition codes, one must be careful to use the most recent condition code values when executing a conditional branch. Since many instructions may be in process at the same time, careful bookkeeping is required so that the condition codes are set and inspected at the correct times. (See [AnST67], for example.) In addition, condition codes typically force the instruction that sets the condition codes to immediately precede the branch instruction. This results in unhidden delay between the issue of the two instructions.

We also separate evaluation of a branch condition and computation of effective target address from the actual transfer of control. In particular, we use a conditional "prepare to branch" (PBR) instruction followed by some later instruction with its opcode "branch exit" or E∧bit set. If the branch condition is satisfied the jump to the branch target address takes place after the execution of the instruction with its E∧bit set. This separation of the actual exit from the rest of the instruction gives the instruction fetch logic advance notice of a branch. This allows the instruction fetch logic to provide a smoother flow of instructions.

This prepare-to-branch/exit method was proposed in [Scho71], and is a more general solution to the problem than is used in [PaSe81, Radi82, HJBG82], where the branch exit takes place following the instruction after the conditional branch instruction. The general method allows more flexibility in the implementation and in code scheduling. Note that the PBR opcodes also have E∧bits. If set, a PBR behaves like a conventional conditional branch instruction.

The PBR instructions are in the 32-bit format. The opcode specifies the condition to be tested, Ri specifies the register to be tested, and the displacement is a program-counter-relative branch target address. The conditional prepare-to-branch instructions are: IPBR(Ri>0) → disp, IPBR(Ri<0) → disp, IPBR(Ri=0) → disp, etc... The "I" prefix character stands for "internal", its meaning will be described in Section 3.2.

There is also an unconditional PBR instruction that branches to the location specified by adding the content of Ri to the displacement: IPBR → (Ri,disp).

2.7. Call/Return Instructions

The speed of call/return is in important factor in determining overall system performance, particularly with modern high level languages and programming techniques. We have taken some special steps to reduce the time consumed by calls and returns.

First, we use the same method outlined in the previous section for transferring control: there are prepare-to-call and prepare-to-return instructions that operate in conjunction with the E∧bits. Again, this permits the instruction fetch logic to get a head start on fetching instructions from the new stream.

Second, to facilitate quick saving and restoring of registers, we have two register files, a foreground file, and a background file. Normal execution is always out of the foreground file. A call or return causes the two files to be switched, and causes the Program Counter to be saved in or restored from background register R7. R7 is ordinarily used to designate a queue, and is, therefore, not used in the backup file for register data. The LDQ should be empty just prior to a call or return.

There are two instructions that allow foreground and background registers to be copied: Ri ← BRj and BRi ← Rj. These use the 16-bit format with field Rk unused; Ri and Rj can be replaced with SDQ and LDQ, respectively, as before.

A procedure stack is maintained by software, with the registers and Program Counter for the top stack frame being held in the background register file. If a procedure is about to make a call, it first copies the current background registers onto the stack in memory. This frees up the background registers so they can be used as

foreground registers after the call. Saving the background registers consists of a number of SDQ ← BRj instructions, putting the background registers in the SDQ, and an equal number of store address instructions causing them to be stored away. These instructions can be scheduled in with other instructions when such a call is anticipated. Similarly, after a return, if another return to a higher level is anticipated, then the background registers must be read from memory and copied into the background registers.

The principal advantage of using this scheme is that the registers and program counter only need be copied to memory by a called procedure that plans to call a second procedure before it returns, or by a calling procedure that plans a return to a higher level before doing any more calls. Even then, only those registers which will be required by the called procedure need be stored. This means that "leaf" procedures, (those that call no others) do not require any saving or restoring of registers. This method is similar to, but not as complete as, the method used in RISC-I [PaSe81]. We choose this method because 1) We plan to use an instruction cache, instead of the large number of registers in the RISC scheme, 2) It probably catches a sizable percentage of all calls and returns. We are planning a study to empirically determine percentages that can be expected. 3) When background registers do need to be saved or restored, we can schedule the code that performs the transfer among the other instructions, using issue time slots that might not otherwise be used. Hence, in conjunction with "prepare to call" and "prepare to return" we can smoothly phase in the call and return functions.

3. Decoupled Architecture

We plan to connect two of the processors just described so that they can cooperate in executing the same process. The two processors communicate data via the load and store queues and control information via branch queues, to be described later. Fig. 2 is a block diagram of such a two-processor system. In order to facilitate communication through the queues, the load/store and conditional branch instruction sets need to be expanded.

3.1. Load/Store Instructions

Both processors have LDQ's and SDQ's as described in Section 2.4. We plan to let one processor initiate a load or store in behalf of the other. That is, the instruction ALDQ ← (Ri,disp) (ALDQ stands for Alternate LDQ) causes data to be loaded from memory and to be placed in the other processor's load queue. A similar thing happens for ALDQ ← (Ri,Rj) and the autoincrement cases.

A processor can also generate a store address for the other processor via the instruction ASAQ ← Ri,disp. The store address, when it reaches the head of the store address queue, must be matched with data at the head of the other processor's SDQ before they both go to memory. One or the other must wait until both are at the heads of their respective queues.

3.2. Branch Instructions

The branch instructions described in Section 2.6 are referred to as "internal branches" because they only affect instruction fetching in the processor in which they are executed.

We add a set of "external branches" that behave just like internal branches, but in addition they send a branch outcome to the other processor via a branch queue. Both processors also have a "prepare to branch from queue" instruction: PBR (Q) → disp. This instruction simply checks the head of the branch queue from the other processor to see if the branch should be taken. By using these paired branches, an external branch in one processor and a branch from queue in the other,

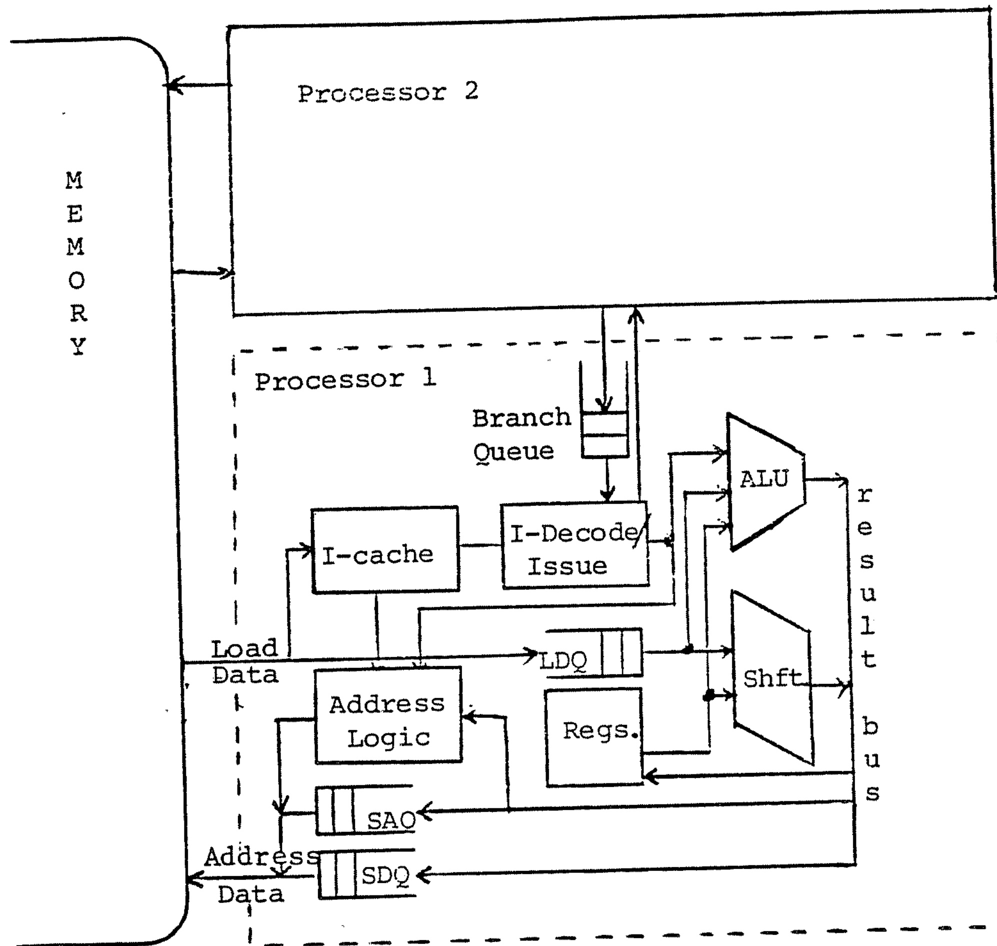

Fig. 2. Block diagram of a two processor organization.

the two processors can be made to "track" each other through sequences of code containing conditional branches.

3.3. Examples

To clarify the operation of the decoupled machine and the way in which the branch instruction and data queues are used, consider the following small section of code:

```
biga := 0;
smalla := 0;

for i := 1 to n do begin
  if a[i] > biga then biga := a[i];
  if a[i] < smalla then smalla := a[i];
  end;
```

This program accesses the elements of a vector with n elements, finding the largest and smallest elements of the vector.

To accent the use of the branch and data queues the program will be translated into two different assembly language programs. The first version of the program is designed to be run on a single processor while the second version is meant to run with two processors, one the execute processor and the other the access processor. Using the mnemonics and operations described in the previous sections, the single processor assembly language version is as shown in Fig. 3. Notice that the incrementing and comparison of the index is not performed immediately before the branch

```
             /*
             /* Single Processor Assembly Language Program
             /*
 1)          R1 ← 0                       /* biga
 2)          R2 ← 0                       /* smalla
 3)          R3 ← a                       /* index i
 4)          R0 ← n+a                     /* end value

 5)   L1:    LDQ ← (R3,1)+                /* fetch a[i] and incr i
 6)          R4 ← LDQ                     /* store a[i] in a reg
 7)          R5 ← R4 - R1                 /* cmp a[i] and biga
 8)          IPBR (R5 ≤ 0) → L2           /* test condition
 9)          R6 ← R2 - R4;exit            /* cmp a[i] and smalla
                                          /* branch on test
                                          /* of a[i] > biga
10)          R1 ← R4                      /* reset biga

11)   L2:    IPBR (R6 ≤ 0) → L3           /* test condition
12)          R5 ← R3 - R0; exit           /* cmp i with end value;
                                          /* branch on test
                                          /* of a[i] < smalla
13)          R2 ← R4                      /* reset smalla

14)   L3:    IPBR (R5 < 0) → L1; exit     /* test condition and
                                          /* branch if i < end value
15)          SDQ ← R1                     /* save biga
16)          SDQ ← R2                     /* save smalla
17)          SAQ ← 0,biga                 /* gen addr for biga
18)          SAQ ← 0,smalla               /* gen addr for smalla
```

Fig. 3. Single Processor Assembly Language Program.

```
              /*
              /* Execute processor program
              /*
E1)           R1 ← 0                       /* biga
E2)           R2 ← 0                       /* smalla
E3)    L1:    R4 ← LDQ                     /* store a[i] in a reg
E4)           R5 ← R4 - R1                 /* cmp a[i] and biga
E5)           IPBR (R5 ≤ 0) → L2           /* test condition
E6)           R6 ← R2 - R4; exit           /* cmp a[i] and smalla
                                           /* branch on test
                                           /* of a[i] > biga
E7)           R1 ← R4                      /* reset biga
E8)    L2:    IPBR (R6 ≤ 0) → L3; exit     /* test condition
                                           /* branch on test
                                           /* of a[i] < smalla
E9)           R2 ← R4                      /* reset smalla
E10)          PBRQ L1; exit
E11)          SDQ ← R1                     /* save biga
E12)          SDQ ← R2                     /* save smalla
```

Fig. 4. Execute Processor Program.

(line 14). Instead these operation are scheduled throughout the rest of the program to improve pipelined performance by reducing dependencies among successive instructions. In line 14 the IPBR instruction is used as a conventional branch instruction. The two other times IPBR is used (lines 8 and 11), the usage is less conventional. That is, although the condition is set with the IPBR instruction, the branch is not taken until the following instruction (lines 9 and 12). Also, notice the use of the data queue for input and output in lines 5, 6, and 15 through 16.

To run this same program on two processors, the addressing portions of the program are separated from the computation portions. That is, the index-related operations are moved to the access processor. In these programs (Figs. 4 and 5), address calculation operations have been removed from the code for the execute processor and placed in the access processor code. In the execute processor, branches are now taken when the branch conditions are evaluated. Also, since the loop control variable (index i) is now located in the access processor, the execute processor

```
              /*
              /* Access processor program
              /*
A1)           R1 ← a                       /* index i
A2)           R2 ← n+a                     /* end value
A3)    L1:    ALDQ ← (R1,1)+               /* fetch a[i] and incr i
A4)           R3 ← R1 - R2                 /* cmp i with end value
A5)    L3:    PBR (R3 < 0) → L1; exit      /* test condition and
                                           /*branch if i ≤ end value
A6)           ASAQ ← 0,biga                /* gen addr for biga
A7)           ASAQ ← 0,smalla              /* gen addr for smalla
```

Fig. 5. Access Processor Program.

checks for the end of the loop by accessing the branch queue (line E10).

4. System Organization

4.1. CPU/memory interface

High performance systems usually depend on sophisticated pipelined memory subsystems to provide a stream of data to meet the processor's demands. Each of the processors will have its own instruction cache, so the backing storage system normally will not supply any instructions. However, it will be required to supply operands at a substantial rate -- approaching one per clock period. In the worst case, when both processors fetch or store data independently, it could be worse than that. The partitioning of the work between the processors, however, leads us to believe that initiating service on memory requests at a maximum rate of one per cycle should not result in any degradation under normal circumstances.

We plan to use one bus for address and memory write data, and a second for memory read data. Sending the write data out over the same lines as the addresses further enhances performance by allowing all lines to be unidirectional. This should not cause any conflict in AE mode, since the address and the data originate in separate processors. In the case where both originate from the same processor, it takes two clocks to initiate a write instruction.

4.2. The Instruction Cache

We intend to implement an instruction cache for each processor. While higher performance could possibly be achieved either by including data in the cache or by maintaining a separate data cache, we decided against it for the following reasons:

(1) A data cache provides lower performance improvements. Instruction fetches constitute a large portion of all memory reads. Instructions also exhibit temporal and spatial locality in a far more consistent manner than data.

(2) An instruction cache is substantially easier to implement than a data cache because writing into the instruction stream (and instruction cache) does not occur. This assumes, of course, that self-modifying code is disallowed, a restriction consistent with modern programming practice.

(3) In the PIPE architecture, operands for a load queue must arrive in order. While supplying some operands from an on-chip cache can be taken into account in maintaining that order, the bookkeeping for this procedure is complicated considerably.

(4) It is not compatible with VLSI, where communication is particularly expensive. The problem of maintaining consistency between two cache memories is well-understood, and widely implemented [IBM74, IBM76, CeFe78] However, the penalty that is paid is heavy communication between the cache controllers. Under these conditions, the cache is much less effective.

We have not entirely ruled out the possibility of a data cache and intend to study it, but our current belief is that it is not cost effective.

4.3. VLSI Implementation

The PIPE architecture is well suited for VLSI implementation. While control logic for a pipelined machine is more complicated than that for a serial machine, we feel that a modest increase in control unit complexity will result in a significant improvement in system performance. Thus, it becomes worthwhile to dedicate some of the increasing number of transistors available per chip to the control unit. However, a careful cost/benefit analysis of the complexity of the control unit is necessary if it is to be implemented in VLSI.

In addition, pipelined architectures exhibit regular data path structures that can be laid out in VLSI in a natural way. This is perhaps one reason why systolic array architectures, a more radical approach to pipelining, have received much attention [Kung79]. Again, the designer must be careful in choosing the number of pipe stages to implement in the architecture, to keep the control and data path delays to reasonable, and nearly equal, levels.

Finally, the PIPE architecture contains several interesting subsystems suitable for custom VLSI implementation, whether or not the processor is built as a monolithic single-chip system. High performance can only be achieved by efficient implementation of certain critical system components. The system's performance is limited by the maximum execution clock rate. Three major performance-limiting system delays are (1) instruction issue logic: it should be possible to issue an instruction in one clock period, (2) memory-processor interface: it should be possible to initiate memory requests at a rate of one per clock period, and (3) instruction cache access: it should be possible to read the instruction cache memory in one clock period. The maximum execution rate will be limited by the slowest of the critical delays. A less significant delay is the ALU processing time: it is desirable to execute one arithmetic operation during each clock period although additional pipeline stages could be inserted if necessary.

By implementing these critical subsystems in VLSI, we hope to gain insights into the overall organization of high performance microprocessor architectures. Many questions remain unanswered. For example, which pipeline delays limit system performance the most? What is the area required by the subsystems? Is it possible to put the entire processor on a single chip, given the available technology and our limited design experience? If not, then how should the system be partitioned? What is the minimum chip to chip communication time, and how can this be achieved? The VLSI implementation aspects of our research are directed towards answering these questions.

5. Software Issues

A decoupled architecture affects system software such as compilers and the operating system. Due to the partitioning of activities, the processors must closely coordinate their work. The architectural description in Section 3 has presented the system features, i.e. data and branch queues, that permit the synchronization of the processors and has indicated how these might be used. In this section, we discuss the issues which arise when writing code for such a system.

5.1. A Compiler for a Decoupled Architecture

Any viable system must have a high-level language compiler. A decoupled architecture presents unique problems in the design of a compiler code generator.

5.1.1. Generating Address Calculation Code

The proposed system is composed of two processors that operate in three different modes. In *independent execution* (IE) mode, the processors operate as dual processors, each working on a task with interaction between the two processors taking place through memory. Code generation is performed as in conventional systems.

In *access/execute* (AE) mode, the compiler's task becomes more difficult. One processor becomes the access processor (AP) and performs all address calculations. The other is called the execute processor (EP) and performs algorithmic calculations. This partitioning is reflected in the generated code. The compiler generates one program with two interacting modules, one for each processor.

The compiler must be capable of isolating those portions of the program associated with the calculation or specification of an operand address from those related to the algorithm being implemented. Easy to isolate is the loop control variable of a DO statement which specifies the array element to be accessed. If two array elements are to be multiplied, the multiplication is assigned to the EP. Array address computation, e.g. multiplying the index by a constant and adding it to a base address, would be compiled into code for the AP.

5.1.2. Address Calculation Code for both Processors

The basic premise of the AE mode of operation is that the tasks performed by the EP take so long that the AP will always have data waiting to be accepted by the EP. This occurs when the EP is performs complex operations. The AP need not always stay ahead of the EP. For some algorithms, address generation may represent a significant portion of a program. To balance the system's performance for address calculation intensive programs, it may be desirable to share the AP's address calculation activity with the EP. One of the processors is nominally the AP; occasionally both processors become APs. This final mode of operation is called *shared access* (SA) mode.

This more complex mode of operation requires a better compiler. In addition to generating code for the AE mode, the compiler must determine when the processors should enter SA mode. The compiler must analyze the time needed to execute EP code and AP code. If the EP execute time is much smaller than AP execute time, and if address calculations can be partitioned between the processors, the compiler may move some of the address calculation code from the AP to the EP.

Code scheduling is an important issue. The execution order of instructions determines the efficiency of CPU pipeline utilization. Dynamically scheduling code to efficiently use a pipeline requires complex circuitry. Leaving code scheduling to the compiler complicates it, and makes it overly dependent on the target machine's implementation. In PIPE, code within a single processor is executed in order. This is effective since the architecture minimizes dependencies between instructions. The instruction execution order when the two processors are considered as a pair is determined dynamically, i.e. when two processors are working on a single program, the instruction execution order will change dynamically, depending on memory activity.

One aspect of code scheduling we have not mentioned is the separation of the prepare to branch (PBR) from the exit. This can be handled by the compiler, providing a simple, effective implementation of dynamic code scheduling.

5.2. Operating System View

The operating system must also perform more work in a decoupled access architecture. In particular, it must be given mode of operation information for a particular program. Scheduling the processors depends on the current operation mode. If in IE mode, a program which runs in AE or SA mode cannot be started until both processors have finished their tasks. The operating system thus monitors the performances of both processors.

Just as any user program may run in AE or SA mode, the operating system may also run in such a mode, or alternate between these modes and IE mode.

6. Summary

We have presented the preliminary PIPE architecture and have proposed an implementation which takes advantage of a simple architecture to achieve a high performance system through pipelining, both in the processors and memory system. We use two processors working in tandem on a single thread of control. Using decoupled processors with instruction caches, we hope to overcome the von Neumann and Flynn

bottlenecks and achieve a combined instruction issue rate greater than one per clock.

The IBM System 360 Model 91 demonstrated that issuing instructions out of sequence can result in a substantial gain in performance. This is a complex, costly solution. PIPE offers the potential for achieving the same effect without the complexity by using two processors communicating through hardware queues.

We have specified a preliminary instruction set which is simple yet compatible with high performance. This instruction set incorporates efficient specification of operands from the queue since the head of the queue is treated as a register. This reduces the number of registers without loss of performance.

Several important aspects of a system design have been recognized but not dealt with here. For example, the handling of interrupts and traps is complicated since the two processors are running asynchronously. While we do not have complete solutions yet, we are investigating the problems.

7. References

[AnST67] Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development*, pp.8-24, January 1967.

[Back78] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM*, Vol. 21, No. 8, pp. 613-641, August 1978.

[Bons69] Bonseigneur, P., "Description of the 7600 Computer System," *Computer Group News*, pp. 11-15, May 1969.

[CeFe78] Censier, L. M. and Feautrier, P., "A new solution to coherence problems in multicache systems," *IEEE Trans. on Computers*, vol. C-27, no. 12, pp. 1112-1118, December 1978.

[CoSt81] Cohler, E. U., and J. E. Storer, "Functionally Parallel Architecture for Array Processors," *Computer*, vol. 14, no. 9, pp.28-36, September 1981.

[Flyn66] Flynn, M. J., "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901-1909, December 1966.

[HJBG82] Hennesy, J., N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, March 1982.

[IBM74] "System/370 model 155 theory of operation/diagrams manual (volume 5): buffer control unit," IBM System Products Division, Poughkeepsie, N.Y., 1974.

[IBM76] "System/370 model 168 theory of operation/diagrams manual (volume 1)," Document No. SY22-6931-3, IBM System Products Division, Poughkeepsie, N.Y., 1976.

[KaCo81] Karplus, W. J., and D. Cohen, "Architectural and Software Issues in the Design and Application of Peripheral Array Processors," *Computer* vol. 14, pp.11-17, September 1981.

[Kung79] Kung, H. T., "Let's Design Algorithms for VLSI Systems," *Proc. Conf. on VLSI: Architecture, Design, Fabrication*, California Institute of Technology, January, 1979.

[PaSe81] Patterson, D. A., and C. H. Sequin, "RISC-I: A Reduced Instruction Set VLSI Computer," *Proc. of the Eighth Annual Symposium on Computer Architecture*, May 1981.

[Ples82] Pleszkun, A. R., "A Structured Memory Access Architecture," CSG Report No. 10, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, August 1982.

[Radi82] Radin, G., "The 801 Minicomputer," *Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47, March 1982.

[Russ78] Russel, R. M., "The CRAY-1 Computer System," *Communications of the ACM*, Vol. 21, No. 1, pp. 63-72, January 1978.

[Scho71] Schorr, H., "Design Principles for a High-Performance System," *Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, pp. 165-173, April 1971.

[Smit82] Smith, J. E., Decoupled Access/Execute Computer Architectures *Proc. of the Ninth Annual Symposium on Computer Architecture*, May 1982.

[Thor70] Thornton, J. E., *Design of a Computer -- The Control Data 6600*, Scott, Foreman and Co., Glenview, IL, 1970.