An Instruction Cache Design for use with a Delayed Branch

Andrew R. Pleszkun and Matthew K. Farrens

Computer Sciences Department
University of Wisconsin
Madison, WI

ABSTRACT

In this paper, we present the design of an instruction cache for a machine that uses an "extended" version of the delayed branch instruction. The extended delayed branch, which we call the *prepare to branch*, or **PBR** instruction, permits the unconditional execution of between 0 and 7 instruction parcels after the branch instruction. The instruction cache is designed to fit on the same chip with the processor and takes advantage of the PBR instruction to minimize the effective latency associated with memory references and the filling of the instruction register. This paper discusses the design and implementation issues associated with realizing such an instruction cache. We present critical aspects of the design and the philosophy used to guide the development of the design. Finally, some timing results are presented which indicate the performance of critical circuits.

1. Introduction

The conditional branch instruction has long been known to challenge the designers and implementors of high performance pipelined machines. It is understood that a conditional branch instruction disrupts the flow of instructions in a pipelined machine. While a conditional branch instruction passes through the stages of the CPU to the pipeline stage where the branch condition is finally evaluated, all the previous stages sit idle. Thus, when the branch target is finally decided and instructions of the branch target begin execution, the full latency of the pipeline is experienced [8,3].

Some implementors have tried to minimize the effects of this latency by guessing that one branch of the target will be taken. Such a technique was reported on the 360 Model 91 [1]. Since the 360/91, other schemes have been proposed to guess which instructions should be *conditionally* executed [14]. Naturally, the cost of such guessing techniques lies in the extra circuitry required to assure that before the conditionally executed instructions change the machine state, these instructions can be canceled, just in case the other target of the branch is taken.

There is a second level of guessing that can take place which does not involve the conditional execution of instructions. This is not as complex as conditional execution and uses buffers to store the first few instructions associated with each target of the branch. Thus when the branch decision is finally made, the full latency of main memory is not experienced. Variations of these techniques include the previously mentioned 360/91 [1] and machines such as the CDC 6600 [18]. Instead of using explicit instruction prefetch buffers, some machine designers have chosen to use a cache to perform the function of an instruction buffer [2]. We will discuss our use of a cache later in this paper.

Recently, projects such as MIPS [6] and the 801 [12] have proposed the use of the delayed branch to make more efficient use of a pipeline. In the IBM 801, no instructions are unconditionally executed after a *normal* branch instruction. Its issue logic, however, always initiates the instruction that has been placed immediately after a *branch with execute* instruction. This permits the 801 to unconditionally execute one instruction beyond the branch instruction. In MIPS [5], those branches involving a memory reference (jump indirect) have a branch delay of two; others have a delay of one. A delay of two is used for the jump indirect because it cannot complete in one clock period since an additional clock period is needed to fetch the branch target from memory.

In our investigations with the PIPE architecture [7,4] we have expanded on this idea and believe that one could do better by unconditionally executing more that one instruction. The PIPE branch instruction permits up to 7 parcels to be unconditionally executed before the actual change in control flow occurs. We call this the *prepare to branch* or **PBR** instruction (see Figure 1 for the instruction format). In the PIPE architecture, a parcel is 16 bit quantity and instructions are 1 or 2 parcels in length. As can be seen from Figure 1, the PBRinstruction is a 16-bit instruction. Bit 0 indicates that this a short or single parcel instruction, while Bit 1 indicates that this is a PBR instruction. The remaining bits indicate how the

branch is to be processed. Bit 2 of the instruction is used when the PIPE processor is used in decoupled operation [16] with 2 processors. A discussion of this is beyond the scope of this paper and does not effect the implementation of the instruction cache. The 3-bit COND field specifies the branch condition to be evaluated, Bit 6 is unused, and Bits 7-9 specify the general purpose register which is to be compared against zero. Since we wanted to make the PBR instruction a single parcel instruction, instead of specifying the branch target address in the instruction, one of 8 Branch Registers is specified. The branch registers are not part of the general purpose registers, but a separate set of registers that are loaded with branch target addresses. The final field (CNT) contains a 3-bit *branch count* field. The branch count specifies the number of parcels to be unconditionally executed after the PBR instruction. The number of unconditionally executed instructions is variable and depends on the program.

As with the delayed branch, the PBR instruction attempts to keep a pipeline stage busy after a conditional branch has been encountered. The merits of the PBR are apparent only when a machine is implemented with significant amounts of pipelining or overlapped operations. (In this discussion, when we use the term pipelining, we include every stage in a machine, such as instruction fetch, decode, instruction issue, etc.) While adequate for small degrees of overlapped operation that only involve simple instruction prefetch, the delayed branch does not continue to make performance gains when the machine implementation is changed to a version having many pipeline stages. The PBR instruction will improve machine performance as long as the number of pipeline stages before the stage that performs the condition evaluation is less than 7 and the program has a sufficient quantity of operations that can be unconditionally performed after the branch instruction. Our experience with the PBR instruction indicates that, for our benchmark programs, a compiler can easily generate code with an average of 4 parcels to be unconditionally executed after the branch instruction [19]. We did not try to schedule more code that this simply because the target implementation does not have more than 4 stages.

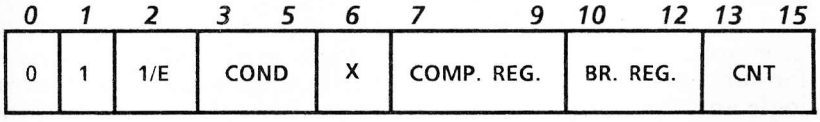

Figure 1. Prepare-to-branch instruction format.

Using the PBR instruction, one can keep the pipeline full without the complex logic required in schemes that rely on guessing. Within the context of VLSI design, this would not really matter if one believes that chip logic is *inexpensive*. The PBR instruction, however, has another impact on the performance of a single chip processor. Since a single chip processor has a limited amount of bandwidth for transferring information on and off the chip, efficient utilization of available bandwidth is extremely important. With the PBR instruction the pipeline is kept full without the need to make guesses; thus, no off-chip bandwidth is wasted in bringing unneeded instructions on-chip.

While the PBR instruction helps keep the pipeline full and guarantees every instruction brought on-chip will be executed, the pins available for transferring items on and off chip are still limited. Thus even with the PBR instruction, chip bandwidth remains a scarce resource. In the PIPE implementation, 16 pins are dedicated for input operations and 16 are dedicated for output operations. Requests for instructions go through the 16 read pins. This means that instruction input must be multiplexed with data input. This is unreasonable and could severely impact overall performance. To limit the effects of this sharing, we decided to include an on-chip instruction cache. Instruction caches have been suggested by others [11,17,15]. The PIPE instruction cache is unique for two reasons; first, it is designed to reside on-chip with the rest of the processor and second, its control logic circuitry can take advantage of the PBR instruction to check for cache hits and schedule main memory requests on a miss.

2. The PIPE Instruction Cache

The structure of the PIPE instruction cache is intimately connected to the pipelined structure of the PIPE machine and in particular the instruction fetch unit. In addition, special considerations must be made if the cache is to take advantage of the PBR instruction.

The PIPE instruction cache is directed mapped and composed of 16 4-word lines for a total of 64 words. Although we would prefer a larger cache, we are limited by both technology and the sharing of chip real estate with the rest of the PIPE processor. This instruction cache, while small, proves sufficient for our purposes. It allows us to verify the design of the control logic and the results of our simulations indicate that each of the Lawrence Livermore Loops [9] easily fit into a cache this large. As will be seen in section 3.1, we use a static memory cell and a bus structure that makes the design relatively insensitive to additional cache lines.

Associated with the cache are several registers (see Figure 2). There is a 64-bit assembly register located above the cache which holds 16-bit quantities as they arrive from memory. Once the 4 entries for a line have been assembled, the entire line is written into the cache. Below the cache are two 64-bit registers, the *instruction queue buffer* (IQB) and the *instruction queue* (IQ).

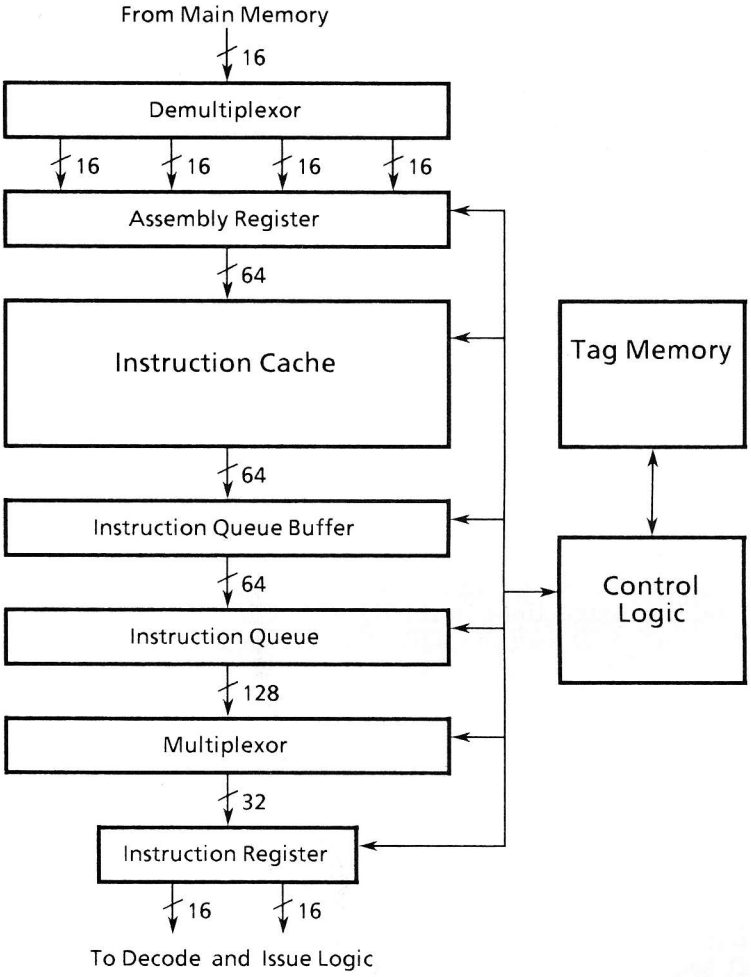

Figure 2. Instruction Cache Organization

Finally there are a pair of 16-bit registers that form the instruction register (IR).

The interaction of these registers can best be explained by considering them in reverse order. The IR is one boundary of the instruction cache/fetch subsystem. The output of the IR feeds into the decode logic. The IR consists of a pair of 16-bit registers since instructions can be one or 2 parcels long. This is similar to the CIP and LIP registers of the Cray1-S [13]. One difference in our implementation is the ability to place 2-parcel instructions into the IR on every clock cycle. In the Cray, consecutive 2 parcel instructions are placed in the CIP-LIP registers on every other clock. The ability to place 2 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 consecutive pair of parcels in the IQ to be placed in the IR during one clock (see Section 3.1).

Above the IR sits the IQ. As mentioned above, this register is 64 bits long, matching the cache line size. Some subset of parcels in this register is always guaranteed to be executed. The number of instructions in the IQ that get executed depends on the existence of a PBR instruction in the IQ, the location of the branch target, and the value of the branch count.

Due to the encoding of the PIPE instruction set, the existence of a branch instruction is determined by 1 bit of the opcode. Furthermore, by looking at 3 more bits, the branch count associated with branch instruction can be calculated. Thus, before the instruction is issued or even loaded into the IR, the cache can *easily* determine whether all the instructions in the IQ will be executed. If the control logic determines that all the instructions in the IQ and at least one instruction sequentially beyond the instructions in the IQ will definitely be executed, the control logic will load the IQB. The IQB acts as a prefetch buffer for the IQ. To permit loading of the IR with 2 parcels per clock, there is a special path from the left-most entry of the IQB through the IQ into the multiplexors feeding the IR. This feature exists to take care of the case when the last (right-most) entry of the IQ is a 2-parcel instruction.

As long as the instructions being executed form straight-line code, instructions are loaded one-by-one into the IR from the IQ. The instructions in the IQ are scanned for a branch, and if no branch is found (as would be the case for sequential code), the IQB is loaded with the next sequential line. If the request for the next sequential line causes a cache miss, a request is made to memory to load a cache line. The incoming parcels are assembled in the AR. When all the requested words have arrived and have been assembled, the contents of the AR are written into the cache and also into the IQB. On occasion, every entry in the IQ will have been used before main memory can fill the AR. In

such a situation, execution is blocked until the AR is filled, at which time its contents are moved directly into the IQ.

When a PBR instruction is encountered, keeping the IQB filled with valid instructions presents a challenge. Depending on the position of the PBR instruction in the IQ and the value of the branch count, it is possible that some of the instructions in the IQ will not be executed. Whether the instructions are executed depends on the outcome of the branch. A further complication with the PBR scheme is the fact that all of the instructions to be unconditionally executed may not fit in the IQ. Due to a large branch count, the unconditionally executed instruction may spill-over into the IQB. In such a case, the control logic will try to fill the IQB and make a memory request should the needed line be missing from the cache. Once the contents of the IQ have all been accessed, the IQB is moved into the IQ. The control logic must remember that a PBR instruction was issued and how many parcels beyond the PBR instruction have been issued. Until all the parcels specified by the branch count have passed into the IQ, the control logic keeps the IQB filled using the same strategy used for strictly sequential code. Once the last of these parcels has moved into the IQ a different strategy is required.

If the branch outcome is not known at the time the IQB becomes empty, by default an attempt is made to load the IQB with the next sequential line. This forces a cache look-up. If the result of the look-up is a cache hit, the line is moved into the IQB. In this case, if the branch is not taken, the cache/fetch unit operates as though sequential code were being executed. If the result of the look-up is a cache miss, the control circuitry waits until the outcome of the branch is determined before attempting to load the IQB. When the branch outcome is determined, a cache look-up is performed. The result of this look-up will be a hit if the branch target is in the cache, and a miss if the next sequential line or the branch target are not in the cache. If a miss is experienced, a request is made to memory for the missing line. Keep in mind that while the memory fetch is being performed, depending on the branch count, the processor can still be performing useful work.

In the above strategy, when there is a branch pending and the next sequential line is not found in the cache, no attempt is made to find the branch target in the cache. The reason for this strategy arises from the use of branch registers and the pipelined structure of the machine. At the time the PBR instruction is encountered in the IQ, there is no guarantee that the branch register specified in the instruction contains the correct branch target address. An instruction that loads the branch register may be in some stage of execution. The hardware only guarantees that by the time the branch condition is evaluated the branch register will have been loaded with the correct value.

These control strategies highlight the basic philosophy of the PIPE instruction cache/fetch operation. When possible, we guess to keep the IQB and IQ filled. The effects 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. We have chosen not to prefetch instructions even though it may appear to be a worthwhile option for a cache with our small line size. This is mainly due to the limited bandwidth of a single chip processor. For straight line code, we achieve the effect of prefetching by checking for branch instructions in the IQ. Requests for a new set of instructions are made well before the instructions in the IQ reach the issue logic. In the case of branches, the degree of achieved prefetching effect depends on the branch count 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. Unlike prefetch strategies in a conventional cache, we are guaranteed that some portion of every line we fetch is executed.

3. Implementation Issues

The implementation issues in the design of the cache can be broadly divided into the design of the datapath and the design of the control logic. The datapath design issues are applicable to general cache design issues, while the those related to the control logic are almost entirely dependent on the functionality needed to take advantage of the PBR instruction. Throughout the implementation a standard 2-phase non-overlapping clock is used.

3.1 Datapath Design

The cache itself has two major sections: (1) the cache cell array and (2) the tag cell array. The same static RAM cell is used in both. This RAM cell is of standard design, using two inverters connected together by a feedback path through a pass transistor. All reads and writes to a RAM cell are performed via the same bus. Reads and writes both occur during phi2 while bus precharge occurs during phi1. A basic cell contains two of these memory cells overlapped back-to-back to minimize total cell area. To improve the speed of a read operation, the output inverter of the memory cell does not follow standard ratioed logic design rules. Since the bus is precharged, we do not need large pullup transistors to drive the bus. Instead an oversized pulldown transistor with a length of 2 lambda and a width of 34 lambda is used, allowing rapid discharge of the bus (see Figure 3). The cell was optimized to make the control lines (read, write, and refresh) as short as possible since they are run in poly. The

unusual appearance of the cell design is due to this optimization and the large size of the pulldown transistor.

These 2-bit cells are arranged in 2 16x16 arrays with the cache decode logic running down the middle between them. The cache cell array was split in half and a superbuffer was placed in the middle of each half to minimize the delay due to the poly control lines. One row in this matrix corresponds to a cache line. The bit lines are run in metal, allowing cache lines to be added

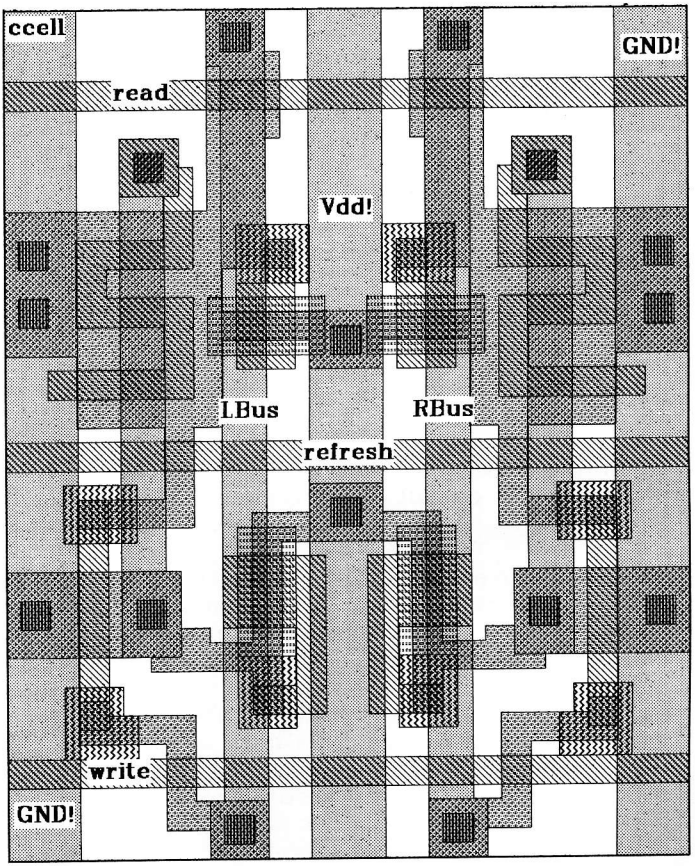

Figure 3. Cache Memory Cell Layout. This basic cell contains 2 bits of the cache memory which are read and written via the LBus and RBus. Refresh is done using the rsh signal.

without significantly effecting cache access time. If on the other hand the line size is increased, significant performance penalties may be experienced. Because the decoders are physically a small part of the entire circuit area and because the speed of generating the cache control signals is critical, the area allocated to the decoders was doubled so that the input signals to the decode logic have fewer gates to drive. Essentially, the input lines to the decode logic are duplicated before being run through the center of the cell array. Figure 4 shows the decode logic for 3 lines of the cache. Notice that there are 16 parallel metal lines that run through the distributed NOR gates which are used to select a row. The left set of 8 signals are duplicates of the 8 signals on the right. With such an arrangement, the superbuffers driving the inputs to the decode logic have half the capacitive load to drive. Each set of 8 signals represent the complemented and non-complemented versions of the cache line address. The criss-crossing that occurs just above the distributed NORs simply switches an input and its complement so that the proper signal is used by the pulldown of the NOR gate. Superbuffers that drive the cache read and write lines are located to the left and right of the decode logic.

The bitlines run vertically through the cache storage arrays, the AR, IQB, and the IQ. A cache line is not physically organized as a sequence of 16-bit words. Instead, Bit 0 of all 4 words are grouped together, followed by Bit 1, Bit 2, etc. The check for the PBR instruction in the IQ is made easier by this arrangement. In addition, when the AR is loaded, this arrangements permits the use of 16 small 1:4 demultiplexors instead of one large demultiplexor that would require massive amounts of line crossovers. This also permits the use of 16 small multiplexors at the output of the cache/fetch unit that feed the IR.

As stated above, the tag array uses the same basic memory cell to store the tags. The tag array also uses the same decode logic as the cache cell array to select a tag. The tags are 10 bits long, one tag associated with each line of the cache. Each tag also has associated with it a valid bit. Given an instruction address, the high order 10 bits (Bits 0-9) are used for the tag. The next 4 bits of the address select a line, while the remaining 2 address bits are used to select the word with in the line. Determining whether a given instruction address generates a cache hit is a straightforward, though time critical operation. Given a valid instruction address on phi2, the tag array will determine if that line is in the cache during phi1, and on the following phi2 the associated cache line can be loaded into the IQB.

The IQ and IQB are composed of cells similar the one used in the cache cell and tag arrays. The major difference is that a

separate bypass bus must be provided from the IQB to the IQ to enable the loading of the IR with a 2-parcel instruction every clock. This bypass bus is used when the the last instruction in the IQ is a 2-parcel instruction. In such a case, the second parcel must be taken from the IQB. The bitline used to perform the bypass is enabled by an extra control line that must be run through the IQB.

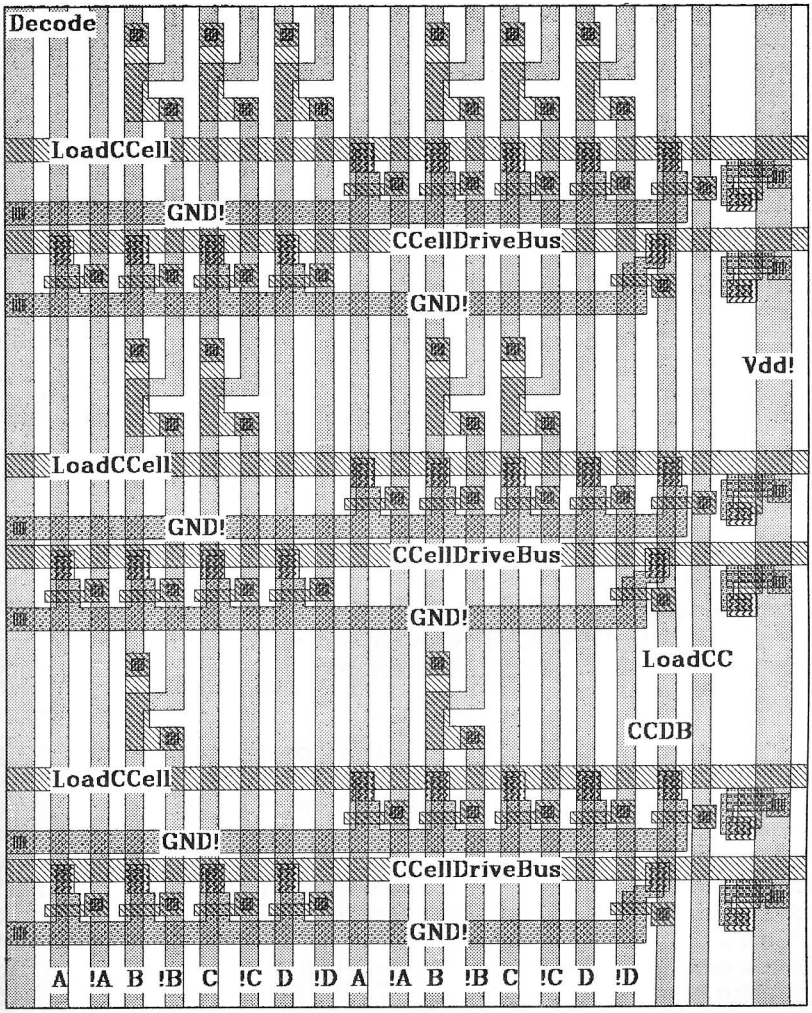

Figure 4. Decode Logic for Selecting a Cache Line. This figure shows the decode logic for 4 cache lines.

The output of the IQ feeds into 16 pairs of 4:1 multiplexors. These multiplexors are not true 4:1 multiplexors, however. The fact that the IR always contains two sequential parcels allows the multiplexor design to combine the 32 logical 4:1 multiplexors that are logically necessary into 16 "4:2" multiplexors that accomplish the same task. This circuit could also be interpreted as a special type of barrel shifter or field extractor. The low-order 2 bits of the PC select the pair of parcels in the IQ to be routed into the IR. The selected pair of parcels are routed directly into the IR where they are latched on phi1 and stored in pairs of bits. The actual separation of bit pairs into 2 distinct words is done as the instruction is transmitted to the decode logic. Deferring this separation until after the instruction has been latched in the IR removes the delays due to interconnection problems from the cache, where timing is critical, to a non-critical path. The first parcel, which is used by the decode logic leaves the IR via metal lines and goes directly to the decode logic. The second parcel crosses over these metal lines in poly and is latched for later use. This way, the delays due to interconnection affect a path whose data (i.e. the second parcel) is not used by the decode or issue logic.

The AR also uses the same basic cell design as the other registers. In addition, the AR never reads a bit line, it only drives it, so there is no write control signal running through the cell. The writing of a cell is controlled via the select signals in the demultiplexor which is in turn driven by the input pins from main memory. All the basic cell designs match pitch so that they can be stacked.

3.2 Control Logic

The control logic is composed of five functional units: (1) program control logic, (2) branch control logic, (3) buffer and queue control logic, (4) validation logic, and (5) main memory interface logic.

The program control logic is responsible for incrementing and loading the program counter. Due to the buffering of instructions and the PBR instruction, this presents some problems. One of these is that in a pipelined machine, it is difficult to define the PC. Should the PC correspond to the instruction just issued, the instruction in the IR, or the next instruction in the IQ? In our case, what would correspond to the PC in a conventional machine is a register whose contents point to the parcel that will be loaded into the IR on the next clock. In reality, the high-order 14 bits of this PC are never needed, so they are not implemented. Only the low-order 2 bits are implemented and used to control multiplexors that route instructions into the IR.

The PC must have the capability of being incremented by 1 or 2 depending on whether a 1 or 2 parcel instruction is being referenced. Updating the PC is complicated when the last instruction in the IQ is a 2-parcel instruction and the IQB does not contain valid data (the next sequential line was not in the cache) and when there is a branch pending. PC incrementation also depends on whether the program counter has just been loaded, whether the branch count is equal to or has gone to zero, and whether there is a valid line in the IQ and/or IQB.

There is an other "PC" that contains the address of the next line to be referenced. We call this the *pending PC*. The next line is either the next sequential line or the branch target. Because a line is being referenced, only the higher order 14 bits are needed. To accomplish this task the pending PC is a chain of ripple counters. A Manchester carry chain was considered but rejected due to the poor performance of such a circuit in our ALU design. The actual measured carry chain performance was much worse than that predicted by Crystal (approximately 45% slower than predicted, which is more noticeable since the rest of the ALU circuit was almost 40% faster than Crystal predictions). Since this pending PC update circuitry lies in a critical timing path of the cache, we cannot afford to spend a clock phase to do the precharging needed with a carry chain. The speed requirements of this circuit outweigh the importance of such considerations as regularity and minimum area.

The branch control logic recognizes that a branch instruction is currently in the IR. It is responsible for keeping track of whether there is a branch pending, and for recognizing as soon as possible the result of the PBR instruction. It is also responsible for decrementing the branch counter. When the branch counter reaches zero, it is responsible for proper transfer of program flow based on the result of the branch. This involves several operations. It must recognize when all the parcels that are guaranteed to execute after the PBR instruction are finally in the IQ. At this point, it must move the contents of the branch register associated with the PBR instruction into the pending PC register so that a cache look-up can begin. The logic for decrementing the branch count value must be capable of decrementing by 1 or 2 depending on whether the instructions to be unconditionally executed following the PBR instruction are 1 or 2 parcel instructions. Single and 2-parcel instruction are indicated by Bit 0 of the instruction.

The buffer and queue control logic is responsible for keeping the IQB and IQ full. This logic determines when to fill the IQ or IQB and what with . It decides which memory cells should drive the bit lines and which cells or registers receive the data. For example, this logic is responsible for transferring the

contents of the IQB into the IQ on demand. It also permits the AR to drive the bit lines and determines whether the cache line, the IQB, or the IQ should receive this data. This control depends on conditions such as: a valid IQB line, a valid IQ line, a cache hit, and a pending branch.

The validation logic determines whether the lines in the IQ and IQB are valid and whether the IR contains a valid instruction. Sometimes, when the IQ and IQB are both invalid, such as on a long memory access, the IR will contain a previously executed instruction. Naturally, this instruction should not be re-issued.

The main memory interface logic is probably the simplest of all the control units. It determines when a main memory access should be initiated based on signals from the other control units. It is also responsible for loading the AR as words are received from main memory.

This is a very brief summary and highlights the function of each of the control units. A detailed description of all the control signals is impossible in this paper. Although there are 5 different logical control units, the current implementation uses 10 PLAs. The assignment of operations to clock phases has been carefully tuned to eliminate **all** idle clock cycles. In addition to these 10 main PLAs there are a small number of random logic circuits. If the logic equations governing the control logic were to be implemented strictly with PLAs, the sizes of the PLAs would cause unacceptable delays.

4. Implementation Status

As mentioned earlier, the cache subsystem is meant to reside on-chip with the rest of the PIPE processor. Currently all sections of the processor have been laid out and with the exception of the cache subsystem and the issue logic subsystem, are being or have been fabricated. The worst-case delay path through the ALU and register file was estimated by the 1985 release of the Crystal circuit simulator from Berkeley to be approximately 74ns. The actual fabricated circuits had on average worst-case delays of slightly under 60ns. Three of the returned chip experience sub-60ns delays while one had a 62.5ns delay. We have used 60ns as the maximum delayed allowed in any circuit. When designing all of the other circuits, we have made sure that they have worst-case delays of less than this value. As far as the cache is concerned, all parts of the cache have been laid out. Simulation results for critical paths in the cache indicate that the worst-case delay path is 59ns. This occurs in the cache hit logic. We are using the MOSIS n-MOS 2 micron technology.

5. Conclusions

In this paper we have discussed the design issues associated with the implementation of an on-chip instruction cache that takes advantage of an extended delayed branch instruction to minimize off-chip memory requests. In the architecture for which this cache is designed, an extended version of the delayed branch instruction is used. In our case, the instruction permits up to 7 instruction parcels to be unconditionally executed after the instruction. The number of instruction parcels to be unconditionally executed vary from 0 to 7 and is specified in the branch instruction. With this type of branch instruction, the machine's pipeline is more easily kept full when a branch occurs. Before all the instructions to be unconditionally have been executed, the branch outcome will have been determined, giving the cache and fetch logic time to access the branch target, minimizing the effects of memory latency and off-chip communications delay, and keeping the pipeline full even when a branch is taken. The instruction cache is used to minimize off-chip memory accesses for instructions. Simulation results indicate that worst-case delays are under 59ns. The logic to control instruction fetching and changing control flow has been moved to reside side-by-side with the cache control logic.

Acknowledgments

This work was funded in through NSF Grant MCS82-02952.

References

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

[2] R. W. Doran, "The Amdahl 470V/8 and the IBM 3033: AComparison of Processor Designs," *Computer*, pp. 27-36, April 1982.

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

[4] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: a VLSI Decoupled Architecture,"*Proc. of the Twelveth Annual Symposium on Computer Architecture*, pp. 20-27, June 1985.

[5] T. R. Gross and J. L. Hennessy, "Optimizing Delayed Branches," *Proceedings, 15th Annual Workshop on Microprogramming*, pp. 114-120, October, 1982.

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

[7] J. T. Hsieh, A. R. Pleszkun and J. R. Goodman, "Performance Evaluation of the PIPE Computer Architecture," Technical Report #566, Computer Sciences Department, University of Wisconsin-Madison, November 1984.

[8] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw Hill, New York, 1981.

[9] F. H. McMahon, "FORTRAN CPU Performance Analysis," Lawrence Livermore Laboratories, Livermore, CA, 1972.

[10] D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, No. 1, pp. 8-21, January 1985.

[11] D. A. Patterson and C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Trans. on Computers*, Vol. C-29, No. 2, February 1980.

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

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

[14] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. of the Eighth Annual Symposium on Computer Architecture*, pp. 135-141, May 1981.

[15] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982.

[16] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Trans. on Computer Systems*, Vol. 2, No. 4, pp. 289-308, November, 1984.

[17] J. E. Smith and J. R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Trans. on Computers*, Vol. C-34, No. 3, pp. 234-241, March 1985.

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

[19] H. C. Young and J. R. Goodman, "A Simulation Study of Architectural Data Queues and Prepare-to-branch Instru ction," *Proceedings, IEEE International Conference on Computer Design*, pp. 544-549, October 1984.