Architecture implications in the design of microprocessors

by R. E. Matick
D. T. Ling

This paper examines how architecture, the definition of the instruction set and other facilities that are available to the user, can influence the implementation of a very large scale integration (VLSI) microsystem. The instruction set affects the system implementation in a number of direct ways. The instruction formats determine the complexity of instruction decoding. The addressing modes available determine not only the hardware needed (multiported register files or three-operand adders), but also the complexity of the overall machine pipeline as greater variability is introduced in the time it takes to obtain an operand. Naturally, the actual operations specified by the instructions determine the hardware needed by the execution unit. In a less direct way, the architecture also determines the memory bandwidth required. A few key parameters are introduced that characterize the architecture and can be simply obtained from a typical workload. These parameters are used to analyze the memory bandwidth required and indicate whether the system is CPU- or memory-limited at a given design point. The implications of caches and virtual memories are also briefly considered.

The rapid advances in density and performance of very large scale integration (VLSI) technology have provided the designer with the opportunity to achieve unprecedented performance on a single chip of silicon. The emerging technologies have the potential of producing microprocessors with performance that matches or exceeds that of medium to large machines today. With this performance, users will come to expect other features normally associated with large systems, such as a powerful instruction set, large main memory, and large virtual address spaces. Furthermore, in order to achieve high performance the microprocessor designer must resort to many of the techniques pioneered in large machines, such as cache, pipelined organization, and parallel functional units. It is within this technological framework that we investigate in this paper the relationships between the architecture and the implementation of microsystems that consist of one or a few VLSI chips.

The structure, architecture, and design of any computing system are affected by an extremely large number of interrelated parameters so that any attempt to optimize a total system by including all parameters is virtually impossible. Not only is there no adequate and well-defined model, but also the complexity and possible variations far exceed the intellectual capability of any individual. Therefore, to reduce the complexity and limit the number of parameters that must be considered simultaneously, a data processing system is typically viewed as a hierarchical structure. Each level of the hierarchy reflects one major aspect of the system and hence contains only a small subset of the total parameters. Each interface between these various levels is usually assumed to be relatively independent of the two levels being interfaced. For instance, at the highest level of the hierarchy is the applications program, often written in a high-level language. It is assumed that the language can be designed without consideration of the remainder of the hierarchy. Below this level lie the compiler and operating system. The

latter manages the system resources and provides basic system services. Below that lies the architectural interface, the nominal boundary between hardware and software. This key interface consists of all facilities and operations that are made available to the programmer; for example, the registers, instruction set, and the addressing capabilities are all part of the architecture. Finally, there is the hardware level, which can further be stratified into the machine organization, including the pipeline structure, microcode and data flow, the circuit family, and the technology itself.

Although the hierarchical structure, in principle, allows each level to be designed separately, closer examination reveals interesting possibilities. Recent papers have suggested that the design of a processing system can be improved if several levels of the hierarchy are considered together. For example, recent designs[1,2] have been guided by the relationship between the architecture and the high-level language compiler. At the architectural level, simple but powerful instructions were chosen to form an efficient compiler target. A simple instruction set made code selection in the compiler much easier. Furthermore, careful choice of instructions resulted in total path lengths comparable to those generated using a more complex instruction set. In other words, complex operations can very efficiently be broken into a sequence of more general, fundamental steps. Finally, by not including memory-to-memory operations, but rather allowing only register stores and loads to and from memory, the compiler can frequently overlap access to storage with execution of other instructions, giving better throughput.

VLSI permits more of the system to be viewed as a unit on a chip.

Another example has been to integrate machine organization with the control program design.[3-5] One way this can be achieved is to create an explicit architecture of the primitive control program operations. For example, in System/370,[6] instructions were added to support dual address spaces, and in both IBM System/38[4] and Intel iAPX 432,[5] instructions are available for process management and interprocess communication. Rao and Rosenfeld[3] discuss other opportunities to exploit the intent of the operating system at the machine organization level.

Thus, by crossing the boundaries between the various levels of the hierarchy, new opportunities are afforded for improving the overall system performance. Since VLSI permits more and more of the system to be viewed as a unit on a chip, not only is the designer permitted the opportunity to cross these boundaries, but in fact it also becomes a necessity. This is especially true at the lower levels of the hierarchy.

In this paper, we examine the interaction between two levels in the hierarchy, namely the architecture and machine organization levels, within the context of a high-performance VLSI-based microsystem. We focus on the following two issues:

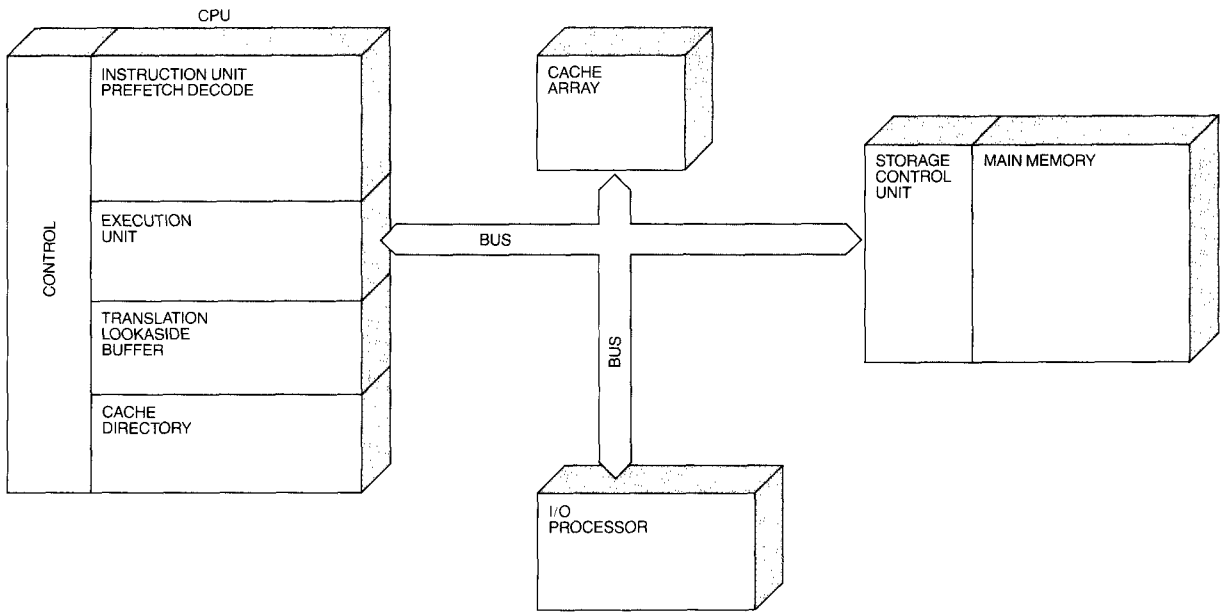
- The effect of the architecture on the instruction decode unit of the CPU.
- The relationship between architecture and the memory subsystem, especially with regard to necessary memory bandwidth and the need to support a demand-paged virtual memory.

With a few simple, general concepts, a number of important principles and tradeoffs concerning the processor and especially the memory system can easily be deduced. In order to do this, it is necessary to understand the nature of a pipeline system, as well as the conflicts and disruptions that can occur in the pipeline flow, because these significantly reduce the system performance. These conflicts and disruptions, together called *hazards*, can occur within the CPU itself or within the memory subsystem. If they occur mainly within the CPU, the CPU is the performance bottleneck. On the other hand, if most of these hazards are caused by the memory system, the memory bandwidth is inadequate, and attempts to improve the overall performance by improving the CPU are of little value. Obviously a balanced design is necessary; therefore, we shall derive the general range of certain simple parameters required for such a balanced design.

The VLSI environment

Since the introduction of the first microprocessor about fourteen years ago, there has been a two-order-

Figure 1 Schematic of a generalized microprocessor system organization



of-magnitude increase in both the density and performance of the chips. The increase in performance has come about not only because of improvements in technology, but also because of better circuit designs and the adoption of more sophisticated machine organizations. Many of these machine organization techniques have been directly modeled after the design of large mainframes. However, many of the approaches used in large mainframes are not appropriate in the microsystem environment. The constraints in the microsystem arena stem from the desire to limit system cost and to emphasize the cost-performance ratio rather than performance alone. As a result, the processor is generally limited to only one or a few VLSI chips. In addition, the number of interchip connections should be kept small because they are expensive from a number of standpoints. First, it is difficult to fabricate and package chips that have a large number of input/output pads. Second, even if a large number of outputs are made available, the number that can be simultaneously switched is limited by the inductive switching noise.[7,8] In addition, the number of interchip connections in a critical path may limit system performance because interchip propagation tends to be much slower than on-chip propagation. A further constraint is that the cooling system for a microsystem must usually be inexpensive and compact, requiring, for example, only a fan. This is very unlike a large mainframe system. Hence the allowed heat dissipation limits the chip power, with appropriate reductions in circuit as well as interchip communication speed.

Partitioning a VLSI microsystem is therefore a critical first task and is determined both by the technology, which establishes the number of circuits that can be placed on the chip, and by the number of I/Os available. In general, the constraints just discussed imply that each chip contains a complete functional unit, such as the entire CPU or the floating point unit, or at least a complete subunit, such as the data flow or microcode. However, even with such a functional partitioning, the need to restrict interchip connections can have a profound effect on the machine organization chosen. For example, if the machine is simply partitioned into microcode and data flow, the size of the microword is determined by the pins available. A more sophisticated partitioning is therefore often desirable.[9] The constraints on the number of pins and simultaneous switching also tend to limit the width of the main processor-memory bus. The implications of this form one of the main themes of this paper.

Another important component of the cost of a microsystem is the design cost, measured both in terms of dollars and design time. From the point of view of physical design (layout), design cost can be re-

duced by using regular physical structures, for example RAMs, register files, and ALUs, so that fewer unique devices have to be drawn.[10] From the point of view of logic design, control logic is far more difficult to design and debug than data flow. (Intensive use of microcode has been one method of dealing with this complexity.) Therefore reducing the complexity of control logic will have greater leverage than simplification of the data flow. This will be seen in the discussion of instruction decoding.

The machine model

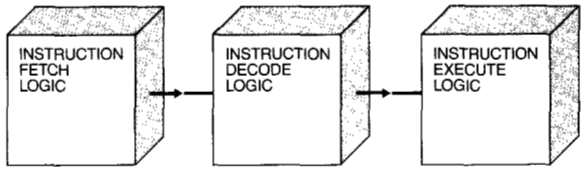
We describe in this section a machine model that serves as the standard example for later discussion. We assume that the processor consists of the following chips (see Figure 1):

- CPU that houses the instruction prefetch and decode unit, the execution unit, the register file, and the address translation logic, as well as cache directory and control logic if cache is used.
- Storage control unit for management of main memory.
- I/O processor to off-load I/O from the main processor and handle Direct Memory Access (DMA) requests.

Cache itself, if present, is assumed to be located external to the CPU on a few (perhaps one to four) high-speed memory chips. A high-speed system bus connects the CPU, storage control unit, and I/O processor and provides a path for cache reload for systems with cache. In general, we assume that this bus is capable of transferring one full memory word (typically four bytes) on every cycle. We also assume that the machine is pipelined to improve performance. By pipelined we mean that the instructions are processed in stages by separate portions of the hardware, much as in an assembly line.[11] The major reason for using such a structure is speed. All data processing can be reduced to a number of fundamental operations, some of which are quite simple and some of which are more complex. The complex operations can be broken into a series of simpler sequential stages.

In Figure 2, we illustrate a simple machine with the following three pipeline stages: instruction fetch, instruction decode, and instruction execution. When an instruction has completed one stage, say instruction fetch, it moves to the next stage, instruction decode. At the same time, the instruction fetch unit starts fetching the subsequent instruction, while the

Figure 2 Schematic of a simple three-stage pipeline processing five instructions

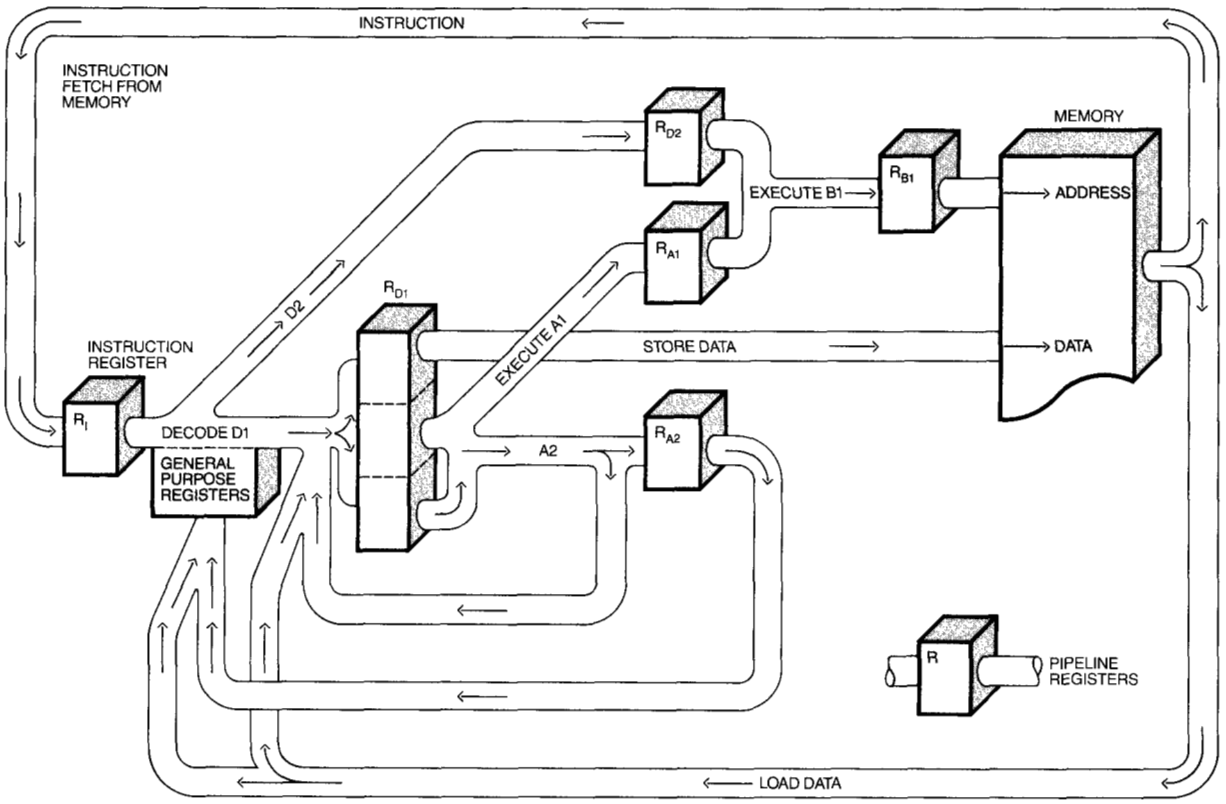


	INSTRUCTIONS I_N PRESENT AT TIME				
	T_1	T_2	T_3	T_4	T_5
INSTRUCTION FETCH LOGIC	I_1	I_2	I_3	I_4	I_5
INSTRUCTION DECODE LOGIC		I_1	I_2	I_3	I_4
INSTRUCTION EXECUTE LOGIC			I_1	I_2	I_3

execution unit starts executing the previous instruction. Thus up to three instructions can be processed at one time. Provided the pipe is kept full, the rate at which instructions are completed is one instruction per stage delay even though it may take multiple stages to process an instruction completely. There are three stages in our example. The stage delay is also known as the machine cycle time. Unfortunately, contention for common resources causes pipeline hazards, so that instructions cannot always flow through the pipe at the maximum rate. This can best be illustrated by a typical pipeline, shown in Figure 3. All instructions must first be fetched from memory on one of the pipe stages. This may require a long or short delay and may possibly require multiple cycles, depending on the type of memory. Then each instruction must be decoded, one instruction at a time if there is only one decode stage. After being decoded, each instruction is executed through the execution part of the pipeline. The exact path and number of stages required for complete execution is a function of the instruction type, complexity, and design of the pipeline. Since there are many different types of instructions, the pipeline can be filled to different levels of capacity, depending on what sequences of instructions have been entered into the decode stage.

If an instruction enters the decode stage on cycle 1 and proceeds down the path A1 B1 toward memory, it arrives at register R_{A1} at the end of cycle 2. Suppose a second instruction enters the pipe decode stage on cycle 2, immediately following instruction 1, proceeds toward memory down path D2, and arrives at register R_{D2} at the end of cycle 2. At the beginning of cycle 3, data in R_{D2} from instruction 2 are ready to enter stage B1 in conflict with data in R_{A1} from

Figure 3 Schematic of part of a typical pipeline showing execution paths of varying numbers of stages (controls are not shown)



instruction 1. Inasmuch as only data from one instruction can enter any pipeline stage during one cycle, obviously one has to be delayed. Furthermore, since instructions must be executed in sequential order, execution of instruction 2 must be held back; in fact, the decode stage is stopped for one cycle. This conflict requires that the third instruction enter the decode stage one cycle later than it otherwise would have done. Other similar conflicts along different paths can arise. Obviously, if we were given a string of instructions to be executed and were free to specify the order in which they would enter the decode stage, we could pick an order to avoid most, if not all, such conflicts. Under such conditions, the pipe could run at full efficiency. However, the sequential order of a program is important and must be maintained, so some conflicts are unavoidable. A conflict and the resulting delay arise because different instructions require different numbers of pipeline stages and can occur in sequences such that a contention for resources occurs. The architecture profoundly affects not only the number of pipeline stages but also the availability of these stages for processing different instruction types. A more complex architecture requires a more complex pipeline with potentially more conflicts. Whether or not, or to what extent, these conflicts actually occur depends on the instruction stream sequence. However, even this can be improved by a *smart compiler*, which is a compiler that knows something about the system pipeline and attempts to avoid certain types of conflict. For instance, a dependent load is an instruction immediately followed by another instruction that requires the data being loaded. The compiler may be able to schedule other instructions between the load and its use, so as to allow additional time for the memory access required by the load, if the program inherently allows it.

In addition to delays resulting from conflicts, there is another class of delays introduced by disruptions to the smooth pipeline flow. Such disruptions arise from any single instruction that requires partial or full completion before the next instruction can be

processed. Successful branches typically make up a significant part of this class, particularly in simple architectures. There are others with varying degrees of significance (e.g., storage-to-storage arithmetic operations, long moves, and load/store multiple). The

The architecture can affect the time to resolve the branch and generate the branch target address.

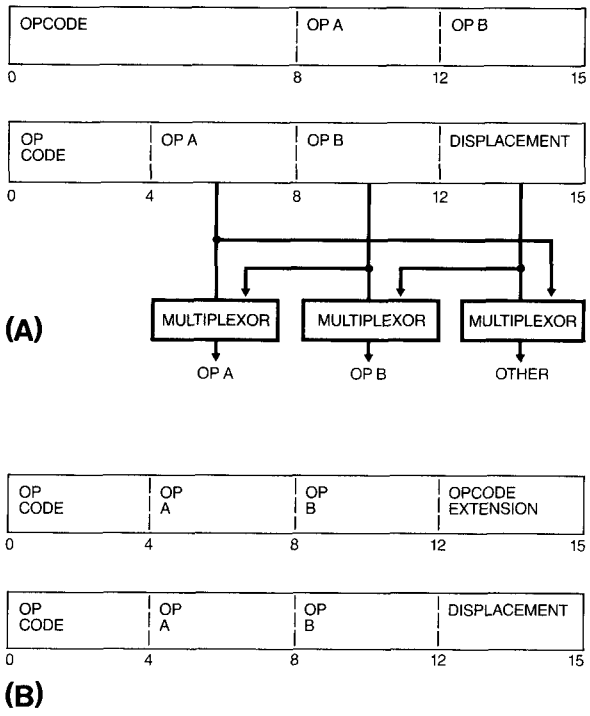
delays resulting from this class do not depend on any particular sequence of instructions. Rather, the mere occurrence of such an instruction results in a pipeline delay. For instance, whenever a conditional branch instruction is decoded, we do not know initially whether the branch is going to be successful and transfer to a new instruction until either the very end of the decode cycle or the beginning of the next cycle. Furthermore, the branch target address may have to be computed. In the meantime, in order to keep the pipeline full, the system will have started decoding the next sequential instruction. If the branch is successful, unless branch target prefetching along both possible paths has been made, a memory reference delay is required until the target instruction is obtained, after which decoding may start once again. The architecture can profoundly affect part of this delay, namely the time to resolve the branch and generate the branch target address. The remaining part, which is the memory access delay for the new instruction, is independent of architecture to a first approximation and depends principally on machine organization, although the word size and boundary alignment due to different instruction lengths have a small effect.

We can illustrate this disruption delay by reference to Figure 3. Suppose a successful branch of a particular type traverses the pipeline toward memory via path A1 B1 to register R_{B1} prior to memory array accessing. The branch is decoded on cycle 1, and the result resides in register R_{D1} at the end of the first cycle. Suppose we know at the very beginning of cycle 2 that the branch is successful, so that the sequential instruction following the branch cannot be decoded. Rather, a delay equal to that of two pipe stages, A1 plus B1, plus the full memory access time from register R_{B1} back to R_I is incurred until the next instruction is ready for decoding. The two pipe stage delays in this case might be address generation and virtual-to-real address translation. If the branch instruction is sufficiently complex, it may require two cycles of address generation. This would be determined mainly by the architecture. If address generation requires the addition of three numbers, say a base, an index, and a displacement (as in System/370), this would most likely require two cycles, with two numbers added on each cycle. This of course gives a very versatile addressing scheme but complicates the pipeline design, increases disruption length, and increases the potential for conflicts. These can all be improved by making the address generation simpler by adding only two numbers, base plus displacement, but obviously this is not as versatile. In any case, the architecture clearly can have a significant effect on one part of the total disruption delay. There are other types of disruptions that can be important, depending on the specific architecture.

Obviously, it is possible that a particular sequence of instructions can produce delays of both the conflict and disruption class simultaneously. For instance, if a load or store operation requires one additional pipeline stage compared to a branch, a load or store followed by a branch results in a conflict delay for the branch, plus the usual branch disruption delay.

In order to analyze such pipeline behavior, the general pipeline structure as well as instruction sequences and some method to simulate the dynamic data flow are required. A full simulation is complex and tedious, and does not easily provide insights into the various design tradeoffs. As a result, the first-pass analysis is simpler, using a quasi-static approach. Initially, the designer has some overall structure in mind for the pipeline in terms of the number of stages desired for each instruction type. From this, the major types of pipeline disruptions and conflicts can be deduced. Sample programs or estimates based on experience are then used to obtain the frequency of occurrence of such major potential delay factors as the average number of loads and stores (including dependent loads) and successful branches. Obviously, the number and types of conflicts or pipeline disruptions that lead to lost cycles depend on the specific instruction stream for any given pipeline. Since this sequence cannot be known in advance,

Figure 4 Generalized instruction format: (A) two formats having fields of different length and function; (B) two formats having fixed length and function



the actual performance of any system cannot be specified exactly. An upper bound on the performance can be obtained by an average analysis of the pipeline where certain types of less-frequent disruptions, conflicts, and combinations are neglected for a first-pass analysis, because they are often small effects. Obviously, if this program were to be processed, the results would be meaningless. However, this analysis gives a good first approximation to system behavior for typical average workloads and can be used to estimate the average number of cycles per instruction, C_I, possible for the architecture and assumed pipeline. If the results do not match the desired performance, various options and tradeoffs must be used to bring them into line; otherwise the goal must be revised. In the next section, we look at the implications of architecture on the value of C_I, what tradeoffs are provided, and how difficult it is to make them effective.

The performance of any machine can be characterized by two parameters, the cycle time (i.e., pipeline stage delay) and the number of cycles required to execute an instruction, C_I, as previously described. This latter number is greater than one because of the pipeline hazards already noted. The net execution rate of a processor can be improved either by lowering the cycle time or by decreasing the average cycles per instruction.

Implications of architecture on instruction decode and operand fetch

In this section we show how the architecture, in particular the length and format of instructions, influences the design of the instruction decode stage of the processor and indeed overall machine performance. Because an efficient pipelined machine should have compact pipe stages of roughly equal lengths, the complexity of the decode stage may determine the length and partitioning of the pipeline and the overall cycle time.

Instruction decode involves identifying the instruction format and extracting the appropriate fields from that format to determine the operand location(s), the destination location(s), and the operation to be performed. If the instruction set is complex and there are many different formats and lengths, it may be necessary to perform the decode process sequentially. For example, suppose there are two formats, as shown in Figure 4A. The *opcode* fields are of different lengths, which forces the operand fields to be misaligned. Until the opcode length is determined, the operand fields cannot be extracted. Even then the extraction probably involves three separate multiplexors. On the other hand, consider the two formats shown in Figure 4B. In this case, the A and B operand fields can be extracted immediately because they are always located in the same fields of all instructions. If the codes are chosen judiciously, a fast predecode can set one multiplexor to gate bits 12–15 to either the decode unit if it is an opcode extension or the execution unit if it is a displacement. Note that the predecode does not delay operand fetch, which is likely to be in the critical path.

Additional complexity is introduced if there are many different instruction lengths. If the machine includes an instruction prefetch stage before the decode stage, there is probably a first-in-first-out (FIFO) prefetch buffer between these two stages of the pipe. This buffer is probably organized into words of the same width as the processor bus. If instructions vary greatly in length and can cross multiple word boundaries or even start at any bit boundary, field extraction can be quite difficult. In a demand-paged

system, attention must also be paid to the case of an instruction crossing a page boundary, because this requires special detection and processing.

Other architectural features that significantly affect design are the inherent features for address formation. If memory addresses for a store operation are formed by adding the contents of one general-purpose register (GPR) to another, in addition to a displacement,[6] both these addresses and the data to be stored must be accessed simultaneously from the GPR array on the decode cycle. Hence the array must have three independent read ports, which are costly in terms of hardware, or multiple decode cycles are needed, which is slow. Furthermore, a three-operand adder is required if address generation is to be performed on one cycle. If a simpler addressing mode is provided, for which an address can be formed only by adding the contents of one GPR to a displacement contained in the instruction, then the GPR array need be capable of only two simultaneous read operations, which is a substantial saving. Although this addressing capability is more limited, it may be a good tradeoff for a microprocessor where space on the chip is at a premium. Alternative methods for achieving versatile addressing often use complex addressing modes, such as indirect addressing. However, these place additional demands on the memory bandwidth for accessing the operand address. Some architectures, such as the PDP-11 and Motorola MC68000, provide both simple and complex addressing capability and leave the choice of mode up to the user. Although this can avoid multiple GPR accesses and multiple additions, the complex modes require additional decode time and memory accesses that can significantly reduce performance if used extensively. Complex modes also require user awareness of the machine organization, if processing time is important.

A complex architecture can have other instruction types that place severe requirements on the GPR array accesses needed for decode. For instance, if an instruction requires two register accesses for a source, and likewise two for a target, then four GPR accesses may be needed. This either requires four separate read ports or multiple decode cycles, both of which are more complex and costly than a simpler instruction set. Such long operations are common in System/370 and represent one of the reasons why complex architectures tend to have a large value of C_I.

The architecture specifies the instruction format which, as previously noted, determines the complexity of instruction decoding. In addition, the instruction format also determines the ease with which the instruction set can be extended in future generations. If the opcode field is too small, it may be difficult to find opcodes for new instructions that preserve the field boundaries for simple decoding. It may be necessary to create extended opcodes, a contingency that can result in awkward, multiple-cycle decoding.

The instruction format determines the ease with which the instruction set can be extended.

Influence of architecture on memory subsystem

The term *memory* is used in a general sense. For systems with a *cache*, the memory is the cache. For systems without cache, memory is *main memory*. Although the architecture does not explicitly define the memory subsystem, it greatly influences the overall design.

Because microprocessor systems can have only a limited number of I/O pins available on the chips or modules, the width of buses and bandwidth between chips are similarly limited. However, the architecture of a microprocessor often evolves from consideration of the general types of instructions that should be provided, without much consideration for the memory bandwidth. In a microprocessor, this effect can be significant. In order to study these effects, the following analysis assumes a pipeline system with a general structure, as previously discussed and shown in Figure 3. With such a structure, the effects of memory bandwidth can most readily be understood by studying the average behavior of the instruction set. To do this, it is assumed that a large number of instructions I_E are executed. The types of instructions executed are examined, and the total number of memory references required can be calculated in terms of a few important parameters that reflect the memory system structure and instruction set com-

plexity. This analysis does not include all possible conflicts and local anomalies that result from instruction sequences; it provides only the lower or upper bounds on parameters and performance.

In this analysis, the most important parameters are the following:

I_{MA} = number of instructions (I) fetched per memory I-fetch cycle.
A_{IT} = total number of memory accesses or references for instructions and data per instruction executed.
θ = fractional percent of the instructions executed, I_E, that are data-reads from memory (exclusive of I-fetches).
δ = fractional percent of I_E that are data-writes to memory.
β = fractional percent of I_E that are successful branches to a new instruction (requires nonsequential I-fetch).
ρ = fractional percent of I_E that do not require a memory access, e.g., register-to-register or other operation.
C_I = average number of machine execution cycles per instruction required for the I_E instruction stream; ideal processor performance is $(C_I T_{CPU})^{-1}$ instructions per second.
T_{CPU} = machine pipeline cycle time in seconds.

Inasmuch as I/O pins are a critical resource that should be minimized, this analysis assumes that the bus width to memory is equal to the maximum instruction length in bits. Hence, if there is only one instruction length, it exactly equals the bus width. Obviously a smaller bus width increases the memory bandwidth requirement. For this analysis to remain valid, we need not make this assumption, but it simplifies certain limits such as the peak bandwidth requirement. From the definitions just given, the following relationship can be obtained:

$$\theta + \delta + \beta + \rho = 1 \tag{1}$$

The total number of memory requests required for the total instructions executed, I_E, is easily deduced as follows. Each of the I_E instructions must be accessed from memory. If an average of I_{MA} instructions are fetched for each memory word referenced, a total of I_E/I_{MA} memory references are required just for those executed instructions. Of these executed instructions, $\theta + \delta$ require additional references for data reads and writes. Hence the total number of memory references required for I_E is the following:

$$\textit{memory references} = \left(\frac{1}{I_{MA}} + \theta + \delta\right) I_E \tag{2}$$

Dividing both sides by I_E normalizes Equation 2 as follows:

$$\frac{\textit{average memory references}}{\textit{instruction}} = A_{IT} = \frac{1}{I_{MA}} + \theta + \delta \tag{3}$$

The average memory bandwidth required by the CPU in terms of accesses per machine cycle can be obtained by dividing Equation 3 by C_I, the average number of machine pipeline cycles per instruction, as follows:

$$\frac{\textit{memory cycles}}{\textit{machine cycle}} = \frac{\frac{1}{I_{MA}} + \theta + \delta}{C_I} \tag{4}$$

Equation 4 expresses the CPU-bound condition and tells us that for the given pipeline design executing an average program, on average, the instruction stream makes this many references to memory on each machine cycle. If the memory system cannot provide such requests fast enough, additional pipeline disruptions occur that retard the flow to a point that the memory can handle. In other words, additional disruption cycles are added to C_I to obtain an effective value C_{IE} that just matches the memory speed:

$$C_{IE} = C_I + \Delta \tag{5}$$

where
Δ = cycles per instruction penalty introduced by memory disruptions,
C_{IE} = net effective cycles per instruction for the total system (processor plus memory).

So the memory-bound relationship is as follows:

$$\frac{\textit{memory cycles}}{\textit{machine cycle}} = \frac{\frac{1}{I_{MA}} + \theta + \delta}{C_{IE}} = \frac{T_{CPU}}{T_m} \tag{6}$$

or

$$\mu_m = \frac{T_m}{T_{CPU}} = \frac{C_{IE}}{A_{IT}} \tag{7}$$

where μ_m is the normalized memory cycle time in machine cycles per memory cycle and T_m is the memory cycle delay time in seconds. T_m increases as

Conflicts arise whenever instructions are of maximum length with each requiring a memory reference.

μ_m increases. Substituting Equation 3 for A_{IT} gives the following expression for the normalized memory cycle time:

$$\mu_m = C_{IE}\left(\frac{1}{I_{MA}} + \theta + \delta\right)^{-1} \tag{8}$$

The system performance in millions of instructions per second (MIPS) is given simply as follows:

$$MIPS(\Delta \geq 0) = (10^6\ T_{CPU}C_{IE})^{-1} \tag{9}$$

$$= [10^6\ T_{CPU}(C_I + \Delta)]^{-1} \tag{10}$$

In Equation 9, C_{IE} is greater than or equal to C_I and Δ is positive. If C_{IE} is less than C_I so that Δ is negative, performance is given by Equation 10, with Δ set equal to 0. As μ_m increases, C_{IE} increases and moves further away from C_I, so that Δ increases. Since the obtainable system performance from Equation 10 is proportional to $1/(C_I + \Delta)$, it is desirable to keep μ_m as small as possible, but only small enough so that $C_I \equiv C_{IE}$. If C_{IE} becomes smaller than C_I, so that Δ becomes negative, the performance is limited entirely by the processor as given by Equation 10, with Δ set equal to 0. If C_{IE} is larger than C_I, the performance is limited by the memory. Thus a balanced design is desired.

To simplify terminology, we use the following expression:

$$R_d = \theta + \delta \tag{11}$$

In Equation 11, R_d presents all the memory references for loads and stores (appropriately adjusted for load/store multiple or similar instructions, if necessary).

We can further characterize the memory bandwidth requirements during periods of peak traffic. This occurs for short periods of time when there are pipeline conflicts for memory requests. These conflicts arise whenever instructions are of maximum length (instantaneous $I'_{MA} = 1$), with each instruction requiring a memory reference (instantaneous $R'_d = 1$), and with the pipeline percolating at one cycle per instruction (instantaneous $C'_I = 1$). We now substitute these values into Equation 6 and solve for maximum memory bandwidth or minimum cycle time as follows:

$$T_m(\min) = \frac{1}{2}\ T_{CPU} \tag{12}$$

While the concepts embodied in Equations 8 through 12 are relatively simple and based on average performance, they can provide insight into many design tradeoffs. Some specific cases are considered first and afterwards some problems and tradeoffs encountered for general ranges of A_{IT}, R_d, and C_I are discussed.

As a first example, consider the memory bandwidth requirements in terms of the architectural parameters just discussed. Our objective is to determine what design tradeoffs are available within the CPU versus those in the memory itself to improve the overall performance and/or to balance the design between processor and memory. On average the memory cycle time requirement is that specified in Equation 7. Let us look at what tradeoffs are possible as we let C_I/A_{IT} take on different values as determined by the architecture.

Case I: $C_I/A_{IT} > 1$

To clarify the example, we let $C_I/A_{IT} = 1.2$. From Equation 7, this case requires a memory of average cycle time $T_m = 1.2\ T_{CPU}$. This would be sufficient to handle the average memory traffic but not the peak traffic. The peak traffic requires a memory cycle time given by Equation 12, which is $T_m = 0.5\ T_{CPU}$. This condition is very difficult and expensive to achieve. On the other hand, a cycle time of $1.2\ T_{CPU}$ is awkward and inefficient. A value of $T_m = T_{CPU}$ is more reasonable. This would provide a memory that is a little faster than the average requirement but slower than the peak requirement. Can something be done to smooth out these fluctuations in memory traffic? Using a memory with $T_m = T_{CPU}$ with the architectural parameters just presented, there are $1/1.2 = 0.833$ memory references required per cycle,

but one such reference available each cycle, or $1 - 0.833 = 0.167$ free memory references each cycle. In other words, there is one full, free memory cycle every $1/0.167 = 6$ machine cycles. We can make use of this average one out of six free memory cycles to help reduce memory conflicts during peak periods by introducing an instruction prefetch buffer (IPB). This buffer is essentially a small stack of the R_I registers shown in Figure 3. Whenever there is an unused memory cycle and simultaneously an empty register in the IPB, the next sequential instruction is fetched. On average this prefetching occurs substantially ahead of the actual decoding, so there is usually an instruction available for decoding during periods of heavy data referencing to memory. Occasionally unavoidable conflicts occur when the IPB is empty simultaneously with some other data references to memory. If the I prefetching is delayed for such conflicts, this makes $\Delta > 0$ in Equation 5. Therefore, such memory conflicts with IPB empty merely add a small Δ to the average number of execution cycles per instruction.

An estimate of the size of the IPB and worst-case conflicts can be obtained as follows. Suppose, for Case I, that $I_{MA} = 1.5$ instructions fetched per memory I-fetch, on average. Every one out of six memory cycles, on average, is free for an I-fetch. Note that an instruction decode can take place during this I-fetch, so that six decode cycles take place for every free I-fetch. These six cycles consume on average $6/1.5 = 4$ registers of the IPB. Thus four registers in the buffer can maintain the average pipe flow.

During long peak periods, the IPB becomes empty. The pipeline start-up time typically has at least four free memory cycles during which the IPB can be filled. Thus the designer has some flexibility afforded by the architecture, which allows certain hidden hardware options for smoothing out the peak-versus-average memory demands that were also created by the architecture.

Case II: $C_I/A_{IT} = 1$

Suppose in Case I that the average memory bandwidth required by the architecture is unity, expressed by the following equation:

$$T_m = C_I/A_{IT} = 1\ T_{CPU}$$

Thus one memory reference for either instructions or data is required for every cycle. The minimum value remains as before, as follows:

$$T_m(\min) = \frac{1}{2}\ T_{CPU}$$

On average, there are no free memory cycles, so it does not seem likely that an IPB would be of much value. However, it might help to some extent, particularly if the typical memory bandwidth requirements consist of short bursts of maximum demand where simultaneous I-fetches and data references are required, followed by minimum demand with only I-fetches required at the cycle time specified by Equation 4, or $1/I_{MA}$ references per machine cycle. If I_{MA} is greater than one, an instruction prefetch buffer can help smooth out some of the memory demands. The extent to which this is effective is highly dependent on the actual instruction stream, but it is not as effective as in Case I. The situation most likely to occur is that the IPB is effective part of the time, but for the remaining time—during heavy traffic to memory—the conflicts arising from limited bandwidth insert additional unproductive Δ cycles in Equation 10, causing the effective C_{IE} to be larger than C_I during these periods. This momentarily makes C_{IE}/A_{IT} larger than one, which is the situation of Case I previously presented. Hence the limited memory bandwidth adjusts the demand upon it to just match its capability. During periods of light data referencing C_{IE} can equal C_I and some instructions can be profitably prefetched.

The major problem is to increase the effective memory bandwidth.

It can be seen from Case II that the designer is always faced with limited memory bandwidth, and any attempt to improve the processor pipeline so as to decrease C_{IE} is an attack on the wrong problem. The designer has to accept the fact that, for these system parameters, the average effective C_{IE} will most likely be substantially above the target value of C_I.

Case III: $C_I/A_{IT} < 1$

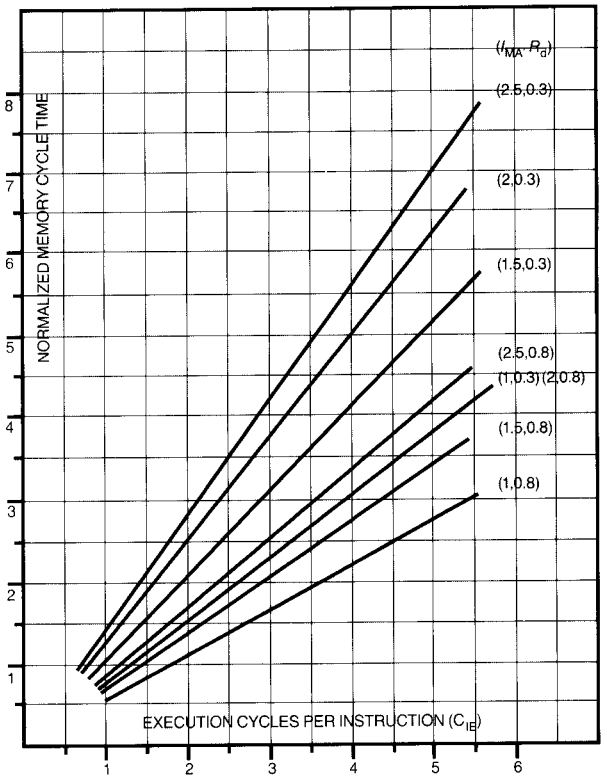
Suppose $C_I/A_{IT} = 0.83 = T_m/T_{CPU}$ so that an average 1.2 memory references are required every cycle. Be-

cause the maximum number of references remains the same at two references per cycle, and because the design point chosen was originally $T_m = T_{CPU}$, there are not only no free memory cycles on average, but there is also insufficient memory bandwidth to sustain the pipeline. Hence an IPB is of little value. The major problem is to increase the effective memory bandwidth. This can be achieved either by increasing the value of I_{MA} or increasing the memory bus width (or both). If the bus width is not changed, the first method requires a change in the architecture, which may or may not be an option. If the bus width is changed, there are several options, but all with inherent difficulties. For a medium or large system under similar conditions, the designer might consider using a separate instruction and data cache, each with $\mu_m = 1$ to achieve the required bandwidth. This may or may not be a reasonable approach, depending on the actual values of I_{MA}, R_d, and C_I. In certain cases, an example of which is given later in this paper, a very unbalanced design results, where the I-cache is fully utilized and the D-cache is seldom used. Nevertheless it may be the only alternative.

For a microprocessor, there may be too few pin I/Os to allow separate I- and D-caches. Thus the designer has little choice. The small value of C_I must be increased, either by design or because the memory system will make Δ of Equation 10 large enough to balance the throughput. The architecture obviously requires too many memory references to support a small value of C_I. The designer should redesign the pipeline and not attempt an impossible task.

The three cases just given have shown that, for a certain range of parameters, the designer has the option of introducing hardware into the CPU to help alleviate peak traffic congestion to the memory. However, outside this range, the problem becomes more and more simply a memory bandwidth limit that cannot be improved by the CPU alone. The designer should be careful to attack the right problem. For instance, there are clever schemes for improving C_I by reducing pipeline disruption length on certain types of instructions. A branch history table and smart decoder can predict and fetch the successful branch targets ahead of time. Note, however, that this does not usually reduce the number of required memory references. Since C_I may actually decrease, the memory bandwidth requirement becomes worse (see Equation 6), because the instructions are processed faster, on average. Thus, when bandwidth is the limiting factor, introducing special hardware into the CPU to improve C_I may be fruitless.

Figure 5 Normalized memory performance versus processor performance for various values of instruction fetch per memory access and data reference per instruction executed



General tradeoffs

Now that some specific examples have been analyzed, let us focus on architectures and resulting tradeoffs in a more general way. To do this, we make use of the normalized memory cycle time μ_m as a function of the effective numbers of cycles per instruction C_{IE}, as obtained from Equation 8 and shown in Figure 5 for various values of I_{MA} and R_d.

As a first example, suppose we have an instruction set that results in significant memory references for data, giving R_d of 0.8 references per instruction executed. If our memory is a cache with $\mu_m = 1$, Figure 5 shows that there is a significant relationship between I_{MA} (instructions fetched per memory access) and C_{IE} (effective execution cycles per instruction). If we try to drive C_{IE} down toward one, I_{MA} must increase. For $C_{IE} = 1.2$ cycles per instruction at $\mu_m = 1$, I_{MA} must have at least the value 2. However, complex instructions tend to push C_{IE} to

higher values, which eases the requirement on I_{MA}. The first conclusion is that, with an architecture that generates large R_d, it is difficult to push C_{IE} toward one cycle per instruction. This is indeed the case in System/370, even in the large mainframe systems. At $C_{IE} = 1$, $R_d = 0.8$, and $I_{MA} = 2$ instructions per memory access, the cache must have an average cycle time of about 0.75 T_{CPU}, i.e., faster than the CPU. Allowing for peak traffic requires the cache to have twice the speed of the processor. This can be achieved with either a dual-ported cache or an extremely fast cache. The former may be unreasonable for a microprocessor, due to the cost in terms of chip count, I/O pins, and bus wires. Currently, if the processor and arrays are fabricated from similar technologies, a fast cache is difficult to achieve without special array design. Hence a microprocessor with such an architecture is more suitable for operation at C_{IE} substantially above the value 1. Two cycles per instruction or greater gives a more balanced design.

If the processor and arrays are fabricated from similar technologies, a fast cache is difficult to achieve.

For an architecture with R_d in the range of 0.3, Figure 5 indicates that it is possible to push C_{IE} toward one cycle per instruction, with memory of cycle time 1 T_{CPU} or even slightly slower if we have I_{MA} of 1.5 instructions per access or greater.

If it is desirable to strive for an architecture with a memory cycle time slower than $\mu_m = 1$, then Figure 5 indicates that C_{IE} must increase. Increasing the value of I_{MA} helps, but it is extremely difficult to push this parameter above 2 to 2.5, even for complex architecture. Word boundary alignment plus instruction complexity keeps the effective value small. If we can increase I_{MA} to 2, while R_d remains at 0.3, and memory cycle time is 3 T_{CPU}, the pipeline then requires a minimum of about 2.5 cycles per instruction, at best.

Note that for an architecture with R_d in the range of 0.3 and $I_{MA} = 1$, the memory access conflicts occur for the average case rather than for just peak periods. Therefore, memory access conflicts cannot be smoothed out with an instruction prefetch buffer. This can be greatly improved by the use of separate instruction and data caches (i.e., I-cache and D-cache), as previously pointed out. However, although this structure is adequate, it is not well balanced, because the I-cache will essentially be accessed on every cycle, while the D-cache will be used only 30 percent of the time. For such a case, a more ideal approach may be to have one memory with two completely separate ports, in and out. This would not only fulfill the average bandwidth requirements, but it would also handle most of the peak conflict traffic as well, since two instructions or two data references could be made simultaneously. However, both approaches are rather costly for a microprocessor and are not considered further here.

If the designer has more than adequate memory bandwidth available, the CPU-bound value of C_I, as in Equation 4, should be the design focus. This can be attacked in many ways such as branch history tables mentioned previously, and more parallel hardware. Many issues and tradeoffs available for reducing C_I have purposely been avoided since they are a separate subject. If one knows or has a target value for C_I desired, the analysis provided here can show the range of parameters available for a balanced design.

Cache reload

In any microprocessor architecture, simple or complex, if a cache is used to improve the bandwidth on the front end between CPU and memory, there still remains a serious bandwidth problem on the back end between main memory and cache during a block reload. A high instantaneous memory bandwidth is needed only occasionally for the reload, but it is a major factor in performance. It is clear that if the cache reloading must take place at a bandwidth that is no faster than normal memory accessing, the only cycles saved by a cache are for any words that are reused, i.e., words accessed more than once. The amount of reuse as well as the effect of reload time can be conveniently represented by the finite cache penalty (FCP). This is used in conjunction with the parameter C_{IE} as follows. The use of C_{IE} does not include the effects of cache misses and reload time. In essence, C_{IE} represents the performance that could be obtained with a very large (say infinite) cache so

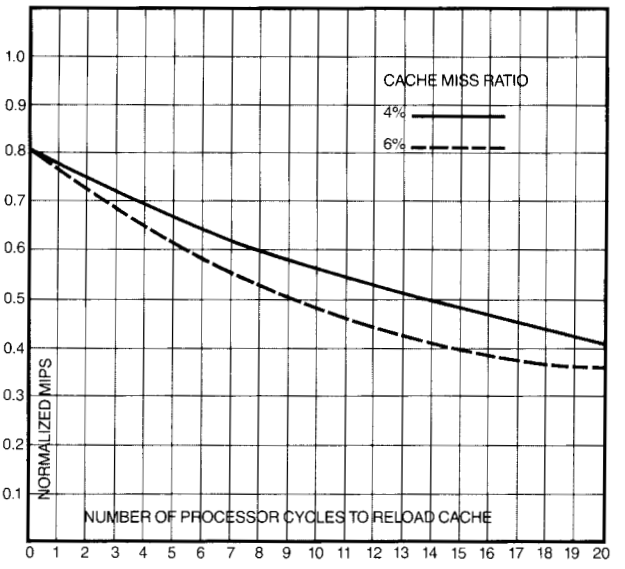
that no misses occur. However, a miss is a pipeline disruption that merely adds an additional term to the value of Δ in Equation 10. This additional term is the finite cache penalty given by the following equation:

$$FCP = A_{IT} MR \frac{T_{Reload}}{T_{CPU}} \quad (13)$$

where MR is the cache miss ratio and T_{Reload}/T_{CPU} is the number of machine cycles needed for reloading a miss. The value of A_{IT} is set by the architecture. The cache miss ratio is influenced slightly by the architecture but mainly by the size of the cache and the blocks that are replaced. Smaller blocks tend to give smaller hit ratios. Also, the size of directory translation hardware needed for the cache increases linearly with the number of blocks. Furthermore, the silicon area available for this directory is very limited, so both these effects favor larger block sizes. Unfortunately, larger blocks take longer to reload and the relationships are nonlinear. The net result for a microprocessor is that the cache and block size are fixed by the available chip area. The designer may be able to change this, but usually only very little within a given technology. Thus the only parameter available for tradeoffs is the reload time. One example of the effects of reload for a high-performance microprocessor with a moderately simple architecture is shown in Figure 6. The x axis is the average number of machine cycles required for reload and must include the time to write-back modified pages to main memory, if appropriate. If nothing special is done to improve reload time, a typical range of 15 to 30 cycles or more might be imposed by the memory system. It can be seen that the performance degradation compared to an infinite cache (zero reload time) can be more than 50 percent. It is desirable to reduce the reload time to a value between 5 and 10 cycles, if at all possible. The only reasonable way to improve this is to load the cache from a main memory system, which—after the initial access time penalty—can stream words in and out either at one word per machine cycle or faster. Streaming at one word every one-half cycle is very desirable but is costly. The clock, storage controller, and cache array chips must be able to support this cycle time.

An attempt to improve the reload time by the use of a multiple word bus width on the back side, coupled with a complex block mapping, is very difficult and costly if at all possible. The total number of data I/O pins to the cache is most likely fixed at the CPU word length, so the maximum bus width is fixed. The only parameter available for improvement is the cycle time.

Figure 6 Normalized millions of instructions per second (MIPS) versus the total number of cycles to reload the cache for two values of cache miss ratio



Memory move operations

Within the CPU, nearly all architectures must and do provide for the manipulation of individual characters or bytes. Since the dawn of the computing era, it has been desirable to do likewise within the memory. However, the bandwidth requirements of memory demand that more than one byte be accessed at one time. Thus memory systems access words, which are typically 2, 4, or 8 bytes, depending on the system. Therefore, words must be accessed on word boundaries, which makes the very common memory MOVE operation quite complex and time-consuming for the general case. In a MOVE operation, the contents of a logical portion of memory are moved from one physical location to another. Such operations can start the move from any byte boundary and place it in a new location starting at any byte boundary. Sometimes the two areas even overlap. The MOVE is such a common operation in both large and small systems that special hardware and instructions are often used to handle it (e.g., Move Character Long in System/370). Most systems do not attempt to solve this problem at its root, but rather attempt to use the CPU to avoid the lack of a byte-addressable memory. For instance, the 801 architecture[1] includes

special rotate and store instructions to improve MOVE operations. Although the process of moving on arbitrary byte boundaries can be aided by the use of special processor instructions, this complicates the processor design, requires extra hardware, and still is not as fast or efficient as a byte boundary addressable memory. However, it has the distinct advantage of being able to use ordinary memory chips, thereby keeping the cost low. In order to make the general move operation simple and efficient, it is desirable to implement a memory system that has the appearance of byte addressability. This can be achieved either by means of a sophisticated controller chip or by enhanced functions added to the memory chips. Such enhanced memory chips may begin to become economically attractive as the density and cost of memory chips continue to decrease. Very simple MOVE instructions may then be likely to appear in the architectures of microprocessors.

Memory management in any high-performance system can be greatly aided by the use of virtual memory.

Virtual memory

The memory management in any high-performance system can be greatly aided by the use of virtual memory. When properly implemented, this concept not only optimizes the use of real memory but also provides the user with a large address space. Thus the user can write programs as though a larger main memory is available than there actually is.

Unfortunately, virtual memory is not free. The most significant design consideration is the translation look-aside buffer (TLB), which must be added to prevent deterioration of system performance due to the translation process. The TLB is a partial directory that provides virtual-to-real address translation of some of the more recently used pages. Each entry to the directory provides translation for one page. The size of the entry is fixed by other parameters, as we shall see, but the number of entries is a variable. More entries give better performance but require more hardware.

A very approximate estimate of the number of page entries required in the TLB can be made, but this requires a new statistic. It is observed over many different programs and operating systems that with a 4K-byte page size and memory size between 0.5 and 1M bytes per MIPS of performance, 20K to 60K instructions are executed between page faults. We represent this variable, instructions between page faults, by I_{pf}. Knowing the value of this variable, the approximate number of pages touched between page faults can be estimated by multiplying I_{pf} by A_{IT} to obtain the following number of memory references between page faults:

$$\textit{references between faults} = A_{pf} = A_{IT} I_{pf} \tag{14}$$

Assuming that every word in memory is referenced once and only once, the total number of pages referenced, N_p, is given by the following equation:

$$N_p = \frac{A_{IT} I_{pf}}{W_p} \tag{15}$$

Here, W_p is the number of words or equivalent number of instructions per page. In typical cases, A_{IT} varies from approximately 1 to 2, and I_{pf} varies between 20K and 60K. Using 4 bytes per word, W_p is typically 4K/4 or 1K words per page. Substituting these into Equation 15 gives N_p as approximately 20 to 120 pages referenced between page faults. If we wish to provide a TLB hardware assist with translation to all pages referenced between page faults, there must be one entry for each of the 20 to 120 possible pages. This keeps the hit ratio to the TLB very large, which is desirable. For various reasons, it is not unusual to have double this number of entries if the cost allows it.

Each entry in the TLB must hold most of the virtual page address, which is typically 23 bits, plus the real main memory address, which is typically 12 bits (for a maximum 16M-byte main memory with 4K-byte pages). The TLB must also hold a valid bit, a memory protect key, and other control bits, which together amount to about six bits. The minimum number of bits per entry is thus in the range of about forty. Thus the TLB can range from about 1K to 8K bits. This is a substantial piece of hardware that is necessary to maintain the performance. In addition to the TLB itself, in order to maintain performance it is

desirable that whenever a miss occurs to the TLB, the reload of that entry be done with special hardware control for speed. Otherwise, a software program that uses the available general hardware is required, which is very slow.

Complex instructions have a more complex pipeline, particularly more parallel paths.

In order to speed up the translation process and page fault detection, as well as simplify the pipeline control, the TLB should be located on the processor chip if at all possible. However, this may limit the number of entries and reduce performance. The effects of TLB reload time can be analyzed in a manner analogous to that of Equation 13 for cache reload. In essence, TLB reload time adds another small value to Δ in Equation 10, but that is not included in this work.

Early attempts to provide a large address space and facilitate memory management in microsystems have included the use of instructions with different addressing modes, such as extended addressing using multiple words and indirect addressing, mentioned previously. However, severe fragmentation and wasted real main memory space can still be encountered. The inclusion of virtual memory in the architecture not only solves the fragmentation and address space problem, but it also eliminates the need for these complex addressing modes. Thus it is likely for virtual memory to become more common in future microprocessor architectures.

Concluding remarks

This paper has shown how architecture influences two major components of a microprocessor, the instruction unit and memory subsystem. Architecture directly affects the instruction decoding processes. A complex instruction set can require a long decode time, with complex field extraction and alignment, multiple accesses to the general-purpose registers or memory, and a difficult address generation cycle. Furthermore, complex instructions have a more complex pipeline, particularly more parallel paths, many of which can have different lengths. This tends to increase pipeline conflicts and disruptions. Maintaining the sequentiality of execution in a complex pipelined system requires additional priority controls, as well as additional staging registers for the pipeline flow. In other words, the pipeline hardware becomes more complex, consumes more silicon area, and most likely produces a slower machine cycle. A simple architecture can significantly minimize all these difficulties, and simplicity is desirable in a microprocessor where cost is of prime importance.

Although less evident at the architectural level, the instruction set has a significant effect on the memory subsystem. It has been shown how a few key parameters that characterize the architecture can be used to analyze the required memory bandwidth and show the designer whether the system performance is CPU- or memory-limited. Achieving a small value for cycles per instruction is generally easier for a simple architecture. Over a certain range of parameters, the system balance is less likely to be limited by the memory bandwidth. Hence various options for reducing the CPU cycles per instruction become available to the designer. Also there are additional CPU options for smoothing out the peaks and valleys in memory traffic. Outside this range, memory bandwidth is a fundamental limit on system performance, even for simple architectures. For such cases, an architecture, which has a target of achieving a small value for cycles per instruction, should strive either to avoid instructions that have multiple accesses to memory or to use instructions of more than one length. The memory bandwidth requirements can become so severe that a cache may be necessary to achieve the desired performance for a given architecture. However, adequate reload capability during a cache miss places new requirements on the memory bandwidth. This pushes the design toward cost-performance considerations rather than the original objective of low cost. The designer of a microsystem is constantly faced with these tradeoffs, which provide many opportunities for innovations as well as radically new approaches.

Acknowledgment

The authors thank Jean Lesser for providing the data for Figure 6.

Cited references

1. G. Radin, "The 801 Minicomputer," *IBM Journal of Research and Development* **27,** No. 3, 237–246 (1983).
2. J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI processor architecture," *Proceedings,* CMU Conference on VLSI Systems and Computations, Computer Science Press, Inc., Rockville, MD (1981), pp. 337–346.
3. G. S. Rao and P. L. Rosenfeld, "Integration of machine organization and control program design—review and direction," *IBM Journal of Research and Development* **27,** No. 3, 247–256 (1983).
4. *IBM System/38 Technical Developments,* GS80-0237 or ISBN 0-933186-00-2, IBM Corporation (1978); available through IBM branch offices.
5. *iAPX 432 General Data Processor Architecture Reference Manual,* Order No. 171860-001, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051 (1981).
6. *IBM System/370 Extended Architecture,* SA22-7085-0, IBM Corporation; available through IBM branch offices.
7. E. E. Davidson, "Electrical design of a high speed computer package," *IBM Journal of Research and Development* **26,** No. 3, 349–361 (1982).
8. N. Raver, "FET off-chip drivers and package disturbs," *Proceedings,* IEEE Custom Integrated Circuits Conference, Rochester, NY (May 1984).
9. P. W. Agnew and A. S. Kellerman, "Microprocessor implementation of mainframe processors by means of architecture partitioning," *IBM Journal of Research and Development* **26,** No. 4, 401–412 (1982).
10. W. W. Lattin, J. A. Bayliss, D. L. Budde, S. R. Cooley, G. W. Cox, A. L. Goodman, J. R. Rattner, W. S. Richardson, and R. C. Swanson, "A 32b VLSI micromainframe computer system," *Proceedings,* IEEE International Solid State Circuits Conference, New York (1981), pp. 110–111.
11. P. M. Kogge, *The Architecture of Pipelined Computers,* McGraw-Hill Book Company, Inc., New York (1981).

General references

R. E. Matick, *Computer Storage Systems and Technology,* John Wiley & Sons, Inc., New York (1977).

Reprint Order No. G321-5224.

Richard E. Matick *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.*

Dr. Matick received his B.S., M.S., and Ph.D. degrees in electrical engineering from Carnegie-Mellon University in 1955, 1956, and 1958. He joined IBM in October 1958 and worked in the areas of thin magnetic films, memories, and ferroelectrics. As manager of the Magnetic Film Memory Group from 1962 to 1964, he received an Outstanding Invention Award for the invention and development of the thick film read-only memory. He spent six months at IBM Hursley, England, developing this read-only memory for System/360 applications. Dr. Matick joined the technical staff of the IBM Director of Research in 1965 and remained until 1972, serving in various staff positions that included coordinator of Research Division plans and Technical Assistant to the Director of Research. He took a sabbatical in 1972 to teach at the University of Colorado and at IBM Boulder. During the summer of 1973 he taught at Stanford University. He is currently working in the areas of VLSI functional memory chip and microprocessor design. Dr. Matick is the author of the books *Transmission Lines for Digital and Communication Networks* and *Computer Storage Systems and Technology.* He is also the author of chapters on memories in *Introduction to Computer Architecture* and *Electronics Engineers' Handbook, Second Edition.* Dr. Matick has written numerous papers on magnetic devices and memories, semiconductor circuits, memory and logic, as well as virtual memory chips and systems. He is the holder of numerous patents and patent publications, and he is a member of the Institute of Electrical and Electronics Engineers and Eta Kappa Nu.

Daniel T. Ling *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.*

Dr. Ling joined IBM at the Thomas J. Watson Research Center in 1979. He is currently manager of exploratory VLSI design and is involved in the design of a high-performance microprocessor. His interests are in the areas of microprocessor and display design, as well as VLSI design tools and methods. Dr. Ling received his B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University. While at Stanford, Dr. Ling was a Fannie and John Hertz Foundation Fellow. He is a member of the Institute of Electrical and Electronics Engineers, the American Physical Society, and Tau Beta Pi.