must be at address 0). It is up to the microassembler to place each microinstruction at a suitable address and link them together in short sequences using the NEXT_ADDRESS field. Each sequence starts at the address corresponding to the numerical value of the IJVM opcode it interprets (e.g., POP starts at 0x57), but the rest of the sequence can be anywhere in the control store, and not necessarily at consecutive addresses.

Now consider the IJVM IADD instruction. The microinstruction branched to by the main loop is the one labeled iadd1. This instruction starts the work specific to IADD:

1. TOS is already present, but the next-to-top word of the stack must be fetched from memory.
2. TOS must be added to the next-to-top word fetched from memory.
3. The result, which is to be pushed on the stack, must be stored back into memory, as well as stored in the TOS register.

In order to fetch the operand from memory, it is necessary to decrement the stack pointer and write it into MAR. Note that, conveniently, this address is also the address that will be used for the subsequent write. Furthermore, since this location will be the new top of stack, SP should be assigned this value. Therefore, a single operation can determine the new value of SP and MAR, decrement SP, and write it into both registers.

These things are accomplished in the first cycle, iadd1, and the read operation is initiated. In addition, MPC gets the value from iadd1's NEXT_ADDRESS field, which is the address of iadd2, wherever it may be. Then iadd2 is read from the control store. During the second cycle, while waiting for the operand to be read in from memory, we copy the top word of the stack from TOS into H, where it will be available for the addition when the read completes.

At the beginning of the third cycle, iadd3, MDR contains the addend fetched from memory. In this cycle it is added to the contents of H, and the result is stored back to MDR, as well as back into TOS. A write operation is also initiated, storing the new top-of-stack word back into memory. In this cycle the goto has the effect of assigning the address of Main1 to MPC, returning us to the starting point for the execution of the next instruction.

If the subsequent IJVM opcode, now contained in MBR, is 0x64 (ISUB), almost exactly the same sequence of events occurs again. After Main1 is executed, control is transferred to the microinstruction at 0x64 (isub1). This microinstruction is followed by isub2 and isub3, and then Main1 again. The only difference between this sequence and the previous one is that in isub3, the contents of H are subtracted from MDR rather than added to it.

The interpretation of IAND is almost identical to that of IADD and ISUB, except that the two top words of the stack are bitwise ANDed together instead of being added or subtracted. Something similar happens for IOR.

If the IJVC opcode is DUP, POP, or SWAP, the stack must be adjusted. The DUP instruction simply replicates the top word of the stack. Since the value of this word is already stored in TOS, the operation is as simple as incrementing SP to point to the new location and storing TOS to that location. The POP instruction is almost as simple, just decrementing SP to discard the top word on the stack. However, in order to maintain the top word in TOS it is now necessary to read the new top word in from memory and write it into TOS. Finally, the SWAP instruction involves swapping the values in two memory locations: the top two words on the stack. This is made somewhat easier by the fact that TOS already contains one of those values, so it need not be read from memory. This instruction will be discussed in more detail later.

The BIPUSH instruction is a little more complicated because the opcode is followed by a single byte, as shown in Fig. 4-18. The byte is to be interpreted as a signed integer. This byte, which has already been fetched into MBR in Main1, must be sign-extended to 32 bits and pushed onto the top of the stack. This sequence, therefore, must sign-extend the byte in MBR to 32 bits, and copy it to MDR. Finally, SP is incremented and copied to MAR, permitting the operand to be written out to the top of stack. Along the way, this operand must also be copied to TOS. Note that before returning to the main program, PC must be incremented and a fetch operation started so that the next opcode will be available in Main1.

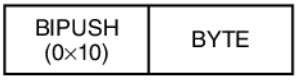


Figure 4-18. The BIPUSH instruction format.

Next consider the ILOAD instruction. ILOAD also has a byte following the opcode, as shown in Fig. 4-19(a), but this byte is an (unsigned) index to identify the word in the local variable space that is to be pushed onto the stack. Since there is only 1 byte, only $2^8 = 256$ words can be distinguished, namely, the first 256 words in the local variable space. The ILOAD instruction requires both a read (to obtain the word) and a write (to push it onto the top of the stack). In order to determine the address for reading, however, the offset, contained in MBR, must be added to the contents of LV. Since both MBR and LV can be accessed only through the B bus, first LV is copied into H (in iload1), then MBR is added. The result of this addition is copied into MAR and a read initiated (in iload2).

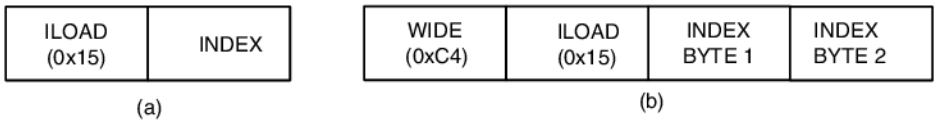


Figure 4-19. (a) ILOAD with a 1-byte index. (b) WIDE ILOAD with a 2-byte index.

However, the use of MBR for an index is slightly different than in BIPUSH, where it was sign-extended. In the case of an index, the offset is always positive, so the byte offset must be interpreted as an unsigned integer, unlike in BIPUSH, where it was interpreted as a signed 8-bit integer. The interface from MBR to the B bus is carefully designed to make both operations possible. In the case of BIPUSH (signed 8-bit integer), the proper operation is sign-extension, that is, the leftmost bit in the 1-byte MBR is copied into the upper 24 bits on the B bus. In the case of ILOAD (unsigned 8-bit integer), the proper operation is zero-fill. Here the upper 24 bits of the B bus are simply supplied with zeros. These two operations are distinguished by separate signals indicating which operation should be performed (see Fig. 4-6). In the microcode, this is indicated by MBR (sign-extended, as in BIPUSH 3) or MBRU (unsigned, as in iload2).

While waiting for memory to supply the operand (in iload3), SP is incremented to contain the value for storing the result, the new top of stack. This value is also copied to MAR in preparation for writing the operand out to the top of stack. PC again must be incremented to fetch the next opcode (in iload4). Finally, MDR is copied to TOS to reflect the new top of stack (in iload5).

ISTORE is the inverse operation of ILOAD, that is, a word is removed from the top of the stack and stored at the location specified by the sum of LV and the index contained in the instruction. It uses the same format as ILOAD, shown in Fig. 4-19(a), except with opcode 0x36 instead of 0x15. This instruction is somewhat different than might be expected because the top word on the stack is already known (in TOS), so it can be stored away immediately. However, the new top-of-stack word must be read from memory. So both a read and a write are required, but they can be performed in any order (or even in parallel, if that were possible).

Both ILOAD and ISTORE are restricted in that they can access only the first 256 local variables. While for most programs this may be all the local variable space needed, it is, of course, necessary to be able to access a variable wherever it is located in the local variable space. To achieve this, IJVM uses the same mechanism employed in JVM to achieve this: a special opcode WIDE, known as a **prefix byte**, followed by the ILOAD or ISTORE opcode. When this sequence occurs, the definitions of ILOAD and ISTORE are modified, with a 16-bit index following the opcode rather than an 8-bit index, as shown in Fig. 4-19(b).

WIDE is decoded in the usual way, leading to a branch to wide1 which handles the WIDE opcode. Although the opcode to widen is already available in MBR, wide1 fetches the first byte after the opcode, because the microprogram logic always expects that to be there. Then a second multiway branch is done in wide2, this time using the byte following WIDE for dispatching. However, since WIDE ILOAD requires different microcode than ILOAD, and WIDE ISTORE requires different microcode than ISTORE, etc., the second multiway branch cannot just use the opcode as the target address, the way Main1 does.

Instead, wide2 ORs 0x100 with the opcode while putting it into MPC. As a result, the interpretation of WIDE ILOAD starts at 0x115 (instead of 0x15), the

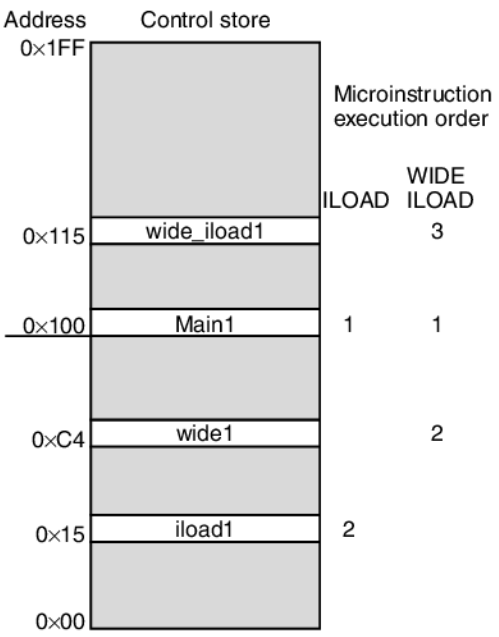


Figure 4-20. The initial microinstruction sequence for ILOAD and WIDE ILOAD. The addresses are examples.

interpretation of WIDE ISTORE starts at 0x136 (instead of 0x36), and so on. In this way, every WIDE opcode starts at an address 256 (i.e., 0x100) words higher in the control store higher than the corresponding regular opcode. The initial sequence of microinstructions for both ILOAD and WIDE ILOAD is shown in Fig. 4-20.

Once the code is reached for implementing WIDE ILOAD (0x115), the code differs from normal ILOAD only in that the index must be constructed by concatenating 2 index bytes instead of simply sign-extending a single byte. The concatenation and subsequent addition must be accomplished in stages, first copying INDEX BYTE 1 into H shifted left by 8 bits. Since the index is an unsigned integer, MBR is zero-extended using MBRU. Now the second byte of the index is added (the addition operation is identical to concatenation since the low-order byte of H is now zero, guaranteeing that there will be no carry between the bytes), with the result again stored in H. From here on, the operation can proceed exactly as if it were a standard ILOAD. Rather than duplicate the final instructions of ILOAD (iload3 to iload5), we simply branch from wide_iload4 to iload3. Note, however, that PC must be incremented twice during the execution of the instruction in order to leave it pointing to the next opcode. The ILOAD instruction increments it once; the WIDE_ILOAD sequence also increments it once.

The same situation occurs for WIDE_ISTORE: after the first four microinstructions are executed (wide_istore1 to wide_istore4), the sequence is the same as the

sequence for ISTORE after the first two instructions, so wide_istore4 branches to istore3.

Our next example is a LDC_W instruction. This opcode is different from ILOAD in two ways. First, it has a 16-bit unsigned offset (like the wide version of ILOAD). Second, it is indexed off CPP rather than LV, since its function is to read from the constant pool rather than the local variable frame. (Actually, there is a short form of LDC_W (LDC), but we did not include it in IJVM, since the long form incorporates all possible variations of the short form, but takes 3 bytes instead of 2.)

The IINC instruction is the only IJVM instruction other than ISTORE that can modify a local variable. It does so by including two operands, each 1 byte long, as shown in Fig. 4-21.

Figure 4-21. The IINC instruction has two different operand fields.

The IINC instruction uses INDEX to specify the offset from the beginning of the local variable frame. It reads that variable, incrementing it by CONST, a value contained in the instruction, and stores it back in the same location. Note that this instruction can increment by a negative amount, that is, CONST is a signed 8-bit constant, in the range −128 to +127. The full JVM includes a wide version of IINC where each operand is 2 bytes long.

We now come to the first IJVM branch instruction: GOTO. The sole function of this instruction is to change the value of PC, so that the next IJVM instruction executed is the one at the address computed by adding the (signed) 16-bit offset to the address of the branch opcode. A complication here is that the offset is relative to the value that PC had at the start of the instruction decoding, not the value it has after the 2 offset bytes have been fetched.

To make this point clear, in Fig. 4-22(a) we see the situation at the start of Main1. The opcode is already in MBR, but PC has not yet been incremented. In Fig. 4-22(b) we see the situation at the start of goto1. By now PC has been incremented but the first offset byte has not yet been fetched into MBR. One microinstruction later we have Fig. 4-22(c), in which the old PC, which points to the opcode, has been saved in OPC and the first offset byte is in MBR. This value is needed because the offset of the IJVM GOTO instruction is relative to it, not to the current value of PC. In fact, this is the reason we needed the OPC register in the first place.

The microinstruction at goto2 starts the fetch of the second offset byte, leading to Fig. 4-22(d) at the start of goto3. After the first offset byte has been shifted left 8 bits and copied to H, we arrive at goto4 and Fig. 4-22(e). Now we have the first offset byte shifted left in H, the second offset byte in MBR, and the base in OPC. By constructing the full 16-bit offset in H and then adding it to the base, we get the

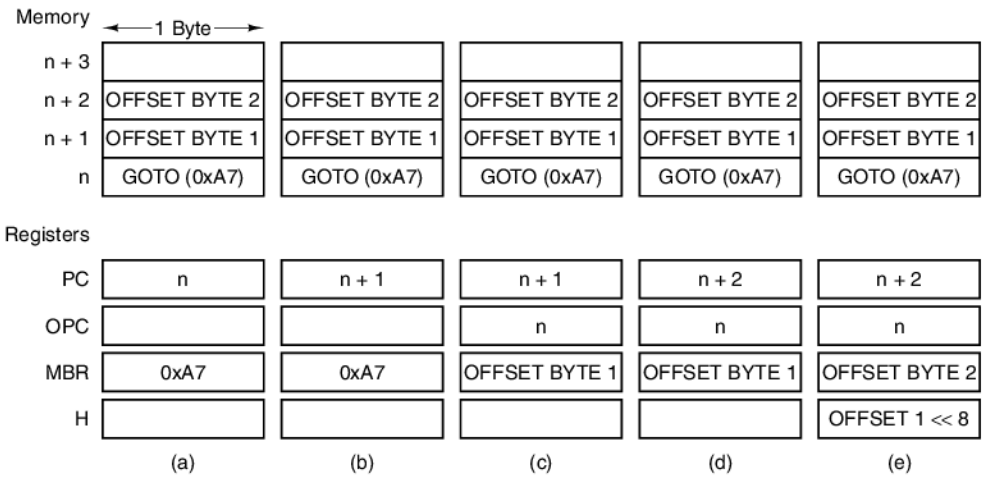


Figure 4-22. The situation at the start of various microinstructions. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

new address to put in PC, in goto5. Note carefully that we use MBRU in goto4 instead of MBR because we do not want sign extension of the second byte. The 16-bit offset is constructed, in fact, by ORing the two halves together. Finally, we have to fetch the next opcode before going back to Main1 because the code there expects the next opcode in MBR. The last cycle, goto6, is necessary because the memory data must be fetched in time to appear in MBR during Main1.

The offsets used in the goto IJVM instruction are signed 16-bit values, with a minimum of −32768 and a maximum of +32767. This means that branches either way to labels more distant than these values are not possible. This property can be regarded as either a bug or a feature in IJVM (and also in JVM). The bug camp would say that the JVM definition should not restrict their programming style. The feature camp would say that the work of many programmers would be radically improved if they had nightmares about the dreaded compiler message

Program is too big and hairy. You must rewrite it. Compilation aborted.

Unfortunately (in our view) this message appears only when a then or else clause exceeds 32 KB, typically at least 50 pages of Java.

Now consider the three IJVM conditional branch instructions: IFLT, IFEQ, and IF_ICMPEQ. The first two pop the top word from the stack, branching if it is less than zero or equal to zero, respectively. IF_ICMPEQ pops the top two words off the stack and branches if and only if they are equal. In all three cases, it is necessary to read in a new top-of-stack word to store in TOS.

The control for these three instructions is similar: the operand or operands are first put in registers, then the new top-of-stack value is read into TOS, finally the test and branch are made. Consider IFLT first. The word to test is already in TOS,

but since IFLT pops a word off the stack, the new top of stack must be read in to store in TOS. This read is started in iflt1. In iflt2, the word to be tested is saved in OPC for the moment, so the new value can be put in TOS shortly without losing the current one. In iflt3 the new top-of-stack word is available in MDR, so it is copied to TOS. Finally, in iflt4 the word to be tested, now saved in OPC, is run through the ALU without being stored and the N bit latched and tested. This microinstruction also contains a branch, choosing either T if the test was successful or F otherwise.

If successful, the remainder of the operation is essentially the same as at the beginning of the GOTO instruction, and the sequence simply continues in the middle of the GOTO sequence, with goto2. If unsuccessful, a short sequence (F, F2, and F3) is necessary to skip over the rest of the instruction (the offset) before returning to Main1 to continue with the next instruction.

The code in ifeq2 and ifeq3 follows the same logic, only using the Z bit instead of the N bit. In both cases, it is up to the assembler for MAL to recognize that the addresses T and F are special and to make sure that their addresses are placed at control store addresses that differ only in the leftmost bit.

The logic for IF_ICMPEQ is roughly similar to IFEQ except that here we need to read in the second operand as well. It is stored in H in if_icmpeq3, where the read of the new top-of-stack word is started. Again the current top-of-stack word is saved in OPC and the new one installed in TOS. Finally, the test in if_icmpeq6 is similar to ifeq4.

Now, we consider the implementation of INVOKEVIRTUAL and IRETURN, the instructions for invoking a procedure call and return, as described in Sec. 4.2.3. INVOKEVIRTUAL, a sequence of 22 microinstructions, is the most complex IJVM instruction implemented. Its operation was shown in Fig 4-12. The instruction uses its 16-bit offset to determine the address of the method to be invoked. In our implementation, the offset is simply an offset into the constant pool. This location in the constant pool points to the method to be invoked. Remember, however, that the first 4 bytes of each method are *not* instructions. Instead they are two 16-bit pointers. The first one gives the number of parameter words (including OBJREF—see Fig. 4-12). The second one gives the size of the local variable area in words. These fields are fetched through the 8-bit port and assembled just as if they were two 16-bit offsets within an instruction.

Then, the linkage information necessary to restore the machine to its previous state—the address of the start of the old local variable area and the old PC—is stored immediately above the newly created local variable area and below the new stack. Finally, the opcode of the next instruction is fetched and PC is incremented before returning to Main1 to begin the next instruction.

IRETURN is a simple instruction containing no operands. It simply uses the address stored in the first word of the local variable area to retrieve the linkage information. Then it restores SP, LV, and PC to their previous values and copies the return value from the top of the current stack onto the top of the original stack, as shown in Fig 4-13.

4.4 DESIGN OF THE MICROARCHITECTURE LEVEL

Like just about everything else in computer science, the design of the microarchitecture level is full of trade-offs. Computers have many desirable characteristics, including speed, cost, reliability, ease of use, energy requirements, and physical size. However, one trade-off drives the most important choices the CPU designer must make: speed versus cost. In this section we will look at this issue in detail to see what can be traded off against what, how high performance can be achieved, and at what price in hardware and complexity.

4.4.1 Speed versus Cost

While faster technology has resulted in the greatest speedup over any period of time, that is beyond the scope of this text. Speed improvements due to organization, while less amazing than that due to faster circuits, have nevertheless been impressive. Speed can be measured in a variety of ways, but given a circuit technology and an ISA, there are three basic approaches for increasing the speed of execution:

1. Reduce the number of clock cycles needed to execute an instruction.
2. Simplify the organization so that the clock cycle can be shorter.
3. Overlap the execution of instructions.

The first two are obvious, but there is a surprising variety of design opportunities that can dramatically affect either the number of clock cycles, the clock period, or—most often—both. In this section, we will give an example of how the encoding and decoding of an operation can affect the clock cycle.

The number of clock cycles needed to execute a set of operations is known as the **path length**. Sometimes the path length can be shortened by adding specialized hardware. For example, by adding an incrementer (conceptually, an adder with one side permanently wired to add 1) to PC, we no longer have to use the ALU to advance PC, eliminating cycles. The price paid is more hardware. However, this capability does not help as much as might be expected. For most instructions, the cycles consumed incrementing the PC are also cycles where a read operation is being performed. The subsequent instruction could not be executed earlier anyway because it depends on the data coming from the memory.

Reducing the number of instruction cycles necessary for fetching instructions requires more than just an additional circuit to increment the PC. In order to speed up the instruction fetching to any significant degree, the third technique—overlapping the execution of instructions—must be exploited. Separating out the circuitry for fetching the instructions—the 8-bit memory port, and the MBR and PC registers—is most effective if the unit is made functionally independent of the main data path. In this way, it can fetch the next opcode or operand on its own,

perhaps even performing asynchronously with respect to the rest of the CPU and fetching one or more instructions ahead.

One of the most time-consuming phases of the execution of many instructions is fetching a 2-byte offset, extending it appropriately, and accumulating it in the H register in preparation for an addition, for example, in a branch to PC $\pm$ n bytes. One potential solution—making the memory port 16 bits wide—greatly complicates the operation, because the memory is actually 32 bits wide. The 16 bits needed might span word boundaries, so that even a single read of 32 bits will not necessarily fetch both bytes needed.

Overlapping the execution of instructions is by far the most interesting approach and offers the most opportunity for dramatic increases in speed. Simple overlap of instruction fetch and execution is surprisingly effective. More sophisticated techniques go much further, however, overlapping execution of many instructions. In fact this idea is at the heart of modern computer design. We will discuss some of the basic techniques for overlapping instruction execution below and motivate some of the more sophisticated ones.

Speed is half the picture; cost is the other half. Cost can also be measured in a variety of ways, but a precise definition of cost is problematic. Some measures are as simple as a count of the number of components. This was particularly true in the days when processors were built of discrete components that were purchased and assembled. Today, the entire processor exists on a single chip, but bigger, more complex chips are much more expensive than smaller, simpler ones. Individual components—for example, transistors, gates, or functional units—can be counted, but often the count is not as important as the amount of area required on the integrated circuit. The more area required for the functions included, the larger the chip. And the manufacturing cost of the chip grows much faster than its area. For this reason, designers often speak of cost in terms of "real estate," that is, the area required for a circuit (presumably measured in pico-acres).

One of the most thoroughly studied circuits in history is the binary adder. There have been thousands of designs, and the fastest ones are much quicker than the slowest ones. They are also far more complex. The system designer has to decide whether the greater speed is worth the real estate.

Adders are not the only component with many choices. Nearly every component in the system can be designed to run faster or slower, with a cost differential. The challenge to the designer is to identify the components in the system to speed up in order to improve the system the most. Interestingly enough, many an individual component can be replaced with a much faster component with little or no effect on speed. In the following sections we will look at some of the design issues and the corresponding trade-offs.

A key factor in determining how fast the clock can run is the amount of work that must be done on each clock cycle. Obviously, the more work to be done, the longer the clock cycle. It's not quite that simple, of course, because the hardware is quite good at doing things in parallel, so it's actually the sequence of operations

that must be performed *serially* in a single clock cycle that determines how long the clock cycle must be.

One aspect that can be controlled is the amount of decoding that must be performed. Recall, for example, that in Fig. 4-6 we saw that while any of nine registers could be read into the ALU from the B bus, we required only 4 bits in the microinstruction word to specify which register was to be selected. Unfortunately, these savings come at a price. The decode circuit adds delay in the critical path. It means that whichever register is to enable its data onto the B bus will receive that command slightly later and will get its data on the bus slightly later. This effect cascades, with the ALU receiving its inputs a little later and producing its results a little later. Finally, the result is available on the C bus to be written to the registers a little later. Since this delay often is the factor that determines how long the clock cycle must be, this may mean that the clock cannot run quite as fast, and the entire computer must run a little slower. Thus there is a trade-off between speed and cost. Reducing the control store by 5 bits per word comes at the cost of slowing down the clock. The design engineer must take the design objectives into account when deciding which is the right choice. For a high-performance implementation, using a decoder is probably not a good idea; for a low-cost one, it might be.

4.4.2 Reducing the Execution Path Length

The Mic-1 was designed to be both moderately simple and moderately fast, although there is admittedly an enormous tension between these two goals. Briefly stated, simple machines are not fast and fast machines are not simple. The Mic-1 CPU also uses a minimum amount of hardware: 10 registers, the simple ALU of Fig. 3-19 replicated 32 times, a shifter, a decoder, a control store, and a bit of glue here and there. The whole system could be built with fewer than 5000 transistors plus whatever the control store (ROM) and main memory (RAM) take.

Having seen how IJVM can be implemented in a straightforward way in microcode with little hardware, let us now look at alternative, faster implementations. We will next look at ways to reduce the number of microinstructions per ISA instruction (i.e., reducing the execution path length). After that, we will consider other approaches.

Merging the Interpreter Loop with the Microcode

In the Mic-1, the main loop consists of one microinstruction that must be executed at the beginning of every IJVM instruction. In some cases it is possible to overlap it with the previous instruction. In fact, this has already been partially accomplished. Notice that when Main1 is executed, the opcode to be interpreted is already in MBR. It is there because it was fetched either by the previous main loop (if the previous instruction had no operands) or during the execution of the previous instruction.

This concept of overlapping the beginning of the instruction can be carried further, and in fact, the main-loop can in some cases be reduced to nothing. This can occur in the following way. Consider each sequence of microinstructions that terminates by branching to Main1. At each of these places, the main loop microinstruction can be tacked on to the end of the sequence (rather than at the beginning of the following sequence), with the multiway branch now replicated many places (but always with the same set of targets). In some cases the Main1 microinstruction can be merged with previous microinstructions, since those instructions are not always fully utilized.

In Fig. 4-23, the dynamic sequence of instructions is shown for a POP instruction. The main loop occurs before and after every instruction; in the figure we show only the occurrence after the POP instruction. Notice that the execution of this instruction takes four clock cycles: three for the specific microinstructions for POP and one for the main loop.

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 4-23. Original microprogram sequence for executing POP.

In Fig. 4-24 the sequence has been reduced to three instructions by merging the main-loop instructions, taking advantage of a clock cycle when the ALU is not used in pop2 to save a cycle and again in Main1. Be sure to note that the end of this sequence branches directly to the specific code for the subsequent instruction, so only three cycles are required total. This little trick reduces the execution time of the next microinstruction by one cycle, so, for example, a subsequent IADD goes from four cycles to three. It is thus equivalent to speeding up the clock from 250 MHz (4-nsec microinstructions) to 333 MHz (3-nsec microinstructions) for free.

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
Main1.pop	PC = PC + 1; fetch	MBR holds opcode; fetch next byte
pop3	TOS = MDR; goto (MBR)	Copy new word to TOS; dispatch on opcode

Figure 4-24. Enhanced microprogram sequence for executing POP.

The POP instruction is particularly well suited for this treatment, because it has a dead cycle in the middle that does not use the ALU. The main loop, however, does use the ALU. Thus to reduce the instruction length by one within an instruction requires finding a cycle in the instruction where the ALU is not in use. Such

dead cycles are not common, but they do occur, so merging Main1 into the end of each microinstruction sequence is worth doing. All it costs is a little control store. Thus we have our first technique for reducing path length:

Merge the interpreter loop into the end of each microcode sequence.

A Three-Bus Architecture

What else can we do to reduce execution path length? Another easy fix is to have two full input buses to the ALU, an A bus and a B bus, giving three buses in all. All (or at least most) of the registers should have access to both input buses. The advantage of having two input buses is that it then becomes possible to add any register to any other register in one cycle. To see the value of this feature, consider the Mic-1 implementation of ILOAD, shown again in Fig. 4-25.

Label	Operations	Comments
iload1	H = LV	MBR contains index; copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 4-25. Mic-1 code for executing ILOAD.

We see here that in iload1 LV is copied into H. The reason is so it can be added to MBRU in iload2. In our original two-bus design, there is no way to add two arbitrary registers, so one of them first has to be copied to H. With our new three-bus design, we can save a cycle, as shown in Fig. 4-26. We have added the interpreter loop to ILOAD here, but doing so neither increases nor decreases the execution path length. Still, the additional bus has reduced the total execution time of ILOAD from six cycles to five cycles. Now we have our second technique for reducing path length:

Go from a two-bus design to a three-bus design.

Label	Operations	Comments
iload1	MAR = MBRU + LV; rd	MAR = address of local variable to push
iload2	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload3	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload4	TOS = MDR	Update TOS
iload5	PC = PC + 1; fetch; goto (MBR)	MBR already holds opcode; fetch index byte

Figure 4-26. Three-bus code for executing ILOAD.

An Instruction Fetch Unit

Both of the foregoing techniques are worth using, but to get a dramatic improvement we need something much more radical. Let us step back and look at the common parts of every instruction: the fetching and decoding of the fields of the instruction. Notice that for every instruction the following operations may occur:

1. The PC is passed through the ALU and incremented.
2. The PC is used to fetch the next byte in the instruction stream.
3. Operands are read from memory.
4. Operands are written to memory.
5. The ALU does a computation and the results are stored back.

If an instruction has additional fields (for operands), each field must be explicitly fetched, 1 byte at a time, and assembled before it can be used. Fetching and assembling a field ties up the ALU for at least one cycle per byte to increment the PC, and then again to assemble the resulting index or offset. The ALU is used nearly every cycle for a variety of operations having to do with fetching the instruction and assembling the fields within the instruction, in addition to the real "work" of the instruction.

In order to overlap the main loop, it is necessary to free up the ALU from some of these tasks. This might be done by introducing a second ALU, though a full ALU is not necessary for much of the activity. Notice that in many cases the ALU is simply used as a path to copy a value from one register to another. These cycles might be eliminated by introducing additional data paths not going through the ALU. Some benefit may be derived, for example, by creating a path from TOS to MDR, or from MDR to TOS, since the top word of stack is frequently copied between those two registers.

In the Mic-1, much of the load can be removed from the ALU by creating an independent unit to fetch and process the instructions. This unit, called an **IFU** (**Instruction Fetch Unit**), can independently increment PC and fetch bytes from the byte stream before they are needed. This unit requires only an incrementer, a circuit far simpler than a full adder. Carrying this idea further, the IFU can also assemble 8- and 16-bit operands so that they are ready for immediate use whenever needed. There are at least two ways this can be accomplished:

1. The IFU can actually interpret each opcode, determining how many additional fields must be fetched, and assemble them into a register ready for use by the main execution unit.
2. The IFU can take advantage of the stream nature of the instructions and make available at all times the next 8- and 16-bit pieces, whether or not doing so makes any sense. The main execution unit can then ask for whatever it needs.

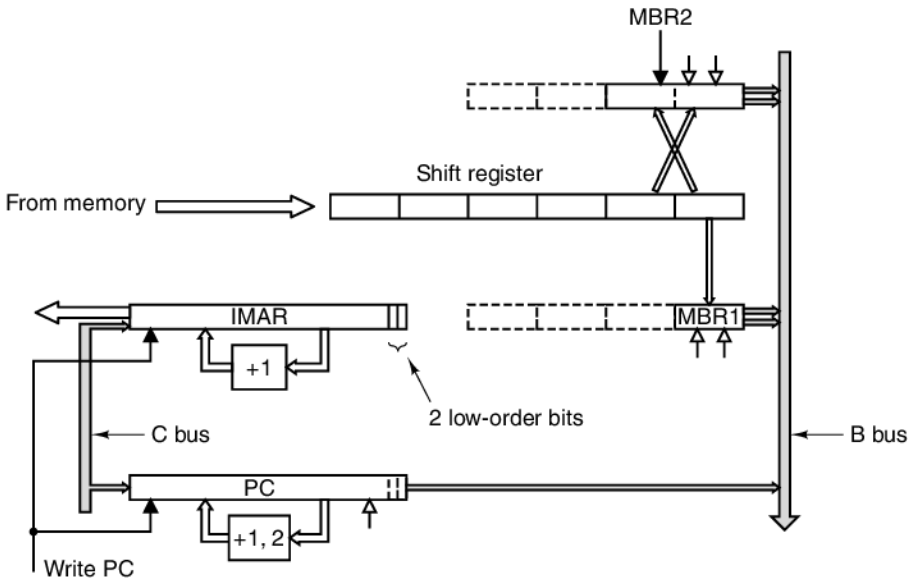


Figure 4-27. A fetch unit for the Mic-1.

We show the rudiments of the second scheme in Fig. 4-27. Rather than a single 8-bit MBR, there are now two MBRs: the 8-bit MBR1 and the 16-bit MBR2. The IFU keeps track of the most recent byte or bytes consumed by the main execution unit. It also makes available in MBR1 the next byte, just as in the Mic-1, except that it automatically senses when the MBR1 is read, prefetches the next byte, and loads it into MBR1 immediately. As in the Mic-1, it has two interfaces to the B bus: MBR1 and MBR1U. The former is sign-extended to 32 bits; the latter is zero-extended.

Similarly, MBR2 provides the same functionality but holds the next 2 bytes. It also has two interfaces to the B bus: MBR2 and MBR2U, gating the 32-bit sign-extended and zero-extended values, respectively.

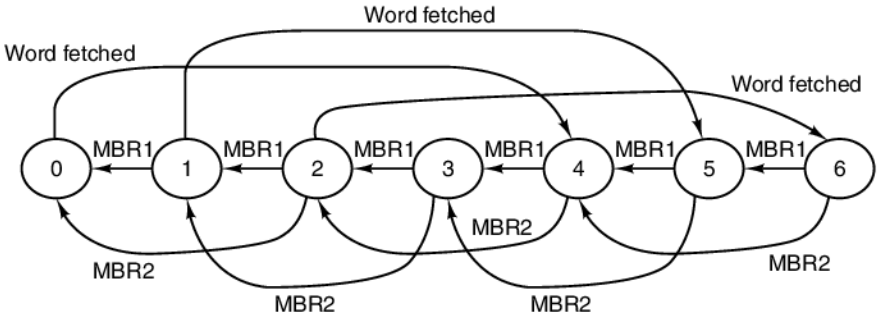
The IFU is responsible for fetching a stream of bytes. It does this by using a conventional 4-byte memory port, fetching entire 4-byte words ahead of time and loading the consecutive bytes into a shift register that supplies them one or two at a time, in the order fetched. The function of the shift register is to maintain a queue of bytes from memory, to feed MBR1 and MBR2.

At all times, MBR1 holds the oldest byte in the shift register and MBR2 holds the oldest 2 bytes (oldest byte on the left), forming a 16-bit integer [see Fig. 4-19(b)]. The 2 bytes in MBR2 may be from different memory words, because IJVM instructions do not align on word boundaries in memory.

Whenever MBR1 is read, the shift register shifts right 1 byte. Whenever MBR2 is read, it shifts right 2 bytes. Then MBR1 and MBR2 are reloaded from the oldest byte and pair of bytes, respectively. If there is sufficient room now left in the shift register for another whole word, the IFU starts a memory cycle in order to read it.

We assume that when any of the MBR registers is read, it is refilled by the start of the next cycle, so it can be read out on consecutive cycles.

The design of the IFU can be modeled by an **FSM** (**Finite State Machine**) as shown in Fig. 4-28. All FSMs consist of two parts: **states**, shown as circles, and **transitions**, shown as arcs from one state to another. Each state represents one possible situation the FSM can be in. This particular FSM has seven states, corresponding to the seven states of the shift register of Fig. 4-27. The seven states correspond to how many bytes are currently in the shift register, a number between 0 and 6, inclusive.



Transitions
MBR1: Occurs when MBR1 is read
MBR2: Occurs when MBR2 is read
Word fetched: Occurs when a memory word is read and 4 bytes are put into the shift register

Figure 4-28. A finite-state machine for implementing the IFU.

Each arc represents an event that can occur. Three different events can occur here. The first event is 1 byte being read from MBR1. This event causes the shift register to be activated and 1 byte shifted off the right-hand end, reducing the state by 1. The second event is 2 bytes being read from MBR2, which reduces the state by two. Both of these transitions cause MBR1 and MBR2 to be reloaded. When the FSM moves into states 0, 1, or 2, a memory reference is started to fetch a new word (assuming that the memory is not already busy reading a word). The arrival of the word advances the state by 4.

To work correctly, the IFU must block when it is asked to do something it cannot do, such as supply the value of MBR2 when there is only 1 byte in the shift register and the memory is still busy fetching a new word. Also, it can do only one thing at a time, so incoming events must be serialized. Finally, whenever PC is changed, the IFU must be updated. Such details make it more complicated than we have shown. Still, many hardware devices are constructed as FSMs.

The IFU has its own memory address register, called IMAR, which is used to address memory when a new word has to be fetched. This register has its own

dedicated incrementer so that the main ALU is not needed to increment it to get the next word. The IFU must monitor the C bus so that whenever PC is loaded, the new PC value is also copied into IMAR. Since the new value in PC may not be on a word boundary, the IFU has to fetch the necessary word and adjust the shift register appropriately.

With the IFU, the main execution unit writes to PC only when the sequential nature of the instruction byte stream must be changed. It writes on a successful branch instruction and on INVOKEVIRTUAL and IRETURN.

Since the microprogram no longer explicitly increments PC as opcodes are fetched, the IFU must keep PC current. It does this by sensing when a byte from the instruction stream has been consumed, that is, when MBR1 or MBR2 (or the unsigned versions) have been read. Associated with PC is a separate incrementer, capable of incrementing by 1 or 2, depending on how many bytes have been consumed. Thus the PC always contains the address of the first byte that has not been consumed. At the beginning of each instruction, MBR contains the address of the opcode for that instruction.

Note that there are two separate incrementers and they perform different functions. PC counts *bytes* and increments by 1 or 2. IMAR counts *words* and increments only by 1 (for 4 new bytes). Like MAR, IMAR is wired to the address bus "diagonally" with IMAR bit 0 connected to address line 2, and so on, to perform an implicit conversion of word addresses to byte addresses.

As we will see shortly in detail, not having to increment PC in the main loop is a big win, because the microinstruction in which PC is incremented often does little else except increment PC. If this microinstruction can be eliminated, the execution path can be reduced. The trade-off here is more hardware for a faster machine, so our third technique for reducing path length is

Have instructions fetched from memory by a specialized functional unit.

4.4.3 A Design with Prefetching: The Mic-2

The IFU can greatly reduce the path length of the average instruction. First, it eliminates the main loop entirely, since the end of each instruction simply branches directly to the next instruction. Second, it avoids tying up the ALU incrementing PC. Third, it reduces the path length whenever a 16-bit index or offset is calculated, because it assembles the 16-bit value and supplies it directly to the ALU as a 32-bit value, avoiding the need for assembly in H. Figure 4-29 shows the Mic-2, an enhanced version of the Mic-1 where the IFU of Fig. 4-27 has been added. The microcode for the enhanced machine is shown in Fig. 4-30.

As an example of how the Mic-2 works, look at IADD. It fetches the second word on the stack and does the addition as before, only now it does not have to go to Main1 when it is done to increment PC and dispatch to the next microinstruction. When the IFU sees that MBR1 has been referenced in iadd3, its internal shift register pushes everything to the right and reloads MBR1 and MBR2. It also makes a

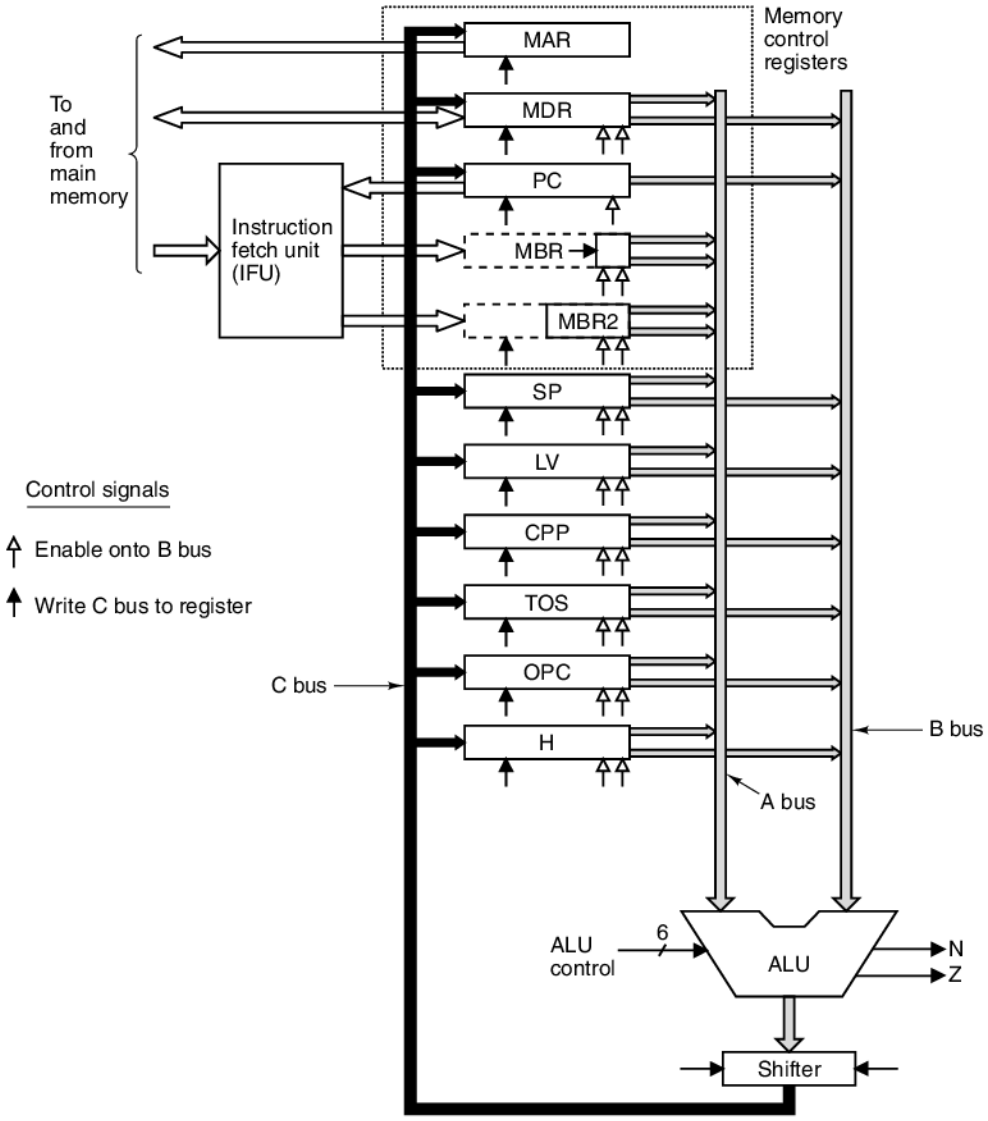


Figure 4-29. The data path for Mic-2.

transition to a state one lower than its current one. If the new state is 2, the IFU starts fetching a word from memory. All of this is in hardware. The microprogram does not have to do anything. That is why IADD can be reduced from four microinstructions to three.

The Mic-2 improves some instructions more than others. LDC_W goes from nine microinstructions to only three, cutting its execution time by a factor of three.

On the other hand, SWAP goes only from eight microinstructions to six. For overall performance, the gain for the more common instructions is what really counts. These include ILOAD (was 6, now 3), IADD (was 4, now 3), and IF_ICMPEQ (was 13, now 10 for the taken case; was 10, now 8 for the not taken case). To measure the improvement, one would have to choose and run some benchmarks, but clearly there is a major gain here.

4.4.4 A Pipelined Design: The Mic-3

The Mic-2 is clearly an improvement over the Mic-1. It is faster and uses less control store, although the cost of the IFU will undoubtedly more than offset the real estate won by having a smaller control store. Thus it is a considerably faster machine at a marginally higher price. Let us see if we can make it faster still.

How about trying to decrease the cycle time? To a considerable extent, the cycle time is determined by the underlying technology. The smaller the transistors and the smaller the physical distances between them, the faster the clock can be run. For a given technology, the time required to perform a full data path operation is fixed (at least from our point of view). Nevertheless, we do have some freedom and we will exploit it to the fullest shortly.

Our other option is to introduce more parallelism into the machine. At the moment, the Mic-2 is highly sequential. It puts registers onto its buses, waits for the ALU and shifter to process them, and then writes the results back to the registers. Except for the IFU, little parallelism is present. Adding parallelism is a real opportunity.

As mentioned earlier, the clock cycle is limited by the time needed for the signals to propagate through the data path. Figure 4-3 shows a breakdown of the delay through the various components during each cycle. There are three major components to the actual data path cycle:

1. The time to drive the selected registers onto the A and B buses.
2. The time for the ALU and shifter to do their work.
3. The time for the results to get back to the registers and be stored.

In Fig. 4-31, we show a new three-bus architecture, including the IFU, but with three additional latches (registers), one inserted in the middle of each bus. The latches are written on every cycle. In effect, these registers partition the data path into distinct parts that can now operate independently of one another. We will refer to this as **Mic-3**, or the **pipelined** model.

How can these extra registers possibly help? Now it takes three clock cycles to use the data path: one for loading the A and B latches, one for running the ALU and shifter and loading the C latch, and one for storing the C latch back into the registers. Surely this is worse than what we already had.

Label	Operations	Comments
nop1	goto (MBR)	Branch to next instruction
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR+H; wr; goto (MBR1)	Add top two words; write to new top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR–H; wr; goto (MBR1)	Subtract TOS from Fetched TOS-1
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	AND Fetched TOS-1 with TOS
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	OR Fetched TOS-1 with TOS
dup1	MAR = SP = SP + 1	Increment SP; copy to MAR
dup2	MDR = TOS; wr; goto (MBR1)	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for read
pop3	TOS = MDR; goto (MBR1)	Copy new word to TOS
swap1	MAR = SP – 1; rd	Read 2nd word from stack; set MAR to SP
swap2	MAR = SP	Prepare to write new 2nd word
swap3	H = MDR; wr	Save new TOS; write 2nd word to stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Write old TOS to 2nd place on stack
swap6	TOS = H; goto (MBR1)	Update TOS
bipush1	SP = MAR = SP + 1	Set up MAR for writing to new top of stack
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Update stack in TOS and memory
iload1	MAR = LV + MBR1U; rd	Move LV + index to MAR; read operand
iload2	MAR = SP = SP + 1	Increment SP; Move new SP to MAR
iload3	TOS = MDR; wr; goto (MBR1)	Update stack in TOS and memory
istore1	MAR = LV + MBR1U	Set MAR to LV + index
istore2	MDR = TOS; wr	Copy TOS for storing
istore3	MAR = SP = SP – 1; rd	Decrement SP; read new TOS
istore4		Wait for read
istore5	TOS = MDR; goto (MBR1)	Update TOS
wide1	goto (MBR1 OR 0x100)	Next address is 0x100 ored with opcode
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Identical to iload1 but using 2-byte index
wide_istore1	MAR = LV + MBR2U; goto istore2	Identical to istore1 but using 2-byte index
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Same as wide_iload1 but indexing off CPP
iinc1	MAR = LV + MBR1U; rd	Set MAR to LV + index for read
iinc2	H = MBR1	Set H to constant
iinc3	MDR = MDR + H; wr; goto (MBR1)	Increment by constant and update
goto1	H = PC – 1	Copy PC to H
goto2	PC = H + MBR2	Add offset and update PC
goto3		Have to wait for IFU to fetch new opcode
goto4	goto (MBR1)	Dispatch to next instruction
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit

Figure 4-30. The microprogram for the Mic-2 (part 1 of 2).

Label	Operations	Comments
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = H – OPC; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	H = PC – 1; goto goto2	Same as goto1
F	H = MBR2	Touch bytes in MBR2 to discard
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Put address of method pointer in MAR
invokevirtual2	OPC = PC	Save Return PC in OPC
invokevirtual3	PC = MDR	Set PC to 1st byte of method code.
invokevirtual4	TOS = SP – MBR2U	TOS = address of OBJREF – 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = address of OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Overwrite OBJREF with link pointer
invokevirtual7	MAR = SP = MDR	Set SP, MAR to location to hold old PC
invokevirtual8	MDR = OPC; wr	Prepare to save old PC
invokevirtual9	MAR = SP = SP + 1	Inc. SP to point to location to hold old LV
invokevirtual10	MDR = LV; wr	Save old LV
invokevirtual11	LV = TOS; goto (MBR1)	Set LV to point to zeroth parameter.
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to read Link ptr
ireturn2		Wait for link ptr
ireturn3	LV = MAR = MDR; rd	Set LV, MAR to link ptr; read old PC
ireturn4	MAR = LV + 1	Set MAR to point to old LV; read old LV
ireturn5	PC = MDR; rd	Restore PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Save return value on original top of stack

Figure 4-30. The microprogram for the Mic-2 (part 2 of 2).

Are we crazy? (*Hint*: No.) The point of inserting the latches is twofold:

1. We can speed up the clock because the maximum delay is now shorter.
2. We can use all parts of the data path during every cycle.

By breaking up the data path into three parts, we reduce the maximum delay with the result that the clock frequency can be higher. Let us suppose that by breaking the data path cycle into three time intervals, each one is about 1/3 as long as the original, so we can triple the clock speed. (This is not totally realistic, since we have also added two more registers into the data path, but as a first approximation it will do.)

Because we have been assuming that all memory reads and writes can be satisfied out of the level 1 cache, and this cache is made out of the same material as the

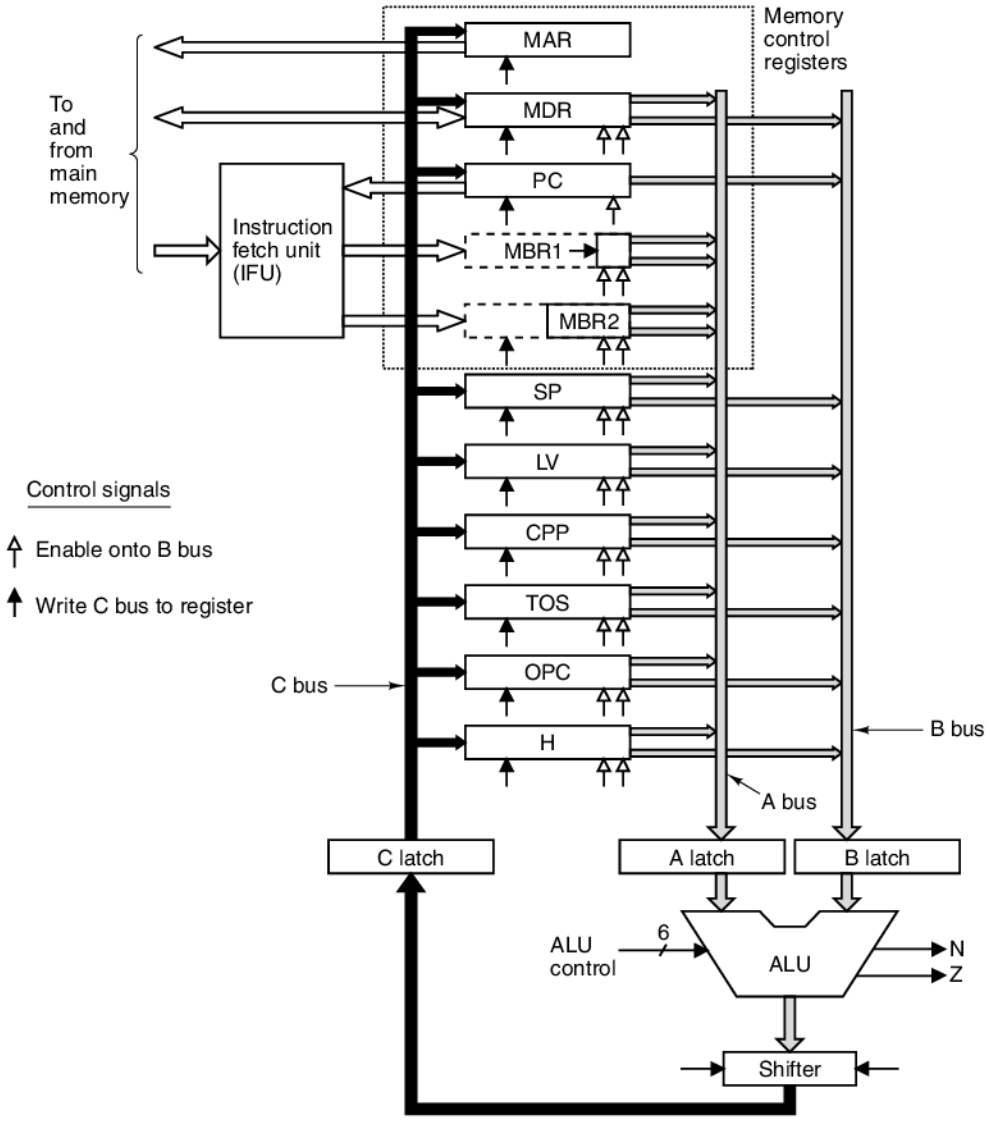


Figure 4-31. The three-bus data path used in the Mic-3.

registers, we will continue to assume that a memory operation takes one cycle. In practice, though, this may not be so easy to achieve.

The second point deals with throughput rather than the speed of an individual instruction. In the Mic-2, during the first and third parts of each clock cycle the ALU is idle. By breaking the data path up into three pieces, we will be able to use the ALU on every cycle, getting three times as much work out of the machine.

Let us now see how the Mic-3 data path works. Before starting, we need a notation for dealing with the latches. The obvious one is to call them A, B, and C and treat them like registers, keeping in mind the constraints of the data path. Figure 4-32 shows an example code sequence, the implementation of SWAP for the Mic-2.

Label	Operations	Comments
swap1	MAR = SP – 1; rd	Read 2nd word from stack; set MAR to SP
swap2	MAR = SP	Prepare to write new 2nd word
swap3	H = MDR; wr	Save new TOS; write 2nd word to stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Write old TOS to 2nd place on stack
swap6	TOS = H; goto (MBR1)	Update TOS

Figure 4-32. The Mic-2 code for SWAP.

Now let us reimplement this sequence on the Mic-3. Remember that the data path now requires three cycles to operate: one to load A and B, one to perform the operation and load C, and one to write the results back to the registers. We will call each of these pieces a **microstep**.

The implementation of SWAP for the Mic-3 is shown in Fig. 4-33. In cycle 1, we start on swap1 by copying SP to B. It does not matter what goes in A because to subtract 1 from B ENA is negated (see Fig. 4-2). For simplicity, we will not show assignments that are not used. In cycle 2 we do the subtraction. In cycle 3 the result is stored in MAR and the read operation is started at the end of cycle 3 (after MAR has been stored). Since memory reads now take one cycle, this one will not complete until the end of cycle 4, indicated by showing the assignment to MDR in cycle 4. The value in MDR may be read no earlier than cycle 5.

Now let us go back to cycle 2. We can now begin breaking up swap2 into microsteps and starting them, too. In cycle 2, we can copy SP to B, then run it through the ALU in cycle 3 and finally store it in MAR in cycle 4. So far, so good. It should be clear that if we can keep going at this rate, starting a new microinstruction every cycle, we have tripled the speed of the machine. This gain comes from the fact that we can issue a new microinstruction on every clock cycle, and the Mic-3 has three times as many clock cycles per second as the Mic-2 has. In fact, we have built a pipelined CPU.

Unfortunately, we hit a snag in cycle 3. We would like to start working on swap3, but the first thing it does is run MDR through the ALU, and MDR will not be available from memory until the start of cycle 5. The situation that a microstep cannot start because it is waiting for a result that a previous microstep has not yet produced is called a **true dependence** or a **RAW dependence**. Dependences are often referred to as **hazards**. RAW stands for Read After Write and indicates that a microstep wants to read a register that has not yet been written. The only sensible thing to do here is delay the start of swap3 until MDR is available, in cycle 5.

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Cy	MAR=SP–1;rd	MAR=SP	H=MDR;wr	MDR=TOS	MAR=SP–1;wr	TOS=H;goto (MBR1)
1	B=SP					
2	C=B–1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B–1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

Figure 4-33. The implementation of SWAP on the Mic-3.

Stopping to wait for a needed value is called **stalling**. After that, we can continue starting microinstructions every cycle as there are no more dependences, although swap6 just barely makes it, since it reads H in the cycle after swap3 writes it. If swap5 had tried to read H, it would have stalled for one cycle.

Although the Mic-3 program takes more cycles than the Mic-2 program, it still runs faster. If we call the Mic-3 cycle time ΔT nsec, then the Mic-3 requires $11\Delta T$ nsec to execute SWAP. In contrast, the Mic-2 takes 6 cycles at $3\Delta T$ each, for a total of $18\Delta T$. Pipelining has made the machine faster, even though we had to stall once to avoid a dependence.

Pipelining is a key technique in all modern CPUs, so it is important to understand it well. In Fig. 4-34 we see the data path of Fig. 4-31 graphically illustrated as a pipeline. The first column represents what is going on during cycle 1, the second column represents cycle 2, and so on (assuming no stalls). The shaded region in cycle 1 for instruction 1 indicates that the IFU is busy fetching instruction 1. One clock tick later, during cycle 2, the registers required by instruction 1 are being loaded into the A and B latches while at the same time the IFU is busy fetching instruction 2, again shown by the two shaded rectangles in cycle 2.

During cycle 3, instruction 1 is using the ALU and shifter to do its operation, the A and B latches are being loaded for instruction 2, and instruction 3 is being fetched. Finally, during cycle 4, four instructions are being worked on at the same time. The results from instruction 1 are being stored, the ALU work for instruction 2 is being performed, the A and B latches for instruction 3 are being loaded, and instruction 4 is being fetched.

If we had shown cycle 5 and subsequent cycles, the pattern would have been the same as in cycle 4: all four parts of the data path that can run independently

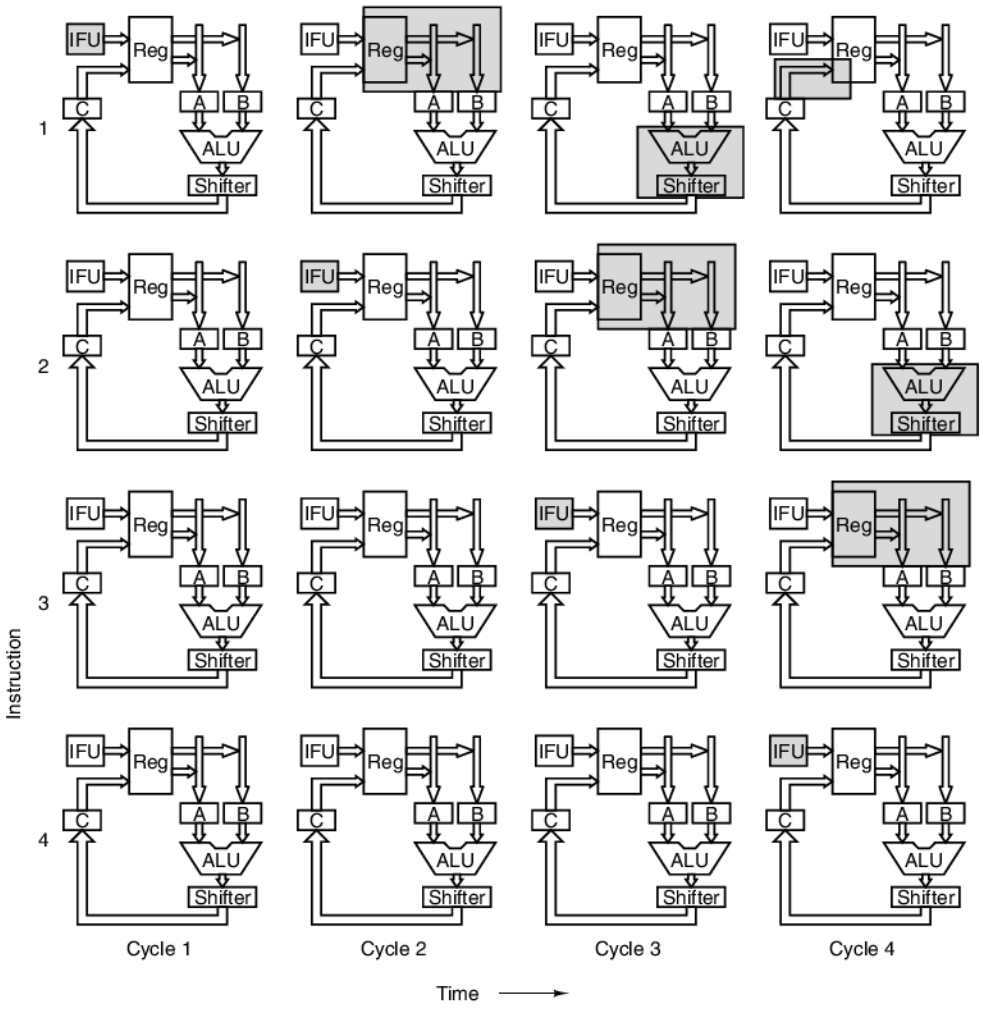


Figure 4-34. Graphical illustration of how a pipeline works.

would be doing so. This design represents a four-stage pipeline, with stages for instruction fetching, operand access, ALU operations, and writeback to the registers. It is similar to the pipeline of Fig. 2-4(a), except without the decode stage. The important point to pick up here is that although a single instruction takes four clock cycles to carry out, on every clock cycle one new instruction is started and one old instruction completes.

Another way to look at Fig. 4-34 is to follow each instruction horizontally across the page. For instruction 1, in cycle 1 the IFU is working on it. In cycle 2, its registers are being put onto the A and B buses. In cycle 3, the ALU and shifter are working for it. Finally, in cycle 4, its results are being stored back into the

registers. The thing to note here is that four sections of the hardware are available, and during each cycle a given instruction uses only one of them, freeing up the other sections for different instructions.

A useful analogy to our pipelined design is an assembly line in a factory that assembles cars. To abstract out the essentials of this model, imagine that a big gong is struck every minute, at which time all cars move one station further down the line. At each station, the workers there perform some operation on the car currently in front of them, like adding the steering wheel or installing the brakes. At each beat of the gong (1 cycle), one new car is injected into the start of the assembly line and one finished car drives off the end. Thus even though it may take hundreds of cycles to complete a car, on every cycle a whole car is completed. The factory can produce one car per minute, independent of how long it actually takes to assemble a car. This is the power of pipelining, and it applies equally well to CPUs as to car factories.

4.4.5 A Seven-Stage Pipeline: The Mic-4

One point we have glossed over is that every microinstruction selects its own successor. Most of them just select the next one in the current sequence, but the last one, such as swap6, often does a multiway branch, which gums up the pipeline since continuing to prefetch after it is impossible. We need a better way of dealing with this point.

Our last microarchitecture is the Mic-4. Its main parts are shown in Fig. 4-35, but a substantial amount of detail has been suppressed for clarity. Like the Mic-3, it has an IFU that prefetches words from memory and maintains the various MBRs.

The IFU also feeds the incoming byte stream to a new component, the **decoding unit**. This unit has an internal ROM indexed by IJVM opcode. Each entry (row) contains two parts: the length of that IJVM instruction and an index into another ROM, the micro-operation ROM. The IJVM instruction length is used to allow the decoding unit to parse the incoming byte stream into instructions, so it always knows which bytes are opcodes and which are operands. If the current instruction length is 1 byte (e.g., POP), then the decoding unit knows that the next byte is an opcode. If, however, the current instruction length is 2 bytes, the decoding unit knows that the next byte is an operand, followed immediately by another opcode. When the WIDE prefix is seen, the following byte is transformed into a special wide opcode, for example, WIDE + ILOAD becomes WIDE_ILOAD.

The decoding unit ships the index into the micro-operation ROM that it found in its table to the next component, the **queueing unit**. This unit contains some logic plus two internal tables, one in ROM and one in RAM. The ROM contains the microprogram, with each IJVM instruction having some number of consecutive entries, called **micro-operations**. The entries must be in order, so tricks like wide_iload2 branching to iload2 in Mic-2 are not allowed. Each IJVM sequence must be spelled out in full, duplicating sequences in some cases.

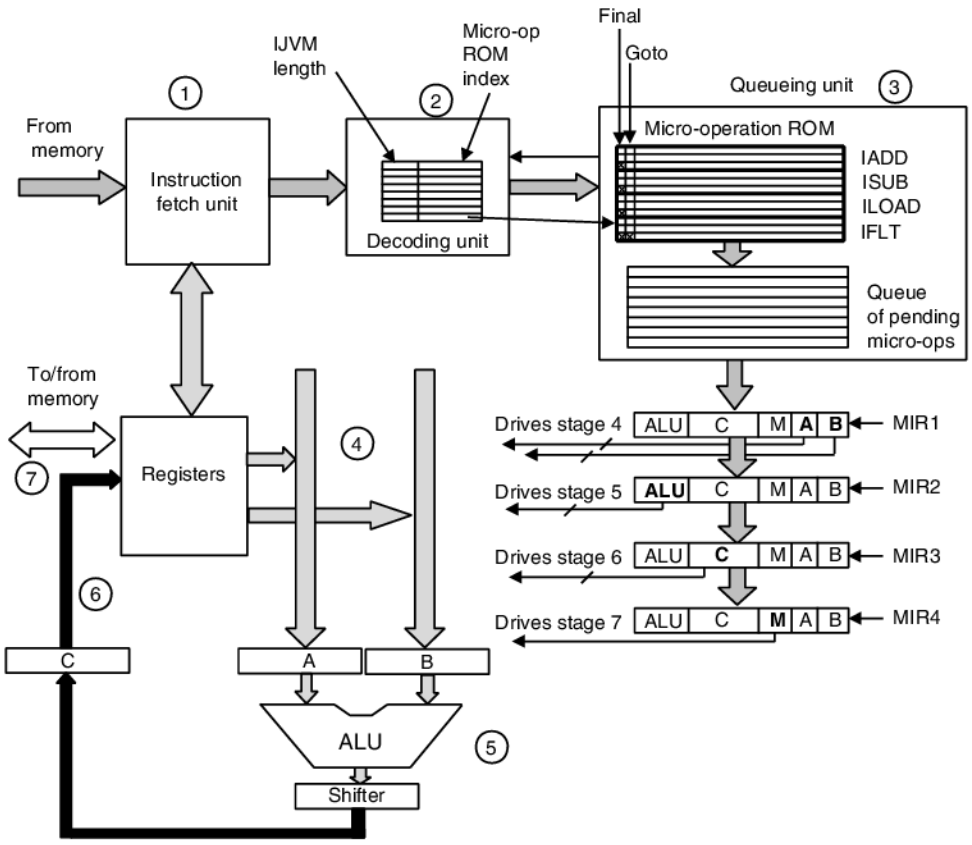


Figure 4-35. The main components of the Mic-4.

The micro-operations are similar to the microinstructions of Fig. 4-5 except that the NEXT_ADDRESS and JAM fields are absent and a new encoded field is needed to specify the A bus input. Two new bits are also provided: Final and Goto. The Final bit is set on the last micro-operation of each IJVM micro-operation sequence to mark it. The Goto bit is set to mark micro-operations that are conditional microbranches. They have a different format from the normal micro-operations, consisting of the JAM bits and an index into the micro-operation ROM. Microinstructions that previously did something with the data path and also performed a conditional microbranch (e.g., iflt4) now have to be split up into two micro-operations.

The queueing unit works as follows. It receives a micro-operation ROM index from the decoding unit. It then looks up the micro-operation and copies it into an internal queue. Then it copies the following micro-operation into the queue as well, and the one after it too. It keeps going until it hits one with the Final bit on. It copies that one, too, and stops. Assuming that it has not hit a micro-operation

with the Goto bit on and still has ample room left in the queue, the queueing unit then sends an acknowledgement signal back to the decoding unit. When the decoding unit sees the acknowledgement, it sends the index of the next IJVM instruction to the queueing unit.

In this way, the sequence of IJVM instructions in memory are ultimately converted into a sequence of micro-operations in a queue. These micro-operations feed the MIRs, which send the signals out to control the data path. However, another factor we now have to consider is that the fields on each micro-operation are not active at the same time. The A and B fields are active during the first cycle, the ALU field is active during the second cycle, the C field is active during the third cycle, and any memory operations take place in the fourth cycle.

To make this work properly, we have introduced four independent MIRs into Fig. 4-35. At the start of each clock cycle (the Δw time in Fig. 4-3), MIR3 is copied to MIR4, MIR2 is copied to MIR3, MIR1 is copied to MIR2, and MIR1 is loaded with a fresh micro-operation from the micro-operation queue. Then each MIR puts out its control signals, but only some of them are used. The A and B fields from MIR1 are used to select the registers that drive the A and B latches, but the ALU field in MIR1 is not used and is not connected to anything else in the data path.

One clock cycle later, this micro-operation has moved on to MIR2 and the registers that it selected are now safely sitting in the A and B latches waiting for the adventures to come. Its ALU field is now used to drive the ALU. In the next cycle, its C field will write the results back into the registers. After that, it will move on to MIR4 and initiate any memory operations needed using the now-loaded MAR (and MDR, for a write).

One last aspect of the Mic-4 needs some discussion now: microbranches. Some IJVM instructions, such as IFLT, need to conditionally branch based on, say, the N bit. When a microbranch occurs, the pipeline cannot continue. To deal with that, we have added the Goto bit to the micro-operation. When the queueing unit hits a micro-operation with this bit set while copying it to the queue, it realizes that there is trouble ahead and refrains from sending an acknowledgement to the decoding unit. As a result, the machine will stall at this point until the microbranch has been resolved.

Conceivably, some IJVM instructions beyond the branch have already been fed into the decoding unit (but not into the queueing unit), since it does not send back an acknowledge (i.e., continue) signal when it hits a micro-operation with the Goto bit on. Special hardware and mechanisms are needed to clean up the mess and get back on track, but they are beyond the scope of this book. When Edsger Dijkstra wrote his famous letter "GOTO Statement Considered Harmful" (Dijkstra, 1968a), he had no idea how right he was.

We have come a long way since the Mic-1. The Mic-1 was a very simple piece of hardware, with nearly all the control done in software. The Mic-4 is a highly pipelined design, with seven stages and far more complex hardware. The pipeline is shown schematically in Fig. 4-36, with the circled numbers keyed back to the

components in Fig. 4-35. The Mic-4 automatically prefetches a stream of bytes from memory, decodes them into IJVM instructions, converts them to a sequence of micro-operations using a ROM, and queues them for use as needed. The first three stages of the pipeline can be tied to the data path clock if desired, but there will not always be work to do. For example, the IFU certainly cannot feed a new IJVM opcode to the decoding unit on every clock cycle because IJVM instructions take several cycles to execute and the queue would rapidly overflow.

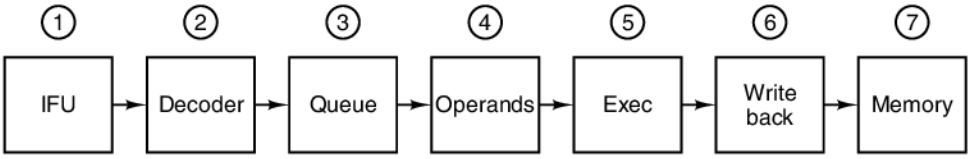


Figure 4-36. The Mic-4 pipeline.

On each clock cycle, the MIRs are shifted forward and the micro-operation at the bottom of the queue is copied into MIR1 to start its execution. The control signals from the four MIRs then spread out through the data path, causing actions to occur. Each MIR controls a different portion of the data path and thus different microsteps.

In this design we have a deeply pipelined CPU, which allows the individual steps to be very short and thus the clock frequency high. Many CPUs are designed in essentially this way, especially those that have to implement an older (CISC) instruction set. For example, the Core i7 implementation is conceptually similar to the Mic-4 in some ways, as we will see later in this chapter.

4.5 IMPROVING PERFORMANCE

All computer manufacturers want their systems to run as fast as possible. In this section, we will look at a number of advanced techniques currently being investigated to improve system (primarily CPU and memory) performance. Due to the highly competitive nature of the computer industry, the lag between new research ideas that can make a computer faster and their incorporation into products is surprisingly short. Consequently, most of the ideas we will discuss are already in use in a wide variety of existing products.

The ideas to be discussed fall into roughly two categories: implementation improvements and architectural improvements. Implementation improvements are ways of building a new CPU or memory to make the system run faster without changing the architecture. Modifying the implementation without changing the architecture means that old programs will run on the new machine, a major selling point. One way to improve the implementation is to use a faster clock, but this is

not the only way. The performance gains from the 80386 through the 80486, Pentium, and later designs like the Core i7 are due to better implementations, as the architecture has remained essentially the same through all of them.

Some kinds of improvements can be made only by changing the architecture. Sometimes these changes are incremental, such as adding new instructions or registers, so that old programs will continue to run on the new models. In this case, to get the full performance, the software must be changed, or at least recompiled with a new compiler that takes advantage of the new features.

However, once in a few decades, designers realize that the old architecture has outlived its usefulness and that the only way to make progress is start all over again. The RISC revolution in the 1980s was one such breakthrough; another one is in the air now. We will look at one example (the Intel IA-64) in Chap. 5.

In the rest of this section we will look at four different techniques for improving CPU performance. We will start with three well-established implementation improvements and then move on to one that needs a little architectural support to work best. These techniques are cache memory, branch prediction, out-of-order execution with register renaming, and speculative execution.

4.5.1 Cache Memory

One of the most challenging aspects of computer design throughout history has been to provide a memory system able to provide operands to the processor at the speed it can process them. The recent high rate of growth in processor speed has not been accompanied by a corresponding speedup in memories. Relative to CPUs, memories have been getting slower for decades. Given the enormous importance of primary memory, this situation has greatly limited the development of high-performance systems and has stimulated research on ways to get around the problem of memory speeds that are much slower than CPU speeds and, relatively speaking, getting worse every year.

Modern processors place overwhelming demands on a memory system, in terms of both latency (the delay in supplying an operand) and bandwidth (the amount of data supplied per unit of time). Unfortunately, these two aspects of a memory system are largely at odds. Many techniques for increasing bandwidth do so only by increasing latency. For example, the pipelining techniques used in the Mic-3 can be applied to a memory system, with multiple, overlapping memory requests handled efficiently. Unfortunately, as with the Mic-3, this results in greater latency for individual memory operations. As processor clock speeds get faster, it becomes more and more difficult to provide a memory system capable of supplying operands in one or two clock cycles.

One way to attack this problem is by providing caches. As we saw in Sec. 2.2.5, a cache holds the most recently used memory words in a small, fast memory, speeding up access to them. If a large enough percentage of the memory words needed are in the cache, the effective memory latency can be reduced enormously.

One of the most effective ways to improve both bandwidth and latency is to use multiple caches. A basic technique that works very effectively is to introduce a separate cache for instructions and data. There are several benefits from having separate caches for instructions and data, often called a **split cache**. First, memory operations can be initiated independently in each cache, effectively doubling the bandwidth of the memory system. This is why it makes sense to provide two separate memory ports, as we did in the Mic-1: each port has its own cache. Note that each cache has independent access to the main memory.

Today, many memory systems are more complicated than this, and an additional cache, called a **level 2 cache**, may reside between the instruction and data caches and main memory. In fact, as more sophisticated memory systems are required, there may be three or more levels of cache. In Fig. 4-37 we see a system with three levels of cache. The CPU chip itself contains a small instruction cache and a small data cache, typically 16 KB to 64 KB. Then there is the level 2 cache, which is not on the CPU chip but may be included in the CPU package, next to the CPU chip and connected to it by a high-speed path. This cache is generally unified, containing a mix of data and instructions. A typical size for the L2 cache is 512 KB to 1 MB. The third-level cache is on the processor board and consists of a few megabytes of SRAM, which is much faster than the main DRAM memory. Caches are generally inclusive, with the full contents of the level 1 cache being in the level 2 cache and the full contents of the level 2 cache being in the level 3 cache.

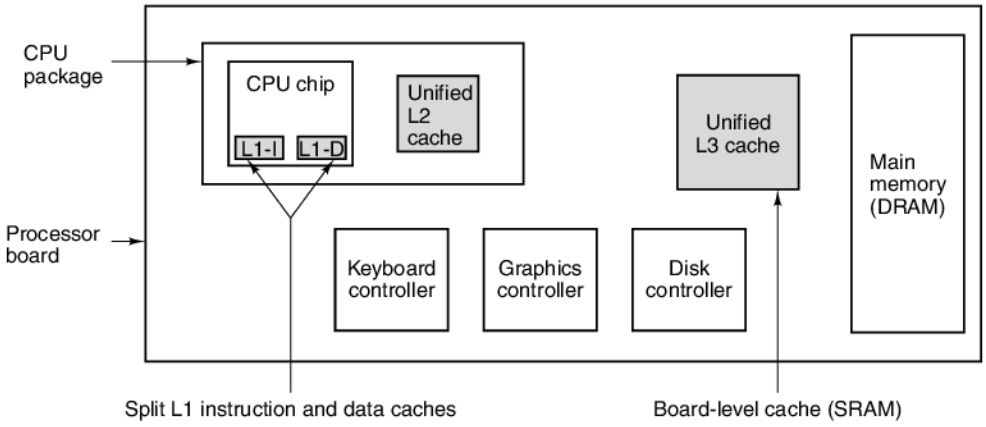


Figure 4-37. A system with three levels of cache.

To achieve their goal, caches depend on two kinds of address locality. **Spatial locality** is the observation that memory locations with addresses numerically similar to a recently accessed memory location are likely to be accessed in the near

future. Caches try to exploit this property by bringing in more data than have been requested, with the expectation that future requests can be anticipated. **Temporal locality** occurs when recently accessed memory locations are accessed again. This may occur, for example, to memory locations near the top of the stack, or instructions inside a loop. Temporal locality is exploited in cache designs primarily by the choice of what to discard on a cache miss. Many cache replacement algorithms exploit temporal locality by discarding those entries that have not been recently accessed.

All caches use the following model. Main memory is divided up into fixed-size blocks called **cache lines**. A cache line typically consists of 4 to 64 consecutive bytes. Lines are numbered consecutively starting at 0, so with a 32-byte line size, line 0 is bytes 0 to 31, line 1 is bytes 32 to 63, and so on. At any instant, some lines are in the cache. When memory is referenced, the cache controller circuit checks to see if the word referenced is currently in the cache. If so, the value there can be used, saving a trip to main memory. If the word is not there, some line entry is removed from the cache and the line needed is fetched from memory or more distant cache to replace it. Many variations on this scheme exist, but in all of them the idea is to keep the most heavily used lines in the cache as much as possible, to maximize the number of memory references satisfied out of the cache.

Direct-Mapped Caches

The simplest cache is known as a **direct-mapped cache**. An example single-level direct-mapped cache is shown in Fig. 4-38(a). This example cache contains 2048 entries. Each entry (row) in the cache can hold exactly one cache line from main memory. With a 32-byte cache line size (for this example), the cache can hold 2048 entries of 32 bytes or 64 KB in total. Each cache entry consists of three parts:

1. The Valid bit indicates whether there is any valid data in this entry or not. When the system is booted (started), all entries are marked as invalid.
2. The Tag field consists of a unique, 16-bit value identifying the corresponding line of memory from which the data came.
3. The Data field contains a copy of the data in memory. This field holds one cache line of 32 bytes.

In a direct-mapped cache, a given memory word can be stored in exactly one place within the cache. Given a memory address, there is only one place to look for it in the cache. If it is not there, then it is not in the cache. For storing and retrieving data from the cache, the address is broken into four components, as shown in Fig. 4-38(b):

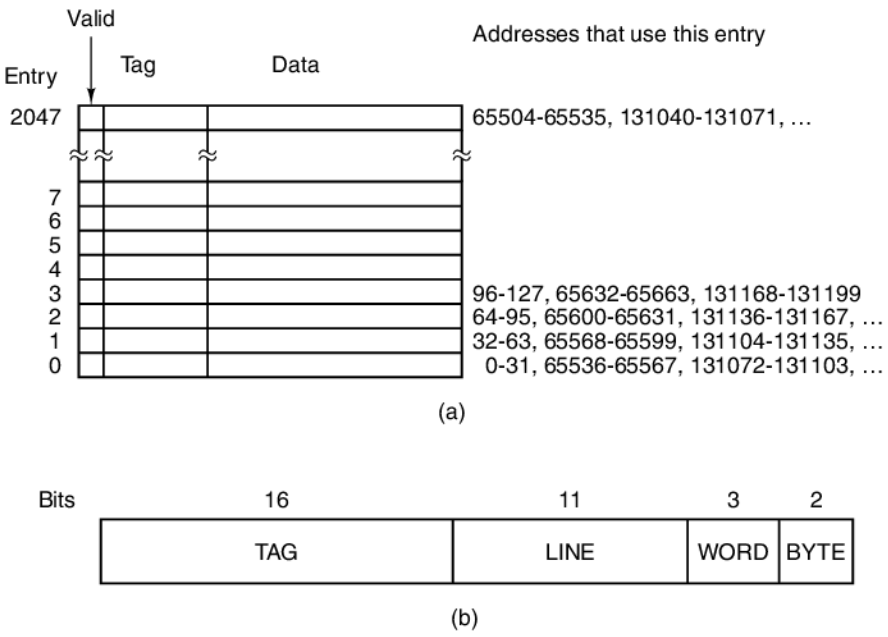


Figure 4-38. (a) A direct-mapped cache. (b) A 32-bit virtual address.

1. The TAG field corresponds to the Tag bits stored in a cache entry.
2. The LINE field indicates which cache entry holds the corresponding data, if they are present.
3. The WORD field tells which word within a line is referenced.
4. The BYTE field is usually not used, but if only a single byte is requested, it tells which byte within the word is needed. For a cache supplying only 32-bit words, this field will always be 0.

When the CPU produces a memory address, the hardware extracts the 11 LINE bits from the address and uses these to index into the cache to find one of the 2048 entries. If that entry is valid, the TAG field of the memory address and the Tag field in cache entry are compared. If they agree, the cache entry holds the word being requested, a situation called a **cache hit**. On a hit, a word being read can be taken from the cache, eliminating the need to go to memory. Only the word actually needed is extracted from the cache entry. The rest of the entry is not used. If the cache entry is invalid or the tags do not match, the needed entry is not present in the cache, a situation called a **cache miss**. In this case, the 32-byte cache line is fetched from memory and stored in the cache entry, replacing what was there. However, if the existing cache entry has been modified since being loaded, it must be written back to main memory before being overwritten.

Despite the complexity of the decision, access to a needed word can be remarkably fast. As soon as the address is known, the exact location of the word is known *if it is present in the cache*. This means that it is possible to read the word out of the cache and deliver it to the processor at the same time that it is being determined if this is the correct word (by comparing tags). So the processor actually receives a word from the cache simultaneously, or possibly even before it knows whether the word is the requested one.

This mapping scheme puts consecutive memory lines in consecutive cache entries. In fact, up to 64 KB of contiguous data can be stored in the cache. However, two lines that differ in their address by precisely 65,536 bytes or any integral multiple of that number cannot be stored in the cache at the same time (because they have the same LINE value). For example, if a program accesses data at location X and next executes an instruction that needs data at location X + 65,536 (or any other location within the same line), the second instruction will force the cache entry to be reloaded, overwriting what was there. If this happens often enough, it can result in poor behavior. In fact, the worst-case behavior of a cache is worse than if there were no cache at all, since each memory operation involves reading in an entire cache line instead of just one word.

Direct-mapped caches are the most common kind of cache, and they perform quite effectively, because collisions such as the one described above can be made to occur only rarely, or not at all. For example, a very clever compiler can take cache collisions into account when placing instructions and data in memory. Notice that the particular case described would not occur in a system with separate instruction and data caches, because the colliding requests would be serviced by different caches. Thus we see a second benefit of two caches rather than one: more flexibility in dealing with conflicting memory patterns.

Set-Associative Caches

As mentioned above, many different lines in memory compete for the same cache slots. If a program using the cache of Fig. 4-38(a) heavily uses words at addresses 0 and at 65,536, there will be constant conflicts, with each reference potentially evicting the other one from the cache. A solution is to allow two or more lines in each cache entry. A cache with n possible entries for each address is called an **n-way set-associative cache**. A four-way set-associative cache is illustrated in Fig. 4-39.

A set-associative cache is inherently more complicated than a direct-mapped cache because, although the correct set of cache entries to examine can be computed from the memory address being referenced, a set of n cache entries must be checked to see if the needed line is present. And they have to be checked very fast. Nevertheless, simulations and experience show that two-way and four-way caches perform well enough to make this extra circuitry worthwhile.

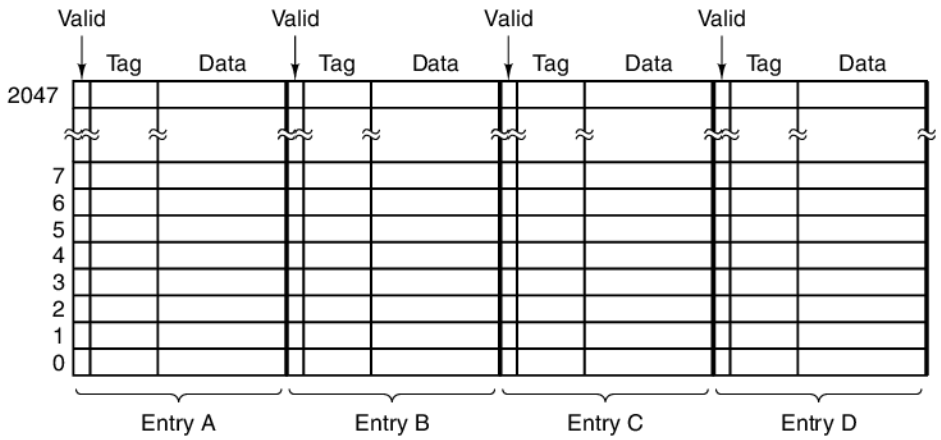


Figure 4-39. A four-way set-associative cache.

The use of a set-associative cache presents the designer with a choice. When a new entry is to be brought into the cache, which of the present items should be discarded? The optimal decision, of course, requires a peek into the future, but a pretty good algorithm for most purposes is **LRU** (**Least Recently Used**). This algorithm keeps an ordering of each set of locations that could be accessed from a given memory location. Whenever any of the present lines are accessed, it updates the list, marking that entry the most recently accessed. When it comes time to replace an entry, the one at the end of the list—the least recently accessed—is the one discarded.

Carried to the extreme, a 2048-way cache containing a single set of 2048 line entries is also possible. Here all memory addresses map onto the single set, so the lookup requires comparing the address against all 2048 tags in the cache. Note that each entry must now have tag-matching logic. Since the LINE field is of 0 length, the TAG field is the entire address except for the WORD and BYTE fields. Furthermore, when a cache line is replaced, all 2048 locations are possible candidates for replacement. Maintaining an ordered list of 2048 entries requires a great deal of bookkeeping, making LRU replacement infeasible. (Remember that this list has to be updated on every memory operation, not just on a miss.) Surprisingly, high-associativity caches do not improve performance much over low-associativity caches under most circumstances, and in some cases actually perform worse. For these reasons, set associativity beyond four-way is relatively unusual.

Finally, writes pose a special problem for caches. When a processor writes a word, and the word is in the cache, it obviously must either update the word or discard the cache entry. Nearly all designs update the cache. But what about updating the copy in main memory? This operation can be deferred until later, when the cache line is ready to be replaced by the LRU algorithm. This choice is difficult,

and neither option is clearly preferable. Immediately updating the entry in main memory is referred to as **write through**. This approach is generally simpler to implement and more reliable, since the memory is always up to date—helpful, for example, if an error occurs and it is necessary to recover the state of the memory. Unfortunately, it also usually requires more write traffic to memory, so more sophisticated implementations tend to employ the alternative, known as **write deferred**, or **write back**.

A related problem must be addressed for writes: what if a write occurs to a location that is not currently cached? Should the data be brought into the cache, or just written out to memory? Again, neither answer is always best. Most designs that defer writes to memory tend to bring data into the cache on a write miss, a technique known as **write allocation**. Most designs employing write through, on the other hand, tend not to allocate an entry on a write because this option complicates an otherwise simple design. Write allocation wins only if there are repeated writes to the same or different words within a cache line.

Cache performance is critical to system performance because the gap between CPU speed and memory speed is very large. Consequently, research on better caching strategies is still a hot topic (Sanchez and Kozyrakis, 2011, and Gaur et. al, 2011).

4.5.2 Branch Prediction

Modern computers are highly pipelined. The pipeline of Fig. 4-36 has seven stages; high-end computers sometimes have 10-stage pipelines or even more. Pipelining works best on linear code, so the fetch unit can just read in consecutive words from memory and send them off to the decode unit in advance of their being needed.

The only minor problem with this wonderful model is that it is not the slightest bit realistic. Programs are not linear code sequences. They are full of branch instructions. Consider the simple statements of Fig. 4-40(a). A variable, *i*, is compared to 0 (probably the most common test in practice). Depending on the result, another variable, *k*, gets assigned one of two possible values.

```
if (i == 0)                          CMP i,0     ; compare i to 0
    k = 1;                           BNE Else    ; branch to Else if not equal
else                        Then:    MOV k,1     ; move 1 to k
    k = 2;                           BR Next     ; unconditional branch to Next
                                Else: MOV k,2     ; move 2 to k
                                Next:
    (a)                                  (b)
```

Figure 4-40. (a) A program fragment. (b) Its translation to a generic assembly language.

A possible translation to assembly language is shown in Fig. 4-40(b). We will study assembly language later in this book, and the details are not important now, but depending on the machine and the compiler, code more or less like that of Fig. 4-40(b) is likely. The first instruction compares *i* to 0. The second one branches to the label *Else* (the start of the else clause) if *i* is not 0. The third instruction assigns 1 to *k*. The fourth instruction branches to the code for the next statement. The compiler has conveniently planted a label, *Next*, there, so there is a place to branch to. The fifth instruction assigns 2 to *k*.

The thing to observe here is that two of the five instructions are branches. Furthermore, one of these, BNE, is a conditional branch (a branch that is taken if and only if some condition is met—in this case, that the two operands in the previous CMP are unequal). The longest linear code sequence here is two instructions. As a consequence, fetching instructions at a high rate to feed the pipeline is very hard.

At first glance, it might appear that unconditional branches, such as the instruction BR Next in Fig. 4-40(b), are not a problem. After all, there is no ambiguity about where to go. Why can the fetch unit not just continue to read instructions from the target address (the place that will be branched to)?

The trouble lies in the nature of pipelining. In Fig. 4-36, for example, we see that instruction decoding occurs in the second stage. Thus the fetch unit has to decide where to fetch from next before it knows what kind of instruction it just got. Only one cycle later can it learn that it just picked up an unconditional branch, and by then it has already started to fetch the instruction following the unconditional branch. As a consequence, a substantial number of pipelined machines (such as the UltraSPARC III) have the property that the instruction *following* an unconditional branch is executed, even though logically it should not be. The position after a branch is called a **delay slot**. The Core i7 [and the machine used in Fig. 4-40(b)] do not have this property, but the internal complexity to get around this problem is often enormous. An optimizing compiler will try to find some useful instruction to put in the delay slot, but frequently there is nothing available, so it is forced to insert a NOP instruction there. Doing so keeps the program correct, but makes it bigger and slower.

Annoying as unconditional branches are, conditional branches are worse. Not only do they also have delay slots, but now the fetch unit does not know where to read from until much later in the pipeline. Early pipelined machines just **stalled** until it was known whether the branch would be taken or not. Stalling for three or four cycles on every conditional branch, especially if 20% of the instructions are conditional branches, wreaks havoc with the performance.

Consequently, what most machines do when they hit a conditional branch is predict whether it is going to be taken or not. It would be nice if we could just plug a crystal ball into a free PCIe slot (or better yet, into the IFU) to help out with the prediction, but so far this approach has not borne fruit.

Lacking such a nice peripheral, various ways have been devised to do the prediction. One very simple way is as follows: assume that all backward conditional

branches will be taken and that all forward ones will not be taken. The reasoning behind the first part is that backward branches are frequently located at the end of a loop. Most loops are executed multiple times, so guessing that a branch back to the top of the loop will be taken is generally a good bet.

The second part is shakier. Some forward branches occur when error conditions are detected in software (e.g., a file cannot be opened). Errors are rare, so most of the branches associated with them are not taken. Of course, there are plenty of forward branches not related to error handling, so the success rate is not nearly as good as with backward branches. While not fantastic, this rule is at least better than nothing.

If a branch is correctly predicted, there is nothing special to do. Execution just continues at the target address. The trouble comes when a branch is predicted incorrectly. Figuring out where to go and going there is not difficult. The hard part is undoing instructions that have already been executed and should not have been.

There are two ways of going about this. The first way is to allow instructions fetched after a predicted conditional branch to execute until they try to change the machine's state (e.g., storing into a register). Instead of overwriting the register, the value computed is put into a (secret) scratch register and only copied to the real register after it is known that the prediction was correct. The second way is to record the value of any register about to be overwritten (e.g., in a secret scratch register), so the machine can be rolled back to the state it had at the time of the mispredicted branch. Both solutions are complex and require industrial-strength bookkeeping to get them right. And if a second conditional branch is hit before it is known whether the first one was predicted right, things can get really messy.

Dynamic Branch Prediction

Clearly, having the predictions be accurate is of great value, since it allows the CPU to proceed at full speed. As a consequence, much ongoing research aims at improving branch prediction algorithms (Chen et al., 2003, Falcon et al., 2004, Jimenez, 2003, and Parikh et al., 2004). One approach is for the CPU to maintain a history table (in special hardware), in which it logs conditional branches as they occur, so they can be looked up when they occur again. The simplest version of this scheme is shown in Fig. 4-41(a). Here the history table contains one entry for each conditional branch instruction. The entry contains the address of the branch instruction along with a bit telling whether it was taken the last time it was executed. Using this scheme, the prediction is simply that the branch will go the same way it went last time. If the prediction is wrong, the bit in the history table is changed.

There are several ways to organize the history table. In fact, these are precisely the same ways used to organize a cache. Consider a machine with 32-bit instructions that are word aligned so that the low-order 2 bits of each memory address are

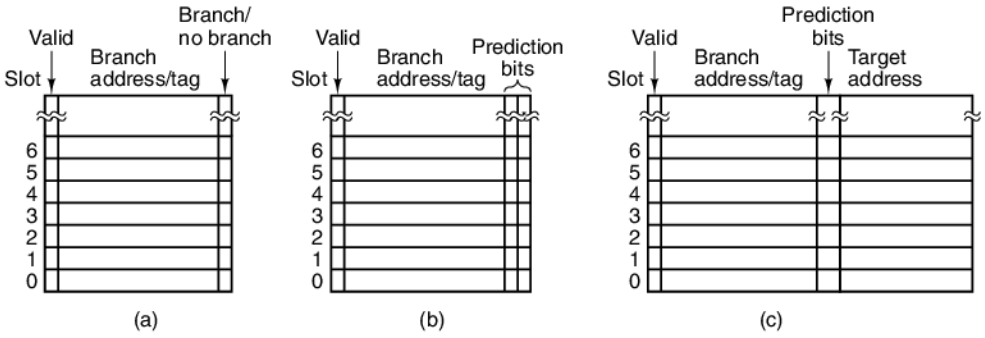


Figure 4-41. (a) A 1-bit branch history. (b) A 2-bit branch history. (c) A mapping between branch instruction address and target address.

00. With a direct-mapped history table containing 2^n entries, the low-order $n + 2$ bits of a branch instruction target address can be extracted and shifted right 2 bits. This n-bit number can be used as an index into the history table where a check is made to see if the address stored there matches the address of the branch. As with a cache, there is no need to store the low-order $n + 2$ bits, so they can be omitted (i.e., just the upper address bits—the tag—are stored). If there is a hit, the prediction bit is used to predict the branch. If the wrong tag is present or the entry is invalid, a miss occurs, just as with a cache. In this case, the forward/backward branch rule can be used.

If the branch history table has, say, 4096 entries, then branches at addresses 0, 16384, 32768, ... will conflict, analogous to the same problem with a cache. The same solution is possible: a two-way, four-way, or n-way associative entry. As with a cache, the limiting case is a single n-way associative entry, which requires full associativity of lookup.

Given a large enough table size and enough associativity, this scheme works well in most situations. However, one systematic problem always occurs. When a loop is finally exited, the branch at the end will be mispredicted, and worse yet, the misprediction will change the bit in the history table to indicate a future prediction of "no branch." The next time the loop is entered, the branch at the end of the first iteration will be predicted wrong. If the loop is inside an outer loop, or in a frequently called procedure, this error can occur often.

To eliminate this misprediction, we can give the table entry a second chance. With this method, the prediction is changed only after two consecutive incorrect predictions. This approach requires having two prediction bits in the history table, one for what the branch is "supposed" to do, and one for what it did last time, as shown in Fig. 4-41(b).

A slightly different way of looking at this algorithm is to see it as a finite-state machine with four states, as depicted in Fig. 4-42. After a series of consecutive successful "no branch" predictions, the FSM will be in state 00 and will predict

"no branch" next time. If that prediction is wrong, it will move to state 01, but predict "no branch" next time as well. Only if this prediction is wrong will it now move to state 11 and predict branches all the time. In effect, the leftmost bit of the state is the prediction and the rightmost bit is what the branch did last time. While this design uses only 2 bits of history, a design that keeps track of 4 or 8 bits of history is also possible.

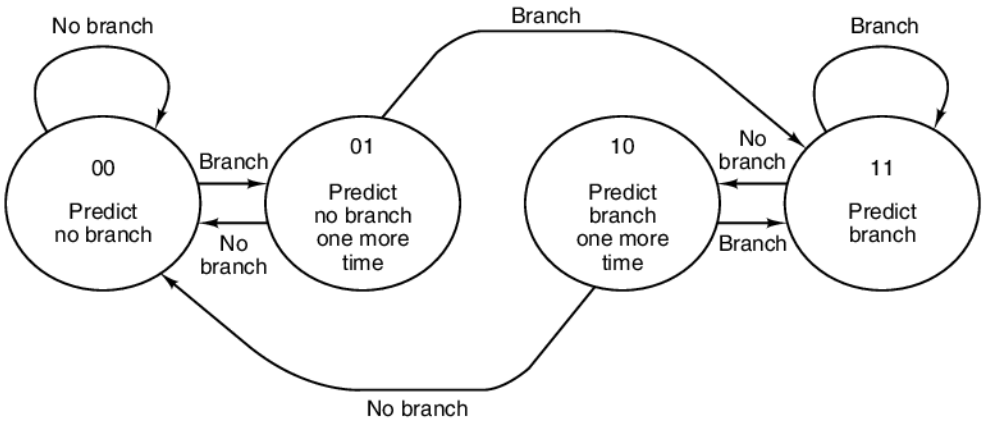


Figure 4-42. A 2-bit finite-state machine for branch prediction.

This is not our first FSM. Figure 4-28 was also an FSM. In fact, all of our microprograms can be regarded as FSMs, since each line represents a specific state the machine can be in, with well-defined transitions to a finite set of other states. FSMs are very widely used in all aspects of hardware design.

So far, we have assumed that the target of each conditional branch was known, typically either as an explicit address to branch to (contained within the instruction itself), or as a relative offset from the current instruction (i.e., a signed number to add to the program counter). Often this assumption is valid, but some conditional branch instructions compute the target address by doing arithmetic on registers, and then going there. Even if the FSM of Fig. 4-42 accurately predicts the branch will be taken, such a prediction is of no use if the target address is unknown. One way of dealing with this situation is to store the actual address branched to the last time in the history table, as shown in Fig. 4-41(c). In this way, if the table says that the last time the branch at address 516 was taken it went to address 4000, if the prediction is now for "branch," the working assumption will be a branch to 4000 again.

A different approach to branch prediction is to keep track of whether the last k conditional branches encountered were taken, irrespective of which instructions they were. This k-bit number, kept in the **branch history shift register**, is then compared in parallel to all the entries of a history table with a k-bit key and, if a hit occurs, the prediction found there used. Somewhat surprisingly, this technique works quite well in actual practice.

Static Branch Prediction

All of the branch prediction techniques discussed so far are dynamic, that is, are carried out at run time while the program is running. They also adapt to the program's current behavior, which is good. The down side is that they require specialized and expensive hardware and a great deal of chip complexity.

A different way to go is to have the compiler help out. When the compiler sees a statement like

```
for (i = 0; i < 1000000; i++) { ... }
```

it knows very well that the branch at the end of the loop will be taken nearly all the time. If only it had a way to tell the hardware, a lot of effort could be saved.

Although this is an architectural change (and not just an implementation issue), some machines, such as the UltraSPARC III, have a second set of conditional branch instructions, in addition to the regular ones (which are needed for backward compatibility). The new ones contain a bit in which the compiler can specify that it thinks the branch will be taken (or not taken). When one of these is encountered, the fetch unit just does what it has been told. Furthermore, there is no need to waste precious space in the branch history table for these instructions, thus reducing conflicts there.

Finally, our last branch prediction technique is based on profiling (Fisher and Freudenberger, 1992). This, too, is a static technique, but instead of having the compiler try to figure out which branches will be taken and which will not, the program is actually run (typically on a simulator) and the branch behavior captured. This information is fed into the compiler, which then uses the special conditional branch instructions to tell the hardware what to do.

4.5.3 Out-of-Order Execution and Register Renaming

Most modern CPUs are both pipelined and superscalar, as shown in Fig. 2-6. What this generally means is that a fetch unit pulls instruction words out of memory before they are needed in order to feed a decode unit. The decode unit issues the decoded instructions to the proper functional units for execution. In some cases it may break individual instructions into micro-ops before issuing them, depending on what the functional units can do.

Clearly, the machine design is simplest if all instructions are executed in the order they are fetched (assuming for the moment that the branch prediction algorithm never guesses wrong). However, in-order execution does not always give optimal performance due to dependences between instructions. If an instruction needs a value computed by the previous instruction, the second one cannot begin executing until the first one has produced the needed value. In this situation (a RAW dependence), the second instruction has to wait. Other kinds of dependences also exist, as we will soon see.

In an attempt to get around these problems and produce better performance, some CPUs allow dependent instructions to be skipped over, to get to future instructions that are not dependent. Needless to say, the internal instruction-scheduling algorithm used must deliver the same effect as if the program were executed in the order written. We will now demonstrate how instruction reordering works using a detailed example.

To illustrate the nature of the problem, we will start with a machine that always issues instructions in program order and also requires them to complete execution in program order. The significance of the latter will become clear later.

Our example machine has eight registers visible to the programmer, R0 through R7. All arithmetic instructions use three registers: two for the operands and one for the result, the same as the Mic-4. We will assume that if an instruction is decoded in cycle n, execution starts in cycle $n + 1$. For a simple instruction, such as an addition or subtraction, the writeback to the destination register occurs at the end of cycle $n + 2$. For a more complicated instruction, such as a multiplication, the writeback occurs at the end of cycle $n + 3$. To make the example realistic, we will allow the decode unit to issue up to two instructions per clock cycle. Commercial superscalar CPUs often can issue four or even six instructions per clock cycle.

Our example execution sequence is shown in Fig. 4-43. Here the first column gives the number of the cycle and the second one gives the instruction number. The third column lists the instruction decoded. The fourth one tells which instruction is being issued (with a maximum of two per clock cycle). The fifth one tells which instruction has been retired (completed). Remember that in this example we are requiring both in-order issue and in-order completion, so instruction $k + 1$ cannot be issued until instruction k has been issued, and instruction $k + 1$ cannot be retired (meaning the writeback to the destination register is performed) until instruction k has been retired. The other 16 columns are discussed below.

After decoding an instruction, the decode unit has to decide whether or not it can be issued immediately. To make this decision, the decode unit needs to know the status of all the registers. If, for example, the current instruction needs a register whose value has not yet been computed, the current instruction cannot be issued and the CPU must stall.

We will keep track of register use with a device called a **scoreboard**, which was first present in the CDC 6600. The scoreboard has a small counter for each register telling how many times that register is in use as a source by currently executing instructions. If a maximum of, say, 15 instructions may be executing at once, then a 4-bit counter will do. When an instruction is issued, the scoreboard entries for its operand registers are incremented. When an instruction is retired, the entries are decremented.

The scoreboard also has counters to keep track of registers being used as destinations. Since only one write at a time is allowed, these counters can be 1-bit wide. The rightmost 16 columns in Fig. 4-43 show the scoreboard.

					Registers being read								Registers being written							
Cy	**#**	**Decoded**	**Iss**	**Ret**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0–R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1							1						
	7	R3=R3*R1	–		1		1							1						
10					1		1							1						
11				6																
12			7			1		1								1				
	8	R1=R4+R4	–			1		1								1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2					1						
17									2					1						
18				8																

Figure 4-43. A superscalar CPU with in-order issue and in-order completion.

In real machines, the scoreboard also keeps track of functional unit usage, to avoid issuing an instruction for which no functional unit is available. For simplicity, we will assume there is always a suitable functional unit available, so we will not show the functional units on the scoreboard.

The first line of Fig. 4-43 shows I1 (instruction 1), which multiplies R0 by R1 and puts the result in R3. Since none of these registers are in use yet, the instruction is issued and the scoreboard is updated to reflect that R0 and R1 are being read and R3 is being written. No subsequent instruction can write into any of these or can read R3 until I1 has been retired. Since this instruction is a multiplication, it will be finished at the end of cycle 4. The scoreboard values shown on each line reflect their state after the instruction on that line has been issued. Blanks are 0s.

Since our example is a superscalar machine that can issue two instructions per cycle, a second instruction (I2) is issued during cycle 1. It adds R0 and R2, storing the result in R4. To see if this instruction can be issued, these rules are applied:

1. If any operand is being written, do not issue (RAW dependence).
2. If the result register is being read, do not issue (WAR dependence).
3. If the result register is being written, do not issue (WAW dependence).

We have already seen RAW dependences, which occur when an instruction needs to use as a source a result that a previous instruction has not yet produced. The other two dependences are less serious. They are essentially resource conflicts. In a **WAR dependence** (Write After Read), one instruction is trying to overwrite a register that a previous instruction may not yet have finished reading. A **WAW dependence** (Write After Write) is similar. These can often be avoided by having the second instruction put its results somewhere else (perhaps temporarily). If none of the above three dependences exist, and the functional unit it needs is available, the instruction is issued. In this case, I2 uses a register (R0) that is being read by a pending instruction, but this overlap is permitted so I2 is issued. Similarly, I3 is issued during cycle 2.

Now we come to I4, which needs to use R4. Unfortunately, we see from line 3 that R4 is being written. Here we have a RAW dependence, so the decode unit stalls until R4 becomes available. While stalled, it stops pulling instructions from the fetch unit. When the fetch unit's internal buffers fill up, it stops prefetching.

It is worth noting that the next instruction in program order, I5, does not have conflicts with any of the pending instructions. It could have been decoded and issued were it not for the fact that this design requires issuing instructions in order.

Now let us look at what happens during cycle 3. I2, being an addition (two cycles), finishes at the end of cycle 3. Unfortunately, it cannot be retired (thus freeing up R4 for I4). Why not? The reason is that this design also requires in-order retirement. Why? What harm could possibly come from doing the store into R4 now and marking it as available?

The answer is subtle, but important. Suppose that instructions could complete out of order. Then if an interrupt occurred, it would be difficult to save the state of the machine so it could be restored later. In particular, it would not be possible to say that all instructions up to some address had been executed and all instructions beyond it had not. This is called a **precise interrupt** and is a desirable characteristic in a CPU (Moudgill and Vassiliadis, 1996). Out-of-order retirement makes interrupts imprecise, which is why some machines complete instructions in order.

Getting back to our example, at the end of cycle 4, all three pending instructions can be retired, so in cycle 5 I4 can finally be issued, along with the newly decoded I5. Whenever an instruction is retired, the decode unit has to check to see if there is a stalled instruction that can now be issued.

In cycle 6, I6 stalls because it needs to write into R1 and R1 is busy. It is finally started in cycle 9. The entire sequence of eight instructions takes 18 cycles to complete due to many dependences, even though the hardware is capable of issuing two instructions on every cycle. Notice, however, when reading down the *Iss* column of Fig. 4-43, that all the instructions have been issued in order. Likewise, the *Ret* column shows that they have been retired in order as well.

Now let us consider an alternative design: out-of-order execution. In this design, instructions may be issued out of order and may be retired out of order as well. The same sequence of eight instructions is shown in Fig. 4-44, only now with out-of-order issue and out-of-order retirement permitted.

					Registers being read								Registers being written							
Cy	**#**	**Decoded**	**Iss**	**Ret**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3	5	R7=R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0–R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*S1	–		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Figure 4-44. Operation of a superscalar CPU with out-of-order issue and out-of-order completion.

The first difference occurs in cycle 3. Even though I4 has stalled, we are allowed to decode and issue I5 since it does not conflict with any pending instruction. However, skipping over instructions causes a new problem. Suppose that I5 had used an operand computed by the skipped instruction, I4. With the current scoreboard, we would not have noticed this. As a consequence, we have to extend the scoreboard to keep track of stores done by skipped-over instructions. This can be done by adding a second bit map, 1 bit per register, to keep track of stores done

by stalled instructions. (These counters are not shown in the figure.) The rule for issuing instructions now has to be extended to prevent the issue of any instruction with an operand scheduled to be stored into by an instruction that came before it but was skipped over.

Now let us look back at I6, I7, and I8 in Fig. 4-43. Here we see that I6 computes a value in R1 that is used by I7. However, we also see that the value is never used again because I8 overwrites R1. There is no real reason to use R1 as the place to hold the result of I6. Worse yet, R1 is a terrible choice of intermediate register, although a perfectly reasonable one for a compiler or programmer used to the idea of sequential execution with no instruction overlap.

In Fig. 4-44 we introduce a new technique for solving this problem: **register renaming**. The wise decode unit changes the use of R1 in I6 (cycle 3) and I7 (cycle 4) to a secret register, S1, not visible to the programmer. Now I6 can be issued concurrently with I5. Modern CPUs often have dozens of secret registers for use with register renaming. This technique can often eliminate WAR and WAW dependences.

At I8, we use register renaming again. This time R1 is renamed into S2 so the addition can be started before R1 is free, at the end of cycle 6. If it turns out that the result really has to be in R1 this time, the contents of S2 can always be copied back there just in time. Even better, all future instructions needing it can have their sources renamed to the register where it really is stored. In any case, the I8 addition got to start earlier this way.

On many real machines, renaming is deeply embedded in the way the registers are organized. There are many secret registers and a table that maps the registers visible to the programmer onto the secret registers. Thus the real register being used for, say, R0 is located by looking at entry 0 of this mapping table. In this way, there is no real register R0, just a binding between the name R0 and one of the secret registers. This binding changes frequently during execution to avoid dependences.

Notice in Fig. 4-44, when reading down the fourth column, that the instructions have not been issued in order. Nor they have been retired in order. The conclusion of this example is simple: using out-of-order execution and register renaming, we were able to speed up the computation by a factor of two.

4.5.4 Speculative Execution

In the previous section we introduced the concept of reordering instructions in order to improve performance. Although we did not mention it explicitly, the focus there was on reordering instructions within a single basic block. It is now time to look at this point more closely.

Computer programs can be broken up into **basic blocks**, each consisting of a linear sequence of code with one entry point on top and one exit on the bottom. A basic block does not contain any control structures (e.g., if statements or while

statements) so that its translation into machine language does not contain any branches. The basic blocks are connected by control statements.

A program in this form can be represented as a directed graph, as shown in Fig. 4-45. Here we compute the sum of the cubes of the even and odd integers up to some limit and accumulate them in *evensum* and *oddsum*, respectively. Within each basic block, the reordering techniques of the previous section work fine.

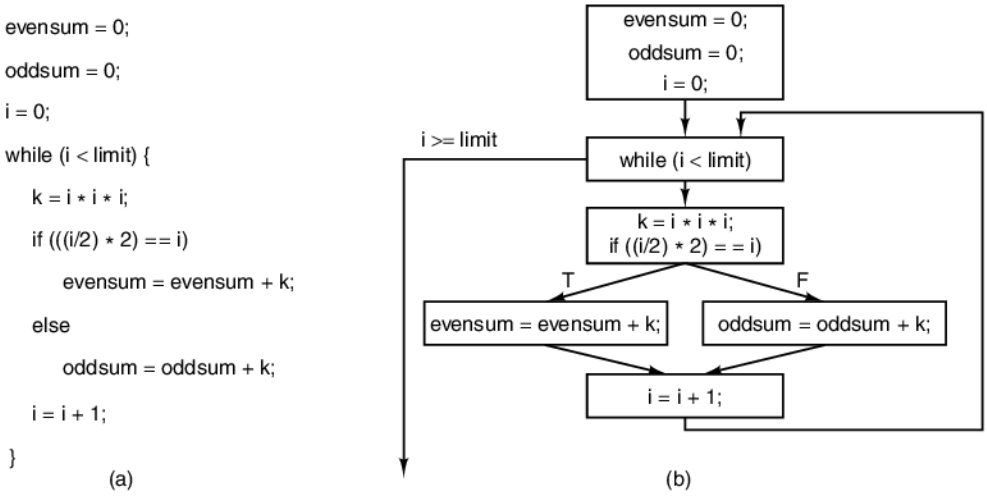


Figure 4-45. (a) A program fragment. (b) The corresponding basic block graph.

The trouble is that most basic blocks are short and there is insufficient parallelism in them to exploit effectively. Consequently, the next step is to allow the reordering to cross basic block boundaries in an attempt to fill all the issue slots. The biggest gains come when a potentially slow operation can be moved upward in the graph to start it early. This might be a LOAD instruction, a floating-point operation, or even the start of a long dependence chain. Moving code upward over a branch is called **hoisting**.

Imagine that in Fig. 4-45 all the variables were kept in registers except *evensum* and *oddsum* (for lack of registers). It might make sense then to move their LOAD instructions to the top of the loop, before computing *k*, to get them started early on, so the values will be available when needed. Of course, only one of them will be needed on each iteration, so the other LOAD will be wasted, but if the cache and memory are pipelined and there are issue slots available, it might still be worth doing this. Executing code before it is known if it is even going to be needed is called **speculative execution**. Using this technique requires support from the compiler and the hardware as well as some architectural extensions. Normally, reordering instructions over basic block boundaries is beyond the capability of hardware, so the compiler must move the instructions explicitly.

Speculative execution introduces some interesting problems. For one, it is essential that none of the speculative instructions have irrevocable results because it may turn out later that they should not have been executed. In Fig. 4-45, it is fine to fetch *evensum* and *oddsum*, and it is also fine to do the addition as soon as *k* is available (even before the if statement), but it is not fine to store the results back in memory. In more complicated code sequences, one common way of preventing speculative code from overwriting registers before it is known if this is desired, is to rename all the destination registers used by the speculative code. In this way, only scratch registers are modified, so there is no problem if the code ultimately is not needed. If the code is needed, the scratch registers are copied to the true destination registers. As you can imagine, the scoreboarding to keep track of all this is not simple, but given enough hardware, it can be done.

However, there is another problem introduced by speculative code that cannot be solved by register renaming. What happens if a speculatively executed instruction causes an exception? A painful, but not fatal, example is a LOAD instruction that causes a cache miss on a machine with a large cache line size (say, 256 bytes) and a memory far slower than the CPU and cache. If a LOAD that is actually needed stops the machine dead in its tracks for many cycles while the cache line is being loaded, well, that's life, since the word is needed. However, stalling the machine to fetch a word that turns out not to be needed is counterproductive. Too many of these "optimizations" may make the CPU slower than if it did not have them at all. (If the machine has virtual memory, which is discussed in Chap. 6, a speculative LOAD might even cause a page fault, which requires a disk operation to bring in the needed page. False page faults can have a terrible effect on performance, so it is important to avoid them.)

One solution present in a number of modern machines is to have a special SPECULATIVE-LOAD instruction that tries to fetch the word from the cache, but if it is not there, just gives up. If the value is there when it is actually needed, it can be used, but if it is not, the hardware must go out and get it on the spot. If the value turns out not to be needed, no penalty has been paid for the cache miss.

A far worse situation can be illustrated with the following statement:

```
if (x > 0) z = y/x;
```

where *x*, *y*, and *z* are floating-point variables. Suppose that the variables are all fetched into registers in advance and that the (slow) floating-point division is hoisted above the if test. Unfortunately, *x* is 0 and the resulting divide-by-zero trap terminates the program. The net result is that speculation has caused a correct program to fail. Worse yet, the programmer put in explicit code to prevent this situation and it happened anyway. This situation is not likely to lead to a happy programmer.

One possible solution is to have special versions of instructions that might cause exceptions. In addition, a bit, called a **poison bit**, is added to each register. When a special speculative instruction fails, instead of causing a trap, it sets the

poison bit on the result register. If that register is later touched by a regular instruction, the trap occurs then (as it should). However, if the result is never used, the poison bit is eventually cleared and no harm is done.

4.6 EXAMPLES OF THE MICROARCHITECTURE LEVEL

In this section, we will show brief examples of three state-of-the-art processors, showing how they employ the concepts explored in this chapter. These will of necessity be brief because real machines are enormously complex, containing millions of gates. The examples are the same ones we have been using so far: Core i7, the OMAP4430, and the ATmega168.

4.6.1 The Microarchitecture of the Core i7 CPU

On the outside, the Core i7 appears to be a traditional CISC machine, with processors that support a huge and unwieldy instruction set supporting 8-, 16-, and 32-bit integer operations as well as 32-bit and 64-bit floating-point operations. It has only eight visible registers per processor and no two of them are quite the same. Instruction lengths vary from 1 to 17 bytes. In short, it is a legacy architecture that seems to do everything wrong.

However, on the inside, the Core i7 contains a modern, lean-and-mean, deeply pipelined RISC core that runs at an extremely fast clock rate that is likely to increase in the years ahead. It is quite amazing how the Intel engineers managed to build a state-of-the-art processor to implement an ancient architecture. In this section we will look at the Core i7 microarchitecture to see how it works.

Overview of the Core i7's Sandy Bridge Microarchitecture

The Core i7 microarchitecture, called the **Sandy Bridge** microarchitecture, is a significant refinement of the previous-generation Intel microarchitectures, including the earlier P4 and P6. A rough overview of the Core i7 microarchitecture is given in Fig. 4-46.

The Core i7 consists of four major subsections: the memory subsystem, the front end, the out-of-order control, and the execution units. Let us examine these one at a time starting at the upper left and going counterclockwise around the chip.

Each processor in the Core i7 contains a memory subsystem with a unified L2 (level 2) cache as well as the logic for accessing the L3 (level 3) cache. A single large L3 cache is shared by all processors, and it is the last stop before leaving the CPU chip and making the very long trip to external RAM over the memory bus. The Core i7's L2 caches are 256 KB in size, and each is organized as an 8-way

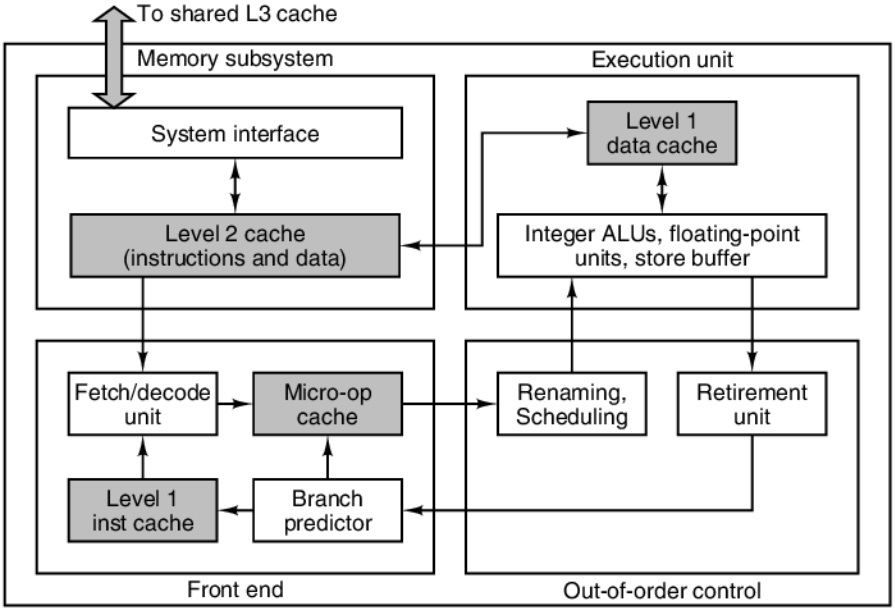


Figure 4-46. The block diagram of the Core i7's Sandy Bridge microarchitecture.

associative cache with 64-byte cache lines. The shared L3 cache varies in size from 1 MB to 20 MB. If you pay more cash to Intel you get more cache in return. Regardless of its size, the L3 is organized as a 12-way associative cache with 64-byte cache lines. In the event that an access to the L3 cache misses, it is sent to external RAM via the DDR3 RAM bus.

Associated with the L1 cache are two prefetch units (not shown in the figure) that attempt to prefetch data from lower levels of the memory system into the L1 before they are needed. One prefetch unit prefetches the next memory block when it detects that a sequential "stream" of memory is being fetched into the processor. A second, more sophisticated, prefetcher tracks the sequence of addresses from specific program loads and stores. If they progress in a regular stride (e.g., 0x1000...0x1020...0x1040...) it will prefetch the next element likely to be accessed in advance of the program. This stride-oriented prefetching does wonders for programs that are marching through arrays of structured variables.

The memory subsystem in Fig. 4-46 is connected to both the front end and the L1 data cache. The front end is responsible for fetching instructions from the memory subsystem, decoding them into RISC-like micro-ops, and storing them into two instruction storage caches. All instructions fetched are placed into the L1 (level 1) instruction cache. The L1 cache is 32 KB in size, organized as an 8-way associative cache with 64-byte blocks. As instructions are fetched from the L1 cache, they enter the decoders which determine the sequence of micro-ops used to

implement the instruction in the execution pipeline. This decoder mechanism bridges the gap between an ancient CISC instruction set and a modern RISC data path.

The decoded micro-ops are fed into the **micro-op cache**, which Intel refers to as the L0 (level 0) instruction cache. The micro-op cache is similar to a traditional instruction cache, but it has a lot of extra breathing room to store the micro-op sequences that individual instructions produce. When the decoded micro-ops rather than the original instructions are cached, there is no need to decode the instruction on subsequent executions. At first glance, you might think that Intel did this to speed up the pipeline (and indeed it does speed up the process of producing an instruction), but Intel claims that the micro-op cache was added to reduce front end power consumption. With the micro-op cache in place, the remainder of the front end sleeps in an unclocked low-power mode 80% of the time.

Branch prediction is also performed in the front end. The branch predictor is responsible for guessing when the program flow breaks from pure sequential fetching, and it must be able to do this long before the branch instructions are executed. The branch predictor in the Core i7 is quite remarkable. Unfortunately for us, the specifics of processor branch predictors are closely held secrets for most designs. This is because the performance of the predictor is often the most critical component to the overall speed of the design. The more prediction accuracy designers can squeeze out of each square micrometer of silicon, the better the performance of the entire design. As such, companies hide these secrets under lock and key and even threaten employees with criminal prosecution should any of them decide to share these jewels of knowledge. Suffice it to say, though, that all of them keep track of which way previous branches went and use this to make predictions. It is the details of precisely what they record and how they store and look up the information that is top secret. After all, if you had a fantastic way to predict the future, you probably would not put it on the Web for the whole world to see.

Instructions are fed from the micro-op cache to the out-of-order scheduler in the order dictated by the program, but they are not necessarily issued in program order. When a micro-op that cannot be executed is encountered, the scheduler holds it but continues processing the instruction stream to issue subsequent instructions all of whose resources (registers, functional units, etc.) are available. Register renaming is also done here to allow instructions with a WAR or WAW dependence to continue without delay.

Although instructions can be issued out of order, the Core i7 architecture's requirement of precise interrupts means that the ISA instructions must be retired (i.e., have their results made visible) in original program order. The retirement unit handles this chore.

In the back end of the processor we have the execution units, which carry out the integer, floating-point, and specialized instructions. Multiple execution units exist and run in parallel. They get their data from the register file and the L1 data cache.