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.

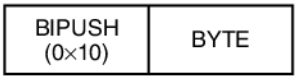


Figure 4-18. The BIPUSH instruction format.

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).

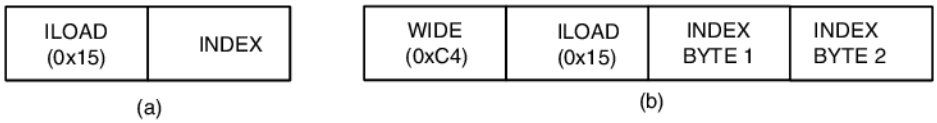


Figure 4-19. (a) ILOAD with a 1-byte index. (b) WIDE ILOAD with a 2-byte index.

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

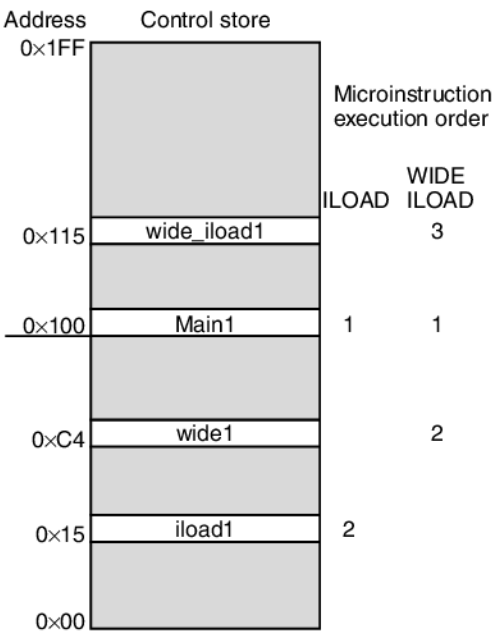


Figure 4-20. The initial microinstruction sequence for ILOAD and WIDE ILOAD. The addresses are examples.

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

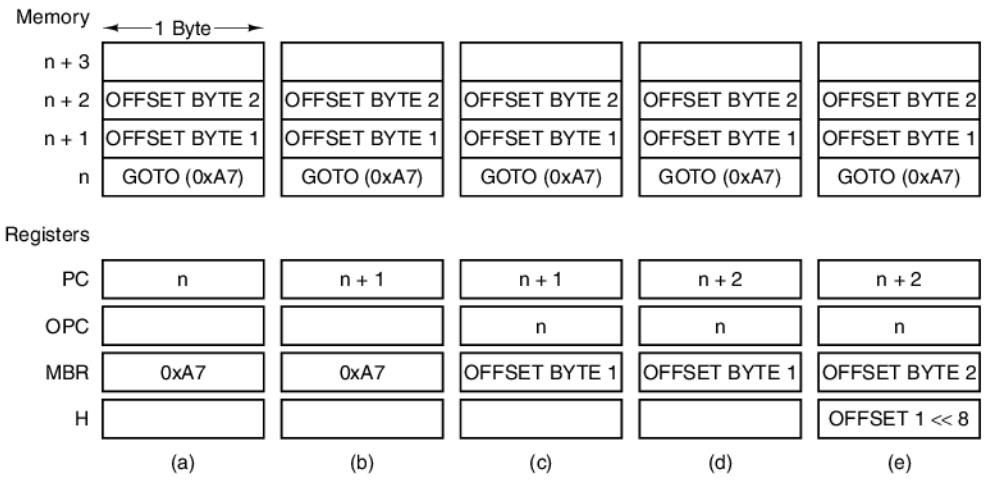


Figure 4-22. The situation at the start of various microinstructions. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

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.

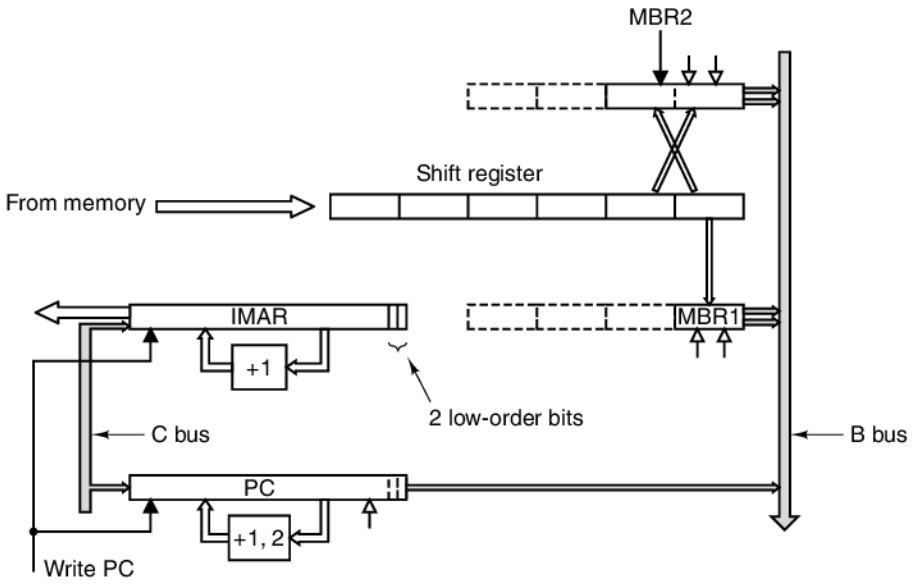


Figure 4-27. A fetch unit for the Mic-1.

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.

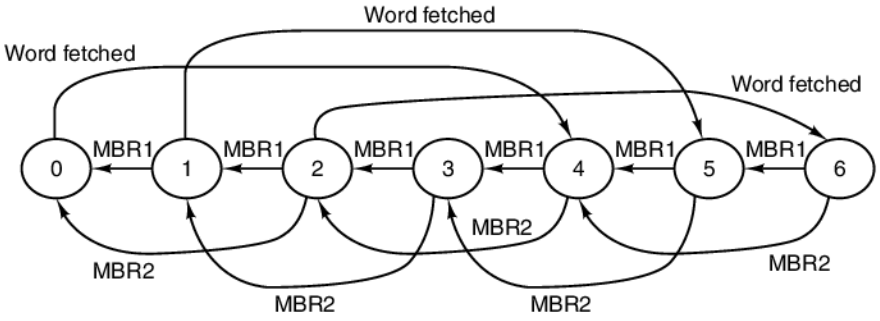
The IFU is responsible for fetching a stream of bytes. It does this by using a conventional 4-byte memory port, fetching entire 4-byte words ahead of time and loading the consecutive bytes into a shift register that supplies them one or two at a time, in the order fetched. The function of the shift register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

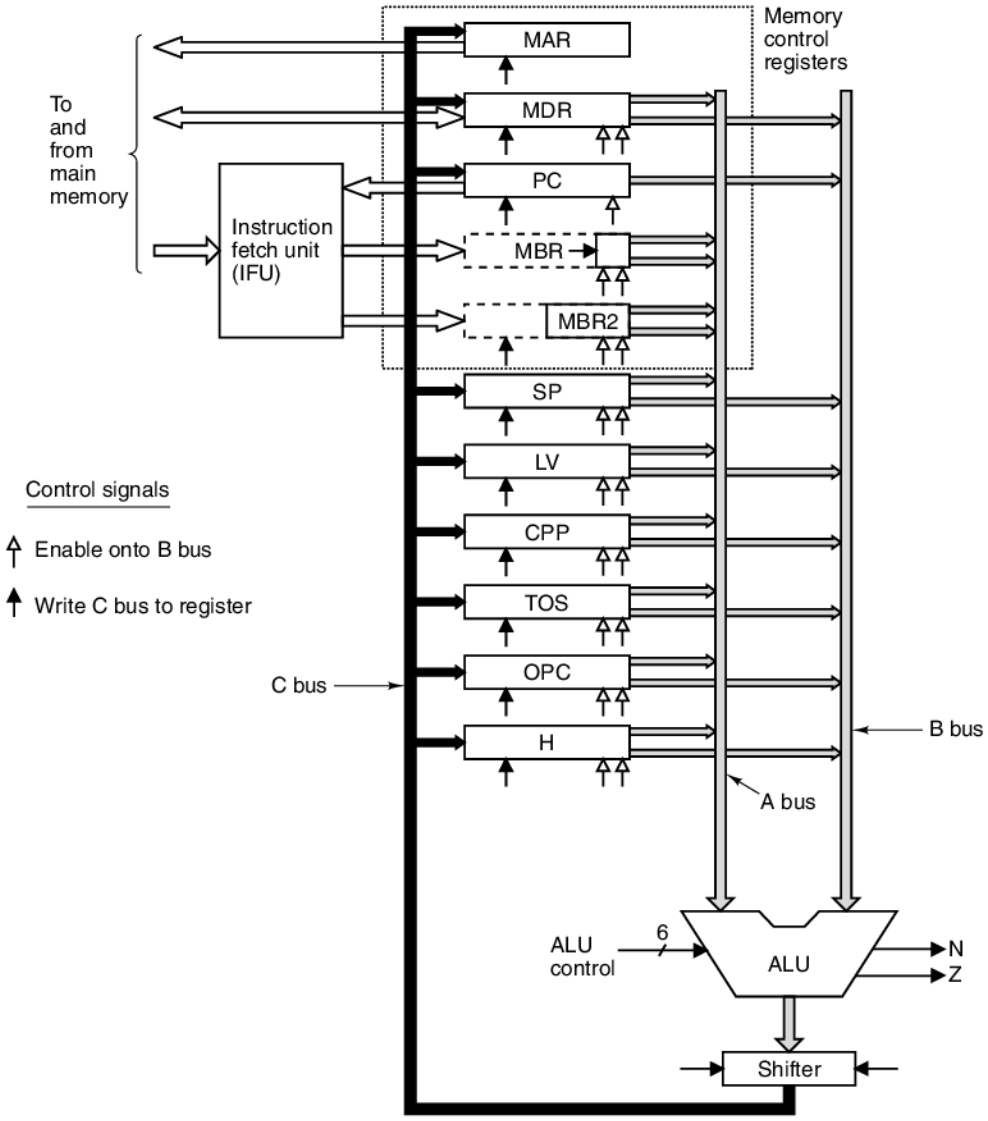


Figure 4-29. The data path for Mic-2.

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

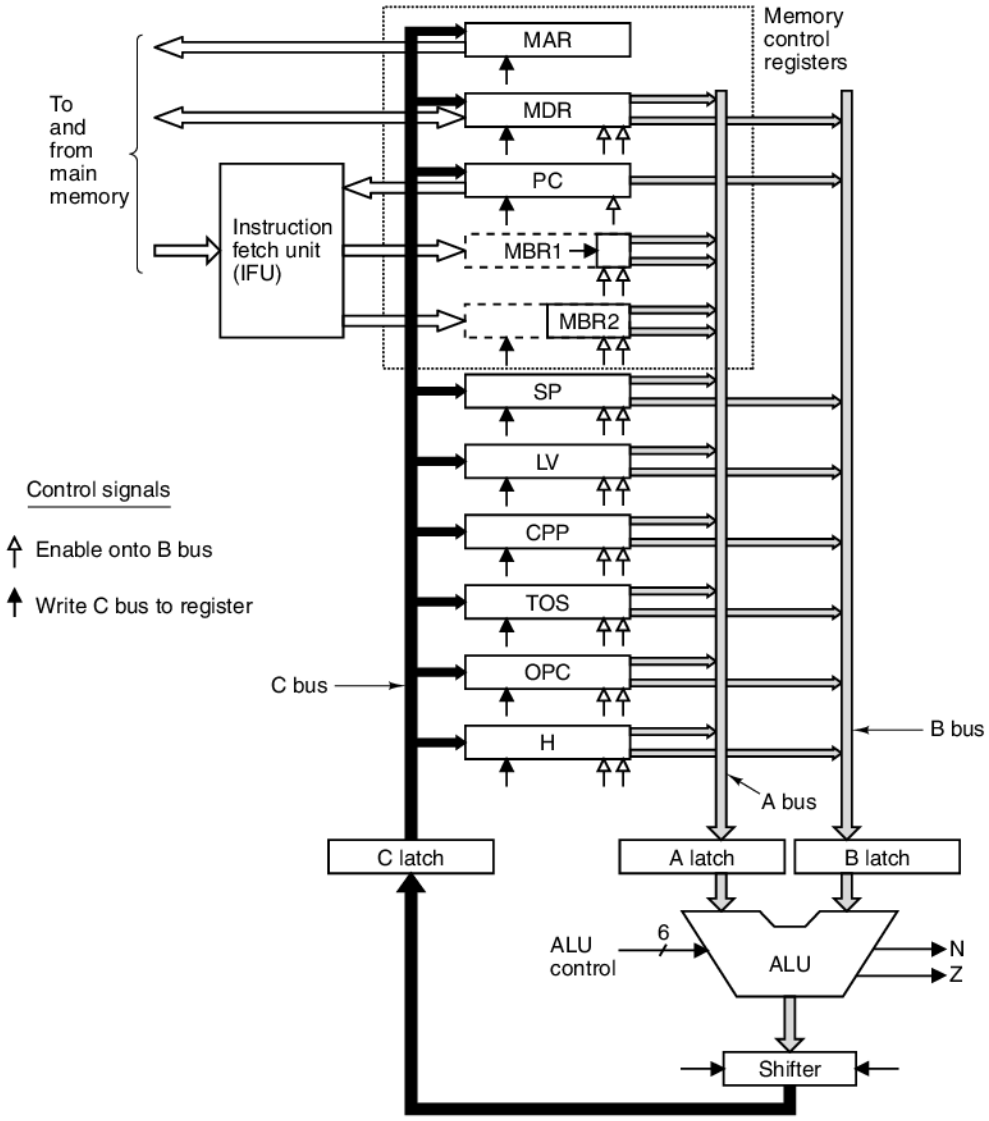


Figure 4-31. The three-bus data path used in the Mic-3.

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

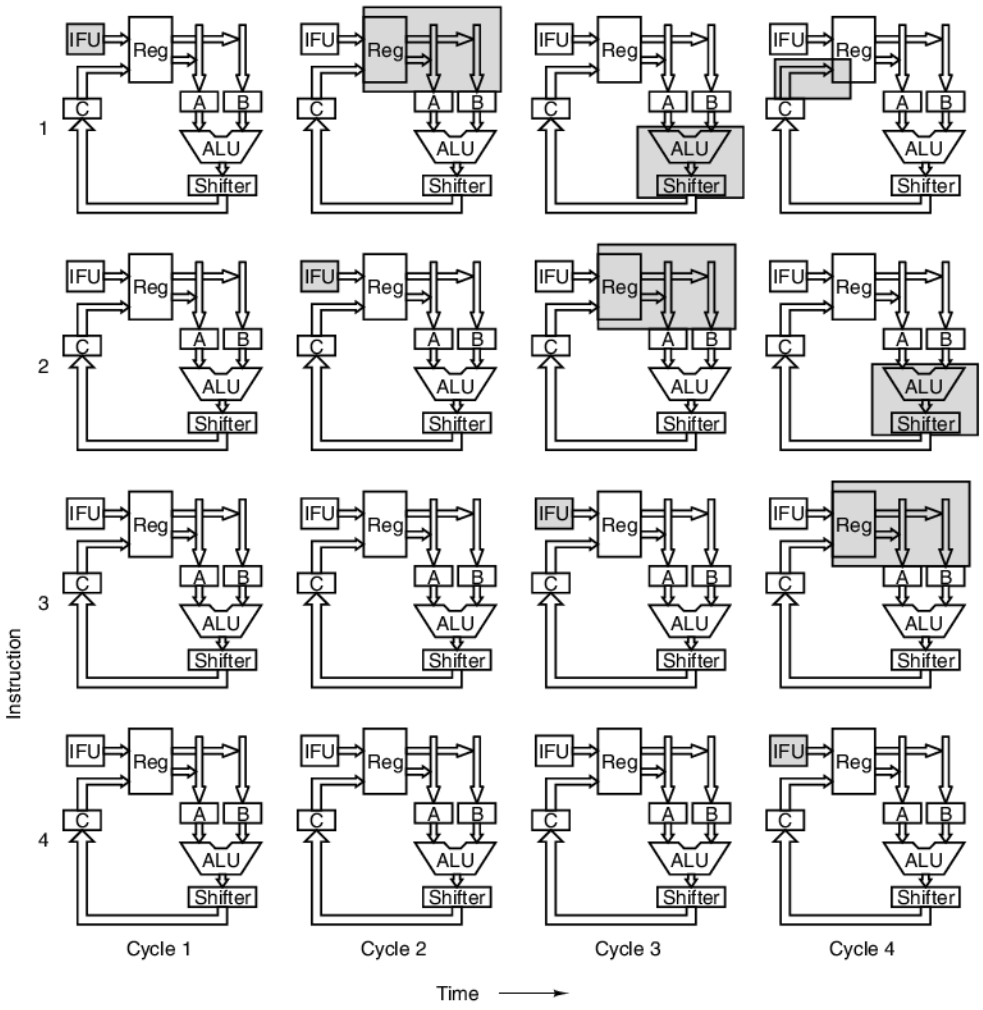


Figure 4-34. Graphical illustration of how a pipeline works.

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.

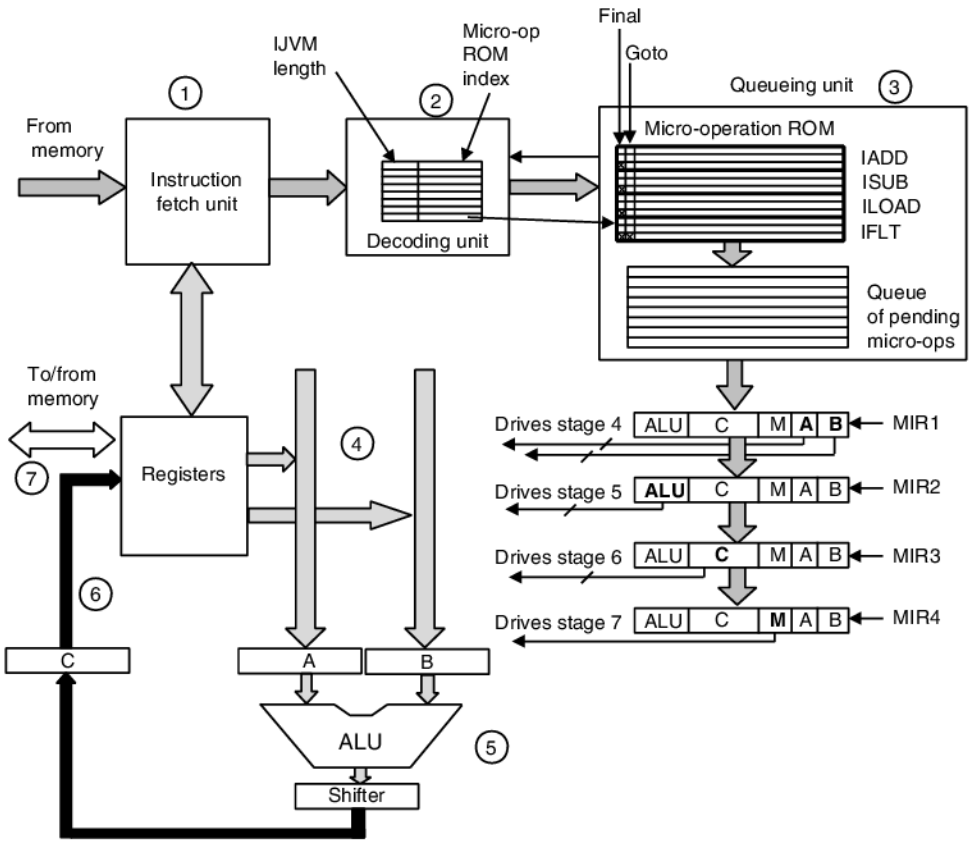


Figure 4-35. The main components of the Mic-4.

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.

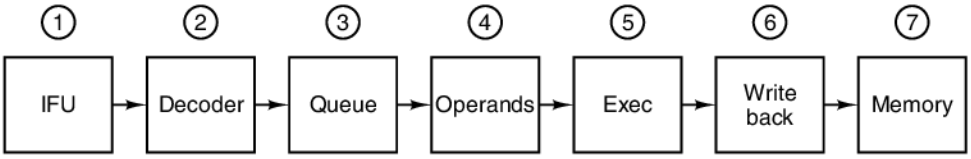


Figure 4-36. The Mic-4 pipeline.

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.

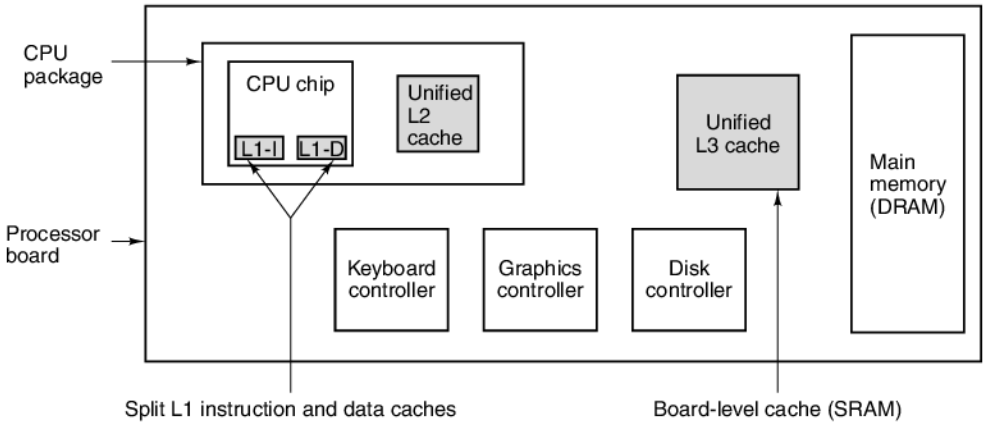


Figure 4-37. A system with three levels of cache.

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):

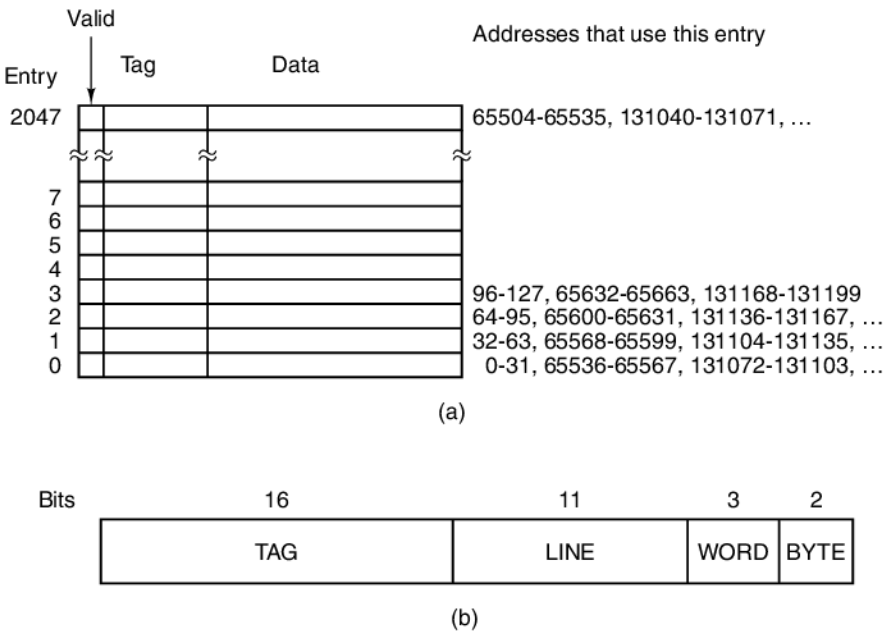


Figure 4-38. (a) A direct-mapped cache. (b) A 32-bit virtual address.

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.

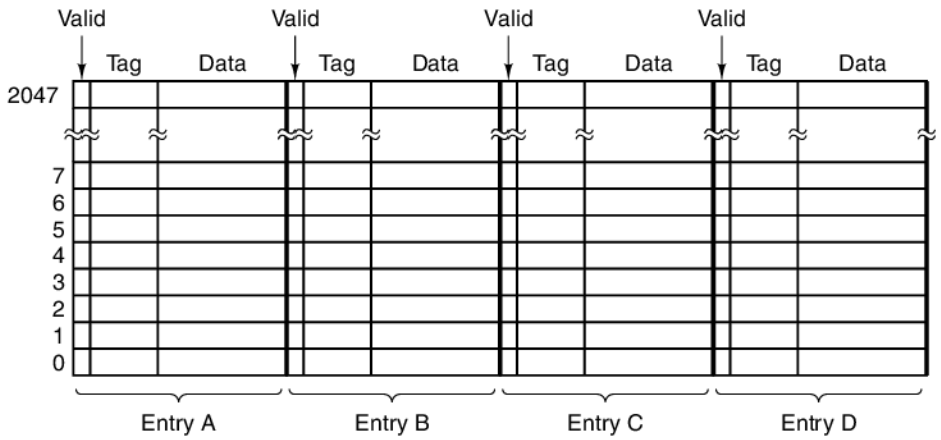


Figure 4-39. A four-way set-associative cache.

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

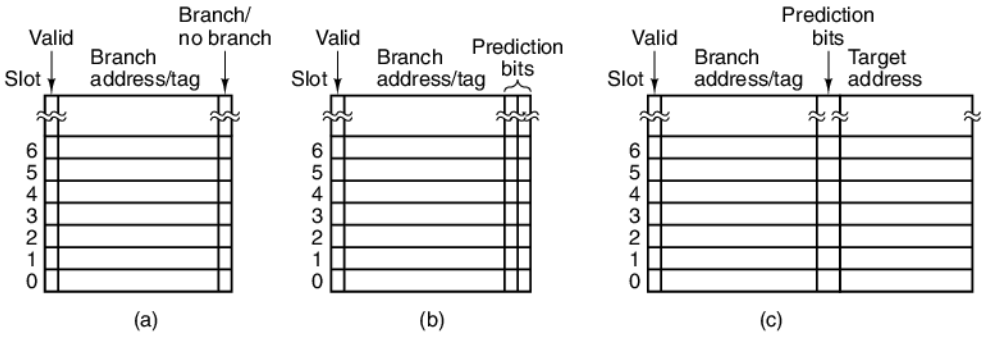


Figure 4-41. (a) A 1-bit branch history. (b) A 2-bit branch history. (c) A mapping between branch instruction address and target address.

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.

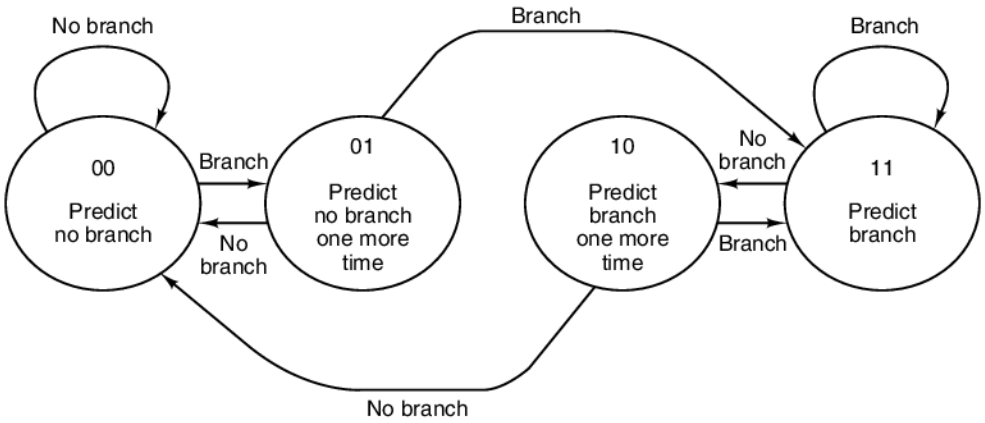


Figure 4-42. A 2-bit finite-state machine for branch prediction.

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.

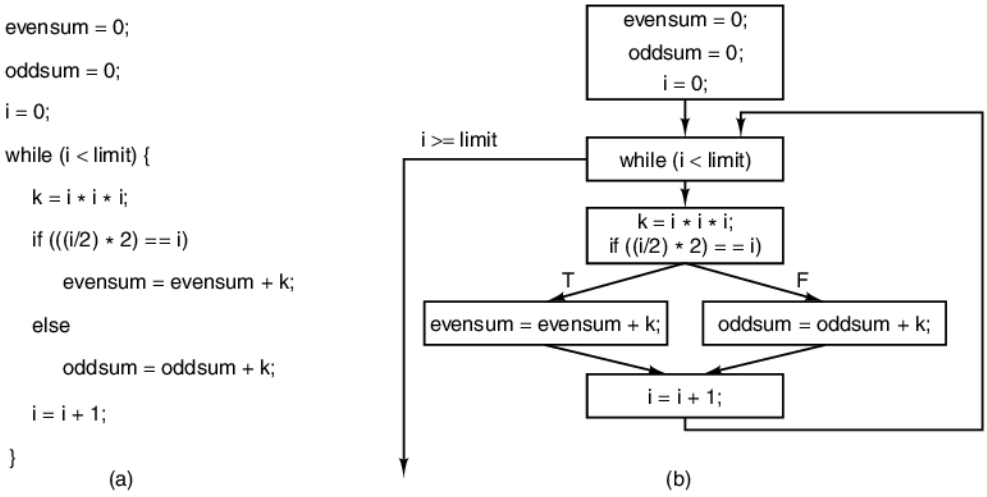


Figure 4-45. (a) A program fragment. (b) The corresponding basic block graph.

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

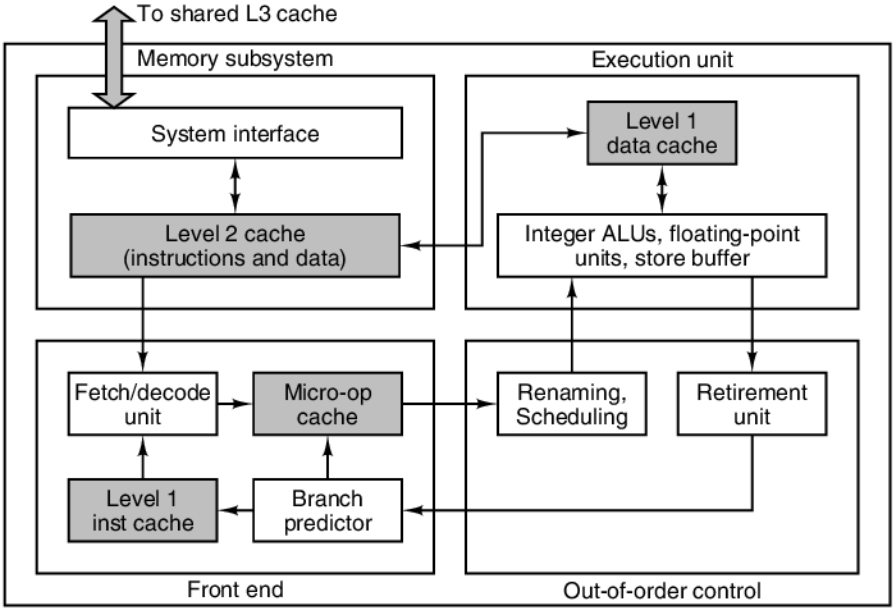


Figure 4-46. The block diagram of the Core i7's Sandy Bridge microarchitecture.

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.

The Core i7's Sandy Bridge Pipeline

Figure 4-47 is a simplified version of the Sandy Bridge microarchitecture showing the pipeline. At the top is the front end, whose job is to fetch instructions from memory and prepare them for execution. The front end is fed new x86 instructions from the L1 instruction cache. It decodes them into micro-ops for storage in the micro-op cache, which holds approximately 1.5K micro-ops. A micro-op cache of this size performs comparably to a 6-KB conventional L0 cache. The micro-op cache holds groups of six micro-ops in a single trace line. For longer sequences of micro-ops, multiple trace lines can be linked together.

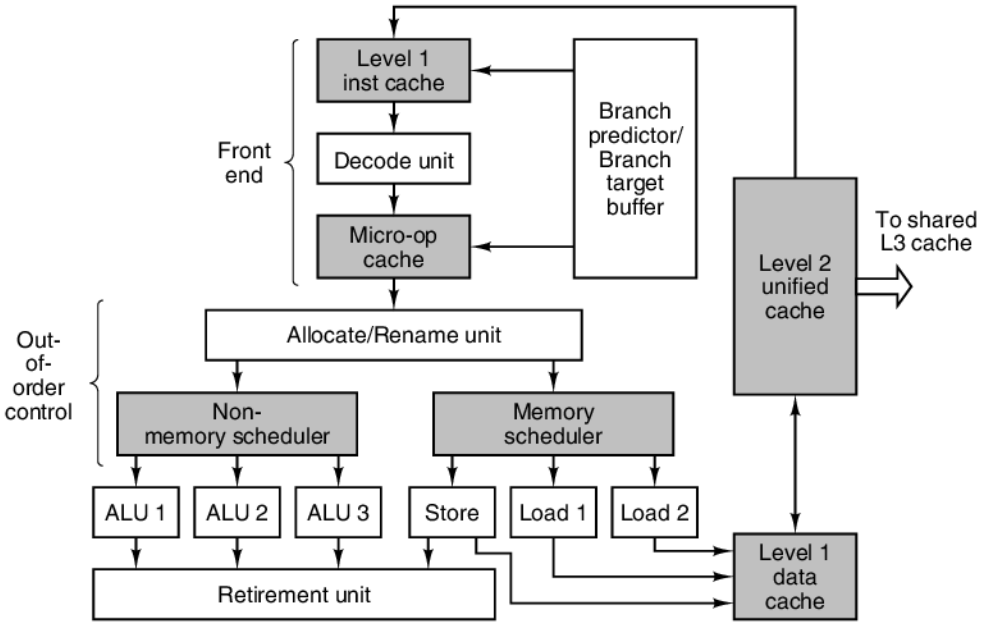


Figure 4-47. A simplified view of the Core i7 data path.

If the decode unit hits a conditional branch, it looks up its predicted direction in the **Branch Predictor**. The branch predictor contains the history of branches encountered in the past, and it uses this history to guess whether or not a conditional branch is going to be taken the next time it is encountered. This is where the top-secret algorithm is used.

If the branch instruction is not in the table, static prediction is used. A backward branch is assumed to be part of a loop and assumed to be taken. The accuracy of these static predictions is extremely high. A forward branch is assumed to be part of an if statement and is assumed not to be taken. The accuracy of these static predictions is much lower than that of the backward branches.

For a taken branch the **BTB** (**Branch Target Buffer**) is consulted to determine the target address. The BTB holds the target address of the branch the last time it was taken. Most of the time this address is correct (in fact, it is always correct for branches with a constant displacement). Indirect branches, such as those used by virtual function calls and C++ `switch` statements, go to many addresses, and they may be mispredicted by the BTB.

The second part of the pipeline, the out-of-order control logic, is fed from the micro-op cache. As each micro-op comes in from the front end, up to four per cycle, the **allocation/renaming unit** logs it in a 168-entry table called the **ROB** (**ReOrder Buffer**). This entry keeps track of the status of the micro-op until it is retired. The allocation/renaming unit then checks to see if the resources the micro-op needs are available. If so, the micro-op is enenqueued for execution in one of the **scheduler** queues. Separate queues are maintained for memory and nonmemory micro-ops. If a micro-op cannot be executed, it is delayed, but subsequent micro-ops are processed, leading to out-of-order execution of the micro-ops. This strategy was designed to keep all the functional units as busy as possible. As many as 154 instructions can be in flight at any instant, and up to 64 of these can be loads from memory and up to 36 can be stores into memory.

Sometimes a micro-op stalls because it needs to write into a register that is being read or written by a previous micro-op. These conflicts are called WAR and WAW dependences, respectively, as we saw earlier. By renaming the target of the new micro-op to allow it to write its result in one of the 160 scratch registers instead of in the intended, but still-busy, target, it may be possible to schedule the micro-op for execution immediately. If no scratch register is available, or the micro-op has a RAW dependence (which can never be papered over), the allocator notes the nature of the problem in the ROB entry. When all the required resources become available later, the micro-op is put into one of the scheduler queues.

The scheduler queues send micro-ops into the six functional units when they are ready to execute. The functional units are as follows:

1. ALU 1 and the floating-point multiply unit.
2. ALU 2 and the floating-point add/subtract unit.
3. ALU 3 and branch processing and floating-point comparisons unit.
4. Store instructions.
5. Load instructions 1.
6. Load instructions 2.

Since the schedulers and the ALUs can process one operation per cycle, a 3-GHz Core i7 has the scheduler performance to issue 18 billion operations per second; however, in reality the processor will never reach this level of throughput. Since the front end supplies at most four micro-ops per cycle, six micro-ops can only be

issued in short bursts since soon the scheduler queues will empty. Also, the memory units each take four cycles to process their operations, thus they could contribute to the peak execution throughput only in small bursts. Despite not being able to fully saturate the execution resources, the functional units do provide a significant execution capability, and that is why the out-of-order control goes to so much trouble to find work for them to do.

The three integer ALUs are not identical. ALU 1 can perform all arithmetic and logical operations and multiplies and divides. ALU 2 can perform only arithmetic and logical operations. ALU 3 can perform arithmetic and logical operations and resolve branches. Similarly, the two floating-point units are not identical either. The first one can perform floating-point arithmetic including multiplies, while the second one can perform only floating-point adds, subtracts, and moves.

The ALU and floating-point units are fed by a pair of 128-entry register files, one for integers and one for floating-point numbers. These provide all the operands for the instructions to be executed and provide a repository for results. Due to the register renaming, eight of them contain the registers visible at the ISA level (EAX, EBX, ECX, EDX, etc.), but which eight hold the "real" values varies over time as the mapping changes during execution.

The Sandy Bridge architecture introduced the Advanced Vector Extensions (AVX), which supports 128-bit data-parallel vector operations. The vector operations include both floating-point and integer vectors, and this new ISA extension represents a two-times increase in the size of vectors now supported compared to the previous SSE and SSE2 ISA extensions. How does the architecture implement 256-bit operations with only 128-bit data paths and functional units? It cleverly coordinates two 128-bit scheduler ports to produce a single 256-bit functional unit.

The L1 data cache is tightly coupled into the back end of the Sandy Bridge pipeline. It is a 32-KB cache and holds integers, floating-point numbers, and other kinds of data. Unlike the micro-op cache, it is not decoded in any way. It just holds a copy of the bytes in memory. The L1 data cache is an 8-way associative cache with 64 bytes per cache line. It is a write-back cache, meaning that when a cache line is modified, that line's dirty bit is set and the data are copied back to the L2 cache when evicted from the L1 data cache. The cache can handle two read and one write operation per clock cycle. These multiple accesses are implemented using **banking**, which splits the cache into separate subcaches (8 in the Sandy Bridge case). As long as as all three accesses are to separate banks, they can proceed in tandem; otherwise, all but one of the conflicting bank accesses will have to stall. When a needed word is not present in the L1 cache, a request is sent to the L2 cache, which either responds immediately or fetches the cache line from the shared L3 cache and then responds. Up to ten requests from the L1 cache to the L2 cache can be in progress at any instant.

Because micro-ops are executed out of order, stores into the L1 cache are not permitted until all instructions preceding a particular store have have been retired. The **retirement unit** has the job of retiring instructions, in order, and keeping track

of where it is. If an interrupt occurs, instructions not yet retired are aborted, so the Core i7 has "precise interrupts" so that upon an interrupt, all instructions up to a certain point have been completed and no instruction beyond that has any effect.

If a store instruction has been retired, but earlier instructions are still in progress, the L1 cache cannot be updated, so the results are put into a special pending-store buffer. This buffer has 36 entries, corresponding to the 36 stores that might be in execution at once. If a subsequent load tries to read the stored data, it can be passed from the pending-store buffer to the instruction, even though it is not yet in the L1 data cache. This process is called **store-to-load** forwarding. While this forwarding mechanism may seem straightforward, in practice it is quite complicated to implement because intervening stores may not have yet computed their addresses. In this case, the microarchitecture cannot definitely know which store in the store buffer will produce the needed value. The process of determining which store provides the value for a load is called **disambiguation**.

It should be clear by now that the Core i7 has a highly complex microarchitecture whose design was driven by the need to execute the old Pentium instruction set on a modern, highly pipelined RISC core. It accomplishes this goal by breaking Pentium instructions into micro-ops, caching them, and feeding them into the pipeline four at time for execution on a set of ALUs capable of executing up to six micro-ops per cycle under optimal conditions. Micro-ops are executed out of order but retired in order, and results are stored into the L1 and L2 caches in order.

4.6.2 The Microarchitecture of the OMAP4430 CPU

At the heart of the OMAP4430 system-on-a-chip are two ARM Cortex A9 processors. The Cortex A9 is a high-performance microarchitecture that implements the ARM instruction set (version 7). The processor was designed by ARM Ltd. and it is included with slight variations in a wide variety of embedded devices. ARM does not manufacture the processor, it only supplies the design to silicon manufacturers that want to incorporate it into their system-on-a-chip design (Texas Instruments, in this case).

The Cortex A9 processor is a 32-bit machine, with 32-bit registers and a 32-bit data path. Like the internal architecture, the memory bus is 32 bits wide. Unlike the Core i7, the Cortex A9 is a true RISC architecture, which means that it does not need a complex mechanism to convert old CISC instructions into micro-ops for execution. The core instructions are in fact already micro-op like ARM instructions. However, in recent years, more complex graphics and multimedia instructions have been added, requiring special hardware facilities for their execution.

Overview of the OMAP4430's Cortex A9 Microarchitecture

The block diagram of the Cortex A9 microarchitecture is given in Fig. 4-48. On the whole, it is much simpler than the Core i7's Sandy Bridge microarchitecture because it has a simpler ISA architecture to implement. Nevertheless, some of

the key components are similar to those used in the Core i7. The similarities are driven mostly by technology, power constraints, and economics. For example, both designs employ a multilevel cache hierarchy to meet the tight cost constraints of typical embedded applications; however, the last level of the Cortex A9's cache memory system (L2) is only 1 MB in size, significantly smaller than the Core i7 which supports last level caches (L3) of up to 20 MB. The differences, in contrast, are due mostly to the difference between having to bridge the gap between an old CISC instruction set and a modern RISC core and not having to do so.

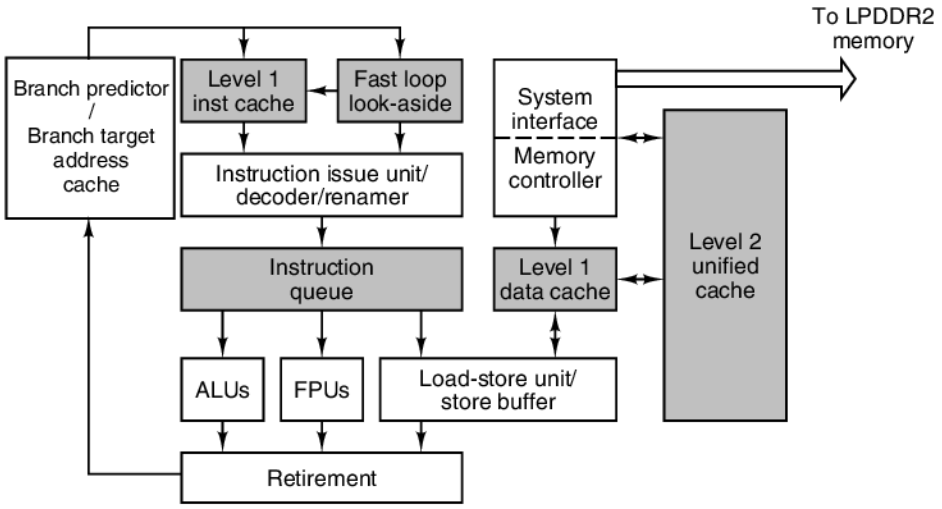


Figure 4-48. The block diagram of the OMAP4430's Cortex A9 microarchitecture.

At the top of Fig. 4-48 is the 32-KB 4-way associative instruction cache, which uses 32-byte cache lines. Since most ARM instructions are 4 bytes, there is room for about 8K instructions here in this cache, quite a bit larger than the Core i7's micro-op cache.

The **instruction issue unit** prepares up to four instructions for execution per clock cycle. If there is a miss on the L1 cache, fewer instructions will be issued. When a conditional branch is encountered, a branch predictor with 4K entries is consulted to predict whether or not the branch will be taken. If predicted taken, the 1K entry branch-target-address cache is consulted for the predicted target address. In addition, if the front end detects that the program is executing a tight loop (i.e., a non-nested small loop), it will load it into the fast-loop look-aside cache. This optimization speeds up instruction fetch and reduces power, since the caches and branch predictors can be in a low-power sleep mode while the tight loop is executing.

The output of the instruction issue unit flows into the decoders, which determine which resources and inputs are needed by the instructions. Like the Core i7,

the instructions are renamed after decode to eliminate WAR hazards that can slow down out-of-order execution. After renaming, the instructions are placed into the instruction dispatch queue, which will issue them when their inputs are ready for the functional units, potentially out of order.

The instruction dispatch queue sends instructions to the functional units, as shown in Fig. 4-48. The integer execution unit contains two ALUs as well as a short pipeline for branch instructions. The physical register file, which holds ISA registers and some scratch registers are also contained there. The Cortex A9 pipeline can optionally contain one more more compute engines as well, which act as additional functional units. ARM supports a compute engine for floating-point computation called **VFP** and integer SIMD vector computation called **NEON**.

The load/store unit handles various load and store instructions. It has paths to the data cache and the store buffer. The **data cache** is a traditional 32-KB 4-way associative L1 data cache using a 32-byte line size. The **store buffer** holds the stores that have not yet written their value to the data cache (at retirement). A load that executes will first try to fetch its value from the store buffer, using store-to-load forwarding like that of the Core i7. If the value is not available in the store buffer, it will fetch it from the data cache. One possible outcome of a load executing is an indication from the store buffer that it should wait, because an earlier store with an unknown address is blocking its execution. In the event that the L1 data cache access misses, the memory block will be fetched from the unified L2 cache. Under certain circumstances, the Cortex A9 also performs hardware prefetching out of the L2 cache into the L1 data cache, in order to improve the performance of loads and stores.

The OMAP 4430 chip also contains logic for controlling memory access. This logic is split into two parts: the system interface and the memory controller. The system interface interfaces with the memory over a 32-bit-wide LPDDR2 bus. All memory requests to the outside world pass through this interface. The LPDDR2 bus supports a 26-bit (word, not byte) address to 8 banks that return a 32-bit data word. In theory, the main memory can be up to 2 GB per LPDDR2 channel. The OMAP4430 has two of them, so it can address up to 4 GB of external RAM.

The memory controller maps 32-bit virtual addresses onto 32-bit physical addresses. The Cortex A9 supports virtual memory (discussed in Chap. 6), with a 4-KB page size. To speed up the mapping, special tables, called **TLBs** (**Translation Lookaside Buffers**), are provided to compare the current virtual address being referenced to those referenced in the recent past. Two such tables are provided for mapping instruction and data addresses.

The OMAP4430's Cortex A9 Pipeline

The Cortex A9 has an 11-stage pipeline, illustrated in simplified form in Fig. 4-49. The 11 stages are designated by short stage names shown on the left-hand side of the figure. Let us now briefly examine each stage. The *Fe*1 (Fetch

#1) stage is at the beginning of the pipeline. It is here that the address of the next instruction to be fetched is used to index the instruction cache and start a branch prediction. Normally, this address is the one following the previous instruction. However, this sequential order can be broken for a variety of reasons, such as when a previous instruction is a branch that has been predicted to be taken, or a trap or interrupt needs to be serviced. Because instruction fetch and branch prediction takes more than one cycle, the *Fe*2 (Fetch #2) stage provides extra time to carry out these operations. In the *Fe*3 (Fetch #3) stage the instructions fetched (up to four) are pushed into the instruction queue.

The *De*1 and *De*2 (Decode) stages decode the instructions. This step determines what inputs instructions will need (registers and memory) and what resources they will require to execute (functional units). Once decode is completed, the instructions enter the *Re* (Rename) stage where the registers accessed are renamed to eliminate WAR and WAW hazards during out-of-order execution. This stage contains the rename table which records which physical register currently holds all architectural registers. Using this table, any input register can be easily renamed. The output register must be given a new physical register, which is taken from a pool of unused physical registers. The assigned physical register will be in use by the instruction until it retires.

Next, instructions enter the *Iss* (Instruction Issue) stage, where they are dropped into the instruction issue queue. The issue queue watches for instructions whose inputs are all ready. When ready, their register inputs are acquired (from the physical register file or the bypass bus), and then the instruction is sent to the execution stages. Like the Core i7, the Cortex A9 potentially issues instructions out of program order. Up to four instructions can be issued each cycle. The choice of instructions is constrained by the functional units available.

The *Ex* (Execute) stages are where instructions are actually executed. Most arithmetic, Boolean, and shift instructions use the integer ALUs and complete in one cycle. Loads and stores take two cycles (if they hit in the L1 cache), and multiplies take three cycles. The *Ex* stages contain multiple functional units, which are:

1. Integer ALU 1.
2. Integer ALU 2.
3. Multiply unit.
4. Floating-point and SIMD vector ALU (optional with VFP and NEON support).
5. Load and store unit.

Conditional branch instructions are also processed in the first *Ex* stage and their direction (branch/no branch) is determined. In the event of a misprediction, a signal is sent back to the *Fe1* stage and the pipeline voided.

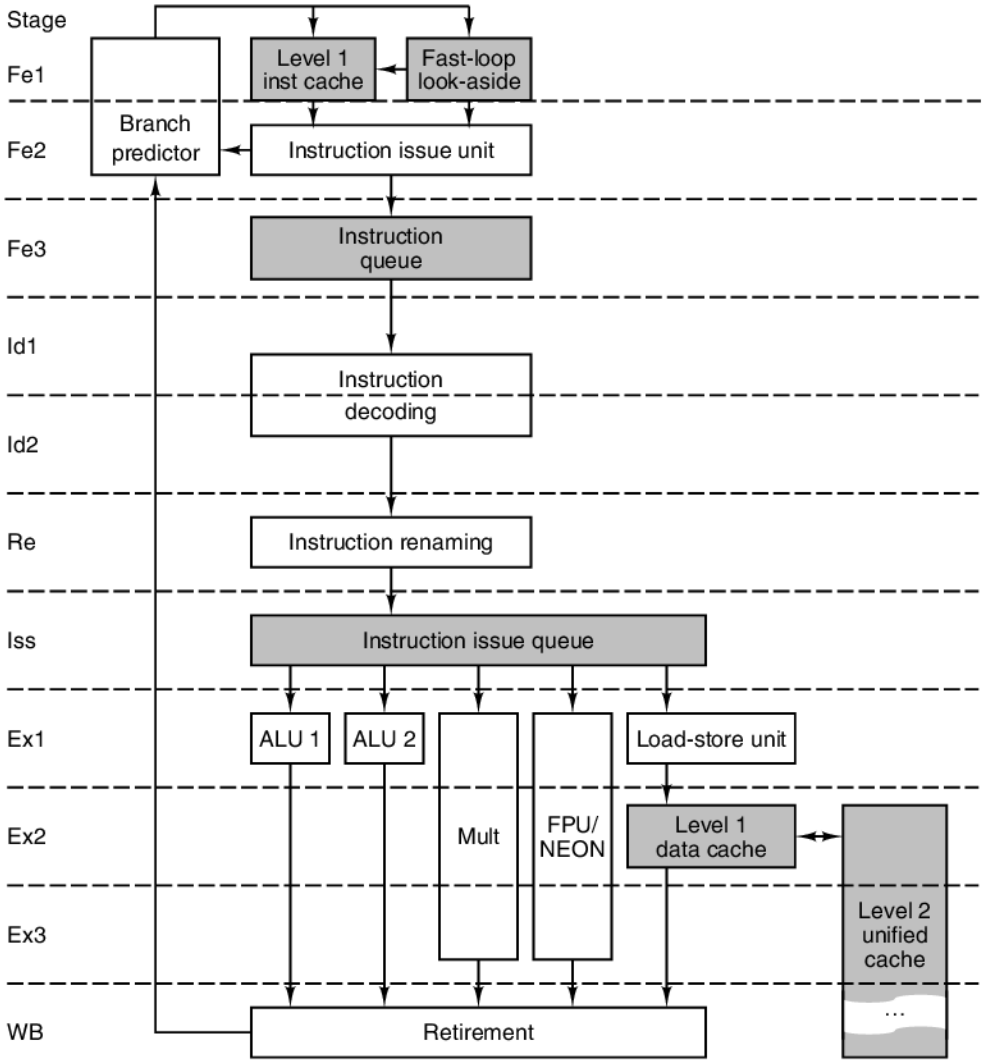


Figure 4-49. A simplified representation of the OMAP4430's Cortex A9 pipeline.

After completing execution, instructions enter the *WB* (WriteBack) stage where each instruction updates the physical register file immediately. Potentially later, when the instruction is the oldest one in flight, it will write its register result to the architectural register file. If a trap or interrupt occurs, it is these values, not those in the physical registers, that are made visible. The act of storing the register in the architectural file is equivalent to retirement in the Core i7. In addition, in the *WB* stage, any store instructions now complete writing their results to the L1 data cache.

This description of the Cortex A9 is far from complete but should give a reasonable idea of how it works and how it differs from the Core i7 microarchitecture.

4.6.3 The Microarchitecture of the ATmega168 Microcontroller

Our last example of a microarchitecture is the Atmel ATmega168, shown in Fig. 4-50. This one is considerably simpler than that of the Core i7 and OMAP4430. The reason is that the chip must be very small and cheap to serve the embedded design market. As such, the primary design goal of the ATmega168 was to make the chip cheap, not fast. Cheap and Simple are good friends. Cheap and Fast are not good friends.

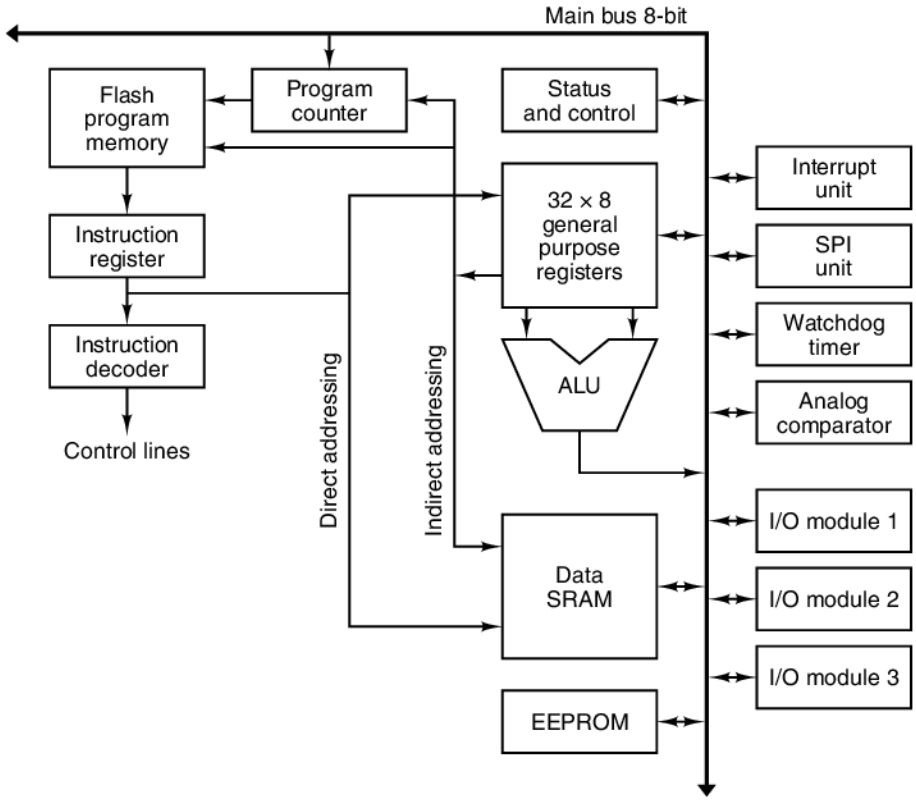


Figure 4-50. The microarchitecture of the ATmega168.

The heart of the ATmega168 is the 8-bit main bus. Attached to it are the registers and status bits, ALU, memory, and I/O devices. Let us briefly describe them now. The register file contains 32 8-bit registers, which are used to store temporary program values. The **status and control** register holds the condition codes of the last ALU operation (i.e., sign, overflow, negative, zero, and carry), plus a bit

that indicates if an interrupt is pending. The **program counter** holds the address of the instruction currently executing. To perform an ALU operation, the operands are first read from the register and sent to the ALU. The ALU output can be written to any of the writable registers via the main bus.

The ATmega168 has multiple memories for data and instructions. The data SRAM is 1 KB, too large to be fully addressed with an 8-bit address on the main bus. Thus, the AVR architecture allows addresses to be constructed with a sequential pair of 8-bit registers, thereby producing a 16-bit address that supports up to 64 KB of data memory. The EEPROM provides up to 1 KB of nonvolatile storage where programs can write variables that need to survive a power outage.

A similar mechanism exists to address program memory, but 64 KB of code is too small, even for low-cost embedded systems. To allow more instruction memory to be addressed the AVR architecture defines three RAM page registers (RAMPX, RAMPY, and RAMPZ), each 8 bits wide. The RAM page register is concatenated with a 16-bit register pair to produce a 24-bit program address, thereby allowing 16 MB of instruction address space.

Stop to think about that for a minute. 64 KB of code is too small for a microcontroller that might power a toy or small appliance. In 1964, IBM released the System 360 Model 30, which had 64 KB of total memory (with no tricks for upgrading it). It sold for $250,000, which is roughly $2 million in today's dollars. The ATmega168 costs about $1, less if you buy a lot of them at once. If you check out, say, Boeing's price list, you will discover that airplane prices have not dropped by a factor of 250,000 in the past 50 or so years. Nor have the prices of cars or televisions or anything except computers.

In addition, the ATmega168 has an on-chip interrupt controller, serial port interface (SPI), and timers, which are essential for real-time applications. There are also three 8-bit digital I/O ports, which allow the ATmega168 to control up to 24 external buttons, lights, sensors, actuators, and so on. It is the presence of the timers and I/O ports more than anything else that makes it possible to use the ATmega168 for embedded applications without any additional chips.

The ATmega168 is a synchronous processor, with most instructions taking one clock cycle, although some take more. The processor is pipelined, such that while one instruction is being fetched, the previous instruction is being executed. The pipeline is only two stages, however, fetch and execute. To execute instructions in one cycle, the clock cycle must accommodate reading the register from the register file, followed by executing the instruction in the ALU, followed by writing the register back to the register file. Because all of these operations occur in one clock cycle, there is no need for bypass logic or stall detection. Program instructions are executed in order, in one cycle, and without overlap with other instructions.

While we could go into more detail about the ATmega168, the description above and Fig. 4-50 give the basic idea. The ATmega168 has a single main bus (to reduce chip area), a heterogeneous set of registers, and a variety of memories and I/O devices hanging off the main bus. On each data path cycle, two operands are

read from the register file and run through the ALU and the results stored back into a register, just as on more modern computers.

4.7 COMPARISON OF THE I7, OMAP4430, AND ATMEGA168

Our three examples are very different, yet even they exhibit a certain amount of commonality. The Core i7 has an ancient CISC instruction set that Intel's engineers would dearly love to toss into San Francisco Bay, except that doing so would violate California's water pollution laws. The OMAP4430 is a pure RISC design, with a lean and mean instruction set. The ATmega168 is a simple 8-bit processor for embedded applications. Yet the heart of each of them is a set of registers and one or more ALUs that perform simple arithmetic and Boolean operations on register operands.

Despite their obvious external differences, the Core i7 and the OMAP4430 have fairly similar execution units. Both of the execution units accept micro-operations that contain an opcode, two source registers, and a destination register. Both of them can execute a micro-operation in one cycle. Both of them have deep pipelines, branch prediction, and split I- and D-caches.

This internal similarity is not an accident or even due to the endless job-hopping by Silicon Valley engineers. As we saw with our Mic-3 and Mic-4 examples, it is easy and natural to build a pipelined data path that takes two source registers, runs them through an ALU, and stores the results in a register. Figure 4-34 shows this pipeline graphically. With current technology, this is the most effective design.

The main difference between the Core i7 and the OMAP4430 is how they get from their ISA instruction set to the execution unit. The Core i7 has to break up its CISC instructions to get them into the three-register format needed by the execution unit. That is what the front end in Fig. 4-47 is all about—hacking big instructions into nice, neat micro-operations. The OMAP4430 does not have to do anything because its native ARM instructions are already nice, neat micro-operations. This is why most new ISAs are of the RISC type—to provide a better match between the ISA instruction set and the internal execution engine.

It is instructive to compare our final design, the Mic-4, to these two real-world examples. The Mic-4 is most like the Core i7. Both of them have the job of interpreting a non-RISC ISA instruction set. Both of them do this by breaking the ISA instructions into micro-operations with an opcode, two source registers, and a destination register. In both cases, the micro-operations are deposited in a queue for execution later. The Mic-4 has a strict in-order issue, in-order execute, in-order retire design, whereas the Core i7 has an in-order issue, out-of-order execute, in-order retire policy.

The Mic-4 and the OMAP4430 are not really comparable at all because the OMAP4430 has RISC instructions (i.e., three-register micro-operations) as its ISA

instruction set. They do not have to be broken up. They can be executed as is, each in a single data path cycle.

In contrast to the Core i7 and the OMAP4430, the ATmega168 is a simple machine indeed. It is more RISC like than CISC like because most of its simple instructions can be executed in one clock cycle and do not need to be decomposed. It has no pipelining and no caching, and it has in-order issue, in-order execute, and in-order retirement. In its simplicity, it is much akin to the Mic-1.

4.8 SUMMARY

The heart of every computer is the data path. It contains some registers, one, two or three buses, and one or more functional units such as ALUs and shifters. The main execution loop consists of fetching some operands from the registers and sending them over the buses to the ALU and other functional unit for execution. The results are then stored back in the registers.

The data path can be controlled by a sequencer that fetches microinstructions from a control store. Each microinstruction contains bits that control the data path for one cycle. These bits specify which operands to select, which operation to perform, and what to do with the results. In addition, each microinstruction specifies its successor, typically explicitly by containing its address. Some microinstructions modify this base address by ORing bits into the address before it is used.

The IJVM machine is a stack machine with 1-byte opcodes that push words onto the stack, pop words from the stack, and combine (e.g., add) words on the stack. A microprogrammed implementation was given for the Mic-1 microarchitecture. By adding an instruction fetch unit to preload the bytes in the instruction stream, many references to the program counter could be eliminated and the machine greatly speeded up.

There are many ways to design the microarchitecture level. Many trade-offs exist, including two-bus versus three-bus designs, encoded versus decoded microinstruction fields, presence or absence of prefetching, shallow or deep pipelines, and much more. The Mic-1 is a simple, software-controlled machine with sequential execution and no parallelism. In contrast, the Mic-4 is a highly parallel microarchitecture with a seven-stage pipeline.

Performance can be improved in a variety of ways. Cache memory is a major one. Direct-mapped caches and set-associative caches are commonly used to speed up memory references. Branch prediction, both static and dynamic, is important, as are out-of-order execution, and speculative execution.

Our three example machines, the Core i7, OMAP4430, and ATmega168, all have microarchitectures not visible to the ISA assembly-language programmers. The Core i7 has a complex scheme for converting the ISA instructions into micro-operations, caching them, and feeding them into a superscalar RISC core for out-of-order execution, register renaming, and every other trick in the book to get

the last possible drop of speed out of the hardware. The OMAP4430 has a deep pipeline, but is further relatively simple, with in-order issue, in-order execution, and in-order retirement. The ATmega168 is very simple, with a straightforward single main bus to which a handful of registers and one ALU are attached.

PROBLEMS

1. What are the four steps CPUs use to execute instructions?
2. In Fig. 4-6, the B bus register is encoded in a 4-bit field, but the C bus is represented as a bit map. Why?
3. In Fig. 4-6 there is a box labeled "High bit." Give a circuit diagram for it.
4. When the JMPC field in a microinstruction is enabled, MBR is ORed with NEXT_ADDRESS to form the address of the next microinstruction. Are there any circumstances in which it makes sense to have NEXT_ADDRESS be 0x1FF and use JMPC?
5. Suppose that in the example of Fig. 4-14(a) the statement

   ```
   k = 5;
   ```

   is added after the if statement. What would the new assembly code be? Assume that the compiler is an optimizing compiler.
6. Give two different IJVM translations for the following Java statement:

   ```
   i = k + n + 5;
   ```

7. Give the Java statement that produced the following IJVM code:

   ```
   ILOAD j
   ILOAD n
   ISUB
   BIPUSH 7
   ISUB
   DUP
   IADD
   ISTORE i
   ```

8. In the text we mentioned that when translating the statement

   ```
   if (Z) goto L1; else goto L2
   ```

   to binary, *L2* has to be in the bottom 256 words of the control store. Would it not be equally possible to have *L1* at, say, 0x40 and *L2* at 0x140? Explain your answer.
9. In the microprogram for Mic-1, in if_icmpeq3, MDR is copied to H. A few lines later it is subtracted from TOS to check for equality. Surely it is better to have one statement:

```
if_cmpeq3       Z = TOS – MDR; rd
```

Why is this not done?

10. How long does a 2.5-GHz Mic-1 take to execute the following Java statement

```
i = j + k;
```

Give your answer in nanoseconds.

11. Repeat the previous question, only now for a 2.5-GHz Mic-2. Based on this calculation, how long would a program that runs for 100 sec on the Mic-1 take on the Mic-2?

12. Write microcode for the Mic-1 to implement the JVM POPTWO instruction. This instruction removes two words from the top of the stack.

13. On the full JVM machine, there are special 1-byte opcodes for loading locals 0 through 3 onto the stack instead of using the general ILOAD instruction. How should IJVM be modified to make the best use of these instructions?

14. The instruction ISHR (arithmetic shift right integer) exists in JVM but not in IJVM. It uses the top two values on the stack, replacing them with a single value, the result. The second-from-top word of the stack is the operand to be shifted. Its content is shifted right by a value between 0 and 31, inclusive, depending on the value of the 5 least significant bits of the top word on the stack (the other 27 bits of the top word are ignored). The sign bit is replicated to the right for as many bits as the shift count. The opcode for ISHR is 122 (0x7A).

a. What is the arithmetic operation equivalent to left shift with a count of 2?
b. Extend the microcode to include this instruction as a part of IJVM.

15. The instruction ISHL (shift left integer) exists in JVM but not in IJVM. It uses the top two values on the stack, replacing the two with a single value, the result. The second-from-top word of the stack is the operand to be shifted. Its content is shifted left by a value between 0 and 31, inclusive, depending on the value of the 5 least significant bits of the top word on the stack (the other 27 bits of the top word are ignored). Zeros are shifted in from the right for as many bits as the shift count. The opcode for ISHL is 120 (0x78).

a. What is the arithmetic operation equivalent to shifting left with a count of 2?
b. Extend the microcode to include this instruction as a part of IJVM.

16. The JVM INVOKEVIRTUAL instruction needs to know how many parameters it has. Why?

17. Implement the JVM DLOAD instruction for the Mic-2. It has a 1-byte index and pushes the local variable at this position onto the stack. Then it pushes the next higher word onto the stack as well.

18. Draw a finite-state machine for tennis scoring. The rules of tennis are as follows. To win, you need at least four points and you must have at least two points more than your opponent. Start with a state (0, 0) indicating that no one has scored yet. Then add a state (1, 0) meaning that *A* has scored. Label the arc from (0, 0) to (1, 0) with an *A*. Now add a state (0, 1) indicating that *B* has scored, and label the arc from (0, 0) with a *B*. Continue adding states and arcs until all the possible states have been included.

19. Reconsider the previous problem. Are there any states that could be collapsed without changing the result of any game? If so, which ones are equivalent?

20. Draw a finite-state machine for branch prediction that is more tenacious than Fig. 4-42. It should change only predictions after three consecutive mispredictions.

21. The shift register of Fig. 4-27 has a maximum capacity of 6 bytes. Could a cheaper version of the IFU be built with a 5-byte shift register? How about a 4-byte one?

22. Having examined cheaper IFUs in the previous question, now let us examine more expensive ones. Would there ever be any point to have a much larger shift register in the IU, say, 12 bytes? Why or why not?

23. In the microprogram for the Mic-2, the code for if_icmpeq6 goes to T when Z is set to 1. However, the code at T is the same as goto1. Would it have been possible to go to goto1 directly? Would doing so have made the machine faster?

24. In the Mic-4, the decoding unit maps the IJVM opcode onto the ROM index where the corresponding micro-operations are stored. It would seem to be simpler to just omit the decoding stage and feed the IJVM opcode into the queueing directly. It could use the IJVM opcode as an index into the ROM, the same way as the Mic-1 works. What is wrong with this plan?

25. Why are computers equipped with multiple layers of cache? Would it not be better to simply have one big one?

26. A computer has a two-level cache. Suppose that 60% of the memory references hit on the first level cache, 35% hit on the second level, and 5% miss. The access times are 5 nsec, 15 nsec, and 60 nsec, respectively, where the times for the level 2 cache and memory start counting at the moment it is known that they are needed (e.g., a level 2 cache access does not even start until the level 1 cache miss occurs). What is the average access time?

27. At the end of Sec. 4.5.1, we said that write allocation wins only if there are likely to be multiple writes to the same cache line in a row. What about the case of a write followed by multiple reads? Would that not also be a big win?

28. In the first draft of this book, Fig. 4-39 showed a three-way associative cache instead of a four-way associative cache. One of the reviewers threw a temper tantrum, claiming that students would be horribly confused by this because 3 is not a power of 2 and computers do everything in binary. Since the customer is always right, the figure was changed to a four-way associative cache. Was the reviewer right? Discuss your answer.

29. Many computer architects spend a lot of time making their pipelines deeper. Why?

30. A computer with a five-stage pipeline deals with conditional branches by stalling for the next three cycles after hitting one. How much does stalling hurt the performance if 20% of all instructions are conditional branches? Ignore all sources of stalling except conditional branches.

31. A computer prefetches up to 20 instructions in advance. However, on the average, four of these are conditional branches, each with a probability of 90% of being predicted correctly. What is the probability that the prefetching is on the right track?

32. Suppose that we were to change the design of the machine used in Fig. 4-43 to have 16 registers instead of 8. Then we change I6 to use R8 as its destination. What happens in the cycles starting at cycle 6?

33. Normally, dependences cause trouble with pipelined CPUs. Are there any optimizations that can be done with WAW dependences that might actually improve matters? What?

34. Rewrite the Mic-1 interpreter but having LV now point to the first local variable instead of to the link pointer.

35. Write a simulator for a 1-way direct mapped cache. Make the number of entries and the line size parameters of the simulation. Experiment with it and report on your findings.

This page intentionally left blank

5

THE INSTRUCTION SET ARCHITECTURE LEVEL

This chapter discusses the Instruction Set Architecture (ISA) level in detail. This level, as we saw in Fig. 1-2, is positioned between the microarchitecture level and the operating system level. Historically, this level was developed before any of the other levels, and, in fact, was originally the only level. To this day this level is sometimes referred to simply as "the architecture" of a machine or sometimes (incorrectly) as "assembly language."

The ISA level has a special importance for system architects: it is the interface between the software and the hardware. While it might be possible to have the hardware directly execute programs written in C, C++, Java, or some other high-level language, this would not be a good idea. The performance advantage of compiling over interpreting would then be lost. Furthermore, to be of much practical use, most computers have to be able to execute programs written in multiple languages, not just one.

The approach that essentially all system designers take is to have programs in various high-level languages be translated to a common intermediate form—the ISA level—and build hardware that can execute ISA-level programs directly. The ISA level defines the interface between the compilers and the hardware. It is the language that both of them have to understand. The relationship among the compilers, the ISA level, and the hardware is shown in Fig. 5-1.

Ideally, when designing a new machine, the architects will spend time talking to both the compiler writers and the hardware engineers to find out what features they want in the ISA level. If the compiler writers want some feature that the en-

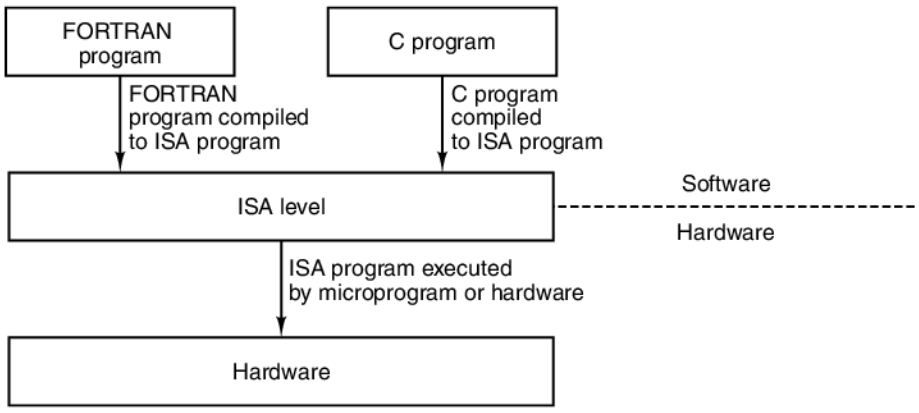


Figure 5-1. The ISA level is the interface between the compilers and the hardware.

gineers cannot implement in a cost-effective way (e.g., a branch-and-do-payroll instruction), it does not go in. Similarly, if the hardware folks have some nifty new feature they want to put in (e.g., a memory in which the words whose addresses are prime numbers are superfast), but the software folks cannot figure out how to generate code to use it, it will die on the drawing board. After much negotiation and simulation, an ISA perfectly optimized for the intended programming languages will emerge and be implemented.

That is the theory. Now the grim reality. When a new machine comes along, the first question all the potential customers ask is: "Is it compatible with its predecessor?" The second is: "Can I run my old operating system on it?" The third is: "Will it run all my existing application programs unmodified?" If any of the answers are "no," the designers will have a lot of explaining to do. Customers are rarely keen on throwing out all their old software and starting all over again.

This attitude puts a great deal of pressure on computer architects to keep the ISA the same between models, or at least make it **backward compatible**. By this we mean that the new machine must be able to run old programs without change. However, it is completely acceptable for the new machine to have new instructions and other features that can be exploited only by new software. In terms of Fig. 5-1, as long as the designers make the ISA backward compatible with the previous models, they are pretty much free to do whatever they want with the hardware, as hardly anyone cares about the real hardware (or even knows what it does). They can switch from a microprogrammed design to direct execution, or add pipelines or superscalar facilities or anything else they want, provided that they maintain backward compatibility with the previous ISA. The goal is to make sure that old programs run on the new machine. The challenge then becomes building better machines subject to the backward compatibility constraint.

The foregoing is not intended to imply that ISA design does not matter. A good ISA has significant advantages over a poor one, particularly in raw computing power vs. cost. For otherwise equivalent designs, different ISAs might account for a difference of as much as 25% in performance. Our point is just that market forces make it hard (but not impossible) to throw out an ancient ISA and introduce a new one. Nevertheless, every once in a while a new general-purpose ISA emerges, and in specialized markets (e.g., embedded systems or multimedia processors) this occurs much more frequently. Consequently, understanding ISA design is important.

What makes a good ISA? There are two primary factors. First, a good ISA should define a set of instructions that can be implemented efficiently in current and future technologies, resulting in cost-effective designs over several generations. A poor design is more difficult to implement and may require many more gates to implement a processor and more memory for executing programs. It also may run slower because the ISA obscures opportunities to overlap operations, requiring much more sophisticated designs to achieve equivalent performance. A design that takes advantage of the peculiarities of a particular technology may be a flash in the pan, providing a single generation of cost-effective implementations, only to be surpassed by more forward-looking ISAs.

Second, a good ISA should provide a clean target for compiled code. Regularity and completeness of a range of choices are important traits that are not always present in an ISA. These are important properties for a compiler, which may have trouble making the best choice among limited alternatives, particularly when some seemingly obvious alternatives are not permitted by the ISA. In short, since the ISA is the interface between the hardware and the software, it should make the hardware designers happy (be easy to implement efficiently) and make the software designers happy (be easy to generate good code for).

5.1 OVERVIEW OF THE ISA LEVEL

Let us start our study of the ISA level by asking what it is. This may seem like a simple question, but it has more complications than one might at first imagine. In the following section we will raise some of these issues. Then we will look at memory models, registers, and instructions.

5.1.1 Properties of the ISA Level

In principle, the ISA level is defined by how the machine appears to a machine-language programmer. Since no (sane) person does much programming in machine language any more, let us redefine this to say that ISA-level code is what a compiler outputs (ignoring operating-system calls and ignoring symbolic assembly language for the moment). To produce ISA-level code, the compiler writer has

to know what the memory model is, what registers there are, what data types and instructions are available, and so on. The collection of all this information is what defines the ISA level.

According to this definition, issues such as whether the microarchitecture is microprogrammed or not, whether it is pipelined or not, whether it is superscalar or not, and so on are not part of the ISA level because they are not visible to the compiler writer. However, this remark is not entirely true because some of these properties do affect performance, and that is visible to the compiler writer. Consider, for example, a superscalar design that can issue back-to-back instructions in the same cycle, provided that one is an integer instruction and one is a floating-point instruction. If the compiler alternates integer and floating-point instructions, it will get observably better performance than if it does not. Thus the details of the superscalar operation *are* visible at the ISA level, so the separation between the layers is not quite as clean as it might appear at first.

For some architectures, the ISA level is specified by a formal defining document, often produced by an industry consortium. For others it is not. For example, the ARM v7 (version 7 ARM ISA) has an official definition published by ARM Ltd. The purpose of a defining document is to make it possible for different implementers to build the machines and have them all run exactly the same software and get exactly the same results.

In the case of the ARM ISA, the idea is to allow multiple chip vendors to manufacture ARM chips that are functionally identical, differing only in performance and price. To make this idea work, the chip vendors have to know what an ARM chip is supposed to do (at the ISA level). Therefore the defining document tells what the memory model is, what registers are present, what the instructions do, and so on, but not what the microarchitecture is like.

Such defining documents contain **normative** sections, which impose requirements, and **informative** sections, which are intended to help the reader but are not part of the formal definition. The normative sections constantly use words like *shall*, *may not*, and *should* to require, prohibit, and suggest aspects of the architecture, respectively. For example, a sentence like

Executing a reserved opcode shall cause a trap.

says that if a program executes an opcode that is not defined, it must cause a trap and not be just ignored. An alternative approach might be to leave this open, in which case the sentence might read

The effect of executing a reserved opcode is implementation defined.

This means that the compiler writer cannot count on any particular behavior, thus giving different implementers the freedom to make different choices. Most architectural specifications are accompanied by test suites that check to see if an implementation that claims to conform to the specification really does.

It is clear why the ARM v7 has a document that defines its ISA level: so that all ARM chips will run the same software. For many years, there was no formal defining document for the IA-32 ISA (sometimes called the **x86** ISA) because Intel did not want to make it easy for other vendors to make Intel-compatible chips. In fact, Intel went to court to try to stop other vendors from cloning its chips, although it lost the case. In the late 1990s, however, Intel finally released a complete specification of the IA-32 instruction set. Perhaps this was because they felt the error of their ways and wanted to help out fellow architects and programmers, or perhaps it was because the United States, Japan, and Europe were all investigating Intel for possibly violating antitrust laws. This well-written ISA reference is still updated today at Intel's developer website (http://developer.intel.com). The version released with Intel's Core i7 weighs in at 4161 pages, reminding us once again that the Core i7 is a *complex* instruction set computer.

Another important property of the ISA level is that on most machines there are at least two modes. **Kernel mode** is intended to run the operating system and allows all instructions to be executed. **User mode** is intended to run application programs and does not permit certain sensitive instructions (such as those that manipulate the cache directly) to be executed. In this chapter we will primarily focus on user-mode instructions and properties.

5.1.2 Memory Models

All computers divide memory up into cells that have consecutive addresses. The most common cell size at the moment is 8 bits, but cell sizes from 1 to 60 bits have been used in the past (see Fig. 2-10). An 8-bit cell is called a **byte** (or **octet**). The reason for using 8-bit bytes is that ASCII characters are 7 bits, so one ASCII character (plus a rarely used parity bit) fits into a byte. Other codes, such as Unicode and UTF-8, use multiples of 8 bits to represent characters.

Bytes are generally grouped into 4-byte (32-bit) or 8-byte (64-bit) words with instructions available for manipulating entire words. Many architectures require words to be aligned on their natural boundaries. For example, a 4-byte word may begin at address 0, 4, 8, etc., but not at address 1 or 2. Similarly, an 8-byte word may begin at address 0, 8, or 16, but not at address 4 or 6. Alignment of 8-byte words is illustrated in Fig. 5-2.

Alignment is often required because memories operate more efficiently that way. The Core i7, for example, fetches 8 bytes at a time from memory using a DDR3 interface which supports only aligned 64-bit accesses.. Thus the Core i7 could not even make a nonaligned memory reference if it wanted to because memory interface requires addresses that are multiples of 8.

However, this alignment requirement sometimes causes problems. On the Core i7, programs are allowed to reference words starting at any address, a property that goes back to the 8088, which had a 1-byte-wide data bus (and thus no requirement about aligning memory references on 8-byte boundaries). If a Core i7

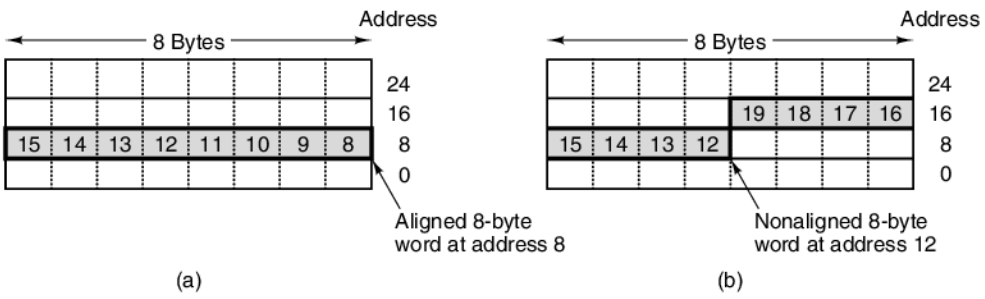


Figure 5-2. An 8-byte word in a little-endian memory. (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

program reads a 4-byte word at address 7, the hardware has to make one memory reference to get bytes 0 through 7 and a second to get bytes 8 through 15. Then the CPU has to extract the required 4 bytes from the 16 bytes read from memory and assemble them in the right order to form a 4-byte word. Doing this on a regular basis does not lead to blinding speed.

Having the ability to read words at arbitrary addresses requires extra logic on the chip, which makes it bigger and more expensive. The design engineers would love to get rid of it and simply require all programs to make word-aligned references to memory. The trouble is, whenever the engineers say: "Who cares about running musty old 8088 programs that reference memory wrong?" the folks in marketing have a succinct answer: "Our customers."

Most machines have a single linear address space at the ISA level, extending from address 0 up to some maximum, often $2^{32} - 1$ bytes or $2^{64} - 1$ bytes. However, a few machines have separate address spaces for instructions and data, so that an instruction fetch at address 8 goes to a different address space than a data fetch at address 8. This scheme is more complex than having a single address space, but it has two advantages. First, it becomes possible to have 2^{32} bytes of program and an additional 2^{32} bytes of data while using only 32-bit addresses. Second, because all writes automatically go to data space, it becomes impossible for a program to accidentally overwrite itself, thus eliminating one source of program bugs. Separating instruction and data spaces also makes attacks by malware much harder to pull off because the malware cannot change the program—it cannot even address it.

Note that having separate address spaces for instructions and data is not the same as having a split level 1 cache. In the former case the total amount of address space is doubled and reads to any given address yield different results, depending on whether an instruction or a data word is being read. With a split cache, there is still just one address space, only different caches store different parts of it.

Yet another aspect of the ISA level memory model is the memory semantics. It is perfectly natural to expect that a LOAD instruction that occurs after a STORE instruction and that references the same address will return the value just stored.

In many designs, however, as we saw in Chap. 4, microinstructions are reordered. Thus there is a real danger that the memory will not have the expected behavior. The problem gets even worse on a multiprocessor, with each of multiple CPUs sending a stream of (possibly reordered) read and write requests to shared memory.

System designers can take any one of several approaches to this problem. At one extreme, all memory requests can be serialized, so that each one is completed before the next is issued. This strategy hurts performance but gives the simplest memory semantics (all operations are executed in strict program order).

At the other extreme, no guarantees of any kind are given. To force an ordering on memory, the program must execute a SYNC instruction, which blocks the issuing of all new memory operations until all previous ones have completed. This design puts a great burden on the compilers because they have to understand how the underlying microarchitecture works in detail, but it gives the hardware designers the maximum freedom to optimize memory usage.

Intermediate memory models are also possible, in which the hardware automatically blocks the issuing of certain memory references (e.g., those involving a RAW or WAR dependence) but not others. While having all these peculiarities caused by the microarchitecture be exposed to the ISA level is annoying (at least to the compiler writers and assembly-language programmers), it is very much the trend. This trend is caused by the underlying implementations such as microinstruction reordering, deep pipelines, multiple cache levels, and so on. We will see more examples of such unnatural effects later in this chapter.

5.1.3 Registers

All computers have some registers visible at the ISA level. They are there to control execution of the program, hold temporary results, and serve other purposes. In general, the registers visible at the microarchitecture level, such as TOS and MAR in Fig. 4-1, are not visible at the ISA level. A few of them, however, such as the program counter and stack pointer, are visible at both levels. On the other hand, registers visible at the ISA level are always visible at the microarchitecture level since that is where they are implemented.

ISA-level registers can be roughly divided into two categories: special-purpose registers and general-purpose registers The special-purpose registers include things like the program counter and stack pointer, as well as other registers with a specific function. In contrast, the general-purpose registers are there to hold key local variables and intermediate results of calculations. Their main function is to provide rapid access to heavily used data (basically, avoiding memory accesses). RISC machines, with their fast CPUs and (relatively) slow memories, usually have at least 32 general-purpose registers, and the trend in new CPU designs is to have even more.

On some machines, the general-purpose registers are completely symmetric and interchangeable. Each one can do anything the others can do. If the registers

are all equivalent, a compiler can use R1 to hold a temporary result, but it can equally well use R25. The choice of register does not matter.

On other machines, however, some of the general-purpose registers may be somewhat special. For example, on the Core i7, there is a register called EDX that can be used as a general register, but which also receives half the product in a multiplication and holds half the dividend in a division.

Even when the general-purpose registers are completely interchangeable, it is common for the operating system or compilers to adopt conventions about how they are used. For example, some registers may hold parameters to procedures called and others may be used as scratch registers. If a compiler puts an important local variable in R1 and then calls a library procedure that thinks R1 is a scratch register available to it, when the library procedure returns, R1 may contain garbage. If there are system-wide conventions on how the registers are to be used, compilers and assembly-language programmers are advised to adhere to them to avoid trouble.

In addition to the ISA-level registers visible to user programs, there are always a substantial number of special-purpose registers available only in kernel mode. These registers control the various caches, memory, I/O devices, and other hardware features of the machine. They are used only by the operating system, so compilers and users do not have to know about them.

One control register that is something of a kernel/user hybrid is the **flags register** or PSW (**Program Status Word**). This register holds various miscellaneous bits that are needed by the CPU. The most important bits are the **condition codes**. These bits are set on every ALU cycle and reflect the status of the result of the most recent operation. Typical condition code bits include

N — Set when the result was Negative.

Z — Set when the result was Zero.

V — Set when the result caused an oVerflow.

C — Set when the result caused a Carry out of the leftmost bit.

A — Set when there was a carry out of bit 3 (Auxiliary carry—see below).

P — Set when the result had even Parity.

The condition codes are important because the comparison and conditional branch instructions (also called conditional jump instructions) use them. For example, the CMP instruction typically subtracts two operands and sets the condition codes based on the difference. If the operands are equal, then the difference will be zero and the *Z* condition code bit in the PSW register will be set. A subsequent BEQ (Branch EQual) instruction tests the *Z* bit and branches if it is set.

The PSW contains more than just the condition codes, but the full contents varies from machine to machine. Typical additional fields are the machine mode

(e.g., user or kernel), trace bit (used for debugging), CPU priority level, and interrupt enable status. Often the PSW is readable in user mode, but some of the fields can be written only in kernel mode (e.g., the user/kernel mode bit).

5.1.4 Instructions

The main feature of the ISA level is its set of machine instructions. These control what the machine can do. There are always LOAD and STORE instructions (in one form or another) for moving data between memory and registers and MOVE instructions for copying data among the registers. Arithmetic instructions are always present, as are Boolean instructions and instructions for comparing data items and branching on the results. We have seen some typical ISA instructions already (see Fig. 4-11) and will study many more in this chapter.

5.1.5 Overview of the Core i7 ISA Level

In this chapter we will discuss three widely different ISAs: Intel's IA-32, as embodied in the Core i7, the ARM v7 architecture, implemented in the OMAP4430 system-on-a-chip, and the AVR 8-bit architecture, used by the ATmega168 microcontroller. The intent is not to provide an exhaustive description of any of the ISAs, but rather to demonstrate important aspects of an ISA, and to show how these aspects can vary from one ISA to another. Let us start with the Core i7.

The Core i7 processor has evolved over many generations, tracing its lineage back to some of the earliest microprocessors ever built, as we discussed in Chap. 1. While the basic ISA maintains full support for execution of programs written for the 8086 and 8088 processors (which had the same ISA), it even contains remnants of the 8080, an 8-bit processor popular in the 1970s. The 8080, in turn, was strongly influenced by compatibility constraints with the still-earlier 8008, which was based on the 4004, a 4-bit chip used back when dinosaurs roamed the earth.

From a pure software standpoint, the 8086 and 8088 were straightforward 16-bit machines (although the 8088 had an 8-bit data bus). Their successor, the 80286, was also a 16-bit machine. Its main advantage was a larger address space, although few programs ever used it because it consisted of 16,384 64-KB segments rather than a linear 2^{30}-byte memory.

The 80386 was the first 32-bit machine in the Intel family. All the subsequent machines (80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino, Core 2 duo, Core i7, etc.) have essentially the same 32-bit architecture as the 80386, called **IA-32**, so it is this architecture that we will focus on here. The only major architectural change after the 80386 was the introduction of the MMX, SSE, and SSE2 instructions in later versions of x86 series. These instructions are highly specialized and designed to improve performance on multimedia applications. Another important extension was 64-bit x86

(often called x86-64), which increased the integer computations and virtual address size to 64 bits. While most extensions were introduced by Intel and later implemented by competitors, this was one case where AMD introduced an extension that Intel had to adopt.

The Core i7 has three operating modes, two of which make it act like an 8088. In **real mode**, all the features that have been added since the 8088 are turned off and the Core i7 behaves like a simple 8088. If any program does something wrong, the whole machine just crashes. If Intel had designed human beings, it would have put in a bit that made them revert back to chimpanzee mode (most of the brain disabled, no speech, sleeps in trees, eats mostly bananas, etc.)

One step up is **virtual 8086 mode**, which makes it possible to run old 8088 programs in a protected way. In this mode, a real operating system is in control of the whole machine. To run an old 8088 program, the operating system creates a special isolated environment that acts like an 8088, except that if its program crashes, the operating system is notified instead of the machine crashing. When a Windows user starts an MS-DOS window, the program run there is started in virtual 8086 mode to protect Windows itself from misbehaving MS-DOS programs.

The final mode is protected mode, in which the Core i7 actually acts like a Core i7 instead of a very expensive 8088. Four privilege levels are available and controlled by bits in the PSW. Level 0 corresponds to kernel mode on other computers and has full access to the machine. It is used by the operating system. Level 3 is for user programs. It blocks access to certain critical instructions and control registers to prevent a rogue user program from bringing down the entire machine. Levels 1 and 2 are rarely used.

The Core i7 has a huge address space, with memory divided into 16,384 segments, each going from address 0 to address $2^{32} - 1$. However, most operating systems (including UNIX and all versions of Windows) support only one segment, so most application programs effectively see a linear address space of 2^{32} bytes, and sometimes part of this is occupied by the operating system. Every byte in the address space has its own address, with words being 32 bits long. Words are stored in little-endian format (the low-order byte has the lowest address).

The Core i7's registers are shown in Fig. 5-3. The first four registers, EAX, EBX, ECX, and EDX, are 32-bit, more-or-less general-purpose registers, although each has its own peculiarities. EAX is the main arithmetic register; EBX is good for holding pointers (memory addresses); ECX plays a role in looping; EDX is needed for multiplication and division, where, together with EAX, it holds 64-bit products and dividends. Each of these registers contains a 16-bit register in the low-order 16 bits and an 8-bit register in the low-order 8 bits. These registers make it easy to manipulate 16- and 8-bit quantities, respectively. The 8088 and 80286 had only the 8- and 16-bit registers. The 32-bit registers were added with the 80386, along with the E prefix, which stands for Extended.

The next four are also somewhat general purpose, but with more peculiarities. The ESI and EDI registers are intended to hold pointers into memory, especially for

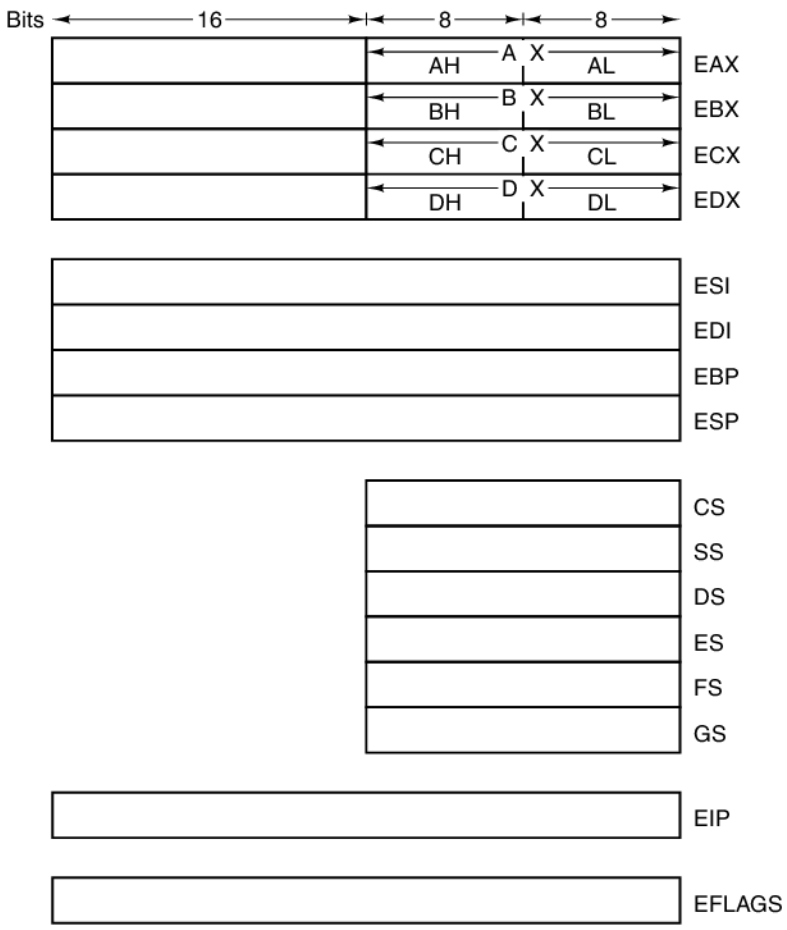


Figure 5-3. The Core i7's primary registers.

the hardware string-manipulation instructions, where ESI points to the source string and EDI points to the destination string. The EBP register is also a pointer register. It is typically used to point to the base of the current stack frame, the same as LV in IJVM. When a register (like EBP) is used to point to the base of the local stack frame, it is usually called the **frame pointer**. Finally, ESP is the stack pointer.

The next group of registers, CS through GS, are segment registers. To some extent, they are electronic trilobites, ancient fossils left over from the time the 8088 attempted to address 2^{20} bytes of memory using 16-bit addresses. Suffice it to say that when the Core i7 is set up to use a single linear 32-bit address space, they can be safely ignored. Next is EIP, which is the program counter (Extended Instruction Pointer). Finally, we come to EFLAGS, which is the PSW.

5.1.6 Overview of the OMAP4430 ARM ISA Level

The ARM Architecture was first introduced in 1985 by Acorn Computer. The architecture was inspired by the research done at Berkeley in the 1980s (Patterson, 1985, and Patterson and Séquin, 1982). The original ARM architecture (called the ARM2) was a 32-bit architecture that supported a 26-bit address space. The OMAP4430 utilizes the ARM Cortex A9 microarchitecture, which implements the version 7 of the ARM architecture, and that is the ISA we will describe in this chapter. For consistency with the rest of the book, we will refer to the OMAP4430 here, but at the ISA level, all designs based on the ARM Cortex A9 core implement the same ISA.

The OMAP4430 memory structure is clean and simple: addressable memory is a linear array of 2^{32} bytes. ARM processors are bi-endian, such that they can access memory with big- or little- endian order. The endian is specified in a system memory block that is read immediately after processor reset. To ensure that the system memory block is read correctly, it must be in little-endian format, even if the machine is to be configured for big-endian operation.

It is important that the ISA have a larger address-space limit than implementations need, because future implementations almost certainly will need to increase the size of memory the processor can access. The ARM ISA's 32-bit address space is giving many designers growing pains, since many ARM-based systems, such as smartphones, already have more than 2^{32} bytes of memory. To date, designers have worked around these problems by making the bulk of the memory flash-drive storage, which is accessed with a disk interface that supports a larger block-oriented address space. To address this potentially market-killing limitation, ARM (the company) recently published the definition of the ARM version 8 ISA, which support 64-bit address spaces.

A serious problem encountered with successful architectures has been that their ISA limited the amount of addressable memory. In computer science, the only error one cannot work around is not enough bits. One day your grandchildren will ask you how computers could do anything in the old days with only 32-bit addresses and only 4 GB of real memory when the average game needs 1 TB just to boot up.

The ARM ISA is clean, though the organization of the registers is somewhat quirky in an attempt to simplify some instruction encodings. The architecture maps the program counter into the integer register file (as register R15), as this allows branches to be created with ALU operations that have R15 as a destination register. Experience has shown that the register organization is more trouble than it is worth, but ye olde backwarde-compatibility rule made it well nigh impossible to get rid of.

The ARM ISA has two groups of registers. These are the 16 32-bit general-purpose registers and the 32 32-bit floating-point registers (if the VFP coprocessor is supported). The general-purpose registers are called R0 through R15,

although other names are used in certain contexts. The alternative names and functions of the registers are shown in Fig. 5-4.

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Figure 5-4. The version 7 ARM's general registers.

All the general registers are 32 bits wide and can be read and written by a variety of load and store instructions. The uses given in Fig. 5-4 are based partly on convention, but also partly on how the hardware treats them. In general, it is unwise to deviate from the uses listed in the figure unless you have a Black Belt in ARM hacking and really, really know what you are doing. It is the responsibility of the compiler or programmer to be sure that the program accesses the registers correctly and performs the correct kind of arithmetic on them. For example, it is very easy to load floating-point numbers into the general registers and then perform integer addition on them, an operation that will produce utter nonsense, but which the CPU will cheerfully perform when so instructed.

The Vx registers are used to hold constants, variables, and pointers that are needed by procedures, and they should be stored and reloaded at procedure entries and exits if need be. The Ax registers are used for passing parameters to procedures to avoid memory references. We will explain how this works below.

Four dedicated registers are used for special purposes. The IP register works around the limitations of the ARM functional call instruction (BL) which cannot fully address all of its 2^{32} bytes of address space. If the target of a call is too far away for the instruction to express, the instruction will call a "veneer" code snippet that uses the address in the IP register as the destination of the function call. The SP register indicates the current top of the stack and fluctuates as words are pushed onto the stack or popped from it. The third special-purpose register is LR. It is used for procedure calls to hold the return address. The fourth special-purpose register, as mentioned earlier, is the program counter PC. Storing a value to this register redirects the fetching of instructions to that newly deposited PC address. Another important register in the ARM architecture is the program status register (PSR), which holds the status of previous ALU computations, including Zero, Negative, and Overflow among other bits.

The ARM ISA (when configured with the VFP coprocessor) also has 32 32-bit floating-point registers. These registers can be accessed either directly as 32 single-precision floating-point values or as 16 64-bit double-precision floating-point

values. The size of the floating-point register accessed is determined by the instruction; in general, all ARM floating-point instructions come in single- and double-precision variants.

The ARM architecture is a **load/store architecture**. That is, the only operations that access memory directly are load and store instructions to move data between the registers and the memory. All operands for arithmetic and logical instructions must come from registers or be supplied by the instruction (not memory), and all results must be saved in a register (not memory).

5.1.7 Overview of the ATmega168 AVR ISA Level

Our third example is the ATmega168. Unlike the Core i7 (which is used primarily in desktop machines and server farms), and the OMAP4430 (which is used primarily in phones, tablets, and other mobile devices), the ATmega168 is used in low-end embedded systems such as traffic lights and clock radios to control the device and manage the buttons, lights, and other parts of the user interface. In this section, we will give a brief technical introduction to the ATmega168 AVR ISA.

The ATmega168 has one mode and no protection hardware since it never runs multiple programs owned by potentially hostile users. The memory model is extremely simple. There is 16 KB of program memory and a second 1 KB of data memory. Each is its own distinct address space, so a particular address will reference different memory depending on whether the access is to the program or data memory. The program and data spaces are split to make it possible to implement the program space in flash and the data space in SRAM.

Several different implementations of memory are possible, depending on how much the designer wants to pay for the processor. In the simplest one, the ATmega48, there is a 4-KB flash for the program and a 512-byte SRAM for data. Both the flash and the RAM are on chip. For small applications, this amount of memory is often enough and having all the memory on the CPU chip is a big win. The ATmega88 has twice as much memory on chip: 8 KB of ROM and 1 KB of SRAM.

The ATmega168 uses a two-tiered memory organization to provide better program security. Program flash memory is divided into the *boot loader section* and *application section*, the size of each being determined by fuse bits that are one-time programmed when the microcontroller is first powered up. For security reasons, only code run from the boot loader section can update flash memory. With this feature, any code can be placed in the application area (including downloaded third-party applications) with confidence that it will never muck with other code in the system (because application code will be running from the application space which cannot write flash memory). To really tie down a system, a vendor can digitally sign code. With signed code, the boot loader loads code into the flash memory only if it is digitally signed by an approved software vendor. As such, the system

will run only code that has been "blessed" by a trusted software vendor. The approach is quite flexible in that even the boot loader can be replaced, if the new code has been properly digitally signed. This is similar to the way that Apple and TiVo ensure that the code running on their devices is safe from mischief.

The ATmega168 contains 32 8-bit general-purpose registers, which are accessed by instructions via a 5-bit field specifying which register to use. The registers are called R0 through R31. A peculiar property of the ATmega168 registers is that they are also present in the memory space. Byte 0 of the data space is equivalent to R0 of register set 0. When an instruction changes R0 and then later reads out memory byte 0, it finds the new value of R0 there. Similarly, byte 1 of memory is R1 and so on, up to byte 31. This arrangement is shown in Fig. 5-5.

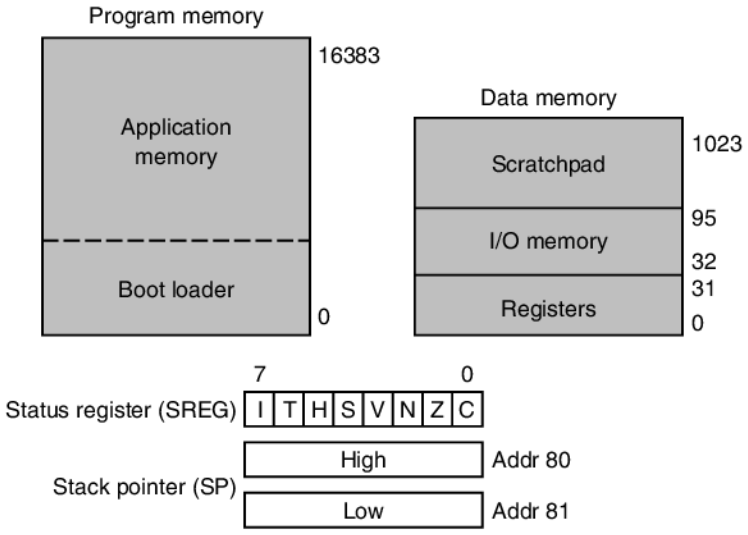


Figure 5-5. On-chip register and memory organization for the ATmega168.

Directly above the 32 general-purpose registers, at memory addresses 32 through 95, are 64 bytes of memory reserved for accessing I/O device registers, including the internal system-on-a-chip devices.

In addition to the four sets of eight registers, the ATmega168 has a small number of special-purpose registers, the most important of which are illustrated in Fig. 5-5. The *status register* contains, from left to right, the interrupt enable bit, the half-carry bit, the sign bit, the overflow bit, the negative flag, the zero flag, and the carry-out bit. All of these status bits, except the interrupt enable bit, are set as a result of arithmetic operations.

The status register I bit allows interrupts to be enabled or disabled globally. If the I bit is 0, all interrupts are disabled. Clearing this bit makes it possible to disable any further interrupts in a single instruction. Setting the bit allows any interrupts currently pending to occur as well as future ones. Each device has associated

with it an interrupt enable bit. If the device enable is set and the global interrupt enable I bit is set, the device can interrupt the processor.

The stack pointer SP holds the current address in data memory where PUSH and POP instructions will access their data, similar to the similarly named instruction in the Java JVM of Chap. 4. The stack pointer is located in I/O memory at address 80. A single 8-bit memory byte is too small to address 1024 bytes of data memory, so the stack pointer is composed of two consecutive locations in memory, forming a 16-bit address.

5.2 DATA TYPES

All computers need data. In fact, for many computer systems, the whole purpose is to process financial, commercial, scientific, engineering, or other data. The data have to be represented in some specific form inside the computer. At the ISA level, a variety of different data types are used. These will be explained below.

A key issue is whether there is hardware support for a particular data type. Hardware support means that one or more instructions expect data in a particular format, and the user is not free to pick a different format. For example, accountants have the peculiar habit of writing negative numbers with the minus sign to the right of the number rather than to the left, where computer scientists put it. Suppose that in an effort to impress his boss, the head of the computer center at an accounting firm changed all the numbers in all the computers to use the rightmost bit (instead of the leftmost bit) as the sign bit. This would no doubt make a big impression on the boss—because all the software would instantly cease to function correctly. The hardware expects a certain format for integers and does not work properly when given anything else.

Now consider another accounting firm, this one just having gotten a contract to verify the federal debt (how much the U.S. government owes everyone). Using 32-bit arithmetic would not work here because the numbers involved are larger than 2^{32} (about 4 billion). One solution is to use two 32-bit integers to represent each number, giving 64 bits in all. If the machine does not support **double-precision** numbers, all arithmetic on them will have to be done in software, but the two parts can be in either order since the hardware does not care. This is an example of a data type without hardware support and thus without a required hardware representation. In the following sections we will look at data types that are supported by the hardware, and thus for which specific formats are required.

5.2.1 Numeric Data Types

Data types can be divided into two categories: numeric and nonnumeric. Chief among the numeric data types are the integers. They come in many lengths, typically 8, 16, 32, and 64 bits. Integers count things (e.g., the number of screwdrivers

a hardware store has in stock), identify things (e.g., bank account numbers), and much more. Most modern computers store integers in two's complement binary notation, although other systems have also been used in the past. Binary numbers are discussed in Appendix A.

Some computers support unsigned as well as signed integers. For an unsigned integer, there is no sign bit and all the bits contain data. This data type has the advantage of an extra bit, so for example, a 32-bit word can hold a single unsigned integer in the range from 0 to $2^{32} - 1$, inclusive. In contrast, a two's complement signed 32-bit integer can handle only numbers up to $2^{31} - 1$, but, of course, it can also handle negative numbers.

For numbers that cannot be expressed as an integer, such as 3.5, floating-point numbers are used. These are discussed in Appendix B. They have lengths of 32, 64, or 128 bits. Most computers have instructions for doing floating-point arithmetic. Many computers have separate registers for holding integer operands and for holding floating-point operands.

Some programming languages, notably COBOL, allow decimal numbers as a data type. Machines that wish to be COBOL-friendly often support decimal numbers in hardware, typically by encoding a decimal digit in 4 bits and then packing two decimal digits per byte (binary code decimal format). However, binary arithmetic does not work correctly on packed decimal numbers, so special decimal-arithmetic-correction instructions are needed. These instructions need to know the carry out of bit 3. This is why the condition code often holds an auxiliary carry bit. As an aside, the infamous Y2K (Year 2000) problem was caused by COBOL programmers who decided that they could represent the year in two decimal digits (8 bits) rather than four decimal digits (or an 8-bit binary number), which can hold even more values (256) than two decimal digits (100). Some optimization!

5.2.2 Nonnumeric Data Types

Although most early computers earned their living crunching numbers, modern computers are often used for nonnumerical applications, such as email, surfing the Web, digital photography, and multimedia creation and playback. For these applications, other data types are needed and are frequently supported by ISA-level instructions. Characters are clearly important here, although not every computer provides hardware support for them. The most common character codes are ASCII and Unicode. These support 7-bit characters and 16-bit characters, respectively. Both were discussed in Chap. 2.

It is not uncommon for the ISA level to have special instructions intended for handling character strings, that is, consecutive runs of characters. These strings are sometimes delimited by a special character at the end. Alternatively a string-length field can be used to keep track of the end. The instructions can perform copy, search, edit, and other functions on the strings.

Boolean values are also important. A Boolean value can take on one of two values: true or false. In theory, a single bit can represent a Boolean, with 0 as false and 1 as true (or vice versa). In practice, a byte or word is used per Boolean value because individual bits in a byte do not have their own addresses and thus are hard to access. A common system uses the convention that 0 means false and everything else means true.

The one situation in which a Boolean value is normally represented by 1 bit is when there is an entire array of them, so a 32-bit word can hold 32 Boolean values. Such a data structure is called a **bit map** and occurs in many contexts. For example, a bit map can be used to keep track of free blocks on a disk. If the disk has n blocks, then the bit map has n bits.

Our last data type is the pointer, which is just a machine address. We have already seen pointers repeatedly. In the Mic-x machines, SP, PC, LV, and CPP are all examples of pointers. Accessing a variable at a fixed distance from a pointer, which is the way ILOAD works, is extremely common on all machines. While pointers are useful, they are also the source of a vast number of programming errors, often with very serious consequences. They must be used with great care.

5.2.3 Data Types on the Core i7

The Core i7 supports signed two's complement integers, unsigned integers, binary coded decimal numbers, and IEEE 754 floating-point numbers, as listed in Fig. 5-6. Due to its origins as a humble 8-bit/16-bit machine, it handles integers of these lengths as well as 32-bits, with numerous instructions for doing arithmetic, Boolean operations, and comparisons on them. The processor can optionally be run in 64-bit mode which also supports 64-bit registers and operations. Operands do not have to be aligned in memory, but better performance is achieved if word addresses are multiples of 4 bytes.

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	× (64-bit)
Unsigned integer	×	×	×	× (64-bit)
Binary coded decimal integer	×			
Floating point			×	×

Figure 5-6. The Core i7 numeric data types. Supported types are marked with ×. Types marked with "64-bit" are only supported in 64-bit mode.

The Core i7 is also good at manipulating 8-bit ASCII characters: there are special instructions for copying and searching character strings. These instructions can be used both with strings whose length is known in advance and with strings whose end is marked. They are often used in string manipulation libraries.

5.2.4 Data Types on the OMAP4430 ARM CPU

The OMAP4430 ARM CPU supports a wide range of data formats, as shown in Fig. 5-7. For integers alone, it can support 8-, 16-, and 32-bit operands, both signed and unsigned. The handling of small data types in the OMAP4430 is slightly more clever than in the Core i7. Internally, the OMAP4430 is 32-bit machine with 32-bit datapaths and instructions. For loads and stores, the program can specify the size and sign of the value to be loaded (e.g., load signed byte: LDRSB). The value is then converted by load instructions into a comparable 32-bit value. Similarly, stores also specify the size and sign of the value to write to memory, and they access only the specified portion of the input register.

Signed integers use two's complement. Floating-point operands of 32 and 64 bits are included and conform to the IEEE 754 standard. Binary coded decimal numbers are not supported. All operands must be aligned in memory. Character and string data types are not supported by special hardware instructions. They are manipulated entirely in software.

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	
Unsigned integer	×	×	×	
Binary coded decimal integer				
Floating point			×	×

Figure 5-7. The OMAP4430 ARM CPU numeric data types. Supported types are marked with ×.

5.2.5 Data Types on the ATmega168 AVR CPU

The ATmega168 has a very limited number of data types. With one exception, all the registers are 8 bits wide, so integers are also 8 bits wide. Characters are also 8 bits wide. In essence the only data type that is really supported by the hardware for arithmetic operations is the 8-bit byte, as shown in Fig. 5-8.

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×			
Unsigned integer	×	×		
Binary coded decimal integer				
Floating point				

Figure 5-8. The ATmega168 numeric data types. Supported types are marked with ×.

To facilitate memory accesses, the ATmega168 also includes limited support for 16-bit unsigned pointers. The 16-bit pointers X, Y, and Z, can be formed from the concatenation of 8-bit register pairs R26/R27, R28/R29, and R30/R31, respectively. When a load uses X, Y, or Z as an address operand, the processor will also optionally increment or decrement the value as needed.

5.3 INSTRUCTION FORMATS

An instruction consists of an opcode, usually along with some additional information such as where operands come from and where results go to. The general subject of specifying where the operands are (i.e., their addresses) is called **addressing** and will be discussed in detail later in this section.

Figure 5-9 shows several possible formats for level 2 instructions. An instruction always has an opcode to tell what the instruction does. There can be zero, one, two, or three addresses present.

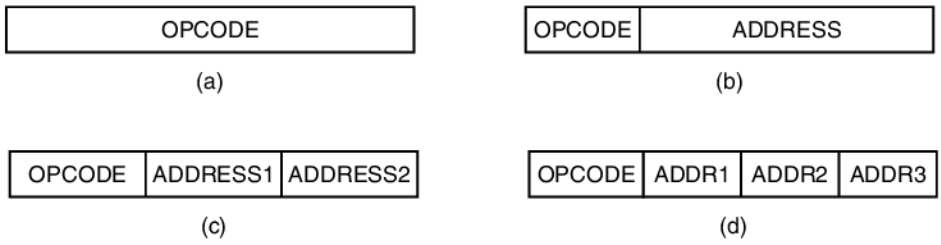


Figure 5-9. Four common instruction formats: (a) Zero-address instruction. (b) One-address instruction (c) Two-address instruction. (d) Three-address instruction.

On some machines, all instructions have the same length; on others there may be many different lengths. Instructions may be shorter than, the same length as, or longer than the word length. Having all the instructions be the same length is simpler and makes decoding easier but often wastes space, since all instructions then have to be as long as the longest one. Other trade-offs are also possible. Figure 5-10 shows some possible relationships between instruction length and word length.

5.3.1 Design Criteria for Instruction Formats

When a computer design team has to choose instruction formats for its machine, they must consider a number of factors. The difficulty of this decision should not be underestimated. The decision about the instruction format must be made early in the design of a new computer. If the computer is commercially successful, the instruction set may survive for 40 years or more. The ability to add

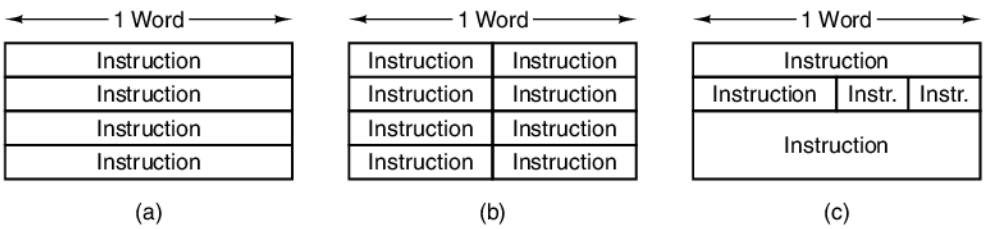


Figure 5-10. Some possible relationships between instruction and word length.

new instructions and exploit other opportunities that arise over an extended period is of great importance, but only if the architecture—and the company building it—survive long enough for the architecture to be a success.

The efficiency of a particular ISA is highly dependent on the technology with which the computer is to be implemented. Over a long period of time, this technology will undergo vast changes, and some of the ISA choices will be seen (with 20/20 hindsight) as unfortunate. For example, if memory accesses are fast, a stack-based design (like IJVM) is a good one, but if they are slow, then having many registers (like the OMAP4430 ARM CPU) is the way to go. Readers who think this choice is easy are invited to find a slip of paper and write down their predictions for (1) a typical CPU clock speed, and (2) a typical RAM access time for computers 20 years in the future. Fold this slip neatly and keep it for 20 years. Then unfold and read it. The humility-challenged can forget the slip of paper and just post their predictions to the Internet now.

Of course, even far-sighted designers may not be able to make all the right choices. And even if they could, they have to deal with the short term, too. If this elegant ISA is a little more expensive than its current ugly competitors, the company may not survive long enough for the world to appreciate the elegance of the ISA.

All things being equal, short instructions are better than long ones. A program consisting of n 16-bit instructions takes up only half as much memory space as n 32-bit instructions. With ever-declining memory prices, this factor might be less important in the future, were it not for the fact that software is metastasizing even faster than memory prices are dropping.

Furthermore, minimizing the size of the instructions may make them harder to decode or harder to overlap. Therefore, achieving the minimum instruction size must be weighed against the time required to decode and execute the instructions.

Another reason for minimizing instruction length is already important and becoming more so with faster processors: memory bandwidth (the number of bits/sec the memory is capable of supplying). The impressive growth in processor speeds over the last few decades has not been matched by equal increases in memory bandwidth. An increasingly common constraint on processors stems from the inability of the memory system to supply instructions and operands as rapidly as

the processor can consume them. Each memory has a bandwidth that is determined by its technology and engineering design. The bandwidth bottleneck applies not only to the main memory but also to all the caches.

If the bandwidth of an instruction cache is t bps and the average instruction length is r bits, the cache can deliver at most t/r instructions per second. Notice that this is an *upper limit* on the rate at which the processor can execute instructions, though there are current research efforts to breach even this seemingly insurmountable barrier. Clearly, the rate at which instructions can be executed (i.e., the processor speed) may be limited by the instruction length. Shorter instructions means a faster processor. Since modern processors can execute multiple instructions every clock cycle, fetching multiple instructions per clock cycle is imperative. This aspect of the instruction cache makes the size of instructions an important design criterion that has major implications for performance.

A second design criterion is sufficient room in the instruction format to express all the operations desired. A machine with 2^n operations with all instructions smaller than n bits is impossible. There simply will not be enough room in the opcode to indicate which instruction is needed. And history has shown over and over the folly of not leaving a substantial number of opcodes free for future additions to the instruction set.

A third criterion concerns the number of bits in an address field. Consider the design of a machine with an 8-bit character and a main memory that must hold 2^{32} characters. The designers could choose to assign consecutive addresses to units of 8, 16, 24, or 32 bits, as well as other possibilities.

Imagine what would happen if the design team degenerated into two warring factions, one advocating making the 8-bit byte the basic unit of memory, and the other advocating the 32-bit word. The former group would propose a memory of 2^{32} bytes, numbered 0, 1, 2, 3, ..., 4,294,967,295. The latter group would propose a memory of 2^{30} words numbered 0, 1, 2, 3, ..., 1,073,741,823.

The first group would point out that in order to compare two characters in the 32-bit word organization, the program would not only have to fetch the words containing the characters but would also have to extract each character from its word in order to compare them. Doing so costs extra instructions and therefore wastes space. The 8-bit organization, on the other hand, provides an address for every character, thus making the comparison much easier.

The 32-bit word supporters would retaliate by pointing out that their proposal requires only 2^{30} separate addresses, giving an address length of only 30 bits, whereas the 8-bit byte proposal requires 32 bits to address the same memory. A shorter address means a shorter instruction, which not only takes up less space but also requires less time to fetch. Alternatively, they could retain the 32-bit address to reference a 16-GB memory instead of a puny 4-GB memory.

This example demonstrates that in order to gain a finer memory resolution, one must pay the price of longer addresses and thus longer instructions. The ultimate in resolution is a memory organization in which every bit is directly addressable

(e.g., the Burroughs B1700). At the other extreme is a memory consisting of very long words (e.g., the CDC Cyber series had 60-bit words).

Modern computer systems have arrived at a compromise that, in some sense, captures the worst of both. They require all the bits necessary to address individual bytes, but memory accesses read one, two, or sometimes four words at a time. Reading 1 byte from memory on the Core i7, for example, brings in a minimum of 8 bytes and probably an entire 64-byte cache line.

5.3.2 Expanding Opcodes

In the preceding section we saw how short addresses and good memory resolution could be traded off against each other. In this section we will examine new trade-offs, involving both opcodes and addresses. Consider an $(n + k)$ bit instruction with a k-bit opcode and a single n-bit address. This instruction allows 2^k different operations and 2^n addressable memory cells. Alternatively, the same $n + k$ bits could be broken up into a $(k - 1)$ bit opcode, and an $(n + 1)$ bit address, meaning only half as many instructions but either twice as much memory addressable, or the same amount of memory but with twice the resolution. A $(k + 1)$ bit opcode and an $(n - 1)$ bit address gives more operations, but the price is either a smaller number of cells addressable, or poorer resolution and the same amount of memory addressable. Quite sophisticated trade-offs are possible between opcode bits and address bits as well as the simpler ones just described. The scheme discussed in the following paragraphs is called an **expanding opcode**.

The concept of an expanding opcode can be most clearly seen by a simple example. Consider a machine in which instructions are 16 bits long and addresses are 4 bits long, as shown in Fig. 5-11. This situation might be reasonable for a machine that has 16 registers (hence a 4-bit register address) on which all arithmetic operations take place. One design would be a 4-bit opcode and three addresses in each instruction, giving 16 three-address instructions.

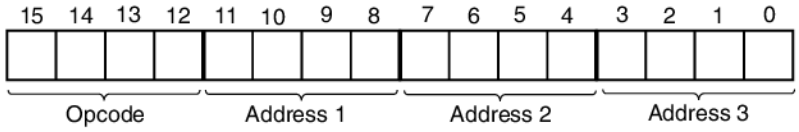


Figure 5-11. An instruction with a 4-bit opcode and three 4-bit address fields.

However, if the designers need 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 instructions with no address at all, they can use opcodes 0 to 14 as three-address instructions but interpret opcode 15 differently (see Fig. 5-12).

Opcode 15 means that the opcode is contained in bits 8 to 15 instead of in bits 12 to 15. Bits 0 to 3 and 4 to 7 form two addresses, as usual. The 14 two-address

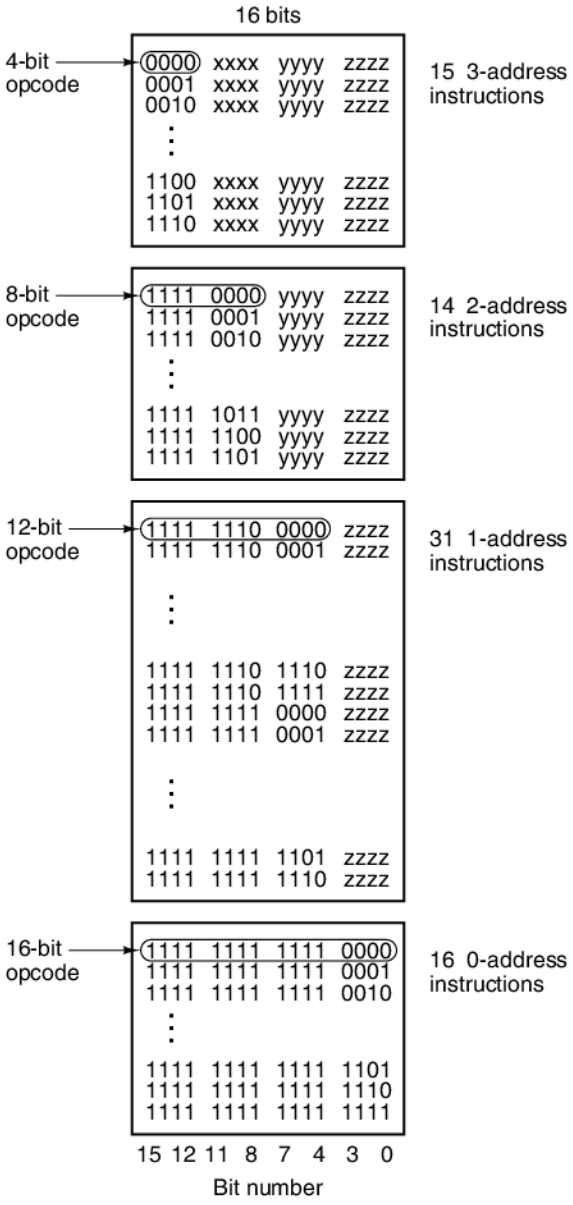


Figure 5-12. An expanding opcode allowing 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 zero-address instructions. The fields marked *xxxx*, *yyyy*, and *zzzz* are 4-bit address fields.

instructions all have 1111 in the leftmost 4 bits, and numbers from 0000 to 1101 in bits 8 to 11. Instructions with 1111 in the leftmost 4 bits and either 1110 or 1111 in bits 8 to 11 will be treated specially. They will be treated as though their opcodes were in bits 4 to 15. The result is 32 new opcodes. Because only 31 are needed, opcode 111111111111 is interpreted to mean that the real opcode is in bits 0 to 15, giving 16 instructions with no address.

As we proceeded through this discussion, the opcode got longer and longer: the three-address instructions have a 4-bit opcode, the two-address instructions have an 8-bit opcode, the one-address instructions have a 12-bit opcode, and the zero-address instructions have a 16-bit opcode.

The idea of expanding opcodes demonstrates a trade-off between the space for opcodes and space for other information. In practice, expanding opcodes are not quite as clean and regular as in our example. In fact, the ability to use variable sizes of opcodes can be exploited in either of two ways. First, the instructions can all be kept the same length, by assigning the shortest opcodes to the instructions that need the most bits to specify other things. Second, the size of the *average* instruction can be minimized by choosing opcodes that are shortest for common instructions and longest for rare instructions.

Carrying the idea of variable-length opcodes to an extreme, it is possible to minimize the average instruction length by encoding every instruction to minimize the number of bits needed. Unfortunately, this would result in instructions of various sizes not even aligned on byte boundaries. While there have been ISAs that had this property (for example, the ill-fated Intel 432), the importance of alignment is so great for the rapid decoding of instructions that this degree of optimization is almost certainly counterproductive.

5.3.3 The Core i7 Instruction Formats

The Core i7 instruction formats are highly complex and irregular, having up to six variable-length fields, of which five are optional. The general pattern is shown in Fig. 5-13. This state of affairs occurred because the architecture evolved over many generations and included some poor choices early on. In the name of backward compatibility, these early decisions could not be reversed later. In general, for two-operand instructions, if one operand is in memory, the other may not be in memory. Thus instructions exist to add two registers, add a register to memory, and add memory to a register, but not to add a memory word to another memory word.

On earlier Intel architectures, all opcodes were 1 byte, though the concept of a prefix byte was used extensively for modifying some instructions. A **prefix byte** is an extra opcode stuck onto the front of an instruction to change its action. The WIDE instruction in IJVM is an example of a prefix byte. Unfortunately, at some point during the evolution, Intel ran out of opcodes, so one opcode, 0xFF, was designated as an **escape code** to permit a second instruction byte.

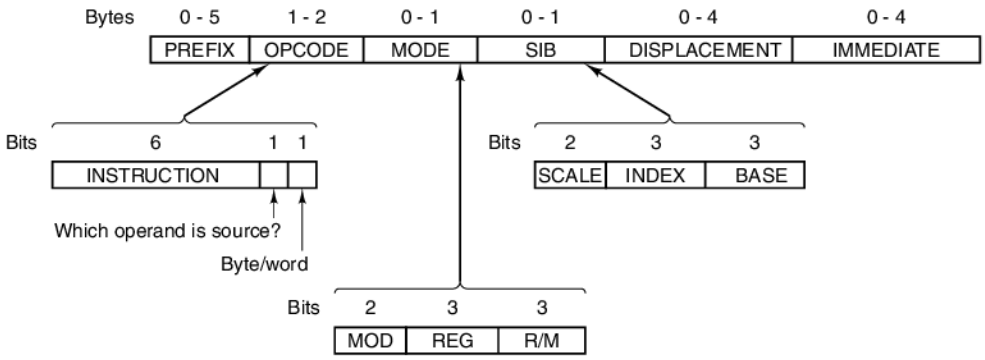


Figure 5-13. The Core i7 instruction formats.

The individual bits in the Core i7 opcodes do not give much information about the instruction. The only structure in the opcode field is the use of the low-order bit in some instructions to indicate byte/word, and the use of the adjoining bit to indicate whether the memory address (if it is present) is the source or the destination. Thus in general, the opcode must be fully decoded to determine what class of operation is to be performed—and thus how long the instruction is. This makes high-performance implementations difficult, since extensive decoding is necessary before it can even be determined where the next instruction starts.

Following the opcode byte in most instructions that reference an operand in memory is a second byte that tells all about the operand. These 8 bits are split up into a 2-bit MOD field and two 3-bit register fields, REG and R/M. Sometimes the first 3 bits of this byte are used as an extension for the opcode, giving a total 11 bits for the opcode. However, the 2-bit mode field means that there are only four ways to address operands and one of the operands must always be a register. Logically, any of EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP should be specifiable as either register, but the encoding rules prohibit some combinations and use them for special cases. Some modes require an additional byte, called **SIB** (**Scale, Index, Base**), giving a further specification. This scheme is not ideal, but a compromise given the competing demands of backward compatibility and the desire to add new features not originally envisioned.

In addition to all this, some instructions have 1, 2, or 4 more bytes specifying a memory address (displacement) and possibly another 1, 2, or 4 bytes containing a constant (immediate operand).

5.3.4 The OMAP4430 ARM CPU Instruction Formats

The OMAP4430 ARM ISA consists of both 16- and 32-bit instructions, aligned in memory. Instructions are generally simple, specifying only a single action. A typical arithmetic instruction specifies two registers to supply the source

operands and a single destination register. The 16-bit instructions are pared-down versions of the 32-bit instruction. They perform the same operations, but allow only two register operands (i.e., the destination register must be the same as one of the inputs) and only the first eight registers can be specified as inputs. The ARM architects called this smaller version of the ARM ISA the Thumb ISA.

Additional variants allows instructions to supply a 3, 8, 12, 16, or 24-bit unsigned constant instead of one of the registers. For a load instruction, two registers (or one register and an 8-bit signed constant) are added together to specify the memory address to read. The data are written into the other register specified.

The format of the 32-bit ARM instructions is illustrated in Fig. 5-14. The careful reader will notice that some of the formats have the same fields (e.g., LONG MULTIPLY and SWAP). In the case of the SWAP instruction, the decoder knows that the instruction is a SWAP only when it sees that the combination of field values for the MUL is illegal. Additional formats have been added for instruction extensions and the Thumb ISA. At the time of this writing, the number of instruction formats was 21 and rising. (Can it be long before we see some company advertising the "World's most complex RISC machine"?) The majority of instructions, however, still use the formats shown in the figure.

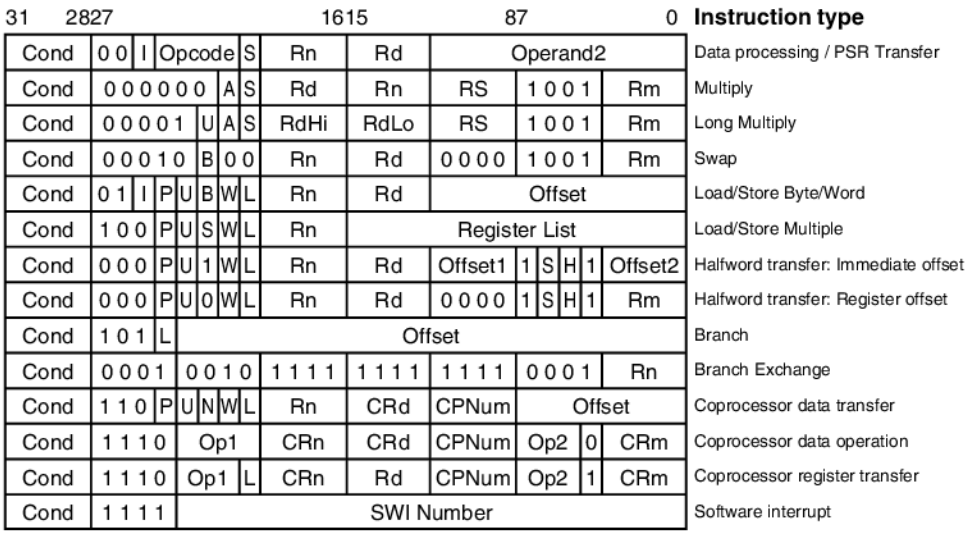


Figure 5-14. The 32-bit ARM instruction formats.

Bits 26 and 27 of every instruction are the first stop in determining the instruction format and tell hardware where to find the rest of the opcode, if there is more. For example, if bits 26 and 27 are both zero, and bit 25 is zero (operand is not an immediate), and the input operand shift is not illegal (which indicates the instruction is a multiply or branch exchange), then both sources are registers. If bit 25 is one, then one source is a register and the other is a constant in the range 0 to 4095.

In both cases, the destination is always a register. Sufficient encoding space is provided for up to 16 instructions, all of which are currently used.

With 32-bit instructions, it is not possible to include a 32-bit constant in the instruction. The MOVT instruction sets the 16 upper bits of a 32-bit register, leaving room for another instruction to set the remaining lower 16 bits. It is the only instruction to use this format.

Every 32-bit instruction has the same 4-bit field in the most significant bits (bits 28 to 31). This is the condition field, which makes any instruction a **predicated instruction**. A predicated instruction executes as normal in the processor, but before writing its result to a register (or memory), it first checks the condition of the instruction. For ARM instructions, the condition is based on the state of the processor status register (PSR). This register holds the arithmetic properties of the last arithmetic operation, (e.g., zero, negative, overflowed, etc). If the condition is not met, the result of the conditional instruction is dropped.

The branch instruction format encodes the largest immediate value, used to compute a target address for branches and function procedure calls. This instruction is special, because it is the only one where 24 bits of data are needed to specify an address. For this instruction, there is a single, 3-bit opcode. The address is the target address divided by four, making the range approximately $\pm 2^{25}$ bytes relative to the current instruction.

Clearly, the ARM ISA designers wanted to fully utilize every bit combination, including otherwise illegal operand combinations, for specifying instructions. The approach makes for extremely complicated decoding logic, but at the same time, it allows the maximum number of operations to be encoded into a fixed-length 16- or 32-bit instruction.

5.3.5 The ATmega168 AVR Instruction Formats

The ATmega168 has six simple instruction formats, as illustrated in Fig. 5-15. Instructions are 2 or 4 bytes in length. Format one consists of an opcode and two register operands, both of which are inputs and one is also the output of the instruction. The ADD instruction for registers uses this format, for example.

Format 2 is also 16 bits, consisting of an additional 16 opcodes and a 5-bit register number. This format increases the number of operations encoded in the ISA at the cost of reducing the number of instruction operands to one. Instructions that use this format perform a unary operation, taking a single register input and writing the output of the operation to the same register. Examples of this type of instruction include "negate" and "increment."

Format 3 has an 8-bit unsigned immediate operand. To accommodate such a large immediate value in a 16-bit instruction, instructions which use this encoding can have only one register operand (used as an input and output) and the register can only be R16–R31 (which limits the operand encoding to 4 bits). Also, the

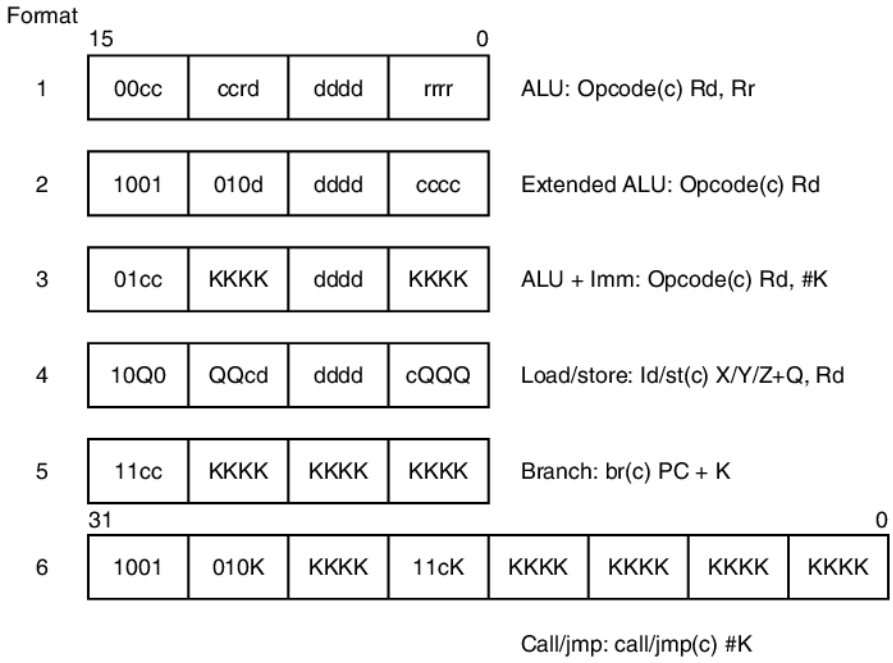


Figure 5-15. The ATmega168 AVR instruction formats.

number of opcode bits is cut in half, allowing only four instructions to use this format (SUBCI, SUBI, ORI, and ANDI).

Format 4 encodes load and store instruction, which includes a 6-bit unsigned immediate operand. The base register is a fixed register not specified in the instruction encoding because it is implied by the load/store opcode.

Formats 5 and 6 are used for jumps and procedure calls. The first format includes a 12-bit signed immediate value that is added to the instruction's PC value to compute the target of the instruction. The last format expands the offset to 22 bits, by making the AVR instruction 32 bits in size.

5.4 ADDRESSING

Most instructions have operands, so some way is needed to specify where they are. This subject, which we will now discuss, is called **addressing**.

5.4.1 Addressing Modes

Up until now, we have paid little attention to how the bits of an address field are interpreted to find the operand. It is now time to investigate this subject, called **address modes**. As we shall see, there are many ways to do it.

5.4.2 Immediate Addressing

The simplest way for an instruction to specify an operand is for the address part of the instruction actually to contain the operand itself rather than an address or other information describing where the operand is. Such an operand is called an **immediate operand** because it is automatically fetched from memory at the same time the instruction itself is fetched; hence it is immediately available for use. A possible immediate instruction for loading register R1 with the constant 4 is shown in Fig. 5-16.

Figure 5-16. An immediate instruction for loading 4 into register 1.

Immediate addressing has the virtue of not requiring an extra memory reference to fetch the operand. It has the disadvantage that only a constant can be supplied this way. Also, the number of values is limited by the size of the field. Still, many architectures use this technique for specifying small integer constants.

5.4.3 Direct Addressing

A method for specifying an operand in memory is just to give its full address. This mode is called **direct addressing**. Like immediate addressing, direct addressing is restricted in its use: the instruction will always access exactly the same memory location. So while the value can change, the location cannot. Thus direct addressing can only be used to access global variables whose address is known at compile time. Nevertheless, many programs have global variables, so this mode is widely used. The details of how the computer knows which addresses are immediate and which are direct will be discussed later.

5.4.4 Register Addressing

Register addressing is conceptually the same as direct addressing but specifies a register instead of a memory location. Because registers are so important (due to fast access and short addresses) this addressing mode is the most common one on most computers. Many compilers go to great lengths to determine which variables will be accessed most often (for example, a loop index) and put these variables in registers.

This addressing mode is known simply as **register mode**. In load/store architectures such as the OMAP4420 ARM architecture, nearly all instructions use this addressing mode exclusively. The only time this addressing mode is not used is when an operand is moved from memory into a register (LDR instruction) or from a register to memory (STR instruction). Even for those instructions, one of the operands is a register—where the memory word is to come from or go to.

5.4.5 Register Indirect Addressing

In this mode, the operand being specified comes from memory or goes to memory, but its address is not hardwired into the instruction, as in direct addressing. Instead, the address is contained in a register. An address used in this manner is called a **pointer**. A big advantage of register indirect addressing is that it can reference memory without paying the price of having a full memory address in the instruction. It can also use different memory words on different executions of the instruction.

To see why using a different word on each execution might be useful, imagine a loop that steps through the elements of a 1024-element one-dimensional integer array to compute the sum of the elements in register R1. Outside the loop, some other register, say, R2, can be set to point to the first element of the array, and another register, say, R3, can be set to point to the first address beyond the array. With 1024 integers of 4 bytes each, if the array begins at *A*, the first address beyond the array will be *A* + 4096. Typical assembly code for doing this calculation is shown in Fig. 5-17 for a two-address machine.

```
        MOV R1,#0          ; accumulate the sum in R1, initially 0
        MOV R2,#A          ; R2 = address of the array A
        MOV R3,#A+4096     ; R3 = address of the first word beyond A
LOOP:   ADD R1,(R2)        ; register indirect through R2 to get operand
        ADD R2,#4          ; increment R2 by one word (4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

Figure 5-17. A generic assembly program for computing the sum of the elements of an array.

In this little program, we use several addressing modes. The first three instructions use register mode for the first operand (the destination) and immediate mode for the second operand (a constant indicated by the # sign). The second instruction puts the *address* of *A* in R2, not the contents. That is what the # sign tells the assembler. Similarly, the third instruction puts the address of the first word beyond the array in R3.

What is interesting to note is that the body of the loop itself does not contain any memory addresses. It uses register and register indirect mode in the fourth instruction. It uses register and immediate mode in the fifth instruction and register mode twice in the sixth instruction. The BLT might use a memory address, but more likely it specifies the address to branch to with an 8-bit offset relative to the BLT instruction itself. By avoiding the use of memory addresses completely, we have produced a short, fast loop. As an aside, this program is really for the Core i7, except that we have renamed the instructions and registers and changed the

notation to make it easy to read because the Core i7's standard assembly-language syntax (MASM) verges on the bizarre, a remnant of the machine's former life as an 8088.

It is worth noting that, in theory, there is another way to do this computation without using register indirect addressing. The loop could have contained an instruction to add *A* to R1, such as

```
ADD R1,A
```

Then on each iteration of the loop, the instruction itself could be incremented by 4, so that after one iteration it read

```
ADD R1,A+4
```

and so on until it was done.

A program that modifies itself like this is called a **self-modifying** program. The idea was thought of by none other than John von Neumann and made sense on early computers, which did not have register indirect addressing. Nowadays, self-modifying programs are considered horrible style and hard to understand. They also cannot be shared among multiple processes at the same time. Furthermore, they will not even work correctly on machines with a split level 1 cache if the I-cache has no circuitry for doing writebacks (because the designers assumed that programs do not modify themselves). Lastly, self-modifying programs will also fail on machines with separate instruction and data spaces. All in all, this is an idea that has come and (fortunately) gone.

5.4.6 Indexed Addressing

It is frequently useful to be able to reference memory words at a known offset from a register. We saw some examples with IJVM where local variables are referenced by giving their offset from LV. Addressing memory by giving a register (explicit or implicit) plus a constant offset is called **indexed addressing**.

Local variable access in IJVM uses a pointer into memory (LV) in a register plus a small offset in the instruction itself, as shown in Fig. 4-19(a). However, it is also possible to do it the other way: the memory pointer in the instruction and the small offset in the register. To see how that works, consider the following calculation. We have two one-dimensional arrays of 1024 words each, *A* and *B*, and we wish to compute A_i AND B_i for all the pairs and then OR these 1024 Boolean products together to see if there is at least one nonzero pair in the set. One approach would be to put the address of *A* in one register, the address of *B* in a second register, and then step through them together in lockstep, analogous to what we did in Fig. 5-17. This way of doing it would certainly work, but it can be done in a better, more general way, as illustrated in Fig. 5-18.

```
        MOV R1,#0          ; accumulate the OR in R1, initially 0
        MOV R2,#0          ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096       ; R3 = first index value not to use
LOOP:   MOV R4,A(R2)       ; R4 = A[i]
        AND R4,B(R2)       ; R4 = A[i] AND B[i]
        OR R1,R4           ; OR all the Boolean products into R1
        ADD R2,#4          ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

Figure 5-18. A generic assembly program for computing the OR of A_i AND B_i for two 1024-element arrays.

Operation of this program is straightforward. We need four registers here:

1. R1 — Holds the accumulated OR of the Boolean product terms.
2. R2 — The index, *i*, that is used to step through the arrays.
3. R3 — The constant 4096, which is the lowest value of *i* not to use.
4. R4 — A scratch register for holding each product as it is formed.

After initializing the registers, we enter the six-instruction loop. The instruction at *LOOP* fetches A_i into R4. The calculation of the source here uses indexed mode. A register, R2, and a constant, the address of *A*, are added together and used to reference memory. The sum of these two quantities goes to the memory but is not stored in any user-visible register. The notation

```
MOV R4,A(R2)
```

means that the destination uses register mode with R4 as the register and the source uses indexed mode, with *A* as the offset and R2 as the register. If *A* has the value, say, 124300, the actual machine instruction for this is likely to look something like the one shown in Fig. 5-19.

MOV	R4	R2	124300

Figure 5-19. A possible representation of MOV R4,A(R2).

The first time through the loop, R2 is 0 (due to it being initialized that way), so the memory word addressed is A_0, at address 124300. This word is loaded into R4. The next time though the loop, R2 is 4, so the memory word addressed is A_1, at 124304, and so on.

As we promised earlier, here the offset in the instruction itself is the memory pointer and the value in the register is a small integer that is incremented during the

calculation. This form requires an offset field in the instruction large enough to hold an address, of course, so it is less efficient than doing it the other way; however, it is nevertheless frequently the best way.

5.4.7 Based-Indexed Addressing

Some machines have an addressing mode in which the memory address is computed by adding up two registers plus an (optional) offset. Sometimes this mode is called **based-indexed addressing**. One of the registers is the base and the other is the index. Such a mode would have been useful here. Outside the loop we could have put the address of *A* in R5 and the address of *B* in R6. Then we could have replaced the instruction at *LOOP* and its successor with

```
LOOP:   MOV R4,(R2+R5)
        AND R4,(R2+R6)
```

If there were an addressing mode for indirecting through the sum of two registers with no offset, that would be ideal. Alternatively, even an instruction with an 8-bit offset would have been an improvement over the original code since we could set both offsets to 0. If, however, the offsets are always 32 bits, then we have not gained anything by using this mode. In practice, however, machines that have this mode usually have a form with an 8-bit or 16-bit offset.

5.4.8 Stack Addressing

We have already noted that making machine instructions as short as possible is highly desirable. The ultimate limit in reducing address lengths is having no addresses at all. As we saw in Chap. 4, zero-address instructions, such as IADD, are possible in conjunction with a stack. In this section we will look at stack addressing more closely.

Reverse Polish Notation

It is an ancient tradition in mathematics to put the operator between the operands, as in $x + y$, rather than after the operands, as in $x\ y +$. The form with the operator "in" between the operands is called **infix** notation. The form with the operator after the operands is called **postfix** or **reverse Polish notation**, after the Polish logician J. Lukasiewicz (1958), who studied the properties of this notation.

Reverse Polish notation has a number of advantages over infix for expressing algebraic formulas. First, any formula can be expressed without parentheses. Second, it is convenient for evaluating formulas on computers with stacks. Third, infix operators have precedence, which is arbitrary and undesirable. For example, we

know that $a \times b + c$ means $(a \times b) + c$ and not $a \times (b + c)$ because multiplication has been arbitrarily defined to have precedence over addition. But does left shift have precedence over Boolean AND? Who knows? Reverse Polish notation eliminates this nuisance.

Several algorithms for converting infix formulas into reverse Polish notation exist. The one given below is an adaptation of an idea due to E. W. Dijkstra. Assume that a formula is composed of the following symbols: variables, the dyadic (two-operand) operators + − * /, and left and right parentheses. To mark the ends of a formula, we will insert the symbol ⊥ after the last symbol and before the first symbol.

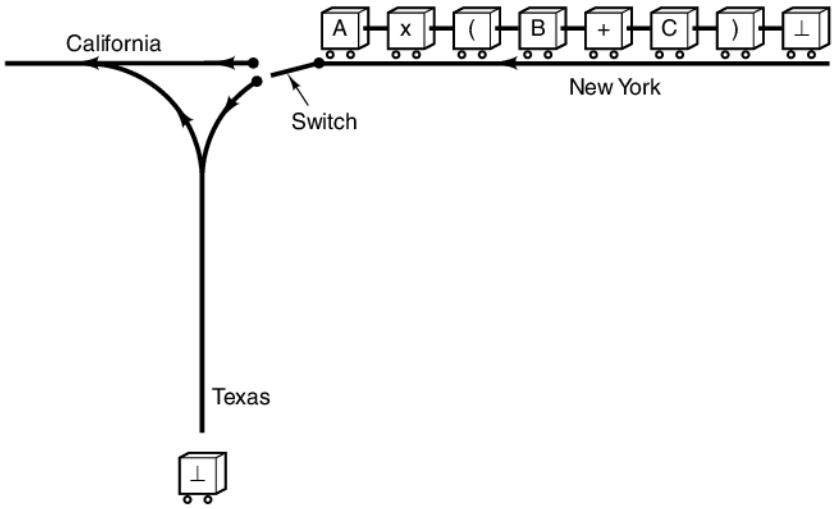


Figure 5-20. Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish notation.

Figure 5-20 shows a railroad track from New York to California, with a spur in the middle that heads off in the direction of Texas. Think of the New York to California line as the main line with the branch down to Texas as a siding for temporary storage. The names and directions are not important. What matters is the distinction between the main line and the alternative place for temporarily storing things. Each symbol in the formula is represented by one railroad car. The train moves westward (to the left). When each car arrives at the switch, it must stop just before it and ask if it should go to California directly or take a side trip to Texas. Cars containing variables always go directly to California and never to Texas. Cars containing all other symbols must inquire about the contents of the nearest car on the Texas line before entering the switch.

The table of Fig. 5-21 shows what happens, depending on the contents of the nearest car on the Texas line and the car poised at the switch. The first ⊥ always goes to Texas. The numbers refer to the following situations:

1. The car at the switch heads toward Texas.
2. The most recent car on the Texas line turns and goes to California.
3. Both the car at the switch and the most recent car on the Texas line are diverted and disappear (i.e., both are deleted).
4. Stop. The symbols now in California represent the reverse Polish notation formula when read from left to right.
5. Stop. An error has occurred. The original formula was not correctly balanced.

Most recently arrived car on the Texas line \ Car at the switch	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Figure 5-21. Decision table used by the infix-to-reverse Polish notation algorithm

After each action is taken, a new comparison is made between the car currently at the switch, which may be the same car as in the previous comparison or may be the next car, and the car that is now the last one on the Texas line. The process continues until step 4 is reached. Notice that the Texas line is being used as a stack, with routing a car to Texas being a push operation, and turning a car already on the Texas line around and sending it to California being a pop operation.

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C − D)	A B + C D − /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Figure 5-22. Some examples of infix expressions and their reverse Polish notation equivalents.

The order of the variables is the same in infix and in reverse Polish notation. The order of the operators, however, is not always the same. Operators appear in

reverse Polish notation in the order they will actually be executed during the evaluation of the expression. Figure 5-22 gives several examples of infix formulas and their reverse Polish notation equivalents.

Evaluation of Reverse Polish Notation Formulas

Reverse Polish notation is the ideal notation for evaluating formulas on a computer with a stack. The formula consists of n symbols, each one either an operand or an operator. The algorithm for evaluating a reverse Polish notation formula using a stack is simple. Scan the reverse Polish notation string from left to right. When an operand is encountered, push it onto the stack. When an operator is encountered, execute the corresponding instruction.

Figure 5-23 shows the evaluation of

$$(8 + 2 \times 5) / (1 + 3 \times 2 - 4)$$

in IJVM. The corresponding reverse Polish notation formula is

$$8\ 2\ 5 \times +\ 1\ 3\ 2 \times +\ 4 - /$$

In the figure, we have introduced IMUL and IDIV as the multiplication and division instructions, respectively. The number on top of the stack is the right operand, not the left operand. This point is important for division (and subtraction) since the order of the operands is significant (unlike addition and multiplication). In other words, IDIV has been carefully defined so that first pushing the numerator, then pushing the denominator, and then doing the operation gives the correct result. Notice how easy code generation is from reverse Polish notation to IJVM: just scan the reverse Polish notation formula and output one instruction per symbol. If the symbol is a constant or variable, output an instruction to push it onto the stack. If the symbol is an operator, output an instruction to perform the operation.

5.4.9 Addressing Modes for Branch Instructions

So far we have been looking only at instructions that operate on data. Branch instructions (and procedure calls) also need addressing modes for specifying the target address. The modes we have examined so far also work for branches for the most part. Direct addressing is certainly a possibility, with the target address simply being included in the instruction in full.

However, other addressing modes also make sense. Register indirect addressing allows the program to compute the target address, put it in a register, and then go there. This mode gives the most flexibility since the target address is computed at run time. It also presents the greatest opportunity for creating bugs that are nearly impossible to find.

Another reasonable mode is indexed mode, which offsets a known distance from a register. It has the same properties as register indirect mode.

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Figure 5-23. Use of a stack to evaluate a reverse Polish notation formula.

Another option is PC-relative addressing. In this mode, the (signed) offset in the instruction itself is added to the program counter to get the target address. In fact, this is simply indexed mode, using PC as the register.

5.4.10 Orthogonality of Opcodes and Addressing Modes

From a software point of view, instructions and addressing should have a regular structure, with a minimum number of instruction formats. Such a structure makes it much easier for a compiler to produce good code. All opcodes should permit all addressing modes wherever that makes sense. Furthermore, all registers should be available for all register modes [including the frame pointer (FP), stack pointer (SP), and program counter (PC)].

As an example of a clean design for a three-address machine, consider the 32-bit instruction formats of Fig. 5-24. Up to 256 opcodes are supported. In format 1, each instruction has two source registers and a destination register. All arithmetic and logical instructions use this format.

The unused 8-bit field at the end can be used for further instruction differentiation. For example, one opcode could be allocated for all the floating-point operations, with the extra field distinguishing among them. In addition, if bit 23 is set, format 2 is used and the second operand is no longer a register but a 13-bit signed immediate constant. LOAD and STORE instructions can also use this format to reference memory in indexed mode.

A small number of additional instructions are needed, such as conditional branches, but they could easily fit in format 3. For example, one opcode could be

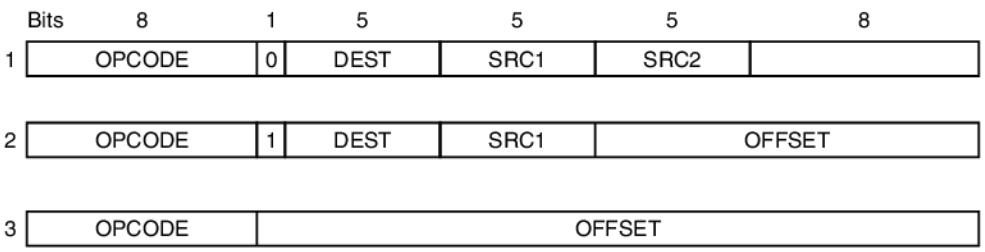


Figure 5-24. A simple design for the instruction formats of a three-address machine.

assigned to each (conditional) branch, procedure call, etc., leaving 24 bits for a PC relative offset. Assuming that this offset counted in words, the range would be ±32 MB. Also a few opcodes could be reserved for LOADs and STOREs that need the long offsets of format 3. These would not be fully general (e.g., only R0 could be loaded or stored), but they would rarely be used.

Now consider a design for a two-address machine that can use a memory word for either operand. It is shown in Fig. 5-25. Such a machine can add a memory word to a register, add a register to a memory word, add a register to a register, or add a memory word to a memory word. At present, memory accesses are relatively expensive, so this design is not currently popular, but if advances in cache or memory technology make memory accesses cheap in the future, it is a particularly easy and efficient design to compile to. The PDP-11 and VAX were highly successful machines that dominated the minicomputer world for two decades using designs similar to this one.

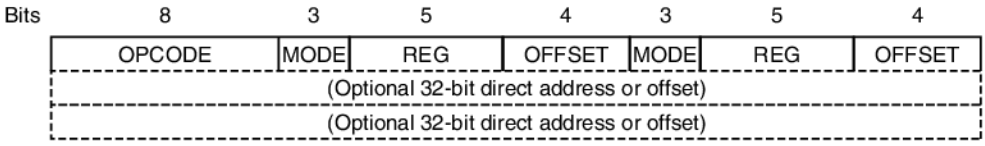


Figure 5-25. A simple design for the instruction formats of a two-address machine.

In this design, we again have an 8-bit opcode, but now we have 12 bits for specifying the source and an additional 12 bits for specifying the destination. For each operand, 3 bits give the mode, 5 bits tell the register, and 4 bits provide the offset. With 3 mode bits, we could support immediate, direct, register, register indirect, indexed, and stack modes, and have room left over for two more future modes. This is a clean and regular design that is easy to compile for and quite flexible, especially if the program counter, stack pointer, and local variable pointer are among the general registers that can be accessed the usual way.

The only problem here is that for direct addressing, we need more bits for the address. What the PDP-11 and VAX did was add an extra word to the instruction for the address of each directly addressed operand. We could also use one of the two available addressing modes for an indexed mode with a 32-bit offset following the instruction. Thus in the worst case, say, a memory-to-memory ADD with both operands directly addressed or using the long indexed form, the instruction would be 96 bits long and use three bus cycles (one for the instruction, two for its addresses). In addition, three more cycles would be needed to fetch the two operands and write the result. On the other hand, most RISC designs would require at least 96 bits, probably more, to add an arbitrary word in memory to another arbitrary word in memory and use at least four bus cycles, depending how the operands were addressed.

Many alternatives to Fig. 5-25 are possible. In this design, it is possible to execute the statement

```
i = j;
```

in one 32-bit instruction, provided that both *i* and *j* are among the first 16 local variables. On the other hand, for variables beyond 16, we have to go to 32-bit offsets. One option would be another format with a single 8-bit offset instead of two 4-bit offsets, plus a rule saying that either the source or the destination could use it, but not both. The possibilities and trade-offs are unlimited, and machine designers must juggle many factors to get a good result.

5.4.11 The Core i7 Addressing Modes

The Core i7's addressing modes are highly irregular and are different depending on whether a particular instruction is in 16-, 32-, or 64-bit mode. Below we will ignore the 16- and 64-bit modes; 32-bit mode is bad enough. The modes supported include immediate, direct, register, register indirect, indexed, and a special mode for addressing array elements. The problem is that not all modes apply to all instructions and not all registers can be used in all modes. This makes the compiler writer's job much more difficult and leads to worse code.

The MODE byte in Fig. 5-13 controls the addressing modes. One of the operands is specified by the combination of the MOD and R/M fields. The other is always a register and is given by the value of the REG field. The 32 combinations that can be specified by the 2-bit MOD field and the 3-bit R/M field are listed in Fig. 5-26. If both fields are zero, for example, the operand is read from the memory address contained in the EAX register.

The 01 and 10 columns involve modes in which a register is added to an 8- or 32-bit offset that follows the instruction. If an 8-bit offset is selected, it is first sign-extended to 32 bits before being added. For example, an ADD instruction with R/M = 011, MOD = 01, and an offset of 6 computes the sum of EBX and 6 and reads the memory word at that address for one of the operands. EBX is not modified.

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

Figure 5-26. The Core i7 32-bit addressing modes. M[x] is the memory word at x.

The MOD = 11 column gives a choice of two registers. For word instructions, the first choice is used; for byte instructions, the second choice. Observe that the table is not entirely regular. For example, there is no way to indirect through EBP and no way to offset from ESP.

In some modes, an additional byte, called **SIB** (**Scale, Index, Base**), follows the MODE byte (see Fig. 5-13). The SIB byte specifies a scale factor as well as two registers. When a SIB byte is present, the operand address is computed by multiplying the index register by 1, 2, 4, or 8 (depending on SCALE), adding it to the base register, and finally possibly adding an 8- or 32-bit displacement, depending on MOD. Almost all the registers can be used as either index or base.

The SIB modes are useful for accessing array elements. For example, consider the Java statement

```
for (i = 0; i < n; i++) a[i] = 0;
```

where a is an array of 4-byte integers local to the current procedure. Typically, EBP is used to point to the base of the stack frame containing the local variables and arrays, as shown in Fig. 5-27. The compiler might keep i in EAX. To access $a[i]$, it would use an SIB mode whose operand address was the sum of 4 × EAX, EBP, and 8. This instruction could store into $a[i]$ in a single instruction.

Is this mode worth the trouble? It is hard to say. Undoubtedly this instruction, when properly used, saves a few cycles. How often it is used depends on the compiler and the application. The problem is that this instruction occupies a certain amount of chip area that could have been used in a different way had this instruction not been present. For example, the level 1 cache could have been made bigger, or the chip could have been made smaller, possibly allowing a slightly higher clock speed.

These are the kinds of trade-offs that designers face constantly. Usually, extensive simulation runs are made before casting anything in silicon, but these simulations require having a good idea of what the workload is like. It is a safe bet that

the designers of the 8088 did not include a Web browser in their test set. Nevertheless, quite a few of that product's descendants are now used primarily for Web surfing, so the decisions made 20 years ago may be completely wrong for current applications. However, in the name of backward compatibility, once a feature gets in there, it is impossible to get it out.

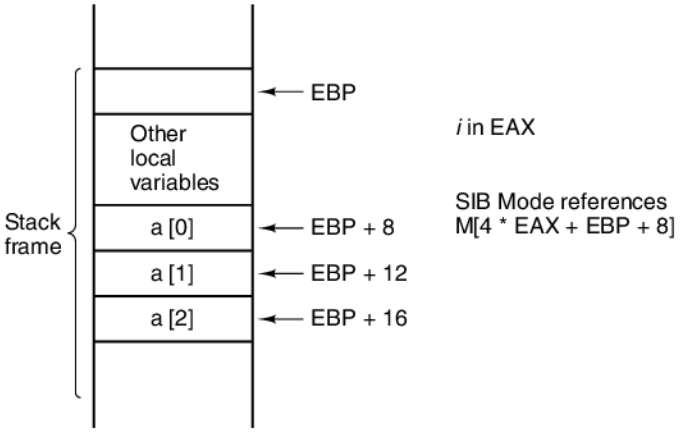


Figure 5-27. Access to $a[i]$.

5.4.12 The OMAP4440 ARM CPU Addressing Modes

In the OMAP4430, all instructions use immediate or register addressing except those that address memory. For register mode, the 5 bits simply tell which register to use. For immediate mode, an (unsigned) 12-bit constant provides the data. No other modes are present for the arithmetic, logical, and similar instructions.

Two kinds of instructions address memory: loads (LDR) and stores (STR). LDR and STR instructions have three modes for addressing memory. The first mode computes the sum of two registers and then indirects through it. The second mode computes the address as the sum of a base register and a 13-bit signed offset. The third addressing mode computes an address equal to the program counter (PC) plus a 13-bit signed offset. This third addressing mode, called PC-relative addressing, is useful for loading program constants which are stored with the program's code.

5.4.13 The ATmega168 AVR Addressing Modes

The ATmega168 has a fairly regular addressing structure. There are four basic modes. The first is register mode, in which the operand is in a register. Registers can be used as both sources and destinations. The second is immediate mode, where an 8-bit unsigned immediate value can be encoded into an instruction.

The remaining modes are usable only by load and store instructions. The third mode is direct addressing, where the operand is in memory at an address contained in the instruction itself. For 16-bit instructions, the direct address is limited to 7 bits (thus only addresses 0 to 127 can be loaded). The AVR architecture also defines a 32-bit instruction as well that accommodates a 16-bit direct address, which supports up to 64 KB of memory.

The fourth mode is register indirect, in which a register contains a pointer to the operand. Since the normal registers are 8 bits wide, the load and store instructions use register pairs to express a 16-bit address. A register pair can address up to 64 KB of memory. The architecture supports the use of three register pairs: X, Y, and Z, which are formed from the register pairs R26/R27, R28/R29, and R30/R31, respectively. To load an address into the X register for example, the program would have to load an 8-bit value into the R26 and R27 registers, using two load instructions.

5.4.14 Discussion of Addressing Modes

We have now studied quite a few addressing modes. The ones used by the Core i7, OMAP4430, and ATmega168 are summarized in Fig. 5-28. As we have pointed out, however, not every mode can be used in every instruction.

Addressing mode	Core i7	OMAP4430 ARM	ATmega168 AVR
Immediate	×	×	×
Direct	×		×
Register	×	×	×
Register indirect	×	×	×
Indexed	×	×	
Based-indexed		×	

Figure 5-28. A comparison of addressing modes.

In practice, not many addressing modes are needed for an effective ISA. Since most code written at this level these days will be generated by compilers (except possibly for the ATmega168), the most important aspect of an architecture's addressing modes is that the choices be few and clear, with costs (in terms of execution time and code size) that are readily computable. What that generally means is that a machine should take an extreme position: either it should offer every possible choice, or it should offer only one. Anything in between means that the compiler is faced with choices that it may not have the knowledge or sophistication to make.

Thus the cleanest architectures generally have only a very small number of addressing modes, with strict limits on their use. In practice, having immediate, direct, register, and indexed mode is generally enough for almost all applications.

Also, every register (including local variable pointer, stack pointer, and program counter) should be usable wherever a register is called for. More complicated addressing modes may reduce the number of instructions, but at the expense of introducing sequences of operations that cannot easily be parallelized with other sequential operations.

We have now completed our study of the various trade-offs possible between opcodes and addresses, and various forms of addressing. When approaching a new computer, you should examine the instructions and addressing modes not only to see which ones are available but also to understand why those choices were made and what the consequences of alternative choices would have been.

5.5 INSTRUCTION TYPES

ISA-level instructions can be approximately divided into a half-dozen groups that are relatively similar from machine to machine, even though they may differ in the details. In addition, every computer has a few unusual instructions, added for compatibility with previous models, or because the architect had a brilliant idea, or possibly because a government agency paid the manufacturer to include it. We will try to briefly cover all the common categories below, without any attempt at being exhaustive.

5.5.1 Data Movement Instructions

Copying data from one place to another is the most fundamental of all operations. By copying we mean the creating of a new object, with the identical bit pattern as the original. This use of the word "movement" is somewhat different from its normal usage in English. When we say that Marvin Mongoose has moved from New York to California, we do not mean that an identical copy of Mr. Mongoose was created in California and that the original is still in New York. When we say that the contents of memory location 2000 have been moved to some register, we always mean that an identical copy has been created there and that the original is still undisturbed in location 2000. Data movement instructions would better be called "data duplication" instructions, but the term "data movement" is already established.

There are two reasons for copying data from one location to another. One is fundamental: the assignment of values to variables. The assignment

```
A = B
```

is implemented by copying the value at memory address *B* to location *A* because the programmer has said to do this. The second reason is to stage the data for efficient access and use. As we have seen, many instructions can access variables only when they are available in registers. Since there are two possible sources for

a data item (memory or register), and there are two possible destinations for a data item (memory or register), four different kinds of copying are possible. Some computers have four instructions for the four cases. Others have one instruction for all four cases. Still others use LOAD to go from memory to a register, STORE to go from a register to memory, MOVE to go from one register to another register, and no instruction for a memory-to-memory copy.

Data movement instructions must indicate the amount of data to be moved. Instructions exist for some ISAs to move variable quantities of data ranging from 1 bit to the entire memory. On fixed-word-length machines, the amount to be moved is often exactly one word. Any more or less must be performed by a software routine using shifting and merging. Some ISAs provide additional capability both for copying less than a word (usually in increments of bytes) and for copying multiple words. Copying multiple words is tricky, particularly if the maximum number of words is large, because such an operation can take a long time, and may have to be interrupted in the middle. Some variable-word-length machines have instructions specifying only the source and destination addresses but not the amount. The move continues until an end-of-data field mark is found in the data.

5.5.2 Dyadic Operations

Dyadic operations combine two operands to produce a result. All ISAs have instructions to perform addition and subtraction on integers. Multiplication and division of integers are nearly standard as well. It is presumably unnecessary to explain why computers are equipped with arithmetic instructions.

Another group of dyadic operations includes the Boolean instructions. Although 16 Boolean functions of two variables exist, few, if any, machines have instructions for all 16. Usually, AND, OR, and NOT are present; sometimes EXCLUSIVE OR, NOR, and NAND are there as well.

An important use of AND is for extracting bits from words. Consider, for example, a 32-bit-word-length machine in which four 8-bit characters are stored per word. Suppose that it is necessary to separate the second character from the other three in order to print it; that is, it is necessary to create a word which contains that character in the rightmost 8 bits, referred to as **right justified**, with zeros in the leftmost 24 bits.

To extract the character, the word containing the character is ANDed with a constant, called a **mask**. The result of this operation is that the unwanted bits are all changed into zeros—that is, masked out—as shown below.

```
10110111 10111100 11011011 10001011  A
00000000 11111111 00000000 00000000  B (mask)
00000000 10111100 00000000 00000000  A AND B
```

The result would then be shifted 16 bits to the right to isolate the character to be extracted at the right end of the word.

An important use of OR is to pack bits into a word, packing being the inverse of extracting. To change the rightmost 8 bits of a 32-bit word without disturbing the other 24 bits, first the unwanted 8 bits are masked out and then the new character is ORed in, as shown below.

```
10110111 10111100 11011011 10001011  A
11111111 11111111 11111111 00000000  B (mask)
10110111 10111100 11011011 00000000  A AND B
00000000 00000000 00000000 01010111  C
10110111 10111100 11011011 01010111  (A AND B) OR C
```

The AND operation tends to remove 1s, because there are never more 1s in the result than in either of the operands. The OR operation tends to insert 1s, because there are always at least as many 1s in the result as in the operand with the most 1s. EXCLUSIVE OR, on the other hand, is symmetric, tending, on the average, neither to insert nor remove 1s. This symmetry with respect to 1s and 0s is occasionally useful, for example, in generating random numbers.

Most computers today also support a set of floating-point instructions, roughly corresponding to the integer arithmetic operations. Most machines provide at least two lengths of floating-point numbers, the shorter ones for speed and the longer ones for occasions when many digits of accuracy are needed. While there are lots of possible variations for floating-point formats, a single standard has now been almost universally adopted: IEEE 754. Floating-point numbers and IEEE 754 are discussed in Appendix B.

5.5.3 Monadic Operations

Monadic operations have one operand and produce one result. Because one fewer address has to be specified than with dyadic operations, the instructions are sometimes shorter, though often other information must be specified.

Instructions to shift or rotate the contents of a word or byte are quite useful and are often provided in several variations. Shifts are operations in which the bits are moved to the left or right, with bits shifted off the end of the word being lost. Rotates are shifts in which bits pushed off one end reappear on the other end. The difference between a shift and a rotate is illustrated below.

```
00000000 00000000 00000000 01110011  A
00000000 00000000 00000000 00011100  A shifted right 2 bits
11000000 00000000 00000000 00011100  A rotated right 2 bits
```

Both left and right shifts and rotates are useful. If an n-bit word is left rotated k bits, the result is the same as if it had been right rotated $n - k$ bits.

Right shifts are often performed with sign extension. This means that positions vacated on the left end of the word are filled up with the original sign bit, 0 or 1. It is as though the sign bit were dragged along to the right. Among other things,

it means that a negative number will remain negative. This situation is illustrated below for 2-bit right shifts.

```
11111111 11111111 11111111 11110000  A
00111111 11111111 11111111 11111100  A shifted without sign extension
11111111 11111111 11111111 11111100  A shifted with sign extension
```

An important use of shifting is multiplication and division by powers of 2. If a positive integer is left shifted k bits, the result, barring overflow, is the original number multiplied by 2^k. If a positive integer is right shifted k bits, the result is the original number divided by 2^k.

Shifting can be used to speed up certain arithmetic operations. Consider, for example, computing $18 \times n$ for some positive integer n. Because $18 \times n = 16 \times n + 2 \times n$, $16 \times n$ can be obtained by shifting a copy of n 4 bits to the left. $2 \times n$ can be obtained by shifting n 1 bit to the left. The sum of these two numbers is $18 \times n$. The multiplication has been accomplished by a move, two shifts, and an addition, which is often faster than a multiplication. Of course, the compiler can use this trick only when one factor is a constant.

Shifting negative numbers, even with sign extension, gives quite different results, however. Consider, for example, the ones' complement number, −1. Shifted 1 bit to the left it yields −3. Another 1-bit shift to the left yields −7:

```
11111111 11111111 11111111 11111110  −1 in ones' complement
11111111 11111111 11111111 11111100  −1 shifted left 1 bit = −3
11111111 11111111 11111111 11111000  −1 shifted left 2 bits = −7
```

Left shifting ones' complement negative numbers does not multiply by 2. Right shifting does simulate division correctly, however.

Now consider a two's complement representation of −1. When right shifted 6 bits with sign extension, it yields −1, which is incorrect because the integral part of −1/64 is 0:

```
11111111 11111111 11111111 11111111  −1 in two's complement
11111111 11111111 11111111 11111111  −1 shifted right 6 bits = −1
```

In general, right shifting introduces errors because it truncates down (toward the more negative integer), which is incorrect for integer arithmetic on negative numbers. Left shifting does, however, simulate multiplication by 2.

Rotate operations are useful for packing and unpacking bit sequences from words. If it is desired to test all the bits in a word, rotating the word 1 bit at a time either way successively puts each bit in the sign bit, where it can be easily tested, and also restores the word to its original value when all bits have been tested. Rotate operations are more pure than shift operations because no information is lost: an arbitrary rotate operation can be negated with another rotate operation.

Certain dyadic operations occur so frequently with particular operands that ISAs sometimes have monadic instructions to accomplish them quickly. Moving

zero to a memory word or register is extremely common when initializing a calculation. Moving zero is, of course, a special case of the general move data instructions. For efficiency, a CLR operation, with only one address, the location to be cleared (i.e., set to zero), is often provided.

Adding 1 to a word is also commonly used for counting. A monadic form of the ADD instruction is the INC operation, which adds 1. The NEG operation is another example. Negating X is really computing $0 - X$, a dyadic subtraction, but again, because of its frequent use, a separate NEG instruction is sometimes provided. It is important to note here the difference between the arithmetic operation NEG and the logical operation NOT. The NEG operation produces the **additive inverse** of a number (the number which when added to the original gives 0). The NOT operation simply inverts all the individual bits in the word. The operations are very similar, and in fact, for a system using ones' complement representation, they are identical. (In twos' complement arithmetic, the NEG instruction is carried out by first inverting all the individual bits, then adding 1.)

Dyadic and monadic instructions are often grouped together by their use, rather than by the number of operands they require. One group includes the arithmetic operations, including negation. The other group includes logical operations and shifting, since these two categories are most often used together to accomplish data extraction.

5.5.4 Comparisons and Conditional Branches

Nearly all programs need the ability to test their data and alter the sequence of instructions to be executed based on the results. A simple example is the square-root function, $\sqrt{x}$. If x is negative, the procedure gives an error message; otherwise it computes the square root. A function *sqrt* has to test x and then branch, depending on whether it is negative or not.

A common method for doing so is to provide conditional branch instructions that test some condition and branch to a particular memory address if the condition is met. Sometimes a bit in the instruction indicates whether the branch is to occur if the condition is met or not met, respectively. Often the target address is not absolute, but relative to the current instruction.

The most common condition to be tested is whether a particular bit in the machine is 0 or not. If an instruction tests the sign bit of a number and branches to *LABEL* if it is 1, the statements beginning at *LABEL* will be executed if the number was negative, and the statements following the conditional branch will be executed if it was 0 or positive.

Many machines have condition code bits that are used to indicate specific conditions. For example, there may be an overflow bit that is set to 1 whenever an arithmetic operation gives an incorrect result. By testing this bit, one checks for overflow on the previous arithmetic operation, so that if an overflow occurred, a branch can be made to an error routine and corrective action taken.

Similarly, some processors have a carry bit that is set when a carry spills over from the leftmost bit, for example, if two negative numbers are added. A carry from the leftmost bit is quite normal and should not be confused with an overflow. Testing of the carry bit is needed for multiple-precision arithmetic (i.e., where an integer is represented in two or more words).

Testing for zero is important for loops and many other purposes. If all the conditional branch instructions tested only 1 bit, then to test a particular word for 0, one would need a separate test for each bit, to ensure that none was a 1. To avoid this situation, many machines have an instruction to test a word and branch if it is zero. Of course, this solution merely passes the buck to the microarchitecture. In practice, the hardware usually contains a register all of whose bits are ORed together to give a single bit telling whether the register contains any 1 bits. The *Z* bit in Fig. 4-1 would normally be computed by ORing all the ALU output bits together and then inverting the result.

Comparing two words or characters to see if they are equal or, if not, which one is greater is also important, in sorting for example. To perform this test, three addresses are needed: two for the data items, and one for the address to branch to if the condition is true. Computers whose instruction format allows three addresses per instruction have no trouble, but those that do not must do something to get around this problem.

One common solution is to provide an instruction that performs a comparison and sets one or more condition bits to record the result. A subsequent instruction can test the condition bits and branch if the two compared values were equal, or unequal, or if the first was greater, and so on. The Core i7, OMAP4430 ARM CPU, and ATmega168 AVR all use this approach.

Some subtle points are involved in comparing two numbers. For example, comparison is not quite as simple as subtraction. If a very large positive number is compared to a very large negative number, the subtraction will result in overflow, since the result of the subtraction cannot be represented. Nevertheless, the comparison instruction must determine whether the specified test is met and return the correct answer—there is no overflow on comparisons.

Another subtle point relating to comparing numbers is deciding whether or not the numbers should be considered signed or not. Three-bit binary numbers can be ordered in one of two ways. From smallest to largest:

Unsigned	Signed	
000	100	(smallest)
001	101	
010	110	
011	111	
100	000	
101	001	
110	010	
111	011	(largest)

The column on the left shows the positive integers 0 to 7 in increasing order. The column on the right shows the two's complement signed integers −4 to +3. The answer to the question "Is 011 greater than 100?" depends on whether or not the numbers are regarded as being signed. Most ISAs have instructions to handle both orderings.

5.5.5 Procedure Call Instructions

A **procedure** is a group of instructions that performs some task and that can be invoked (called) from several places in the program. The term **subroutine** is often used instead of procedure, especially when referring to assembly-language programs. In C, procedures are called **functions**, even though they are not necessarily functions in the mathematical sense. In Java, the term used is **method**. When the procedure has finished its task, it must return to the statement after the call. Therefore, the return address must be transmitted to the procedure or saved somewhere so that it can be located when it is time to return.

The return address may be placed in any of three places: memory, a register, or the stack. Far and away the worst solution is putting it in a single, fixed memory location. In this scheme, if the procedure called another procedure, the second call would cause the return address from the first one to be lost.

A slight improvement is having the procedure-call instruction store the return address in the first word of the procedure, with the first executable instruction being in the second word. The procedure can then return by branching indirectly to the first word or, if the hardware puts the opcode for branch in the first word along with the return address, branching directly to it. The procedure may call other procedures, because each procedure has space for one return address. If the procedure calls itself, this scheme fails, because the first return address will be destroyed by the second call. The ability for a procedure to call itself, called **recursion**, is exceedingly important for both theorists and practical programmers. Furthermore, if procedure *A* calls procedure *B*, and procedure *B* calls procedure *C*, and procedure *C* calls procedure *A* (indirect or daisy-chain recursion), this scheme also fails. This scheme for storing the return address in the first word of a procedure was used on the CDC 6600, the fastest computer in the world during much of the 1960s. The main language used on the 6600 was FORTRAN, which forbade recursion, so it worked then. But it was, and still is, a terrible idea.

A bigger improvement is to have the procedure-call instruction put the return address in a register, leaving the responsibility for storing it in a safe place to the procedure. If the procedure is recursive, it will have to put the return address in a different place each time it is called.

The best thing for the procedure-call instruction to do with the return address is to push it onto a stack. When the procedure has finished, it pops the return address off the stack and stuffs it into the program counter. If this form of procedure call is available, recursion does not cause any special problems; the return address will

automatically be saved in such a way as to avoid destroying previous return addresses. Recursion works just fine under these conditions. We saw this form of saving the return address in IJVM in Fig. 4-12.

5.5.6 Loop Control

The need to execute a group of instructions a fixed number of times occurs frequently and thus some machines have instructions to facilitate doing this. All the schemes involve a counter that is increased or decreased by some constant once each time through the loop. The counter is also tested once each time through the loop. If a certain condition holds, the loop is terminated.

One method initializes a counter outside the loop and then immediately begins executing the loop code. The last instruction of the loop updates the counter and, if the termination condition has not yet been satisfied, branches back to the first instruction of the loop. Otherwise, the loop is finished and it falls through, executing the first instruction beyond the loop. This form of looping is characterized as test-at-the-end (or post-test) type looping, and is illustrated in C in Fig. 5-29(a). (We could not use Java here because it does not have a goto statement.)

```
        i = 1;
L1:     first-statement;
        .
        .
        .
        last-statement;
        i = i + 1;
        if (i < n) goto L1;
```

(a)

```
        i = 1;
L1:     if (i > n) goto L2;
        first-statement;
        .
        .
        .
        last-statement
        i = i + 1;
        goto L1;
L2:
```

(b)

Figure 5-29. (a) Test-at-the-end loop. (b) Test-at-the-beginning loop.

Test-at-the-end looping has the property that the loop will always be executed at least once, even if n is less than or equal to 0. Consider, as an example, a program that maintains personnel records for a company. At a certain point in the program, it is reading information about a particular employee. It reads in n, the number of children the employee has, and executes a loop n times, once per child, reading the child's name, sex, and birthday, so that the company can send him or her a birthday present, one of the company's fringe benefits. If the employee does not have any children, n will be 0 but the loop will still be executed once sending presents and giving erroneous results.

Figure 5-29(b) shows another way of performing the test (pretest) that works properly even for n less than or equal to 0. Notice that the testing is different in the two cases, so that if a single ISA instruction does both the increment and the test, the designers are forced to choose one method or the other.

Consider the code that should be produced for the statement

```
for (i = 0; i < n; i++) { statements }
```

If the compiler does not have any information about n, it must use the approach of Fig. 5-29(b) to correctly handle the case of $n \leq 0$. If, however, it can determine that $n > 0$, for example, by seeing where n is assigned, it may use the better code of Fig. 5-29(a). The FORTRAN standard formerly stated that all loops were to be executed once, to allow the more efficient code of Fig. 5-29(a) to be generated all the time. In 1977, that defect was corrected when even the FORTRAN community began to realize that having a loop statement with outlandish semantics that sometimes gave the wrong answer was not a good idea, even if it did save one branch instruction per loop. C and Java have always done it right.

5.5.7 Input/Output

No other group of instructions exhibits as much variety from machine to machine as the I/O instructions. Three different I/O schemes are in current use in personal computers. These are

1. Programmed I/O with busy waiting.
2. Interrupt-driven I/O.
3. DMA I/O.

We now discuss each of these in turn.

The simplest possible I/O method is **programmed I/O**, which is commonly used in low-end microprocessors, for example, in embedded systems or in systems that must respond quickly to external changes (real-time systems). These CPUs usually have a single input instruction and a single output instruction. Each of these instructions selects one of the I/O devices. A single character is transferred between a fixed register in the processor and the selected I/O device. The processor must execute an explicit sequence of instructions for each and every character read or written.

As a simple example of this method, consider a terminal with four 1-byte registers, as shown in Fig. 5-30. Two registers are used for input, status and data, and two are used for output, also status and data. Each one has a unique address. If memory-mapped I/O is being used, all four registers are part of the computer's memory address space and can be read and written using ordinary instructions. Otherwise, special I/O instructions, say, IN and OUT, are provided to read and write them. In both cases, I/O is performed by transferring data and status information between the CPU and these registers.

The keyboard status register has 2 bits that are used and 6 that are not. The leftmost bit (7) is set to 1 by the hardware whenever a new character arrives. If the software has previously set bit 6, an interrupt is generated, otherwise it is not

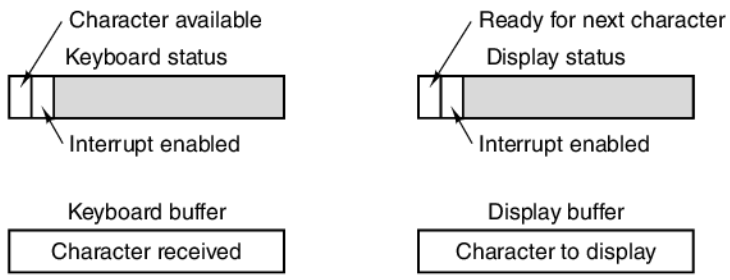


Figure 5-30. Device registers for a simple terminal.

(interrupts will be discussed shortly). When using programmed I/O, to get input, the CPU normally sits in a tight loop, repeatedly reading the keyboard status register, waiting for bit 7 to go on. When this happens, the software reads in the keyboard buffer register to get the character. Reading the keyboard data register causes the CHARACTER AVAILABLE bit to be reset to 0.

Output works in a similar way. To write a character to the screen, the software first reads the display status register to see if the READY bit is 1. If not, it loops until the bit goes to 1, indicating that the device is ready to accept a character. As soon as the terminal is ready, the software writes a character to the display buffer register, which causes it to be transmitted to the screen, and also causes the device to clear the READY bit in the display status register. When the character has been displayed and the terminal is prepared to handle the next character, the READY bit is automatically set to 1 again by the controller.

As an example of programmed I/O, consider the Java procedure of Fig. 5-31. It is called with two parameters: a character array to be output, and the count of characters present in the array, up to 1K. The body of the procedure is a loop that outputs characters one at a time. For each character, first the CPU must wait until the device is ready, then the character is output. The procedures *in* and *out* would typically be assembly-language routines to read and write the device registers specified by the first parameter from or to the variable specified as the second parameter. The implicit division by 128 by shifting gets rid of the low-order 7 bits, leaving the READY bit in bit 0.

The primary disadvantage of programmed I/O is that the CPU spends most of its time in a tight loop waiting for the device to become ready. This approach is called **busy waiting**. If the CPU has nothing else to do (e.g., the CPU in a washing machine), busy waiting may be OK (though even a simple controller often needs to monitor multiple, concurrent events). However, if there is other work to do, such as running other programs, busy waiting is wasteful, so a different I/O method is needed.

The way to get rid of busy waiting is to have the CPU start the I/O device and tell it to generate an interrupt when it is done. Looking at Fig. 5-30, we show how

```
public static void output_buffer(char buf[ ], int count) {
   // Output a block of data to the device
   int status, i, ready;

   for (i = 0; i < count; i++) {
      do {
         status = in(display_status_reg);        // get status
         ready = (status >> 7) & 0x01;           // isolate ready bit
      } while (ready != 1);
      out(display_buffer_reg, buf[i]);
   }
}
```

Figure 5-31. An example of programmed I/O.

this is done. By setting the INTERRUPT ENABLE bit in a device register, the software can request that the hardware give it a signal when the I/O is completed. We will study interrupts in detail later in this chapter when we come to flow of control.

It is worth mentioning that in many computers, the interrupt signal is generated by ANDing the INTERRUPT ENABLE bit with the READY bit. If the software first enables interrupts (before starting I/O), an interrupt will happen immediately, because the READY bit will be 1. Thus it may be necessary to first start the device, then immediately afterward enable interrupts. Writing a byte to the status register does not change the READY bit, which is read only.

Although interrupt-driven I/O is a big step forward compared to programmed I/O, it is far from perfect. The problem is that an interrupt is required for every character transmitted. Processing an interrupt is expensive. A way is needed to get rid of most of the interrupts.

The solution lies in going back to programmed I/O, but having somebody else do it. (The solution to many problems lies in having somebody else do the work.) Figure 5-32 shows how this is arranged. Here we have added a new chip, a **DMA** (**Direct Memory Access**) controller to the system, with direct access to the bus.

The DMA chip has (at least) four registers inside it, all of which can be loaded by software running on the CPU. The first contains the memory address to be read or written. The second contains the count of how many bytes (or words) are to be transferred. The third specifies the device number or I/O space address to use, thus specifying which I/O device is desired. The fourth tells whether data are to be read from or written to the I/O device.

To write a block of 32 bytes from memory address 100 to a terminal (say, device 4), the CPU writes the numbers 32, 100, and 4 into the first three DMA registers, and then the code for WRITE (say, 1) in the fourth one, as illustrated in Fig. 5-32. Once initialized like this, the DMA controller makes a bus request to read byte 100 from the memory, the same way the CPU would read from the memory. Having gotten this byte, the DMA controller then makes an I/O request to device 4 to write the byte to it. After both of these operations have been completed,

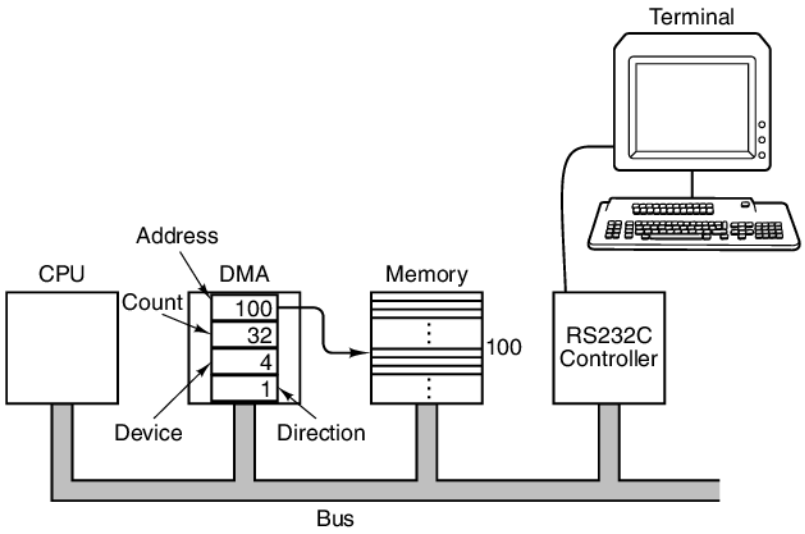


Figure 5-32. A system with a DMA controller.

the DMA controller increments its address register by 1 and decrements its count register by 1. If the count register is still greater than 0, another byte is read from memory and then written to the device.

When the count finally goes to 0, the DMA controller stops transferring data and asserts the interrupt line on the CPU chip. With DMA, the CPU only has to initialize a few registers. After that, it is free to do something else until the complete transfer is finished, at which time it gets an interrupt from the DMA controller. Some DMA controllers have two, or three, or more sets of registers, so they can control multiple simultaneous transfers.

While DMA greatly relieves the CPU from the burden of I/O, the process is not totally free. If a high-speed device, such as a disk, is being run by DMA, many bus cycles will be needed, both for memory references and device references. During these cycles the CPU will have to wait (DMA always has a higher bus priority than the CPU because I/O devices frequently cannot tolerate delays). The process of having a DMA controller take bus cycles away from the CPU is called **cycle stealing**. Nevertheless, the gain in not having to handle one interrupt per byte (or word) transferred far outweighs the loss due to cycle stealing.

5.5.8 The Core i7 Instructions

In this section and the next two, we will look at the instruction sets of our three example machines: the Core i7, the OMAP4430 ARM CPU, and the ATmega168 AVR. Each has a core of instructions that compilers normally generate, plus a set

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOVcc DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract SRC from DST
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract SRC & carry from DST
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TEST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT n	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

Figure 5-33. A selection of the Core i7 integer instructions.

of instructions that are rarely used, or are used only by the operating system. In our discussion, we will focus on the common instructions. Let us start with the Core i7. It is the most complicated. Then it is all downhill from there.

The Core i7 instruction set is a mixture of instructions that make sense in 32-bit mode and those that hark back to its former life as an 8088. In Fig. 5-33 we show a small selection of the more common integer instructions that compilers and programmers are likely to use these days. This list is far from complete, as it does not include any floating-point instructions, control instructions, or even some of the more exotic integer instructions (such as using an 8-bit byte in AL to perform table lookup). Nevertheless, it does give a good feel for what the Core i7 can do.

Many of the Core i7 instructions reference one or two operands, either in registers or in memory. For example, the two-operand ADD instruction adds the source to the destination and the one operand INC instruction increments (adds 1 to) its operand. Some of the instructions have several closely related variants. For example, the shift instructions can shift either left or right and can treat the sign bit specially or not. Most of the instructions have a variety of different encodings, depending on the nature of the operands.

In Fig. 5-33, the SRC fields are sources of information and are not changed. In contrast, the DST fields are destinations and are normally modified by the instruction. There are some rules about what is allowed as a source or a destination, varying somewhat erratically from instruction to instruction, but we will not go into them here. Many instructions have three variants, for 8-, 16-, and 32-bit operands, respectively. These are distinguished by different opcodes and/or a bit in the instruction. The list of Fig. 5-33 emphasizes the 32-bit instructions.

For convenience, we have divided the instructions into several groups. The first group contains instructions that move data around the machine, among registers, memory, and the stack. The second group does arithmetic, both signed and unsigned. For multiplication and division, the 64-bit product or dividend is stored in EAX (low-order part) and EDX (high-order part).

The third group does Binary Coded Decimal (BCD) arithmetic, treating each byte as two 4-bit **nibbles**. Each nibble holds one decimal digit (0 to 9). Bit combinations 1010 to 1111 are not used. Thus a 16-bit integer can hold a decimal number from 0 to 9999. While this form of storage is inefficient, it eliminates the need to convert decimal input to binary and then back to decimal for output. These instructions are used for doing arithmetic on the BCD numbers. They are heavily used by COBOL programs.

The Boolean and shift/rotate instructions manipulate the bits in a word or byte in various ways. Several combinations are provided.

The next two groups deal with testing and comparing, and then jumping based on the results. The results of test and compare instructions are stored in various bits of the EFLAGS register. Jxx stands for a set of instructions that conditionally jump, depending on the results of the previous comparison (i.e., bits in EFLAGS).

The Core i7 has several instructions for loading, storing, moving, comparing, and scanning strings of characters or words. These instructions can be prefixed by a special byte called REP, which cause them to be repeated until a certain condition is met, such as ECX, which is decremented after each iteration, reaching 0. In this

way, arbitrary blocks of data can be moved, compared, and so on. The next group manages the condition codes.

The last group is a hodge-podge of instructions that do not fit in anywhere else. These include conversions, stack frame management, stopping the CPU, and I/O.

The Core i7 has a number of **prefixes**, of which we have already mentioned one (REP). Each of these prefixes is a special byte that can precede most instructions, analogous to WIDE in IJVM. REP causes the instruction following it to be repeated until ECX hits 0, as mentioned above. REPZ and REPNZ repeatedly execute the following instruction until the *Z* condition code is set, or not set, respectively. LOCK reserves the bus for the entire instruction, to permit multiprocessor synchronization. Other prefixes are used to force an instruction to operate in 16-bit mode, or in 32-bit mode, which not only changes the length of the operands but also completely redefines the addressing modes. Finally, the Core i7 has a complex segmentation scheme with code, data, stack, and extra segments, a holdover from the 8088. Prefixes are provided to force memory references to use specific segments, but these will not be of concern to us (fortunately).

5.5.9 The OMAP4430 ARM CPU Instructions

Nearly all of the user-mode integer ARM instructions that a compiler might generate are listed in Fig. 5-34. Floating-point instructions are not given here, nor are control instructions (e.g., cache management, system reset), instructions involving address spaces other than the user's, or instruction extensions such as Thumb. The set is surprisingly small: the OMAP4430 ARM ISA really is a reduced instruction set computer.

The LDR and STR instructions are straightforward, with versions for 1, 2, and 4 bytes. When a number less than 32 bits is loaded into a (32-bit) register, the number can be either sign extended or zero extended. Instructions for both exist.

The next group is for arithmetic, which can optionally set the processor status register's condition code bits. On CISC machines, most instructions set the condition codes, but on a RISC machine that is undesirable because it restricts the compiler's freedom to move instructions around when trying to schedule them to tolerate instruction delays. If the original instruction order is A ... B ... C with A setting the condition codes and B testing them, the compiler cannot insert C between A and B if C sets the condition codes. For this reason, two versions of many instructions are provided, with the compiler normally using the one that does not set the condition codes, unless it is planning to test them later. The programmer specifies the setting of the condition codes by adding an "S" to the end of the instruction opcode name, for example, ADDS. A bit in the instruction indicates to the processor that the condition codes should be set. Multiplication and multiply-accumulate are also supported.

The shift group contains one left shift and two right shifts, each of which operate on 32-bit registers. The rotate right instruction does a circular rotation of bits

Loads

LDRSB DST,ADDR	Load signed byte (8 bits)
LDRB DST,ADDR	Load unsigned byte (8 bits)
LDRSH DST,ADDR	Load signed halfwords (16 bits)
LDRH DST,ADDR	Load unsigned halfwords (16 bits)
LDR DST,ADDR	Load word (32 bits)
LDM S1,REGLIST	Load multiple words

Stores

STRB DST,ADDR	Store byte (8 bits)
STRH DST,ADDR	Store halfword (16 bits)
STR DST,ADDR	Store word (32 bits)
STM SRC,REGLIST	Store multiple words

Arithmetic

ADD DST,S1,S2IMM	Add
ADD DST,S1,S2IMM	Add with carry
SUB DST,S1,S2IMM	Subtract
SUB DST,S1,S2IMM	Subtract with carry
RSB DST,S1,S2IMM	Reverse subtract
RSC DST,S1,S2IMM	Reverse subtract with carry
MUL DST,S1,S2	Multiply
MLA DST,S1,S2,S3	Multiple and accumulate
UMULL D1,D2,S1,S2	Unsigned long multiple
SMULL D1,D2,S1,S2	Signed long multiple
UMLAL D1,D2,S1,S2	Unsigned long MLA
SMLAL D1,D2,S1,S2	Signed long MLA
CMP S1,S2IMM	Compare and set PSR

Shifts/rotates

LSL DST,S1,S2IMM	Logical shift left
LSR DST,S1,S2IMM	Logical shift right
ASR DST,S1,S2IMM	Arithmetic shift right
ROR DSR,S1,S2IMM	Rotate right

Boolean

TST DST,S1,S2IMM	Test bits
TEQ DST,S1,S2IMM	Test equivalence
AND DST,S1,S2IMM	Boolean AND
EOR DST,S1,S2IMM	Boolean Exclusive-OR
ORR DST,S1,S2IMM	Boolean OR
BIC DST,S1,S2IMM	Bit clear

Transfer of control

Bcc IMM	Branch to PC+IMM
BLcc IMM	Branch with link to PC+IMM
BLcc S1	Branch with link to reg add

Miscellaneous

MOV DST,S1	Move register
MOVT DST,IMM	Move imm to upper bits
MVN DST,S1	NOT register
MRS DST,PSR	Read PSR
MSR PSR,S1	Write PSR
SWP DST,S1,ADDR	Swap reg/mem word
SWPB DST,S1,ADDR	Swap reg/mem byte
SWI IMM	Software interrupt

S1 = source register
S2IMM = source register or immediate
S3 = source register (when 3 are used)
DST = destination register
D1 = destination register (1 of 2)
D2 = destination register (2 of 2)

ADDR = memory address
IMM = immediate value
REGLIST = list of registers
PSR = processor status register
cc = branch condition

Figure 5-34. The primary OMAP4430 ARM CPU integer instructions.

within the register, such that bits that rotate off the least significant bit reappear as the most significant bit. The shifts are mostly used for bit manipulation. Rotates are useful for cryptographic and image-processing operations. Most CISC machines have a vast number of shift and rotate instructions, nearly all of them totally useless. Few compiler writers will spend restless nights mourning their absence.

The Boolean instruction group is analogous to the arithmetic one. It includes AND, EOR, ORR, TST, TEQ, and BIC. The latter three are of questionable value, but they can be done in one cycle and require almost no additional hardware so they got thrown in. Even RISC machine designers sometimes succumb to temptation.

The next instruction group contains the control transfers. *Bcc* represents a set of instructions that branch on the various conditions. *BLcc* is similar in that it

branches on various conditions, but it also deposits the address of the next instruction in the link register (R14). This instruction is useful to implement procedure calls. Unlike all other RISC architectures, there is no explicit branch to register address instruction. This instruction can be easily synthesized by using a MOV instruction with the destination set to the program counter (R15).

Two ways are provided for calling procedures. The first *BLcc* instruction uses the "Branch" format of Fig. 5-14 with a 24-bit PC-relative *word* offset. This value is enough to reach any instruction within 32 megabytes of the called in either direction. The second *BLcc* instruction jumps to the address in the specified register. This can be used to implement dynamically bound procedure calls (e.g., C++ virtual functions) or calls beyond the reach of 32 megabytes.

The last group contains some leftovers. MOVT is needed because there is no way to get a 32-bit immediate operand into a register. The way it is done is to use MOVT to set bits 16 through 31 and then have the next instruction supply the remaining bits using the immediate format. The MRS and MSR instructions allow reading and writing of the processor status word (PSR). The SWP instructions perform atomic swaps between a register and a memory location. These instruction implement the multiprocessor synchronization primitives that we will learn about in Chap. 8. Finally, the SWI instruction initiates a software interrupt, which is an overly fancy way of saying that it initiates a system call.

5.5.10 The ATmega168 AVR Instructions

The ATmega168 has a simple instruction set, shown in Fig. 5-35. Each line gives the mnemonic, a brief description, and a snippet of pseudocode that details the operation of the instruction. As is to be expected, there are a variety of MOV instructions for moving data between the registers. There are instructions for pushing and popping from a stack, which is pointed to by the 16-bit stack pointer (SP) in memory. Memory can be accessed with either an immediate address, register indirect, or register indirect plus a displacement. To allow up to 64 KB of addressing, the load with an immediate address is a 32-bit instruction. The indirect addressing modes utilize the X, Y, and Z register pairs, which combine two 8-bit registers to form a single 16-bit pointer.

The ATmega168 has simple arithmetic instructions for adding, subtracting, and multiplying, the latter of which use two registers. Incrementing and decrementing are also possible and commonly used. Boolean, shift, and rotate instructions are also present. The branch and call instruction can target an immediate address, a PC-relative address, or an address contained in the Z register pair.

5.5.11 Comparison of Instruction Sets

The three example instruction sets are very different. The Core i7 is a classic two-address 32-bit CISC machine, with a long history, peculiar and highly irregular addressing modes, and many instructions that reference memory.

Instruction	Description	Semantics
ADD DST,SRC	Add	DST ← DST + SRC
ADC DST,SRC	Add with Carry	DST ← DST + SRC + C
ADIW DST,IMM	Add Immediate to Word	DST+1:DST ← DST+1:DST + IMM
SUB DST,SRC	Subtract	DST ← DST - SRC
SUBI DST,IMM	Subtract Immediate	DST ← DST - IMM
SBC DST,SRC	Subtract with Carry	DST ← DST - SRC - C
SBCI DST,IMM	Subtract Immediate with Carry	DST ← DST - IMM – C
SBIW DST,IMM	Subtract Immediate from Word	DST+1:DST ← DST+1:DST - IMM
AND DST,SRC	Logical AND	DST ← DST AND SRC
ANDI DST,IMM	Logical AND with Immediate	DST ← DST AND IMM
OR DST,SRC	Logical OR	DST ← DST OR SRC
ORI DST,IMM	Logical OR with Immediate	DST ← DST OR IMM
EOR DST,SRC	Exclusive OR	DST ← DST XOR SRC
COM DST	One's Complement	DST ← 0xFF - DST
NEG DST	Two's Complement	DST ← 0x00 - DST
SBR DST,IMM	Set Bit(s) in Register	DST ← DST OR IMM
CBR DST,IMM	Clear Bit(s) in Register	DST ← DST AND (0xFF - IMM)
ING DST	Increment	DST ← DST + 1
DEC DST	Decrement	DST ← DST - 1
TST DST	Test for Zero or Minus	DST ← DST AND DST
CLR DST	Clear Register	DST ← DST XOR DST
SER DST	Set Register	DST ← 0xFF
MUL DST,SRC	Multiply Unsigned	R1:R0 ← DST * SRC
MULS DST,SRC	Multiply Signed	R1:R0 ← DST * SRC
MULSU DST,SRC	Multiply Signed with Unsigned	R1:R0 ← DST * SRC
RJMP IMM	PC-relative Jump	PC ← PC + IMM + 1
IJMP	Indirect Jump to Z	PC ← Z (R30:R31)
JMP IMM	Jump	PC ← IMM
RCALL IMM	Relative Call	STACK ← PC+2, PC ← PC + IMM + 1
ICALL	Indirect Call to (Z)	STACK ← PC+2, PC ← Z (R30:R31)
CALL	Call	STACK ← PC+2, PC ← IMM
RET	Return	PC ← STACK
CP DST,SRC	Compare	DST – SRC
CPC DST,SRC	Compare with Carry	DST – SRC – C
CPI DST,IMM	Compare with Immediate	DST – IMM
BRcc IMM	Branch on Condition	if cc(true) PC ← PC + IMM + 1
MOV DST,SRC	Copy Register	DST ← SRC
MOVW DST,SRC	Copy Register Pair	DST+1:DST ← SRC+1:SRC
LDI DST,IMM	Load Immediate	DST ← IMM
LDS DST,IMM	Load Direct	DST ← MEM[IMM]
LD DST,XYZ	Load Indirect	DST ← MEM[XYZ]
LDD DST,XYZ+IMM	Load Indirect with Displacement	DST ← MEM[XYZ+IMM]
STS IMM,SRC	Store Direct	MEM[IMM] ← SRC
ST XYZ,SRC	Store Indirect	MEM[XYZ] ← SRC
STD XYZ+IMM,SRC	Store Indirect with Displacement	MEM[XYZ+IMM] ← SRC
PUSH REGLIST	Push Register on Stack	STACK ← REGLIST
POP REGLIST	Pop Register from Stack	REGLIST ← STACK
LSL DST	Logical Shift Left by One	DST ← DST LSL 1
LSR DST	Logical Shift Right by One	DST ← DST LSR 1
ROL DST	Rotate Left by One	DST ← DST ROL 1
ROR DST	Rotate Right by One	DST ← DST ROR 1
ASR DST	Arithmetic Shift Right by One	DST ← DST ASR 1

SRS = source register
DST = destination register
IMM = immediate value

XYZ = X, Y, or Z register pair
MEM[A] = access memory at address A

Figure 5-35. The ATmega168 AVR instruction set.

The OMAP4430 ARM CPU is a modern three-address 32-bit RISC, with a load/store architecture, hardly any addressing modes, and a compact and efficient instruction set. The ATmega168 AVR architecture is a tiny embedded processor designed to fit on a single chip.

Each machine is the way it is for a good reason. The Core i7's design was determined by three major factors:

1. Backward compatibility.
2. Backward compatibility.
3. Backward compatibility.

Given the current state of the art, no one would now design such an irregular machine with so few registers, all of them different. This makes compilers hard to write. The lack of registers also forces compilers to constantly spill variables into memory and then reload them, an expensive business, even with two or three levels of caching. It is a testimonial to the quality of Intel's engineers that the Core i7 is so fast, even with the constraints of this ISA. But as we saw in Chap. 4, the implementation is exceedingly complex.

The OMAP4430 ARM represents a state-of-the-art ISA design. It has a full 32-bit ISA. It has many registers, an instruction set that emphasizes three-register operations, plus a small group of LOAD and STORE instructions. All instructions are the same size, although the number of formats has gotten a bit out of hand. Still, it lends itself to a straightforward and efficient implementation. Most new designs tend to look like the OMAP4430 ARM architecture, but with fewer instruction formats.

The ATmega168 AVR has a simple and fairly regular instruction set with relatively few instructions and few addressing modes. It is distinguished by having 32 8-bit registers, rapid access to data, a way to access registers in the memory space, and surprisingly powerful bit-manipulation instructions. Its main claim to fame is that it can be implemented with a very small number of transistors, thus making it possible to put a large number on a die, which keeps the cost per CPU very low.

5.6 FLOW OF CONTROL

Flow of control refers to the sequence in which instructions are executed dynamically, that is, during program execution. In general, in the absence of branches and procedure calls, successively executed instructions are fetched from consecutive memory locations. Procedure calls cause the flow of control to be altered, stopping the procedure currently executing and starting the called procedure. Coroutines are related to procedures and cause similar alterations in the flow of control. They are useful for simulating parallel processes. Traps and interrupts

also cause the flow of control to be altered when special conditions occur. All these topics will be discussed in the following sections.

5.6.1 Sequential Flow of Control and Branches

Most instructions do not alter the flow of control. After an instruction is executed, the one following it in memory is fetched and executed. After each instruction, the program counter is increased by the instruction length. If observed over an interval of time that is long compared to the average instruction time, the program counter is approximately a linear function of time, increasing by the average instruction length per average instruction time. Stated another way, the dynamic order in which the processor actually executes the instructions is the same as the order in which they appear on the program listing, as shown in Fig. 5-36(a). If a program contains branches, this simple relation between the order in which instructions appear in memory and the order in which they are executed is no longer true. When branches are present, the program counter is no longer a monotonically increasing function of time, as shown in Fig. 5-36(b). As a result, it becomes difficult to visualize the instruction execution sequence from the program listing.

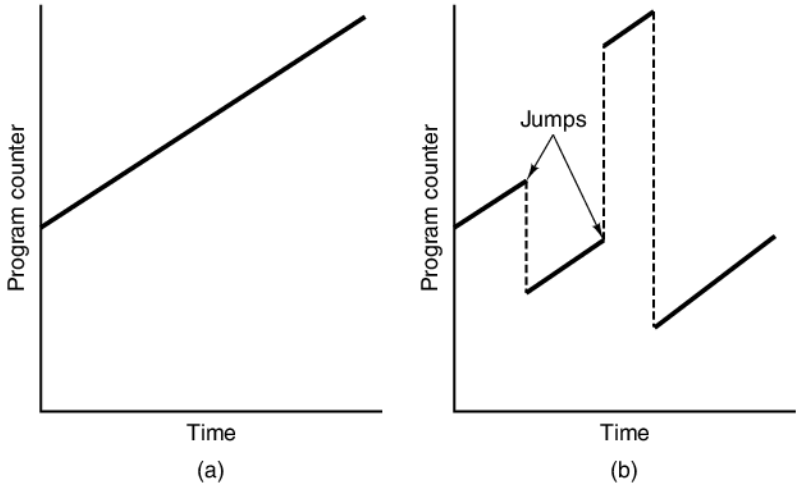


Figure 5-36. Program counter as a function of time (smoothed). (a) Without branches. (b) With branches.

When programmers have trouble keeping track of the sequence in which the processor will execute the instructions, they are prone to make errors. This observation led Dijkstra (1968a) to write a then-controversial letter entitled "GO TO Statement Considered Harmful," in which he suggested avoiding goto statements. That letter gave birth to the structured programming revolution, one of whose tenets is the replacement of goto statements with more structured forms of flow

control, such as while loops. Of course, these programs compile down to level 2 programs that may contain many branches, because the implementation of if, while, and other high-level control structures requires branching around.

5.6.2 Procedures

The most important technique for structuring programs is the procedure. From one point of view, a procedure call alters the flow of control just as a branch does, but unlike the branch, when finished performing its task, it returns control to the statement or instruction following the call.

However, from another point of view, a procedure body can be regarded as defining a new instruction on a higher level. From this standpoint, a procedure call can be thought of as a single instruction, even though the procedure may be quite complicated. To understand a piece of code containing a procedure call, it is only necessary to know *what* it does, not *how* it does it.

One particularly interesting kind of procedure is the **recursive procedure**, that is, a procedure that calls itself, either directly or indirectly via a chain of other procedures. Studying recursive procedures gives considerable insight into how procedure calls are implemented, and what local variables really are. Now we will give an example of a recursive procedure.

The "Towers of Hanoi" is an ancient problem that has a simple solution involving recursion. In a certain monastery in Hanoi, there are three gold pegs. Around the first one were a series of 64 concentric gold disks, each with a hole in the middle for the peg. Each disk is slightly smaller in diameter than the disk directly below it. The second and third pegs were initially empty. The monks there are busily transferring all the disks to peg 3, one disk at a time, but at no time may a larger disk rest on a smaller one. When they finish, it is said the world will come to an end. If you wish to get hands-on experience, it is all right to use plastic disks and fewer of them, but when you solve the problem, nothing will happen. To get the end-of-world effect, you need 64 of them and in gold. Figure 5-37 shows the initial configuration for $n = 5$ disks.

The solution of moving n disks from peg 1 to peg 3 consists first of moving $n - 1$ disks from peg 1 to peg 2, then moving 1 disk from peg 1 to peg 3, then moving $n - 1$ disks from peg 2 to peg 3. This solution is illustrated in Fig. 5-38.

To solve the problem we need a procedure to move n disks from peg i to peg j. When this procedure is called, by

```
towers(n, i, j)
```

the solution is printed out. The procedure first makes a test to see if $n = 1$. If so, the solution is trivial, just move the one disk from i to j. If $n \neq 1$, the solution consists of three parts as discussed above, each being a recursive procedure call.

The complete solution is shown in Fig. 5-39. The call

```
towers(3, 1, 3)
```

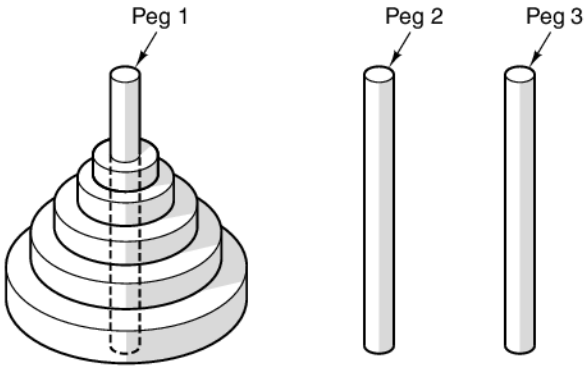


Figure 5-37. Initial configuration for the Towers of Hanoi problem for five disks.

to solve the problem of Fig. 5-38 generates three more calls. Specifically, it makes the calls

```
towers(2, 1, 2)
towers(1, 1, 3)
towers(2, 2, 3)
```

The first and third will generate three calls each, for a total of seven.

In order to have recursive procedures, we need a stack to store the parameters and local variables for each invocation, the same as we had in IJVM. Each time a procedure is called, a new stack frame is allocated for the procedure on top of the stack. The frame most recently created is the current frame. In our examples, the stack grows upward, from low memory addresses to high ones, just like in IJVM. So the most recent frame has higher addresses than all the others.

In addition to the stack pointer, which points to the top of the stack, it is often convenient to have a frame pointer, FP, which points to a fixed location within the frame. It could point to the link pointer, as in IJVM, or to the first local variable. Figure 5-39 shows the stack frame for a machine with a 32-bit word. The original call to *towers* pushes *n*, *i*, and *j* onto the stack and then executes a CALL instruction that pushes the return address onto the stack, at address 1012. On entry, the called procedure stores the old value of FP on the stack at 1016 and then advances the stack pointer to allocate storage for the local variables. With only one 32-bit local variable (*k*), SP is incremented by 4 to 1020. The situation, after all these things have been done, is shown in Fig. 5-39(a).

The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit), copy SP into FP, and possibly increment by one word, depending on where in the new frame FP points. In this example, FP points to the first local variable, but in IJVM, LV pointed to the link pointer. Different

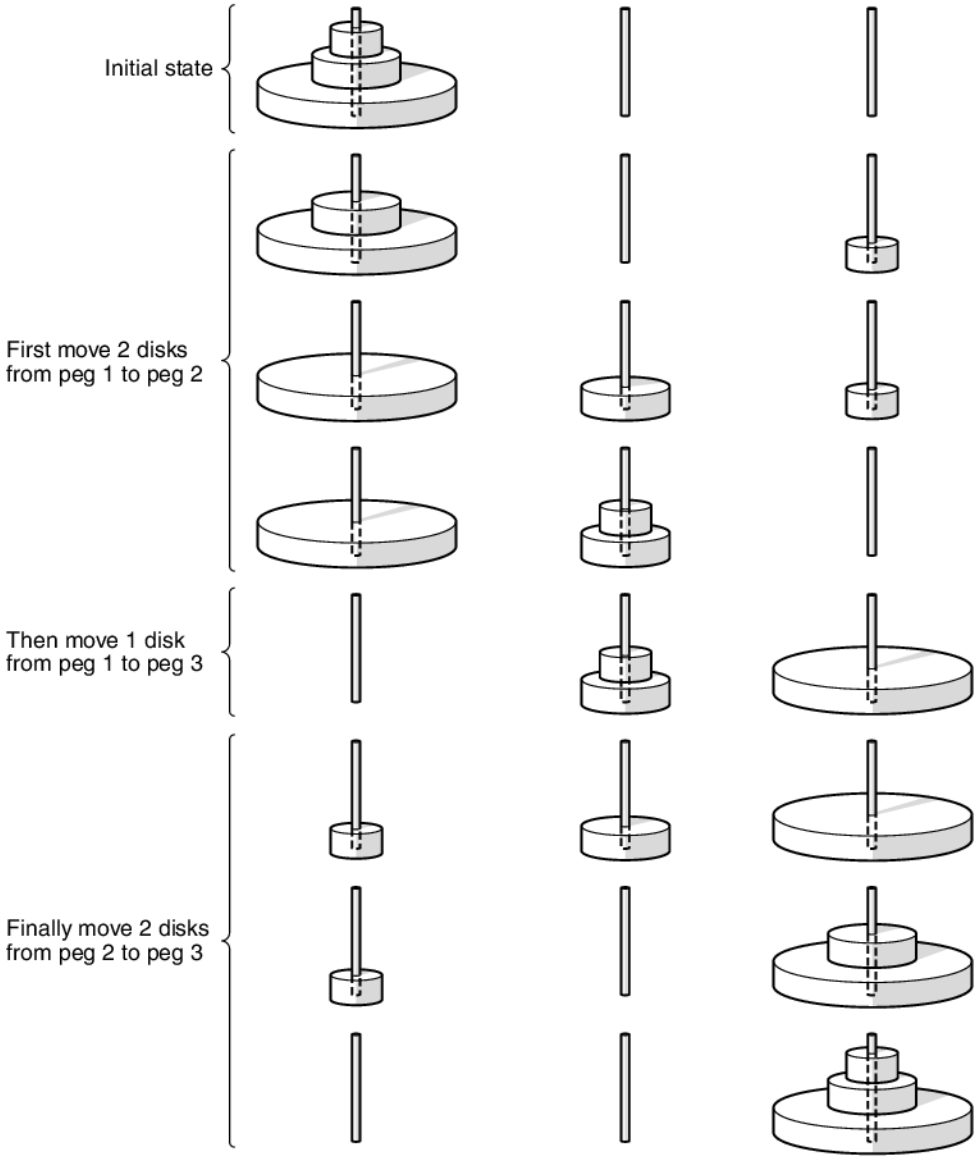


Figure 5-38. The steps required to solve the Towers of Hanoi for three disks.

machines handle the frame pointer slightly differently, sometimes putting it at the bottom of the stack frame, sometimes at the top, and sometimes in the middle as in Fig. 5-40. In this respect, it is worth comparing Fig. 5-40 with Fig. 4-12 to see two different ways to manage the link pointer. Other ways are also possible. In all cases, the key is the ability to later be able to do a procedure return and restore the

```
public void towers(int n, int i, int j) {
   int k;

   if (n == 1)
      System.out.println("Move a disk from " + i + " to " + j);
   else {
      k = 6 – i – j;
      towers(n – 1, i, k);
      towers(1, i, j);
      towers(n – 1, k, j);
 }
}
```

Figure 5-39. A procedure for solving the Towers of Hanoi.

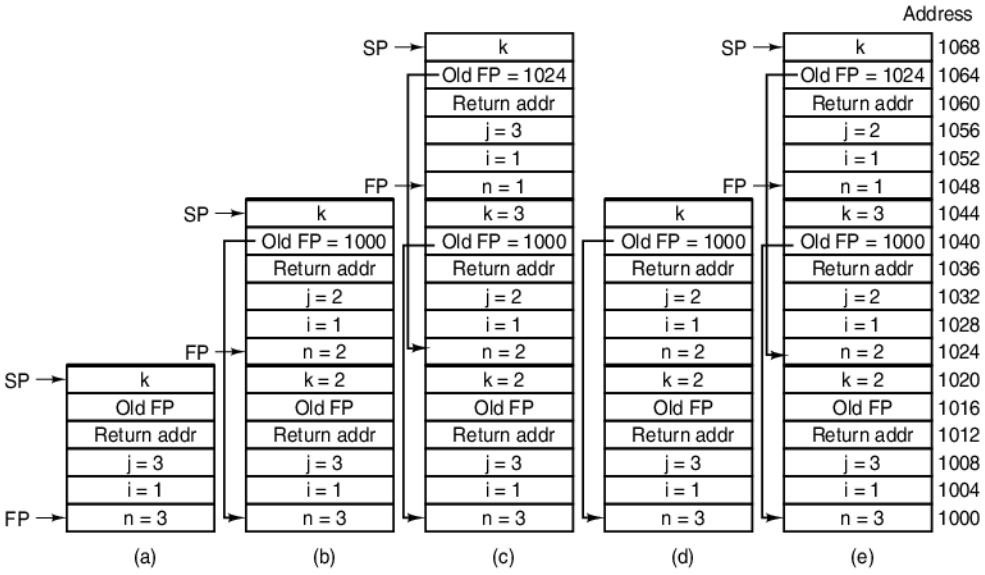


Figure 5-40. The stack at several points during the execution of Fig. 5-39.

state of the stack to what it was just prior to the current procedure invocation.

The code that saves the old frame pointer, sets up the new one, and advances the stack pointer to reserve space for local variables is called the **procedure prolog**. Upon procedure exit, the stack must be cleaned up again, something called the **procedure epilog**. One of the most important characteristics of any computer is how short and fast it can make the prolog and epilog. If they are long and slow, procedure calls will be expensive. Programmers who worship at the altar of efficiency will learn to avoid writing many short procedures and write large, monolithic, unstructured programs instead. The Core i7 ENTER and LEAVE instructions

have been designed to do most of the procedure prolog and epilog work efficiently. Of course, they have a particular model of how the frame pointer should be managed, and if the compiler has a different model, they cannot be used.

Now let us get back to the Towers of Hanoi problem. Each procedure call adds a new frame to the stack and each procedure return removes a frame from the stack. In order to illustrate the use of a stack in implementing recursive procedures, we will trace the calls starting with

```
towers(3, 1, 3)
```

Figure 5-40(a) shows the stack just after this call has been made. The procedure first tests to see if $n = 1$, and on discovering that $n = 3$, fills in k and makes the call

```
towers(2, 1, 2)
```

After this call is completed the stack is as shown in Fig. 5-40(b), and the procedure starts again at the beginning (a called procedure always starts at the beginning). This time the test for $n = 1$ fails again, so it fills in k again and makes the call

```
towers(1, 1, 3)
```

The stack then is as shown in Fig. 5-40(c) and the program counter points to the start of the procedure. This time the test succeeds and a line is printed. Next, the procedure returns by removing one stack frame, resetting FP and SP to Fig. 5-40(d). It then continues executing at the return address, which is the second call:

```
towers(1, 1, 2)
```

This adds a new frame to the stack as shown in Fig. 5-40(e). Another line is printed; after the return a frame is removed from the stack. The procedure calls continue in this way until the original call completes execution and the frame of Fig. 5-40(a) is removed from the stack. To best understand how recursion works, it is recommended that you simulate the complete execution of

```
towers(3, 1, 3)
```

using pencil and paper.

5.6.3 Coroutines

In the usual calling sequence, there is a clear distinction between the calling procedure and the called procedure. Consider a procedure *A*, on the left which calls a procedure *B* on the right in Fig. 5-41.

Procedure *B* computes for a while and then afterwards returns to *A*. At first sight you might consider this situation symmetric, because neither *A* nor *B* is a main program, both being procedures. (Procedure *A* may have been called by the

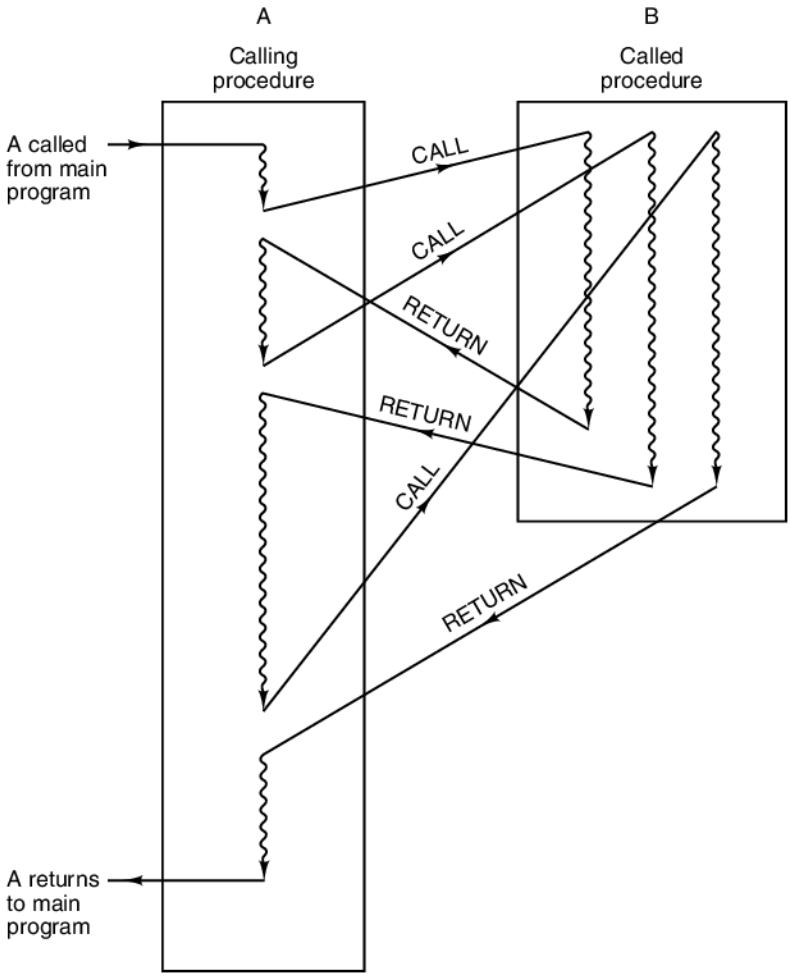


Figure 5-41. When a procedure is called, execution of the procedure always begins at the first statement of the procedure.

main program but that is irrelevant.) Furthermore, first control is transferred from *A* to *B*—the call—and later control is transferred from *B* to *A*—the return.

The asymmetry arises from the fact that when control passes from *A* to *B*, procedure *B* begins executing at the beginning; when *B* returns to *A*, execution starts not at the beginning of *A* but at the statement following the call. If *A* runs for a while and calls *B* again, execution starts at the beginning of *B* again, not at the statement following the previous return. If, in the course of running, *A* calls *B* many times, *B* starts at the beginning all over again each and every time, whereas *A* never starts over again. It just keeps going forward.

This difference is reflected in the method by which control is passed between *A* and *B*. When *A* calls *B*, it uses the procedure call instruction, which puts the return address (i.e., the address of the statement following the call) somewhere useful, for example, on top of the stack. It then puts the address of *B* into the program counter to complete the call. When *B* returns, it does not use the call instruction but instead it uses the return instruction, which simply pops the return address from the stack and puts it into the program counter.

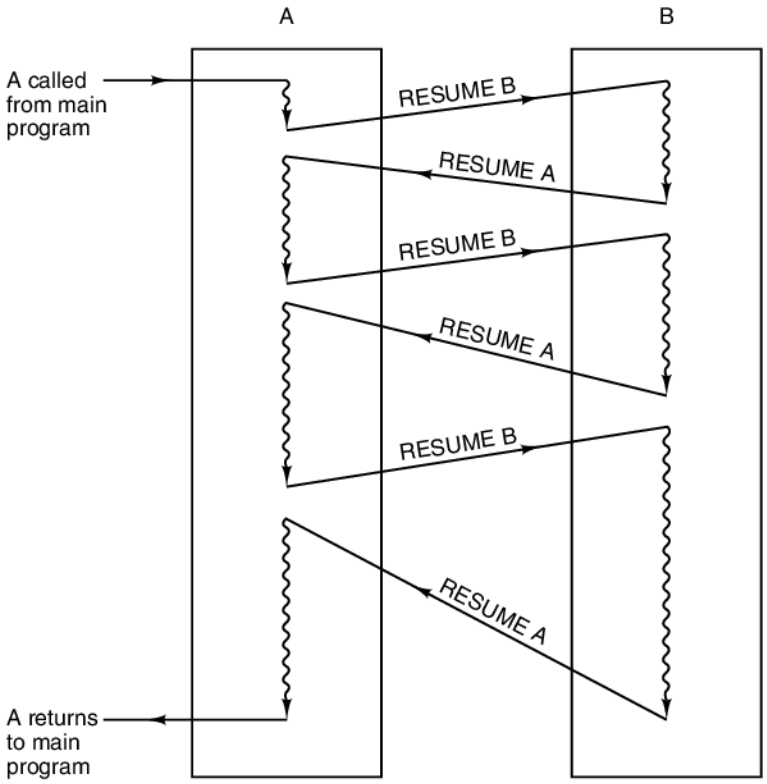


Figure 5-42. When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.

Sometimes it is useful to have two procedures, *A* and *B*, each of which calls the other as a procedure, as shown in Fig. 5-42. When *B* returns to *A*, it branches to the statement following the call to *B*, as above. When *A* transfers control to *B*, it does not go to the beginning (except the first time) but to the statement following the most recent "return," that is, the most recent call of *A*. Two procedures that work this way are called **coroutines**.

A common use for coroutines is to simulate parallel processing on a single CPU. Each coroutine runs in pseudoparallel with the others, as though it had its

own CPU. This style of programming makes programming some applications easier. It also is useful for testing software that will later actually run on a multiprocessor.

Neither the usual CALL nor the usual RETURN instruction works for calling coroutines, because the address to branch to comes from the stack like a return, but, unlike a return, the coroutine call itself puts a return address somewhere for the subsequent return to it. It would be nice if there were an instruction to exchange the top of the stack with the program counter. In detail, this instruction would first pop the old return address off the stack into an internal register, then push the program counter onto the stack, and finally, copy the internal register into the program counter. Because one word is popped off the stack and one word is pushed onto the stack, the stack pointer does not change. This instruction rarely exists, so in most cases it has to be simulated as several instructions.

5.6.4 Traps

A **trap** is a kind of automatic procedure call initiated by some condition caused by the program, usually an important but rarely occurring condition. A good example is overflow. On many computers, if the result of an arithmetic operation exceeds the largest number that can be represented, a trap occurs, meaning that the flow of control is switched to some fixed memory location instead of continuing in sequence. At that fixed location is a branch to a procedure called the **trap handler**, which performs some appropriate action, such as printing an error message. If the result of an operation is within range, no trap occurs.

The essential point about a trap is that it is initiated by some exceptional condition caused by the program itself and detected by the hardware or microprogram. An alternative method of handling overflow is to have a 1-bit register that is set to 1 whenever an overflow occurs. A programmer who wants to check for overflow must include an explicit "branch if overflow bit is set" instruction after every arithmetic instruction. Doing so is both slow and wasteful of space. Traps save both time and memory compared with explicit programmer-controlled checking.

The trap may be implemented by an explicit test performed by the microprogram (or hardware). If an overflow is detected, the trap address is loaded into the program counter. What is a trap at one level may be under program control at a lower level. Having the microprogram make the test still saves time compared to a programmer test, because it can be easily overlapped with something else. It also saves memory, because it need occur only in one place, for example, the main loop of the microprogram, independent of how many arithmetic instructions occur in the main program.

A few of the common conditions that can cause traps are floating-point overflow, floating-point underflow, integer overflow, protection violation, undefined opcode, stack overflow, attempt to start a nonexistent I/O device, attempt to fetch a word from an odd-numbered address, and division by zero.

5.6.5 Interrupts

Interrupts are changes in the flow of control caused not by the running program, but by something else, usually related to I/O. For example, a program may instruct the disk to start transferring information, and set the disk up to provide an interrupt as soon as the transfer is finished. Like the trap, the interrupt stops the running program and transfers control to an interrupt handler, which performs some appropriate action. When finished, the interrupt handler returns control to the interrupted program. It must restart the interrupted process in exactly the same state that it was in when the interrupt occurred, which means restoring all the internal registers to their preinterrupt state.

The essential difference between traps and interrupts is this: *traps* are synchronous with the program and *interrupts* are asynchronous. If the program is rerun a million times with the same input, the traps will reoccur in the same place each time but the interrupts may vary, depending, for example, on precisely when a person at a terminal hits carriage return. The reason for the reproducibility of traps and irreproducibility of interrupts is that traps are caused directly by the program and interrupts are, at best, indirectly caused by the program.

To see how interrupts really work, let us consider a common example: a computer wants to output a line of characters to a terminal. The system software first collects all the characters to be written to the terminal together in a buffer, initializes a global variable *ptr*, to point to the start of the buffer, and sets a second global variable *count* equal to the number of characters to be output. Then it checks to see if the terminal is ready and if so, outputs the first character (e.g., using registers like those of Fig. 5-30). Having started the I/O, the CPU is then free to run another program or do something else.

In due course of time, the character is displayed on the screen. The interrupt can now begin. In simplified form, the steps are as follows.

HARDWARE ACTIONS

1. The device controller asserts an interrupt line on the system bus to start the interrupt sequence.
2. As soon as the CPU is prepared to handle the interrupt, it asserts an interrupt acknowledge signal on the bus.
3. When the device controller sees that its interrupt signal has been acknowledged, it puts a small integer on the data lines to identify itself. This number is called the **interrupt vector**.
4. The CPU removes the interrupt vector from the bus and saves it temporarily.
5. Then the CPU pushes the program counter and PSW onto the stack.

6. The CPU then locates a new program counter by using the interrupt vector as an index into a table in low memory. If the program counter is 4 bytes, for example, then interrupt vector n corresponds to address $4n$. This new program counter points to the start of the interrupt service routine for the device causing the interrupt. Often the PSW is loaded or modified as well (e.g., to disable further interrupts).

SOFTWARE ACTIONS

7. The first thing the interrupt service routine does is save all the registers it uses so they can be restored later. They can be saved on the stack or in a system table.
8. Each interrupt vector is generally shared by all the devices of a given type, so it is not yet known which terminal caused the interrupt. The terminal number can be found by reading some device register.
9. Any other information about the interrupt, such as status codes, can now be read in.
10. If an I/O error occurred, it can be handled here.
11. The global variables, *ptr* and *count*, are updated. The former is incremented, to point to the next byte, and the latter is decremented, to indicate that 1 byte fewer remains to be output. If *count* is still greater than 0, there are more characters to output. Copy the one now pointed to by *ptr* to the output buffer register.
12. If required, a special code is output to tell the device or the interrupt controller that the interrupt has been processed.
13. Restore all the saved registers.
14. Execute the RETURN FROM INTERRUPT instruction, putting the CPU back into the mode and state it had just before the interrupt happened. The computer then continues from where it was.

A key concept related to interrupts is **transparency**. When an interrupt happens, some actions are taken and some code runs, but when everything is finished, the computer should be returned to exactly the same state it had before the interrupt. An interrupt routine with this property is said to be transparent. Having all interrupts be transparent makes interrupts much easier to understand.

If a computer has only one I/O device, then interrupts always work as we have just described, and there is nothing more to say about them. However, a large computer may have many I/O devices, and several may be running at the same time, possibly on behalf of different users. A nonzero probability exists that while an interrupt routine is running, a second I/O device wants to generate *its* interrupt.

Two approaches can be taken to this problem. The first one is for all interrupt routines to disable subsequent interrupts as the very first thing they do, even before saving the registers. This approach keeps things simple, as interrupts are then taken strictly sequentially, but it can lead to problems for devices that cannot tolerate much delay. If the first one has not yet been processed when the second one arrives, data may be lost.

When a computer has time-critical I/O devices, a better design approach is to assign each I/O device a priority, high for very critical devices and low for less critical ones. Similarly, the CPU should also have priorities, typically determined by a field in the PSW. When a priority n device interrupts, the interrupt routine should also run at priority n.

While a priority n interrupt routine is running, any attempt by a device with a lower priority to cause an interrupt is ignored until the interrupt routine is finished and the CPU goes back to running lower-priority code. On the other hand, interrupts from higher-priority devices should be allowed to happen with no delay.

With interrupt routines themselves subject to interrupt, the best way to keep the administration straight is to make sure that all interrupts are transparent. Let us consider a simple example of multiple interrupts. A computer has three I/O devices, a printer, a disk, and an RS232 (serial) line, with priorities 2, 4, and 5, respectively. Initially ($t = 0$), a user program is running, when suddenly at $t = 10$ a printer interrupt occurs. The printer Interrupt Service Routine (ISR) is started up, as shown in Fig. 5-43.

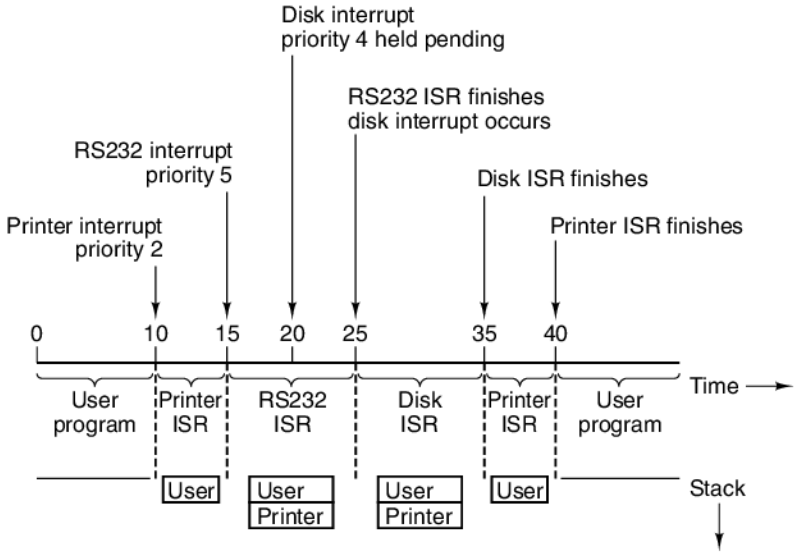


Figure 5-43. Time sequence of multiple-interrupt example.

At $t = 15$, the RS232 line wants attention and generates an interrupt. Since the RS232 line has a higher priority (5) than the printer (2), the interrupt happens. The

state of the machine, which is now running the printer interrupt service routine, is pushed onto the stack, and the RS232 interrupt service routine is started.

A little later, at $t = 20$, the disk is finished and wants service. However, its priority (4) is lower than that of the interrupt routine currently running (5), so the CPU hardware does not acknowledge the interrupt, and it is held pending. At $t = 25$, the RS232 routine is finished, so it returns to the state it was in just before the RS232 interrupt happened, namely, running the printer interrupt service routine at priority 2. As soon as the CPU switches to priority 2, before even one instruction can be executed, the disk interrupt at priority 4 now is allowed in, and the disk service routine runs. When it finishes, the printer routine gets to continue. Finally, at $t = 40$, all the interrupt service routines have completed and the user program continues from where it left off.

Since the 8088, the Intel CPU chips have had two interrupt levels (priorities): maskable and nonmaskable. Nonmaskable-interrupts are generally used only for signaling near-catastrophes, such as memory parity errors. All the I/O devices use the one maskable interrupt.

When an I/O device issues an interrupt, the CPU uses the interrupt vector to index into a 256-entry table to find the address of the interrupt service routine. The table entries are 8-byte segment descriptors and the table can begin anywhere in memory. A global register points to its start.

With only one usable interrupt level, there is no way for the CPU to let a high-priority device interrupt a medium-priority interrupt service routine while prohibiting a low-priority device from doing so. To solve this problem, the Intel CPUs are normally used with an external interrupt controller (e.g., an 8259A). When the first interrupt comes in, say at priority n, the CPU is interrupted. If a subsequent interrupt comes in at a higher priority, the interrupt controller interrupts a second time. If the second interrupt is at a lower priority, it is held until the first one is finished. To make this scheme work, the interrupt controller must know when the current interrupt service routine is finished, so the CPU must send it a command when the current interrupt has been fully processed.

5.7 A DETAILED EXAMPLE: THE TOWERS OF HANOI

Now that we have studied the ISA of three machines, let us put all the pieces together by taking a close look at the same example program for the two larger machines. Our example is the Towers of Hanoi program. We gave a Java version of this program in Fig. 5-39. In the following sections we will give assembly-code programs for the Towers of Hanoi.

However, we will cheat a tiny bit. Rather than give the translation of the Java version, for the Core i7 and OMAP4430 ARM CPU we will give the translation of a C version to avoid some problems with Java I/O. The only difference is the replacement of the Java call to *println* with the standard C statement

```
printf("Move a disk from %d to %d\n", i, j)
```

For our purposes, the syntax of *printf* format strings is unimportant (basically, the string is printed literally except that *%d* means print the next integer in decimal). The only thing that is relevant here is that the procedure is called with three parameters: a format string and two integers.

The reason for using the C version for the Core i7 and OMAP4430 ARM CPU is that the Java I/O library is not available in native form for these machines; the C library is. The difference is minimal, affecting only the print statement.

5.7.1 The Towers of Hanoi in Core i7 Assembly Language

Figure 5-44 gives a possible translation of the C version of the Towers of Hanoi for the Core i7. For the most part, it is fairly straightforward. The EBP register is used as the frame pointer. The first two words are used for linkage, so the first actual parameter, n (or N here, as MASM is case insensitive), is at EBP + 8, followed by i and j at EBP +12 and EBP + 16, respectively. The local variable, k, is in EBP + 20.

The procedure begins by establishing the new frame at the end of the old one. It does this by copying ESP to the frame pointer, EBP. Then it compares n to 1, branching off to the else clause if $n > 1$. The then code pushes three values on the stack: the address of the format string, i, and j, and calls itself.

The parameters are pushed in reverse order, which is required for C programs. This is necessary to put the pointer to the format string on top of the stack. Since *printf* has a variable number of parameters, if the parameters were pushed in forward order, *printf* would not know how deep in the stack the format string was.

After the call, 12 is added to ESP to remove the parameters from the stack. Of course, they are not really erased from memory, but the adjustment of ESP makes them inaccessible via the normal stack operations.

The else clause, which starts at *L1*, is straightforward. It first computes $6 - i - j$ and stores this value in k. No matter what values i and j have, the third peg is always $6 - i - j$. Saving it in k saves the trouble of recomputing it the second time.

Next, the procedure calls itself three times, with different parameters each time. After each call, the stack is cleaned up. That is all there is to it.

Recursive procedures sometimes confuse people at first, but when viewed at this level, they are straightforward. All that happens is that the parameters are pushed onto the stack and the procedure itself is called.

5.7.2 The Towers of Hanoi in OMAP4430 ARM Assembly Language

Now let us try again, only this time for the OMAP4430 ARM. The code is listed in Fig. 5-45. Because the OMAP4430 ARM code is especially unreadable, even for assembly code and even after a lot of practice, we have taken the liberty to

```
        .686                            ; compile for Core i7 class processor
.MODEL FLAT
PUBLIC _towers                          ; export 'towers'
EXTERN _printf:NEAR                     ; import printf
.CODE
_towers: PUSH EBP                       ; save EBP (frame pointer) and decrement ESP
        MOV EBP, ESP                    ; set new frame pointer above ESP
        CMP [EBP+8], 1                  ; if (n == 1)
        JNE L1                          ; branch if n is not 1
        MOV EAX, [EBP+16]               ; printf(" ...", i, j);
        PUSH EAX                        ; note that parameters i, j and the format
        MOV EAX, [EBP+12]               ; string are pushed onto the stack
        PUSH EAX                        ; in reverse order. This is the C calling convention
        PUSH OFFSET FLAT:format         ; offset flat means the address of format
        CALL _printf                    ; call printf
        ADD ESP, 12                     ; remove params from the stack
        JMP Done                        ; we are finished
L1:     MOV EAX, 6                      ; start k = 6 – i – j
        SUB EAX, [EBP+12]               ; EAX = 6 – i
        SUB EAX, [EBP+16]               ; EAX = 6 – i – j
        MOV [EBP+20], EAX               ; k = EAX
        PUSH EAX                        ; start towers(n – 1, i, k)
        MOV EAX, [EBP+12]               ; EAX = i
        PUSH EAX                        ; push i
        MOV EAX, [EBP+8]                ; EAX = n
        DEC EAX                         ; EAX = n – 1
        PUSH EAX                        ; push n – 1
        CALL _towers                    ; call towers(n – 1, i, 6 – i – j)
        ADD ESP, 12                     ; remove params from the stack
        MOV EAX, [EBP+16]               ; start towers(1, i, j)
        PUSH EAX                        ; push j
        MOV EAX, [EBP+12]               ; EAX = i
        PUSH EAX                        ; push i
        PUSH 1                          ; push 1
        CALL _towers                    ; call towers(1, i, j)
        ADD ESP, 12                     ; remove params from the stack
        MOV EAX, [EBP+12]               ; start towers(n – 1, 6 – i – j, i)
        PUSH EAX                        ; push i
        MOV EAX, [EBP+20]               ; EAX = k
        PUSH EAX                        ; push k
        MOV EAX, [EBP+8]                ; EAX = n
        DEC EAX                         ; EAX = n–1
        PUSH EAX                        ; push n – 1
        CALL _towers                    ; call towers(n – 1, 6 – i – j, i)
        ADD ESP, 12                     ; adjust stack pointer
Done:   LEAVE                           ; prepare to exit
        RET 0                           ; return to the called
.DATA
format  DB "Move disk from %d to %d\n"  ; format string
END
```

Figure 5-44. The Towers of Hanoi for the Core i7.

define a few symbols in the beginning to clean it up. To make this work, the program has to be run through a program called *cpp*, the C preprocessor, before assembling it. Also we have used lowercase letters here because the OMAP4430 ARM assembler insists on them (in case any readers wish to type the program in).

Algorithmically, the OMAP4430 ARM version is identical to the Core i7 version. Both test n to start with, branching to the else clause if $n > 1$. The main complexity of the ARM version is due to some properties of the ISA.

To start with, the OMAP4430 ARM has to pass the address of the format string to *printf*, but the machine cannot just move the address to the register that holds the outgoing parameter because there is no way to put a 32-bit constant in a register in one instruction. It takes two instructions to do this, MOVW and MOVT.

The next thing to notice is that the stack adjustments are automatically handled by the PUSH and POP instructions at the beginning and end of functions. These instructions also handle the saving and restoring of the return address, by saving the *LR* register on entry and restoring the PC on function exit.

5.8 THE IA-64 ARCHITECTURE AND THE ITANIUM 2

Around year 2000, some engineers inside Intel felt the company was getting to the point where it had just about squeezed every last drop of juice out of the IA-32 line of processors. New models were still seeing benefits from advances in manufacturing technology, which means smaller transistors (hence faster clock speeds). However, finding new tricks to speed up the implementation even more was getting harder and harder as the constraints imposed by the IA-32 ISA were looming larger all the time.

Some engineers felt that the only real solution was to abandon the IA-32 as the main line of development and go to a completely new ISA. This is, in fact, what Intel started working toward. In fact, it had plans for two new architectures. The EMT-64 is a wider version of the traditional Pentium ISA, with 64-bit registers and a 64-bit address space. This new ISA solves the address-space problem but still has all the implementation complexities of its predecessors. It can best be thought of as a wider Pentium.

The other new architecture, which was developed jointly by Intel and Hewlett Packard, was named the **IA-64**. It is a full 64-bit machine from beginning to end, not an extension of an existing 32-bit machine. Furthermore, it is a radical departure from the IA-32 architecture in many ways. The initial market was for high-end servers, but Intel had hoped it would catch on in the desktop world eventually. That did not happen. Bad as it was, the customers refused to abandon the IA-32. Nevertheless, the architecture is so radically different from anything we have studied so far that it is worth examining just for that reason. The first implementation of the IA-64 architecture is the Itanium series. In the remainder of this section we will study the IA-64 architecture and the Itanium 2 CPU that implements it.

```
#define Param0                          r0
#define Param1                          r1
#define Param2                          r2
#define FormatPtr                       r0
#define k                               r7
#define n_minus_1                       r5

        .text
towers: push {r3, r4, r5, r6, r7, lr}   @ save return addr and touched regs
        mov r4, Param1
        mov r6, Param2
        cmp Param0, #1                  @ is (n == 1)?
        bne else                        @ if not, jump to else code sequence
        movw FormatPtr, #:lower16:format @ load format string pointer
        movt FormatPtr, #:upper16:format
        bl printf                       @ print move
        pop {r3, r4, r5, r6, r7, pc}

else:   rsb k, r1, #6                   @ k = 6 – i – j
        subs k, k, r2
        add n_minus_1, r0, #-1          @ compute (n-1) for recursive call
        mov r0, n_minus_1
        mov r2, k
        bl towers                       @ call towers(n–1, i, k)
        mov r0, #1
        mov r1, r4
        mov r2, r6
        bl towers                       @ call towers(1, k, j)
        mov r0, n_minus_1
        mov r1, k
        mov r2, r6
        bl towers                       @ call towers(n–1, k, j)
        pop {r3, r4, r5, r6, r7, pc}    @ restore touched registers and return to called

        .global main
main:   push {lr}                       @ save called's return address
        mov Param0, #3
        mov Param1, #1
        mov Param2, Param0
        bl towers                       @ call towers(3, 1, 3)
        pop {pc}                        @ pop return address, return to called

format: .ascii "Move a disk from %d to %d\n\0"
```

Figure 5-45. The Towers of Hanoi for the OMAP4430 ARM CPU.

5.8.1 The Problem with the IA-32 ISA

Before getting into the details of the IA-64 and Itanium 2, it is useful to review what is wrong with the IA-32 ISA to see what problems Intel was trying to solve with the new architecture. The main fact of life that causes all the trouble is that

IA-32 is an ancient ISA with all the wrong properties for current technology. It is a CISC ISA with variable-length instructions and a myriad of different formats that are hard to decode quickly on the fly. Current technology works best with RISC ISAs that have one instruction length and a fixed-length opcode that is easy to decode. The IA-32 instructions can be broken up into RISC-like micro-operations at execution time, but doing so requires hardware (chip area), takes time, and adds complexity to the design. That is strike one.

The IA-32 is also a two-address memory-oriented ISA. Most instructions reference memory, and most programmers and compilers think nothing of referencing memory all the time. Current technology favors load/store ISAs that reference memory only to get the operands into registers but otherwise perform all their calculations using three-address memory register instructions. And with CPU clock speeds going up much faster than memory speeds, the problem will get worse with time. That is strike two.

The IA-32 also has a small and irregular register set. Not only does this tie compilers in knots, but the small number of general-purpose registers (four or six, depending on how you count ESI and EDI) requires intermediate results to be spilled into memory all the time, generating extra memory references even when they are not logically needed. That is strike three. The IA-32 is out.

Now let us start the second inning. The small number of registers causes many dependences, especially unnecessary WAR dependences, because results have to go somewhere and no extra registers are available. Getting around the lack of registers requires the implementation to do renaming internally—a terrible hack if ever there was one—to secret registers inside the reorder buffer. To avoid blocking on cache misses too often, instructions have to be executed out of order. However, the IA-32's semantics specify precise interrupts, so the out-of-order instructions have to be retired in order. All of these things require a lot of very complex hardware. Strike four.

Doing all this work quickly requires a deep pipeline. In turn, the deep pipeline means that instructions entered into it take many cycles before they are finished. Consequently, very accurate branch prediction is essential to make sure the right instructions are being entered into the pipeline. Because a misprediction requires the pipeline to be flushed, at great cost, even a fairly low misprediction rate can cause a substantial performance degradation. Strike five.

To alleviate the problems with mispredictions, the processor has to do speculative execution, with all the problems it entails, especially when memory references on the wrong path cause an exception. Strike six.

We are not going to play the whole baseball game here, but it should be clear by now that there is a problem. And we have not even mentioned that IA-32's 32-bit addresses limit individual programs to 4 GB of memory, which is a big problem on servers. The EMT-64 solves this problem but not all the others.

All in all, the situation with IA-32 can be favorably compared to the state of celestial mechanics just prior to Copernicus. The then-current theory dominating

astronomy was that the earth was fixed and motionless in space and that the planets moved in circles with epicycles around it. However, as observations got better and more deviations from this model could be clearly observed, epicycles were added to the epicycles until the whole model just collapsed from its internal complexity.

Intel is in the same pickle now. A huge fraction of all the transistors on the Core i7 are devoted to decomposing CISC instructions, figuring out what can be done in parallel, resolving conflicts, making predictions, repairing the consequences of incorrect predictions, and other bookkeeping, leaving surprisingly few for doing the real work the user asked for. The conclusion that Intel is being inexorably driven to is the only sane conclusion: junk the whole thing (IA-32) and start all over with a clean slate (IA-64). The EMT-64 provides some breathing room, but it really papers over the complexity issue.

5.8.2 The IA-64 Model: Explicitly Parallel Instruction Computing

The key idea behind the IA-64 is moving work from run time to compile time. On the Core i7, during execution the CPU reorders instructions, renames registers, schedules functional units, and does a lot of other work to determine how to keep all the hardware resources fully occupied. In the IA-64 model, the compiler figures out all these things in advance and produces a program that can be run as is, without the hardware having to juggle everything during execution. For example, rather than tell the compiler that the machine has eight registers when it actually has 128 and then try to figure out at run time how to avoid dependences, in the IA-64 model, the compiler is told how many registers the machine really has so it can produce a program that does not have any register conflicts to start with. Similarly, in this model, the compiler keeps track of which functional units are busy and does not issue instructions that use functional units that are not available. The model of making the underlying parallelism in the hardware visible to the compiler is called **EPIC** (**Explicitly Parallel Instruction Computing**). To some extent, EPIC can be thought of as the successor to RISC.

The IA-64 model has a number of features that speed up performance. These include reducing memory references, instruction scheduling, reducing conditional branches, and speculation. We will now examine each of these in turn and discuss how they are implemented in the Itanium 2.

5.8.3 Reducing Memory References

The Itanium 2 has a simple memory model. Memory consists of up to 2^{64} bytes of linear memory. Instructions are available to access memory in units of 1, 2, 4, 8, 16, and 10 bytes, the latter for 80-bit IEEE 745 floating-point numbers. Memory references need not be aligned on their natural boundaries, but a performance penalty is incurred if they are not. Memory can be either big endian or little endian, determined by a bit in a register loadable by the operating system.

Memory access is a huge bottleneck in all modern computers because CPUs are so much faster than memory. One way to reduce memory references is to have a large level 1 cache on chip and an even larger level 2 cache close to the chip. All modern designs have these two caches. But one can go beyond caching to look for other ways to reduce memory references, and the IA-64 uses some of these ways.

The best way to speed up memory references is to avoid having them in the first place. The Itanium 2 implementation of the IA-64 model has 128 general-purpose 64-bit registers. The first 32 of these are static, but the remaining 96 are used as a register stack, very similar to the register window scheme in other RISC processors, such as the UltraSPARC. However, unlike the UltraSPARC, the number of registers visible to the program is variable and can change from procedure to procedure. Thus each procedure has access to 32 static registers and some (variable) number of dynamically allocated registers.

When a procedure is called, the register stack pointer is advanced so the input parameters are visible in registers, but no registers are allocated for local variables. The procedure itself decides how many registers it needs and advances the register stack pointer to allocate them. These registers need not be saved on entry or restored on exit, although if the procedure needs to modify a static register it must take care to explicitly save it first and restore it later. By making the number of registers available variable and tailored to what each procedure needs, scarce registers are not wasted and procedure calls can go deeper before registers have to be spilled to memory.

The Itanium 2 also has 128 floating-point registers in IEEE 745 format. They do not operate as a register stack. This very large number of registers means that many floating-point computations can keep all their intermediate results in registers and avoid having to store temporary results in memory.

There are also 64 1-bit predicate registers, eight branch registers, and 128 special-purpose application registers used for various purposes, such as passing parameters between application programs and the operating system. An overview of the Itanium 2's registers is given in Fig. 5-46.

5.8.4 Instruction Scheduling

One of the main problems in the Core i7 is the difficulty of scheduling the various instructions over the various functional units and avoiding dependences. Exceedingly complex mechanisms are needed to handle these issues at run time, and a large fraction of the chip area is devoted to managing them. The IA-64 and Itanium 2 avoid all these problems by having the compiler do the work. The key idea is that a program consists of a sequence of **instruction groups**. Within certain boundaries, all the instructions within a group do not conflict with one another, do not use more functional units and resources than the machine has, do not contain RAW and WAW dependences, and have only certain restricted WAR dependences. Consecutive instruction groups give the appearance of being executed

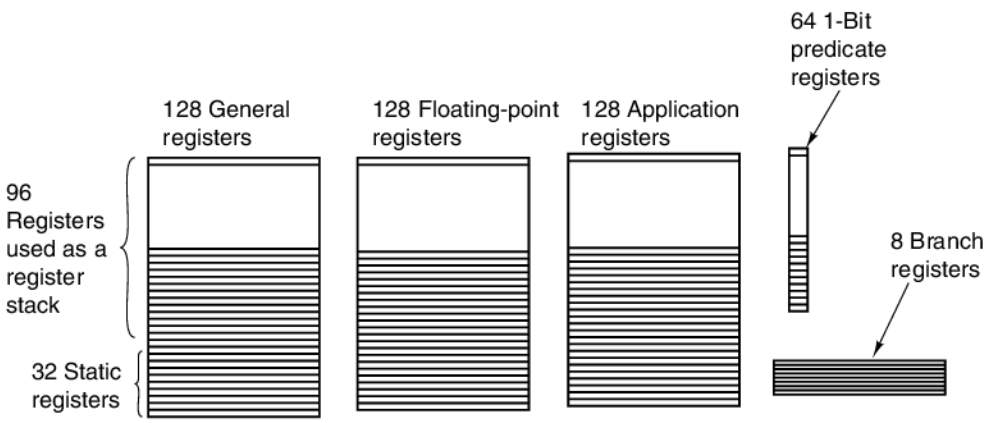


Figure 5-46. The Itanium 2's registers.

strictly sequentially with the second group not starting until the first one has completed. The CPU may, however, start the second group (in part) as soon as it feels it is safe to.

As a consequence of these rules, the CPU is free to schedule the instructions within a group in any order it chooses, possibly in parallel if it can, without having to worry about conflicts. If the instruction group violates the rules, program behavior is undefined. It is up to the compiler to reorder the assembly code generated from the source program to meet all these requirements. For rapid compilation while a program is being debugged, the compiler can put every instruction in a different group, which is easy to do but gives poor performance. When it is time to produce production code, the compiler can spend a long time optimizing it.

Instructions are organized into 128-bit **bundles** as shown at the top of Fig. 5-47. Each bundle contains three 41-bit instructions and a 5-bit template. An instruction group need not be an integral number of bundles; it can start and end in the middle of a bundle.

Over 100 instruction formats exist. A typical one, in this case for ALU operations such as ADD, which sums two registers into a third one, is shown in Fig. 5-47. The first field, the OPERATION GROUP, is the major group and mostly tells the broad class of the instruction, such as an integer ALU operation. The next field, the OPERATION TYPE, gives the specific operation required, such as ADD or SUB. Then come the three register fields. Finally, we have the PREDICATE REGISTER, to be described shortly.

The bundle template essentially tells which functional units the bundle needs and also the position of an instruction-group boundary present, if any. The major functional units are the integer ALU, non-ALU integer instructions, memory operations, floating-point operations, branching, and other. Of course, with six units and three instructions, complete orthogonality would require 216 combinations,