protocols make for a modular and flexible design, and have been widely used in the world of network software for decades. What is new here is building them into the "bus" hardware.

The PCI Express protocol stack is shown in Fig. 3-57(a). It is discussed below.

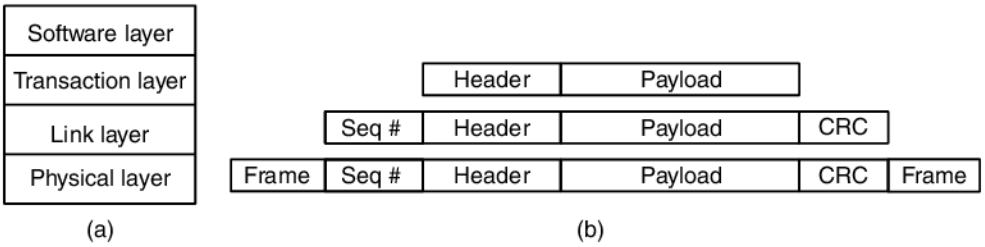


Figure 3-57. (a) The PCI Express protocol stack. (b) The format of a packet.

Let us examine the layers from the bottom up. The lowest is the **physical layer**. It deals with moving bits from a sender to a receiver over a point-to-point connection. Each point-to-point connection consists of one or more pairs of simplex (i.e., unidirectional) links. In the simplest case, there is one pair in each direction, but having 2, 4, 8, 16, or 32 pairs is also allowed. Each link is called a **lane**. The number of lanes in each direction must be the same. First-generation products must support a data rate each way of at least 2.5 Gbps, but the speed is expected to migrate to 10 Gbps each way fairly soon.

Unlike the ISA/EISA/PCI buses, PCI Express does not have a master clock. Devices are free to start transmitting as soon as they have data to send. This freedom makes the system faster but also leads to a problem. Suppose that a 1 bit is encoded as +3 volts and a 0 bit as 0 volts. If the first few bytes are all 0s, how does the receiver know data are being transmitted? After all, a run of 0 bits looks the same as an idle link. The problem is solved using what is called **8b/10b encoding**. In this scheme, 10 bits are used to encode 1 byte of actual data in a 10-bit symbol. Of the 1024 possible 10-bit symbols, the legal ones have been chosen to have enough clock transitions to keep the sender and receiver synchronized on the bit boundaries even without a master clock. A consequence of 8b/10b encoding is that a link with a gross capacity of 2.5 Gbps can carry only 2 Gbps of (net) user data.

Whereas the physical layer deals with bit transmission, the **link layer** deals with packet transmission. It takes the header and payload given to it by the transaction layer and adds to them a sequence number and an error-correcting code called a **CRC** (**Cyclic Redundancy Check**). The CRC is generated by running a certain algorithm on the header and payload data. When a packet is received, the receiver performs the same computation on the header and data and compares the result with the CRC attached to the packet. If they agree, it sends back a short **acknowledgment packet** affirming its correct arrival. If they disagree, the receiver asks for a retransmission. In this manner, data integrity is greatly improved

over the PCI bus system, which does not have any provision for verification and retransmission of data sent over the bus.

To prevent having a fast sender bury a slow receiver in packets it cannot handle, a **flow control** mechanism is used. The mechanism is that the receiver gives the transmitter a certain number of credits, basically corresponding to the amount of buffer space it has available to store incoming packets. When the credits are used up, the transmitter has to stop sending until it is given more credits. This scheme, which is widely used in all networks, prevents losing data due to a mismatch of transmitter and receiver speeds.

The **transaction layer** handles bus actions. Reading a word from memory requires two transactions: one initiated by the CPU or DMA channel requesting some data and one initiated by the target supplying the data. But the transaction layer does more than handle pure reads and writes. It adds value to the raw packet transmission offered by the link layer. To start with, it can divide each lane into up to eight **virtual circuits**, each handling a different class of traffic. The transaction layer can tag packets according to their traffic class, which may include attributes such as high priority, low priority, do not snoop, may be delivered out of order, and more. The switch may use these tags when deciding which packet to handle next.

Each transaction uses one of four address spaces:

1. Memory space (for ordinary reads and writes).
2. I/O space (for addressing device registers).
3. Configuration space (for system initialization, etc.).
4. Message space (for signaling, interrupts, etc.).

The memory and I/O spaces are similar to what current systems have. The configuration space can be used to implement features such as plug-and-play. The message space takes over the role of many of the existing control signals. Something like this space is needed because none of the PCI bus' control lines exist in PCI Express.

The **software layer** interfaces the PCI Express system to the operating system. It can emulate the PCI bus, making it possible to run existing operating systems unmodified on PCI Express systems. Of course, operating like this will not exploit the full power of PCI Express, but backward compatibility is a necessary evil that is needed until operating systems have been modified to fully utilize PCI Express. Experience shows this can take a while.

The flow of information is illustrated in Fig. 3-57(b). When a command is given to the software layer, it hands it to the transaction layer, which formulates it in terms of a header and a payload. These two parts are then passed to the link layer, which attaches a sequence number to the front and a CRC to the back. This enlarged packet is then passed on to the physical layer, which adds framing information on each end to form the physical packet that is actually transmitted. At the

receiving end, the reverse process takes place, with the link header and trailer being stripped and the result being given to the transaction layer.

The concept of each layer adding additional information to the data as it works its way down the protocol has been used for decades in the networking world with great success. The big difference between a network and PCI Express is that in the networking world the code in the various layers is nearly always software that is part of the operating system. With PCI Express it is all part of the device hardware.

PCI Express is a complicated subject. For more information see Mayhew and Krishnan (2003) and Solari and Congdon (2005). It is also still evolving. In 2007, PCIe 2.0 was released. It supports 500 MB/s per lane up to 32 lines, for a total bandwidth of 16 GB/sec. Then came PCIe 3.0 in 2011, which changed the encoding from 8b/10b to 128b/130b and can run at 8 billion transactions/sec, double PCIe 2.0.

3.6.3 The Universal Serial Bus

The PCI bus and PCI Express are fine for attaching high-speed peripherals to a computer, but they are too expensive for low-speed I/O devices such as keyboards and mice. Historically, each standard I/O device was connected to the computer in a special way, with some free ISA and PCI slots for adding new devices. Unfortunately, this scheme has been fraught with problems from the beginning.

For example, each new I/O device often comes with its own ISA or PCI card. The user is often responsible for setting switches and jumpers on the card and making sure the settings do not conflict with other cards. Then the user must open up the case, carefully insert the card, close the case, and reboot the computer. For many users, this process is difficult and error prone. In addition, the number of ISA and PCI slots is very limited (two or three typically). Plug-and-play cards eliminate the jumper settings, but the user still has to open the computer to insert the card and bus slots are still limited.

To deal with this problem, in 1993, representatives from seven companies (Compaq, DEC, IBM, Intel, Microsoft, NEC, and Northern Telecom) got together to design a better way to attach low-speed I/O devices to a computer. Since then, hundreds of other companies have joined them. The resulting standard, officially released in 1998, is called **USB** (**Universal Serial Bus**) and it is now widely implemented in personal computers. It is described further in Anderson (1997) and Tan (1997).

Some of the goals of the companies that originally conceived of the USB and started the project were as follows:

1. Users must not have to set switches or jumpers on boards or devices.
2. Users must not have to open the case to install new I/O devices.
3. There should be only one kind of cable to connect all devices.

4. I/O devices should get their power from the cable.
5. Up to 127 devices should be attachable to a single computer.
6. The system should support real-time devices (e.g., sound, telephone).
7. Devices should be installable while the computer is running.
8. No reboot should be needed after installing a new device.
9. The new bus and its I/O devices should be inexpensive to manufacture.

USB meets all these goals. It is designed for low-speed devices such as keyboards, mice, still cameras, snapshot scanners, digital telephones, and so on. Version 1.0 has a bandwidth of 1.5 Mbps, which is enough for keyboards and mice. Version 1.1 runs at 12 Mbps, which is enough for printers, digital cameras, and many other devices. Version 2.0 supports devices with up to 480 Mbps, which is sufficient to support external disk drives, high-definition webcams, and network interfaces. The recently defined USB version 3.0 pushes speeds up to 5 Gbps; only time will tell what new and bandwidth-hungry applications will spring forth from this ultra-high-bandwidth interface.

A USB system consists of a **root hub** that plugs into the main bus (see Fig. 3-51). This hub has sockets for cables that can connect to I/O devices or to expansion hubs, to provide more sockets, so the topology of a USB system is a tree with its root at the root hub, inside the computer. The cables have different connectors on the hub end and on the device end, to prevent people from accidentally connecting two hub sockets together.

The cable consists of four wires: two for data, one for power (+5 volts), and one for ground. The signaling system transmits a 0 as a voltage transition and a 1 as the absence of a voltage transition, so long runs of 0s generate a regular pulse stream.

When a new I/O device is plugged in, the root hub detects this event and interrupts the operating system. The operating system then queries the device to find out what it is and how much USB bandwidth it needs. If the operating system decides that there is enough bandwidth for the device, it assigns the new device a unique address (1–127) and downloads this address and other information to configuration registers inside the device. In this way, new devices can be added on-the-fly, without requiring any user configuration or the installation of any new ISA or PCI cards. Uninitialized cards start out with address 0, so they can be addressed. To make the cabling simpler, many USB devices contain built-in hubs to accept additional USB devices. For example, a monitor might have two hub sockets to accommodate the left and right speakers.

Logically, the USB system can be viewed as a set of bit pipes from the root hub to the I/O devices. Each device can split its bit pipe up into a maximum of 16 subpipes for different types of data (e.g., audio and video). Within each pipe or

subpipe, data flows from the root hub to the device or the other way. There is no traffic between two I/O devices.

Precisely every 1.00 ± 0.05 msec, the root hub broadcasts a new frame to keep all the devices synchronized in time. A frame is associated with a bit pipe and consists of packets, the first of which is from the root hub to the device. Subsequent packets in the frame may also be in this direction, or they may be back from the device to the root hub. A sequence of four frames is shown in Fig. 3-58.

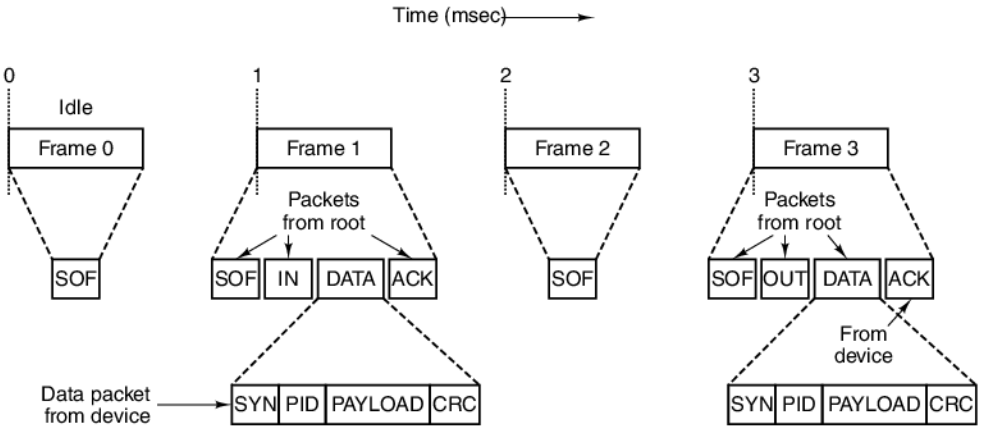


Figure 3-58. The USB root hub sends out frames every 1.00 msec.

In Fig. 3-58 there is no work to be done in frames 0 and 2, so all that is needed is one SOF (Start of Frame) packet. This packet is always broadcast to all devices. Frame 1 is a poll, for example a request to a scanner to return the bits it has found on the image it is scanning. Frame 3 consists of delivering data to some device, for example to a printer.

USB supports four kinds of frames: control, isochronous, bulk, and interrupt. Control frames are used to configure devices, give them commands, and inquire about their status. Isochronous frames are for real-time devices such as microphones, loudspeakers, and telephones that need to send or accept data at precise time intervals. They have a highly predictable delay but provide no retransmissions in the event of errors. Bulk frames are for large transfers to or from devices with no real-time requirements, such as printers. Finally, interrupt frames are needed because USB does not support interrupts. For example, instead of having the keyboard cause an interrupt whenever a key is struck, the operating system can poll it every 50 msec to collect any pending keystrokes.

A frame consists of one or more packets, possibly some in each direction. Four kinds of packets exist: token, data, handshake, and special. Token packets are from the root to a device and are for system control. The SOF, IN, and OUT packets in Fig. 3-58 are token packets. The SOF packet is the first in each frame and marks the beginning of the frame. If there is no work to do, the SOF packet is the

only one in the frame. The IN token packet is a poll, asking the device to return certain data. Fields in the IN packet tell which bit pipe is being polled so the device knows which data to return (if it has multiple streams). The OUT token packet announces that data for the device will follow. A fourth type of token packet, SETUP (not shown in the figure), is used for configuration.

Besides the token packet, three other kinds exist. These are DATA (used to transmit up to 64 bytes of information either way), handshake, and special packets. The format of a DATA packet is shown in Fig. 3-58. It consists of an 8-bit synchronization field, an 8-bit packet type (PID), the payload, and a 16-bit **CRC** (**Cyclic Redundancy Check**) to detect errors. Three kinds of handshake packets are defined: ACK (the previous data packet was correctly received), NAK (a CRC error was detected), and STALL (please wait—I am busy right now).

Now let us look at Fig. 3-58 again. Every 1.00 msec a frame must be sent from the root hub, even if there is no work. Frames 0 and 2 consist of just an SOF packet, indicating that there was no work. Frame 1 is a poll, so it starts out with SOF and IN packets from the computer to the I/O device, followed by a DATA packet from the device to the computer. The ACK packet tells the device that the data were received correctly. In case of an error, a NAK would be sent back to the device and the packet would be retransmitted for bulk data (but not for isochronous data). Frame 3 is similar in structure to frame 1, except that now the data flow is from the computer to the device.

After the USB standard was finalized in 1998, the USB designers had nothing to do so they began working on a new high-speed version of USB, called **USB 2.0**. This standard is similar to the older USB 1.1 and backward compatible with it, except that it adds a third speed, 480 Mbps, to the two existing speeds. There are also some minor differences, such as the interface between the root hub and the controller. With USB 1.1 two new interfaces were available. The first one, **UHCI** (**Universal Host Controller Interface**), was designed by Intel and put most of the burden on the software designers (read: Microsoft). The second one, **OHCI** (**Open Host Controller Interface**), was designed by Microsoft and put most of the burden on the hardware designers (read: Intel). In USB 2.0 everyone agreed to a single new interface called **EHCI** (**Enhanced Host Controller Interface**).

With USB now operating at 480 Mbps, it clearly competes with the IEEE 1394 serial bus popularly called FireWire, which runs at 400 Mbps or 800 Mbps. Because virtually every new Intel-based PC now comes with USB 2.0 or USB 3.0 (see below) 1394 is likely to vanish in due course. This disappearance is not so much due to obsolescence as to turf wars. USB is a product of the computer industry whereas 1394 comes from the consumer electronics industry. When it came to connecting cameras to computers, each industry wanted everyone to use its cable. It looks like the computer folks won this one.

Eight years after the introduction of USB 2.0, the **USB 3.0** interface standard was announced. USB 3.0 supports a whopping 5-Gbps bandwidth over the cable, although the link modulation is adaptive, and one is likely to achieve this top speed

only with professional-grade cabling. USB 3.0 devices are structurally identical to earlier USB devices, and they fully implement the USB 2.0 standard. If plugged into a USB 2.0 socket, they will operate correctly.

3.7 INTERFACING

A typical small- to medium-sized computer system consists of a CPU chip, chipset, memory chips, and some I/O devices, all connected by a bus. Sometimes, all of these devices are integrated into a system-on-a-chip, like the TI OMAP4430 SoC. We have already studied memories, CPUs, and buses in some detail. Now it is time to look at the last part of the puzzle, the I/O interfaces. It is through these I/O ports that the computer communicates with the external world.

3.7.1 I/O Interfaces

Numerous I/O interfaces are already available and new ones are being introduced all the time. Common interfaces include UARTs, USARTs, CRT controllers, disk controllers, and PIOs. A **UART** (**Universal Asynchronous Receiver Transmitter**) is an I/O interface that can read a byte from the data bus and output it a bit at a time on a serial line for a terminal, or input data from a terminal. UARTs usually allow various speeds from 50 to 19,200 bps; character widths from 5 to 8 bits; 1, 1.5, or 2 stop bits; and provide even, odd, or no parity, all under program control. **USART**s (**Universal Synchronous Asynchronous Receiver Transmitters**) can handle synchronous transmission using a variety of protocols as well as performing all the UART functions. Since UARTs have become less important as telephone modems are vanishing, let us now study the parallel interface as an example of an I/O chip.

PIO Interfaces

A typical **PIO** (**Parallel Input/Output**) interface (based on the classic Intel 8255A PIO design) is illustrated in Fig. 3-59. It has a collection of I/O lines (e.g., 24 I/O lines in the example in the figure) that can interface to any digital logic device interface, for example, keyboards, switches, lights, or printers. In a nutshell, the CPU program can write a 0 or 1 to any line, or read the input status of any line, providing great flexibility. A small CPU-based system using a PIO interface can control a variety of physical devices, such as a robot, toaster, or electron microscope. Typically, PIO interfaces are found in embedded systems.

The PIO interface is configured with a 3-bit configuration register, which specifies if the three independent 8-bit ports are to be used for digital signal input (0) or output (1). Setting the appropriate value in the configuration register will allow any combination of input and output for the three ports. Associated with

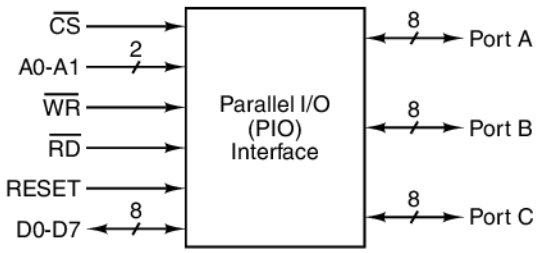


Figure 3-59. A 24-bit PIO Interface.

each port is an 8-bit latch register. To set the lines on an output port, the CPU just writes an 8-bit number into the corresponding register, and the 8-bit number appears on the output lines and stays there until the register is rewritten. To use a port configured for input, the CPU just reads the corresponding 8-bit latch register.

It is possible to build more sophisticated PIO interfaces. For example, one popular operating mode provides for handshaking with external devices. For example, to output to a device that is not always ready to accept data, the PIO can present data on an output port and wait for the device to send a pulse back saying that it has accepted the data and wants more. The necessary logic for latching such pulses and making them available to the CPU includes a ready signal plus an 8-bit register queue for each output port.

From the functional diagram of the PIO we can see that in addition to 24 pins for the three ports, it has eight lines that connect directly to the data bus, a chip select line, read and write lines, two address lines, and a line for resetting the chip. The two address lines select one of the four internal registers, corresponding to ports A, B, C, and the port configuration register. Normally, the two address lines are connected to the low-order bits of the address bus. The chip select line allows the 24-bit PIO to be combined to form larger PIO interfaces, by adding additional address lines and using them to select the proper PIO interface by asserting its chip select line.

3.7.2 Address Decoding

Up until now we have been deliberately vague about how chip select is asserted on the memory and I/O chips we have looked at. It is now time to look more carefully at how this is done. Let us consider a simple 16-bit embedded computer consisting of a CPU, a 2KB $\times$ 8 byte EPROM for the program, a $2-$KB $\times$ 8 byte RAM for the data, and a PIO interface. This small system might be used as a prototype for the brain of a cheap toy or simple appliance. Once in production, the EPROM might be replaced by a ROM.

The PIO interface can be selected in one of two ways: as a true I/O device or as part of memory. If we choose to use it as an I/O device, then we must select it

using an explicit bus line that indicates that an I/O device is being referenced, and not memory. If we use the other approach, **memory-mapped I/O**, then we must assign it 4 bytes of the memory space for the three ports and the control register. The choice is somewhat arbitrary. We will choose memory-mapped I/O because it illustrates some interesting issues in I/O interfacing.

The EPROM needs 2 KB of address space, the RAM also needs 2K of address space, and the PIO needs 4 bytes. Because our example address space is 64K, we must make a choice about where to put the three devices. One possible choice is shown in Fig. 3-60. The EPROM occupies addresses to 2K, the RAM occupies addresses 32 KB to 34 KB, and the PIO occupies the highest 4 bytes of the address space, 65532 to 65535. From the programmer's point of view, it makes no difference which addresses are used; however, for interfacing it does matter. If we had chosen to address the PIO via the I/O space, it would not need any memory addresses (but it would need four I/O space addresses).

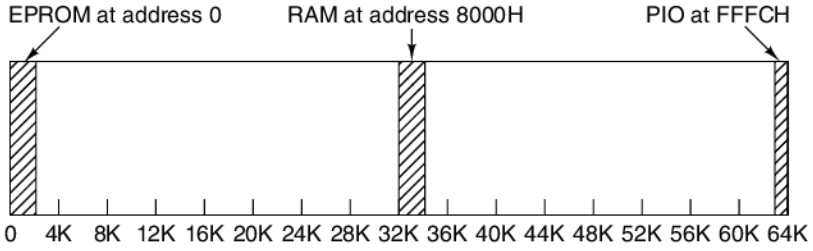


Figure 3-60. Location of the EPROM, RAM, and PIO in our 64-KB address space.

With the address assignments of Fig. 3-60, the EPROM should be selected by any 16-bit memory address of the form 00000xxxxxxxxxxx (binary). In other words, any address whose 5 high-order bits are all 0s falls in the bottom 2 KB of memory, hence in the EPROM. Thus, the EPROM's chip select could be wired to a 5-bit comparator, one of whose inputs was permanently wired to 00000.

A better way to achieve the same effect is to use a five-input OR gate, with the five inputs attached to address lines A11 to A15. If and only if all five lines are 0 will the output be 0, thus asserting $\overline{\text{CS}}$ (which is asserted low). This addressing approach is illustrated in Fig. 3-60(a) and is called full-address decoding.

The same principle can be used for the RAM. However, the RAM should respond to binary addresses of the form 10000xxxxxxxxxxx, so an additional inverter is needed as shown in the figure. The PIO address decoding is somewhat more complicated, because it is selected by the four addresses of the form 11111111111111xx. A possible circuit that asserts $\overline{\text{CS}}$ only when the correct address appears on the address bus is shown in the figure. It uses two eight-input NAND gates to feed an OR gate.

However, if the computer really consists of only the CPU, two memory chips, and the PIO, we can use a trick to greatly simplify the address decoding. The trick

is based on the fact that all EPROM addresses, and only EPROM addresses, have a 0 in the high-order bit, A15. Therefore, we can just wire $\overline{\text{CS}}$ to A15 directly, as shown in Fig. 3-61(b).

At this point the decision to put the RAM at 8000H may seem much less arbitrary. The RAM decoding can be done by noting that the only valid addresses of the form 10xxxxxxxxxxxxx are in the RAM, so 2 bits of decoding are sufficient. Similarly, any address starting with 11 must be a PIO address. The complete decoding logic is now two NAND gates and an inverter.

The address decoding logic of Fig. 3-61(b) is called **partial address decoding**, because the full addresses are not used. It has the property that a read from addresses 0001000000000000, 0001100000000000, or 0010000000000000 will give the same result. In fact, every address in the bottom half of the address space will select the EPROM. Because the extra addresses are not used, no harm is done, but if one is designing a computer that may be expanded in the future (an unlikely occurrence in a toy), partial decoding should be avoided because it ties up too much address space.

Another common address-decoding technique is to use a decoder, such as that shown in Fig. 3-13. By connecting the three inputs to the three high-order address lines, we get eight outputs, corresponding to addresses in the first 8K, second 8K, and so on. For a computer with eight RAMs, each 8K $\times$ 8, one such chip provides the complete decoding. For a computer with eight 2K $\times$ 8 memory chips, a single decoder is also sufficient, provided that the memory chips are each located in distinct 8-KB chunks of address space. (Remember our earlier remark that the position of the memory and I/O chips within the address space matters.)

3.8 SUMMARY

Computers are constructed from integrated circuit chips containing tiny switching elements called gates. The most common gates are AND, OR, NAND, NOR, and NOT. Simple circuits can be built up by directly combining individual gates.

More complex circuits are multiplexers, demultiplexers, encoders, decoders, shifters, and ALUs. Arbitrary Boolean functions can be programmed using a FPGA. If many Boolean functions are needed, FPGAs are often more efficient. The laws of Boolean algebra can be used to transform circuits from one form to another. In many cases more economical circuits can be produced this way.

Computer arithmetic is done by adders. A single-bit full adder can be constructed from two half adders. An adder for a multibit word can be built by connecting multiple full adders in such a way as to allow the carry out of each one feed into its left-hand neighbor.

The components of (static) memories are latches and flip-flops, each of which can store one bit of information. These can be combined linearly into latches and

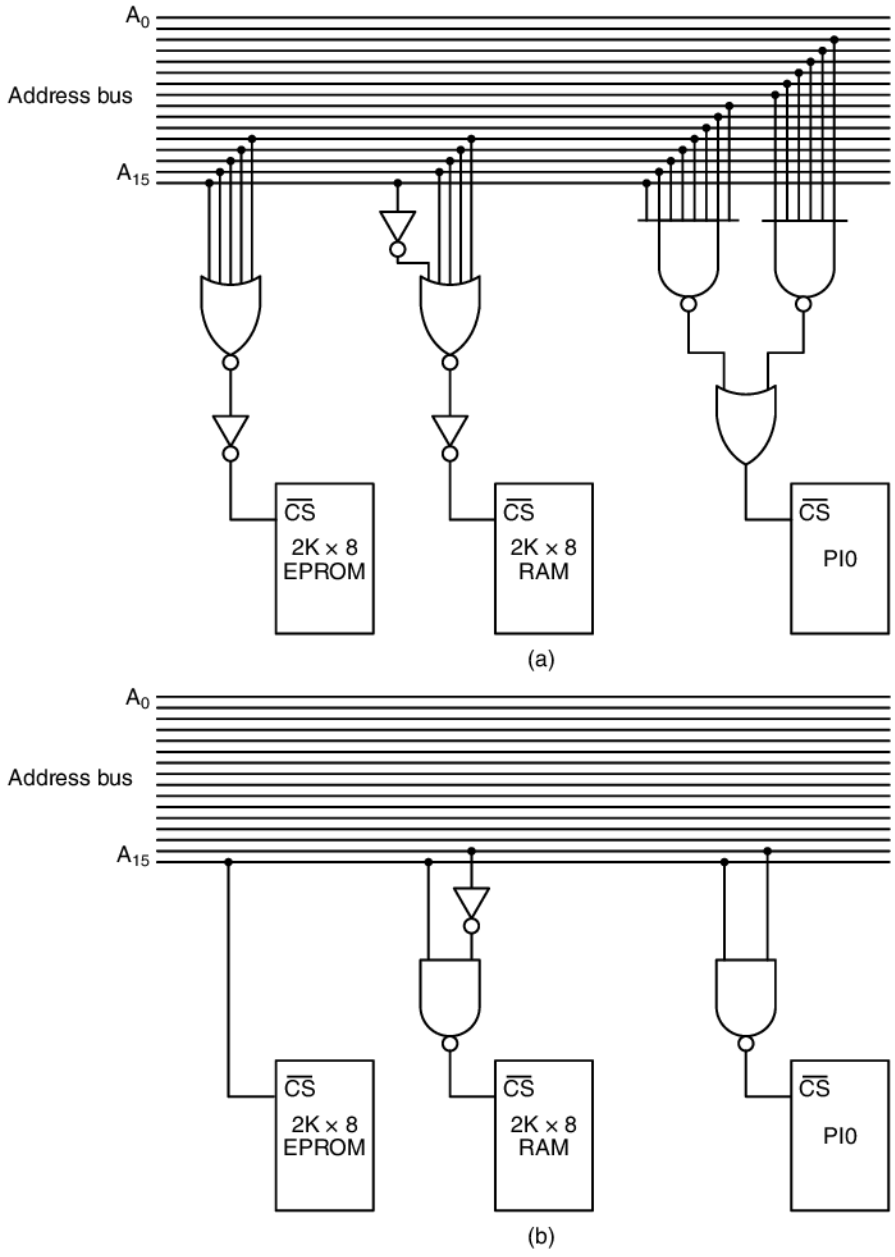


Figure 3-61. (a) Full address decoding. (b) Partial address decoding.

flip-flops for memories with any word size desired. Memories are available as RAM, ROM, PROM, EPROM, EEPROM, and flash. Static RAMs need not be refreshed; they keep their stored values as long as the power remains on. Dynamic RAMs, on the other hand, must be refreshed periodically to compensate for leakage from the little capacitors on the chip.

The components of a computer system are connected by buses. Many, but not all, of the pins on a typical CPU chip directly drive one bus line. The bus lines can be divided into address, data, and control lines. Synchronous buses are driven by a master clock. Asynchronous buses use full handshaking to synchronize the slave to the master.

The Core i7 is an example of a modern CPU. Modern systems using it have a memory bus, a PCIe bus, and a USB bus. The PCIe interconnect is the dominant way to connect the internal parts of a computer at high speeds. The ARM is also a modern high-end CPU but is intended for embedded systems and mobile devices where low power consumption is important. The Atmel ATmega168 is an example of a low-priced chip good for small, inexpensive appliances and many other price-sensitive applications.

Switches, lights, printers, and many other I/O devices can be interfaced to computers using parallel I/O interfaces. These chips can be configured to be part of the I/O space or the memory space, as needed. They can be fully decoded or partially decoded, depending on the application.

PROBLEMS

1. Analog circuits are subject to noise that can distort their output. Are digital circuits immune to noise? Discuss your answer.

2. A logician drives into a drive-in restaurant and says, "I want a hamburger or a hot dog and french fries." Unfortunately, the cook flunked out of sixth grade and does not know (or care) whether "and" has precedence over "or." As far as he is concerned, one interpretation is as good as the other. Which of the following cases are valid interpretations of the order? (Note that in English "or" means "exclusive or.")

a. Just a hamburger.
b. Just a hot dog.
c. Just french fries.
d. A hot dog and french fries.
e. A hamburger and french fries.
f. A hot dog and a hamburger.
g. All three.
h. Nothing—the logician goes hungry for being a wiseguy.

3. A missionary lost in Southern California stops at a fork in the road. He knows that two motorcycle gangs inhabit the area, one of which always tells the truth and one of which always lies. He wants to know which road leads to Disneyland. What question should he ask?

4. Use a truth table to show that $X = (X \text{ AND } Y) \text{ OR } (X \text{ AND NOT } Y)$.

5. There exist four Boolean functions of a single variable and 16 functions of two variables. How many functions of three variables are there? Of n variables?

6. There exist four Boolean functions of a single variable and 16 functions of two variables. How many functions of four variables are there?

7. Show how the AND function can be constructed from two NAND gates.

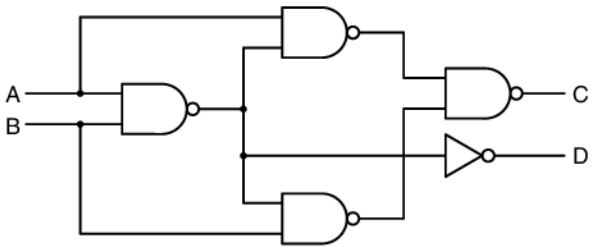
8. Using the three-variable multiplexer chip of Fig. 3-12, implement a function whose output is the parity of the inputs, that is, the output is 1 if and only if an even number of inputs are 1.

9. Put on your thinking cap. The three-variable multiplexer chip of Fig. 3-12 is actually capable of computing an arbitrary function of *four* Boolean variables. Describe how, and as an example, draw the logic diagram for the function that is 0 if the English word for the truth-table row has an even number of letters, 1 if it has an odd number of letters (e.g., 0000 = zero = four letters $\rightarrow$ 0; 0111 = seven = five letters $\rightarrow$ 1; 1101 = thirteen = eight letters $\rightarrow$ 0). *Hint*: If we call the fourth input variable D, the eight input lines may be wired to V_{cc}, ground, D, or $\bar{D}$.

10. Draw the logic diagram of a 2-bit encoder, a circuit with four input lines, exactly one of which is high at any instant, and two output lines whose 2-bit binary value tells which input is high.

11. Draw the logic diagram of a 2-bit demultiplexer, a circuit whose single input line is steered to one of the four output lines depending on the state of the two control lines.

12. What does this circuit do?



13. A common chip is a 4-bit adder. Four of these chips can be hooked up to form a 16-bit adder. How many pins would you expect the 4-bit adder chip to have? Why?

14. An n-bit adder can be constructed by cascading n full adders in series, with the carry into stage i, C_i, coming from the output of stage $i - 1$. The carry into stage 0, C_0, is 0. If each stage takes T nsec to produce its sum and carry, the carry into stage i will not be valid until iT nsec after the start of the addition. For large n the time required for the

carry to ripple through to the high-order stage may be unacceptably long. Design an adder that works faster. *Hint*: Each C_i can be expressed in terms of the operand bits A_{i-1} and B_{i-1} as well as the carry C_{i-1}. Using this relation it is possible to express C_i as a function of the inputs to stages 0 to $i-1$, so all the carries can be generated simultaneously.

15. If all the gates in Fig. 3-18 have a propagation delay of 1 nsec, and all other delays can be ignored, what is the earliest time a circuit using this design can be sure of having a valid output bit?

16. The ALU of Fig. 3-19 is capable of doing 8-bit 2's complement additions. Is it also capable of doing 2's complement subtractions? If so, explain how. If not, modify it to be able to do subtractions.

17. A 16-bit ALU is built up of 16 1-bit ALUs, each one having an add time of 10 nsec. If there is an additional 1-nsec delay for propagation from one ALU to the next, how long does it take for the result of a 16-bit add to appear?

18. Sometimes it is useful for an 8-bit ALU such as Fig. 3-19 to generate the constant -1 as output. Give two different ways this can be done. For each way, specify the values of the six control signals.

19. What is the quiescent state of the S and R inputs to an SR latch built of two NAND gates?

20. The circuit of Fig. 3-25 is a flip-flop that is triggered on the rising edge of the clock. Modify this circuit to produce a flip-flop that is triggered on the falling edge of the clock.

21. The 4×3 memory of Fig. 3-28 uses 22 AND gates and three OR gates. If the circuit were to be expanded to 256×8, how many of each would be needed?

22. To help meet the payments on your new personal computer, you have taken up consulting for fledgling SSI chip manufacturers. One of your clients is thinking about putting out a chip containing four D flip-flops, each containing both Q and $\bar{Q}$, on request of a potentially important customer. The proposed design has all four clock signals ganged together, also on request. Neither preset nor clear is present. Your assignment is to give a professional evaluation of the design.

23. As more and more memory is squeezed onto a single chip, the number of pins needed to address it also increases. It is often inconvenient to have large numbers of address pins on a chip. Devise a way to address 2^n words of memory using fewer than n pins.

24. A computer with a 32-bit wide data bus uses $1\text{M} \times 1$ dynamic RAM memory chips. What is the smallest memory (in bytes) that this computer can have?

25. Referring to the timing diagram of Fig. 3-38, suppose that you slowed the clock down to a period of 20 nsec instead of 10 nsec as shown but the timing constraints remained unchanged. How much time would the memory have to get the data onto the bus during T_3 after $\overline{\text{MREQ}}$ was asserted, in the worst case?

26. Again referring to Fig. 3-38, suppose that the clock remained at 100 MHz, but T_{DS} was increased to 4 nsec. Could 10-nsec memory chips be used?

27. In Fig. 3-38(b), T_{ML} is specified to be at least 2 nsec. Can you envision a chip in which it is negative? Specifically, could the CPU assert $\overline{\text{MREQ}}$ before the address was stable? Why or why not?

28. Assume that the block transfer of Fig. 3-42 were done on the bus of Fig. 3-38. How much more bandwidth is obtained by using a block transfer over individual transfers for long blocks? Now assume that the bus is 32 bits wide instead of 8 bits wide. Answer the question again.

29. Denote the transition times of the address lines of Fig. 3-39 as T_{A1} and T_{A2}, and the transition times of $\overline{\text{MREQ}}$ as T_{MREQ1} and T_{MREQ2}, and so on. Write down all the inequalities implied by the full handshake.

30. Multicore chips, with multiple CPUs on the same die, are becoming popular. What advantages do they have over a system consisting of multiple PCs connected by Ethernet?

31. Why have multicore chips suddenly appeared? Are there technological factors that have paved the way? Does Moore's law play a role here?

32. What is the difference between the memory bus and the PCI bus?

33. Most 32-bit buses permit 16-bit reads and writes. Is there any ambiguity about where to place the data? Discuss.

34. Many CPUs have a special bus cycle type for interrupt acknowledge. Why?

35. A 64-bit computer with a 400-MHz bus requires four cycles to read a 64-bit word. How much bus bandwidth does the CPU consume in the worst case, that is, assuming back-to-back reads or writes all the time?

36. A 64-bit computer with a 400-MHz bus requires four cycles to read a 64-bit word. How much bus bandwidth does the CPU consume in the worst case, that is, assuming back-to-back reads or writes all the time?

37. A 32-bit CPU with address lines A2–A31 requires all memory references to be aligned. That is, words have to be addressed at multiples of 4 bytes, and half-words have to be addressed at even bytes. Bytes can be anywhere. How many legal combinations are there for memory reads, and how many pins are needed to express them? Give two answers and make a case for each one.

38. Modern CPU chips have one, two, or even three levels of cache on chip. Why are multiple levels of cache needed?

39. Suppose that a CPU has a level 1 cache and a level 2 cache, with access times of 1 nsec and 2 nsec, respectively. The main memory access time is 10 nsec. If 20% of the accesses are level 1 cache hits and 60% are level 2 cache hits, what is the average access time?

40. Calculate the bus bandwidth needed to display 1280×960) color video at 30 frames/sec. Assume that the data must pass over the bus twice, once from the CD-ROM to the memory and once from the memory to the screen.

41. Which of the signals of Fig. 3-55 is not strictly necessary for the bus protocol to work?

42. A PCI Express system has 10 Mbps links (gross capacity). How many signal wires are needed in each direction for 16x operation? What is the gross capacity each way? What is the net capacity each way?

43. A computer has instructions that each require two bus cycles, one to fetch the instruction and one to fetch the data. Each bus cycle takes 10 nsec and each instruction takes 20 nsec (i.e., the internal processing time is negligible). The computer also has a disk with 2048 512-byte sectors per track. Disk rotation time is 5 msec. To what percent of its normal speed is the computer reduced during a DMA transfer if each 32-bit DMA transfer takes one bus cycle?

44. The maximum payload of an isochronous data packet on the USB bus is 1023 bytes. Assuming that a device may send only one data packet per frame, what is the maximum bandwidth for a single isochronous device?

45. What would the effect be of adding a third input line to the NAND gate selecting the PIO of Fig. 3-61(b) if this new line were connected to A13?

46. Write a program to simulate the behavior of an $m \times n$ array of two-input NAND gates. This circuit, contained on a chip, has j input pins and k output pins. The values of j, k, m, and n are compile-time parameters of the simulation. The program should start off by reading in a "wiring list," each wire of which specifies an input and an output. An input is either one of the j input pins or the output of some NAND gate. An output is either one of the k output pins or an input to some NAND gate. Unused inputs are logical 1. After reading in the wiring list, the program should print the output for each of the 2^j possible inputs. Gate array chips like this one are widely used for putting custom circuits on a chip because most of the work (depositing the gate array on the chip) is independent of the circuit to be implemented. Only the wiring is specific to each design.

47. Write a program in your favorite programming language to read in two arbitrary Boolean expressions and see if they represent the same function. The input language should include single letters, as Boolean variables, the operands AND, OR, and NOT, and parentheses. Each expression should fit on one input line. The program should compute the truth tables for both functions and compare them.

This page intentionally left blank

4

THE MICROARCHITECTURE LEVEL

Above the digital logic level is the microarchitecture level. Its job is to implement the ISA (Instruction Set Architecture) level above it, as illustrated in Fig. 1-2. Its design depends on the ISA being implemented, as well as the cost and performance goals of the computer. Many modern ISAs, particularly RISC designs, have simple instructions that can usually be executed in a single clock cycle. More complex ISAs, such as the Core i7 instruction set, may require many cycles to execute a single instruction. Executing an instruction may require locating the operands in memory, reading them, and storing results back into memory. The sequencing of operations within a single instruction often leads to a different approach to control than that for simple ISAs.

4.1 AN EXAMPLE MICROARCHITECTURE

Ideally, we would like to introduce this subject by explaining the general principles of microarchitecture design. Unfortunately, there are no general principles; every ISA is a special case. Consequently, we will discuss a detailed example instead. For our example ISA, we have chosen a subset of the Java Virtual Machine. This subset contains only integer instructions, so we have named it **IJVM** to emphasize it deals only with integers.

We will start out by describing the microarchitecture on top of which we will implement IJVM. IJVM has some complex instructions. Many such architectures

have been implemented through microprogramming, as discussed in Chap. 1. Although IJVM is small, it is a good starting point for describing the control and sequencing of instructions.

Our microarchitecture will contain a microprogram (in ROM), whose job is to fetch, decode, and execute IJVM instructions. We cannot use the Oracle JVM interpreter because we need a tiny microprogram that drives the individual gates in the actual hardware efficiently. In contrast, the Oracle JVM interpreter was written in C for portability and cannot control the hardware at all.

Since the actual hardware used consists only of the basic components described in Chap. 3, in theory, after fully understanding this chapter, the reader should be able to go out and buy a large bag full of transistors and build this subset of the JVM machine. Students who successfully accomplish this task will be given extra credit (and a complete psychiatric examination).

As a convenient model for the design of the microarchitecture we can think of it as a programming problem, where each instruction at the ISA level is a function to be called by a master program. In this model, the master program is a simple, endless loop that determines a function to be invoked, calls the function, then starts over, very much like Fig. 2-3.

The microprogram has a set of variables, called the **state** of the computer, which can be accessed by all the functions. Each function changes at least some of the variables making up the state. For example, the Program Counter (PC) is part of the state. It indicates the memory location containing the next function (i.e., ISA instruction) to be executed. During the execution of each instruction, the PC is advanced to point to the next instruction to be executed.

IJVM instructions are short and sweet. Each instruction has a few fields, usually one or two, each of which has some specific purpose. The first field of every instruction is the **opcode** (short for **operation code**), which identifies the instruction, telling whether it is an ADD or a BRANCH, or something else. Many instructions have an additional field, which specifies the operand. For example, instructions that access a local variable need a field to tell *which* variable.

This model of execution, sometimes called the **fetch-decode-execute cycle**, is useful in the abstract and may also be the basis for implementation for ISAs like IJVM that have complex instructions. Below we will describe how it works, what the microarchitecture looks like, and how it is controlled by the microinstructions, each of which controls the data path for one cycle. Together, the list of microinstructions forms the microprogram, which we will present and discuss in detail.

4.1.1 The Data Path

The **data path** is that part of the CPU containing the ALU, its inputs, and its outputs. The data path of our example microarchitecture is shown in Fig. 4-1. While it has been carefully optimized for interpreting IJVM programs, it is fairly similar to the data path used in most machines. It contains a number of 32-bit

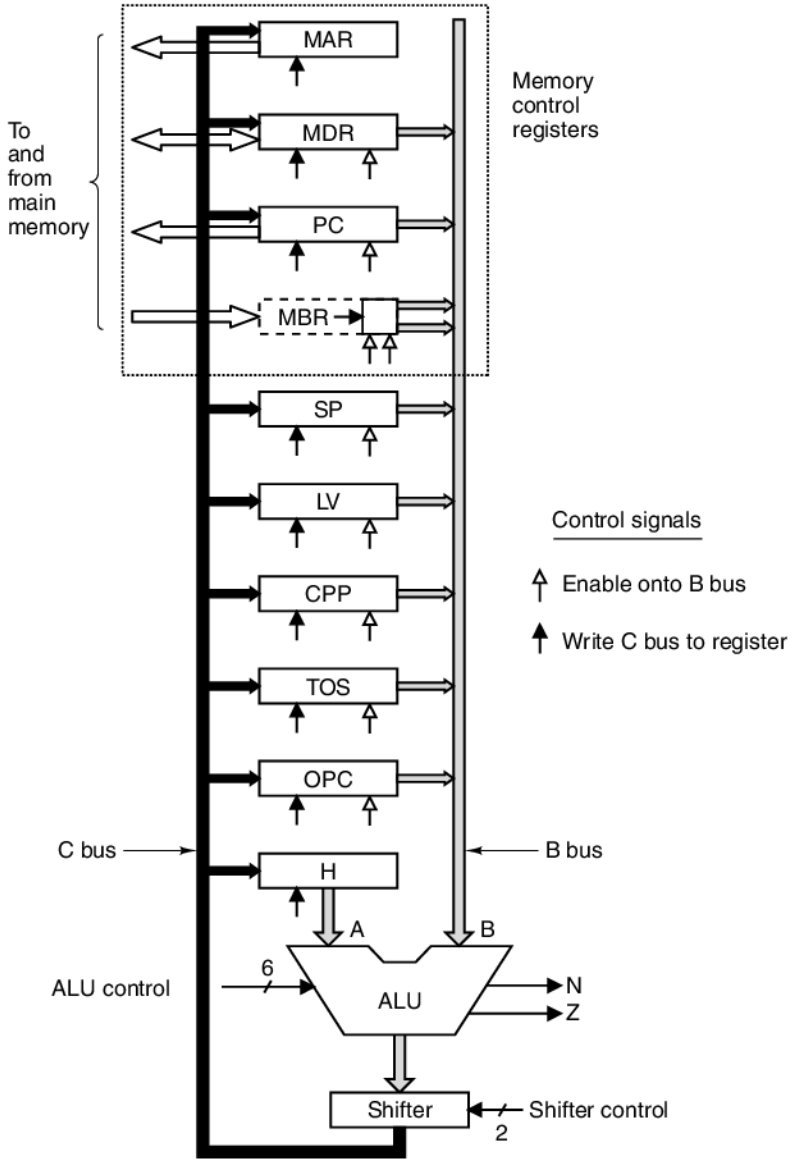


Figure 4-1. The data path of the example microarchitecture used in this chapter.

registers, to which we have assigned symbolic names such as PC, SP, and MDR. Though some of these names are familiar, it is important to understand that these registers are accessible only at the microarchitecture level (by the microprogram). They are given these names because they usually hold a value corresponding to the variable of the same name in the ISA level architecture. Most registers can drive

their contents onto the B bus. The output of the ALU drives the shifter and then the C bus, whose value can be written into one or more registers at the same time. There is no A bus for the moment; we will add one later.

The ALU is identical to the one shown in Figs. 3-18 and 3-19. Its function is determined by six control lines. The short diagonal line labeled "6" in Fig. 4-1 indicates that there are six ALU control lines. These are F_0 and F_1 for determining the ALU operation, ENA and ENB for individually enabling the inputs, INVA for inverting the left input, and INC for forcing a carry into the low-order bit, effectively adding 1 to the result. However, not all 64 combinations of ALU control lines do something useful.

Some of the more interesting combinations are shown in Fig. 4-2. Not all of these functions are needed for IJVM, but for the full JVM many of them would come in handy. In many cases, there are multiple possibilities for achieving the same result. In this table, + means arithmetic plus and − means arithmetic minus, so, for example, $-A$ means the two's complement of A.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

Figure 4-2. Useful combinations of ALU signals and the function performed.

The ALU of Fig. 4-1 needs two data inputs: a left input (A) and a right input (B). Attached to the left input is a holding register, H. Attached to the right input is the B bus, which can be loaded from any one of nine sources, indicated by the nine gray arrows touching it. An alternative design, with two full buses, has a different set of trade-offs and will be discussed later in this chapter.

H can be loaded by choosing an ALU function that just passes the right input (from the B bus) through to the ALU output. One such function is adding the ALU inputs, only with ENA negated so the left input is forced to zero. Adding zero to the value on the B bus just yields the value on the B bus. This result can then be passed through the shifter unmodified and stored in H.

In addition to the above functions, two other control lines can be used independently to control the output from the ALU. SLL8 (Shift Left Logical) shifts the contents left by 1 byte, filling the 8 least significant bits with zeros. SRA1 (Shift Right Arithmetic) shifts the contents right by 1 bit, leaving the most significant bit unchanged.

It is explicitly possible to read and write the same register on one cycle. For example, it is allowed to put SP onto the B bus, disable the ALU's left input, enable the INC signal, and store the result in SP, thus incrementing SP by 1 (see the eighth line in Fig. 4-2). How can a register be read and written on the same cycle without producing garbage? The solution is that reading and writing are actually performed at different times within the cycle. When a register is selected as the ALU's right input, its value is put onto the B bus early in the cycle and kept there continuously throughout the entire cycle. The ALU then does its work, producing a result that passes through the shifter onto the C bus. Near the end of the cycle, when the ALU and shifter outputs are known to be stable, a clock signal triggers the store of the contents of the C bus into one or more of the registers. One of these registers may well be the one that supplied the B bus with its input. The precise timing of the data path makes it possible to read and write the same register on one cycle, as described below.

Data Path Timing

The timing of these events is shown in Fig. 4-3. Here a short pulse is produced at the start of each clock cycle. It can be derived from the main clock, as shown in Fig. 3-20(c). On the falling edge of the pulse, the bits that will drive all the gates are set up. This takes a finite and known time, Δw. Then the register needed on the B bus is selected and driven onto the B bus. It takes Δx before the value is stable. Then the ALU and shifter, which as combinational circuits have been running continuously, finally have valid data to operate on. After another Δy, the ALU and shifter outputs are stable. After an additional Δz, the results have propagated along the C bus to the registers, where they can be loaded on the rising edge of the next pulse. The load should be edge triggered and fast, so that even if some of the input registers are changed, the effects will not be felt on the C bus until long after the registers have been loaded. Also on the rising edge of the pulse, the register driving the B bus stops doing so, in preparation for the next cycle. MPC, MIR, and the memory are mentioned in the figure; their roles will be discussed shortly.

It is important to realize that even though there are no storage elements in the data path, there is a finite propagation time through it. Changing the value on the

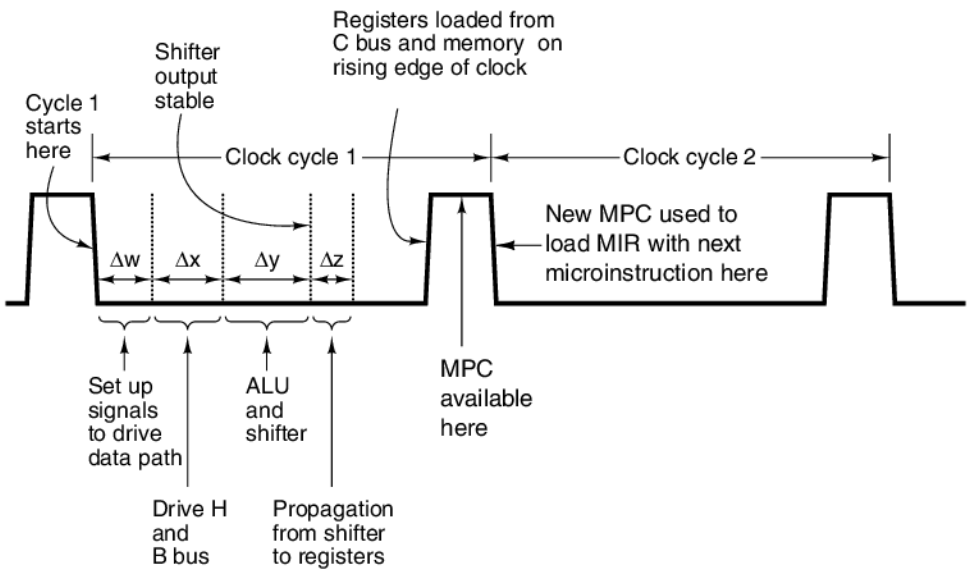


Figure 4-3. Timing diagram of one data path cycle.

B bus does not cause the C bus to change until a finite time later (due to the finite delays of each step). Consequently, even if a store changes one of the input registers, the value will be safely tucked away in the register long before the (now-incorrect) value being put on the B bus (or H) can reach the ALU.

Making this design work requires rigid timing, a long clock cycle, a known minimum propagation time through the ALU, and a fast load of the registers from the C bus. However, with careful engineering, the data path can be designed so that it functions correctly all the time. Actual machines work this way.

A somewhat different way to look at the data path cycle is to think of it as broken up into implicit subcycles. The start of subcycle 1 is triggered by the falling edge of the clock. The activities that go on during the subcycles are shown below along with the subcycle lengths (in parentheses).

1. The control signals are set up (Δw).
2. The registers are loaded onto the B bus (Δx).
3. The ALU and shifter operate (Δy).
4. The results propagate along the C bus back to the registers (Δz).

The time interval after Δz provides some tolerance since the times are not exact. At the rising edge of the next clock cycle, the results are stored in the registers.

We said that the subcycles can be best thought of as being *implicit*. By this we mean there are no clock pulses or other explicit signals to tell the ALU when to operate or tell the results to enter the C bus. In reality, the ALU and shifter run all the time. However, their inputs are garbage until a time $\Delta w + \Delta x$ after the falling edge of the clock. Likewise, their outputs are garbage until $\Delta w + \Delta x + \Delta y$ has elapsed after the falling edge of the clock. The only explicit signals that drive the data path are the falling edge of the clock, which starts the data path cycle, and the rising edge of the clock, which loads the registers from the C bus. The other subcycle boundaries are implicitly determined by the inherent propagation times of the circuits involved. It is the design engineers' responsibility to make sure that the time $\Delta w + \Delta x + \Delta y + \Delta z$ comes sufficiently in advance of the rising edge of the clock to have the register loads work reliably all the time.

Memory Operation

Our machine has two different ways to communicate with memory: a 32-bit, word-addressable memory port and an 8-bit, byte-addressable memory port. The 32-bit port is controlled by two registers, MAR (**Memory Address Register**) and MDR (**Memory Data Register**), as shown in Fig. 4-1. The 8-bit port is controlled by one register, PC, which reads 1 byte into the low-order 8 bits of MBR. This port can only read data from memory; it cannot write data to memory.

Each of these registers (and every other register in Fig. 4-1) is driven by one or two **control signals**. An open arrow under a register indicates a control signal that enables the register's output onto the B bus. Since MAR does not have a connection to the B bus, it does not have an enable signal. H does not have one either because, being the only possible left ALU input, it is always enabled.

A solid black arrow under a register indicates a control signal that writes (i.e., loads) the register from the C bus. Since MBR cannot be loaded from the C bus, it does not have a write signal (although it does have two other enable signals, described below). To initiate a memory read or write, the appropriate memory registers must be loaded, then a read or write signal issued to the memory (not shown in Fig. 4-1).

MAR contains *word* addresses, so that the values 0, 1, 2, etc. refer to consecutive words. PC contains *byte* addresses, so that the values 0, 1, 2, etc. refer to consecutive bytes. Thus putting a 2 in PC and starting a memory read will read out byte 2 from memory and put it in the low-order 8 bits of MBR. Putting a 2 in MAR and starting a memory read will read out bytes 8–11 (i.e., word 2) from memory and put them in MDR.

This difference in functionality is needed because MAR and PC will be used to reference two different parts of memory. The need for this distinction will become clearer later. For the moment, suffice it to say that the MAR/MDR combination is used to read and write ISA-level data words and the PC/MBR combination is used

to read the executable ISA-level program, which consists of a byte stream. All other registers that contain addresses use word addresses, like MAR.

In the actual physical implementation, there is only one real memory and it is byte oriented. Allowing MAR to count in words (needed due to the way JVM is defined) while the physical memory counts in bytes is handled by a simple trick. When MAR is placed on the address bus, its 32 bits do not map onto the 32 address lines, 0–31, directly. Instead MAR bit 0 is wired to address bus line 2, MAR bit 1 to address bus line 3, and so on. The upper 2 bits of MAR are discarded since they are needed only for word addresses above 2^{32}, none of which are legal for our 4-GB machine. Using this mapping, when MAR is 1, address 4 is put onto the bus; when MAR is 2, address 8 is put onto the bus, and so forth. This trick is illustrated in Fig. 4-4.

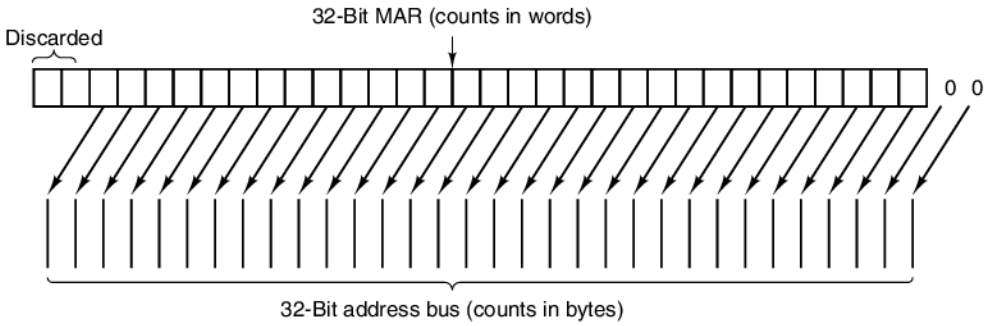


Figure 4-4. Mapping of the bits in MAR to the address bus.

As mentioned above, data read from memory through the 8-bit memory port are returned in MBR, an 8-bit register. MBR can be gated (i.e., copied) onto the B bus in one of two ways: unsigned or signed. When the unsigned value is needed, the 32-bit word put onto the B bus contains the MBR value in the low-order 8 bits and zeros in the upper 24 bits. Unsigned values are useful for indexing into a table, or when a 16-bit integer has to be assembled from 2 consecutive (unsigned) bytes in the instruction stream.

The other option for converting the 8-bit MBR to a 32-bit word is to treat it as a signed value between −128 and +127 and use this value to generate a 32-bit word with the same numerical value. This conversion is done by duplicating the MBR sign bit (leftmost bit) into the upper 24 bit positions of the B bus, a process known as **sign extension**. When this option is chosen, the upper 24 bits will either be all 0s or all 1s, depending on whether the leftmost bit of the 8-bit MBR is a 0 or a 1.

The choice of whether the 8-bit MBR is converted to an unsigned or a signed 32-bit value on the B bus is determined by which of the two control signals (open arrows below MBR in Fig. 4-1) is asserted. The need for these two options is why two arrows are present. The ability to have the 8-bit MBR act like a 32-bit source to the B bus is indicated by the dashed box to the left of MBR in the figure.

4.1.2 Microinstructions

To control the data path of Fig. 4-1, we need 29 signals. These can be divided into five functional groups, as described below.

9 Signals to control writing data from the C bus into registers.

9 Signals to control enabling registers onto the B bus for ALU input.

8 Signals to control the ALU and shifter functions.

2 Signals (not shown) to indicate memory read/write via MAR/MDR.

1 Signal (not shown) to indicate memory fetch via PC/MBR.

The values of these 29 control signals specify the operations for one cycle of the data path. A cycle consists of gating values out of registers and onto the B bus, propagating the signals through the ALU and shifter, driving them onto the C bus, and finally writing the results in the appropriate register or registers. In addition, if a memory read data signal is asserted, the memory operation is started at the end of the data path cycle, after MAR has been loaded. The memory data are available at the very end of the *following* cycle in MBR or MDR and can be used in the cycle *after that*. In other words, a memory read on either port initiated at the end of cycle k delivers data that cannot be used in cycle $k+1$, but only in cycle $k+2$ or later.

This seemingly counterintuitive behavior is explained by Fig. 4-3. The memory control signals are not generated in clock cycle 1 until just after MAR and PC are loaded at the rising edge of the clock, toward the end of clock cycle 1. We will assume the memory puts its results on the memory buses within one cycle so that MBR and/or MDR can be loaded on the next rising clock edge, along with the other registers.

Put in other words, we load MAR at the end of a data path cycle and start the memory shortly thereafter. Consequently, we cannot really expect the results of a read operation to be in MDR at the start of the next cycle, especially if the clock pulse is narrow. There is just not enough time if the memory takes one clock cycle. One data path cycle must intervene between starting a memory read and using the result. Of course, other operations can be performed during that cycle, just not those that need the memory word.

The assumption that the memory takes one cycle to operate is equivalent to assuming that the level 1 cache hit rate is 100%. This assumption is never true, but the complexity introduced by a variable-length memory cycle time is more than we want to deal with here.

Since MBR and MDR are loaded on the rising edge of the clock, along with all the other registers, they may be read during cycles when a new memory read is being performed. They return the old values, since the read has not yet had time to overwrite them. There is no ambiguity here; until new values are loaded into MBR

and MDR at the rising edge of the clock, the previous values are still there and usable. Note that it is possible to perform back-to-back reads on two consecutive cycles, since a read takes only 1 cycle. Also, both memories may operate at the same time. However, trying to read and write the same byte simultaneously gives undefined results.

While it may be desirable to write the output on the C bus into more than one register, it is never desirable to enable more than one register onto the B bus at a time. (In fact, some real implementations will suffer physical damage if this is done.) With a small increase in circuitry, we can reduce the number of bits needed to select among the possible sources for driving the B bus. There are only nine possible input registers that can drive the B bus (where the signed and unsigned versions of MBR each count separately). Therefore, we can encode the B bus information in 4 bits and use a decoder to generate the 16 control signals, 7 of which are not needed. In a commercial design, the architects would experience an overwhelming urge to get rid of one of the registers so that 3 bits would do the job. As academics, we have the enormous luxury of being able to waste 1 bit to give a cleaner and simpler design.

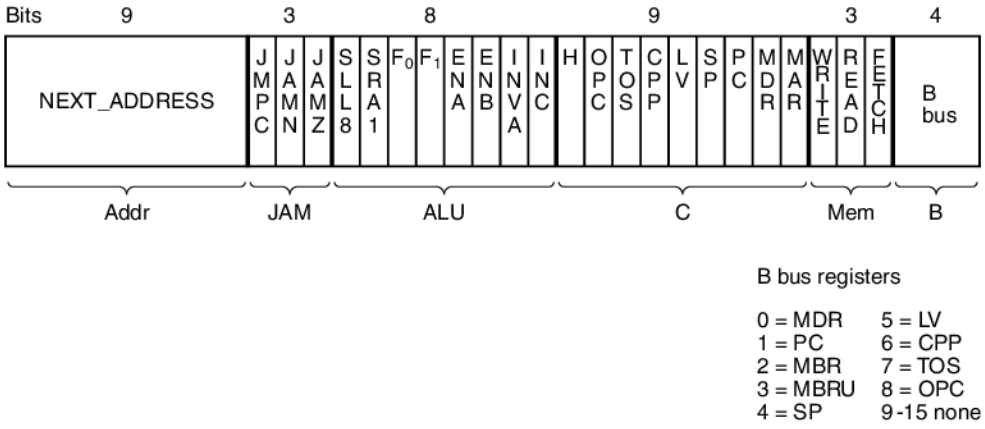


Figure 4-5. The microinstruction format for the Mic-1 (to be described shortly).

At this point we can control the data path with 9 + 4 + 8 + 2 + 1 = 24 signals, hence 24 bits. However, these 24 bits control the data path for only one cycle. The second part of the control is to determine what is done on the following cycle. To include this in the design of the controller, we will create a format for describing the operations to be performed using the 24 control bits plus two additional fields: NEXT_ADDRESS and JAM. Their contents will be discussed shortly. Figure 4-5 shows a possible format, divided into the six groups (listed below the instruction) and containing the following 36 signals:

Addr – Contains the address of a potential next microinstruction.

JAM – Determines how the next microinstruction is selected.

ALU – ALU and shifter functions.

C – Selects which registers are written from the C bus.

Mem – Memory functions.

B – Selects the B bus source; it is encoded as shown.

The ordering of the groups is, in principle, arbitrary, although we have actually chosen it very carefully to minimize line crossings in Fig. 4-6. Line crossings in schematic diagrams like Fig. 4-6 often correspond to wire crossings on chips, which cause trouble in two-dimensional designs and are best minimized.

4.1.3 Microinstruction Control: The Mic-1

So far we have described how the data path is controlled, but we have not yet described how it is decided which control signals should be enabled on each cycle. This is determined by a **sequencer** that is responsible for stepping through the sequence of operations for the execution of a single ISA instruction.

The sequencer must produce two kinds of information each cycle:

1. The state of every control signal in the system.
2. The address of the microinstruction that is to be executed next.

Figure 4-6 is a detailed block diagram of the complete microarchitecture of our example machine, which we will call the **Mic-1**. It may look intimidating initially but is worth studying carefully. When you fully understand every box and every line in this figure, you will be well on your way to understanding the microarchitecture level. The block diagram has two parts: the data path, on the left, which we have already discussed in detail, and the control section, on the right, which we will now look at.

The largest and most important item in the control portion of the machine is a memory called the **control store**. It is convenient to think of it as a memory that holds the complete microprogram, although it is sometimes implemented as a set of logic gates. In general, we will refer to it as the control store, to avoid confusion with the main memory, accessed through MBR and MDR. Functionally, however, the control store is a memory that simply holds microinstructions instead of ISA instructions. For our example machine, it contains 512 words, each consisting of one 36-bit microinstruction of the kind illustrated in Fig. 4-5. Actually, not all of these words are needed, but (for reasons to be explained shortly) we need addresses for 512 distinct words.

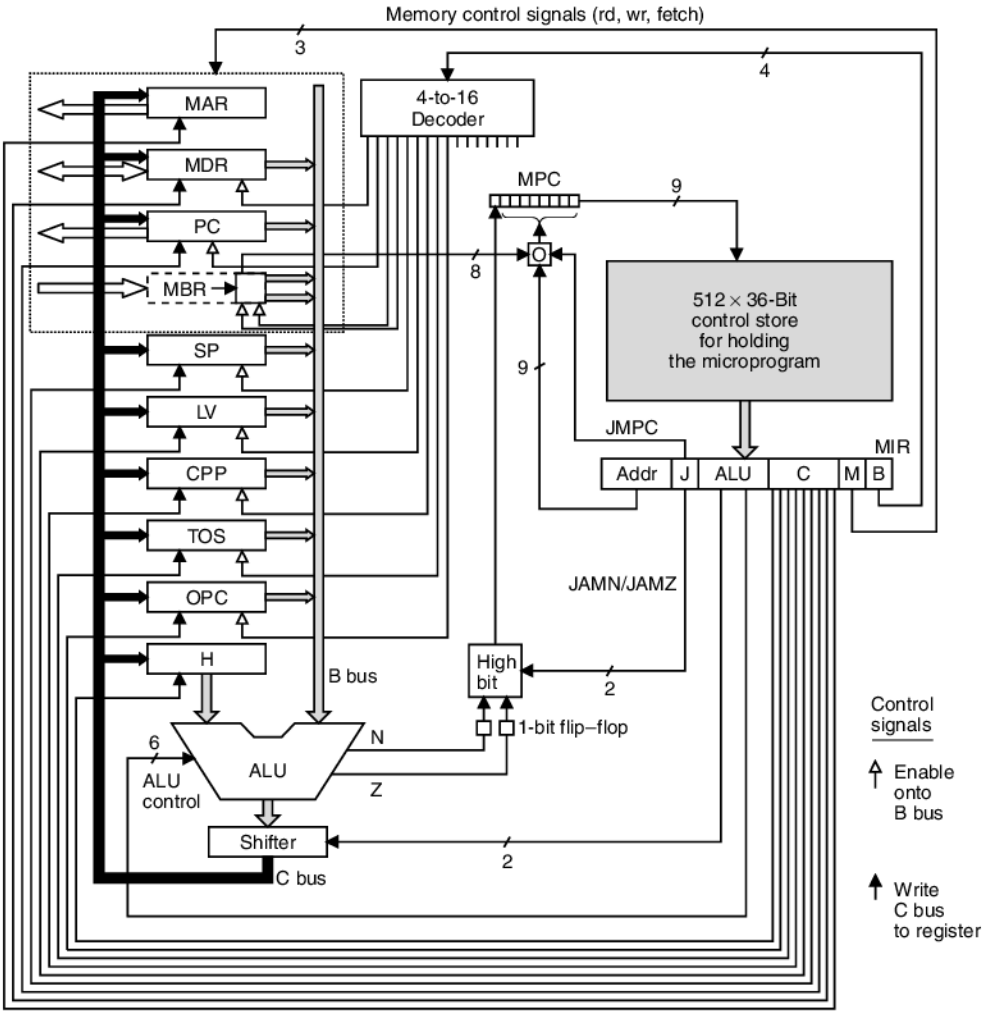


Figure 4-6. The complete block diagram of our example microarchitecture, the Mic-1.

In one important way, the control store is quite different from the main memory: instructions in main memory are always executed in address order (except for branches); microinstructions are not. The act of incrementing the program counter in Fig. 2-3 reflects the fact that the default instruction to execute after the current one is the instruction following the current one in memory. Microprograms need more flexibility (because microinstruction sequences tend to be short), so they usually do not have this property. Instead, each microinstruction explicitly specifies its successor.

Since the control store is functionally a (read-only) memory, it needs its own memory address register and memory data register. It does not need read and write

signals, because it is continuously being read. We will call the control store's memory address register **MPC** (**MicroProgram Counter**). This name is ironic since the locations in it are explicitly not ordered, so the concept of counting is not useful (but who are we to argue with tradition?). The memory data register is called **MIR** (**MicroInstruction Register**). Its function is to hold the current microinstruction, whose bits drive the control signals that operate the data path.

The MIR register in Fig. 4-6 holds the same six groups as in Fig. 4-5. The Addr and J (for JAM) groups control the selection of the next microinstruction and will be discussed shortly. The ALU group contains the 8 bits that select the ALU function and drive the shifter. The C bits cause individual registers to load the ALU output from the C bus. The M bits control memory operations.

Finally, the last 4 bits drive the decoder that determines what goes onto the B bus. In this case we have chosen to use a standard 4-to-16 decoder, even though only nine possibilities are required. In a more finely tuned design, a 4-to-9 decoder could be used. The trade-off here is using a standard circuit taken from a library of circuits versus designing a custom one. Using the standard circuit is simpler and is unlikely to introduce any bugs. Rolling your own uses less chip area but takes longer to design and you might get it wrong.

The operation of Fig. 4-6 is as follows. At the start of each clock cycle (the falling edge of the clock in Fig. 4-3), MIR is loaded from the word in the control store pointed to by MPC. The MIR load time is indicated in the figure by Δw. If one thinks in terms of subcycles, MIR is loaded during the first one.

Once the microinstruction is set up in MIR, the various signals propagate out into the data path. A register is put out onto the B bus, the ALU knows which operation to perform, and there is lots of activity out there. This is the second subcycle. After an interval $\Delta w + \Delta x$ from the start of the cycle, the ALU inputs are stable.

Another Δy later, everything has settled down and the ALU, N, Z, and shifter outputs are stable. The N and Z values are then saved in a pair of 1-bit flip-flops. These bits, like all the registers that are loaded from the C bus and from memory, are saved on the rising edge of the clock, near the end of the data path cycle. The ALU output is not latched but just fed into the shifter. The ALU and shifter activity occurs during subcycle 3.

After an additional interval, Δz, the shifter output has reached the registers via the C bus. Then the registers can be loaded near the end of the cycle (at the rising edge of the clock pulse in Fig. 4-3). Subcycle 4 consists of loading the registers and N and Z flip-flops. It terminates a little after the rising edge of the clock, when all the results have been saved and the results of the previous memory operations are available and MPC has been loaded. This process goes on and on until somebody gets bored with it and turns the machine off.

In parallel with driving the data path, the microprogram also has to determine which microinstruction to execute next, as they need not be executed in the order they happen to appear in the control store. The calculation of the address of the

next microinstruction begins after MIR has been loaded and is stable. First, the NEXT_ADDRESS field is copied to MPC. While this copy is taking place, the JAM field is inspected. If it has the value 000, nothing else is done; when the copy of NEXT_ADDRESS completes, MPC will point to the next microinstruction.

If one or more of the JAM bits are 1, more work is needed. If JAMN is set, the 1-bit N flip-flop is ORed into the high-order bit of MPC. Similarly, if JAMZ is set, the 1-bit Z flip-flop is ORed there. If both are set, both are ORed there. The reason that the N and Z flip-flops are needed is that after the rising edge of the clock (while the clock is high), the B bus is no longer being driven, so the ALU outputs can no longer be assumed to be correct. Saving the ALU status flags in N and Z makes the correct values available and stable for the MPC computation, no matter what is going on around the ALU.

In Fig. 4-6, the logic that does this computation is labeled "High bit." The Boolean function it computes is

$$F = (\text{JAMZ AND Z}) \text{ OR } (\text{JAMN AND N}) \text{ OR } \text{NEXT\_ADDRESS}[8]$$

Note that in all cases, MPC can take on only one of two possible values:

1. The value of NEXT_ADDRESS.
2. The value of NEXT_ADDRESS with the high-order bit ORed with 1.

No other possibilities make sense. If the high-order bit of NEXT_ADDRESS was already 1, using JAMN or JAMZ makes no sense.

Note that when the JAM bits are all zeros, the address of the next microinstruction to be executed is simply the 9-bit number in the NEXT_ADDRESS field. When either JAMN or JAMZ is 1, there are two potential successors: NEXT_ADDRESS and NEXT_ADDRESS ORed with 0x100 (assuming that NEXT_ADDRESS $\leq$ 0xFF). (Note that 0x indicates that the number following it is in hexadecimal.) This point is illustrated in Fig. 4-7. The current microinstruction, at location 0x75, has NEXT_ADDRESS = 0x92 and JAMZ set to 1. Consequently, the next address of the microinstruction depends on the Z bit stored on the previous ALU operation. If the Z bit is 0, the next microinstruction comes from 0x92. If the Z bit is 1, the next microinstruction comes from 0x192.

The third bit in the JAM field is JMPC. If it is set, the 8 MBR bits are bitwise ORed with the 8 low-order bits of the NEXT_ADDRESS field coming from the current microinstruction. The result is sent to MPC. The box with the label "O" in Fig. 4-6 does an OR of MBR with NEXT_ADDRESS if JMPC is 1 but just passes NEXT_ADDRESS through to MPC if JMPC is 0. When JMPC is 1, the low-order 8 bits of NEXT_ADDRESS are normally zero. The high-order bit can be 0 or 1, so the NEXT_ADDRESS value used with JMPC is normally 0x000 or 0x100. The reason for sometimes using 0x000 and sometimes using 0x100 will be discussed later.

The ability to OR MBR together with NEXT_ADDRESS and store the result in MPC allows an efficient implementation of a multiway branch (jump). Notice that

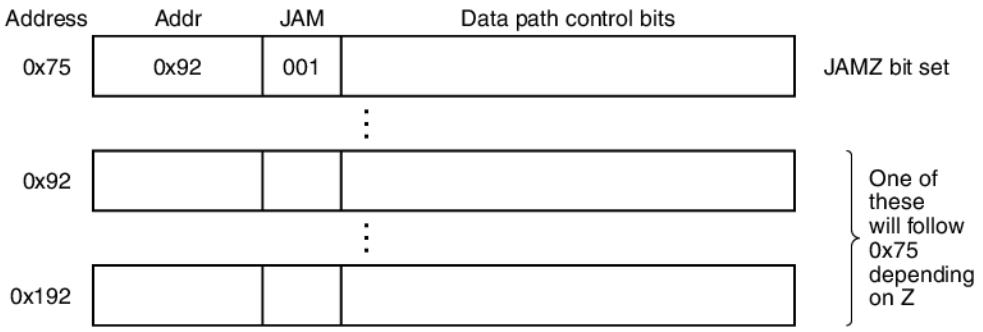


Figure 4-7. A microinstruction with JAMZ set to 1 has two potential successors.

any of 256 addresses can be specified, determined solely by the bits present in MBR. In a typical use, MBR contains an opcode, so the use of JMPC will result in a unique selection for the next microinstruction to be executed for every possible opcode. This method is useful for quickly branching directly to the function corresponding to the just-fetched opcode.

Understanding the timing of the machine is critical to what will follow, so it is perhaps worth repeating. We will do it in terms of subcycles, since this is easy to visualize, but the only real clock events are the falling edge, which starts the cycle, and the rising edge, which loads the registers and the N and Z flip-flops. Please refer to Fig. 4-3 once more.

During subcycle 1, initiated by the falling edge of the clock, MIR is loaded from the address currently held in MPC. During subcycle 2, the signals from MIR propagate out and the B bus is loaded from the selected register. During subcycle 3, the ALU and shifter operate and produce a stable result. During subcycle 4, the C bus, memory buses, and ALU values become stable. At the rising edge of the clock, the registers are loaded from the C bus, N and Z flip-flops are loaded, and MBR and MDR get their results from the memory operation started at the end of the previous data path cycle (if any). As soon as MBR is available, MPC is loaded in preparation for the next microinstruction. Thus MPC gets its value sometime during the middle of the interval when the clock is high but after MBR/MDR are ready. It could be either level triggered (rather than edge triggered), or edge trigger a fixed delay after the rising edge of the clock. All that matters is that MPC is not loaded until the registers it depends on (MBR, N, and Z) are ready. As soon as the clock falls, MPC can address the control store and a new cycle can begin.

Note that each cycle is self contained. It specifies what goes onto the B bus, what the ALU and shifter are to do, where the C bus is to be stored, and finally, what the next MPC value should be.

One final note about Fig. 4-6 is worth making. We have been treating MPC as a proper register, with 9 bits of storage capacity that is loaded while the clock is high. In reality, there is no need to have a register there at all. All of its inputs can be fed directly through, right to the control store. As long as they are present at the

control store at the falling edge of the clock when MIR is selected and read out, that is sufficient. There is no need to actually store them in MPC. For this reason, MPC might well be implemented as a **virtual register**, which is just a gathering place for signals, more like an electronic patch panel, than a real register. Making MPC a virtual register simplifies the timing: now events happen only on the falling and rising edges of the clock and nowhere else. But if it is easier for you to think of MPC as a real register, that is also a valid viewpoint.

4.2 AN EXAMPLE ISA: IJVM

Let us continue our example by introducing the ISA level of the machine to be interpreted by the microprogram running on the microarchitecture of Fig. 4-6 (IJVM). For convenience, we will sometimes refer to the Instruction Set Architecture as the **macroarchitecture**, to contrast it with the microarchitecture. Before we describe IJVM, however, we will digress slightly to motivate it.

4.2.1 Stacks

Virtually all programming languages support the concept of procedures (methods), which have local variables. These variables can be accessed from inside the procedure but cease to be accessible once the procedure has returned. The question thus arises: "Where should these variables be kept in memory?"

The simplest solution, to give each variable an absolute memory address, does not work. The problem is that a procedure may call itself. We will study these recursive procedures in Chap. 5. For the moment, suffice it to say that if a procedure is active (i.e., called) twice, its variables cannot be stored in absolute memory locations because the second invocation will interfere with the first.

Instead, a different strategy is used. An area of memory, called the **stack**, is reserved for variables, but individual variables do not get absolute addresses in it. Instead, a register, say, LV, is set to point to the base of the local variables for the current procedure. In Fig. 4-8(a), a procedure *A*, which has local variables *a1*, *a2*, and *a3*, has been called, so storage for its local variables has been reserved starting at the memory location pointed to by LV. Another register, SP, points to the highest word of *A*'s local variables. If LV is 100 and words are 4 bytes, then SP will be 108. Variables are referred to by giving their offset (distance) from LV. The data structure between LV and SP (and including both words pointed to) is called *A*'s **local variable frame**.

Now let us consider what happens if *A* calls another procedure, *B*. Where should *B*'s four local variables (*b1, b2, b3, b4*) be stored? *Answer*: On the stack, on top of *A*'s, as shown in Fig. 4-8(b). Notice that LV has been adjusted by the procedure call to point to *B*'s local variables instead of *A*'s. We can refer to *B*'s local variables by giving their offset from LV. Similarly, if *B* calls *C*, LV and SP are adjusted again to allocate space for *C*'s two variables, as shown in Fig. 4-8(c).

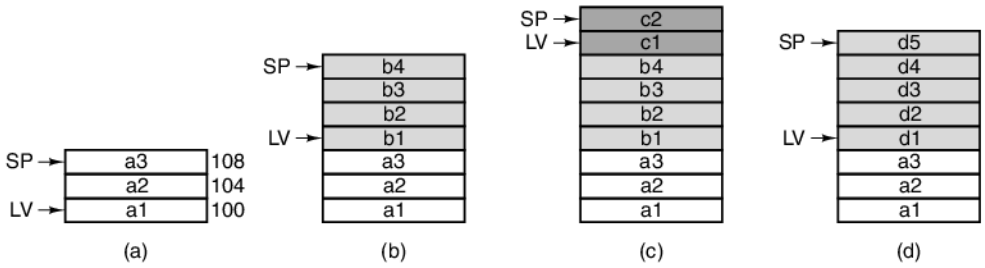


Figure 4-8. Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

When *C* returns, *B* becomes active again, and the stack is adjusted back to Fig. 4-8(b) so that LV now points to *B*'s local variables again. Likewise, when *B* returns, we get back to the situation of Fig. 4-8(a). Under all conditions, LV points to the base of the stack frame for the currently active procedure, and SP points to the top of the stack frame.

Now suppose that *A* calls *D*, which has five local variables. We get the situation of Fig. 4-8(d), in which *D*'s local variables use the same memory that *B*'s did, as well as part of *C*'s. With this memory organization, memory is allocated only for procedures that are currently active. When a procedure returns, the memory used by its local variables is released.

Besides holding local variables, stacks have another use. They can hold operands during the computation of an arithmetic expression. When used this way, the stack is referred to as the **operand stack**. Suppose, for example, that before calling *B*, *A* has to do the computation

```
a1 = a2 + a3;
```

One way of doing this sum is to push *a2* onto the stack, as shown in Fig. 4-9(a). Here SP has been incremented by the number of bytes in a word, say, 4, and the first operand stored at the address now pointed to by SP. Next, *a3* is pushed onto the stack, as shown in Fig. 4-9(b). (As an aside on notation, we will typeset all program fragments in Helvetica, as above. We will also use this font for assembly-language opcodes and machine registers, but in running text, program variables and procedures will be given in *italics*. The difference is that variables and procedure names are chosen by the user; opcodes and register names are built in.)

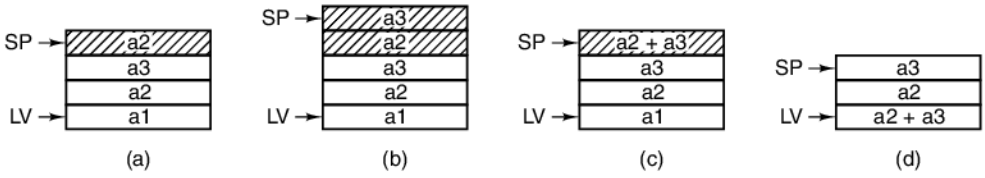


Figure 4-9. Use of an operand stack for doing an arithmetic computation.

The actual computation can now be done by executing an instruction that pops two words off the stack, adds them together, and pushes the result back onto the stack, as shown in Fig. 4-9(c). Finally, the top word can be popped off the stack and stored back in local variable *a1*, as illustrated in Fig. 4-9(d).

The local variable frames and the operand stacks can be intermixed. For example, when computing an expression like $x^2 + f(x)$, part of the expression (e.g., x^2) may be on the operand stack when a function f is called. The result of the function is left on the stack, on top of x^2, so the next instruction can add them.

It is worth noting that while all machines use a stack for storing local variables, not all use an operand stack like this for doing arithmetic. In fact, most of them do not, but JVM and IJVM work like this, which is why we have introduced stack operations here. We will study them in more detail in Chap. 5.

4.2.2 The IJVM Memory Model

We are now ready to look at the IJVM's architecture. Basically, it consists of a memory that can be viewed in either of two ways: an array of 4,294,967,296 bytes (4 GB) or an array of 1,073,741,824 words, each consisting of 4 bytes. Unlike most ISAs, the Java Virtual Machine makes no absolute memory addresses directly visible at the ISA level, but there are several implicit addresses that provide the base for a pointer. IJVM instructions can access memory only by indexing from these pointers. At any time, the following areas of memory are defined:

1. *The constant pool.* This area cannot be written by an IJVM program and consists of constants, strings, and pointers to other areas of memory that can be referenced. It is loaded when the program is brought into memory and not changed afterward. There is an implicit register, CPP, that contains the address of the first word of the constant pool.

2. *The Local variable frame.* For each invocation of a method, an area is allocated for storing variables during the lifetime of the invocation. It is called the **local variable frame**. At the beginning of this frame reside the parameters (also called arguments) with which the method was invoked. The local variable frame does not include the operand stack, which is separate. However, for efficiency reasons, our implementation chooses to implement the operand stack immediately above the local variable frame. An implicit register contains the address of the first location in the local variable frame. We will call this register LV. The parameters passed at the invocation of the method are stored at the beginning of the local variable frame.

3. *The operand stack.* The stack frame is guaranteed not to exceed a certain size, computed in advance by the Java compiler. The operand stack space is allocated directly above the local variable frame, as

illustrated in Fig. 4-10. In our implementation, it is convenient to think of the operand stack as part of the local variable frame. In any case, an implicit register contains the address of the top word of the stack. Notice that, unlike CPP and LV, this pointer, SP, changes during the execution of the method as operands are pushed onto the stack or popped from it.

4. *The method area.* Finally, there is a region of memory containing the program, referred to as the "text" area in a UNIX process. An implicit register contains the address of the instruction to be fetched next. This pointer is referred to as the Program Counter, or PC. Unlike the other regions of memory, the method area is treated as a byte array.

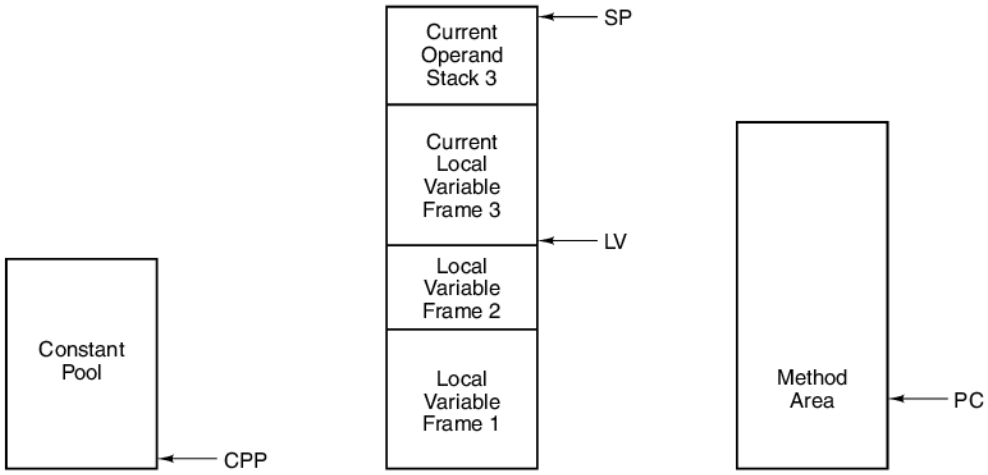


Figure 4-10. The various parts of the IJVM memory.

One point needs to be made regarding the pointers. The CPP, LV, and SP registers are all pointers to *words*, not *bytes*, and are offset by the number of words. For the integer subset we have chosen, all references to items in the constant pool, the local variables frame, and the stack are words, and all offsets used to index into these frames are word offsets. For example, LV, LV + 1, and LV + 2 refer to the first three words of the local variables frame. In contrast, LV, LV + 4, and LV + 8 refer to words at intervals of four words (16 bytes).

In contrast, PC contains a byte address, and an addition or subtraction to PC changes the address by a number of bytes, not a number of words. Addressing for PC is different from the others, and this fact is apparent in the special memory port provided for PC on the Mic-1. Remember that it is only 1 byte wide. Incrementing PC by one and initiating a read results in a fetch of the next *byte*. Incrementing SP by one and initiating a read results in a fetch of the next *word*.

4.2.3 The IJVM Instruction Set

The IJVM instruction set is shown in Fig. 4-11. Each instruction consists of an opcode and sometimes an operand, such as a memory offset or a constant. The first column gives the hexadecimal encoding of the instruction. The second gives its assembly-language mnemonic. The third gives a brief description of its effect.

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 4-11. The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

Instructions are provided to push a word from various sources onto the stack. These sources include the constant pool (LDC_W), the local variable frame (ILOAD), and the instruction itself (BIPUSH). A variable can also be popped from the stack and stored into the local variable frame (ISTORE). Two arithmetic operations (IADD and ISUB) as well as two logical (Boolean) operations (IAND and IOR) can be performed using the two top words on the stack as operands. In all the arithmetic and logical operations, two words are popped from the stack and the result pushed back onto it. Four branch instructions are provided, one unconditional (GOTO) and three conditional ones (IFEQ, IFLT, and IF_ICMPEQ). All the branch instructions, if taken, adjust the value of PC by the size of their (16-bit signed) offset, which follows the opcode in the instruction. This offset is added to the address of the opcode. There

are also IJVM instructions for swapping the top two words on the stack (SWAP), duplicating the top word (DUP), and removing it (POP).

Some instructions have multiple formats, allowing a short form for commonly used versions. In IJVM we have included two of the various mechanisms JVM uses to accomplish this. In one case we have skipped the short form in favor of the more general one. In another case we show how the prefix instruction WIDE can be used to modify the ensuing instruction.

Finally, there is an instruction (INVOKEVIRTUAL) for invoking another method, and another instruction (IRETURN) for exiting the method and returning control to the method that invoked it. Due to the complexity of the mechanism we have slightly simplified the definition, making it possible to produce a straightforward mechanism for invoking a call and return. The restriction is that, unlike Java, we allow a method to invoke only a method existing within its own object. This restriction severely cripples the object orientation but allows us to present a much simpler mechanism, by avoiding the requirement to locate the method dynamically. (If you are not familiar with object-oriented programming, you can safely ignore this remark. What we have done is turn Java back into a nonobject-oriented language, such as C or Pascal.) On all computers except JVM, the address of the procedure to call is determined directly by the CALL instruction, so our approach is actually the normal case, not the exception.

The mechanism for invoking a method is as follows. First, the callr pushes onto the stack a reference (pointer) to the object to be called. (This reference is not needed in IJVM since no other object may be specified, but it is retained for consistency with JVM.) In Fig. 4-12(a) this reference is indicated by OBJREF. Then the caller pushes the method's parameters onto the stack, in this example, *Parameter 1*, *Parameter 2*, and *Parameter 3*. Finally, INVOKEVIRTUAL is executed.

The INVOKEVIRTUAL instruction includes a displacement which indicates the position in the constant pool that contains the start address within the nethod area for the method being invoked. However, while the method code resides at the location pointed to by this pointer, the first 4 bytes in the method area contain special data. The first 2 bytes are interpreted as a 16-bit integer indicating the number of parameters for the method (the parameters themselves have previously been pushed onto the stack). For this count, OBJREF is counted as a parameter: parameter 0. This 16-bit integer, together with the value of SP, provides the location of OBJREF. Note that LV points to OBJREF rather than the first real parameter. The choice of where LV points is somewhat arbitrary.

The second 2 bytes in the method area are interpreted as another 16-bit integer indicating the size of the local variable area for the method being invoked. This is necessary because a new stack will be established for the method, beginning immediately above the local variable frame. Finally, the fifth byte in the method area contains the first opcode to be executed.

The actual sequence that occurs for INVOKEVIRTUAL is as follows and is depicted in Fig. 4-12. The two unsigned index bytes that follow the opcode are

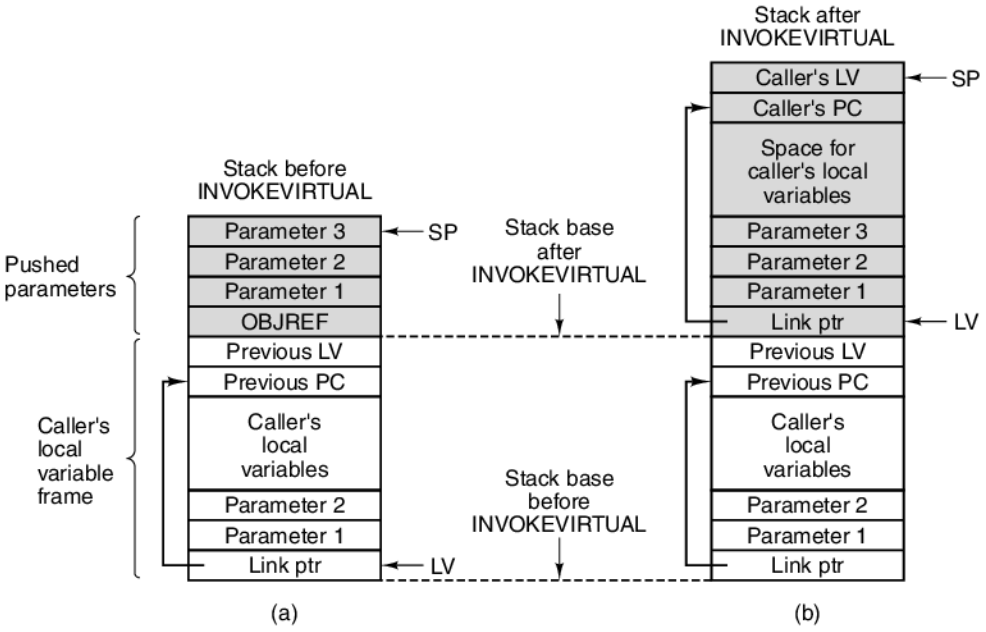


Figure 4-12. (a) Memory before executing INVOKEVIRTUAL. (b) After executing it.

used to construct an index into the constant pool table (the first byte is the high-order byte). The instruction computes the base address of the new local variable frame by subtracting off the number of parameters from the stack pointer and setting LV to point to OBJREF. At this location, overwriting OBJREF, the implementation stores the address of the location where the old PC is to be stored. This address is computed by adding the size of the local variable frame (parameters + local variables) to the address contained in LV. Immediately above the address where the old PC is to be stored is the address where the old LV is to be stored. Immediately above that address is the beginning of the stack for the newly called procedure. SP is set to point to the old LV, which is the address immediately below the first empty location on the stack. Remember that SP always points to the top word on the stack. If the stack is empty, it points to the first location below the end of the stack because our stacks grow upward, toward higher addresses. In our figures, stacks always grow upward, toward the higher address at the top of the page.

The last operation needed to carry out INVOKEVIRTUAL is to set PC to point to the fifth byte in the method code space.

The IRETURN instruction reverses the operations of the INVOKEVIRTUAL instruction, as shown in Fig. 4-13. It deallocates the space that was used by the returning method. It also restores the stack to its former state, except that (1) the (now overwritten) OBJREF word and all the parameters have been popped from the stack, and (2) the returned value has been placed at the top of the stack, at the

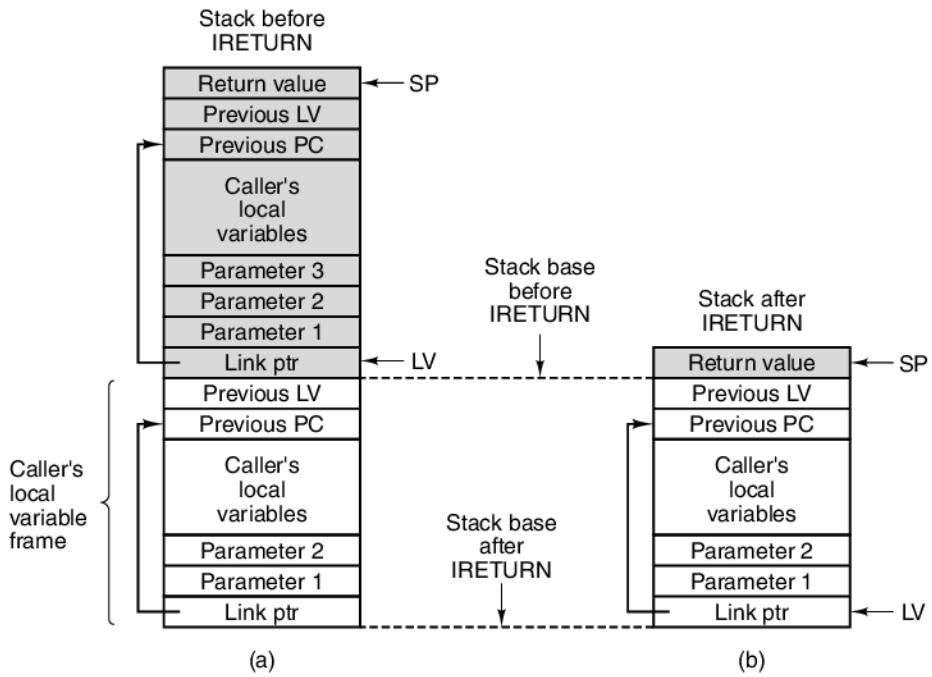


Figure 4-13. (a) Memory before executing IRETURN. (b) After executing it.

location formerly occupied by OBJREF. To restore the old state, the IRETURN instruction must be able to restore the PC and LV pointers to their old values. It does this by accessing the link pointer (which is the word identified by the current LV pointer). In this location, remember, where the OBJREF was originally stored, the INVOKEVIRTUAL instruction stored the address containing the old PC. This word and the word above it are retrieved to restore PC and LV, respectively, to their old values. The return value, which is stored at the top of the stack of the terminating method, is copied to the location where the OBJREF was originally stored, and SP is restored to point to this location. Control is therefore returned to the instruction immediately following the INVOKEVIRTUAL instruction.

So far, our machine does not have any input/output instructions. Nor are we going to add any. It does not need them any more than the Java Virtual Machine needs them, and the official specification for JVM never even mentions I/O. The theory is that a machine that does no input or output is "safe." (Reading and writing are performed in JVM by means of calls to special I/O methods.)

4.2.4 Compiling Java to IJVM

Let us now see how Java and IJVM relate to one another. In Fig. 4-14(a) we show a simple fragment of Java code. When fed to a Java compiler, the compiler would probably produce the IJVM assembly-language shown in Fig. 4-14(b). The

line numbers from 1 to 15 at the left of the assembly language program are not part of the compiler output. Nor are the comments (starting with //). They are there to help explain a subsequent figure. The Java assembler would then translate the assembly program into the binary program shown in Fig. 4-14(c). (Actually, the Java compiler does its own assembly and produces the binary program directly.) For this example, we have assumed that *i* is local variable 1, *j* is local variable 2, and *k* is local variable 3.

```
i = j + k;                1     ILOAD j          // i = j + k      0x15 0x02
if (i == 3)               2     ILOAD k                         0x15 0x03
    k = 0;                3     IADD                            0x60
else                      4     ISTORE i                        0x36 0x01
   j = j – 1;             5     ILOAD i          // if (i == 3)    0x15 0x01
                          6     BIPUSH 3                        0x10 0x03
                          7     IF_ICMPEQ L1                    0x9F 0x00 0x0D
                          8     ILOAD j          // j = j – 1      0x15 0x02
                          9     BIPUSH 1                        0x10 0x01
                         10     ISUB                            0x64
                         11     ISTORE j                        0x36 0x02
                         12     GOTO L2                         0xA7 0x00 0x07
                         13  L1: BIPUSH 0         // k = 0        0x10 0x00
                         14     ISTORE k                        0x36 0x03
                         15  L2:
   (a)                             (b)                              (c)
```

Figure 4-14. (a) A Java fragment. (b) The corresponding Java assembly language. (c) The IJVM program in hexadecimal.

The compiled code is straightforward. First *j* and *k* are pushed onto the stack, added, and the result stored in *i*. Then *i* and the constant 3 are pushed onto the stack and compared. If they are equal, a branch is taken to *L1*, where *k* is set to 0. If they are unequal, the compare fails and code following IF_ICMPEQ is executed. When it is done, it branches to *L2*, where the then and else parts merge.

The operand stack for the IJVM program of Fig. 4-14(b) is shown in Fig. 4-15. Before the code starts executing, the stack is empty, indicated by the horizontal line above the 0. After the first ILOAD, *j* is on the stack, as indicated by the boxed *j* above the 1 (meaning instruction 1 has executed). After the second ILOAD, two words are on the stack, as shown above the 2. After the IADD, only one word is on the stack, and it contains the sum $j + k$. When the top word is popped from the stack and stored in *i*, the stack is empty, as shown above the 4.

Instruction 5 (ILOAD) starts the if statement by pushing *i* onto the stack (in 5) Next comes the constant 3 (in 6). After the comparison, the stack is empty again (7). Instruction 8 is the start of the else part of the Java program fragment. The else part continues until instruction 12, at which time it branches over the then part and goes to label *L2*.

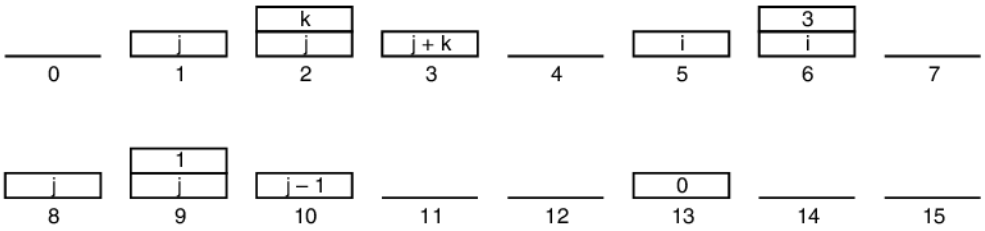


Figure 4-15. The stack after each instruction of Fig. 4-14(b).

4.3 AN EXAMPLE IMPLEMENTATION

Having specified both the microarchitecture and the macroarchitecture in detail, the remaining issue is the implementation. In other words, what does a program running on the former and interpreting the latter look like, and how does it work? Before we can answer these questions, we must carefully consider the notation we will use to describe the implementation.

4.3.1 Microinstructions and Notation

In principle, we could describe the control store in binary, 36 bits per word. But as in conventional programming languages, there is great benefit in introducing notation that conveys the essence of the issues we need to deal with while obscuring the details that can be ignored or better handled automatically. It is important to realize here that the language we have chosen is intended to illustrate the concepts rather than to facilitate efficient designs. If the latter were our goal, we would use a different notation to maximize the flexibility available to the designer. One aspect where this issue is important is the choice of addresses. Since the memory is not logically ordered, there is no natural "next instruction" to be implied as we specify a sequence of operations. Much of the power of this control organization derives from the ability of the designer (or the assembler) to select addresses efficiently. We therefore begin by introducing a simple symbolic language that fully describes each operation without explaining fully how all addresses may have been determined.

Our notation specifies all the activities that occur in a single clock cycle in a single line. We could, in theory, use a high-level language to describe the operations. However, cycle-by-cycle control is very important because it gives the opportunity to perform multiple operations concurrently, and we need to be able to analyze each cycle to understand and verify the operations. If the goal is a fast, efficient implementation (other things being equal, fast and efficient is always better than slow and inefficient), then every cycle counts. In a real implementation, many subtle tricks are hidden in the program, using obscure sequences or operations in order to save a single cycle. There is a high payoff for saving cycles: a four-cycle

instruction that can be reduced by two cycles now runs twice as fast. And this speedup is obtained every time we execute the instruction.

One possible approach is simply to list the signals that should be activated each clock cycle. Suppose that in one cycle we want to increment the value of SP. We also want to initiate a read operation, and we want the next instruction to be the one residing at location 122 in the control store. We might write

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

where WSP means "write the SP register." This notation is complete but hard to understand. Instead we will combine the operations in a natural and intuitive way to capture the effect of what is happening:

```
SP = SP + 1; rd
```

Let us call our high-level Micro Assembly Language "MAL" (French for "sick," something you become if you have to write too much code in it). MAL is tailored to reflect the characteristics of the microarchitecture. During each cycle, any of the registers can be written, but typically only one is. Only one register can be gated to the B side of the ALU. On the A side, the choices are +1, 0, –1, and the register H. Thus we can use a simple assignment statement, as in Java, to indicate the operation to be performed. For example, to copy something from SP to MDR, we can say

```
MDR = SP
```

To indicate the use of the ALU functions other than passing through the B bus, we can write, for example,

```
MDR = H + SP
```

which adds the contents of the H register to SP and writes the result into MDR. The + operator is commutative (which means that the order of the operands does not matter), so the above statement can also be written as

```
MDR = SP + H
```

and generate the same 36-bit microinstruction, even though strictly speaking H must be the left ALU operand.

We have to be careful to use only permitted operations. The most important of them are shown in Fig. 4-16, where SOURCE can be any of MDR, PC, MBR, MBRU, SP, LV, CPP, TOS, or OPC (MBRU implies the unsigned version of MBR). These registers can all act as sources to the ALU on the B bus. Similarly, DEST can be any of MAR, MDR, PC, SP, LV, CPP, TOS, OPC, or H, all of which are possible destinations for the ALU output on the C bus. This format is deceptive because many seemingly reasonable statements are illegal. For example,

```
MDR = SP + MDR
```

looks perfectly reasonable, but there is no way to execute it on the data path of Fig. 4-6 in one cycle. This restriction exists because for an addition (other than increment or decrement) one of the operands must be the H register. Likewise,

H = H – MDR

might be useful, but it, too, is impossible, because the only possible source of a subtrahend (the value being subtracted) is the H register. It is up to the assembler to reject statements that look valid but are, in fact, illegal.

DEST = H
DEST = SOURCE
DEST = $\overline{\text{H}}$
DEST = $\overline{\text{SOURCE}}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

Figure 4-16. All permitted operations. Any of the above operations may be extended by adding "<< 8" to them to shift the result left by 1 byte. For example, a common operation is H = MBR << 8.

We extend the notation to permit multiple assignments by the use of multiple equal signs. For example, adding 1 to SP and storing it back into SP as well as writing it into MDR can be accomplished by

SP = MDR = SP + 1

To indicate memory reads and writes of 4-byte data words, we will just put rd and wr in the microinstruction. Fetching a byte through the 1-byte port is indicated by fetch. Assignments and memory operations can occur in the same cycle. This is indicated by writing them on the same line.

To avoid any confusion, let us repeat that the Mic-1 has two ways of accessing memory. Reads and writes of 4-byte data words use MAR/MDR and are indicated in the microinstructions by rd and wr, respectively. Reads of 1-byte opcodes from the

instruction stream use PC/MBR and are indicated by fetch in the microinstructions. Both kinds of memory operations can proceed simultaneously.

However, the same register may not receive a value from memory and the data path in the same cycle. Consider the code

```
MAR = SP; rd
MDR = H
```

The effect of the first microinstruction is to assign a value from memory to MDR at the end of the second microinstruction. However, the second microinstruction also assigns a value to MDR at the same time. These two assignments are in conflict and are not permitted, as the results are undefined.

Remember that each microinstruction must explicitly supply the address of the next microinstruction to be executed. However, it commonly occurs that a microinstruction is invoked only by one other microinstruction, namely, by the one on the line immediately above it. To ease the microprogrammer's job, the microassembler normally assigns an address to each microinstruction (not necessarily consecutive in the control store) and fills in the NEXT_ADDRESS field so that microinstructions written on consecutive lines are executed consecutively.

However, sometimes the microprogrammer wants to branch away, either unconditionally or conditionally. The notation for unconditional branches is easy:

```
goto label
```

can be included in any microinstruction to explicitly name its successor. For example, most microinstruction sequences end with a return to the first instruction of the main loop, so the last instruction in each such sequence typically includes

```
goto Main1
```

Note that the data path is available for normal operations even during a microinstruction that contains a goto. After all, every single microinstruction contains a NEXT_ADDRESS field. All goto does is instruct the microassembler to put a specific value there instead of the address where it has decided to place the microinstruction on the next line. In principle, every line should have a goto statement. As a convenience to the microprogrammer, when the target address is the next line, it may be omitted.

For conditional branches, we need a different notation. Remember that JAMN and JAMZ use the N and Z bits, which are set based on the ALU output. Sometimes we need to test a register to see if it is zero, for example. One way to do this would be to run it through the ALU and store it back in itself. Writing

```
TOS = TOS
```

looks peculiar, although it does the job (setting the Z flip-flop based on TOS). However, to make microprograms look nicer, we now extend MAL, adding two new imaginary registers, N and Z, which can be assigned to. For example,

```
Z = TOS
```

runs TOS through the ALU, thus setting the Z (and N) flip-flops, but it does not do a store into any register. What using Z or N as a destination really does is tell the microassembler to set all the bits in the C field of Fig. 4-5 to 0. The data path executes a normal cycle, with all normal operations allowed, but no registers are written to. Note that it does not matter whether the destination is N or Z; the microinstruction generated by the microassembler is identical. Programmers who intentionally choose the "wrong" one should be forced to work on a 4.77-MHz original IBM PC for a week as punishment.

The syntax for telling the microassembler to set the JAMZ bit is

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

Since the hardware requires these two addresses to be identical in their low-order 8 bits, it is up to the microassembler to assign them such addresses. On the other hand, since *L2* can be anywhere in the bottom 256 words of the control store, the microassembler has a lot of freedom in finding an available pair.

Normally, these two statements will be combined, for example,

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

The effect of this statement is that MAL generates a microinstruction in which TOS is run through the ALU (but not stored anywhere) so that its value sets the Z bit. Shortly after Z has been loaded from the ALU condition bit, it is ORed into the high-order bit of MPC, forcing the address of the next microinstruction to be fetched from either *L2* or *L1* (which must be exactly 256 more than *L2*). MPC will be stable and ready to use for fetching the next microinstruction.

Finally, we need a notation for using the JMPC bit. The one we will use is

```
goto (MBR OR value)
```

This syntax tells the microassembler to use *value* for NEXT_ADDRESS and set the JMPC bit so that MBR is ORed into MPC along with NEXT_ADDRESS. If *value* is 0, which is the normal case, it is sufficient to just write

```
goto (MBR)
```

Note that only the low-order 8 bits of MBR are wired to MPC (see Fig. 4-6), so the issue of sign extension (i.e., MBR versus MBRU) does not arise here. Also note that the MBR available at the end of the current cycle is the one used. A fetch started in *this* microinstruction is too late to affect the choice of the next microinstruction.

4.3.2 Implementation of IJVM Using the Mic-1

We have finally reached the point where we can put all the pieces together. Figure 4-17 is the microprogram that runs on Mic-1 and interprets IJVM. It is a surprisingly short program—only 112 microinstructions total. Three columns are

Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
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 Main1	Add top two words; write to 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 Main1	Do subtraction; write to top of stack
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 Main1	Do AND; write to new top of stack
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 Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
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
swap1	MAR = SP − 1; rd	Set MAR to SP − 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP − 1; wr	Set MAR to SP − 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	Update TOS
bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack
bipush2	PC = PC + 1; fetch	Increment PC, fetch next opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Sign-extend constant and push on stack
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
istore1	H = LV	MBR contains index; copy LV to H
istore2	MAR = MBRU + H	MAR = address of local variable to store into
istore3	MDR = TOS; wr	Copy TOS to MDR; write word
istore4	SP = MAR = SP − 1; rd	Read in next-to-top word on stack
istore5	PC = PC + 1; fetch	Increment PC; fetch next opcode
istore6	TOS = MDR; goto Main1	Update TOS
wide1	PC = PC + 1; fetch;	Fetch operand byte or next opcode
wide2	goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into
ldc_w1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
ldc_w2	H = MBRU << 8	H = 1st index byte << 8
ldc_w3	H = MBRU OR H	H = 16-bit index into constant pool
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = address of constant in pool

Label	Operations	Comments
iinc1	H = LV	MBR contains index; copy LV to H
iinc2	MAR = MBRU + H; rd	Copy LV + index to MAR; read variable
iinc3	PC = PC + 1; fetch	Fetch constant
iinc4	H = MDR	Copy variable to H
iinc5	PC = PC + 1; fetch	Fetch next opcode
iinc6	MDR = MBR + H; wr; goto Main1	Put sum in MDR; update variable
goto1	OPC = PC − 1	Save address of opcode.
goto2	PC = PC + 1; fetch	MBR = 1st byte of offset; fetch 2nd byte
goto3	H = MBR << 8	Shift and save signed first byte in H
goto4	H = MBRU OR H	H = 16-bit branch offset
goto5	PC = OPC + H; fetch	Add offset to OPC
goto6	goto Main1	Wait for fetch of next opcode
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
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 = OPC − H; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	OPC = PC − 1; goto goto2	Same as goto1; needed for target address
F	PC = PC + 1	Skip first offset byte
F2	PC = PC + 1; fetch	PC now points to next opcode
F3	goto Main1	Wait for fetch of opcode
invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP − H	TOS = address of OBJREF − 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV
invokevirtual20	MDR = LV; wr	Save old LV above saved PC
invokevirtual21	PC = PC + 1; fetch	Fetch first opcode of new method.
invokevirtual22	LV = TOS; goto Main1	Set LV to point to LV Frame

Figure 4-17. The microprogram for the Mic-1 (part 1 on facing page, part 2 above, part 3 on next page).

Label	Operations	Comments
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

Fig. 4-17. The microprogram for the Mic-1 (part 3 of 3).

given for each microinstruction: the symbolic label, the actual microcode, and a comment. Note that consecutive microinstructions are not necessarily located in consecutive addresses in the control store, as we have already pointed out.

By now the choice of names for most of the registers in Fig. 4-1 should be obvious: CPP, LV, and SP are used to hold the pointers to the constant pool, local variables, and the top of the stack, respectively, while PC holds the address of the next byte to be fetched from the instruction stream. MBR is a 1-byte register that sequentially holds the bytes of the instruction stream as they come in from memory to be interpreted. TOS and OPC are extra registers. Their use is described below.

At certain times, each of these registers is guaranteed to hold a certain value, but each can be used as a temporary register if needed. At the beginning and end of each instruction, TOS contains the value of the memory location pointed to by SP, the top word on the stack. This value is redundant since it can always be read from memory, but having it in a register often saves a memory reference. For a few instructions maintaining TOS means *more* memory operations. For example, the POP instruction throws away the top word and therefore must fetch the new top-of-stack word from the memory into TOS.

The OPC register is a temporary (i.e., scratch) register. It has no preassigned use. It is used, for example, to save the address of the opcode for a branch instruction while PC is incremented to access parameters. It is also used as a temporary register in the IJVM conditional branch instructions.

Like all interpreters, the microprogram of Fig. 4-17 has a main loop that fetches, decodes, and executes instructions from the program being interpreted, in this case, IJVM instructions. Its main loop begins on the line labeled Main1. It starts with the invariant that PC has previously been loaded with an address of a memory location containing an opcode. Furthermore, that opcode has already been fetched into MBR. Note this implies, however, that when we get back to this location, we must ensure that PC has been updated to point to the next opcode to be interpreted and the opcode byte itself has already been fetched into MBR.

This initial instruction sequence is executed at the beginning of every instruction, so it is important that it be as short as possible. Through careful design of the Mic-1 hardware and software, we have reduced the main loop to only a single microinstruction. Once the machine has started, every time this microinstruction is

executed, the IJVM opcode to execute is already present in MBR. What the microinstruction does is branch to the microcode for executing this IJVM instruction and also begin fetching the byte following the opcode, which may be either an operand byte or the next opcode.

Now we can reveal the real reason each microinstruction explicitly names its successor, instead of having them executed sequentially. All the control store addresses corresponding to opcodes must be reserved for the first word of the corresponding instruction interpreter. Thus from Fig. 4-11 we see that the code that interprets POP starts at 0x57 and the code that interprets DUP starts at 0x59. (How does MAL know to put POP at 0x57? Possibly there is a file that tells it.)

Unfortunately, the code for POP is three microinstructions long, so if placed in consecutive words, it would interfere with the start of DUP. Since all the control store addresses corresponding to opcodes are effectively reserved, the microinstructions other than the initial one in each sequence must be stuffed away in the holes between reserved addresses. For this reason, there is a great deal of jumping around, so having an explicit microbranch (a microinstruction that branches) every few microinstructions to hop from hole to hole would be very wasteful.

To see how the interpreter works, let us assume, for example, that MBR contains the value 0x60, that is, the opcode for IADD (see Fig. 4-11). In the one-microinstruction main loop we accomplish three things:

1. Increment the PC, leaving it containing the address of the first byte after the opcode.
2. Initiate a fetch of the next byte into MBR. This byte will always be needed sooner or later, either as an operand for the current IJVM instruction or as the next opcode (as in the case of the IADD instruction, which has no operand bytes).
3. Perform a multiway branch to the address contained in MBR at the start of Main1. This address is equal to the numerical value of the opcode currently being executed. It was placed there by the previous microinstruction. Note carefully that the value being fetched in this microinstruction does not play any role in the multiway branch.

The fetch of the next byte is started here so it will be available by the start of the third microinstruction. It may or may not be needed then, but it will be needed eventually, so starting the fetch now cannot do any harm in any case.

If the byte in MBR happens to be all zeros, the opcode for a NOP instruction, the next microinstruction is the one labeled nop1, fetched from location 0. Since this instruction does nothing, it simply branches back to the beginning of the main loop, where the sequence is repeated, but with a new opcode having been fetched into MBR.

Once again we emphasize that the microinstructions in Fig. 4-17 are not consecutive in memory and that Main1 is not at control store address 0 (because nop1