15.1 Organization versus Architecture

Up to this point, the discussion has focused on the components from which computers are built, i.e., computer organization. In contrast, computer architecture is the science of integrating those components to achieve a level of functionality and performance. It is as if computer organization examines the lumber, bricks, nails, and other building material while computer architecture looks at the design of the house.

We've already discussed a number of the components of computer architecture. For example, when we discussed memory in Chapter 12, we introduced the interface that the processor uses to communicate with the memory and other peripherals of the system. Chapter 13 showed how internal registers and the cache RAM improve the processor's performance.

This chapter puts these components together and introduces a few new ones to complete the architecture of a general purpose processor. A few advanced architecture topics are also examined to see how the general architecture can modified to deliver improved performance.

15.2 Components

Before going into detail on how the processor operates, we need to discuss some of its sub-assemblies. The following sections discuss some of the general components upon which the processor is built.

15.2.1 Bus

As shown in Chapter 12, a bus is a bundle of wires grouped together to serve a single purpose. The main application of the bus is to transfer data from one device to another. The processor's interface to the bus includes connections used to pass data, connections to represent the address with which the processor interested, and control lines to manage and synchronize the transaction. These lines are "daisy-chained" from one device to the next.

The concept of a bus is repeated here because the memory bus is not the only bus used by the processor. There are internal buses that the processor uses to move data, instructions, configuration, and status

between its subsystems. They typically use the same number of data lines found in the memory bus, but the addressing is usually simpler. This is because there are only a handful of devices between which the data is passed.

In this chapter we will introduce new control lines that go beyond the read control, write control, and timing signals discussed in Chapter 12. These new lines are needed by the processor in order to service external devices and include interrupt and device status lines.

15.2.2 Registers

As stated when they were introduced in Chapter 13, a register stores a binary value using a group of latches. For example, if the processor wishes to add two integers, it may place one of the integers in a register labeled A and the second in a register labeled B. The contents of the latches can then be added by connecting their Q outputs to the addition circuitry described in Chapter 8. The output of the addition circuitry is then directed to another register in order to store the result. Typically, this third register is one of the original two registers, e.g., A = A + B.

Although variables and pointers used in a program are all stored in memory, they are moved to registers during periods in which they are the focus of operation. This is so that they can be manipulated quickly. Once the processor shifts its focus, it stores the values it doesn't need any longer back in memory.

The individual bit positions of the register are identified by the power of two that the position represents as an integer. In other words, the least significant bit is bit 0, the next position to the left is bit 1, the next is bit 2, and so on.

For the purpose of our discussion, registers may be used for one of four types of operations.

- ***Data registers*** – These registers hold the values on which to perform arithmetic or logical functions.
- ***Address registers*** – Sometimes, the processor may need to store an address rather than a value. A common use of an address register is to hold a pointer to an array or string. Another application is to hold the address of the next instruction to execute.
- ***Instruction registers*** – Remember that instructions are actually numeric values stored in memory. Each number represents a different command to be executed by the processor. Some registers

are meant specifically to hold instructions so that they can be interpreted to see what operation is to be performed.

- ***Flag registers*** – The processor can also use individual bits grouped together to represent the status of an operation or of the processor itself. The next section describes the use of flags in greater detail.

15.2.3 Flags

Picture the instrumentation on the dash board of a car. Beside the speedometer, tachometer, fuel gauge, and such are a number of lights unofficially referred to as "idiot lights". Each of these lights has a unique purpose. One comes on when the fuel is low; another indicates when the high beams are on; a third warns the driver of low coolant. There are many more lights, and depending on the type of car you drive, some lights may even replace a gauge such as oil pressure.

How is this analogous to the processor's operation? There are a number of indicators that reveal the processor's status much like the car's idiot lights. Most of these indicators represent the results of the last operation. For example, the addition of two numbers might produce a negative sign, an erroneous overflow, a carry, or a value of zero. Well, that would be four idiot lights: sign, overflow, carry, and zero.

These indicators, otherwise known as flags, are each represented with a single bit. Going back to our example, if the result of an addition is negative, the sign flag would equal 1. If the result was not a negative number, (zero or greater than zero) the sign flag would equal 0.

For the sake of organization, these flags are grouped together into a single register called the ***flags register*** or the ***processor status register***. Since the values contained in its bits are typically based on the outcome of an arithmetic or logical operation, the flags register is connected to the mathematical unit of the processor.

One of the primary uses of the flags is to remember the results of the previous operation. It is the processor's short term memory. This function is necessary for ***conditional branching***, a function that allows the processor to decide whether or not to execute a section of code based on the results of a condition statement such as "if".

The piece of code shown in Figure 15-1 calls different functions based on the relative values of *var1* and *var2*, i.e., the flow of the program changes depending on whether *var1* equals *var2*, *var1* is greater than *var2*, or *var1* is less than *var2*. So how does the processor determine whether one variable is less than or greater than another?

```
if(var1 == var2)
    equalFunction();
else if(var1 > var2)
    greaterThanFunction();
else
    lessThanFunction();
```

Figure 15-1 Sample Code Using Conditional Statements

The processor does this using a "virtual subtract." This is a subtraction that occurs in the mathematical unit of the processor where it affects the flags, but the result is discarded.

Referring back to our example, the results of a subtraction of *var2* from *var1* is used to select one of three paths through the code.

- ***var1* is equal to *var2*** – When one value is subtracted from an equal value, the result is zero. Therefore, if the zero flag is set after the subtraction, the function *equalFunction()* should be executed.
- ***var1* is greater than *var2*** – If *var1* is larger, then no borrow is needed in the subtraction which results in a non-zero value. (A borrow will set the carry flag.) Therefore, after a subtraction, if the carry flag and the zero flag are both cleared, *var1* was greater than *var2* and the function *greaterThanFunction()* is called.
- ***var1* is less than *var2*** – If *var1* is smaller, then a borrow is needed setting the carry flag. Therefore, after a subtraction, if the carry flag is set, *var1* was less than *var2* and *lessThanFunction()* is called.

Later in this chapter, there is a more detailed examination of this process including a list of the many other program flow control options that are available, each of which tests the flags to determine which code to jump to after one of these virtual subtracts.

15.2.4 Buffers

Rarely does a processor operate in isolation. Typically there are multiple processors supporting the operation of the main processor. These include video processors, the keyboard and mouse interface processor, and the processors providing data from hard drives and CDROMs. There are also processors to control communication

interfaces such as USB, Firewire, and Ethernet networks. These processors all operate independently, and therefore one may finish an operation before a second processor is ready to receive the results.

If one processor is faster than another or if one processor is tied up with a process prohibiting if it from receiving data from a second process, then there needs to be a mechanism in place so that data is not lost. This mechanism takes the form of a block of memory that can hold data until it is ready to be picked up. This block of memory is called a buffer. Figure 15-2 presents the basic block diagram of a system that incorporates a buffer.

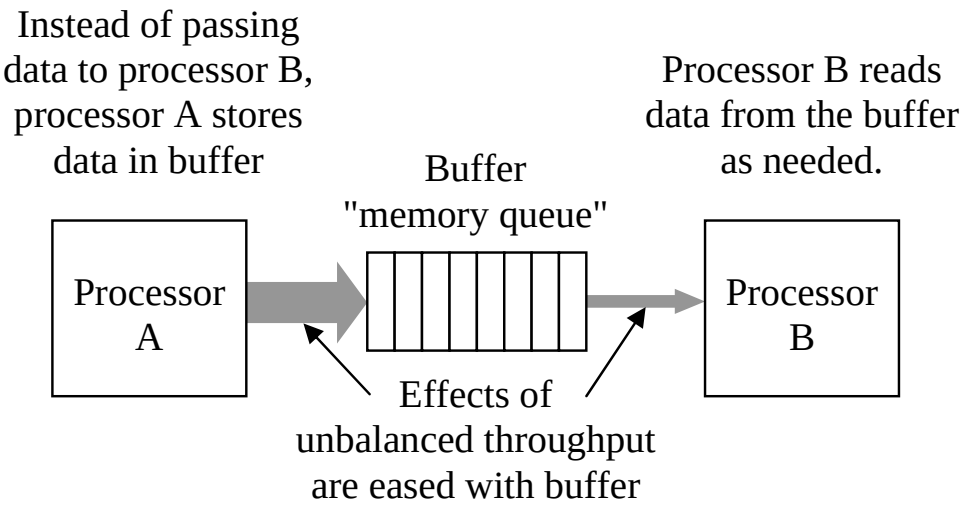


Figure 15-2 Block Diagram of a System Incorporating a Buffer

The concept of buffers is presented here because the internal structure of a processor often relies on buffers to store data while waiting for an external device to become available.

15.2.5 The Stack

During the course of normal operation, there will be a number of times when the processor needs to use a temporary memory, a place where it can store a number for a while until it is ready to use it again. For example, every processor has a finite number of registers. If an application needs more registers than are available, the register values that are not needed immediately can be stored in this temporary memory. When a processor needs to jump to a subroutine or function, it needs to remember the instruction it jumped from so that it can pick

back up where it left off when the subroutine is completed. The return address is stored in this temporary memory.

The ***stack*** is a block of memory locations reserved to function as temporary memory. It operates much like the stack of plates at the start of a restaurant buffet line. When a plate is put on top of an existing stack of plates, the plate that was on top is now hidden, one position lower in the stack. It is not accessible until the top plate is removed.

The processor's stack works in the same way. When a processor puts a piece of data, a plate, on the top of the stack, the data below it is hidden and cannot be removed until the data above it is removed. This type of buffer is referred to as a "last-in-first-out" or LIFO buffer.

There are two main operations that the processor can perform on the stack: it can either store the value of a register to the top of the stack or remove the top piece of data from the stack and place it in a register. Storing data to the stack is referred to as "pushing" while removing the top piece of data is called "pulling" or "popping".

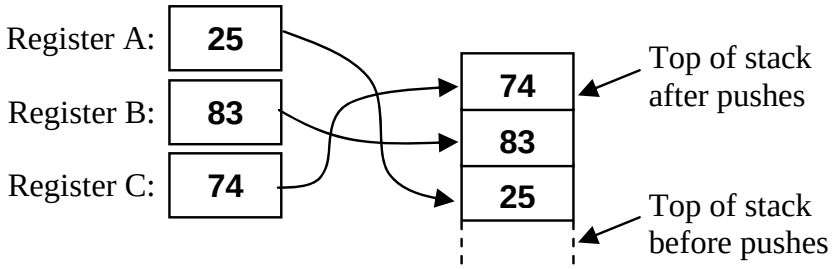
The LIFO nature of the stack makes it so that applications must remove data items in the opposite order from which they were placed on the stack. For example, assume that a processor needs to store values from registers A, B, and C onto the stack. If it pushes register A first, B second, and C last, then to restore the registers it must pull in order C, then B, then A.

Example

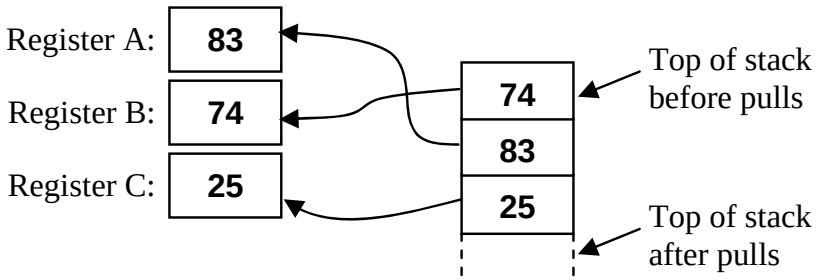
Assume registers A, B, and C of a processor contain 25, 83, and 74 respectively. If the processor pushes them onto the stack in the order A, then B, then C then pulls them off the stack in the order B, then A, then C, what values do the registers contain afterwards?

Solution

First, let's see what the stack looks like after the values from registers A, B, and C have been pushed. The data from register A is pushed first placing it at the bottom of the stack of three data items. B is pushed next followed by C which sits at the top of the stack. In the stack, there is no reference identifying which register each piece of data came from.



When the values are pulled from the stack, B is pulled first and it receives the value from the top of the stack, i.e., 74. Next, A is pulled. Since the 74 was removed and placed in B, A gets the next piece of data, 83. Last, 25 is placed in register C.



15.2.6 I/O Ports

Input/output ports or I/O ports refer to any connections that exist between the processor and its external devices. A USB printer or scanner, for example, is connected to the computer system through an I/O port. The computer can issue commands and send data to be printed through this port or receive the device's status or scanned images.

As described in the section on memory mapping in Chapter 12, some I/O devices are connected directly to the memory bus and act just like memory devices. Sending data to the port is done by storing data to a memory address and retrieving data from the port is done by reading from a memory address.

In some cases, however, the processor has special hardware just for I/O ports. This is done in one of two ways: either the device interface hardware is built into the processor or the processor has a second bus designed to communicate with the I/O devices. In Chapter 16 we will see that the Intel 80x86 family of processors uses the later method.

If the device is incorporated into the processor, then communication with the port is done by reading and writing to registers. This is sometimes the case for simple serial and parallel interfaces such as a printer port or keyboard and mouse interface.

15.3 Processor Level

Figure 15-3 presents the generic block diagram of a processor system. It represents the interface between the processor, memory, and I/O devices through the bus that we discussed in the section on memory interfacing in Chapter 12.

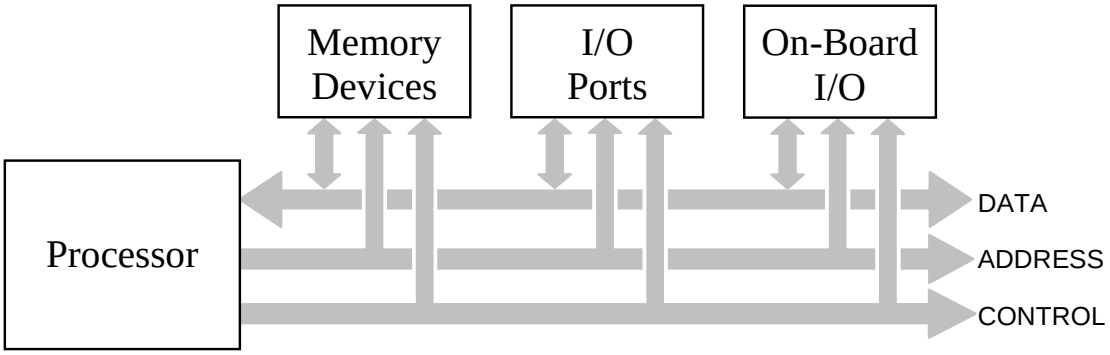


Figure 15-3 Generic Block Diagram of a Processor System

The internals of a processor are a microcosm of the processor system shown in Figure 15-3. Figure 15-4 shows a central processing unit (CPU) acting as the brains of the processor connected to memory and I/O devices through an internal bus within a single chip.

The internal bus is much simpler than the bus the processor uses to connect its external devices. There are a number of reasons for this. First, there are fewer devices to interface with, so the addressing scheme does not need to be that complex. Second, the external bus needs to be able to adapt to many different configurations using components from many different manufacturers. The internal bus will never change for that particular model of processor. Third, the CPU accesses the internal components in a well-defined, synchronized manner allowing for more precise timing logic.

The following is a description of the components of the processor shown in Figure 15-4.

- ***Central processing unit (CPU)*** – This is the brain of the processor. The execution of all instructions occurs inside the CPU along with the computation required to determine addressing.
- ***Internal memory*** – A small, but extremely quick memory. It is used for any internal computations that need to be done fast without the added overhead of writing to external memory. It is also used

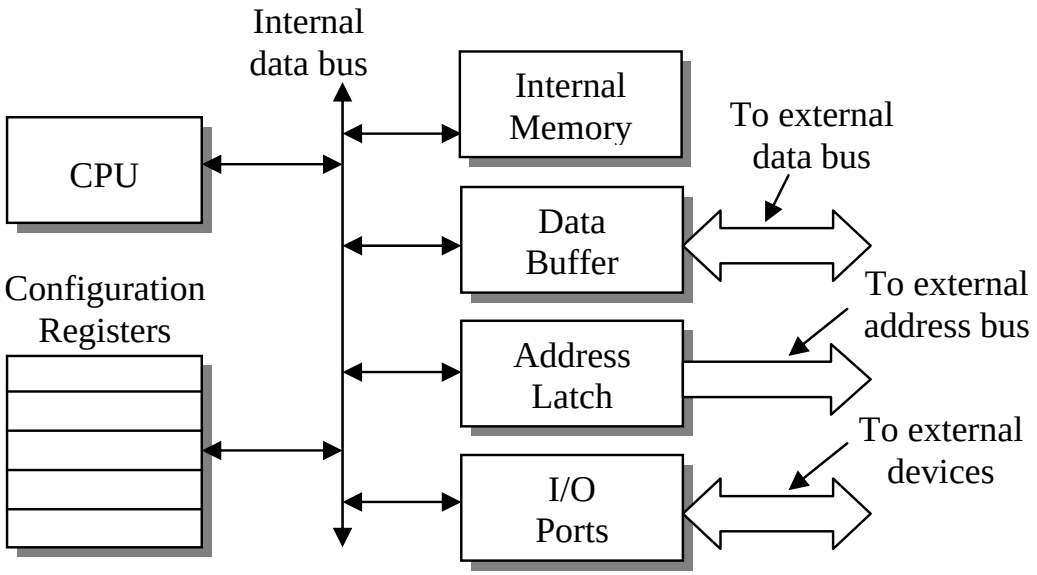


Figure 15-4 Generic Block Diagram of Processor Internals

for storage by processes that are transparent to the applications, but necessary for the operation of the processor.

- ***Data buffer*** – This buffer is a bidirectional device that holds outgoing data until the memory bus is ready for it or incoming data until the CPU is ready for it. This circuitry also provides signal conditioning ensuring the output signals are strong enough and the fragile internal components of the CPU are protected.
- ***Address latch*** – This group of latches maintains the address that the processor wishes to exchange data with on the memory bus. It also provides signal conditioning and circuit protection for the CPU.
- ***I/O ports*** – These ports represent the device interfaces that have been incorporated into the processor's hardware.
- ***Configuration registers*** – A number of features of the processor are configurable. These registers contain the flags that represent the current configuration of the processor. These registers might also contain addressing information such as which portions of memory are protected and which are not.

15.4 CPU Level

If we look at the organization inside the CPU, we see that it in turn is a microcosm of the processor block diagram of Figure 15-4. Figure 15-5 presents the organization inside a typical CPU.

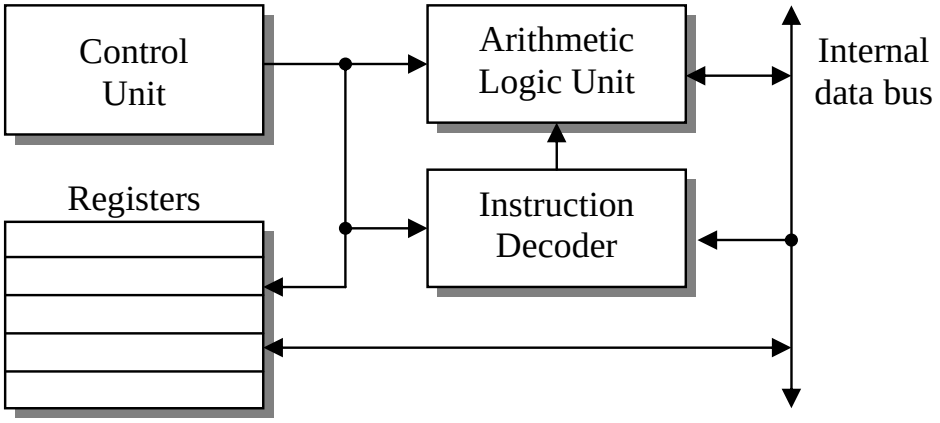


Figure 15-5 Generic Block Diagram of a Typical CPU

- ***Control unit*** – Ask anyone who has worked in a large business what middle management does and they might say something like, "Not a darn thing." Ask them what expertise middle management has and you are likely to get a similar answer. This of course is not true. Middle management has a very important task: they know what needs to be done, who best can do it, and when it needs to be done. This is the purpose of the control unit. It knows the big picture of what needs to be done, it knows which of the CPU's components can do it, and it controls the timing to do it.
- ***Arithmetic logic unit (ALU)*** – The ALU is a collection of logic circuits designed to perform arithmetic (addition, subtraction, multiplication, and division) and logical operations (not, and, or, and exclusive-or). It's basically the calculator of the CPU. When an arithmetic or logical operation is required, the values and command are sent to the ALU for processing.
- ***Instruction decoder*** – All instructions are stored as binary values. The instruction decoder receives the instruction from memory, interprets the value to see what instruction is to be performed, and tells the ALU and the registers which circuits to energize in order to perform the function.
- ***Registers*** – The registers are used to store the data, addresses, and flags that are in use by the CPU.

15.5 Simple Example of CPU Operation

Each component of the CPU has a well-defined allocation of duties. In addition, the interaction between the components is based on a lock-

step communication scheme that places data where it is needed when it is needed. The power of the modern processor is the combination of its ability to execute digital commands quickly and the compiler's ability to take a complex program written in a high-level language and convert it to an efficient sequence of digital commands to be used by the CPU.

Let's examine a short piece of code to see how the CPU might execute it. The following for-loop is presented to show how a compiler might transform it to a sequence of processor commands.

```
int sum = 0, max = 0;
for (int i=0; i<100; i ++)
{
    sum += array[i];
    if (max < array[i]) max = array[i];
}
```

The first thing a compiler might do to create executable code for the processor is to determine how it is going to use its internal registers. It needs to decide which pieces of data require frequent and fast operations and which pieces can be kept in the slower main memory.

First, the index *i* is accessed repeatedly throughout the block of code, so the compiler would assign one of the data registers inside the CPU to contain *i*. Depending on the size of the registers provided by the CPU, it would only need to be an 8-bit register.

Second only to *i* in the frequency of their use are the values *sum* and *max*. They too would be assigned to registers assuming that enough registers existed in the CPU to support three variables. Since *sum* and *max* are defined as integers, they would need to be assigned to registers equivalent to the size of an integer as defined for this CPU. In the Pentium processor, this would be a 32-bit register.

The data contained in *array* would not be loaded into a register, at least not all at once. First of all, each element of *array* is accessed only once, and it isn't even modified during that access. Second, and more important, only a few special application processors have enough registers to hold 100 data elements.

There is one element of *array* that will be stored in a register, and that is the pointer or address that identifies where *array* is stored in memory. Each time the code needs to access an element of *array*, it multiplies the index *i* by the size of an integer, then adds it to the base address of *array*. This provides a pointer to the specific element of *array* in which the CPU is interested.

The sequence shown below is one possible way that a compiler might convert the sample for-loop into CPU commands.

Step 1: Clear registers assigned for *i*, *sum*, and *max*
Step 2: Initialize an address register to point to start of *array*
Step 3: Use address generated by adding *i* multiplied by the size of an integer to the starting address of *array* to retrieve *array*[*i*] from memory
Step 4: Add retrieved value to register assigned to *sum*
Step 5: Compare retrieved value to register assigned to *max*
Step 6: If the value in the register assigned to *max* was less than retrieved value, jump to Step 8
Step 7: Copy retrieved value to register assigned to *max*
Step 8: Increment register assigned to *i*
Step 9: Compare register assigned to *i* to 100
Step 10: If register assigned to *i* is less than 100, jump to Step 3
Step 11: Store values in registers assigned to *sum* and *max* to the appropriate memory locations for later use. Since *i* is visible only within this loop, it does not need to be stored.

There are two things to notice about these steps. First, the steps are very minimal. The instruction set that a CPU uses for its operation is made from short, simple commands. The typical instruction for a CPU involves either a single transaction of data (movement from a register to a register, from memory to a register, or from a register to memory), or a simple operation such as the addition of two registers.

The second thing to notice is that this simple sequence uses a two-step process to handle program flow control. In section 15.2.3, it was shown how a "virtual subtraction" is performed to compare two values. This operation sets or clears the zero flag, the sign flag, the carry flag, and the overflow flag depending on the relationship of the magnitude of the two values. For our example, this virtual subtraction occurs in Step 5 where *max* is compared to the next value retrieved from *array* and in Step 9 where *i* is compared to the constant 100.

Every compare is followed immediately by a ***conditional jump*** that checks the flags to see if the flow of the program needs to be shifted to a new address or if it can just continue to the next address in the sequence. There are many more options for conditional jumps than were presented in the processor flags section. For example, a

conditional "jump if greater than" might work differently when using 2's complement values rather than unsigned integer values.

Table 15-1 presents some of the many options that can be used for conditional jumps after a compare. High-level language compilers use these conditional jumps to transform if-statements, for-loops, while-loops, and switch-case blocks into code useable by the processor. Even though programmers are told to avoid using any type of "jump" commands in their code, compiled CPU instructions are full of them.

Table 15-1 Conditional Jumps to be Placed After a Compare

Jump to new address if…	Flag conditions
equal	zero flag = 1
not equal	zero flag = 0
greater than or equal (unsigned)	carry flag = 0
greater than (unsigned)	carry flag = 0 & zero flag = 0
less than or equal (unsigned)	carry flag = 1 or zero flag = 1
less than (unsigned)	carry flag = 1
greater than or equal (signed)	sign flag = overflow flag
greater than (signed)	sign flag = overflow flag & zero flag = 0
less than or equal (signed)	sign flag != overflow flag or zero flag = 1
less than (signed)	sign flag != overflow flag

The application of conditional jumps is not limited only to use with a compare command. Any operation that affects the flags can be used to change the flow of the code using conditional jumps. For example, a section of code may need to be executed if the result of a multiplication is negative while another section is to be executed if the result is positive. Table 15-2 presents some of the options that can be used for conditional jumps after an arithmetic instruction that affects the flags.

Notice that the flag settings for a conditional jump checking for equality and the conditional jump checking for a zero are the same in both Table 15-1 and Table 15-2. The processor treats these instructions the same. In fact, the processor thinks they are exactly the same command and they are represented in memory using the same code.

The only reason there are two different commands is to assist the programmer by creating syntax that makes more sense linguistically.

Table 15-2 Conditional Jumps to be Placed After an Operation

Jump to new address if…	Flag conditions
result is zero	zero flag = 1
result is not zero	zero flag = 0
result is positive	sign flag = 0
result is negative	sign flag = 1
operation generated a carry	carry flag = 1
operation generated no carry	carry flag = 0

15.6 Assembly and Machine Language

Processor designers create a basic set of instructions for every processor they design. As we have already discussed, these instructions are very simplistic, mere baby steps as compared with high-level languages such as C, C++, or BASIC. In order for the instruction decoder to decipher what an instruction represents, the instruction itself must be a number. These numbers are referred to as ***machine code***. Machine code is the instruction set that the processor uses.

Humans, however, understand words, so each machine code is given a lexical equivalent. These instructions in text form are called ***assembly language***. There is a one-to-one correlation between assembly language instructions and the machine code.

These definitions do not do a good job of showing how processors execute code. For that, let's design the instruction set for a mock processor and use those instructions to create some short programs.

To begin with, assume our mock processor has two registers, A and B. Next, let's assume that the processor is an 8-bit machine, i.e., both A and B are 8-bit registers and can hold unsigned values from 0 to 255 or signed values from –128 to 128. Lastly, let's assume that the processor has 16 address lines. This will give us a memory space of 2^{16} = 64K.

Now let's begin creating the instruction set by brainstorming a list of possible operations we could perform on these two registers and some of the conditional branches that we might need. Of course if you do this exercise on your own, you will come up with a completely different list of operations. Below is the instruction set we will use for our example.

- Move data from A to memory
- Move data from memory to A
- Load A with a constant
- Move data from B to memory
- Move data from memory to B
- Load B with a constant
- Exchange values contained in A and B
- Add A and B and put result in A
- Take the 2's complement of A (make A negative)
- Take the 2's complement of B (make B negative)
- Compare A and B
- Compare A to a constant
- Compare B to a constant
- Jump if equal
- Jump if first value is greater than second value (signed)
- Jump if first value is less than second value (signed)
- Unconditional jump (jump always)

This is a good start except that processors understand binary values, not English. By numbering the instructions, the instruction decoder can identify the requested operation by matching it with the corresponding integer (machine code). Table 15-3 presents one possible numbering.

Unfortunately, human beings are not very adept at programming with numbers. Words are far more natural for us, so each machine code instruction is given a text abbreviation to describe its operation. The resulting collection of words is called ***assembly language***. The one-to-one correspondence between machine code and assembly language is used by a program called an assembler to create the machine code that will be executed by the CPU. Table 15-4 presents a suggested assembly language for the instruction set of our imaginary processor.

We need to define one last item for our instruction set before we can begin programming. Some of the processor's instructions require additional information in order to be executed. This might be a constant to be loaded into a register, an address pointing to a memory location, or some other attribute that the CPU needs in order to properly execute the instruction. These additional pieces of data are called ***operands***. Table 15-5 takes the list of instructions for our processor and shows the size and type of operand that would be needed with each.

Table 15-3 Numbered Instructions for Imaginary Processor

Machine code	Instruction
01	Move data from A to memory
02	Move data from memory to A
03	Load A with a constant
04	Move data from B to memory
05	Move data from memory to B
06	Load B with a constant
07	Exchange values contained in A and B
08	Add A and B and put result in A
09	Take the 2's complement of A (negative)
0A	Take the 2's complement of B (negative)
0B	Compare A to B
0C	Compare A to a constant
0D	Compare B to a constant
0E	Jump if equal
0F	Jump if first value is greater than second value
10	Jump if first value is less than second value
11	Jump unconditionally (jump always)

Table 15-4 Assembly Language for Imaginary Processor

Machine code	Assembly language	Instruction
01	STORA	Move data from A to memory
02	LOADA	Move data from memory to A
03	CNSTA	Load A with a constant
04	STORB	Move data from B to memory
05	LOADB	Move data from memory to B
06	CNSTB	Load B with a constant
07	EXCAB	Exchange values in A and B
08	ADDAB	Add A and B and put result in A
09	NEGA	Take the 2's complement of A
0A	NEGB	Take the 2's complement of B
0B	CMPAB	Compare A to B
0C	CMPAC	Compare A to a constant
0D	CMPBC	Compare B to a constant
0E	JEQU	Jump if equal
0F	JGT	Jump if first value is greater
10	JLT	Jump if second value is greater
11	JMP	Jump always

Table 15-5 Operand Requirements for Imaginary Processor

Instruction	Operands required
Move data from A to memory (STORA)	16-bit memory address
Move data from memory to A (LOADA)	16-bit memory address
Load A with a constant (CNSTA)	8-bit constant
Move data from B to memory (STORB)	16-bit memory address
Move data from memory to B (LOADB)	16-bit memory address
Load B with a constant (CNSTB)	8-bit constant
Exchange values in A & B (EXCAB)	None
Add A and B and put result in A (ADDAB)	None
Take the 2's complement of A (NEGA)	None
Take the 2's complement of B (NEGB)	None
Compare A to B (CMPAB)	None
Compare A to a constant (CMPAC)	8-bit constant
Compare B to a constant (CMPBC)	8-bit constant
Jump if equal (JEQU)	16-bit destination address
Jump if 1st val. Is greater than 2nd val. (JGT)	16-bit destination address
Jump if 1st val. Is less than 2nd val. (JLT)	16-bit destination address
Jump always (JMP)	16-bit destination address

Now that we have a set of instructions, let's create a simple program. This first program adds two variables together and puts the result into a third variable. In a high-level language, this is a single line of code.

RESULT = VAR1 + VAR2

To do this in assembly language, however, takes a few more steps. First, our instruction set does not support the addition of variables in memory. Therefore, the data will need to be copied from memory into registers where the addition can be performed. Second, since the result of the addition will be in a register, we will need to store the data back to memory in order to free up the register. The code below is the assembly language equivalent of RESULT = VAR1 + VAR2.

```
LOADA     VAR1
LOADB     VAR2
ADDAB
STORA     RESULT
```

The next step is to have an assembler convert this assembly language code to machine language so the processor can execute it.

There is another thing that must be done before a processor can execute code: the variable names must be converted into addresses. For the purpose of our example, assume that VAR1 is stored at address $5E00_{16}$, VAR2 is stored at $5E01_{16}$, and RESULT is stored at $5E02_{16}$. By using Table 15-4 to convert the assembly language to machine code and by substituting the addresses shown above, the assembly language program becomes the following sequence of numbers. (All of the values are shown in hexadecimal.)

```
02   5E00
05   5E01
08
01   5E02
```

This is what the processor reads and executes. In memory, it appears as a sequence of binary values, but to the instruction decoder, each byte becomes executable code and data. The following sequence of values is how the data would appear in memory.

```
02 5E 00 05 5E 01 08 01 5E 02
```

Now that it has been shown how assembly language is converted into machine code, let's go the other way and see how the CPU might interpret a sequence of numbers stored as code in memory. Table 15-6 presents a sample of some code stored in memory starting at address 1000_{16}. Each location stores a byte which is the size of a single machine code instruction, an 8-bit constant, or one half of a 16-bit address. All of the values are shown in hexadecimal.

Table 15-6 A Simple Program Stored at Memory Address 1000_{16}

Address	Data	Address	Data	Address	Data
1000_{16}	02_{16}	1005_{16}	$0F_{16}$	$100A_{16}$	05_{16}
1001_{16}	12_{16}	1006_{16}	10_{16}	$100B_{16}$	08_{16}
1002_{16}	$3E_{16}$	1007_{16}	09_{16}	$100C_{16}$	01_{16}
1003_{16}	$0C_{16}$	1008_{16}	09_{16}	$100D_{16}$	12_{16}
1004_{16}	FF_{16}	1009_{16}	06_{16}	$100E_{16}$	$3E_{16}$

Assuming that the instruction decoder is told to begin executing code starting at address 1000_{16} and by using the machine code to assembly language translations found in Table 15-4, this string of values can be decoded into executable instructions. Starting at address 1000_{16}, we see that the first instruction is 02_{16}. Table 15-4 equates 02_{16} to the LOADA instruction while Table 15-5 shows that LOADA uses a 16-bit address. Therefore, the next two bytes in memory (addresses 1001_{16} and 1002_{16}) contain the address from which register A will be loaded. This gives us the first instruction: LOADA 123E.

The next instruction comes after the operands of the LOADA instruction. This puts us at address 1003_{16}. Address 1003_{16} contains $0C_{16}$ which we see from Table 15-4 represents CMPAC, i.e., compare A with a constant. Table 15-5 shows that CMPAC uses a single 8-bit constant as its operand. Since 1004_{16} contains FF_{16}, the 2's complement representation of –1, the next instruction is CMPAC –1.

The CMPAC –1 instruction is followed by the machine code $0F_{16}$ at address 1005_{16}. $0F_{16}$ represents the assembly language JGT, "Jump if first value is greater than second value." When this instruction is executed, it will jump if the value loaded into accumulator A is greater than -1, i.e., if it is a positive number or zero. The next two bytes represent the address that will be jumped to, 1009_{16}.

By continuing this process for the remainder of the code, the assembly language program that is represented by this machine code is revealed. Figure 15-6 presents the final code with the leftmost column presenting the address where the instruction begins and the rightmost column representing an in-line comment field.

```
1000₁₆    LOADA    123E₁₆    ;Put data from address 123E₁₆ in A
1003₁₆    CMPAC    -1        ;Compare A to –1
1005₁₆    JGT      1009₁₆    ;If A>–1, jump to address 1009₁₆
1008₁₆    NEGA               ;A = –A
1009₁₆    CNSTB    5         ;Put a constant 5 in B
100B₁₆    ADDAB              ;A = A + B
100C₁₆    STORA    123E₁₆    ;Store A at address 123E₁₆
```

Figure 15-6 Decoded Assembly Language from Table 15-6

Notice that if A is positive or zero, the compare and subsequent JGT at addresses 1003_{16} and 1005_{16} respectively will force the processor to

skip over the instruction at 1008_{16} and execute the CNSTB 5 at address 1009_{16}. In a high-level language, the code above might look like the following two instructions where the address of VAR is $123E_{16}$.

```
if (VAR > -1) VAR = -VAR;
VAR = VAR + 5;
```

It is important to note that not only does machine language require variable names to be replaced with references to memory addresses, but jumps must also use addresses. Second, note that a comment field has been added to the code in Figure 15-6. All assembly languages have a provision commenting. Usually it is of the in-line variety where a character, in this case a semi-colon (;), is used to comment out all of the subsequent characters until the end of the line is reached.

Every processor has an assembly language associated with it. Since the processors have different architectures, functions, and capabilities, the languages are usually quite different. There are, however, similarities. For example, there are three general categories of instructions for all processors: data transfer, data manipulation, and program control Data transfer instructions are used to pass data between different parts of the processor and memory. These include:

- Register-to-register transfers
- Register-to-memory or port transfers
- Memory or port-to-register transfers
- Memory or port-to-memory or port transfers

Data manipulation instructions make use of the ALU to operate on values contained in the registers or in memory. These include:

- Math operations such as add, subtract, multiply, and divide
- Logic operations such as and, or, xor, and not
- Bit manipulation such as shifting

Within the CPU is a register that contains an address pointing to the next instruction to be executed. There are a number of different names given to this register such as ***program counter*** or ***instruction pointer***. Every time an instruction is executed, this pointer is modified so that it points to the next instruction to be executed. Program control

instructions are used to assign new values to this register so that control can jump to a new position in the program. Some of the program control instructions use the CPU's flags to determine whether a jump in the code will be performed or not. These are the conditional jumps described earlier. The following is a short list of some of the major program control instructions:

- Jump to a new address of the code
- Jump to a subroutine or function
- Return from a subroutine or function
- Conditional jumps

There are a number of reasons to program in assembly language just as there are a number of reasons to avoid it. The tiny, almost primitive processor dependent assembly language instructions cause many problems for programmers. The result is code that is:

- complicated to learn and use;
- hard to debug;
- more time consuming to write;
- unable to be directly transferred to a different processor; and
- harder to decipher if the programmer is unfamiliar with it.

The main benefits of programming in assembly language are due to the fact that the programmer is working much closer to the electronics of the processor. This makes it so that the details of the processor are not hidden by the operating system or compiler. Programming in assembly language gives the programmer:

- full access to all processor resources;
- the ability to make much faster code; and
- the ability to make far more compact code.

15.7 Big-Endian/Little-Endian

In the previous section, some of the operands were 16-bits in length and had to be broken into 8-bit values in order to be stored in memory. It is not much of a problem to store numbers larger than the width of the data bus in memory. By partitioning the value to be stored into

chunks that are the size of the data bus, the processor simply uses sequential memory locations to store large values. For example, if a processor with an 8-bit data bus needs to store the 32-bit value $3A2B48CA_{16}$, it uses four memory locations: one to store $3A_{16}$, one for $2B_{16}$, one for 48_{16}, and one for CA_{16}. When it retrieves the data, it reads all four values and reconstructs the data in one of its registers. The processor designer must ensure that the order in which the smaller chunks are stored remains consistent for both reading and writing, or the value will become corrupted. This should not be a problem.

It can become a problem, however, when data is being transferred between processors that use different orders. Big-endian and little-endian are terms used to identify the order in which the smaller words or bytes are stored. Big-endian means that the first byte or word stored is the most significant byte or word. Little-endian means that the first byte or word stored is the least significant byte or word. The method selected does not affect the starting address, nor does it affect the ordering of items in a data structure.

15.8 Pipelined Architectures

Microprocessor designers, in an attempt to squeeze every last bit of performance from their designs, try to make sure that every circuit of the CPU is doing something productive at all times. Circuitry is added that tries to predict what each CPU component should be doing as soon as it finishes its current task. Even if the prediction was wrong, nothing is lost; the result is simply ignored. If, however, the outcome was useful, then time has been saved and code executed faster.

The most common application of this practice applies to the execution of instructions. It is based on the fact that there are steps to the execution of an instruction, each of which uses entirely different components of the CPU.

Let's begin our discussion by assuming that the execution of a machine code instruction can be broken into three stages:

- ***Fetch*** – get the next instruction to execute from its location in memory
- ***Decode*** – determine which circuits to energize in order to execute the fetched instruction
- ***Execute*** – use the ALU and the processor to memory interface to execute the instruction

By comparing the definitions of the different components of the CPU shown in Figure 15-5 with the needs of these three different stages or cycles, it can be seen that three different circuits are used for these three tasks.

- The internal data bus and the instruction pointer perform the fetch.
- The instruction decoder performs the decode cycle.
- The ALU and CPU registers are responsible for the execute cycle.

Once the logic that controls the internal data bus is done fetching the current instruction, what's to keep it from fetching the next instruction? It may have to guess what the next instruction is, but if it guesses right, then a new instruction will be available to the instruction decoder immediately after it finishes decoding the previous one.

Once the instruction decoder has finished telling the ALU what to do to execute the current instruction, what's to keep it from decoding the next instruction while it's waiting for the ALU to finish? If the internal data bus logic guessed right about what the next instruction is, then the ALU won't have to wait for a fetch and subsequent decode in order to execute the next instruction.

This process of creating a queue of fetched, decoded, and executed instructions is called ***pipelining***, and it is a common method for improving the performance of a processor.

Figure 15-7 shows the time-line sequence of the execution of five instructions on a non-pipelined processor. Notice how a full fetch-decode-execute cycle must be performed on instruction 1 before instruction 2 can be fetched. This sequential execution of instructions allows for a very simple CPU hardware, but it leaves each portion of the CPU idle for 2 out of every 3 cycles. During the fetch cycle, the instruction decoder and ALU are idle; during the decode cycle, the bus interface and the ALU are idle; and during the execute cycle, the bus interface and the instruction decoder are idle.

Figure 15-8 on the other hand shows the time-line sequence for the execution of five instructions using a pipelined processor. Once the bus interface has fetched instruction 1 and passed it to the instruction decoder for decoding, it can begin its fetch of instruction 2. Notice that the first cycle in the figure only has the fetch operation. The second cycle has both the fetch and the decode cycle happening at the same time. By the third cycle, all three operations are happening in parallel.

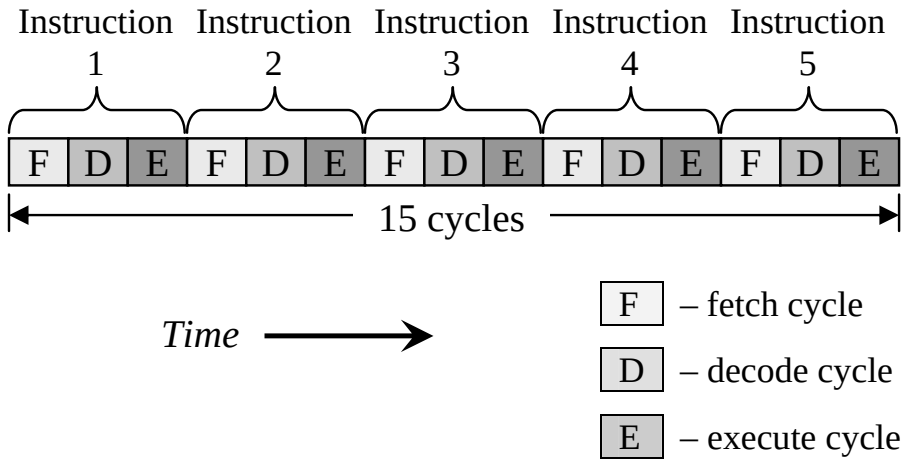


Figure 15-7 Non-Pipelined Execution of Five Instructions

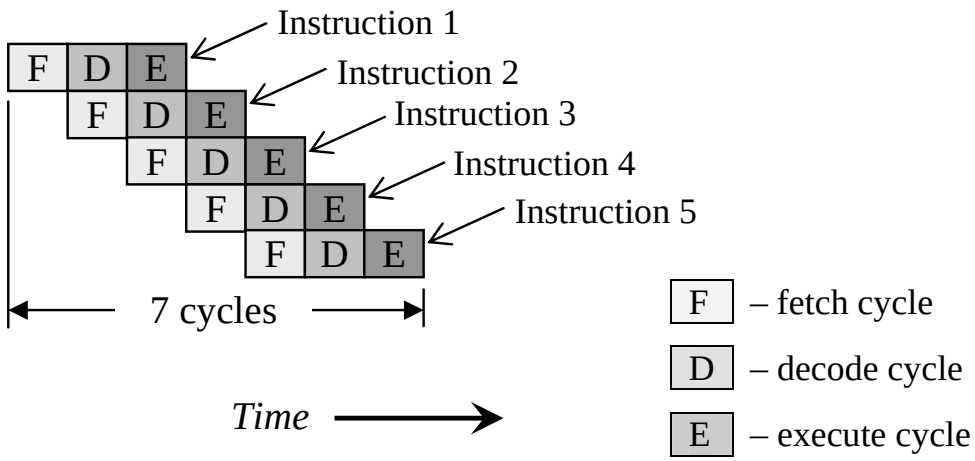


Figure 15-8 Pipelined Execution of Five Instructions

Without pipelining, five instructions take 15 cycles to execute. In a pipelined architecture, those same five instructions take only 7 cycles to execute, a savings of over 50%.

In general, the number of cycles it takes for a non-pipelined architecture using three cycles to execute an instruction is equal to three times the number of instructions.

Num. of cycles (non-pipelined) = 3 × number of instructions (15.1)

For the pipelined architecture, it takes two cycles to "fill the pipe" so that all three CPU components are fully occupied. Once this occurs,

then an instruction is executed once every cycle. Therefore, the formula used to determine the number of cycles used by a pipelined processor to execute a specific number of instructions is:

Num. of cycles (pipelined) = 2 + number of instructions (15.2)

As the number of instructions grows, the number of cycles required of a pipelined architecture approaches 1/3 that of the non-pipelined.

Example

Compare the number of cycles required to execute 50 instructions between a non-pipelined processor and a pipelined processor.

Solution

Using equations 15.1 and 15.2, we can determine the number of cycles necessary for both the non-pipelined and the pipelined CPUs.

number of cycles (non-pipelined) = 3 * 50 = 150 cycles

number of cycles (pipelined) = 2 + 50 = 52 cycles

By taking the difference, we see that the pipelined architecture will execute 50 instructions in 98 fewer cycles.

There is one more point that needs to be addressed when discussing pipelined architectures. In order for the bus interface logic to retrieve the next instruction, it needs to know where to find it. For most instructions, it is only a matter of knowing how many memory locations to move forward from the current position.

For example, assume that the bus interface logic for our mock processor has retrieved the machine code 03. It doesn't need to know that this instruction is CNSTA, "Load A with a constant," it only needs to know how many memory locations the instruction uses. From Table 15-5 we see that CNSTA uses an 8-bit operand. Therefore, including the instruction itself, this particular instruction uses 2 bytes in memory. This means that the bus interface logic needs to increment 2 positions in order to point to the next instruction.

The address of the next instruction can be found even for the unconditional jump instruction, JMP. In this case, the bus interface

logic needs to load the instruction pointer with the two bytes following the JMP = 11_{16} machine code to point to the next instruction to fetch.

There is one group of instructions for which there is no method to reliably predict where to find the next instruction in memory: conditional jumps. For our mock processor, this group of instructions includes "Jump if equal" (JEQU), "Jump if first value is greater than second value" (JGT), and "Jump if first value is less than second value" (JLT). Each of these instructions has two possible outcomes: either control is passed to the next instruction or the processor jumps to a new address. The decision, however, cannot be made until after the instruction is executed, the last cycle of the sequence. This is because the flags from the previous instruction must be evaluated before the processor knows which address to load into the instruction pointer.

There are a number of methods used to predict what the next instruction will be, but if this prediction fails, the pipeline must be flushed of all instructions fetched after the conditional jump. The bus interface logic then starts with a new fetch from the address determined by the execution of the conditional jump. Each time the pipeline is flushed, two cycles are added to the execution time of the code.

15.9 Passing Data To and From Peripherals

Although the vast majority of data transactions within a computer occur between the processor and its memory, sometimes the processor must communicate with external devices. This means that the processor must be able to transfer data to and from devices such as a hard drive or a flash RAM, receive data from inputs such as the keyboard and mouse, and send data to outputs such as the video system.

Every year brings technology that allows for higher and higher densities of digital circuitry. This makes it so that every new processor design contains greater functionality. One of these improvements is to incorporate greater levels of interface circuitry into the processor. This might include a built-in keyboard/mouse interface or a communication interface. When this is done, exchanging data with the interface is performed by reading from or writing to a set of special registers contained within the processor.

Sometimes though, the processor will still need a special interface to an external device. In these cases, the external device can be connected through the same bus that the processor uses to communicate with the memory.

15.9.1 Memory-Mapped I/O

Recall the process that the processor uses to read and write from memory. It begins by placing the address of the memory location it wishes to exchange data with on its address lines. If it is writing data, it places the data to store in memory on the data lines and pulls the write line low while leaving the read line high. If it is reading data, it pulls the read line low while leaving the write line high, then retrieves the data from the data lines.

Sending data to and receiving data from an external input/output (I/O) device can be done using the same process. The major difference is that a memory device will have a great deal more memory locations than an I/O device. Where a memory device may require an address space on the order of Megabytes, an I/O device may require only a few addresses. These addresses may be used for configuring the device, reading its status, receiving captured data, or sending data.

The chip select design discussion in Chapter 12 showed that the address lines are divided into two groups, one that specifies the chip select bit pattern and one that is used to determine the address within the memory device. The number of bits used for the address within the memory device is determined by the size of the device itself. For example, a 256 Meg device uses 28 address lines (2^{28} = 256 Meg).

Assume that an interface needs to be designed for an I/O device that has two registers that are written to, one for writing a configuration and one for writing data, and two registers that are read from, one for reading the device's status and one for reading data. This means that the device requires only two addresses. This can be handled with a single address line, A_0. Table 15-7 presents the signal settings for communicating with such a device.

Table 15-7 Signal Values for Sample I/O Device

A_0	R	W	Function
0	0	1	Reading from device's status register
1	0	1	Reading from device's data register
0	1	0	Writing to device's configuration register
1	1	0	Writing to device's data register
X	1	1	No data transaction

By using the remaining address lines for the chip select, this I/O device can be inserted into the memory map of the processor using the processor's memory bus. This method of interfacing an I/O device to a processor is called ***memory mapping***. Figure 15-9 shows a basic memory mapped device circuit that uses four addresses.

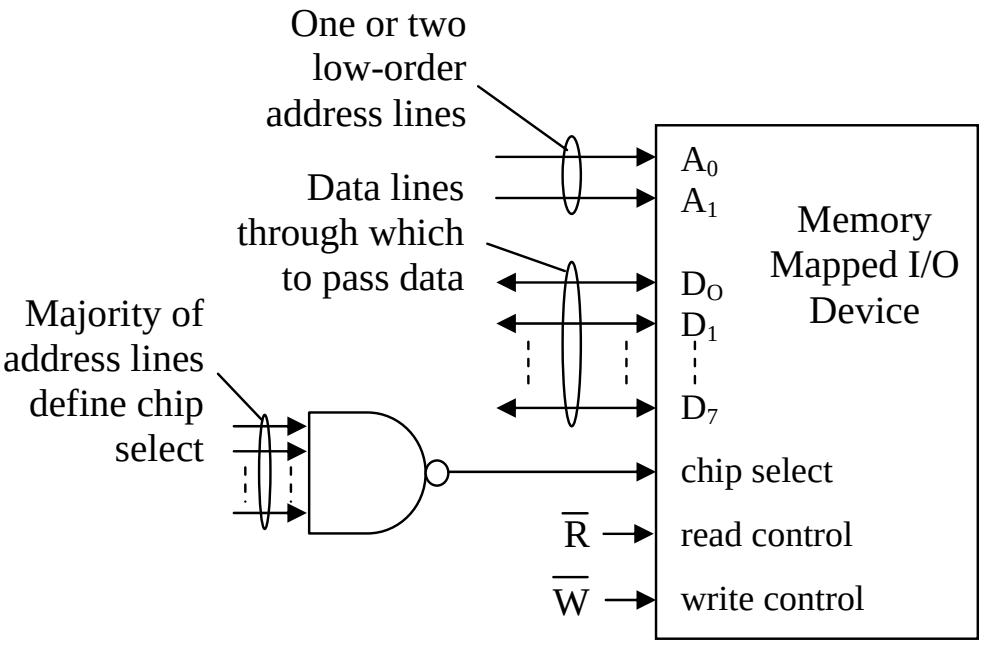


Figure 15-9 Sample Memory Mapped Device Circuit

Some processors add a second read control line and a second write control line specifically for I/O devices. These new lines operate independently of the read and write control lines set up for memory. This does two things for the system. First, it allows the I/O devices to be added to the main processor bus without stealing memory addresses from the memory devices. Second, it makes it so that the I/O devices are not subject to the memory handling scheme of the operating system.

Typically, there is a different set of assembly language instructions that goes along with these new control lines. This is done to distinguish a read or write with a memory device from a read or write with an I/O device. Table 15-8 summarizes how the processor uses the different read and write control lines to distinguish between an I/O device transaction and a memory transaction.

Table 15-8 Control Signal Levels for I/O and Memory Transactions

$\overline{R_{memory}}$	$\overline{W_{memory}}$	$\overline{R_{I/O\ device}}$	$\overline{W_{I/O\ device}}$	Operation
0	1	1	1	Reading from memory
1	0	1	1	Writing to memory
1	1	0	1	Reading from I/O device
1	1	1	0	Writing to I/O device
1	1	1	1	Bus is idle

The methods used to physically connect the processor with an I/O device are only half of the story. The next thing to understand is how the operating system or the software application accesses the device while maintaining responsibility for its other duties.

15.9.2 Polling

The method used by the operating system and its software applications to communicate with I/O devices directly affects the performance of the processor. This is due to the asynchronous nature of I/O. In other words, the I/O device is never ready exactly when the processor needs it to be. For example, the processor cannot predict when a user might press a key, a network connection is not as fast as the processor that's trying to send data down it, and the mechanical nature of a hard drive means that the processor will have to wait for the data it requested. If an I/O interface is not designed properly, the processor will be stalled as it waits for access to the I/O device.

There are four basic methods used for communicating with an I/O device: polling, interrupts, direct memory access, and I/O channels. The first of these, ***polling***, is by far the lowest performer, but it is presented here due to its simplicity.

When an I/O device needs attention from the processor, it usually indicates this by changing a flag in one of its status registers. For example, a network interface may have a bit in one of its status registers that is set to a one when its receive buffer is full. If the processor does not attend to this situation immediately, new incoming data may overwrite the buffer causing the old data to be lost.

In the polling method, the processor continually reads the status registers of the I/O device to see if it needs attention. There are two problems with this method. First, data might be missed if the register is not read often enough. Second, by forcing the processor to

continuously monitor the I/O inputs, considerable processing time is eaten up without having much to show for it. The majority of the reads are not going to show any change in the input values.

15.9.3 Interrupts

The problems caused by using the polling method of communication with an I/O device can be solved if a mechanism is added to the system whereby each I/O device could "call" the processor when it needed attention. This way the processor could tend to its more pressing duties and communicate with the I/O device only when it is asked to. If each call was handled with enough priority, the chance of losing data would be greatly reduced.

This system of calling the processor is called ***interrupt driven I/O***. Each device is given a software or hardware interface that allows it to request the processor's attention. This request might be to tell the processor that new data is available to be read, that the device is ready to receive data, or that a process has completed. The call to the processor requesting service is called an ***interrupt***.

It is as if someone was reading a book when the telephone rings. The reader, concerned about keeping her place in the book, places a book mark to indicate where she left off. She then answers the phone and carries on a conversation while the book "waits" for her attention to return. While chatting on the phone, the person notices the dog standing at the door waiting to be let out. She tells the person on the other end of the line, "Hold that thought, I'll be right back." After she lets out the dog, she returns to the phone call, picks up where she left off. When she finishes talking on the phone, she hangs up and returns to her reading exactly where she left off.

The processor handles devices that need service in a similar way. When the processor receives a device interrupt, it needs to remember exactly what it was doing when it was interrupted. This includes the current condition of its registers, the address of the line of code it was about to execute, and the settings of all of its flags. It does this by storing its registers and instruction pointer to the stack using pushes.

Once its current status is stored, the processor executes a function to handle the device's request. This function is called an ***interrupt service routine (ISR)***. There could be a single ISR for a group of devices or a different ISR for each device. By using interrupts and ISRs, the

processor is able to concentrate on running applications while it is the responsibility of the devices themselves to monitor their condition.

It is important to note that unlike subroutines, ISRs are not called with function calls from the application or operating system code. The processor maintains a list of the ISRs that correspond to each device. When a device interrupts the processor, the processor halts the execution of the main code, looks up the address of the appropriate ISR, and jumps to it. Once the ISR is complete, the processor restores its previous condition by pulling the register values and instruction pointer from the stack so as to pick up the main code where it left off. Figure 15-10 presents a basic diagram of this operation.

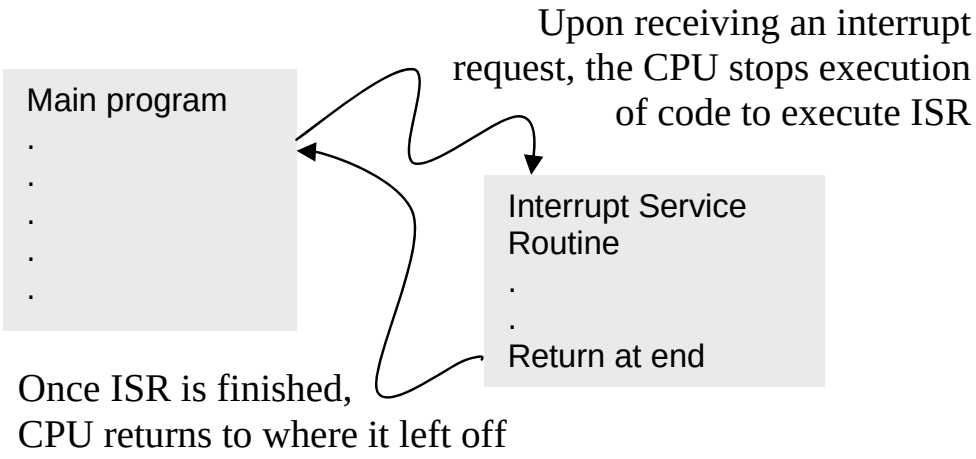


Figure 15-10 Basic Operation of an ISR

Although interrupts greatly improve the performance of a system by requiring the processor's attention only when it is needed, there is still a large burden placed on the processor if the device requires the transfer of a large block of data.

15.9.4 Direct Memory Access

Assume that a communication device receives a large block of data that needs to be placed into memory. It interrupts the processor which in turn initiates the execution of an ISR. The function of the ISR is to make the processor read the data one piece at a time from the device, then store it to memory. This repetitive read-write process takes processing time away from the applications. In addition, each piece of

data goes through a two step process, a read from the device then a store to memory, in order to complete a transfer.

It would be far more efficient for the data to be transferred directly from the I/O device to memory. A process such as this would not need to involve the processor at all. If the processor could remain off of the bus long enough for the device to perform the transfer, the processor would only need to be told when the transfer was completed. It could even continue to perform functions that did not require bus access.

This type of data transfer is called ***direct memory access (DMA)***, and although it still requires an interrupt, it is far more efficient since the processor does not need to perform the data transfer. The typical system uses a device called a DMA controller that is used to take over the bus when the device needs to make a transfer to or from memory. The controller either waits for a time when the processor does not need the bus or it sends the processor a signal asking it to suspend its bus access for one cycle while the I/O device makes a transfer.

A DMA transaction involves a three step process. In the first step, the processor sets up the transfer by telling the DMA controller the direction of the transfer (read or write), which I/O device is to perform the transfer, the address of the memory location where the data will be stored to or read from, and the amount of data to be transferred.

Once the processor has set up the transfer, it relinquishes control to the DMA controller. As the I/O device receives or requires data, it communicates directly with memory under the supervision of the DMA controller. The last step comes when the transfer is complete. At this point, the DMA controller interrupts the processor to tell it that the transfer is complete.

15.9.5 I/O Channels and Processors

As I/O devices become more sophisticated, more and more of the processing responsibility can be taken off of the processor and placed on the I/O device itself. Some I/O devices can access and execute application software directly from main memory without any processor intervention. These are ***I/O channels***. Other I/O devices, ***I/O processors***, are computer systems in their own right taking the functionality of the processor and distributing it to the end devices.

15.10 What's Next?

At this point, the reader should have enough of a background in computer architecture to begin examining a specific processor. In Chapter 16, we will study the Intel 80x86 processor architecture from the point of view of the hardware. Following that, Chapter 17 presents a basic introduction to the Intel 80x86 assembly language.

Problems

1. List the types of registers utilized by the processor and describe their operation.
2. Determine the settings of the zero flag, the carry flag, the overflow flag, and the sign flag for each of the following 8-bit operations.

10110110	01011011	10011001
+ 01001010	+ 01110010	– 00001000

3. If registers A, B, and C contain the values 12, 65, and 87 respectively, and they are pushed to the stack in the order A, then B, then C, what values do A, B, and C have if they then are pulled from the stack in the order C, then A, then B?
4. List and describe the purpose of each of the components of the processor.
5. List and describe the purpose of each of the components of the CPU.
6. Using Tables 15-4 and 15-5, convert the following assembly language to machine code.

```
                LOADA  1000₁₆
                LOADB  1001₁₆
                CMPAB
                JGT    AGREATER
                EXCAB
AGREATER:       STORA  1002₁₆
```

7. What is the purpose of an instruction pointer?

8. List the four drawbacks presented in the text to programming in assembly language.
9. List the three benefits presented in the text to programming with assembly language.
10. Using Tables 15-4 and 15-5, convert the following machine code to assembly language starting at address 2000_{16}.

Address	Data
2000_{16}	02
2001_{16}	13
2002_{16}	4E
2003_{16}	05
2004_{16}	13
2005_{16}	4F
2006_{16}	08
2007_{16}	05
2008_{16}	13
2009_{16}	50
$200A_{16}$	0A
$200B_{16}$	08
$200C_{16}$	01
$200D_{16}$	13
$200E_{16}$	51

11. What type of instruction might force the processor to flush the pipeline?
12. List the two benefits of using separate read/write control lines for I/O devices instead of using memory mapped I/O.
13. What two problems does the polling method to monitor the I/O devices have that are solved by interrupt-driven I/O?
14. What problem does non-DMA interrupt-driven I/O have that is solved by DMA?
15. How would the 32-bit value $1A2B3C4D_{16}$ be stored in an 8-bit memory with a processor that used big-endian? Little-endian?