CHAPTER 1

Fundamentals of a Computer System

There are many resources online to get you started programming, but if you don't have training in computer science, there are certain fundamental concepts that you may not have learned yet that will help you avoid getting frustrated, such as choosing the wrong programming language for the task at hand or feeling overwhelmed. We wrote this book to help you understand computer science basics, whether you already started programming or you are just getting started. We will touch on the topics someone with a computer science degree learns above and beyond the semantics and syntax of a programming language. In this first chapter, we will cover a brief history and evolution of a computer system and the fundamentals of how it operates. We will cover some low-level computer architecture and programming concepts in this chapter, but subsequent chapters will cover higher-level programming concepts that make it much easier to program the computer.

von Neumann Architecture

You've probably heard stories about computers the size of an entire room in the 1940s into the 1970s, built with thousands of vacuum tubes, relays, resistors, capacitors, and other components. Using these various

© Paul D. Crutcher, Neeraj Kumar Singh, and Peter Tiegs 2021
P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0_1

components, scientists invented the concept of gates, buffers, and flip-flops, the standard building blocks of electronic circuits today. In the 1970s, Intel invented the first general-purpose microprocessor, called the 8088, that IBM used to make the first PC that was small enough for personal use. Despite the continuous advancements that have made it possible to shrink the microprocessor, as you'll see, the core elements of today's desktop or laptop computer are consistent with the first computers designed in the 1940s!

In 1945, John von Neumann documented the primary elements of a computer in the "First Draft of a Report on the EDVAC" based on the work he was doing for the government. EDVAC stands for Electronic Discrete Variable Automatic Computer, which was the successor to the Electronic Numerical Integrator and Computer (ENIAC), the first general-purpose computer developed during World War II to compute ballistic firing tables. EDVAC was designed to do more general calculations than calculating ballistic firing tables. As depicted in Figure 1-1, von Neumann described five subdivisions of the system: central arithmetic and central control (C), main memory (M), input (I), output (O), and recording medium (R). These five components and how they interact is still the standard architecture of most computers today.

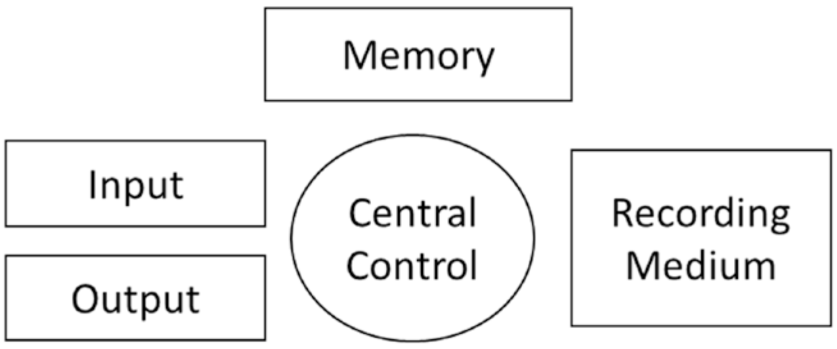

Figure 1-1. *Primary Architecture Elements of a Computer*

In his paper, von Neumann called the central arithmetic and control unit the central control organ and the combination of central control and main memory as corresponding to associative neurons. Even today, people refer to the central processing unit, or CPU, as the "brain" of the computer. Don't be fooled, though, because a computer based on this architecture does exactly what it is programmed to do, nothing more and nothing less. Most often the difficulties we encounter when programming computers are due to the complex nature of how your code depends on code written by other people (e.g., the operating system), combined with your ability to understand the nuances of the programming language you're using. Despite what a lot of people might think, there's no magic to how a computer works, but it can be complicated!

CPU: Fetch, Decode, Execute, and Store

The CPU's job is to fetch, decode, execute, and store the results of instructions. There are many improvements that have been invented to do it as efficiently as possible, but in the end, the CPU repeats this cycle over and over until you tell it to stop or remove power. How this cycle works is important to understand as it will help you debug multi-threaded programs and code for multicore or multiprocessor systems.

Note Threads are a mechanism used to simulate executing a set of instructions in parallel (at the same time), whereas multiple cores in the same system actually do execute instructions in parallel.

The basic blocks of a CPU are shown in Figure 1-2. The CPU needs a clock that sends an electric pulse at a regular interval, called a frequency. The frequency of the clock dictates how fast the CPU can execute its internal logic. The control unit drives the fetch, decode, execute, and store function of the processor. The arithmetic and logic unit, or ALU, performs math operations and digital logic operations like AND, OR, XOR, and so on. The CPU has an internal memory unit for registers and one or more high-speed memory caches to store data proactively pulled in from main memory.

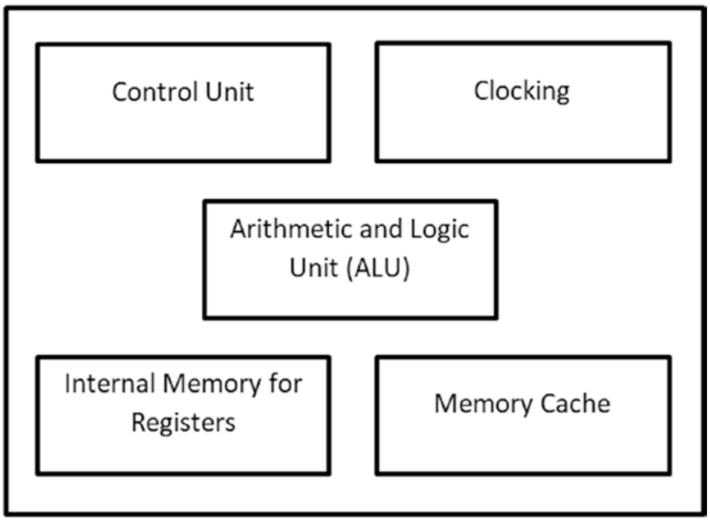
Control Unit
Clocking
Arithmetic and Logic Unit (ALU)
Internal Memory for Registers
Memory Cache

Figure 1-2. *Basic Blocks Inside a CPU*

Fetch

The CPU fetches instructions from memory using addresses. Consider your home's mailbox; it has an address and, if it's anything like my mailbox, contains junk mail and a letter from my mom, if I'm lucky. Like the mail in your mailbox, instructions sit in memory at a specific address. Your mailbox is probably not much bigger than a shoebox, so it has a limit to how much mail the mail carrier can put into it. Computer memory is similar in that each address location has a specific size. This is an important concept to grasp because much of computer programming has to do with data and instructions stored at an address in memory, the size of the memory location, and so on.

When the CPU turns on, it starts executing instructions from a specific location as specified by the default value of its instruction pointer. The instruction pointer is a special memory location, called a register, that stores the memory address of the next instruction.

Here's a simple example of instructions in memory that add two numbers together:

Address	**Instruction**	**Human-Readable Instruction**
`200`	`B80A000000`	`MOV EAX,10`
`205`	`BB0A000000`	`MOV EBX,10`
`20A`	`01D8`	`ADD EAX,EBX`

The first column is the address in memory where the instruction is stored, the second column is the instruction itself, and the third column is the human-readable version of the instruction. The address and instruction numbers are in hexadecimal format. Hexadecimal is a base 16 number system, which means a digit can be `0–F`, not just `0–9` as with the decimal system. The address of the first instruction is `200`, and the instruction is "`mov eax,10`," which means "move the number `10` into the `EAX` register." B8 represents "move something into `EAX`," and `0A000000` is the value. Hexadecimal digit `A` is a `10` in decimal, but you might wonder why it's in that particular position.

It turns out that CPUs work with ones and zeros, which we call binary. The number 10 in binary is `1010`. `B8` is `10111000` in binary, so the instruction `B80A000000` in binary would be `1011 1000 0000 1010 0000 0000 0000 0000 0000 0000`. Can you imagine having to read binary numbers? Yikes!

In this binary format, a single digit is called a "bit." A group of 8 bits is called a "byte." This means the maximum value of a byte would be `1111 1111`, which is `255` in decimal and `FF` in hexadecimal. A word is 2 bytes, which is `16` bits. In this example, the "`MOV EAX`" instruction uses a byte for the instruction and then 4 words for the data. If you do the math, 4 words is 8 bytes, which is 32 bits. But if you are specifying the number 10 (or 0A in hexadecimal) to be moved into the EAX register, why is it 0A000000? Wouldn't that be 167,772,160 in decimal? It would, but it turns out processors don't expect numbers to be stored in memory that way.

```
bit          0 or 1
byte         8 bits
word         2 bytes = 16 bits
dword        2 words = 4 bytes = 32 bits
```

Most CPUs expect the lower byte of the word to be before the upper byte of the word in memory. A human would write the number 10 as a hexadecimal word like this: 000A. The first byte, 00, would be considered the most significant byte; and the second byte, 0A, would be the least significant. The first byte is more significant than the second byte because it's the larger part of the number. For example, in the hexadecimal word 0102, the first byte 01 is the "most significant" byte. In this case, it represents the number 256 (0100 in hexadecimal is 256). The second 02 byte represents the number 2, so the decimal value of the hexadecimal word 0102 is 258. Now, let's look at the "MOV EAX,10" instruction as a stream of bytes in memory:

```
200:  B8    <- Instruction (MOV EAX)
201:  0A    <- Least significant byte of 1st word
202:  00    <- Most significant byte of 1st word
203:  00    <- Least significant byte of 2nd word
204:  00    <- Most significant byte of 2nd word
205:  ??    <- Start of next instruction
```

The instruction is a single byte, and then it expects 4 bytes for the data, or 2 words, also called a "double word" (programmers use DWORD for short). A double word, then, is 32 bits. If you are adding a hexadecimal number that requires 32 bits, like 0D0C0B0A, it will be in this order in memory: `0A0B0C0D`. This is called “little-endian.” If the most significant byte is first, it’s called “big-endian.” Most CPUs use “little-endian,” but in some cases data may be written in “big-endian” byte order when sent between devices, for instance, over a network, so it’s good to understand the byte order you’re dealing with.

For this example, the CPU’s instruction pointer starts at address `200`. The CPU will fetch the instruction from address `200` and advance the instruction pointer to the location of the next instruction, which in this case is address 205.

The examples we’ve been studying so far have been using decimal, binary, and hexadecimal number conventions. Sometimes it is hard to tell what type of number is being used. For example, 10 in decimal is 2 in binary and 16 in hexadecimal. We need to use a mechanism so that it is easy to tell which number system is being used. The rest of this book will use the following notation:

Decimal: No modifier. Example: `10`

Hexadecimal: Starts with `0x` or ends in `h`. Example: `0x10` or `10h`

Binary: Ends in `b`. Example: `10b`

Instruction Set Architecture

Instructions are defined per a specification, called instruction set architecture, or ISA. There are two primary approaches to instruction set architecture that have evolved over time: complex instruction sets and reduced instruction sets. A system built with a complex instruction set is called a complex instruction set computer, abbreviated as CISC. Conversely, a system built with a reduced instruction set is referred to as a reduced instruction set computer, abbreviated as RISC. A reduced instruction set is an optimized set of instructions that the CPU can execute quickly, maybe in a single cycle, and typically involves fewer memory accesses.

Complex instructions will do more work in a single instruction and take as much time to execute as needed. These are used as guiding principles when designing the instruction set, but they also have a profound impact on the microarchitecture of the CPU. Microarchitecture is how the instruction set is implemented. There are multiple microarchitectures that support the same ISA, for example, both Intel and AMD (Advanced Micro Devices) make processors that support the x86 ISA, but they have a different implementation, or microarchitecture. Because they implement the same ISA, the CPU can run the exact same programs as they were compiled and assembled into binary format. If the ISA isn't the same, you have to recompile and assemble your program to use it on a different CPU.

Note A compiler and an assembler are special programs that take code written by humans and convert it into instructions for a CPU that supports a specific instruction set architecture (ISA).

Whether it is complex or reduced, the instruction set will have instructions for doing arithmetic, moving data between memory locations (registers or main memory), controlling the flow of execution, and more. We will use examples based on the x86 ISA to understand how the CPU decodes and executes instructions in the following sections.

Registers

CPUs have special memory locations called registers. Registers are used to store values in the CPU that help it execute instructions without having to refer back to main memory. The CPU will also store results of operations in registers. This enables you to instruct the CPU to do calculations between registers and avoid excess memory accesses. Table 1-1 is the original x86 ISA base register set.

Table 1-1. *x86 Base Register Set*

	64 bits	**32 bits**	**16 bits(8086)**	
	(x86_64)	**(x86)**	**8 bits**	**8 bits**
Accumulator	RAX	EAX	AX	
			AH	AL
Base register	RBX	EBX	BX	
			BH	BL
Counter	RCX	ECX	CX	
			CH	CL
Data	RDX	EDX	DX	
			DH	DL
Base pointer	RBP	EBP	BP	
				BPL
Source index	RSI	ESI	SI	
				SIL
Destination index	RDI	EDI	DI	
				DIL
Stack pointer	RSP	ESP	SP	
				SPL
General purpose	R8-R15	R8D-R15D	R8W-R15W	
				R8B-R15B

It's important to understand how the registers are used by the CPU for the given ISA. For example, the 32-bit counter, in this case ECX, will be automatically decremented by the loop instruction. Another example is the stack pointer where you can directly manipulate it, but it's modified by many other instructions (we will explore the concept of a stack later in this chapter).

The x86 register set has evolved over time and is meant to be backward compatible with older versions of x86 CPUs. You can see the progression from the original 16-bit processor to 32-bit and the now more common 64-bit memory address sizes. As the memory address size increased, so did the register size, and new names were given to allow using the different register sizes with the appropriate instructions. Even when in 64-bit mode, the 32-bit register names enable programs written for 32 bits to run on 64-bit machines.

A typical ISA will have multiple register sets. For example, x86 has a floating-point register set and another register set for handling large data sets. The popular ARM architecture also has multiple register sets. The register set and the ISA go hand in hand!

Decode, Execute, and Store

Decoding is when the CPU interprets the instruction and transfers the data needed to execute the instruction into the CPU to prepare to execute the instruction.

Instructions are formatted in a particular way to enable efficient decoding. The instruction format specifies the opcode (the operation to be performed), the operands (the registers or data needed for the operation), and the addressing mode. The number and order of the operands depends on the instruction addressing mode as follows:

Register Direct: Both operands are registers:

```
ADD EAX, EAX
```

Register Indirect: Both operands are registers, but one contains the address where the operand is stored in memory:

```
MOV ECX, [EBX]
```

Immediate: The operand is included immediately after the instruction in memory:

```
ADD EAX, 10
```

Indexed: The address is calculated using a base address plus an index, which can be another register:

```
MOV AL,  [ESI+0x401000]
MOV EAX, [EBX+EDI]
```

The CPU control unit decodes the instruction and then, based on the addressing scheme, moves data from memory into the appropriate registers. At this point, the instruction is ready, and the control unit drives the ALU to do its work. For example, `ADD EAX, 10` will add the number `10` to the current value of the `EAX` register and store the result in the `EAX` register.

The ALU can support typical math instructions like add (`ADD`), multiply (`MUL`), and divide (`DIV`) for integer numbers. The original arithmetic unit doesn't handle floating-point numbers directly. For example, when you divide a number using the `DIV` instruction, you put the dividend in `EAX` and the divisor in `ECX` and then issue the divide instruction:

```
MOV EDX, 0
MOV EAX, 13
MOV ECX, 2
DIV ECX
```

Since 13 is not an even number, there will be a remainder. The instruction deals with integers only, so the quotient, 6, is stored in `EAX`, and the remainder, 1, is stored in `EDX`. `ECX` will still be 2. You can use other registers for the divisor, but the quotient and remainder will be stored in `EAX` and `EDX`. In 16-bit mode, they are stored in `AX` and `DX`, and in 8-bit mode, this pattern breaks and the quotient is stored in `AL` with the remainder in `AH`.

Just like division has special handling for remainders, addition and subtraction have special handling for carrying and borrowing. For example, a binary number is either 0 or 1. The number 2 is represented as 10b in binary. When you add two bits together (1b + 1b), a carry occurs. This is easily represented digitally by an XOR logic gate and an AND logic gate. A logic gate is a set of transistors that perform logical operations on binary inputs. Figure 1-3 shows how the XOR and the AND gates are wired together to form a half adder circuit. The output of an XOR gate is “one or the other but not both,” so it will be 0 if both inputs are 1. The output of an AND gate is 1 only if both inputs are 1. The output of the AND gate is used to set the carry bit for the add operation.

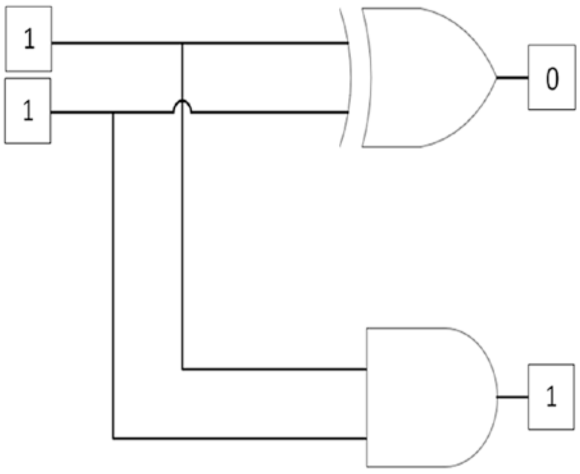

Figure 1-3. *Half Adder Circuit*

The ALU uses many different combinations of logic gates to implement the various instructions. In addition, the ALU also supports logic operations such as OR and AND, shifting bits, comparing, incrementing, decrementing, and more. We’re just scratching the surface here, so if you’re interested in more, we encourage you to study the ISA for your processor.

Controlling the Flow

A very important instruction is one that tells the CPU to start executing instructions from a different location, which is typically referred to as a "jump" instruction. You can program the CPU to perform calculations and then jump (change the instruction pointer) to a different location in memory based on the outcome of the calculations. This technique is used to perform a loop operation. In the following example, we will initialize the `ECX` counter register to 4 and the `ESI` index register to 0. Then we will increment the `ESI` register and call the `LOOP` instruction. The `LOOP` instruction has a special relationship with the `ECX` register. It will automatically decrement the register by one and, if it is greater than zero, jump to the specified location:

```
Address   Instruction       Human-Readable Instruction
0x0200    0xB904000000      MOV ECX,0x4
0x0205    0xBE00000000      MOV ESI,0x0
0x020A    0x46              INC ESI
0x020B    0xE2FD            LOOP 0x020A
```

Let's look at a slightly more complex example. Suppose you have two lists of numbers and you want to add them together and store the result somewhere else in memory:

List 1		List 2		List 3 (results)	
Address	Data	Address	Data	Address	Data
0x401000	01	0x402000	04	0x403000	00
0x401001	02	0x402001	03	0x403001	00
0x401002	03	0x402002	02	0x403002	00
0x401004	04	0x402003	01	0x403003	00

The following instructions add a number from List 1 to the corresponding number in List 2 and put the result in List 3. Again, we will use the ECX as a counter, so we initialize it to 4 since there are four elements in each list. Next, we initialize our source index register (ESI) and destination index register (EDI) to zero. Starting at address 0x0214, we move a byte from the first list into the AL register and a byte from the second list into the AH register. Next, starting at address 0x0220, we move one byte into our destination and then add the other byte to that same location. ESI is added to the address, and then the data located at that calculated address is moved into the AL register. Since we are adding ESI and EDI to the addresses, we increment both of them with the INC instruction before the LOOP instruction. The LOOP instruction automatically decrements ECX and jumps to address 0x214 as long as ECX is greater than zero. There are several other conditional loops and jump instructions that enable you to control program flow in a variety of ways:

Address	**Instruction**	**Human-Readable Instruction**
0x0205	0xB904000000	MOV ECX,0x4
0x020A	0xBE00000000	MOV ESI,0x0
0x020F	0xBF00000000	MOV EDI,0x0
0x0214	0x8A8600104000	MOV AL,[ESI+0x401000]
0x021A	0x8AA600204000	MOV AH,[ESI+0x402000]
0x0220	0x888700304000	MOV [EDI+0x403000],AL
0x0226	0x00A700304000	ADD [EDI +0x403000],AH
0x022C	0x46	INC ESI
0x022D	0x47	INC EDI
0x022E	0xE2E4	LOOP 0x0214

What if you needed to do this operation often? It would be of help if you could put this set of instructions in your program and jump to it from other parts of your program whenever you need to add two lists of numbers together, right? You would need to pass information to this code for the location of the two lists in memory, how many numbers are in the lists, and another memory location to store the results. Also, when the code is done, you need to tell the processor to return to the location it came from so it can continue execution instructions. We call this a function or routine, and thankfully the processor has special instructions and registers to keep track of the input to the function and where to jump to when the function is done doing its work. These special instructions store the needed information on the stack.

The Stack

The stack works on a Last In, First Out (LIFO) principle. Imagine a card game between two people sitting at a table. There are just a few simple rules. First, if there are no cards on the table, you can put a card on the table. If there's a card on the table, you must put the next card on top of the existing card, or stack them. Second, if either of you wants to take a card from the table, you have to take the card from the top of the stack. Thus, the last card put on the top of the stack is always the first one to come off the stack of cards. Of course, we're talking about computers, not people, so in a computer, the table is memory, the people are functions of your program, and the cards are data being passed back and forth. To make it more interesting, some CPUs require that the table is upside down!

For the x86 ISA, there are two instructions to work with the stack: `PUSH` and `POP`. There's also a special register called the extended stack pointer (`ESP`). The x86 stack always starts at a high memory address. As data is pushed onto the stack, the `ESP` decrements to the next address. When the pop instruction is executed, the `ESP` increments to reveal the previous item on the stack. Here is an empty 32-bit stack with ESP set to the address of the first available position:

Address	Data (DWORD)	ESP
0x01000000	0x00000000	0x01000000
0x00FFFFFC	0x00000000	
0x00FFFFF8	0x00000000	
0x00FFFFF4	0x00000000	

Let's push a value onto the stack and look at the result:

```
MOV  EAX, 10
PUSH EAX
```

This is what the stack will look like and the value of the ESP register:

Address	Data (DWORD)	ESP
0x01000000	0x00000000	
0x00FFFFFC	0x0A000000	0x00FFFFFC
0x00FFFFF8	0x00000000	
0x00FFFFF4	0x00000000	

Notice anything? The value is actually stored in the next available spot, not the current location ESP was referring to! The push instruction decrements the address in the ESP register by 4, and then it stores the value at the location. The POP instruction does the opposite; it moves the value at the current address in the ESP register and then increments the ESP register by 4. If we do POP EAX, which means "take the value on the stack and put it in EAX," the stack will look like this in 32-bit mode:

Address	Data (DWORD)	ESP
0x01000000	0x00000000	0x01000000
0x00FFFFFC	0x0A000000	
0x00FFFFF8	0x00000000	
0x00FFFFF4	0x00000000	

The ESP register is now back to `0x01000000`; however, the `0A` value is still sitting at location `0x00FFFFFC`! The `POP` instruction doesn't touch the data; it just copies it to the register you specify and changes the address value stored in `ESP`. However, you can't count on that data staying there as the next push command will overwrite it.

Now that we know how the stack pointer works, let's look at calling the routine we talked about earlier that adds the elements of two lists of numbers and stores the result in memory. Our routine needs the address of the two lists of numbers and the address where to store the results. It also needs to know the number of items in these lists, so let's push these items onto the stack:

```
Address Instruction   Human-Readable Instruction
0x0200  0x6800104000  PUSH DWORD 0x401000
0x0205  0x6800204000  PUSH DWORD 0x402000
0x020A  0x6800304000  PUSH DWORD 0x403000
0x020F  0x6A04        PUSH BYTE +0x4
```

We use the DWORD and BYTE modifiers as hints to the compiler how to treat the numbers. We will cover compiling, linking, and loading in the next chapter. We also need to push an address on the stack so the routine knows where to tell the processor to return to when it is done and then jump to our routine. It turns out that the CALL instruction does this for us, so now we just need to call our routine, which is at address 0x024C in this example:

```
0x0211   0xE836000000    CALL 0x024C
0x0216   ;address of next instruction
```

Now the stack looks like this:

Address	Data (DWORD)	ESP
0x01000000	0x0401000	+16
0x00FFFFFC	0x0402000	+12
0x00FFFFF8	0x0403000	+8
0x00FFFFF4	0x0000004	+4
0x00FFFFF0	0x0000216	0x00FFFFF0

We can reference the parameters on the stack in relation to the current stack pointer. The beginning of our routine will use this technique to put the number of bytes in the lists into ECX, the destination for the results into EDI, the address of the second list of numbers in EBX, and the address of the first list of numbers in EDX. Then, we will do add the numbers together from the two lists and store them at the location stored in EDI. The code has changed a bit because we're using registers in a slightly different way, but it has the same outcome. Note that the ret instruction will use the address at ESP to jump to address 216 to continue executing the next instruction after the call to our routine:

```
0x024C  0x8B4C2404          MOV ECX,[ESP+4]
0x0250  0x8B7C2408          MOV EDI,[ESP+8]
0x0254  0x8B5C240C          MOV EBX,[ESP+12]
0x0258  0x8B542410          MOV EDX,[ESP+16]
```

```
0x025C  0xB800000000        MOV EAX,0x0
0x0261  0xBE00000000        MOV ESI,0x0

0x0266  0x8A0432            MOV AL,[EDX+ESI]
0x0269  0x8A2433            MOV AH,[EBX+ESI]
0x026C  0x00E0              ADD AL,AH
0x026E  0x8807              MOV [EDI],AL
0x0270  0x46                INC ESI
0x0271  0x47                INC EDI
0x0272  0xE2F2              LOOP 0x266

0x0274  0xC3                RET
```

Our routine is simpler than the first list addition example; it doesn't need to use any temporary variables to get its job done. But if we did need temporary variables, there's a way to use the stack to store those variables so that you do not have to allocate them in memory and then have to remember to free that memory. If you use the stack, when your function returns, the stack pointer is adjusted appropriately. It's like a free scratch space for storing information. The way you accomplish this is to simply add the amount of space you want to allocate to the stack pointer, like this:

```
ADD    ESP, 24
```

One problem, though, is as routines call other routines (so-called subroutines), the stack will grow. The stack pointer will continue to grow downward as you push items onto it and call other functions. Within your routine, you need some way to reference your local variables. We use the EBP register, also called the base pointer, to save the value of ESP before we change it. There's a trick, though, because the routine that called our routine may also be using the base pointer to keep track of its local variable space. To avoid any issues, we push the current base pointer, set the base pointer to the current stack pointer, and then move the stack pointer, like this:

```
PUSH  EBP          ;save current base pointer
MOV   EBP, ESP     ;set base pointer to ESP
ADD   ESP, 24      ;move ESP down the stack
```

The area on the stack we use for this purpose is called the "stack frame." To reference this space, we can now subtract from the base pointer, EBP. For example, to initialize three locations on this space, you could do this:

```
MOV [EBP-4], 1
MOV [EBP-8], 2
MOV [EBP-12],4
```

Now we can reference those locations throughout our routine. When we exit our routine, we need to do some cleanup before calling the return function. Basically, we need to restore the stack pointer and then pop the EBP register off the stack to restore the stack frame to what our caller expected:

```
MOV ESP, EBP
POP EBP
RET
```

Remember how we pushed parameters on the stack before calling our function? We definitely want to clean those up. That can be done either by our routine using the RET (short for "return" to the caller) instruction, or we can expect the caller to clean up the stack. This is referred to as the "calling convention" for a routine. It's important to understand the calling convention that the code you are calling uses, and you should pick a consistent calling convention when you write code. Luckily, higher-level programing languages do this for us, but as we write assembly code to work with those higher-level languages, we need to follow those language conventions.

Instruction Pipeline

CPUs are designed to fetch, decode, and execute instructions as efficiently as possible. The circuitry of the CPU is designed in stages that can run in parallel, called parallel execution units. For example, when the CPU is performing the second stage of an instruction, it can start executing the next instruction's first phase. This allows the CPU to use all of its circuitry and execute instructions faster. The stages of executing an instruction are referred to as a pipeline.

A simple five-stage pipeline would have stages for fetching (F), decoding (D), executing (E), accessing memory (M), and writing to a register (R). Here are instructions executing without a pipeline:

T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1	D1	E1	M1	R1					
					F2	D2	E2	M2	R2

The first row is time (T1–T10), the second row is the first instruction, and the third row is the second instruction. In this example, we can't fetch the second instruction until the first instruction completes all five stages:

Utilizing parallel stages in the pipeline, we can start fetching the second instruction after the first one moves to the second stage. This will enable the CPU to greatly decrease the amount of time it takes to execute the two instructions. Instead of ten steps, the instructions are done in only six steps, as follows:

T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1	D1	E1	M1	R1					
	F2	D2	E2	M2	R2				

There are instances where the pipeline will not work well, for example, when the next instruction is relying on the result from a previous instruction. This is called a data hazard. If you're writing code in assembly language, you need to consider how you're using registers to ensure that these hazards are avoided. For higher-level languages, the compiler and assembler will optimize the machine language to ensure the pipeline is executing efficiently to get the best performance out of the processor.

Modern processors have a deep pipeline consisting of over 30 stages! They also use very fast internal memory called a cache to prefetch instructions and data and even execute instructions proactively by predicting the control flow.

Flynn's Taxonomy

Let's revisit the code we wrote to add the values of two lists of numbers. In that example, we were using the add instruction repeatedly on the data in memory. Each instruction was executed against a single piece of data. What if you could tell the processor to execute the add instruction on all of that data with a single instruction? Well, you can. It's called a single instruction, multiple data (SIMD) operation. In 1966, Michael J. Flynn proposed a taxonomy for the different ways that instructions operate on data.

Flynn defined a taxonomy in 1966 to classify parallel computing scenarios. In a parallel computing environment, you have multiple independent processors that can execute concurrently. Today, CPUs have multiple cores that can execute tasks in parallel, so they can execute parallel instructions. Flynn defined four classes, or scenarios:

Single instruction, single data (SISD)	Single instruction, multiple data (SIMD)
Multiple instruction, single data (MISD)	Multiple instruction, multiple data (MIMD)

We've been focusing on SISD, single instruction, single data, which is typical in a single-processor scenario. Let's look at our example of adding two lists of numbers together. The two lists of numbers are multiple data inputs, and it turns out there are instructions in the x86 instruction set that support multiple data inputs, or SIMD instructions as defined by Flynn. It's kind of interesting how it works. We will use the x86 `PADDB` instruction to add the values of both lists together in one shot. `PADDB` stands for "add packed integers." To use the `PADDB` instruction, you need to "pack" the data into a register using the `MOVDQU` instruction. `MOVDQU` stands for "move aligned double quadword." A double quadword is 128 bits (2 × 4 × 16) and is also referred to as an "`OWORD.`" If you remember, our previous example used lists that had 4 bytes. If we increase those to hold 16 bytes, then we have 128 bits. We can "pack" those 128 bits of contiguous data into a 128-bit register using the `MOVDQU` instruction, use `PADDB` to do the addition in one instruction, and then move the result to the destination passed in on the stack as follows:

```
0x00000256  0x8B7C2404       MOV EDI,[RSP+4]
0x0000025A  0x8B5C2408       MOV EBX,[RSP+8]
0x0000025E  0x8B54240C       MOV EDX,[RSP+12]
0x00000262  0xF30F6F02       MOVDQU XMM0, OWORD [RDX]
0x00000266  0xF30F6F0B       MOVDQU XMM1, OWORD [RBX]
0x0000026A  0x660FFCC1       PADDB XMM0,XMM1
0x0000026E  0xF30F7F07       MOVDQU OWORD [RDI],XMM0
0x00000272  0xC3             RET
```

Using the `PADDB` instruction, we've removed the loop entirely! Packing your data into the XMM registers is the trick that makes it work. This implies that these instructions have limitations as to the amount of data you can pack and add at a time, so if the data set is large, you would still have to write a loop to complete the operation, but in the end it should be faster.

Multiple instruction, multiple data, or MIMD, is the case where you have multiple CPUs or CPU cores operating on multiple data streams at once. This is a typical multiprocessor scenario that happens often in today's seemingly single-processor systems. Most CPUs today have multiple cores built into them that can truly execute instructions in parallel. Most of the coordination of running programs on different cores in parallel is handled by the operating system. As a programmer, you write a program, and within that program if you want to execute multiple instructions concurrently on different CPUs, you create execution threads for those instructions with some help from the operating system.

Multiple instruction, single data (MISD) is a less common technique. A good example of MISD would be a fault-tolerant system where you may have processors run a known algorithm on the same data set. If the results don't match, the system knows one of the processors is malfunctioning, at which point it can stop using it and let humans know to replace it!

Main Memory and Secondary Storage

We've covered how the CPU fetches information from memory using addresses and how it decodes and executes instructions with help from special memory locations called registers. We also now know that information in memory is stored in byte-sized chunks (8 bits per byte) and that the CPU keeps track of the next instruction using an instruction pointer. To do its job effectively, the CPU must be able to access any location in memory quickly, which means the main memory must support random access. We call this type of memory "random access memory," or RAM. The main memory must be very fast and is implemented using electronic circuits consisting of capacitors and transistors that work together to store bits. Electronic circuits can only save information while they have power, so that type of memory is called "volatile memory." Therefore, a computer system also needs "non-volatile memory" that will save instructions when there's no power. This type of memory is called secondary storage.

Originally, instructions were encoded on punch cards that were fed by hand into memory. This was very cumbersome! Magnetic tape was originally invented to store audio in the late 1800s and further refined in the early 1900s. In 1950, the first tape recorder was created for storing digital information to be used by a computer. Information on a reel of magnetic tape could not be accessed randomly; instead, it had to be accessed from beginning to end, or sequentially. The tape drive is connected to the computer in a way that the computer can send the drive commands to start reading data from the tape and store it in a particular location in memory. After the instructions from the tape are loaded into memory, the CPU instruction pointer is set to start reading those instructions. This was better than punch cards, but still relatively slow, especially as the number of instructions and data used to run a program increased.

Researchers invented the "hard drive" to provide random access to instructions and data. Hard drives store data on magnetic disks housed in a special container. The disks spin at a high rate, and the mechanism to read the data is on an arm that moves left and right across the surface of the disk to read the data. This provided a cheaper and faster way to read programs from secondary storage into the much faster main memory.

Floppy disks are another type of magnetic media invented after tape. The advantage of a floppy disk was that it could be inserted into a drive that had a head that moved left and right while the disk was spinning to read blocks of data in a more random fashion (but still much, much slower than RAM). They were called floppy drives because they were somewhat flexible when not inserted into the drive.

Secondary storage technology has continued to evolve from high-density CD ROM, which is read and written to using a laser, to solid-state drives (SSDs) that have no moving parts at all. The evolution will continue with the advent of persistent memory that has the potential to be an alternative for main memory that does not lose its content when power is removed or lost. Imagine the implications of a system where the main memory is persistent and instructions no longer have to be moved from secondary storage to main memory before the CPU starts its fetch, decode, and execute cycle.

Input and Output (I/O)

We've talked about how the CPU needs to load the instructions from secondary storage into main memory. But how is that actually done? In modern computers, devices are connected to the same address bus as the CPU and main memory, as depicted in Figure 1-4. This enables CPU instructions to use memory addresses to perform input and output (I/O) operations with devices, which is called "memory-mapped I/O (MMIO)."

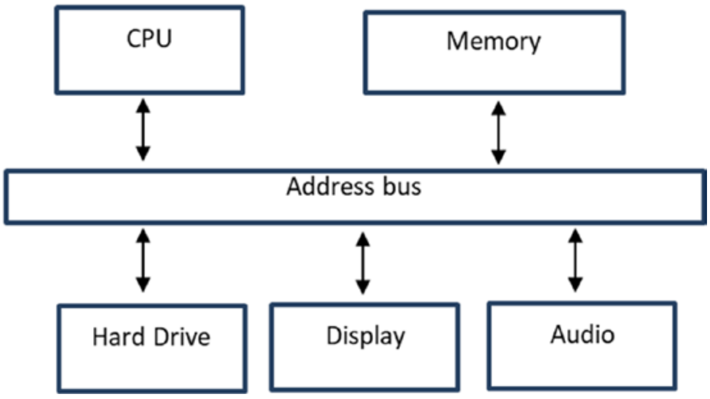

Figure 1-4. *Devices Connected to the Address Bus*

In early x86 processors, there were `input` and `output` instructions that read and wrote to I/O port addresses. Using other CPU instructions with the I/O port addresses would reference main memory, not the intended device. The only way to interact with the device was by using `input` and `output` instructions to load data into CPU registers and then execute instructions using those registers as operands. With memory-mapped I/O, you can simply use the address location for the device as the operand for any CPU instruction. The `input` and `output` instructions still exist in the x86 ISA, but aren't used except by the operating system and some common device drivers.

Through these memory accesses, the CPU can set values that the device interprets as commands. A common scenario is the CPU telling the device to transfer data into main memory, for example, having a hard drive transfer data from its disk to main memory, which is called direct memory access, or DMA. After telling a device to initiate direct memory access, the CPU is free to execute other instructions.

When a device completes its operation, it will let the CPU know it is done through an interrupt, which is a signal connected to the CPU that the device raises or lowers to get the CPU's attention. When the CPU receives the signal, it can stop executing instructions and switch to a special routine that takes care of the interrupt.

Summary

In this chapter, we learned about the fundamentals of a computer system:

- von Neumann Architecture: Central arithmetic and central control (C), main memory (M), input (I), output (O), and recording medium (R)
- Operation of a CPU: Fetch, decode, execute, and store
- Instruction set architecture and register sets
- Controlling the flow of execution and using the stack to implement routines
- Classifying parallel instruction and data using Flynn's taxonomy
- Understanding the difference between main memory and secondary storage
- Input and Output: Memory-mapped I/O and interrupts

Now that we have a basic understanding and hopefully appreciation of computer fundamentals, we can move on to Chapter 2.

References and Further Reading

- The ENIAC Story: https://web.archive.org/web/20110814181522/http://ftp.arl.mil/~mike/comphist/eniac-story.html
- Intel 8088 Microprocessor Family: www.cpu-world.com/CPUs/8088/
- “First Draft of a Report on the EDVAC”: https://web.mit.edu/STS.035/www/PDFs/edvac.pdf
- History of Magnetic Tape: https://history-computer.com/ModernComputer/Basis/tape.html
- Introduction to Dynamic Random Access Memory: www.allaboutcircuits.com/technical-articles/introduction-to-dram-dynamic-random-access-memory/
- John L. Patterson, David A. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier Science Ltd, 2007
- Intel 64 and IA-32 Architectures Software Developer Manuals: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- ARM Developer Documentation: https://developer.arm.com/documentation