

This chapter utilizes narrative, hands-on experimentation, and review questions to introduce computer science and technology concepts.



A Balanced Introduction to Computer Science

David Reed ©Prentice Hall, 2004

Chapter 14: Inside the Computer – The von Neumann Architecture

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

John von Neumann draws attention to what seemed to him a contrast. He remarked that for simple mechanisms, it is often easier to describe how they work than what they do, while for more complicated mechanisms, it is usually the other way around.

Edsger Dijkstra

As was discussed in Chapter 1, virtually all modern computers have the same basic layout, known as the von Neumann architecture. This layout divides the hardware of a computer into three main components: memory, a Central Processing Unit (CPU), and input/output devices. The first component, memory, provides storage for data and program instructions. The CPU is in charge of fetching instructions and data from memory, executing the instructions, and then storing the resulting values back in memory. Input devices (such as the keyboard, mouse, and microphone) allow a person to interact with the computer by entering information and commands, whereas output devices (such as the screen, speakers and printer) are used to communicate data, instructions, and the results of computations.

This chapter explores the details of the von Neumann architecture by describing the inner workings of a computer. We develop our explanation incrementally, starting with a simple representation of the CPU datapath and then adding main memory and a Control Unit. When combined with input and output devices, these components represent an accurate (albeit simplified) model of a modern, programmable computer. Software simulators (originally

developed by Grant Braught at Dickinson College) are provided for each model to facilitate experimentation and hands-on learning.

CPU Subunits and Datapath

As we saw in Chapter 1, the CPU acts as the brain of the computer. It is responsible for obtaining data and instructions from memory, carrying out the instructions, and storing the results back in memory. Each computer's CPU can understand and execute a particular set of instructions, known as that computer's *machine language*. In Chapter 8, we explained that programmers can control a computer by defining instructions for its CPU—this is accomplished either by writing programs directly in machine language, or by writing programs in a high-level language and then translating them into machine language. Even programs that exhibit complex behavior are specified to the CPU as sequences of simple machine-language commands, each performing a task no more complicated than adding two numbers or copying data to a new location. However, the CPU can execute these instructions at such a high speed that complex programmatic behavior is achieved.

CPU Subunits

The CPU itself is comprised of several subunits, each playing a specific role in the processor's overall operation. These subunits are the Arithmetic Logic Unit (ALU), the registers, and the Control Unit (Figure 14.1).

- The *Arithmetic Logic Unit (ALU)* is the collection of circuitry that performs actual operations on data. Basic operations might include addition, subtraction, and bit manipulations (such as shifting or combining bits).
- *Registers* are memory locations that are built into the CPU. Since registers are integrated directly into the CPU circuitry, data in registers can be accessed more quickly (as much as 5-10 times faster) than data in main memory can. However, due to the limited number of registers in the CPU (commonly 16 or 32), these memory locations are reserved for data that the CPU is currently using. To function efficiently, the computer must constantly copy data back and forth between registers and main memory. These transfers occur across collections of wires called a *bus*, which connects the registers to main memory. A separate set of buses connect the registers to the ALU, allowing the ALU to receive data for processing and then store the results of computations back in the registers.
- The *Control Unit (CU)* can be thought of as “the brain within the brain,” in that it oversees the various functions of the CPU. The Control Unit is the circuitry in charge of fetching data and instructions from main memory, as well as controlling the flow of data from the registers to the ALU and back to the registers.

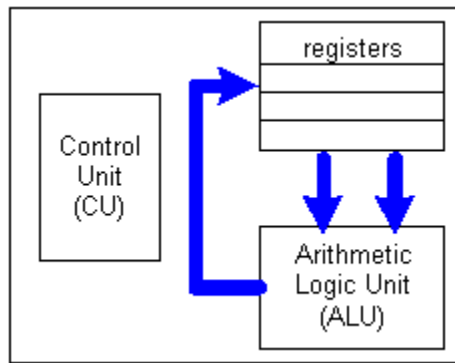


Figure 14. 1: Central Processing Unit (CPU) subunits. Since ALU operations such as addition and subtraction operate on two values, there are two buses connecting the registers to the ALU. The result of the ALU operation is passed back to the registers via a single bus.

CPU Datapath Cycles

The path that data follows within the CPU, traveling along buses from registers to the ALU and then back to registers, is known as the *CPU datapath*. Every task performed by a computer, from formatting a document to displaying a page in a Web browser, is broken down into sequences of simple operations; the computer executes each individual operation by moving data from the registers to the ALU, performing computations on that data within the ALU, and then storing the result in the registers. A single rotation around the CPU datapath is referred to as a *CPU datapath cycle*, or *CPU cycle*.

Recall that, in Chapter 1, we defined CPU speed as measuring the number of instructions that a CPU can carry out in one second. Since each instruction requires a single CPU cycle to execute, we can infer that a CPU's speed will equal the number of CPU cycles that occur per second. For example, an 800-MHz CPU can perform 800 million CPU cycles per second, whereas a 1.4-GHz CPU can perform 1.4 billion CPU cycles per second. However, CPUs cannot be compared solely on the basis of their processor speeds. This is because two machine languages might divide the same task into different sets of instructions, and one set might be more efficient than the other. That is, one CPU might be able to complete a task in a single cycle, whereas another might require several cycles to complete the same task. In order to accurately evaluate a CPU's performance, you must consider the instruction set for that CPU, as well as the number of registers and the size of the buses that carry data between components.

Datapath Simulator

To help you visualize the behavior of the CPU datapath, a simple simulator has been designed to accompany the text. The CPU Datapath Simulator (accessible at <http://www.creighton.edu/~csc107/Sims/datapath.html>) models a simple CPU containing four registers. Using this simulator, you can follow the progress of data as it traverses the CPU datapath, from the registers to the ALU and back to the registers. To keep things simple, we have avoided including an explicit Control Unit in this simulator. Instead, the user

must serve as the Control Unit, selecting the desired input registers, ALU function, and output register by clicking the knob images. Note that not all features of the simulator will be demonstrated in this chapter. In particular, the ALU has two bit-manipulation operations, $\&$ and $|$, which involve combining individual bits in the two input values. Likewise, there are status boxes within the ALU that identify when an operation results in a negative number, zero, or an overflow (a value too large to be represented). You are free to experiment with these features, but all of the examples in this chapter will involve simple addition and subtraction.

Figures 14.2 through 14.5 demonstrate using the simulator to add two numbers together—a task that can be completed during a single CPU cycle.

- This simulator uses text boxes to represent registers, enabling the user to enter data by typing in the boxes. The knobs, which allow the user to specify how data moves along the CPU datapath and what operations the ALU performs on the data, are images that change when the user clicks them. In Figure 14.2, the user has entered the numbers 43 and -296 in registers R0 and R1, respectively. After inputting these values, the user clicked the A Bus knob so that it selects R0 and the B Bus knob so that it selects R1. This will cause the numbers stored in these two registers to be transferred along the indicated buses to the ALU, which will perform an operation on them. Since the user has set the ALU Operation knob to addition and the C Bus knob to R2, the ALU will add the two numbers together, and the result will travel along the C Bus to be stored back in register R2.
- After entering the desired settings, the user initiates a CPU cycle within the simulator by clicking the button labeled "Execute". Figure 14.3 depicts the state of the CPU as the values in R0 and R1 travel along the A and B buses to the ALU. The arrows that represent the buses blink red, and the numbers being transferred are displayed in text boxes next to the buses.
- Figure 14.4 illustrates the state of the CPU after the ALU has received the numbers and performed the specified operation. Since the user set the ALU Operation knob to addition, the ALU adds the two numbers, 43 and -296. The result, -253, is then sent out along the C Bus.
- Finally, Figure 14.5 shows the end result of the CPU cycle. Since the user set the C Bus knob to R2, the value -253 (which travels along the C Bus) is ultimately stored in register R2.

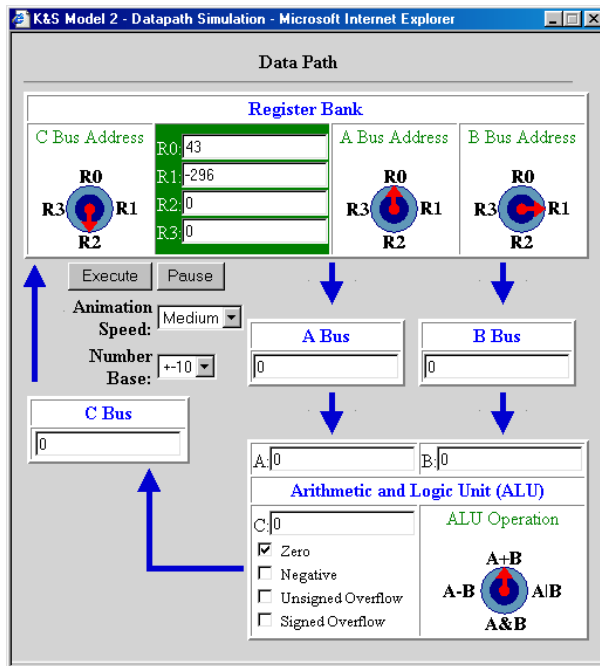


Figure 14.2: Initial settings of the simulator.

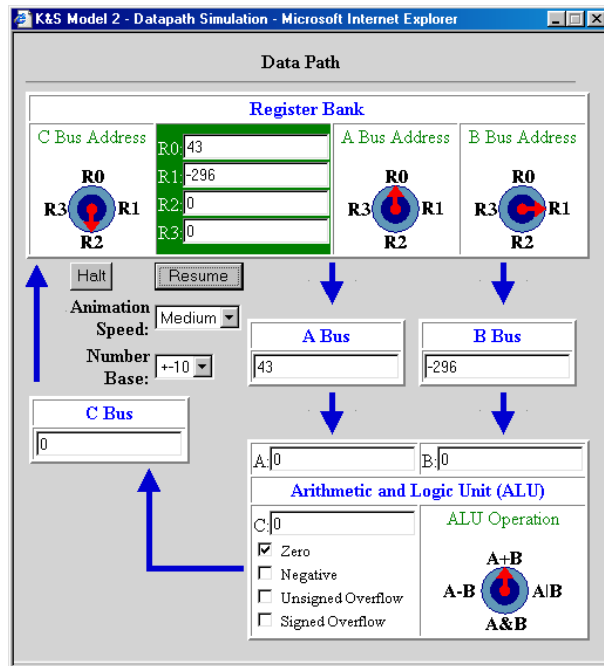


Figure 14.3: Data traveling from registers to the ALU.

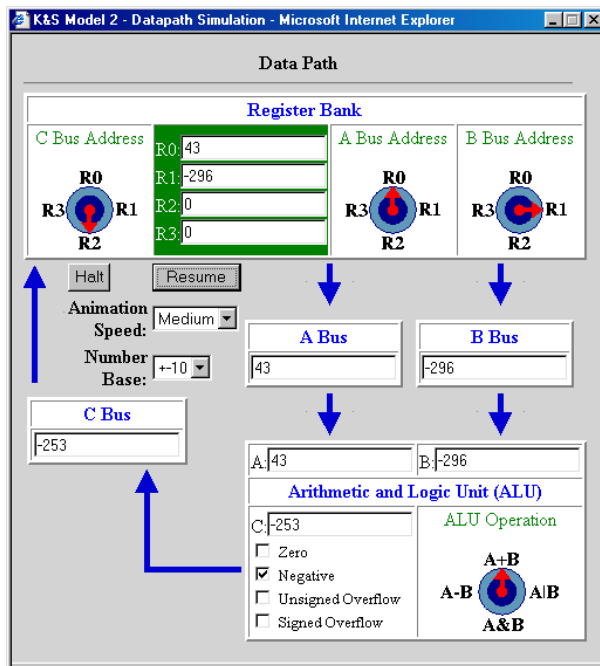


Figure 14.4: Data traveling from ALU to registers.

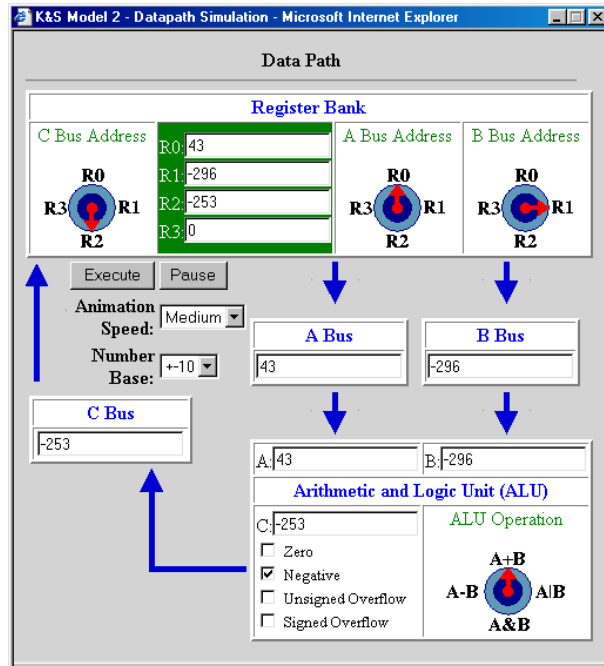


Figure 14.5: Final result of the CPU cycle.

HANDS-ON EXERCISES: Experiment with this simulator until you are familiar with the CPU datapath and the events that occur within a CPU cycle. Then, use the simulator to answer the following questions:

- 14.1. What would happen if you placed the number 100 in R0, then set the knobs so that A Bus = R0, B Bus = R0, ALU = A-B, and C Bus = R0?
- 14.2. Describe the settings that would cause the value stored in R2 to be doubled.
- 14.3. How many cycles are required to add the contents of R0, R1, and R2 and then place the sum in R3? Describe the settings for each cycle.

CPU and Main Memory

Although the CPU datapath describes the way in which a computer manipulates data stored in registers, we have not yet explained how data gets into the registers in the first place, or how the results of ALU operations are accessed outside the CPU. Both these tasks involve connections between the CPU and main memory. Recall from Chapter 1 that all active programs and data are stored in the main memory of a computer. We can think of main memory as a large collection of memory locations, in which each location is accessible via an address. Similar to the way in which a street address (e.g., 27 Maple Drive) allows a mail carrier to find and access a mailbox, a memory address (e.g., memory location 27) allows the CPU to find and access a particular piece of main memory. A bus connects main memory to the CPU, enabling the computer to copy data and instructions to registers and then copy the results of computations back to main memory. Figure 14.6 illustrates the interaction between a computer's main memory and CPU; the darker arrows represent the CPU datapath, whereas the lighter arrow represents the bus that connects main memory to the registers.

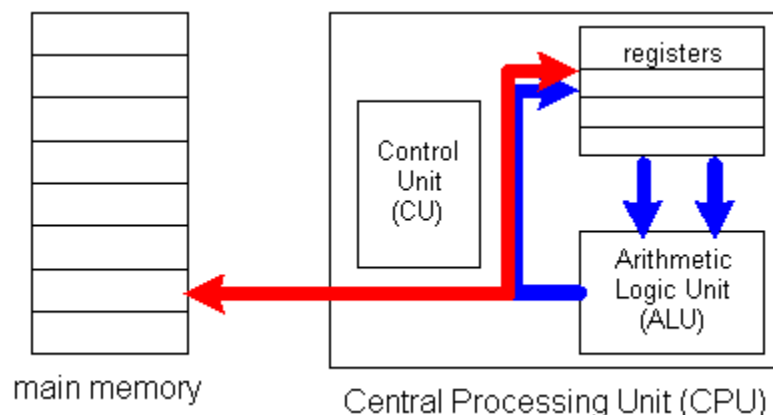


Figure 14. 6: A bus connects Main Memory to the CPU.

Transferring Data to and from Main Memory

As a program is executed, the Control Unit processes the program instructions and identifies which data values are needed to carry out the specified tasks. The required values are then fetched from main memory along the main memory bus, loaded into registers, and utilized in ALU operations.

As a concrete example, imagine that you had a file containing 1,000 numbers and needed to compute the sum of those numbers. To begin, the computer would need to load the data from the file into main memory—for example, at memory locations 500 through 1499. Then, the Control Unit would carry out the following steps to add the numbers and store the resulting sum back in main memory.

1. Initialize one of the registers, say R0, to 0. This register will store the running total of the numbers.
2. For each number stored at memory addresses 500 through 1499:
 - a. Copy the number from main memory to another register, say R1.
 - b. During one cycle around the CPU datapath, add the contents of R0 and R1 and store the result back in R0.
3. When all the numbers in the file have been processed, the value in R0 will represent their sum. This value can then be copied back to a main memory location.

Note that each number must be transferred into a register before it can be added to the sum. In practice, transferring data between main memory and the CPU takes much longer than executing a single CPU cycle. This is mainly due to the fact that the electrical signals must travel a greater distance—for example, from a separate RAM chip to the CPU chip. In the time it takes for data to traverse the main memory bus and reach the registers, several CPU cycles may actually occur. Modern processors compensate for this delay with special hardware that allows multiple instructions to be fetched at once. By fetching several instructions ahead, the processor can often identify instructions that are not dependent on the current one, and execute them while the current data transfer is in progress. Thus, the CPU can perform useful computations as opposed to sitting idle while an instruction waits for data to be copied from main memory to the registers.

Datapath with Memory Simulator

Our next version of the CPU DataPath Simulator has been augmented so that it illustrates the relationship between the CPU and main memory. The main memory incorporated in this extended simulator (accessible at

<http://www.creighton.edu/~csc107/Sims/dpandmem.html>) can store up to 32 16-bit numbers, with addresses 0 through 31. A new bus, labeled "Main Memory Bus," connects the main memory to the CPU, allowing data and computation results to be transferred between the main memory and the registers. As in our previous example, this version of the simulator does

not contain an explicit Control Unit. The user must serve as the Control Unit, selecting the desired settings on the Main Memory Bus and C Bus to control the data flow. Note that these buses can move data either between main memory and the registers or from the ALU to main memory, depending on how the user sets the switches. The user can open and close these bus switches by clicking them, effectively disconnecting and connecting the buses.

Figures 14.7 through 14.9 demonstrate using the simulator to add two numbers stored in main memory.

- When the CPU must add two numbers stored in main memory, the first step is to copy one of the numbers into a register. In Figure 14.7, the user has selected the first number to be added, currently in main memory location 0, by clicking the R/W button next to that location (R/W refers to the fact that the button selects which memory location will be *read from* or *written to*). The user has also configured the arrows surrounding the Main Memory Bus so that they connect main memory to the registers. Once inside the CPU, the Main Memory Bus connects to the C Bus, which loads the number into register R0 (since the C Bus knob is set to R0).
- Figure 14.8 illustrates the next step in our example, which involves copying the second number into a register. Since the user has highlighted the R/W button next to main memory location 1 and set the C Bus knob to R1, the contents of location 1 are fetched and stored in register R1.
- Figure 14.9 depicts the CPU cycle during which the ALU adds the contents of R0 and R1. Note that the A Bus, B Bus, and ALU Operation knob settings are the same as those in Figures 14.3 through 14.6; this is because the two examples portray the same task (i.e., adding the numbers stored in R0 and R1). In Figure 14.9, however, the switches on the Main Memory bus are set so that the result of the addition is sent to main memory, rather than to the registers.

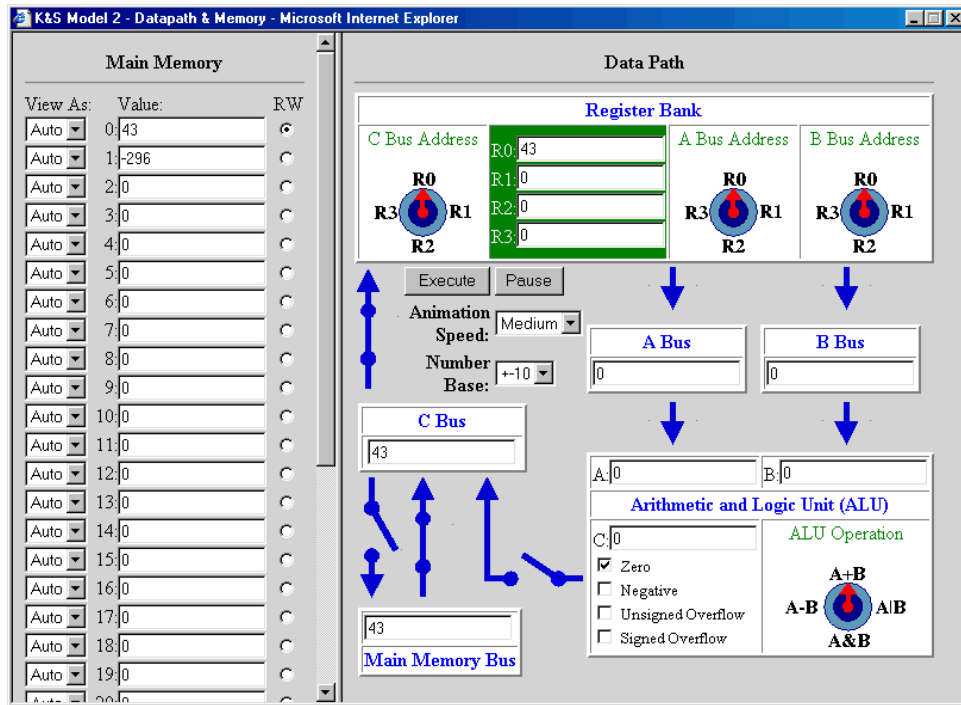


Figure 14. 7: First, 43 is loaded from memory into R0.

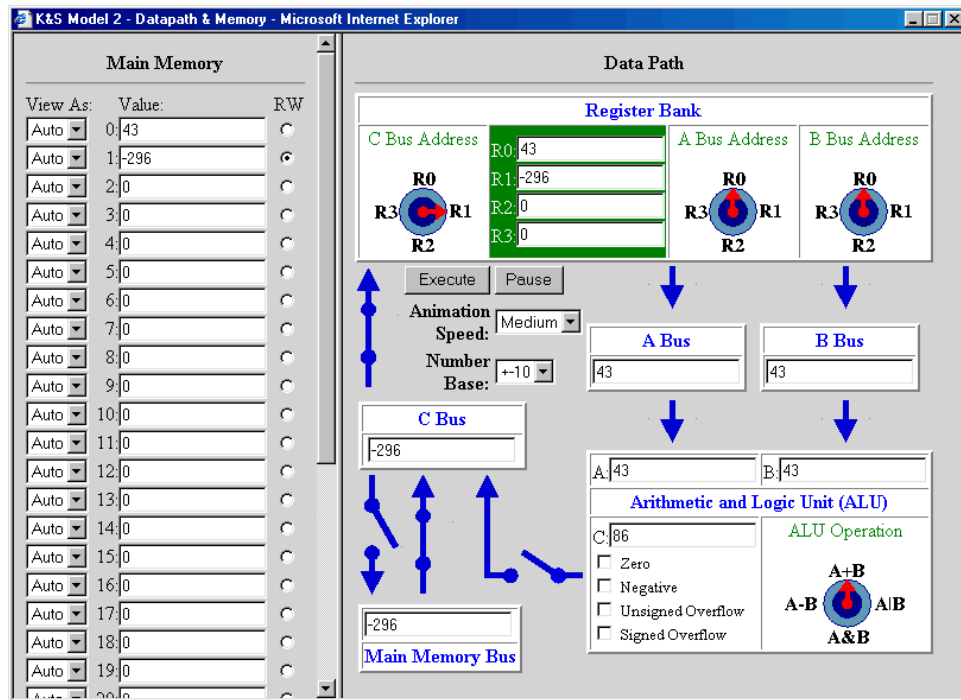


Figure 14. 8: Second, -296 is loaded from main memory into R1.

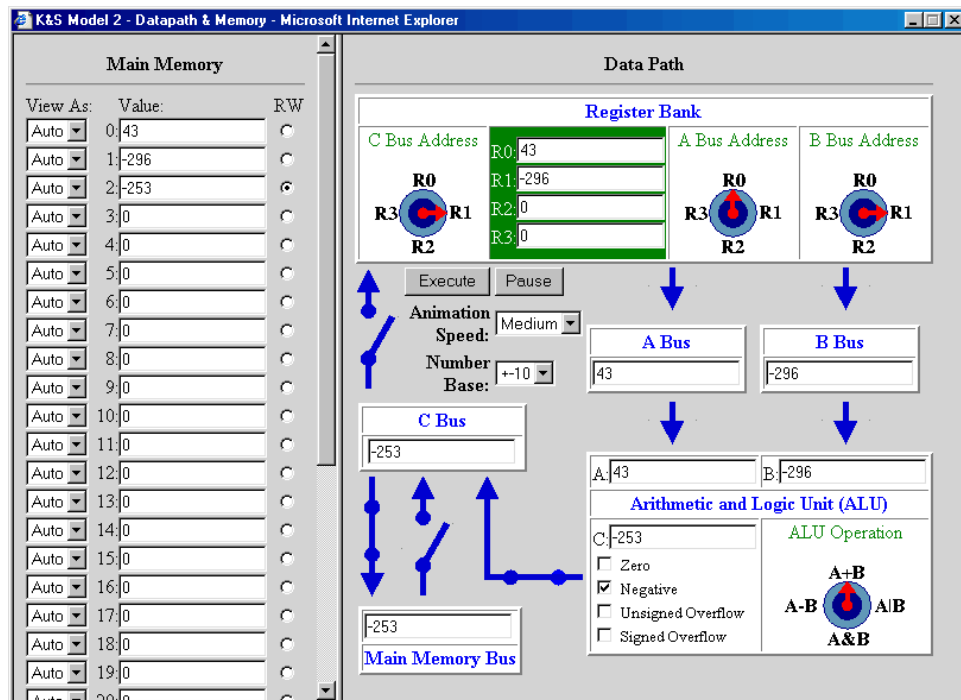


Figure 14. 9: Finally, the values are added, and their sum is stored back in main memory.

Two interesting observations can be made concerning the behavior of the simulator. First, the simulator requires more time to transfer data between main memory and the CPU than it does to perform a CPU datapath cycle. This delay is meant to simulate the slower access times associated with main memory. In a real computer, as many as 10 CPU cycles might occur in the time it takes to transfer data between the CPU and main memory. The second observation is that, even while data is being fetched from main memory, operations are still performed on the CPU datapath. For example, in Figure 14.8, the number in R0 (43) is sent along both the A and B Buses to the ALU, yielding the sum 86. This might seem wasteful, since the result of the ALU operation is ignored (due to the disconnected C Bus). Surprisingly, this is an accurate reflection of a CPU's internal workings. It is more efficient for the ALU to perform needless computations while data is being transferred to or from main memory than it would be to add extra circuitry to recognize whether the C Bus is connected.

HANDS-ON EXERCISES: Experiment with this simulator until you are familiar with the interactions between main memory and the CPU datapath. Then, use the simulator to answer the following questions:

- 14.4. What settings would result in the sum of registers R0 and R3 being stored in memory location 4?
- 14.5. What settings would cause the contents of memory address 4 to be copied into register R0?
- 14.6. Assuming that data can be copied to and from main memory in a single CPU cycle, how many cycles are required to add the contents of memory addresses 5 and 6 and then store the result in memory address 7? Describe the settings for each cycle.

Stored-Program Computer

Now that we have explored how main memory works, we are ready to focus on the last component of the CPU: a fully functioning, automatic Control Unit. To understand the role of the Control Unit, recall the tasks that you performed while using the simulators. When you experimented with the Datapath Simulator (Exercises 14.1 – 14.3), you defined the computation carried out during a CPU cycle by selecting the registers and ALU operation via knobs. In the datapath and main memory simulator (Exercises 14.4 – 14.6), you controlled the flow of information between the datapath and main memory via switches on the buses. The key idea behind a stored-program computer is that tasks such as these can be represented as instructions, stored in main memory along with data, and then carried out by the Control Unit.

Machine Languages

As we explained in Chapter 8, a machine language is a set of binary codes corresponding to the basic tasks that a CPU can perform. In essence, each machine-language instruction specifies how various hardware components must be configured in order for a CPU cycle to perform a particular computation. Thus, we could define machine-language instructions for our simulator by enumerating all the physical settings of the knobs and switches. For example, the settings:

A Bus = R0	ALU Switch = closed
B Bus = R1	MMIn Switch = open
ALU = A+B	MMOut Switch = open
C Bus = R2	C Switch = closed

would define a configuration in which the contents of R0 and R1 are added and stored back in R2. This notation might suffice to control the behavior of a very simple machine, such as the one represented in our simulator; however, real-world CPUs contain an extremely large number of physical components, and specifying the status of all these parts during every CPU cycle would be impossible. Furthermore, since machine-language instructions are stored in memory along with data, the instructions must ultimately be represented as bit patterns.

Figure 14.10 describes a simple machine language that has been designed for our simulator. Since the main memory locations in our simulator can hold a maximum of 16 bits, our language represents each instruction as a 16-bit pattern. The initial bits indicate the type of task that the CPU must perform, whereas the subsequent bits indicate the registers and/ or memory locations involved in the task. Since there are only four registers, two bits suffice to represent a register number; since there are 32 main memory locations, five bits suffice to represent a memory address. For instance, all instructions that involve adding the contents of two registers begin with the bit pattern: 1010000100. The final six bits of an addition instruction represent the destination register (i.e., the register where the result will be stored) and the source registers (i.e., the registers whose contents will be added by the ALU), respectively. As an example, suppose that you wanted to add the contents of R0 and R1 and then store the result in R2—i.e., $R2 = R0 + R1$. The bit patterns for R2 ($2 = 10_2$), R0 ($0 = 00_2$), and R1 ($1 = 01_2$) would be appended to the initial bit pattern for addition (1010000100), yielding the machine-language instruction:

1010000100100001. Similarly, if the intent were $R3 = R0 + R1$, then the bit pattern for R3 (3 = 11₂) would replace that of R2: 1010000100110001.

Operation	Machine-Language Instruction	Example
add contents of two registers, store result in another register e.g., $R0 = R1 + R2$	1010000100 RR RR RR	1010000100 00 01 10 will add contents of R1 (01) and R2 (10), then store the result in R0 (00)
subtract contents of two registers, store result in another register e.g., $R0 = R1 - R2$	1010001000 RR RR RR	1010001000 00 01 10 will take contents of R1 (01), subtract R2 (10), then store the result in R0 (00)
load contents of memory location into register e.g., $R3 = M[5]$	100000010 RR MMMM	100000010 11 00101 will load contents of memory location 5 (00101) into R3 (11)
store contents of register in memory location e.g., $M[5] = R3$	100000100 RR MMMM	100000100 11 00101 will store contents of memory location 7 (00101) in R3 (11)
move contents of one register into another register e.g., $R1 = R0$	100100010000 RR RR	100100010000 01 00 will move contents of R0 (00) to R1 (01)
halt the machine	1111111111111111	N/A

Figure 14. 10: Machine Language for Computer Simulator

The first two machine-language instructions in Figure 14.10 correspond to tasks that users can perform with the CPU Datapath simulator—i.e., selecting an ALU operation to be executed and the registers to be operated on during a CPU cycle. The next three instructions correspond to tasks that users can perform with the datapath and memory version of the simulator—i.e., controlling the flow of information between the main memory and the datapath. The last instruction, HALT, tells the Control Unit when a sequence of instructions terminates. Of course, a real CPU would require many more instructions than these. For example, if a CPU executes programs that include conditional statements (such as JavaScript if statements and while loops), its machine language must support branching instructions that allow the CPU to jump from one instruction to another. However, Figure 14.10's limited instruction set is sufficient to demonstrate the workings of a basic CPU and its Control Unit.

Control Unit

Once a uniform machine language for a particular CPU is established, instructions can be stored in main memory along with data. It is the job of the Control Unit to obtain each machine-language instruction from memory, interpret its meaning, carry out the specified CPU cycle, and then move on to the next instruction. Since instructions and data are both stored in the same

memory, the Control Unit must be able to recognize where a sequence of instructions begins and ends. In real computers, this task is usually performed by the operating system, which maintains a list of each program in memory and its location. For simplicity, our simulator assumes that the first instruction is stored in memory location 0. The end of the instruction sequence is explicitly identified using the HALT bit pattern.

In order to track the execution of an instruction sequence, the Control Unit maintains a Program Counter (PC), which stores the address of the next instruction to be executed. Since we are assuming that all programs start at address 0, the PC's value is initialized to 0 before program execution begins. When the Control Unit needs to fetch and execute an instruction, it accesses the PC and then obtains the instruction stored in the corresponding memory location. After the Control Unit fetches the instruction, the PC is incremented so that it identifies the next instruction in the sequence.

The steps carried out by the Control Unit can be defined as a general algorithm, in which instructions are repeatedly fetched and executed:

Fetch-Execute Algorithm carried out by the Control Unit:

1. Initialize PC = 0.
2. Fetch the instruction stored at memory location PC, and set PC = PC + 1.
3. As long as the current instruction is not the HALT instruction:
 - a. Decode the instruction – that is, determine the CPU hardware settings required to carry it out.
 - b. Configure the CPU hardware to match the settings indicated in the instruction.
 - c. Execute a CPU datapath cycle using those settings.
 - d. When the cycle is complete, fetch the next instruction from memory location PC, and set PC = PC + 1.

For example, suppose that main memory contained the program and data shown in Figure 14.11.

0:	1000000100000101	// load memory location 5 into R0
1:	1000000100100110	// load memory location 6 into R1
2:	1010000100100001	// add R0 and R1, store result in R2
3:	1000001001000111	// copy R2 to memory location 7
4:	1111111111111111	// halt
5:	0000000000001001	// data to be added: 9
6:	0000000000000001	// data to be added: 1
7:	0000000000000000	// location where sum is to be stored

Figure 14.11: Machine-language program for adding two numbers in memory.

The first five memory locations (addresses 0 through 4) contain machine-language instructions for adding two numbers and storing their sum back in memory. The numbers to be added are stored in memory locations 5 and 6. To execute this program, the Control Unit would carry out the following steps:

1. First, the Program Counter is initialized: $PC = 0$.
2. The instruction at memory location 0 (corresponding to the current value of PC) is fetched, and the PC is incremented: $PC = 0 + 1 = 1$.
3. Since this instruction (1000000100000101) is not a HALT instruction, it is decoded: the CPU hardware is configured so that it will load the contents of memory location 5 into register R0, and a CPU cycle is executed.
4. The next instruction (at memory location 1, corresponding to the current value of PC) is fetched, and the PC is incremented: $PC = 1 + 1 = 2$.
5. Since this instruction (1000000100100110) is not a HALT instruction, it is decoded: the CPU hardware is configured so that it will load the contents of memory location 6 into register R1, and a CPU cycle is executed.
6. The next instruction (at memory location 2, corresponding to the current value of PC) is fetched, and the PC is incremented: $PC = 2 + 1 = 3$.
7. Since this instruction (1010000100100001) is not a HALT instruction, it is decoded: the CPU hardware is configured so that it will add the contents of registers R0 and R1 and store the result in register R2, and a CPU cycle is executed.
8. The next instruction (at memory location 3, corresponding to the current value of PC) is fetched, and the PC is incremented: $PC = 3 + 1 = 4$.
9. Since this instruction (1000001001000111) is not a HALT instruction, it is decoded: the CPU hardware is configured so that it will copy the contents of register R2 to memory location 7, and a CPU cycle is executed.
10. The next instruction (at memory location 4, corresponding to the current value of PC) is fetched, and the PC is incremented: $PC = 4 + 1 = 5$.
11. Since this instruction (1111111111111111) is a HALT instruction, the Control Unit recognizes the end of the program and stops executing.

Stored-Program Computer Simulator

The Stored-Program Computer Simulator (accessible at <http://www.creighton.edu/~csc107/Sims/computer.html>) models the behavior of a complete, stored-program computer. Instructions and data can be entered into memory, with the first instruction assumed to be at memory location 0. The Control Unit is responsible for fetching and interpreting the machine-language instructions, as well as carrying out the tasks specified by those instructions.

The simulator contains several display boxes to illustrate the Control Unit's inner workings. As we described in the previous section, the Program Counter (PC) lists the address of the next instruction to be executed. In addition to the PC, CPUs also maintain an Instruction Register (IR), which lists the instruction that the Control Unit is currently executing. The IR is displayed

in the simulator as an additional text box. Above these boxes, the simulator exhibits the actual knob and switch settings defined by the current instruction – this makes the correspondence between the machine-language instruction and the CPU hardware settings more obvious. Knob settings are specified as 2-bit binary numbers: 00 represents a knob pointing straight up, 01 represents a knob pointing to the right, 10 represents a knob pointing down, and 11 represents a knob pointing to the left. The four switch settings are represented by a four-bit pattern, with a 1 bit indicating a closed switch and a 0 bit indicating an open switch (the topmost switch in the simulator, the C Bus, corresponds to the first bit, followed by the three remaining switches from left to right).

Figures 14.12 through 14.17 demonstrate using the simulator to execute the example machine-language program from Figure 14.11.

- Figure 14.12 depicts the initial state of the simulator. The machine-language instructions are stored in main memory, starting at address 0. The data required to execute the instructions is also stored in memory, at addresses 5 and 6, immediately following the last instruction. Within the Control Unit, the Program Counter (PC) is initialized to 0, so the instruction at address 0 will be the first to be loaded and executed. To assist the user, the page includes a link to a reference page that summarizes all of the machine and assembly language instructions, labeled "Machine/Assembly Language Instructions."
- Figure 14.13 portrays the simulator after the first instruction has been executed. Within the Control Unit, the instruction from address 0 has been loaded into the Instruction Register and translated into the knob and switch settings required to carry out the specified CPU datapath cycle. Once the Control Unit determines the correct knob and switch settings, it carries out the corresponding CPU datapath cycle. In this case, the value from memory location 5, the number 9, is loaded into register R0. Note that the PC is automatically incremented after the instruction has been executed, so the next instruction to be fetched and executed will be the instruction at address 1.
- Figure 14.14 shows the simulator after the next instruction, from memory location 1, has been executed. Here, the value from memory location 6, the number 1, is loaded into register R1.
- Figure 14.15 depicts the result of executing the next instruction, from memory location 2, which adds the contents of registers R0 and R1 and stores the result in register R2.
- Figure 14.16 portrays the result of executing the next instruction, from memory location 3, which copies the result of the addition, stored in R2, into memory location 7.
- Finally, Figure 14.17 shows the computer after the program terminates. Recall that the HALT instruction 1111111111111111 tells the Control Unit to stop processing. Since no datapath cycle is executed once the HALT instruction is recognized by the Control Unit, the knob and switch settings within the datapath are not changed from the previous cycle.

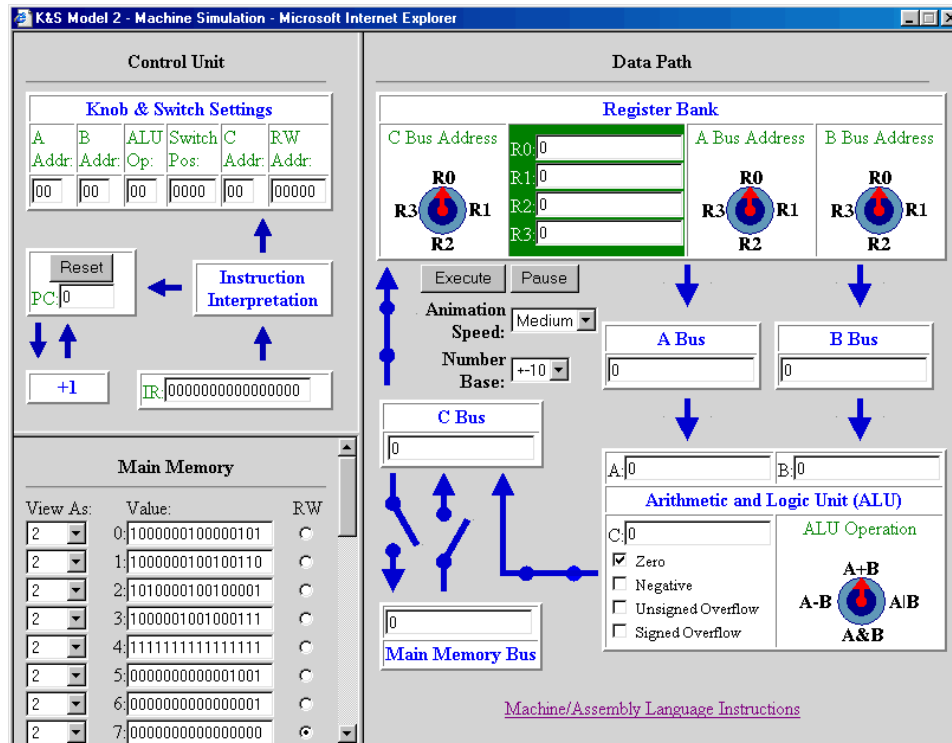


Figure 14. 12: Initial state of the simulator, with program stored in main memory.

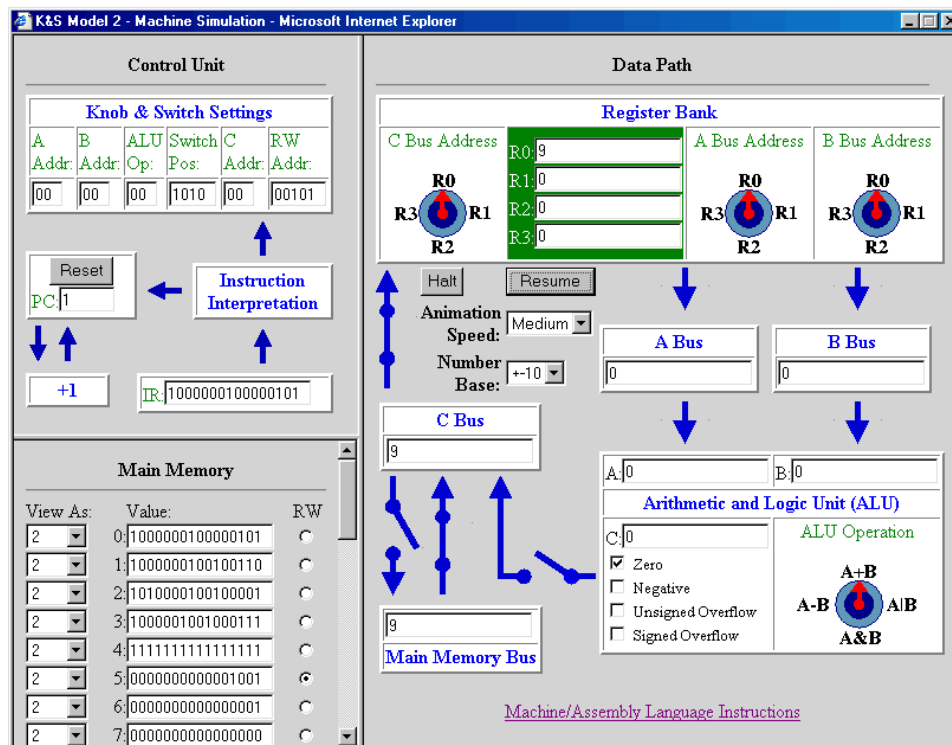


Figure 14. 13: Simulator after the first instruction has been executed (R0 = MM5).

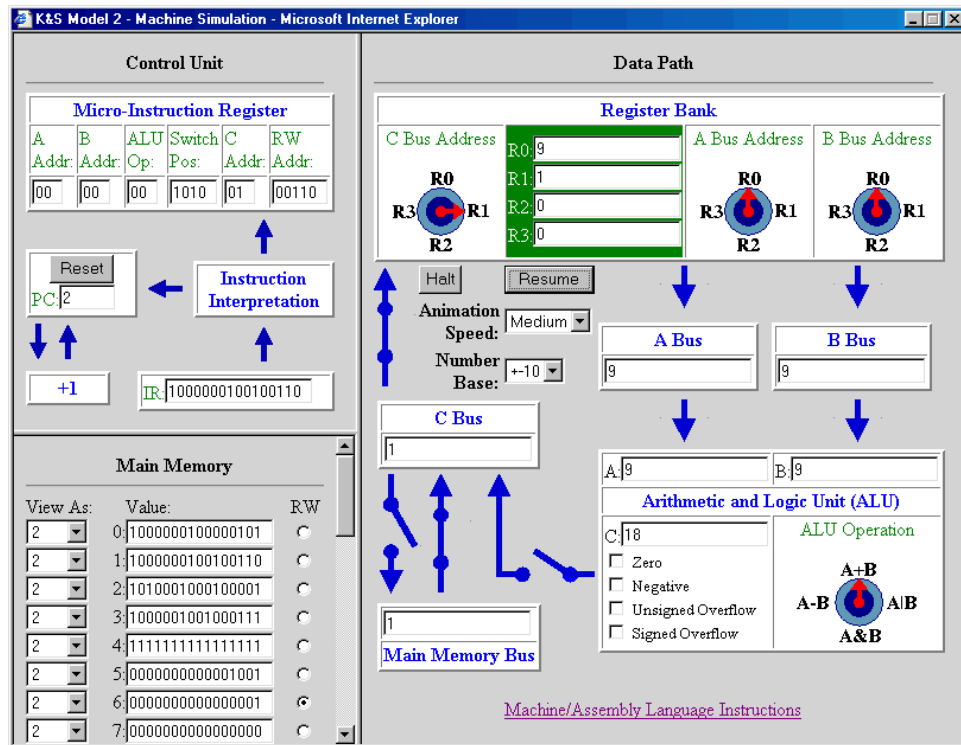


Figure 14. 14: Simulator after the second instruction has been executed (R1 = MM6).

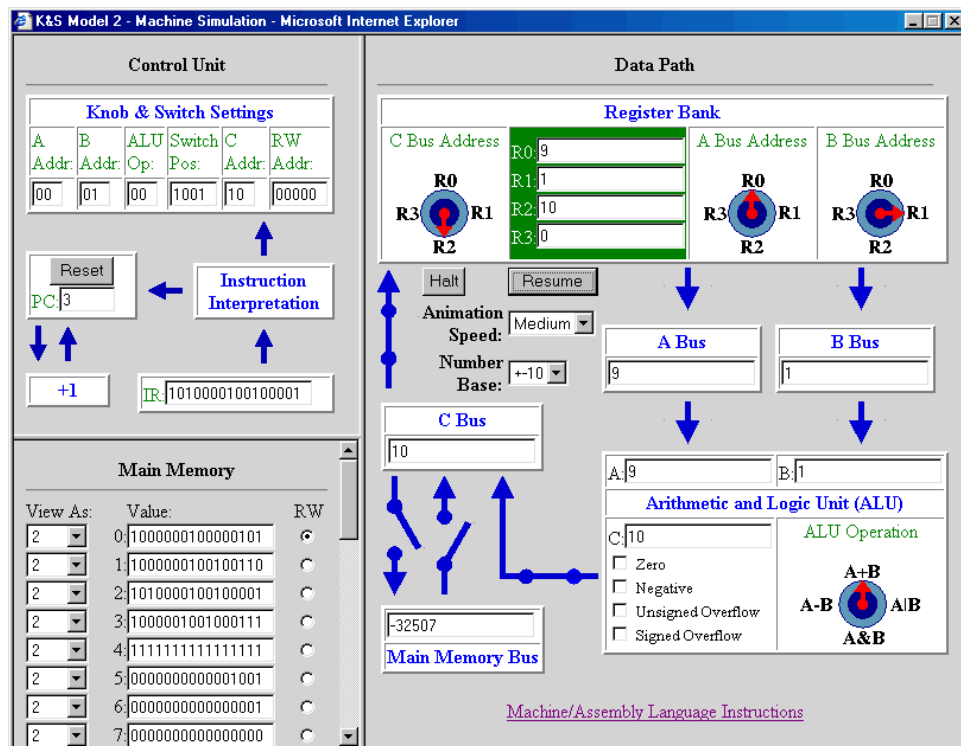


Figure 14. 15: Simulator after the third instruction has been executed (R2 = R0 + R1).

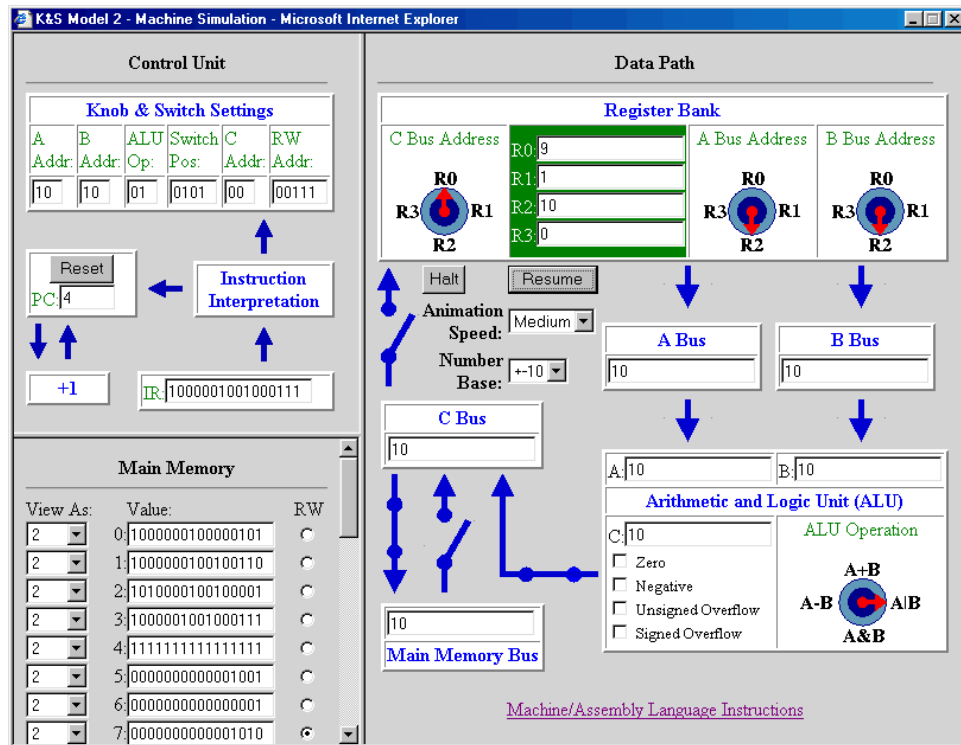


Figure 14. 16: Simulator after the fourth statement has been executed (MM7 = R2).

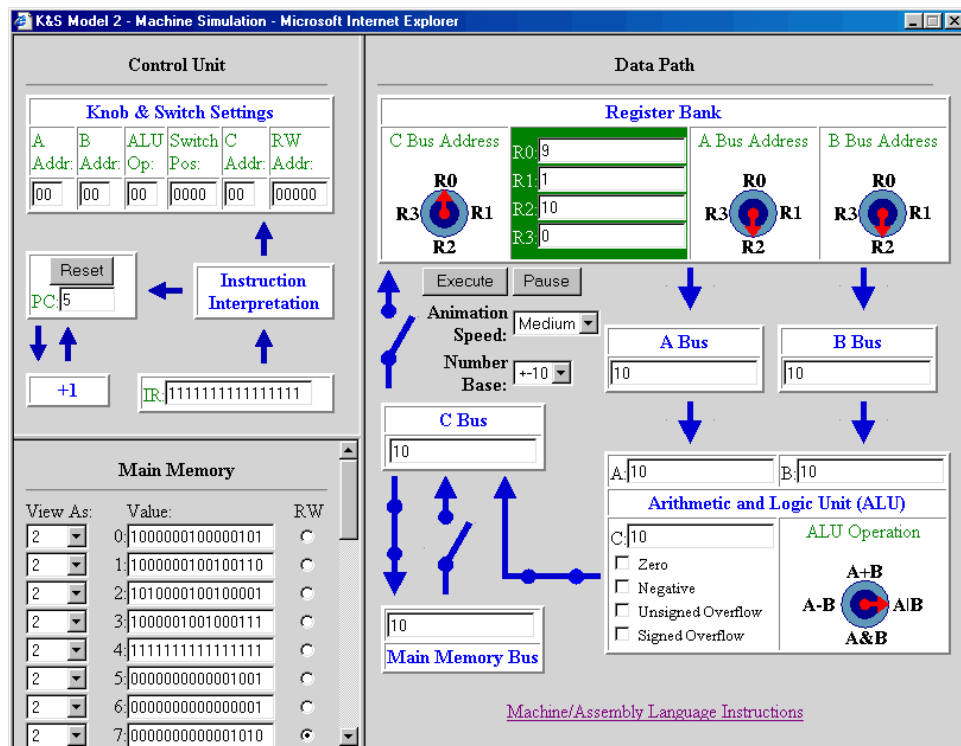


Figure 14. 17: Simulator after the fifth statement has been executed (HALT).

The simulator is designed so that the user can enter values in main memory as either decimal or binary numbers. By default, values entered by the user are assumed to be decimal numbers. However, the user can always select 2 from the `View As` box to the left of a memory location in order to view the contents in binary. Before entering a machine-language instruction in a memory cell, the user must first set the value of the `View As` box to 2, since machine-language instructions are represented in binary.

HANDS-ON EXERCISES: Experiment with this simulator until you are familiar with its behavior. Then, use the simulator to answer the following questions:

14.7. What task would the following machine-language program perform?

```
0: 1010001000000000
1: 1000001000000011
2: 1111111111111111
```

14.8. What sequence of machine-language instructions would cause the contents of the four registers to be copied into memory locations 7, 8, 9, and 10, respectively?

14.9. What sequence of machine-language instructions would cause the simulator to add the contents of memory locations 10, 11, and 12 and then store the result in memory location 13?

14.10. What do you think would happen if you forgot to place a HALT instruction at the end of a machine-language program? How would the Control Unit react? Use the simulator to test your prediction, then report the results.

The Role of Input/Output Devices

To complete our description of the stored-program computer, we must at least briefly discuss the role of input and output devices. Input devices such as keyboards, mice, and scanners allow the user to communicate with the computer by entering data and instructions, which are then stored in memory and accessed by the CPU. Likewise, output devices such as display screens, speakers, and printers allow the user to view the current status of the computer and access computation results that are stored in memory.

Computers that are designed to run one program at a time, such as the first programmable computers (introduced in the 1950s) and the first personal computers (introduced in the 1970s), provide relatively straightforward methods of user interaction. The user enters program instructions and data directly into main memory locations via input devices such as keyboards or tape readers. Then, by flipping a switch or entering a specific command, the user instructs the CPU to fetch the program instructions from memory and execute them in sequential order. Once a particular computation has been completed, the user can view its result by sending the contents of memory to a printer or display screen. This process is closely modeled by our simulator, in which the user enters instructions and data by typing them in main memory boxes, then initiates execution by clicking a button. In the case of the simulator, however, the final step of sending

results to an output device is not necessary – the contents of the registers and main memory are already visible.

As we discussed in Chapters 6 and 10, most modern computers allow for multitasking, meaning that multiple programs can be loaded in main memory and be active simultaneously. When multiple programs are to be executed, user interactions and the computations that result from those interactions become more complex. In particular, the instructions and data associated with each program must be loaded into separate portions of main memory. When the user switches from one program to another (say by clicking on a different window in the graphical user interface), the CPU must save the state of the current program and be able to locate the portion of memory associated with the new program. These tasks, and many others involving the coordination of programs and CPU processing, are managed by the operating system.

Machine v. Assembly Languages

It is important to note that the machine language and hardware configurations associated with our simulator are much simpler than those of any actual computer. A real machine language might encompass tens or hundreds of instructions, and a real CPU might contain hundreds or thousands of configurable components. However, our model is sufficient to demonstrate computer behavior at its lowest level.

The simulator is also useful in representing the difficulty and tedium of programming in a machine language. As you learned in Chapter 8, writing, debugging, and understanding bit-sequence instructions can be mind-numbing work. Over the past fifty years, computer programming has advanced significantly, and most modern programmers are able to avoid direct machine-language programming. Some of the earliest programmer tools were assembly languages, which substitute words for bit patterns, allowing the programmer to write:

```
ADD R0 R1 R2
```

instead of the machine-language instruction:

```
1010000100000110
```

It is much easier for programmers to remember and understand assembly-language instructions than patterns of 0s and 1s. Furthermore, most assembly languages support the use of variable names, enabling programmers to specify memory locations by descriptive names, rather than by numerical addresses. This greatly simplifies the programmer's task, since she no longer needs to worry about the physical location of data and how locations might shift as new instructions are inserted into memory.

Figure 14.18 lists one possible set of assembly-language instructions that correspond to the machine-language instructions from Figure 14.11.

Operation	Machine-Language Instruction	Assembly-Language Instruction
add contents of two registers, then store result in another register e.g., $R0 = R1 + R2$	1010000100 RR RR RR e.g., 1010000100 00 01 10	ADD [REG] [REG] [REG] e.g., ADD R0 R1 R2
subtract contents of two registers, then store result in another register e.g., $R0 = R1 - R2$;	1010001000 RR RR RR e.g., 1010001000 00 01 10	SUB [REG] [REG] [REG] e.g., SUB R0 R1 R2
load contents of memory location into register e.g., $R3 = MM5$	100000010 RR MMMM e.g., 100000010 11 00101	LOAD [REG] [MEM] e.g., LOAD R3 5
store contents of register into memory location e.g., $MM5 = R3$	100000100 RR MMMM e.g., 100000100 11 00101	STORE [MEM] [REG] e.g., STORE 5 R3
move contents of one register into another register e.g., $R1 = R0$	100100010000 RR RR e.g., 100100010000 01 00	MOVE [REG] [REG] e.g., MOVE R1 R0
halt the machine	1111111111111111	HALT

Figure 14. 18: Assembly-language instructions.

Within the Stored Program Computer Simulator, the user may enter assembly-language instructions directly in memory. The default mode for displaying the contents of memory locations, labeled "Auto" in the View As box, will automatically recognize assembly-language instructions and will display them as text (changing the label to "Inst" to acknowledge that they are instructions). After inputting the instructions, the user can switch between viewing the instruction in assembly- or machine-language form by selecting Inst (for assembly-language instructions) or 2 (for machine-language instructions in binary form) in the View As box to the left of the instruction. For example, Figure 14.19 depicts the same program that is pictured in Figure 14.12, but here the instructions are formatted as Inst.

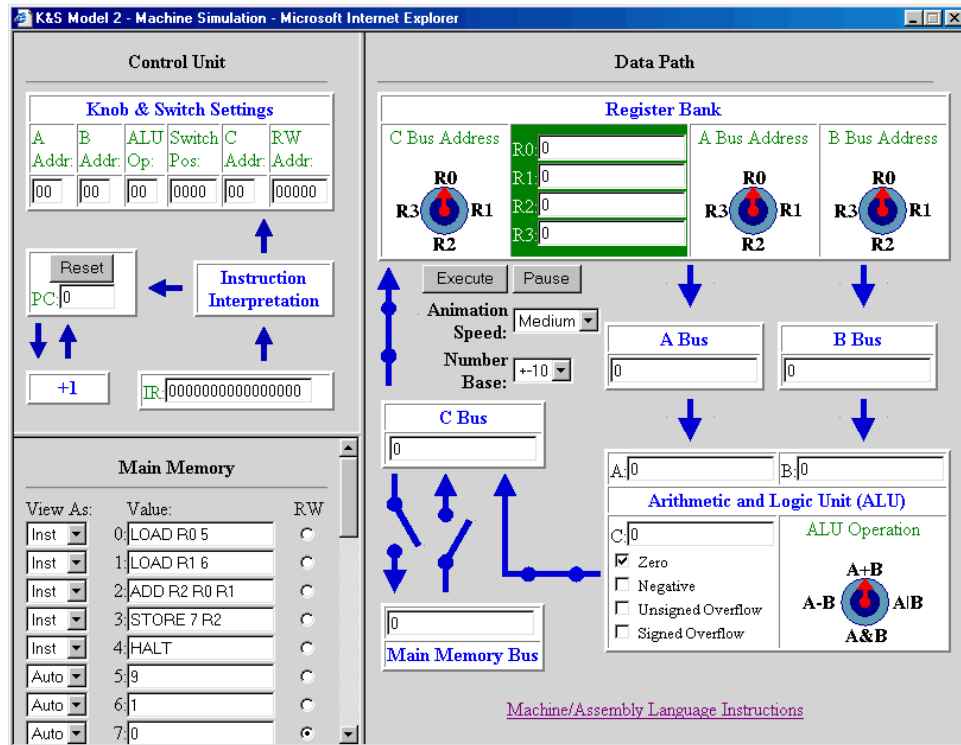


Figure 14. 19: Assembly-language program displayed in computer simulator.

HANDS-ON EXERCISES: Experiment with the simulator until you are comfortable with the correspondence between machine-language and assembly-language instructions. Then, use the simulator to answer the following questions:

- 14.11. What sequence of assembly-language instructions corresponds to the machine-language instruction set from Exercise 14.7?
- 14.12. What sequence of assembly-language instructions corresponds to the machine-language instruction set you wrote in Exercise 14.9?
- 14.13. Write a sequence of assembly-language instructions that multiplies the contents of memory location 10 by four. For example, if the number 10 were stored in memory location 10, executing your instructions would cause the simulator to store 40 there. Note: although the ALU Operation knob does not provide a multiplication setting, a number can be multiplied via repeated additions (e.g., $10 * 4 = 10 + 10 + 10 + 10$).

Looking Ahead...

In this chapter, you studied the internal workings of computers, focusing on the CPU, main memory, and their interactions. By experimenting with the various simulators, you saw how the CPU breaks up even the most complex computing tasks into sequences of very simple instructions, each of which can be executed during a single CPU datapath cycle. Although buses that connect the CPU to main memory enable the manipulation of large amounts of data, the computer must transfer each data value into a CPU register, perform operations on the value within the ALU, and then store the result of the computation back in main memory. The combination of a Control Unit and main memory allows computers to process stored programs. This behavior was modeled by our most sophisticated simulator, in which programs consisting of machine-language instruction sequences were stored in memory, fetched and decoded by the Control Unit, then executed in a series of CPU datapath cycles.

Building upon your general understanding of computer components and their organization, Chapter 16 will delve deeper into the details of their design and construction. In particular, you will study the construction and behavior of transistors and integrated circuits. However, before you consider how these technologies are used to build the hardware components of computer, Chapter 15 will describe a general methodology for designing and implementing software components. The object-oriented approach to software development attempts to simplify the task of designing and testing software systems by focusing on programming structures that model real-world objects. For example, a JavaScript string is an object that models a word or phrase, with useful operations on a string (such as capitalizing it or extracting a substring) provided as part of the language. Chapter 15 will describe the JavaScript string type, and explore applications that involve storing and manipulating text

Review Questions

1. TRUE or FALSE? Any piece of memory that is used to store the sum of numeric values is known as a register.
2. TRUE or FALSE? The path that data follows within the CPU, traveling along buses from registers to the ALU and then back to registers, is known as the *CPU* datapath.
3. TRUE or FALSE? All modern CPUs provide the same set of basic operations that can be executed in a single CPU cycle.
4. TRUE or FALSE? The size of main memory is generally measured in MHz or GHz.
5. TRUE or FALSE? Suppose you wish to add two numbers that are stored in memory. Before the Arithmetic Logic Unit (ALU) can add the numbers, they must first be transferred to registers within the CPU.
6. TRUE or FALSE? In real computers, it takes roughly the same amount of time to transfer data from main memory to registers as it does to add two numbers in registers.
7. TRUE or FALSE? Within the CPU, the Control Unit is responsible for fetching machine-language instructions from memory, interpreting their meaning, and carrying out the specified CPU cycles.
8. TRUE or FALSE? Suppose a CPU contains eight registers. Within a machine-language instruction, at least three bits would be required to uniquely identify one of the registers.
9. TRUE or FALSE? In a multitasking computer, the Program Counter keeps track of how many programs are currently loaded into main memory.
10. TRUE or FALSE? In a stored-program computer, both machine-language instructions and the data operated on by those instructions can reside in main memory at the same time.
11. Name the three subunits of the CPU, and describe the role of each subunit in carrying out computations.
12. Describe how data values move around the CPU datapath and what actions occur during a single CPU cycle. How does the datapath relate to CPU speed?
13. Consider two computer systems that are identical except for their CPUs. System 1 contains a 1.8 GHz Pentium 4, whereas System 2 contains a 1.8 GHz PowerPC processor. Will these two systems always require the same amount of time to execute a given program? Justify your answer.
14. Consider the following tasks: (1) adding 100 numbers stored in main memory, and (2) adding a number to itself 100 times. Although both tasks require 100 additions, the second would be executed much more quickly than the first would. Why?
15. Machine languages are machine-specific, meaning that each type of computer has its own machine language. Explain why this is the case.
16. Within the Control Unit, what is the role of the Program Counter (PC)? That is, how is the PC used in fetching and executing instructions?
17. In a stored-program computer, both instructions and data are stored in main memory. How does the Control Unit know where the program instructions begin? How does it know where the instructions end?
18. Describe two advantages of assembly languages over machine languages.

References

- Brain, Marshall. "How Microprocessors Work." *HowStuffWorks*, January 2003.
Online at <http://www.howstuffworks.com/microprocessor.htm>
- Brought, G. "Computer Organization in the Breadth First Course." *Journal of Computing in Small Colleges* 16(4), 2001.
- Brought, Grant, and David Reed. "The Knob & Switch Computer: A Computer Architecture Simulator for Introductory Computer Science." *ACM Journal of Educational Resources in Computing* 1(4), 2001.
- Malone, Michael. *The Microprocessor: A Biography*. New York, NY: Springer Verlag, 1995.
- "The PC Technology Guide – Components/Processors". *PCTechGuide*, December 2002.
Online at <http://www.pctechguide.com/02procs.htm>
- Stallings, William. *Computer Organization and Architecture: Designing for Performance*, 6th Edition. Upper Saddle River, NJ: Prentice Hall, 2003.
- Tanenbaum, Andrew S. *Structured Computer Organization*, 4th Edition. Upper Saddle River, NJ: Prentice Hall, 1999.