Chapter 11: Computer System Design

This chapter presents the design of a simple computer system that will illustrate the use of many of the Verilog modeling techniques covered in this book. The goal of this chapter is not to provide an in-depth coverage of modern computer architecture, but rather to present a simple operational computer that can be implemented in Verilog to show how to use many of the modeling techniques covered thus far. The chapter begins with some architectural definitions so that consistent terminology can be used throughout the computer design example.

Learning Outcomes—After completing this chapter, you will be able to:

11.1 Describe the basic components and operation of computer hardware.
11.2 Describe the basic components and operation of computer software.
11.3 Design a fully operational computer system using Verilog.

11.1 Computer Hardware

A computer accomplishes tasks through an architecture that uses both *hardware* and *software*. The hardware in a computer consists of many of the elements that we have covered so far. These include registers, arithmetic and logic circuits, finite state machines, and memory. What makes a computer so useful is that the hardware is designed to accomplish a predetermined set of **instructions**. These instructions are relatively simple, such as moving data between memory and a register or performing arithmetic on two numbers. The instructions comprise binary codes that are stored in a memory device and represent the sequence of operations that the hardware will perform to accomplish a task. This sequence of instructions is called a computer **program**. What makes this architecture so useful is that the preexisting hardware can be *programmed* to perform an almost unlimited number of tasks by simply defining the sequence of instructions to be executed. The process of designing the sequence of instructions, or program, is called *software development* or *software engineering*.

The idea of a general-purpose computing machine dates back to the nineteenth century. The first computing machines were implemented with mechanical systems and were typically analog in nature. As technology advanced, computer hardware evolved from electromechanical switches to vacuum tubes and ultimately to integrated circuits. These newer technologies enabled switching circuits and provided the capability to build binary computers. Today's computers are built exclusively with semiconductor materials and integrated circuit technology. The term *microcomputer* is used to describe a computer that has its processing hardware implemented with integrated circuitry. Nearly all modern computers are binary. Binary computers are designed to operate on a fixed set of bits. For example, an 8-bit computer would perform operations on 8-bits at a time. This means it moves data between registers and memory and performs arithmetic and logic operations in groups of 8-bits.

Computer hardware refers to all of the physical components within the system. This hardware includes all circuit components in a computer such as the memory devices, registers, and finite state machines. Figure 11.1 shows a block diagram of the basic hardware components in a computer.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-030-10552-5_11

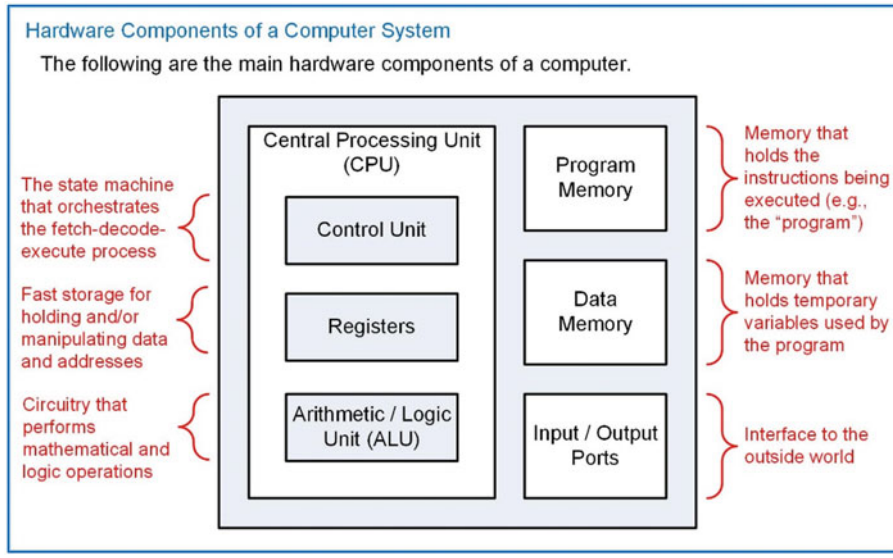


Fig. 11.1
Hardware components of a computer system

11.1.1 Program Memory

The instructions that are executed by a computer are held in *program memory*. Program memory is treated as read only during execution in order to prevent the instructions from being overwritten by the computer. Programs are typically held in non-volatile memory so that the computer system does not lose its program when power is removed. Modern computers will often copy a program from non-volatile memory (e.g., a hard disk drive) to volatile memory (i.e., SRAM or DRAM) after startup in order to speed up instruction execution as volatile memory is often a faster technology.

11.1.2 Data Memory

Computers also require *data memory*, which can be written to and read from during normal operation. This memory is used to hold temporary variables that are created by the software program. This memory expands the capability of the computer system by allowing large amounts of information to be created and stored by the program. Additionally, computations can be performed that are larger than the width of the computer system by holding interim portions of the calculation (e.g., performing a 128-bit addition on a 32-bit computer). Data memory is typically implemented with volatile memory as it is often faster than read-only memory technology.

11.1.3 Input/Output Ports

The term *port* is used to describe the mechanism to get information from the output world into or out of the computer. Ports can be input, output, or bidirectional. I/O ports can be designed to pass information in a serial or parallel format.

11.1.4 Central Processing Unit

The *central processing unit* (CPU) is considered the *brains* of the computer. The CPU handles reading instructions from memory, decoding them to understand which instruction is being performed, and executing the necessary steps to complete the instruction. The CPU also contains a set of registers

that are used for general-purpose data storage, operational information, and system status. Finally, the CPU contains circuitry to perform arithmetic and logic operations on data.

11.1.4.1 Control Unit

The *control unit* is a finite state machine that controls the operation of the computer. This FSM has states that perform fetching the instruction (i.e., reading it from program memory), decoding the instruction, and executing the appropriate steps to accomplish the instruction. This process is known as *fetch, decode, and execute* and is repeated each time an instruction is performed by the CPU. As the control unit state machine traverses through its states, it asserts control signals that move and manipulate data in order to achieve the desired functionality of the instruction.

11.1.4.2 Data Path: Registers

The CPU groups its registers and ALU into a subsystem called the *data path*. The data path refers to the fast storage and data manipulations within the CPU. All of these operations are initiated and managed by the control unit state machine. The CPU contains a variety of registers that are necessary to execute instructions and hold status information about the system. Basic computers have the following registers in their CPU:

- **Instruction Register (IR)**—The instruction register holds the current binary code of the instruction being executed. This code is read from program memory as the first part of instruction execution. The IR is used by the control unit to decide which states in its FSM to traverse in order to execute the instruction.
- **Memory Address Register (MAR)**—The memory address register is used to hold the current address being used to access memory. The MAR can be loaded with addresses in order to fetch instructions from program memory or with addresses to access data memory and/or I/O ports.
- **Program Counter (PC)**—The program counter holds the address of the current instruction being executed in program memory. The program counter will increment sequentially through the program memory reading instructions until a dedicated instruction is used to set it to a new location.
- **General-Purpose Registers**—These registers are available for temporary storage by the program. Instructions exist to move information from memory into these registers and to move information from these registers into memory. Instructions also exist to perform arithmetic and logic operations on the information held in these registers.
- **Condition Code Register (CCR)**—The condition code register holds status flags that provide information about the arithmetic and logic operations performed in the CPU. The most common flags are *negative* (N), zero (Z), two's complement overflow (V), and carry (C). This register can also contain flags that indicate the status of the computer, such as if an interrupt has occurred or if the computer has been put into a low-power mode.

11.1.4.3 Data Path: Arithmetic Logic Unit (ALU)

The *arithmetic logic unit* is the system that performs all mathematical (i.e., addition, subtraction, multiplication, and division) and logic operations (i.e., and, or, not, shifts). This system operates on data being held in CPU registers. The ALU has a unique symbol associated with it to distinguish it from other functional units in the CPU.

Figure 11.2 shows the typical organization of a CPU. The registers and ALU are grouped into the data path. In this example, the computer system has two general-purpose registers called A and B. This CPU organization will be used throughout this chapter to illustrate the detailed execution of instructions.

Typical CPU Organization

A CPU is functionally organized into a control unit and a data path. The control unit contains the FSM to orchestrate the fetch-decode-execute process. The registers and ALU are grouped into a unit called the data path. The control unit sends control signals to the data path to move and manipulate data. The control unit uses status signals from the data path to decide which states to traverse in its FSM.

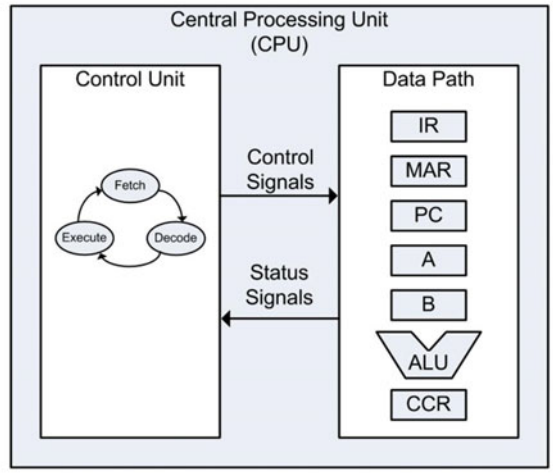


Fig. 11.2
Typical CPU organization

11.1.5 A Memory Mapped System

A common way to simplify moving data in or out of the CPU is to assign a unique address to all hardware components in the memory system. Each input/output port and each location in both program and data memory are assigned a unique address. This allows the CPU to access everything in the memory system with a dedicated address. This reduces the number of lines that must pass into the CPU. A *bus system* facilitates transferring information within the computer system. An address bus is driven by the CPU to identify which location in the memory system is being accessed. A data bus is used to transfer information to/from the CPU and the memory system. Finally, a control bus is used to provide other required information about the transactions such as *read* or *write* lines. Figure 11.3 shows the computer hardware in a memory mapped architecture.

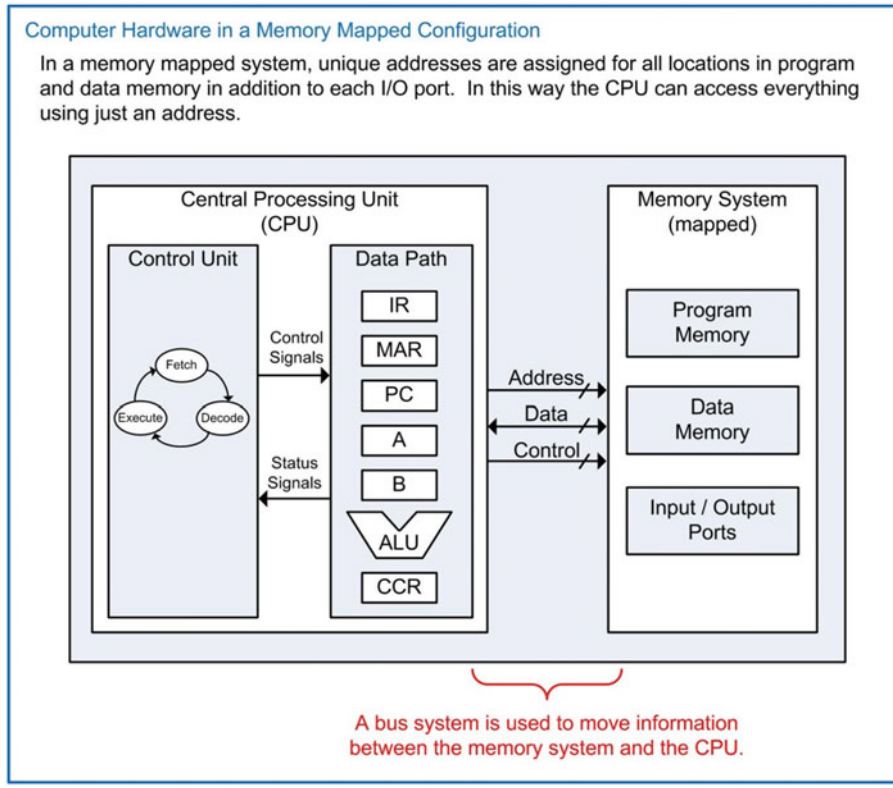
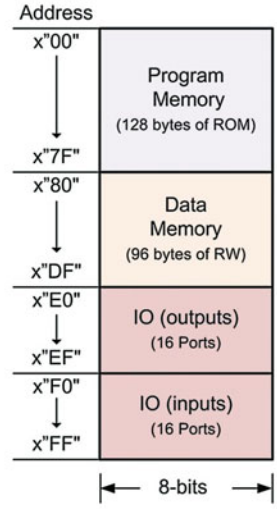


Fig. 11.3
Computer hardware in a memory mapped configuration

To help visualize how the memory addresses are assigned, a *memory map* is used. This is a graphical depiction of the memory system. In the memory map, the ranges of addresses are provided for each of the main subsections of memory. This gives the programmer a quick overview of the available resources in the computer system. Example 11.1 shows a representative memory map for a computer system with an address bus with a width of 8-bits. This address bus can provide 256 unique locations. For this example, the memory system is also 8-bits wide, thus the entire memory system is 256 $\times$ 8 in size. In this example, 128 bytes are allocated for program memory; 96 bytes are allocated for data memory; 16 bytes are allocated for output ports; and 16 bytes are allocated for input ports.

Example: Memory Map for a 256x8 Memory System

The following is a memory map for an example 8-bit computer system.



Example 11.1
Memory map for a 256 × 8 memory system

Concept Check

CC11.1 Is the hardware of a computer programmed in a similar way to a programmable logic device?

(A) Yes. The control unit is reconfigured to produce the correct logic for each unique instruction just like a logic element in an FPGA is reconfigured to produce the desired logic expression.

(B) No. The instruction code from program memory simply tells the state machine in the control unit which path to traverse in order to accomplish the desired task.

11.2 Computer Software

Computer software refers to the instructions that the computer can execute and how they are designed to accomplish various tasks. The specific group of instructions that a computer can execute is known as its **instruction set**. The instruction set of a computer needs to be defined first before the computer hardware can be implemented. Some computer systems have a very small number of instructions in order to reduce the physical size of the circuitry needed in the CPU. This allows the CPU to execute the instructions very quickly but requires a large number of operations to accomplish a given task. This architectural approach is called a **reduced instruction set computer** (RISC). The alternative to this approach is to make an instruction set with a large number of dedicated instructions that can accomplish a given task in fewer CPU operations. The drawback of this approach is that the physical size of the CPU must be larger in order to accommodate the various instructions. This architectural approach is called a **complex instruction set computer** (CISC).

11.2.1 Opcodes and Operands

A computer instruction consists of two fields, an *opcode* and an *operand*. The opcode is a unique binary code given to each instruction in the set. The CPU decodes the opcode in order to know which instruction is being executed and then takes the appropriate steps to complete the instruction. Each opcode is assigned a **mnemonic**, which is a descriptive name for the opcode that can be used when discussing the instruction functionally. An operand is additional information for the instruction that may be required. An instruction may have any number of operands including zero. Figure 11.4 shows an example of how the instruction opcodes and operands are placed into program memory.

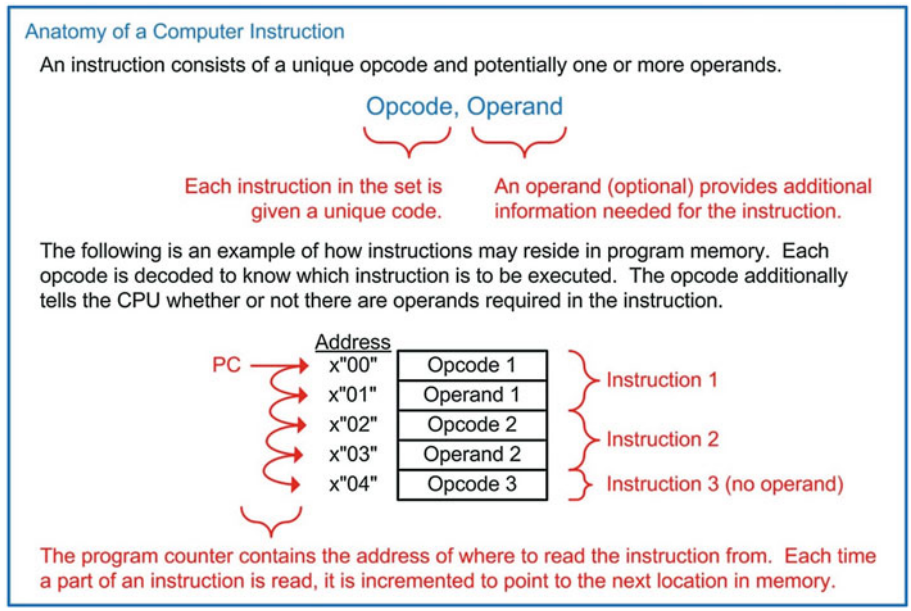


Fig. 11.4
Anatomy of a computer instruction

11.2.2 Addressing Modes

An *addressing mode* describes the way in which the operand of an instruction is used. While modern computer systems may contain numerous addressing modes with varying complexities, we will focus on just a subset of basic addressing modes. These modes are immediate, direct, inherent, and indexed.

11.2.2.1 Immediate Addressing (IMM)

Immediate addressing is when the operand of an instruction *is* the information to be used by the instruction. For example, if an instruction existed to put a constant into a register within the CPU using immediate addressing, the operand would *be* the constant. When the CPU reads the operand, it simply inserts the contents into the CPU register and the instruction is complete.

11.2.2.2 Direct Addressing (DIR)

Direct addressing is when the operand of an instruction contains the *address* of where the information to be used is located. For example, if an instruction existed to put a constant into a register within the CPU using direct addressing, the operand would contain the address of *where* the constant was located

in memory. When the CPU reads the operand, it puts this value out on the address bus and performs an additional read to retrieve the contents located at that address. The value read is then put into the CPU register and the instruction is complete.

11.2.2.3 Inherent Addressing (INH)

Inherent addressing refers to an instruction that does not require an operand because the opcode itself contains all of the necessary information for the instruction to complete. This type of addressing is used on instructions that perform manipulations on data held in CPU registers without the need to access the memory system. For example, if an instruction existed to increment the contents of a register (A), then once the opcode is read by the CPU, it knows everything it needs to know in order to accomplish the task. The CPU simply asserts a series of control signals in order to increment the contents of A and then the instruction is complete. Notice that no operand is needed for this task. Instead, the location of the register to be manipulated (i.e., A) is inherent within the opcode.

11.2.3 Classes of Instructions

There are three general classes of instructions: (1) loads and stores; (2) data manipulations; and (3) branches. To illustrate how these instructions are executed, examples will be given based on the computer architecture shown in Fig. 11.3.

11.2.3.1 Loads and Stores

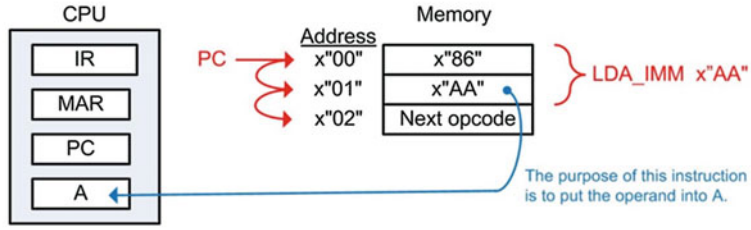
This class of instructions accomplishes moving information between the CPU and memory. A **load** is an instruction that moves information from memory *into* a CPU register. When a load instruction uses immediate addressing, the operand of the instruction *is* the data to be loaded into the CPU register. As an example, let's look at an instruction to load the general-purpose register A using immediate addressing. Let's say that the opcode of the instruction is x"86", has a mnemonic LDA_IMM, and is inserted into program memory starting at x"00". Example 11.2 shows the steps involved in executing the LDA_IMM instruction.

Example: Execution of an Instruction to "Load Register A Using Immediate Addressing"

A load instruction using immediate addressing will put the value of the operand into a CPU register. Let's create a program that will load register A in the CPU with the value x"AA". The program is as follows:

Using Mnemonics		Using Hex Values
LDA_IMM x"AA"	or	x"86" x"AA"

When the opcode and operand are put into program memory at x"00", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"00", meaning that this address is the location of the first instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the IR holds x"86" and the PC holds x"01".

Step 2 – Decode the instruction

The CPU decodes x"86" and understands that it is a "load A with immediate addressing". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"01") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is placed into register A. After this step, A=x"AA". Also in this step, the PC is incremented to point to the next location in memory (x"02"), which holds the opcode of the next instruction to be executed.

Example 11.2
Execution of an instruction to "load register A using immediate addressing"

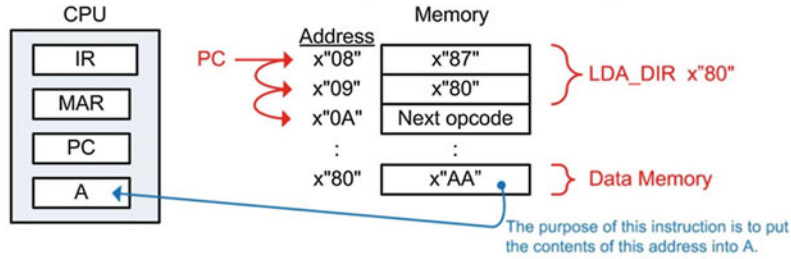
Now let's look at a load instruction using direct addressing. In direct addressing, the operand of the instruction is the *address* of where the data to be loaded resides. As an example, let's look at an instruction to load the general-purpose register A. Let's say that the opcode of the instruction is x"87", has a mnemonic LDA_DIR, and is inserted into program memory starting at x"08". The value to be loaded into A resides at address x"80", which has already been initialized with x"AA" before this instruction. Example 11.3 shows the steps involved in executing the LDA_DIR instruction.

Example: Execution of an Instruction to "Load Register A Using Direct Addressing"

A load instruction using direct addressing will put the value located at the address provided by the operand into a CPU register. Let's create a program that will load register A in the CPU with the contents located at address x"80", which has already been initialized to x"AA". The program is as follows:

Using Mnemonics		Using Hex Values
LDA_DIR x"80"	or	x"87" x"80"

When the opcode and operand are put into program memory at x"08", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"08", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the IR holds x"87" and the PC holds x"09".

Step 2 – Decode the instruction

The CPU decodes x"87" and understands that it is a "load A with direct addressing". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"09") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is the address that contains the value to be put into A. The operand is immediately put on the address bus using the MAR and another read is performed. The value read from address x"80" is placed into register A. After this step, A=x"AA". Also in this step, the PC is incremented to point to the next location in memory (x"0A"), which holds the opcode of the next instruction to be executed.

Example 11.3
Execution of an instruction to "load register A using direct addressing"

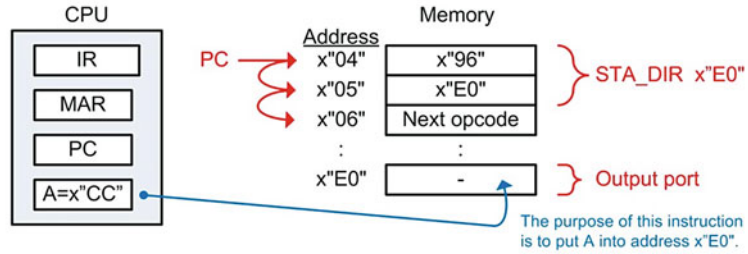
A **store** is an instruction that moves information from a CPU register *into* memory. The operand of a store instruction indicates the address of where the contents of the CPU register will be written. As an example, let's look at an instruction to store the general-purpose register A into memory address x"E0". Let's say that the opcode of the instruction is x"96", has a mnemonic STA_DIR, and is inserted into program memory starting at x"04". The initial value of A is x"CC" before the instruction is executed. Example 11.4 shows the steps involved in executing the STA_DIR instruction.

Example: Execution of an Instruction to "Store Register A Using Direct Addressing"

A store instruction using direct addressing will put the value held in a CPU register into memory at the address provided by the operand. Let's create a program that will store register A in the CPU to address location x"E0". We can assume A holds x"CC" prior to this instruction. The program is as follows:

Using Mnemonics		Using Hex Values
STA_DIR x"E0"	or	x"96" x"E0"

When the opcode and operand are put into program memory at x"04", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"04", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the IR holds x"96" and the PC holds x"05".

Step 2 – Decode the instruction

The CPU decodes x"96" and understands that it is a "store A with direct addressing". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"05") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is the address of where A will be written. The operand is immediately put on the address bus using the MAR, A is put on the data bus, and a write is performed. After this step, location x"E0" in memory contains x"CC". Also in this step, the PC is incremented to point to the next location in memory (x"06"), which holds the opcode of the next instruction to be executed. The write did not effect register A so it still contains x"CC" after the instruction completes.

Example 11.4
Execution of an instruction to "store register A using direct addressing"

11.2.3.2 Data Manipulations

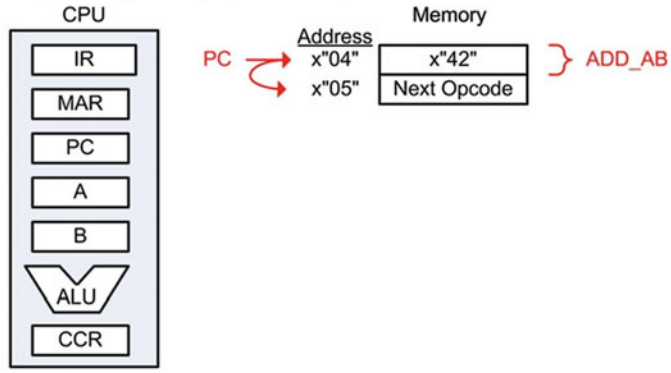
This class of instructions refers to ALU operations. These operations act on data that resides in the CPU registers. These instructions include arithmetic, logic operators, shifts and rotates, and tests and compares. Data manipulation instructions typically use inherent addressing because the operations are conducted on the contents of CPU registers and don't require additional memory access. As an example, let's look at an instruction to perform addition on registers A and B. The sum will be placed back in A. Let's say that the opcode of the instruction is x"42", has a mnemonic ADD_AB, and is inserted into program memory starting at x"04". Example 11.5 shows the steps involved in executing the ADD_AB instruction.

Example: Execution of an Instruction to "Add Registers A and B"

This instruction adds A and B and puts the sum back into A (A = A+B). This instruction does not require an operand because the inputs and output of the operation reside completely within the CPU. This type of instruction uses inherent addressing, meaning that the location of the information impacted is inherent in the opcode. Let's create a program to perform this addition. The program is as follows:

Using Mnemonics		Using Hex Values
ADD_AB	or	x"42"

When the opcode is put into program memory at x"04", it looks like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"04", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the PC holds x"05" and the IR holds x"42".

Step 2 – Decode the instruction

The CPU decodes x"42" and understands that it is an "Add A and B". It also knows that there is no operand associated with this instruction.

Step 3 – Execute the instruction

The CPU asserts the necessary control signals to route A and B to the ALU, performs the addition, and places the sum back into A. The CCR is also updated to provide additional status information about the operation.

Example 11.5
Execution of an instruction to "add registers A and B"

11.2.3.3 Branches

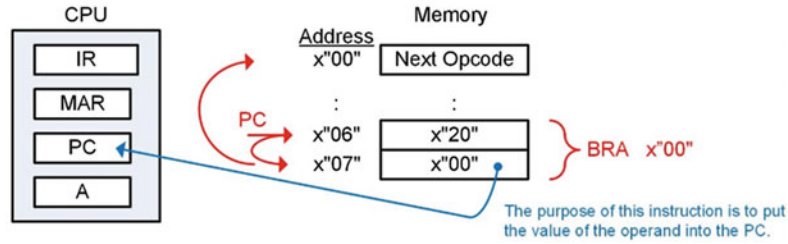
In the previous examples the program counter was always incremented to point to the address of the next instruction in program memory. This behavior only supports a linear execution of instructions. To provide the ability to specifically set the value of the program counter, instructions called *branches* are used. There are two types of branches: **unconditional** and **conditional**. In an unconditional branch, the program counter is always loaded with the value provided in the operand. As an example, let's look at an instruction to *branch always* to a specific address. This allows the program to perform loops. Let's say that the opcode of the instruction is x"20", has a mnemonic BRA, and is inserted into program memory starting at x"06". Example 11.6 shows the steps involved in executing the BRA instruction.

Example: Execution of an Instruction to "Branch Always"

A *branch always* instruction will set the program counter to the value provided by the operand. Let's create a program that will set the program counter to x"00". The program is as follows:

Using Mnemonics		Using Hex Values
BRA x"00"	or	x"20" x"00"

When the opcode and operand are put into program memory at x"06", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"06", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the PC holds x"07" and the IR holds x"20".

Step 2 – Decode the instruction

The CPU decodes x"20" and understands that it is a "branch always". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"07") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is the address to load into the PC. The operand is latched into the PC and the instruction is complete. After this instruction, the PC=x"00" and the program will begin executing instructions at that address.

Example 11.6
Execution of an instruction to "branch always"

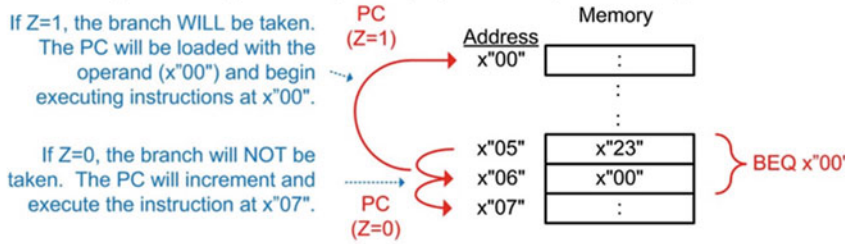
In a conditional branch, the program counter is only updated if a particular condition is true. The conditions come from the status flags in the condition code register (NZVC). This allows a program to selectively execute instructions based on the result of a prior operation. Let's look at an example instruction that will branch only if the *Z* flag is asserted. This instruction is called a *branch if equal to zero*. Let's say that the opcode of the instruction is x"23", has a mnemonic BEQ, and is inserted into program memory starting at x"05". Example 11.7 shows the steps involved in executing the BEQ instruction.

Example: Execution of an Instruction to "Branch if Equal to Zero"

This instruction will update the program counter with the address in the operand if the zero flag (Z) in the condition code register is asserted (Z=1). If Z=0, the program counter will simply increment to the next location in program memory. Let's look at how this program is executed. The instruction resides in program memory at addresses x"05" and x"06".

Using Mnemonics		Using Hex Values
BEQ x"00"	or	x"23" x"00"

When the opcode and operand are put into program memory at x"02", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"05", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the PC holds x"06" and the IR holds x"23".

Step 2 – Decode the instruction

The CPU decodes x"23" and understands that it is a "branch if equal to zero". It also knows from the opcode that the instruction has an operand that exists at the next address location. The FSM now looks at the Z flag and decides which path in the FSM to take in order to execute the instruction properly.

Step 3 – Execute the instruction

Z=1 – The branch will be taken by loading the PC with the operand. It places the PC address (x"06") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is then loaded into the PC. If this action is taken, the PC=x"00".

Z=0 – The branch will not be taken. Instead, the PC is simply incremented to point to the next location in memory, bypassing the operand. If this action is taken, the PC=x"07".

Example 11.7
Execution of an instruction to "branch if equal to zero"

Conditional branches allow computer programs to make *decisions* about which instructions to execute based on the results of previous instructions. This gives computers the ability to react to input signals or act based on the results of arithmetic or logic operations. Computer instruction sets typically contain conditional branches based on the NZVC flags in the condition code registers. The following instructions are a set of possible branches that could be created using the values of the NZVC flags.

- BMI—Branch if minus ($N = 1$)
- BPL—Branch if plus ($N = 0$)
- BEQ—Branch if equal to Zero ($Z = 1$)
- BNE—Branch if not equal to Zero ($Z = 0$)
- BVS—Branch if two's complement overflow occurred, or V is set ($V = 1$)

- BVC—Branch if two's complement overflow did not occur, or V is clear ($V = 0$)
- BCS—Branch if a carry occurred, or C is set ($C = 1$)
- BCC—Branch if a carry did not occur, or C is clear ($C = 0$)

Combinations of these flags can be used to create more conditional branches.

- BHI—Branch if higher ($C = 1$ and $Z = 0$)
- BLS—Branch if lower or the same ($C = 0$ and $Z = 1$)
- BGE—Branch if greater than or equal (($N = 0$ and $V = 0$) or ($N = 1$ and $V = 1$)), only valid for signed numbers
- BLT—Branch if less than (($N = 1$ and $V = 0$) or ($N = 0$ and $V = 1$)), only valid for signed numbers
- BGT—Branch if greater than (($N = 0$ and $V = 0$ and $Z = 0$) or ($N = 1$ and $V = 1$ and $Z = 0$)), only valid for signed numbers
- BLE—Branch if less than or equal (($N = 1$ and $V = 0$) or ($N = 0$ and $V = 1$) or ($Z = 1$)), only valid for signed numbers

Concept Check

CC11.2 Software development consists of choosing which instructions, and in what order, will be executed to accomplish a certain task. The group of instructions is called the *program* and is inserted into program memory. Which of the following might a software developer care about?

(A) Minimizing the number of instructions that need to be executed to accomplish the task in order to increase the computation rate.

(B) Minimizing the number of registers used in the CPU to save power.

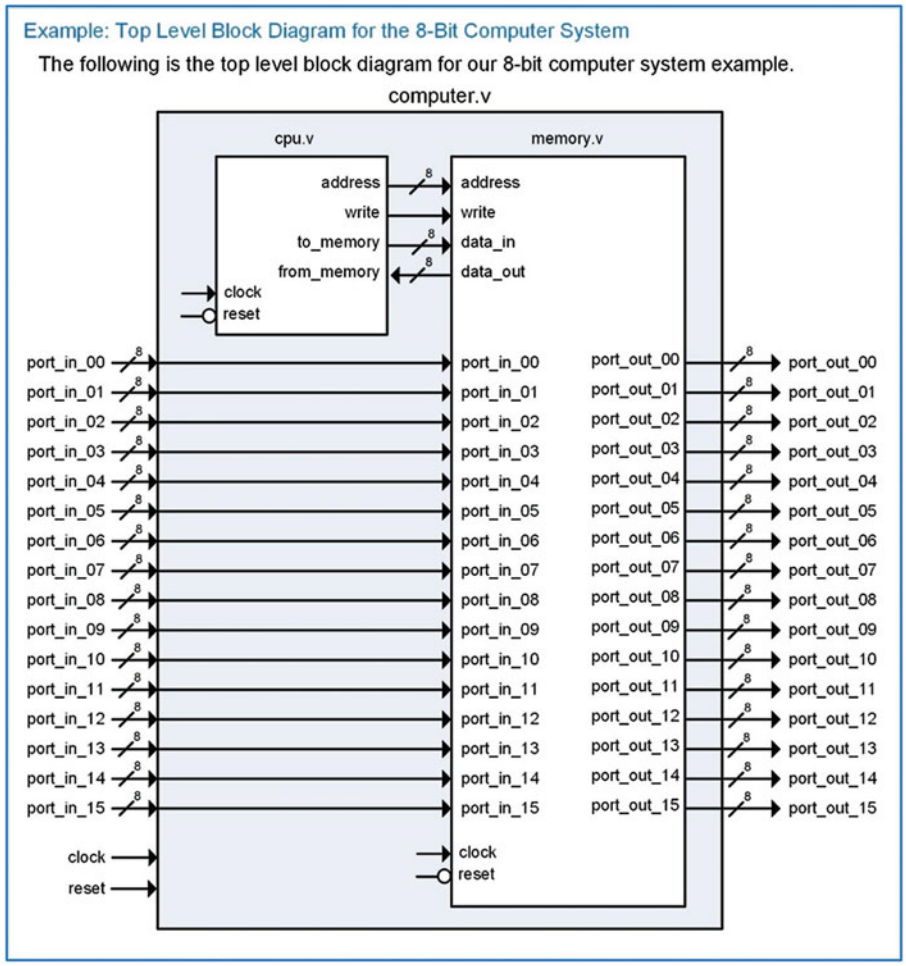
(C) Minimizing the overall size of the program to reduce the amount of program memory needed.

(D) Both A and C.

11.3 Computer Implementation: An 8-Bit Computer Example

11.3.1 Top-Level Block Diagram

Let's now look at the detailed implementation and instruction execution of a computer system. In order to illustrate the detailed operation, we will use a simple 8-bit computer system design. Example 11.8 shows the block diagram for the 8-bit computer system. This block diagram also contains the Verilog file and module names, which will be used when the behavioral model is implemented.



Example 11.8
Top-level block diagram for the 8-bit computer system

We will use the memory map shown in Example 11.1 for our example computer system. This mapping provides 128 bytes of program memory, 96 bytes of data memory, 16× output ports, and 16× input ports. To simplify the operation of this example computer, the address bus is limited to 8-bits. This only provides 256 locations of memory access but allows an entire address to be loaded into the CPU as a single operand of an instruction.

11.3.2 Instruction Set Design

Example 11.9 shows a basic instruction set for our example computer system. This set provides a variety of loads and stores, data manipulations, and branch instructions that will allow the computer to be programmed to perform more complex tasks through software development. These instructions are sufficient to provide a baseline of functionality in order to get the computer system operational. Additional instructions can be added as desired to increase the complexity of the system.

Example: Instruction Set for the 8-Bit Computer System

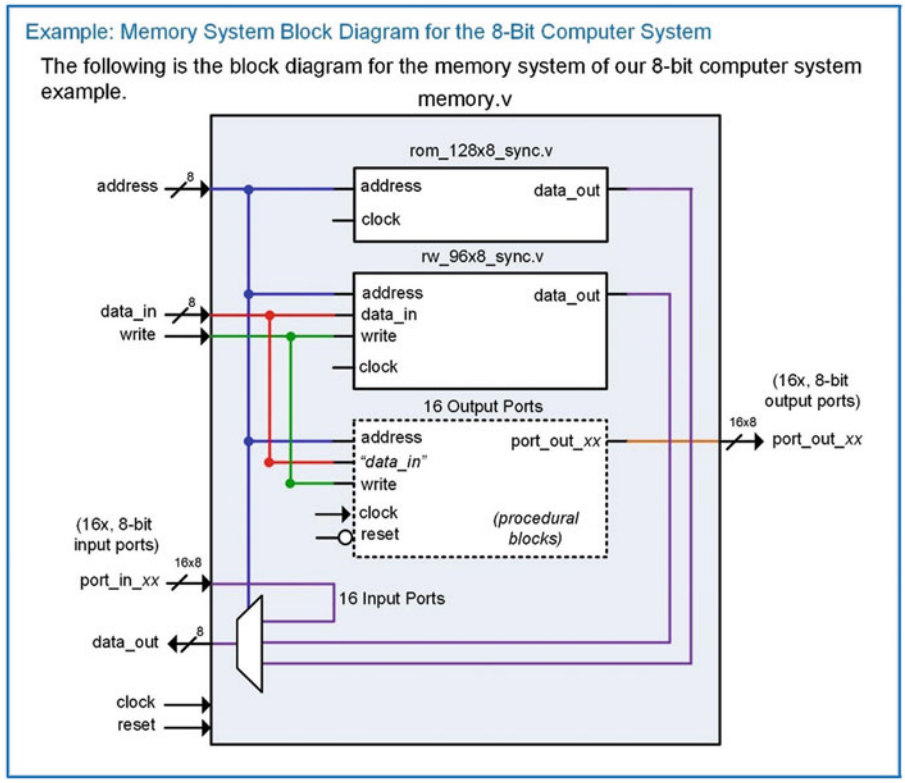
The following is a base set of instructions that the 8-bit computer system will be able to perform. Each instruction is given a descriptive mnemonic, which allows the system implementation and the programming to be more intuitive. Each instruction is also provided with a unique binary opcode. Some instructions have an operand, which provides additional information necessary for the instruction. If an instruction contains an operand, a description is provided as to how it is used (e.g., as data or as an address).

Mnemonic	Opcode, Operand	Description
"Loads and Stores"		
LDA_IMM	x"86", <*data*>	Load Register A using Immediate Addressing
LDA_DIR	x"87", <*addr*>	Load Register A using Direct Addressing
LDB_IMM	x"88", <*data*>	Load Register B with Immediate Addressing
LDB_DIR	x"89", <*addr*>	Load Register B with Direct Addressing
STA_DIR	x"96", <*addr*>	Store Register A to Memory using Direct Addressing
STB_DIR	x"97", <*addr*>	Store Register B to Memory using Direct Addressing
"Data Manipulations"		
ADD_AB	x"42"	A = A + B (plus)
SUB_AB	x"43"	A = A - B (minus)
AND_AB	x"44"	A = A · B (AND)
OR_AB	x"45"	A = A + B (OR)
INCA	x"46"	A = A + 1 (plus)
INCB	x"47"	B = B + 1 (plus)
DECA	x"48"	A = A - 1 (minus)
DECB	x"49"	B = B - 1 (minus)
"Branches"		
BRA	x"20", <*addr*>	Branch Always to Address Provided
BMI	x"21", <*addr*>	Branch to Address Provided if N=1
BPL	x"22", <*addr*>	Branch to Address Provided if N=0
BEQ	x"23", <*addr*>	Branch to Address Provided if Z=1
BNE	x"24", <*addr*>	Branch to Address Provided if Z=0
BVS	x"25", <*addr*>	Branch to Address Provided if V=1
BVC	x"26", <*addr*>	Branch to Address Provided if V=0
BCS	x"27", <*addr*>	Branch to Address Provided if C=1
BCC	x"28", <*addr*>	Branch to Address Provided if C=0

Example 11.9
Instruction set for the 8-bit computer system

11.3.3 Memory System Implementation

Let's now look at the memory system details. The memory system contains program memory, data memory, and input/output ports. Example 11.10 shows the block diagram of the memory system. The program and data memory will be implemented using lower-level components (rom_128x8_sync.v and rw_96x8_sync.v), while the input and output ports can be modeled using a combination of RTL blocks and combinational logic. The program and data memory subsystems contain dedicated circuitry to handle their addressing ranges. Each output port also contains dedicated circuitry to handle its unique address. A multiplexer is used to handle the signal routing back to the CPU based on the address provided.



Example 11.10
Memory system block diagram for the 8-bit computer system

11.3.3.1 Program Memory Implementation in Verilog

The program memory can be implemented in Verilog using the modeling techniques presented in Chap. 12. To make the Verilog more readable, the instruction mnemonics can be declared as parameters. This allows the mnemonic to be used when populating the program memory array. The following Verilog shows how the mnemonics for our basic instruction set can be defined as parameters.

```
parameter LDA_IMM = 8'h86;  //-- Load Register A with Immediate Addressing
parameter LDA_DIR = 8'h87;  //-- Load Register A with Direct Addressing
parameter LDB_IMM = 8'h88;  //-- Load Register B with Immediate Addressing
parameter LDB_DIR = 8'h89;  //-- Load Register B with Direct Addressing
parameter STA_DIR = 8'h96;  //-- Store Register A to memory (RAM or IO)
parameter STB_DIR = 8'h97;  //-- Store Register B to memory (RAM or IO)
parameter ADD_AB  = 8'h42;  //-- A <= A + B
parameter BRA     = 8'h20;  //-- Branch Always
parameter BEQ     = 8'h23;  //-- Branch if Z=1
```

Now the program memory can be declared as an array type with initial values to define the program. The following Verilog shows how to declare the program memory and an example program to perform a load, store, and a branch always. This program will continually write x“AA” to port_out_00.

```
reg[7:0] ROM[0:127];

initial
  begin
    ROM[0] = LDA_IMM;
    ROM[1] = 8'hAA;
    ROM[2] = STA_DIR;
    ROM[3] = 8'hE0;
    ROM[4] = BRA;
    ROM[5] = 8'h00;
  end
```

The address mapping for the program memory is handled in two ways. First, notice that the array type defined above uses indices from 0 to 127. This provides the appropriate addresses for each location in the memory. The second step is to create an internal enable line that will only allow assignments from ROM to data_out when a valid address is entered. Consider the following Verilog to create an internal enable (EN) that will only be asserted when the address falls within the valid program memory range of 0 to 127.

```
always @ (address)
  begin
    if ( (address >= 0) && (address <= 127) )
       EN = 1'b1;
    else
       EN = 1'b0;
  end
```

If this enable signal is not created, the simulation and synthesis will fail because data_out assignments will be attempted for addresses outside of the defined range of the ROM array. This enable line can now be used in the behavioral model for the ROM as follows:

```
always @ (posedge clock)
  begin
    if (EN)
      data_out = ROM[address];
  end
```

11.3.3.2 Data Memory Implementation in Verilog

The data memory is created using a similar strategy as the program memory. An array signal is declared with an address range corresponding to the memory map for the computer system (i.e., 128 to 223). An internal enable is again created that will prevent data_out assignments for addresses outside of this valid range. The following is the Verilog to declare the R/W memory array:

```
reg[7:0] RW[128:223];
```

The following is the Verilog to model the local enable and signal assignments for the R/W memory:

```
always @ (address)
  begin
    if ( (address >= 128) && (address <= 223) )
      EN = 1'b1;
    else
      EN = 1'b0;
  end
```

```
always @ (posedge clock)
  begin
    if (write && EN)
       RW[address] = data_in;
    else if (!write && EN)
       data_out = RW[address];
  end
```

11.3.3.3 Implementation of Output Ports in Verilog

Each output port in the computer system is assigned a unique address. Each output port also contains storage capability. This allows the CPU to update an output port by writing to its specific address. Once the CPU is done storing to the output port address and moves to the next instruction in the program, the output port holds its information until it is written to again. This behavior can be modeled using an RTL procedural block that uses the address bus and the write signal to create a synchronous enable condition. Each output port is modeled with its own block. The following Verilog shows how the output ports at x"E0" and x"E1" are modeled using address specific procedural blocks.

```
//-- port_out_00 (address E0)
   always @ (posedge clock or negedge reset)
     begin
       if (!reset)
          port_out_00 <= 8'h00;
       else
          if ((address == 8'hE0) && (write))
             port_out_00 <= data_in;
     end

 //-- port_out_01 (address E1)
   always @ (posedge clock or negedge reset)
     begin
       if (!reset)
          port_out_01 <= 8'h00;
       else
          if ((address == 8'hE1) && (write))
             port_out_01 <= data_in;
     end

                    :
    "the rest of the output port models go here..."
                    :
```

11.3.3.4 Implementation of Input Ports in Verilog

The input ports do not contain storage but do require a mechanism to selectively route their information to the data_out port of the memory system. This is accomplished using the multiplexer shown in Example 11.10. The only functionality that is required for the input ports is connecting their ports to the multiplexer.

11.3.3.5 Memory data_out Bus Implementation in Verilog

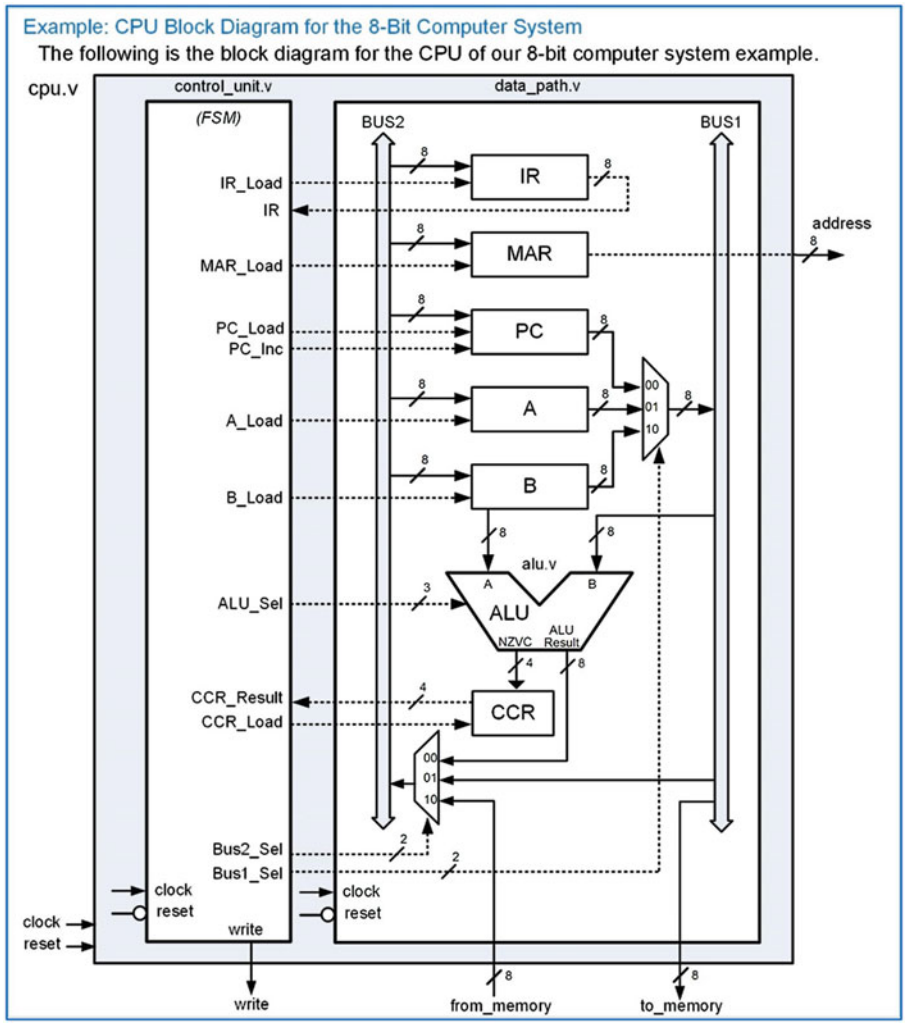
Now that all of the memory functionality has been designed, the final step is to implement the multiplexer that handles routing the appropriate information to the CPU on the data_out bus based on the incoming address. The following Verilog provides a model for this behavior. Recall that a multiplexer is combinational logic, so if the behavior is to be modeled using a procedural block, all inputs must be listed in the sensitivity list and blocking assignments are used. These inputs include the outputs from the program and data memory in addition to all of the input ports. The sensitivity list must also include the

address bus as it acts as the select input to the multiplexer. Within the block, an if-else statement is used to determine which subsystem drives data_out. Program memory will drive data_out when the incoming address is in the range of 0 to 127 (x“00” to x“7F”). Data memory will drive data_out when the address is in the range of 128 to 223 (x“80” to x“DF”). An input port will drive data_out when the address is in the range of 240 to 255 (x“F0” to x“FF”). Each input port has a unique address, so the specific addresses are listed as nested if-else clauses.

```
always @ (address, rom_data_out, rw_data_out,
          port_in_00, port_in_01, port_in_02, port_in_03,
          port_in_04, port_in_05, port_in_06, port_in_07,
          port_in_08, port_in_09, port_in_10, port_in_11,
          port_in_12, port_in_13, port_in_14, port_in_15)

   begin: MUX1

     if ( (address >= 0) && (address <= 127) )
        data_out = rom_data_out;
     else if ( (address >= 128) && (address <= 223) )
        data_out = rw_data_out;
     else if (address == 8'hF0) data_out = port_in_00;
     else if (address == 8'hF1) data_out = port_in_01;
     else if (address == 8'hF2) data_out = port_in_02;
     else if (address == 8'hF3) data_out = port_in_03;
     else if (address == 8'hF4) data_out = port_in_04;
     else if (address == 8'hF5) data_out = port_in_05;
     else if (address == 8'hF6) data_out = port_in_06;
     else if (address == 8'hF7) data_out = port_in_07;
     else if (address == 8'hF8) data_out = port_in_08;
     else if (address == 8'hF9) data_out = port_in_09;
     else if (address == 8'hFA) data_out = port_in_10;
     else if (address == 8'hFB) data_out = port_in_11;
     else if (address == 8'hFC) data_out = port_in_12;
     else if (address == 8'hFD) data_out = port_in_13;
     else if (address == 8'hFE) data_out = port_in_14;
     else if (address == 8'hFF) data_out = port_in_15;

   end
```

11.3.4 CPU Implementation

Let’s now look at the central processing unit details. The CPU contains two components, the control unit (control_unit.v) and the data path (data_path.v). The data path contains all of the registers and the ALU. The ALU is implemented as a subsystem within the data path (alu.v). The data path also contains a bus system in order to facilitate data movement between the registers and memory. The bus system is implemented with two multiplexers that are controlled by the control unit. The control unit contains the finite state machine that generates all control signals for the data path as it performs the fetch-decode-execute steps of each instruction. Example 11.11 shows the block diagram of the CPU in our 8-bit microcomputer example.



Example 11.11
CPU block diagram for the 8-bit computer system

11.3.4.1 Data Path Implementation in Verilog

Let's first look at the data path bus system that handles internal signal routing. The system consists of two 8-bit busses (Bus1 and Bus2) and two multiplexers. Bus1 is used as the destination of the PC, A, and B register outputs, while Bus2 is used as the input to the IR, MAR, PC, A, and B registers. Bus1 is connected directly to the *to_memory* port of the CPU to allow registers to write data to the memory system. Bus2 can be driven by the *from_memory* port of the CPU to allow the memory system to provide data for the CPU registers. The two multiplexers handle all signal routing and have their select lines (Bus1_Sel and Bus2_Sel) driven by the control unit. The following Verilog shows how the multiplexers are implemented. Again, a multiplexer is combinational logic, so all inputs must be listed in the sensitivity list of its procedural block and blocking assignments are used. Two additional signal assignments are also required to connect the MAR to the address port and to connect Bus1 to the to_memory port.

```
always @ (Bus1_Sel, PC, A, B)
  begin: MUX_BUS1
    case (Bus1_Sel)
      2'b00    : Bus1 = PC;
      2'b01    : Bus1 = A;
      2'b10    : Bus1 = B;
      default  : Bus1 = 8'hXX;
    endcase
  end

always @ (Bus2_Sel, ALU_Result, Bus1, from_memory)
  begin: MUX_BUS2
    case (Bus2_Sel)
      2'b00    : Bus2 = ALU_Result;
      2'b01    : Bus2 = Bus1;
      2'b10    : Bus2 = from_memory;
      default  : Bus1 = 8'hXX;
    endcase
  end

always @ (Bus1, MAR)
  begin
    to_memory = Bus1;
    address = MAR;
  end
```

Next, let's look at implementing the registers in the data path. Each register is implemented using a dedicated procedural block that is sensitive to clock and reset. This models the behavior of synchronous latches, or registers. Each register has a synchronous enable line that dictates when the register is updated. The register output is only updated when the enable line is asserted and a rising edge of the clock is detected. The following Verilog shows how to model the instruction register (IR). Notice that the signal IR is only updated if IR_Load is asserted and there is a rising edge of the clock. In this case, IR is loaded with the value that resides on Bus2.

```
always @ (posedge clock or negedge reset)
  begin: INSTRUCTION_REGISTER
    if (!reset)
      IR <= 8'h00;
    else
      if (IR_Load)
        IR <= Bus2;
  end
```

A nearly identical block is used to model the memory address register. A unique signal is declared called *MAR* in order to make the Verilog more readable. MAR is always assigned to address in this system.

```
always @ (posedge clock or negedge reset)
  begin: MEMORY_ADDRESS_REGISTER
    if (!reset)
      MAR <= 8'h00;
    else
     if (MAR_Load)
       MAR <= Bus2;
  end
```

Now let's look at the program counter block. This register contains additional functionality beyond simply latching in the value of Bus2. The program counter also has an increment feature that will take place synchronously when the signal PC_Inc coming from the control unit is asserted. This is handled using an additional nested if-else clause under the portion of the block handling the rising edge of clock condition.

```
always @ (posedge clock or negedge reset)
  begin: PROGRAM_COUNTER
    if (!reset)
      PC <= 8'h00;
    else
      if (PC_Load)
        PC <= Bus2;
      else if (PC_Inc)
        PC <= MAR + 1;
  end
```

The two general-purpose registers A and B are modeled using individual procedural blocks as follows:

```
always @ (posedge clock or negedge reset)
  begin: A_REGISTER
    if (!reset)
      A <= 8'h00;
    else
      if (A_Load)
        A <= Bus2;
  end

always @ (posedge clock or negedge reset)
  begin: B_REGISTER
    if (!reset)
      B <= 8'h00;
    else
      if (B_Load)
        B <= Bus2;
  end
```

The condition code register latches in the status flags from the ALU (NZVC) when the CCR_Load line is asserted. This behavior is modeled using a similar approach as follows:

```
always @ (posedge clock or negedge reset)
  begin: CONDITION_CODE_REGISTER
    if (!reset)
      CCR_Result <= 8'h00;
    else
      if (CCR_Load)
        CCR_Result <= NZVC;
  end
```

11.3.4.2 ALU Implementation in Verilog

The ALU is a set of combinational logic circuitry that performs arithmetic and logic operations. The output of the ALU operation is called *Result*. The ALU also outputs 4 status flags as a 4-bit bus called *NZVC*. The ALU behavior can be modeled using case and if-else statements that decide which operation to perform based on the input control signal *ALU_Sel*. The following Verilog shows an example of how to implement the ALU addition functionality. A case statement is used to decide which operation is being performed based on the ALU_Sel input. Under each operation clause, a series of procedural statements are used to compute the result and update the NZVC flags. Each of these flags is updated individually. The *N* flag can be simply driven with position 7 of the ALU result since this bit is the sign bit for signed numbers. The *Z* flag can be driven using an if-else condition that checks whether the result was x"00". The V flag is updated based on the type of the operation. For the addition operation, the V flag will be asserted if a POS + POS = NEG or a NEG + NEG = POS. These conditions can be checked by looking at the sign bits of the inputs and the sign bit of the result. Finally, the *C* flag can be computed as the 8th bit in the addition of *A* + *B*.

```
always @ (A, B, ALU_Sel)
  begin
    case (ALU_Sel)
      3'b000  : begin //-- Addition

                //-- Sum and Carry Flag
                  {NZVC[0], Result} = A + B;

                //-- Negative Flag
                   NZVC[3] = Result[7];

                //-- Zero Flag
                  if (Result == 0)
                     NZVC[2] = 1;
                   else
                     NZVC[2] = 0;

                //-- Two's Comp Overflow Flag
                  if ( ((A[7]==0) && (B[7]==0) && (Result[7] == 1)) ||
                       ((A[7]==1) && (B[7]==1) && (Result[7] == 0)) )
                     NZVC[1] = 1;
                   else
                     NZVC[1] = 0;

                end

                      :
      //-- other ALU operations go here...
                      :

      default  : begin
                    Result = 8'hXX;
                    NZVC   = 4'hX;
                 end
    endcase

  end
```

11.3.4.3 Control Unit Implementation in Verilog

Let's now look at how to implement the control unit state machine. We'll first look at the formation of the Verilog to model the FSM and then turn to the detailed state transitions in order to accomplish a variety of the most common instructions. The control unit sends signals to the data path in order to move data in and out of registers and into the ALU to perform data manipulations. The finite state machine is implemented with the behavioral modeling techniques presented in Chap. 9. The model contains three processes in order to implement the state memory, next state logic, and output logic of the FSM. Parameters are created for each of the states defined in the state diagram of the FSM. The states associated with fetching (S_FETCH_0, S_FETCH_1, S_FETCH_2) and decoding the opcode (S_DECODE_3) are performed each time an instruction is executed. A unique path is then added after the decode state to perform the steps associated with executing each individual instruction. The FSM can be created one instruction at a time by adding additional state paths after the decode state. The following Verilog code shows how the user-defined state names are created for nine basic instructions (LDA_IMM, LDA_DIR, STA_DIR, LDB_IMM, LDB_DIR, STB_DIR, ADD_AB, BRA, and BEQ). Eight-bit state variables are created for current_state and next_state to accommodate future state codes. The state codes are assigned in binary using integer format to allow additional states to be easily added.

```
reg    [7:0] current_state, next_state;
parameter S_FETCH_0  = 0,    //-- Opcode fetch states
          S_FETCH_1  = 1,
          S_FETCH_2  = 2,

          S_DECODE_3 = 3,    //-- Opcode decode state

          S_LDA_IMM_4 = 4,   //-- Load A (Immediate) states
          S_LDA_IMM_5 = 5,
          S_LDA_IMM_6 = 6,

          S_LDA_DIR_4 = 7,   //-- Load A (Direct) states
          S_LDA_DIR_5 = 8,
          S_LDA_DIR_6 = 9,
          S_LDA_DIR_7 = 10,
          S_LDA_DIR_8 = 11,

          S_STA_DIR_4 = 12,  //-- Store A (Direct) States
          S_STA_DIR_5 = 13,
          S_STA_DIR_6 = 14,
          S_STA_DIR_7 = 15,

          S_LDB_IMM_4 = 16,  //-- Load B (Immediate) states
          S_LDB_IMM_5 = 17,
          S_LDB_IMM_6 = 18,

          S_LDB_DIR_4 = 19,  //-- Load B (Direct) states
          S_LDB_DIR_5 = 20,
          S_LDB_DIR_6 = 21,
          S_LDB_DIR_7 = 22,
          S_LDB_DIR_8 = 23,

          S_STB_DIR_4 = 24,  //-- Store B (Direct) States
          S_STB_DIR_5 = 25,
          S_STB_DIR_6 = 26,
          S_STB_DIR_7 = 27,

          S_BRA_4   = 28,    //-- Branch Always States
          S_BRA_5   = 29,
          S_BRA_6   = 30,

          S_BEQ_4   = 31,    //-- Branch if Equal States
          S_BEQ_5   = 32,
          S_BEQ_6   = 33,
          S_BEQ_7   = 34,

          S_ADD_AB_4 = 35;   //-- Addition States
```

Within the control unit module, the state memory is implemented as a separate procedural block that will update the current state with the next state on each rising edge of the clock. The reset state will be the first fetch state in the FSM (i.e., S_FETCH_0). The following Verilog shows how the state memory in the control unit can be modeled. Note that this block models sequential logic, so non-blocking assignments are used.

```
always @ (posedge clock or negedge reset)
  begin: STATE_MEMORY
    if (!reset)
       current_state <= S_FETCH_0;
    else
       current_state <= next_state;
  end
```

The next state logic is also implemented as a separate procedural block. The next state logic depends on the current state, instruction register (IR), and the condition code register (CCR_Result). The following Verilog gives a portion of the next state logic process showing how the state transitions can be modeled.

```
always @ (current_state, IR, CCR_Result)
  begin: NEXT_STATE_LOGIC
    case (current_state)
      S_FETCH_0   : next_state = S_FETCH_1;   //-- Path for FETCH instruction
      S_FETCH_1   : next_state = S_FETCH_2;
      S_FETCH_2   : next_state = S_DECODE_3;

      S_DECODE_3 : if    (IR == LDA_IMM) next_state = S_LDA_IMM_4;   //-- Register
                                                                        A
                   else if (IR == LDA_DIR) next_state = S_LDA_DIR_4;
                   else if (IR == STA_DIR) next_state = S_STA_DIR_4;
                   else if (IR == LDB_IMM) next_state = S_LDB_IMM_4; //-- Register
                                                                        B
                   else if (IR == LDB_DIR) next_state = S_LDB_DIR_4;
                   else if (IR == STB_DIR) next_state = S_STB_DIR_4;
                   else if (IR == BRA)    next_state = S_BRA_4;       //-- Branch
                                                                        Always
                   else if (IR == ADD_AB) next_state = S_ADD_AB_4;  //-- ADD
                   else                   next_state = S_FETCH_0;           //-- others
                                                                        go here

        S_LDA_IMM_4 : next_state = S_LDA_IMM_5;  //-- Path for LDA_IMM instruction
        S_LDA_IMM_5 : next_state = S_LDA_IMM_6;
        S_LDA_IMM_6 : next_state = S_FETCH_0;

                                   :
          Next state logic for other states goes here...
                                   :
    endcase
   end
```

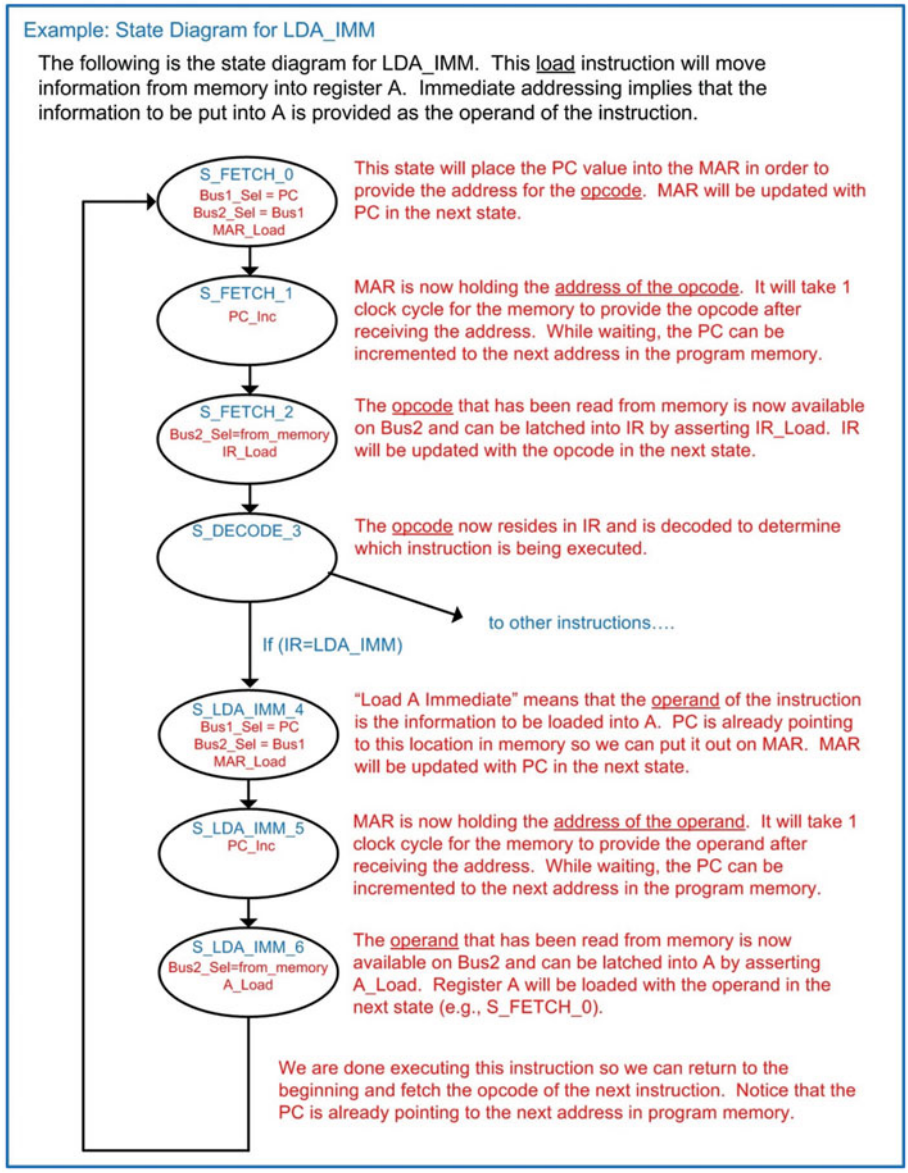
Finally, the output logic is modeled as a third, separate procedural block. It is useful to explicitly state the outputs of the control unit for each state in the machine to allow easy debugging and avoid synthesizing latches. Our example computer system has Moore type outputs, so the process only depends on the current state. The following Verilog shows a portion of the output logic process.

```
always @ (current_state)
  begin: OUTPUT_LOGIC
    case (current_state)

      S_FETCH_0  : begin          //-- Put PC onto MAR to provide address of Opcode
                     IR_Load  = 0;
                     MAR_Load = 1;
                     PC_Load  = 0;
                     PC_Inc   = 0;
                     A_Load   = 0;
                     B_Load   = 0;
                     ALU_Sel  = 3'b000;
                     CCR_Load = 0;
                     Bus1_Sel = 2'b00; //-- "00"=PC, "01"=A,  "10"=B
                     Bus2_Sel = 2'b01; //-- "00"=ALU, "01"=Bus1, "10"=from_memory
                     write    = 0;
                   end
```

```
        S_FETCH_1  : begin     //-- Increment PC, Opcode will be available next state
                       IR_Load  = 0;
                       MAR_Load = 0;
                       PC_Load  = 0;
                       PC_Inc   = 1;
                       A_Load   = 0;
                       B_Load   = 0;
                       ALU_Sel  = 3'b000;
                       CCR_Load = 0;
                       Bus1_Sel = 2'b00; //--"00"=PC, "01"=A,    "10"=B
                       Bus2_Sel = 2'b00; //--"00"=ALU,"01"=Bus1,"10"=from_memory
                       write    = 0;
                     end;

                                 :
              Output logic for other states goes here...
                                 :
      endcase
    end
```

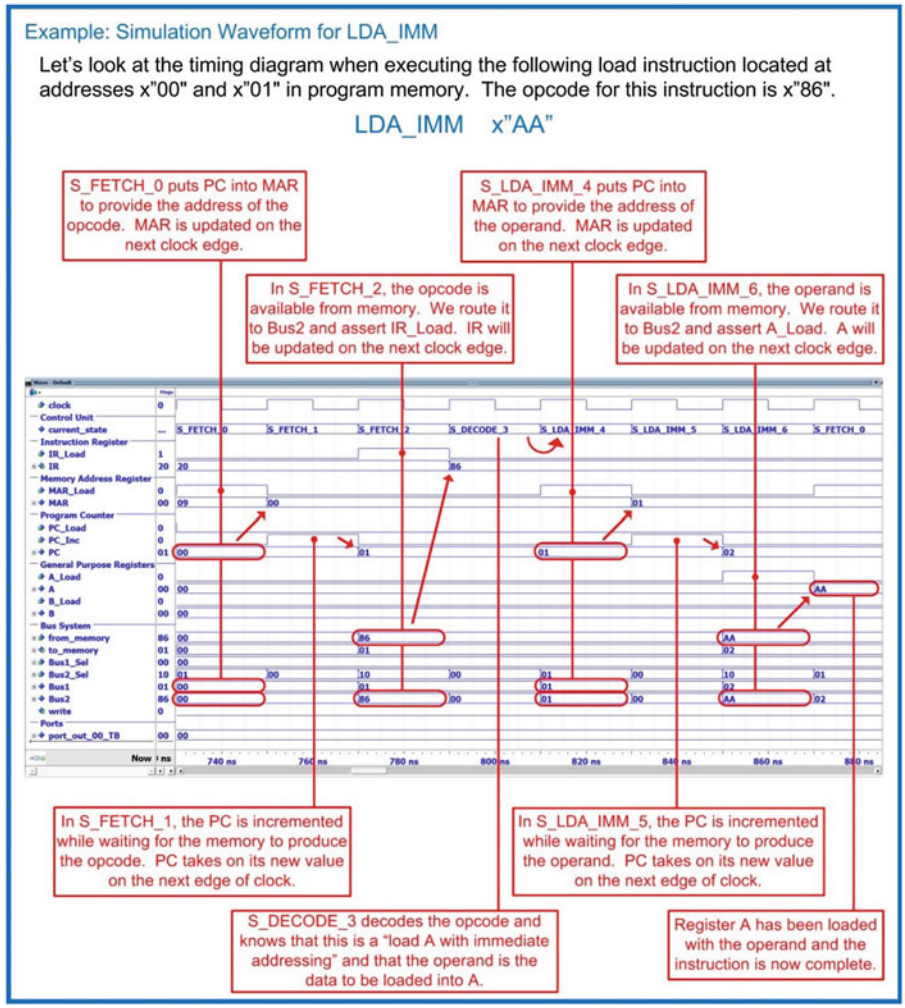
11.3.4.3.1 Detailed Execution of LDA_IMM

Now let's look at the details of the state transitions and output signals in the control unit FSM when executing a few of the most common instructions. Let's begin with the instruction to load register A using immediate addressing (LDA_IMM). Example 11.12 shows the state diagram for this instruction. The first three states (S_FETCH_0, S_FETCH_1, S_FETCH_2) handle fetching the opcode. The purpose of these states is to read the opcode from the address being held by the program counter and put it into the instruction register. Multiple states are needed to handle putting PC into MAR to provide the address of the opcode, waiting for the memory system to provide the opcode, latching the opcode into IR, and incrementing PC to the next location in program memory. Another state is used to decode the opcode (S_DECODE_3) in order to decide which path to take in the state diagram based on the instruction being executed. After the decode state, a series of three more states are needed (S_LDA_IMM_4, S_LDA_IMM_5, S_LDA_IMM_6) to execute the instruction. The purpose of these states is to read the operand from the address being held by the program counter and put it into A. Multiple states are needed to handle putting PC into MAR to provide the address of the operand, waiting for the memory system to provide the operand, latching the operand into A, and incrementing PC to the next location in program memory. When the instruction completes, the value of the operand resides in A and PC is pointing to the next location in program memory, which is the opcode of the next instruction to be executed.



Example 11.12
State diagram for LDA_IMM

Example 11.13 shows the simulation waveform for executing LDA_IMM. In this example, register A is loaded with the operand of the instruction, which holds the value x"AA".



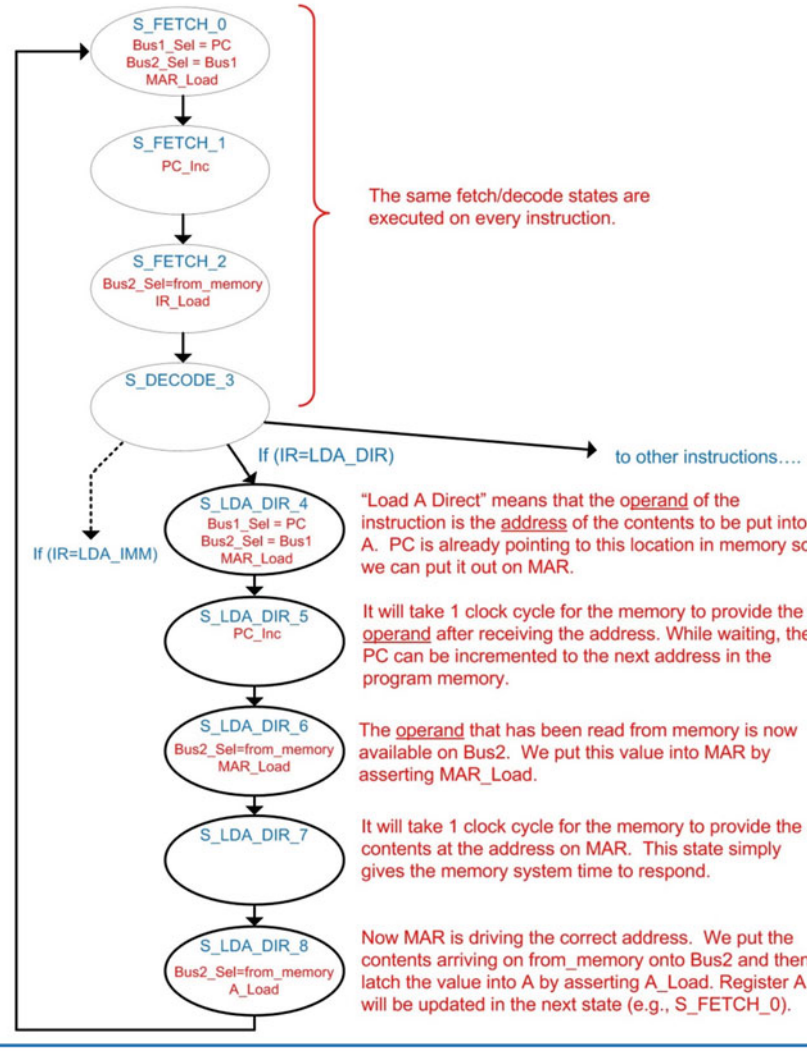
Example 11.13
Simulation waveform for LDA_IMM

11.3.4.3.2 Detailed Execution of LDA_DIR

Now let's look at the details of the instruction to load register A using direct addressing (LDA_DIR). Example 11.14 shows the state diagram for this instruction. The first four states to fetch and decode the opcode are the same states as in the previous instruction and are performed each time a new instruction is executed. Once the opcode is decoded, the state machine traverses five new states to execute the instruction (S_LDA_DIR_4, S_LDA_DIR_5, S_LDA_DIR_6, S_LDA_DIR_7, S_LDA_DIR_8). The purpose of these states is to read the operand and then use it as the address of where to read the contents to put into A.

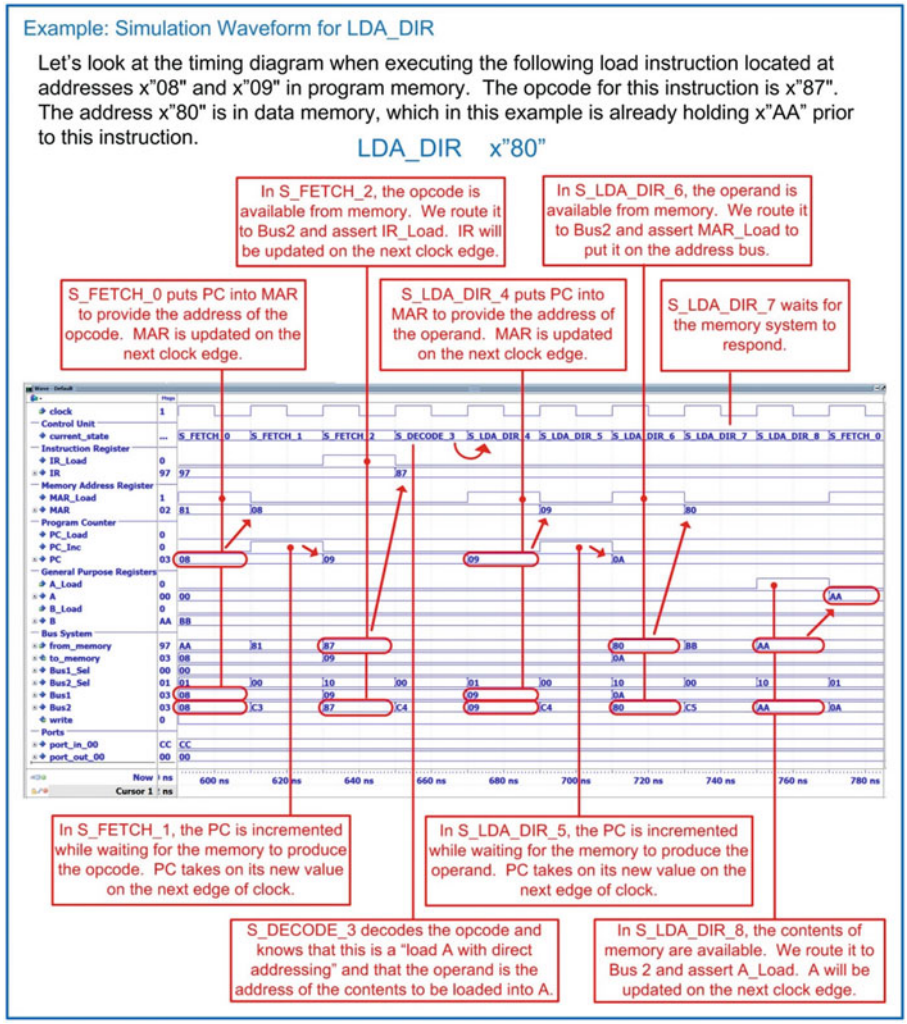
Example: State Diagram LDA_DIR

The following is the state diagram for LDA_DIR. This load instruction will move information from memory into register A. Direct addressing implies that the information to be put into A is located at the address provided as the operand of the instruction.



Example 11.14
State diagram for LDA_DIR

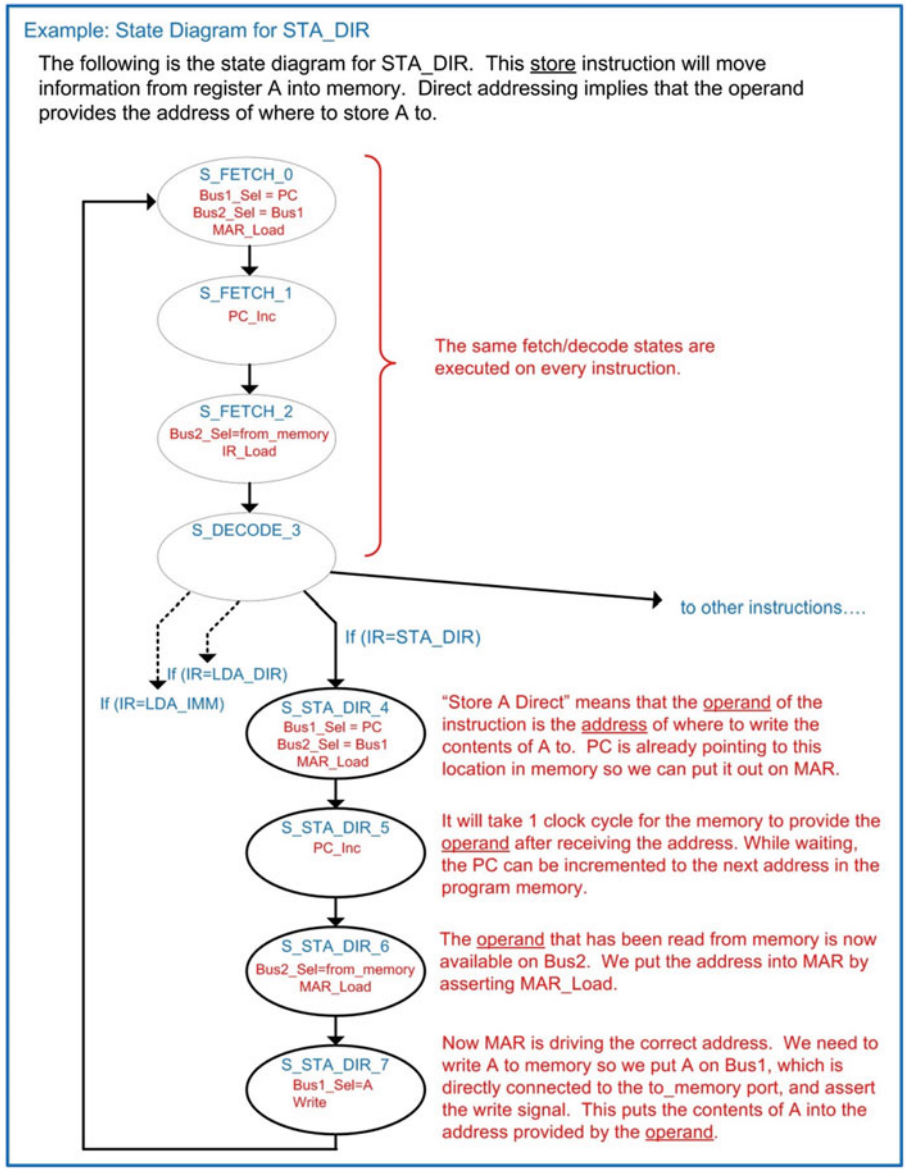
Example 11.15 shows the simulation waveform for executing LDA_DIR. In this example, register A is loaded with the contents located at address x"80", which has already been initialized to x"AA".



Example 11.15
Simulation waveform for LDA_DIR

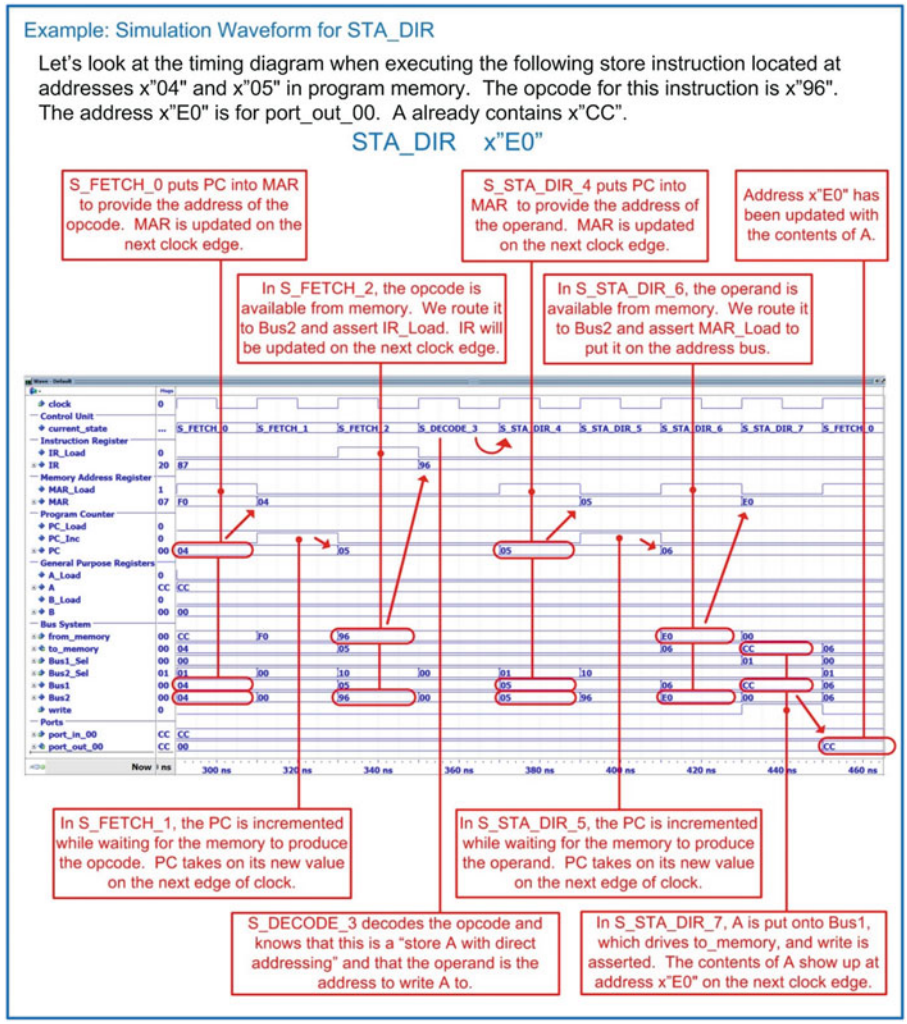
11.3.4.3.3 Detailed Execution of STA_DIR

Now let's look at the details of the instruction to store register A to memory using direct addressing (STA_DIR). Example 11.16 shows the state diagram for this instruction. The first four states are again the same as prior instructions in order to fetch and decode the opcode. Once the opcode is decoded, the state machine traverses four new states to execute the instruction (S_STA_DIR_4, S_STA_DIR_5, S_STA_DIR_6, S_STA_DIR_7). The purpose of these states is to read the operand and then use it as the address of where to write the contents of A to.



Example 11.16
State diagram for STA_DIR

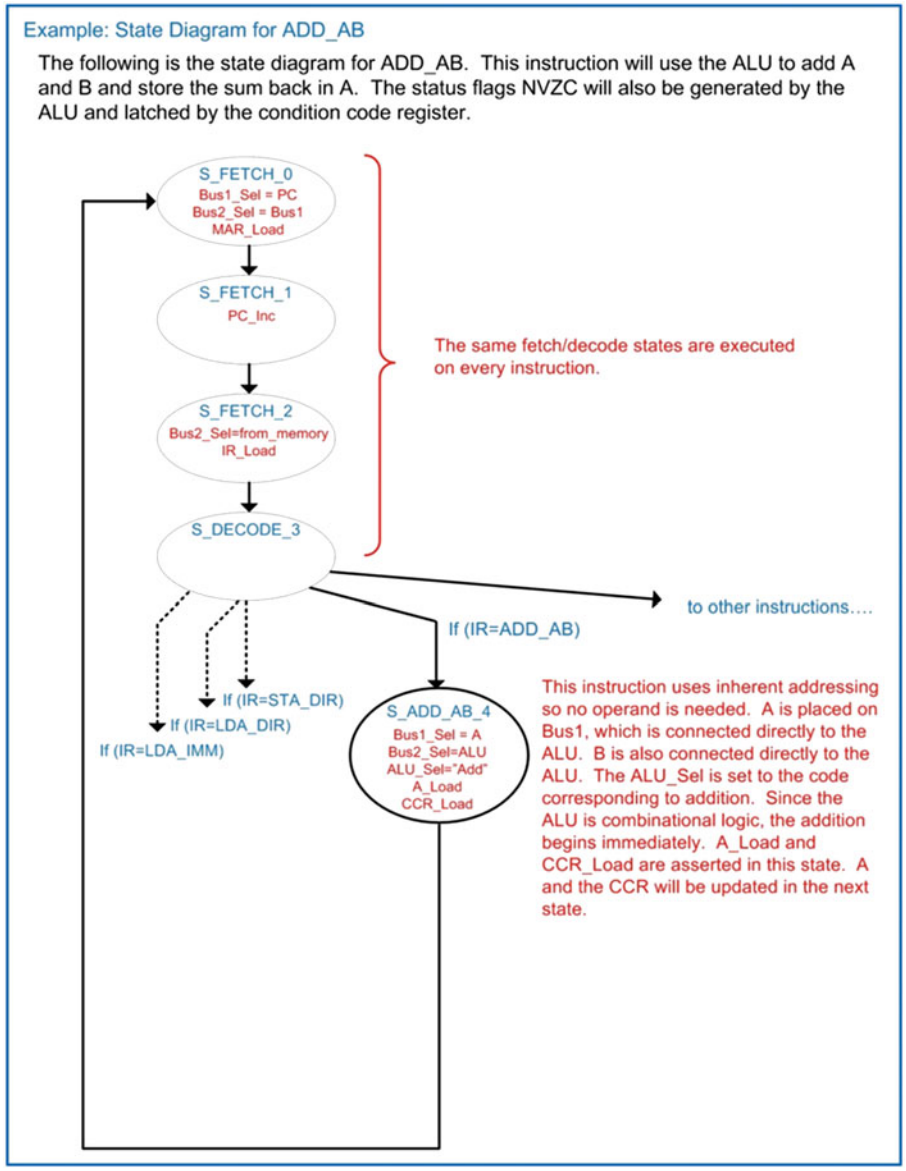
Example 11.17 shows the simulation waveform for executing STA_DIR. In this example, register A already contains the value x“CC” and will be stored to address x“E0”. The address x“E0” is an output port (port_out_00) in our example computer system.



Example 11.17
Simulation waveform for STA_DIR

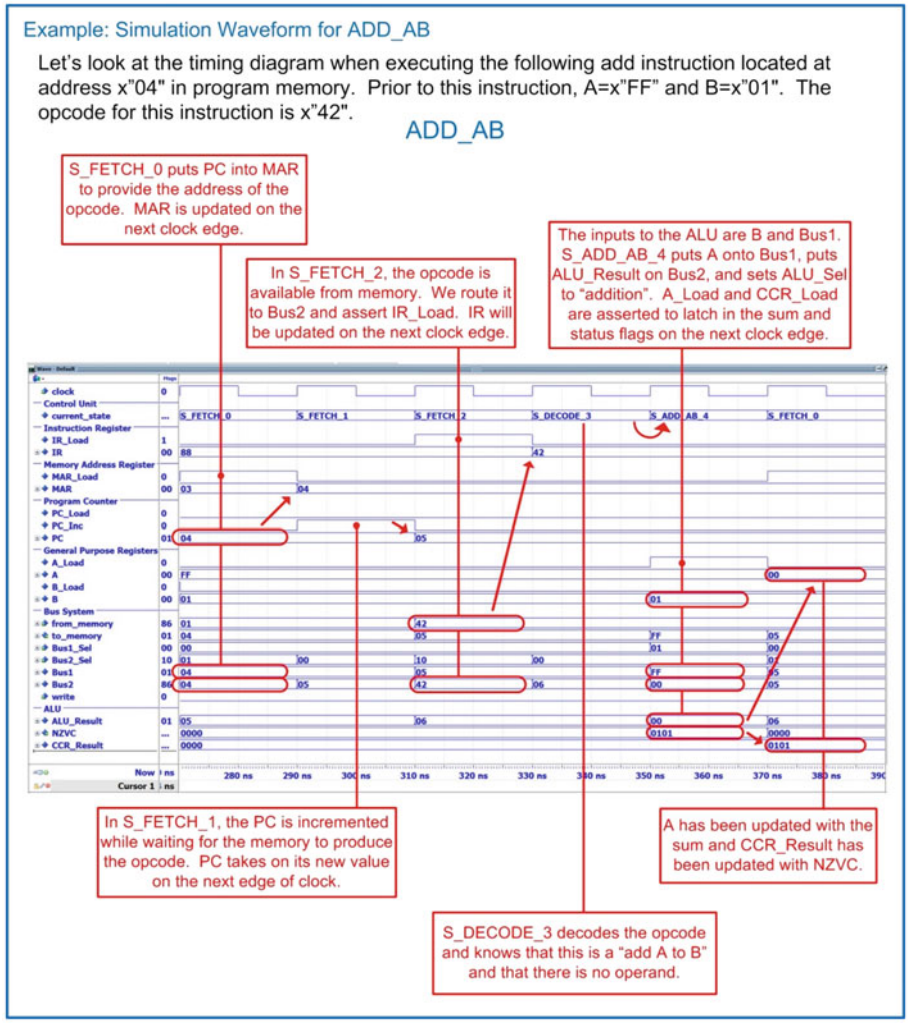
11.3.4.3.4 Detailed Execution of ADD_AB

Now let's look at the details of the instruction to add A to B and store the sum back in A (ADD_AB). Example 11.18 shows the state diagram for this instruction. The first four states are again the same as prior instructions in order to fetch and decode the opcode. Once the opcode is decoded, the state machine only requires one more state to complete the operation (S_ADD_AB_4). The ALU is combinational logic, so it will begin to compute the sum immediately as soon as the inputs are updated. The inputs to the ALU are Bus1 and register B. Since B is directly connected to the ALU, all that is required to start the addition is to put A onto Bus1. The output of the ALU is put on Bus2 so that it can be latched into A on the next clock edge. The ALU also outputs the status flags NZVC, which are directly connected to the condition code register. A_Load and CCR_Load are asserted in this state. A and CCR_Result will be updated in the next state (i.e., S_FETCH_0).



Example 11.18
State diagram for ADD_AB

Example 11.19 shows the simulation waveform for executing ADD_AB. In this example, two load immediate instructions were used to initialize the general-purpose registers to A = x“FF” and B = x“01” prior to the addition. The addition of these values will result in a sum of x“00” and assert the carry (C) and zero (Z) flags in the condition code register.



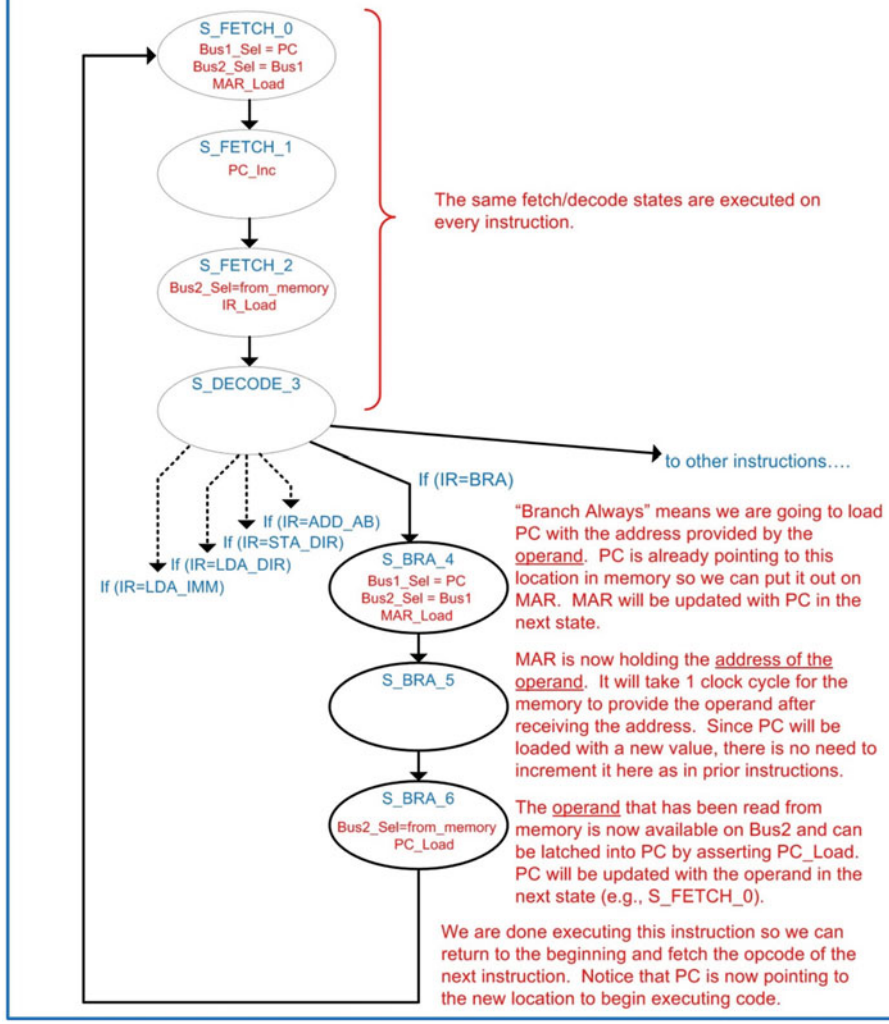
Example 11.19
Simulation waveform for ADD_AB

11.3.4.3.5 Detailed Execution of BRA

Now let's look at the details of the instruction to branch always (BRA). Example 11.20 shows the state diagram for this instruction. The first four states are again the same as prior instructions in order to fetch and decode the opcode. Once the opcode is decoded, the state machine traverses four new states to execute the instruction (S_BRA_4, S_BRA_5, S_BRA_6). The purpose of these states is to read the operand and put its value into PC to set the new location in program memory to execute instructions.

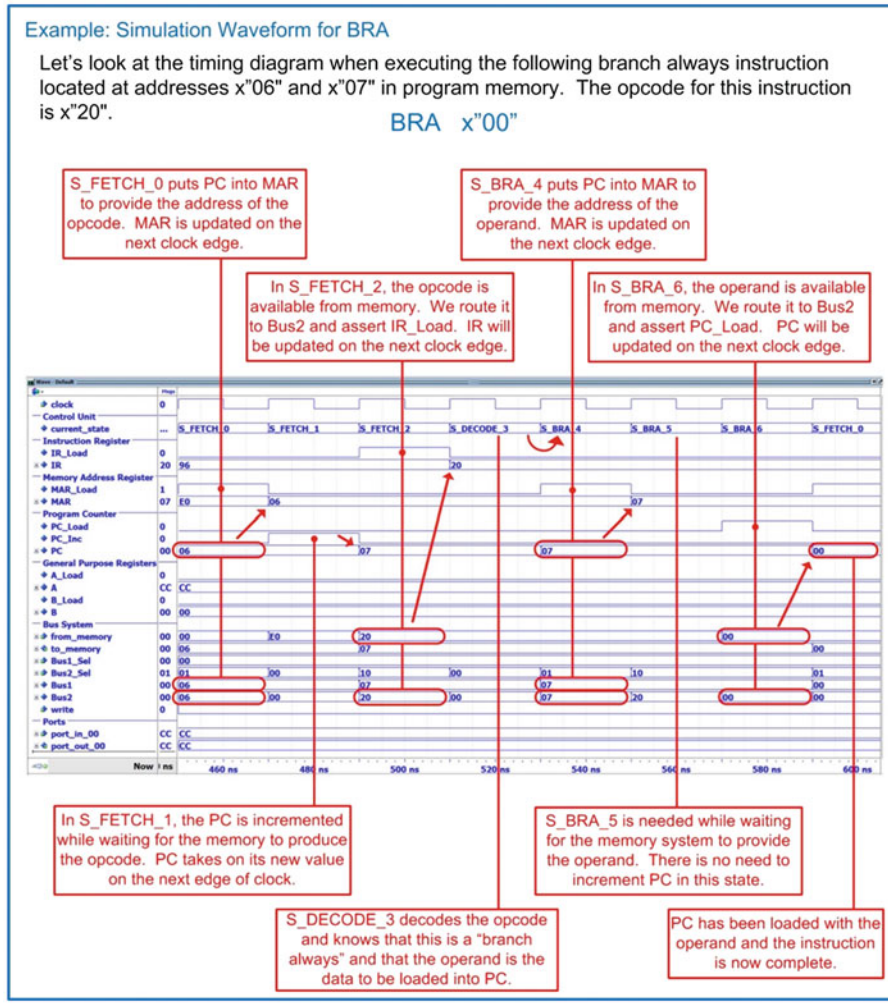
Example: State Diagram for BRA

The following is the state diagram for BRA. This instruction will load the program counter with the address supplied by the operand of the instruction. This has the effect of setting the address of the next instruction to be executed to a new location in program memory.



Example 11.20
State diagram for BRA

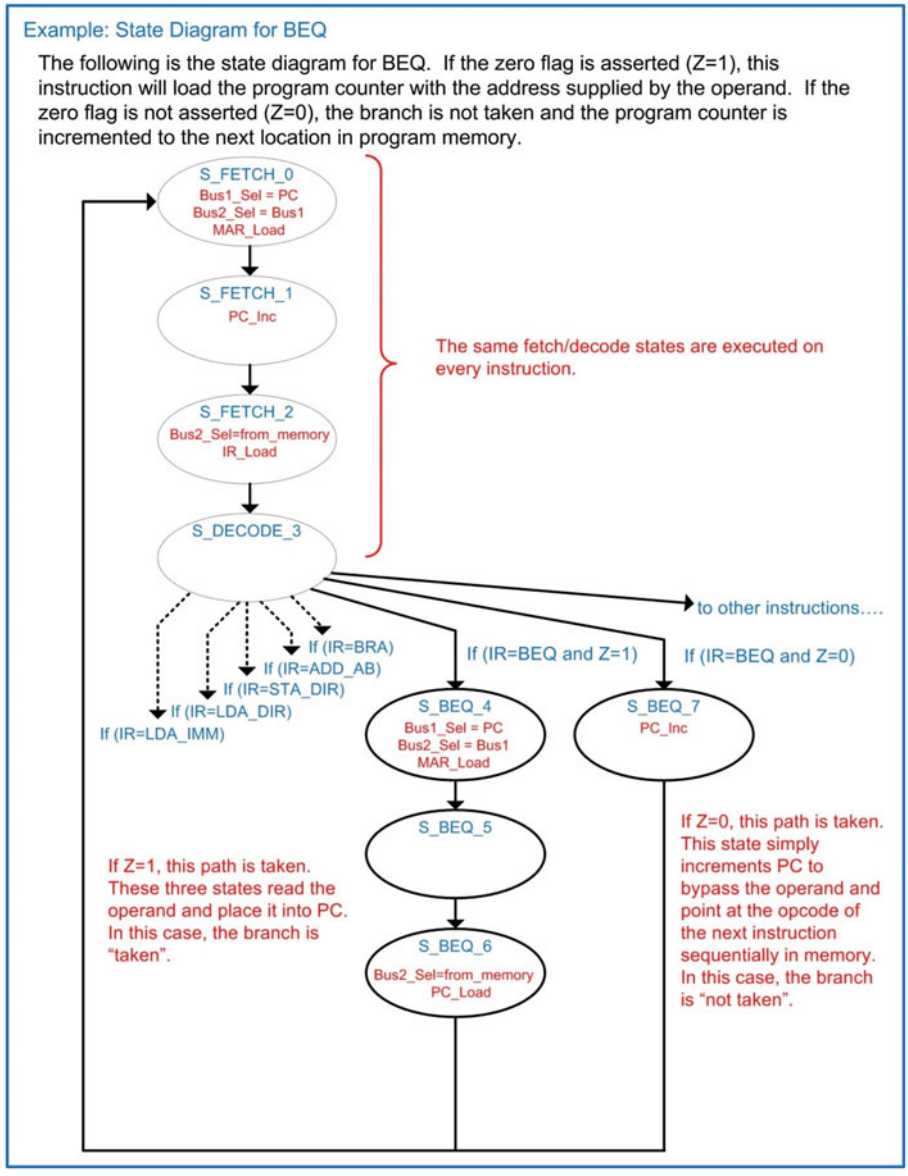
Example 11.21 shows the simulation waveform for executing BRA. In this example, PC is set back to address x“00”.



Example 11.21
Simulation waveform for BRA

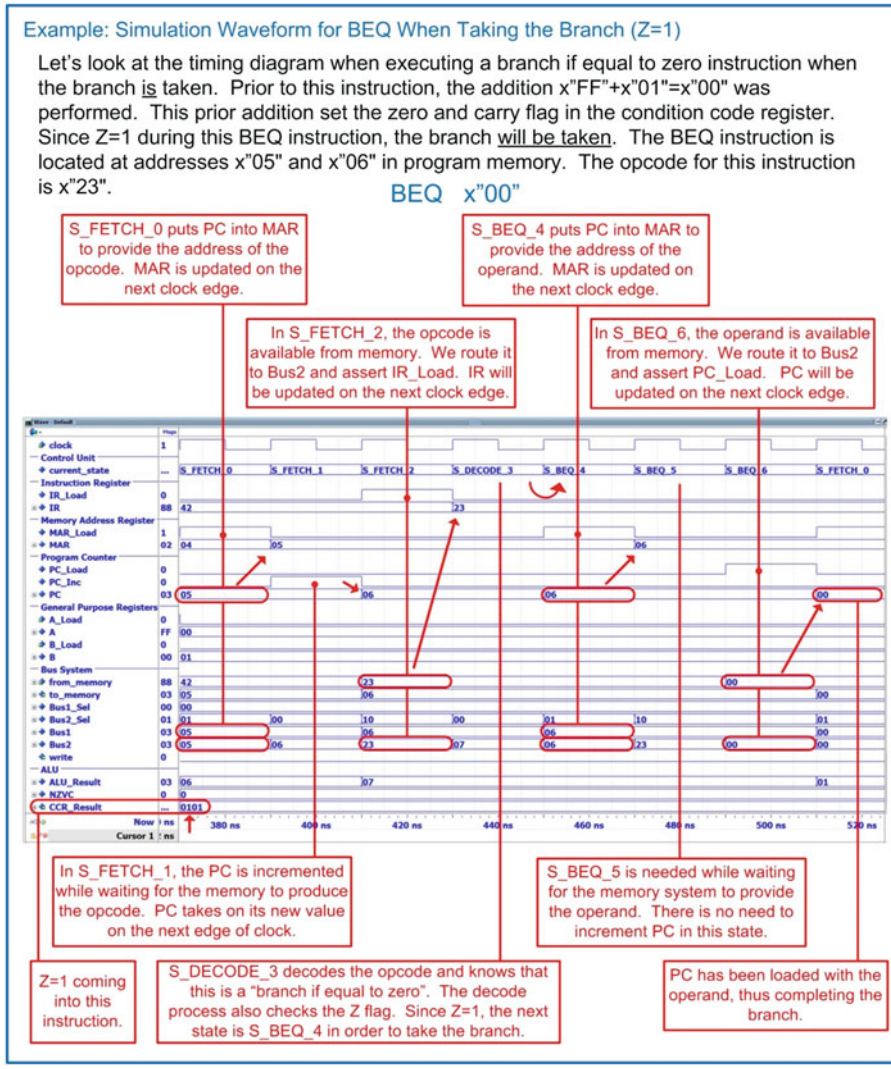
11.3.4.3.6 Detailed Execution of BEQ

Now let's look at the branch if equal to zero (BEQ) instruction. Example 11.22 shows the state diagram for this instruction. Notice that in this conditional branch, the path that is taken through the FSM depends on both IR and CCR. In the case that $Z = 1$, the branch is taken, meaning that the operand is loaded into PC. In the case that $Z = 0$, the branch is not taken, meaning that PC is simply incremented to bypass the operand and point to the beginning of the next instruction in program memory.



Example 11.22
State diagram for BEQ

Example 11.23 shows the simulation waveform for executing BEQ when the branch *is taken*. Prior to this instruction, an addition was performed on x"FF" and x"01". This resulted in a sum of x"00", which asserted the *Z* and *C* flags in the condition code register. Since $Z = 1$ when BEQ is executed, the branch is taken.

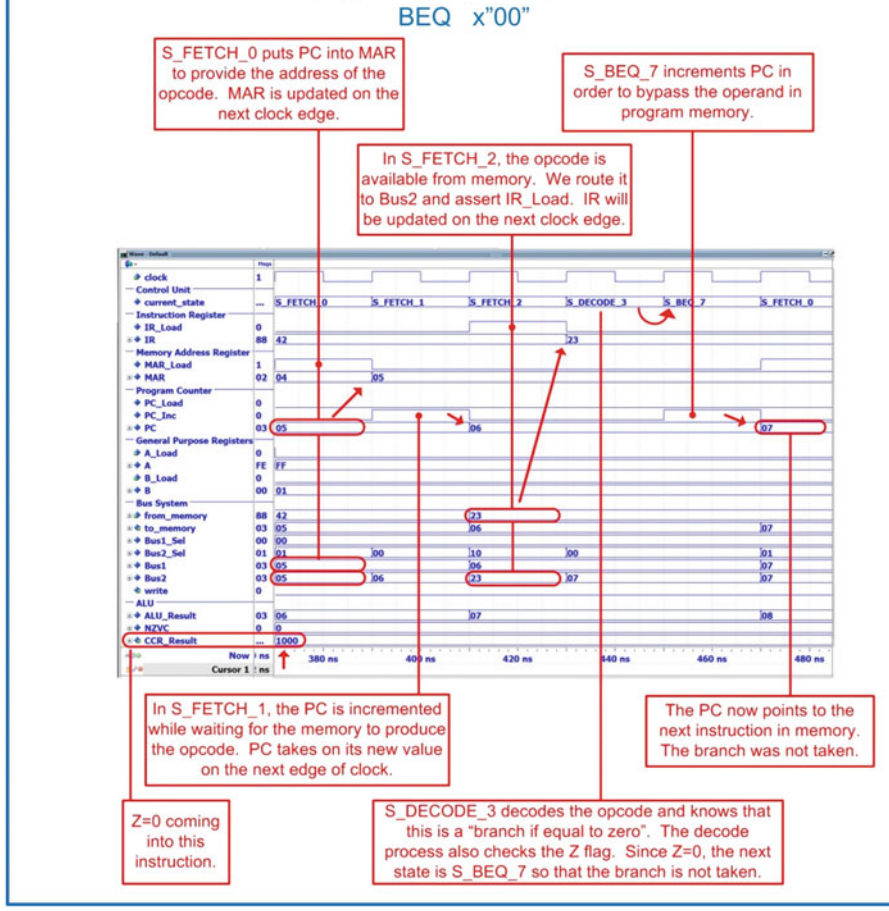


Example 11.23
Simulation waveform for BEQ when taking the branch ($Z = 1$)

Example 11.24 shows the simulation waveform for executing BEQ when the branch *is not taken*. Prior to this instruction, an addition was performed on x"FE" and x"01". This resulted in a sum of x"FF", which did not assert the Z flag. Since $Z = 0$ when BEQ is executed, the branch is not taken. When not taking the branch, PC must be incremented again in order to bypass the operand and point to the next location in program memory.

Example: Simulation Waveform for BEQ When the Branch is Not Taken (Z=0)

Let's look at the timing diagram when executing a branch if equal to zero instruction when the branch is not taken. Prior to this instruction, the addition x"FE"+x"01"=x"FF" was performed. This addition did not set the zero in the condition code register. Since this operation resulted in Z=0, the branch will not be taken. The BEQ instruction is located at addresses x"05" and x"06" in program memory. The opcode for this instruction is x"23".



Example 11.24
Simulation waveform for BEQ when the branch is not taken ($Z = 0$)

Concept Check

CC11.3 The 8-bit microcomputer example presented in this section is a very simple architecture used to illustrate the basic concepts of a computer. If we wanted to keep this computer an 8-bit system but increase the depth of the memory, it would require adding more address lines to the address bus. What changes to the computer system would need to be made to accommodate the wider address bus?

(A) The width of the program counter would need to be increased to support the wider address bus.

(B) The size of the memory address register would need to be increased to support the wider address bus.

(C) Instructions that use direct addressing would need additional bytes of operand to pass the wider address into the CPU 8-bits at a time.

(D) All of the above

Summary

- A computer is a collection of hardware components that are constructed to perform a specific set of instructions to process and store data. The main hardware components of a computer are the central processing unit (CPU), program memory, data memory, and input/output ports.
- The CPU consists of registers for fast storage, an arithmetic logic unit (ALU) for data manipulation, and a control state machine that directs all activity to execute an instruction.
- A CPU is typically organized into a *data path* and a *control unit*. The data path contains circuitry used to store and process information. The data path includes registers and the ALU. The control unit is a large state machine that sends control signals to the data path in order to facilitate instruction execution.
- The control unit performs a *fetch-decode-execute* cycle in order to complete instructions.
- The instructions that a computer is designed to execute is called its *instruction set*.
- Instructions are inserted into *program memory* in a sequence that when executed will accomplish a particular task. This sequence of instructions is called a computer *program*.
- An instruction consists of an *opcode* and a potential *operand*. The opcode is the unique binary code that tells the control state machine which instruction is being executed. An operand is additional information that may be needed for the instruction.
- An *addressing mode* refers to the way that the operand is treated. In *immediate* addressing the operand is the actual data to be used. In *direct* addressing the operand is the address of where the data is to be retrieved or stored. In *inherent* addressing all of the information needed to complete the instruction is contained within the opcode, so no operand is needed.
- A computer also contains *data memory* to hold temporary variables during run time.
- A computer also contains input and output ports to interface with the outside world.
- A *memory mapped* system is one in which the program memory, data memory, and I/O ports are all assigned a unique address. This allows the CPU to simply process information as data and addresses and allows the program to handle where the information is being sent to. A *memory map* is a graphical representation of what address ranges various components are mapped to.
- There are three primary classes of instructions. These are loads and stores, data manipulations, and branches.
- Load instructions move information from memory into a CPU register. A load instruction takes multiple read cycles. Store instructions move information from a CPU register into memory. A store instruction

takes multiple read cycles and at least one write cycle.

- Data manipulation instructions operate on information being held in CPU registers. Data manipulation instructions often use inherent addressing.
- Branch instructions alter the flow of instruction execution. *Unconditional branches* always change the location in memory of where the CPU is executing instructions. *Conditional branches* only change the location of instruction execution if a status flag is asserted.
- Status flags are held in the condition code register and are updated by certain instructions. The most commonly used flags are the negative flag (N), zero flag (Z), two's complement overflow flag (V), and carry flag (C).

Exercise Problems

Section 11.1: Computer Hardware

11.1.1 What computer hardware subsystem holds the temporary variables used by the program?

11.1.2 What computer hardware subsystem contains fast storage for holding and/or manipulating data and addresses?

11.1.3 What computer hardware subsystem allows the computer to interface to the outside world?

11.1.4 What computer hardware subsystem contains the state machine that orchestrates the fetch-decode-execute process?

11.1.5 What computer hardware subsystem contains the circuitry that performs mathematical and logic operations?

11.1.6 What computer hardware subsystem holds the instructions being executed?

Section 11.2: Computer Software

11.2.1 In computer software, what are the names of the most basic operations that a computer can perform?

11.2.2 Which element of computer software is the binary code that tells the CPU which instruction is being executed?

11.2.3 Which element of computer software is a collection of instructions that perform a desired task?

11.2.4 Which element of computer software is the supplementary information required by an instruction such as constants or which registers to use?

11.2.5 Which class of instructions handles moving information between memory and CPU registers?

11.2.6 Which class of instructions alters the flow of program execution?

11.2.7 Which class of instructions alters data using either arithmetic or logical operations?

Section 11.3: Computer Implementation: An 8-Bit Computer Example

11.3.1 Design the example 8-bit computer system presented in this chapter in Verilog with the ability to execute the three instructions LDA_IMM, STA_DIR, and BRA. Simulate your computer system using the following program that will continually write the patterns x"AA" and x"BB" to output ports port_out_00 and port_out_01:

```
initial
  begin
   ROM[0]  = LDA_IMM;
   ROM[1]  = 8'hAA;
   ROM[2]  = STA_DIR;
   ROM[3]  = 8'hE0;
   ROM[4]  = STA_DIR;
   ROM[5]  = 8'hE1;
   ROM[6]  = LDB_IMM;
   ROM[7]  = 8'hBB;
   ROM[8]  = STB_DIR;
   ROM[9]  = 8'hE0;
   ROM[10] = STB_DIR;
   ROM[11] = 8'hE1;
   ROM[12] = BRA;
   ROM[13] = 8'h00;
  end
```

11.3.2 Add the functionality to the computer model from 11.3.1 the ability to perform the LDA_DIR instruction. Simulate your computer system using the following program that will continually read from port_in_00 and write its contents to port_out_00:

```
initial
  begin
   ROM[0] = LDA_DIR;
   ROM[1] = 8'hF0;
   ROM[2] = STA_DIR;
   ROM[3] = 8'hE0;
   ROM[4] = BRA;
   ROM[5] = 8'h00;
  end
```

11.3.3 Add the functionality to the computer model from 11.3.2 the ability to perform the instructions LDB_IMM, LDB_DIR, and STB_DIR. Modify the example programs given in Exercises 11.3.1 and 11.3.2 to use register B in order to simulate your implementation.

11.3.4 Add the functionality to the computer model from 11.3.3 the ability to perform the addition instruction ADD_AB. Test your addition instruction by simulating the following program. The first addition instruction will perform x“FE” + x“01” = x“FF” and assert the negative (N) flag. The second addition instruction will perform x“01” + x“FF” = x“00” and assert the carry (C) and zero (Z) flags. The third addition instruction will perform x“7F” + x“7F” = x“FE” and assert the two’s complement overflow (V) and negative (N) flags.

```
initial
  begin
    ROM[0]  = LDA_IMM; //-- test 1
    ROM[1]  = 8'hFE;
    ROM[2]  = LDB_IMM;
    ROM[3]  = 8'h01;
    ROM[4]  = ADD_AB;
    ROM[5]  = LDA_IMM; //-- test 2
    ROM[6]  = 8'h01;
    ROM[7]  = LDB_IMM;
    ROM[8]  = 8'hFF;
    ROM[9]  = ADD_AB;
    ROM[10] = LDA_IMM; //-- test 3
    ROM[11] = 8'h7F;
    ROM[12] = LDB_IMM;
    ROM[13] = 8'h7F;
    ROM[14] = ADD_AB;
    ROM[15] = BRA;
    ROM[16] = 8'h00;
  end
```

11.3.5 Add the functionality to the computer model from 11.3.4 the ability to perform the *branch if equal to zero* instruction BEQ. Simulate your implementation using the following program. The first addition in this program will perform x“FE” + x“01” = x“FF” (Z = 0). The subsequent BEQ instruction should NOT take the branch. The second addition in this program will perform x“FF” + x“01” = x“00” (Z = 1) and SHOULD take the branch. The final instruction in this program is a BRA that is inserted for safety. In the event that the BEQ is not operating properly, the BRA will set the program counter back to x“00” and prevent the program from running away.

```
initial
 begin
   ROM[0]  = LDA_IMM; //-- test 1
   ROM[1]  = 8'hFE;
   ROM[2]  = LDB_IMM;
   ROM[3]  = 8'h01;
   ROM[4]  = ADD_AB;
   ROM[5]  = BEQ;      //--NO branch
   ROM[6]  = 8'h00;

   ROM[7]  = LDA_IMM; //-- test 2
   ROM[8]  = 8'h01;
   ROM[9]  = LDB_IMM;
   ROM[10] = 8'hFF;
   ROM[11] = ADD_AB;
   ROM[12] = BEQ;      //-- Branch
   ROM[13] = 8'h00;

   ROM[14] = BRA;
   ROM[15] = 8'h00;
 end
```