

## SUPPLEMENT to DDPP-5E

### DESIGN OF A SIMPLE COMPUTER

Before we launch into the details associated with a relatively complex, contemporary microcontroller, it will be helpful for us to examine the design and implementation of a *simple* computer. In particular, the overall approach – based on a *top-down* specification of functionality, followed by a *bottom-up* implementation of the various functional blocks – will prove useful to our basic understanding of how a “real” microcontroller works.

*top-down,  
bottom-up*

Earlier we learned about a number of digital system building blocks. This included combinational elements such as decoders, priority encoders, and multiplexers as well as sequential elements such as latches and flip-flops. We then reviewed how these combinational and sequential elements can be combined to build digital systems. We also reviewed how digital systems could be specified using a hardware description language and subsequently implemented using *programmable logic devices* (PLDs).

*programmable  
logic devices*

Our purpose here is to apply this background to the design of a simple computer. Before we go any further, though, some basic definitions are in order. First, what is a *computer*? What distinguishes computers from random combinations of logic or from simple “light flashing” state machines? Simply stated, a computer is a device that *sequentially executes a stored program*. The program executed is typically called *software* if it is a user-programmable (“general purpose”) computer system; or called *firmware* if it is a single-purpose, non-user-programmable system (also referred to as a “turn-key” system). A given program consists of a *series of instructions* that the machine understands. Instructions are simply bit patterns that tell the computer what operation to perform on specified data. That a program is *stored* implies the existence of *memory*. To perform the series of instructions stored in memory, two basic operations need to be performed. First, an instruction must be *fetched* (read) from memory. Second, that instruction must be *executed*, e.g., two numbers are added together to produce a result. The memory that is used to store a program can take many different forms – ranging from removable media devices such as CD-ROMs to patterns in the metal layer of an integrated circuit. While the physical implementation of the memory in which the program is

*computer*

*stored program*

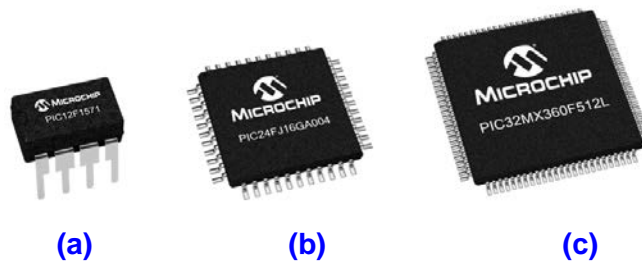
*software  
firmware*

*memory*

stored may vary, the information stored in memory is interpreted (i.e., fetched and executed) the same way.

Given the basic definition of a computer, above, what is a *microprocessor*? Classically, it is a single-chip embodiment of the major functional blocks of a computer. Today, though, the term “microprocessor” is often applied to a wide range of single- and multi-chip computational devices, ranging from “mainframes on a chip” (used in personal computers and workstations) to small dedicated controllers (used in a wide variety of “intelligent” devices). They can range in physical size from packages with several hundred pins to packages with only a few pins; some examples are illustrated in Figure S-1. They can range in cost from less than one dollar to hundreds of dollars. The simple computer we will be designing here can be implemented using a modest-size PLD; we could therefore rightfully call this single-chip embodiment of our simple computer a “microprocessor.”

*microprocessor*



**Figure S-1** Contrasting contemporary microcontrollers: (a) 8-bit PIC 12F family; (b) 16-bit PIC 24F family; and (c) 32-bit PIC 32M family. Illustrations © 1998-2016 Microchip Corporation, used by permission.

Finally, what is a *microcontroller*, and how does it differ from a microprocessor? Typically a microcontroller integrates, in addition to a microprocessor, a number of *peripheral devices* that are commonly used in control-type applications onto a single integrated circuit (and are thus often referred to as “single-chip microcontrollers”). Peripheral devices get their name from the fact that they provide interfaces with devices that are external (i.e., “peripheral”) to the computer. For example, a common series of operations often performed in control applications is: (1) input analog signals from sensors, (2) process them according to some algorithm, (3) and output analog control voltages to actuators. A device that digitizes an analog input voltage is called an analog-to-digital (A-to-D) converter. Conversely, a device that produces an analog output voltage based on a digital code is called a digital-to-analog (D-to-A) converter. A-to-D and D-to-A converters are examples of peripherals one might find integrated onto a microcontroller chip.

*microcontroller*

*peripheral devices*

Other common peripherals include communication controllers, timer modules, and pulse-width modulation (PWM) generators. In post-requisite courses we will see a variety of applications for all of these integrated peripherals.

## S.1 Computer Design Basics

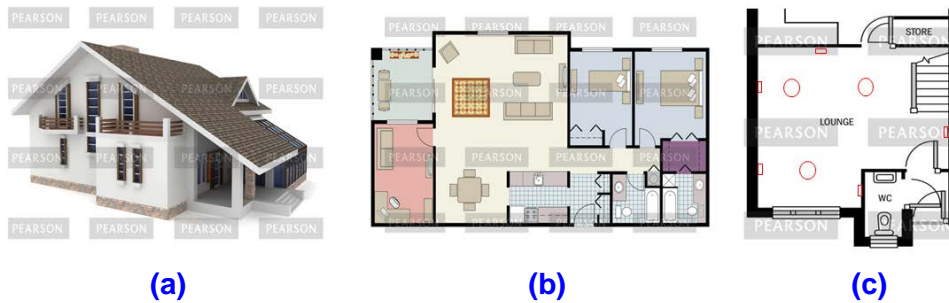
How can we apply what we have learned thus far about basic digital system building blocks toward building a simple computer? Basically, what we need is some way to structure and break down this design problem, because now it is a somewhat bigger than drawing a single state transition diagram or filling out a truth table. We will need a structured approach that enables us to take a written description of the functions performed by our simple computer and create a high-level block diagram. Based on this diagram, we can proceed to define what each block does, and ultimately design the circuitry required to implement each block.

Before starting this process, though, we need to define what we mean by the *structure* of a computer. “Architecture” is a word commonly used to depict the arrangement and interconnection of a computer’s functional blocks. While some might argue that this definition of computer architecture is a bit simplistic, it will serve our purposes for the discussion that follows.

*architecture*

Before starting to design our simple computer, let us first consider a “real world” analogy: building a house. Where is the logical place to start? Probably with a “big picture” – i.e., an *exterior elevation* or *plan view* of the entire project. Of course, the floor plan and exterior elevation are greatly influenced by the size, shape, and grade of the lot chosen for the house. Once we know the physical constraints dictated by our choice of lot, we can then begin to develop a floor plan. At this stage we can define the overall “functionality” of the house, i.e., the purpose of each room. Once we have defined the functionality of each room, the next step is to determine their arrangement and interconnection. Once we have a working floor plan, we can begin to embellish it with a number of details – for example, the location and size of windows, the location of light fixtures and their associated wall switches, the location of power outlets, the routing of plumbing, etc. The important thing to note from this analogy is that we have described a top-down design process: starting with a “big picture”, and progressively embellishing it with layers of details. Figure S-2 depicts such a progression.

*big picture*



**Figure S-2** Top-down design of a house: (a) the “big picture”, (b) the floor plan, (c) details of a particular room.

Once all the design specifications have been formulated, how would we proceed to build our house? From the ground up – assuming we have adequate financing, of course. We have to dig a hole first (perhaps analogous to going into debt), then pour a foundation, “stick-build” the basic structure, put a roof on it, complete the exterior walls, and finally embellish each room with its finishing details. Note that the *order* in which this “bottom up” implementation proceeds is quite important – certainly one would not wish to start hanging drywall before the roof is in place, or run plumbing lines before the floor joists are in place. Clearly, there is a *structured, ordered way* in which the entire process must take place – an approach strikingly similar to the one we will follow in designing our simple computer.

What would be a good name for the overall process described above? Ignoring the financial aspects for a moment, we could aptly call it the top-down specification of functionality followed by bottom-up implementation of each basic step (or “block”). More succinctly, we could call it *top-down specification and bottom-up implementation*. This is the process we will apply to the design and implementation of our simple computer.

*top-down  
specification*

*bottom-up  
implementation*

First, a disclaimer. The initial machine we design will be very, very simple. It will be an 8-bit machine with just a few instructions. Further, there will be a single *instruction format* (layout of bit patterns) as well as a single *addressing mode* (way that the processor accesses operands in memory). By the time we finish this “first phase” design, however, we will find out that even this rather simple machine is fairly complex in terms of implementation details.

*instruction  
format*

*addressing  
mode*

Once we have mastered our simple computer, we will then add “modern conveniences” such as input and output (or “I/O”), transfer of

control instructions, stack manipulation instructions, and subroutine linkage instructions. We will have the makings of a “socially redeeming” computer once we get done, plus have a firm footing upon which to understand the architecture and instruction set of a “real” microcontroller.

*socially  
redeeming*

## S.2 Simple Computer Big Picture

Just as one might begin the design of a house by sketching an exterior elevation view, we will begin the design of our simple computer with a “big picture” of its control console. In the “old days” (which was actually not so long ago), computers had lots of lights and switches on their front panels. The Digital Equipment Corporation PDP-8 (the first commercial “minicomputer”), illustrated in Figure S-3, was a good example of such a computer. The Intellect 8 microcomputer system (one of the first commercially-available microprocessor development systems) from Intel, based on the 8008 microprocessor, was another example. Frankly, these ground-breaking computer systems were a lot more interesting (and fun) to watch “crunch numbers” than today’s computers...and a lot less irritating than the “this application has performed an illegal function and will be shut down” message we’ve all become accustomed to today.

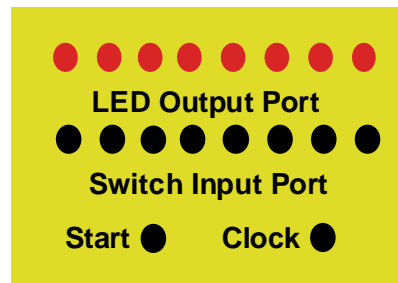
*old days*

*minicomputer*

*crunch numbers*



**Figure S-3** World’s first “desktop” mini-computer, the PDP-8 (photo by David Gesswein, retrieved from [www.pdp8.net](http://www.pdp8.net)).



**Figure S-4** Our simple computer console.

Our computer’s console, then, will have some lights that indicate the result of the most recent computation along with some switches that will be used to input data. A “START” pushbutton will be included to get the machine into a known initial state (in preparation for “running” a program), and a “CLOCK” pushbutton will be included to facilitate debugging (as we manually clock the machine from state-to-state). An “artist’s conception” of our simple computer’s console is shown in Figure S-4.

Returning to the “house analogy” for a moment, the floor plan of a computer is basically its *instruction set* and *programming model*. The instruction set is simply the list of operations that the computer performs. There are five fundamental groups (or categories) of machine instructions: data transfer, arithmetic, logical (or “Boolean”), transfer of control, and machine control. (Some computers include a sixth group dedicated to specific applications, e.g., multimedia extensions, digital signal processing, or graphics support.) The *addressing modes* that instructions can use to access operands in memory are also a key aspect of a computer’s instruction set.

*instruction set**programming model**addressing modes*

The *programming model* of a computer is the software writer’s view of the machine. Basically, it tells what resources are available for the programmer’s use, in particular, the machine’s registers. A register is simply a “memory location” within the processor that can be used to store intermediate results and/or as an operand (or as a *pointer* to an operand) used in a computation.

*pointer*

As alluded to above, the programming model and instruction set of our computer will be relatively simple. Initially there will only be one register, called the accumulator (or “A” register), so-named because it is the register in which the result of computations accumulate. Our computer will also include several condition code bits: a zero flag (ZF), negative flag (NF), overflow flag (VF), and carry flag (CF). Before we complete this chapter, we will add a stack pointer register and discuss the role of index registers.

*condition**code bits**ZF**NF**VF**CF*

The instructions executed by our simple computer will be of the fixed-length variety (i.e., all 8-bits in size, hence its designation as an “8-bit” computer) that consist of two fixed-length fields. The upper 3-bits of each instruction will indicate the operation to be performed, and is therefore called the *operation code* field (or “opcode” field). The lower 5-bits will indicate the memory address in which the operand is located (or, a result is to be stored). The 5-bit memory address dictates a maximum memory size of  $2^5 = 32$  locations. For those who have become jaded by multi-megabyte (even gigabyte) length programs that appear to do trivial things, this may not seem like much memory! Fortunately, though, it will be enough to illustrate basic principles of instruction execution, despite being too small to contain a “practical” (i.e., useful and socially redeeming) program.

*opcode field*

In addition to fixed-field decoding, another simplification in our initial design will be a single *addressing mode*. An addressing mode is the mechanism (or “function”) used to generate what is often called the

*addressing mode*

*effective address* of an operand, i.e., the actual address in memory where an operand is stored. The addressing mode our machine will support might aptly be called “absolute” addressing, based on the fact that this 5-bit field directly indicates the effective address in memory where the operand is stored. It is important to note at this point that not all manufacturers of microprocessors agree on the names ascribed to certain addressing modes. What we have just referred to as an “absolute” addressing mode is also called “extended” or “direct.”

*effective  
address*

*absolute  
addressing  
mode*

One other bit of terminology worth mentioning before delving into the instruction set concerns the number of addresses a given instruction (or more generally, a machine) can accommodate. Our simple computer here could be described as a “two address” machine, which means that two different locations (at two different addresses) are used in a given operation, e.g., ADD. In our computer, one location will be the “A” register (the accumulator), and the other will be contained in memory. Note that a “side-effect” of such an arrangement is that the result of the computation will overwrite one of the operands, here the value in the “A” register (the operand in memory will be unaffected). As one might guess, there are a lot of variations in instruction format and addressing capability, ranging from single-address instructions to three-address (or more) instructions.

*two-address  
machine*

### S.3 Simple Computer Floor Plan

We are now ready to introduce the “floor plan” (instruction set) of our simple computer. Note that we will initially define six of the eight possible instructions afforded by our 3-bit opcode field. We will save the last two opcode bit patterns to define some extensions to our instruction set later in this chapter. Our simple computer’s instruction set is given in Table S-1.

**Table S-1** Simple computer instruction set.

<i>Opcode</i>	<i>Mnemonic</i>	<i>Function Performed</i>
0 0 0	<b>HLT</b>	Halt – Stop, discontinue execution
0 0 1	<b>LDA</b> <i>addr</i>	Load A with contents of location <i>addr</i>
0 1 0	<b>ADD</b> <i>addr</i>	Add contents of <i>addr</i> to contents of A
0 1 1	<b>SUB</b> <i>addr</i>	Subtract contents of <i>addr</i> from contents of A
1 0 0	<b>AND</b> <i>addr</i>	AND contents of <i>addr</i> with contents of A
1 0 1	<b>STA</b> <i>addr</i>	Store contents of A at location <i>addr</i>

The “LDA” and “STA” instructions are examples of data transfer group instructions. As their *assembly mnemonics* imply, these instructions transfer data between the “A” register (accumulator) and memory. For the “load A” (LDA) instruction, the source of the data is memory location *addr*, and the destination is the “A” register. For the “store A” (STA) instruction, it is just the opposite: here, *addr* indicates the location in memory where the value in A (also referred to as the *contents of A*) is to be stored. As it turns out, “load” and “store” instructions are the “most popular” instructions in any machine’s instruction set, often comprising as much as 30% of the compiled code for typical applications.

*data transfer  
group instructions*

*assembly  
mnemonics*

**LDA**  
**STA**

A “shorthand” notation we will use throughout the remainder of this text is the use of parenthesis to indicate “the contents of” a particular register or memory location. This allows us to describe what an LDA instruction does as simply “ $(A) \leftarrow (\text{addr})$ ” and what an STA does as “ $(\text{addr}) \leftarrow (A)$ ”. An important point to note in both cases is that the *source* of the data transfer – i.e.,  $(\text{addr})$  for LDA and  $(A)$  for STA – *does not change* (or, *is unaffected*) as a result of the instruction execution.

Continuing down the list of available instructions, we next find two *arithmetic group* instructions: ADD and SUB. The ADD instruction performs the operation  $(A) \leftarrow (A) + (\text{addr})$  using *radix* (or *two’s complement*) arithmetic, and sets the *condition code* bits based on the result obtained. (Details on radix arithmetic and condition codes can be found in the in Chapter 2 of *DDPP*.) The SUB instruction performs the operation  $(A) \leftarrow (A) - (\text{addr})$  and sets the condition code bits accordingly. Following an ADD or SUB, the carry flag is the carry out of the most significant (or *sign*) position. One application of the carry flag is implementation of *extended precision arithmetic*, where the carry-out produced by adding a lower-order set of bits serves as the carry-in for the next set of higher-order bits (i.e. a *carry propagated forward*). For subtraction, the carry out produced is interpreted as the *complement of a borrow propagated forward*. This is identical to the convention for the carry flag used by ARM architectures.

*arithmetic group  
instructions*

**ADD**  
**SUB**

*two’s complement  
arithmetic*

*carry flag*

*extended precision  
arithmetic*

Moving down the chart, we find that our next instruction, AND, is from the logical (or “Boolean”) group. Because logical group instructions perform bit-wise operations, they are sometimes referred to as *bit manipulation* instructions. At minimum, most microprocessors worth their silicon generally have at least three Boolean instructions: AND, OR, and NOT (many also include XOR). Our simple computer, however, will just implement the first of these operations, which can be

*logical group  
instructions  
bit manipulation  
instructions*

**AND, OR,**  
**NOT, XOR**



described using the notation  $(A) \leftarrow (A) \cap (\text{addr})$ , where the “ $\cap$ ” symbol is used to denote the bit-wise logical AND of the two operands to produce the corresponding result bits.

Finally, no instruction set would be complete without a way to stop the machine. Our final (for now) instruction, HLT (for “halt”) serves this purpose. The HLT instruction is an example of a *machine control group* instruction. Execution of the HLT instruction will “freeze” the machine at its current point in the program being executed, and prevent the machine from fetching or executing any additional instructions until it is restarted (by pressing the START pushbutton described previously).

*HLT**machine control  
group instructions*

## S.4 Simple Computer Programming Example

To better understand how our simple computer operates, we will “walk through” the execution of a short program. This program will exercise each instruction in our simple computer’s repertoire. An important point to consider before proceeding is that it would be rather difficult to design a “simple” computer that directly interprets the instruction mnemonics (i.e., LDA, STA, etc.) we have defined. Rather, it is much easier to design a machine that directly interprets bit patterns (0’s and 1’s) that represent these instructions. This means that, before we can place our program in memory, we must translate the instruction mnemonics into bit patterns (“code”) the machine understands, called *machine code*. This translation process is called *assembly*, since machine code is created directly (“assembled”) based on instruction mnemonics. As one might guess, instruction mnemonics are typically referred to as *assembly level* mnemonics, or simply *assembly language*. A software program that translates assembly level mnemonics into machine code is called an *assembler*. If one is unfortunate enough to perform the translation by hand, the process is called *hand assembly*.

*machine code**assembly  
language**hand assembly*

Fortunately, most computer programming is done at a higher level of abstraction, using *high-level languages* such as “C”. Here, a *compiler* program is used to translate code written in high-level language into assembly code. An assembler program is then used to translate the compiler’s output into machine code for the target processor. We will find, though, that a firm grasp of assembly language programming techniques is essential for effectively utilizing the resources integrated into a modern microcontroller. Once we master assembly-level programming, we’ll consider how to program a microcontroller using

*high-level  
language  
compiler*

“C”. But to get there, we need to start at the “basic bit” level – so let’s return to the illustrative simple computer program in Table S-2.

**Table S-2** Programming example.

<i>Addr</i>	<i>Instruction</i>	<i>Comments</i>
00000	LDA 01011	Load A with contents of location 01011
00001	ADD 01100	Add contents of location 01100 to A
00010	STA 01101	Store contents of A at location 01101
00011	LDA 01011	Load A with contents of location 01011
00100	AND 01100	AND contents of 01100 with contents of A
00101	STA 01110	Store contents of A at location 01110
00110	LDA 01011	Load A with contents of location 01011
00111	SUB 01100	Subtract contents of location 01100 from A
01000	STA 01111	Store contents of A at location 01111
01001	HLT	Stop – discontinue execution

One of the first things we need to know is *where* in memory our program needs to be located. The logical thing to do is place our program at the *beginning* of memory, i.e., starting at location 00000<sub>2</sub>. We can then design the circuitry that, after the START pushbutton is pressed, begins fetching instructions from memory at location 00000<sub>2</sub>. Recalling that instructions are of fixed length (8 bits) and that memory locations are 8-bits wide, we realize that consecutive instructions will occupy consecutive memory locations. We can then imagine a “pointer” that tells us which instruction is to be executed, and that gets incremented after each instruction is fetched. Such a pointer is typically referred to as either an *instruction pointer* or a *program counter*.

*instruction pointer*  
*program counter*

A “snapshot” of what our short program looks like in memory prior to execution is provided in Figure S-5 (just the “first half” of memory, from locations 00000<sub>2</sub> to 01111<sub>2</sub> is shown). The lightly shaded part corresponds to the assembled machine code. Referring back to Table S-2, note that the first instruction (at address 00000<sub>2</sub>) is *load accumulator* (LDA) with the contents of memory location 01011<sub>2</sub>. Since the 3-bit opcode for LDA is “001”, this instruction is encoded as the bit pattern “001 01011” in memory. Stated another way, the instruction “LDA 01011” has been *assembled* into the machine code “001 01011”. We could go through a similar “hand assembly” process for the rest of the instructions that comprise the program, up to and including the HLT instruction at location 01001<sub>2</sub> (note that the address field of this instruction is not used, and is shown here to be “00000”).

Location	Contents
00000	001 01011
00001	010 01100
00010	101 01101
00011	001 01011
00100	100 01100
00101	101 01110
00110	001 01011
00111	011 01100
01000	101 01111
01001	000 00000
01010	
01011	10101010
01100	01010101
01101	
01110	
01111	

### ***Beam in the Bits, Scotty!***

One important detail we will ignore for the moment is how these bit patterns get loaded into memory. In a later chapter, we'll discuss how to write what's called a "loader" program, which – as its name implies – does just that. For now, assume Scotty (of *Star Trek* fame, for those of you much younger than the author) has used a molecular beam transporter to "beam the bits" into memory.

The operands used by each arithmetic (ADD, SUB) or logical (AND) operation will be stored at locations 01011<sub>2</sub> and 01100<sub>2</sub> (in the darker shaded area of Figure S-5); note that we have initialized these two locations to *arbitrarily chosen values*. The results of each operation (ADD, AND, SUB) will be stored in three consecutive locations, starting at location 01101<sub>2</sub>. Note that our computer's memory will contain a mix of instructions and data (operands and results).

### ***No Stopping It Now***

What happens if the HLT instruction is omitted? Perhaps even worse than "not stopping", the computer will start *executing data*, which, as one might imagine, is not a pretty sight (or, stated less formally, causes "bits to fly all over the place") and, at best, leads to *very strange* program behavior. Any "honest" programmer however, will confess that he/she has inadvertently done this "at least once..."

*executing data*

*honest programmer*

Given that our computer only understands 0's and 1's rather than the more human-friendly assembly mnemonics, the question that begs is: "How is our computer able to distinguish between instructions and data?" The hopefully obvious answer is: "It can't!" Rather, it has to be

*told* which locations contain instructions and which contain data. The convention we will use to make this distinction is that our programs will always start at location 00000<sub>2</sub> and continue until they reach a “halt” (HLT) instruction; any locations following the HLT instruction may be used for data (operands or results).

Location	Contents
00000	001 01011
00001	010 01100
00010	101 01101
00011	001 01011
00100	100 01100
00101	101 01110
00110	001 01011
00111	011 01100
01000	101 01111
01001	000 00000
01010	
01011	10101010
01100	01010101
01101	11111111 ← ADD
01110	
01111	

Add:

```

10101010
+01010101
-----
11111111

```

CF = 0  
NF = 1  
VF = 0  
ZF = 0

**Figure S-6** Result after executing the first three instructions.

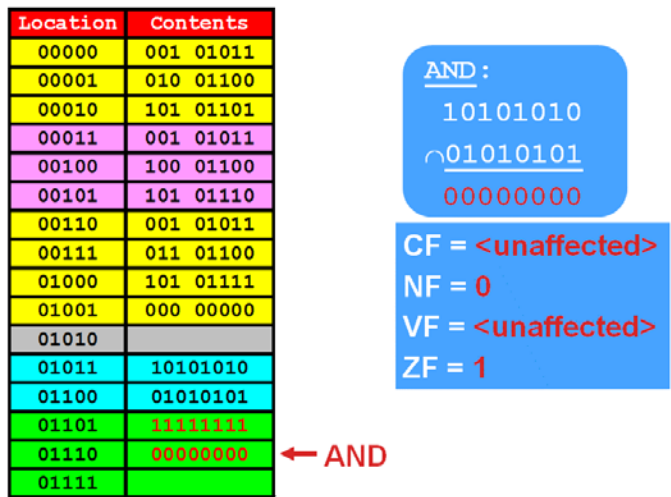
We are now ready to step through the execution of this program. Referring back to Table S-2, we see that the purpose of the first three instructions is to add the two operands (at locations 01011<sub>2</sub> and 01100<sub>2</sub>, respectively) and store the result at location 01101<sub>2</sub>. As illustrated in Figure S-6, the result obtained will be 11111111<sub>2</sub> (recall that this is the 8-bit representation for “-1” in two’s complement notation). Also, the negative flag (NF) will be set to “1”, the carry flag (CF) will be cleared to “0”, the overflow flag (VF) will be cleared to “0”, and the zero flag (ZF) will be cleared to “0”.

### Self-Perpetrating Programs

It is entirely possible to contrive a program that writes data into locations that contain instructions yet to be executed. The name “self-modifying code” has been used to describe such a creation. A self-modifying program, as one might guess, could prove to be excruciatingly difficult to debug. In a word, *don’t try this at home!* (And, don’t try to convince your boss that you’ve invented a new way to write “interesting” programs!).

*self-modifying  
code*

Again referring back to Table S-2, we see that the purpose of the next three instructions is to logically AND the two operands and store the result at location 01110<sub>2</sub>. Note that, for the AND operation, the carry flag (CF) and overflow flag (VF) are meaningless, and therefore should be *unaffected* by the execution of the AND instruction. The result obtained, however, may be negative (in a two’s complement sense) or zero, so the negative flag (NF) and zero flag (ZF) should be affected. A snapshot of memory following execution of the three AND-related instructions is provided in Figure S-7. Note that, since the result obtained is 00000000<sub>2</sub>, the zero flag is set to “1”.



**Figure S-7** Result after executing the “middle” three instructions.

The purpose of the next group of three instructions is to take the difference of the two operands at locations 01011<sub>2</sub> and 01100<sub>2</sub>. Specifically, we are going to subtract (SUB) the operand at location 01100<sub>2</sub> from the operand at location 01011<sub>2</sub>, and place the result at location 01111<sub>2</sub>. Recall that a radix subtraction is realized by forming the two’s complement of the subtrahend (here, the operand at location 01100<sub>2</sub>) and adding it to the minuend (the operand at location 01011<sub>2</sub>). Further, the easiest way to generate the radix complement of a signed number is to add one to its *diminished radix* complement (or *ones’ complement*). Figure S-8 shows what happens. Note that, while the result 01010101<sub>2</sub> will be stored at location 01111<sub>2</sub>, it is *invalid* because overflow has occurred (denoted by VF set to “1”). Note also that CF is set to “1” due to its interpretation here as the *complement* of the *borrow flag*: a carry flag of “1” following a subtract operation essentially means that “no borrow is propagated forward” to the next higher order set of bits.

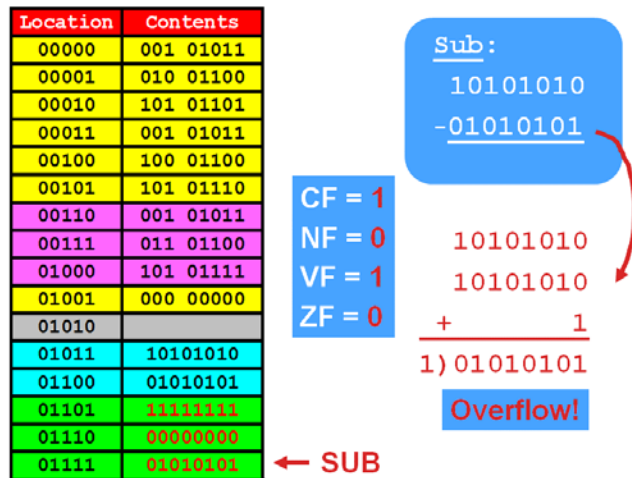


Figure S-8 Result after executing the last group of three instructions.

### Bumblng Borrow

Perhaps the single-most issue that causes students consternation is that of carries and borrows propagated forward. The interpretation of a “carry propagated forward” following an addition is no problem; but when it gets to subtraction, all “bits are off” (pardon the very bad pun). Here, the proper interpretation is as a “borrow propagated forward” to the next-most significant group of digits in an extended precision subtraction. The carry flag, when *cleared* (i.e. meaning there *is* a borrow propagated forward), is basically telling that next group of digits to “reduce its result by one” because the previous stage “has borrowed from it.” The best real-world analogy that comes to mind is that of a statement from your friendly, local banking institution listing the service charge they have extracted from your account for the privilege of serving you. The point is: since they have already taken the money, you need to adjust your idea of how much money you have left!

Before we leave this last block of code, yet another question that comes to mind is: “How should error conditions like overflow be handled?” As one might guess, we will need some “new” instructions that allow us to test the state of the various condition codes (here, VF) and transfer control to a different part of the program (typically called an “exception handler”) if an error has occurred. Before we finish this chapter, we will learn how to implement such “conditional transfer of control” instructions.

The final instruction in our short program, HLT, simply tells our computer to “stop executing”. Once the program has stopped, we could presumably look at the contents of each location to determine the results of the program execution. What we should find is the memory image depicted in Figure S-8 (note that memory location 01010<sub>2</sub> was unused by our example program and may contain a “random” value).

## S.5 Simple Computer Block Diagram

Now that we know *how* our simple computer works, we are ready to consider the functional blocks necessary to *make* it work. Basically we want to build what appears to be a “big state machine” that performs the calculations just done by hand. At a fundamental level, there are two basic steps associated with the processing of each instruction. The first step is to read the instruction from memory, called an *instruction fetch cycle*. The second step is to extract the opcode and address fields from the instruction just fetched and perform the operation specified by the opcode on the data located at the specified address; this step is referred to as an *instruction execute cycle*.

*instruction  
fetch cycle*

*instruction  
execute cycle*

What are the basic functional blocks, then, that are necessary to implement the simple computer described here? Clearly, a memory unit – for storing instructions and data – is one of the major functional blocks necessary. This memory unit needs to be capable of reading the contents of a specified location (indicated on its address lines) as well as writing a new value to a specified location.

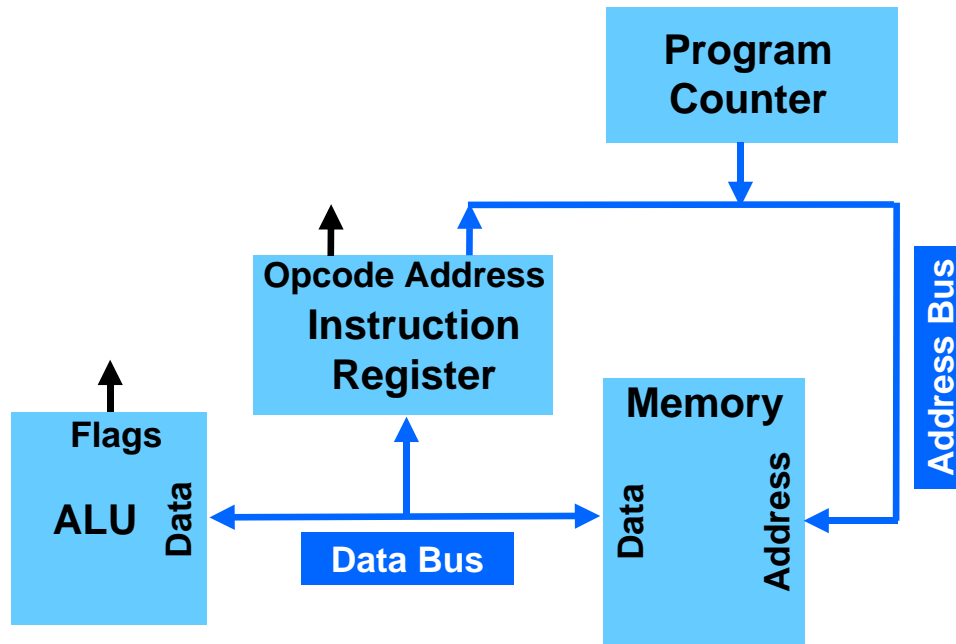
*memory unit*

Another major functional block needed is one that will keep track of which instruction is next in line to be executed. In our simple computer, the instructions are stored in consecutive memory locations, starting at location 00000<sub>2</sub>. What is needed is a pointer that keeps track of which instruction is next. Because this block is nothing more than a binary counter, we will call it the *program counter* (PC).

*program counter  
PC*

Once it is fetched from memory, a place is needed to temporarily “stage” an instruction while the opcode field is decoded and the address field is extracted. We can think of this block as a place to hold the instruction just fetched while it is being “digested”. While more creative, biologically inspired names for it are certainly possible, we will simply call this functional block the *instruction register* (IR).

*instruction register  
IR*



**Figure S-9** Simple computer core block diagram.

Next we realize the need for a functional block that performs the arithmetic and logical operations we have defined in the simple computer's instruction set. Not surprisingly, this block is usually called an *arithmetic logic unit*, or simply ALU. Note that the accumulator ("A" register) and condition code bits (CF, NF, VF, ZF) are part of the ALU.

*arithmetic logic unit*  
*ALU*

Finally, we realize that our simple computer needs a "manager" – a functional block that orchestrates the activities of all the other functional blocks delineated above. This "manager" is responsible for indicating whether a fetch or an execute cycle is to be performed and, once an instruction is fetched, for decoding the opcode field of that instruction and telling the other blocks in the system what to do in order to execute it. Because our simple computer's "manager" controls the sequencing of events that, taken together, constitute the completion of a machine instruction, we often refer to the state machine part of the manager's personality as a *micro-sequencer* (similar to, perhaps, but not to be confused with a "micro-manager"). And because decoding the opcode field of the instruction is an essential part of the sequencing process, we award our simple computer's manager the grand and glorious name: *instruction decoder and micro-sequencer* (IDMS). This more extravagant sounding name helps prevent images of "kicking bits around" that might be associated with a "manager" (think baseball).

*manager*

*micro-sequencer*

*IDMS*



Returning to the “house” analogy for a moment, what we have just done is “define the rooms” of the “structure” (or system) we wish to build. What we have not yet done, however, is interconnect the functional blocks into a working “floor plan”. In order to do this, we need an understanding of the “traffic patterns” (here, of address, data, and control information) that need to flow among the various functional blocks.

Starting with the memory unit, we note that a series of address lines tell which location is being accessed; the collection of address lines is referred to as the *address bus*. (Recall that a *bus* is a set of signal lines that have a *common purpose*.) At the location in memory accessed, data can be *read* (output) or *written* (input); the memory’s data lines (and the associated data bus) must therefore be *bi-directional*. Further, control signals need to be supplied to the memory unit that tell whether or not it is *enabled* to respond (or *selected*), and, if enabled to respond, whether it should perform a read operation or a write operation.

*address bus*

*bi-directional*

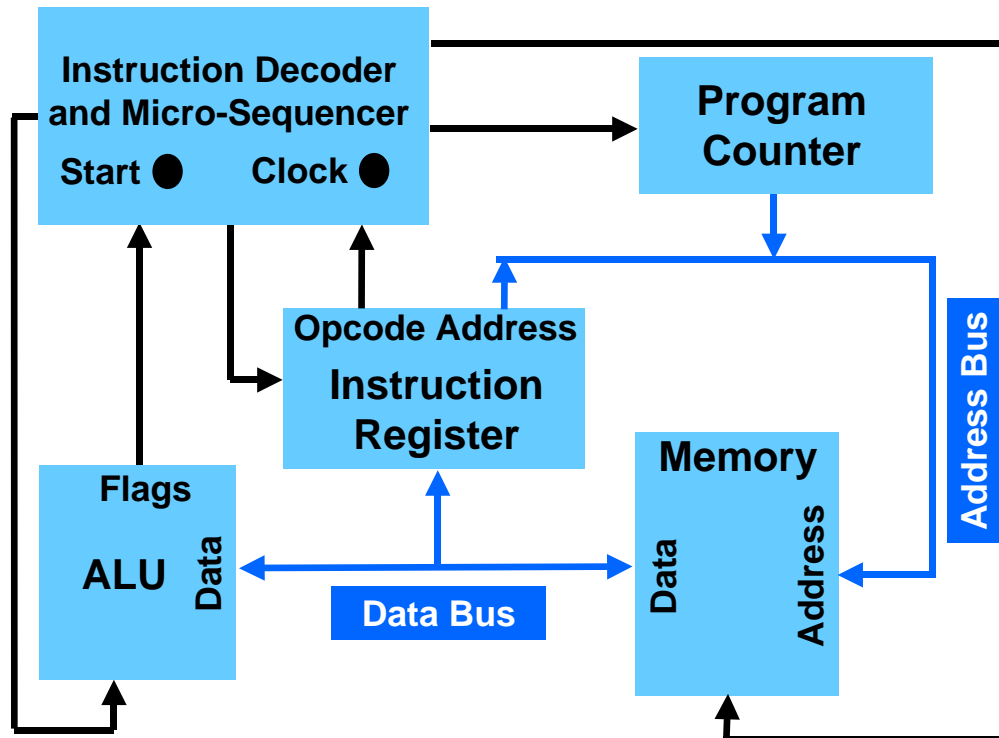
Next, we realize that the program counter (PC) will supply the instruction address to memory during a fetch cycle, and that the instruction register (IR) will be used to temporarily stage the instruction after it has been read from memory. Further, on an execute cycle, the IR will supply the operand address to memory, and the destination (or source) of the data in this transaction is the “A” register of the ALU. Thus, there are two potential sources of address information – the PC and the IR – on the address bus. Since only one device can “talk” on the bus at a given instant in time, we will need to provide each of these functional blocks with *three-state output* capability – and it will be our “manager’s” job to keep them from talking at the same time!

*three-state output  
capability*

Further, there are two potential destinations of data read from memory. On a fetch cycle, an instruction destined for the IR is read from memory. On an execute cycle, an operand destined for the ALU is read from memory (alternately, data in the ALU is destined for memory if an STA instruction is being executed). Again, we note the need for three-state buffers in all the functional blocks involved with driving the data bus.

Putting this all together, the “core” of our simple computer is depicted in Figure S-9. Left on their own, however, these functional blocks are incapable of doing anything “intelligent”, let alone successfully executing instructions. Hence the need for a “manager” – the instruction decoder and micro-sequencer – to tell each block what to

do when. As such, the IDMS can aptly be thought of as the “heart” of the machine. The simple computer augmented with an IDMS is shown in Figure S-10.



**Figure S-10** Complete simple computer block diagram.

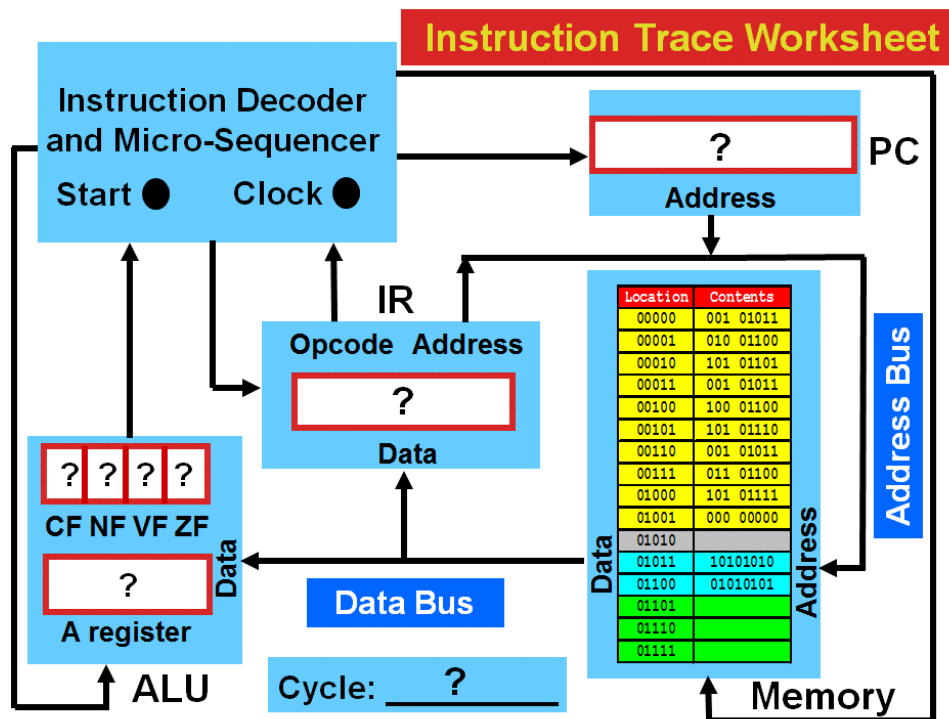
We now have a complete “floor plan” for our “digital house”, that we have specified in a top-down fashion. Before actually building it, though, let’s make sure we understand how the “rooms” work together.

## S.6 Instruction Execution Tracing

To get a better idea of how the various functional blocks of our simple computer work in concert to process instructions, we will return to our short program of Table S-2 and use a technique called *instruction tracing* to help us visualize the flow of information. On a cycle-by-cycle basis, we will examine the address and data paths as well as the bit patterns in each register for the first three instructions of this short program. Recall that we used the term “micro-sequencer” because there is a sequence of events associated with processing an instruction: here, a fetch cycle followed by an execute cycle.

*instruction tracing*

The instruction trace worksheet in Figure S-11 sets the stage for this exercise, which shows the initial state of the machine after START is asserted. Note that there are several things we will keep track of as our machine executes the program. In particular, we will be monitoring what happens to the PC, IR, and “A” register as well as the contents of memory. We will also practice naming each cycle as it occurs.



**Figure S-11** Instruction trace worksheet for machine state after START is asserted, prior to first fetch cycle.

Recall that pressing the START pushbutton places the machine in a known initial state: the PC is reset to “00000” and the state counter (in the IDMS) is set to “fetch”. Note that the initial state of the IR and ALU may be “random” and that memory is initialized to the values indicated (although at this point we “don’t care” what is in the unused location 01010<sub>2</sub> or the locations where the results will be stored, 01101<sub>2</sub>–01111<sub>2</sub>).

During the first fetch cycle, shown in Figure S-12, the instruction at memory location 00000<sub>2</sub> is read and placed in the IR. As the IR is being loaded with the instruction, the PC is incremented by one (i.e., once the fetch of the current cycle is complete, the PC is pointing to the *next* instruction to execute). Note that the values in each register are those obtained *after* the “fetch LDA” cycle is *complete*.

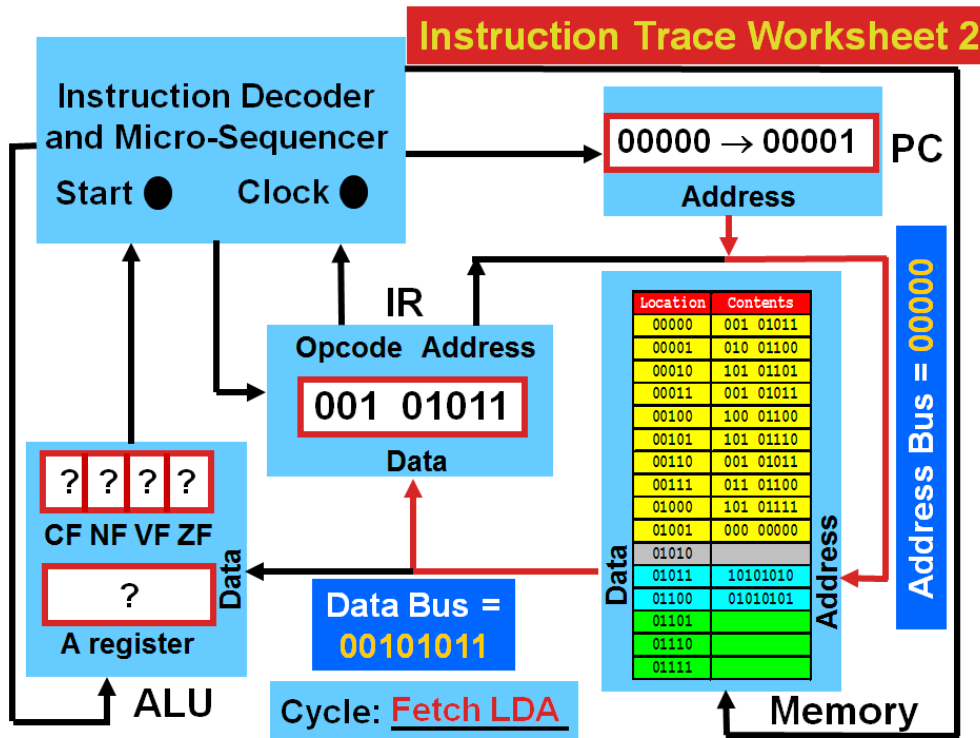


Figure S-12 Instruction trace worksheet for first fetch cycle.

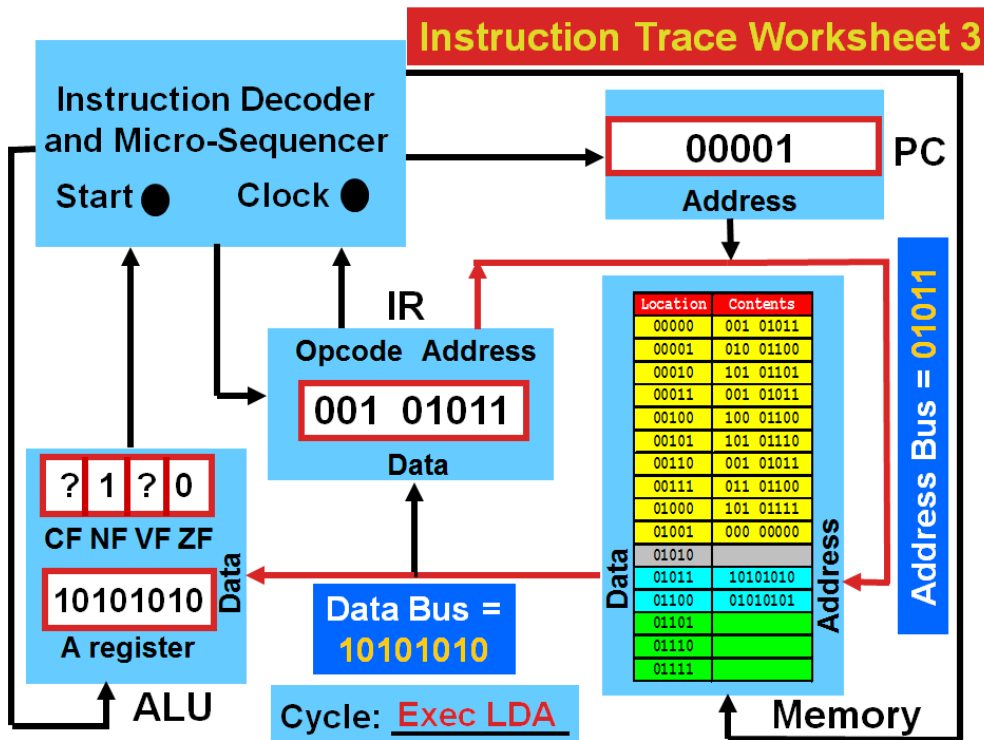


Figure S-13 Instruction trace worksheet for first execute cycle.

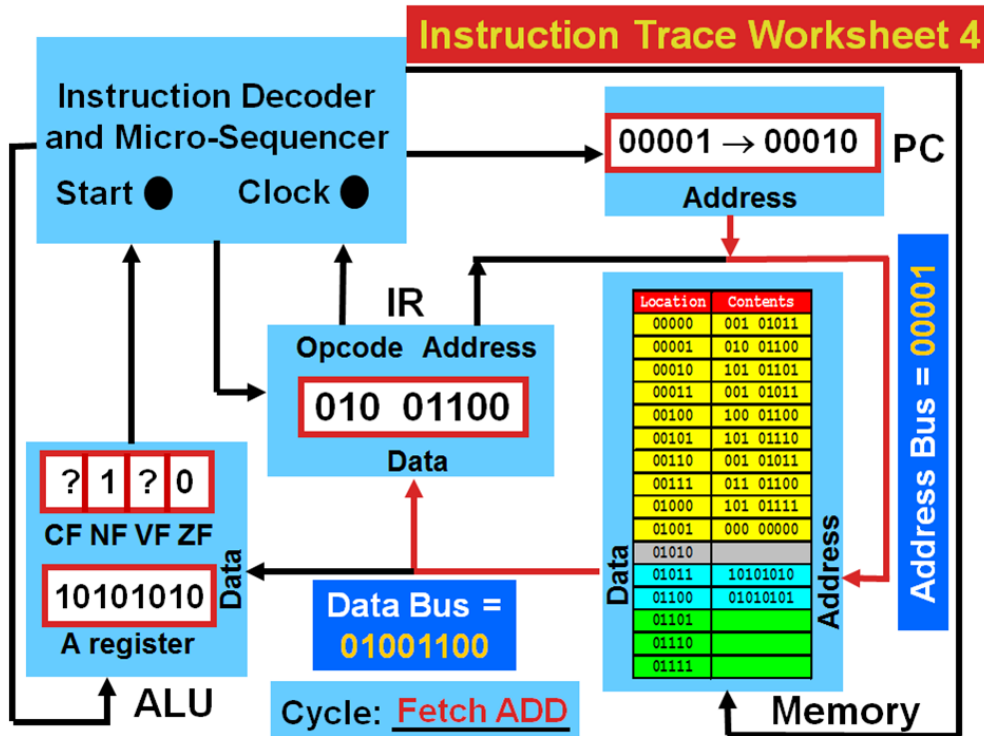


Figure S-14 Instruction trace worksheet for second fetch cycle.

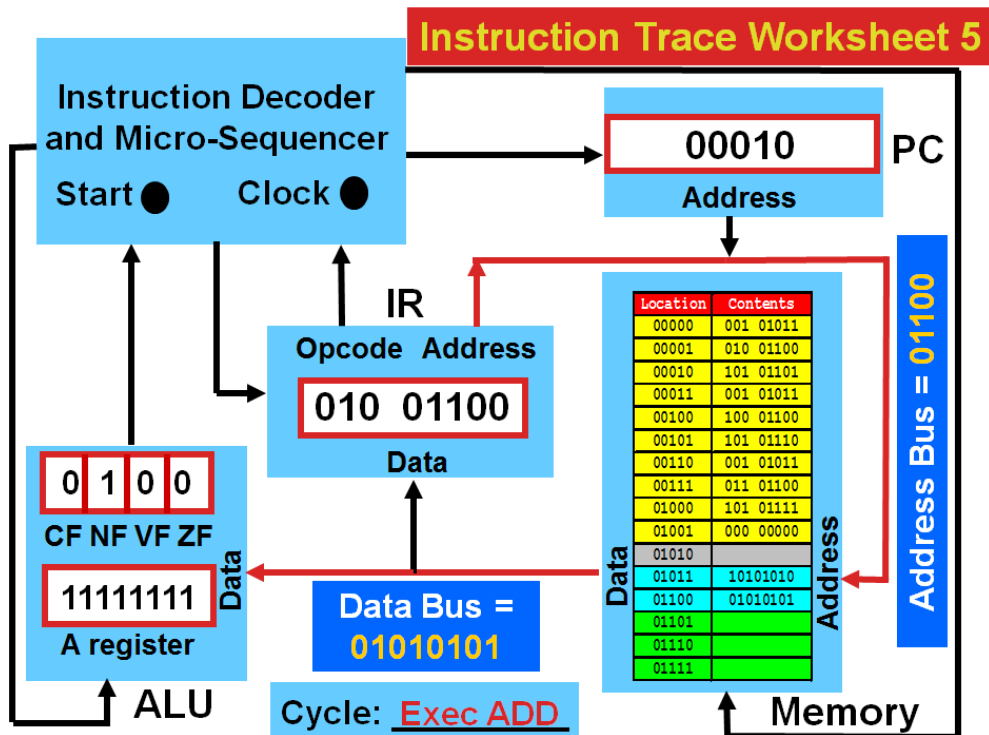


Figure S-15 Instruction trace worksheet for second execute cycle.

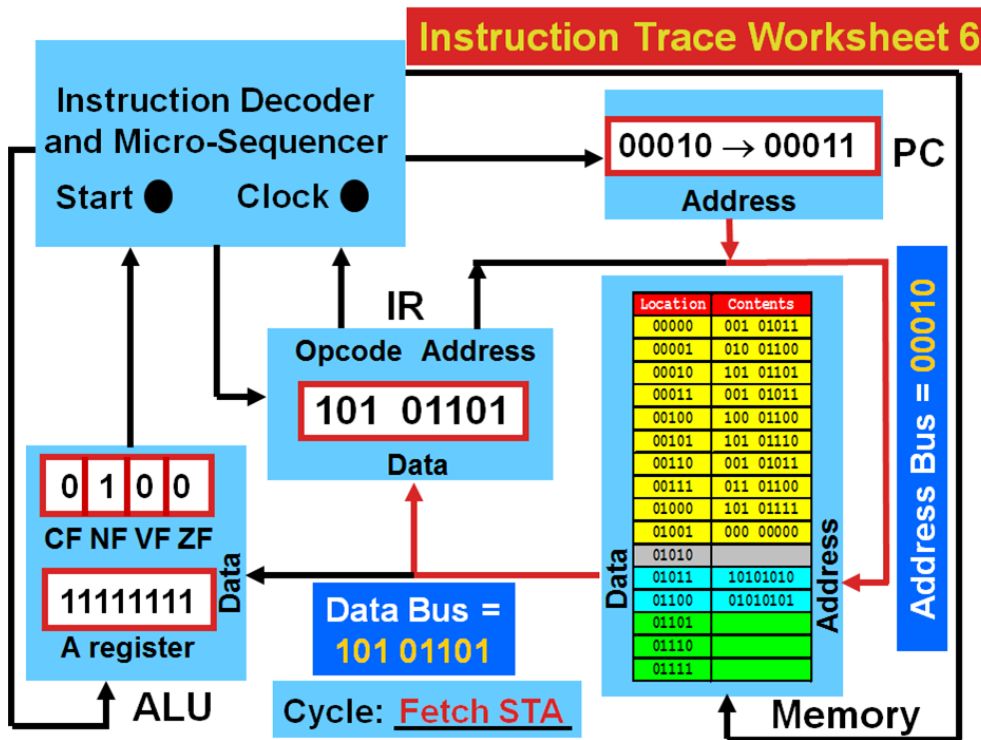


Figure S-16 Instruction trace worksheet for third fetch cycle.

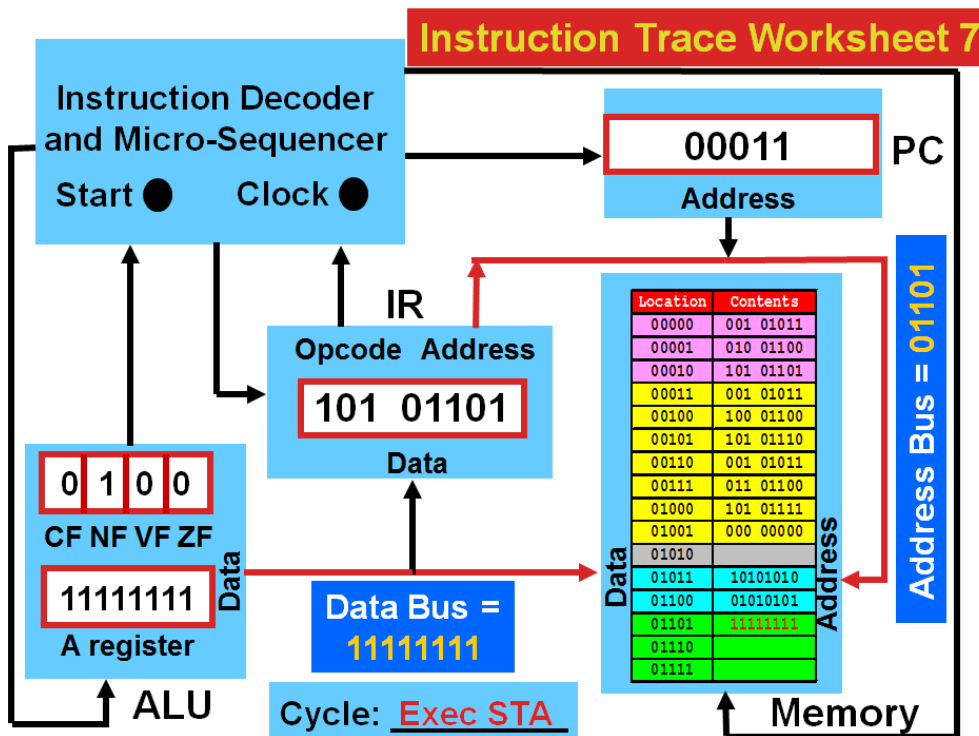


Figure S-17 Instruction trace worksheet for third execute cycle.

During the first execute cycle, shown in Figure S-13, the “LDA 01011” instruction in the IR is executed. When this cycle is complete, the “A” register contains the contents of memory location  $01011_2$ , i.e., the value  $10101010_2$ . Note also that the NF is set to “1” and ZF is cleared to “0”. The “execute LDA” cycle does not, however, affect the contents of any memory location, nor does it change the contents of IR or PC (condition code bits CF and VF are also unaffected).

We are now ready for the second fetch cycle (“fetch ADD”), shown in Figure S-14. Here, the instruction at memory location  $00001_2$  is fetched and placed into the IR, and as that occurs, the value in the PC is incremented by one. The results of executing the ADD instruction are shown in Figure S-15. Here, the contents of memory location  $01100_2$  (i.e., the value  $01010101_2$ ) are added to the value previously loaded into the “A” register. A result of  $11111111_2$  is obtained, along with condition code bits CF = “0”, NF = “1”, ZF = “0”, and VF = “0”.

This brings us to the third fetch cycle (“fetch STA”) of our tracing example, shown in Figure S-16. Here, the instruction at memory location  $00010_2$  is fetched and placed into the IR, and as that occurs, the value in the PC is incremented by one. The results of executing the STA instruction are shown in Figure S-17. Here, the contents of the “A” register are stored at the memory location indicated in the instruction’s address field:  $01101_2$ . When the “execute STA” cycle is complete, then, memory location  $01101_2$  contains the value  $11111111_2$ . Note, however, that the “A” register as well as the condition code bits are unchanged.

Several observations are in order. First, all of our simple computer’s fetch cycles are identical (i.e., they are independent of the instruction opcode). In fact, this *has* to be the case, since our machine basically knows nothing about the instruction being fetched until it is placed in the IR. Second, it may appear “strange” that our simple computer is incrementing the value in the PC on the same cycle that it is being used as a pointer to memory. Another way to say this is that the increment of PC is *overlapped* with the fetch of the instruction. The reason this can happen will become apparent when we start implementing each functional block in the next section. For now, though, suffice it to say that because each register will be implemented using edge-triggered flip-flops, the same clock edge that causes the IR to load the instruction being fetched also causes the PC to increment. The IR, though, will be loaded with the value on the data bus *prior* to the clock edge, while the value output by the PC (driving the address bus) will change *after* the clock edge – thus facilitating the desired

*overlapped*

overlap. This is an important point that we will revisit several times before the end of this chapter.

One final suggestion before we move to the “bottom-up” phase of our simple computer design process. Practice the “instruction tracing” process outlined in this section on other code segments to become more familiar with “what happens when” as each instruction is fetched and executed. As we say in the education industry, this is a “good test question” (GTQ)!

*good test  
question*

## S.7 Bottom-Up Implementation of Simple Computer

Armed with a thorough understanding of how our simple computer works, we are now ready to start building it from the bottom-up. In practice, the preferred approach is to implement and test each block as it is designed. Then, when we put the various functional blocks together, we have a much better chance of the entire system working “the first time”.

### S.7.1 Memory

The block we will start with is memory. Although most of the time we would simply choose a “memory chip” of appropriate size and speed, a knowledge of “what’s under the hood” is essential to understanding how the various functional blocks of our simple computer work together.

First, some terminology. Normally, we think of memory as an entity that, from the computer’s perspective, can be “read” or “written”. In “read” mode, the memory unit simply outputs, on its data bus lines, the contents of the location indicated on its address bus inputs. In “write” mode, the memory unit stores the bit pattern present on its data bus lines at the location indicated on its address bus inputs. The correct acronym to describe such a “read/write memory” is RWM. Despite valiant efforts, the name RWM never caught on. Instead, it is more popular to refer to these devices as “random access memories” or RAMs – so-named because any (random) location can be accessed in the same amount of time (not because something random is read after a given value is written).

The specific type of RAM we wish to concentrate on here is *static* RAM, or SRAM. This is in contrast to *dynamic* RAM (DRAM), which requires constant refreshing to retain information. (In DRAM, data is

*static RAM (SRAM)*  
*dynamic ram (DRAM)*



stored as a charge on a capacitor – since the charge dissipates over time, it must be periodically refreshed.) SRAM consists of a collection of D latches that will retain data (without the need for refreshing) as long as power is applied. Once power is turned off, however, all information previously stored in the SRAM is lost (this is referred to as a *volatile* memory).

*volatile  
memory*

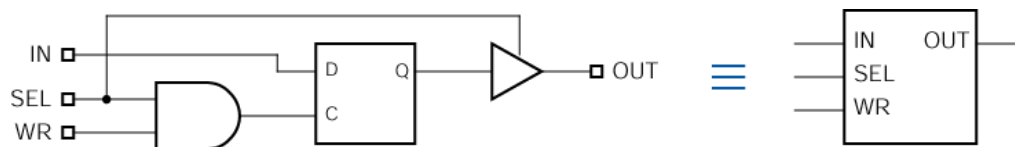
In addition to address and data bus connections (where, for our simple computer, the address bus is 5-bits wide and the data bus is 8-bits wide), an SRAM needs three control signals. First, an SRAM needs an overall enable, typically called a “chip select” (CS) or “chip enable” (CE). This enable signal is needed to differentiate among multiple SRAMs or, as we will see later in this chapter, between memory and input/output devices. Second, an SRAM needs an output enable (OE) signal which, provided the SRAM is selected, turns on a series of three-state buffers that drive the data from the addressed location out onto the data bus. Finally, an SRAM needs a write enable (WE) signal which, if the SRAM is selected, opens the row of latches associated with the addressed location and allows it to take on the value presented to the SRAM on the data bus.

*chip select  
(CS)*

*output enable  
(OE)*

*write enable  
(WE)*

The basic building block of an SRAM is a memory cell, such as the one depicted in Figure S-18, consisting of a D-latch and a three-state buffer. When the *select* (SEL) signal is asserted, the three-state buffer is enabled, placing the data stored in the latch on the cell’s OUT line. When both SEL and WR are asserted, the latch opens and accepts the data present on the IN line (by virtue of asserting the latch enable or “C” input of the D-latch). When WR is negated, the latch closes and retains the new value.

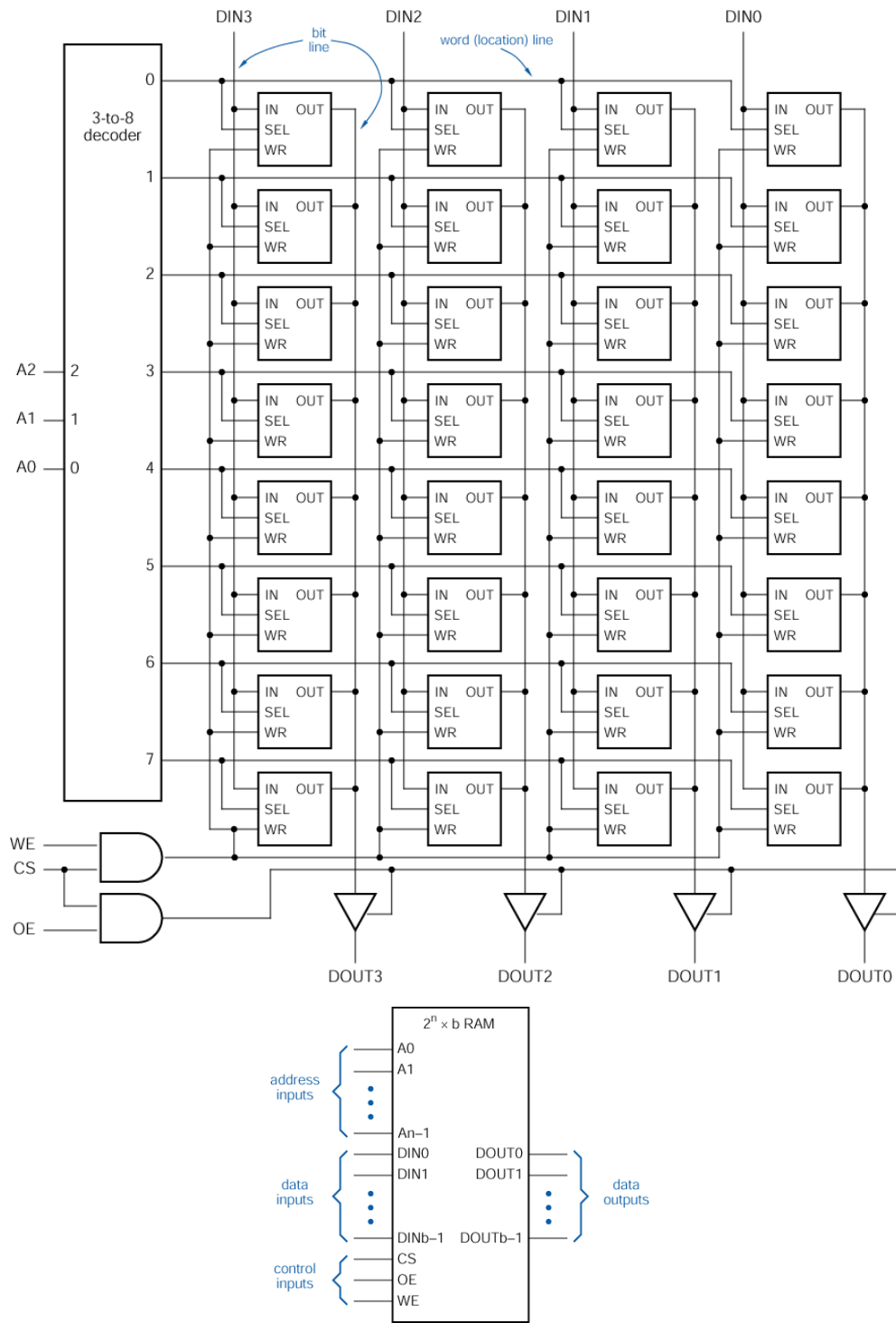


**Figure S-18** SRAM cell (adapted from *DDPP-4E*).

A complete SRAM can be constructed by combining an array of memory cells with a (large) decoder plus some additional logic. The internal structure of an eight location, 4-bit wide (or, “8x4”) SRAM is shown in Figure S-19. Note that the number of address lines needed is  $\log_2(\text{number\_of\_locations})$ ; here,  $\log_2(8) = 3$ . Stated another way, the number of locations in an SRAM is  $2^n$ , where  $n$  is the number of address lines. A “location” in the SRAM corresponds to a *row* of

*memory  
location*

memory cells; to select a particular row, an  $n$ -to- $2^n$  binary decoder is needed.



**Figure S-19** SRAM internal structure and symbol (adapted from DDPP).

### GigaBiga Dittos

The prefixes K (kilo-), M (mega-), G (giga-), and T (tera-), when referring to memory sizes, mean  $2^{10} = 1024$  (“about one thousand”),  $2^{20} = 1,048,576$  (“about one million”),  $2^{30} = 1,073,741,824$  (“about one billion”), and  $2^{40} = 1,099,511,627,776$  (“about one trillion”), respectively. This brings up a very important question: Does this mean the once-feared “Y2K bug” is yet to occur (in year 2048)? An even more important question, though, might be: Instead of calling a billion bytes a “gigabyte”, wouldn’t a better name be “bigabyte” (as in *Biga* (short for “Bigger”) *Bytes of Digital Wisdom*, the subtitle for this supplemental text)?

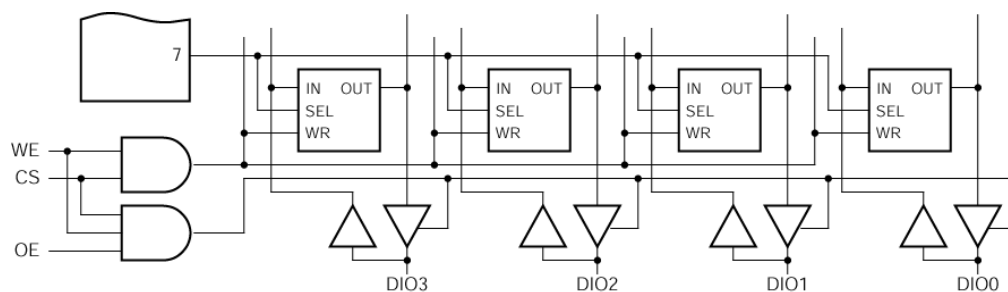
*kilo-, mega-,  
giga-, tera-*

*bigabyte*

In addition to a decoder, some logic is needed to “qualify” the actions associated with the OE and WE signals based on the assertion of CS (the overall chip enable). When WE is asserted in conjunction with CS, the data present on the DIN pins (DIN3 – DIN0) is written at the location specified on the address lines (note that the operation completes upon negation of the WE signal). When OE is asserted in conjunction with CS, the data output by a given row is routed to the three-state buffers that drive the external data lines.

Since the read and write operations are mutually exclusive, however, there is usually no need for separate data input and output lines. Instead, the data input and output lines are tied together and connected to the rest of the system using a *bi-directional* data bus. Such a configuration is shown in Figure S-20. Note that an additional buffer is used to receive the incoming data during a write operation, to reduce the load seen by the entity driving the bus.

*bi-directional  
data bus*



**Figure S-20** SRAM bi-directional data bus (adapted from *DDPP*).

Before moving on, a few notes concerning memory timing are in order. Because an SRAM read operation is a purely combinational function, the *order* in which the address and control signals (CS and OE) are asserted is of no consequence. A future topic, though, concerns how each of these signals represents a *critical timing path* with respect to receiving valid data from memory on a read cycle:  $t_{AA}$  is the address access (propagation delay) time,  $t_{CS}$  is the chip select access time, and  $t_{OE}$  is the output enable access time. When interfacing an SRAM to a computer, all of these “read” paths need to be analyzed.

*critical  
timing path*

$t_{AA}$   
 $t_{CS}$   
 $t_{OE}$

Since a “D” latch is used to store each bit of data in an SRAM, the *timing relationship* between the information on the address and data buses as well as the requisite control signals (CS and WE) is *more stringent* than for a read cycle. In particular, the address information needs to be stable, and the chip select (CS) needs to be asserted, for some time ( $t_{CW}$ ) before WE is asserted (opening the set of latches associated with the selected location). Also, the information supplied to the SRAM on the data bus must be stable  $t_{SETUP}$  prior to the negation of the WE signal, and  $t_{HOLD}$  following the negation of the WE signal. The *consequence* of violating the data setup or hold timing specifications of an SRAM, or of not asserting the WE control signal for a sufficient period of time, is the *possibility of metastable behavior*. All of these “write”-related timing parameters need to be analyzed when interfacing an SRAM to a computer.

$t_{CW}$

$t_{SETUP}$   
 $t_{HOLD}$

*metastable  
behavior*

Returning to our simple computer, we note that by simply doubling the “width” of the SRAM depicted in Figure S-19 (from 4-bits to 8-bits) and quadrupling the “length” (from 8 locations to 32 locations), as well as adding the bi-directional data bus interface shown in Figure S-20, we will have the exact structure of SRAM needed. The only difference is the “unique” names we will use for our simple computer’s memory control signals: “MSL” for the memory select signal, “MOE” for the memory output enable, and “MWE” for the memory write enable.

**MSL**  
**MOE**  
**MWE**

## S.7.2 Program Counter

The next functional block we wish to address is the program counter (PC). Basically, this is nothing more than a (5-bit) binary “up” counter with an asynchronous reset and three-state outputs. The asynchronous reset (ARS) will be connected to the START pushbutton, so that the first instruction fetched is from location  $00000_2$ . There are two other control signals needed: one that enables the PC to increment by one when a low-to-high (“positive edge”) of the system CLOCK signal occurs, which we will call PCC; and one that turns on

**ARS**

the three-state buffers that “gate” the value in the PC onto the address bus, which we will call POA. Note that if PCC is negated while a positive CLOCK edge occurs, the program counter should simply retain its current state.

*PCC*  
*POA*

To document the design of each functional block, we will present a Verilog code module. The Verilog source file for the program counter module is shown in Table S-3.

*Verilog*

**Table S-3** Program counter module.

```

/* Program Counter Module */

module pc(CLK, PCC, POA, RST, ADRBUS_z);

    input wire CLK;
    input wire PCC;    // PC count enable
    input wire POA;    // PC output on address bus tri-state enable
    input wire RST;    // asynchronous reset (connected to START)
    output wire [4:0] ADRBUS_z;

    wire [4:0] next_PC;
    reg [4:0] PC;

    assign ADRBUS_z = POA ? PC : 5'bZZZZZ;

    always @ (posedge CLK, posedge RST) begin
        if (RST == 1'b1)
            PC <= 5'b00000;
        else
            PC <= next_PC;
        end

    //          (PCC) ? count up : retain value;
    assign next_PC = (PCC) ? (PC+1) : PC;

endmodule

```

Examining the Verilog code, we see that when PCC is negated, the next state is simply the current state. When PCC is asserted, the equations for a synchronous 5-bit binary “up” counter determine the next state. Assertion of POA causes the three-state buffers associated with each register bit to be enabled, and assertion of ARS causes each flip-flop comprising the PC to be asynchronously reset.

### S.7.3 Instruction Register

The instruction register (IR) has a very simple mission: temporarily hold (“stage”) the instruction fetched from memory so that it can be “peeled apart” and executed. As such, it is simply a series of D flip-flops with two control signals. The first control signal, which we will call IRL, enables the instruction register to be loaded with the instruction read from memory; the load should occur on the positive edge of the system CLOCK. The second control signal, which we will call IRA, turns on the three-state buffers of the lower 5-bits of the IR, to “gate” the address field of the instruction onto the address bus.

IRL

IRA

**Table S-4** Instruction register module.

```

/* Instruction Register Module */

module ir(CLK, IR_z, DB_z, IRL, IRA);

    input wire CLK;
    input wire IRL;           // IR load enable
    input wire IRA;          // IR output on address bus enable
    input wire [7:0] DB_z;   // data bus
    output wire [7:0] IR_z;  // IR_z[4]..IR_z[0] connected to address bus
                           // IR_z[7]..IR_z[5] supply opcode to IDMS

    reg [7:0] IR;
    wire [7:0] next_IR;

    assign IR_z[4:0] = IRA ? IR[4:0] : 5'bZZZZZ;
    assign IR_z[7:5] = IR[7:5];

    always @ (posedge CLK) begin
        IR <= next_IR;
    End

    // (IRL) ? load : retain state
    assign next_IR = (IRL) ? DB_z : IR;

endmodule

```

Several items in the IR module, shown in Table S-4, deserve explanation. First, when IRL is negated, note that the IR simply retains its current state. Second, note that, unlike the PC, there is no need to asynchronously reset the IR when the START pushbutton is pressed, since its (random) initial value is of no consequence. Finally, note that IRA only controls the three-state outputs associated with the lower 5-bits of the IR, and that the upper 3-bits (i.e., the opcode bits) are simply transmitted directly to the IDMS (i.e., they do not drive a bus). Recall

that the IDMS uses the opcode bits to determine which system control signals are asserted on the next cycle, when the instruction is executed.

### S.7.4 Arithmetic Logic Unit

As mentioned earlier, the arithmetic logic unit (ALU) is so-named because it performs the arithmetic (add, subtract, etc.) and logical (“Boolean”) operations defined by the instruction set. A “real” ALU performs a wide range of arithmetic and logical functions on operands stored in either registers or in memory. Fortunately, our ALU is relatively simple: it performs four different functions on a single register (which we have called the accumulator, or “A” register) and sets four condition code bits (or flags) based on the result obtained. As such, only four control signals are needed: an overall enable, which we will call ALE; two “function select” lines, which we will call ALX and ALY; and a three-state output enable for “gating” the value in the “A” register onto the data bus, which we will call AOE. The data bus interface must be *bi-directional*, in order to input data supplied by memory on LDA, ADD, SUB, and AND operations; and to output data to memory for STA operations. The condition code bits (CF, NF, VF, ZF) are output directly to the IDMS (we will see how these flags can be used to implement conditional transfer of control instructions later).

*arithmetic and  
logical operations  
ALU*

*ALE  
ALX  
ALY*

*condition  
code bits*

A multiplexer block diagram of the ALU is depicted in Figure S-21, and the corresponding Verilog module is shown in Tables S-5 and S-6. The block diagram provides a visualization for how the equations are derived. The control signal ALE causes its respective 2:1 multiplexer to select either the current value of the AQ bit (when ALE=0) or the multiplexer above it (when ALE=1) as the source for the accumulator next state (ALU bits). The control signal ALX causes its respective 2:1 multiplexer to select either the full adder output (when ALX=0) or the multiplexer above it (when ALX=1). If ALX=0, ALY is used to select whether the ALU is in either ADD (when ALY=0) or SUB (when ALY=1) mode. Note that, given that the LSB Cin is connected to ALY, the radix complement of the subtrahend is formed by taking its diminished radix (1’s) complement (using the XOR gates) and adding one to it. If ALX=1, the 2:1 multiplexed controlled by ALY selects either LDA (when ALY=0) or AND (when ALY=1).

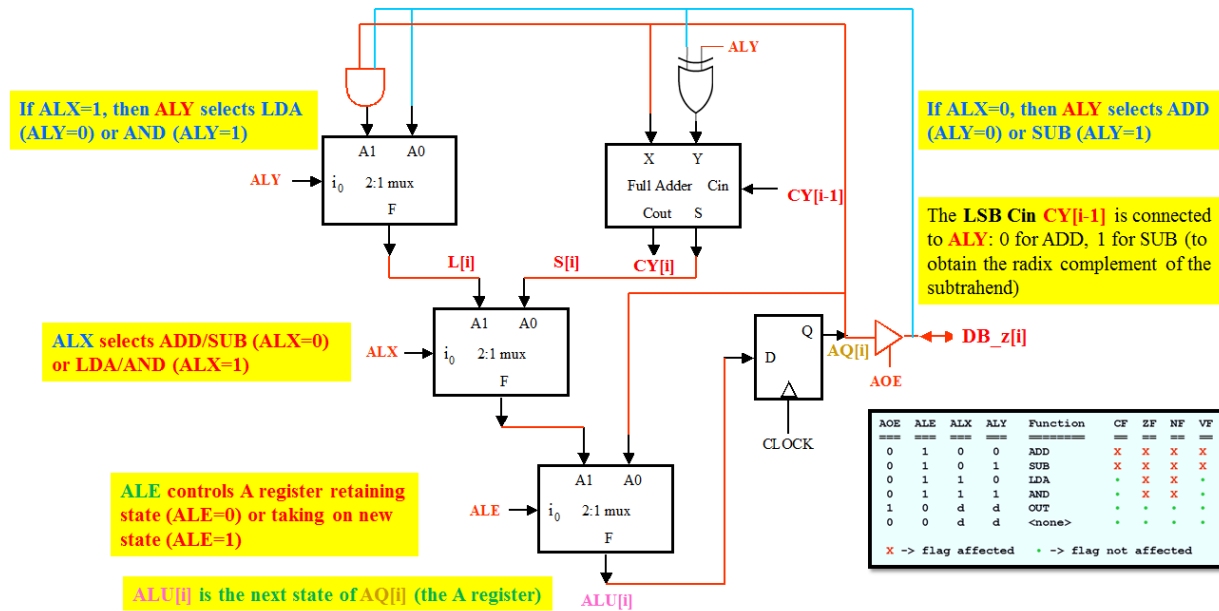


Figure S-21 ALU multiplexer block diagram bit.

Table S-5 Header section of ALU module.

```

/* ALU Module */
module alu(CLK, ALE, AOE, ALX, ALY, DB_z, CF, VF, NF, ZF);

/* 8-bit, 4-function ALU with bi-directional data bus
Accumulator register is AQ, tri-state data bus is DB_z

ADD: (AQ[7:0]) <- (AQ[7:0]) + DB_z[7:0]
SUB: (AQ[7:0]) <- (AQ[7:0]) - DB_z[7:0]
LDA: (AQ[7:0]) <- DB_z[7:0]
AND: (AQ[7:0]) <- (AQ[7:0]) & DB_z[7:0]
OUT: Value in AQ[7:0] output on data bus DB_z[7:0]

AOE  ALE  ALX  ALY  Function    CF  ZF  NF  VF
===  ===  ===  ===  =====  ==  ==  ==  ==
0    1    0    0    ADD          X  X  X  X
0    1    0    1    SUB          X  X  X  X
0    1    1    0    LDA          .  X  X  .
0    1    1    1    AND          .  X  X  .
1    0    d    d    OUT          .  .  .  .
0    0    d    d    <none>       .  .  .  .

X -> flag affected    . -> flag not affected

Note: If ALE = 0, the state of all register bits should be retained */

```



**Table S-6** Continuation of ALU module.

```

input wire CLK;
// ALU control lines
input wire ALE;           // overall ALU enable
input wire AOE;           // data bus tri-state output enable
input wire ALX, ALY;      // function select
inout wire [7:0] DB_z;    // bidirectional 8-bit tri-state data bus
output reg CF, VF, NF, ZF; // condition code register bits (flags)
                           // Carry, Overflow, Negative, Zero

// Carry equations
wire [7:0] CY;
// Combinational ALU outputs
wire [7:0] ALU;
wire [7:0] S;             // Adder/subtractor sum
wire [7:0] L;             // LDA/AND multiplexer output
reg [7:0] AQ;             // A register flip-flops
// Next state variables
reg next_CF, next_VF, next_NF, next_ZF;
reg [7:0] next_AQ;

// Declaration of intermediate equations
// Least significant bit carry in (0 for ADD, 1 for SUB => ALY)
assign CIN = ALY;
// Intermediate equations for adder/subtractor SUM (S) selected when ALX = 0
assign S = AQ ^ (DB_z ^ ALY) ^ {CY[6:0],CIN};
// Ripple carry equations (CY[7] is COUT, DB_z is data from data bus)
assign CY = AQ&(ALY ^ DB_z) | AQ&{CY[6:0],CIN} | ALY&DB_z&{CY[6:0],CIN};
// Intermediate equations for LOAD and AND, selected when ALX = 1
//      (ALY)?   AND   : LDA   (select LDA or AND based on ALY)
assign L = ALY ? AQ & DB_z : DB_z ;
// Combinational ALU outputs
//      (ALX)? L : S   (select LDA/AND or ADD/SUB based on ALX)
assign ALU = ALX ? L : S;
// Register bit and data bus control equations
always @(posedge CLK) begin
    AQ <= next_AQ;
end
always @ (AQ, ALE, ALU) begin
    next_AQ = ALE ? ALU : AQ;
end
assign DB_z = AOE ? AQ : 8'bZZZZZZZZ;
// Condition code register state equations
always @ (posedge CLK) begin
    CF <= next_CF;
    ZF <= next_ZF;
    NF <= next_NF;
    VF <= next_VF;
end
always @ (CF, NF, ZF, VF, ALE, ALX, ALY, CY) begin
    next_CF = ALE ? (ALX ? CF : (CY[7]))           : CF;
    next_ZF = ALE ? (ALU == 8'b00000000)          : ZF;
    next_NF = ALE ? ALU[7]                        : NF;
    next_VF = ALE ? (ALX ? VF : (CY[7] ^ CY[6]))  : VF;
end
endmodule

```

Last, but not least, are the equations that govern the four condition code bits. All of these flags retain their current state if ALE is negated. The carry flag (CF) and overflow flag (VF) are only affected by the ADD and SUB instructions. For ADD and SUB, the CF bit is set to the carry out of the most significant position (here, CY[7]). The VF bit is simply the XOR of the *carry in* to the sign bit (CY[6]) with the *carry out* of the sign bit (CY[7]).

The negative flag (NF) and zero flag (ZF) are affected by all four functions implemented by our ALU. The NF bit is simply the sign bit (ALU7) of the result generated by the ALU, while the ZF bit is set to “1” if all the ALU result bits are zero.

Before moving on to the final block of our simple computer design, there is an important practical point worth noting. All of the functional blocks designed thus far – the memory, PC, IR, and ALU – can be independently implemented (or simulated) and tested (as well as debugged) before they are all “assembled together” into a completed computer. Independent testing and debugging of each functional block, in fact, is an important aspect of the “top-down, bottom-up” strategy we have espoused in this chapter.

*independent  
testing and  
debugging*

### S.7.5 Instruction Decoder and Micro-sequencer

As described previously, there are two basic steps involved with “processing” each instruction, the combination of which is referred to as a micro-sequence. During a fetch cycle, the instruction pointed to by the PC is read from memory and loaded into the IR; the PC is incremented by one as the instruction is loaded. During the ensuing execute cycle, the instruction staged in the IR is “peeled” apart into an *opcode* field and an *operand address* field; the opcode field indicates the operation to be performed using data obtained from (or destined for) the memory location specified by the address field. The functional block that orchestrates the sequencing of these activities is called the *instruction decoder and micro-sequencer* (IDMS).

Since, in this initial version of our simple computer, there are only two different kinds of cycles (*fetch* and *execute*), a single flip-flop can be used as a *state counter* (SQ). In reality, this state counter is simply a single-bit binary counter (i.e., it simply *toggles* between “0” and “1”). Note that the state counter must be placed in the “fetch” state when START is pressed; therefore, it makes sense to assign the “reset” state

*state counter (SQ)  
toggles*

of the SQ flip-flop (SQ=0) to the fetch cycle, and the “set” state of the SQ flip-flop (SQ=1) to the execute cycle.

With the structure of the state counter established, the next step is to determine which control signals (of the functional blocks designed previously) need to be asserted when SQ=0 (fetch) and SQ=1 (execute). To accomplish this, we will need to refer back to each of the previous sub-sections (on the design of the individual functional blocks) as well as the instruction tracing worksheets completed previously.

Referring again to Figure S-12, we note that the following signals need to be asserted to complete a fetch cycle. First, to “gate” the value in the PC onto the address bus, the signal POA needs to be asserted by the IDMS. To read the instruction, the memory needs to be selected (MSL asserted) and its data bus output enabled (MOE asserted). To load the instruction read from memory into the IR, the signal IRL needs to be asserted. Finally, to increment the PC as the instruction is loaded, the signal PCC needs to be asserted. A total of five system control signals, therefore, needed to be asserted by the IDMS during a fetch cycle (when SQ=0): POA, MSL, MOE, IRL, and PCC.

The control signals that need to be asserted during an “ALU function” execute cycle (i.e., LDA, ADD, SUB, AND operation) can be inferred from Figure S-13. First, to “gate” the operand address staged in the IR onto the address bus, the signal IRA needs to be asserted by the IDMS. To read the operand, the memory needs to be selected (MSL asserted) and its data bus output enabled (MOE asserted). To perform the operation specified by the instruction opcode (supplied to the IDMS from the upper 3-bits of the IR), ALE needs to be asserted along with the prescribed combination of ALX and ALY (based on the ALU design documented in Table S-5).

The “store A” (STA) instruction execute cycle is similar, but notably different, than an “ALU function” execute cycle. Here, the address supplied to memory (from the IR, upon assertion of IRA) specifies the destination for the data in the “A” register. To complete the write to memory, it needs to be selected (MSL asserted) and write enabled (MWE asserted). To “gate” the data in the “A” register onto the data bus, AOE needs to be asserted. A total of four control signals need to be asserted, then, to execute a “store A” (STA) instruction: IRA, MSL, MWE, and AOE.

A succinct summary of all the system control signal assertions is provided in Table S-7. Note that, for the sake of clarity, signal assertions are denoted using “H” (signals that are either negated or “don’t care” are left blank). By way of contrast, the control signal negations that are effected by execution of the HLT (halt) instruction are denoted using “L”.

**Table S-7** System control table.

Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY
S0	—	H	H		H	H	H					
S1	HLT	L			L		L			L		
S1	LDA	H	H					H		H	H	
S1	ADD	H	H					H		H		
S1	SUB	H	H					H		H		H
S1	AND	H	H					H		H	H	H
S1	STA	H		H				H	H			

The Verilog source file for the simple computer’s IDMS module is shown in Table S-8. Referring first to the declarations, we find decoded opcode definitions (using the instruction mnemonics as pseudonyms for the corresponding opcode bit patterns) and decoded machine state definitions (S[0] for fetch, S[1] for execute). The purpose of defining an intermediate equation for each opcode combination is simply to make the job of writing the system control equations easier. Perhaps if we were more “clever”, we might have used the name “fetch” (instead of S[0]) and “execute” (instead of S[1]) to help make the subsequent equations a bit more clear (albeit more cumbersome to write).

Continuing with the IDMS equations, we discover three basic components: the state counter, the run/stop flip-flop, and the system control equations. Looking first at the state counter, we note that if the machine RUN enable is high (i.e., the machine is “running”), the state counter flip-flop merely “toggles” each time a positive CLOCK edge occurs. If RUN is negated, SQ is reset to “0” (i.e., the “fetch” state). Pressing the START pushbutton also resets SQ to the “fetch” state.

*run/stop  
flip-flop*

**Table S-8** IDMS module.

```

/* Instruction Decoder and Microsequencer */
module idms(CLK,START,OP,MSL,MOE,MWE,PCC,POA,ARS,IRL,IRA,ALE,ALX,ALY,AOE);
  input wire CLK;
  input wire START; // Asynchronous START pushbutton
  input wire [2:0] OP; // opcode bits (input from IR[7:5])
  output wire MSL, MOE, MWE; // Memory control signals
  output wire PCC, POA, ARS; // PC control signals
  output wire IRL, IRA; // IR control signals
  output wire ALE, ALX, ALY, AOE; // ALU control signals
  reg SQ, next_SQ; // State counter
  reg RUN, next_RUN; // RUN/HLT state
  wire LDA, STA, ADD, SUB, AND, HLT; // Opcode names
  wire [1:0] S; // State variables
  wire RUN_ar; // Asynchronous reset for RUN
  assign HLT = ~OP2 & ~OP1 & ~OP0; // HLT opcode = 000
  assign LDA = ~OP2 & ~OP1 & OP0; // LDA opcode = 001
  assign ADD = ~OP2 & OP1 & ~OP0; // ADD opcode = 010
  assign SUB = ~OP2 & OP1 & OP0; // SUB opcode = 011
  assign AND = OP2 & ~OP1 & ~OP0; // AND opcode = 100
  assign STA = OP2 & ~OP1 & OP0; // STA opcode = 101
  // Decoded state definitions
  assign S[0] = ~SQ; // fetch
  assign S[1] = SQ; // execute
  // State counter
  always @ (posedge CLK, posedge START) begin
    if(START == 1'b1) // start in fetch state
      SQ <= 1'b0;
    else // if RUN negated, resets SQ
      SQ <= next_SQ;
    end
  always @ (SQ, RUN) begin
    next_SQ = RUN & ~SQ
  end
  // Run/stop
  assign RUN_ar = S[1] & HLT;
  always @ (posedge CLK, posedge RUN_ar, posedge START) begin
    if(START == 1'b1) // RUN set to 1 when START asserted
      RUN <= 1'b1;
    else if(RUN_ar == 1'b1) // RUN is cleared when HLT is executed
      RUN <= 1'b0;
    end
  // System control equations
  assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
  assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
  assign MWE = S[1] & STA;
  assign ARS = START;
  assign PCC = RUN & S[0];
  assign POA = S[0];
  assign IRL = RUN & S[0];
  assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
  assign AOE = S[1] & STA;
  assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
  assign ALX = S[1] & (LDA | AND);
  assign ALY = S[1] & (SUB | AND);
endmodule

```

The run/stop flip-flop is defined next in the IDMS module code (Table S-8). Here we note that pressing the START pushbutton asynchronously sets the RUN flip-flop, thereby enabling our simple computer to start executing instructions. Once set, the RUN signal remains asserted until asynchronously reset through execution of an HLT instruction.

We see how the RUN signal is used to enable/disable machine activity in the system control equations that follow. Note that if RUN is high, the system control signals are asserted according to the table in Table S-8, as described previously. For example, MSL is asserted if a *fetch cycle* is being performed (S[0] high); *or*, an *execute cycle* is being performed (S[1] high) of an LDA instruction, an STA instruction, an ADD instruction, a SUB instruction, or an AND instruction. If RUN is low, however, all of the pertinent system control signals are negated. Note that it is only necessary to negate the system control signals responsible for causing the various functional blocks to *change state* (i.e., it is *not necessary* to negate function select signals such as ALX and ALY, nor is it necessary to negate three-state output enables).

This completes the “bottom-up” phase of the design process for the initial version of our simple computer. All of the Verilog code described in this section could be implemented using a single, modest-size PLD. The addition of a conventional memory chip would yield a working computer. Before augmenting the instruction set with some useful extensions, though, let’s take a closer look at system timing.

## S.8 System Timing Analysis

When we designed the program counter in Section S.7.2, there was the appearance of “cheating” – specifically, of using the current value in the PC to access an instruction in memory while, at apparently the same time, telling the PC to increment. This is an issue that deserves further scrutiny.

To gain a better understanding of the timing relationship among different activities within our computer, we need to understand two basic hardware-imposed constraints. The first is that only one device (functional block) can drive a bus on a given bus cycle, i.e., “bus fighting” must be avoided. The second is that data can only “pass through” one edge-triggered flip-flop per cycle. Thus, it is not possible to load a value into a register and expect to “use it” (have the value available on the register’s outputs) on the same cycle.

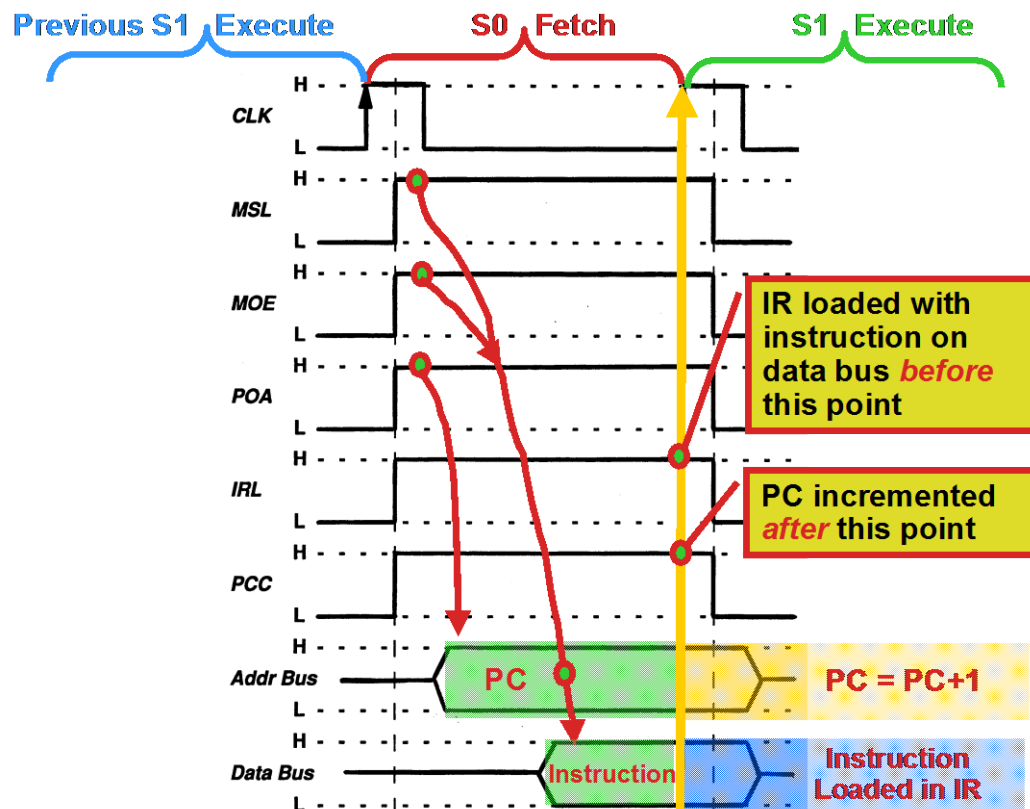
*bus fighting*

Given these constraints, we are now prepared to examine in detail the sequence of activities that occur during a fetch cycle. A “qualitative” timing diagram is provided in Figure S-21 for this purpose (by *qualitative* we mean that we’re not interested in the exact number of nanoseconds between one signal assertion and another, just the fact that there is a *delay*). Depicted in this diagram is the sequencing that occurs as the machine finishes an execute cycle, performs a fetch of the next instruction, and subsequently proceeds to execute the instruction just fetched. Our focus here is on the events that constitute a fetch cycle.

*qualitative timing diagram*

The first thing to note is that, since the functional blocks of the machine were designed using positive-edge-triggered flip-flops, the *clock edges* “drive” the machine from state-to-state. Thus, a “fetch cycle” is the time between the clock edge that drives the machine from the previous execute cycle to the current fetch cycle, and the subsequent clock edge that transitions the machine from the fetch cycle to an execute cycle. Shortly after the first clock edge in Figure S-21, then, the control signals MSL, MOE, POA, IRL, and PCC are asserted (the delay relative to the clock edge in generating these signals is due to the propagation delay of the state counter plus the delay associated with the system control equations – see Table S-8).

*clock edges*



**Figure S-21** Fetch cycle event timing relationship.

The assertion of POA causes the three-state buffers of the PC to turn on and drive its value onto the address bus. The value on the address bus, in conjunction with the MSL and MOE signal assertions, causes the memory to drive the addressed instruction onto the data bus (note that, in most practical systems, this constitutes a substantial part of the cycle time). Provided the instruction is on the data bus at least  $t_{\text{SETUP}}$  (of the D flip-flop) prior to the next clock edge, it is successfully loaded into the IR (because the IRL signal is asserted) when that edge occurs.

While this may seem to be “enough” activity already, we realize that a related “housekeeping” activity can be accomplished on this cycle as well: incrementing the value in the PC, so it points to the next instruction (in preparation for the next fetch). Again, based on the use of edge-triggered flip-flops in our design, we note that the value on the data bus *just prior* to the clock edge that loads the IR determines the next state of the IR. It follows, then, that we can use that *same clock edge* to drive the PC to its next state – this is why PCC is also asserted during a fetch cycle. Note that the PC state change will occur *after* the clock edge, i.e., after the instruction has been safely loaded into the IR. This allows us to effectively *overlap* the load of the IR with the increment of the PC on the same cycle. We will make use of this same principle when we add some extensions to our machine later in this chapter.

*overlap*

One might ask at this point, “Could we have delayed the increment of the PC until the execute cycle?” In the initial version of our simple computer, it would clearly be possible: here, the “new value” in the PC would be available shortly after the commencement of the fetch cycle, thus enabling the correct instruction to be loaded into the IR (the only consequence might be a small amount of additional propagation delay for the “new” value to become stable). When we add subroutine linkage instructions to our computer, however, we will find it useful to have the “new” value of the PC available during the first execute cycle (to serve as the “return address” for a “subroutine call” instruction). In anticipation of this extension, we will include the increment of the PC as an integral part of the fetch cycle.



## S.9 Simple Computer Extensions

When we originally designed our instruction set, we purposefully left two opcode bit patterns “uncommitted”. The reason we did this was to provide room for expansion. We will, then, add a “pair” of instructions at a time to our “base” instruction set. The “pairs” we will add include input/output (IN/OUT) instructions, transfer of control instructions (JMP/JZF), stack manipulation instructions (PSH/POP), and subroutine linkage instructions (JSR/RTS).

### S.9.1 Input/Output Instructions

When we first drew the “big picture” of our simple computer (see Figure S-4), we included a switch “input port” and an LED “output port”. As evident from the initial version of our instruction set, we included no provision for using these. It makes sense, then, to add instructions for providing our machine with the “modern convenience” of data input and output (“I/O”).

*input port*  
*output port*

First, we need to establish the *destination* that will be used for data input (or *read*) from the “outside world”, as well as the *source* for data that will be output (or *written*). Given that our machine has but one register that participates in data transactions – namely, the “A” register – it is the most likely candidate to serve as the destination/source of data that is input/output, respectively. Thus, our new “IN” instruction will function in a manner similar to an LDA instruction, *except* the source of data will be the “outside world” and the address field will be used as a pointer to an “input device” (instead of to memory). Similarly, our new “OUT” instruction will function in a manner similar to an STA instruction, *except* the destination of data will be the “outside world” and the address field will be used as a pointer to an “output device”. A name commonly used for this input/output strategy is *accumulator-mapped I/O*.

Second, we need to establish how data will be communicated to/from the ubiquitous “outside world”. Basically, a “gateway” is needed between the system data bus and the external input and output devices, along with some new system control signals that enable a “read” (IOR) or a “write” (IOW) via this gateway. Also, a means of decoding the I/O addresses (typically called *port* or *device numbers*) into individual “device selects” (or *enables*) is needed. A diagram illustrating the placement of the “I/O block” is provided in Figure S-22; a Verilog module is given in Table S-9.

*IOR*

*IOW*

*port numbers*  
*device numbers*  
*I/O block*

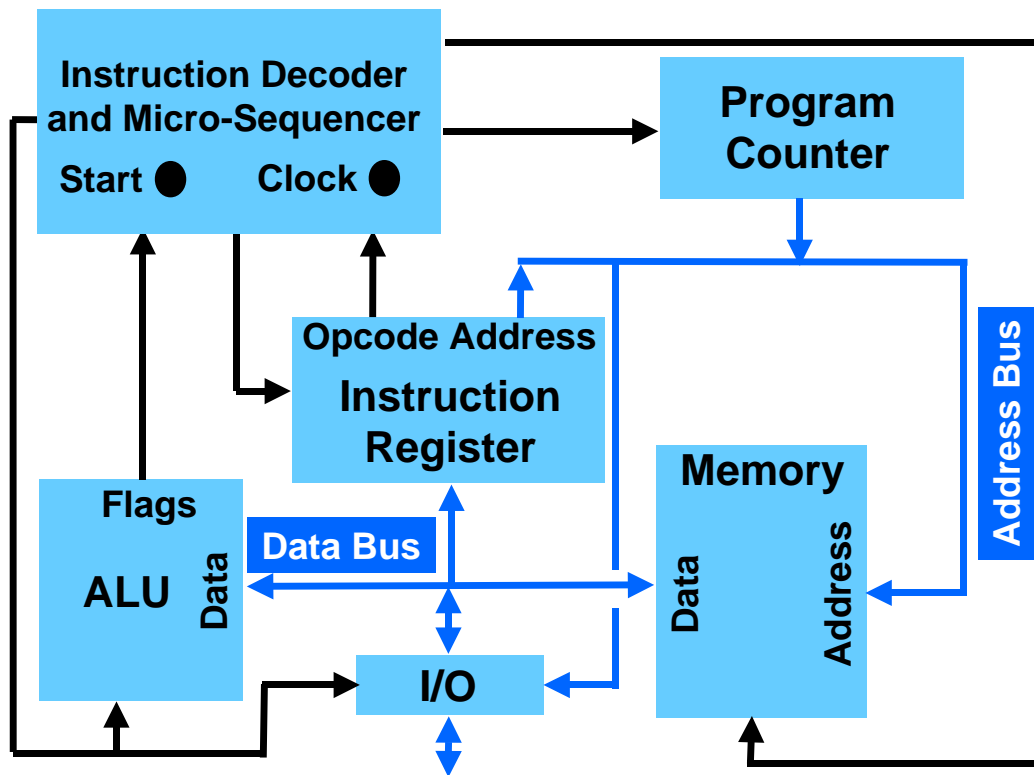


Figure S-22 Placement of I/O block in Simple Computer.

Table S-9 Basic I/O module.

```

/* Input/Output Port 00000 */
module io(ADRBUS_z, IN, OUT, IOR, IOW, DB_z);
  input wire [4:0] ADRBUS_z; // address bus
  input wire [7:0] IN; // input port
  input wire IOR; // input port read
  input wire IOW; // input port write
  output wire [7:0] OUT; // output port
  inout wire [7:0] DB_z; // bidirectional data bus
  wire PS;

  // Port select equation for port address 00000
  assign PS = (ADRBUS_z == 5'b00000);
  assign DB_z = IOR & PS ? IN : 8'bZZZZZZZZ;
  assign OUT = IOW & PS ? DB_z : 8'bZZZZZZZZ;

endmodule

```

Referring to the Verilog module, we see that it contains a specific port address decoding equation, here for port address  $00000_2$ . When the pattern on the address bus matches this value, an I/O transaction via this port address is enabled. If an IN instruction is being executed, assertion of the IOR signal (by the IDMS) causes the value on the “IN pins” ( $IN[7:0]$ ) to be gated onto the system data bus, allowing it to be loaded into the “A” register. If an OUT instruction is being executed, assertion of the IOW signal causes the value on the data bus (supplied by the “A” register) to be gated to the “OUT pins” ( $OUT[7:0]$ ).

There is a limitation, however, inherent in the I/O port design shown in Table S-9: the value output (when an OUT instruction is executed) is only “active” for a very short time (specifically, the amount of time the IOW signal is asserted by the IDMS). For devices such as light emitting diodes (LEDs), the brief assertion of IOW will not provide a satisfactory display. A better solution is to *latch* the value sent to the output port, and retain it until execution of a subsequent OUT instruction changes the value. An I/O module that provides a latched output port is provided in Table S-10. Here, assertion of IOW in conjunction with the proper port address opens a transparent latch, which then assumes the new value sent on the data bus. The latch closes (retains its value) when IOW is negated. Note that an `if` construct without an `else` creates an *inferred latch*, a “feature” that some Verilog purists may find the use of objectionable.

*latched  
output port*

The augmented system control table for our simple computer plus I/O is given in Table S-11. Note that there are two “new” equations (for IOR and IOW), along with four equations that need to be updated (for IRA, AOE, ALE, and ALX). The updated system control equations are given in Table S-12.

**Table S-10** Latched I/O port.

```

/* Input/Output Port 00000 - with Output Latch */

module io(ADRBUS_z, IN, OUT, IOR, IOW, DB_z);
  input wire [4:0] ADRBUS_z; // address bus
  input wire [7:0] IN;      // input port
  input wire IOR;          // input port read
  input wire IOW;          // input port write
  output wire [7:0] OUT;   // output port
  inout wire [7:0] DB_z;   // bidirectional data bus
  wire PS;

  // Port select equation for port address 00000
  assign PS = (ADRBUS_z == 5'b00000);
  assign DB_z = IOR & PS ? IN : 8'bZZZZZZZZ;

  // Transparent latch for output port
  always @ (IOW, PS, DB_z) begin
    if((IOW & PS) == 1'b1)
      OUT = DB_z;
  end

endmodule

```

**Table S-11** System control table modified for I/O.

Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY	IOR	IOW
S0	—	H	H		H	H	H							
S1	HLT	L			L		L			L				
S1	LDA	H	H					H		H	H			
S1	ADD	H	H					H		H				
S1	SUB	H	H					H		H		H		
S1	AND	H	H					H		H	H	H		
S1	STA	H		H				H	H					
S1	IN							H		H	H		H	
S1	OUT							H	H					H

**Table S-12** System control equations modified for I/O.

```

// System control equations modified to support IN/OUT

assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
assign MWE = S[1] & STA;
assign ARS = START;
assign PCC = RUN & S[0];
assign POA = S[0];
assign IRL = RUN & S[0];
assign IRA = S[1] & (LDA | STA | ADD | SUB | AND | IN | OUT);
assign AOE = S[1] & (STA | OUT);
assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND | IN);
assign ALX = S[1] & (LDA | AND);
assign ALY = S[1] & (SUB | AND);

assign IOR = S[1] & IN;
assign IOW = S[1] & OUT;

endmodule

```

## S.9.2 Transfer-of-Control Instructions

Any program worth the silicon it runs on typically does more than execute “straight line” code. Instead, execution transfers to different parts of the program based on various conditions encountered. Generically, we refer to the instructions that allow program execution to “jump around” as *transfer-of-control* instructions.

*straight line code*

*transfer-of-control instructions*

There are two basic types of transfer-of-control instructions. If the address field of the instruction contains the (absolute) address in memory at which execution should continue, it is most often referred to as a “jump” instruction. If the address field instead represents the (signed) “distance” the next instruction is from the transfer-of-control instruction, it is referred to as a “branch”. (There is not universal agreement on this nomenclature, however – see sidebar.) Jumps (or branches) that “always happen” are called *unconditional*; those that happen only if a certain combination of condition codes exists are called *conditional*.

*jump instruction*

*branch instruction*

*unconditional conditional*

The addition of transfer-of-control instructions to our simple computer will require modifications to the PC (as well as to the IDMS). Specifically, we will need to provide a mechanism for loading a new value into the PC to implement “jump-style” instructions, or for adding a

signed offset to the value in the PC to implement “branch-style” instructions. Here we will focus on the modifications necessary to implement jump-style instructions. A Verilog module for the modified PC is provided in Table S-13. Note that it is the same as the “original” PC (see Table S-3), except that a “load from address bus” function (and associated control signal, PLA) has been added. Recall that the “new value” with which the PC is to be loaded is staged in the IR, and can therefore be conveniently “transported” to the PC via the address bus.

PLA

### A Branch by Any Other Name

Regrettably, there is no “universal agreement” among manufacturers of microcontrollers concerning the names used for the basic transfer-of-control instruction types. Since this is primarily a text dealing with Motorola products, we will use the names they commonly use: “jump” for absolute transfer, and “branch” for relative transfer. Be advised, though, that another “major manufacturer” (Intel) uses *just the opposite* designation: “branch” for absolute transfer, and “jump” for relative transfer. Although the author cut his “digital teeth” on Intel processors, he prefers the Motorola adopted names.

**Table S-13** PC modifications to support transfer-of-control instructions.

```

/* Modified Program Counter with Load Capability */
module pc(CLK, PCC, POA, ADRBUS_z, PLA, RST);
  input wire CLK;
  input wire PCC; // PC count enable
  input wire POA; // PC output on address bus tri-state enable
  input wire PLA; // PC load from address bus enable
  input wire RST; // Asynchronous reset (connected to START)
  inout wire [4:0] ADRBUS_z; // address bus

  // NOTE: Assume PCC and PLA are mutually exclusive
  reg [4:0] PC, next_PC;
  assign ADRBUS_z = POA ? PC : 5'bZZZZZ;
  always @ (posedge CLK, posedge RST) begin
    if (RST == 1'b1)
      PC <= 5'b00000;
    else
      PC <= next_PC;
    end
  always @ (PCC, PC) begin
    if (PLA == 1'b1) // load
      next_PC = ADRBUS_z;
    else if (PCC == 1'b1) // count up by 1
      next_PC = PC + 1;
    else // retain state
      next_PC = PC;
    end
  end
endmodule

```

The system control table, modified to include an “unconditional jump” instruction (JMP) along with a “jump if zero flag set” (JZF) instruction, is shown in Table S-14. As its name implies, the JZF instruction causes a transfer-of-control to the address following the opcode if the zero flag (ZF) is set, i.e., the result of the most recent ALU operation has generated a result of zero in the “A” register. (As it turns out, this is a fairly “popular” condition to check in practical applications.) If the condition specified by a “conditional jump” instruction (like JZF) is *not* met, however, *nothing happens* (often called a *no operation*, or “NOP”) – execution merely continues with the instruction that follows. In order to effect the load of the jump address, the IDMS needs to know the state of the various condition code bits generated by the ALU. The equations for IRA and PLA, then, will be a function of ZF for the new instructions added to the machine in Table S-17.

**JMP**  
**JZF**

*no operation*  
**NOP**

**Table S-14** System control table modified for transfer-of-control instructions.

Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY	PLA
S0	—	H	H		H	H	H						
S1	HLT	L			L		L			L			
S1	LDA	H	H					H		H	H		
S1	ADD	H	H					H		H			
S1	SUB	H	H					H		H		H	
S1	AND	H	H					H		H	H	H	
S1	STA	H		H				H	H				
S1	JMP							H					H
S1	JZF							ZF					ZF

**Table S-15** IDMS modifications to support transfer-of-control.

```
// System control equations modified to support JMP/JZF

assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
assign MWE = S[1] & STA;
assign ARS = START;
assign PCC = RUN & S[0];
assign POA = S[0];
assign IRL = RUN & S[0];
assign IRA = S[1] & (LDA | STA | ADD | SUB | AND | JMP | JZF&ZF);
assign AOE = S[1] & STA;
assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
assign ALX = S[1] & (LDA | AND);
assign ALY = S[1] & (SUB | AND);
assign PLA = S[1] & (JMP | JZF & ZF);
endmodule
```

One could imagine, at this point, a number of other conditions that would be useful for determining whether or not a jump or branch should be “taken”. In addition to a separate “jump on condition” instruction dedicated to each flag (CF, NF, VF, ZF), there are various *Boolean combinations* of these flags that are of interest as well (e.g., testing for “greater than” or “less than or equal to”). All of these variations will be explored when we tackle the instruction set of a “real” microcontroller in the next chapter.

*Boolean combinations of flags*

### S.9.3 Multiple Execute Cycle Instructions

To this point, all of the instructions we originally defined or added to our simple computer required a single fetch cycle followed by a single execute cycle. As the functions performed by an individual instruction become more complex, however, additional execute cycles become necessary. On the surface, this would appear to be a relatively straightforward extension, accomplished by simply adding extra bits to the state counter in the IDMS, along with a binary decoder to decode the various states. Adding one additional bit to our original state counter would provide us with four possible states: a fetch state (S[0]), followed by three execute states (S[1], S[2], S[3]).

S[1]  
S[2]  
S[3]

The “complication” that arises is that, despite this addition, we want our original “single execute state” instructions to still execute in a single state. Further, we want any new instructions that require two execute states to consume only two execute states, and new instructions that require all three execute states to consume exactly three execute states. More succinctly, we want our state counter to be able to accommodate *variable-length execution cycles* (here, from 1 to 3).

*variable-length execution cycles*

One way this can be accomplished is by adding a *synchronous reset* capability to our (now 2-bit) state counter. For this purpose, we will add a new signal (RST) to our system control table that, when asserted, causes the state counter to *reset to zero* when the next clock edge occurs. In the system control table, this signal will be asserted on the *final execute cycle* of each instruction. For single execute cycle instructions (such as LDA, STA, ADD, AND, SUB), the RST signal will be asserted during S[1] (the first execute cycle), ensuring that the next cycle will be a “fetch”. For instructions requiring two execute cycles, the RST signal will be asserted during S[2] (the second execute cycle). Finally, for three-execute-cycle instructions, the RST signal will be asserted during S[3] (note that, if RST is *not* asserted at this point, the state counter will “wrap around” to zero automatically, thus ensuring that the next cycle is a “fetch” regardless).

*synchronous reset*



**Table S-16** IDMS modifications for multi-execute-cycle instructions.

```

/* Instruction Decoder and Microsequencer with Multi-Execution States */
module idmsr(CLK,START,OP,MSL,MOE,MWE,PCC,POA,ARS,IRL,IRA,ALE,ALX,ALY,AOE);
  input wire CLK;
  input wire START;
  input wire [2:0] OP;
  output wire MSL, MOE, MWE;
  output wire PCC, POA, ARS;
  output wire IRL, IRA;
  output wire ALE, ALX, ALY, AOE;
  reg SQA, SQB;
  reg RUN;
  wire RST;
  wire LDA, STA, ADD, SUB, AND, HLT;
  wire [3:0] S;
  reg next_SQA, next_SQB;
  wire RUN_ar;

  // Decoded opcode definitions
  assign HLT = ~OP2 & ~OP1 & ~OP0;
  assign LDA = ~OP2 & ~OP1 & OP0;
  assign ADD = ~OP2 & OP1 & ~OP0;
  assign SUB = ~OP2 & OP1 & OP0;
  assign AND = OP2 & ~OP1 & ~OP0;
  assign STA = OP2 & ~OP1 & OP0;

  // Decoded state definitions
  assign S[0] = ~SQB & ~SQA;
  assign S[1] = ~SQB & SQA;
  assign S[2] = SQB & ~SQA;
  assign S[3] = SQB & SQA;

  // State counter
  always @ (posedge CLK, posedge START) begin
    if(START == 1'b1) begin
      SQA <= 1'b0;
      SQB <= 1'b0;
    end else begin
      SQA <= next_SQA;
      SQB <= next_SQB;
    end
  end

  always @ (RST, RUN, SQA, SQB) begin
    next_SQA = ~RST & RUN & ~SQA;
    next_SQB = ~RST & RUN & (SQA ^ SQB);
  end
end

```

**Table S-17** IDMS modifications for multi-execute-cycle instructions, continued.

```

assign RUN_ar = S[1] & HLT;

// Run/stop
always @ (posedge CLK, posedge RUN_ar, posedge START) begin
    if(START == 1'b1)           // start with RUN set to 1
        RUN <= 1'b1;
    else if(RUN_ar == 1'b1)     // RUN is cleared when HLT is executed
        RUN <= 1'b0;
end

// System control equations
assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND));
assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
assign MWE = S[1] & STA;
assign ARS = START;
assign PCC = RUN & S[0];
assign POA = S[0];
assign IRL = RUN & S[0];
assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
assign AOE = S[1] & STA;
assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
assign ALX = S[1] & (LDA | AND);
assign ALY = S[1] & (SUB | AND);
assign RST = S[1] & (LDA | STA | ADD | SUB | AND);

endmodule

```

The state counter modifications necessary to accommodate multiple execute cycles are shown in Tables S-16 and S-17. Following conventional notation, bit “A” of the modified state counter is the least significant bit, and bit “B” is the most significant bit. Note that if RUN is negated, or RST is asserted, the state counter is reset to “00”. Pressing the START pushbutton also resets the state counter to zero.

In the sections that follow, we will see examples of instructions that require two or three execute states. The system control tables for these “new” instruction sets will therefore include the RST signal.

### S.9.4 Stack Manipulation Instructions

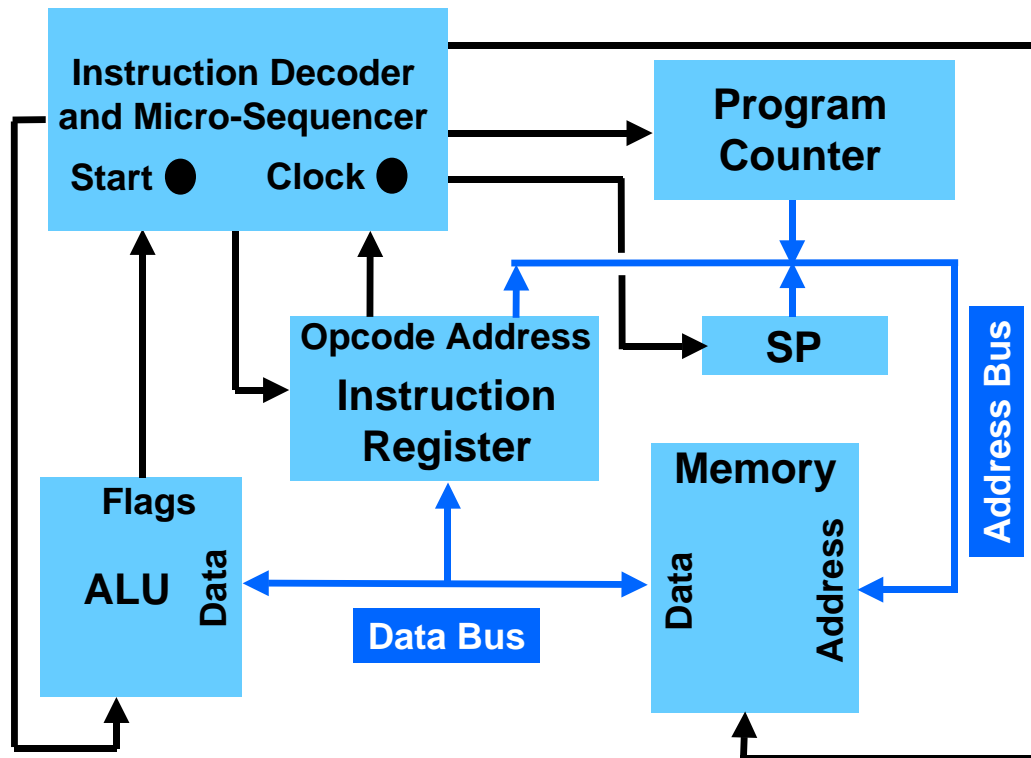
An important “modern convenience” that most “real” computers enjoy is a stack mechanism. Stacks – also referred to as *last-in, first-out* (LIFO) data structures – facilitate a number of capabilities, including

*last-in, first-out*  
**LIFO**

expression evaluation, subroutine linkage, and parameter passing. While there are many variations on stack implementation, the most common strategy is to place the stack contents in the uppermost portion of (read/write) memory, and add a new register to the machine that serves as a pointer to the top item on the stack. Not surprisingly, this register is called the *stack pointer* (SP). An augmented system block diagram illustrating the placement of the SP register in our simple computer is given in Figure S-23.

*expression evaluation*  
*subroutine linkage*  
*parameter passing*

*stack pointer*  
*SP*



**Figure S-23** Block diagram of simple computer with stack.

Since program “growth” (or *execution direction*) is toward *increasing* addresses (starting in “low” memory), it makes sense that *stack growth* should be toward *decreasing* addresses (starting in “high” memory). The stack grows as items are “pushed” onto it, which means the SP register must decrement as it grows; conversely, as items are “popped” off the stack and its size diminishes, the SP register must increment.

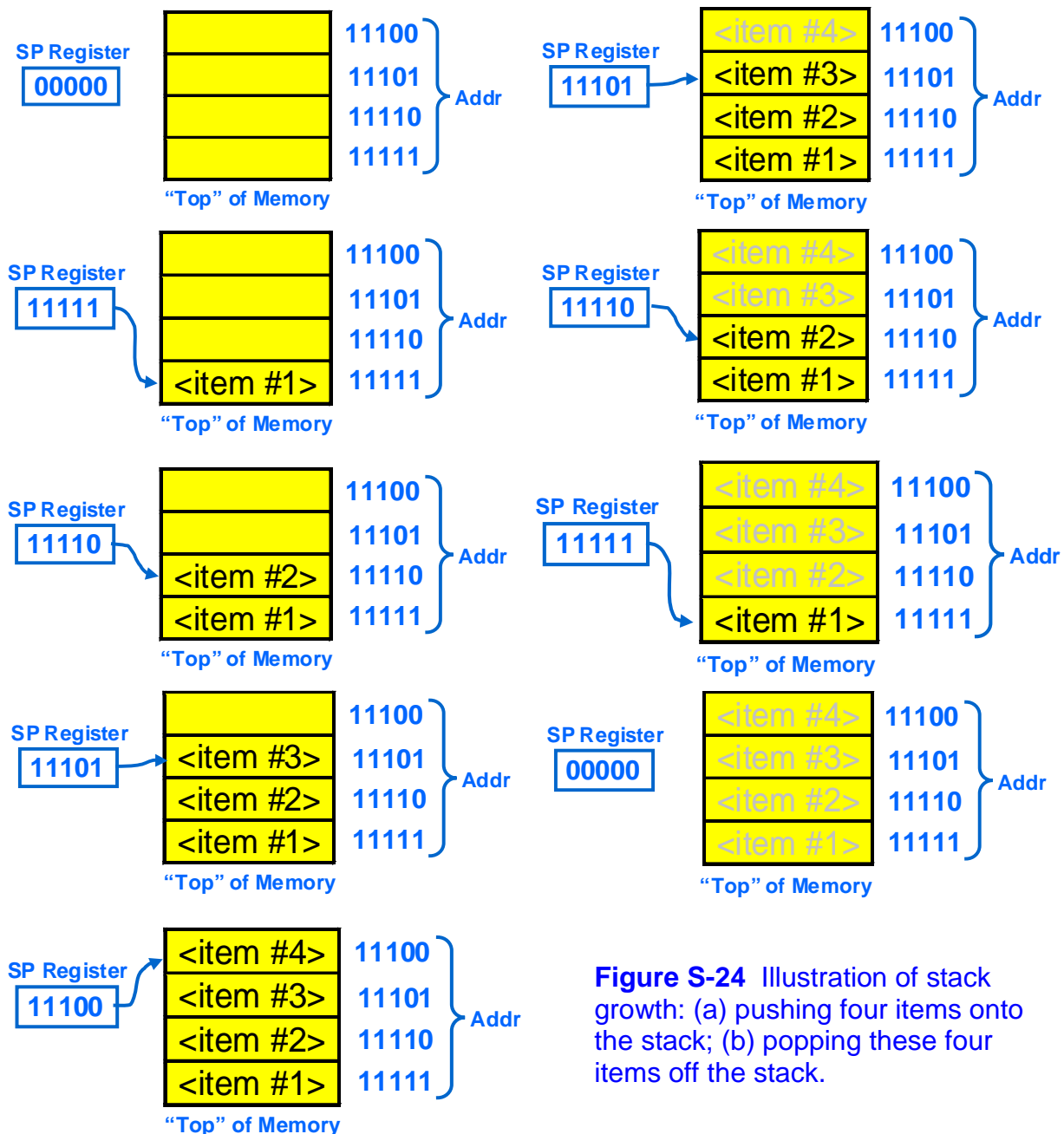
*execution direction*

*stack growth*

At this point, we realize there are two possible conventions that can be used as a “stack pointer paradigm” – we can choose to have the SP register point to the *top stack item*, or we can choose to have it point to

*stack convention*  
*top stack item*  
*next available location*

the *next available location*. The most commonly used convention (and the one we will adopt here) is to have the SP register point to the top stack item. Based on this choice, we realize that the initial value of the SP register needs to be *one greater* than the address in which the first stack item is placed. Because the SP register points to the top stack item, it must be decremented in order to allocate space for a new item during a “push” operation. If the stack starts in the *uppermost location* of memory (for our simple computer, location  $11111_2$ ), the SP register should be initialized to  $00000_2$  (i.e., one greater than  $11111_2$ , modulo  $2^5$ ). Stack growth and retraction based on this “conventional convention” is illustrated in Figure S-24. Note that items popped off the stack are merely *de-allocated* from the stack area, *not erased*.



**Figure S-24** Illustration of stack growth: (a) pushing four items onto the stack; (b) popping these four items off the stack.

Based on an understanding of how the stack mechanism works, we can now consider the design of the SP register module, documented in Table S-18. The first thing we note is that the SP register is simply an “up/down” binary counter, with three-state output buffers and an asynchronous reset. The IDMS, then, needs to supply the SP register with four control signals: an asynchronous reset (ARS), an increment enable (SPI), a decrement enable (SPD), and a three-state buffer enable (SPA) that gates the value in the SP register onto the address bus.

**ARS**  
**SPI**  
**SPD**  
**SPA**

**Table S-18** Stack pointer module.

```

/* Stack Pointer */

module sp(CLK, SPI, SPD, SPA, ARS, ADRBUS_z);
  // NOTE: Assume SPI and SPD are mutually exclusive
  input wire CLK;
  input wire SPI, SPD;           // SP increment, decrement
  input wire SPA;               // SP output on address but tri-state enable
  input wire ARS;               // asynchronous reset (connected to START)
  output wire [4:0] ADRBUS_z; // address bus
  reg [4:0] SP, next_SP;

  assign ADRBUS_z = SPA ? SP : 5'bzzzzz;

  always @ (posedge CLK, posedge ARS) begin
    if (ARS == 1'b1)
      SP <= 5'b00000;
    else
      SP <= next_SP;
  end

  always @ (SPI, SPD, SP) begin
    if (SPI == 1'b1)           // increment
      next_SP = SP + 1;
    else if (SPD == 1'b1)     // decrement
      next_SP = SP - 1;
    else                       // retain state
      next_SP = SP;
  end
endmodule

```

We now have all the “ingredients” available to create two new *stack manipulation* instructions: push the contents of the “A” register onto the stack (PSH), and pop the top stack item into the “A” register (POP). One possible application for such a pair of instructions is expression evaluation. Here, intermediate results of a calculation can be placed on the stack and retrieved when needed. For example, to evaluate the expression  $(W+X) - (Y-Z)$ , we could first calculate the quantity  $(Y-Z)$  and push it onto the stack, next calculate the quantity  $(W+X)$ , and finally pop the stack and subtract that value from our “running total”. Formal methods exist for transforming an arbitrarily complex, parenthesized expression into *postfix* form.

*stack manipulation  
instructions  
PSH  
POP*

*postfix*

Implementation of the PSH instruction requires two execute states. Here, the SP register must first be *decremented* in order to allocate space for the new item (given the convention we have adopted that SP points to the *top stack item*). After the SP has been decremented, it can be used as a pointer to indicate where in memory the contents of “A” should be stored.

For POP, however, the SP register is already pointing to the “right place”, enabling the “A” register to be loaded with the contents of that location on the first execute cycle. The “bookkeeping” step of de-allocating the item just popped off the stack (accomplished by incrementing the SP register) needs to follow, which at first glance appears to require a second execute cycle. Here, though, the same clock edge that is used to load the “A” register (with the value pointed to by the SP register) can be used to increment the SP register, since its value will not change until after the load has safely completed. The POP instruction, then, can be implemented using a single execute cycle. (Note the similarity between the *overlap* employed here and the overlap of the PC increment used previously in the fetch cycle.)

*de-allocation*

*overlap*

A modified system control table illustrating the addition of PSH and POP to our simple computer’s instruction set is given in Table S-19. Here, only one of the instructions listed (PSH) requires a second execute state (S[2]); the remaining instructions complete in a single execute cycle. Note, therefore, that RST is not asserted until the S[2] state of the PSH instruction, while for the other instructions RST is asserted during the S[1] state. A modified Verilog source file for the IDMS that corresponds to this version of our instruction set is given in Table S-20.

**Table S-19** System control table modifications for stack manipulation instructions.

Decoded State	Instruction Mnemonic	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY	SPI	SPD	SPA	RST
S0	—	H	H		H	H	H									
S1	HLT	L			L		L			L						
S1	LDA	H	H					H		H	H					H
S1	ADD	H	H					H		H						H
S1	SUB	H	H					H		H		H				H
S1	AND	H	H					H		H	H	H				H
S1	STA	H		H				H	H							H
S1	PSH													H		
S1	POP	H	H							H	H		H		H	H
S2	PSH	H		H					H						H	H

**Table S-20** IDMS modifications for stack manipulation instructions.

```
// System control equations modified to support PSH/POP instructions

assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND | POP)
                | S[2] & PSH);
assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND);
assign MWE = S[1] & STA | S[2] & PSH;
assign ARS = START;
assign PCC = RUN & S[0];
assign POA = S[0];
assign IRL = RUN & S[0];
assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
assign AOE = S[1] & STA;
assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
assign ALX = S[1] & (LDA | AND | POP);
assign ALY = S[1] & (SUB | AND);
assign SPI = S[1] & POP;
assign SPD = S[1] & PSH;
assign SPA = S[1] & POP | S[2] & PSH;
assign RST = S[1] & (LDA | STA | ADD | SUB | AND | POP) | S[2] & PSH;
```

Before adding our final set of simple computer extensions, some additional comments on PSH/POP are in order. Virtually every computer that has a stack mechanism implements some variation of the basic push/pop instruction pair, typically for each “important” register in the machine’s architecture. Other variations – which would be particularly useful for performing expression evaluation on our simple computer – include “pop and add” (i.e., pop the stack and add that item to the contents of the “A” register), “pop and subtract”, etc. In fact, instructions like “pop and add” are simple variations of the “basic POP” instruction, and can be implemented with only minor modifications to the Verilog source files given.

*pop and add*  
*pop and subtract*

### S.9.5 Subroutine Linkage Instructions

Another important “modern convenience” that most computers enjoy is a subroutine linkage mechanism, which is the final extension to our simple computer we will explore in this chapter. A very effective way to provide this capability is to utilize a stack. While there are other ways that subroutine linkage can be implemented in practice, use of a stack is attractive because it: (a) allows *arbitrary nesting* of subroutine calls; (b) provides a mechanism for passing parameters to subroutines; (c) allows recursion (the ability of a subroutine to call itself); and (d) allows reentrancy (the ability of a code module to be shared among quasi-simultaneously executing tasks).

*arbitrary*  
*nesting*

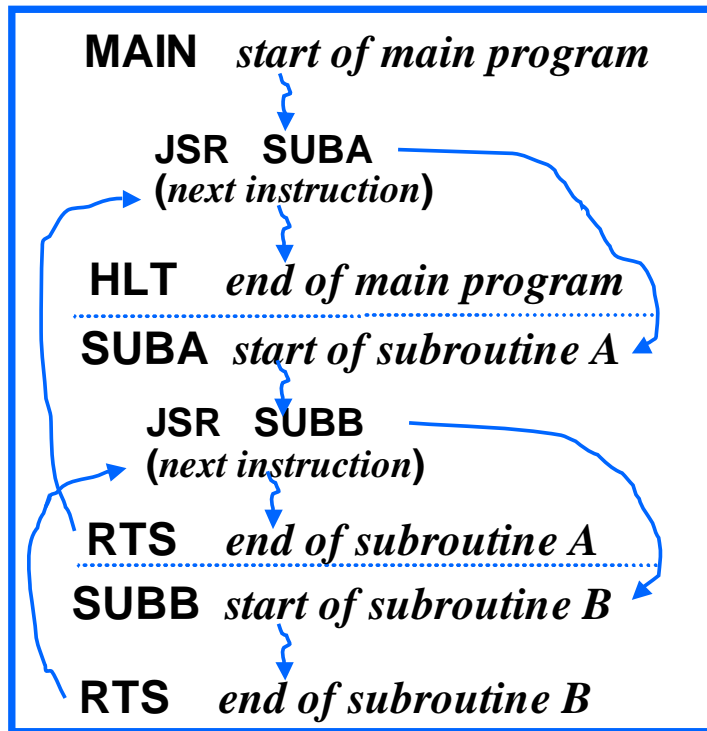
*recursion*  
*reentrancy*

The two subroutine-linkage instructions we will add to our “base” instruction set are “jump to subroutine” (JSR) and “return from subroutine” (RTS). Generically, we can simply refer to these as (subroutine) “call” and “return” instructions. As can be seen from the “subroutine linkage in action” illustration (Figure S-25), one of the key things the “call” instruction must do is establish a “return path” to the calling program (hence the name “linkage”). Placing the calling program’s *return address* on the stack affords nesting of subroutine calls (i.e., one subroutine calls another, which then calls another, etc.).

*return address*

Note that the return address is simply the address of the instruction that *follows* the JSR. Recalling that the PC is automatically incremented as part of the fetch cycle, we realize that the desired return address has already been calculated. The value in the PC simply needs to be pushed onto the stack when a JSR instruction is executed. Conversely, when a return from subroutine (RTS) instruction is executed, the top stack item needs to be popped off the stack and placed into the PC.

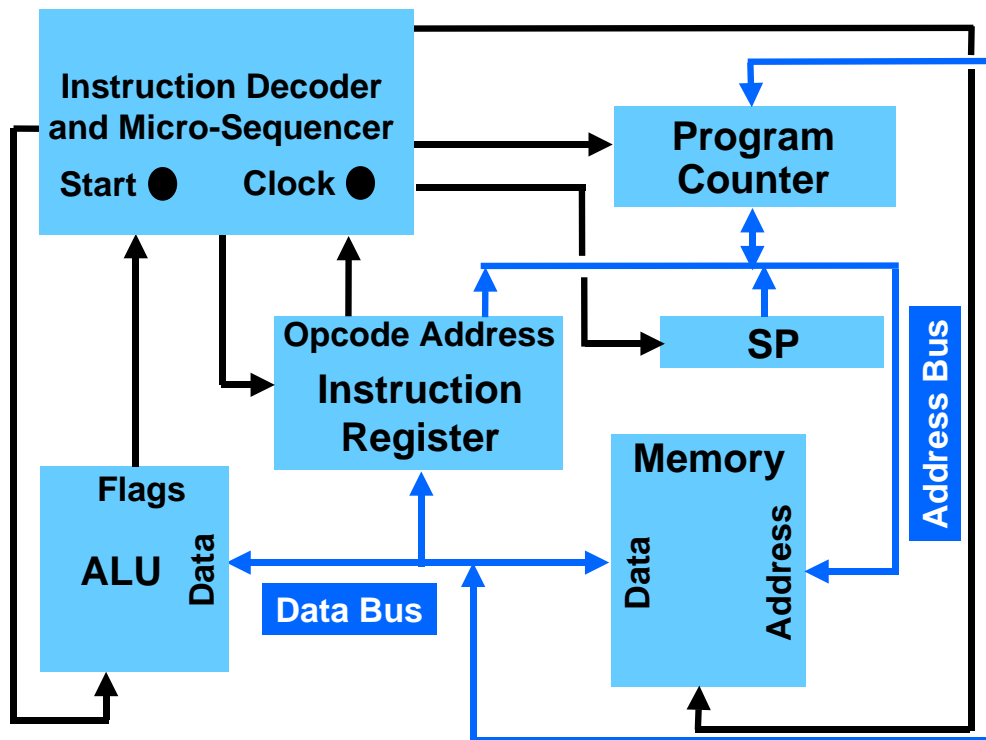




**Figure S-25** Subroutine linkage in action.

These observations indicate that, in order to add JSR and RTS instructions to our machine, the PC register needs to be modified. Specifically, a bi-directional interface to the system data bus needs to be added so that the value in the PC can be pushed/popped. Two new control signals need to be added to the PC for this purpose: PLD, for loading the PC with the value on the data bus (popped off the stack when an RTS instruction is executed); and POD, for gating the value in the PC onto the data bus (so that it can be pushed onto the stack when a JSR instruction is executed). A block diagram depicting the modified system is given in Figure S-26. A Verilog file for the modified PC is given in Table S-21.

Upon examining the block diagram of the modified system, one might initially be “disturbed” by the fact that the *width* (i.e., number of bits) of the PC register does not match that of data bus and/or memory – here, the PC register is only 5-bits wide, while the memory is 8-bits wide. In practice, though, this is of no consequence – we will simply use the lower 5-bits of the addressed memory location to store the value of the PC when it is pushed onto the stack. In most “real” computers, there is usually a better “match” between the PC and memory width (e.g., 32-bit address space and 32-bit wide memory).



**Figure S-26** Block diagram of simple computer with subroutine linkage mechanism.

We are now ready to outline the steps needed to execute the JSR and RTS instructions. First, we realize there are two fundamental steps associated with performing a JSR: (a) push the return address (the value in the PC register) onto the stack, and (b) jump to the location indicated by the instruction's address field. Step (a) is accomplished in a manner similar to the PSH instruction described in Section S.9.4: during the first execute cycle, the stack pointer is decremented; during the second execute cycle, the new item (here, the PC) is written to the location pointed to by the SP register. Step (b) is accomplished the same way as the unconditional "jump" instruction (JMP) described in Section S.9.3: the location at which execution of the subroutine is to commence is simply transferred from the IR to the PC via the address bus. Adding it all up, we find that a total of three execute states are needed to perform a JSR instruction.

**Table S-21** Modified PC for subroutine linkage.

```

/* Program Counter with Data Bus interface */
module pc(CLK, PCC, PLA, POA, RST, ADRBUS_z, DB_z, PLD, POD, PC);
  input wire CLK;
  input wire PCC;                // PC count enable
  input wire PLA;                // PC load from address bus enable
  input wire POA;                // PC output on address bus tri-state enable
  input wire RST;                // Asynchronous reset (connected to START)
  input wire PLD;                // PC load from data bus enable
  input wire POD;                // PC output on data bus tri-state enable
  inout wire [4:0] ADRBUS_z;     // address bus (5-bits wide)
  inout wire [7:0] DB_z;        // data bus (8-bits wide)
  output reg [4:0] PC;

  reg [4:0] next_PC;
  always @ (posedge CLK, posedge RST) begin
    if (RST == 1'b1)
      PC <= 5'b00000;
    else
      PC <= next_PC;
  end

  always @ (PLA, PLD, PCC, AB_z, DB_z, PC) begin
    // synchronous control signals PLA, PLD, and PCC are mutually exclusive
    if (PLA == 1'b1)            // load PC from address bus
      next_PC = ADRBUS_z;
    else if (PLD == 1'b1)       // load PC from data bus
      next_PC = DB_z;
    else if (PCC == 1'b1)       // increment PC
      next_PC = PC + 1;
    else                          // retain state
      next_PC = PC;
  end

  assign ADRBUS_z = POA ? PC[4:0] : 5'bZZZZZ;

  // pad upper 3 bits of DB with 000
  assign DB_z = POD ? {3'b000, PC[4:0]} : 8'bZZZZZZZZ;

endmodule

```

By way of contrast, execution of an RTS instruction requires only a single fundamental step: pop the return address off the stack and place it into the PC register. This is really not much different than the “basic pop” instruction (POP) described in Section S.9.4, except here the destination is the PC rather than the “A” register. Also, because RTS is merely a “pop PC” operation, it can be performed in a single execute cycle, just like the “pop A” (POP) instruction.

**Table S-22** System control table modifications for subroutine linkage instructions.

Dec. State	Instr. Mnem.	MSL	MOE	MWE	PCC	POA	IRL	IRA	AOE	ALE	ALX	ALY	PLA	POD	PLD	SPI	SPD	SPA	RST
S0	—	H	H		H	H	H												
S1	HLT	L			L		L			L									
S1	LDA	H	H					H		H	H								H
S1	ADD	H	H					H		H									H
S1	SUB	H	H					H		H		H							H
S1	AND	H	H					H		H	H	H							H
S1	STA	H		H				H	H										H
S1	JSR																H		
S1	RTS	H	H												H	H		H	H
S2	JSR	H		H										H					H
S3	JSR							H					H						H

**Table S-23** IDMS modifications for subroutine linkage instructions.

```
// System control equations modified to support JSR/RTS instructions

assign MSL = RUN & (S[0] | S[1] & (LDA | STA | ADD | SUB | AND | RTS)
                | S[2] & JSR);
assign MOE = S[0] | S[1] & (LDA | ADD | SUB | AND | RTS);
assign MWE = S[1] & STA | S[2] & JSR;
assign ARS = START;
assign PCC = RUN & S[0];
assign POA = S[0];
assign PLA = S[3] & JSR;
assign POD = S[2] & JSR;
assign PLD = S[1] & RTS;
assign IRL = RUN & S[0];
assign IRA = S[1] & (LDA | STA | ADD | SUB | AND);
assign AOE = S[1] & STA;
assign ALE = RUN & S[1] & (LDA | ADD | SUB | AND);
assign ALX = S[1] & (LDA | AND);
assign ALY = S[1] & (SUB | AND);
assign SPI = S[1] & RTS;
assign SPD = S[1] & JSR;
assign SPA = S[1] & RTS | S[2] & JSR;
assign RST = S[1] & (LDA | STA | ADD | SUB | AND | RTS) | S[3] & JSR;

endmodule
```

The system control table, modified to include the new JSR and RTS instructions, is shown in Table S-22. A Verilog file for the modified IDMS is given in Table S-23. Note that, since the JSR consumes all three execute cycles available, it technically “doesn’t matter” whether or not the RST signal is asserted during S[3] (since the 2-bit state counter will automatically “wrap around” to S[0] when the next clock edge occurs). It’s probably a good idea, though, to show RTS as being asserted on S[3], just in case future extensions to the instruction set require a state counter with additional bits.

### S.9.6 Other Possibilities

Having established the “basic modern conveniences” needed to implement a very simple computer, our imaginations could “go wild” thinking up new instructions and architectural extensions. We could accommodate additional instructions (opcodes) by simply increasing the number of opcode bits (an 8-bit opcode would give us 256 possibilities). And we could incorporate a more reasonably-sized memory by simply increasing the number of address bits. We could add new registers, such as an additional accumulator or an index register, as well as new addressing modes. An index register could be used as a pointer to memory, and facilitate implementation of a variety of new addressing modes.

## S.10 Summary and References

In this chapter we have introduced the design and implementation of a simple computer and progressively embellished it with a number of extensions. In addition to reviewing a “top-down, bottom-up” strategy for designing digital systems, we have provided a “bridge” between the basic digital logic design topics introduced earlier and microcontroller-oriented topics that will be the focus of post-requisite courses.

There are a number of texts that delve into the myriad of topics associated with computer architecture and design, written at a variety of levels. One of the best (and most widely used) introductory texts is Patterson and Hennessey’s *Computer Architecture: The Hardware-Software Interface* (Morgan Kaufmann). Their earlier text, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann), is an authoritative “advanced” text on the subject, used in numerous graduate programs.

Other highly regarded texts on computer architecture include Mano's *Computer Engineering Hardware Design* (Prentice-Hall), Stalling's *Computer Organization and Architecture* (Macmillan), Haye's *Computer Architecture and Organization*, and Hamacher's *Computer Organization*.

One of the best sources for unbiased reviews of the "latest and greatest" microprocessors is *Microprocessor Report* – a subscriber-supported periodical published by Cahners Electronics Group. Another excellent source of information on recent developments in microprocessor architecture is *IEEE Micro*, a publication of the IEEE Computer Society.

For information on embedded microcontrollers and applications, *Circuit Cellar Inc.* magazine is the source of choice. Web sites of the major manufacturers (Intel, Motorola, Texas Instruments, ST, Hitachi, etc.) continue to be the best sources for detailed information concerning specific microprocessors and microcontrollers.