10

A SIMPLE CPU DESIGN

The design of a simplified central processing unit (CPU) is covered in this chapter. This design exemplifies a somewhat more involved and practical design than the examples studied in the previous chapter. This entire chapter basically is a huge example that shows the most important considerations when designing a simple CPU. We start defining the CPU instruction set and the machine instruction word. What the instructions do. The registers, memory, and combinational logic blocks are the components that the CPU requires to be able to execute the defined instruction set. We will also cover the design of the sequencer or control section of the machine with the details of its state diagram and circuit implementation. Finally, a system section covers some of the most important aspects, and sometimes overlooked issues of embedded system design: clocks, resets, power decoupling, and timing. The goal of this chapter is not only to cover a simple CPU but at the same time a complex enough design example that is more comprehensive than previously covered design examples. The basic approach taken is mostly bottom up.

10.1 OUR SIMPLE CPU INSTRUCTION SET

This section introduces the reader to our small CPU instruction set. The instruction set is carefully picked such that various types of the most popular machine language instructions are represented. We will not categorize this

Electrical, Electronics, and Digital Hardware Essentials for Scientists and Engineers, First Edition. Ed Lipiansky.

design neither as a *CISC* or *RISC* example. *CISC* stands for *Complex Instruction Set Computer* and *RISC* stands for *Reduced Instruction Set Computer*. From a computer architecture point of view our design is closer to a *von Neumann* machine. This is an architecture that consists of a stored-program digital machine that has a central processing unit and a single separate memory unit that holds program instructions as well as data. An example of a *RISC* and *CISC* is covered in References 3, 4, and 6 in the Further Reading section.

The instruction set architecture (ISA) that our simple CPU supports consists of a few but very significant instructions that all real-world machines support. The purpose of studying a very simple CPU is to prove basic architecture concepts to the reader, which later on we will use to add real world factors that embedded systems face. Such factors over and above the computer architecture are timing analysis of the CPU, how to clock the machine, the reset logic, and integrated circuit power decoupling.

The basic instructions that our simple CPU supports are: *LOAD, STORE, AND, ADD*, (unconditional jump) *JMP*, (conditional branch) *BRNA*, and (complement) *CMPA*.

Our CPU has a single accumulator register or simply register *A*. Register *A* is a *16-bit* wide register. The computer memory has *4096 16-bit* wide words, that is, *4K* words. Since the memory has *4K* memory locations, the address width required to address each word uniquely, is *12* since 2^{12} is *4096*. The program counter register or the PC is *12* bits wide and it is used to store the address of the instruction to be executed immediately after the currently being executed instruction. Summarizing our CPU has *16*-bit wide data paths between register *A* and its memory. All memory accesses are done with a *12-bit* wide address. The PC stores the address of the to-be executed instruction. Our simple CPU has a single *16-bit* word instruction word. The lower *12* bits are used as an address to memory for those instructions that require such address, while the upper four bits are allocated as operational code bits *(opcode bits)* format. Opcode bits are unique binary codes defined for every unique instruction that the CPU supports. Table 10.1 below depicts the organization of our CPU instruction word format.

$$\text{Opcode} = \text{IWF}[15{:}12] \tag{10.1}$$

$$\text{Address X} = \text{IWF}[11{:}0]. \tag{10.2}$$

Now we are ready to explain what each instruction does from a programmer's model point of view.

Table 10.1 Instruction word format (IWF) bit assignments

MSB															LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operational code				Memory address X											

10.2 INSTRUCTION SET DETAILS: REGISTER TRANSFER LANGUAGE (RTL)

Micro-operations are the most basic actions that digital computers make. Examples of micro-operations are: register to register transfers, register to memory location transfers, memory location to register transfers, perform a logic or arithmetic operation between a register and the contents of a memory location, storing the result in the register or in the memory location. Data transfers are indicated as: $X \leftarrow Y$, where X and Y are registers. The contents of register Y, the source register, get transferred (in actuality is copied) to register X, the destination register. The original contents of Y are preserved. In addition to the data transfer itself; there may be conditions under which a transfer takes place. For example: If "a bit of some register is set" (i.e., the condition) transfer the contents of *Memory* location whose address is in the *memory address register MAR* into register X. This is indicated as: $X \leftarrow M\ [MAR]$, that is, specific sequences of micro-operations constitute macro-instructions or machine language instructions. Such instructions, in binary form, are loaded into the main memory of our simple CPU and the CPU fetches, decodes, and executes them. Assembly language is a symbolic language that allows programmers to more easily write low-level machine language. An assembler typically translates the assembly language instructions into machine language before execution. Let us get started with our very simple CPU instruction set.

LOAD Instruction: Syntax *LDA A, (X)*. This *opcode* is defined as 0000_2. This instruction reads the contents of memory location whose address is X and copies such contents into register A. It is important to recall that X is a *12-bit* address and that the memory contents at any memory address and the contents of register A are all *16* bits wide. The *16*-bit data transfer that the *LOAD* instruction produces is indicated as:

$$\mathrm{A} \leftarrow (\mathrm{X}). \tag{10.3}$$

The *LDA* instruction does not affect the state of the *S* (Sign) bit. The S bit is usually part of a condition codes register (CCR) or Processor Status Word (PSW). This register holds bits that are set or reset based on some arithmetic or logic operation outcomes. Our simple CPU will one have a *Sign bit* in its CCR. The sign bit is the *MSB* of the results produced by the Arithmetic and Logic Unit (ALU), to be discussed shortly.

STORE Instruction: Syntax *STA (X), A:* Its opcode is 0001_2. This instruction reads the contents of register A and copies them into memory location whose address is X. The *16*-bit data transfer that the *STORE* instruction produces is indicated as:

$$(\mathrm{X}) \leftarrow \mathrm{A}. \tag{10.4}$$

The *STA* instruction does not affect the state of the *S* (Sign) bit.

ADD Instruction: Syntax *ADD A, (X):* Its opcode is 0010_2. The *ADD* instruction reads the contents of a memory location whose address is *X*, adds them to the value contained in register *A*, prior to the execution of the *ADD*, and produces the sum of these two *16*-bit quantities, storing the result in register *A*. It is the programmer's responsibility to have some desired value in register *A* prior to the execution of the *ADD* instruction. After the *ADD* instruction is executed the original contents of *A* are overwritten. The *16*-bit addition and data transfer that the *ADD* instruction produces is indicated as:

$$\mathrm{A} \leftarrow \mathrm{A} + (\mathrm{X}). \tag{10.5}$$

Execution of this instruction sets the sign *(S)* flag. The *S* flag is a registered copy of the accumulator *MSB*. When the accumulator is zero or positive the *S* bit is zero, when the accumulator is negative the *S* bit is one. The *S* bit has the same meaning as the *MSB* in a 2's Complement number.

AND Instruction: Syntax *AND A, (X):* Its opcode is 0011_2. The *AND* instruction reads the contents of a memory location whose address is *X*, performs a bit-to-bit logical *AND* of the read memory contents and register *A* bits, and stores the *ANDing* of these two *16*-bit quantities in register *A*. It is the programmer's responsibility to have some desired value in register *A* prior to the execution of the *AND* instruction. After the *AND* instruction is executed the original contents of *A* are overwritten. The *16*-bit data transfer that the *AND* instruction produces is indicated as:

$$\mathrm{A} \leftarrow \mathrm{A}.(\mathrm{X}). \tag{10.6}$$

Execution of the *AND* instruction sets the sign flag accordingly.

JMP Instruction: Syntax *JPM X:* Its opcode is 0100_2. Upon execution of this unconditional instruction the PC gets loaded with address *X*, which is *bits 11:0* from the fetched instruction.

$$\mathrm{PC} \leftarrow \mathrm{X}. \tag{10.7}$$

The *JMP* instruction does not affect the *S* flag.

BRNA Instruction: Syntax *BRNA X*: Its opcode is 0101_2. This instruction is called branch on negative accumulator. This instruction looks at the *Sign* (*S*) bit, if the *S* bit is *1* (i.e., a negative *2's* Complement number is in the accumulator) the PC gets loaded with address *X*, which are bits *11:0* from the fetched instruction. When the *S* bit is zero (accumulator is zero or positive) no change to the PC takes place. That is the PC remains incremented by one from the fetch cycle. This may sound a little confusing but it will be understood better when we will study the data path architecture of our simple CPU:

$$\text{If } (\mathrm{S} = 1) \text{ then } \mathrm{PC} \leftarrow \mathrm{X}. \tag{10.8}$$

Table 10.2 Simple CPU instruction set

Instruction Syntax	Opcode (Binary) IWF [15:12]	Address X IWF [11:0]	Description of What Gets Executed	Affects Sign Flag?
LD A, (X)	0000	A valid address	$A \leftarrow (X)$	No
STA (X), A	0001	A valid address	$(X) \leftarrow A$	No
ADD A, (X)	0010	A valid address	$A \leftarrow A + (X)$	Yes
AND A, (X)	0011	A valid address	$A \leftarrow A \,.\, (X)$	Yes
BRNA X	0101	A valid address	If S = 1 then $PC \leftarrow X$	No
JPM X	0100	A valid address	$PC \leftarrow X$	No
CMP A	0110	Bits [11:0] are ignored	$A \leftarrow \bar{A}$	Yes

The *BRNA* instruction does not affect the *S* flag; however it uses the *S* setting made by some prior instruction to the *BRNA* to make a decision.

CMP Instruction: Syntax *CMP A:* Its opcode is 0110_2. Upon execution of this instruction the contents of accumulator register *A* become *1's* complemented.

$$\bar{A} \leftarrow A. \tag{10.9}$$

Execution of this instruction sets the *S* bit accordingly.

Table 10.2 summarizes the instruction set of our simple CPU.

10.3 BUILDING A SIMPLE CPU: A BOTTOM-UP APPROACH

Our CPU is required to have the registers and memory access that support the above-described instruction set. The CPU registers are part the programmer's model of the CPU. However, there will invariably be other registers, mechanisms and devices that are totally transparent (or not visible) to the programmer. So before looking at the big view of the data path we will study bits and pieces of the fundamental elements that constitute such data path architecture.

10.3.1 The Registers

Our CPU needs registers. Registers are used to hold memory addresses, memory data read or memory data to be written. Registers also hold the operand of the *ADD* and *AND* instructions. Registers are typically built with *D* type flip-flops; however they are not just a free running group of flip-flops. Why not? If the registers were free running they would get loaded on every

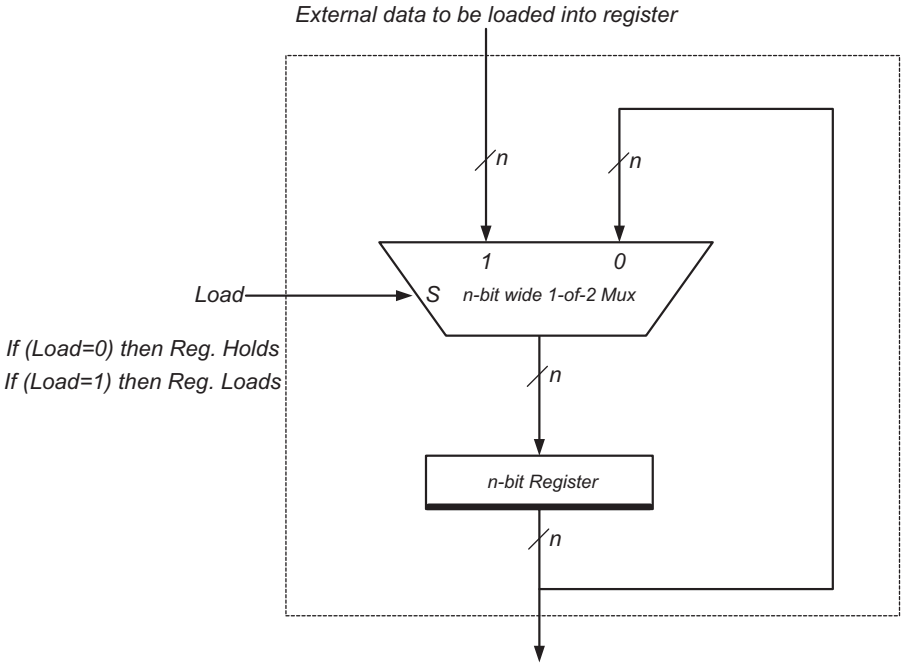


Figure 10.1 Implementation of a register with synchronous load control.

single active edge of the clock. To achieve a selective load of a register we place a *1-of-2 Mux* as shown in Figure 10.1. When the CPU control circuit drives a *zero* onto the *mux* select line, the register holds it data indefinitely because it keeps reloading itself with its own outputs for as long as clocks keep coming into the register. When a control circuit drives a *one* onto the *mux* select line, the external data placed on *channel 1* of the mux gets loaded upon the active edge of the clock clocking the register.

For example if we have a data path like the one shown in Figure 10.2, which has four 8-bit registers *A, B*, *C*, *and D*. Assume that somehow all four registers have their own initial values. Let us assume that on the next clock edge we would like the contents of register *A* to get transferred to register *B*, overwriting the current value of *B* and preserving the current values of *A*, *C*, and *D*. The control required to do that has to assert the load input for register *B* and keep the load inputs of registers *A, C*, and *D* negated. The mux select lines have to select input channel 0, which feeds the contents of register *A* onto the inputs of all four registers. Upon the active clock edge only register *B* will get written with the contents of *A* because register *B* has its load control input asserted while the other three registers do not. So upon the active edge of the clock clocking all four registers synchronously perform the following data transfers:

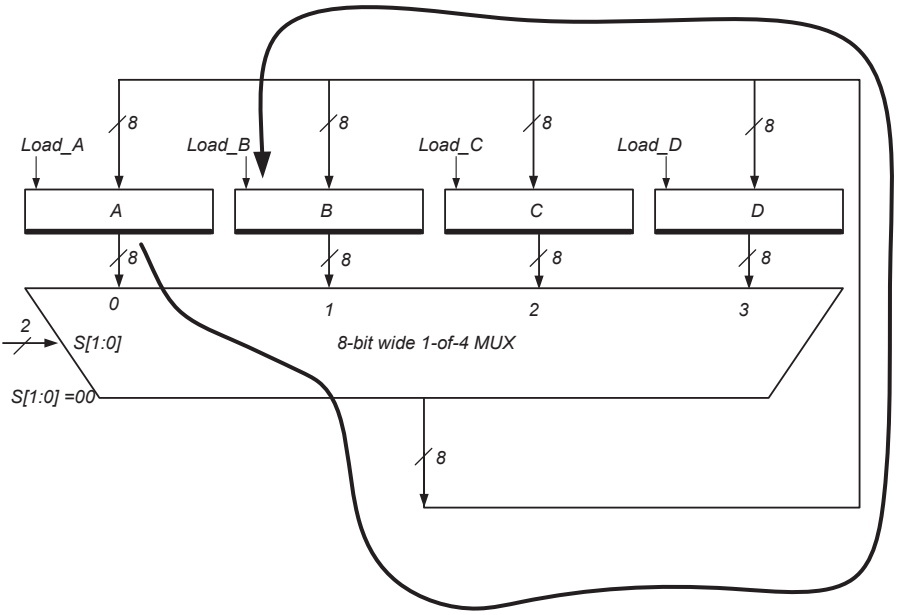


Figure 10.2 Simple data path architecture to show synchronous data transfers.

$$B \leftarrow A \tag{10.10}$$

$$A \leftarrow A \tag{10.11}$$

$$C \leftarrow C \tag{10.12}$$

$$D \leftarrow D. \tag{10.13}$$

Note from Equations (10.10) through (10.13) only *B* gets loaded with the contents of register *A*. Data transfers (Eqs. 10.11 through 10.13) show that *A*, *C*, and *D* preserve or hold their original contents, because their load control inputs are negated upon the assertion of the clock edge. It is very important to observe that although data transfers (Eqs. 10.10 through 10.13) are written in a sequence, actually all four of them take place concurrently. This is what a synchronous state machine does. For correct data transfers to take place, set-up and hold times of all flip-flops need to be met. We will address timing when we get to the control section of our simple CPU. Note that based on the simple data path depicted by Figure 10.2 we can transfer the contents of any one register to any two, or any three or all four registers. This is accomplished by asserting the load lines of all the registers we want the new data to get loaded into and by selecting the mux channel of the register that we want to source or provide the data.

10.3.2 The Memory Access Path or Memory Interface

Our CPU main memory or simply its memory is an array of 4096 16-bit words or 4Kx16. Twelve address lines are required to access 4096 locations since $2^{12} = 4096$. Each word is 16 bits wide so the data path to memory has 16 data lines. Memory is designed in such way that one 16-bit data word out of 4096 words can be accessed at any given time. Our memory has two control input lines: a READ and a WRITE. READ and WRITE can be both negated (non-asserted), but only one control input can be asserted at any given time. This means that we can only read a word from memory or write a word into memory at a time.

Practically two registers are needed to interface the CPU with its memory. The Memory Address Register or MAR register holds a memory address. The Memory Buffer Register or MBR receives the data read from memory on memory READS; or holds the data to-be-written into memory on memory WRITES. The interfacing protocol between the CPU and its memory is performed via the MAR and MBR registers.

Using the *MAR* and the *MBR* the data path transfers below are required to read memory:

1. *MAR* ← *Address;* places desired address to read memory from, in the *MAR*.
2. *MBR* ← *Memory [MAR];* retrieves the contents of memory location whose address is in the *MAR*.

For a memory write:

1. *MAR* ← *Address;* places desired memory address to write to in the *MAR*.
2. *MBR* ← Data; places desired data to be written into memory in the *MBR*.
3. *Memory [MAR]* ← *MBR;* performs the write. Transfers data from the *MBR* into the memory location whose address is in the *MAR*.

Figure 10.3 depicts the data path and interfacing registers with our memory array. Note that the *MAR* is *12* bits wide, because it has to hold a *12*-bit address. The *MBR* is *16* bits wide because the memory data is *16* bits wide. The memory access path not only shows the path for the memory data but also the memory address path. Data path pictures usually do not include control signals, like the *READ* and *WRITE* controls for the memory. Register *Q* outputs are represented with a heavy line, refer to *MAR* and *MBR* in Figure 10.3. The register clock is implicit. So when we see a rectangle with one of its sides being very thick, it means that we have a register. Usually, the width of the register is indicated with a little forward slash followed by the width of the input and output buses in bits.

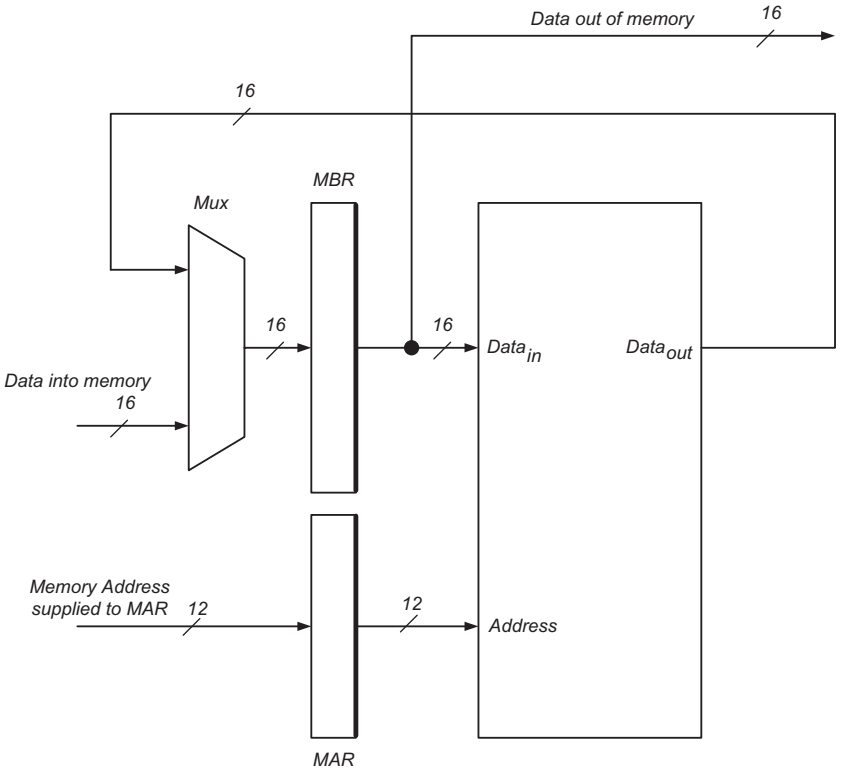


Figure 10.3 Memory data path and interfacing registers.

10.3.3 The Arithmetic and Logic Unit (ALU)

Instructions *ADD* and *AND* respectively require arithmetic and logic to be performed on its operands. We will see within this chapter that two registers will hold the operands that the ALU receives on its two input legs *P* and *Q*. Both the *ADD* and the *AND* operations are performed with combinational logic. Such logic constitutes the ALU. Figure 10.4 shows a high-level block diagram of our ALU. Essentially our ALU has two *16*-bit wide input legs, *P* and *Q* and a *16*-bit output leg *Z*. The ALU is designed to perform the operations listed in Table 10.3. Although it may not be clear as to why we need all those operations now, please hang on and everything will come together when we stitch together all the components of our simple CPU.

10.3.4 The Program Counter (PC)

We mentioned earlier that the *12*-bit wide PC holds the address of the to-be-fetched instruction when an instruction is currently being executed. The PC

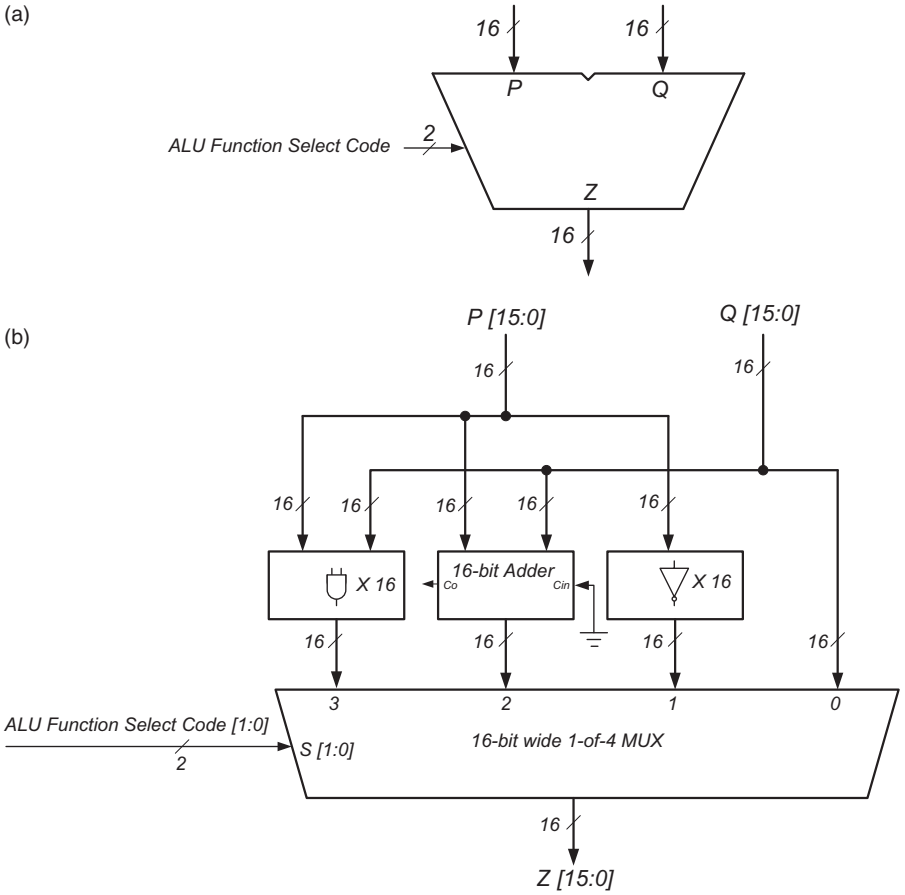


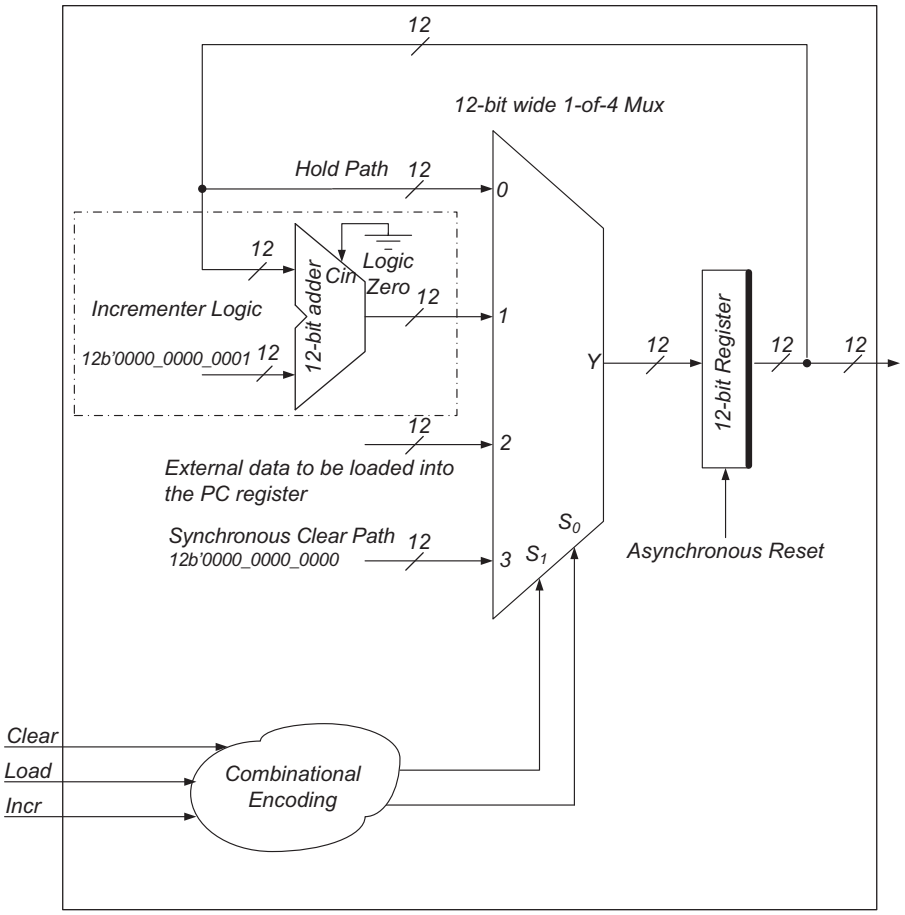
Figure 10.4 (a) ALU high-level block diagram; (b) ALU block diagram.

Table 10.3 ALU Basic operations

ALU Operation	High Level Description	ALU Function Select Code
Connect input leg Q to output leg Z	PASS Q to Z	00
Complement input leg P and connect it to output leg Z	$\overline{P}$ to Z	01
Arithmetic ADD of input legs P and Q send result to output leg Z	(P + Q) to Z	10
Logical bit-to-bit AND of input legs P and Q send result to output leg Z	(P . Q) to Z	11

keeps track of which instruction within a program the CPU is at. A program: "a sequence of instructions with a defined purpose" is an oversimplified but correct view of what a program is. During the execution of most of the CPU instructions the PC gets incremented by one, since all of our instructions are one *16*-bit word in length. So unless our program encounters an unconditional jump or a conditional branch instruction, the PC is always incremented by one. Upon a jump or branch instruction the PC wants to be loaded with a new destination or jump-to address. Such address, called address *X*, is fetched from memory along with the four-bit opcode. Once the PC gets loaded with this destination address *X* the jump will always fetch the next instruction from this new destination address. In the case of the *BRNA (branch on negative accumulator)* the branching will occur if the condition of a negative accumulator (*S* flag set) was met before the BRNA instruction. If the branch condition is met the next instruction fetch takes place from the destination address *X*. If the branch condition is false the branch does not occur and the PC remains loaded with the address of the previous instruction incremented by one. From all of the above we need to have the PC perform at least three distinct functions: hold its contents, auto-increment by one or get loaded with a new destination address. Additionally the PC will have an asynchronous *Reset* input line to ensure that its contents are cleared upon power-up reset. Because of all of that functionality the PC is a little more involved that the accumulator *A*, *MAR* or *MBR* registers. One way of implementing the PC is with a counter, to obtain the auto-increment feature. The PC could use the CPU ALU to increment its contents by one, but this is not desirable because the PC would be using the ALU, which is a valuable resource of the CPU. Figure 10.5 depicts an implementation of the PC. Table 10.4 describes all the operations that the PC performs. Note that the PC implementation in itself is a simple synchronous state machine. Its basic functions are: *(1) hold, (2) increment by one, (3) external data synchronous load*, and *(4) synchronous reset (or clear)*. Note that the combinational logic between the PC input control lines *(Incr, Load & Clear)* and the *12*-bit wide *1-of-4 mux* has two select lines S_1 and S_0, is designed according to Table 10.4, look under columns PC *Control Inputs* and *Mux Select Inputs*. Note that Rows *1* through *4* of Table 10.4 defines the main functions of the PC combinational logic. In general control lines *Incr, Load & Clear* must be asserted in a mutually exclusive fashion, when all control inputs are negated the PC holds its previously clocked state. Assertion of two or three PC control inputs is not meaningful and to avoid this illegal condition whenever they are asserted the PC register will simply hold its previously clocked state. Refer to Table 10.4 Rows *5* through *8*. Finally Row *9* indicates that when the clock into the PC is not active the PC register holds the previously clocked state. The asynchronous *Reset* line is not shown in Table 10.4 since this table is busy enough as it is. Figure 10.5 depicts a functional block diagram of the PC register architecture.

Table 10.4 describes the operation or characteristic table of the PC register. Carefully read Table 10.4 while and also inspect Figure 10.5.



NOTE:Register clock signal not explicitly shown.

Figure 10.5 The program counter register (PC).

Table 10.4 PC register characteristic table

Row	Clock	High-true PC Control Inputs			Mux Select Inputs		PC Next State Output Becomes	
#	Active Edge	Clear	Load	Incr	S_1	S_0	$Q_{n+1}[11:0]$	PC Function
1	↑	0	0	0	0	0	$Q_{n+1} \leftarrow Q_n[11:0]$	Hold
2	↑	0	0	1	0	1	$Q_{n+1} \leftarrow Q_n+1$	Increment
3	↑	0	1	0	1	0	$Q_{n+1} \leftarrow$ Ext-$Data_n[11:0]$	Load
4	↑	1	0	0	1	1	$Q_{n+1} \leftarrow 0$	Synchronous Clear
5	↑	0	1	1	0	0	$Q_{n+1} \leftarrow Q_n[11:0]$	Hold
6	↑	1	1	0	0	0	$Q_{n+1} \leftarrow Q_n[11:0]$	Hold
7	↑	1	0	1	0	0	$Q_{n+1} \leftarrow Q_n[11:0]$	Hold
8	↑	1	1	1	0	0	$Q_{n+1} \leftarrow Q_n[11:0]$	Hold
9	Inactive	X	X	X	X	X	$Q_{n+1} \leftarrow Q_n[11:0]$	Hold

10.4 DATA PATH ARCHITECTURE: PUTTING THE LOGIC BLOCKS TOGETHER

Do not read this section until you have a good understanding of everything in this chapter that precedes this section. We will be heavily referring to previous sections, figures and tables. Once you are are ready we will begin to discuss how the individual logic blocks from Section 10.3 fit together. When our CPU has to execute an instruction it does it in three basic stages: *(1) instruction fetch, (2) instruction decode (3) instruction execution*. From a digital design standpoint each of the stages mentioned may be have one or more states.

10.4.1 Data Path: *LDA* Instruction Fetch, Decode and Execution RTL

When an instruction has to be fetched from memory the *Program Counter Register* (PC), which should have the address of the to-be-executed instruction has to transfer its contents to the *MAR*. To fetch an instruction means that the instruction has to be read from memory and be placed in some register within the CPU. Remember that our CPU instructions are only one *16*-bit word long and it is not a multi-word instruction like in some advanced machines. The instruction upper four bits are the opcode and the lower *12* bits are address *X*. Refer to the *ISW* in Table 10.1. Upon being fetched, the instruction needs to be decoded; this tells the CPU what instruction was just fetched from memory and what else it needs to do. Upon the CPU figuring out which instruction it fetched, and assuming that in our example it was a *LDA A, (X)*; the CPU knows that it requires bringing a word of data from a memory location whose address is *X*. Such data are copied from memory into the memory *MBR*. Lastly the CPU transfers such word, now in the *MBR*, to accumulator register *A*. This last data path transfer finalizes the execution phase of the instruction. That is, $A \leftarrow (X)$, refer to Table 10.2.

Let us look at the data path architecture diagram of Figure 10.6. Our complete simple CPU data path and its main memory interface consist of four registers and the ALU. The PC register holds the address of the to-be executed instruction for the currently being executed instruction. The *MAR* register holds the address of a memory location the CPU wants to access. The *MBR* register is used to read data from and write data to memory. Remember that the PC register shown in Figure 10.6 is actually all the logic of Figure 10.5. We will explain the need for the *MAR* and the *MBR* multiplexers in the data path as we explain the operation of key instructions that use such muxes. Going back one more time to our *LDA* instruction, the fetch cycles consist of:

$$1.\ \mathrm{MAR} \leftarrow \mathrm{PC}. \tag{10.14}$$

For this transfer to happen the selection of channel *1* of the *MAR_MUX* enables the PC to the *MAR* path. The transfer of the PC contents into the *MAR* is setting up the address from where to do the *LDA* instruction fetch

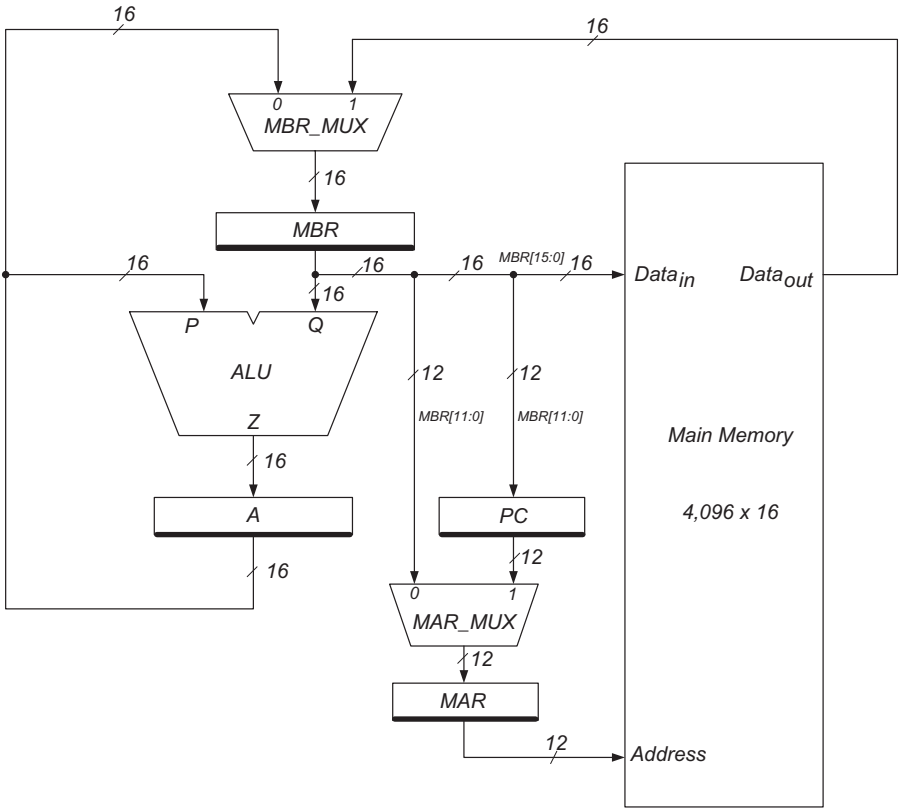


Figure 10.6 Data path architecture of our simple CPU.

from memory. Remember our instructions are all *16* bits wide and one word long. If Step (1) above happens to be the very first cycle that the CPU has to perform upon power-up reset or cold-start, the PC is previously cleared via its asynchronous clear line by the power-up reset circuitry (not shown), refer to Figure 10.5 to see the asynchronous clear line into the PC. The second data transfer that the *LDA* instruction requires is a memory read (or actual instruction fetch) that is:

$$2.\ \mathrm{MBR} \leftarrow \mathrm{M[MAR]};\ \mathrm{PC} \leftarrow \mathrm{PC} + 1. \tag{10.15}$$

Also at this time the contents of the PC are incremented by one. When the instruction completes with all its micro-operations the PC will already be pointing to the next instruction in memory. There is no reason to delay incrementing the PC or yet worse, to do it with a whole separate microinstruction. We will see shortly that if the PC needs to get loaded with a different value

(instead of its incremented value) when *JMP* and *BRNA* instructions are executed, the PC will get overwritten with the address that these instructions jump or branch to. After micro-operation (2) the *MBR* has fetched complete instruction, *4*-bit opcode and *12*-bit address *X*. Notice that at the completion of step (2) both the *MBR* gets loaded with the contents of memory pointed to by the address held in the *MAR* and the PC is incremented by one. The is no resource conflict for those two operations to be performed on the same state (or clock); that is because we designed the increment PC function such that it does not use the CPU ALU to do this. Refer to Figure 10.5 note that the PC has its own increment control line.

The next step for the CPU is to decode the opcode bits, which are in bits *MBR [15:12]*. Hence:

$$3.\ \text{Decode MBR [15:12]; MAR} \leftarrow \text{MBR [11:0]}. \tag{10.16}$$

Step 3 is a good time to transfer *MBR [11:0]* (address *X*) to the *MAR*, since address *X* will be needed to get the operand from memory. The operand refers to the data in memory that needs to be copied into the *MBR*. The two micro-operations in step three occur concurrently. Refer to Equation (10.16).

Now the CPU knows that the fetched instruction was an *LDA* and that data from memory location whose address is *X* has to be brought into the CPU *MBR*. Thus:

$$4.\ \text{MBR} \leftarrow \text{M [MAR]}. \tag{10.17}$$

Remember that the address in the *MAR* is already *X* from Step 3.

The heart of the CPU data path is its ALU and accumulator register *A*. Note that the ALU can *PASS* the contents of the *MBR* connected to the ALU *Q* input leg straight into its accumulator register *A*. This portion of the data transfer is required for the final execution path of the *LDA* instruction. So the ALU is placed in *PASS Q* mode by the control logic and the data from memory, now in the *MBR* gets transferred to register *A*, in one clock cycle. Hence:

$$5.\ \text{A} \leftarrow \text{MBR}. \tag{10.18}$$

Summarizing, the complete sequence of micro-operations to fully fetch, decode and execute our *LDA* instruction follows:

$$1.\ \text{MAR} \leftarrow \text{PC} \tag{10.19}$$

$$2.\ \text{MBR} \leftarrow \text{M [MAR]; PC} \leftarrow \text{PC} + 1 \tag{10.20}$$

$$3.\ \text{Decode MBR [15:12]; MAR} \leftarrow \text{MBR [11:0]} \tag{10.21}$$

$$4.\ \text{MBR} \leftarrow \text{M [MAR]} \tag{10.22}$$

$$5.\ \text{A} \leftarrow \text{MBR}. \tag{10.23}$$

We will see later that the five micro-instructions given by Equations (10.19) through (10.23) occur in five clocks.

10.4.2 All Other Instructions: Fetch, Decode and Execution: RTL

Having gone through the *LDA* instruction fetch, decode and execution in detail we will go over the rest of the instructions a little faster. We will emphasize the differences that each instruction presents with respect to a previously described instruction. To start with, it is important for the reader to know that all seven instructions of our simple CPU have the same identical fetch and decode steps. We will see in the control section that what we are calling steps are actually states of the controller state machine that steers the data transfers throughout the data path of the machine. In summary all instructions perform the same three states to do the instruction fetch (first two states) and instruction decode (third state). For the reader's convenience these three states are repeated here:

$$1.\ \mathrm{MAR} \leftarrow \mathrm{PC} \tag{10.24}$$

$$2.\ \mathrm{MBR} \leftarrow \mathrm{M}[\mathrm{MAR}];\ \mathrm{PC} \leftarrow \mathrm{PC}+1 \tag{10.25}$$

$$3.\ \text{Decode MBR [15:12]};\ \mathrm{MAR} \leftarrow \text{MBR [11:0]}. \tag{10.26}$$

10.4.2.1 Store Instruction Having said that, let us look into the *STA* instruction and its difference with respect to *LDA*. Referring to Table 10.2 *STA* stores data from register *A* into a memory location whose address is *X*. That is:

$$(\mathrm{X}) \leftarrow \mathrm{A}. \tag{10.27}$$

The *STA (X), A* instruction stores or writes the contents of register *A* into memory location whose address is *X*. *LDA* on the other hand reads memory from address *X*, refer to Equation (10.22). Continuing with our *STA* instruction, in its fourth state we need to transfer register *A* data into the *MBR* and in the fifth state we write *A* to memory. Note that the *MAR* is already loaded with address *X* from state (3), refer to Equation (10.26). Summarizing the five states of the *STA (X), A:*

$$1.\ \mathrm{MAR} \leftarrow \mathrm{PC} \tag{10.28}$$

$$2.\ \mathrm{MBR} \leftarrow \mathrm{M}[\mathrm{MAR}];\ \mathrm{PC} \leftarrow \mathrm{PC}+1 \tag{10.29}$$

$$3.\ \text{Decode MBR [15:12]};\ \mathrm{MAR} \leftarrow \text{MBR [11:0]} \tag{10.30}$$

$$4.\ \mathrm{MBR} \leftarrow \mathrm{A} \tag{10.31}$$

$$5.\ \mathrm{M}[\mathrm{MAR}] \leftarrow \mathrm{MBR}. \tag{10.32}$$

Basically states (4) and (5) (Eqs. 10.31 and 10.32) respectively, accomplish Equation (10.27), that is,

$$(\mathrm{X}) \leftarrow \mathrm{A}.$$

10.4.2.2 Add Instruction Referring one more time to Table 10.2 the *ADD A, (X)* performs:

$$\mathrm{A} \leftarrow \mathrm{A} + (\mathrm{X}). \tag{10.33}$$

ADD needs to read contents of memory location whose address is *X* and then add them to the existing contents of register *A* and store the results in *A*. So state four is identical to state four of the *LDA* instruction, which reads memory from location *X* into the *MBR*. The fifth state adds the read data now in the *MBR* to *A* and places the result in *A*. This last step overwrites the previous contents of *A* and sets the *S* bit accordingly. Thus *ADD* is:

$$1.\ \mathrm{MAR} \leftarrow \mathrm{PC} \tag{10.34}$$

$$2.\ \mathrm{MBR} \leftarrow \mathrm{M}[\mathrm{MAR}];\ \mathrm{PC} \leftarrow \mathrm{PC} + 1 \tag{10.35}$$

$$3.\ \text{Decode MBR [15:12]};\ \mathrm{MAR} \leftarrow \text{MBR [11:0]} \tag{10.36}$$

$$4.\ \mathrm{MBR} \leftarrow \mathrm{M}[\mathrm{MAR}] \tag{10.37}$$

$$5.\ \mathrm{A} \leftarrow \mathrm{A} + \mathrm{MBR}. \tag{10.38}$$

The produce Equation (10.38) the CPU controller has to select the ALU *ADD* function select lines. Refer to Table 10.3.

10.4.2.3 And Instruction From a data path or register transfer language (*RTL)* viewpoint *AND* is virtually identical to *ADD*. The sole difference is that the CPU controller selects the *AND* function of the ALU instead of the *ADD*. So for the AND instruction we have that:

$$1.\ \mathrm{MAR} \leftarrow \mathrm{PC} \tag{10.39}$$

$$2.\ \mathrm{MBR} \leftarrow \mathrm{M}[\mathrm{MAR}];\ \mathrm{PC} \leftarrow \mathrm{PC} + 1 \tag{10.40}$$

$$3.\ \text{Decode MBR [15:12]};\ \mathrm{MAR} \leftarrow \text{MBR [11:0]} \tag{10.41}$$

$$4.\ \mathrm{MBR} \leftarrow \mathrm{M}[\mathrm{MAR}] \tag{10.42}$$

$$5.\ \mathrm{A} \leftarrow \mathrm{A}.\ \mathrm{MBR}. \tag{10.43}$$

10.4.2.4 Conditional Branch Instruction Branch on negative accumulator needs to check the state of the *S* bit, upon this bit being one it loads the PC with address *X*, which as usual is already in the *MAR* from state 3. If the

S bit is zero then the PC remains with its previous contents, which are PC + 1 from state 2. Hence:

$$1.\ \text{MAR} \leftarrow \text{PC} \tag{10.44}$$

$$2.\ \text{MBR} \leftarrow \text{M}[\text{MAR}];\ \text{PC} \leftarrow \text{PC} + 1 \tag{10.45}$$

$$3.\ \text{Decode MBR}[15{:}12];\ \text{MAR} \leftarrow \text{MBR}[11{:}0] \tag{10.46}$$

$$4.\ \text{If S bit} = 1? \tag{10.47}$$

$$5.\ \text{Then: PC} \leftarrow \text{MAR}. \tag{10.48}$$

10.4.2.5 Unconditional Jump Instruction Simply requires loading the PC with the MAR that already has address X from state 3. This instruction only has four states, which are:

$$1.\ \text{MAR} \leftarrow \text{PC} \tag{10.49}$$

$$2.\ \text{MBR} \leftarrow \text{M}[\text{MAR}];\ \text{PC} \leftarrow \text{PC} + 1 \tag{10.50}$$

$$3.\ \text{Decode MBR}[15{:}12];\ \text{MAR} \leftarrow \text{MBR}[11{:}0] \tag{10.51}$$

$$4.\ \text{PC} \leftarrow \text{MAR}. \tag{10.52}$$

10.4.2.6 Complement Accumulator Instruction This is an instruction that already has its operand, that is, register *A*, in the CPU itself, it does not need to go to memory to read a location to get the operand like the *AND* and *ADD* instructions do. To generate $A \leftarrow \bar{A}$, the controller simply needs to select the ALU *PASS* $\bar{P}$ *to Z mode* to complement the accumulator and store it back into itself. Thus the *CMP A* instruction looks like:

$$1.\ \text{MAR} \leftarrow \text{PC} \tag{10.53}$$

$$2.\ \text{MBR} \leftarrow \text{M}[\text{MAR}];\ \text{PC} \leftarrow \text{PC} + 1 \tag{10.54}$$

$$3.\ \text{Decode MBR}[15{:}12];\ \text{MAR} \leftarrow \text{MBR}[11{:}0] \tag{10.55}$$

$$4.\ \text{A} \leftarrow \bar{A}. \tag{10.56}$$

Figure 10.7 depicts a complete state diagram of all seven instructions. The state assignment of each state is not done in Figure 10.7 yet. This will be addressed when we design the CPU controller state machine. Notice that for all seven instructions we used the numbers (1), (2), etc. just to indicate the sequence of states in time. Those numbers should not be construed as the state number assignment.

10.5 THE SIMPLE CPU CONTROLLER

The controller is the state machine that orchestrates the functioning of the data path data transfers, registers loading, PC incrementing, synchronous

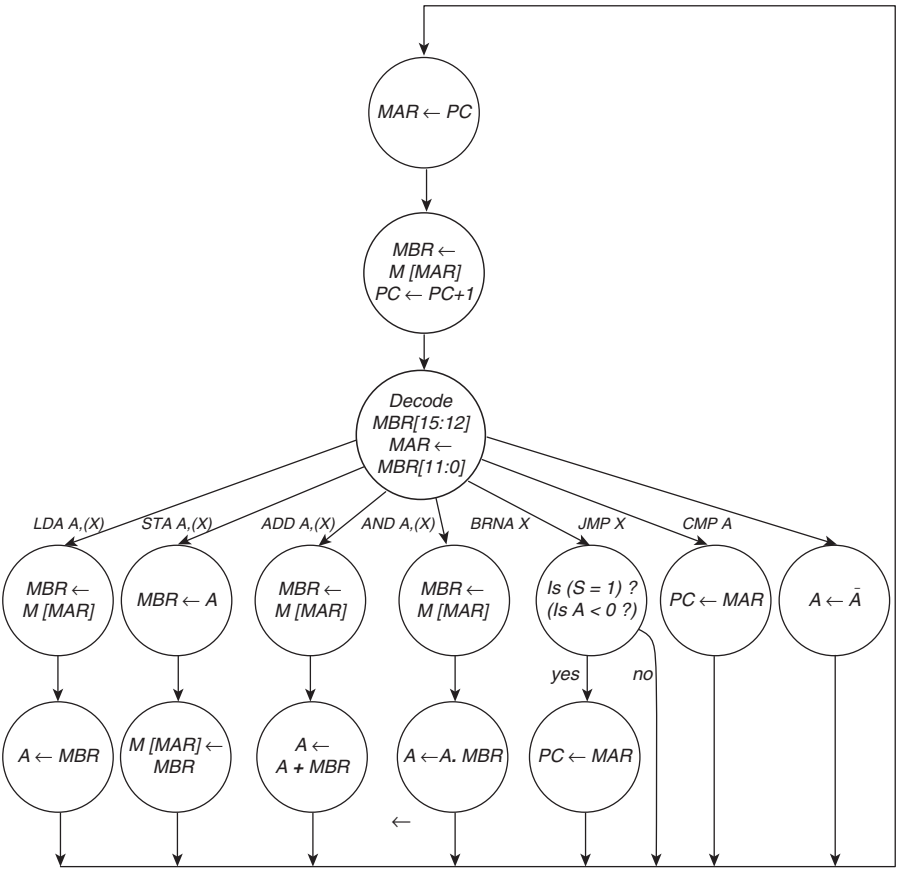


Figure 10.7 Simple CPU state diagram.

clearing, ALU function selection, *MAR* and *MBR* multiplexers steering, memory reads and writes, so that the instruction set is executed as described by its state diagram (Fig. 10.7). We need to first identify all the inputs and the outputs that our controller needs. This process has to be done by mainly careful inspections of Figures 10.6 and 10.7 and it is greatly a comprehensive process. Input signals to our controller are: (1) *MBR [15:12]* the opcode of each instruction. (2) The accumulator *MSB* stored in the *S* bit flip-flop (not explicitly shown on the data path diagram). (3) A system level asynchronous reset to clear the PC upon power-up. Outputs of the controller are listed by functional block and are the following: (1) For register *A*: *LOAD_A*, (2) for the *MAR: LOAD_MAR*, (3) for the *MBR: LOAD_MBR*, (4) for the PC: *INCR_PC*, *LOAD_PC* and *CLEAR_PC* all three signal being synchronous controls. (5) The asynchronous clear control for the PC: *ASYNC_CLEAR_PC* will be generated by the reset logic (not covered yet) and not by the controller. (6)

The *MAR mux* needs a *MAR_MUX_SEL* line, (7) the *MBR mux* needs a *MBR_MUX_SEL* line, (8) the ALU needs two bits of *ALU_FUNCTION_SELECT* bits. (9) We need a *READ* and a *WRITE* control signals for the memory array. Finally four bits of state are needed for our controller because since our state diagram has *16* states or less (actually *15*). In summary we have:

Five input bits for the controller and one input bit for the asynchronous *CLEAR* for the PC from the reset logic. Twelve control outputs and four state bits. State bits are outputs too. It is interesting to mention that a relatively large number of control signals are needed even though we are dealing with a very simple CPU.

10.5.1 State Assignments and Controller Implementation

Carefully reviewing *RTL* micro-operations (Eqs. 10.14 through 10.56) and the state diagram of Figure 10.7 we will make the following state assignments and justify their selection later. Starting with the first state at the top of Figure 10.7 we assign to it the value of *0*, then state *1* and *2* for the fetching and decoding states. For the *LDA* instruction, we assign states *3* and *4*. States *5* and *6* for *STA*; states *7* and *8* for *ADD*; states *9* and *10* for *AND*; states *11* and *12* for the *BRNA*, state *13* for *JMP*, and state *14* for *CMP A*. There is only one state that remains unassigned, state *15*. Since state *15* is an unused state we can design our controller state machine assigning unused state 15 to unconditionally go to the machine instruction fetch, state *0*. Another option is to make state *15* an isolated state (Fig. 10.8). Alternatively, if the CPU reaches state *15*, for example upon power up reset or due to some failure mechanism, we may choose to force the user to re-start or power cycle the CPU, since the CPU would freeze. Although this may seem a little unreasonable, it may be a better choice than letting the computer start fetching instructions from perhaps not the correct memory address. Or perhaps with some other register contents corrupted.

Let's talk about the state transitions that exist with the current state assignment. All transitions from state *0* to *1*, from state *1* to *2*, from state *3* to *4*, from state *5* to *6*, from state *7* to *8*, from state *9* to *10* and from state *11* to *12*, are achieved by incrementing the previous state by one. For example, you reach state 10 by incrementing state 9 by one. All state transitions from state *4 to 0, 6 to 0, 8 to 0, 10 to 0, 12 to 0, 13 to 0*, and *14 to 0* are reached by clearing a state register of the to-be-designed state machine controller. Finally all transitions from state *2* to state *3, 5, 7, 9, 11, 13*, and *14* are attained by loading the corresponding state into the state register. This has an important implication when we need to design the controller for our simple CPU. The reason is that the hardware implementation of the controller is greatly simplified because there are only three types of state transitions: that is, *Increment, Load or Clear*. This is the same register architecture as the one used for the PC register, refer one more time to Figure 10.5. We will implement the controller with a *4*-bit state register and three *1-of-16* multiplexers. Figure 10.9 depicts the hardwired

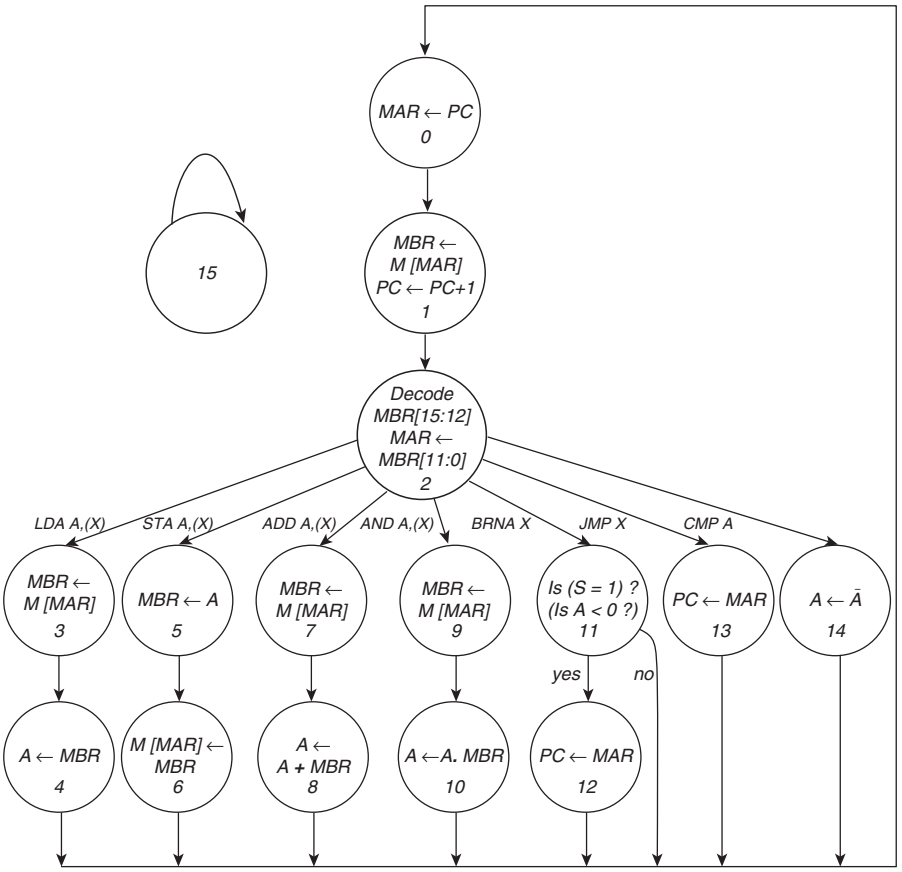


Figure 10.8 State assignments of our simple CPU state diagram.

implementation of the controller. It is called a hardwired implementation because it is not programmable or as easily changeable as it would be if we used a programmable memory device, such device is referred to as the micro-sequencer or controller micro-store. Because of space reasons this book does not deal with micro-store based controllers. However, the reader can find material for further study under the Further Reading section at the end of this chapter.

The hardwired controller of Figure 10.9 is a clean and simple implementation. The logic has not been minimized in any way. The multiplexers are there to emphasize the function we want the state counter to take; these functions are increment, load or clear, all of them synchronous functions. The initial clearing of the state counter or register is accomplished with it asynchronous reset supplied by the reset or power-on circuit, not shown. Careful analysis of

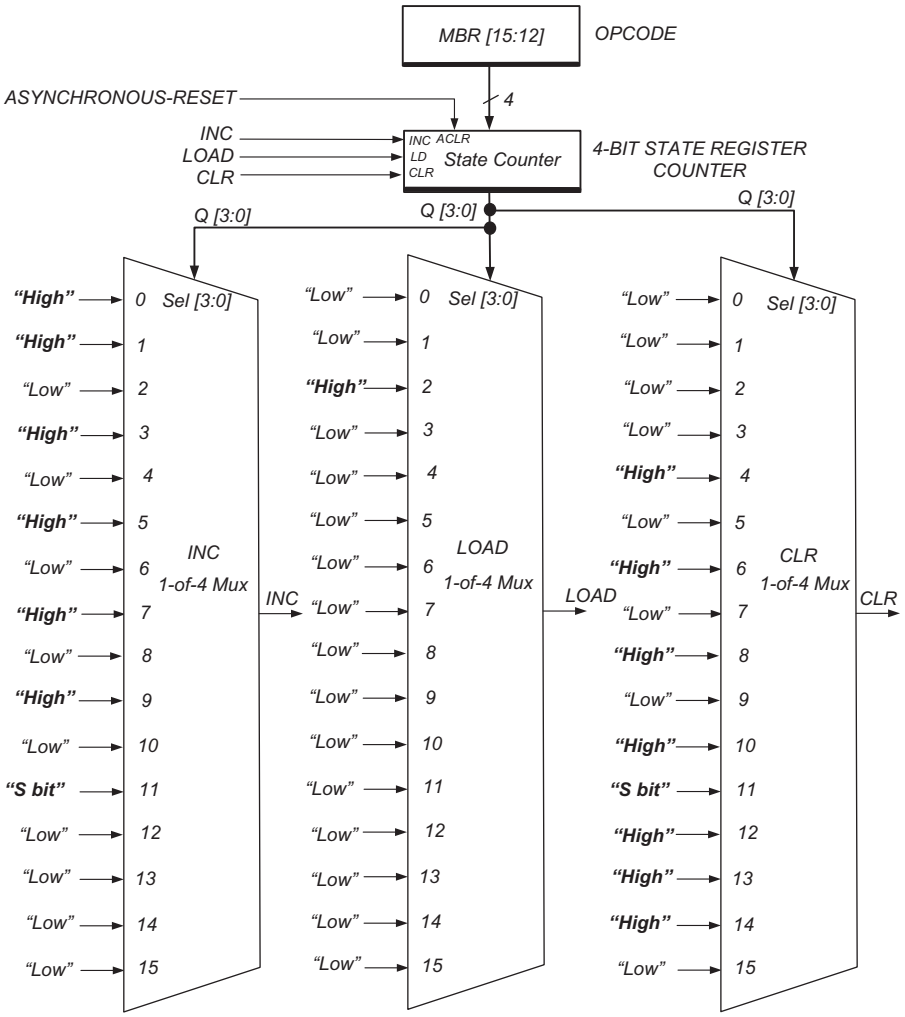


Figure 10.9 Hardwired implementation of our simple CPU controller or sequencer.

Figure 10.9 in conjunction with 10.8 allows the reader to understand the way the controller *walks* through each and every one of the assigned states, depending on the instruction that is presented for external loading to the state counter. Table 10.5 shows the mapping that needs to be produced from each state to the appropriate assertion of the output control signals. The state bits to output signal mapping can be implemented with a *Read Only Memory (ROM)* or combinational logic gates to do the decoding to assert the appropriate control outputs. The reader is asked, as an exercise, to design the combinational logic described by Table 10.5. Hint: The inputs of the logic should be *Q[3:0]* the state bits of the controller state counter.

Table 10.5 Combinational logic or ROM table to drive the CPU controller control outputs

State	Control Outputs										
Q[3:0]	LD_A	LD_MAR	LD_MBR	LD_PC	INC_PC	CLR_PC	MRB_ MUX_SEL	MAR_ MUX_SEL	ALU_FUNC_ SEL[1:0]	M_RD	M_WR
0	0	1	0	0	0	0	X	1	XX	0	0
1	0	0	1	0	1	0	1	X	XX	1	0
2	0	1	0	0	0	0	X	0	XX	0	0
3	0	0	1	0	0	0	1	X	XX	1	0
4	1	0	0	0	0	0	X	X	00	0	0
5	0	0	1	0	0	0	0	X	XX	0	0
6	0	0	0	0	0	0	X	X	XX	0	1
7	0	0	1	0	0	0	1	X	XX	1	0
8	1	0	0	0	0	0	X	X	10	0	0
9	0	0	1	0	0	0	1	X	XX	1	0
10	1	0	0	0	0	0	X	X	11	0	0
11	0	0	0	0	0	0	X	X	XX	0	0
12	0	1	0	0	0	0	X	1	XX	0	0
13	0	1	0	0	0	0	X	1	XX	0	0
14	1	0	0	0	0	0	X	X	01	0	0
15	0	0	0	0	0	0	X	X	XX	0	0

In Table 10.5 several control output names have been abbreviated: for example *LOAD_A* is renamed: *LD_A*, to have more room on the table. Let us refer to the *LDA* instruction *RTL* given by Equations (10.19) through (10.23). Referring to Table 10.5, Figure 10.6 and the *LDA RTL* repeated for the reader's convenience, we verify how Table 10.5 was filled in.

$$\text{State 0: MAR} \leftarrow \text{PC} \tag{10.57}$$

$$\text{State 1: MBR} \leftarrow \text{M [MAR]; PC} \leftarrow \text{PC} + 1 \tag{10.58}$$

$$\text{State 2: Decode MBR [15:12]; MAR} \leftarrow \text{MBR [11:0]} \tag{10.59}$$

$$\text{State 3: MBR} \leftarrow \text{M [MAR]} \tag{10.60}$$

$$\text{State 4: A} \leftarrow \text{MBR.} \tag{10.61}$$

In state *0:* we need to assert *LD_MAR = 1* and *MAR_MUX_SEL = 1* to transfer data from the PC into the *MAR*, and all other control signals need to be negated upon receiving the active edge of the clock. Once in state *1: LD_MBR = 1, MBR_MUX_SEL = 1* and *M_RD = 1* reads memory location pointed to by the *MAR* and saves the read data in the *MBR*. Simultaneously the *INC_PC = 1* to increment the PC by one. Once on state *2*: The *MBR[15:12]* get looked at (decoded) by the controller and *MBR[11:0]* transferred to the *MAR*, so that *LD_MAR = 1, MAR_MUX_SEL = 0*. This micro-operation just copied address *X* into the *MAR*. The controller asserts its *LOAD Mux* output to load a jump to state *3* to go to the *LDA* instruction execution sequence. On state *3 M_RD = 1*, *MBR_MU_SEL = 1* and *LD_MBR = 1* reads memory data from address *X*. Finally once in state *4 ALU_FUNC_SEL = 00 (PASS Q to Z)* and *LD_A = 1* transfers the contents in the *MBR* to register *A*.

For the rest of the instructions the reader should refer to the state diagram with state assignments of Figure 10.8 and with the aid of Figure 10.6 (data path architecture) start verifying the correctness of the contents of the rest of Table 10.5. Please allocate, as long a time as you need, the first time going through the complete table may take more than a few hours and more than one sitting. I have been there; please do not feel frustrated, this will only make sense once you go over the material exhaustively.

10.6 CPU TIMING REQUIREMENTS

This section will be concerned with identifying the timing paths in the machine data path, memory interface and controller. To study these paths we will make a reasonable assumption, which is that all clocks arrive at their registers clock inputs at the same time. This means that we assume that there is zero clock-skew. Starting with the data path architecture of Figure 10.6 timing paths are simply identified starting from the *Q* outputs of a register working your way

though a combinational data path or simply through a bus wire into the *D* inputs of either the same or another register.

The following are the long paths in Figure 10.6:

1. *PC Q* outputs through *MAR_MUX* into *MAR D* inputs.
2. *MRB Q* outputs through *MAR_MUX* into *MAR D* inputs.
3. *MBR Q* outputs through *ALU leg Q* into *A* register *D* inputs.
4. Register *A Q* outputs through *ALU leg P* into register *A D* inputs.
5. Register *A Q* outputs through *MBR_MUX* into *MBR D* inputs.
6. *MBR Q o*utputs into *PC D* inputs.
7. *MAR Q* outputs through memory address to data out through *MBR_MUX* into *MBR D* inputs.
8. *MBR Q* outputs though memory data in and data out, through *MBR_MUX* into *MBR D* inputs.

For the logic of the CPU controller of Figure 10.9 we can identify the following paths:

9. *MBR Q* outputs to state counter *D* inputs.
10. State counter *Q* outputs to *INC MUX* select lines to state counter *INC* input.
11. State counter *Q* outputs to *LOAD MUX* select lines to state counter *LOAD* input.
12. State counter *Q* outputs to *CLR MUX* select lines to state counter *CLR* input.
13. *S*-bit flip-flop *Q* output through *INC Mux INC* output into state counter *INC* input.
14. *S*-bit flip-flop *Q* output through *CLR Mux CLR* output into state counter *CLR* input.

When we want to study for each timing path their corresponding long path, to calculate if the set-up time is met we proceed as follows with path (1): We use the maximum clock-to-output time of the sourcing register, the maximum propagation delay of the combinational logic and the minimum set-up time required by the *D* input of the receiving register.

So when we want to study path (1) as a long path it becomes:

$$1.\ \mathrm{PC}\ t_{\text{clock-to-output max delay}} + t_{\text{max pd MAR\_MUX}} + \mathrm{MAR}\ t_{\mathrm{SU}}.$$

Finally to determine if such path meets the set-up time requirement of the MAR register we need to check if:

$$\mathrm{PC}\ t_{\text{clock-to-output max delay}} + t_{\text{max pd MAR\_MUX}} + \mathrm{MAR}\ t_{\mathrm{SU}} \leq T_{\mathrm{clkmin}}.$$

When we want to study the effect of the same path (1) as a short path, we consider the fastest or shortest delay at which the *Q* outputs of the PC travel plus the shortest delay through the *MAR_MUX* combinational logic. Since we want to analyze the corresponding short path we want the check if the following inequality is met:

$$\text{PC } t_{\text{clock-to-output min delay}} + t_{\text{min pd MAR\_MUX}} \geq \text{MAR } t_{\text{HOLD}}.$$

The reader is strongly encouraged to establish all 14 long path and 14 short path equations for the complete CPU. In general we find that the longest path usually is the path through the ALU because it has the largest amount of combinational logic. The usual short paths are those that connect registers *Q* outputs through short wires straight into registers *D* inputs. One such example in our CPU design is the *MBR[15:12]* to state counter inputs.

10.7 OTHER SYSTEM PIECES: CLOCK, RESET AND POWER DECOUPLING

10.7.1 Clock

The clock of a synchronous state machine is typically generated with a crystal-based oscillator with good stability. Clocks are buffered and distributed to its loads in a point-to-point fashion. With the current sub-nanosecond rise/ fall times and clock frequencies of today, virtually all signals are interconnected point-to-point for optimal signal integrity and timing behavior. Clock buffers used should be preferably packaged within the same *IC* package and the wire lengths should be matched. When the clock frequencies are high, like several hundred megahertz crystal oscillators are not available so *Phase Lock Loop* circuits (*PLL)* are used to generate gigahertz range frequencies. PLLs are not within the scope of this book and an excellent reference [7] to this topic is given in the Further Reading section of this chapter.

10.7.2 Reset

Upon good power being applied to a CPU-based system, reset is the first hardware signal that the system requires to power-up correctly. In our simple CPU, reset clears the PC and the CPU controller state counter register. In real world systems reset initializes or clears all the appropriate registers on board or within programmable devices such as *CLPDs* and *FPGAs*, it also allows the clock generating circuit to start running. It is extremely important for the reset or *Power-On Reset (POR)* signal not to glitch, because that may cause registers to come out of reset at different times and the system to behave unpredictably. It is desirable to assert reset asynchronously to ensure that all resettable devices are cleared regardless of the state of their clock. However,

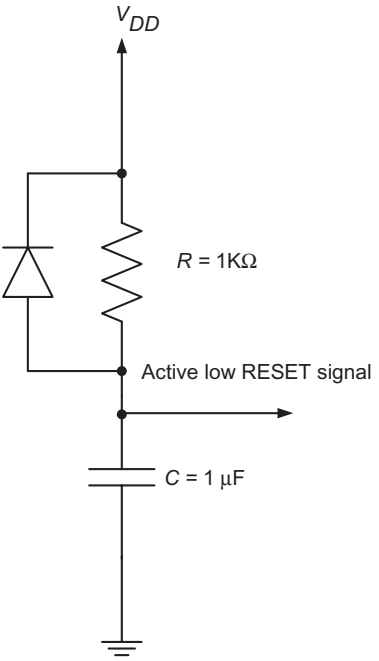


Figure 10.10 A poor example of a reset circuit.

it is more advisable to negate reset synchronously. Reset should be released (negated) synchronously, because releasing reset in an uncontrolled environment (i.e., asynchronously) may cause its flip-flops or registers to go metastable. Two problems may occur upon an asynchronous reset release: (a) the reset recovery time may be violated and (b) reset negation may occur at different clock cycles for different clocked elements. The reset recovery time is the time between when reset is negated and the time that the clock edge signal goes active again. Figure 10.10 depicts a very bad example of a reset circuit. It has multiple problems. Upon power-on, assuming a discharged capacitor, the low-true RESET signal will start at zero and has an exponentially increasing waveform. This waveform may not be suitable for the ICs that receive the reset, because it may stay at the IC logic threshold too long. It may not apply reset for the required time because the exponential ramp up time is not very precise. Chips generally have a normal operating voltage plus and minus 5 or 10 percentage points of such voltage. The assertion of reset with the circuit of Figure 10.10 is not very well defined. The diode that the circuit has across the 10 kΩ resistor is to provide a quick discharge path of the capacitor if a user turns off the system and turns it back on quickly. A better scheme would use a Schottky diode since it has a lower forward voltage drop.

A somewhat improved reset is depicted in Figure 10.11.

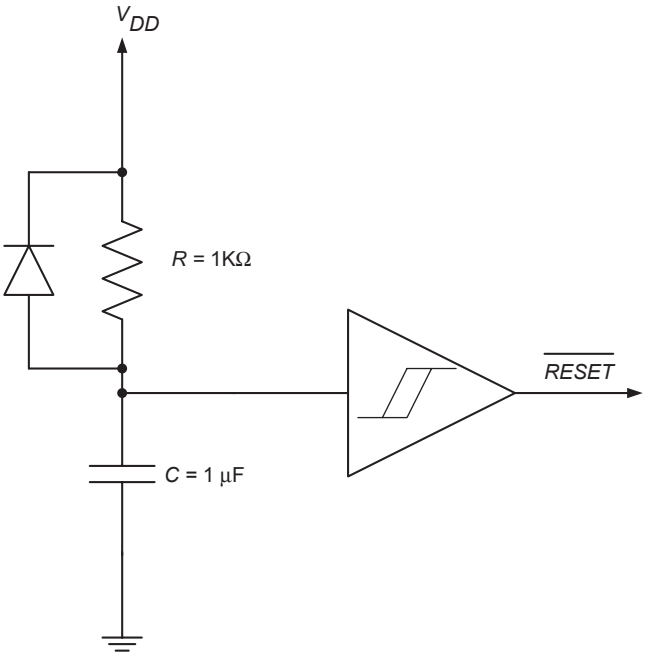


Figure 10.11 Improved reset circuit.

The advantage of this circuit is that a Schmitt trigger logic gate is used to shape the reset pulse. A Schmitt trigger gate has built-in hysteresis properties, thus filtering out short-lived glitches and reset pulse variations.

An even better approach can be implemented with two D flip-flops synchronously clocked, but having asynchronous reset inputs. The second flip-flop reduces metastability. The operation of this reset circuit is clean. Reset out is active low. It produces an asynchronous active high reset, even before the clock runs. But upon active low Reset_Out being released a low level (i.e., no reset) is synchronously clocked into active low reset_out. The two synchronizing flip-flops reduce the probability of metastable behavior to practically negligible levels. Figure 10.12 depicts such reset circuit. This circuit is commonly used at board level as well as in programmable devices level designs.

At the board level reset chips are available from several IC manufacturers, such as Analog Devices (ADI), Intersil, Maxim, ST Microelectronics, Texas Instruments (TI) and several others. Such ICs are referred to as *supply voltage supervisors or reset chips*. Supervisor circuits monitor system voltages from a range that may vary from about 0.5V to some upper voltage limit like 5 V. When the voltage dips below a preset threshold or when a manual reset (typically a pushbutton) drops to a logic low (active low manual reset) the active low open drain reset output asserts. It usually remains asserted low for a user-programmed time delay. An external resistor and capacitor time

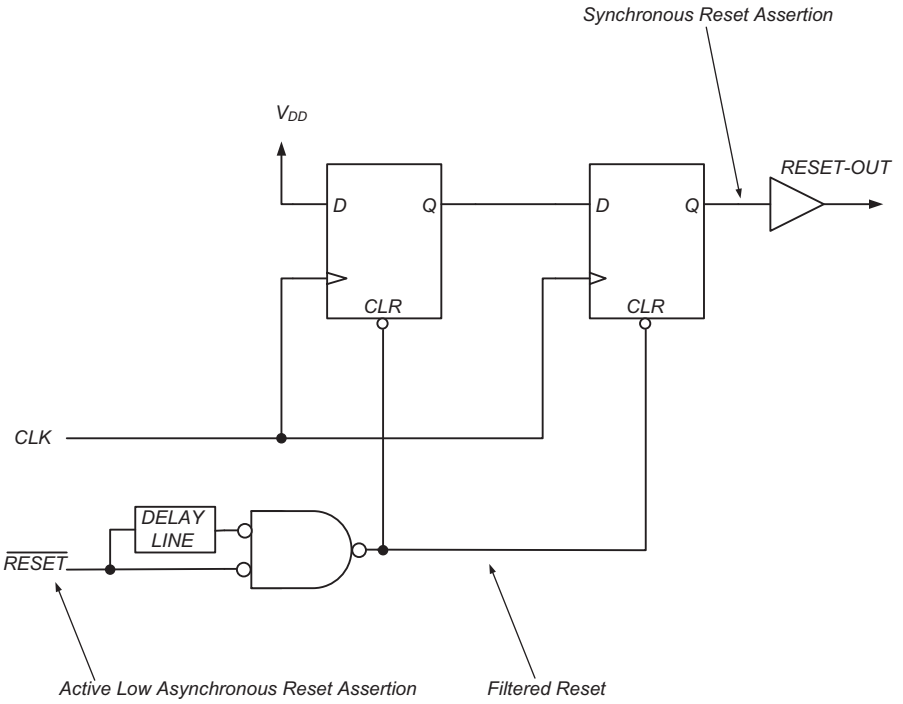


Figure 10.12 Reset circuit with asynchronous assertion and synchronous negation.

constant usually control such time delay. These supervisor chips use a precision reference voltage to achieve good threshold accuracy (typically 1% or better). When the *DC* voltage to an embedded system dips below the required minimum voltage for proper operation, the supervisor circuit asserts the reset signal; this initiates a system shut down. Current flow stops and the voltage to the supervisor may increase due to decreased IR drop. This produces a false reset negation from the supervisor circuit, that is, the supervisor incorrectly turns the system back on. To mitigate that problem sensing voltage hysteresis is provided to the supervisory circuit. For more details on supervisory circuits refer to the websites of the IC manufacturers mentioned above. Examples of some power-on reset or supervisory ICs are: Intersil ISL6131, Maxim MAX691A, MAX700 and MAX800 series, Analog Devices ADM63xx series, TI TPS3808 series, and many others.

10.7.3 Power Decoupling

Just like any *IC* on a system good decoupling with low equivalent series resistance (*ESR)* and low lead inductance capacitors must be supplied to every *IC* on the board. What is decoupling for? Power supplies provide voltage and

current to integrated circuits over the time they need to be operational. IC's make sudden transitions, usually in the nanosecond range, of one or many of their inputs and output pins. Such fast signals transitions produce high current demands as the power across the IC droops a few millivolts. The power across the IC power pins droops, because there is no power supply that can have a time response to such a fast power demand. The power supply and its power distribution scheme, cables, traces and wires that route the power to the point of consumption, both have a finite and definitely non-zero response to current transients. For example *300 kHz* switching power supplies, may have a bandwidth or capacity to respond to current transients of about *10* microseconds. During these *10* microseconds the decoupling capacitors, placed in extremely close proximity to the IC being decoupled, provide the amount of current for the amount of time that the power supply requires to react and start to provide current to the IC. This indicates that the decoupling capacitor in a very first pass approximation has to be able to provide enough current for a certain minimum time allowing its voltage to droop no more than a predetermined limit. Such calculation is based on the basic equation than links current voltage and time in a capacitor. From Chapter 1 we know that such relationship is:

$$i_C(t) = C\frac{dv_C(t)}{dt} \tag{10.62}$$

Let us consider the following numerical example to illustrate how to calculate the value of decoupling capacitance needed. Assume that our power supply has a bandwidth of *100 kHz*, which means that it will be able to respond to current transients after *10 microseconds* from the beginning of the current transient event. Assume that we want to have a maximum voltage droop across the IC power pins of no more than 300 mV. Finally assume that the current transient demanded by the IC is *1 A*. From Equation (10.62) we calculate C as:

$$C = \frac{i_C d(t)}{dv_C(t).} \tag{10.63}$$

Replacing differentials with finite time and voltage increments or decrements in Equation (10.63) we obtain:

$$C = \frac{1\times 10\times 10^{-6}}{0.300} = 33.333\,\mu\text{F}.$$

When decoupling calculations need to done more accurately equivalent series resistance *(ESR)* and lead inductance *(ESL)* of the capacitor should be taken into account. A very comprehensive treatment of this topic can be found in [1,2].

10.8 SUMMARY

This chapter defined the instruction set of a very simple CPU. A data path architecture was presented to support the defined instruction set. Using such data path we covered the microinstructions of every machine language instruction or simply called a macroinstruction. We further developed the state diagram of the complete instruction set, and the sequencer design.. The longest instructions take five states or clocks to fully execute and the shortest ones take four clocks. We further look into some system level issues: timing, clocks, and their distribution, resets and power decoupling.

FURTHER READING

1. Larry Smith, *Decoupling Capacitor Calculations for CMOS Circuits*, Electrical Performance of Electronic Packaging Conference, 1994.
2. Larry Smith, et al., *Power Distribution System Design Methodology and Capacitor Selection for Modern CMOS Technology*, 1999.
3. David Money Harris and Sarah L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann Publishers, San Francisco, CA, 2007.
4. David A. Patterson and John L. Hennessy, *Computer Organization and Design the Hardware/Software Interface*, Morgan Kaufmann Publishers, San Mateo, CA, 1994.
5. John Mick and Jim Brick, *Bit-Slice Microprocessor Design*, McGraw-Hill, New York, 1980.
6. Carl Hamacher, Zvonko Vranesic, and Safwat Zaky, *Computer Organization*, 5th ed., McGraw-Hill, New York, 1978.
7. Roland Best, *Phase Lock Loops*, 6th ed., McGraw-Hill, New York, 2007.

PROBLEMS

10.1 Design the two new instructions shown at the bottom of Table 10.6
Note:

1. The new instruction set supports two more instructions: *LD B, (X)* and *STA (X), B.*

2. The CPU has one new 16- bit register: that is, register B.

(a) Make any needed modifications to the data path architecture of Figure 10.6.

(b) Draw a complete state diagram only of the two new instructions shown in Table 10.6.

10.2 In reference to Figure 10.9 the CPU controller:

(a) Redesign the control logic of Figure 10.9 using a minimally sized ROM and D-type registers for the new instruction set of Table 10.6.

Table 10.6 New simple CPU instruction set

Instruction Syntax	Opcode (Binary) IWF [15:12]	Address X IWF [11:0]	Description of What Gets Executed	Affects Sign Flag?
LD A, (X)	0000	A valid address	$A \leftarrow (X)$	No
STA (X), A	0001	A valid address	$(X) \leftarrow A$	No
ADD A, (X)	0010	A valid address	$A \leftarrow A + (X)$	Yes
AND A, (X)	0011	A valid address	$A \leftarrow A.\ (X)$	Yes
BRNA X	0101	A valid address	If S = 1 then $PC \rightarrow X$	No
JPM X	0100	A valid address	$PC \leftarrow X$	No
CMP A	0110	Bits [11:0] are ignored	$A \leftarrow \bar{A}$	Yes
LD B, (X)	0111	A valid address	$B \leftarrow (X)$	No
STA (X), B	1000	A valid address	$(X) \leftarrow B$	No

(b) Draw the circuit schematic of the new simple CPU controller logic.

(c) Write the micro-code that the ROM needs to perform the complete instruction set given by Table 10.6.

10.3 Assume we want to add another new instruction to Table 10.6 New Instruction Set. This new instruction is:

$$\text{OR A, (X); A} \leftarrow \text{A} + \text{(X)}$$

The *OR* instruction does a bit-to-bit *OR* operation between the contents of memory location whose address is *X* and the contents of register *A*, it finally stores the result in register A, overwriting its original contents. Remember that the new instruction has to increment the contents of the PC just as any other instruction does.

(a) Enumerate and describe all the required changes to the data path architecture, and (b) the state diagram.

10.4 Assume we want to add another new instruction to Table 10.6 New Instruction Set. This new instruction is:

NOP; 5 clock cycles doing nothing

The NOP should have a normal instruction fetch and decode cycles. The actual execute cycle should be long enough to use 5 cycles of the clock, including the fetch and decode phases.

Remember the new instruction has to increment the contents of the PC just as any other instruction does. However, the NOP must *not* change the contents of registers A and B and must not affect the ALU flag bit S (sign bit).

(a) Enumerate and describe all the changes to the data path architecture, and the state diagram that apply.

10.5 Create an algorithm that allows one to add two 4-bit unsigned binary numbers and produce the resulting sum in BCD.

For example: 1001 + 0001 = 1010, binary nine plus binary one equals binary ten. Your algorithm should report the sum as: 0001_0000, which stands for ten in binary coded decimal (BCD). Similarly 0110 + 0101 = 1011 should be reported as 0001_0001 (eleven in BCD). And a last example, 0011 + 0010 = 0101, binary 3 plus binary 2 equals equals five in BCD.

10.6 Design the logic hardware to implement the algorithm found in Problem 10.5. Draw the circuit schematics of the logic.

10.7 Let us assume that we want to design a MOVE instruction that copies the contents of a memory location whose address is X, into another memory location whose address is Y. Such instruction must not affect ALU flags, it must increment the PC just as any other instruction does. The original (and likely unknown) contents of memory location Y are overwritten with the data copied from address X. Original contents of memory location X remain unchanged. The syntax for such new instruction is:

$$\text{MOVE (Y), (X); (Y)} \leftarrow \text{(X)}$$

(a) Enumerate and describe any (and all) required changes to the data path architecture, (b) instruction word format (IWF), and (c) the state diagram.

For part (a) draw and show needed changes to the data path architecture of Figure 10.6. For part (b) generate the complete state diagram for the new instruction only.

10.8 For the circuit schematics of Figure 10.13 assume the following timing parameters:

D-type Flip-flop:

$$t_{CLK\text{-}to\text{-}QMAX} = 2 \text{ ns}$$

$$t_{CLK\text{-}to\text{-}Qmin} = 1 \text{ ns}$$

$$t_{SU} = 1.5 \text{ ns}$$

$$t_{H} = 0.5 \text{ ns}$$

The circuit represents a hardwired shift register without external reset line or controls.

Assume that the wires are ideal and have no delays. Assume that the clock arrives at all flip-flop clock inputs at the same time, no clock skew.

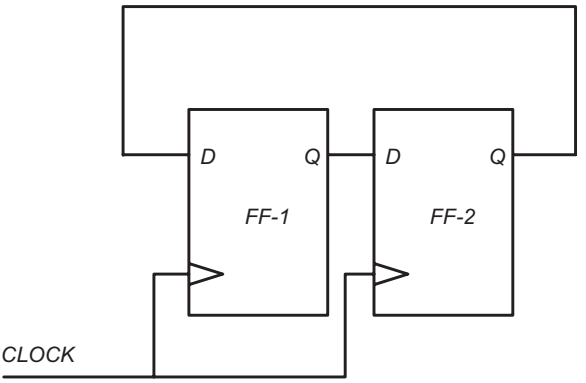


Figure 10.13 Circuit for Problem 10.8.

(a) Determine the maximum frequency at which the shift register can be reliably clocked.

(b) Determine the available hold time (short path) available to each flip-flop.

(c) Assume that flip-flop 1 clock has a rising edge at time 0 ns and flip-flop 2 receives the same logically rising edge 1 ns later. Determine the available hold time (short path) available to each flip-flop. (This point and the next one no longer assume zero clock-skew.)

(d) Draw the following waveforms: clock 1, clock 2, FF1 input data, FF1 Q output, FF2 input data, and FF2 Q output, showing clock skews.