Ch 7

Designing a Minicomputer

You are ready to tackle a really substantial project to round out your study of hardwired design. Nothing will sharpen your design skills more than wading through the design of a complex project from start to finish. Thus far, you have studied pieces of the design process; in the next three chapters we will help you forge your knowledge into an integrated and workable design tool. What project should we choose? Such an undertaking should be detailed yet elegant, large yet not too large. Let's design a computer!

Immediately we are faced with dillemas, some unexpected. What to build? Surely a modern processor embodying cuurent architectural concepts is a prime candidate, or is it? We desire a project simple enough to be completed in an elementary course, a serious constraint which rules out likely candidates, in fact all current CPU's. This is unfortunate but we have not found a way around it.

Further, all modern processors are meant to be *thrown away when defective* whereas ancient computers were expensive and intended for field repair with tools called diagnostic programs to assist field engineers in detecting indivdual failed components for replacement. Diagnostics are an indispensible tool as you build and test your system and trying to build even a simple project without their support borders on cruel and unusual punishment. This immediately takes us back to old times when these tools were commonplace and an integral part of any computer installation.

This is not necessisarily a show stopper, back up for a minute and reflect on this book's subtitle, "Algorithms in Silicon"; computers are simply a common target for hardware implementation and we are interested in the process, the target is secondary so let's pick the simplest one we can find.

The exposition in this chapter is not a mere "paper design"; real working hardware has been built following the design path outlined here. We strongly encourage you to follow through and build it either on a FPGA or on a simulator as outlined in one of the appendices; there is no better way of solidfying design concepts than putting them to work and verifying your construction.

Our aim is to design an entire operational, real, computer system, taking no shortcuts, leaving nothing out. Toy computers can be simple but less satisfying. Immediately we are faced with a conflicting set of requirements. Most computers, even the smallest microcomputers, are highly complex structures—too complex to be a suitable teaching illustration at the gate level of design. Instead, we choose the first minicomputer, the Digital Equipment Corporation PDP-8.

The PDP-8 has had a successful history. More than 50,000 units have been installed, some of which are still in use. The PDP-8 also has an extensive library of software and is a good machine for illustrating device interfacing.

The great advantage of the PDP-8 for our purpose is that it has a simple structure with only eight basic instructions. It exists in several models; each executes the same basic set of instructions, but they differ in minor ways. We will use the PDP-8I as the basis for our exercise. We will develop our design from first principles and make no reference to the Digital Equipment Corporation's design. The result will be functionally equivalent to the PDP-8I—for example, it will run PDP-8I software—but we will use top-down design techniques. The only detailed information we need about the PDP-8I is a description of the action of each instruction.

Let us list the pro's and con's of the PDP-8 as our first project:

Pro's

It is a real computer

It has a dirt simple data path

Its instruction set is simple yet powerful enough for real programming

There is a complete suite of diagnostic programs so you can test your simulated machine

There is an amazing library of programs, chess, LISP, an interpreter similar to BASIC, and a host of other applications (including multi user time sharing, all in a 4k memory!)

There are several PDP-8simulators on the internet with a complete set of utilities, assemblers, etc. Some are so good they give the illusion you are running on a real PDP-8

It is a historically important machine

It is simple enough for a first time student to simulate on any of a number of cheap commercial simulators

Con's

It is a historically important machine, but now extinct

It is a dirt simple computer without many of the bells and whistles of modern machines. In spite of its overall simplicity it does have a rather involvrd fetch cycle.

Its addressing modes are severely constrained by its 12-bit word length. This forces some rather quaint (ugly?) modes that modern computers bypass with their wider word lengths.

Modern features like stacks, varied indexing modes, register files, vectored interrupts, pipelining, memory hierarchy, and protection schemes are absent.

In our experience there is only one machine suitable for a first introduction, the PDP-8. Later we will go through the same process for a thoroughly modern computer and the more adventuresome may wish to skip the PDP-8. However, it has been our experience that a beginner can do both machines in sequence in less time than skipping the '8 and starting off with the more complex example. The statement of the problem is brief: build a computer that will execute the PDP-8I instruction set.

PART 1

PDP-81 SPECIFICATIONS and INSTRUCTION SET

The first step is the obvious one of studying the PDP-8I to see what we must emulate. The major characteristics of the PDP-8I are:

(a) A *12-bit word size.* This is quite small and will cause memory-addressing limitations. If a memory word is used to hold an address, it can refer to only 4096 (2^{12}) different locations. Therefore, the standard PDP-8 is limited to 4096 words of addressable memory.

(b) A *single accumulator.* Several instructions refer to an accumulator (AC), used to store intermediate results for later manipulation. Having only one accessible register forces a programmer to use care in saving and restoring vital data in the AC, for example upon subroutine entry and exit. Many computers use register files, which can speed the execution of programs but which expose the programmer to subtle bugs if the data in all registers is not properly handled. In many applications the single AC is a blessing!

(c) A *3-bit operation code.* Each instruction occupies a 12-bit word, of which 3 bits are devoted to the operation code. This provides eight basic commands—an adequate but hardly abundant number. Only 9 bits remain in the instruction for such purposes as addressing memory, whereas the 4096-word memory requires a full 12-bit address.

(d) *Paging.* Addressing limitations in minicomputers and microcomputers have forced computer architects to find a number of ingenious solutions. The PDP-8's method is based on memory pages of 128 (2^7) words. The 4096-word address space is divided into 32 pages, and each memory-referencing instruction has 7 bits to address a word within a page. The missing 5 bits of the address are not a part of the instruction, but are derived implicitly from the context. Without some trick of this sort there would be no way to pack a 3-bit command and an address into a 12-bit word. Maneuvers such as this were common features of early minicomputers. The paging mechanism of the PDP-8 is perhaps the simplest technique and serves as a foundation for studying more complicated schemes used in other

computers.

Throughout this design exercise, we will use the octal numbering system to specify particular values of the PDP-8's instructions, addresses, and so on.

Any such numbers not in octal will have an explicitly designated base. Thus 305 is 305 octal, 1011_2 is 1011 binary, and 42_{10} is 42 decimal.

PDP-8I Instructions

All computers go through a fetch-execute sequence. It is the function of fetch to get and decode instructions and prepare operands for the execute cycle. This may be simple or complex depending on the instruction's addressing mode, but at the end of fetch either an address, the EA (Effective Address), or an operand, the EO (Effective Operand) will be presented to the execute apparatus for processing by the instruction that was decoded during fetch.

Branches, or jumps, only need an EA, the target location where the jump tells the CPU to find its next instruction. Data operations, such as an ADD, on the other hand require a value, the EO. For example an add instruction will add the EO to a memory location, register, or accumulator.

Splitting the computer into fetch and execute sections is an example of divide and conquer, a powerful tool for managing complexity. Divide a complex task into smaller units that can be easily understood and debugged, then assemble the subunits into a finished product; a paradigm used in both software and hardware. After fetch, execute will have the data it needs without needing to know how fetch got it, and similarly, fetch can go about its business of getting instructions and operands without knowing how the execute apparatus will process them

An instruction set characterizes a computer, and therefore we must carefully study the PDP-8I's instructions. The effective address EA points to a memory location whose contents are the EO, the effective operand. EA and EO notations allow a compact description of memony-referencing instructions. A compact notation for Memory addressed by EA is MEM(EA), thus EO=MEM(EA)

AND (Twelve-bit logical AND). Operation code $000_2 = 0_8$.

$$AC \bullet EO \rightarrow AC$$

This is a bit-by-bit AND of the AC with the effective address contents. For example,

AC	=	$001\ 101\ 111\ 000_2$
EO	=	$110\ 111\ 101\ 100_2$
AC•EO	=	$000\ 101\ 101\ 000_2$

The value of AC•EO replaces the old contents of the AC.

TAD (Two's-complement add). Operation code $001_2 = 1_8$

$$AC(+)EO \rightarrow AC$$

The addition is performed in the two's-complement mode; that is, the instruction implies that the numbers are 12-bit signed quantities represented in the two's-complement notation. If carry out occurs, the CPU toggles (complements) a special flag called the link bit (LINK).

ISZ (Increment and skip if 0).). Operation code $010_2 = 2_8$

EO (+) 1 → MEM(EA) **if** EO (+) 1 = 0 **then** skip the next instruction
else execute the next instruction

This instruction is useful in controlling loop execution.

DCA (Deposit and clear AC). Operation code $011_2 = 3_8$

$$AC \rightarrow MEM(EA);\ \text{then}\ 0 \rightarrow AC$$

The contents of the AC goes into the specified memory location, then the AC is set to 0.

JMP (Jump). Operation code $101_2 = 5_8$

Jump to location with address EA for the next instruction.

JMS (Jump to subroutine Operation code $100_2 = 4_8$

Store the address of the word following the JMS instruction (i.e., the return location) in the memory word with address EA. Then jump to the location with address EA (+) 1 for the next instruction.

The return location is the word after the JMS instruction. This instruction stores the return address in the first word of the subroutine and then jumps to the second word, which must contain the starting instruction for the subroutine. The normal entry to a subroutine X is thus with a JMS X, which saves the return address in location X. The normal exit from the subroutine is with a JMP *X (indirect jump through location X).

OP (Operate). Operation code $111_2 = 7_8$.

This is by far the most complex command in the PDP-8. It does not reference memory, so the address field bits are available for other purposes. The Operate instruction permits the following basic actions:

Clear accumulator: 0 → AC

Clear link bit: 0 → L I N K

Complement accumulator: $\overline{AC}$ → AC

Complement link bit: $\overline{LINK}$ → L I N K

Increment accumulator: AC (+) 1 → AC

Rotate the concatenated accumulator and link bit right or left, 1 or 2 bit positions.

OR console switches with AC SR + AC→AC

Skip on various conditions of the accumulator or link bit.

Halt the computer.

Each of these operations is controlled by 1 or more bits in the address field of the instruction. These are sometimes called *microcoded instructions* on *microinstructions*. The programmer may invoke combinations of these microinstructions within one Operate instruction. There is a huge number of possible combinations; about 20 of these are useful to the programmer. These combinations of microinstructions ease the pinch of having only eight basic instructions in the PDP-8.

The operation code 111_2 occupies bits 11 through 9, as usual. Instruction bits 8 through 0 have individual functions. The Operate instruction on the PDP-8I is split into two groups, group 1 (G1) and group 2 (G2). Bit 8 specifies the group: in group 1, bit 8 = 0; in group 2, bit 8 = 1

The format for group 1 is

11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	CLA	CLL	CMA	CML	RAR	RAL	Rotate twice	IAC
Operate			Group 1								

The meaning of the microcode bits in GI is

Bit	Mnemonic	Name
7	CLA	Clear accumulator
6	CLL	Clear link
5	CMA	Complement accumulator
4	CML	Complement link
3	RAR	Rotate accumulator and link right
2	RAL	Rotate accumulator and link left
1	----	0=one bit rotation; 1=2-bit rotation

0	IAC	Increment accumulator

The format for group 2 is

11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	CLA	SMA	SZA	SNL	Skip sense	OSR	HLT	0
Operate			Group 2								

The meaning of the microcode bits in G2 is

Bit	Mnemonic	Name
7	CLA	Clear accumulator
6	SMA	Skip on minus accumulator
5	SZA	Skip on zero accumulator
4	SNL	Skip on nonzero link
3	Skip sense	(specifies sense of skips; see discussion)
2	OSR	OR switch register into accumulator
1	HLT	Halt the computer

(In group 2 micro-operations, bit 0 is 0. On the PDP-8I, the condition of bit 0 is irrelevant, but some other models of the PDP-8 computer have another set of microinstructions, group 3, identified by bits 8 and 0, both of which are set to 1.)

To find the exact result of combining microinstructions, we must define the sequence in which the operations of each group occur. The PDP-8 describes the sequence in terms of *priorities.* There are four priority levels, I through 4: priority 1 operations occur before priority 2, and so on. The priority sequences of the micro-operations of G 1 and G2 are:

Priority	Group 1	Group 2
1	CLA CLL	Skips
2	CMA CML	CLA
3	IAC	OSR HLT
4	Rotates	

The group 2 "skip" microinstructions require further explanation. There are three conditions for skipping: SMA, SZA, and SNL. Bit 8 determines the skip mode. The operations are as follows:

If (bit 8 = 0) **then** a skip occurs if *any* of the chosen conditions is satisfied
else no skip occurs.

If (bit 8 = 1**) then** *no* skip occurs if *any* of the chosen conditions is satisfied
else a skip occurs.

IOT (input-output transfer). Operation code $110_2 = 6_8$ The PDP-8 has a primitive but adequate facility for the input and output of data. We will discuss the IOT instruction more thoroughly later; but now we will note how data enters and leaves the computer. Outgoing data (from the PDP-8 to the external world) comes from the AC. Incoming data reaches the AC by being ORed with the existing contents of the AC. There is a programmable facility for clearing the AC prior to accepting incoming data. Thus the basic input operations are

$$0 \rightarrow AC \text{ (optional)}$$
$$\text{Input .Data} + AC \rightarrow AC$$

The IOT instruction also permits the programmer to enable and disable the PDP-8's interrupt system. These IOT subcommands are ION (Interrupt System On) and IOF (Interrupt System Off), and have instruction bit patterns 6001_8 and 6002_8, respectively. The presence of interrupt commands alerts us to the need to investigate the interrupt mechanism.

Interrupts. (Voltage driven subroutine calls)

The PDP-8 specification requires that the machine be able to sense the presence of an external interrupt

request. This request originates in some peripheral device and means that the device wishes to report an event of interest to the computer program. Any number of devices can request interrupt processing through this one external interrupt request line. When the PDP-8's interrupt system is activated, the computer monitors the interrupt request signal to see if any device needs servicing. If so, then at an appropriate time in the normal instruction processing cycle, the PDP-8 will force an automatic subroutine jump (JMS) to a fixed memory location (cell 0000). It is the programmer's responsibility to see that a valid subprogram for processing interrupts begins at location 0000. This subroutine is responsible for reading data from the peripheral device, writing data, or perhaps placing control information into the device. The characteristics of the device generating the interrupt determine what the interrupt subprogram must do. Therefore, the interrupt subprogram must determine which device is responsible for the interrupt and then perform actions tailored to that device. After servicing the interrupt, the subprogram will make a normal subroutine return through cell 0000 and processing of regular instructions will resume.

Interrupt requests originate from external devices running at their own pace, and may interrupt the program at any time. This is both a blessing and a curse to the programmer. Interrupt requests can occur whenever a peripheral device decides it needs service from the main computer. This is a potent programming tool, since the computer program need not waste time continually checking its peripheral devices to see if one needs service.

Interrupts are powerful; they are also tricky. The difficulty arises because interrupt requests originate from external devices and are therefore not reproducible. An interrupt may occur when the resident computer program is not prepared to handle it. For instance, suppose that the programmer has not established an interrupt service routine beginning at memory location 0000. Then the program will not run correctly if the computer recognizes an interrupt and jumps to location 0000. Even if the interrupt service program is present, it may not properly treat all the interrupt requests that may arise. These problems are difficult to diagnose, since the debugger of the program cannot reproduce the exact sequence of instructions that led to the difficulty. Interrupt programming requires much more foresight and care than conventional programming.

To allow more control over this difficult programming task, computers with interrupts always allow the programmer to *enable* (turn on) and *disable* (turn off) the computer's interrupt detection apparatus. The programmer may select those times when interrupt requests may result in the interruption of the program. Some computers permit the handling of several types of interrupts, each type having its own interrupt jump location. We will not pursue this subject, because our focus is on the PDP-8's interrupt capabilities.

The PDP-8 programmer may enable or disable the recognition of interrupts by using the ION (Interrupt System On) and IOF (Interrupt System Off) sub-commands of the IOT instruction. The PDP-8's hardware will automatically disable the interrupt system whenever an interrupt causes a jump to location 0000. This action is needed to give the programmer's interrupt service routine enough time to react to one interrupt without the danger of another interrupt occurring in the middle of the processing of the first interrupt. It is the programmer's responsibility to enable the interrupt system again at the proper time, to permit the detection of further interrupts. This gives rise to a subtle problem. The interrupt subroutine will normally leave the interrupt system disabled until it is time to return to the main (interrupted) program. At this time the interrupt subprogram must enable the interrupt system and return. The last two instructions of the subprogram are:

```
        --.
        --.
        --.
ION     (Turn on interrupt system)
JMP *0  (Indirect jump to point of interruption)
```

We must make sure that we can execute the return jump to get back to the main program. Consider what would happen if an interrupt request is pending at the time the ION command is executed. The ION would re-enable the interrupt system and the computer would immediately jump again to location 0000 *without* executing the jump instruction after the ION. The interrupt-forced JMS 0 causes cell 0000 to receive the address of the point of interruption—the address following the ION in this example. This act destroys the old return address in location 0000 which the unexecuted JMP *0 instruction wanted to use. The PDP-8's solution to this dilemma is to inhibit the recognition of interrupt requests for one instruction following an

ION command, thus allowing the program the time to execute the crucial JMP *0 to return to the interrupted program before the computer recognizes any additional interrupt requests. Interrupts are a complex feature of computers, and they place a heavy responsibility on the programmer. Whether or not the programmer does the job correctly, the computer must faithfully perform its assigned duty of detecting interrupt requests and forcing subroutine jumps to location 0000 whenever the interrupt system is enabled.

PDP-8 Memory Addressing

In many memory addressing schemes for small instructions, the location of the current instruction is used to specify part of the operand address. For example, assume a program with five instructions stored sequentially, starting at location 300. Call these instructions CM0 (command 0) through CM4 (command 4). A memory map of this program would be:

Location	Contents
300	CM0
301	CM1
302	CM2
303	CM3
304	CM4

If instruction CM3 is being executed, we know that it is located at address 303, since that is where we placed it. Instruction CM3 can employ a subset of the 12 bits in its word to reference data located close to location 303. In the PDP-8, "close to" means in the same page.

The PDP-8 splits 4096 words of memory into 32 pages of 128 words each, as shown in Fig. 7-1. Instruction CM3 is in page 1; 7 bits are sufficient for that instruction to access any word in that page.

Page 0	0-177
Page 1	200-377
Page2	400-577
	. . . . .
Page 31_{10}	7600-7777

All address are octal
Each page contains
$2^7=128_{10}=200_8$ words

Figure 7-1 Page structure of PDP8 memory

We now have a mechanism such that an instruction needs only 7 bits to access a memory cell in one particular page. Let us call these 7 bits the *page offset,* and let the page offset occupy the rightmost 7 bits of a PDP-8 instruction:

Op code 3 bits	Uncommitted 2 bits	Page offset 7 bits

Suppose location 301 contains the 12 bits 001 *XY1* 000 101_2, (for the moment we will ignore the 2 bits *X* and *Y).* The operation code is 001_2, which means an addition of the AC and the contents of a memory location. Which location? The 7 page-offset bits are 1 000 101_2 = 105_8. The instruction is to add the contents of location 105 in this page (the page containing the add instruction) to the accumulator. We know that the instruction is at location 301, and since the instruction is in page 1, the page offset is referring to page 1. Thus we will get the contents of word 105 in page 1 and add it to the AC.

What if instructions in different pages require the same data? It would be nice if some common page could be accessed by instructions in any page. In the PDP-8, page 0 has this function. We have two precious unused bits in the instruction, and we need one of them to tell if we want word 105 in the current page (page 1 in our example) or word 105 in the common page (page 0). In the PDP-8, the *Y* bit is used for this page selection; we call it the *page bit.*

If the page bit is 1, the page address of the current instruction is concatenated with the 7-bit offset in the instruction to form a full 12-bit address, which is sufficient to identify any word of the 4096-word memory. If the page bit is 0, the reference will be to a word in page 0 of the memory.

If we execute an instruction at location 301 that contains 001 011 000 101_2, we will add the contents of location 105 in page 1 to the *AC*. Location 105 in page 1 is memory location 305

000 01	1 000 101	=	305_8
Page address	Page offset		

If location 301 contains 001 001 000 101_2, the instruction would mean to add the contents of location 105 in page 0 to the AC. Location 105 in page 0 is memory location 105.

Indirect addressing. We have shown how the page bit and the page offset combine to yield an address either in page 0 or in the current instruction page. What happens if a command in page 2 needs to access a location in page 7? We must use all 12 bits of a word as address bits. We can do this if the word accessed by an instruction is treated not as an operand but as the *address* of an operand. This extra step is called *indirect addressing*. The PDP-8 uses the remaining instruction bit *X* as the *indirect bit* to specify indirect addressing. The complete format of a memory referencing instruction is

Op code 3 bits	**Indirect** bit	**Page** bit	**Page** offset 7 bits

In the previous examples, the contents of locations 305 or 105 (for page bits 1 or 0) were treated as 12-bit *data* words. If the indirect bit is on, these contents are treated as 12-bit *addresses of* data. We require one extra memory cycle to access this final indirectly addressed data location.

Indirect addressing is a powerful concept since it provides a way to specify arbitrary 12-bit addresses. Into some memory word *IND* that is close to our instruction or in page 0, we load the address *of* the final location that we wish to access. We can then access the location by indirectly addressing it through *IND*.

It is useful to have a shorthand for the final memory location reference in an instruction after all applicable paging and indirect addressing are invoked We call this final location the effective address, EA. The contents of location EA is called the contents of the effective address, EO. (EO is sometimes called the effective operand.) Using EA and EO, we can compactly describe the memory references of any PDP-8 instruction.

Examples of memory addressing. Here are some examples of referencing memory using PDP-8 instructions. The addresses will have 12 bits, since the PDP-8 has 4096 words of memory. We refer to the contents of an addressed memory location by enclosing the address in parentheses: If location 0301 contains 0305, then (0301) = 0305. Note that (EA) = EO.

Now assume that the following memory locations have been loaded with the data shown:

(0301) = 1305	(0305) = 1234
(0302) = 1105	(0105) = 4321
(0303) = 1705	(1234) = 5567
(0304) = 1505	(4321) = 7765

(a) What are the EA and the EO for the instruction located at 0301? **EX1**

(0301)=1305

1305 =	001	0	1	$1\ 000\ 101_2$
	TAD	Indirect bit	Page bit	Page offset =105

EA: concatenate page address with the page offset when the page bit=1
(symbol for concatenate operator is ∩)

EA =	000 01	∩	$1\ 000\ 101_2$	= 0305
	Page address		Page offset	

EO = contents of memory at address 0305 (EO =1234)
This instruction would add the quantity 1234 to the contents of the AC.

(b) What are the EA and the EO for the instruction located at 0302? **EX2**

(0302) = 1105 = 001 0 0 $1\ 000\ 101_2$
TAD Indirect bit Page bit Page offset =105

EA = 0105
EO = (0105) = 4321

This instruction would add the quantity 4321 to the contents of the AC.

(c) What are the EA and the EO for the instruction located at 0303? **EX3**

(0303) = 1705 = 001 1 1 $1\ 000\ 101_2$
TAD Indirect bit Page bit Page offset =105

EA = (0305) = 1234
EO = (1234) = 5567

This instruction would add the quantity 5567 to the contents of the AC.

(d) What are the EA and the EO for the instruction located at 0304? **EX4**

(0304) = 1505 = 001 1 0 $1\ 000\ 101_2$
TAD Indirect bit Page bit Page offset =105

EA = (0105) = 4321
EO = (4321) = 7765

This instruction would add the quantity 7765 to the contents of the AC.

Auto indexing. The PDP-8 has a feature called *auto indexing* that provides some flexibility in addressing. Most large computers have index registers to facilitate array access. Unfortunately, specifying an index register takes one or more bits of the instruction and we have no bits left. The PDP-8's auto indexing is a primitive way to index without using instruction bits. An *auto index register* is a word in memory that will automatically increment every time it is used as the source of an indirect address. The word is incremented before it is used as an address. Repeated use of the same auto index register will sequence the effective address, EA, throughout the full memory address space. There are 8 auto index registers in PDP-8 main memory, locations 10_8 through 17_8. When not performing auto indexing, these locations behave like normal memory words.

Here are some examples of auto indexing. Assume the following locations have the contents shown:

(0013) = 4102
(4102) = 1111
(4103) = 2000

(a) **EX5**

Instruction = 1013 = 001 0 0 $0\ 001\ 011_2$
TAD Indirect bit Page bit

EA = 0013
EO = (0013) = 4102

Although location 0013 is the address, there is no auto indexing because the indirect bit is 0. This instruction adds the quantity 4102 to the contents of the AC.

(b) **EX6**

Instruction = 1413 = 001 1 0 $0\ 001\ 011_2$
TAD Indirect bit Page bit

EA = 0013

The initial address is 0013. This is an auto index location used as an indirect address. The auto indexing feature causes

(0013) (+) 1 → (0013) or
4102 (+) 1 → (0013)

Then

EA = 4103
EO = 2000

The effect of executing this instruction is to increment the contents of location 0013 by 1, and to add the quantity 2000 to the contents of the AC.

Whew! Frankly, this is pretty ugly and the EA,EO gyrations of the PDP8 are due solely to its 12-bit memory limitation. Modern giga-byte RAMs cost only a few dollars but you can't project that mind set back to the PDP8 era where memory was enormously expensive and dictated CPU architectures of the day. Regardless, all processors, old or modern, break down instruction execution into a fetch phase which is responsible for generating an EA and EO and the fetch ASM will have to provide them to the execute apparatus. (Much of the driving force behind modern RISC architectures is dedicated to minimizing the time required to generate EA and EO but that's a subject you will explore in detail later in a course in computer architecture).

PART 2

PRELIMINARY DATA PATH ARCHITECTURE

The instruction set already tells us a good deal about the machine's major architectureal elements and we can embark on a preliminary specification before formulating an ASM. In order to be consistent with the principles set forth in Chapter 5, we plan for our design to be synchronous and static. Further, since this is a pedagogical exercise for the budding designer, we should strive to develop good design habits. All major building blocks should be "out in the open." We wish to use building blocks of the right scale for our example. We could build everything from transistors or from AND, OR, and NOT circuit elements, as the original designers were forced to do, but such approaches are hopelessly outdated and would not teach you how to use higher-level building blocks, such as those discussed in Chapter 3. We will rigorously follow the 8I's instruction set but there is no reason to restrict ourselves to the original rather primitive control panel so we will take some liberties there.

The major building blocks are:

- **Memory (MEM):** At this point we can't specify more, it could be any variety of RAM and we are free to choose the type but it must conform to the 12-bit address limitation, ie, it will be 4k x 12. As discussed in Chaper 3, we can assume that it will be accompanied by a 12-bit address register and a 12-bit data register to hold the data to be written into memory or read from memory. Traditionally, the address register is called the **MA** (memory address register) and the data register the **MB** (memory buffer register).
- **Program Counter (PC):** Although not explicitly specified by the instruction set, every computer needs some way to specify the memory location of the current instruction in memory; without further ado we can assume our machine wil contain a **PC** and its size, 12-bits, will be determined by the 4k x 12 memory.
- **Instruction Register (IR):** Although not explicitly specified by the instruction set, every computer needs some way to hold an instruction for decoding and execution.
- **Accumulator (AC)**: A 12-bit register to hold the results of arithmetic and logical operations. Although, at this point, we might suspect that the AC could be used to accomplish some of the instruction set's operations we must emphasize that this decision should be postponed until after we have explored the machines's ASM. Never restrict your options before you have to.
- **Link**: A 1-bit register to handle overflow from the AC. We must be able to SET, CLEAR, and TOGGLE the Link as well as load bits shifted out from the AC.

- **A control panel:** The instruction set assumes there is a means of OR'ing a 12-bit manual operand (SR, the switch register) to the AC. Most machines also use the control panel to start and stop execution and we can assume this functionality even though it is not explicitly laid out in the machine specification. Starting the machine also implies that we know where in program memory we should start.; this implies that the PC can be initialized from the SR. If you were to build a true hardware implementation rather than a simulation you would certainly wish to display all the registers on a liquid crystal display, alternately, you might want to be more modern and replace the control panel with a conventional keyboard/monitor setup, but that is left as an exercise.
- **ALU:** machine must be able to do the following arithmetic and logical operations:

  (a) 12-bit logical AND (AC) • EO → AC

  (b) 12-bit PLUS (AC) (+) EO → AC

  (c) Increment AC and PC AC (+) 1→ AC is part of the microcoded instructions. PC (+) 1 → PC is part of normal instruction sequencing as well as the conditional skip in the ISZ command

  (d) 12-bit logical OR **(**SR) + (AC) → AC

  (e) 12-bit logocal NOT (1's complement)

  (f) Register Shifts, Clears, and Loads. For bookkeeping purposes we list these operations under the ALU heading but suspect that we may want to offload these operations to the target registers themselves although we may want to rout the data through the ALU.

Data Paths

We have identified major elements of the PDP8's architecture by looking at the PDP-8's functions. Now that we have this set of elements, how do we put them together? In true top-down spirit, we will leave them scattered about on the desk and back off far enough to ask a question. How does the PDP-8 instruction set say they should be connected? Contemplating this question will lead us surprisingly close to the final architecture. We assume that the ALU has two 12-bit input paths and that it will handle all logical and arithmetic operations. Let's see how the PDP-8's instructions guide us to a model of the data paths among the building blocks.

(a) IAC, Increment Accumulator: AC (+) 1 → AC. If the ALU does the addition, the AC and the ALU must be connected in a manner similar to Fig. 7-2. (In this and subsequent figures, the data paths are all 12 bits wide, and we call the output of the ALU the ALUBUS.) Execution of the IAC microinstruction must result in setting the ALU control lines to force the ALU to increment the input and place the result on the ALUBUS.

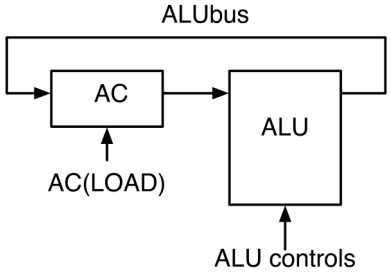


Figure 7-2 Data flow for incrementing the AC

(b) TAD, Two's-Complement Add: AC (+) CA→ AC. This instruction requires an architecture like that in Fig. 7-3. Here the ALU's control lines must make the ALU add its two data input quantities and place the result on the ALUBUS.

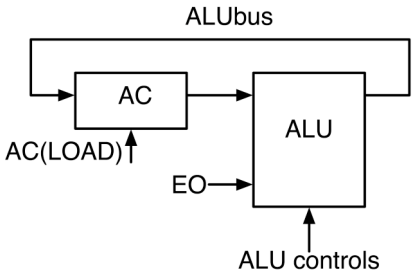


Figure 7-3 Data flow for addition

(c) AND: AC•CA **-4** AC. This instruction again leads to Fig. 7-3, where this time the ALU must perform the bit-by-bit logical AND of its data inputs.

(d) OSR OR Switch Register: AC + SR→ AC. Both the switch register and the AC must be inputs to the ALU, as shown in Fig. 7-4.

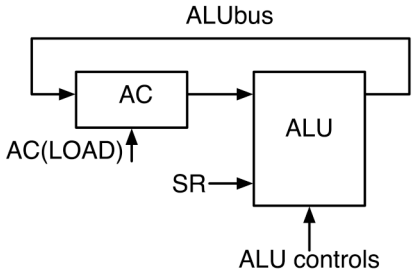


Figure 7-4 Data flow for the OR Switch Register (OSR) operation

(e) Increment the program counter: PC (+) 1 → PC. This operation is implied in any computer, since after one instruction is completed the next one will be executed. This normal sequencing will continue, instruction after instruction, until a programmed branch operation causes the PC to be set to a branch address. Normal (nonbranch) sequencing leads to Fig. 7-5, in which the ALU performs the increment operation.

(f) Load the program counter: This results from the branching case mentioned above and requires the operation EA → PC. At this point we could establish a private data path into the PC so that EA → PC; but look at the requirements of operations (a) through (e) that are leading to a common architecture. Examining those cases, we may reason as follows

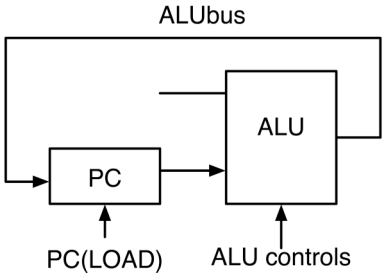


Figure 7-5 Data flow for incrementing the program counter (PC)

Developing the main bus structure. The AC seems to have pinned down one of the two data inputs that are a part of the ALU building block, let's arbitrarily call it the "B" input. The other input comes from a variety of sources: the contents of the effective address EO, the switch register SR, and the program counter PC, so we require a way of selecting the proper input to the ALU, lets call it the "A" input. Recall the discussions of data busing in Chapters 3 and 4. One good way to route several data sources to an output is to have three-state buffers on each source output. Enabling one of the source buffers lets that source "talk" to the ALU, as shown in Fig. 7-6. (Remember, all data paths are 12 bits wide.) This is an economical way to route the data. Some register integrated circuits include three-state output control; using such chips might eliminate

some of the separate buffers in Fig. 7-6.

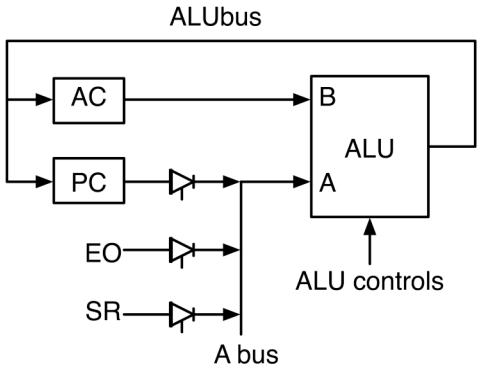


Figure 7-6 Controlling the ALU input with 3-state buffers

Another approach to the selection of the ALU's input is to use the multiplexer building block to *select* the appropriate input (see Fig. 7-7).

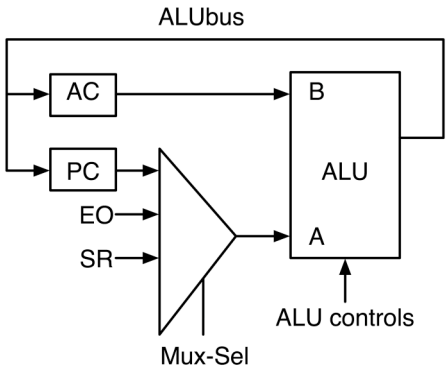


Figure 7-7 Selecting ALU inputs with a mux

There is one multiplexer for each bit of the 12-bit data path. We now have two choices for busing the ALU input. Using tri-state buffers or mux's to put sources on a bus is technology dependent, and also a matter of personal style. Using mux's guarantees that there is only one "talker", but at the cost of having to generate mux select controls. Tri-state buffers require care to avoid the multiple talker problem. On the other hand tri-state enables can be read directly from the ASM chart, a trivial task. Our strong personal preference is to use tri-state and our exposition of the PDP8 data path will be based on tri-state protocols.

Our study of PDP-8 operations has led to this architecture. We may ask if the structure could be generalized to handle the case EA→ PC, which arises when the PC register is loaded with a branch address. The answer is, of course, yes: we add an EA tri-state buffer to the ALU input bus. To execute the branch operation, we must set tri-state enable to select EA, and must set the ALU controls to pass this ALU input unchanged to the ALUBUS. If a load signal to the PC accompanies these operations, we would perform EA→ PC. It is significant that we have accomplished the goal by generalization rather than specialization.

This is a discovery that we should try to carry as far as we can. We now have a mechanism for moving data from one register to any other, as well as a way of performing logical or arithmetic operations on the ALU data inputs. These capabilities are nearly all the structure we need to build a computer. Again, let us note that we were led to the architecture in a direct manner by a careful consideration of the fundamental requirements of the PDP-8's instruction set.This is not the way some designers proceed. They start with a structure (a guess, really) based on a certain set of library modules, and then try to bend the design to fit that guess. Sometimes the bending can require enormous leverage, when it could be side-stepped by redefining the architecture. Unfortunately, at that point it is too late; the designer's mind is already in a groove and it is difficult to jolt it out. If you let your target lead you to the architecture, you can avoid this

groove until the last possible moment and save yourself anguish.

Adding memory. Now it is a simple matter to expand the tentative architecture of the PDP8 to handle the remaining data transfers. We need to take care of memory and its interaction with the memory address and memory buffer registers MA and MB. The MA register gives memory the address at which to perform a read or write operation; therefore, MA must be wired directly to the memory. MA must be loaded with address data from various sources (e.g., EA), so we make MA a destination on the ALUBUS. So that data from the memory can reach other destinations, the output of memory connects to the data bus. Memory write operations require input from MB in addition to MA. The MB holds the write data and must be connected directly to the memory. The memory buffer register's input may come from a variety of sources yet to be specified, so we will make MB a destination on the ALUBUS. Figure 7-8 shows the proposed routing of the memory data.

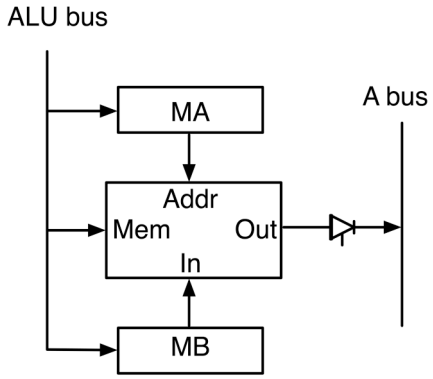


Figure 7-8 Memory data flow

Handling EO. This is a good time to investigate EO, the contents of address EA. EO is not a register; rather, it is a concept that we introduced to aid our understanding of the operations of PDP-8 instructions. Where do values for EO come from? Remember that EO = MEM(EA): EO is the contents of the location referenced by EA. Thus EO comes from the memory. We have just proposed an architecture for reading and writing memory data. If we route an address EA into MA over the existing paths, then a subsequent memory read command will give us EO. Therefore, we may eliminate EO from our A bus inputs, realizing that our recent addition of the memory system incorporates the data movement for EO.

Fetching instructions. Can we use our basic data routing scheme to acquire the next instruction from memory and move it to the instruction register IR? The memory is already a source on the data bus, so we simply make the IR a destination on the ALUBUS. At the time of instruction fetch, our design must select MEM on the A bus, cause the ALU to pass its input without modification, and then cause the IR to load the result from the ALUBUS.

Handling shift operations. Last, we must lay a plan for the data movements for the left and right shift instructions, which involve the AC and the link bit. One approach would be to include the shifting capability in the ALU building block. This would be acceptable design practice. On the other hand, one of our basic modules is the parallel-in parallel-out shift register, which will shift or retain its value or load a new value upon command. Since the shift operations in the PDP-8 involve only the AC and LINK, let's make the AC a shift register in addition to its earlier assignment as a holding register. In this event it appears unlikely that the shift operations will affect the basic routing of the data.

The PDP8 data bus. The initial proposal for the architecture of the data path is complete, and appears in Fig. 7-9. We have derived the structure from first principles based on the requirements of the PDP-8's instructions, and although we have used our knowledge of good building blocks we have not committed ourselves to any particular set of modules. This initial architecture turns out to be very close to the final requirements that will emerge from more detailed study.

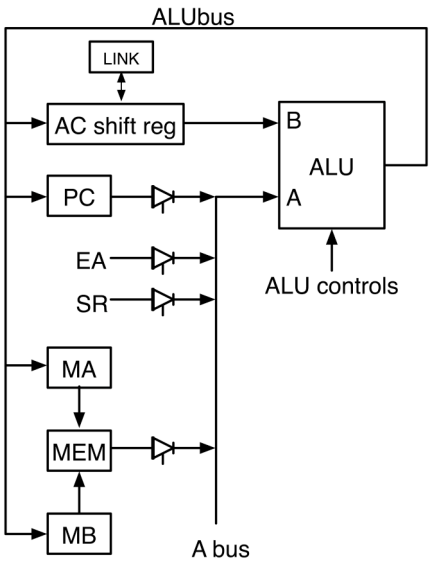


Figure 7-9 An initial proposal for the architecture of the PDP8 data path

It is now time to consider the PDP8's control algorithm. We use our tentative architecture as a framework for developing the control; understanding the control will, in turn, lead us to a refined architecture. Throughout all this, we remain aloof from the actual hardware until we thoroughly understand both the architecture and the control.

Designing an ASM

Our philosophy: the first ASM iteration should focus on correctness, not efficiency. Make an ASM that is transparent as possible, test it and work out the inevitable kinks. If you're not sure you can share a register-don't, just create a new one, above all, keep it simple. After you have a working ASM it is always easy to go back and look for optimizations.

Our first "divide and conquer" step is traditional: we break the complete machine cycle into a fetch phase responsible for getting a new instruction, decoding that instruction, preparing the EA and EO, and then passing that information to the ASM's execute phase. Execute will need the instruction, EA, and EO in standard locations and we can use registers IR, EA, and MB for that purpose. MA works for the EA since we no longer need the MA after the EA is calculated and we can make that register do double duty. Similar arguments apply to EO and MB.

The EA, EO portion of the FETCH ASM:

ASM notation conventions:

Up to now we have used the (REG) notation to signify the contents of REG and this appropriate when discussing the meaning of various instruction operations where precision is important. This gets slightly clumsy in an ASM where compactness is a virtue and we can relax the rigidity as long as we know what we mean. In an ASM, REG on the left side of an arrow means the contents of that REG; on the right side it will mean the name of a destination register.

And lastly, we will deviate from the DEC convention for numbering bit positions. Nearly every modern computer labels the least significant bit (LSB, or right most bit) as BIT_0, whereas DEC labels that as BIT_{11}. This grates on the author's sensibilities and from now on we will use the modern convention where the most significant bit (MSB) will be BIT_{11}. You will not detect any problems unless you delve into original DEC documentation where you have to be aware of the changed convention.

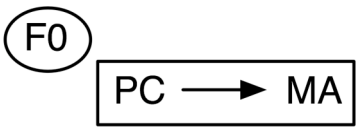
F0, (Fetch State 0, load the MA):

As a starting point, assume we are in the middle of an executing program, bypassing for the moment how we got here. What do we know? Not much really, but enough. The PC points to the next instruction so lets

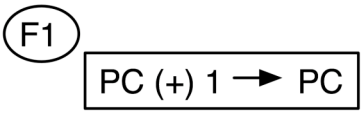
get it and save it in a hidden hardware register called the IR (instruction register). At this point the need for a separate IR register is an assumption that elaboration of the ASM will either justify or refute but for now it is a harmless assumption and a nice aid in organizing our thought processes, (see our philosophy above).

To access memory we will have to load an address, in this case the contents of the PC, into the memory address register and wait for the memory to deliver the contents of that cell. In this case we can use the IR as a destination register. For the PDP8's microscopic memory, (4k x 12), there is no reason to use anything but static RAM with its simple interface. The machines clock speed will be determined by the memory cycle time so we can assume that memory operations will complete by the end of one clock cycle.

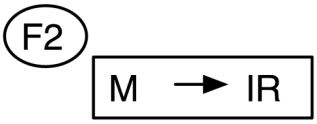
In F0 we present the (PC) to the MA's inputs, wait during F0 for the input to stabilize, then load the MA on the clock edge ending state F0.



F1, In all computers the default instruction sequencing is one-after-the-other unless a JUMP instruction alters flow. So we might as well take care of the default early in the fetch cycle and increment the PC. This is an example of parallel execution, while the ALU is incrementing the PC, the autonomous memory unit automatically starts a read operation when the new address is loaded into MA at the start of F1.

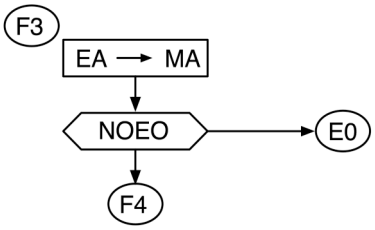


F2, (load the new instruction into the IR)



F3, This is where it starts to get interesting; can you think of cases where fetch has finished its work and no more memory accesses are required? How about opcode7, the OP instruction requires neither an EA or EO. The other cases are instructions that requires an EA but not an EO. For direct JMP, JMS, and DCA the EA is all that is needed. JMP and JMS go to MEM(EA) and DCA stores something in MEM(EA).

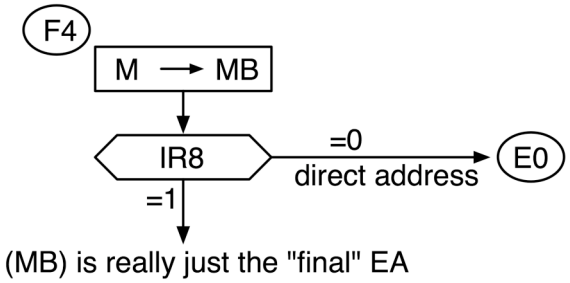
The IR now contains enough information to allow the EA calculation so we can branch to E0 after placing the EA where execute expects to find it (in the MA register). We can lump these cases together as NOEO (No memory access required to retrieve an EO, this will be true at this point in the ASM only for the OP instruction or if JMP, JMS, and DCA are *direct* instructions. (Indirect versions will be handled at F7).



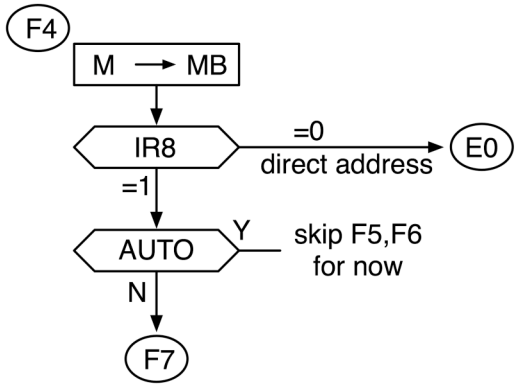
A small alarm bell should go off any time you load a new value into the MA register; this automatically starts a new autonomous memory read, in this case at he end of F3, and if we take the NOEO branch, continuing during the E0 clock cycle. E0 better not be wanting to use the memory during this time so make a mental note to go back and check for this when you design the execute ASM. The same is true on the F4 branch, but in this case this is precisely what we want the memory to be doing.

F4, This is the most complex fetch state. The simplest case is for directly addressed operands where IR8=0. In this case the EA computed in F3 points to the EO and the memory is already busy retrieving it. All we

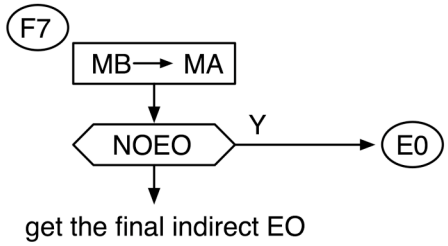
have to do is wait for memory stability, load it into the standard EO register, (which is MB), and branch to E0.



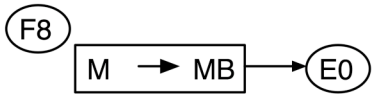
However, if IR8=1 we have an indirect address and have to do it all over again; this is sometimes hard to fathom unless you have programmed in assembler and are familiar with indirect addressing. You may want to go back to the EA,EO examples above. Until you get your mind wrapped around indirect addressing we will ignore auto indexing that is another complexity best revisited later.



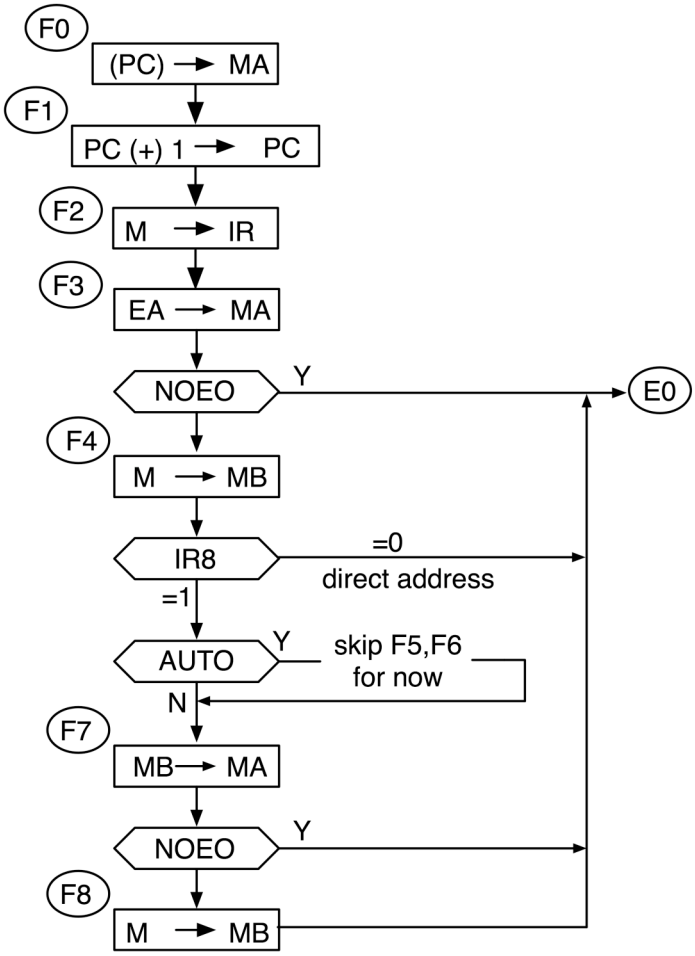
F7, Here we handle the final calculations of EA for indirect addressing. As in F4 we have two cases: those that don't require an EO and those that do.



F8, Get the EO and place it where Execute expects to find the EO.



Putting it all together we get this preliminary ASM:



Experienced designers treasure well defined hardware or software modules that have few connections to their enveloping environment; FETCH fits this bill nicely. There are only 2 inputs, PC and Memory, and 3 outputs, IR EA, and EO. You must *not* accept this preliminary ASM on faith!! Now is the time to debug it before the complexity of the full machine confuses things. It will pay big dividends for you to manually rework examples EX1 – EX4, following the ASM to see if you wind up in E0 with the proper values for EA and EO.

If you are building real hardwware in an FPGA or simulating it (and we hope you are doing one or the other) now is the time to see if you can bring this preliminary FETCH to life with gates, registers, and a primitive ALU, which at this point in our development must be capable of conditionally incrementing its "A" input. (hint: review half adders).

What about F5 and F6? Auto indexing requires us to increment one of the memory locations between 0010_8 and 0017_8 *before* using it as an EA. Further, the incremented value must be written back to the same auto index location.