CHAPTER 4
DESIGN Of A SIMPLE CPU

Section 1. Introduction

In Chapter 1 we studied circuit design at what is commonly referred to as the digital logic (or gate) level. Here the fundamental unit for processing is a binary, digital, electronic signal, and the basic components of circuits are gates. Connections among gates made with individual conducting lines.

As part of our study of digital logic circuits, however, we designed circuits capable of accepting multi-bit inputs, producing multi-bit output, and undergoing multi-bit changes of state. Among some of these circuits are multi-bit gates, multiplexers, decoders, bit-sliced alu's, various forms of registers (parallel in/out, counters, shift registers), etc. These circuits are among the fundamental components for a level of circuit design immediately above the digital logic level in a digital system design hierarchy. In recognition of the omnipresence of registers at this level, it is commonly referred to as the register level (or register-transfer level).

In Chapter 2 we introduced one register-level structure when we introduced main memory as an integrated collection of registers (which we call words or bytes when they are used in reference to memory). In this chapter we take on another register-level structure when we introduce important concepts behind the design of central processing units (CPU's), and incorporate these concepts into the design of a simple processor.

Recall that the primary function of a CPU is to execute programs expressed in the processor's own machine language. During their execution programs and their accompanying data are stored wholly or in part in a main memory M which lies outside the CPU. To actually execute the program the CPU must perform the following actions:

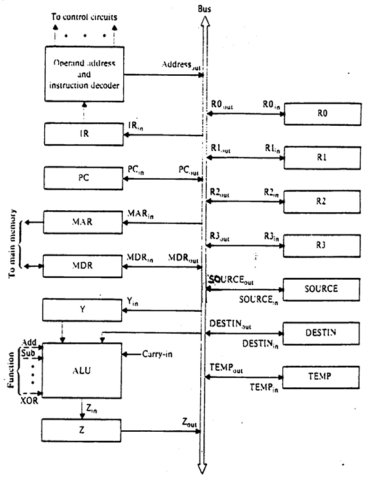
1. Fetch I from M by performing one or more memory read operations.
2. Determine the address in M of the next machine language instruction I (sometimes included in step 1).
3. Decode I to determine the operation to be performed.
4. If the instruction uses a word in memory, determine where it is.
5. If necessary from step 4, retrieve the word from memory and copy it into a CPU register. .
6. Perform the operations specified by I. This may involve writing a word to memory.
7. Go back to step 1 to begin executing the next instruction.

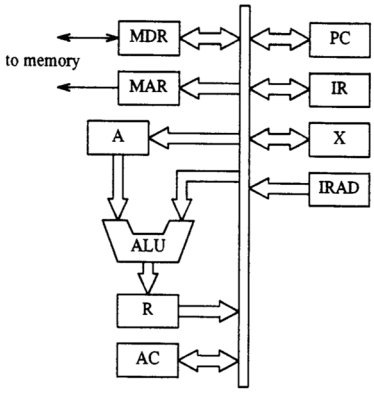
These steps comprise what is known as the *instruction cycle* or *fetch-execute cycle*.

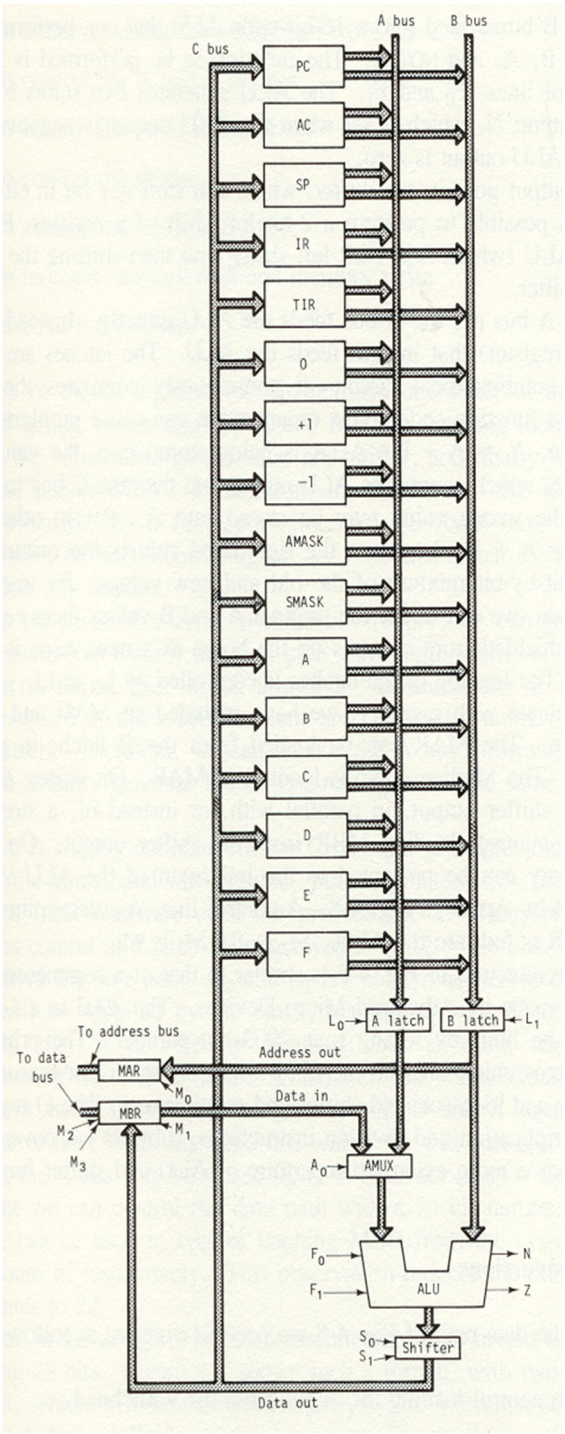
During normal execution of a program the CPU repeatedly goes through the instruction cycle. The circuitry within the CPU to implement this process consists of:

1. An appropriate sized alu.
2. A variety of registers for the temporary storage of addresses, instructions and data.
3. Control circuitry to properly sequence the transfers of data among the CPU's alu and internal registers that are needed to implement the steps of the instruction cycle for each machine language instruction.

The organization used to connect the registers and CPU is often referred to as the *data path* of the CPU. On the next page we show the data paths for some simple, hypothetical, CPU's. In section 2 we shall describe the data path for our own hypothetical CPU and use this CPU as a vehicle for introducing processor organization principles.


To control circuits
Bus
Operand address and instruction decoder
Address out
IR
IR in
PC
PC in
PC out
MAR
MAR in
MDR
MDR in
MDR out
To/from memory
Y
Y in
Add
Sub
AND
OR
XOR
Function
ALU
Carry-in
Z in
Z
Z out
R0 out
R0 in
R0
R1 out
R1 in
R1
R2 out
R2 in
R2
R3 out
R3 in
R3
SOURCE out
SOURCE in
SOURCE
DESTIN out
DESTIN in
DESTIN
TEMP out
TEMP in
TEMP


MDR
to memory
MAR
A
ALU
R
AC
PC
IR
X
IRAD


A bus
B bus
C bus
PC
AC
SP
IR
TIR
0
+1
−1
AMASK
SMASK
A
B
C
D
E
F
L0
A latch
B latch
L1
To address bus
To data bus
MAR
MBR
M0
M1
M2
M3
Address out
Data in
A0
AMUX
F0
F1
ALU
N
Z
S0
S1
Shifter
Data out

Section 2. Data Path for a Hypothetical CPU

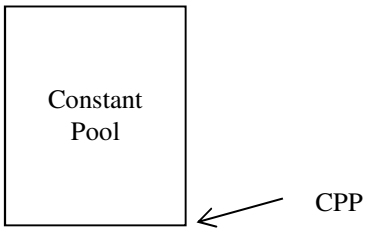
On page 245 of your textbook is the data path for a hypothetical CPU we shall design for this course.

We note that the data path for our CPU uses a triple bus organization with all of the registers allowing only uni-directional transfers either to or from the bus. (Note, one of the buses, bus A, only goes from register H to the alu). The register-level components visible in our data path are:

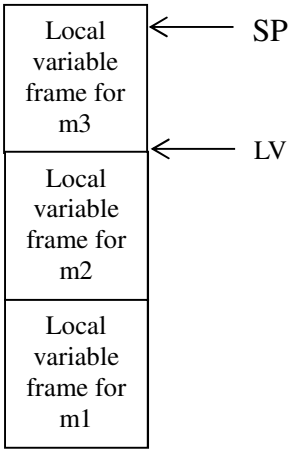
1. *A 32-bit ALU* capable of performing sixteen different functions, as determined by six control lines – F_1, F_0, ENA, ENB, INVA, and INC. A table showing the relationship between control signal values and ALU functions is given on page 246. The ALU also has two output lines N and Z that tell us whether the result of an operation (when regarded as a signed integer) is negative (N = 1) or zero (Z = 1).
2. A holding register (H) that is connected to the left input of the ALU. H is a 32-bit register and is connected to the ALU by a data bus A with 32 data lines.
3. A 32-bit shift register (Shifter) that is connected to the output lines of the ALU. The shift register has two control lines SLL8 and SRA1. When active, the signal SLL8 will cause the contents of the register to shift left by one byte and fill the rightmost 8 bits with zeros; When the control signal SRA1 is active it will cause the register to shifts its bits one bit to the right, but leaves the leftmost bit unchanged.
4. A 32-bit memory address register (MAR) and an-bit program counter (PC) to address values in main memory. The MAR addresses 32-bit words in memory while the PC addresses 8-bit bytes.
5. A 32-bit memory data register (MDR) that contains a 32-bit word that has been read from memory, or which will be written to memory.
6. An 8-bit memory buffer register (MBR) which contains a byte of memory.
7. A 32-bit stack pointer (SP). The purpose of this register will become clear later.
8. A 32-bit local variable register (LV) The purpose of this register will become clear later.
9. A 32-bit top of stack register (TOS). The purpose of this register will become clear later.
10. A 32-bit old program counter register (OPC). The purpose of this register will become clear later.
11. A 32-bit constant pool pointer (CPP). The purpose of this register will become clear later.

We are going to assume that our CPU will implement a small subset—IJVM— of the Java virtual machine (JVM). In addition we assume the following memory model for IJVM:

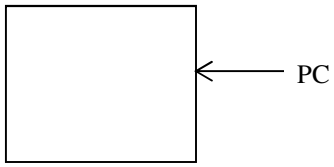
1. ***Constant Pool***: There is an area of main memory known as the "constant pool" that stores all constants, strings, and references to (other) areas of main memory. The register CPP in our data path contains the address of the first *word* of this constant pool.



2. ***Local Variable Frame*****:** Whenever a method of a Java class is invoked, an area known as a *local variable frame* is created for the method's parameters and local variables. The values of any parameters that are passed are stored at the beginning of the local variable frame. The register LV in our data path contains the address of the word that begins the local variable frame of the currently executing method, while the register SP contains the address of the word of the last of the local variables of the currently executing method. Note, as methods invoke other methods the local variable frames of the called methods are stacked atop those of the calling methods, with LV referencing the starting address of the most recently called method. Thus, if method m1 called method m2, which in turn called method m3, then the local variable frames would look like the following:



3. ***Operand Stack*:** The space immediately above that for the local variable frame of the currently executing method is known as the ***operand stack***. The operand stack is used to hold operands during the computation of an arithmetic expression and the value of SP will point at the word immediately before that where the next operand to be placed on the stack will go. We will also assume that at the beginning and end of each machine language instruction that the register TOS will contain a copy of the word being referenced by SP
4. ***Method Area:*** The (byte-addressable) region of memory that contains the code of the program being executed is known as the *method area*. The register PC in our data path stores a byte indicating the address of the program instruction to be addressed next.



We now give the IJVM instruction set (that is, the machine language for the CPU we are designing)

IJVM Instruction Set for Hypothetical CPU			
Op-code (hex)	Mnemonic	Meaning	Notes
10	`BIPUSH` *`byte`*	Push byte onto stack	*`byte`* is 1 byte
59	`DUP`	Copy top word on stack and push onto stack	
A7	`GOTO` *`offset`*	Unconditional branch	*`offset`* is 2 bytes
60	`IADD`	Pop two words from operand stack and push their sum	
7E	`IAND`	Pop two words from operand stack and push their bit-wise AND	
99	`IFEQ` *`offset`*	Pop one word from the operand stack and branch if it is zero	*`offset`* is 2 bytes
9B	`IFLT` *`offset`*	Pop one word from the operand stack and branch if it is less than zero	*`offset`* is 2 bytes
9F	`IF_ICMPEQ` *`offset`*	Pop two words from the operand stack and branch if they are equal	*`offset`* is 2 bytes
84	`IINC` *`varnum const`*	Add a constant to a local variable	*`varnum`* is 1 byte; *`const`* is 1 byte

15	ILOAD *varnum*	Push local variable onto the operand stack	
B6	INVOKEVIRTUAL *disp*	Invoke a method	*disp* is 2 bytes
80	IOR	Pop two words from the operand stack and push their bit-wise OR	
AC	IRETURN	Return from a method with an integer return value	
36	ISTORE *varnum*	Pop a words from the operand stack and store it in a local variable	*varnum* is 1 byte
64	ISUB	Pop two words from the operand stack and push their difference onto the stack	
13	LDC_W *index*	Push a constant from the constant pool onto the operand stack	*index* is 2 bytes
00	NOP	Do nothing	
57	POP	Pop the word on top of the operand stack	
5F	SWAP	Swap the top two words on the operand stack	
C4	WIDE	A prefix instruction; the next instruction has a 16-bit index	

Data Path Timing and Register Transfers for IJVM

In this subsection we develop the data transfers that will be needed to implement each of the IJVM instructions using the given data path. We assume that these data transfers will be coordinated by a clock pulse being transmitted throughout the data path. It is important to know, however, just how much can be done in our data path in a single clock pulse. This is shown in Figure 4-3 on page 248 of your textbook. What is most significant for us at this time about this clock cycle is that it is significantly long to allow us to read from and write to the same register in one clock cycle.

Another point to remember is that it is possible to access memory in two ways from our data path:

- 4-byte words can be read from or written to memory using the MAR and MDR registers. Here MAR contains the address of the word in memory involved in the transaction and MDR either receives the word read or holds the word to be written. The appropriate control signal **read (**or **rd)** or **write** (or **wr)** will indicate the memory operation to perform.

- 1-byte words can be read (only) using the PC and MBR registers in a manner similar to that above. To distinguish a read operation using PC/MBR from one using MAR/MDR, we use the designation **fetch** for a PC/MBR read.

Register Transfer Notation: We complete this section by giving the sequences of registers transfers needed to implement each of the IJVM instructions. In showing these registers transfers we adopt the following notational conventions (this is slightly different from your author's, but I think it makes certain things clearer):

1. D = [S]

   Transfer the content of source register S to destination registers D.

2. D1=D2= …=Dn = [S]

   Transfer the content of source register S to destination registers D1, D2,...,Dn.

3. Op1; Op2

   Perform operations op1 and op2 at the same time.

4. D = f([R1],...,[Rn])

   The function f is performed using the contents of registers R1 ,..., Rn as operands and its value is transferred to register D. If f is a binary operator (i.e. n = 2) we use infix notation rather than prefix notation.

5. a symbol for a data line, e.g. X

   Generate a signal on the indicated line: 1 if the symbol is uncomplemented (e.g., X) and 0 if the symbol is complemented (e.g. $\overline{X}$).

6. if *condition* then *RT-statement*

   Here *condition* has the form data-line or register = signal value(s). The statement indicates that the given register-transfer (RT) statement is to be performed if the indicated data line or register has the specified value.

Examples of registers transfers implementing the instruction cycle for IJVM: We now give examples of the register transfers needed to implement the instruction cycle for our CPU. In doing so we note the following:

- At the beginning and end of each IJVM instruction the register TOS will contain a copy of the word being referenced by SP; otherwise it can be used as a temporary storage register.
- The register OPC also can be used as a temporary (or scratch) register. It gets its name from its use in storing previous values ("old") values of the PC register.
- If a memory read is indicated (by either the **read** or **fetch**) the read operation starts at the end of the data path cycle using the value in the MAR or PC as is appropriate. Our discussion of the data path cycle showed that it is possible for PC or MAR to attain new values before the end of the cycle, however, meaning it is possible to change the value of PC or MAR and initiate a read operation with this new value.
- Continuing our discussion of memory reads, we shall assume that memory completes its operation within one cycle. This means that following the initiation of a memory read, we can assume the data being read is available in the MDR or MDR at the *end* of the next data path cycle, but not at the beginning of this cycle. Consequently we must wait until the cycle after that before it can be used.
- Whenever the value of the MBR (which is only 8 bits) is placed on the B bus two options will be available:
  a. Put the value in MBR in the low order bits and append twenty-four 0s for the high order bits.
  b. Treat the value of the MBR as an 8-bit signed integer and generate a 32-bit word with the same value by extending the sign bit of the 8-bit value into the twenty-four high order bits (meaning either twenty four 0s or twenty four 1s are copied).

Examples of Register Transfers Needed for various IJVM Instructions:

FETCH
POP
IADD
DUP
ILOAD
GOTO
IFLT
INVOKEVIRTUAL