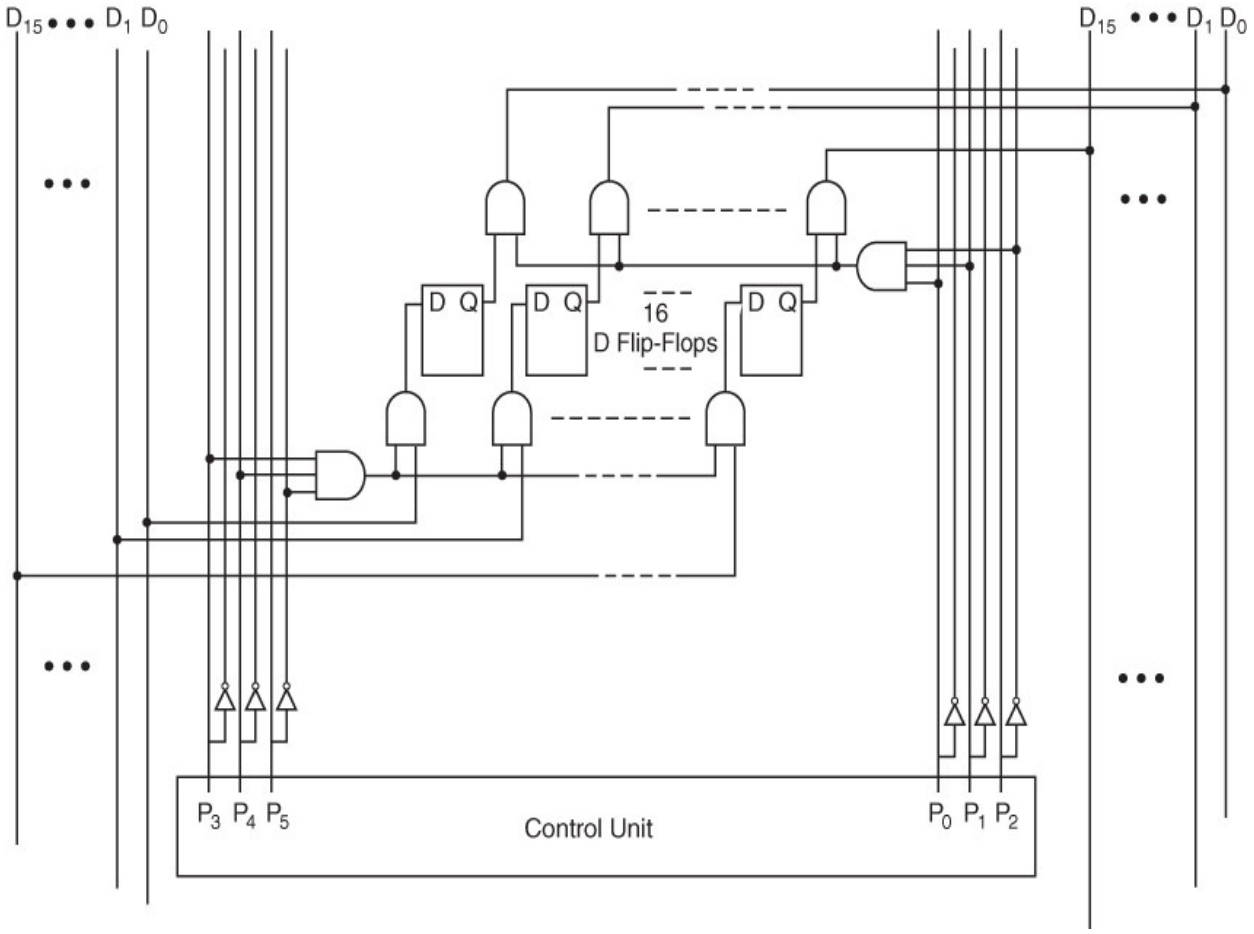
Supplement for Section 4.13 of the textbook



The text's Figure 4.15 (above) shows the connection of MARIE's MBR (Reg. #3) to the Datapath. The control unit uses $P_5P_4P_3$ to encode the register number receiving the data (D_{15} - D_0) and $P_2P_1P_0$ to encode the register number sending the data (D_{15} - D_0). Unfortuately, this figure has several problems.

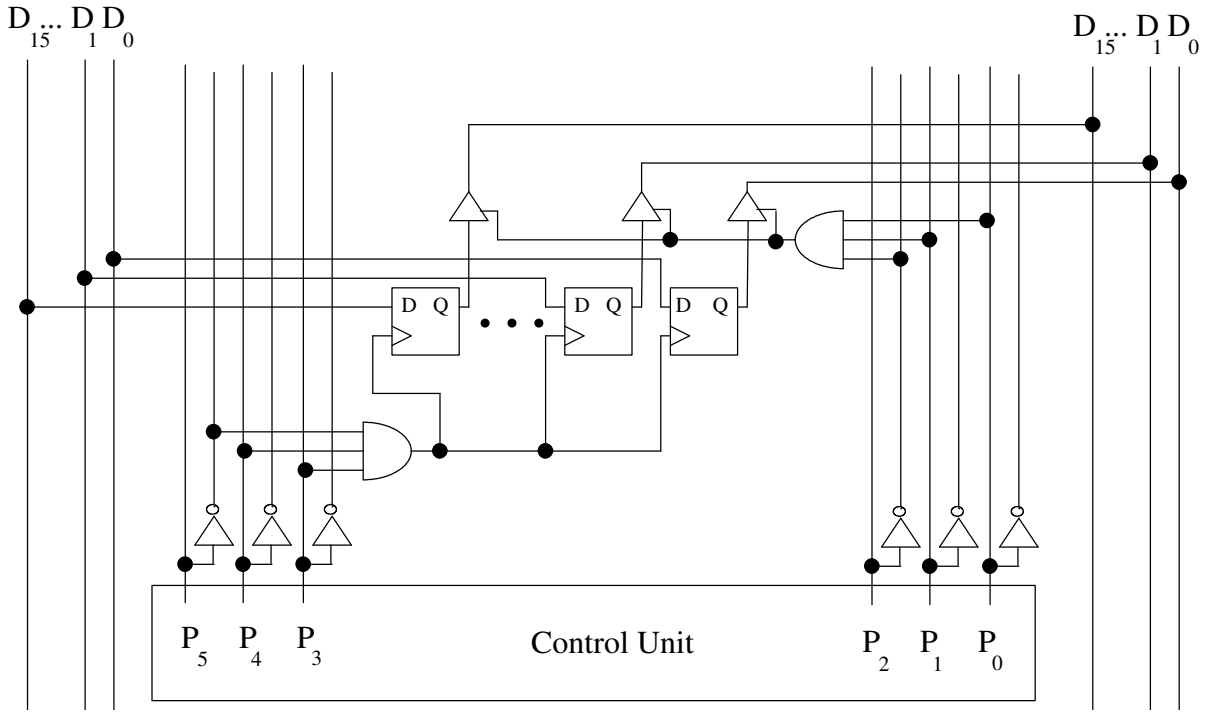
The first problem is way registers write their values to the data (D_{15} - D_0) wires. In Figure 4.15, the right-most AND gate is acting like a decoder and looking for the MBR's register number (Reg. #3) encoded on $P_2P_1P_0$. Since $3_{10} = 011_2$, this AND computes $\overline{P_2}P_1P_0$ which correctly outputs a 1 when the MBR is selected by the control unit to write to the data (D_{15} - D_0) wires. The output of this decoder AND is fed as input to the ANDs with the Qs from the D flip-flops. Since the decoder AND's output is a 1, these other ANDs allow the Qs to pass to the data (D_{15} - D_0) wires. On the surface, all seems good. However, consider that all the registers (MAR, PC, MBR, etc.) are connected to the bus similarly. When the MBR is selected, all the other registers NOT selected will have decoder ANDs that output 0s. These 0s will be fed as input to the ANDs with the Qs from their respective D flip-flops. Since the decoder ANDs' output is 0, all these other ANDs will try to write 0s to the data (D_{15} - D_0) wires. Thus, interferring with the MBR's attempt to put it's register value on the data wires.

The solution to this problem is straight forward. We can keep the decoder ANDs and feed its output to the control of tri-state buffers that replace the ANDs with inputs of the Qs from the D flip-flops. This way the tri-state buffers for the selected register connects the Qs of the D flip-flops to the data (D_{15} - D_0) wires, and the non-selected registers are disconnected by the tri-state buffers from the data wires.

The second problem with Figure 4.15 is the way registers read their values from the data (D_{15} - D_0) wires. Recall that D flip-flops should have a Clock input that can be used to signal when to load a new value on the D input. Notice that the D flip-flops in Figure 4.15 have no Clock inputs. Figure 4.15 correctly has a decoder AND gate that looks for the register number (Reg. #3 for the MBR) encoded on $P_5P_4P_3$. The output of this decoder AND

can be used as input into the Clock input of the D flip-flops. The D inputs of the flip-flops can be fed directly from the data (D_{15} - D_0) wires.

Below is a revised Figure 4.15 that shows how the MBR (register 3) should be connected to the bus.



Implementing the Control Unit: The control unit is reponsible for generating the sequence of control signals to drive the datapath, bus interface, memory, ALU through the fetch-decode-execute cycle. There are two general types of control units:

- hardwired - a combinational circuit as shown generally in Figure 4.17
- microprogrammed - a fixed program stored and run inside the control unit to cause the fetch-decode-execute of the user program's instructions

Hardwired Control Unit:

On page 216 of the text, the "signal patterns" for the ADD RTN seem wrong in a number of ways:

- The timing signs used are T_0 - T_3, when using T_4 - T_6 would make more sense since the "fetch" steps should be T_0 - T_3.
- "P_0 P_2 T_1: MBR← M[MAR]" should be "P_4 P_3 T_5: MBR← M[MAR]"
- For "AC ← AC + MBR", I'm not sure the P_0 and P_1 signals are needed since the AC and MBR are connected directly to the ALU. The P_5 signal makes some amount of sense if it triggers a "LOAD" of the AC.

The corrected signal patterns on page 216 should be:

P_3 P_2 P_1 P_0 T_4: MAR ← IR[11-0]
P_4 P_3 T_5: MBR ← M[MAR]
C_r A_0 P_5 T_6: AC ← AC + MBR and [Reset the clock cycle counter.]

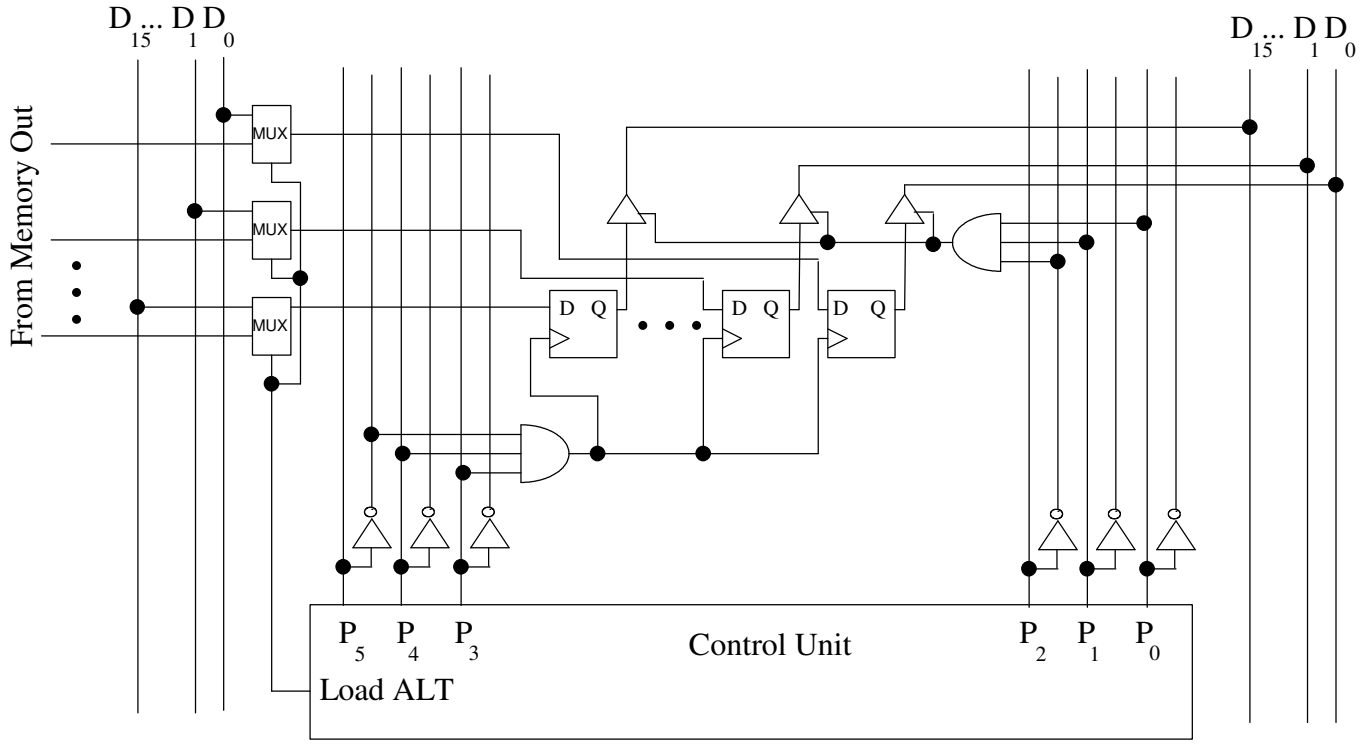
We can use the following table to determine which control signals should contain a "1" for each steps in the ADD instruction.

Supplement for Section 4.13 of the textbook

Step	RTN	Step #	P_5	P_4	P_3	P_2	P_1	P_0	C_r	Inc PC	Mem Read	Load ALT	Mem Write
Fetch	MAR ← PC	T_0	0	0	1	0	1	0	0	0	0	0	0
	MBR ← M[MAR]	T_1	0	1	1	-	-	-	0	0	1	1	0
	IR ← MBR	T_2	1	1	1	0	1	1	0	0	0	0	0
	PC ← PC + 1	T_3	0	0	0	-	-	-	0	1	0	0	0
Decode IR[15-12]	MAR ← IR[11-0]	T_4	0	0	1	1	1	1	0	0	0	0	0
Get operand	MBR ← M[MAR]	T_5	0	1	1	-	-	-	0	0	1	1	0
Execute	AC ← AC + MBR	T_6	1	0	0	-	-	-	1	0	0	1	0
		T_7											

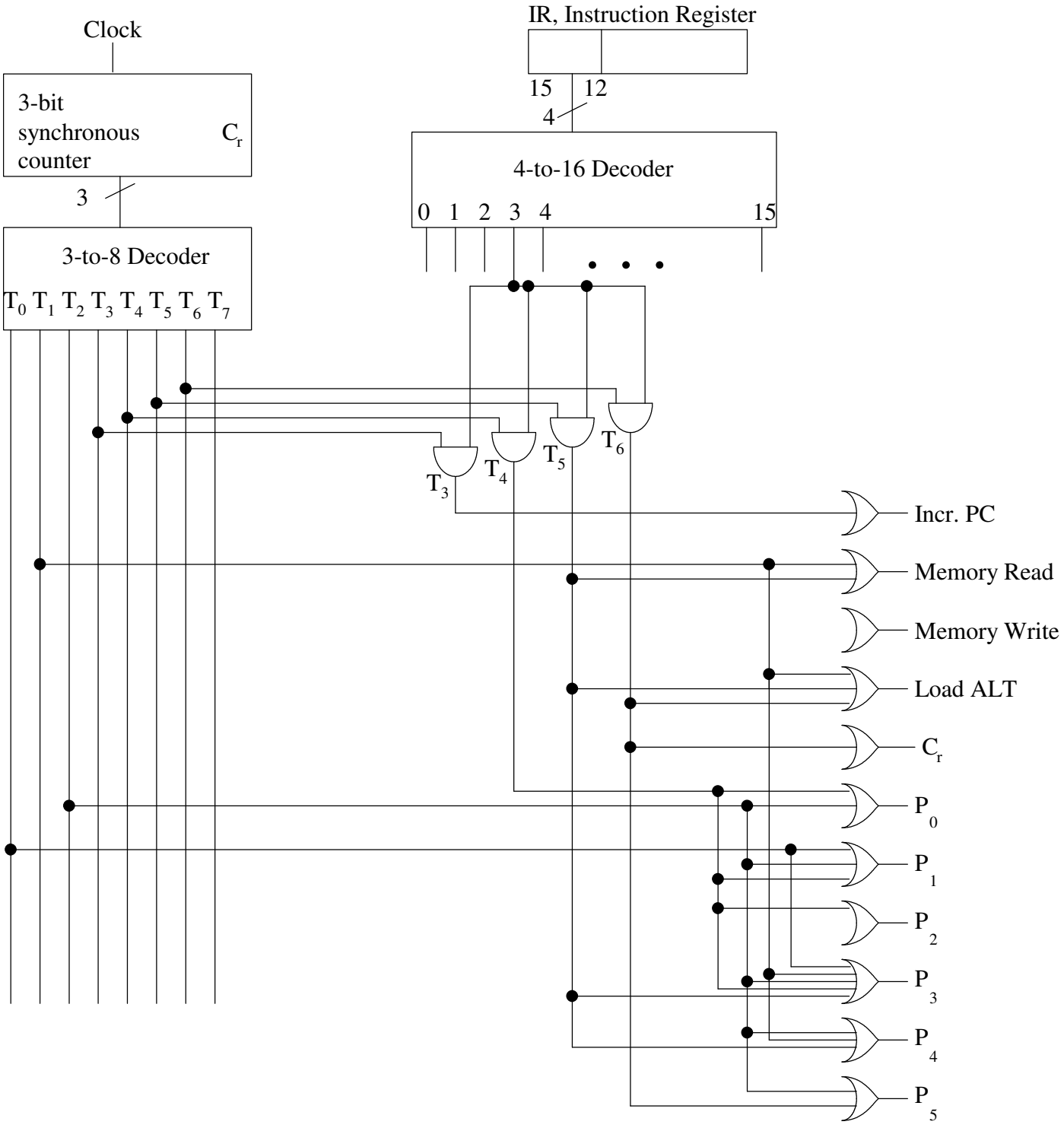
I've added a couple of columns for additional control signals:

- Inc PC - is used to tell the PC to increment itself (which we'll assume it is able to do)
- Mem Read - tells the memory that a read is being performed using the MAR as the address and the MBR as the receiver of the data read from memory
- Mem Write - tells the memory that a write is being performed using the MAR as the address and the MBR as the data being written to memory
- Load ALT - some of the registers (MBR, AC, and InREG) can load values from the data (D_{15} to D_0) wires or an alternate location (e.g., the AC can load a value from the ALU, and the MBR can load a value from the memory). 2-input MUXs with the Load ALT as the control could be used to select which value should be loaded into the register. For example, the alternate inputs to load the MBR from memory would be:



We can implement the a partial hardwire control unit that would generate the signals needed to handle the ADD instruction. To sequence the microoperations, we can build a cycle counter driven by the clock pulse to move a single "1" signal across the T_0 wire, to the T_1 wire, to the T_2 wire, etc. until the synchronous counter rolls over

back to T_0 or receive a clock reset (C_r) signal. The Fetch part of the above table T_0 to T_2 is always the same regardless of what instruction is being fetched. The remaining cycles (> T_2) are all dependent on the type of instruction that was fetched into the IR. Since the opcode part of the instruction is four bits (IR[15-12]), we can use a 4-to-16 decoder to tell us which instruction was fetched. For an ADD instruction whose opcode is 3, the 3 output of this decoder will contain a 3 with all other outputs having a 0. The four AND gates check to make sure that an ADD instruction is being executed and a cycle greater than T_2. The control signals (Incr PC, Memory Read, ..., P_5) each are generated by an OR gate since many more situations call for these control signal, especially other instructions.

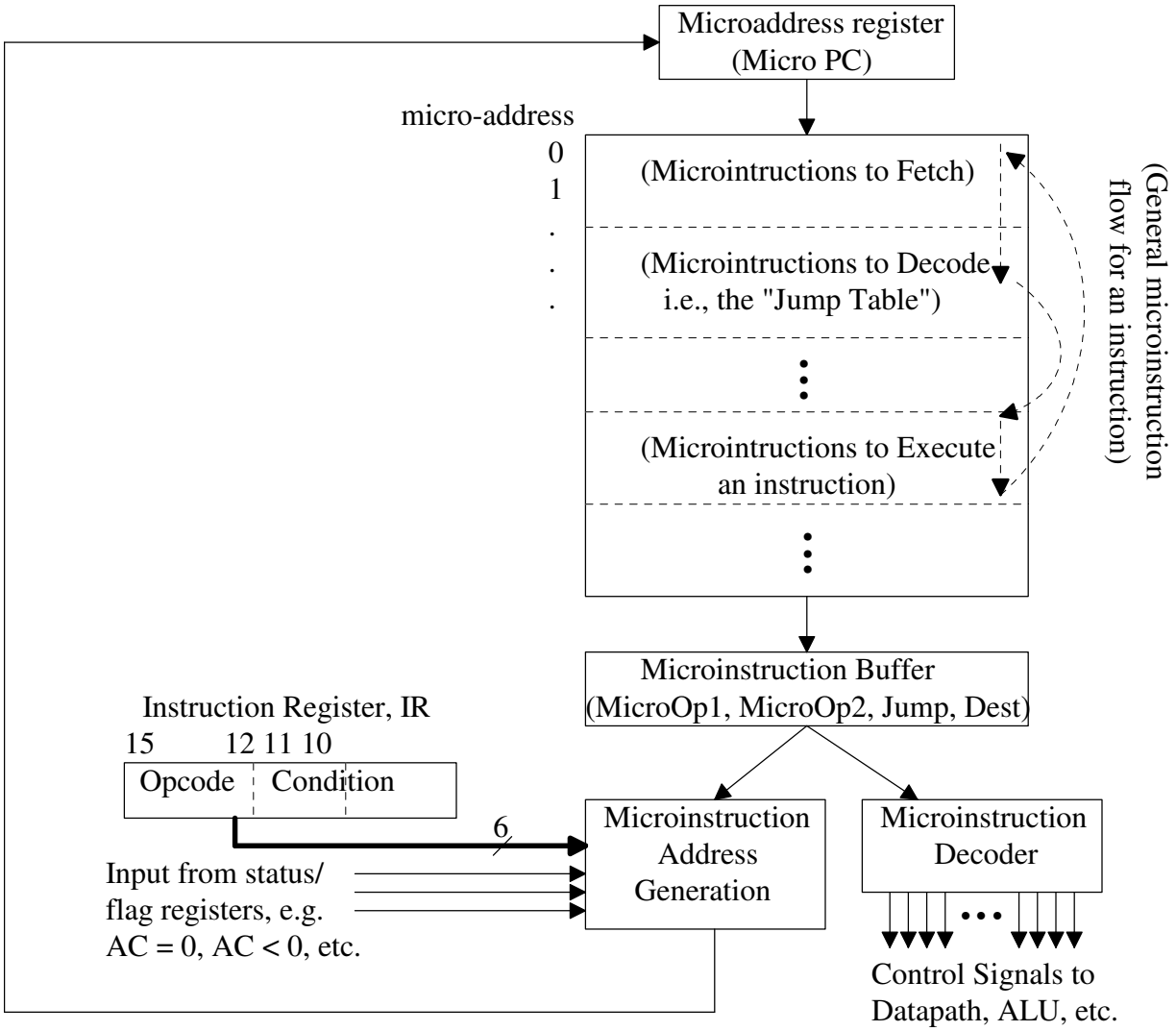


Supplement for Section 4.13 of the textbook

Microprogrammed Control Unit

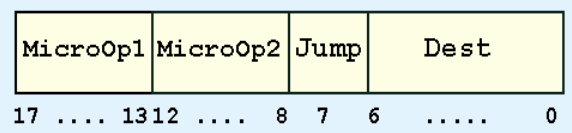
- A hardwired control unit uses a circuit to repeatedly generates control signals to fetch-decode-execute the next machine-language instruction of the program. A microprogrammed control unit running a special purpose "microprogram" inside the control unit to generates control signals to fetch-decode-execute the next machine-language instruction of the program.
- Microinstructions are fetched, decoded, and executed in the same manner as regular instructions. This extra level of instruction interpretation is what makes microprogrammed control slower than hardwired control.
- The advantages of microprogrammed control are that it can support very complicated instructions and only the microprogram might need to be changed if the instruction set changes (or an error is found).

Revised Figure 4.19 Microprogrammed Control Unit



Supplement for Section 4.13 of the textbook

The microprogrammed version of MARIE executes a fixed microprogam to perform the fetch-decode-execute cycle. The instruction format for the microinstructions look like:



`MicroOp1` encodes the type of register transfer notation (RTN) to perform (e.g., AC ← 0 is 00010_2)

`MicroOp2` is used only when decoding the instruction. It contains the binary codes for each instruction to allow comparison to the IR opcode. (Since the MARIE opcodes are only 4-bits long, only bits 12 - 9 are used and bit 8 is unused.

`Jump` is a single bit indicating that the value in the **Dest** field is a valid micro-address and should be placed in the microsequencer; if **`Jump`** is "FALSE" (0), then increment to the next microinstruction.

Table 4.8. Microoperation Codes and Corresponding MARIE RTN (p. 221)

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01100	MBR ← M[MAR]
00001	AC ← 0	01101	OutREG ← AC
00010	AC ← AC - MBR	01110	PC ← IR[11-0]
00011	AC ← AC + MBR	01111	PC ← MBR
00100	AC ← InREG	10000	PC ← PC + 1
00101	IR ← M[MAR]	10001	If AC = 00
00110	M[MAR] ← MBR	10010	If AC > 0
00111	MAR ← IR[11-0]	10011	If AC < 0
01000	MAR ← MBR	10100	If IR[11-10] = 00
01001	MAR ← PC	10101	If IR[11-10] = 01
01010	MAR ← X	10110	If IR[11-10] = 10
01011	MBR ← AC	10111	If IR[15-12] = MicroOp2[4-1]

We need to augment this table to include a few omitted microoperations and because we modified Figure 4.9 to remove the Memory from direct connection to the datapath. The following additional microoperations are used.

MicroOp Code	Microoperation
00101*	IR ← MBR
11000	AC ← MBR

* This microop code is being reused.

Below is a revised Figure 4.21 which is a partial microprogram for MARIE. The Fetch part of the Fetch-Decode-Execute cycle starts at microaddress (μAddr) 0 and continues through microaddress 3. For these microinstructions:

- MicroOp1 encodes for the familar RTN of the Fetch.
- MicroOp2 is not used since we are Decoding a MARIE instruction in IR yet
- Jump is 0 since we just want these microinstructions to be executed sequentially
- Dest is ignored since we are not jumping

Supplement for Section 4.13 of the textbook

The Decode part of the Fetch-Decode-Execute cycle starts at microaddress (μAddr) 4 and continues through microaddress 16. This is also known as the *Jump Table*. For these microinstructions:

- MicroOp1 encodes for "If IR[15-12] = MicroOp2[4-1]" which means that the Opcode of the instruction just fetched into the IR should be compared to the first four bits of the MicroOp2. If they match, jump to the microaddress specified in the Dest field; otherwise fetch the next microinstruction sequentially
- MicroOp2 is varied to check all the different MARIE opcodes
- Jump is 1 since we might jump to microinstruction specified in Dest depending on the comparison
- Dest contains the microaddress of the first Execution microinstruction corresponding to the MARIE Opcode in MicroOp2

Revised Figure 4.21 Partial Microprogram

Part of Cycle	RTN (of MicroOp1)	μAddr	MicroOp1	MicroOp2	Jump	Dest
Fetch	MAR ← PC	0	01001	0000	0	0
	MBR ← M[MAR]	1	01100	0000	0	0
	IR ← MBR	2	00101	0000	0	0
	PC ← PC + 1	3	10000	0000	0	0
Decode	If ADD, Jump	4	10111	00110	1	17_{10}
("Jump Table")	If LOAD, Jump	5	10111	00010	1	20_{10}
	If STORE, Jump	6	10111	00100	1	23_{10}
	If SKIPCOND, Jump	7	10111	10000	1	26_{10}
	If SUBT, Jump	8	10111	01000	1	
	If JUMP, Jump	9	10111	10010	1	
	If ADDI, Jump	10	10111	10110	1	
	If CLEAR, Jump	11	10111	10100	1	
	If JNS, Jump	12	10111	00000	1	
	If JUMPI, Jump	13	10111	11000	1	
	If INPUT, Jump	14	10111	01010	1	
	If OUTPUT, Jump	15	10111	01100	1	
	If HALT, Jump	16	10111	01110	1	0
Execute ADD	MAR ← IR[11-0]	17	00111	00000	0	0
	MBR ← M[MAR]	18	01100	00000	0	0
	AC ← AC + MBR	19	00011	00000	1	0
Execute LOAD	MAR ← IR[11-0]	20	00111	00000	0	0
	MBR ← M[MAR]	21	01100	00000	0	0
	AC ← MBR	22	11000	00000	1	0
Execute STORE	MAR ← IR[11-0]	23	00011	00000	0	0
	MBR ← AC	24	01011	00000	0	0
	M[MAR] ← MBR	25	00110	00000	1	0
Execute SKIPCOND	If IR[11-10]=00, Jump	26	10100	00000	1	30_{10}
	If IR[11-10]=01, Jump	27	10101	00000	1	32_{10}
(here only if	If AC > 0, Jump	28	10010	00000	1	34_{10}
IR[11-10]=10)	NOOP, Jump to Fetch	29	00000	00000	1	0
	If AC < 0, Jump	30	10011	00000	1	34_{10}
	NOOP, Jump to Fetch	31	00000	00000	1	0
	If AC = 0, Jump	32	10001	00000	1	34_{10}
	NOOP, Jump to Fetch	33	00000	00000	1	0
	PC ← PC + 1	34	10011	00000	1	0
Execute SUBT		35	...			

Supplement for Section 4.13 of the textbook

The Execute part of the Fetch-Decode-Execute cycle starts at microaddress (μAddr) 17 for the rest of the microprogram. The execution microinstructions for each MARIE instruction are grouped together is this part. For the microinstructions that cause the execution of a MARIE instruction:

- MicroOp1 encodes for the familiar RTN of the Execution
- MicroOp2 is not used since we have already Decoding the MARIE instruction in the IR
- Jump is 0 for all but the last microinstruction of the Execution since we just want these microinstructions to be executed sequentially. (the exception to this is the SKIPCOND instruction -- explanation to follow) The last microinstruction of an Execution sequence has a 1 for Jump and Dest of 0 so we can start the next Fetch-Decode-Execute cycle.
- Dest is 0 and either ignored if the Jump is 0 or causes the Fetch at microaddress 0 to be executed if Jump is 1 (the exception to this is the SKIPCOND instruction -- explanation to follow)

The execution microinstructions for the SKIPCOND instruction starts with a "mini" jump table at microaddresses 26 and 27 to decode which condition (bits 11 and 10 of the IR) should be compared. Once the condition has be determined, the microinstruction to perform the corresponding comparison is performed. If the result of the comparison is True, the microprogram jumps to microaddress 34 to increment the PC and thus causes the instruction after the SKIPCOND to be skipped. If the result of the comparison is False, the jump is not taken and the next microinstruction is executed which is a NOOP (NO OPeration or do nothing), but Jumps back to the fetch at microaddress 0.