CHAPTER 8

PROGRAMMABLE PROCESSORS

8.1 EXERCISES

Section 8.2: Basic Architecture

8.1. If a processor's program counter is 20-bits wide, up to how many words can the processor's instruction memory hold (ignoring any special tricks to expand the instruction memory size)?

$2^{20} = 1{,}048{,}576$

8.2 Which of the following are legal single-cycle datapath operations for the datapath in Figure 8.2? Explain your answer.

a. Copy data from a memory location into another memory location.
b. Copy two register locations into two memory locations.
c. Add data from a register file location and a memory location, storing the result in a memory location.

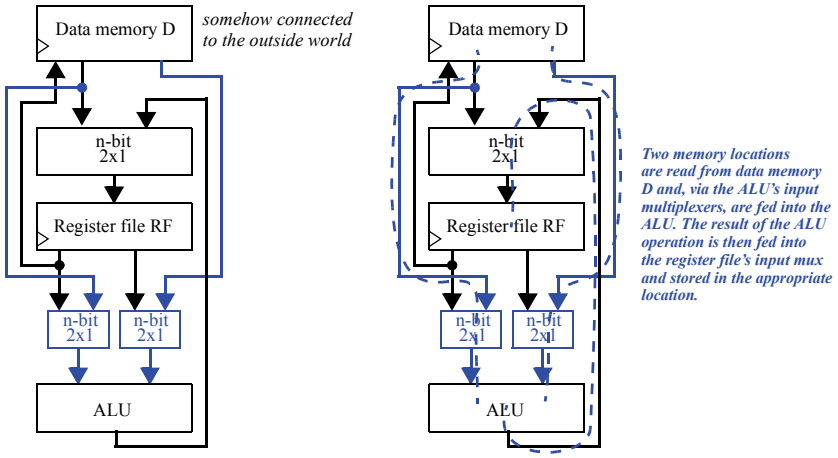
a) Invalid. Data must first be loaded into the register file then stored into the destination memory location.
b) Invalid. Only one register file to memory location copy is permitted during a single cycle.
c) Invalid. Data must first be loaded into a register file, then the addition must be performed, then the sum must be stored into a memory location. The entire sequence of operations would take three cycles.

8.3 Which of the following are legal single-cycle datapath operations for the datapath in Figure 8.2? Explain your answer.

a. Copy data from a register file location into a memory location.
b. Subtract data from two memory locations and store the result in another memory location.
c. Add data from a register file location and a memory location, storing the result in the same memory location.

a) Valid operation.
b) Invalid. Two cycles are required to load the two operands. One cycle is required to perform the subtraction. One cycle is required to store the difference. Four cycles total are needed to perform this sequence of operations.
c) Invalid. Three cycles are required (Load, Add, Store).

8.4 Assume we are using a dual-port memory from which we can read two locations simultaneously. Modify the datapath of the programmable processor of Figure 8.2 to support an instruction that performs an ALU operation on any two memory locations and stores the result in a register file location. Trace through the execution of this operation, as illustrated in Figure 8.3.



8.5 Determine the operations required to instruct the datapath of Figure 8.2 to perform the operation: D[8] = (D[4] + D[5]) - D[7], where D represents the data memory.

1) Load D[4] into the register file (R[0])
2) Load D[5] into the register file (R[1])
3) Add R[0] and R[1] and store the result in the register file (R[2])
4) Load D[7] into the register file (R[0])
5) Subtract R[0] from R[2] and store the result in the register file (R[1])
6) Store R[1] in the data memory location D[8]

Section 8.3: A Three-Instruction Programmable Processor

8.6 If a processor's instruction has 4 bits for the opcode, how many possible instructions can the processor support?

$2^4 = 16$

8.7 What does the following assembly program, which uses the three-instruction instruction set of this chapter, compute? MOV R5, 19; ADD R5, R5, R5; MOV 20, R5.

D[20] = D[19] + D[19]

8.8 What does the following assembly program, which uses the three-instruction instruction set of this chapter, compute? MOV R4, 20; MOV R9, 18; ADD R4, R4, R9; MOV R5, 30; ADD R9, R4, R5; MOV 20, R9.

D[20] = D[20] + D[18] + D[30]

8.9 Using the three-instruction instruction set of this chapter, write an assembly program that updates the data memory D as follows: D[0]=D[0]+D[1].

```
MOV R0, 0
MOV R1, 1
ADD R0, R0, R1
MOV 0, R0
```

8.10 Using the three-instruction instruction set of this chapter, write an assembly program that updates the data memory D as follows: D[4]=D[1]*2+D[2].

```
MOV R0, 1
ADD R0, R0, R0
MOV R1, 2
ADD R0, R0, R1
MOV 4, R0
```

8.11 Convert the following assembly program to machine code based on the three-instruction instruction set of this chapter: MOV R5, 19; ADD R5, R5, R5; MOV 20, R5.

```
0000 1001 00010011
0010 1001 1001 1001
0001 1001 00010100
```

8.12 List the basic register/memory transfers and operations that occur during each clock cycle for the following program, based on the three-instruction instruction set of this chapter: MOV R0, 1; MOV R1, 9; ADD R0, R0, R1;

1) Fetch Instruction #1
2) Decode Instruction #1
3) The FSM sets the control lines on the memory and register file to load D[1] into RF[0]
4) Fetch Instruction #2
5) Decode Instruction #2
6) The FSM sets the control lines on the memory and register file to load D[9] into RF[1]
7) Fetch Instruction #3
8) Decode Instruction #3
9) The FSM sets the control lines on the ALU and register file to effect RF[0] := RF[0] + RF[1]

Section 8.4: A Six-Instruction Programmable Processor

8.13 List the basic register/memory transfers and operations that occur during each clock cycle for the following program, based on the six-instruction instruction set of this chapter, assuming that the content of D[9] is 0: MOV R6, #1; MOV R5, 9; JMPZ R5, label1; ADD R5, R5, R6; label1: ADD R5, R5, R6. What is the value in R5 after the program completes?

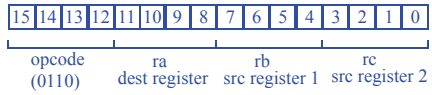
1) Fetch Instruction #1
2) Decode Instruction #1
3) The FSM sets the control lines on the register file and RF write mux to load the constant value '1' to RF[6]
4) Fetch Instruction #2
5) Decode Instruction #2
6) The FSM sets the conrol lines on the register file, RF write mux, and memory to load the contents of D[9] (which contains '0') to RF[5]
7) Fetch Instruction #3
8) Decode Instruction #3
9) The FSM sets the control lines on the register file to test whether RF[5] is '0'
10) RF[5] was '0', so the PC gets loaded with PC + 2 - 1 (the offset of label1)
11) Fetch Instruction #5
12) Decode Instruction #5
13) The FSM sets the control lines on the register file, the RF write mux, and the ALU to effect RF[5] := RF[5] + RF[6]

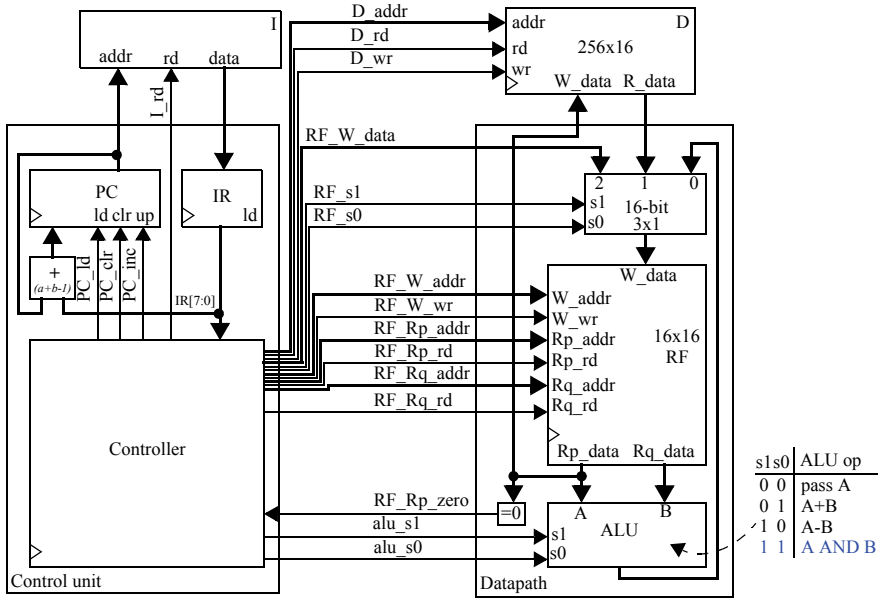
After the program completes, RF[5] is 1.

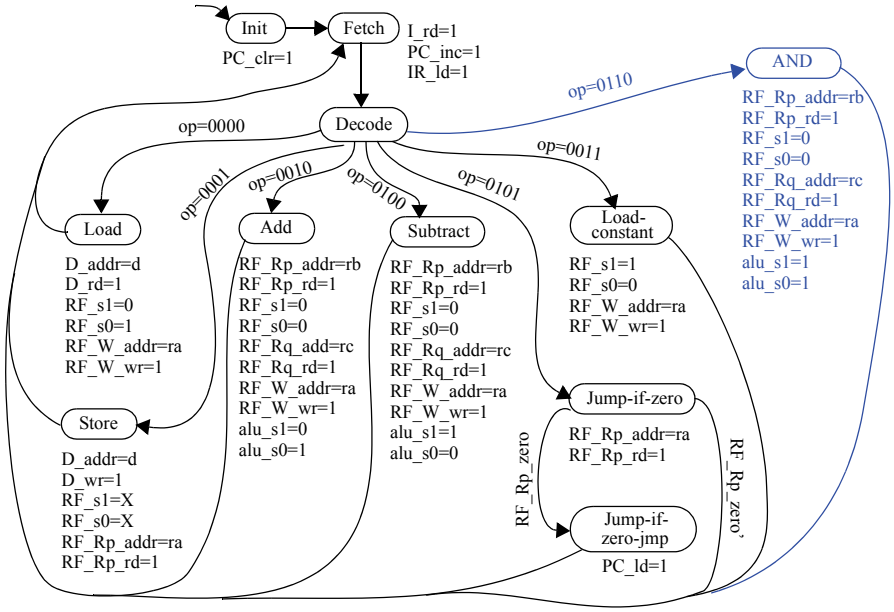
8.14 Add a new instruction to the six-instruction instruction set of this chapter that performs a bitwise AND of two registers and stores the result in a third register. Extend the datapath, control unit, and the controller's FSM as needed.

We'll use the opcode 0110 for the AND operation. We'll modify the ALU to perform the AND operation when the ALU's s1s0=11

AND ra, rb, rc

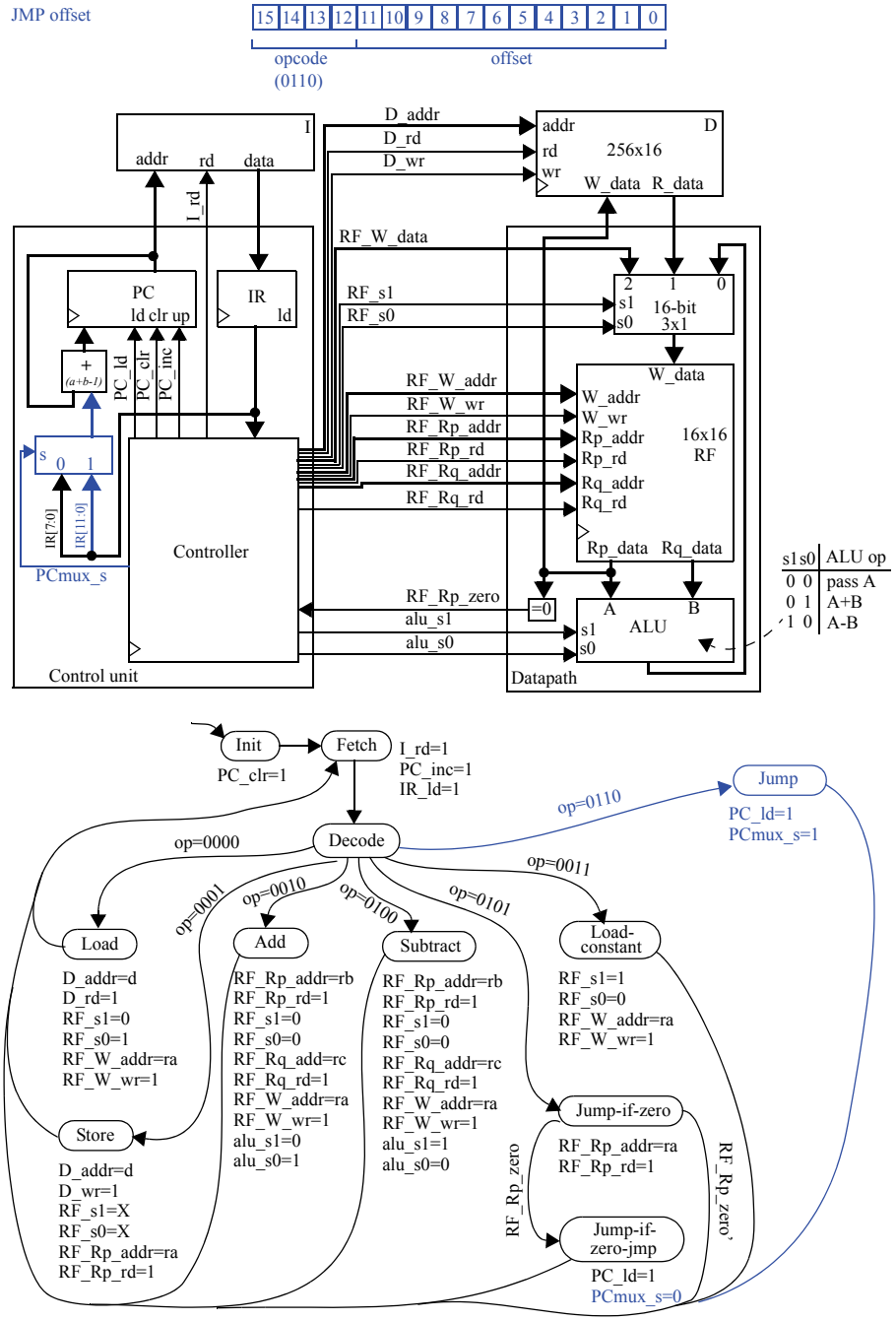





Init
PC_clr=1
Fetch
I_rd=1
PC_inc=1
IR_ld=1
Decode
op=0000
op=0001
op=0010
op=0100
op=0101
op=0011
op=0110
Load
D_addr=d
D_rd=1
RF_s1=0
RF_s0=1
RF_W_addr=ra
RF_W_wr=1
Store
D_addr=d
D_wr=1
RF_s1=X
RF_s0=X
RF_Rp_addr=ra
RF_Rp_rd=1
Add
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_add=rc
RF_Rq_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s1=0
alu_s0=1
Subtract
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_addr=rc
RF_Rq_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s1=1
alu_s0=0
Load-
constant
RF_s1=1
RF_s0=0
RF_W_addr=ra
RF_W_wr=1
Jump-if-zero
RF_Rp_addr=ra
RF_Rp_rd=1
RF_Rp_zero
Jump-if-
zero-jmp
PC_ld=1
RF_Rp_zero'
AND
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_addr=rc
RF_Rq_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s1=1
alu_s0=1

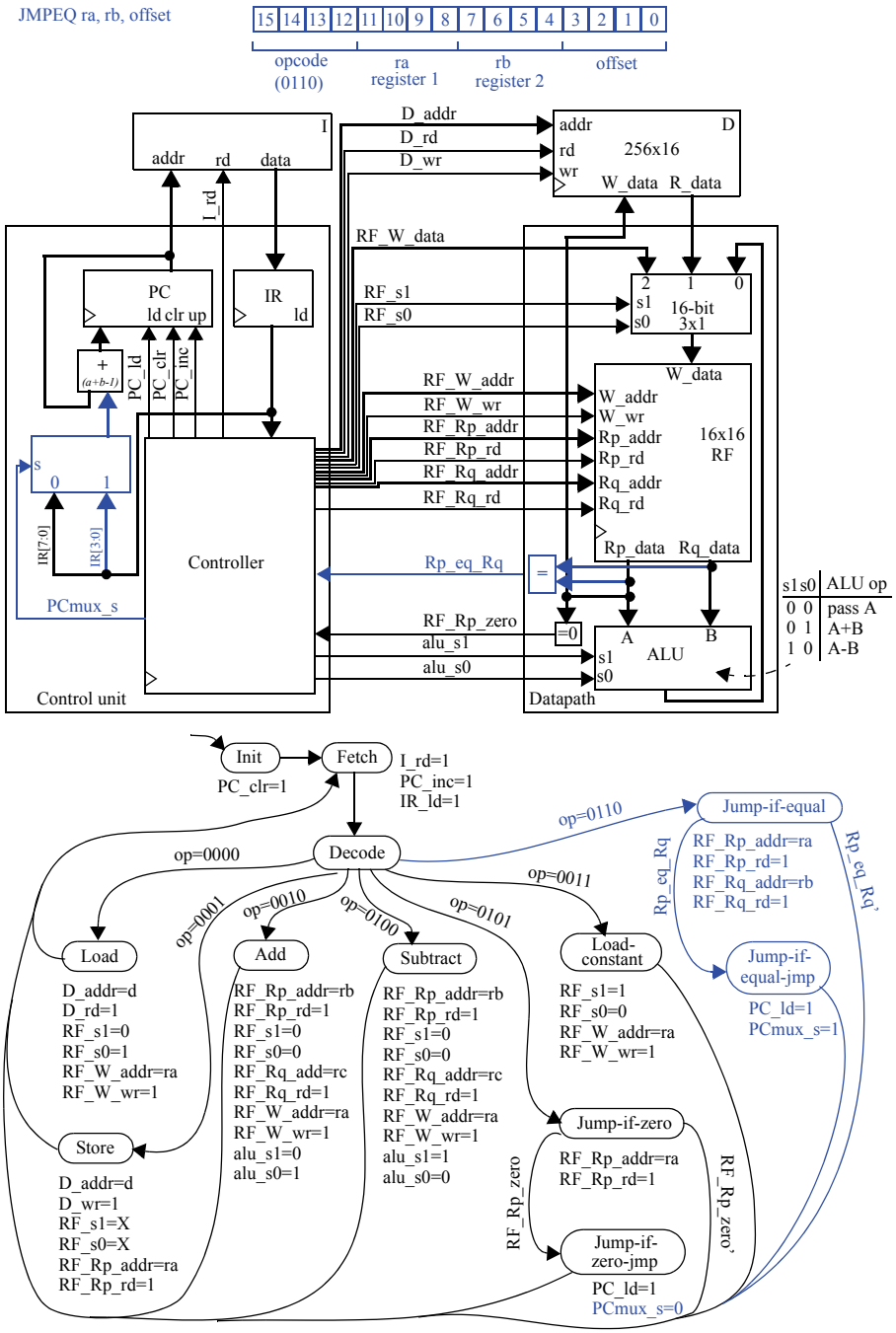
8.15 Add a new instruction to the six-instruction instruction set of this chapter that performs an unconditional jump (jumps always) to a location specified by a 12-bit offset. Extend the datapath, control unit, and the controller's FSM as needed.

We'll use opcode 0110.



8.16 Add a new instruction to the six-instruction instruction set of this chapter that performs a jump if two registers are equal, to a location specified by a 4-bit offset. Extend the datapath, control unit, and the controller's FSM as needed.

We'll use opcode 0110.



8.17 Using the six-instruction instruction set of this chapter, write an assembly program for the following C code, which computes the sum of the first N numbers, where N is another name for D[9]. Hint: use a register to first store N..

```
i=0;
sum=0;
while ( i!=N ) {
  sum = sum + i;
  i = i + 1;
}
```

```
        MOV R0, #0        // R0 is "i"
        MOV R1, #0        // R1 is "sum"
        MOV R2, #1        // R2 is the constant "1"
        MOV R3, 9         // R3 is "N" or "D[9]"
        MOV R4, #0        // R4 is the constant "0" (for looping)
loop:   SUB R5, R3, R0    // R4 = N - i
        JMPZ R5, done     // if i==N, end while loop
        ADD R1, R1, R0    // sum = sum + i
        ADD R0, R0, R2    // i = i + 1
        JMPZ R4, loop     // continue through while loop
done:
```

8.18 Using the extended instruction set you designed in Exercise 8.16, write an assembly program for the C code in Exercise 8.17.

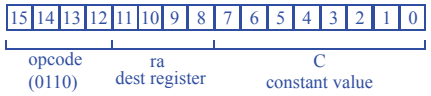
```
        MOV R0, #0            // R0 is "i"
        MOV R1, #0            // R1 is "sum"
        MOV R2, #1            // R2 is the constant "1"
        MOV R3, 9             // R3 is "N" or "D[9]"
        MOV R4, #0            // R4 is the constant "0" (for looping)
loop:   JMPEQ R0, R3, done    // end while loop if i==N
        ADD R1, R1, R0        // sum = sum + i
        ADD R0, R0, R2        // i = i + 1
        JMPZ R4, loop         // continue through while loop
done:
```

Section 8.5: Example Assembly and Machine Programs

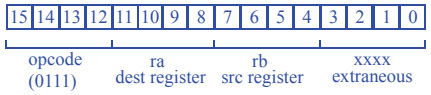
8.19 Define two new data movement instructions for the six-instruction instruction set of this chapter. Extend the datapath, control unit, and the controller's FSM as needed.

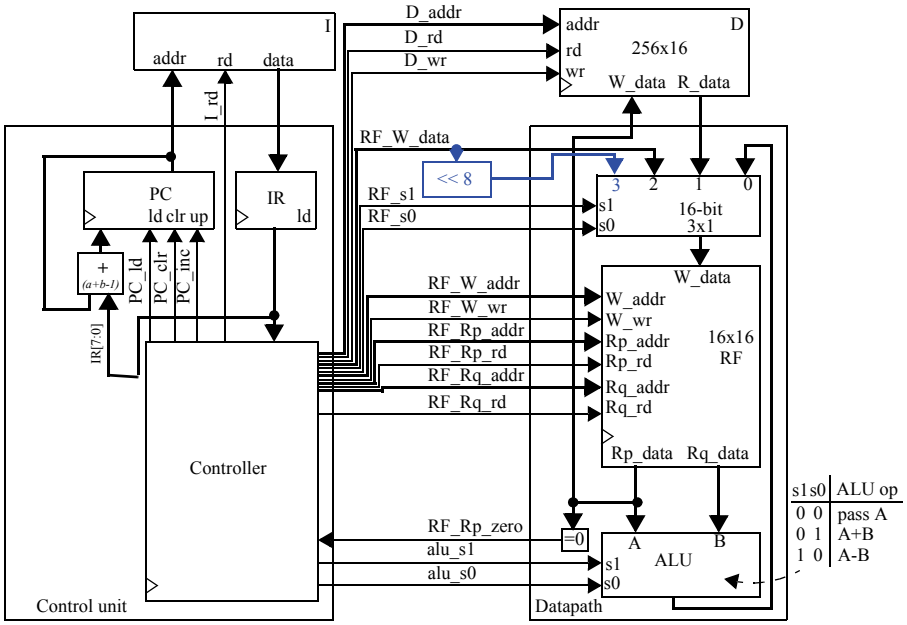
We'll define LUI and MOVR, with opcodes 0110 and 0111.
LUI will act just as "MOV Ra, #C" but will load #C into the upper 8 bits of Ra.
MOVR will allow us to duplicate the contents of one register into another, eliminating the need to use memory or initialize another register to zero. Its syntax is "MOVR Ra, Rb", where Ra is assigned Rb's value.

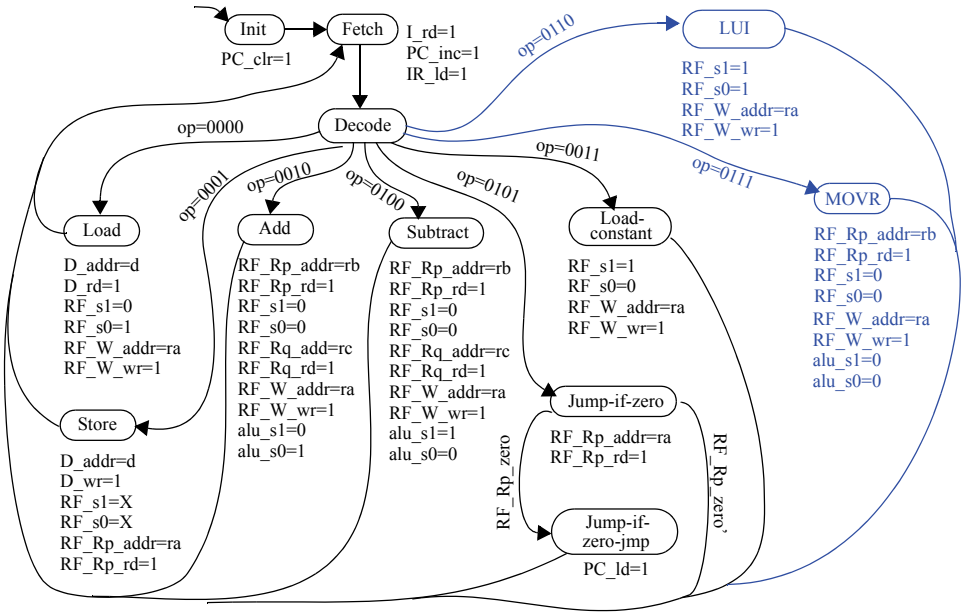
LUI ra, #C



MOVR ra, rb



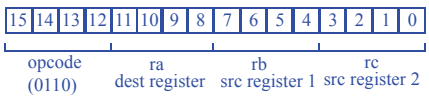




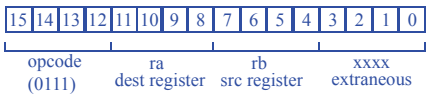
8.20 Define two new arithmetic/logic instructions for the six-instruction instruction set of this chapter. Extend the datapath, control unit, and the controller's FSM as needed.

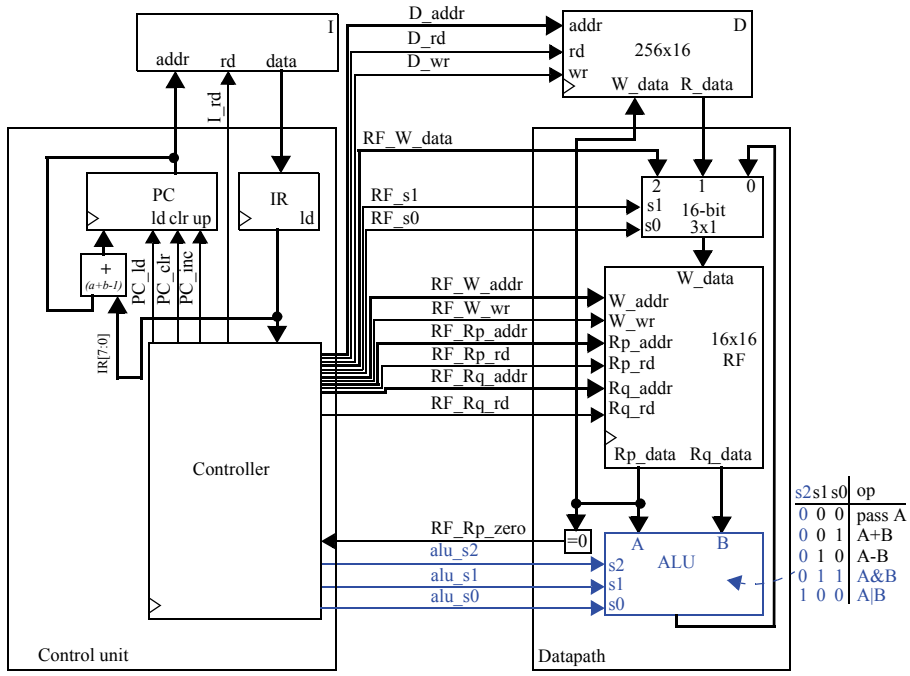
We'll define AND and NOT, with opcodes 0110 and 0111. The syntax for AND will be "AND Ra, Rb, Rc", where Ra gets the bitwise AND of the contents of of Rb and Rc. The syntax for NOT will be "NOT Ra, Rb", where Ra gets the logical complement of the contents of Rb.

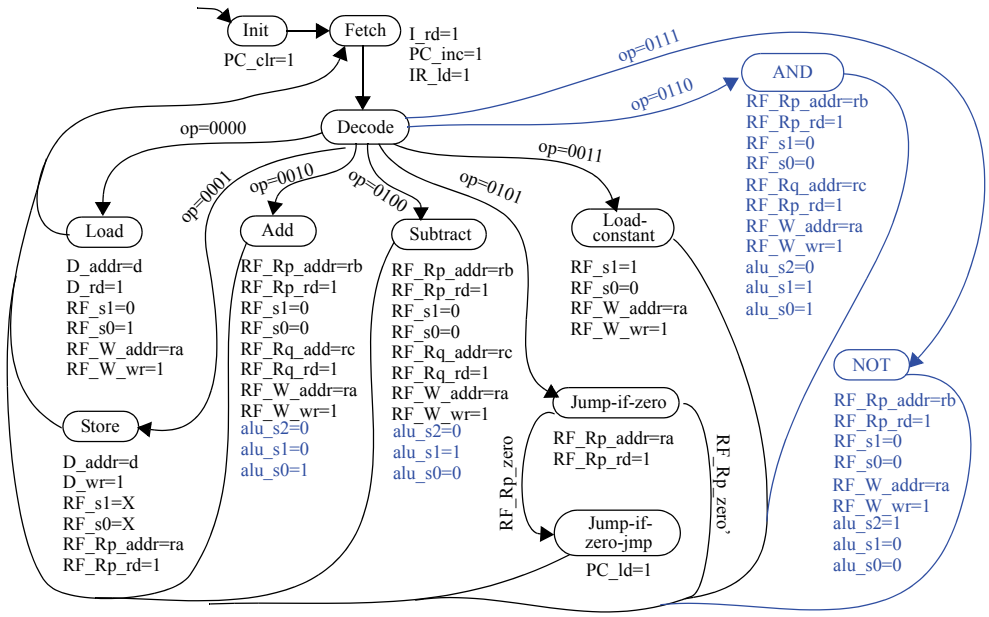
AND ra, rb, rc



NOT ra, rb

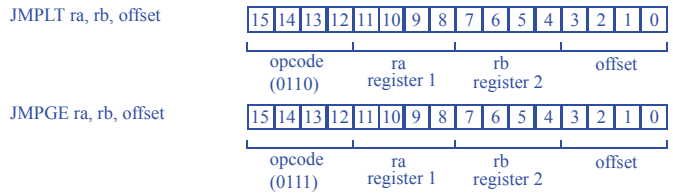


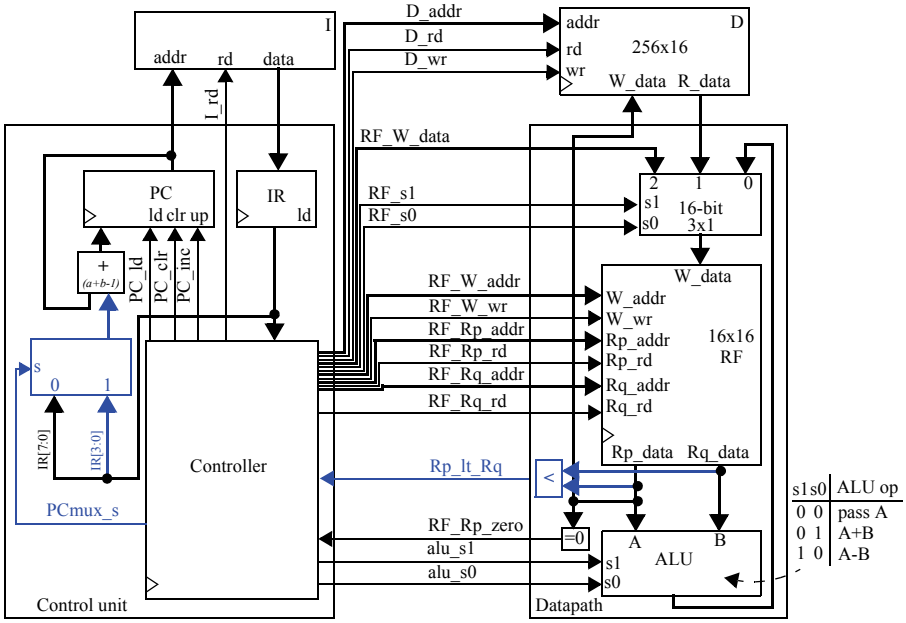

addr rd data
I_rd
D_addr
D_rd
D_wr
addr D
rd 256x16
wr
W_data R_data
RF_W_data
PC
ld clr up
IR
ld
RF_s1
RF_s0
2 1 0
s1 16-bit
s0 3x1
+
(a+b-1)
PC_ld
PC_clr
PC_inc
IR[7:0]
W_data
RF_W_addr
RF_W_wr
RF_Rp_addr
RF_Rp_rd
RF_Rq_addr
RF_Rq_rd
W_addr
W_wr
Rp_addr 16x16
Rp_rd RF
Rq_addr
Rq_rd
Rp_data Rq_data
Controller
RF_Rp_zero
alu_s2
alu_s1
alu_s0
=0
A B
s2 ALU
s1
s0
s2 s1 s0 op
0 0 0 pass A
0 0 1 A+B
0 1 0 A-B
0 1 1 A&B
1 0 0 A|B
Control unit
Datapath

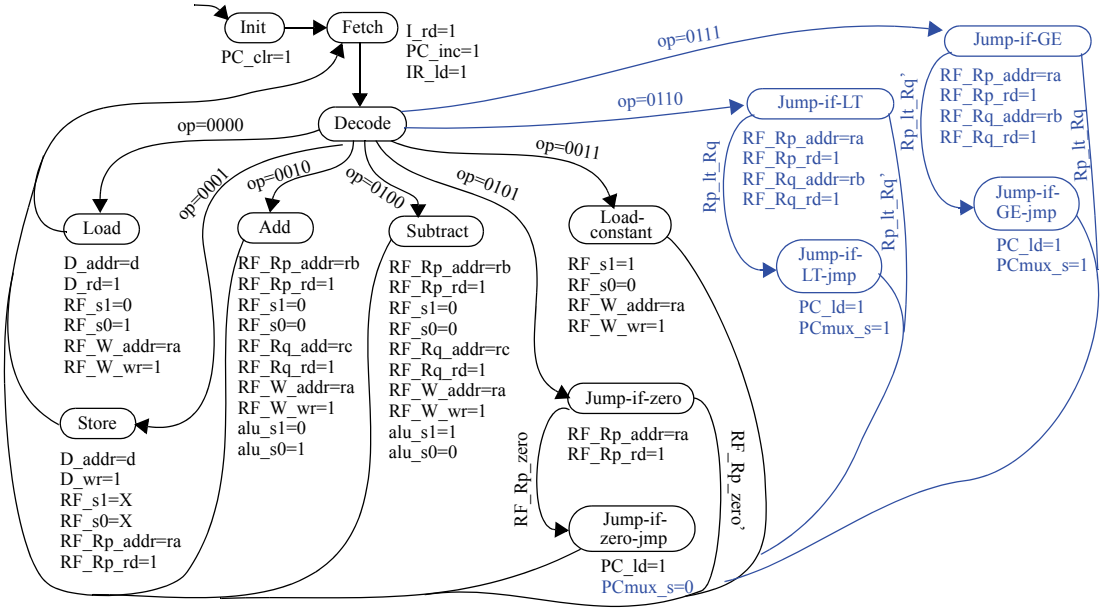

Init
PC_clr=1
Fetch
I_rd=1
PC_inc=1
IR_ld=1
Decode
op=0000
op=0001
op=0010
op=0100
op=0101
op=0011
op=0110
op=0111
Load
D_addr=d
D_rd=1
RF_s1=0
RF_s0=1
RF_W_addr=ra
RF_W_wr=1
Store
D_addr=d
D_wr=1
RF_s1=X
RF_s0=X
RF_Rp_addr=ra
RF_Rp_rd=1
Add
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_add=rc
RF_Rq_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s2=0
alu_s1=0
alu_s0=1
Subtract
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_addr=rc
RF_Rq_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s2=0
alu_s1=1
alu_s0=0
Load-
constant
RF_s1=1
RF_s0=0
RF_W_addr=ra
RF_W_wr=1
Jump-if-zero
RF_Rp_addr=ra
RF_Rp_rd=1
RF_Rp_zero
Jump-if-
zero-jmp
PC_ld=1
RF_Rp_zero'
AND
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_Rq_addr=rc
RF_Rp_rd=1
RF_W_addr=ra
RF_W_wr=1
alu_s2=0
alu_s1=1
alu_s0=1
NOT
RF_Rp_addr=rb
RF_Rp_rd=1
RF_s1=0
RF_s0=0
RF_W_addr=ra
RF_W_wr=1
alu_s2=1
alu_s1=0
alu_s0=0

8.21 Define two new flow-of-control instructions for the six-instruction instruction set of this chapter. Extend the datapath, control unit, and the controller's FSM as needed.

We'll define JMPLT and JMPGE, with opcodes 0110 and 0111. The syntax for JMPLT will be "JMPLT Ra, Rb, offset", where we jump to the offset if the contents of Ra are less than the contents of Rb. The syntax for JMPGE will be "JMPGE Ra, Rb, offset", where we jump to the offset if the contents of Ra are greater than or equal to the contents of Rb.







8.22 Assuming that the microprocessor's external pins *I0..I7* and *P0..P7* are mapped to data memory locations as in Figure 8.15 and an AND instruction has been added to the six-instruction instruction set of this chapter, create an assembly program that will output 0 on *P4* if all eight inputs *I0..I7* are `1`s.

```
        MOV R0, #1          // R0 is the constant "1"
        MOV R1, 240         // R1 gets the value of I0
        MOV R2, 241         // R2 gets the value of I1
        AND R2, R1, R2      // R2 = I0 ANDI1
        MOV R1, 242         // R1 = I2
        AND R2, R1, R2      // R2 = R2 AND I2
        MOV R1, 243         // R1 = I3
        AND R2, R1, R2      // R2 = R2 AND I3
        MOV R1, 244         // R1 = I4
        AND R2, R1, R2      // R2 = R2 AND I4
        MOV R1, 245         // R1 = I5
        AND R2, R1, R2      // R2 = R2 AND I5
        MOV R1, 246         // R1 = I6
        AND R2, R1, R2      // R2 = R2 AND I6
        MOV R1, 247         // R1 = I6
        AND R2, R1, R2      // R2 = R2 AND I7
        SUB R2, R2, R0      // R2 = R2 - 1
        MOV R0, #0          // R0 is the constant "0"
        JMPZ R2, output     // If R2-1==0, then I7..I0 were all 1s
        JMPZ R0, done       // exit program
output: MOV 252, R0         // P4 = 0
done:
```