

Answers

Q3-1.

Most computers use memory mapped I/O because it provides a very simple mechanism for implementing I/O. The processor's own microarchitecture need not actually be changed to support memory mapping and no new instructions need be added to its instruction set. Hence, Implementing I/O because as simple as decoding the address bus and routing the data to the proper address space or device.

Q3-2.

```
ADR r4, ds1
Test: LDR r3, [r4]
      CMP r3, #0
      BE Test
```

Q3-3.

```
Test: R0=DM(ds1);
      COMP(R0, #0);
      IF EQ JUMP Test;
```

Q3-4.

```
ADR r4, ds1
Test: LDR r3, [r4]
      AND r3, r3, #1
      CMP r3, #0
      BE Test
```

Q3-5. We assume that arguments, return values, and return addresses are stored on the stack. We don't use the ARM standard procedure call linkage here for simplicity, since that won't be introduced until Chapter 5.

```
peek: LDR r0, [r13] ;location on the stack for the argument
      LDR r0,[r0]
      STR r0, [r13, -4] ; push back onto stack at return val location
      MOV r15, r14
```

```
poke: LDR r0, [r13]
      LDR r1, [r13, -4]
      STR r0, [r1] ;assumes that location to and date are on the stack
      MOV r15, r14
```

Q3-6.

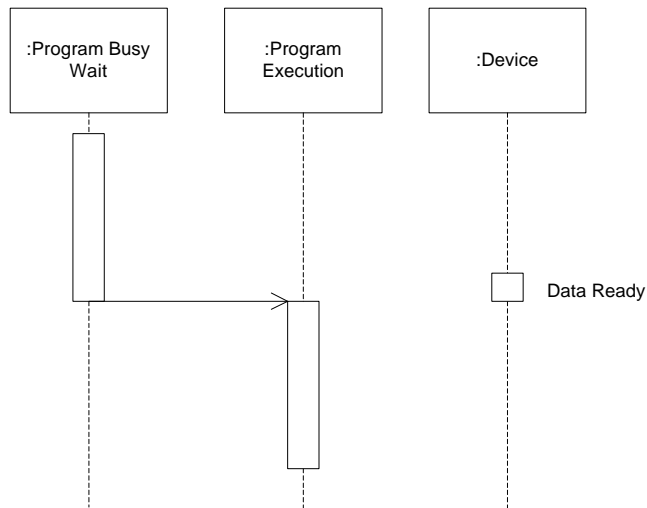
```
r1=1;
test: r0=DM(ds1);
      BIST r0 by r1
      IF NE JUMP test
```

Q3-7. Note: Uses I1 to point to top of stack and assumes that M1=1

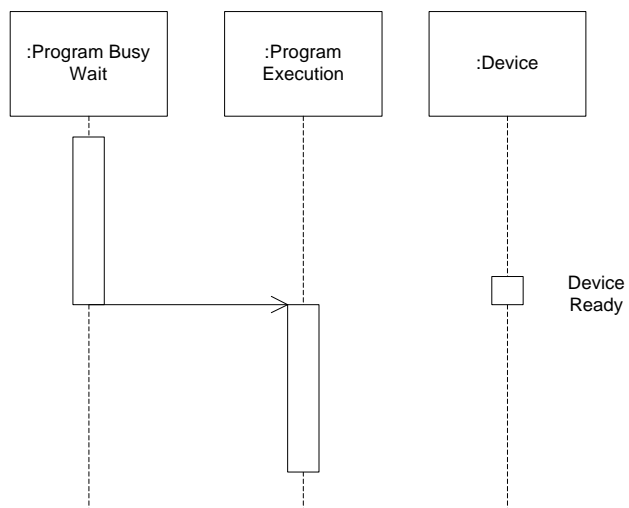
```
peek: R0=DM(I1, -1);  
      R0=DM(R0);  
      DM(I1, -2)=R0;    ! assumes space was set aside on stack  
      RTS;
```

```
poke: R0=DM(I1, -1);    !new value  
      R1=DM(I1, -2);    !location  
      DM(R1)=R0;  
      RTS;
```

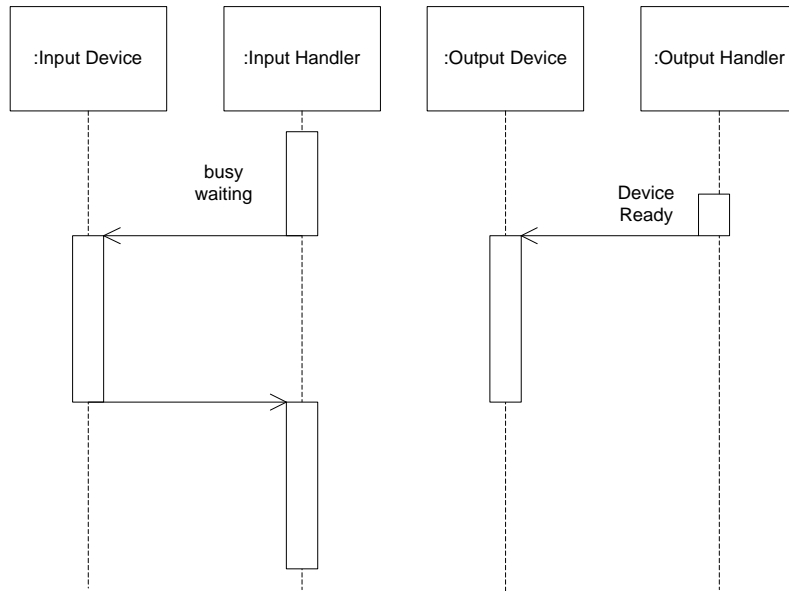
Q3-8.



Q3-9.

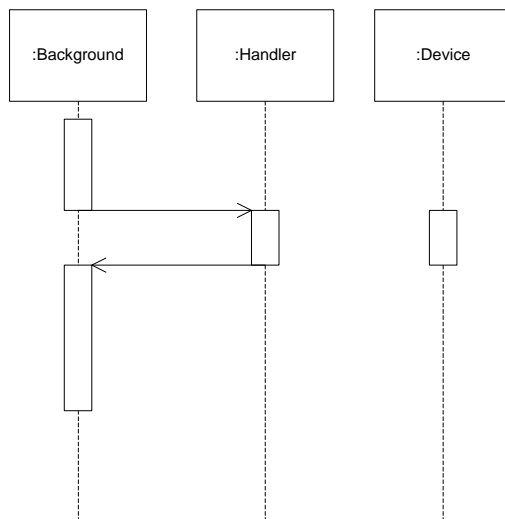


Q3-10.

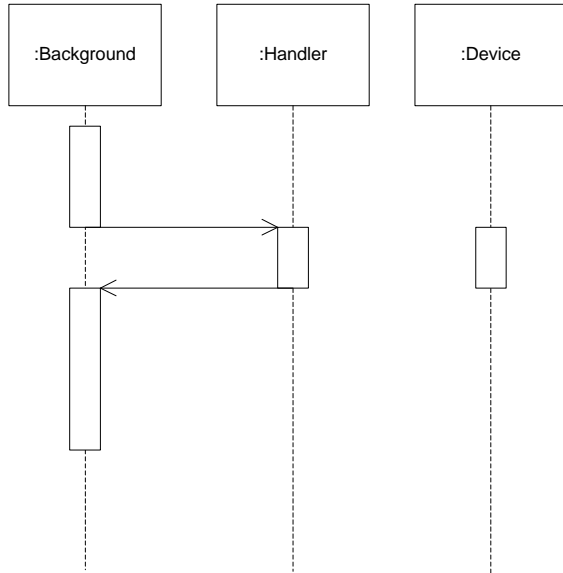


Q3-11. Busy-wait I/O may be the cheapest implementation for systems that need to do only one thing at a time. Interrupt driven I/O is preferable when the I/O is time critical as well as in a multi-threaded process/system.

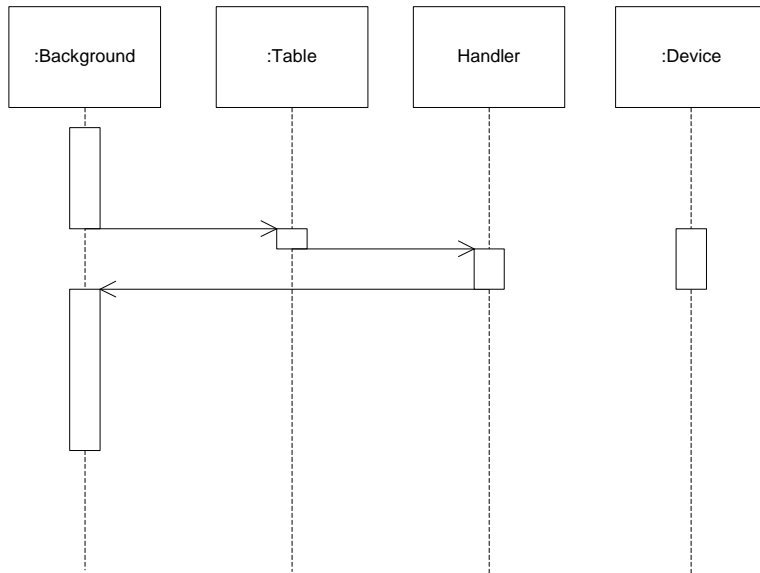
Q3-12.



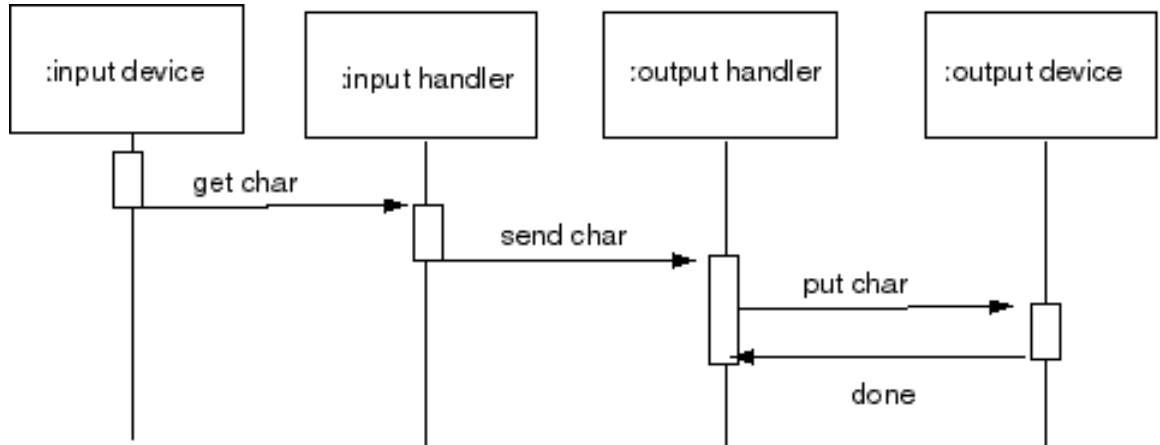
Q3-13.



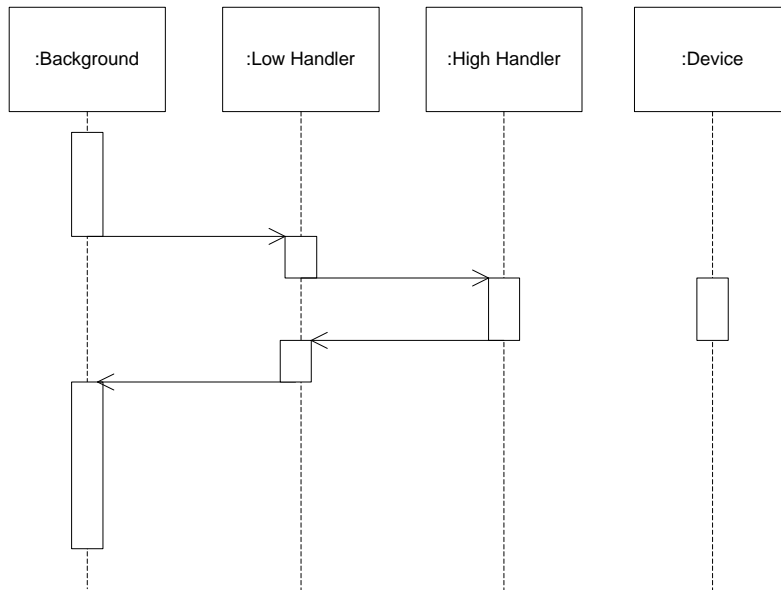
Q3-14.



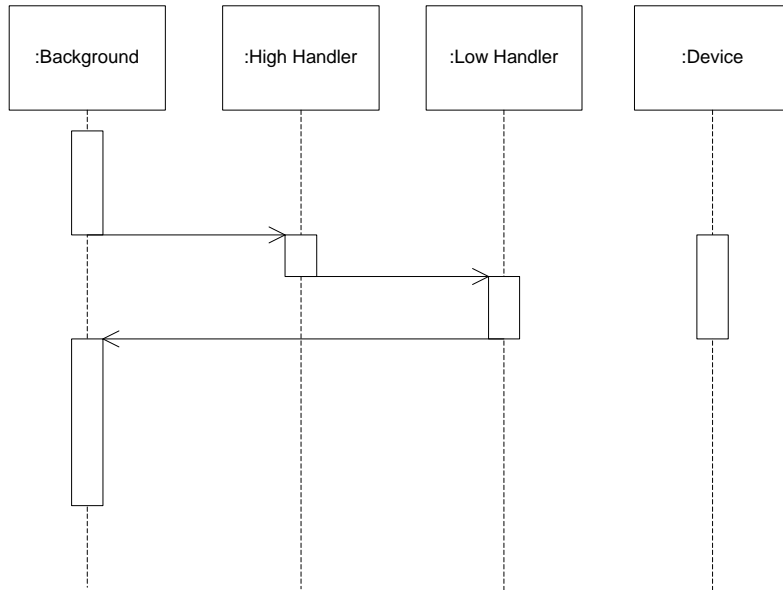
Q3-15.



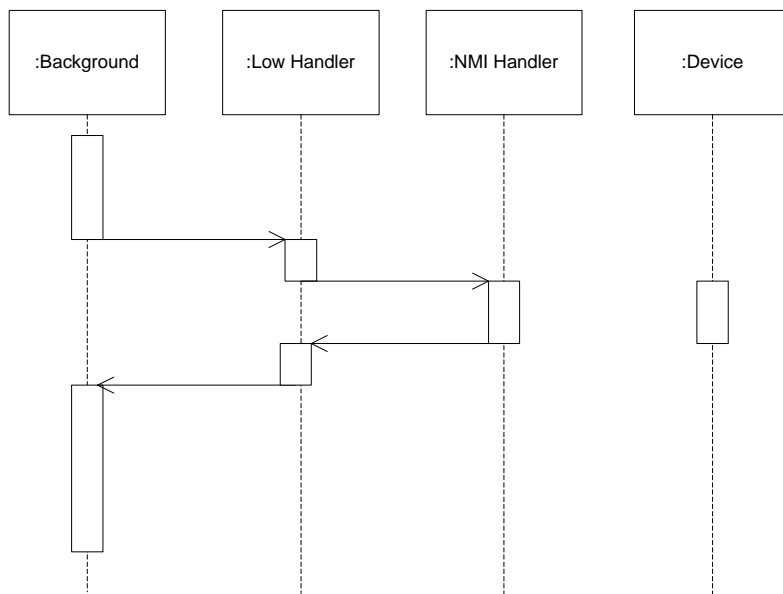
Q3-16.



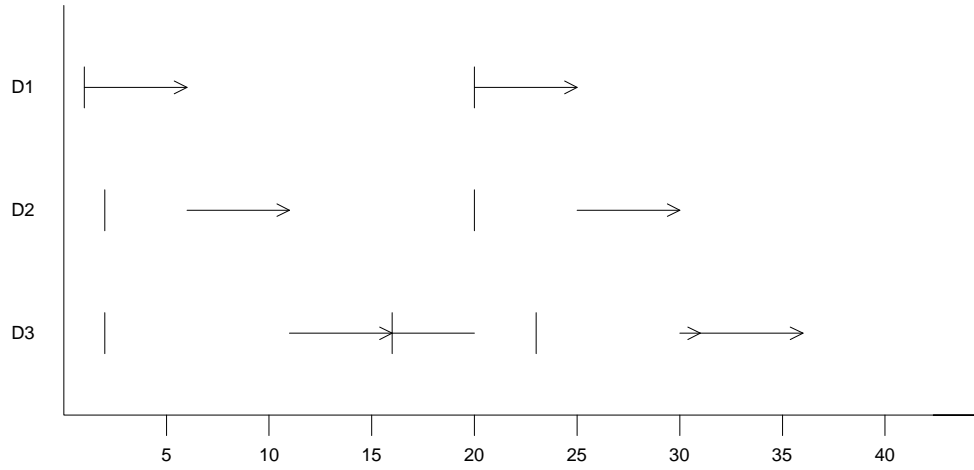
Q3-17.



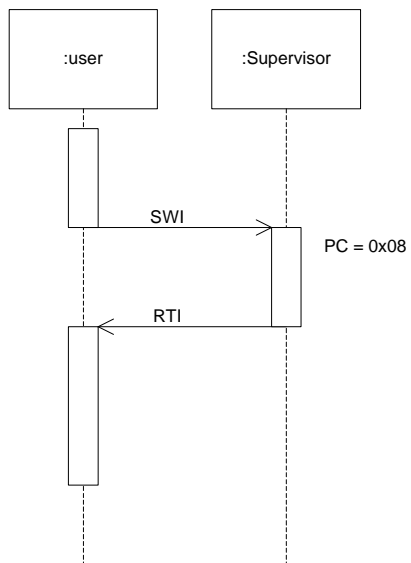
Q3-18.



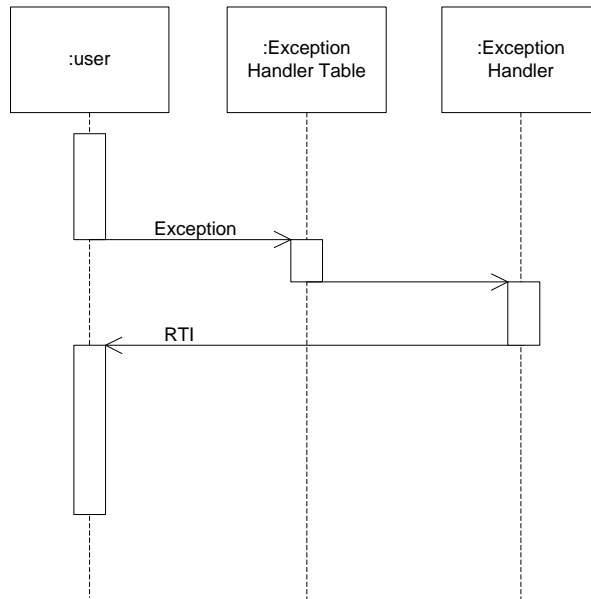
Q3-19.



Q3-20.



Q3-21.



Q3-22.

- ✓ Compulsory Miss – a cache miss that occurs the first time a location is accessed. An example is what will happen at boot time when the cache is completely empty
- ✓ Capacity Miss – a cache miss that occurs when the working set is too large. This type of miss could occur while executing a loop. If the length of the loop is too long, entries that could repeatedly execute will get bumped.
- ✓ Conflict Miss – a cache miss that occurs because 2 memory locations map to the same cache location. This could occur if, for instance, two global variables (or 2 variables in general) were multiples of the cache size. Hence, they could get mapped to the same cache location

Q3-23. Assuming a 1 level cache:

$$\text{Access time} = .93(5) + (.07)(80) = 10.35 \text{ ns}$$

Q3-24.

$$6.5 = 5x + (1-x)80$$

$$6.5 = 80 - 75x;$$

$$x = .98$$

A 98% cache hit rate is required

Q3-25.

$$.9(4) + .1(.97)(15) + .1(.03)(80) = 5.3 \text{ ns}$$

Q3-26.

<i>Address</i>	<i>Data</i>
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

Memory Access: 001, 010, 011, 100, 101, 111

After 001 access:

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	-	-	-	-
01	0	1111	-	-
10	-	-	-	-
11	-	-	-	-

After 010 access:

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	-	-	-	-
01	0	1111	-	-
10	0	0000	-	-
11	-	-	-	-

After 011 access:

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	-	-	-	-
01	0	1111	-	-
10	0	0000	-	-
11	0	0110	-	-

After 100 access:

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	1	1000	-	-
01	0	1111	-	-
10	0	0000	-	-
11	0	0110	-	-

After 101 access:

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	1	1000	-	-
01	0	1111	1	0001
10	0	0000	-	-
11	0	0110	-	-

After 111 access:

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	1	1000	-	-
01	0	1111	1	0001
10	0	0000	-	-
11	0	0110	1	0100

Q3-27.

Assume that the following code fragment was loaded at 0x0000

```

0000      MOV r0, #0
0001      LDR r2, #10
0010      MOV r2, #0
0011      ADR r3, c
0100      ADR r5, x
0101      loop CMP r0, r1
0110      BGE loopend
0111      LDR r4, [r3, r0]
1000      LDR r6, [r5, r0]
1001      MUL r4,r4,r6
1010      ADD r2, r2, r4
1011      ADD r0, r0, #1
1100      B loop

```

a. Direct Mapped, four lines:

<i>Block</i>	<i>Tag</i>	<i>Data</i>
00	11	B Loop
01	10	MUL r4, r4, r6
10	10	ADD r2, r2, r4
11	10	ADD r0, r0 #1

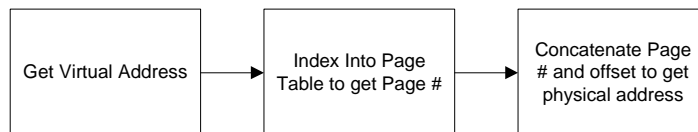
b. Direct Mapped, eight lines:

<i>Block</i>	<i>Tag</i>	<i>Data</i>
000	1	LDR r6, [r5, r0]
001	1	MUL r4, r4, r6
010	1	Add r2, r2, r4
011	1	ADD r0, r0, #1
100	1	B loop
101	0	CMP r0, r1
110	0	BGE loopend
111	0	LDR r4, [r3, r0]

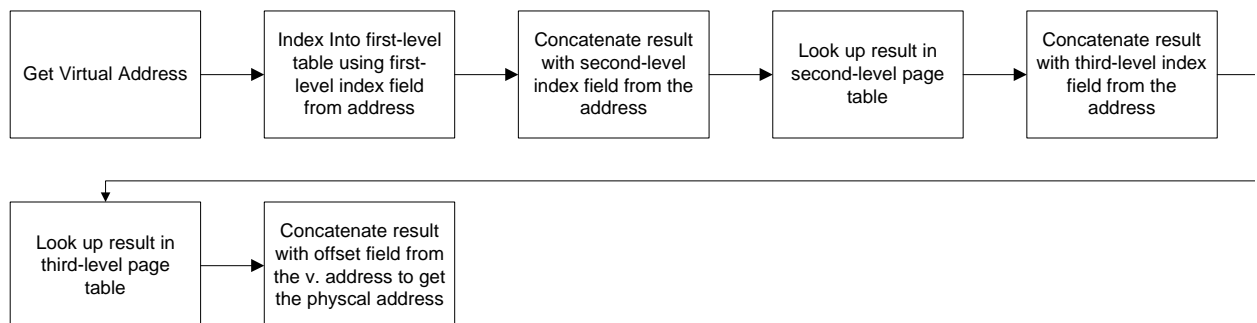
c. Two-way set-associative, four lines per set

<i>Set</i>	<i>Block 0 Tag</i>	<i>Data</i>	<i>Block 1 Tag</i>	<i>Data</i>
00	10	LDR r6, [r5, r0]	11	B loop
01	10	MUL r4, r4, r6	01	CMP r0, r1
10	10	Add r2, r2, r4	01	BGE loopend
11	10	ADD r0, r0, #1	01	LDR r4, [r3, r0]

Q3-28.



Q3-29.



Q3-30.

The three stages are:

- ✓ Fetch: The instruction is fetched from memory
- ✓ Decode: The instruction's opcode and operands are decoded to determine what function to perform
- ✓ Execute: The decoded instruction is executed

Q3-31.

Latency is the average number of clock cycles that it takes an instruction to execute while throughput is the average number of instructions that complete on any given cycle.

Q3-32.

a. Not taken:

BZ loop	Fetch	Decode	Execute			
SUB r2, r3, r6		Fetch	Decode	Execute		
ADD r4, r5, r7			Fetch	Decode	Execute	

b. Taken:

BZ loop	Fetch	Decode	Execute			
SUB r2, r3, r6		Fetch 	Decode 	Execute 		
ADD r4, r5, r7			Fetch 	Decode 	Execute 	
Loop:			Fetch	Decode	Execute	

Q3-33.

Power supply voltage, toggling, and leakage are three mechanisms by which a CMOS processor consumes power.

Q3-34.

- a. An example of static power management is a sleep-mode that a user can enter with an instruction.
- b. An example of dynamic power management is when the software running on a device (like a cellphone) determines that certain parts of the system (like the radio) are not currently needed. The software is then able to take steps to turn the unneeded components off.

Q3-35.

In a sleep state, the CPU is actually in reset. Hence, there must be a way to save state before sleep and to restore it afterwards. This is in contrast to an Idle state where the CPU is essentially stalled and can easily continue execution by restarting its clock.