**CS 201 Section 1 Homework 3 Suggested Solutions**

1. (5.1) Here's one solution, using the notation of Fig 5-12. Note that only one expanding op-code is needed; there's plenty of space to encode the 1-address and 0-address instructions in a more straightforward manner. I've used a single bit of opcode to distinguish these two classes, which should make it easier to decode the layout of the remainder of the instruction bits.

```
           ----------------------------------------
           | 000xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyrrr |
  3-bit    | 001xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyrrr |  7 2-address
  opcode   | .                                    |  instructions
           | .                                    |
           | .                                    |
           | 110xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyrrr |
           ----------------------------------------
           | 1110000000000-----yyyyyyyyyyyyyyyrrr |
  13-bit   | 1110000000001-----yyyyyyyyyyyyyyyrrr |  500 1-address
  opcode   | .                                    |  instructions
           | .                                    |
           | .                                    |
           | 1110111110011-----yyyyyyyyyyyyyyyrrr |
           ----------------------------------------
           | 1111000000---------------------------|
  10-bit   | 1111000001---------------------------|
  opcode   | .                                    |  50 0-address
           | .                                    |  instructions
           | .                                    |
           | 1111110010---------------------------|
           ----------------------------------------
```

2. (5.4)

There are $16 - 2 \times 6 = 4$ bits available to hold the two-address instruction opcodes together with any expansion "escape" values used to indicate that the remainder of the opcode is elsewhere. That means we can have $n - 2^4$ different escape values. There are 6 additional bits available to hold extended opcodes for one-address instructions. Thus we can have $2^6 = 64$ one-address instructions for every different escape value, yielding a total of $64 \times (16 - n)$ one-address instructions. Note, therefore, that if there are to be any one-address instructions at all, there can be at most 15 two-address instructions.

3. (5.6)

We assume that the variables `A,B,C,D,E,F,X` are all in memory, and (except for `X`) their values must not be altered by the computation. Also, we assume the usual order of operations (multiplication and division before addition and subtraction), and we make use of the commutativity of addition.

We consider each machine in turn.

**0-address machine**

(Assume that the second operand for each binary operator is at the top of the stack, with the first operand just below it.)

Code:

```
PUSH A
PUSH B
PUSH C
MUL
ADD
PUSH D
PUSH E
PUSH F
MUL
SUB
DIV
POP X
```

Instruction space costs:

`ADD,SUB,MUL,DIV` : 8 bits for opcode = 8 bits total
`PUSH, POP`: 8 bits for opcode + 16 bits for address = 24 bits total.

Cost of this sequence: $5 \times 8 + 7 \times 24 = 208$ bits.

**1-Address Machine**

Code (using `X` as a temporary location):

```
LOAD E
MUL F
STORE X
LOAD D
SUB X
STORE X
LOAD B
MUL C
ADD A
DIV X
STORE X
```

Instruction space costs:

Each instruction: 8 bits for opcode + 16 bits for address = 24 bits total.

Cost of this sequence: $11 \times 24 = 264$ bits.

**2-Address Machine**

Code:

```
MOV R1,B
MUL R1,C
ADD R1,A
MOV R2,E
MUL R2,F
MOV R3,D
SUB R3,R2
DIV R1,R3
MOV X,R1
```

Instruction space costs:

Instructions with two register operands: 8 bits for opcode + 4 bits for each register = 16 bits
Instructions with one register operand and one address operand: 8 bits for opcode + 4 bits for register + 16 bits for address = 28 bits

Cost of this sequence: $2 \times 16 + 7 \times 28 = 228$ bits

**3-Address Machine**

Code:

```
MUL R1,B,C
ADD R1,A,R1
MUL R2,E,F
SUB R2,D,R2
DIV X,R1,R2
```

Instruction space costs:

Instructions with two register operands and one address: 8 bits for opcode + 4 bits for each register + 16 bits for address = 32 bits
Instructions with one register operand and two address operands: 8 bits for opcode + 4 bits for register + 16 bits for each address = 44 bits

Cost of this sequence: $3 \times 32 + 2 \times 44 = 184$ bits

4. (5.26)

We use the symbol $\oplus$ to mean XOR. Recall the truth table for XOR is

| $a$ | $b$ | $a \oplus b$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The word-level `XOR` instruction available on many machines simply performs the a bit-wise $\oplus$ operation in parallel on each bit of the word.

A key property of $\oplus$ is that, for any $a$ and $b$, $(a \oplus b) \oplus b = a$. Also, $\oplus$ is associative and commutative.

Thus, the following sequence of assignments will have the effect of swapping the contents of `A` and `B`. The comments show the contents of `A` and `B` after each step, assuming that initially `A` contains $a$ and `B` contains $b$:

```
A := A ⊕ B   /* A = a ⊕ b; B = b */
B := A ⊕ B   /* A = a ⊕ b; B = (a ⊕ b) ⊕ b = a */
A := A ⊕ B   /* A = (a ⊕ b) ⊕ a = (b ⊕ a) ⊕ a = b; B = a */
```

5. If no `sethi` instruction were available, we could still use a series of `mov`,`or`, and `sll` operations to load an arbitrary 32-bit constant into a register. The key point is that we have to divide the constant into three pieces, each no more than 13 bits long, which can then be represented as immediate values within arithmetic or logical operations. There are several essentially similar ways to do this; let's assume that we choose pieces of length 11 bits, 11 bits, and 10 bits.

Suppose the constant we want to load is $c$. Then if we define the following pseudo-instructions (much like the existing `%hi`):

```
%hi11(c)  = (c & 0xffe00000) >> 21
%mid11(c) = (c & 0x001ffc00) >> 10
```

together with the existin pseudo-instruction

```
%lo(c)  = (c & 0x000003ff)
```

where `&` means bitwise AND and `>>` means logical right shift (without sign extension), we can load $c$ into register $r$ by the following sequence:

```
mov %hi11(c),r
sll r,11,r
or r,%mid11(c),r
sll r,11,r
or r,%lo(c),r
```