EECS 470 Homework 5

1. In class we discussed the following code fragment:

```
        r1=MEM[r2+0]        //A
        r1=r1*2             //B
        MEM[r2+0]=r1        //C
        r2=r2+4             //D
        bne r2 r3 Loop      //E
```

   We showed that using software pipelining it could be converted to:

```
        r4=MEM[r2+0]        //A1
        r1=r4*2             //B1
        r4=MEM[r2+4]        //A2
Loop:   MEM[r2+0]=r1        //C(n)
        r1=r4*2             //B(n+1)
        r4=MEM[r2+8]        //A(n+2)
        r2=r2+4             //D(n)
        bne r2 r3 Loop      //E(n)
```

   a) As discussed in class, the above loop will execute the load two extra times, and multiplication one extra time. Rewrite the loop so that software pipelining is still in use, but each of the instructions are executed the correct number of times. [3]
   b) Another problem with the above code is that currently the pipelined code fragment cannot execute an iteration of D and one of E in the same loop body. Unroll your solution to a) once to fix this problem. [5]

2. Rewrite using the following code fragment using CMOV so that it has the same exception behavior but does not branch. Use as few additional registers as possible. You may not perform any more loads or stores than required. You may assume that loading from address 11000 will not cause any exceptions. [4]

```
        if(R2>10000)
              R3=MEM[R2+R1]
        else
              R3=100
```

3. 4.21 [4]

4. Consider the example that starts on page 317. The caption of figure 4.5 claims that it requires the use of at least 8 floating point registers.
   a) Is that correct? Explain your answer. [2]
   b) In that example the loop was unrolled 7 times, took 9 cycles to execute 23 instructions, uses 15 floating-point registers and requires some number of floating-point registers (from part a). How would those numbers change if it were unrolled 9 times? [4]

5. Define the following terms and why they are important to compilers. [3]
    a) Live value
    b) Register spill
    c) Register preasure
    d) Caller/callee save registers

6. Consider the following C code:

```
for ( i = 0; i < MAX; i++ ) {
            a[ i ] = a[ i ] + b[ i ];
      } //end for
```

That C code is translated into the following x86-like assembly language:
(note: the ++ indicates the "autoincrement" addressing mode. See page 98 if needed.)
You can probably find the answer to this by doing a web search. At least try to do it yourself first…

```
      mov   r1, addr( a )  -- address of a[ 0 ] into r1
      mov   r2, addr( b )  -- address of b[ 0 ] into r2
      mov   rx, MAX        -- Number of iterations into rx
l1:
      ld    r3, (r1)       -- load indirect into r3 through r1
      ld    r4, (r2)++     -- what r2 points to loaded in r4
      fadd  r5, r3, r4     -- r5 holds sum of two elements
      st    r5, (r1)++     -- store result and post-increment
      loop  l1             -- does an autodecrement (by 1) of rx
                           -- if rx isn't zero branches to l1
```

And then that assembly code is software pipelined.

```
-- Initialization:
      mov   r0, addr( a )     -- r0 is pointer to a[0]
      mov   r1, r0            -- copy address of a[0] into r1
      mov   r2, addr( b )     -- r2 is pointer to b[0]
      ___blank A_______
      ___blank B_______
      ___blank C_______
      fadd  r5, r3, r4
      ld    r3, (r1)++
      ld    r4, (r2)++
l2:   st    r5, (r0)++
      fadd  r5, r3, r4
      ld    r3, (r1)++
      ld    r4, (r2)++
      loop  l2                -- decrement rx, if != 0 jump to l2
      ___blank D_______
      fadd  r5, r3, r4
      st    r5, (r0)
```

a) Supply the missing code for each blank [1 point each]
b) If, in the original C code, MAX is ***less than*** _____ the software-pipelined loop will behave incorrectly. [1]