CSE 30321 – Computer Architecture I – Fall 2010
Homework 01 – Introduction to Programmable Processors – 75 points
Assigned: August 31, 2010 **– Due:** September 7, 2010

Problems 1-4a are designed to review and reinforce fundamental tasks and concepts covered in Lectures 02 and 03. Problem 4b-e and Problem 5 extend this material slightly. For these problems, the emphasis is not so much on writing syntactically correct assembly code, etc. Instead, they have been designed to emphasize how the design of an instruction set architecture (ISA) can ultimately impact performance of (even relatively simple) code written in a high-level language. These problems emphasize the importance of an ISA as the hardware-software interface.

Problem 1: (10 points)
Machine code is a useful abstraction in that it gives you a feel for how some pseudo code might actually be executed on a given microprocessor, yet also abstracts away some of the "gory detail" associated with the process (i.e. a "1s" and "0s" representation). In this question we start with machine code, "look up" in Part A, and "look down" in Part B.

Assume that you have the following sequence of instructions (all are for the 6-instruction processor discussed in lecture):

0. MOV R4, #9
1. MOV R1, #4
2. MOV R2, #5
3. SUB R3, R2, R1
4. JUMPZ R3, 4
5. MOV R4, #7
6. MOV 20, R4
7. JUMPZ R0, 2
8. MOV 21, R4
9. <out of code>

Part A: (6 points)
After the above instructions have been executed, what has been done?

(please answer in terms similar to: d(x) = d(y) + d(z) OR x = y * z – in other words, don't just translate the instructions to register transfer language, but think about what they might do at a higher-level)

Part B: (4 points)
Write the machine code for the instructions at addresses 3-through-6. Please answer in hex.

Problem 2: (10 points)

The question below requires you to write some 6-instruction processor code. It is included to give you more practice working with instruction mnemonics. You might want to refer to the datapath diagram for the 6-instruction processor in your notes (or the copy linked on the website).

Translate the following C statement to 6-instrution processor assembly language. Be sure to pay attention to order of operations!

```
y = a * b + c * d – p + e * f + t;
```

You should assume that c, e, and t are at memory locations 30, 36, and 50, and that Y maps to R10. You may also assume that other variables have already been mapped to registers. The original value(s) of a given variable does not have to be preserved. Do this with as few instructions as possible. Report the total number of different registers used (lower is better).

Hint: Construct your answer as shown below. More detailed comments are well-correlated to more partial credit.

```
MULT  R1, R1, R2          # R1 ← R1 * R2        (a ← a * b)
```

Problem 3: (10 points)
The questions listed below all require short answers. The primary purpose of these questions is to consider how simpler operations (instructions) can – or can't – execute statements that you might write in some high-level language.

Question A: (5 points)
Would it be practical/feasible to use 6-instruction processor instructions to write/generate assembly language that does the following:

```
if (x < 10)
      y = x * z;
else
      y = x + z;
```

Note that you don't have to write the code itself, just explain – although you are welcome to show code snippets to explain your answer and/or for partial credit. Also, if it is *not* possible: (a) explain why and (b) suggest instruction support that would make it possible.

Question B: (5 points)
Would it be practical/feasible to use 6-instruction processor instructions to write/generate assembly language that does the following:

```
i = 30;
while (i < 36)
      y(i) = x + z;
      i = i + 1;
}
```

Note that you don't have to write the code itself, just explain – although you are welcome to show code snippets to explain your answer and/or for partial credit. Also, if it is *not* possible: (a) explain why and (b) suggest instruction support that would make it possible.

Problem 4: (35 points)

The focus of the questions below is architectural and hardware support for high-level language programming constructs. (In other words, if there are constructs in a high-level language that are frequently used, it's generally a good idea to support the realization of said constructs with appropriate hardware … and the instruction set architecture is this interface.)

To begin, below, I've written 6-instruciton processor assembly language that is equivalent to the following C for loop. Refer to this code to answer Questions A and B.

C-code

```
for(i=0; i<5; i++) {
      x = x + x;
      x = x + y;
}
```

6-instruction code

```
      MOV R1, #1
      MOV R10, #0
      MOV R5, #5
P:    ADD R2, R2, R2
      ADD R2, R2, R3
      ADD R10, R10, R1
      SUB R6, R5, R10
      JUMPZ R6, Q
      JUMPZ R0, P
Q:
```

Question A: (7 points)

Any kind of loop structure is an excellent example of a construct that you frequently use in code that is written in a high-level language. We would like to improve the performance of loops by expanding the existing 6-instruction processor ISA. **Using the code above as example, suggest a *new* instruction that can be used to reduce the number of clock cycles required to execute the above for loop in 6-instruction processor assembly.** (Adding a new instruction is possible because, thus far, we have only used 6 of the possible 16 opcodes.) Note that the hallmark of a "good" instruction is broad applicability and not something that is specific to a particular piece of code. (So keep this in mind when you answer this question!) Other instructions could be re-written if necessary to better leverage your new instruction. **Be sure to explain how your new instruction helps.**

Question B: (8 points)

How many cycles were eliminated? (Hint: Calculate the number of cycles required for the original code, and then show how many cycles are required with your new code.)

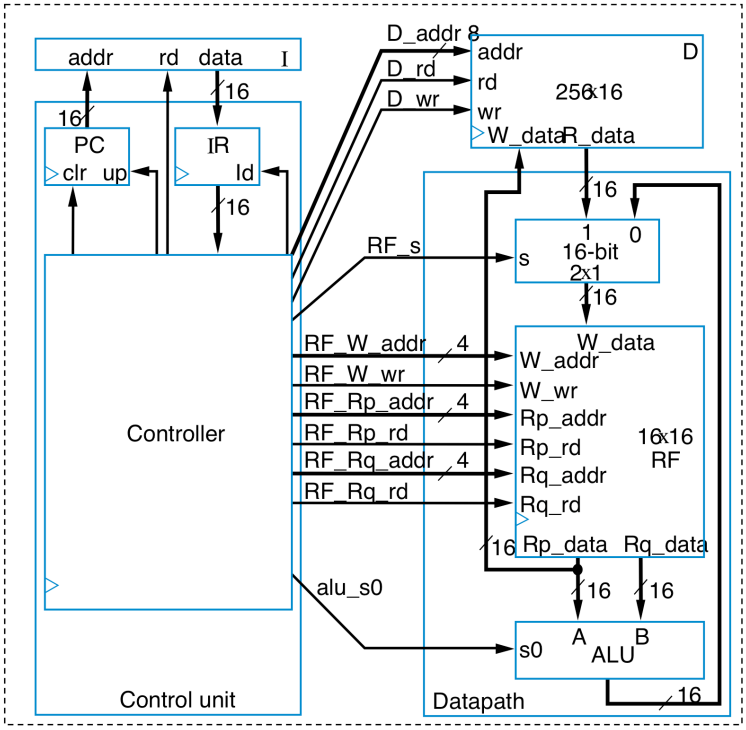
Question C: (8 points)
What does the code shown below accomplish? You should answer in terms of some high-level language pseudo code. Your variables names can simply be the register names/numbers used. Again, don't simply translate the code. Think about the higher-level functionality.

Address	Instruction
0	MOV R10, #1
1	MOV R1, #27
2	MOV R2, #31
3	MOV R3, 27
4	MOV R4, 3
5	ADD R4, R4, R10
6	MOV 3, R4
7	ADD R9, R3, R9
8	ADD R1, R1, R10
9	SUB R6, R2, R1
10	JUMPZ R6, 2
11	JUMPZ R0, -8
12	MOV 40, R3

Question D: (7 points)
You can assume that the assembly code shown below does the exact same thing that the code in Question C does. Note that a new type of instruction has been added. (I will leave it for you to figure out where, but it should be fairly obvious as there is at least one instance of an instruction whose syntax does not match that shown in your class notes.) Explain what the new instruction does.

Address	Instruction
0	MOV R10, #1
1	MOV R1, #27
2	MOV R2, #31
3	MOV R3, R1
4	ADD R9, R3, R9
5	ADD R1, R1, R10
6	SUB R6, R2, R1
7	JUMPZ R6, 2
8	JUMPZ R0, -5
9	MOV 40, R3

Question E: (5 points)
Describe how the datapath that we discussed in lecture would need to be modified to support the new instruction used in Part D. For reference, a picture is shown below.


addr
rd
data
I
16
16
PC
clr
up
IR
ld
16
Controller
Control unit
D_addr 8
D_rd
D_wr
addr
rd
wr
W_data
D
256x16
R_data
16
RF_s
1
0
s
16-bit
2x1
16
RF_W_addr
4
RF_W_wr
RF_Rp_addr
4
RF_Rp_rd
RF_Rq_addr
4
RF_Rq_rd
W_data
W_addr
W_wr
Rp_addr
Rp_rd
Rq_addr
Rq_rd
16x16
RF
16
Rp_data
Rq_data
alu_s0
16
16
s0
A
B
ALU
16
Datapath

Problem 5: (10 points)

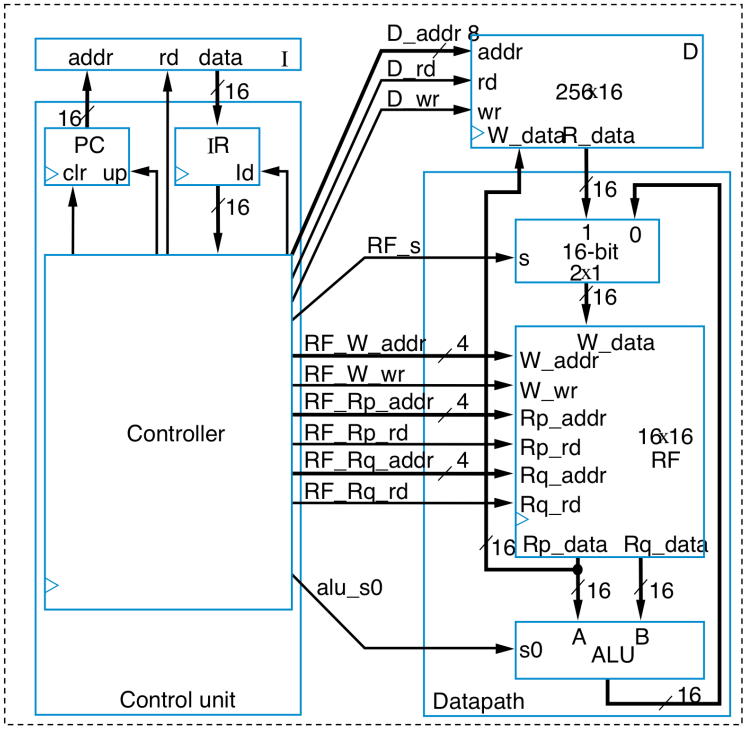
A common operation in digital signal processing (DSP) applications is "multiply and accumulate" (MAC). As such, in processors that are used heavily for DSP, it is common to support this operation architecturally. (In other words, there will be a multiply and accumulate instruction.)

Question A: (5 points)

Assume that the register transfer language for a MAC instruction in the context of the 6-instruction processor is:

MAC R1, R2, R3 # R1 ← R1 + (R2*R3)

Describe how the datapath that we discussed in lecture would need to be modified to support the MAC instruction. For reference, a picture is shown below.



Question B: (5 points)

Given the code below, how many CCs are saved by using a MAC instruction. Note: You may assume that the MAC takes 4 CCs to execute. Also, you may presume the existence of a 3 clock cycle multiply instruction in addition to a 3 clock cycle ADD and SUB. Finally, you *do not* have to write any assembly code for this question. An answer in the form of a parameterized expression is sufficient.

```
for (i=0; i<N; i++) {
      x = x + (y*z);
}
```