CSE 30321 – Computer Architecture I – Fall 2011

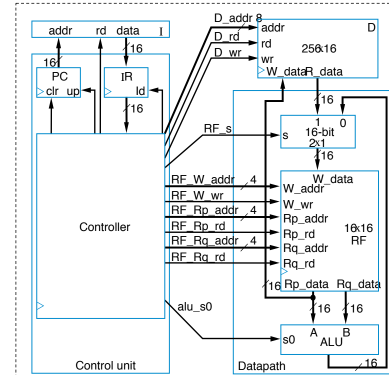
Homework 01 – Introduction to Programmable Processors – 70 points

Assigned: August 30, 2011 **– Due:** September 6, 2011

Useful Information

For all of the problems shown below, it may be useful to refer to (a) the instruction set for the 6-instruction processor discussed in class and (b) the datapath diagram associated with this ISA. Copies are also included here for your reference

Instruction	Meaning	Opcode
Mov Ra, d	RF[a] = D[d]	0000
Mov d, Ra	D[d] = RF[a]	0001
Add Ra, Rb, Rc	RF[a] = RF[b] + RF[c]	0010
Mov Ra, #C	RF[a] = #C	0011
Sub Ra, Rb, Rc	RF[a] = RF[b] - RF[c]	0100
Jmpz Ra, offset	PC = PC + offset if RF[a] = 0	0101



Problem 1: (20 points)

Machine code is a useful abstraction in that it gives you a feel for how some pseudo code and/or high-level language (HLL) 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).

Assume that you have the following sequence of instructions

- All are for the 6-instruction processor discussed in lecture…

```
        MOV     R1,    #7
        MOV     R2,    #8
        MOV     R3,    #7
*1      MOV     R10,   #24
*2      SUB     R4, R3, R3
        SUB     R5, R1, R4
        JUMPZ   R5, P
        SUB     R5, R2, R4
        JUMPZ   R5, Q
        ADD     R11, R10, R3
        JUMPZ   R0, END
P:      ADD     R11, R3, R3
        JUMP    R0, END
Q:      ADD     R11, R3, R10
END:
```

Part A: (10 points)

After the above instructions have been executed, what has been done?

- You should answer in terms similar to: d(x) = d(y) + d(z), x = y * z, etc.
- In other words, don't just translate the instructions to register transfer language, but think about what they might represent in terms of some HLL construct; in other words, DON'T just explain how data is moved around in registers/memory locations.
- Assume that R4 maps to the variable *x* and R11 maps to variable *y.*
- To simplify your answer, I have added explicit labels in the JUMPZ instructions. This avoids the trouble of computing addresses (which is an easy way to make a mistake).
- You should assume that R0 always has the value 0.

Part B: (4 points)

Write the machine code for the instructions *1 and *2. Please answer in hex.

Part C: (6 points)

How many CC's are required to execute the assembly code in Part C?

- Your answer should consider what instructions are actually executed, not the time that would be required if all instructions were executed.
- Remember that each instruction takes multiple CCs.

Problem 2: (10 points + 3 bonus 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.

Translate the following C code to 6-instruction processor assembly language.

```
i = 24;
while (i < 35) {
        y(1) = y(1) + i;
        i = i + 1;
}
```

- An EXACT translation is not required; your 6-instruction processor assembly code simply needs to provide the same functionality – however, you CANNOT make 10 copies of the loop body…

- As in problem 1, you can just use a label in a JUMPZ instruction if this type of instruction is required in your answer.

- When writing your assembly code, think about how you might reduce the number of instructions that are executed – and hence overall execution time. For this problem, there is at least 1 way to significantly reduce execution time. Answer that incorporate this solution will receive 3 bonus points.

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

```
Add     R1, R1, R2          # R1 ← R1 + R2       (a ← a + b)
```

Problem 3: (15 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. The instruction set architecture is this interface.

For example, a useful addition to the 6-instruction processor ISA might be a new type of load or store instruction like that specified below:

Opcode	Syntax	Register transfer language and encoding	Example
1110	LOAD Rw, Rp	Rw ← Memory(Rp) 1110 I wwww I pppp I unused	Assume - Rw is R1 - Rp is R2 - R2 contains the number 9 - Memory address 9 contains the number 18 This instruction will then: - Send address 9 to memory - The data contained in address 9 – the number 18 – will be copied to register R1
1111	STORE Rp, Rq	Memory(Rp) ← Rq 1111 I unused I pppp I qqqq	Assume - Rp is R1 - Rq is R2 - R1 contains the number 12 - R2 contains the number 24 This instruction will then: - Store the number 24 at memory location 12.

Part A: (7 points)

Describe how the datapath on page 1 would need to be modified to support these new instructions.

- Note that this is the same 6-instruction datapath discussed in lecture.
- For reference, a picture is also linked on the course website.

Part B: (8 points)

Explain why these 2 additions to the original 6-instruction ISA are useful?

- Hint:
    - Think about what common HLL statements might be enabled – and that are either (a) not possible or (b) extremely difficult to do given only the initial 6 instructions. Providing this type of example is a good way to answer this question.

Problem 4: (5 points)

The focus of the question below is also on architectural and hardware support for high-level language programming constructs.

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;
```

Problem 5: (20 points)

In this last question, we consider a more sophisticated ISA – namely a subset of the ARM ISA.

- How common are ARM chipsets? Per the Wall Street Journal, "approximately 95% of the world's mobile handsets and more than one-quarter of all electronic devices use an ARM chip."[1]
- A subset of the ARM ISA is shown in the table below – note that many arithmetic instructions (e.g. ADD, SUB) are quite similar to their 6-instruciton ISA counterpart.
- However, the instructions that access memory, conditional branch instructions, etc. are more sophisticated than their 6-instruction counterparts.
- Examples that illustrate what each ARM instruction does are also included in the table below. Be sure that you understand how each instruction works, and then answer the questions below.

Q: Why the change in ISAs?

A: You should become comfortable with interpreting machine code for different microprocessor architectures. A goal of this course is not to teach you MIPS or ARM assembly, but rather to (a) understand how the machine instructions accomplish some set of tasks specified by HLL code and (b) understand how changing code written in some HLL and/or a given processor-memory configuration can affect performance.

---

[1] http://blogs.wsj.com/tech-europe/2010/11/19/intel-microprocessor-business-doomed-claims-arm-co-founder/

Instructions:

Instruction	Syntax	What happens…	Comments
ADD	ADD R1, R2, R3	R1 ← R2 + R3	• Assume R2 = 10, R3 = 20 • 30 will be stored in R1
	ADD R1, R2, 100	R1 ← R2 + 100	• Assume R2 = 10 • 110 will be stored in R1 • (It is possible to encode a constant value directly in the instruction itself.)
B	B <addr>	PC ← PC + <addr>	• This is an "unconditional" branch instruction – the value of the PC is automatically updated. • Assume PC = 100, <addr> = 200 • The next instruction will be fetched from address 300 • (i.e. PC = 300)
B <CC>	BGT <addr>	If the GT (greater than) flag is set: • PC ← PC + <addr> Otherwise: • The value of the PC is unchanged	• <CC> stands for "condition code" • Condition codes are set after certain instructions are executed and can be thought of as TRUE or FALSE flags. • Other common condition codes – and hence instructions – are shown in the table below.
CMP	CMP R1, R2	Status = R1-R2 • Status flags are updated accordingly	Example: • Assume R1 = 10, R2 = 9 • Status = 1 • GT = TRUE (R1 is > R2) • GE = TRUE (R1 is ≥ R2) • LT = FALSE (R1 is not < R2) • LE = FALSE (R1 is not ≤ R2) • EQ = FALSE (R1 ≠ R2)
	CMP R1, #	Status = R1 – #	• Like above, but R9 replaced with constant value
LDR	LDR R2, [R1]	R2 ← Memory(R1)	• R1 contains the address that is sent to memory • The data at the address specified by R1 is stored in R2
	LDR R2, [R1, 4]	R2 ← Memory (R1 + 4)	• The address sent to memory is the sum of the data in R1 + the constant 4 • The data at the address specified by R1 + 4 is stored in R2
MUL	MUL R1, R2, R3	R1 ← R2 x R3	• Assume R2 = 10, R3 = 20 • 200 will be stored in R1
STR	STR R2, [R1]	Memory(R1) ← R2	• R1 contains the address that is sent to memory • The data stored in R2 is copied to the memory address stored in R1
	STR R2, [R1, 4]	Memory (R1 + 4) ← R2	• The address sent to memory is the sum of the data in R1 + the constant 4 • The data stored in memory is contained in R2
SUB	SUB R1, R2, R3	R1 = R2-R3	• Assume R2 = 30, R3 = 20 • 10 will be stored in R1

Condition codes:

Code	Explanation
GT	If a number is greater than a reference number, the GT flag is TRUE, otherwise it is FALSE
GE	If a number is greater than or equal to a reference number, the GE flag is TRUE, otherwise it is FALSE
LT	If a number is less than a reference number, the LT flag is TRUE, otherwise it is FALSE
LE	If a number is less than or equal to a reference number, the LE flag is TRUE, otherwise it is FALSE
EQ	If a number is equal to a reference number, the EQ flag is TRUE, otherwise it is FALSE

<u>Part A</u>: (5 points)

- Assume that you want to compare the data stored in R3 to the data stored in R4
  - If R4 is less than or equal to R3, you should add R10 to R11 and store the result in R12
  - Otherwise, you should subtract R10 from R11 and store the result in R12

Write the ARM assembly language to do this. Use as few of instructions as possible from the subset described above.

<u>Part B</u>: (5 points)

Write ARM assembly for the C code given in Problem 4 (and reproduced here).

- You should assume that:
  - x maps to R1
  - y maps to R2
  - z maps to R3

- C code:

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

<u>Question C</u>: (10 points)

Explain what the following sequence of ARM instructions does. (Use HLL / pseudo code.)

```
        ADD    R1, R0, #100
Q:      CMP    R1, #1000
        BGT    P
        STR    R1, R1
        ADD    R1, R1, #100
        B      Q
P:
```