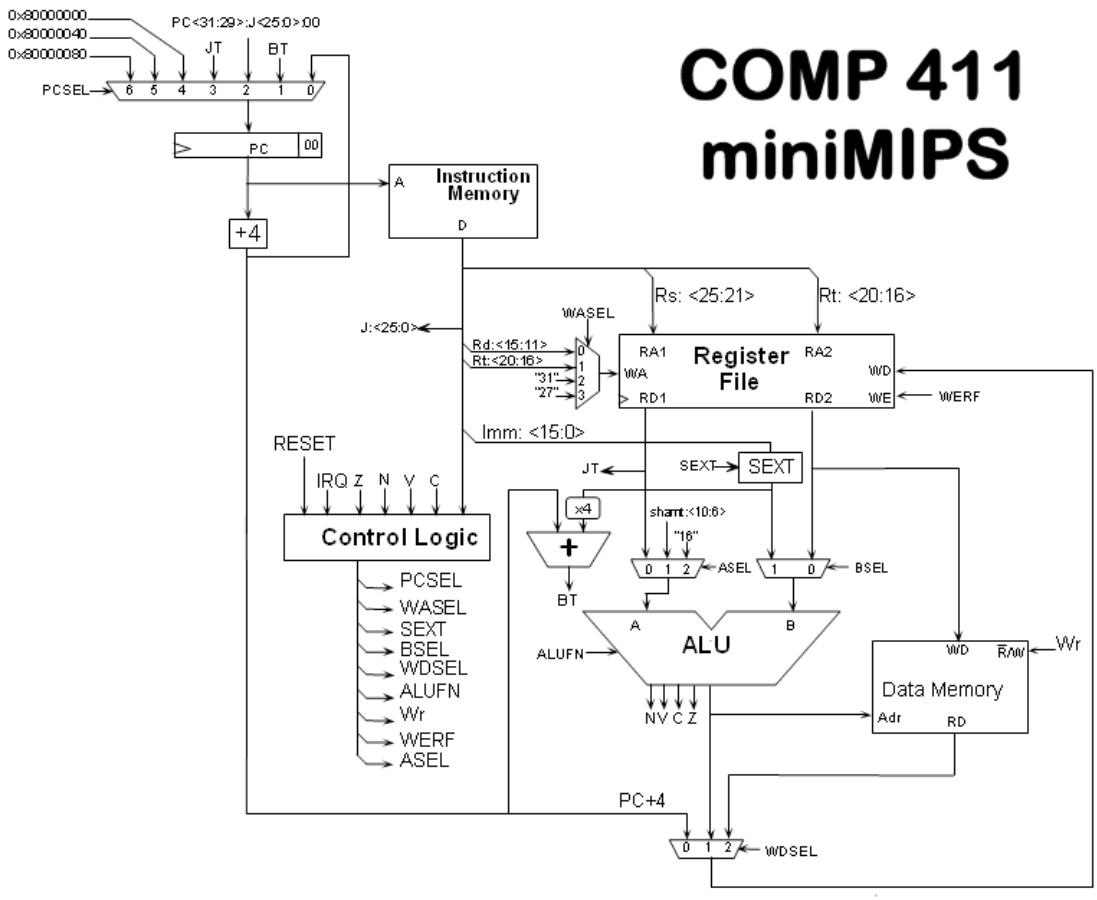
The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 411 Computer Organization
Fall 2008

Problem Set #7
Issued Thursday, 11/6/08; Due Thursday, 11/13/08

Homework Information: Some of the problems are probably too long to be done the night before the due date, so plan accordingly. Late homework will not be accepted. Feel free to get help from others, but the work you hand in should be your own.

Problem 1. "Out of Control" (20 points)



Fill in the missing entries of the Control Logic ROM in the table below, based on the data path shown above.

Opcode	PCSEL	WASEL	SEXT	BSEL	WDSEL	ALUFN Sub	ALUFN Bool	ALUFN Shft	ALUFN Math	Wr	WERF	ASEL
`sub`	0	0	X	0	1	1	XX	0	1	0	1	0
`xor`	0	0								0		
`addi`	0	1										
`sll`	0	0	X	0	1					0	1	
`andi`	0									0		
`lw`					2					0		
`sw`										1		
`j`						X	XX	X	X	0		
`jal`						X	XX	X	X	0		
`lui`										0		

Problem 2. "Simplified Shifts" (40 points)

Lori Acan, a budding computer architect, realized that the hardware implementation of the `sll`, `slr`, `sar`, and `lui` instructions could be simplified, if they were encoded as follows:

op=000000	shamt	rt	rd	0	func=000000	`sll rd, rt, shamt`
op=000000	shamt	rt	rd	0	func=000010	`srl rd, rt, shamt`
op=000000	shamt	rt	rd	0	func=000011	`sra rd, rt, shamt`
op=001111	10000	rt	16-bit immediate			`lui rt, imm`

(A) Comment on how Lori's new encoding approach impacts the hardware implementation (i.e., relate it to the miniMIPS block diagram discussed in class, and also shown on the previous page).

(B) Does Lori's new encoding impact the hardware implementation of the variable shift instructions (`sllv`, `srav`, and `srlv`)? Comment briefly.

Lori has also suggested that the `lui` instruction be replaced with the following more general instruction:

op=001111	shamt	rt	16-bit immediate	`lsi rt, imm, shamt`

This new instruction, called `lsi,` works as follows: Load the specified register, `rt`, with the value of the signed-immediate constant shifted left by the unsigned instruction field, `shamt`.

(C) Does Lori's proposed `lsi` instruction require any additional hardware modification to the miniMIPS data path beyond those needed for her new encodings in part (A)?

(D) Discuss the utility of Lori's new instruction. Specifically, what capabilities does it provide over `lui`. Comment on whether `lui` is a subset of the `lsi` instruction's functionality?

(E) Discuss the implications of Lori's choice to treat the 16-bit immediate value as a signed number. Does it impact the data path? How does the set of constants that can be generated vary in comparison to an unsigned implementation?

Problem 3. Delayed Decisions (40 points)
Many modern instructions set architectures include special conditional instructions designed to avoid branch delays and to pipeline stalls related to determining a branch target. Consider the following proposed extension to the miniMIPS ISA.

```
abnz  rt,rs,label
```

Meaning: Add the contents of register `rs` to those of register `rt`, if the result is not zero branch to label. In the C language, this could be written as:

```
if (Reg[rt] + Reg[rs] != 0) {
        PC ← PC + 4 + 4*sign_extend(imm16);
}
Reg[rt] ← Reg[rt] + Reg[rs]
```

(A) What instruction format would the `abnz` instruction use?

(B) How should each miniMIPS control signal (see the table in Problem 1) be set to implement the `abnz` instruction (assume the unpipelined miniMIPS implementation). (*Hint:* You should specify PCSEL as a function of the ALU's Z flag.)

Consider the following two implementations of the procedure `int sum(int N)`. The first uses only standard MIPS instructions while the second takes advantage of the `abnz` instruction:

```
sum1: addu  $sp,$sp,-24
      move  $v0, $0
loop: add   $v0,$v0,$a0
      addi  $a0,$a0,-1
      bne   $a0,$0,loop
      addu  $sp,$sp,24
      j $31
```

```
sum2: addu  $sp,$sp,-24
      move  $v0,$0
      addi  $t0,$0,-1
loop: add   $v0,$v0,$a0
      abnz  $a0,$t0,loop
      addu  $sp,$sp,24
      j $31
```

(C) For what values of the argument N is sum2 is at least 25% faster than sum1?

Despite the apparent advantages of the `abnz` instruction (it requires no additional H/W and it improves the performance of some loops), there are still significant reasons for not including it. In particular, it can be difficult for a compiler to take advantage of it. Consider the following C-code fragment:

```
int sum = 0;
for (int i = 0; i < N; i = i + 1)
      sum = sum + x[i];
```

(D) Write a MIPS assembly language code fragment for the loop given above using the standard MIPS branch instructions, and then recode your fragment incorporating the `abnz` instruction. Comment on the coding and conceptual difficulties associated with incorporating the `abnz` instruction in this loop (Note: In order to support debugging it is required that the sum be computed in the same order as specified by the C-code).