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

Comp 411 Computer Organization
Spring 2012

Problem Set #5
Issued Wednesday, 3/21/12; Due Wednesday, 4/9/12

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. "Simplified Shifts"

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. what bits fetched from the instruction memory would need to be rerouted, and to where).

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

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`

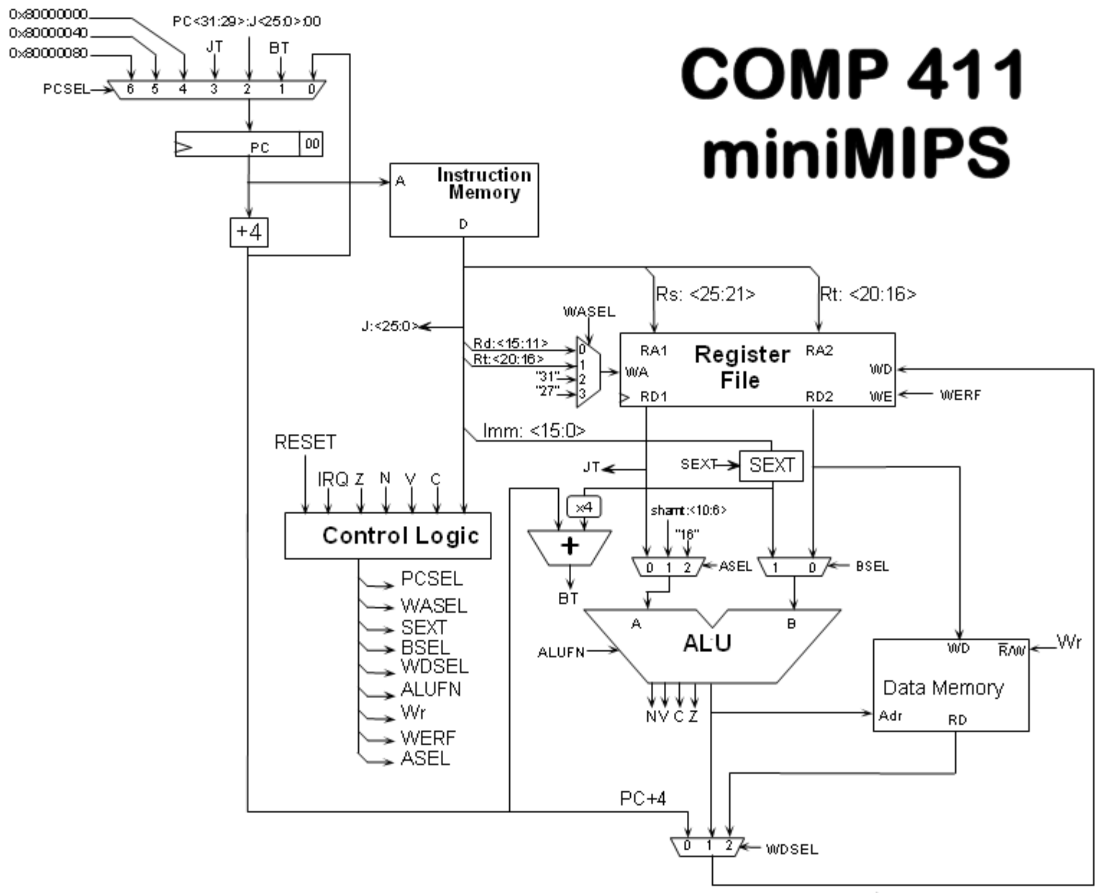
Load the specified register, `rt`, with the value of the signed-immediate constant shifted left by the unsigned instruction field, `shamt`.

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

(D) Discuss the utility of Lori's new instruction. Specifically, what capabilities does it provide over `lui`. Comment on whether `lui` 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 2. "Out of Control"



Fill in the entries of the Control Logic ROM, based on the given data path. Feel free to print this page, fill in your answers, staple it to your answers for problems 1-2, and turn it in.

Opcode	PCSEL	WASEL	SEXT	BSEL	WDSEL	ALUFN				Wr	WERF	ASEL
						Sub	Bool	Shft	Math			
`subu`												
`xor`												
`addiu`												
`sll`												
`andi`												
`lw`												
`sw`												
`j`												
`jal`												
`lui`												

Problem 3. Delayed Decisions
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
```

Add the contents of register `rs` to those of register `rt`, if the result is not zero branch to label.

```
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 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?

(C) At first glance it might seem that subtracting Reg[rs] from Reg[rt] is a more natural instruction choice. Explain why this change would require datapath modifications.

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

(D) 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.

(E) One problem with the `abnz` instruction is that it is difficult to pipeline. At what stage in the miniMIPS 5-stage pipeline is the branch decision made for the `abnz` instruction? How many delay slots would a straightforward implementation of it require? Describe the additional logic that would be required to compute an early branch decision in the Register-Fetch pipeline stage for the `abnz` instruction. How does the complexity, and likely propagation delay, of the early branch-decision hardware required for the `abnz` instruction compare to that of the `bne` and `beq` instructions of the standard MIPS ISA.

Another difficulty associated with special-purpose branching instructions is that it is often difficult for compilers to take advantage of them. Consider the following C-code fragment:

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

(F) 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).

Problem 4. Flexible Pipes

Bud LeVile has suggested a modification to the 5-stage miniMIPS pipeline discussed in class. Having noticed that the MEM stage is only used for load and store instructions, he proposes omitting that pipeline stage entirely whenever the memory isn't accessed, as illustrated below:

Instruction	*t*	*t+1*	*t+2*	*t+3*	*t+4*
`lw` and `sw`	IF	RF	ALU	MEM	WB
Other instructions	IF	RF	ALU	WB	

Bud reasons that instructions which skip the MEM stage can complete a cycle earlier, thus, allowing most programs will run as much as 20% faster! In your answers below assume that both the original and the Bud-modified pipelined implementations are fully and properly bypassed.

(A) Explain briefly to Bud why decreasing the latency of a single instruction does not necessarily have an impact on the throughput of the processor (Hint: Consider how long it would take the original pipelined miniMIPS to complete a sequence of 1000 adds. Then compare that with how long a Bud-modified miniMIPS would take to complete the same sequence).

(B) Consider a sequence of alternating `lw` and `add` instructions. Assuming that the `lw` instructions use different source and destination registers than the `add` instructions (i.e., there are no pipeline stalls introduced due to data dependencies), what is the instruction completion rate of the original, unmodified 5-stage miniMIPS pipeline?

(C) Now show how the same sequence of instructions will perform on a processor modified as Bud has suggested. Assume that the hardware will stall an instruction if it requires a pipeline stage that is currently being used by a previous instruction. For example, if two instructions both want to use the Write-Back pipeline stage in the same cycle, the instruction that started later will be forced to wait a cycle. Draw a pipeline diagram showing where the stalls need to be introduced to prevent pipe stage conflicts.

(D) Did Bud's idea improve performance? Explain why or why not?

Problem 5. Stage Three

Suppose that the behavior of the `lw` and `sw` instructions were redefined as follows:

`lw rt, (rs)` Reg[rt] ← Mem[Reg[rs]]

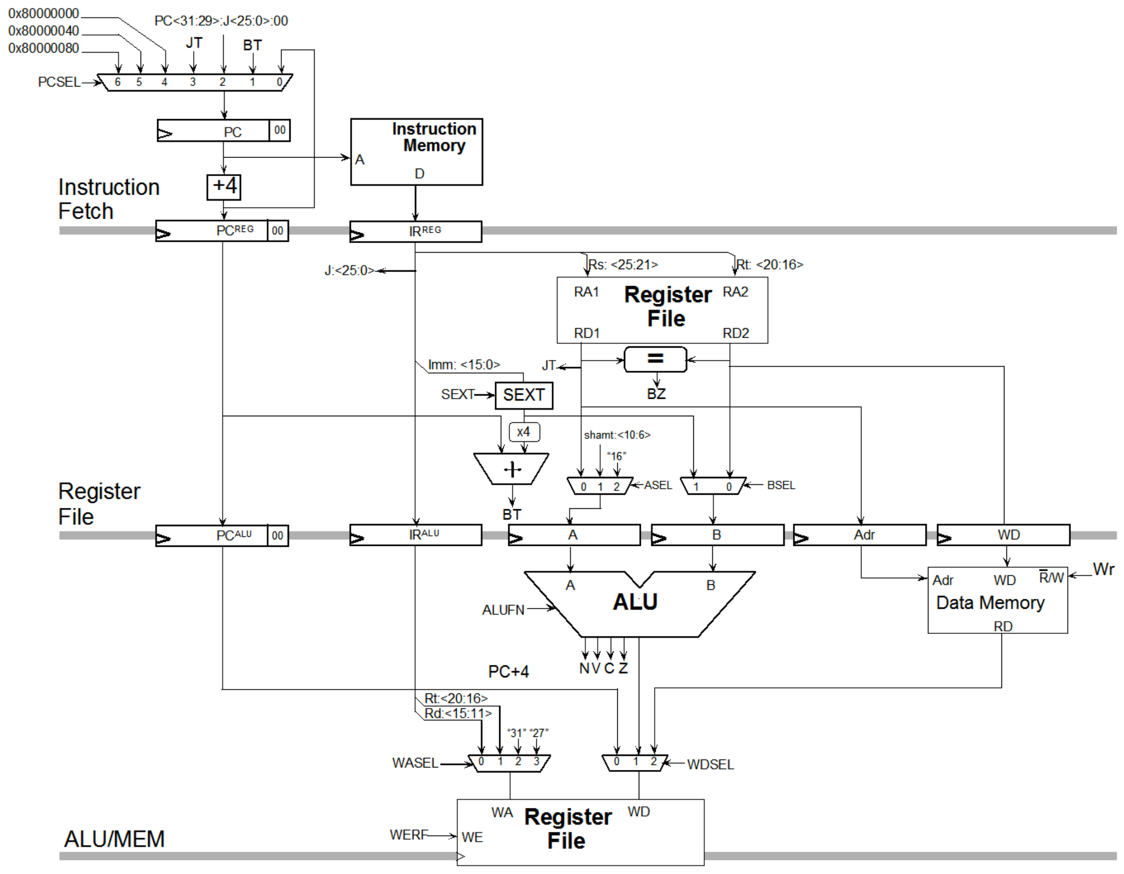
Load register rt with the contents of the memory location specified register rs.

`sw rt, (rs)` Mem[Reg[rs]] ← Reg[rt]

Store the contents of register rt at the memory location specified register rs.

(A) Give instruction sequences that emulate the operation of the original `lw` and `sw` instructions as pseudoinstuctions using the redefined versions. Note: Use register `$as` to store any required intermediate values.

These ISA changes enable memory accesses and ALU operations to be overlapped in the same pipeline stage (ALU/MEM). They also allow for the construction of a meaningful 3-stage miniMIPS pipeline, whose datapath is illustrated below:



(B) Discuss where and the how many bypass paths are needed for this modified architecture. Give an example instruction sequences that exercises each bypass path.

(C) Does this modified 3-stage pipeline architecture require pipeline interlocks on load instructions? Explain why or why not.