CSE 378 Autumn 2007 Midterm Exam Solution
Machine Organization & Assembly Language

Write your answers on these pages. Additional pages may be attached (with staple) if necessary. Please ensure that your answers are legible. Please show your work. Write your name at the top of each page.

Total points: 100

1. [10 Points] **Calling Conventions**.

(a) What are calling conventions?

Answer:

Calling conventions are a protocol governing the use of registers and the stack by procedure calls. They specify how procedure parameters and return values are to be passed from function to function.

Rubric:
4 points for mentioning register saving across function calls, stack usage, or parameter/return value passing.
0 points for confusing calling conventions with coding conventions.

(b) Why are they necessary?

Answer:

Calling conventions are necessary

- to allow assembly code compiled separately (by different people with different compilers) to interoperate
- to know when to save registers to avoid data loss
- to avoid unnecessary saving of registers that could hurt performance

Rubric:
4 points for mentioning at least one of the above.
0 points for confusing calling conventions with coding conventions.

(c) Give an example of a MIPS calling convention, and say why it is useful.

Answer:

- Having specific registers be callee-saved (and others caller-saved) are a calling convention; they are useful because having all registers be callee- or caller-saved would be inefficient.
- Passing arguments via \$a0-\$a3 allows for more efficient procedure calls than always using the stack.

Rubric:
1 point for giving a valid MIPS calling convention.
1 point for saying why the convention you gave is useful.

2. [30 points] **MIPS Programming**

   The following C code does a binary search for a value in a *sorted* array `A` of integers (each integer is 32 bits in size, and they are packed together in the array). Translate this C code into MIPS assembly using the template on the next page. *Hint*: you may find the `BGTZ`/`BLTZ` instructions useful.

   Your solution will not be graded for syntax, but you must use the proper opcode and register names. You should make use of the following assumptions:

   - `$a0` contains `A`
   - `$a1` contains `lengthOfA`
   - `$a2` contains `value`
   - `$v0` should be used to hold the return value
   - your function will be called by some other function.
   - you are allowed to use pseudo-instructions.

   C version

```
int binSearch(int *A, int lengthOfA, int value) {
   int low = 0;
   int high = lengthOfA - 1;
   int mid = 0;

   while (low <= high) {
       mid = (low + high) / 2;
       if (A[mid] > value)
            high = mid - 1;
       else if (A[mid] < value)
            low = mid + 1;
       else
            return mid; /* found, return array index */
   }
   return -1; /* not found */
}
```

   MIPS assembly version (next page)

MIPS assembly version

```
binSearch:
    li $t0 <- 0            # low
    addi $t1 <- $a1, -1    # high
    li $t2 <- 0            # mid
```

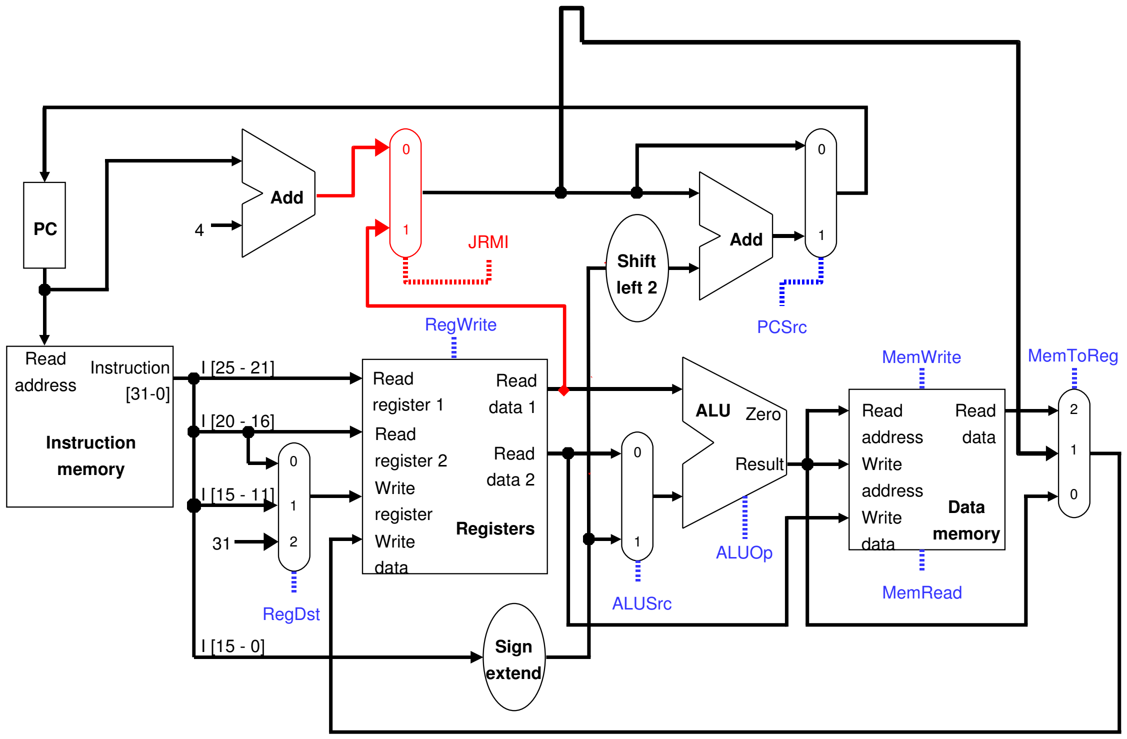
Answer:

```
    li $v0 <- -1           # assume not found
loop:
    bgt $t0, $t1, end

    add $t2 <- $t0, $t1    # compute mid
    srl $t2 <- $t2, 1      # truncate...
    sll $t3 <- $t2, 2      # ...and divide by 2

    add $t3 <- $t3, $a0    # compute &A[mid]
    lw $t4 <- 0($t3)       # load A[mid]

    bgt $t4, $a2, greater # A[mid] > value
    blt $t4, $a2, less     # A[mid] < value
    addi $v0 <- $t2, 0     # found it!
    j end

greater:
    addi $t1 <- $t2, -1    # high = mid - 1
    j loop

less:
    addi $t0 <- $t2, 1     # low = mid + 1
    j loop

end:
    jr $ra
```

3. [15 Points] **Datapath.**

Taking the single-cycle processor developed in class, we want to add a new instruction `jrmi imm16(rs)` which executes a jump to the instruction at the specified address in memory. `jrmi` is an I-type instruction. The 16-bit immediate (a word offset) and the register `rs` specify an address via register base + offset (in the same manner as `lw`/`sw`).

(a) Draw the necessary modifications to implement the above instruction on the figure of the single-cycle data-path provided below.

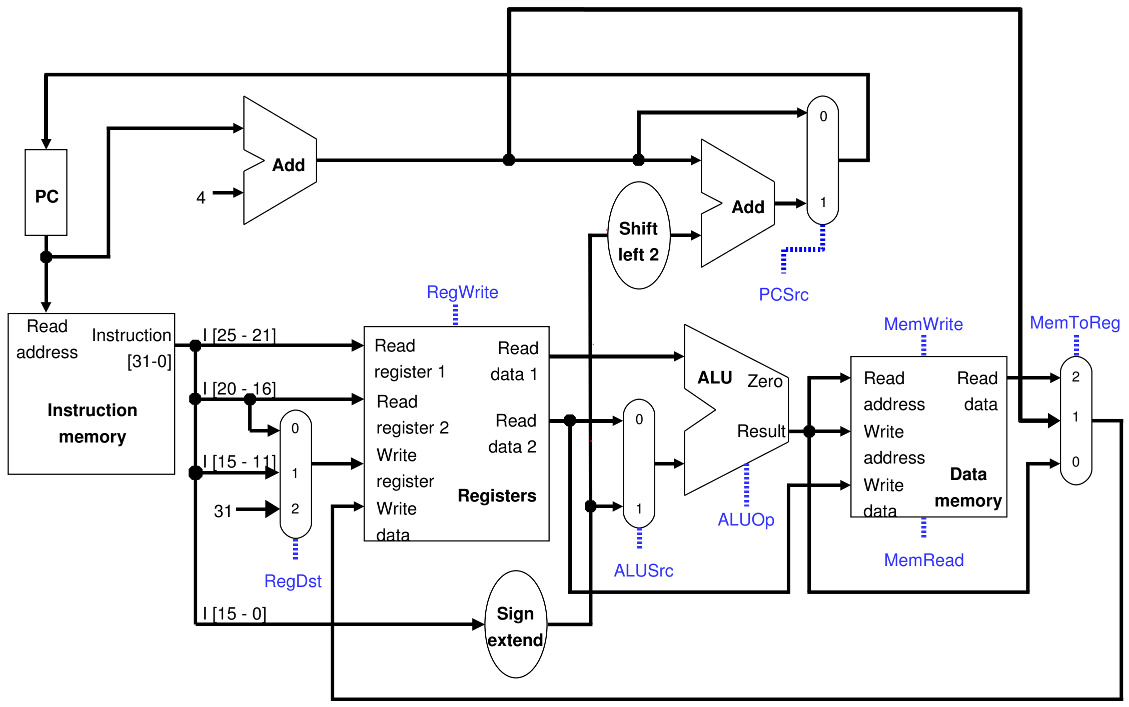


Answer:
The new logic for `jrmi` is drawn in red.

(b) What is/are the new control signal(s) required to implement `jrmi`?

Answer:
We introduced one new mux, and that mux needs a 1-bit control signal, `JRMI`.

4. [20 Points] **Datapath Control Signals.**



Given the single-cycle datapath above (control signals are marked with dashed lines), fill in the blanks in the table below. You must either give the control signals (0, 1, X) for a particular MIPS instruction, or give the MIPS instruction that uses the specified control signals.

Each control signal must be specified as 0, 1 or X (don't care). Writing a 0 or 1 when an X is more accurate is *not* correct.

Opcode	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg	PCSrc
`nor`	1	1	0	nor	0	0	0	0
`beq`	X	0	0	sub	0	0	0	$1 \wedge$ ALU.Zero
jal	2	1	X	X	0	0	1	1

Answer: The answers appear in the table above.

Rubric:
1 point for each correct control signal
3 points for stating the missing opcode as a "jump" instruction, but not `jal`.
6 points for stating the missing opcode as `jal`.

5. [15 Points] **Pipeline Hazards.**

   Consider the sequence of MIPS instructions below:

```
add $2 <- $3, $4
or $5 <- $2, $4
lw $6 <- 0($4)
addi $7 <- $6, 0x5
sub $8 <- $8, $4
```

   (a) Draw arrows on the instructions above indicating all the data dependences.

   Answer: There is a dependence between the `add` and the `or` insn via `$2`, and another between the `lw` and the `addi` via `$6`.

   Rubric:
   -1 point for drawing arrows for any false dependences.

   (b) Reorder the instructions into a new schedule that will execute without any stalls on a 5-stage pipelined processor with forwarding. For reference, you may refer to the pipeline diagram below.

   Answer:

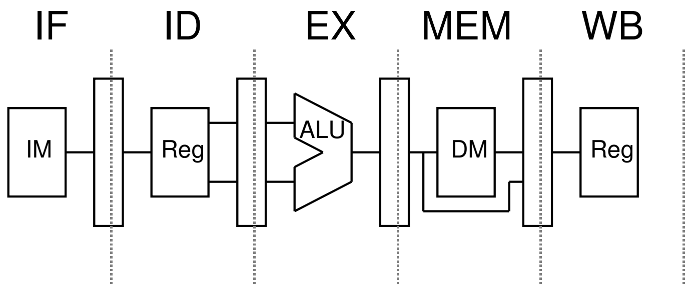
   We need to avoid the 1-cycle load-use stall between the lw and the addi. We can simply put the sub in between.

```
add $2 <- $3, $4
or $5 <- $2, $4
lw $6 <- 0($4)
sub $8 <- $8, $4
addi $7 <- $6, 0x5
```

   Rubric:
   -3 points for each unnecessary stall.

   (c) Reorder the instructions into a new schedule that will execute without any stalls on a 5-stage pipelined processor *without* forwarding. For reference, you may refer to the pipeline diagram below.

   Answer:



The or has a data dependence on the add, and the addi on the lw. Since we have no forwarding, we need to allow 2 cycles between the dependent instructions, so that the producer can reach WB when the consumer is in ID.

```
add $2 <- $3, $4
lw $6 <- 0($4)
sub $8 <- $8, $4
or $5 <- $2, $4
addi $7 <- $6, 0x5
```

Rubric:
-4 points for each unnecessary stall.

6. [10 Points] **Pipelining.**

Suppose you have a program that is 1 instruction long. Will it execute faster on a pipelined processor than on a single-cycle processor (assuming equal processor frequencies)? Why or why not?

Answer:

The program will actually execute more quickly on the single-cycle design: it will only take 1 cycle. On the pipelined design, it will take 5 cycles, as it has to traverse the entire pipe.

Pipelines trade off latency for throughput: each instruction has higher latency, but throughput is much higher than for a single-cycle design.

Rubric:
3 points for stating that the single-cycle execution will be faster.
7 points for explaining why.