CSE 378 Winter 2009 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] **The Stack**.

   (a) Why is the stack a convenient data structure for function calls?

   Answer:
   The stack is convenient because it models the process of control exchange well. When a processes calls another processes the callee process (and all subsequent calls) must fully finish before the calling process can resume. This closely resembles the stack's first-in-last-out design. Also the local variables of individual functions can be safely preserved in this manner, as the local stack frame of subsequent function calls will be placed after that of the current.

   (b) What is normally stored on the stack?

   Answer:
   As a function is called, it places all of its local variables at a particular place on the stack. If it calls another function, then that function would place its local variables directly below that and so on, corresponding to pushing things onto a stack. Once the function returns, the space it used is freed, and the stack pointer points to the function directly before it, resuming that function. This corresponds to popping something off of a stack.

   (c) What is the stack pointer?

   Answer:
   Points to top of the stack

   (d) How do you push data onto the stack?

   Answer:
   Move the stack pointer and put the data on the stack.

   (e) How do you pop data off of the stack?

   Answer:
   Move the stack pointer

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

   The strstr() function finds the position of the first occurrence of a substring within a bigger string. For example, if a program calls strstr("stack", "ack") the function should return **2** (the the beginning position of "ack" within "stack"). Should a program call strstr("stack", "acc") the function should return **-1**, since "acc" is not a substring within the source string.

   Here are some examples:
   strstr("cse378","37") returns 3
   strstr("cse378","cs") returns 0
   strstr("cse378","bob") returns -1

   Translate this C code into MIPS assembly using the template on the next page. 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:

   - `$a0` contains the address of the first char of `string1`
   - `$a1` contains the address of the first char of `string2`
   - `$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 strstr(char* haystack, char* needle) {
 int i, j;
 // outer loop
 for(i=0; haystack[i] != '\0'; i++) {
   // inner loop
   for(j=0; needle[j] != '\0' && needle[j] == haystack[i+j]; j++) {
     // just move to next character
   }
   if (needle[j] == '\0') { // if needle is exhausted,
     return i;              // needle found at this position
   }
 }
 return -1; // no match
}
```

   MIPS assembly version (next page)

MIPS assembly version

```
strstr:
addiu $sp, $sp, -4
sw $ra, 0($sp)
add $s0, $a0, $zero   # base pointer for string1
add $s1, $a1, $zero   # base pointer for string2
addi $t0, $zero, 0        # i
addi $t1, $zero, 0        # j



outer_loop:


inner_loop:
```

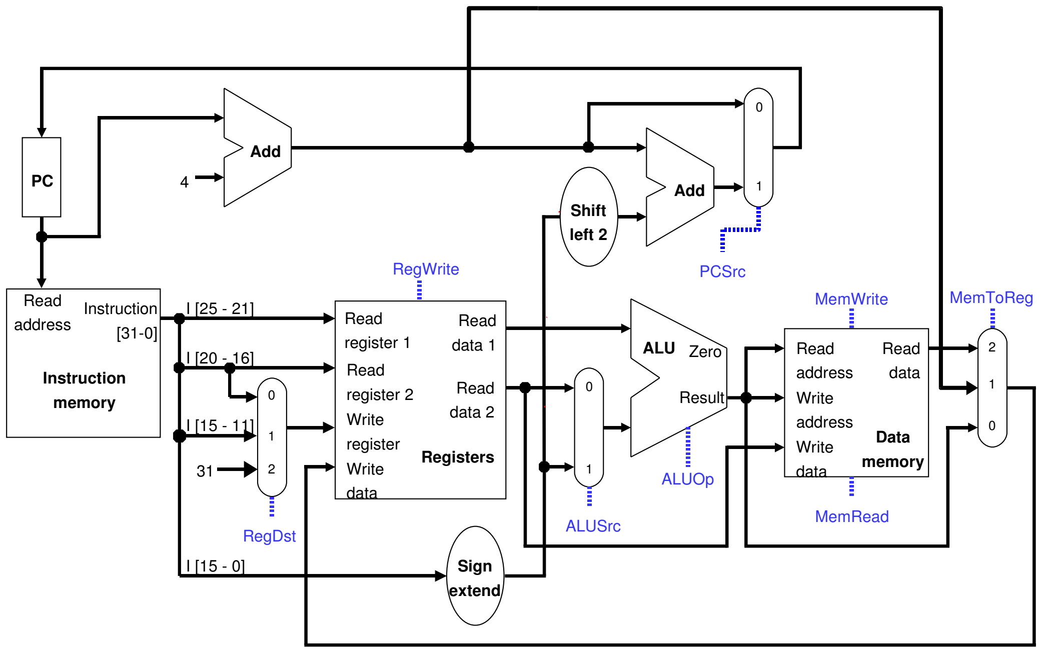
```
end:
lw $ra, 0($sp)
addiu $sp, $sp, 4
jr $ra
```

Answer:

```
addi $v0, $zero, -1 # assume not found, set $v0 = -1
outer_loop:
add $t2, $t0, $s0   # adjust pointer to next char in string1
lb $t2, 0($t2)   # get that next char
beq $t2, $zero, end # null terminator for string1

inner_loop:
add $t3, $t1, $s1 # adjust pointer to next char in string2
lb $t3, 0($t3) # get that next char

add $t4, $t0, $t1 # i + j for inner loop
add $t4, $t4, $s0 # string1[i+j]
lb $t4, 0($t4)

beq $t3, $zero, end_outer_loop # null terminator for string2
bne $t3, $t4, end_outer_loop # string2[j] != string1[i + j]
addi $t1, $t1, 1 # j++
j inner_loop

end_outer_loop:
beq $t3, $zero, found # found substring else iterate outer loop
addi $t0, $t0, 1   # i++
j outer_loop

found:
add $v0, $t0, $zero   # return i
end:
lw $ra, 0($sp)
addiu $sp, $sp, 4
jr $ra
```

3. [20 Points] **Datapath.**

Let's add a Jump Indirect instruction to the single-cycle processor developed in class. This I-format instruction `jin imm16(rs)` will cause the processor to jump to the address stored in the word at memory location `imm16 + R[rs]` (the same address computed by `lw` and `sw`).



(a) Draw the necessary modifications to implement the `jin` instruction on the figure of the single-cycle datapath provided above.

Answer:

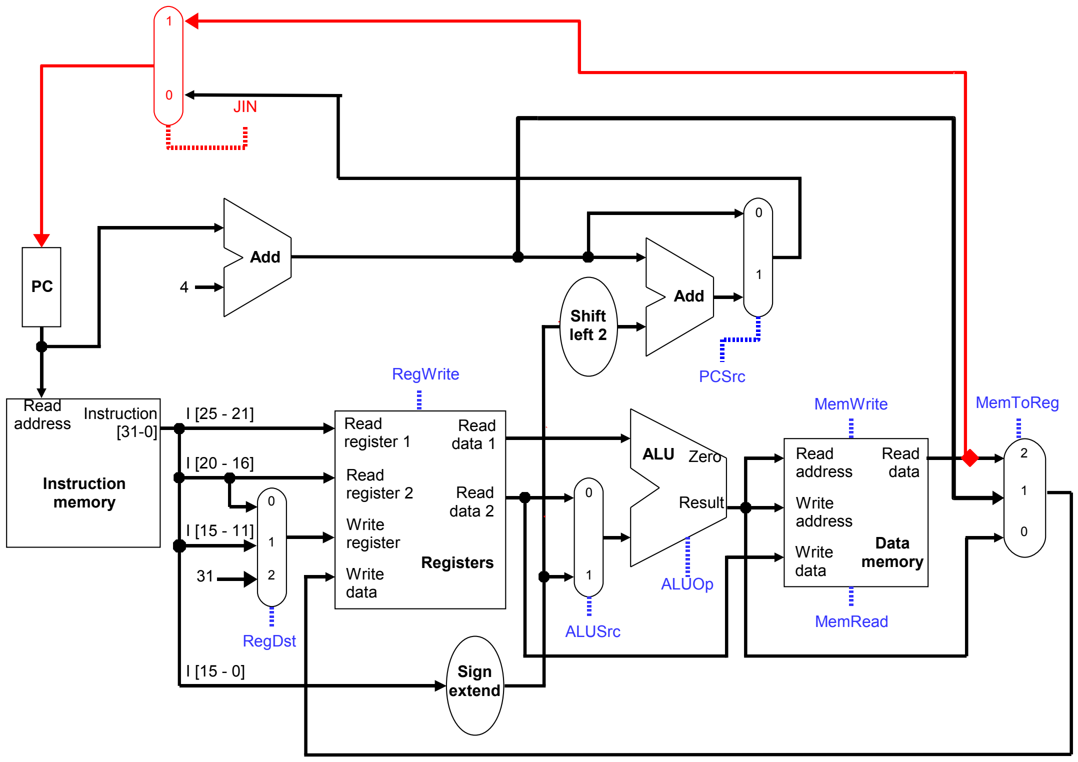
The new logic for `jin` is drawn in red on the next page.

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

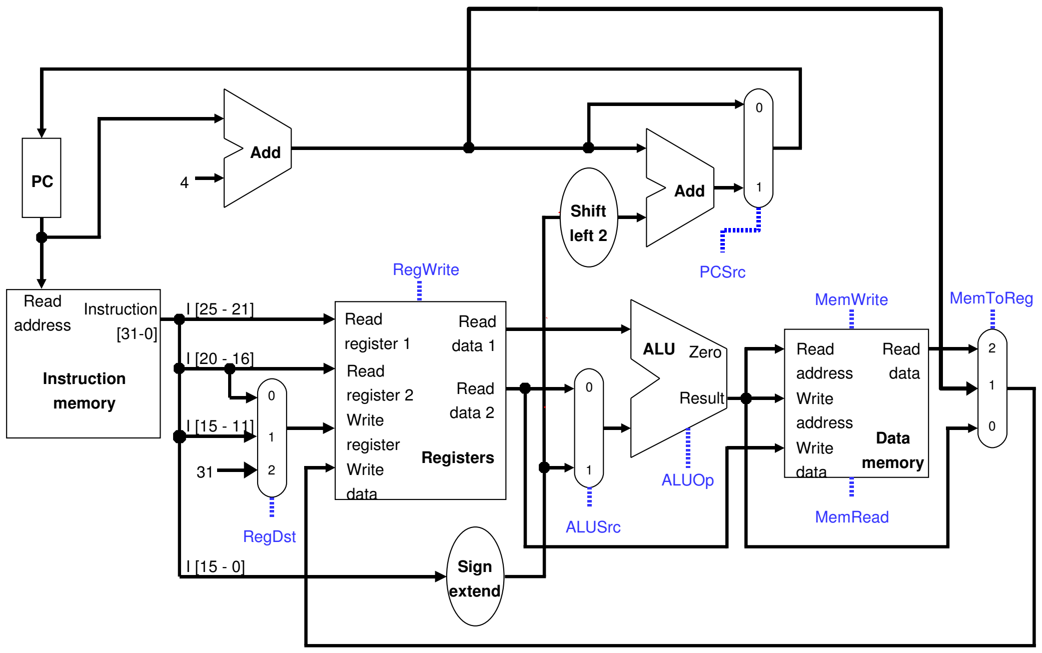
Answer:

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

Name: ______________________________________


1
0
JIN
PC
Add
4
Add
0
1
Shift left 2
PCSrc
RegWrite
MemWrite
MemToReg
Read address
Instruction [31-0]
Instruction memory
I [25 - 21]
I [20 - 16]
I [15 - 11]
31
0
1
2
RegDst
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALU
Zero
Result
ALUOp
0
1
ALUSrc
Read address
Write address
Write data
Read data
Data memory
MemRead
2
1
0
I [15 - 0]
Sign extend

4. [10 Points] **Control Unit.**



A new I-format instruction `swu` has been added to the MIPS instruction set. Its format is `swu rt, I(rs)`. It takes as arguments register `rt`, register `rs`, and immediate `I`, and it stores the contents of `R[rt]` at the memory address `(R[rs] + I)` and then increments `R[rs]` by `I`.

Given the single-cycle datapath above (control signals are marked with dashed lines), fill in the blanks in the table below for this new instruction. You must give the control signals (0, 1, 2, X) for the MIPS instruction.

Each control signal must be specified as 0, 1, 2 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
`swu`	0	1	1	add	1	0	0	0

Answer: The answers appear in the table above.

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

   Consider the sequence of MIPS instructions below:

```
add  $8, $4, $1
lw   $2, 8($5)
sllv $2, $2, $4
andi $5, $5, 0x3
sw   $5, 0($6)
```

   (a) Draw arrows from producer to consumer on the instructions above indicating all the data dependences.

   Answer: There is a dependence between the `lw` and the `sllv` instruction via `$2`, and another between the `andi` and the `sw` via `$5`.

   (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 sllv. We can simply put the add in between.

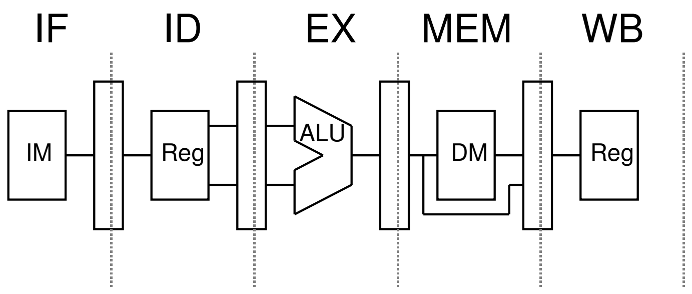
```
lw   $2, 8($5)
add  $8, $4, $1
sllv $2, $2, $4
andi $5, $5, 0x3
sw   $5, 0($6)
```

   (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 sllv has a data dependence on the lw, and the sw on the andi. 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.

```
lw   $2, 8($5)
andi $5, $5, 0x3
```



```
add  $8, $4, $1
sllv $2, $2, $4
sw   $5, 0($6)
```

6. [15 Points] **Pipelining.**

   (a) Suppose you have a 5 stage pipeline in which the IF, ID, EX, MEM, WB stages took 3ns, 2ns, 2ns, 3ns and 2ns respectively. Explain performance impact in comparison with a single cycle implementation in each of the following scenarios.

   i. How long does a single instruction take in the a single-cycle implementation?

   Answer: (3+2+2+3+2) = 12ns

   ii. How long does it take N instructions to complete in pipelining, where N is some arbitrary large number?

   Answer: 3*N + 12

   iii. What is the speedup of pipelining?

   Answer: 4x speedup

   iv. Is that the max speedup? Why or why not?

   Answer: No, this is not the ideal speed-up, as the pipeline stages are not balanced. In otherwords, since the IF and MEM stages take 3ns, that will be the lenght of each stage, regardless of the fact that the other 3 stages take less time.

   (b) Why do pipelined machines in general offer improved performance?

   Answer: Pipelined machines offer better throughput because when pipeline multiple stages are utilized at the same time, pipelined machines complete one instruction in the time it takes a single stage to complete. This is faster than the time it takes for the entire processor to complete every stage before fetching a new instruction.