CSE 378 Winter 1998

Machine Organization and Assembly Language Programming

Midterm

Friday February 13th

NAME : _____

Do all your work on these pages. Do not add any pages. Use back pages if necessary. Show your work to get partial credit.

This exam is worth 50 points. After each question, you will find the number of points it is worth. You should spend approximately x minutes on a question worth x points (e.g., 5 minutes on question 1 worth 5 points). That will leave you with 10 minutes to look over your work.

1. 5 points _____
2. 6 points _____
3. 10 points _____
4. 5 points _____
5. 12 points _____
6. 12 points _____

1. (5 points)

Suppose register $t1 contains 0xFFFFFFF6 and register $t2 contains 0x0000000A. What is the result (in hexadecimal) in register $t0 after each of the following instruction executes (if there is an overflow/underflow, say so):

```
add     $t0, $t1, $t2          #$t0 = 0x00000000

sub     $t0, $t1, $t2          #$t0 = 0xFFFFFFEC

sll     $t0, $t1, 3            #$t0 = 0xFFFFFFB0

sra     $t0, $t1, 3            #$t0 = 0xFFFFFFFE
```

Consider the instruction

```
Loop:   jal     foo
```

and suppose that `Loop` is at address 0x00040024 and `foo` at address 0x00041234. What is the content of Register $ra (register 31) after the `jal` instruction has been executed?

0x00040028

2. (6 points)

Consider two implementations M1 and M2 of the same ISA. We are interested in the performances of two programs P1 and P2. P1 and P2 have the following respective instruction mixes:

Operations	P1	P2
Load/Store	40%	50%
ALU operations	50%	20%
Branches	10%	30%

and the CPI's for each machine are:

(a) Assume that the clock rate of M1 is 200MHz. What should be the clock rate of M2 so that both machines have the same execution time for P1.

Operations	M1	M2
Load/Store	2	2
ALU operations	1	2
Branches	3	2

Since both machines execute the same number of instructions and take the same time for execution, we must have:

$$(CPI\ on\ M1)/(M1\ clock\ rate) = (CPI\ on\ M2)/(M2\ clock\ rate)$$

$$(.4 \times 2 + .5 \times 1 + 0.1 \times 3)/200 = 2/(M2\ clock\ rate)$$

i.e.,

$$(M2\ clock\ rate) = (2 \times 200)/1.6 = 250MHz$$

(b) Assume now that both machines have the same clock rate and that P1 and P2 execute the same number of instructions. Which machine is faster for a workload consisting of equal runs of P1 and P2.

$$\frac{Ex.\ time\ M1}{Ex.\ time\ M2} = \frac{CPI\ on\ M1}{CPI\ on\ M2} = (1.6 + 2.1)/4 = 3.7/4.0$$

M1 is faster by 8% (4/3.7 = 1.08)

(c) Find a workload (using only P1 and P2) that makes M1 and M2 have the same performance when they have the same clock rate.

If x is the proportion of P1 runs, we must have:

$$x \times (CPI\ of\ P1\ on\ M1) + (1 - x) \times (CPI\ of\ P2\ on\ M1) = 2$$

$$x = 0.2$$

We need 1 run of P1 and 4 runs of P2

3. (10 points)

Suppose you had one opcode left and you wanted to add one instruction to the MIPS ISA to manufacture the MIPS 9999.

(a) Which of the following 3 instructions could be *encoded* using one of the current MIPS instruction formats (justify your answer by showing how the instruction is encoded and the other two cannot be encoded):

```
ADDW     $rt, address($rs)          # rt = rt + Mem[rs + address]
```

```
ADD2W    $rd, $rt, address($rs)    # rd = rt + Mem[rs + address]

ADDIW    $rd, immed, address($rs)  # rd = immed +  Mem[rs + address]
```

where `address` and `immed` are 16-bit signed integers.

ADDW could be encoded like a "LW" (6 bits for opcode, 5 each for the registers, and 16 bits for the offset of the address).

ADD2W needs another 5 bit field for the third register. Hence it is not possible.

ADDIW needs (wrt to ADDW) another 16 bit for the immediate. Not possible.

(b) Assuming that you now include the instruction that you can encode in the MIPS 9999, can you still say that the MIPS 9999 is a RISC machine? Justify your answer in one sentence.

No. The ADDW concept violates the "load-store" concept that all non-memory operations only use registers as operands

(c) You now decide to build the MIPS 9999 using a multiple cycle implementation. In addition to adding some busses and extending some multiplexers, do you need to add major resources (ALU, Memory, Registers) to the data path shown for the MIPS in the book (reproduced on the next page)? If so which one, if not why not (you do NOT need to trace the data path in either case).

How many cycles would it take to execute the new instruction?

You do not need any new resources. Proceed like in a LW but instead of storing the MDR in the register file in cycle 5, you add it to the register and, in a 6th cycle, write the result.

You need to extend the multiplexers that control the inputs to the ALU and add a bus from the MDR to one of these multiplexers.

4. (5 points)

(a) In the MIPS ISA which class(es) of instructions rely on the fact that instructions always start on a word boundary?

Branch and jump instructions

(b) Is the call/callee protocol relative to procedures described in class the only possible one with respect to the $s and $t registers? If not, give another possibility.

No, there are zillions of possibilities. For example, have the caller save all $s and $t registers.

(c) Is an instruction like LBU (load byte unsigned) absolutely necessary? If not, is there a sequence of MIPS instructions that could be used to achieve the same effect and if so what instructions would be used in that sequence. (You don't have to write the sequence; just describe it. Also assume that memory can only be accessed through a LW (load word) instruction that accesses memory locations on a word boundary.)

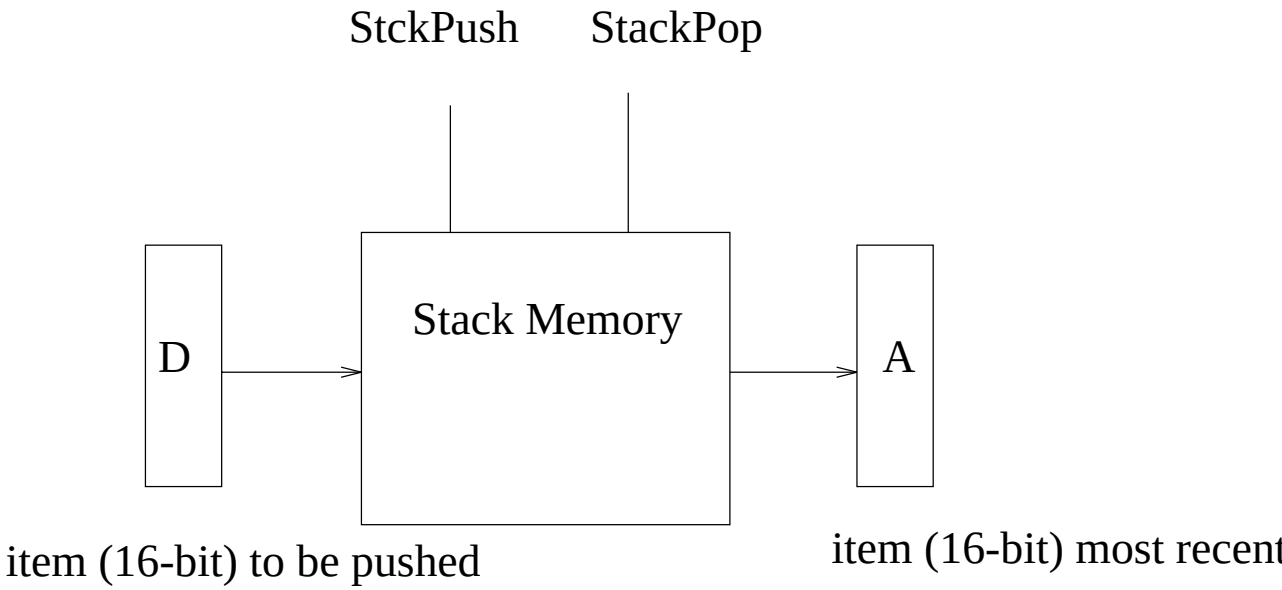
LBU is not necessary. Perform a LW followed by shifts left and right of either 8, 16, or 24 bits depending on the byte adddress. You could also use shift and some logical operations with adequate masks.

5. (12 points)

You are to design a subset of the data path and indicate the control lines for a stack based machine with the following specifications.

All instructions are 16 bits long and operate on 16 bit data. Instead of a register file, data is accessed from stack memory which supports pushes and pops as shown below. There is data memory, distinct from stack memory, and instruction memory. Data, stack and instruction memory all use 16 bit adresses.

In this machine the stack pointer is hardwired and stack memory operates as follows:



On a "push", triggered by the signal *StckPush*, the contents of register D are stored on the top of the stack and the hardwired stackpointer is incremented

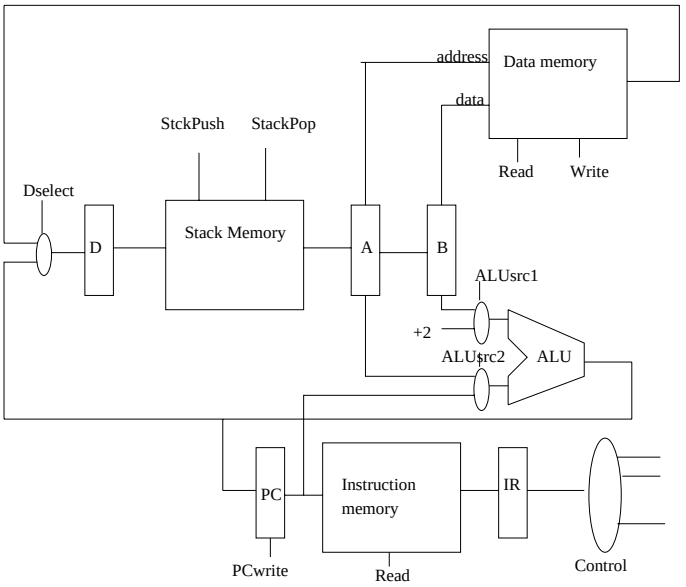
by 2. On a "pop", triggered by the signal *StckPop*, the hardwired stackpointer is decremented by 2 and the top of the stack is written in register A.

Sketch (on the next page) the datapath and label the control signals necessary to support the 3 operations **Store, Load, ADD** (this is not an all inclusive list but you only need to concern yourself with these 3). Note that the semantics of these operations are slightly different from those of the JVM that you programmed; This is to make your task easier.

- **Store**: pop data to be stored in data memory from stack, pop address at which to store the data from the stack, perform the store (write) in data memory.
- **Load**: pop address at which to load the data from data memory, load (read) the desired data, push data onto the stack memory.
- **ADD**: pop data value1, pop data value 2, add, push result onto stack.

Use a multiple-cycle datapath design. Assume that only one memory (either instruction, data, or stack) can be accessed in each cycle. Use a single ALU. You may use as many multiplexers, registers, and buses as necessary. Describe succinctly what happens in each cycle.

Hint: in addition to the D and A registers of the figure, you'll need **AT LEAST** a PC, an IR and another "temporary" register.



6. (12 points)

In this question, you are asked to write a small program in MIPS assembly language. You won't be graded down if you forget the exact name of an instruction or the place of a comma as long as you respect the spirit of the MIPS instruction set. You can use pseudo-instructions, such as `la` used below, as long as they would be recognized by SPIM.

The program you have to write is a *linear search* function, i.e., a function that searches for a *key* in an array of integer elements (32-bit) *array* of size *size* elements.

The calling sequence is:

```
la     $a0, arradd          #1st parameter is address of array
lw     $a1, size            #2nd parameter is size of array
lw     $a2, key             #3rd parameter is key to search for
jal    linear
```

You should NOT assume that *linear* is a leaf procedure (i.e., in further extensions, it might be that *linear* would call another procedure).

Your function should return in $vo the index of the first element of the array that matches the key and return the value -1 in $vo if there is no match.

Without trying to optimize and without saving the parameters:

```
linear:   sw    $ra, 0($sp)          # save return address
          subu  $sp,$sp,4            # since not leaf procedure
          li    $t0,0                # t0 will be index
loop:     beq   $t0,$a1,notfnd       # no match
          lw    $t1,0($a0)           # get next element
          beq   $t1,$a2,match        # found a match
          addi  $a0,$a0,4            # otherwise compute address of
                                     # next element
          addi  $t0,$t0,1            # and index of next element
          j     loop                 # and iterate
match:    move  $v0,$t0              # index in result register
          j     out
notfnd:   li    $v0,-1               # not found flag
out:      addu  $sp,$sp,4            # restore return address
          lw    $ra, 0($sp)          #
          jr    $ra
```

If you don't like destroying/using the $a registers, move them to $t registers.

If you are a super-optimizer, you could save 1 or 2 instructions by using $v0 everywhere there is $t0.