**CSE 378 Spring 2010** **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.

   (d) What registers may need to be saved by a recursive function?

   **Answer**: Both callee-saved and caller-saved registers may need to be saved by a recursive function.

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

In this question, you'll implement the core calculation function of a prefix calculator. The calculator takes a string containing a prefix expression of operators and numbers, seperated by a single space, and returns the value of the expression. Each number is a non-negative integer, and each operator takes exactly two operands. The value returned is a signed integer. Here are some examples:

| Call | Return value |
|---|---|
| `calc("12345")` | 12345 |
| `calc("+ 1 1")` | 2 |
| `calc("- 0 5")` | -5 |
| `calc("/ * 3 + 4 5 + 8 1")` | 3 |

The `calc()` function you need to implement is given below. It's written in a C-like language with tuples, allowing for multiple return values. Translate the function into MIPS assembly. `calc()` calls two other functions, also described below; don't implement these functions.

Please demonstrate your understanding of the MIPS calling convention by saving registers you are required to save.

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

- All numbers in the string are positive integers; a number will always start with a digit.
- Tokens in the string are separated by a single space character.
- Any input string will be correctly formatted; you don't need to check this or handle errors.
- Any character that's not a digit is an operand, and you can pass it to `compute` without checking it.
- You may load the ASCII value of a character by writing something like `li $t0, '9'`.
- The pointer returned by the toplevel call to `calc` can be anything.

`calc` is called with a pointer to the first character of an expression in `$a0`, and returns the value of the expression in `$v0` and a pointer to the next subexpression in `$v1`.

```
(int, char*) calculate(char *buffer) {
  if ('0' <= *buffer && *buffer <= '9') {
    return getnum(buffer);
  } else {
    char* firstArg = buffer+2;     // move pointer past operand and space
    (int arg1, char* nextArg) = calc(firstArg); // get first operand
    (int arg2, char* next)    = calc(nextArg);  // get scond operand
    int result = compute(op, arg1, arg2);       // compute result
    return (result, next);   // return result, next subexpression pointer
  }
}
```

`getnum` takes a pointer to the start of a number in the string in `$a0`, and returns the value of that number in `$v0` and a pointer to the next argument in the string (after the space following the number) in `$v1`.

```
(int, char*) getnum(char* buffer);
```

`compute` takes the operator character in `$a0` and the two integer arguments in `$a1` and `$a2`, and returns the result of performing the operation on the operands in `$v0`.

```
int compute(char op, int arg1, int arg2);
```

**MIPS assembly version of `calc` (next page)**

**MIPS assembly version of `calc`**

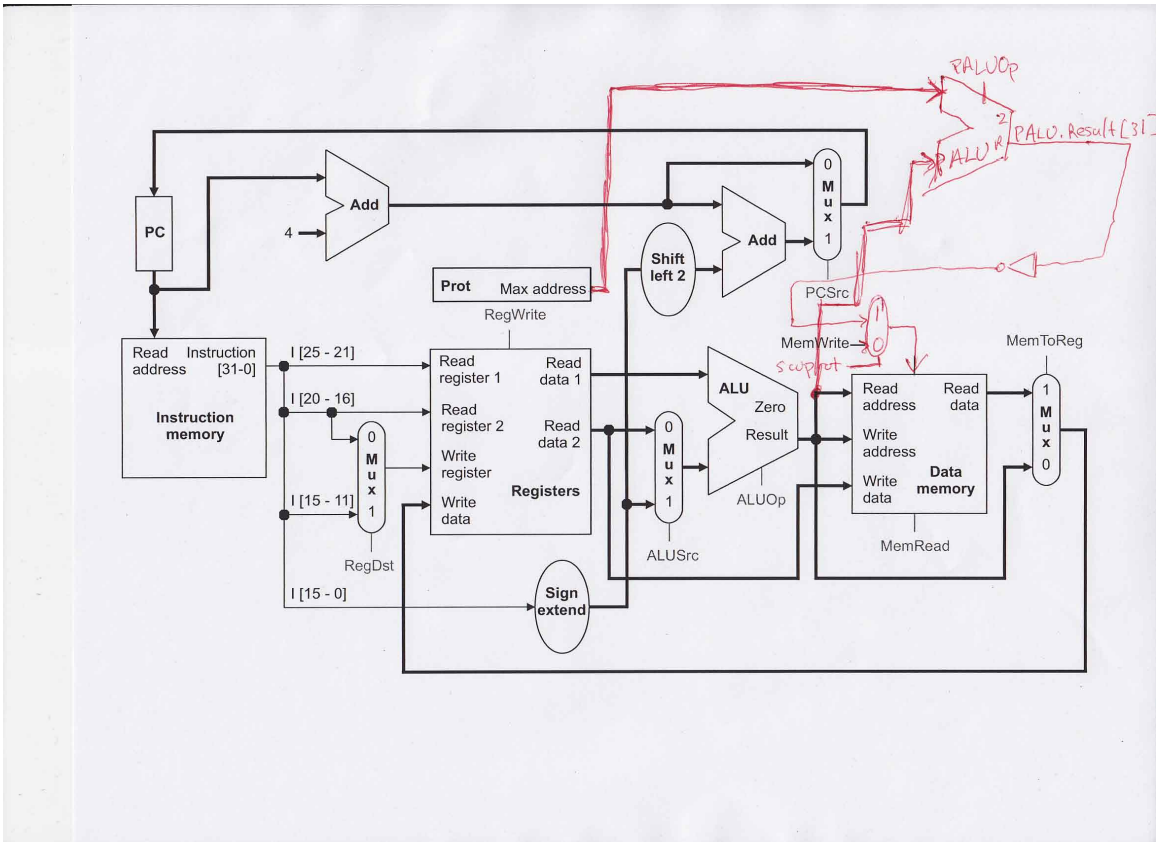
**Answer**:

```
calc:
        addiu $sp, $sp, -16     # allocate space on stack
        sw $ra, 0($sp)          # save return address

        lb $t0, 0($a0)          # get first byte
        sw $t0, 4($sp)          # store for later

num_or_op:
        li $t1, '0'             # is it a digit?
        blt $t0, $t1, op
        li $t1, '9'
        bgt $t0, $t1, op

num:
        jal getnum              # it's a digit; get it and return
        j end                   # just pass along $v0 and $v1

op:
        ## skip op
        addiu $a0, $a0, 2

        ## get first arg
        jal calc                # recursive call
        sw $v0, 8($sp)          # store first result
        move $a0, $v1           # get pointer to next op

        ## get second arg
        jal calc                # recursive call
        sw $v1, 12($sp)         # store pointer to next op

        ## compute
        lw $a0, 4($sp)          # get op byte
        lw $a1, 8($sp)          # get first arg
        move $a2, $v0           # get second arg
        jal compute             # do computation

        ## return
        lw $v1, 12($sp)         # restore pointer to next op

end:
        lw $ra, 0($sp)          # restore return address
        addiu $sp, $sp, 16      # deallocate stack space
        jr $ra                  # return
```

3. [30 Points] **Datapath.**

The NSA has taken notice of your knowledge of buffer overflow attacks and methods of prevention and has recruited you to help build a new super-secure MIPS processor. They want to add a new instruction `swprot $rt, imm16($rs)` to the MIPS instruction set which performs a bounds-check store on the instruction's effective memory address. In order to do so the NSA's architects have added a Prot register to the datapath which contains the maximum memory address that can be stored to. If the effective address is less than or equal to the value in the Prot register, the store is allowed. Otherwise, the store is not performed.

(a) Draw the necessary modifications to implement the `swprot` instruction on the figure of the single-cycle datapath provided below, adding functional units and logical primitives as necessary.




**Answer**:

The new logic for `swprot` is drawn in red.

(b) Including new signals you've added, what are the values of the control signals required to implement `swprot`? You may use opcodes for ALUOp (e.g. add, sub, xor) and values of `X` to indicate a don't care.

| RegDst | RegWrite | ALUSrc | ALUOp | PCSrc | MemWrite | MemRead | MemToReg |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

New signals and their values:

**Answer**:

| RegDst | RegWrite | ALUSrc | ALUOp | PCSrc | MemWrite | MemRead | MemToReg |
|---|---|---|---|---|---|---|---|
| X | 0 | 1 | add | 0 | 1 | 0 | X |

New signals: PALU.Op = sub, swprot = 1

4. [15 Points] **Pipeline Hazards.**

   Consider the sequence of MIPS instructions below:

```
sub  $8, $4, $1
lw   $2, 8($5)
or   $3, $2, $4
andi $5, $5, 0x3
add  $6, $5, $2
```

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

   **Answer**: There are dependences between the `lw` and the `or` instruction via `$2`, between the `andi` and the `add` via `$5`, and between the `lw` and the `add` via `$2`.

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

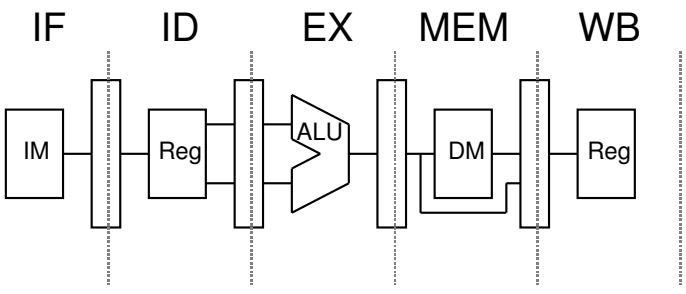
   (b) With a single reordering, rearrange the instructions into a new schedule that will execute without any stalls on a 5-stage pipelined processor with forwarding. For reference, you may refer to the pipeline diagram below.

   **Answer**:
   We need to avoid the 1-cycle load-use stall between the `lw` and the `or`. We can simply put the `sub` in between.

```
lw   $2, 8($5)
sub  $8, $4, $1
or   $3, $2, $4
andi $5, $5, 0x3
add  $6, $5, $2
```

   Another apporach would be to swap the `or` and the `andi`.

```
sub  $8, $4, $1
lw   $2, 8($5)
andi $5, $5, 0x3
or   $3, $2, $4
add  $6, $5, $2
```



**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**:
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
sub  $8, $4, $1
or   $3, $2, $4
add  $6, $5, $2
```

**Rubric:**
-4 points for each unnecessary stall.

5. [15 Points] **Performance.**

Suppose we have the following latencies for each of the functional units in our datapath:

| Functional Unit | Latency |
|---|---|
| Instruction Memory | 2 ns |
| Register File | Read: 1 ns, Write: 1 ns |
| ALU | 2 ns |
| Data Memory | 2 ns |

(a) Considering the above latencies, how long must the clock cycle be for the pipelined processor? (For reference, there is an image of the pipelined datapath on the next page)

**Answer**: 2 ns, as long as the maximum functional unit latency

**Rubric:**
3 points for correct answer

(b) How long does it take to complete each instruction in the pipelined processor?

**Answer**: 10 ns, clk_cycle_len * num_stages

**Rubric:**
3 points for correct answer

(c) How long does it take for `n` instructions to complete in the pipelined processor?

**Answer**: (8 + 2N) ns, (num_stages-1 + N) * clk_cycle_len or intuitively, time_to_fill + one_instr_ex_time * N

**Rubric:**
4 points for correct answer

(d) What is the approximate speedup of executing 100 instructions in the pipelined processor versus in the single-cycle processor? You may assume the instructions execute without any stalling.

**Answer**: 4, formulaically: 8N / (8 + 2N) = 800 / 208

**Rubric:**
5 points for correct answer