MIPS and SPIM

This is written as a brief introduction to mips and spim for students doing the CS75 course project.

MIPS (www.mips.com) is a reduced instruction set computer (RISC), meaning that it contains a small number of simple instructions (x86 is an example of a complex instruction set computer (CISC)) All MIPS instructions are the same size (4 bytes), and there is a simple five stage instruction pipeline.

MIPS is a register based architecture, meaning that instruction operands are in registers. Java VM, on the other hand, is a stack based architecture where instruction operands are pushed and poped off an instruction stack. MIPS has a three-address instruction set (instructions have 3 operands). For example:

```
add  $t0, $t1, $t2    # add values in reg t1 and t2 and put result in t0
```

MIPS is also a load/store architecture, meaning that values are loaded from memory address into registers and stored from registers to memory address. For example:

```
lw  $t5, 4($t1)   # load $t5 with the word at mem address 4 + address value stored in register $t1
```

MIPS load and store instructions are the only ones that can directly address memory.

MIPS assembely

labels

A label is a string of chars, digits, dot, or underscore charaters, followed by a colon that is on a line by itself. labels are used to associate a name with a line of MIPS code (i.e. an address of an instruction). Some MIPS instructions can have label operands. For example:

```
.L2:                # a label
  ...
  la $t0, .L2      # load the address assocated with the label .L2 into $t0
  ...
  b .L2            # branch to the instruction at label .L2
```

assembler directives:

These are a subset of the MIPS assembler directives that are used to tell the assembler something about names that are used in a MIPS assembly file:

```
.text    # the succeeding lines contain instructions

.data    # the succeeding lines contain data

.globl name  # name is global symbol (visible to code in other files)

.asciiz  "a string\n"   # stores a null terminated string in memory

example:
--------
# single line comments in MIPS start with the '#' character

        .text            # what follows are instructions
        .globl main      # main is a global name (can be referenced in other files)

main:                    # main is a label (a name assoc w/memory location)

        sub $sp, $sp, 32
        ...
```

```
loop:                   # loop is a local label


        .data            # what follows are data

str:                                    # str is a local label
       .asciiz  "hello world\n"         # this is a string associated with str
```

registers 32 general purpose registers (32 bits each)

```
$at, $k0, $k1: reserved for OS and assembler

$a0-$a3:  used to pass first 4 arguments to routine (rest passed on stack)

$v0, $v1: used for return values from functions

$t0-$t9:  caller-saved registers, used to hold temporaries,
          not perserved across calls

$s0-$s7: callee-saved registers, hold long lived data
         should be preserved across calls

$gp:  global pointer points to middle of 64K block in static data segment

        addresses in .data are given relative to it:

        lw  $v0, 0x20($gp)

$sp: the stack pointer, points to last location on the stack

$fp: frame pointer

$ra: the return address from procedure call
```

procedure call convention

```
  stack:  sp points to the top of the stack (grows into lower addresses)
  ------
          fp points to the bottom of the current stack frame

             -----------------
sp --------> local variables

             saved registers
             -----------------
fp --------> argument 5
             argument 6
             ...

  function call:
  =============

  caller:
  ------
  (1) saves registers a0-a3 and t0-t9 if caller needs their values upon return

        callee will assume these are okay to use

  (2) sets up arguments

      passed arguments:  1st four passed in registers a0-a3

                     remaining arguments are pushed onto the stack
                     appear at the begining of the called func's stack frame
```

```
(3) execute a jal instruction to jump to address of callee's the first instr

      jal saves return address in register $ra

callee:
------
(1) allocate stack memory for the frame (change value of $sp)

(2) save callee-saved registers in the frame

    $s0-$s7, $fp, $ra: these are registers the caller expects to be restored

      $fp: caller's frame pointer

      $ra: return address

      others: only if callee uses them

              in your code, you will always save all of them

      (3) change value of $fp to point to bottom of callee stack frame


function return
===============

callee:
------
 (1) return value placed in $v0

 (2) restore all callee-saved registers that were saved on func entry

 (3) pop the stack frame by adding the frame size to $sp

 (4) return by jumping to the address in register $ra

caller:
-------
 (1) may "pop" arguments off the stack
```

MIPS assembley hides some details of real instruction set:

1. assembly code doesn't have delay slot instructions (you don't have to worry about delay slot instructions in assembly code you write)

   Some MIPS instructions need an extra cycle to execute (branch, jump, load and store instructions). A compiler that generates MIPS machine code, either put a nop instruction following these instructions (in their delay slot), or tries to put in the delay slot an instruction that is executed no matter if the branch is taken or not.

   When you write MIPS assembly code, you do not need to insert a delay slot instruction. For example if your assembly looks like:

   ```
   bne
   next instr    # only executed if bne is not taken
                 # this is NOT a delay slot instruction
   ```

   The MIPS assembler will automatically put a nop in the delay slot of the bne, producing MIPS code that looks like:

   ```
   bne
   nop           # delay slot instruction
   next instr    # only executed if bne is not taken
                 # this is NOT a delay slot instruction
   ```

2. MIPS assembly has some pseudo-instructions that do not correspond to real MIPS instructions, but that make

writing assembly code a bit easier. The MIPS assembler will re-write these instructions as real MIPS.

The MIPS assembler will fill delay slots and generate real MIPS code for pseudo-instructions.

Some MIPS instructions

MIPS is one of the most RISC of the RISC instruction sets, and still we will end up using a subset of its instrutions. Here are some of the instructions you may use:

```
Arithmetic Instructions
-----------------------
add  rd,  rs,  rt              #  rd <-- rs + rt
sub  rd, rs, rt
mulo rd, rs, rt                #  rd <-- rs * rt
div  rd, rs, rt                #  rd <-- rs / rt
neg rdest, rsrc                #  rdest <-- -rsrc


# there are also immediate forms you could use:
ori rd, rs, immed             # rd <-- rs || immed


comparison instructions
----------------------
slt rd, rs, rt           #   rd <-- 1 if rs < rt,  0 otherwise
sle rd, rs, rt           #   rd <-- 1 if rs <= rt,  0 otherwise
sgt rd, rs, rt           #   >
sge rd, rs, rt           #   >=
seq rd, rs, rt           #   ==
sne rd, rs, rt           #   !=

branch instructions
-------------------
b    label              # unconditional branch to label
beq  rs, rt, label      # conditional branch to label if rs == rt
bgez rs, label          # conditional branch to label if rs >= 0
...                     # and more branch instructions that will be useful

# use branches with other ops to do && || and !
#
#  x || y   if x is non-zero, then x || y is 1
#           else if y is non-zero, then x || y is 1
#           else x || y is 0


j   target              # unconditional jump to instruction at target

jal target              # unconitionally jump to instruction at target
                        # save addresss of next instruction in $ra (func call)

jr  rs                  # jump to instruction whose address is in reg rs

load and store instructions
---------------------------

lb rd, addr       # load the byte at addr into $rd
                  # (see addressing modes below for how addr can be specified)

lw rd, addr       # load the word (32 bit value) at addr into $rd

la rd, addr       # load the value of addr into $rd (NOT the contents at addr)

li rd, immed      # load the value immed into $rd


sb  rt, addr      # store the low byte from $rt to address addr
```

```
sw  rt, addr       # store the word from $rt to address addr

# example: (global addresses are given as offset from $gp)

  lw      $t5,    0($gp)        # loads global value at offset 0 from gp into t5

        la      $t4,    0($gp)        # loads the address of the global at offset 0

addressing modes:
-----------------

lw  rd,  imm               # load value at address imm

lw  rd,  ($rs)             # load value at address contents of reg rs

lw  rd,  imm($rs)          # load value at address contents of reg rs + imm

li  rd,  imm               # load immediate:  load value imm into register rd

data movement instructions
---------------------------

move rd, rs       # move value in register rs to rd
```

SPIM

Spim (pages.cs.wisc.edu/~larus/spim.html) is a MIPS simulator. Input to spim is a MIPS assembly file that spim will execute.

some of the things that spim and xspim support that we will use:

1. some debugging options like steping through instruction of single instructions
2. seeing contents of registers and stack
3. some system calls

```
we will use print_int and print_str to print "\n"

    li  $v0, 1      # load v0 with system call number (print_int)
    li  $a0, 5      # load a0 with integer to print
    syscall         # print it
```

4. shows real instructions in place of pseudo-instructions but does not re-order instructions to fill the delay slot