MIPS coding

Review

- Shifting
  - Shift Left Logical (sll)
  - Shift Right Logical (srl)
  - Moves all of the bits to the left/right and fills in gap with 0's
  - For most cases, equivalent to multiplying/dividing by 2^n where n is the number of bits being shifted
  - Be careful of overflow and using srl on negative numbers

Review

- Branching
  - Branch If Equal (beq)
  - Branch If Not Equal (bne)
  - Jump (j)
  - Changes point of execution:
    - Conditionally only if clause is true for beq/bne (otherwise, the next instruction is the one executed)
    - Unconditionally for j
  - Used to make if statements and loops in higher level languages

In Class Exercise

- Write the MIPS assembly code for the following C code segment:

```
If (A[1] < A[2]) {
        A[0] = A[1] & 5;
}
else {
        A[0] = A[2] & 5;
}
```

- Assume the starting address of array A is stored in $s0. Use only the instructions
- covered in class, i.e. add, addi, sub, or, ori, and, andi, xor, xori, nor, lw, sw, srl, sll, beq,
- bne, j.

In Class Exercise – C Code

```
If (A[1] < A[2]) {
        A[0] = A[1] & 5;
}
else {
        A[0] = A[2] & 5;
}
```

In Class Exercise – Set up

```
ori $t0, $zero, 5                # Set up constant used in if
lw $t1, 4($s0)                   # Get value from A[1] and place in $t1
lw $t2, 8($s0)                   # Get value from A[2] and place in $t2
If (A[1] < A[2]) {
        A[0] = A[1] & 5;
}
else {
        A[0] = A[2] & 5;
}
sw $t0, 0($s0)                   # Store $t0 to A[0]
```

In Class Exercise – If Bodies

```
ori $t0, $zero, 5                # Set up constant used in if
lw $t1, 4($s0)                   # Get value from A[1] and place in $t1
lw $t2, 8($s0)                   # Get value from A[2] and place in $t2
If ($t1 < $t2) {
        and $t0, $t1, $t0        # And constant and A[1]
}
else {
        and $t0 $t2, $t0         # And constant and A[2]
}
sw $t0, 0($s0)                   # Store $t0 to A[0]
```

In Class Exercise – Change Compare Operator

```
ori $t0, $zero, 5                # Set up constant used in if
lw $t1, 4($s0)                   # Get value from A[1] and place in $t1
lw $t2, 8($s0)                   # Get value from A[2] and place in $t2
sub $t3, $t1, $t2                # Negative if <, Zero/Positive if >=
srl $t3, $t3, 31                 # Discard everything but the sign bit
If ($t3 != $zero) {
        and $t0, $t0, $t1        # And constant and A[1]
}
else {
        and $t0, $t0, $t2        # And constant and A[2]
}
sw $t0, 0($s0)                   # Store $t0 to A[0]
```

In Class Exercise – Change If Statement to BEQ

```
        ori $t0, $zero, 5       # Set up constant used in if
        lw $t1, 4($s0)          # Get value from A[1] and place in $t1
        lw $t2, 8($s0)          # Get value from A[2] and place in $t2
        sub $t3, $t1, $t2       # Negative if <, Zero/Positive if >=
        srl $t3, $t3, 31        # Discard everything but the sign bit
        beq $t3, $zero, ELSE
        and $t0, $t0, $t1       # And constant and A[1]
        j EXIT                  # Skip over ELSE branch
ELSE:
        and $t0, $t0, $t2       # And constant and A[2]
EXIT:
        sw $t0, 0($s0)          # Store $t0 to A[0]
```

In Class Exercise – Convert to Exercise Given (<=)

```
        ori $t0, $zero, 5          # Set up constant used in if
        lw $t1, 4($s0)             # Get value from A[1] and place in $t1
        lw $t2, 8($s0)             # Get value from A[2] and place in $t2
        sub $t3, $t1, $t2          # Negative if <, Zero/Positive if >=
        bne $t3, $zero, REST       # Skip if the two numbers are not equal
        and $t0, $t0, $t1          # Same as true branch below
        j EXIT                     # Skip over everything else
REST:
        srl $t3, $t3, 31           # Discard everything but the sign bit
        beq $t3, $zero, ELSE
        and $t0, $t0, $t1          # And constant and A[1]
        j EXIT                     # Skip over ELSE branch
ELSE:
        and $t0, $t0, $t2          # And constant and A[2]
EXIT:
        sw $t0, 0($s0)             # Store $t0 to A[0]
```

slt, slti

- `slt $t3, $t1, $t2`
  - set `$t3` to be 1 if `$t1 < $t2`; else clear `$t3` to be `0`.
  - "Set Less Than."
- `slti $t3, $t1, 100`
  - set `$t3` to be 1 if `$t1 < 100`; else clear `$t3` to be `0`.

Using `slt`

```
  slt $t3, $t1, $t2
  beq $t3, $zero, ELSE
  andi $t0, $t1, 5
  j EXIT
ELSE:
  andi $t0, $t2, 5
EXIT:
```

Complete MIPS code

- The text segment in the source code usually starts with

```
    .text
    .globl main
main:
```

  where ``main'' is the label associated with the address of the first instruction of the code.
- And the code usually ends with

```
    li $v0,10 # telling the simulator to stop
    syscall
```

- Comment with `#'

In Class Exercise

```
            .text
            .globl MAIN
MAIN:
            ori $t0, $zero, 5          # Set up constant used in if
            lw $t1, 4($s0)             # Get value from A[1] and place in $t1
            lw $t2, 8($s0)             # Get value from A[2] and place in $t2
            sub $t3, $t1, $t2          # Negative if <, Zero/Positive if >=
            bne $t3, $zero, REST       # Skip if the two numbers are not equal
            and $t0, $t0, $t1          # Same as true branch below
            j EXIT                     # Skip over everything else
REST:
            srl $t3, $t3, 31           # Discard everything but the sign bit
            beq $t3, $zero, ELSE
            and $t0, $t0, $t1          # And constant and A[1]
            j EXIT                     # Skip over ELSE branch
ELSE:
            and $t0, $t0, $t2          # And constant and A[2]
EXIT:
            sw $t0, 0($s0)             # Store $t0 to A[0]
            li $v0, 10                 # Sets the syscall operation
            syscall                    # Exits the program
```

SPIM

- Run codes with **SPIM**. SPIM is a simulator.
  - Use any editor to write the source file, save it as an .asm file.
  - Run SPIM, load the source file.
  - F10 to step through the code. Monitor how the registers change.
  - F5 to run the code
  - Can set breakpoints for debugging
- SPIM can be downloaded at
  http://sourceforge.net/projects/spimsimulator/files/
- Lots of good references online, like
  https://www.cs.tcd.ie/~waldroj/itral/spim_ref.html

Working with the simulator

- Can check
  - How the program runs
  - How the instructions are encoded, addressed
  - How to monitor the change of the registers
  - Later, how the memory is used to store data

Some Comments

- Being able to write if-else, we can have all other fancy things like for loop, while loop….
- That is why we do not have an instruction for the for loop or while loop, but we build it from the if-else.

Compiling a while loop in C

- How to translate the following to MIPS assembly?

```
while (save[i] == k)
  i += 1;
```

  - We first translate into a C program using if and goto

```
Loop: if (save[i] != k) goto Exit;
      i = i + 1;
      goto Loop;
Exit:
```

Compiling a while loop in C

- Assume that `i` and `k` correspond to registers `$s3` and `$s5` and starting address of array save is in `$s6`

```
while (save[i] == k)
  i += 1;
```

```
Loop: if (save[i] != k) goto Exit;
      i = i + 1;
      goto Loop;
Exit:
```

Compiling a while loop in C

- Assume that `i` and `k` correspond to registers `$s3` and `$s5` and starting address of array save is in `$s6`

```
while (save[i] == k)
  i += 1;
```

```
Loop: if (save[i] != k) goto Exit;
      i = i + 1;
      goto Loop;
Exit:
```

```
Loop: sll  $t1, $s3, 2     # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6   # $t1 = address of save [i]
      lw   $t0, 0($t1)     # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3, $s3, 1     # i = i + 1
      j    Loop            # go to Loop
Exit:
```

While Loop

- How many instructions will be executed for the following array save?

  10, 10, 10, 10, 10, 10, 10, 10, 10, 0

  - Assume that k = 10 and i = 0 initially

```
Loop: sll  $t1, $s3, 2     # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6   # $t1 = address of save [i]
      lw   $t0, 0($t1)     # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3, $s3, 1     # i = i + 1
      j    Loop            # go to Loop
Exit:
```

  - (6 loop lines * 9 loops) + 4 lines in last iteration
  - = 58 lines

Optimized

```
      sll  $t1, $s3, 2    # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6  # $t1 = address of save[i]
      lw   $t0, 0($t1)    # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit # go to Exit if save[i] ≠ k
Loop: addi $s3, $s3, 1    # i = i + 1
      addi $t1, $t1, 4    # $t1 = address of save[i]
      lw   $t0, 0($t1)    # Temp reg $t0 = save[i]
      beq  $t0, $s5, Loop # go to Loop if save[i] = k
Exit:
```

- How many instructions now?
  - Assume k = 10 and i = 0 initially

    10, 10, 10, 10, 10, 10, 10, 10, 10, 0
  - 4 preloop lines + (4 loop lines * 9 loop iterations) + 4 lines in last iteration
  - = 44 lines

The loop code

```
    .data
save:.word 10, 10, 10, 10, 10, 11, 12,

    .text
    .globl main
main:
    li $s3, 0
    li $s5, 10
    la $s6, save

Loop:
    sll $t1, $s3, 2
    add $t1, $t1, $s6
    lw $t0, 0($t1)
    bne $t0, $s5, Exit
    addi $s3, $s3, 1
    j Loop
Exit:

done:
    li $v0, 10 # these two lines are to tell the simulator to stop
    syscall
```

Data segment and code segment

- The code has a **data** segment and a **code (text)** segment.
- The beginning of the data segment in the assembly source code is indicated as

  ```
  .data
  ```

  and followed by several declarations such as
  - `A:  .word 0,1,2,3,4,5,6,7,8,9`

    meaning an array of words whose starting address is associated with label ``A.''
  - Several notes:
    - It will allocate continuous spaces in the memory for the data
    - .word means everything is 4 bytes
    - save: is a **label** associated with the address of the first byte allocated. Like the label for the instructions, label for an address is also an address.