CSI 333 – Lecture 13

MIPS Assembly Language (MAL): Part I
(Basic Instructions)
Ref: Chapter 2 and Appendix B of Patterson & Hennessey.

Some Details Regarding MIPS:

- RISC architecture.
- Word size = 32 bits (4 bytes).
- Word and byte addressable. (Each word or byte address is 32 bits long.) The address of each word must be a multiple of 4.
- Maximum memory size = $2^{30}$ words (or $2^{32}$ bytes).
- 32 registers (denoted by $0$ through $31$). Each register is 32 bits long.
Some Details Regarding MIPS (continued):

- All arithmetic operations involve registers. (Reason: speed)

- Load/Store Architecture (i.e., only load and store instructions need operand fetch).

Special Roles of Some Registers:

- Register $0$ always contains the value 0 (zero).

- Register $1$ is reserved by the assembler.

- Register $29$ is used as the stack pointer for the system stack.

- Registers $2$ ($v0$), $4$ ($a0$) and $5$ ($a1$) are used in performing I/O operations using syscall instruction.

- Register $31$ is normally used to store return address for procedure calls.
Goals:

1. To introduce MAL instructions.
2. To convey the level of detail involved in writing MAL programs.

Example 1 – A Segment in C:

```c
int f, g, h, i, j; f = (g + h) - (i + j);
```

Register Assignment:

- f: $15  g: $16  h: $17  i: $18  j: $19

Equivalent MAL Segment:

```mal
add $8, $16, $17  #$8 has g + h.
add $9, $18, $19  #$9 has i + j.
sub $15, $8, $9  #$15 has f.
```
Example 2 – A Segment in C:

```c
int f, g, h, i, j;
if (i == j)
    f -= i;
else
    f = g + h;
```

Register Assignment: (same as in Example 1)

```
f: $15 g: $16 h: $17 i: $18 j: $19
```

Equivalent MAL Segment:

```mal
beq $18, $19, then
add $15, $16, $17 #f = g+h.
exit
then: sub $15, $15, $18 #f -= i.
exit:
```
Example 3:  

C segment: Assume all variables are of type int.

```c
if (a > 0) {
    x += y; p *= q;
}
```

Register Assignment:

```
a: $15  x: $16  y: $17  p: $18  q: $19
```

Equivalent MAL Segment:  (Version I)

```
bg $15, $0, then
j  end_if
then:  add $16, $16, $17
mul $18, $18, $19
end_if:
```
Example 3 – Equivalent MAL Segment:  (Version II)

```
ble $15, $0, end_if
add $16, $16, $17
mul $18, $18, $19
end_if:
```

Example 4: Translating a for loop.

**C Segment:** Assume all variables are of type int.

```c
j = 0;
for (i = 1; i <= n; i++) {
    j += 2 * i;
}
```

**Register Assignment:**

```
i: $15  j: $16  n: $17
```

Also, $18 is used as a temporary.

**Equivalent MAL Segment:** Handout 13.1.
Example 5: Translating a switch statement.

**Segment in C:** Assume all variables are of type int.

```c
switch(k) {
    case 0: f = i+j; break;
    case 1: f = g+h; break;
    case 2: f = g-h; break;
    case 3: f = i-j; break;
}
```

**Register Assignment:** Variables $f$, $g$, $h$, $i$, $j$ and $k$ are in registers $15$, $16$, $17$, $18$, $19$ and $20$ respectively. Register $4$ is used for scratch work.

**Equivalent MAL Segment:** Handout 13.2.
List of Important MAL Instructions

**Load, Store and Move:** (General form)

- `lw` R, label #Load reg.
- `li` R, constant #Load immediate.
- `sw` R, label #Store reg.
- `move` Rd, Rs #Reg. to reg.

**Example:**

```
.data
val: .word 24
.text
lw $5, val
li $6, -21
move $7, $6
sw $7, val
```

**Arithmetic Instructions:** (General form)

- `op` Rd, Rs1, Rs2
Possible Opcodes:

add  sub  mul  div  rem

Examples and Notes:

- The instruction
  
  sub  $5, $3, $7

  subtracts the contents of $7 from that of $3 and stores the result in $5.

- The instruction
  
  div  $5, $3, $7

  divides the contents of $3 by the contents of $7 and stores the quotient in $5. (The remainder is ignored.)
Examples and Notes (continued):

- The instruction

  \[
  \text{rem} \quad \$5, \$3, \$7
  \]

  divides the contents of $3$ by the contents of $7$ and stores the remainder in $5$. (The quotient is ignored.)

Bitwise Operators:

  and  or  xor  nor  not

Examples:

  and  \$3, \$5, \$7
  nor  \$6, \$9, \$8
  not  \$4, \$5
Shift Instructions: (General form)

sll  Rd, Rs, const  
srl  Rd, Rs, const  
sra  Rd, Rs, const

Notes on Shift Instructions:

- The sll and srl instructions fill the vacated spots with zeros.
- The sra instruction fills the vacated spots with the sign bit.

Examples:

sll  $5, $5, 4  
srl  $6, $9, 3  
sra  $10, $10, 2
Jump Instructions:

(a) Unconditional Jump:

    j label
    b label

(b) Conditional Jump:

    beq  R1, R2, label
    beqz R1, label

Other Possible Jump Conditions:

    bne    blt    bgt    ble    bge
    bnez   bltz   bgtz   blez   bgez
**Example 1:** See Handout 13.3.

**Notes on Example 1:**

- No I/O statements are used.
- The `.data` line begins the Data Segment of the program.
- The `.text` line begins the Text Segment of the program.
- The `.globl` line specifies the label of the first instruction that should be executed.
- The program exits using a syscall instruction.
The syscall Instruction

- Provided by the spim system.
- Think of syscall as a macro (expanded by the assembler).
- The syscall instruction is used for I/O as well as for stopping the program.
- The operation to be carried out is specified as as a command (in register $v0) to syscall.

**Additional Notes on syscall:**

- $v0 is a synonym for $2. This register must contain the command for syscall. This register also contains the return value (if any) produced by executing the command.
- $a0 and $a1 are synonyms for $4 and $5 respectively. These registers must contain suitable values if such values are needed for the command.
Using syscall to Read an Integer

**Steps:**

1. Move the value 5 (the code for read-int command) into $v0.
2. Issue syscall. (The integer read from the keyboard is returned in $v0.)

**MAL Code:**

```
li $v0, 5
syscall
```
Using syscall to Print an Integer

Steps:
1. Move the value to be printed into $a0.
2. Move the value 1 (the code for print-int command) into $v0.
3. Issue syscall.

MAL Code: Assume that we want to print the value in $12.

```
move   $a0, $12
li     $v0, 1
syscall
```
Using syscall to Print a String

**Steps:**

1. The string to be printed must have the terminating '\0' character.
2. Store in $a0 the starting address of the string to be printed.
3. Move the value 4 (the code for print-string command) into $v0.
4. Issue syscall.

**MAL Code:** The following segment prints the string Hello followed by the newline character.

```
.data
hstr: .asciiz "Hello\n"
.text
la $a0, hstr
li $v0, 4
syscall
```
Using **syscall** to Read a String

**Note:** Let $n$ denote the maximum length of the string to be read.

**Steps:**

1. Store in $a0$ the starting address of the buffer into which string must be read. (The buffer must have at least $n + 1$ bytes, to hold the $n$ characters from the input plus the '\0' character.)

2. Store the value $n + 1$ in $a1$.

3. Move the value 8 (the code for read-string command) into $v0$.

4. Issue syscall.

**Remarks:**

- The system adds the '\0' character at the end.
- Note the similarity with the `fgets` function of C.
**MAL Code:** The following MAL segment prompts the user for a string and reads in a string consisting of at most 10 characters, including the newline character.

```
.data
hstr: .asciiz "Input: "
buf: .space 11
.text
#Prompt the user.
    la     $a0, hstr
    li     $v0, 4
    syscall
#Read string.
    la     $a0, buf
    li     $a1, 11
    li     $v0, 8
    syscall
```
1 Send the command 10 to syscall.

**MAL Code:**

```
li $v0, 10
syscall
```

**Summary of syscall:**

```
Command ($v0)
Arg. 1 (if any) ($a0)
Arg. 2 (if any) ($a1)
);
```

**Note:** For further information on syscall, refer to Appendix B of Patterson & Hennessey.
**Example 2:** Handout 13.4.

**Notes on Example 2:**

- Uses `syscall` to do I/O as well as stop.
- The `.asciiz` directive specifies a string terminated by `\0` (as in C).
- The character `\n` denotes newline (as in C).
- The ASCII codes of '0' and '9' are used to check if the input character is a digit.

**Example 3:** Handout 13.5.
Saving and Restoring Registers

**Note:** Assume that we want to save and restore $2$ and $4$.

```assembly
.data
s2: .word 0
s4: .word 0
.text
    .<-- Code uses $2$ and $4$.
    sw $2, s2 #Save registers.
sw $4, s4
    .
    .<-- Code uses $2$ and $4$ for I/O (or other purposes)
    lw $2, s2 #Restore registers.
lw $4, s4
```

**More Common Method:** Use the system stack (to be discussed later).