# Comp lecture #13

## Elements of MAL

- Directives start with '.':
- .data data segment,
- .text code segment.
- Labels end with ':'
- v:, main:, loop: and endloop:.
- Registers start with '$'.
- Comments start with '#'.
- Instructions and macros (pseudo-instructions).
- Constants.

# Basic MAL Directives

| | |
|---|---|
| `.globl sym` | Declares `sym` as global (accessible from other files) |
| `.extern sym size` | Declares `sym` as global with specified `size` |
| `.ascii str` | Saves `str` in memory |
| `.asciiz str` | Saves `str` and adds '\0' at the end |
| `.byte b1, b2, ..., bn` | Sequentially saves `b1`, `b2`, ..., `bn` allocating 8 bits for each |
| `.half h1, h2, ..., hn` | Sequentially saves `h1`, `h2`, ..., `hn` allocating 16 bits for each |
| `.word w1, w2, ..., wn` | Sequentially saves `w1`, `w2`, ..., `wn` allocating 32 bits for each |
| `.dword dw1, dw2, ..., dwn` | Sequentially saves `dw1`, `dw2`, ..., `dwn` allocating 64 bits for each |
| `.float f1, f2, ..., fn` | Sequentially saves floats `f1`, `f2`, ..., `fn` |
| `.double d1, d2, ..., dn` | Sequentially saves double floats `d1`, `d2`, ..., `dn` |
| `.space n` | Allocates n bytes |
| `.align n` | Aligns data to 2^n bytes |

# Labels in MAL

- in the .data segment, data labels define addresses of variables.
- in the .text segment instruction labels define addresses of instruction.

# MIPS Registers

- MIPS has 32 main register and 32 co processor registers
- Size of a register is 32 bits
- Registers saves address, data and instruction
- Max mem = $2^{32}$ 4G bytes (doesnt make sense to use 64 bit if CPU is only 32-bit)
- All MIPS arithmetic operations use registers.
- Register's name starts with '$' and has two synonymic versions:
    - Number
    - symbolic name.

| Register Number | Conventional Name | Usage |
|---|---|---|
| $0 | $zero | Hard-wired to 0 |
| $1 | $at | Reserved for pseudo-instructions |
| $2 - $3 | $v0, $v1 | Return values from functions |
| $4 - $7 | $a0 - $a3 | Arguments to functions - not preserved by subprograms |
| $8 - $15 | $t0 - $t7 | Temporary data, not preserved by subprograms |
| $16 - $23 | $s0 - $s7 | Saved data registers, preserved by subprograms |
| $24 - $25 | $t8 - $t9 | More temporary registers, not preserved by subprograms |
| $26 - $27 | $k0 - $k1 | Reserved for kernel. Do not use. |
| $28 | $gp | Global Area Pointer (base of global data segment) |
| $29 | $sp | Stack Pointer |
| $30 | $fp | Frame Pointer |
| $31 | $ra | Return Address |

# MIPS Registers for Floating Point Operations

| Register Number | Conventional Name | Usage |
|---|---|---|
| $f0 - $f3 | - | Floating point return values |
| $f4 - $f10 | - | Temporary registers, not preserved by subprograms |
| $f12 - $f14 | - | First two arguments to subprograms, not preserved by subprograms |
| $f16 - $f18 | - | More temporary registers, not preserved by subprograms |
| $f20 - $f31 | - | Saved data registers, preserved by subprograms |

# MAL Data Types

| Type | Example | Size |
|---|---|---|
| Byte | `a: .byte 0` | 8 bit |
| Halfword | `b: .half 0` | 16 bit |
| Word | `c: .word 0` | 32 bit |
| Doubleword | `d: .dword 0` | 64 bit |

- **In MAL a type defines only the size of data.**
- **The interpretation of binary code depends on instruction.**

# Examples of Data Definitions

```
.data
  var1:  .byte  'A', 0xF3, 127, -1, '\n'
  var2:  .half  -10, 0xffff
  var3:  .word  0x12345678
  var4:  .float 12.3, -0.1
  var5:  .double        1.5e-10
  var6:  .dword 0x1234567812345678
  str1:  .ascii "i love mips\n"
  str2:  .asciiz        "zero-finished string"
  array: .space 100
```

- •asciiz differe from .ascii in that .asciiz terminates
-

# Source Code

- Each line of code can be
    - Instruction (?:label)
    - Single directive
    - Empty line
    - Comment
- Comment starts:#, ends:newline

# Special Instruction syscall

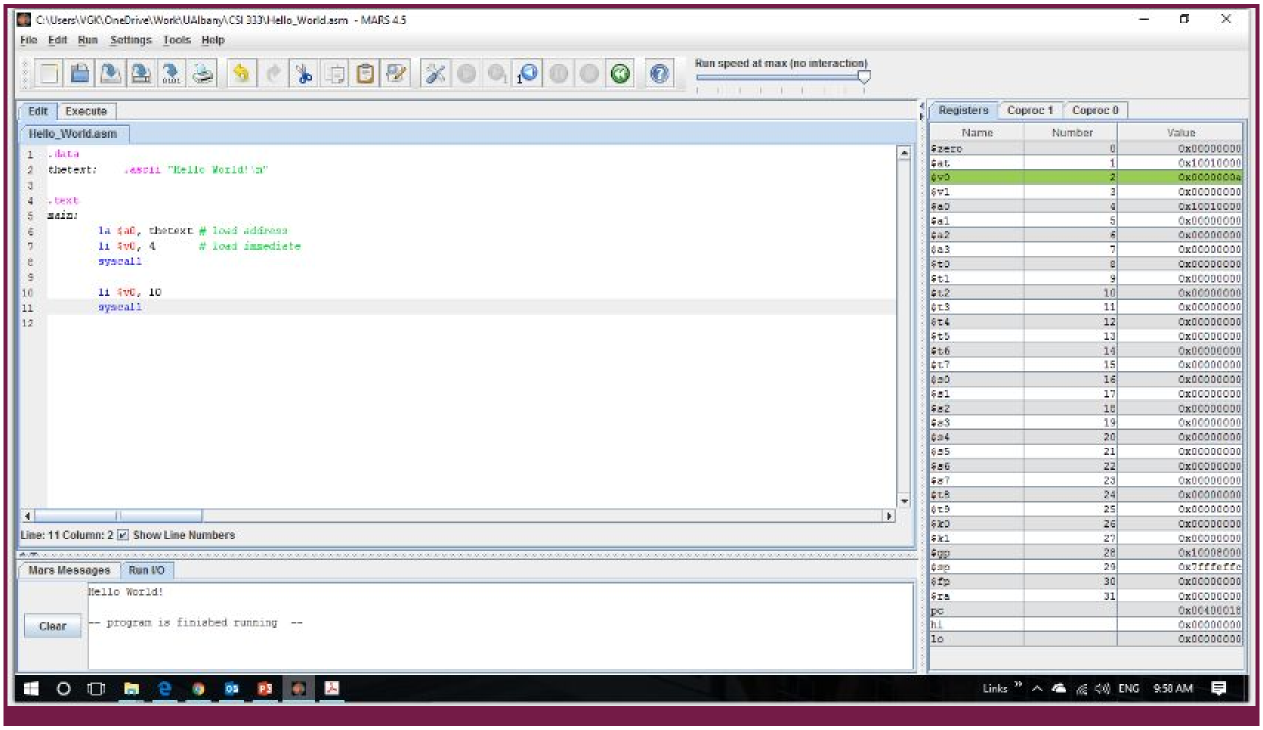
| Service | Code in $v0 | Arguments |
|---|---|---|
| Print integer | 1 | `$a0` = integer to be printed |
| Print float | 2 | `$f12` = float to be printed |
| Print double | 3 | `$f12` = double to be printed |
| Print string | 4 | `$a0` = address of string in memory |
| Read integer to $v0 | 5 | |
| Read float to `$f0` | 6 | |
| Read double to `$f0` | 7 | |
| Read string | 8 | `$a0` = memory address of string input buffer<br>`$a1` = length of string buffer (n) |
| Dynamic memory allocation | 9 | `$a0` = amount |
| Exit | 10 | |
| Print character | 11 | `$a0` = character to be printed |
| Read character in `$v0` | 12 | |

- Lets your code commmunicate with os
- If something goes wrong the OS usually tells you in C
- If you mess up your processor, you can destroy your machine
- Use software (simulators) to run your mips code

# *MAL Simulators*

- Direct experiments with CPU can destroy it
- Use simulators such as SPIM and MARS to test your code

# MAL Simulators




- First part data that defines the strings
- The list to your right are 32 registers
- Before execution all registers are zero
- The values in the register change as they need to
- Change value of regsiter v0, to 4 becuase next instruction is syscall
- Goes to register v0 to find what to do, print string at register (memory location) a0

- In MIPS you can execute your code step by step to see what is happening
- Exit called done by syscall, if  code does not do it itself
- Values placed into registers are binary
- Use hexadecimals to save space
- 0x tells you if a number is hexadecimal

## SPIM Interface

- Make sure code runs well with SPIM simulator

## MAL Instructions

- Psuedo-instructions in the previous example
  - la rdest, addr
- translates to
  - lui $at, hi(addr)
  - ori rdest, $at, lo(addr)

- Too hard to remember all instruction, but have to know where code is
- Use psuedo-instruction

## Types of Instructions

- Type R(register)
  - Three operands,
    - Destination register ($rd)
    - First argument ($rs)
    - Seconde argument ($rt)
    - add $t2, $t0, $t1
    - $t0 + $t1 -> $t2
- Type I (immediate)
  - Two registers and constant
  - addi $t3, $t2, 12
  - $t2 + 12 -> $t3

- Type J (jump)
  - one operand: new address for PC.
  - E.g. J 128.

TRANSlating C segments into MAL

- Assembly language can take a long time
- To introduce MAL instructions
- To convery level of detail involved

# Example 1

**A Segment in C:**

```
int f, g, h, i, j;
/* ... */
f = (g + h)-(i + j);
```

**Equivalent MAL Segment:**

Register Assignment:

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

```
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:**

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

**Equivalent MAL Segment:**

Register Assignment:

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

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

- Here the else statement comes right after tthe conditional and exits
- The then label comes to as representation of the if statement

# Example 3

**A Segment in C:**

Assume all variables are of type `int`.

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

**Equivalent MAL Segment:**

Register Assignment:

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

```
        bgt $15, $0, then
        j end_if
then:   add $16, $16, $17
        mul $18, $18, $19
end_if:
```

# alt

**A Segment in C:**

Assume all variables are of type `int`.

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

**Equivalent MAL Segment (alternative solution):**

Register Assignment:

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

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

# C and Assembly Conditional Operators

| C Conditional Operator | MIPS Assembly Instruction |
|---|---|
| `a == b` | `beq $t0, $t1, then` |
| `a != b` | `bne $t0, $t1, then` |
| `a < b` | `blt $t0, $t1, then` |
| `a > b` | `bgt $t0, $t1, then` |
| `a <= b` | `ble $t0, $t1, then` |
| `a >= b` | `bge $t0, $t1, then` |
| `a == 0` | `beqz $t0, then` |

# Loops

## Translating C Segments into MAL – For Loop

**A Segment in C :**

Assume all variables are of type `int`.

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

See **Handout 13.1**.

**Equivalent MAL Segment:**

Register Assignment:

i: \$15   j: \$16   n: \$17   temporary: \$18.

```
    move $16, $0      #Set j = 0.
    addi $15, $0, 1     #Set i = 1.
loop:  bgt $15, $17, end_loop
    #If i > n, loop is over.
    mul $18, $15, 2     #Set temp = 2 * i.
    add $16, $16, $18   #Set j += temp.
    addi $15, $15, 1    #i++
    j loop
end_loop:
```

## Important MAL Instructions: Load, Store and Move

**General form:**

```
lw R, label     #Load to register from memory.
li R, constant   #Load to register immediate.
sw R, label     #Store in memory from register.
move Rd, Rs     #Move from register to register.
```

**Example:**

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

## Important MAL Instructions: Arithmetic Instructions

General form:

```
add Rd, Rs1, Rs2      #Add
sub Rd, Rs1, Rs2      #Subtract
mul Rd, Rs1, Rs2      #Multiply
div Rd, Rs1, Rs2      #Divide
rem Rd, Rs1, Rs2      #Remainder
```

Examples:

```
sub $5, $3, $7
#subtracts the contents of $7 from that of $3 and
#stores the result in $5.
div $5, $3, $7
#divides the contents of $3 by the contents of $7 and
#stores the quotient in $5. (The remainder is ignored.)
rem $5, $3, $7
#divides the contents of $3 by the contents of $7 and
#stores the remainder in $5. (The quotient is ignored.)
```

## Important MAL Instructions: Logical Bitwise Operators

Opcodes:

```
and
or
xor
nor
not
```

| NOR Truth Table | | |
|---|---|---|
| Input A | Input B | Output Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Examples:

```
and $3, $5, $7
nor $6, $9, $8
not $4, $5
```

# Important MAL Instructions:

# Shift Instructions

**General form:**

```
sll Rd, Rs, const #Shift left logical
srl Rd, Rs, const #Shift right logical
sra Rd, Rs, const #Shift right arithmetic
```

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
```

# Important MAL Instructions: Jump Instructions

- Unconditional Jump:

```
j label   #Jump using absolute addresses
b label   #Branch using PC-relative addresses
```

*Note:* The difference is important for position-independent code.

- Conditional Jump:

```
beq R1, R2, label
beqz R1, label
```

- Other Possible Jump Conditions:

`bne blt bgt ble bge` (see above)

`bnez bltz bgtz blez bgez`

# A Simple MAL Program

For 13.3 notes

## The syscall Instruction

- Used for I/O and for stopping the program
- The operation to be carried out is specified as a command (in register $v0) to 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

MAL Code:

```
li $v0, 5
syscall
```

### Using syscall to Print an Integer

**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

**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, cont'd

| Service | $v0 | Arguments |
|---|---|---|
| Print integer | 1 | $a0 = integer to be printed |
| Print float | 2 | $f12 = float to be printed |
| Print double | 3 | $f12 = double to be printed |
| Print string | 4 | $a0 = address of string in memory |
| Read integer to $v0 | 5 | |

| | | |
|---|---|---|
| Read float to $f0 | 6 | |
| Read double to $f0 | 7 | |
| Read string | 8 | $a0 = memory address of string input buffer<br>$a1 = length of string buffer (n) |
| Dynamic memory allocation | 9 | $a0 = amount |
| Exit | 10 | |
| Print character | 11 | $a0 = character to be printed |
| Read character in $v0 | 12 | |

**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

        la $a0, buf   #Read string.
        li $a1, 11
        li $v0, 8
        syscall
```

## Using syscall to Stop a Program

**MAL Code:**

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

# Saving and Restoring Registers

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

```
        .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).