# Week 8 Review


## Warmup

- What are the following assembly language instructions doing?

`sub $t7, $t0, $t1` → Subtract register `$t1` from `$t0` and placing the result into `$t7`

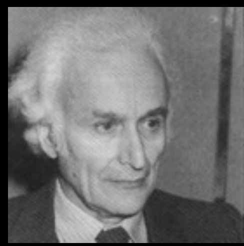
`andi $t7, $t0, 15` → Bitwise AND between register `$t0` and 15 (`1111`), with the result placed into register `$t7`

`sra $t2, $t1, 2` → Arithmetic shift of register `$t1` two bits to the right, with the result stored in `$t2`

What is the instruction type of sra? R-type!

# Writing Assembly

- Assembly is not sophisticated.
  - You have to tell it to do everything.
  - The difficulty comes from its simplicity.
- Making an assembly program:
  1. Split to very basic steps.
  2. Understand what variables you'll need and set aside registers.
  3. Add labels, branches as needed.
  4. Thank previous generations for compilers.



Corrado Böhm

Grace Hopper

# MIPS Register File Registers

| Number | Name | Use |
|---|---|---|
| 0 | `$0, $zero` | Always the constant zero |
| 1 | `$at` | reserved for assembler (pseudo instructions) |
| 2 – 3 | `$v0 - $v1` | function return values |
| 4 – 7 | `$a0 - $a3` | function arguments |
| 8 – 15 | `$t0 - $t7` | temporary variables |
| 16 – 23 | `$s0 - $s7` | saved temporaries |
| 24 – 25 | `$t8 - $t9` | temporary variables |
| 26 – 27 | `$k0 - $k1` | reserved for operating system kernel |
| 28 | `$gp` | global pointer to data segment |
| 29 | `$sp` | stack pointer to top of stack |
| 30 | `$fp` | frame pointer to function frame start |
| 31 | `$ra` | return address from function |

# Types of Asm Instructions

- Arithmetic `add, mult, …`
- Logical `and, or, …`
- Bit shifting `sll, sra, …`
- Data movement `mflo,mfhi, …`
- Branch `beq, bgtz, …`
- Jump `j, jr, …`
- Comparison `slt, sltu, …`
- Memory `lw, sw, …`

# Arithmetic instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **add** | 100000 | $d, $s, $t | $d = $s + $t |
| **addu** | 100001 | $d, $s, $t | $d = $s + $t |
| **addi** | 001000 | $t, $s, i | $t = $s + SE(i) |
| **addiu** | 001001 | $t, $s, i | $t = $s + SE(i) |
| **div** | 011010 | $s, $t | lo = $s / $t; hi = $s % $t |
| **divu** | 011011 | $s, $t | lo = $s / $t; hi = $s % $t |
| **mult** | 011000 | $s, $t | hi:lo = $s * $t |
| **multu** | 011001 | $s, $t | hi:lo = $s * $t |
| **sub** | 100010 | $d, $s, $t | $d = $s - $t |
| **subu** | 100011 | $d, $s, $t | $d = $s - $t |

**Notes: "hi" and "lo" refer to the HI and LO registers**
**"SE" = "sign extend".**

# Question #1

- Write a piece of assembly code to swap the values in `$t0` and `$t1`, using `$t2` as a temp value.

```
add $t2, $zero, $t0
add $t0, $zero, $t1
add $t1, $zero, $t2
```

# Logical instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **and** | 100100 | \$d, \$s, \$t | \$d = \$s & \$t |
| **andi** | 001100 | \$t, \$s, i | \$t = \$s & ZE(i) |
| **nor** | 100111 | \$d, \$s, \$t | \$d = ~(\$s \| \$t) |
| **or** | 100101 | \$d, \$s, \$t | \$d = \$s \| \$t |
| **ori** | 001101 | \$t, \$s, i | \$t = \$s \| ZE(i) |
| **xor** | 100110 | \$d, \$s, \$t | \$d = \$s ^ \$t |
| **xori** | 001110 | \$t, \$s, i | \$t = \$s ^ ZE(i) |

Note: ZE = zero extend (pad upper bits with 0 value).

# Shift instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **sll** | 000000 | $d, $t, a | $d = $t << a |
| **sllv** | 000100 | $d, $t, $s | $d = $t << $s |
| **sra** | 000011 | $d, $t, a | $d = $t >> a |
| **srav** | 000111 | $d, $t, $s | $d = $t >> $s |
| **srl** | 000010 | $d, $t, a | $d = $t >>> a |
| **srlv** | 000110 | $d, $t, $s | $d = $t >>> $s |

- Order is $d, $t, $s or $d, $t, a (not $d, $s, $t as before!)
- `srl` = "shift right logical"
- `sra` = "shift right arithmetic".
- The "`v`" denotes a variable number of bits, specified by `$s`.
- `a` is shift amount, and is stored in `shamt` when encoding the R-type machine code instructions.

# lui – load upper immediate

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **lui** | 001111 | $t, i | $t = i << 16 |

- Load 16-bit immediate into upper half of the register.
- The lower 16 bits of the register are set to zero.

`iiiiiiiiiiiiiiii0000000000000000`

# `li` pseudoinstruction

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **li** | N/A | \$t, i | \$t ← i |

- Load immediate into register.
- If immediate fits in 16-bit, uses `addiu`
- If immediate is 32-bit, uses `lui` followed by `ori`

```
li $s0,0x1234ABCD
```

→

```
lui $s0,0x1234
ori $s0,$s0,0xABCD
```

# Pseudoinstructions

- Move data from \$t4 to \$t5?
  - `move $t5,$t4` →

```
add $t5,$t4,$zero
```

- Multiply and store in \$s3?
  - `mul $s1, $t4, $t5` →

```
mult $t4,$t5
mflo $s1
```

- Branches (later today…)

# Question #2

- Translate this C-style code into 4 lines of MIPS assembly code:

```
int t1 = 10
int t2 = 3;
int t3 = t1 + 2*t2;
```

# Question #2

- Translate this C-style code into 4 lines of MIPS assembly code:

```
int t1 = 10
int t2 = 3;
int t3 = t1 + 2*t2;
```

- Solution:

```
li $t1, 10
li $t2, 3
sll $t2, $t2, 1
add $t3, $t2, $t1
```

# Question #2

- Translate this C-style code into 4 lines of MIPS assembly code:

```
int t1 = 10
int t2 = 3;
int t3 = t1 + 2*t2;
```

- Solution:

```
li $t1, 10
li $t2, 3
sll $t2, $t2, 1
add $t3, $t2, $t1
```

we are overwriting $t2!

# Question #2

- Translate this C-style code into 4 lines of MIPS assembly code:

```
int t1 = 10
int t2 = 3;
int t3 = t1 + 2*t2;
```

Use $t3 instead

- Solution:

```
li $t1, 10
li $t2, 3
sll $t3, $t2, 1
add $t3, $t3, $t1
```

# Formatting Assembly Code

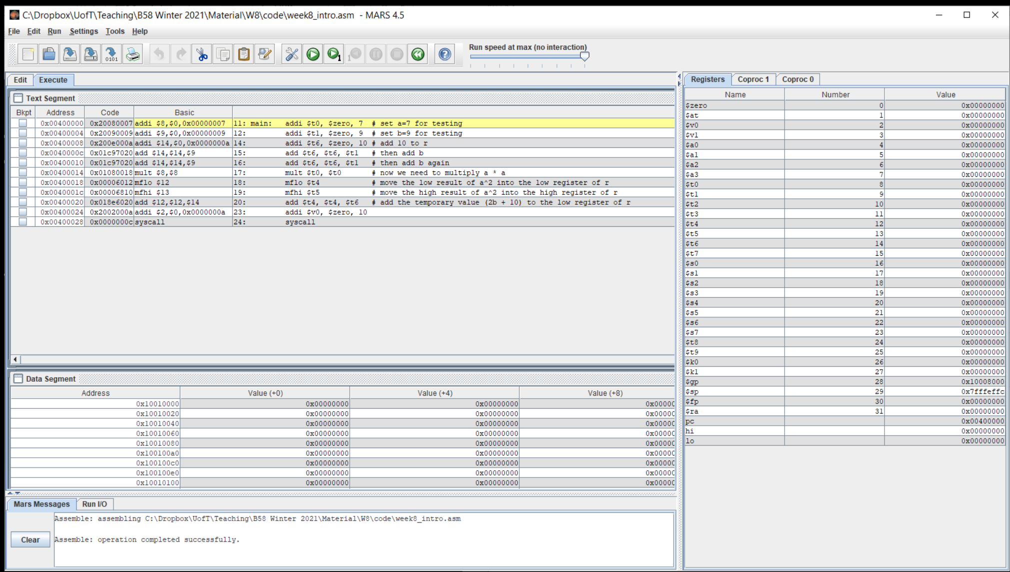
- Start file with `.text`
  - (we'll see other options later)
- Follow this with:
- `.globl main`
  - (Makes the main label visible to the OS)
- `main:`
  - (Tells OS which line of code should run first.)
- Write instructions
  - `label: <instr> <params>   # comments`
  - Labels and comments as needed
- Use `#` for comments. Comments are critical!
- At the end of the program, tell the OS to finish:
  ```
  li $v0, 10
  syscall
  ```

```
.text

.globl main
main:
      <code>

      li $v0, 10
      syscall
```

```
# Compute the following result: r = a^2 + 2b + 10
.text

.globl main
# $t0 will be a,   $t1 will be b,   $t5 will be r
# $t6 will be temp
main:
      addi $t0, $zero, 7   # set a=7 for testing
      addi $t1, $zero, 9   # set b=9 for testing

      addi $t6, $zero, 10  # add 10 to r
      add $t6, $t6, $t1    # then add b
      add $t6, $t6, $t1    # then add b again
      mult $t0, $t0        # multiply a * a
      mflo $t4             # move the low result of a^2
                           # into the register for r
      add $t4, $t4, $t6    # add the temporary value
                           # (2b + 10) to the result

      addi $v0, $zero, 10  # end program
      syscall
```

# MARS Simulator

- Use it to write and run MIPS assembly programs.




# Control Flow in Assembly

- Assembly is not sophisticated.
  - We have to tell it manually where to go and when.
- Labels indicate points in the program we might need to jumpy to.
- Branch instructions tell the CPU to go somewhere based on some condition.

  `beq $t0, $t5, label` → if \$to = \$t5, jump to `label`
- We can also jump unconditionally.

  `j label` → jump to `label` (always)

# Warmup

- What are the following assembly language instructions doing?

`bgtz $t2, TOP` → Jump to the line with label "TOP" if register `$t2` is greater than `0` (`$zero`) Comparison is signed!

`jalr $t0` → Store the current PC location into $ra (register $31) and jump to the location whose address is stored in register `$t0`

Practice, and learn the meaning behind the names of instructions

# Branch instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **beq** | 000100 | \$s, \$t, label | if (\$s == \$t) pc ← label |
| **bgtz** | 000111 | \$s, label | if (\$s > 0) pc ← label |
| **blez** | 000110 | \$s, label | if (\$s ≤ 0) pc ← label |
| **bne** | 000101 | \$s, \$t, label | if (\$s != \$t) pc ← label |

- These comparisons are signed.
- Branch operations are key to implementing if/else statements and loops.
- The labels are memory locations, assigned to each label at compile time.

# Jump instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **j** | 000010 | label | pc ← label |
| **jal** | 000011 | label | $ra = pc+4; pc ← label |
| **jalr** | 001001 | $s | $ra = pc+4; pc = $s |
| **jr** | 001000 | $s | pc = $s |

- `jal` = "jump and link".
  - Register $31 (aka $ra) stores the address that's used when returning from a subroutine.
- Note: `jr` and `jalr` are not j-type instructions.
  - We can tell because they have $s

# Comparison instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **slt** | 101010 | $d, $s, $t | $d = ($s < $t) |
| **sltu** | 101001 | $d, $s, $t | $d = ($s < $t) |
| **slti** | 001010 | $t, $s, i | $t = ($s < SE(i)) |
| **sltiu** | 001001 | $t, $s, i | $t = ($s < SE(i)) |

- "slt" = "Set Less Than"
- Comparison operation stores one (1) in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.
- Signed: `0x8000000` is less than all other numbers
- Unsigned: `0 - 0x7FFFFFFF` are less than `0x8000000`
  - Immediate is sign-extended even in `sltiu`

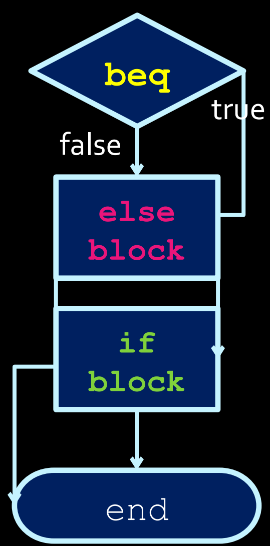
# Branch Pseudoinstructions

- Implemented using `slt` variants and branches.
- You are allowed to use them unless we say otherwise.

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| **blt** | N/A | \$s, \$t, label | if (\$s < \$t) pc ← label |
| **bltu** | N/A | \$s, \$t, label | if (\$s < \$t) pc ← label |
| **bgt** | N/A | \$s, \$t, label | if (\$s > \$t) pc ← label |
| **bgtu** | N/A | \$s, \$t, label | if (\$s > \$t) pc ← label |
| **ble** | N/A | \$s, \$t, label | if (\$s ≤ \$t) pc ← label |
| **bleu** | N/A | \$s, \$t, label | if (\$s ≤ \$t) pc ← label |
| **bge** | N/A | \$s, \$t, label | if (\$s ≥ \$t) pc ← label |
| **bgeu** | N/A | \$s, \$t, label | if (\$s ≥ \$t) pc ← label |

# If statements

```
if ( i == j )
   i = i+1;
else
   j = j-1;
j = j+i;
```




```
#  $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF
         addi $t2, $t2, -1
         j END
IF:      addi $t1, $t1, 1
END:     add $t2, $t2, $t1
```

# If/Else using beq

```
if ( i == j )
  i = i+1;
else
  j = j-1;
j = j+i;
```

```
#   $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF     # branch if ( i == j )
         addi $t2, $t2, -1     # j--
         j END                 # jump over IF
IF:      addi $t1, $t1, 1      # i++
END:     add $t2, $t2, $t1     # j += i
```

# If/Else using bne

```
if ( i == j )
  i = i+1;
else
  j = j-1;
j = j+i;
```

```
#   $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE   # branch if ( i != j )
         addi $t1, $t1, 1      # i++
         j END                 # jump over ELSE
ELSE:    addi $t2, $t2, -1     # j--
END:     add $t2, $t2, $t1     # j += i
```

# Multiple Conditions Inside If

```
if ( i == j && i == k )
    i++ ;   // if-body
else
    j-- ;   // else-body
j = i + k ;
```

- Multiple branches whose flow matches if logic.

```
#  $t1 = i, $t2 = j, $t3 = k
main:  bne $t1, $t2, ELSE    # cond1: branch if ( i != j )
       bne $t1, $t3, ELSE    # cond2: branch if ( i != k )
IF:    addi $t1, $t1, 1      # if (i==j|i==k) → i++
       j END                 # jump over else
ELSE:  addi $t2, $t2, -1     # else-body: j--
END:   add $t2, $t1, $t3     # j = i + k
```

# Question #3

- Write the following in assembly:

```
if ((a > b) and (c > b))
    b++;
else
    b-- ;
c = a + b;
```

# Loop example in MIPS

```
j = 0;
for ( i=0 ; i!=100 ; i++ ) {
      j = j + i;
 }
```

- This translates to:

```
#  $t0 = i, $t1 = j
main:    add $t0, $zero, $zero        # set i to 0
         add $t1, $zero, $zero        # set j to 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE:  addi $t0, $t0, 1             # i++
         j START
EXIT:
```

- `while` loops are the same, without the initialization and update sections.

# Question #4

- Given value X in $to and value Y in $t1, compute $X^Y$ and store the result in $t6
  - Assume no overflow, assume result fits in 32-bits.
  - Assume Y is larger than or equal to zero ($X^0$ = 1).
- Hints:
  - Suppose you know Y=3 always, and you need compute $X^3$, how would you do it?
  - You need a loop. How do you make that happen?
  - What is the stopping condition?
  - What needs to be done at the beginning?
  - What if Y is zero?

# Question #5

- Write the following in assembly:

```
j = 258;
for(i = 42 ; j > i ; i = i * 2)
     j++;
i = j;
```

# Question 6

- Fibonacci sequence:
  - Convert this into assembly?

```
int n = 10;
int f1 = 1, f2 = 1;
int temp;

while (n != 0) {
   temp = f1
   f1 = f1 + f2;
   f2 = temp;
   n = n - 1;
}
# result is f1
```

# Lab 4

- Install MARS
- Compute the number of solutions for the quadratic equation
  - Compute discriminant
  - Use multiple if/else to decide if 0, 1, or 2 solutions
- Compute Greatest Common Divisor using a variation of Euclid's algorithm
  - Basically a loop