# MIPS ASSEMBLER





## Week 14 Practical

# ASSIGNMENT

**Ethiopian Multiplication** allows you to multiply two numbers by using simple operations such as shift and add



- This method is sometimes referred to by other names





- Task
  - Build a procedure that takes integers a and b
  - The two numbers are multiplied and the result is returned
  - a and b are provided via memory
  - The result is printed out by the calling function

```
.data
res:        .word 0                              #memory holding result
num:        .word 10, 12                         #memory holding a, b
result_s:   .asciiz "The result is: "            #output preamble
error_s:    .asciiz "Multiplication Error"       #error message

.text
  j main #used to jump over the following function

#the multiplication function
#$a0 contains pointer to array (num in our case)
#$v0 is used to return result 0 or 1
mult:
  << LOAD a and b using address in $a0 >>
  << multiply a and b >>
  << store result in array using address in $a0 >>
  << return either 0 or 1 in $v0 depending on successful must >>

main:
  << put address of array num in $a0 >>
  << call mult function >>
  << print result of mult or error depending on $v0 >>
```

# ETHIOPIAN MULTIPLICATION

- Multiply **a** and **b** (for example, **a**=17 **b**=34)
- Create 2 columns
- Halve a until reaching 1; double b
- Add 2nd column if **a** is odd



```
17      34
8      (68) <-- even
4     (136) <-- even
2     (272) <-- even
1      544
      -----
       578
```

- This can be implemented using MIPS assembler ...

# OTHER OPTIONS

- Obviously we could simply use

```
addi $t0, $zero, 17 # a is 17 ($t0=a)
addi $t1, $zero, 34 # b is 34 ($t1=b)
mult $t1,$t0        # multiply
```



- However, assume we would not have a **`mult`** instruction
- We could just run a loop and add **`a`** times **`b`**

```
addi $t0, $zero, 17 # a is 17 ($t0=a)
addi $t1, $zero, 34 # b is 34 ($t1=b)
addi $t2, $zero, 0  # $t2 is loop counter
loop:
addi $t2,$t2,1      # add 1 to loop counter
add $s0,$s0,$t1     # add one b to result
bne $t2,$t0, loop   # add b a times
```

# OTHER OPTIONS

- Problem of loop with adding
  - It can take a lot of cycles to compute the result
  - It will take time to compute the result for large numbers



- How can we cut down time?
  - Use something like Ethiopian Multiplication
  - Requires less instructions to compute the result

- However
  - It depends on the numbers which method is faster!

# WHAT DO WE NEED?

- We need to load **a** and **b** from memory
- We need to store the result in memory
- We need to implement a procedure call
- We need to implement a loop
- We need to half and double numbers
- We need to test for a condition "even number"
- We need `syscall` to print the result

# WHAT DO WE NEED?

- **We need to load a and b from memory**
- **We need to store the result in memory**



- We need to implement a procedure call
- We need to implement a loop
- We need to half and double numbers
- We need to test for a condition "even number"
- We need `syscall` to print the result

# ACCESSING MEMORY

```
lw $s1,100($s2)
```

"load word"

destination register

source address of word



```
sw $s1,100($s2)
```

"store word"

source register

destination address

`lw`, `sw` are **I-type** Instructions

# WORD AND BYTE

"big endian"

most significant bit (MSB) of data word

least significant bit (LSB) of data word



1 word = 32 bits = 4 bytes

8 bits

address

| | | | | |
|---|---|---|---|---|
| 0x10010010 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 |
| 0x1001000C | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 |
| 0x10010008 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 |
| 0x10010004 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 |
| 0x10010000 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 |
| | 0x10010001 | 0x10010002 | 0x10010003 | |

address

# WHAT DO WE NEED?

- We need to load **a** and **b** from memory
- We need to store the result in memory



- **We need to implement a procedure call**



- We need to implement a loop
- We need to half and double numbers
- We need to test for a condition "even number"
- We need `syscall` to print the result

# MIPS REGISTERS FOR PROCEDURE CALLS

`$a0-$a3:`

"argument" registers in which to pass parameters



`$v0 and $v1:`



return value registers

`$ra:`

return address register

# MIPS INSTRUCTIONS FOR PROCEDURE CALLS

```
jal ProcedureAddress
```



"jump and link"



label to jump to



- `jal` stores the address of the next instruction in `$ra`
- ...and then jumps to `ProcedureAddress`
- to get back, we use "jump register"

```
jr $ra
```

"jump register"

# PRESERVING REGISTERS

- Sometimes a procedure needs to use more registers than just four arguments and two return values.
- Register content must be preserved during procedure call
- Moving the contents of registers to main memory is called **spilling registers**.
- Registers are stored to memory using a conceptual data structure known as a **stack**.
- The stack pointer register `$sp` points to the contents of the register most recently pushed onto the stack.

# WHAT DO WE NEED?

- We need to load **a** and **b** from memory
- We need to store the result in memory
- We need to implement a procedure call
- **We need to implement a loop**
- We need to half and double numbers
- We need to test for a condition "even number"
- We need `syscall` to print the result

# CONDITIONAL BRANCH

- Different conditions can be tested
  - branch on equal: **beq**
  - branch on not equal: **bne**
  - branch if zero: **beqz**



- Branch destination
  - given by a label
  - translated by assembly to a PC relative address

# BRANCH EXAMPLE
# IF-THEN-ELSE

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



```
        beq $s3,$s4,Then
        sub $s0,$s1,$s2
        j Exit
Then:   add $s0,$s1,$s2
Exit:
```

register mapping

```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```

# WHAT DO WE NEED?

- We need to load **a** and **b** from memory
- We need to store the result in memory
- We need to implement a procedure call
- We need to implement a loop
- **We need to half and double numbers**
- We need to test for a condition "even number"
- We need `syscall` to print the result

# HALF AND DOUBLE

- Doubling a number can be implemented by a shift left

```
sll $t1,$t1,1      # multiply value in $t1 by 2
```



- Example:



```
34:    0010 0010
68:    0100 0100
```



- Halving without remainder can be implemented by shift right

```
srl $t1,$t1,1      # devide value in $t1 by 2
```

- Example:

```
17:    0001 0001
8 :    0000 1000
```

# WHAT DO WE NEED?

- We need to load **a** and **b** from memory
- We need to store the result in memory



- We need to implement a procedure call
- We need to implement a loop
- We need to half and double numbers
- **We need to test for a condition "even number"**
- We need `syscall` to print the result

# ODD/EVEN

- Checking if a number is odd by looking at the last bit!

```
17:     0001 0001
8 :     0000 1000
```

1, odd number

0, even number



- A possible test using **and**



```
17:     0001 0001
1 :     0000 0001
        ---------
17 and 1    0000 0001
```



- Result of **and** is 0: even number
- Result of **and** is 1: odd number

# WHAT DO WE NEED?

- We need to load **a** and **b** from memory
- We need to store the result in memory
- We need to implement a procedure call
- We need to implement a loop
- We need to half and double numbers
- We need to test for a condition "even number"
- **We need syscall to print the result**

# SYSCALL EXECUTION

1. Load the service number in register `$v0`
2. Load argument values, if any, in `$a0`, `$a1`, `$a2`, or `$f12`
3. Issue the **SYSCALL** instruction
4. Retrieve return values, if any, from result registers



**Note:** MIPS register contents are not affected by a system call, except for result registers as specified.

# SYSCALL SERVICE NUMBERS

| Service | System call code | Arguments | Result |
|---|---|---|---|
| print_int | 1 | $a0=integer | |
| print_float | 2 | $f12=float | |
| print_double | 3 | $f12=double | |
| print_string | 4 | $a0=string | |
| read_int | 5 | | integer ($v0) |
| read_float | 6 | | float ($f0) |
| read_double | 7 | | double ($f0) |
| read_string | 8 | $a0=buffer, $a1=length | |
| sbrk | 9 | $a0=amount | |
| exit | 10 | | |

# SUMMARY

- Overview of the week 14 task



- Next
  - Some more examples