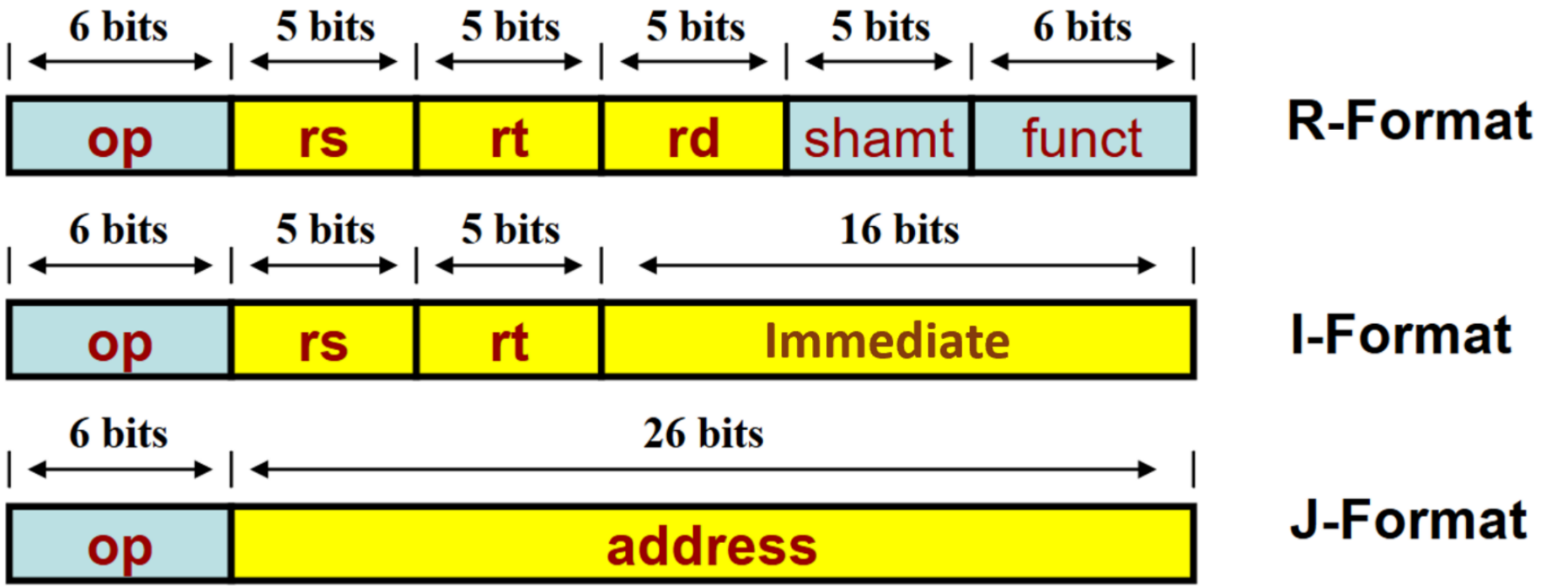
TUTORIAL 7 MIPS MACHINE CODE AND PROCEDURE

Overview

- **You will review the following in this tutorial:**
  - MIPS machine code
  - Sign extension and zero extension
- **You will practice to write simple MIPS procedures**

MIPS Instruction Format

- MIPS has three instruction formats. All uses 32 bits.



Warm up exercise

- Write down the corresponding MIPS machine code (in binary) of the instructions

a) add $s0, $s1, $t2

Solution:
000000 10001 01010 10000 00000 100000

b) lw $s0, 16($t0)

Solution:
100011 01000 10000 0000 0000 0001 0000

MIPS Assembly Examples: Symbolic

```
.text # text segment
.globl _main
_main:
# it's not a valid code sequence
# R-format
add $s2, $s1, $s0
sub $s2, $s1, $s0
or $s2, $s1, $s0
sll $s1, $s0, 2
# I-format
lw $s1, 16($s0)
lb $s1, 16($s0)
addi $s2, $s1, -1
beq $s1, $s0, exit
# J-format
j exit
exit:
jr $ra  # is it really J-type?
```

Corresponding MIPS Machine Code

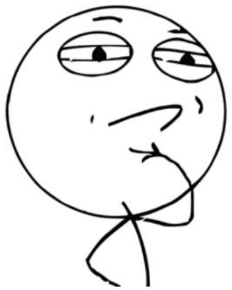
Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x02309020	add $18,$17,$16	7: add $s2, $s1, $s0
☐	0x00400004	0x02309022	sub $18,$17,$16	8: sub $s2, $s1, $s0
☐	0x00400008	0x02309025	or $18,$17,$16	9: or $s2, $s1, $s0
☐	0x0040000c	0x00108880	sll $17,$16,0x00000002	10: sll $s1, $s0, 2
☐	0x00400010	0x8e110010	lw $17,0x00000010($16)	12: lw $s1, 16($s0)
☐	0x00400014	0x82110010	lb $17,0x00000010($16)	13: lb $s1, 16($s0)
☐	0x00400018	0x2232ffff	addi $18,$17,0xffff...	14: addi $s2, $s1, -1
☐	0x0040001c	0x12300001	beq $17,$16,0x00000001	15: beq $s1, $s0, exit
☐	0x00400020	0x08100009	j 0x00400024	17: j exit
☐	0x00400024	0x03e00008	jr $31	19: jr $ra

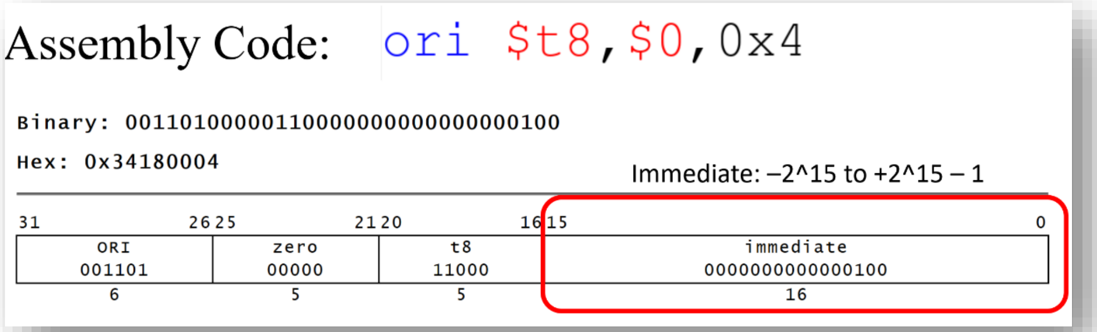
Zero Extension or Sign Extension

- **Extension is needed when there is a size mis-match**
  - Source and destination in assignment
  - Two operands of a calculation

Type	Example	Extension
Arithmetic Instructions	addi/addiu	Always sign-extend
Load/store Instructions	lb	Sign-extend
	lbu	Zero-extend
Logical instructions	ori, andi	Always zero-extend



Zero Extension



- **ori** is I format
- 0x4 is stored in the instruction with 16 bits 0000 0000 0000 0100
- It is to be bitwise ORed with the thirty-two bits of register zero,

  0000 0000 0000 0000 0000 0000 0000 0000
- This would not ordinarily be possible because the operands are different lengths. However, MIPS zero extends the sixteen-bit operand so the operands are the same length.
- Sometimes this is also called **padding with zeros**.

0000 0000 0000 0000 0000 0000 0000 0100 -- zero extended immediate operand
0000 0000 0000 0000 0000 0000 0000 0000 -- data in register $0
0000 0000 0000 0000 0000 0000 0000 0100 -- result, put in register $t8

Sign Extension for Signed Numbers

- We often need to represent a value given in a certain number of bits by using a larger number of bits.
- You can **repeat the sign bit** of the number as many times as it is needed to the left.
- **Sign extension preserves the original value!!**

- Examples:
  - Positive Number:

0111→ 0000 0000 0000 0111

  - Negative Number:

1010→ 1111 1111 1111 1010

Example: Sign/Zero Extension

Edit | Execute

proc_sum | procedure_print.asm | procedure_cuboid | machinecode.asm | extension.asm

```
.data
bytearray: .byte 0x7f 0xff
wordarray: .word 0xffffffff
.text # text segment
.globl _main
_main:
# it's not a valid code sequence
ori $s0, $0, 0xFF41 # zero extension
addi $s1, $0, -1 # sign extension

la $s2, bytearray
lb $s3, 0($s2) # sign extension
lbu $s4, 0($s2) #zero extension
lb $s5, 1($s2) # sign extension
lbu $s6, 1($s2) # zero extension

la $s7, wordarray
li $t0, 0x1234ABCD # pseudo instruction
sb $t0, 0($s7) # store the rightmost byte to memory
```

Exercise 1

- Write down the shortest sequence (any one) of MIPS instructions for the following C++ code, assuming variable a and b are stored in \$s0 and \$s1 respectively. You can use some registers for storing temporary values.

```
b = a + 0x37cf0010;
```

Solution:

```
lui $t0, 0x37cf
addi $t0, $t0, 0x0010
add $s1, $s0, $t0
```

Exercise 2

- Write down the shortest sequence (any one) of MIPS instructions for the following C++ code, assuming variable a and b are stored in $s0 and $s1 respectively. You can use some registers for storing temporary values.

```
b = a + 0x37cff346;
```

Solution:

```
lui $t0, 0x37cf
#addi doesn't work here because of sign-extension
ori $t0, $t0, 0xf346
add $s1, $s0, $t0
```

Register Convention for Procedures

- **General purpose registers for procedure calling:**
  - `$a0 – $a3:` arguments (reg's 4 – 7)
  - `$v0, $v1:` result values (reg's 2 and 3)
  - `$t0 – $t9:` temporaries
    - Can be overwritten by callee
  - `$s0 – $s7:` saved
    - Must be saved/restored by callee
  - `$gp:` global pointer for static data (reg 28)
  - `$sp:` stack pointer (reg 29)
  - `$fp:` frame pointer (reg 30)
  - `$ra:` return address (reg 31)

- **Program counter (PC) or instruction address register:**
  - Register that holds address of the current instruction being executed. It is updated after executing the current instruction.
  - PC = PC + 4 *or* PC = branch target address

Caller and Callee Coordination

- **The calling program (caller)**
  - Passing parameters:
    - Puts the parameter values in `$a0` - `$a3`
    - Invokes `jal X` to jump to procedure X
- **Procedure X (callee)**
  - Performs the calculations
  - To return the results, place the results in `$v0` - `$v1`
  - Returns control to the caller using `jr $ra`
  - Caller picks up the result from `$v0` - `$v1`

Example 1: Print

Edit | Execute

proc_sum | procedure_print.asm | procedure_cuboid | machinecode.asm | extension.asm

```
#procedure print
.data #data segment
message1: .asciiz "\nHello World!\n"
message2: .asciiz "\nPrinted by procedure.\n"

.text #text segment
.globl _main
_main: # caller
        la $a0, message1 # caller prepares arguments in $a
        jal print # print(message1)
        la $a0, message2
        jal print # print(message2)
        li $v0, 10
        syscall
print: # callee void print(char* str)
        li $v0, 4
        syscall
        jr $ra # return

```

Exercise 3A

- Translate the following C++ function into a MIPS function, using the registers `$a0` and `$a1` for its parameters and the register `$v0` for its return value.

```
int equal (int p1, int p2) {
    if (p1 == p2)
        return 1;
    return 0;
```

```
equal:
beq $a0, $a1, true
addi $v0, $zero, 0
jr $ra
true: addi $v0, $zero, 1
jr $ra
```

Exercise 3B

- Write down the MIPS code segment that make the following call to the C++ function in the previous exercise, assuming the variable `b` is stored in the register `$s0`.

```
int b = equal (3, 4);
```

Solution:

```
addi $a0, $zero, 3
addi $a1, $zero, 4
jal equal
addi $s0, $v0, 0
```

Example 2: Integer Comparison

Edit | Execute

proc_sum | procedure_print.asm | procedure_cuboid | machinecode.asm | extension.asm | procedure_cmp.asm

```
#procedure comparison
.data #data segment
equal_msg: .asciiz "\nTwo numbers are equal!\n"
not_equal_msg: .asciiz "\nTwo numbers are not equal.\n"
.text #text segment
.globl _main
_main: # caller
        addi $a0, $0, 11 # caller prepares arguments in $a
        addi $a1, $0, 11
        jal cmp # cmp($a0, $a1)
        li $v0, 10
        syscall #exit
cmp:    # callee void cmp(int a, int b)
        bne $a0, $a1, not_equal
        la $a0, equal_msg # no need to preserve $a registers
        j cmp_exit
not_equal:
        la $a0, not_equal_msg
cmp_exit:
        li $v0, 4 # print msg
        syscall
        jr $ra # return
```

Example 3: Multiplication

Edit | Execute

procedure_mult.asm | procedure_sum.asm | procedure_print.asm | procedure_cuboid.asm | procedure_cmp.asm

```
#procedure multiplication
.text #text segment
.globl _main
_main: # caller
        addi $a0, $0, 2 # calculate 2x6
        addi $a1, $0, 6 # caller prepares arguments in $a
        jal mult_with_add # mult_with_add($a0, $a1)
        add $s0, $0, $v0 # grab return value in $v
        li $v0, 10 # exit main
        syscall
mult_with_add: # callee int mult_with_add(int a, int b)
        add $v0, $0, $0 # init $v0=0
loop:
        beq $a1, $0, loop_done
        add $v0, $v0, $a0 # add $a0 for $a1 times
        addi $a1, $a1, -1 # $a1--
        j loop
loop_done:
        jr $ra # $v0 = $a0x$a1, return back to caller
```

Example 4: Integer Cuboid

Edit | Execute

proc_sum | procedure_print.asm | procedure_cuboid* | procedure_cmp.asm | procedure_mult.asm

```
# procedure cuboid
# use procedure mult_with_add
.data
sides: .word 2 3 4 # sides of the cuboid
result: .word 0 # volume of the cuboid
.text #text segment
.globl _main
_main: # caller
        la $s0, sides
        la $s1, result
        lw $a0, 0($s0) # side a
        lw $a1, 4($s0) # side b
        jal mult_with_add # mult_with_add($a0, $a1)
        add $a0, $v0, $0 # $a0 = a x b
        lw $a1, 8($s0)
        jal mult_with_add # $v0 = a x b x c
        sw $v0, 0($s1)
        li $v0, 10 # exit main
        syscall
```

Example 5: Array Sum

Edit | Execute

proc_sum | procedure_print.asm | procedure_cuboid | procedure_cmp.asm | procedure_mult.asm

```
.data
arr: .word 1 -2 3 -4 5 -6 7 -8
msg1: .asciiz "\n The sum of positive values: "
msg2: .asciiz "\n The sum of negative values: "
.text #text segment
.globl _main
_main: # caller
        la $a0, arr # starting addr of array
        addi $a1, $0, 8 # array size
        jal sum # call sum(arr, size), it returns two values in $v
        add $s0, $v0, $0 # $s0 positive sum
        add $s1, $v1, $0 # $s1 negative sum
        la $a0, msg1
        li $v0, 4
        syscall
        add $a0, $s0, $0 # print positive sum
        li $v0, 1
        syscall
        la $a0, msg2
        li $v0, 4
        syscall
        add $a0, $s1, $0 # print negative sum
        li $v0, 1
        syscall
        li $v0, 10 # exit
        syscall
```

Edit | Execute

proc_sum | procedure_print.asm | procedure_cuboid | procedure_cmp.asm | procedure_mult.asm

```
sum: # callee
        add $v0, $0, $0 # positive sum
        add $v1, $0, $0 # negative sum
        add $t0, $0, $0 # loop iterator
loop:
        slt $t1, $t0, $a1
        beq $t1, $0, loop_done # while (i < size)
        sll $t2, $t0, 2 # $t2 = 4 x i
        add $t2, $t2, $a0 # $t2 = addr of arr[i]
        lw $t3, 0($t2) # $t3 = arr[i]
        bltz $t3, negative # arr[i]<=0?
        add $v0, $v0, $t3 # $v0 += positive element
        j increment
negative:
        add $v1, $v1, $t3 # $v1 += negative element
increment:
        addi $t0, $t0, 1 # i++
        j loop
loop_done:
        jr $ra
```

Extra Exercise

- The following C++ function takes as inputs the base address of an int array `A` and returns the minimum value in `A`. Use the registers `$a0` and `$a1` as arguments to the function, `$v0` as return value, `$ra` as function return address. Translate the C++ function into a MIPS function.

```
int minArray (int A[], int arraySize){
    int min = A[0];
    int i = 1;
    while (i < arraySize) {
        if (min > A[i])
            min = A[i];
        i++;
    }
return min;
}
```

```
minArray:
lw $v0, 0($a0)
addi $t1, $zero, 1
loop:   slt $t2, $t1, $a1
        beq $t2, $zero, loopend
        sll $t3, $t1, 2
        add $t4, $a0, $t3
        lw $t5, 0($t4)
        slt $t6, $t5, $v0
        beq $t6, $zero, endif
        addi $v0, $t5, 0
endif: addi $t1, $t1, 1
        j loop
loopend: jr $ra
```