Computer Organization/ Architecture (COMP2825)

Instructor: Maryam Tanha

Fall 2021

Learning Outcomes of This Lecture

- **By the end of this lecture you will be able to**
  - Explain what are the instructions for making decisions in MIPS and apply them.
  - Explain what happens when we a have a procedure call.

Agenda

- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware

Agenda

- **Instructions for Making Decisions**
- Supporting Procedures in Computer Hardware

Instructions for Making Decisions

- **Conditional Branch Instruction**
  - branch to a labeled instruction if a condition is true; otherwise, continue sequentially
  - used in if statements and loops
  - beq register1, register2, L1
    - branch if equal
    - if (register1 == register2) branch to instruction labeled L1
  - bne register1, register2, L1
    - branch if not equal
    - if (register1 != register2) branch to instruction labeled L1

Instructions for Making Decisions - Continued

- **Unconditional Branch Instruction**
  - j L1
    - ✓jump to instruction labeled L1
    - ✓used in if statements and loops

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Variable	Register
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Answer (Assembly code):

```
bne $s3,$s4,Else                    # go to Else if i ≠ j
```

Variable	Register
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Answer (Assembly code):

```
bne $s3,$s4,Else        # go to Else if i ≠ j

add $s0, $s1, $s2       # f = g + h (skipped if i ≠ j)
```

Variable	Register
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Answer (Assembly code):

```
bne $s3,$s4,Else        # go to Else if i ≠ j
add $s0, $s1, $s2       # f = g + h (skipped if i ≠ j)
j Exit                  # go to Exit
```

Variable	Register
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Answer (Assembly code):

```
bne $s3,$s4,Else        # go to Else if i ≠ j
add $s0, $s1, $s2       # f = g + h (skipped if i ≠ j)
j Exit                  # go to Exit
Else: sub $s0, $s1, $s2 # f = g – h (skipped if i = j)
```

Variable	Register
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Instructions for Making Decisions – Example (if statement)

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

If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

Answer (Assembly code):

```
bne $s3,$s4,Else        # go to Else if i ≠ j
add $s0, $s1, $s2       # f = g + h (skipped if i ≠ j)
j Exit                  # go to Exit
Else: sub $s0, $s1, $s2 # f = g – h (skipped if i = j)
Exit:
```

Variable	Register
f	$s0
g	$s1
h	$s2
i	$s3
j	$s4

Instructions for Making Decisions – Continued

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Instructions for Making Decisions – Example (loop)

```
while (A[i] == k)
    i += 1;
```

Assume that *i* and *k* correspond to registers $s3 and $s5 and the base of the array *A* is in $s6. What is the MIPS assembly code corresponding to this C segment?

Instructions for Making Decisions – Example (loop)

```
while (A[i] == k)
    i += 1;
```

Assume that *i* and *k* correspond to registers $s3 and $s5 and the base of the array *A* is in $s6. What is the MIPS assembly code corresponding to this C segment?

Variable	Register
i	$s3
k	$s5

Instructions for Making Decisions – Example (loop)

```
while (A[i] == k)
    i += 1;
```

Answer:

Get the address of A[i]:

```
Loop: sll $t1, $s3, 2      # Temp reg $t1 = i * 4
add $t1, $t1, $s6          # $t1 = address of A[i]
```

Variable	Register
i	$s3
k	$s5

Instructions for Making Decisions – Example (loop)

```
while (A[i] == k)
    i += 1;
```

Answer:

Get the address of A[i]:

```
Loop: sll $t1, $s3, 2      # Temp reg $t1 = i * 4
add $t1, $t1, $s6          # $t1 = address of A[i]
```

load A[i] into a temporary register:

```
lw $t0, 0($t1)             # Temp reg $t0 = A[i]
```

Variable	Register
i	$s3
k	$s5

Instructions for Making Decisions – Example (loop)

```
while (A[i] == k)
    i += 1;
```

Answer:

Get the address of A[i]:

```
Loop: sll $t1, $s3, 2      # Temp reg $t1 = i * 4
add $t1, $t1, $s6          # $t1 = address of A[i]
```

load A[i] into a temporary register:

```
lw $t0, 0($t1)             # Temp reg $t0 = A[i]
```

loop test, exiting if A[i] ≠ k:

```
bne $t0, $s5, Exit         # go to Exit if A[i] ≠ k
```

Variable	Register
i	$s3
k	$s5

Instructions for Making Decisions – Example (loop)

```
while (A[i] == k)
    i += 1;
```

Answer – continued:

Add 1 to i:

```
addi $s3, $s3, 1        # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. Add the Exit label after it:

```
j Loop                          # go to Loop

Exit:
```

Variable	Register
i	$s3
k	$s5

Instructions for Making Decisions – Example (loop)

C Code:

```
while (A[i] == k)
      i += 1;
```

MIPS assembly code:

```
Loop: sll $t1, $s3, 2      # Temp reg $t1 = i * 4
add $t1, $t1, $s6          # $t1 = address of A[i]
lw $t0, 0($t1)             # Temp reg $t0 = A[i]
bne $t0, $s5, Exit         # go to Exit if A[i] ≠ k
addi $s3, $s3, 1           # i = i + 1
 j Loop                            # go to Loop
Exit:
```

Other Conditional Operations

- **Check if a variable is less than another variable**
  - set result to 1 if a condition is true; otherwise, set to 0
  - **comparing signed integers**
    - slt (set on less than) → **R-type** slt $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4
    - slti (set on less than immediate) → **I-type** slti $t0, $s2, 10 # $t0 = 1 if $s2 < 10
  - **comparing unsigned integers**
    - sltu (set on less than unsigned) → **R-type**
    - sltiu (set on less than immediate unsigned) → **I-type**

Other Conditional Operations

- **Check if a variable is less than another variable**
  - set result to 1 if a condition is true; otherwise, set to 0
  - **comparing signed integers**
    - ✓slt (set on less than) → **R-type** slt $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4
    - ✓slti (set on less than immediate) → **I-type** slti $t0, $s2, 10 # $t0 = 1 if $s2 < 10
  - **comparing unsigned integers**
    - ✓sltu (set on less than unsigned) → **R-type**
    - ✓sltiu (set on less than immediate unsigned) → **I-type**

slt/sltu, slti/sltiu, beq, bne, and the fixed value of 0 (register $zero) are used by MIPS compiler to produce all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

Agenda

- Instructions for Making Decisions
- **Supporting Procedures in Computer Hardware**

Supporting Procedures in Computer Hardware

- **Procedure:** a stored subroutine that performs a specific task based on the parameters with which it is provided.
- Procedure is one way to implement abstraction in software.
- **Steps to execute a procedure**:
  1. Place parameters in registers.
  2. Transfer control to procedure.
  3. Acquire storage resources for the procedure.
  4. Perform the desired task.
  5. Place result in register for caller.
  6. Return to place of call (point of origin).

Supporting Procedures in Computer Hardware

- **Registers used in procedure calling:**
  - **$a0 – $a3**: four argument registers to pass parameters (reg 4 – 7)
  - **$v0, $v1**: two value registers to return values (reg 2 and 3)
  - **$ra**: return address register to return to the point of origin (reg 31)
  - **$t0 – $t9**: temporary registers
    - ✓can be overwritten by callee
  - **$s0 – $s7**: saved (reg 16 - 23)
    - ✓must be saved/restored by callee
  - **PC**: program counter, a register to hold the address of the current instruction being executed (also called instruction address register)
  - **$gp**: global pointer for static data (reg 28)
  - **$sp**: stack pointer (reg 29)
  - **$fp**: frame pointer (reg 30)

Supporting Procedures in Computer Hardware

- **Procedure call instructions**
  - **jal ProcedureAddress**
    - ✓jal (jump-and-link instruction)
    - ✓jumps to an address and simultaneously saves the address of the following instruction in register $ra (PC + 4 is stored in register $ra)
  - **jr $ra**
    - ✓an unconditional jump to the address specified in register $ra (**return address**)

Supporting Procedures in Computer Hardware - Continued

The calling program, or **caller**, puts the parameter values in $a0–$a3 and uses jal X to jump to procedure X (sometimes named the **callee**). The callee then performs the calculations, places the results in $v0 and $v1, and returns control to the caller using jr $ra.

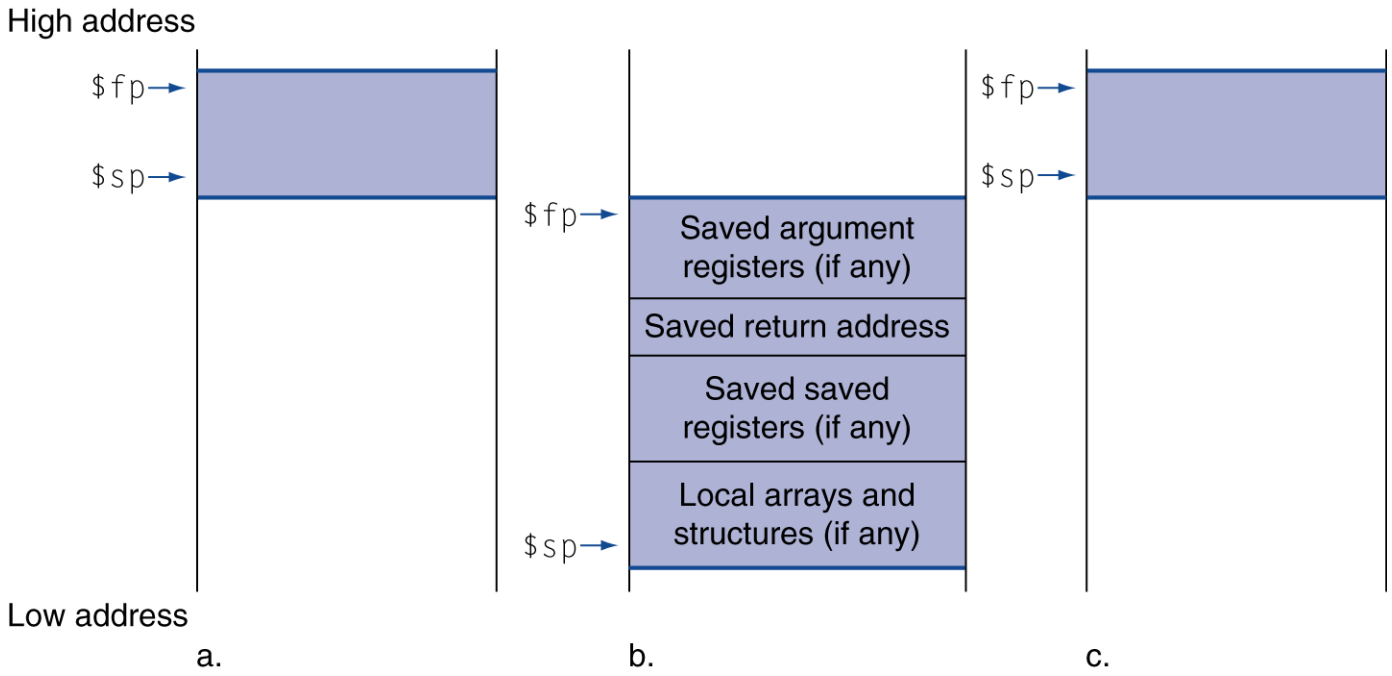
Supporting Procedures in Computer Hardware – Continued

What if there are more than four parameters (input arguments)?

Supporting Procedures in Computer Hardware – Continued

- When compiler needs more registers for a procedure than the four argument and two return value registers:
  - **it spills registers to the memory**
    - after the procedure finishes the task, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked.
  - stack data structure (last-in-first-out) is used for spilling registers.
    - stack pointer: register $sp that holds a value denoting the most recently allocated address in a stack that shows where the next procedure should place the registers to be spilled or where old register values are found.
    - push: add element to stack
    - pop: remove element from stack
    - **stacks "grow" from higher addresses to lower addresses** (i.e., you push values onto the stack by subtracting from the stack pointer)

Stack Allocation (a) before (b) during (c) after the Procedure Call



- Procedure frame (activation record): the segment of the stack containing a procedure's saved registers and local variables.
  - **$fp: points to the first word of the procedure frame**
- Stack is **also** used to store variables that are local to the procedure (callee) but do not fit in registers, such as local arrays or structures.

Supporting Procedures in Computer Hardware – Example

Consider the following C code:

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Assume f is in $s0, what is the compiled MIPS assembly code?

Supporting Procedures in Computer Hardware – Example

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Answer:

- parameters g, h, i, j correspond to argument registers $a0, $a1, $a2, $a3
- f corresponds to $s0 (hence, need to save $s0 on stack)
- Two temporary registers needed for the assignment and must be saved on stack.
- result in $v0

Supporting Procedures in Computer Hardware – Example

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Answer:

```
proc_example:
addi $sp, $sp, –12      # adjust stack to make room for 3 items
sw $t1, 8($sp)          # save register $t1 for use afterwards
sw $t0, 4($sp)          # save register $t0 for use afterwards
sw $s0, 0($sp)          # save register $s0 for use afterwards
```

Saving two temporary registers and $s0 on stack.

Supporting Procedures in Computer Hardware – Example

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Answer:

```
proc_example:
addi $sp, $sp, –12      # adjust stack to make room for 3 items
sw $t1, 8($sp)          # save register $t1 for use afterwards
sw $t0, 4($sp)          # save register $t0 for use afterwards
sw $s0, 0($sp)          # save register $s0 for use afterwards
```

Saving two temporary registers and $s0 on stack.

```
add $t0,$a0,$a1         # register $t0 contains g + h
add $t1,$a2,$a3         # register $t1 contains i + j
sub $s0,$t0,$t1         # f = $t0 – $t1, which is (g + h)–(i + j)
```

body of the procedure

Supporting Procedures in Computer Hardware – Example

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Answer – continued:

```
add $v0,$s0,$zero      # returns f ($v0 = $s0 + 0)
```

Supporting Procedures in Computer Hardware – Example

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Answer – continued:

```
add $v0,$s0,$zero      # returns f ($v0 = $s0 + 0)
lw $s0, 0($sp)         # restore register $s0 for caller
lw $t0, 4($sp)         # restore register $t0 for caller
lw $t1, 8($sp)         # restore register $t1 for caller
addi $sp,$sp,12         # adjust stack to delete three items
```

Restoring the old values of registers we saved by popping them from the stack.

Supporting Procedures in Computer Hardware – Example

```
int proc_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

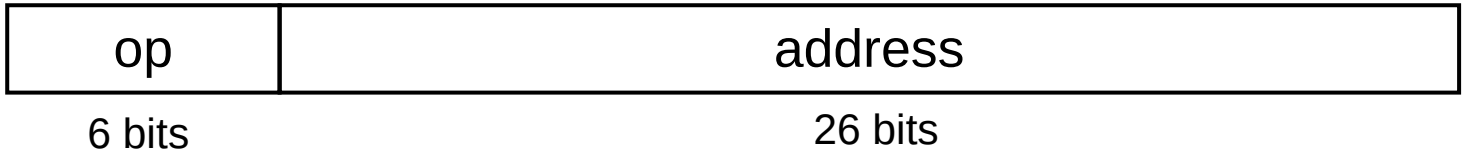
Answer – continued:

```
add $v0,$s0,$zero      # returns f ($v0 = $s0 + 0)
lw $s0, 0($sp)         # restore register $s0 for caller
lw $t0, 4($sp)         # restore register $t0 for caller
lw $t1, 8($sp)         # restore register $t1 for caller
addi $sp,$sp,12         # adjust stack to delete three items
jr $ra                 # jump back to calling routine
```

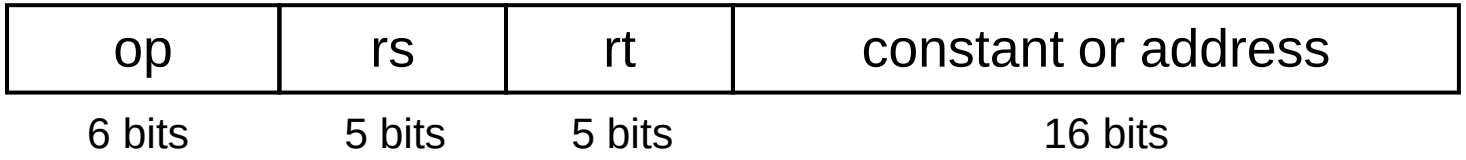
Restoring the old values of registers we saved by popping them from the stack.

Branches and Jumps Instruction Formats

- **MIPS jump instructions (e.g., j, jal)**
  - **J-type instruction format**



  - **note that jr (jump register) has R-type instruction format**
- **MIPS conditional branch instructions (e.g., beq, bne)**
  - **I-type instruction format**
  - must specify two operands in addition to the branch address



Addressing in Branches and Jumps

- **PC-relative addressing**
  - target address = PC + offset × 4
  - PC already incremented by 4 and it points to the next instruction

MIPS uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch (an example of making the common case fast).

Addressing in Branches and Jumps

- **PC-relative addressing**
  - target address = PC + offset × 4
  - PC already incremented by 4 and it points to the next instruction

MIPS uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch (an example of making the common case fast).

jump-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

References

[1] David A. Patterson and John L. Hennessy, Computer Organization and Design:

The Hardware/Software Interface, 6th Ed, 2020, Morgan Kaufmann Publishers, Inc.