Department of Electrical Engineering, Feng-Chia University

# Chapter 2
# Instructions: Language of the Computer
# (Part 2)

王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw



Department of Electrical Engineering, Feng-Chia University

## Outline

2.7 Instructions for Making Decisions
2.8 Supporting Procedures in Computer Hardware
2.9 Communicating with People
2.10 MIPS Addressing for 32-Bit Immediates and Addresses

Text Book : P105

# Change of the control flow

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

```
beq $t0, $t1, Label        # branch if equal
bne $t0, $t1, Label        # branch if not equal
```

- Example: `if (i==j) h = i + j;`

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:      ...
```

- MIPS unconditional branch instructions:

```
j  label
```



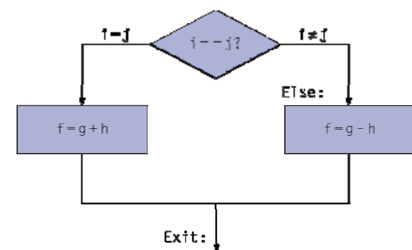


Text Book : P106

# Example

- Compiling *if-then-else* into Conditional Branches
  - C Code:

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




  - MIPS Code:

```
        bne     $s3, $s4, Else
        add     $s0, $s1, $s2
        j       Exit
Else:   sub     $s0, $s1, $s2
Exit:   …
```

Assembler calculates addresses

| Var. | Reg. |
|---|---|
| f | $s0 |
| g | $s1 |
| h | $s2 |
| i | $s3 |
| j | $s4 |

Text Book : P107 ~ P108

# Example

- Compiling a *while* Loop in C

| Var. | Reg. |
|---|---|
| i | $s3 |
| k | $s5 |
| $save_{base}$ | $s6 |

  - C Code:

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

  - MIPS Code:

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





# So far:

- Instruction / Meaning

```
add  $s1,$s2,$s3      $s1 = $s2 + $s3
sub  $s1,$s2,$s3      $s1 = $s2 – $s3
lw   $s1,100($s2)     $s1 = Memory[$s2+100]
sw   $s1,100($s2)     Memory[$s2+100] = $s1
bne  $s4,$s5,Label    Next instr. is at Label if $s4 ≠ $s5
beq  $s4,$s5,Label    Next instr. is at Label if $s4 = $s5
j    Label            Next instr. is at Label
```

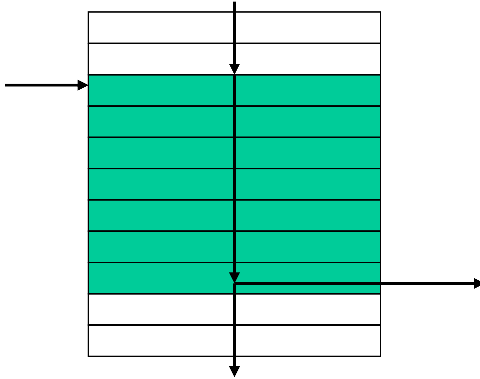
- Formats:

| | | | | | | |
|---|---|---|---|---|---|---|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

Text Book : P108

# Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks





Text Book : P109

# Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0
```

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

*Text Book : P109 ~ P110*

# Signed vs. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example

$s0 = 1111 1111 1111 1111 1111 1111 1111 1111

$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

```
slt  $t0, $s0, $s1  # signed
```

- –1 < +1 ⇒ $t0 = 1

```
sltu $t0, $s0, $s1  # unsigned
```

- +4,294,967,295 > +1 ⇒ $t0 = 0





# Conditional Branch Options

- **Condition Code**
  - Tests special bits set by ALU operations, possibly under program control.
  - Examples: 80x86, ARM, PowerPC …
- **Condition Register**
  - Tests arbitrary register with the result of a comparison.
  - Examples: MIPS, Alpha …
- **Compare and Branch**
  - Compare is part of the branch. Often compare is limited to subset.
  - Examples: PA-RISC, VAX

| HLL | Condition Code | Condition Register | Compare & Branch |
|---|---|---|---|
| if ( a < b )<br>*Statement 1 ;*<br>else<br>*Statement 2 ;* | CMP Ra, Rb<br>J.NEG Label<br>*Statement 2*<br>J Exit<br>Label: *Statement 1*<br>Exit: ... | S.LT Rt, Ra, Rb<br>J.C Rt, Label<br>*Statement 2*<br>J Exit<br>Label: *Statement 1*<br>Exit: ... | J.LT Ra, Rb, Label<br>*Statement 2*<br>J Exit<br>Label: *Statement 1*<br>Exit: ... |

Department of Electrical Engineering, Feng-Chia University

## Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≧, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise



Department of Electrical Engineering, Feng-Chia University

## Outline

2.7 Instructions for Making Decisions

2.8 Supporting Procedures in Computer Hardware

2.9 Communicating with People

2.10 MIPS Addressing for 32-Bit Immediates and Addresses

Text Book : P112

# Procedure Call

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.





Text Book : P113

# Policy of Use Conventions

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| $zero | 0 | the constant value 0 | – |
| $at | 1 | reserved for assembler | – |
| $v0-$v1 | 2-3 | values for results and expression evaluation | no |
| $a0-$a3 | 4-7 | arguments | no |
| $t0-$t7 | 8-15 | temporaries | no |
| $s0-$s7 | 16-23 | saved temporaries | yes |
| $t8-$t9 | 24-25 | more temporaries | no |
| $k0-$k1 | 26-27 | reserved for OS kernel | – |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return address | yes |

Department of Electrical Engineering, Feng-Chia University

*Text Book : P113*

# Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Address of following instruction put in $ra
  - Jumps to target address

- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements



Department of Electrical Engineering, Feng-Chia University

*Text Book : P114 ~ P115*

# Compiling a Procedure Call

- C Code

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

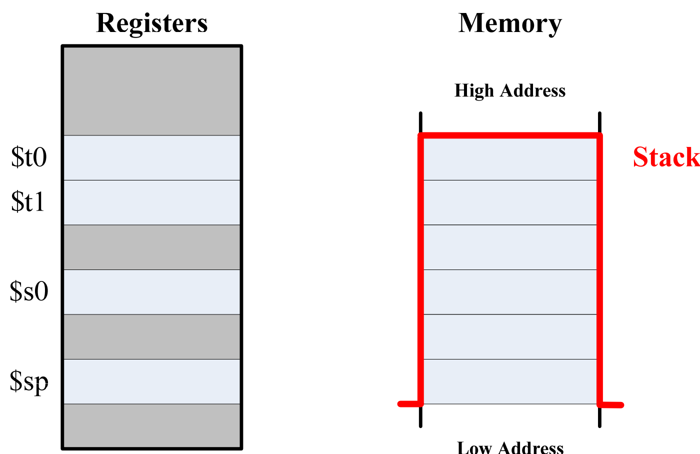
| Var. | Reg. |
|---|---|
| g | $a0 |
| h | $a1 |
| i | $a2 |
| j | $a3 |
| f | $s0 |

- MIPS Code

```
addi    $sp, $sp, -12
sw      $t1, 8($sp)
sw      $t0, 4($sp)
sw      $s0, 0($sp)

add     $t0, $a0, $a1
add     $t1, $a2, $a3
sub     $s0, $t0, $t1
add     $v0, $s0, $zero

lw      $s0, 0($sp)
lw      $t0, 4($sp)
lw      $t1, 8($sp)
addi    $sp, $sp, 12
jr      $ra
```



*Text Book : P115, P118*

# What is and what is not preserved across a procedure call

| Preserved | Not preserved |
|---|---|
| Saved registers: $s0 ~ $s7 | Temporary register: $t0 ~ $t9 |
| Stack pointer register: $sp | Argument register: $a0 ~ $a3 |
| Return address register: $ra | Return value register: $v0 ~ $v1 |
| Stack above the stack pointer | Stack below the stack pointer |

- Caller saving
  - Caller saves the registers that are preserved.
- Callee saving
  - Called procedure saves the registers that are preserved.





*Text Book : P116 ~ P117*

# Non-Leaf Procedure Example

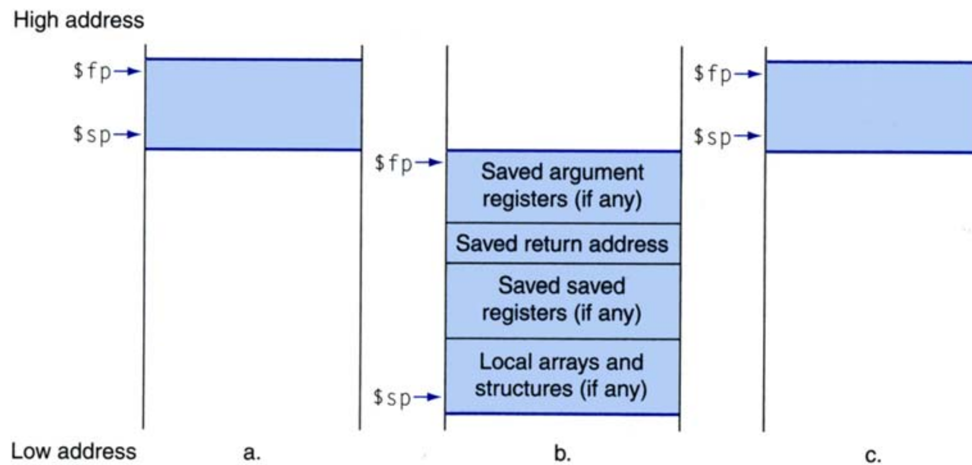
- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call
- Recursive call
  - Argument n in $a0
  - Result in $v0

```
int  fact ( int n )
{
    if ( n<1)
          return (1);
    else
          return ( n * fact( n-1 ) );
}
```

*Text Book : P119*

# Stack Allocation

- Stack can stores variables (large object) that are local to the procedure.
- $fp is fixed at a stable location so it offers good reference point for local memory references.



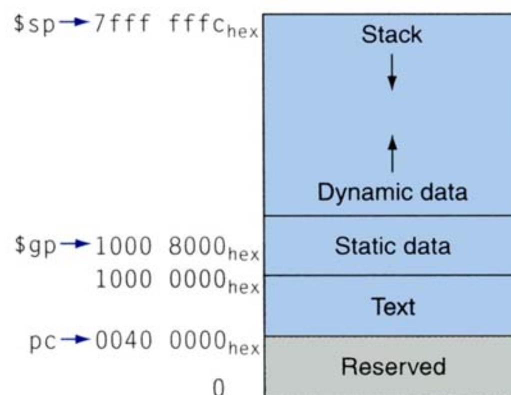





*Text Book : P120*

# MIPS memory allocation

- Text segment : the home of the MIPS machine code
- Static data segment : constants and other static variables
- Dynamic data (Heap) segment : data structures like linked lists tend to grow and shrink during their lifetimes.
  - E.g., malloc in C, new in Java
- Stack segment : starts in the high end of memory and grows down



Department of Electrical Engineering, Feng-Chia University

# Outline


2.7 Instructions for Making Decisions

2.8 Supporting Procedures in Computer Hardware

2.9 Communicating with People

2.10 MIPS Addressing for 32-Bit Immediates and Addresses




Department of Electrical Engineering, Feng-Chia University

*Text Book : P123*

# Communicating with People

- Load byte and store byte ( ASCII characters )

```
lb    $t0, 0($sp)      # Read byte from source
sb    $t0, 0($sp)      # Write byte to destination
```

- Load halfword and store halfword ( Unicode characters )

```
lh    $t0, 0($sp)      # Read halfword (16 bits) from source
sh    $t0, 0($sp)      # Write halfword (16 bits) to destination
```

| Hex code | ASCII character | Hex Code | ASCII character | Hex code | ASCII character | Hex code | ASCII character |
|---|---|---|---|---|---|---|---|
| 00 | NUL | 20 | SP | 40 | @ | 60 | ` |
| 01 | SOH | 21 | ! | 41 | A | 61 | a |
| 02 | STX | 22 | “ | 42 | B | 62 | b |
| 03 | ETX | 23 | # | 43 | C | 63 | c |
| 04 | EOT | 24 | $ | 44 | D | 64 | d |
| 05 | ENQ | 25 | % | 45 | E | 65 | e |
| 06 | ACK | 26 | & | 46 | F | 66 | f |
| 07 | BEL | 27 | ’ | 47 | G | 67 | g |
| 08 | BS | 28 | ( | 48 | H | 68 | h |
| 09 | HT | 29 | ) | 49 | I | 69 | i |
| 0A | LF | 2A | * | 4A | J | 6A | j |
| 0B | VT | 2B | + | 4B | K | 6B | k |
| 0C | FF | 2C | , | 4C | L | 6C | l |
| 0D | CR | 2D | – | 4D | M | 6D | m |
| 0E | SO | 2E | . | 4E | N | 6E | n |
| 0F | SI | 2F | / | 4F | O | 6F | o |
| 10 | DLE | 30 | 0 | 50 | P | 70 | p |
| 11 | DC1 | 31 | 1 | 51 | Q | 71 | q |
| 12 | DC2 | 32 | 2 | 52 | R | 72 | r |
| 13 | DC3 | 33 | 3 | 53 | S | 73 | s |
| 14 | DC4 | 34 | 4 | 54 | T | 74 | t |
| 15 | NAK | 35 | 5 | 55 | U | 75 | u |
| 16 | SYN | 36 | 6 | 56 | V | 76 | v |
| 17 | ETB | 37 | 7 | 57 | W | 77 | w |
| 18 | CAN | 38 | 8 | 58 | X | 78 | x |
| 19 | EM | 39 | 9 | 59 | Y | 79 | y |
| 1A | SUB | 3A | : | 5A | Z | 7A | z |
| 1B | ESC | 3B | ; | 5B | [ | 7B | { |
| 1C | FS | 3C | < | 5C | \ | 7C | \| |
| 1D | GS | 3D | = | 5D | ] | 7D | } |
| 1E | RS | 3E | > | 5E | ^ | 7E | ~ |
| 1F | US | 3F | ? | 5F | _ | 7F | DEL |






*Text Book : P124 ~ P125*


# String Copy

➢ C Code

```
void strcpy ( char x[ ], char y[ ] )
{
    int  i ;
    i = 0 ;
    while ( ( x[i] = y[i] ) != ‘\0’ )
       i += 1 ;
}
```

| Var. | Reg. |
|---|---|
| $X_{base}$ | $a0 |
| $Y_{base}$ | $a1 |
| i | $s0 |

➢ MIPS Code

```
strcpy:  addi    $sp, $sp, -4
         sw      $s0, 0($sp)
         add     $s0, $zero, $zero
L1:      add     $t1, $s0, $a1
         lbu     $t2, 0($t1)
         add     $t3, $s0, $a0
         sb      $t2, 0($t3)
         beq     $t2, $zero, L2
         addi    $s0, $s0, 1
         j       L1
L2:      lw      $s0, 0($sp)
         addi    $sp, $sp, 4
         jr      $ra
```




Department of Electrical Engineering, Feng-Chia University

# Outline


2.7 Instructions for Making Decisions
2.8 Supporting Procedures in Computer Hardware
2.9 Communicating with People
2.10 MIPS Addressing for 32-Bit Immediates and Addresses






# Constants

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:

```
addi   $29, $29, 4
slti   $8, $18, 10
andi   $29, $29, 6
ori    $29, $29, 4
```

- Design Principle: Make the common case fast. *Which format?*

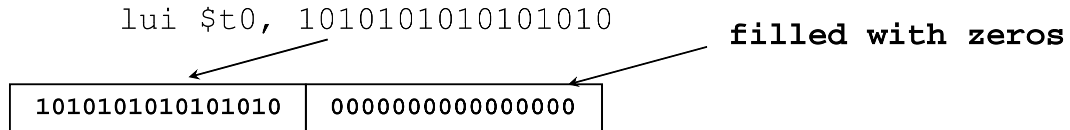
Department of Electrical Engineering, Feng-Chia University

*Text Book : P128*

# How about larger constants?

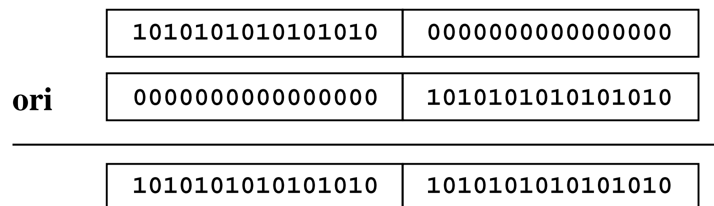
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```




- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```






Department of Electrical Engineering, Feng-Chia University

# Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudo-instructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Department of Electrical Engineering, Feng-Chia University

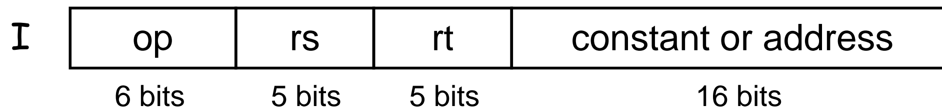
Text Book : P130

# Branch Addressing

- Instructions:

```
bne $t4,$t5,Label    # Next instruction is at Label if $t4≠$t5
beq $t4,$t5,Label    # Next instruction is at Label if $t4=$t5
```

- Most branch targets are near branch
  - Forward or backward




- Addresses are not 32 bits
  — How do we handle this with load and store instructions?
- PC-relative addressing
  - Target address = PC + offset $\times$ 4
  - PC already incremented by 4 by this time

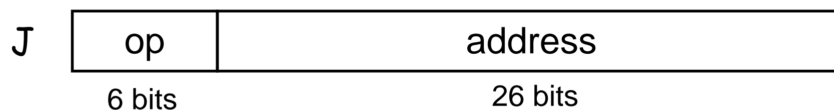


Department of Electrical Engineering, Feng-Chia University

Text Book : P130

# Jump Addressing

- Jump (j and jal ) targets could be anywhere in text segment
  - Encode full address in instruction




- (Pseudo) Direct jump addressing
  - Jump instructions just use high order bits of PC
  - Target address = $PC_{31 \ldots 28}$ : (address $\times$ 4)
  - address boundaries of 256 MB

Department of Electrical Engineering, Feng-Chia University

Text Book : P131

# Showing Branch Offset

- MIPS assembly code:

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

- MIPS machine code:

| Address | | | | | | |
|---|---|---|---|---|---|---|
| 80000 | 0 | 0 | 19 | 9 | 2 | 0 |
| 80004 | 0 | 9 | 22 | 9 | 0 | 32 |
| 80008 | 35 | 9 | 8 | 0 | | |
| 80012 | 5 | 8 | 21 | 2 | | |
| 80016 | 8 | 19 | 19 | 1 | | |
| 80020 | 2 | 2000 | | | | |
| 80024 | … | | | | | |



Department of Electrical Engineering, Feng-Chia University

Text Book : P132

# Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

- Example

```
      beq  $s0, $s1,   L1
               ↓
      bne  $s0, $s1,   L2
      j   L1
L2:   …
```

# Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

| | | | | | | |
|---|---|---|---|---|---|---|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

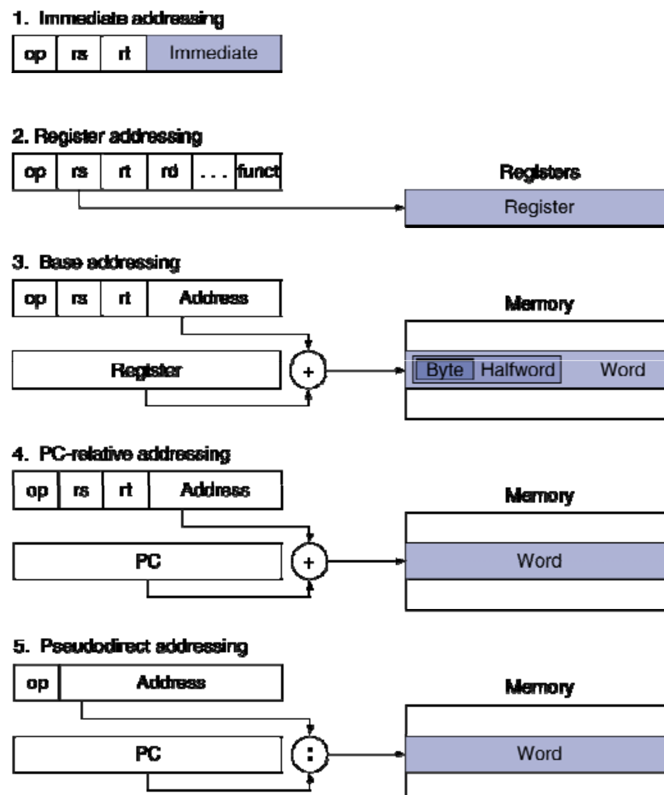
- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can





*Text Book : P133*

# MIPS Addressing Mode Summary



# MIPS Operand Summary

- $2^{30}$ Memory words
- 32 registers

| Name | Register number | Usage |
|---|---|---|
| $zero | 0 | the constant value 0 |
| $at | 1 | reserved for assembler |
| $v0-$v1 | 2-3 | values for results and expression evaluation |
| $a0-$a3 | 4-7 | arguments |
| $t0-$t7 | 8-15 | temporaries |
| $s0-$s7 | 16-23 | saved temporaries |
| $t8-$t9 | 24-25 | more temporaries |
| $k0-$k1 | 26-27 | reserved for OS kernel |
| $gp | 28 | global pointer |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | return address |



Department of Electrical Engineering, Feng-Chia University

# MIPS Instruction Summary

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $s1,$s2,$s3 | $s1 = $s2 + $s3 | Three register operands |
| | subtract | sub $s1,$s2,$s3 | $s1 = $s2 – $s3 | Three register operands |
| | add immediate | addi $s1,$s2,100 | $s1 = $s2 + 100 | Used to add constants |
| Data transfer | load word | lw $s1,100($s2) | $s1 = Memory[$s2 + 100] | Word from memory to register |
| | store word | sw $s1,100($s2) | Memory[$s2 + 100] = $s1 | Word from register to memory |
| | load half | lh $s1,100($s2) | $s1 = Memory[$s2 + 100] | Halfword memory to register |
| | store half | sh $s1,100($s2) | Memory[$s2 + 100] = $s1 | Halfword register to memory |
| | load byte | lb $s1,100($s2) | $s1 = Memory[$s2 + 100] | Byte from memory to register |
| | store byte | sb $s1,100($s2) | Memory[$s2 + 100] = $s1 | Byte from register to memory |
| | load upper immed. | lui $s1,100 | $s1 = 100 * $2^{16}$ | Loads constant in upper 16 bits |
| Logical | and | and $s1,$s2,$s3 | $s1 = $s2 & $s3 | Three reg. operands; bit-by-bit AND |
| | or | or $s1,$s2,$s3 | $s1 = $s2 \| $s3 | Three reg. operands; bit-by-bit OR |
| | nor | nor $s1,$s2,$s3 | $s1 = ~ ($s2 \|$s3) | Three reg. operands; bit-by-bit NOR |
| | and immediate | andi $s1,$s2,100 | $s1 = $s2 & 100 | Bit-by-bit AND reg with constant |
| | or immediate | ori $s1,$s2,100 | $s1 = $s2 \| 100 | Bit-by-bit OR reg with constant |
| | shift left logical | sll $s1,$s2,10 | $s1 = $s2 << 10 | Shift left by constant |
| | shift right logical | srl $s1,$s2,10 | $s1 = $s2 >> 10 | Shift right by constant |
| Conditional branch | branch on equal | beq $s1,$s2,25 | if ($s1 == $s2) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | bne $s1,$s2,25 | if ($s1 != $s2) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | slt $s1,$s2,$s3 | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than; for beq, bne |
| | set less than immediate | slti $s1,$s2,100 | if ($s2 < 100) $s1 = 1; else $s1 = 0 | Compare less than constant |
| Unconditional jump | jump | j 2500 | go to 10000 | Jump to target address |
| | jump register | jr $ra | go to $ra | For switch, procedure return |
| | jump and link | jal 2500 | $ra = PC + 4; go to 10000 | For procedure call |