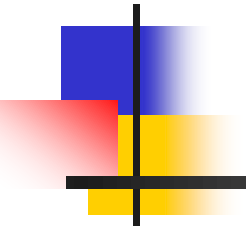


# Chapter2

## Instructions: Language of the Computer

臺大電機系　吳安宇教授
Speaker: Edan

2018/04/22 v3

Minor update on slides

# outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 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
- 2.11 Translating and Starting a Program

# Introduction

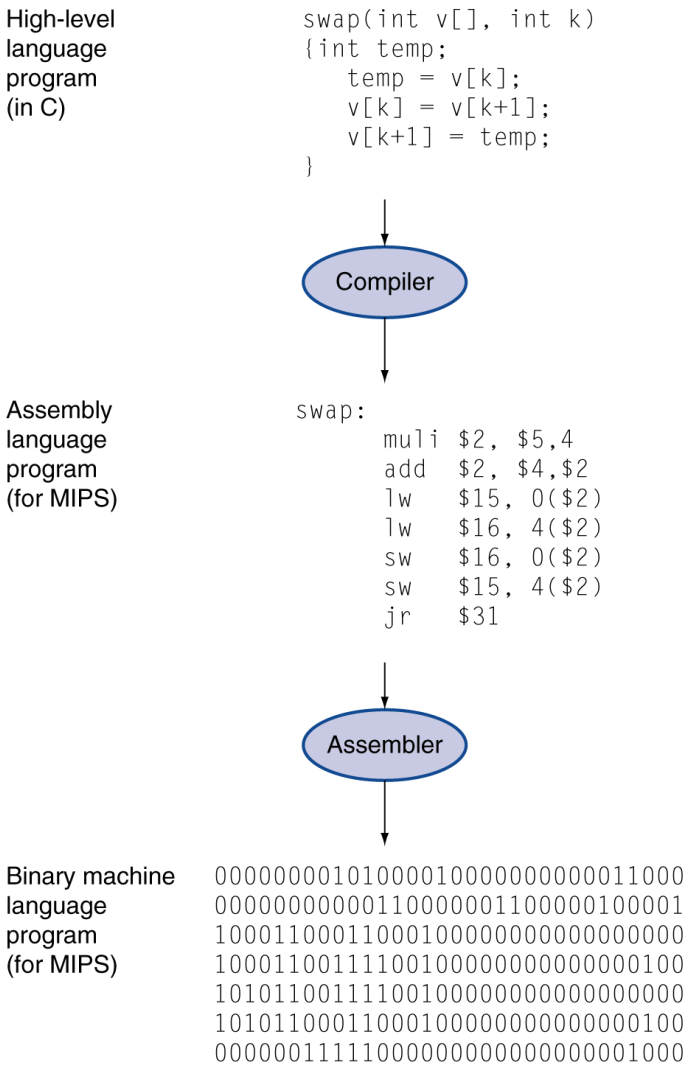
- To command a computer's hardware, you must speak its language.
  - The words of a machine's language are called "Instructions".
  - The vocabulary is called an "Instruction set".
  - Machine's languages are simpler and easier, once you learn one, you can pick up others easily.
  - Reason:

    (1) Hardware and operating principle are similar.

    (2) Basic operations (**add, mul, move**) are similar.
  - Purpose of this chapter: to learn "assembly language".

# Levels of Program Code

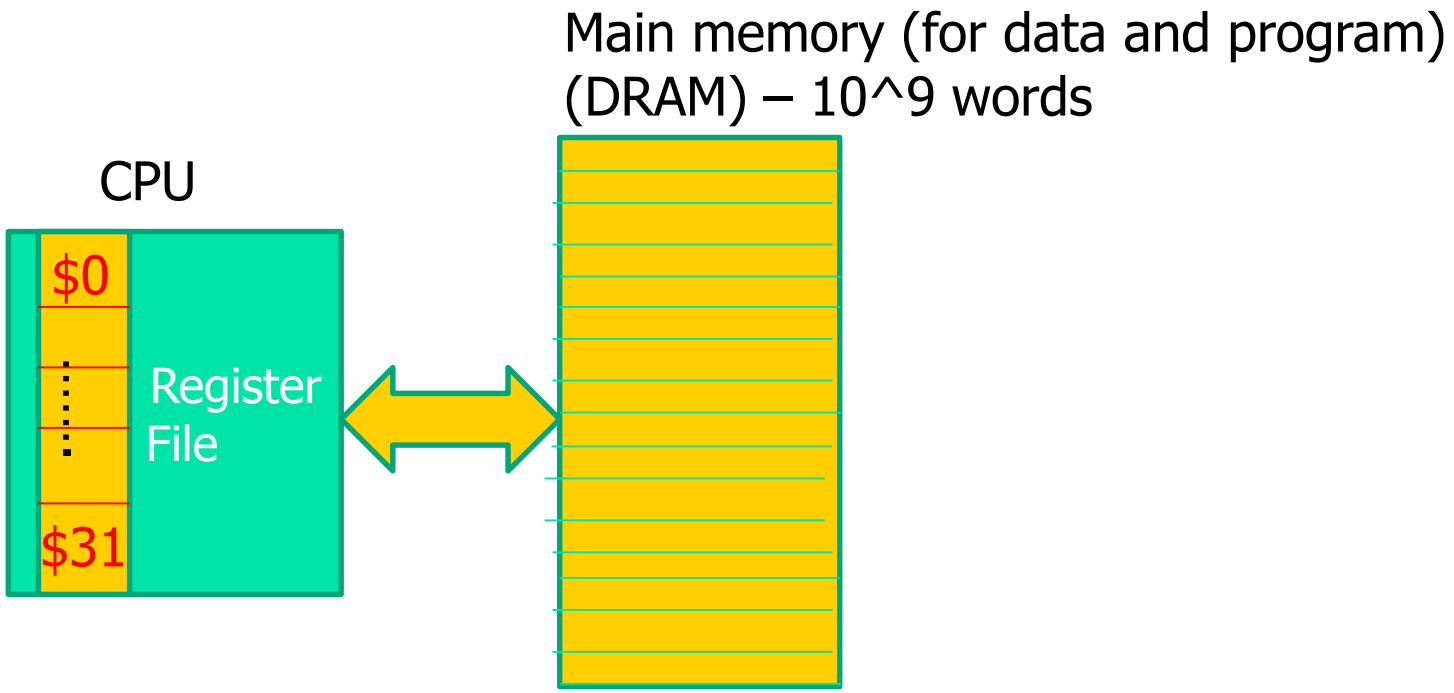
- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



# Overview of MIPS RISC CPU (1/2)

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |



# Overview of MIPS CPU (2/2)

**MIPS assembly language**

| 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 $s1,$s2,20` | $s1 = $s2 + 20 | Used to add constants |
| Data transfer | load word | `lw $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | `sw $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Word from register to memory |
| | load half | `lh $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | load half unsigned | `lhu $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | store half | `sh $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Halfword register to memory |
| | load byte | `lb $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | load byte unsigned | `lbu $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | store byte | `sb $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Byte from register to memory |
| | load linked word | `ll $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Load word as 1st half of atomic swap |
| | store condition. word | `sc $s1,20($s2)` | Memory[$s2+20]=$s1;$s1=0 or 1 | Store word as 2nd half of atomic swap |
| | load upper immed. | `lui $s1,20` | $s1 = 20 * $2^{16}$ | 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 $s1,$s2,20` | $s1 = $s2 & 20 | Bit-by-bit AND reg with constant |
| | or immediate | `ori $s1,$s2,20` | $s1 = $s2 \| 20 | 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 on less than unsigned | `sltu $s1,$s2,$s3` | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than unsigned |
| | set less than immediate | `slti $s1,$s2,20` | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant |
| | set less than immediate unsigned | `sltiu $s1,$s2,20` | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant unsigned |
| 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 |

# Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 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
- 2.11 Translating and Starting a Program

# Operations of the computer hardware

- All instructions have 3 operands
- Operand order is fixed (destination first)
  - Example:

    In C language: a = b + c

    In MIPS: **add** a, b, c

    (we'll talk about registers in a bit)

- The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of

***keeping the hardware as simple as possible!***

# Operations of the computer hardware

- Example1: add 4 numbers (b, c, d, e)

```
add a, b, c  # a=b+c
add a, a, d  # a=a+d (a=b+c+d)
add a, a, e  # a=a+e (a=b+c+d+e)
```

-- It takes three instructions

-- # stands for comment ( for human reader )

--Imply " simplicity favors regularity " (Principle 1)
(Use three-operand instruction can handle all kinds of addition)

-- Some CISC computers can add 3 numbers in one instruction

# Operations of the computer hardware

- Syntax

| Category | instruction | example | meaning | comments |
|---|---|---|---|---|
| Arithmetic | add | add a,b,c | a = b + c | Always 3 **operands** |
| | subtract | sub a,b,c | a = b - c | |

- Writing assembly programs (Example1):
  - In C program: a = b + c;
    d = a – e;
    *There are 5 variables (a,b,c,d,e).*
  - In assembly language: (produced by C compiler)
    add a, b, c
    sub d ,a ,e

# Example 2

- In C program:

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

- In assembly language:

```
add  t0, g, h  # temp variable t0 contains g+h
add  t1, i, j  # temp variable t1 contains i+j
sub  f, t0, t1  # f=(g+h)-(i+j)
```

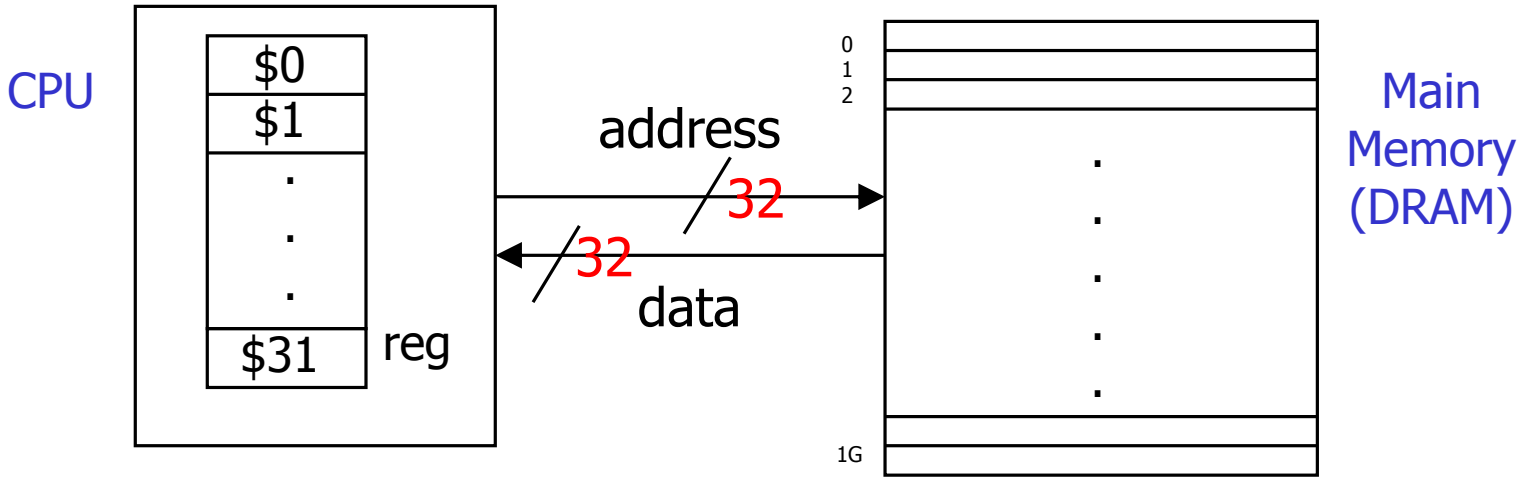
-- Note that compiler "created" t0 and t1 to express the program in the restricted three-operands-per-instruction notation.

# Outline

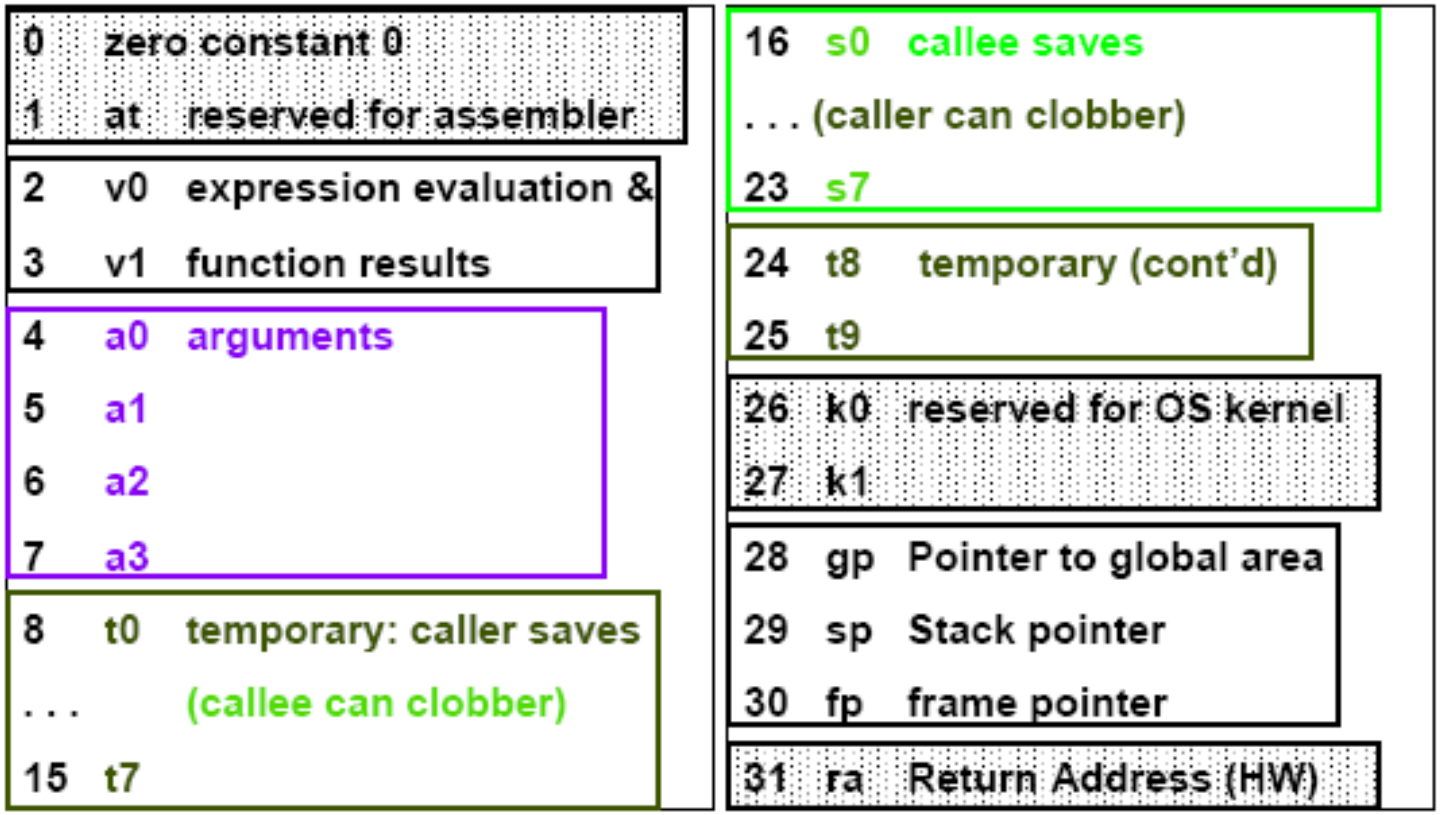
- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program

# Registers

- Main (and the only) Storage within CPU.
- The size of a register in the MIPS architecture is 32-bit.
- "word" = 32 bits in MIPS.
- MIPS has 32 registers, using the notation $0, $1, ……, $31 to represent them.
- Smaller is faster (design principle 2)



# MIPS Register Convention



# Compiler → Assembly Language

- For now on, we use ***$s0, $s1***, …..for registers that correspond to variables in C and Java programs

- Use ***$t0, $t1,*** …..for temporary registers needed to compile the program into MIPS instructions.

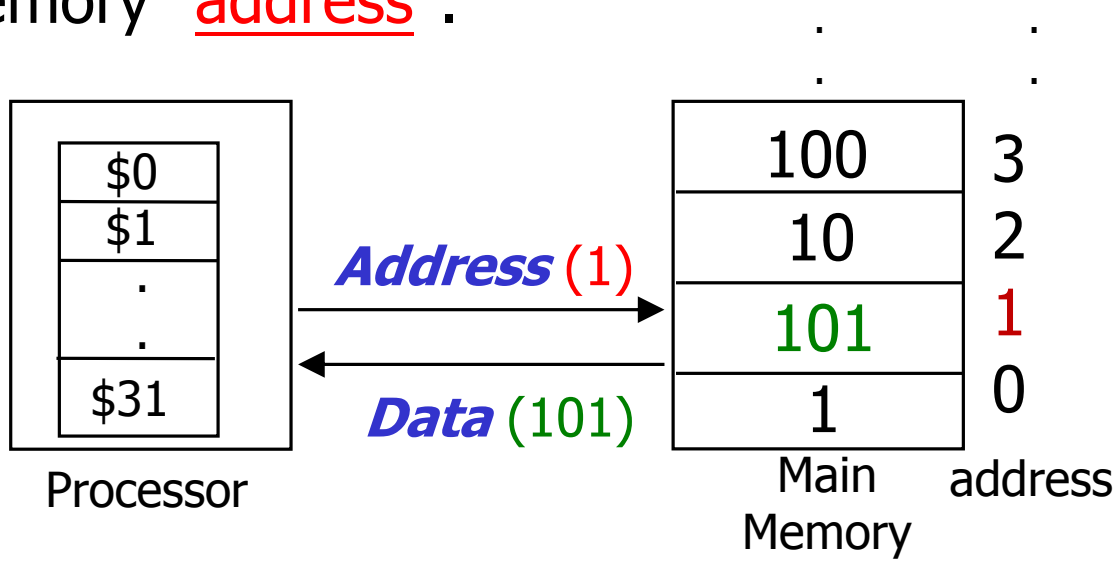
- Example: f = ( g + h ) – ( i + j )
  - Assignment: (by compiler)

    f = $s0, g = $s1, h = $s2, i = $s3, j = $s4
  - Compiler → assembly language

```
add $t0, $s1, $s2     # t0 = g+h
add $t1, $s3 $s4      # t1= i+j
sub $s0, $t0, $t1     # f = (g+h)-(i+j)
```

# Data transfer instructions

- CPU can store only a small amount of data in registers ($0, $1, ……, $31).
- Arithmetic operations occur "ONLY" on registers in MIPS.
- MIPS needs instructions that transfer data between memory and registers.
- To access a word in memory, the instruction must supply the memory "address".



# LW: Load Word from Memory

- Load
  - The data transfer instruction that moves data from memory to a register.
  - MIPS instruction: lw = load word
  - Example:

    Assume that A is an array of 100 words (**int A[100] in C**), g=$s1, h=$s2, and the starting address (base address) of the array **= $s3.**

    ```
    g = h + A[8];
    ```

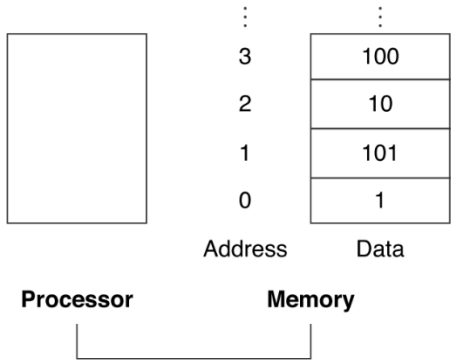
    The address of the array element is the **sum** of the **base of the Array A (stored in $s3)** plus the number to select element 8.

    ```
    lw   $t0, 8($s3)        # $t0 ← A[8] = MEM[A+8]
    add  $s1, $s2, $t0      # g = h + A[8]
    ```

  - The constant in a data transfer instruction is called the offset.
  - The register added to from the address is called the base register.

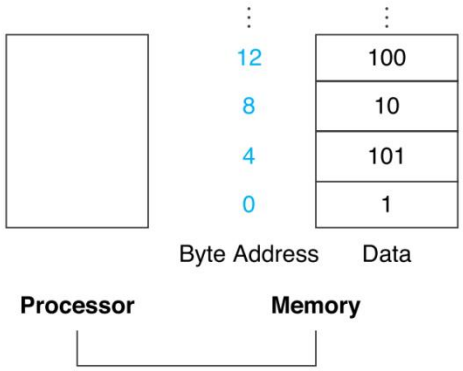
# Address format in MIPS

- 1 word (32bits) = **4 bytes**
- Memory unit is usually "**Byte (8 bits)**"




Byte = 8 bits

**FIGURE 2.2 Memory addresses and contents of memory at those locations.** If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.




Word = 4 Bytes

**FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words.** The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

# SW: Store Word to Memory

- Store
  - The data transfer instruction that moves data from a register to memory.
  - MIPS instruction: **sw = store word**
  - Example:

    Assume that h=$s2, and the Base address of **Array A= $s3**

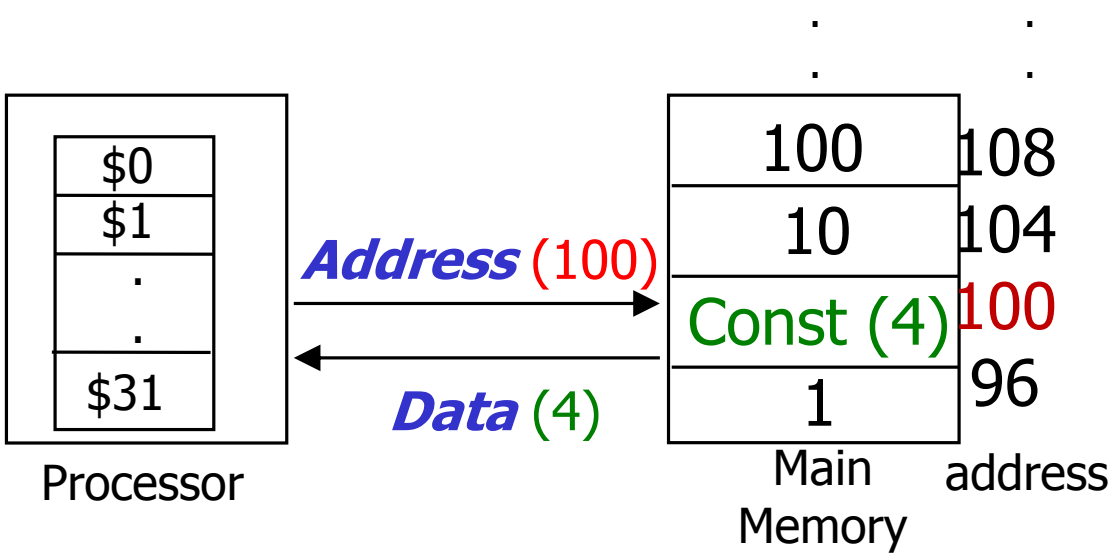
    A[12] = h + A[8];  → C Program

    Compiler → Assembly language

```
lw   $t0, 32($s3)      # temp $t0 = A[8]
add $t0, $s2, $t0      # temp $t0= h + A[8]
sw  $t0, 48($s3)       # A[12] = h +A[8]
```

# Operands with Constant

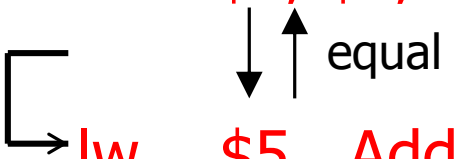
- Assume that AddrConstant4 is the memory address of the constant 4.
- MEM[AddrConstant4 + $1] ← Constant 4
- lw $5 , AddrConstant4 ($1) # $5 = constant 4



# Operands with Constant

- Constant as Operand (Immediate Addressing mode)
  - addi = add immediate  # add an immediate constant

```
addi $3, $3, 4                          # $3 = $3 + 4
```




```
lw   $5 , AddrConstant4 ($1) # $5 = constant 4
add  $3, $3, $5              # $3 = $3 + $5  ($5 =4)
```

Assume that AddrConstant4 is the memory address of the constant 4.

# Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 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
- 2.11 Translating and Starting a Program

# Positive numbers (addr & data)

- $1011_2 = 1x2^3+0x2^2+1x2^1+1x2^0$
  $= 8+2+1 = 11_{10}$
- In MIPS word ( = 32 bits)

| 0000 | 0000 | …… | 0000 | 1011 |
|---|---|---|---|---|

  - Bit_31= Leftmost bit = ***Most significant bit (MSB)***
  - Bit_0 = Rightmost bit = ***Least significant bit (LSB)***

We can represent positive integers (without negative number):

0 = $0000……0000_2$

1 = $0000……0001_2$

2 = $0000……0010_2$

:

:

$2^{32}$-1 = $4,294,967,295_{10}$ = $1111……1111_2$

# 2's Complement Representation (data only)

MSB=sign bit

- 0: Positive number or zero
- 1: Negative number

- Formula

Value in Base-10

$= -2^{31}xbit31 + 2^{30}\,xbit30 + 2^{29}xbit29 + \ldots\ldots + 2^{0}xbit0$

- Example
  - $1111\ 1111\ \ldots\ldots\ 1111\ 1100_2$

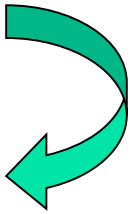
  $= -2^{31}+2^{30}+2^{29}+\ldots\ldots+2^2 = -4_{10}$

  - Flip and plus one (negation)

    $0000\ 0000\ \ldots\ldots\ 0000\ 0011 + 1 = 4_{10}$

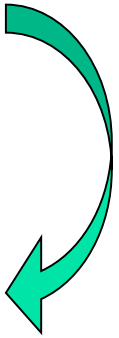
# Shortcut of Negation

- Negate $2_{10}$, and then check the result by negating $-2_{10}$
- $2_{10}$=0000 0000 0000 0000 0000 0000 0000 $0010_2$



Flip & Add 1

$$\begin{array}{l} \quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\ + \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_2 \\ \hline = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 \\ = -2_{10} \end{array}$$



Flip & Add 1

$$\begin{array}{l} \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \\ + \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_2 \\ \hline = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 \\ = 2_{10} \end{array}$$

# Sign Extension

- Convert 16-bit binary version of $2_{10}$ and $-2_{10}$ to 32-bit binary numbers.
- $0000\ 0000\ 0000\ 0010_2 = 2_{10}$

Make 16 copies of the value in MSB. Then, make 2's complement

➔ $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

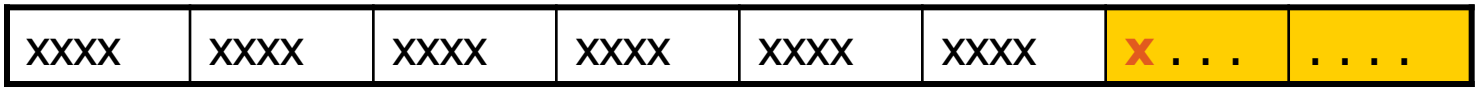
$$
\begin{array}{rl}
 & 1111\ 1111\ 1111\ 1101_2 \\
+ & \underline{\qquad\qquad\qquad\qquad\quad 1_2} \\
= & \mathbf{1}111\ 1111\ 1111\ 1110_2 \\
= & -2_{10}
\end{array}
$$

Make 16 copies of the value in MSB (the sign bit)

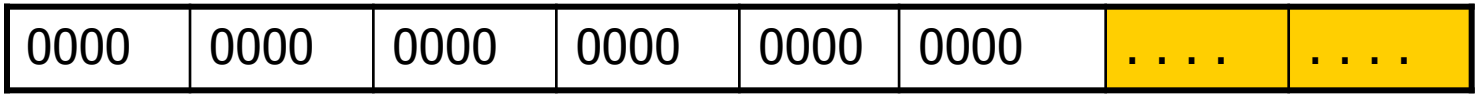
➔ $1111\ 1111\ 1111\ 1111\ \mathbf{1}111\ 1111\ 1111\ 1110 = -2_{10}$

# Sign Extension

- **lb** (**load byte** from memory to 32-bit register)
  Need Sign extension: fill with x's (x=0 or 1)

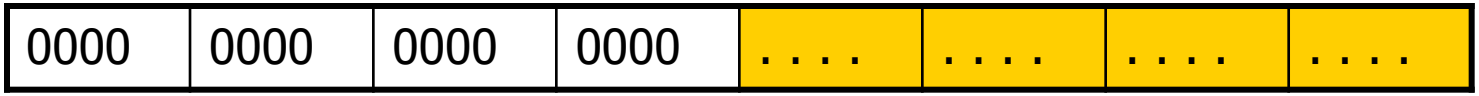



- **lbu (load byte unsigned** from memory to register)
  fill with 0's




- **lh** (**load half word**)


- **lhu (load half word unsigned**)



# Summary

**MIPS assembly language**

| 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 $s1,$s2,20 | $s1 = $s2 + 20 | Used to add constants |
| Data transfer | load word | lw $s1,20($s2) | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | sw $s1,20($s2) | Memory[$s2 + 20] = $s1 | Word from register to memory |
| | load half | lh $s1,20($s2) | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | load half unsigned | lhu $s1,20($s2) | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | store half | sh $s1,20($s2) | Memory[$s2 + 20] = $s1 | Halfword register to memory |
| | load byte | lb $s1,20($s2) | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | load byte unsigned | lbu $s1,20($s2) | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | store byte | sb $s1,20($s2) | Memory[$s2 + 20] = $s1 | Byte from register to memory |
| | load linked word | ll $s1,20($s2) | $s1 = Memory[$s2 + 20] | Load word as 1st half of atomic swap |
| | store condition. word | sc $s1,20($s2) | Memory[$s2+20]=$s1;$s1=0 or 1 | Store word as 2nd half of atomic swap |
| | load upper immed. | lui $s1,20 | $s1 = 20 * $2^{16}$ | Loads constant in upper 16 bits |

**MIPS assembly language**

| 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 $s1,$s2,20 | $s1 = $s2 + 20 | Used to add constants |
| Data transfer | load word | lw $s1,20($s2) | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | sw $s1,20($s2) | Memory[$s2 + 20] = $s1 | Word from register to memory |
| | load half | lh $s1,20($s2) | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | load half unsigned | lhu $s1,20($s2) | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | store half | sh $s1,20($s2) | Memory[$s2 + 20] = $s1 | Halfword register to memory |
| | load byte | lb $s1,20($s2) | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | load byte unsigned | lbu $s1,20($s2) | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | store byte | sb $s1,20($s2) | Memory[$s2 + 20] = $s1 | Byte from register to memory |
| | load linked word | ll $s1,20($s2) | $s1 = Memory[$s2 + 20] | Load word as 1st half of atomic swap |
| | store condition. word | sc $s1,20($s2) | Memory[$s2+20]=$s1;$s1=0 or 1 | Store word as 2nd half of atomic swap |
| | load upper immed. | lui $s1,20 | $s1 = 20 * $2^{16}$ | 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 $s1,$s2,20 | $s1 = $s2 & 20 | Bit-by-bit AND reg with constant |
| | or immediate | ori $s1,$s2,20 | $s1 = $s2 \| 20 | 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 on less than unsigned | sltu $s1,$s2,$s3 | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than unsigned |
| | set less than immediate | slti $s1,$s2,20 | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant |
| | set less than immediate unsigned | sltiu $s1,$s2,20 | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant unsigned |
| 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 |

# Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 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
- 2.11 Translating and Starting a Program

# Review of Number Systems (I)

- The hexadecimal-to-binary conversion table

| Hexadecimal | Binary | Hexadecimal | Binary | Hexadecimal | Binary | Hexadecimal | Binary |
|---|---|---|---|---|---|---|---|
| $0_{hex}$ | $0000_{two}$ | $4_{hex}$ | $0100_{two}$ | $8_{hex}$ | $1000_{two}$ | $c_{hex}$ | $1100_{two}$ |
| $1_{hex}$ | $0001_{two}$ | $5_{hex}$ | $0101_{two}$ | $9_{hex}$ | $1001_{two}$ | $d_{hex}$ | $1101_{two}$ |
| $2_{hex}$ | $0010_{two}$ | $6_{hex}$ | $0110_{two}$ | $a_{hex}$ | $1010_{two}$ | $e_{hex}$ | $1110_{two}$ |
| $3_{hex}$ | $0011_{two}$ | $7_{hex}$ | $0111_{two}$ | $b_{hex}$ | $1011_{two}$ | $f_{hex}$ | $1111_{two}$ |

**FIGURE 2.4 The hexadecimal-binary conversion table.** Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

# Review of Number System (II)

- Example1:

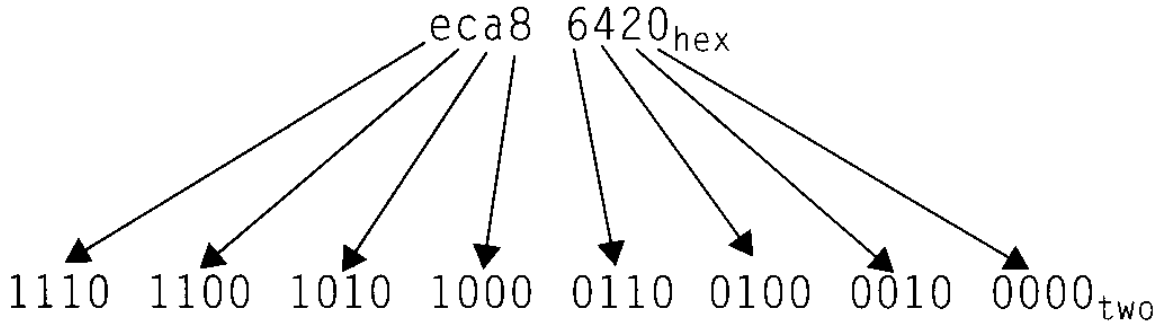
$123_{10}$ = $1111011_2$

(base 10) (base 2)

- Example2:

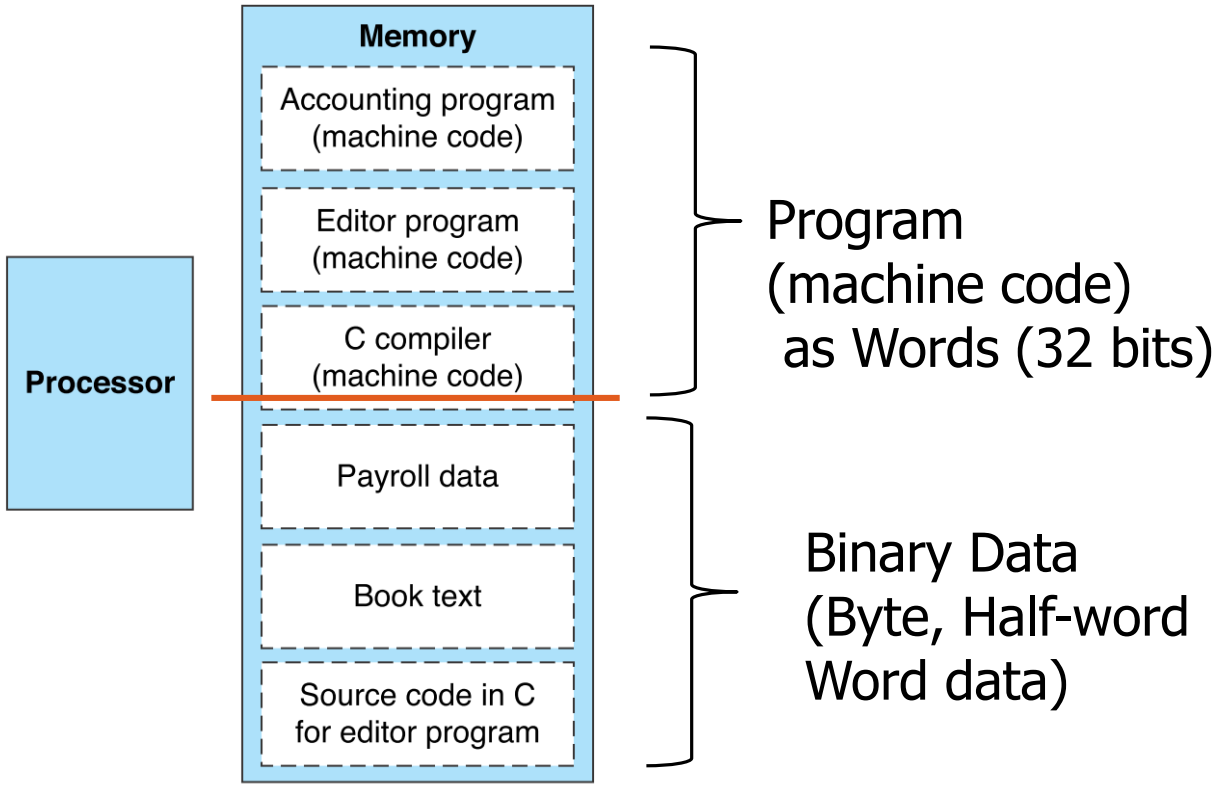
$eca8\ 6420_{16}$ = $1110\ 1100\ 1010\ 1000\ 0110\ 0100\ 0010\ 0000_2$

(base 16) (base 2)

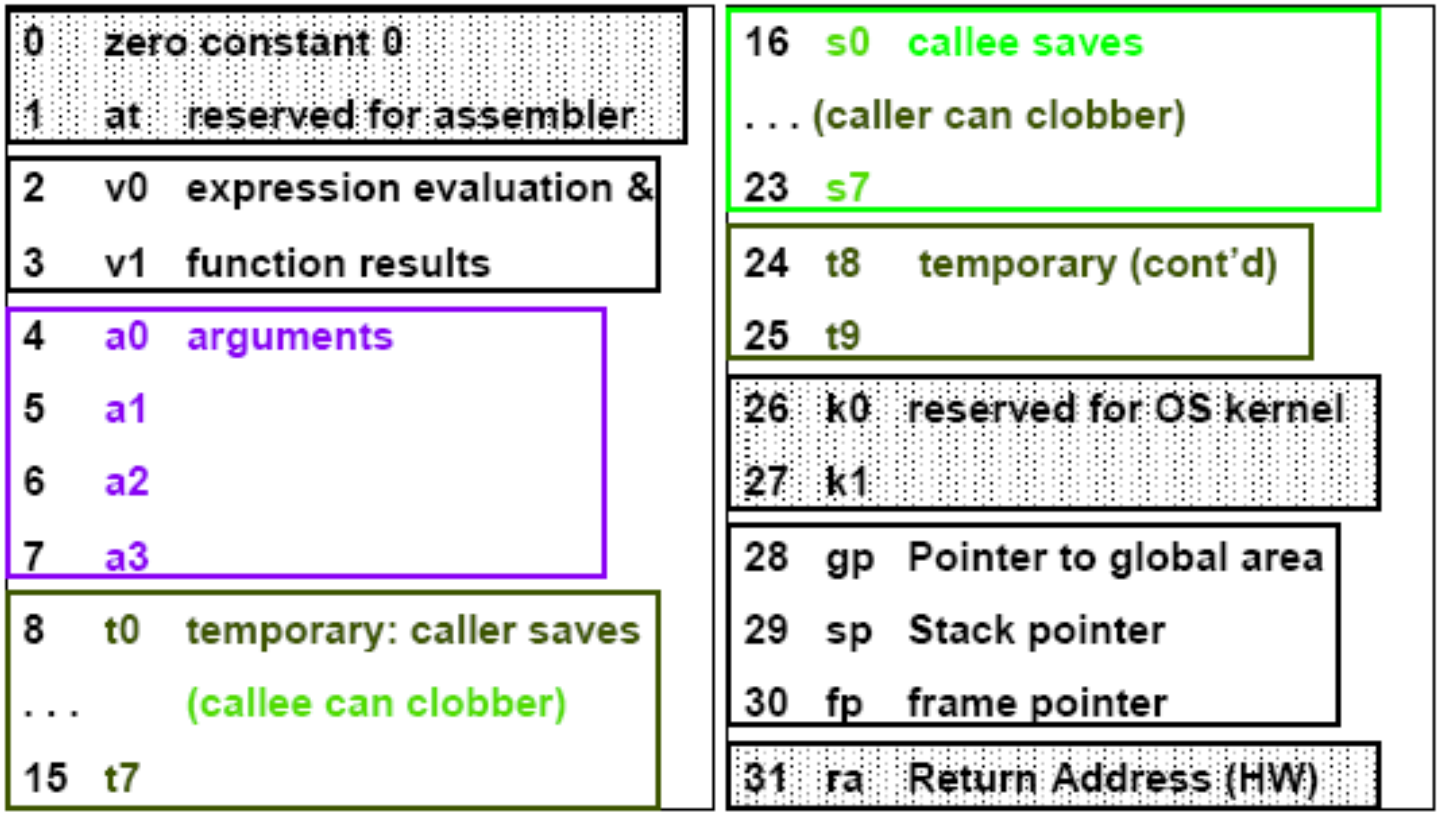


# "Stored-program" Concept

- Instructions are represented as (32-bit) numbers!!
- Programs can be stored in memory to be read or written just like numbers.
- Most important concept in computer history (*von Neumann architecture*)



# Review of MIPS register conventions



# Representing Instructions in Binary Number

- MIPS instruction Example: add $t0, $s1, $s2

| 0 | 17 | 18 | 8 | 0 | 32 |
|---|---|---|---|---|---|

  - The first and last fields (containing 0 and 32 in this case) in combination ➔ addition operation
  - The 2nd field (17 = $s1) ➔ the number of the register that is the first source operand of the addition operation.
  - The 3rd field (18 = $s2) ➔ the number of the register that is the second source operand of the addition operation.
  - The 4th field (8 = $t0) ➔ the number of the register that is to receive the sum.
  - Binary numbers (for machine to use) → **Machine language (32 bits)**

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |
|---|---|---|---|---|---|
| **6 bits** | **5 bits** | **5 bits** | **5 bits** | **5 bits** | **6 bits** |

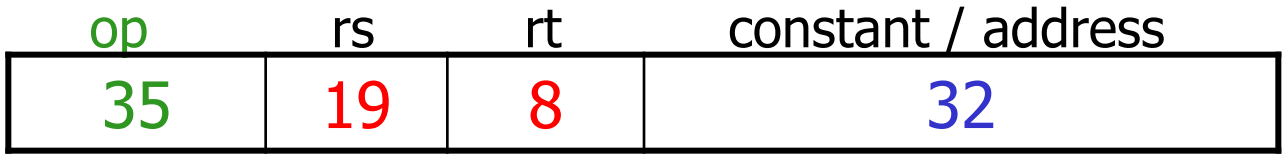
# Instruction format in MIPS (R-type)

- R- type (register related)


- meaning:
  - op : operation of the instruction
  - rs : the first register source operand
  - rt : the second register source operand
  - rd : the register destination operand
  - Shamt : shift amount
  - funct : function; this field selects the variant of the operation in the op field (e.g., add, sub)

# Representing Instructions in the Computer

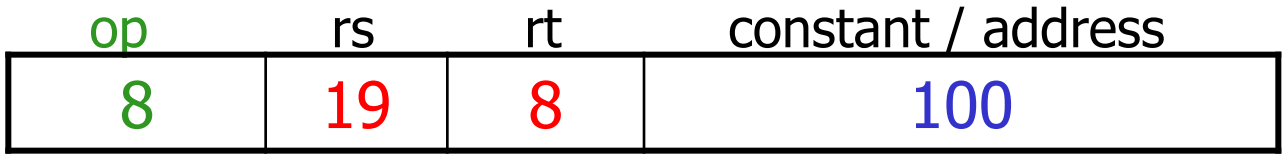
- Example: lw $t0 , 32($s3)




- 19 (for $s3) is placed in the rs field.
- 8 (for $t0) is placed in the rt field.
- 32 (offset) is placed in the constant/address field.
- Note: In a load word instruction, **rt** field specifies the destination register, which receives the result of the load.

# Representing Instructions in the Computer

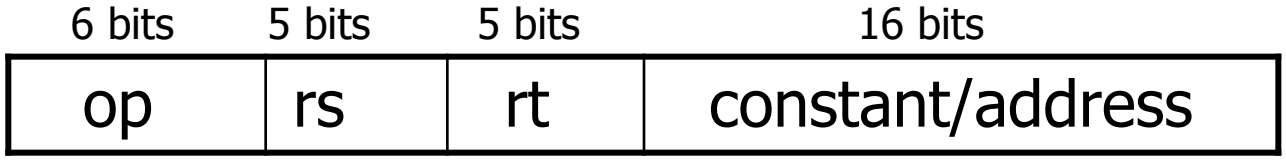
- Example: addi $t0 , $s3, 100




- 19 (for $s3) is placed in the rs field.
- 8 (for $t0) is placed in the rt field.
- 100 (constant) is placed in the constant/address field.
- Note: In a load word instruction**, rt** field specifies the destination register, which receives the result of the load.

# Instruction format in MIPS (I-type)

- I- type (data transfer, or immediate mode)




  - Meaning:

    op : operation of the instruction

    rs : the source register (usually the **base register**)

    rt : the register to receive the result of the operation

    address : the **offset** from the base register

  - **Constant/address (16-bit):**

- The 16-bit address means a Load Word instruction can load any word within a region of $\pm 2^{15}$ or 32768 **bytes** ($\pm 2^{13}$ or 8192 **words**) of the address in the base register **rs**

  - Similarly, the immediate is limited **to constants <= $\pm 2^{15}$**

# Instruction format in MIPS

- Two major classes of **instruction format** **in MIPS**

  (1) R- type (register related) : add, sub

  (2) I - type (data transfer) : lw, sw

| Instruction | Format | op | rs | rt | rd | shamt | funct | address |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | reg | reg | reg | 0 | $32_{ten}$ | n.a. |
| sub (subtract) | R | 0 | reg | reg | reg | 0 | $34_{ten}$ | n.a. |
| add immediate | I | $8_{ten}$ | reg | reg | n.a. | n.a. | n.a. | constant |
| lw (load word) | I | $35_{ten}$ | reg | reg | n.a. | n.a. | n.a. | address |
| sw (store word) | I | $43_{ten}$ | reg | reg | n.a. | n.a. | n.a. | address |

  - Design issue : keep format similar
  - Use first field **(op)** to distinguish lower 16 bits
    - *R-type* : three fields (rd, shamt, function)
    - *I-type* : single field for 16-bit address/immediate data

# Representing Instructions in the Computer

- Example:

  Assume h=$s2 and $t1 has the base of the array A.

  A[300] = h +A[300];

- Compiler → assembly language

```
lw   $t0, 1200($t1)     # temp $t0 = A[300]
add $t0, $s2, $t0       # temp $t0= h + A[300]
sw  $t0, 1200($t1)      # A[300] = h +A[300]
```

- Assembler → machine language (in decimal number)

| op | rs | rt | (rd) | (shamt) | address/ (Function) |
|---|---|---|---|---|---|
| 35 | 9 | 8 | 1200 | | |
| 0 | 18 | 8 | 8 | 0 | 32 |
| 43 | 9 | 8 | 1200 | | |

# Representing Instructions in the Computer

- The base register **9 ($t1)** is specified in the second field (rs).
- The destination register **8 ($t0)** is specified in the third field (rt).
- The offset to select A[300] (1200 = 300 * 4) is found in the final field (constant/address).

- Final look in MIPS machine language (32-bit width):

| op | rs | rt | (rd) | (shamt) | address/ function |
|---|---|---|---|---|---|
| 100011 | 01001 | 01000 | 0000 0100 1011 0000 | | |
| 000000 | 10010 | 01000 | 01000 | 00000 | 100000 |
| 101011 | 01001 | 01000 | 0000 0100 1011 0000 | | |

# MIPS Machine Language

**MIPS machine language**

| Name | Format | Example | | | | | | Comments |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | 18 | 19 | 17 | 0 | 32 | add $s1,$s2,$s3 |
| sub | R | 0 | 18 | 19 | 17 | 0 | 34 | sub $s1,$s2,$s3 |
| addi | I | 8 | 18 | 17 | 100 | | | addi $s1,$s2,100 |
| lw | I | 35 | 18 | 17 | 100 | | | lw $s1,100($s2) |
| sw | I | 43 | 18 | 17 | 100 | | | sw $s1,100($s2) |
| Field size | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | All MIPS instructions are 32 bits long |
| R-format | R | op | rs | rt | rd | shamt | funct | Arithmetic instruction format |
| I-format | I | op | rs | rt | address | | | Data transfer format |

# Pseudo Instruction

- Note:
  - op code:
    - lw (35) : 1 0 0 0 1 1
    - sw (43): 1 0 1 0 1 1 → to simplify the HW design
  - Reg $0 is always **0**.
  - Usage : Assembly language: move $8, $18 # $8=$18

**Assembler (system software)**

Machine language: add $8, $0 ,$18 # $8=$18+0

→ "move" is a "pseudo instruction" which simplifies our programming in writing Assembly language, **not implemented** in the HW design.

# Summary of MIPS Inst. Set & Format

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | $s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at | Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register $zero always equals 0, and register $at is reserved by the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

**MIPS assembly language**

| 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 $s1,$s2,20 | $s1 = $s2 + 20 | Used to add constants |
| Data transfer | load word | lw $s1,20($s2) | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | sw $s1,20($s2) | Memory[$s2 + 20] = $s1 | Word from register to memory |

**MIPS machine language**

| Name | Format | Example | | | | | | Comments |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | 18 | 19 | 17 | 0 | 32 | add $s1,$s2,$s3 |
| sub | R | 0 | 18 | 19 | 17 | 0 | 34 | sub $s1,$s2,$s3 |
| addi | I | 8 | 18 | 17 | 100 | | | addi $s1,$s2,100 |
| lw | I | 35 | 18 | 17 | 100 | | | lw $s1,100($s2) |
| sw | I | 43 | 18 | 17 | 100 | | | sw $s1,100($s2) |
| Field size | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | All MIPS instructions are 32 bits long |
| R-format | R | op | rs | rt | rd | shamt | funct | Arithmetic instruction format |
| I-format | I | op | rs | rt | address | | | Data transfer format |

# Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 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
- 2.11 Translating and Starting a Program

# Logical Operations

| Operation | C | Java | MIPS |
|---|---|---|---|
| Shift left | << | << | sll |
| Shift right | >> | >>> | srl |
| Bitwise AND | & | & | and, andi |
| Bitwise OR | \| | \| | or, ori |
| Bitwise NOT | ~ | ~ | nor |

- Example:

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_2 = 9_{10}$

  **Shift left by 4** → $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_2 = 144_{10}$

- sll (shift left logical) (c.f. Shift right logical, srl)

```
sll  $t2, $s0, 4          # $t2 = $s0 << 4 bits
```

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | unused | 16 | 10 | 4 | 0 |
| 000000 | 00000 | 10000 | 01010 | 00100 | 000000 |

# Logical Operations

- Example: **($t2 as mask)**

  assume $t2 = 0000 0000 0000 0000 **1**000 **1111** 0000 $0000_2$

  $t1 = 0000 0000 0000 0000 0011 1100 0000 $0000_2$

(1) AND operation (set to 0 OR mask of 1)

**and** $t0 , $t1, $t2 # $t0 = $t1 & $t2

| 0 | 9 | 10 | 8 | 0 | 36 |
|---|---|---|---|---|---|

→ $t0 = 0000 0000 0000 0000 0000 1100 0000 $0000_2$

(2) OR operation (set to 1)

**or** $t0 , $t1, $t2 # $t0 = $t1 | $t2

| 0 | 9 | 10 | 8 | 0 | 37 |
|---|---|---|---|---|---|

→ $t0 = 0000 0000 0000 0000 1011 1111 0000 $0000_2$

# Logical Operations

- Example:

  assume $t2 = 0000 0000 0000 0000 0000 1101 0000 $0000_2$

  $t1 = 0000 0000 0000 0000 0011 1100 0000 $0000_2$

(3) NOR operation (R-type)

Example:

**nor $t0 , $t1, $t2** **# $t0 = ~ ( $t1 | $t2)**

➔ $t0 = 1111 1111 1111 1111 1100 0011 1111 $1111_2$

| 0 | 9 | 10 | 8 | 0 | 39 |
|---|---|---|---|---|---|

(4) Pseudo Inst.: Perform **NOT operation using NOR Instruction**

A **NOR** 0 = **NOT** (A **OR** 0) = NOT (A)

**nor $t0 , $t1, $0** # $t0 = ~ ( $t1 | 0)

# MIPS Instructions

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

**MIPS assembly language**

| 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 $s1,$s2,20` | $s1 = $s2 + 20 | Used to add constants |
| 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 $s1,$s2,20` | $s1 = $s2 & 20 | Bit-by-bit AND reg with constant |
| | or immediate | `ori $s1,$s2,20` | $s1 = $s2 \| 20 | 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 |
| Data transfer | load word | `lw $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | `sw $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Word from register to memory |

# Outline

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Representing Instructions in the Computer
- 2.5 Logical Operations
- 2.6 Instructions for Making Decisions
- 2.7 Supporting Procedures in Computer Hardware
- 2.8 Communicating with People
- 2.9 MIPS Addressing for 32-bit Immediates and Addresses
- 2.10 Translating and Starting a Program

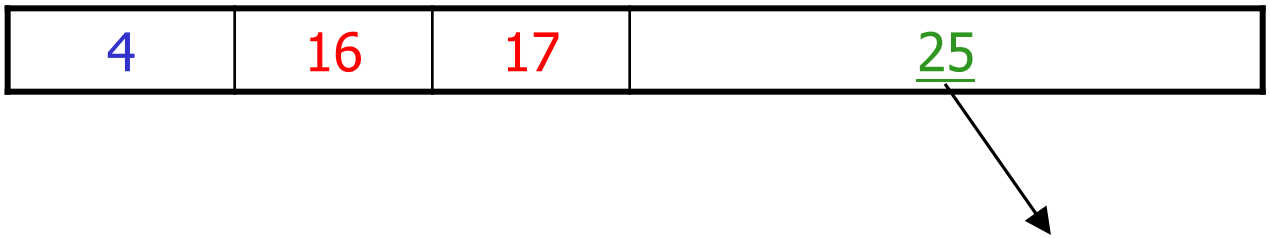
# Instructions for Making Decisions

- (a) beq reg1 reg2 L1 ***beq = "branch equal"***

  # if (reg1 value) == (reg2 value)
  then goto statement labled L1

  - Example:

    beq $s1, $s2, L1 # if ($s1==$s2) goto location (PC+4)+100




Branch by 25 **Instructions**
→ New location of Instruction in memory:
Next Program Counter (PC+4) + 25 x 4 (=100)

# Instructions for Making Decisions

- (b) bne reg1 reg2 L1 ***bne = "branch not equal"***

  # if (reg1 value) != (reg2 value)
  then goto statement labled L1

  - Example:
    bne $s1, $s2, 100 # if ($16!=$17) goto location (PC+4)+100

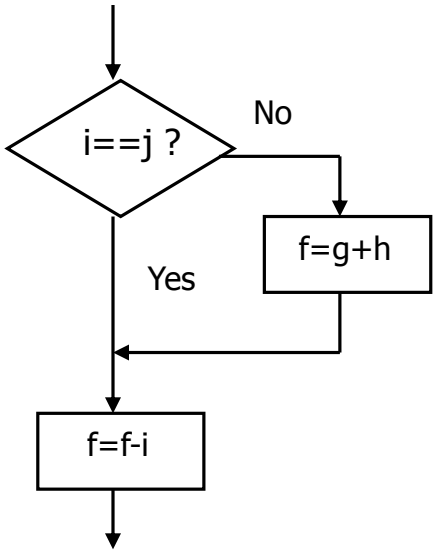

- The two instructions are traditionally called **conditional branches.**

# Instructions for Making Decisions

- Example:

  In C language :

```
        If ( i == j ) goto L1
        f = g + h;
L1:     f = f – i;
```



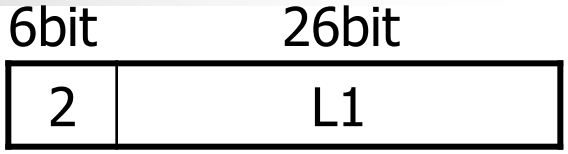

In MIPS: assume i=$s1, j =$s2, f=$s3, g=$s4, h=$s5

PC (Program Counter)

```
100   beq $s1, $s2, L1     # if (i==j) goto L1
104   add $s3, $s4, $s5    # f=g+h (skipped if i==j)
L1    sub $s3, $s3, $s1    # f=f-i (always executed)
```

L1 = address of the subtract instruction ***(L1 =108)***

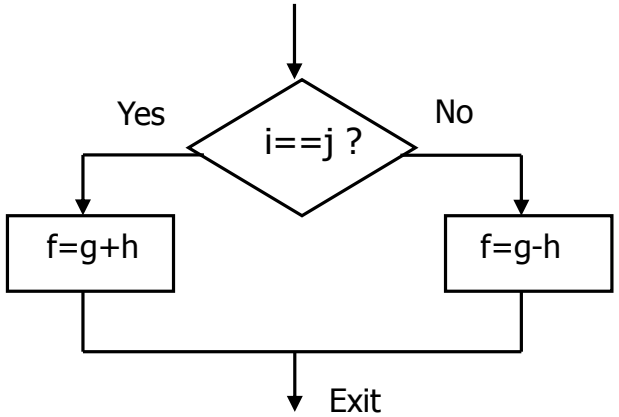
# Instructions for Making Decisions




- (c) jump (unconditional branch) : j L1
  - Example:

In C language :

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




In MIPS: assume i=$3, j =$s4, f=$s0, g=$s1, h=$s2

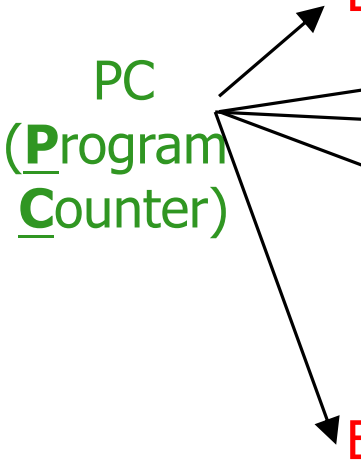
```
100     bne $s3, $s4, Else      #  if (i!=j) goto Else (+2)
104     add $s0, $s1, $s2       #  f=g+h
108     J  Exit                 #  jump Exit
Else:   sub $s0, $s1, $s2       #  f=g-h,   (Else=112)
Exit:   ..................      #  (Exit = 116)
        ..................
```

# Instructions for Making Decisions

- Loops: **In C language :**

```
while ( A[i] == k )
        i+=1;     # counting continuous k value in A[]
```

In MIPS: assume i=$s3, k=$s5,
and the base register of the array **A** is in $s6




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

# Instructions for Making Decisions (machine code)

- Addressing in branches and jumps
  - J-type : consists of 6 bits for the operation field and the rest of the bits for the address.


  - Example: j 10000 # go to the location 10000


  - Unlike the jump instruction, the conditional branch instruction must specify 2 operands in addition to the branch address.

bne $s0, $s1, Exit # if ($s0!=$s1) go to Exit (+/- 100)


- **Program Counter (PC) = (PC + 4, next inst) + branch address x4**

# Instructions for Making Decisions: Slt

- Basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

- slt reg3, reg1, reg2 **slt : set ON less than *(R-type)***
  - ➔ Compare two registers and set
    if (reg1 < reg2) reg3=1; Otherwise, reg3=0;
  - Example:
    slt $s1, $s2, $s3 #if ($s2<$s3) set $s1=1, else $s1=0.

| 0 | 18 | 19 | 17 | 0 | 42 |
|---|---|---|---|---|---|

- "Set on less than" immediate (slti) – *I-type (page P.B-58)*
  - Example: slti $t0, $s2, 10
    # if ($s2 < 10), $t0 =1; Otherwise, $t0 =0.

# Set ON Less Than (slt)

- Signed integer (+/-) : Normal numbers (usually)
- Unsigned integer (+) : Memory addressing

→ **slt (slti) : set ON less than, signed integer (immediate)**
**sltu (sltiu) : set ON less than, unsigned integer (immediate)**

- Example

$s1 = 1111 1111 …… $1111_2$ (**= -1 for signed or $(2^{32}-1)$ for unsigned**)
$s2 = 0000 0000 …… $0001_2$ (**= +1 for signed and for unsigned**)

**slt $t0, $s1, $s2 # signed comparison → $t0 =1**
**sltu $t0, $s1, $s2 # unsigned comparison → $t0=0**

| Inst | Example | Meaning (unsigned comparison) |
|---|---|---|
| sltu | sltu $s1,$s2,$s3 | If($s2<$s3),$s1=1; else $s1=0 |
| sltiu | sltiu $s1,$s2,100 | If($s2<100),$s1=1; else $s1=0 |

# Summary of MIPS Instructions

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

**MIPS assembly language**

| 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 $s1,$s2,20` | `$s1 = $s2 + 20` | Used to add constants |
| Data transfer | load word | `lw $s1,20($s2)` | `$s1 = Memory[$s2 + 20]` | Word from memory to register |
| | store word | `sw $s1,20($s2)` | `Memory[$s2 + 20] = $s1` | Word from register to memory |
| 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 $s1,$s2,20` | `$s1 = $s2 & 20` | Bit-by-bit AND reg with constant |
| | or immediate | `ori $s1,$s2,20` | `$s1 = $s2 \| 20` | 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 $s1,$s2,20` | if (`$s2 < 20`) `$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 |

# MIPS machine language

**MIPS machine language**

| Name | Format | Example | | | | | | Comments |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | 18 | 19 | 17 | 0 | 32 | add $s1,$s2,$s3 |
| sub | R | 0 | 18 | 19 | 17 | 0 | 34 | sub $s1,$s2,$s3 |
| lw | I | 35 | 18 | 17 | 100 | | | lw $s1,100($s2) |
| sw | I | 43 | 18 | 17 | 100 | | | sw $s1,100($s2) |
| and | R | 0 | 18 | 19 | 17 | 0 | 36 | and $s1,$s2,$s3 |
| or | R | 0 | 18 | 19 | 17 | 0 | 37 | or $s1,$s2,$s3 |
| nor | R | 0 | 18 | 19 | 17 | 0 | 39 | nor $s1,$s2,$s3 |
| andi | I | 12 | 18 | 17 | 100 | | | andi $s1,$s2,100 |
| ori | I | 13 | 18 | 17 | 100 | | | ori $s1,$s2,100 |
| sll | R | 0 | 0 | 18 | 17 | 10 | 0 | sll $s1,$s2,10 |
| srl | R | 0 | 0 | 18 | 17 | 10 | 2 | srl $s1,$s2,10 |
| beq | I | 4 | 17 | 18 | 25 | | | beq $s1,$s2,100 |
| bne | I | 5 | 17 | 18 | 25 | | | bne $s1,$s2,100 |
| slt | R | 0 | 18 | 19 | 17 | 0 | 42 | slt $s1,$s2,$s3 |
| j | J | 2 | 2500 | | | | | j 10000 (see Section 2.9) |
| Field size | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | All MIPS instructions 32 bits |
| R-format | R | op | rs | rt | rd | shamt | funct | Arithmetic instruction format |
| I-format | I | op | rs | rt | address | | | Data transfer, branch format |