# COMP2611 COMPUTER ORGANIZATION
# INSTRUCTIONS: LANGUAGE OF THE COMPUTER

# Major Goal

**To command a computer's hardware, you must speak its language**

- The "words" of a machine language are called **instructions**
- Its "vocabulary" is called an **instruction set**
- In the form written by the programmer → **assembly language**
- In the form the computer can understand → **machine language**
- To learn the design principles for **instruction set architecture (ISA)**

# Instruction Set Architecture (ISA)

- **instruction set architecture (ISA), is the part of the processor that is visible to the programmer or compiler writer**
  - the native data types,
  - instructions,
  - registers,
  - addressing modes,
  - memory architecture,
  - interrupt and exception handling,
  - and external I/O.

- **An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor**

# The MIPS Instruction Set

- **MIPS (Microprocessor without Interlocked Pipeline Stages)**
  - An example of RISC (Reduced Instruction Set Computer) design
  - Utilizes a small, highly-optimized set of instructions
  - Simple, clean MIPS processor whose architecture is easy to learn and understand
  - RISC ISAs now have large share of embedded core market

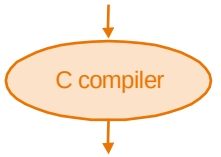
- **MIPS Reference Data green card**

# ISA vs. Assembly Language

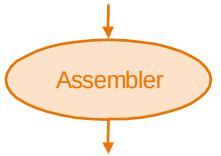
## ISA vs. Assembly Language

- "ISA" standardizes a public interface to a processor that should be used as the basis to write programs
- Assembly Language is a term for a programming language.
- Ideally for each ISA there is an Assembly Language,
- But it is not all that uncommon that more than one or subtle variations can exist between Assembly Languages for a specific ISA.
- The Assembly Language is essentially defined by the Assembler

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```




```
swap:
    muli $2, $5,4
    add  $2, $4,$2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```




```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

# Why ISA?

- **Imagine that processors do not have an ISA**
  - It means the vocabulary of hardware can change from time to time
  - No guarantee of how the instructions will look like in the next product
- **From programmers' point of view**
  - Potentially **need to re-program** every time we upgrade our computer. It would be a nightmare!
- **From system designers' point of view:**
  - Hardware improvement (for performance, or power efficiency, etc) may lead to **incompatibility** with existing applications
- **With ISA abstraction, all these problem are resolved**
  - All software developers have to do is conforming to machine's ISA
  - No need to worry about how hardware implements the instructions
  - All system designers have to do is making sure new processor implementation backward compatible to ISA's definition, instead of existing applications
- **Therefore, it is critical to have a good ISA design!**

# Design Principles for (RISC) ISA

- Simplicity favors regularity
- Smaller is faster
- Make common case fast
- Good design demands good compromises

# MIPS INSTRUCTIONS FOR DATA PROCESSING

# Arithmetic Operations

- Every computer must be able to perform arithmetic operations

- Add and subtract, three operands
- Two sources and one destination

- Examples

```
add a, b, c
```

  - It is equivalent to a = b + c in C++

```
sub x, y, z
```

  - It is equivalent to x = y - z in C++

# Arithmetic Example

- C++ code:

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

- Compiled MIPS code:

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

Notes:

- Each line of code contain at most **one** instruction
- Words to the right of # symbol are **comments** for the human reader
- **Comments** are entirely ignored by the computer

# MIPS Design Principle #1

**Design Principle #1**

**Simplicity favors regularity**

- Each instruction has a **fixed number** of operands in MIPS
- Intel architecture supports a variable number of operands

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

# Register Operands

**MIPS data processing instructions operate on registers**

- **Registers are fast temporary storage inside the processor used to hold variables.**
- **Variable vs. Register**
  - Variable is a logical storage, # of variables can be unlimited
  - Register is a physical storage, # of registers is limited

# MIPS Registers

- **MIPS has 32 general purpose registers, each is of 32 bits in size**
  - Use for frequently accessed data (32-bit data called a “word”)
  - Numbered 0 to 31
  - Saved registers **`$s0`**, **`$s1`**, … , **`$s7`** (or **`$16`** - **`$23`**): usually used to correspond to variables in a high-level program
  - Temporary registers **`$t0`**, **`$t1`**, … , **`$t7`** (or **`$8`** - **`$15`**): additional registers for variables, temporary data
  - **`$zero`** (or **`$0`**): read-only, holding a constant value 0

# Register Operand Example

- C++ code: `f = (g + h) - (i + j);`
- Using registers `$s0`, `$s1`, `$s2`, `$s3`, `$s4`, to hold variables f, g, h, i, j

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

# How Many Registers Do We Need?

- **If the number of registers is**
  - Too few: not enough to hold large number of variables in a program
  - Too many: more complicated processor design, increased clock cycle time, obstacle to improving performance
- **The computer architect should strike a good balance**
  - providing enough number of registers, and
  - keeping the clock cycle short

# MIPS Design Principle #2

| Design Principle #2 |
| --- |
| Smaller is faster |

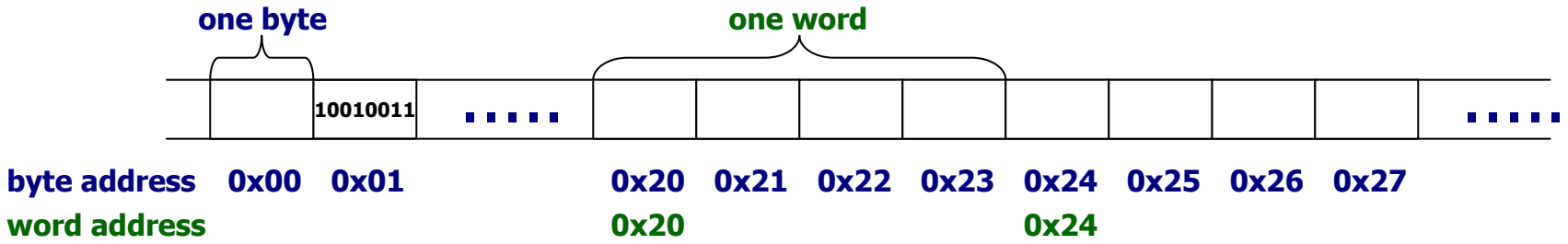
- Having a small enough number of registers leads to a faster processor
- Larger number of registers, longer electronic signals must travel

# Data Transfer in MIPS

- **Main memory provides large storage for millions of data elements**
  - Large composite data like arrays, structures and dynamic data have to be kept in the memory
- **MIPS' design disallows values stored in memory to be manipulated directly**
  - Data must be transferred from memory to a register before manipulation and the results are stored back to memory
- **We need data transfer instructions**
- **load moves data from memory to a register, e.g. lw (load a word)**
- **store moves data from a register to memory, e.g. sw (store a word)**

# Addressing Memory Location

- **Memory** is a consecutive arrangement of storage locations

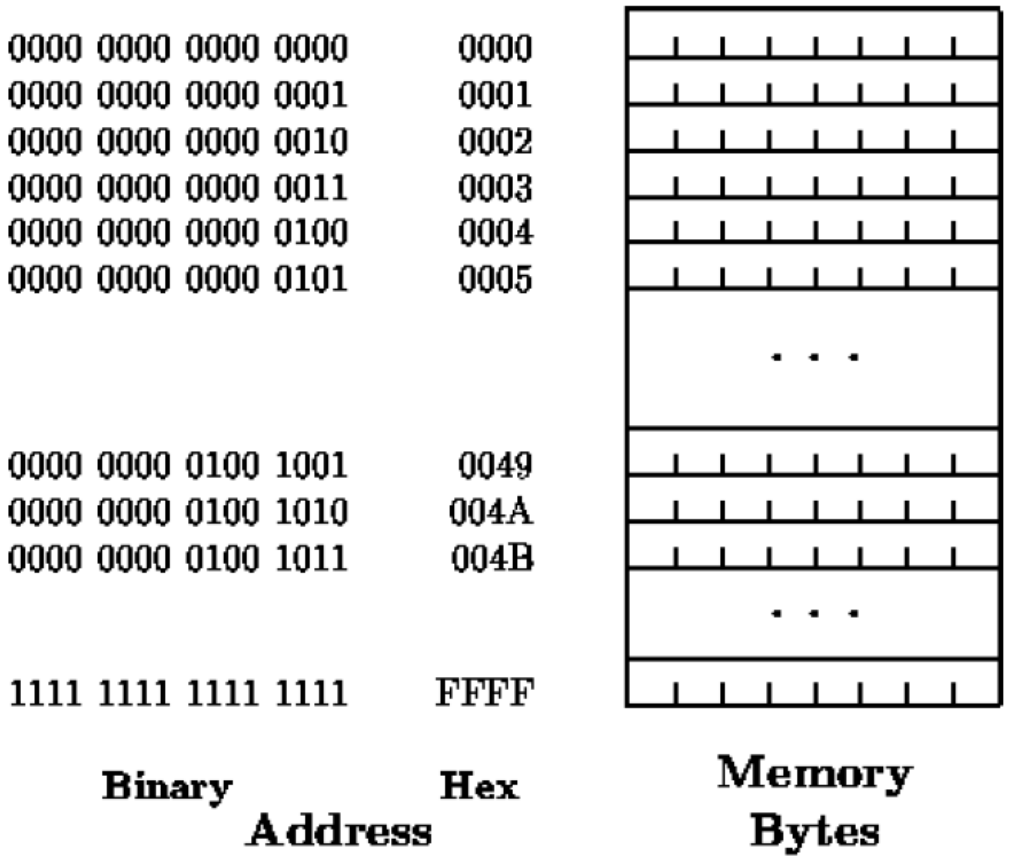



- **Each memory location is indexed by an address**
  - Most architectures address individual bytes
  - Addresses of sequential 8-bit bytes differ by 1
  - Addresses of sequential 32-bit words differ by 4

- **To specify the address of the memory location of any array element in assembly language, we need two parts:**
  - **Base address**: starting address of an array
  - **Offset**: distance of target location from starting address. A constant that can be either positive or negative

# Address Space Example

- A memory with 16-bit address
- Max memory capacity is $2^{16}$ bytes



# Memory Operand

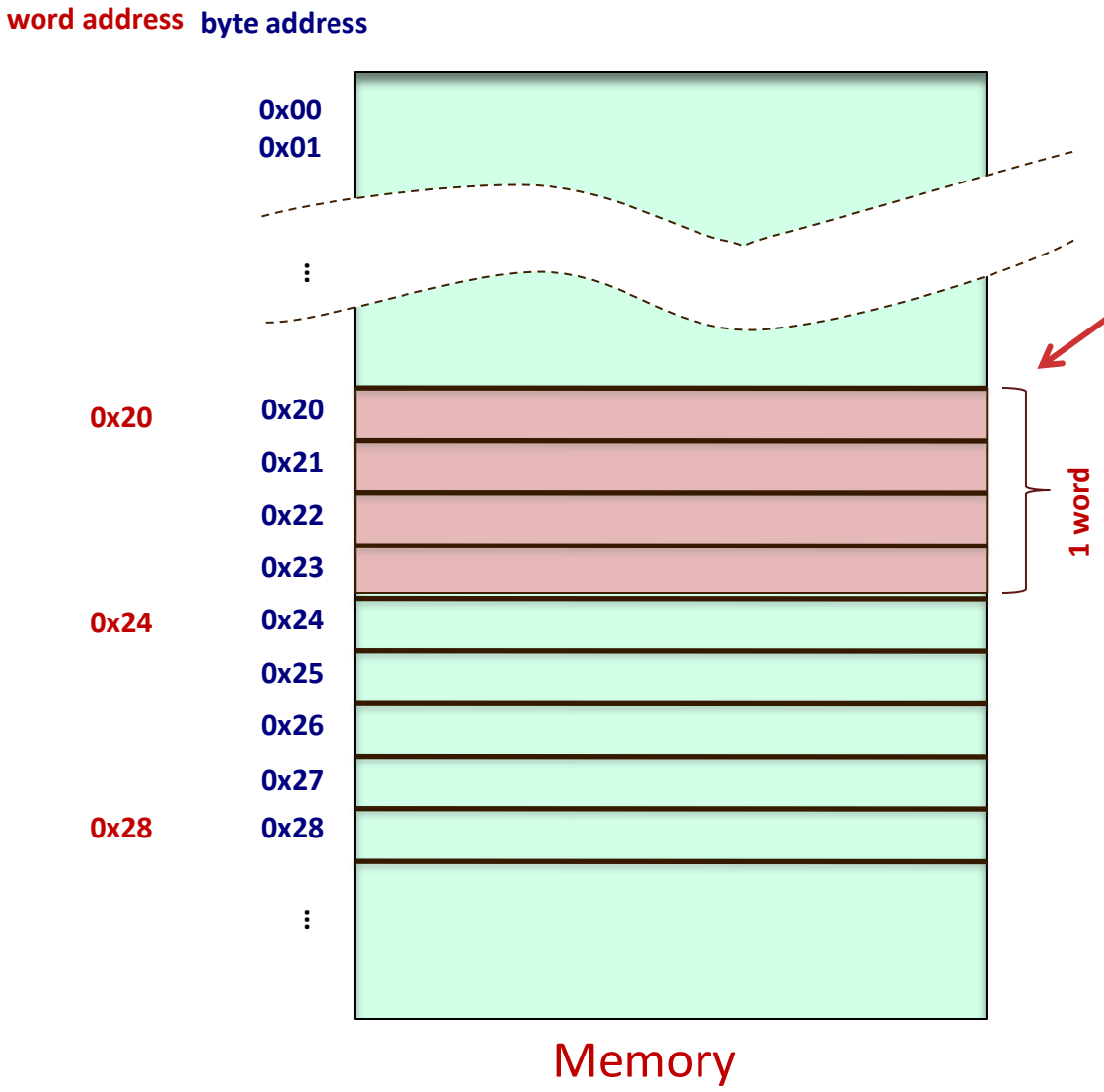
- **`A` is an array of 100 words, perform `A[12] = h + A[8];` in MIPS**
  - Each element in `A` takes 4 bytes to store
  - `$s0` stores the starting location of array `A` (i.e., address of A[0])
  - `$s1` stores the value of `h`

- **Answer:**

```
# temp reg $t0 gets A[8]
lw  $t0, 32($s0)          # address of A[8] : $s0 + 32
# $t0 = h + A[8]
add $t0, $s1, $t0
# stores h + A[8] to A[12]
sw  $t0, 48($s0)
```

- **Remark: `$s0` is used as a base register, "32" and "48" are offsets**

# Endianness (byte order)

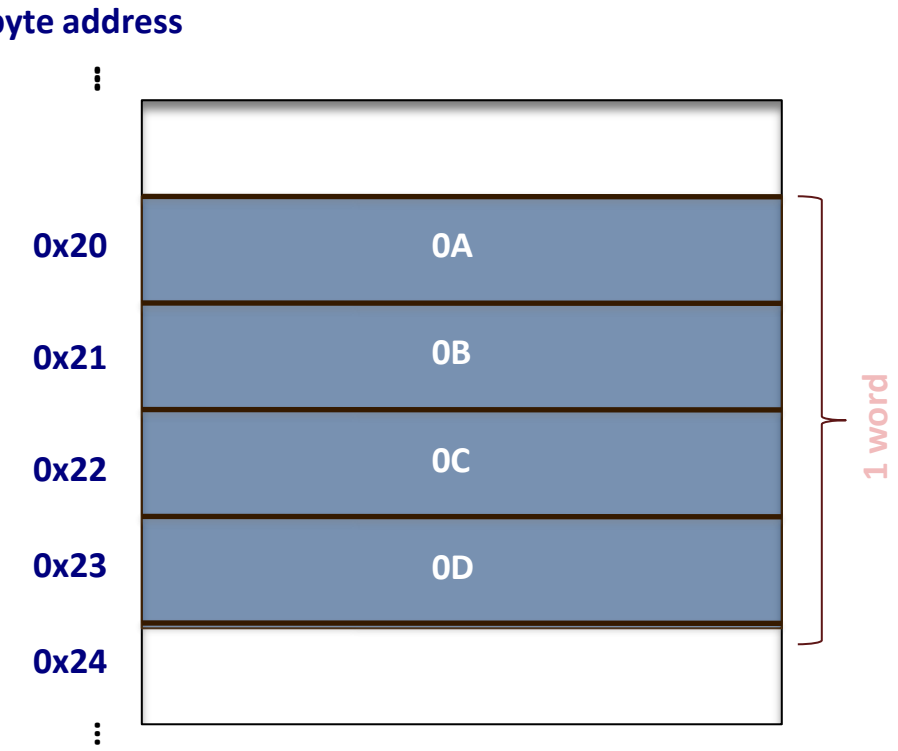



- In which order is the word **0A0B0C0D** stored in Memory?
- **Endianness** is the order of the bytes comprising a digital word in computer memory.
- It also describes the order of byte transmission over a digital link.

# Big-Endian vs. Little Endian

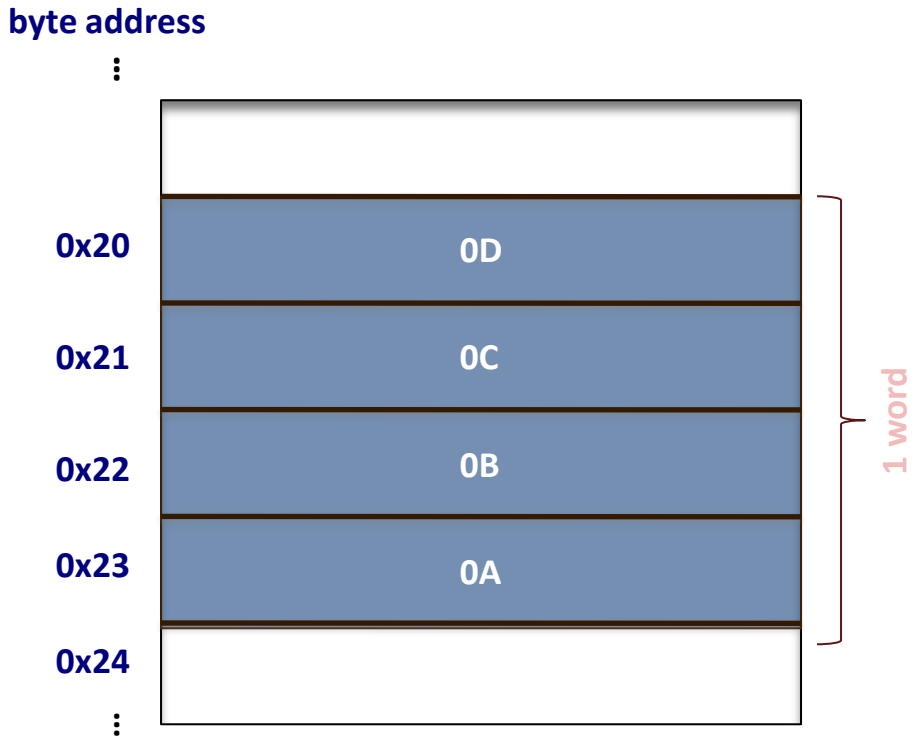
## Big-Endian

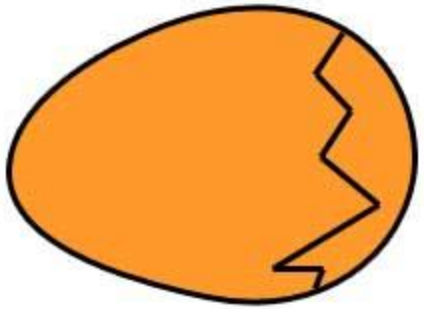
end of the word matches big addresses



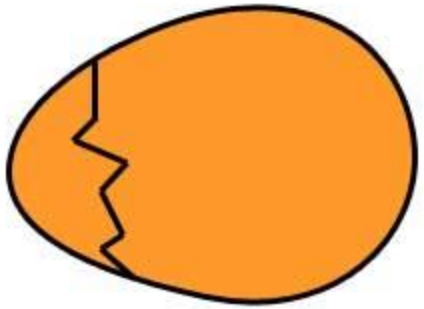

## Little-Endian

end of the word matches little addresses





BIG ENDIAN - The way people always broke their eggs in the Lilliput land



LITTLE ENDIAN - The way the king then ordered the people to break their eggs

# Immediate Operands

- To do the expression register1 = register2 +/- constant
- addi means add immediate (constant)
  - Constant part is always the last operand of this instruction

```
addi $t0, $s1, 8      # $t0 = $s1 + 8
addi $t0, $t0, -1     # $t0 = $t0 - 1 (no subi)
addi $t0, $t0, -1     # $t0 = $t0 - 1
```

- Constant Zero
  - MIPS register 0 ($zero) is the constant 0.
  - Read-only, cannot be overwritten

```
add $t2, $s1, $zero   # move between registers
addi $t0, $zero, 5    # initialize to constant
```

# Memory vs. Register vs. Constant

- **Major difference is the instruction's execution time**
  - **Memory** is outside the processor; far from the processing unit
    - Memory operand takes a long time to load/store
  - **Register** is inside the processor; close to the processing unit
    - Register operand takes a short time to get to the value
  - **Constant** already encoded in the instruction
    - Constant operand value is immediately available

**A program is a mixture of these three types of operations**

# Design Principle #3

**Design Principle #3**

**Make the common case fast**

- **Constant operands occur frequently!**
- **Encode constants inside arithmetic instructions,**
  - They are much faster than if constants were loaded from memory

# Logical Operations – and, or, nor

- `and`, `or`, `nor`: bit-by-bit operation

| bit 1 | bit 2 | and | or | nor |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

- **Example**

given register $t1 and $t2

$t1 = 0011 1100 0000 0000$_2$
$t2 = 0000 1101 0000 0000$_2$

```
and  $t0, $t1, $t2
or   $t0, $t1, $t2
nor  $t0, $t1, $zero
```

$t0 = 0000 1100 0000 0000$_2$
$t0 = 0011 1101 0000 0000$_2$
$t0 = 1100 0011 1111 1111$_2$

- `andi` : and with an immediate operand
- `ori` : or with an immediate operand

# Logical Operations – shift

**shift**

- Move all the bits in a word to the left or right
- Filling the emptied bits with 0s
- Example

`0000 0000 0000 0000 0000 0000 0000 1001 = 9`$_{10}$

shift left (<<) by 4

`0000 0000 0000 0000 0000 0000 1001 0000 = 144`$_{10}$

shifting left by k bits gives the same result as multiplying by $2^k$

MIPS shift instructions:

`sll` ('shift left logical'), `srl` ('shift right logical')

- Example

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

# MIPS Instructions Learned So Far

## Three types of instructions: arithmetic, logical, data transfer

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $s1, $s2, $s3 | $s1 = $s2 + $s3 | 3 operands |
| | subtract | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 | 3 operands |
| | add immediate | addi $s1, $s2, 100 | $s1 = $s2 + 100 | 2 operands, 1 constant |
| Logical | and | and $s1, $s2, $s3 | $s1 = $s2 & $s3 | 3 operands, bit-by-bit and |
| | or | or $s1, $s2, $s3 | $s1 = $s2 \| $s3 | 3 operands, bit-by-bit or |
| | nor | nor $s1, $s2, $s3 | $s1 = ~($s2 \| $s3) | 3 operands, bit-by-bit or |
| | and immediate | andi $s1, $s2, 100 | $s1 = $s2 & 100 | 2 operands, 1 constant, bit-by-bit |
| | or immediate | ori $s1, $s2, 100 | $s1 = $s2 \| 100 | 2 operands, 1 constant, bit-by-bit |
| | 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 $s1, 100($s2) | $s1 = memory[$2+100] | Word from mem to reg |
| | store word | sw $s1, 100($s2) | Memory[$2+100] = $1 | Word from reg to mem |

# MIPS PROGRAM

# Writing a Program in MIPS

- How can I declare an array / a variable in a MIPS program?
- How can I obtain the starting address of an array?
- How does the program run?

# Sample Program

```
##############
# - We need to declare "variables" & "Arrays" used in the program in a data segment.
# - The compiler recognize .data as the beginning of data segment
.data                          # assembler directive
h: .word 1 2 3 4               # h is an array of size 4, each element is a word (32 bit)
s: .word 5

# The 3 lines below let the system know the program begins here
.text
.globl __start
__start:

# Write your program code here
la $s0, h                      # Obtain starting address of array h,  s0 = x (a constant)
lw $s1,8($s0)                  # $s1 = content in memory address x + 8 = 3 = h[2]

la $s2, s
lw $s3, -12($s2)               # $s2 = content of address of s -12 = ?
sub $s3, $s3, $s1              # Q1: Guess what is the value of $s3 ?
sw $s3, 0($s0)                 # Q2: How are the values of array h changed ?
```

# How Does It Work?

- When the program is about to run, the data (variables, arrays) declared will be fed into memory consecutively

```
h: .word 1 2 3 4 # h is an array of size 4
s: .word 5
```

| | Address | Value | Array element |
|---|---|---|---|
| h → | X -th byte | 1 | h[0] |
| | X+4 -th byte | 2 | h[1] |
| | X+8 -th byte | 3 | h[2] |
| | X+12 -th byte | 4 | h[3] |
| s → | X+16 -th byte | 5 | |

- h & s are called "labels", they can be viewed as the bookmarks of the program
- When la $s0, h is executed, the address (in byte) referenced by h will be assigned to register $s0
- e.g. if X = 10000, then $s0 = 10000. This means the values of the array h store between the $10000^{th}$ and $10015^{th}$ byte of the memory

# Assembler Directive

- Directives are commands that are part of the assembler syntax but are not related to the processor instruction set.
- MIPS assembler directives begin with a period (.)

| | |
|---|---|
| `.data` | The following data items should be stored in the data segment. |
| `.align n` | Align the next datum on a $2^n$ byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives. |
| `.ascii str` | Store the string in memory, but do not null-terminate it. |
| `.asciiz str` | Store the string in memory and null-terminate it. |
| `.byte b1,..., bn` | Store the n values in successive bytes of memory. |
| `.word w1,..., wn` | Store the n 32-bit quantities in successive memory words. |
| `.double d1,..., dn` | Store the n floating point double precision numbers in successive memory locations. |
| `.text <addr>` | The next items are put in the user text segment. |
| `.globl sym` | Declare that symbol sym is global and can be referenced from other files. |

# MIPS INSTRUCTIONS FOR CONTROL FLOW

# Instructions for Making Decisions

**What distinguishes a computer from a simple calculator is its ability to make decisions based on input data or values obtained during the computation**

- **In high-level programming languages, decision-making instruction:**
  - **if** statement

- **In MIPS, decision-making instructions (or conditional branches):**
  - **`beq`** ('branch if equal'):
    - e.g. **`beq reg1, reg2, L1`**
    - go to statement labeled **`L1`** if **`reg1`** and **`reg2`** have the same value
  - **`bne`** ('branch if not equal' ):
    - e.g. **`bne reg1, reg2, L1`**
    - go to statement labeled **`L1`** if **`reg1`** and **`reg2`** do not have the same value

# Example

- In the following C++ code segment, `f`, `g`, `h`, `i`, and `j` are variables:

```
        if (i == j) goto L1;
        f = g + h;
  L1:   f = f - i;
```

Assuming that the five variables `f` through `j` correspond to five registers `$s0` through `$s4`, what is the compiled MIPS code?

- Answer:

```
      beq $s3, $s4, L1     # go to L1 if i==j
      add $s0, $s1, $s2    # f = g + h (skipped if i==j)
L1:   sub $s0, $s0, $s3    # f = f - i (always executed)
```

Notes:

`L1` corresponds to the memory address of the `sub` instruction.

# More on Branches

- Compilers frequently create branches and labels where they do not appear in the programming language.
- Alternative no goto version for previous example

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

- Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is one of the reasons why coding is faster at that level.

- Besides conditional branches, we also have unconditional jumps:
  - `j` ('jump'):
    - e.g. `j L1`
    - always go to statement labeled `L1`

# Example

- Assume, as before, that the five variables `f` through `j` correspond to registers `$s0` through `$s4`. What is the compiled MIPS code for this?

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

- Answer:

```
        bne $s3, $s4, ElseIf    #if(i!=j) goto Elseif
        add $s0, $s1, $s2
        j   Exit
ElseIf: bne $s3, $s1, Else      #if(i!=j) goto Else
        sub $s0, $s1, $s2
        j   Exit
Else:   add $s0, $s1, $s4
Exit:
```

# Example (Cont'd)

- Another Solution:

```
                    beq $s3, $s4, if_match
                    beq $s3, $s1, elseif_match
                    j   else_match
if_match:           add $s0, $s1, $s2
                    j   exit
elseif_match:       sub $s0, $s1, $s2
                    j   exit
else_match:         add $s0, $s1, $s4
exit:
```

- Although this solution is longer, it is more similar to C++ version & looks closer to a switch-case statement
- Could be easier to debug if you need to check for more conditions

# Loops

- **Decisions are important both for**
  - choosing between two alternatives–found in **if** statement
  - iterating a computation–found in **loops**

- **In loops, decisions are needed to determine when to stop looping**

- **Commonly used loop constructs in high-level programming languages**
  - **while, do-while**
  - **for**

# Example

- Here is a traditional loop in C:

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

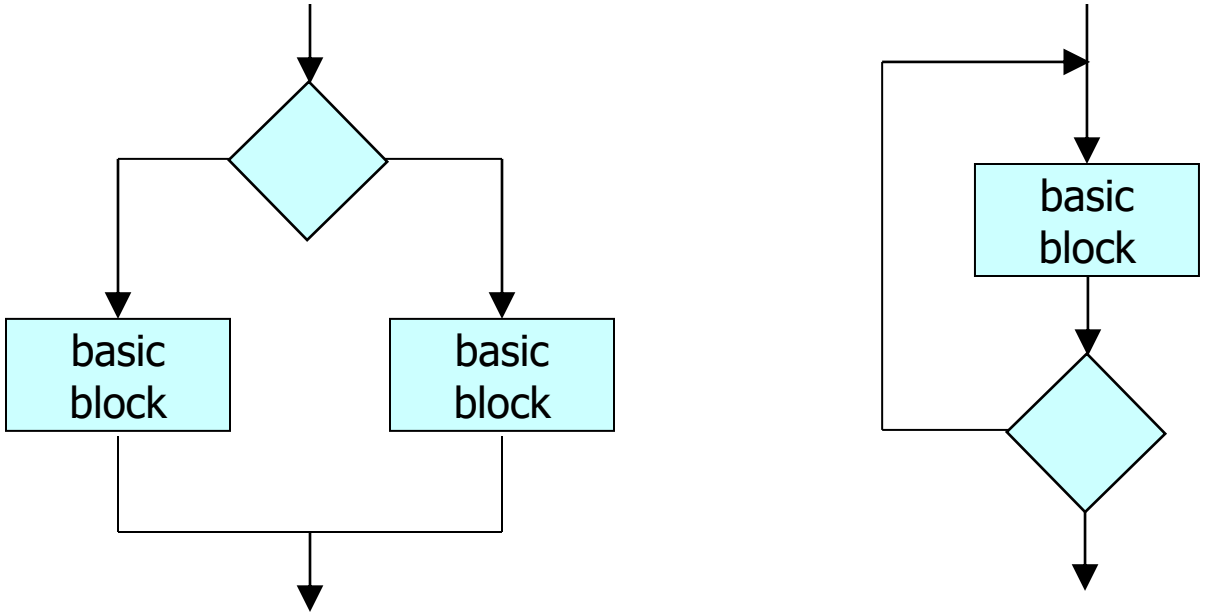
Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C Segment?

- Answer:

```
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
Exit:
```

# Basic Blocks

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




- One of the first early phases of compilation is breaking the program into basic blocks
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

# Example of Basic Blocks

- #1

```
Loop:  sll $t1, $s3, 2
       add $t1, $t1, $s6
       lw  $t0, 0($t1)
       bne $t0, $s5, Exit

       add $s3, $s3, 1
       j   Loop
Exit:
```

- #2

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

# 'Less Than' Test

- **Besides testing for equality or inequality, it is often useful to see if a variable is less than another variable.**
  - e.g., exit from a loop when the array index is less than a variable

- **`slt` ('set on less than'):**
  - `slt  reg1, reg2, reg3`
  - register **`reg1`** is set to 1 if the value in **`reg2`** is less than the value in **`reg3`**; otherwise, register **`reg1`** is set to 0

- **`slti` ('set on less than immediate')**
  - `slti  $t0, $s2, 10`  `# $t0=1 if $s2 < 10`

# Branch Instruction Design

- **MIPS compilers use `beq`, `bne`, `slt`, `slti` and the fixed value of 0 (always available by reading register `$zero`) to create all comparison operations:**
  - **equal, not equal**
  - **less than, less than or equal**
  - **greater than, greater than or equal**

- **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

# Example

- Give the MIPS code that tests if variable `a` (corresponding to register `$s0`) is less than variable `b` (register `$s1`) and then branch to label `L` if the condition holds.

- Answer:

```
slt $t0, $s0, $s1   # $t0 gets 1 if $s0 < $s1
bne $t0, $zero, L   # go to L if $t0 != 0
```

- Remark:
  - Instead of providing a separate 'branch if less than' instruction which will complicate the instruction set, the MIPS architecture chooses to do this operation using two faster MIPS instructions – similar for other conditional branches.

# Branch with Zero

Branch on greater than or equal to zero

- bgez $s, label # if ($s >= 0)

Branch on greater than zero

- bgtz $s, label # if ($s > 0)

Branch on less than or equal to zero

- blez $s, label # if ($s <= 0)

Branch on less than zero

- bltz $s, label # if ($s < 0)

# ‘Jump Register’ Instruction

- **Another unconditional jump instruction:**
  - `jr` (‘jump register’):
    - e.g. `jr reg`
    - jump to address specified in register `reg`

- **It is usually used for procedure call and case/switch statement**

# Exercise

- Consider the program below: What are the values stored in Array after the program is executed? (It depends on where 'jr' goes)

```
.data
Array: .word 4 8 12 16 20

.text
.globl __start
__start:

la $t0, Array
lw $t1, 4($t0)
lw $t2, 8($t0)
la $s0, Label
add $s0, $s0, $t1
jr $s0

Label:
add $t1, $t1, $t1
add $t1, $t2, $t2
sw $t1, 12($t0)
```

i) What are the values of t1 & t2 ?

ii) If the instruction add $t1, $t1, $t1 stores at address 10000, what is the value of s0 & what jr $s0 does ?

Assume this add instruction stores at address 10000

iii) Where is this instruction stored ?

# Exercise – Solution

```
Array: .word 4 8 12 16 20

la $t0, Array
lw $t1, 4($t0)
lw $t2, 8($t0)
```

i) So, t1 = 8, t2 = 12

```
la $s0, Label
add $s0, $s0, $t1
jr $s0
```

Array →

| Address | Value | Array element |
|---|---|---|
| t0 | 4 | Array1[0] |
| t0+4 | 8 | Array1[1] |
| t0+8 | 12 | Array1[2] |
| t0+12 | 16 | Array1[3] |
| t0+16 | 20 | Array1[4] |

ii) s0 = 10000 after `la $s0, Label1` is executed.
Hence, the next instruction to be run after `jr $s0` is stored at 10000 + 8 = 10008$^{th}$ byte of the memory

```
Label:
add $t1, $t1, $t1
add $t1, $t2, $t2
sw $t1, 12($t0)
```

Label →

| Address | Instruction |
|---|---|
| 10000 | add $t1, $t1, $t1 |
| 10004 | add $t1, $t1, $t2 |
| 10008 | sw $t1, 12($t0) |

iii) All MIPS instructions are fixed as 4 bytes long. So, `sw $t1, 12($t0)` should be executed after `jr $s0` (2 instructions skipped). Array1[3] = t1 = 8 at the end

# IMPLEMENT MIPS INSTRUCTIONS

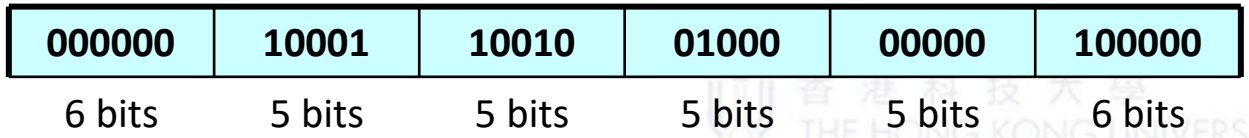
# Representing Instructions in the Computer

- Computer "see" the instructions as machine language or machine code; encoded in binary
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Broken up into a number of fields
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Example:

```
add $t0, $s1, $s2
```

  - Decimal representation:


  - Binary representation:



# Instruction Formats

- **All MIPS instructions have fixed length, but different instructions may have different formats**

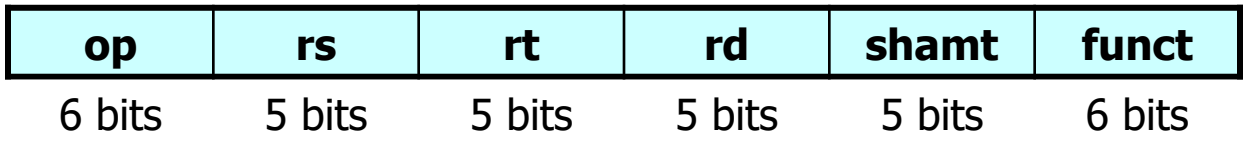
- **Three types of instruction formats in MIPS**
  - R-type or R-format for register
  - I-type or I-format for immediate
  - J-type or J-format for jump

- **Each format is assigned a distinct set of values for the 1st field**
  - Hardware can interpret the instruction just by examining this field
  - This field is so-called opcode

- **Using multiple formats complicates hardware design, but complexity can be reduced by keeping the formats similar (will see in next slides)**

# MIPS R-format Instructions

## R-type or R-format




- **Instruction fields: (6 fields)**
  - **op**: basic operation of instruction, traditionally called **opcode**
  - **rs**: first register source operand
  - **rt**: second register source operand
  - **rd**: register destination operand, which gets result of operation
  - **shamt**: shift amount (number of positions to shift)
  - **funct**: function code (extends opcode)

# R-Format Example

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

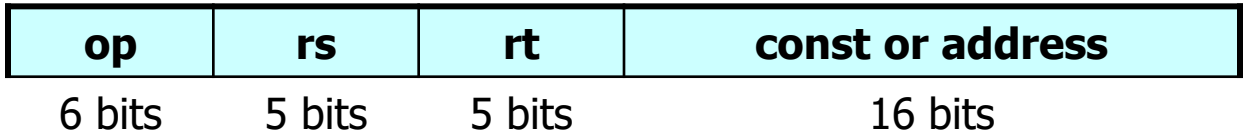
```
add $t0, $s1, $s2
```

| op | $s1 | $s2 | $t0 | 0 | add |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 8 | 0 | 32 |
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

$00000010001100100100000000100000_2 = 02324020_{16}$

# MIPS I-format Instructions

## I-type or I-format




- **Immediate arithmetic and load/store instructions**
- **Instruction fields:**
  - **op**: as before
  - **rs**: base register
  - **rt**: register source operand (for **`sw`**) *or* destination operand (for **`lw`**)
  - **constant**: $-2^{15}$ to $+2^{15} - 1$
  - **address**: offset added to base address in rs

# Design Principle #4

**Design Principle #4**

**Good design demands good compromises**

- **Ways to encode instructions:**
  - **Variable** length *or* **fixed** length
- **How to choose?**
  - Use **variable length** to optimize code size (i.e. to save storage)
  - Use **fixed length** to optimize performance and reduce complexity
- **Compromise MIPS chose is to keep all instructions the same length**
  - Why?
  - Hardware to fetch & decode an instruction is simpler and faster

# MIPS Instruction Encoding

| Instruction | Type | op | rs | rt | rd | shamt | funct | const/address |
|---|---|---|---|---|---|---|---|---|
| `add` | R | 0 | reg | reg | reg | 0 | $32_{10}$ | - |
| `sub` | R | 0 | reg | reg | reg | 0 | $34_{10}$ | - |
| `and` | R | 0 | reg | reg | reg | 0 | $36_{10}$ | - |
| `or` | R | 0 | reg | reg | reg | 0 | $37_{10}$ | - |
| `sll` | R | 0 | 0 | reg | reg | constant | 0 | - |
| `srl` | R | 0 | 0 | reg | reg | constant | $2_{10}$ | - |
| `addi` | I | $8_{10}$ | reg | reg | - | - | | constant |
| `lw` | I | $35_{10}$ | reg | reg | - | - | | address |
| `sw` | I | $43_{10}$ | reg | reg | - | - | | address |

- reg: a register number between 0 and 31
- constant/address: a constant or a 16-bit address (offset)
- Special opcode 0: e.g. both `add` and `sub` have the same value in op field but different values (32 for `add`; 34 for `sub`) in the funct field.

# Instruction Format Example

## Description:

- Suppose **`$t1`** stores the base address of array **`A`** and **`$s2`** is associated with **`h`**, the following C assignment statement

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

is compiled into

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

## Problem to solve:

- What is the MIPS machine code for these three instructions?

# Answer

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

- Decimal representation:

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

- Binary representation:

| | | | | | |
|---|---|---|---|---|---|
| 100011 | 01001 | 01000 | 0000 0100 1011 0000 | | |
| 000000 | 10010 | 01000 | 01000 | 00000 | 100000 |
| 101011 | 01001 | 01000 | 0000 0100 1011 0000 | | |

# Other Important MIPS Instructions

**~~Unsigned~~ arithmetic**

- `addu $rd, $rs, $rt` `#addition ignoring overflow`
- `subu $rd, $rs, $rt`

**Load/store a byte**

- `lb $rt, offset($rs)` `#sign extend to 32 bits in $rt`
- `lbu $rt, offset($rs)` `#zero extend to 32 bits in $rt`
- `sb $rt, offset($rs)` `#store rightmost byte in $rt`

**Logical operation**

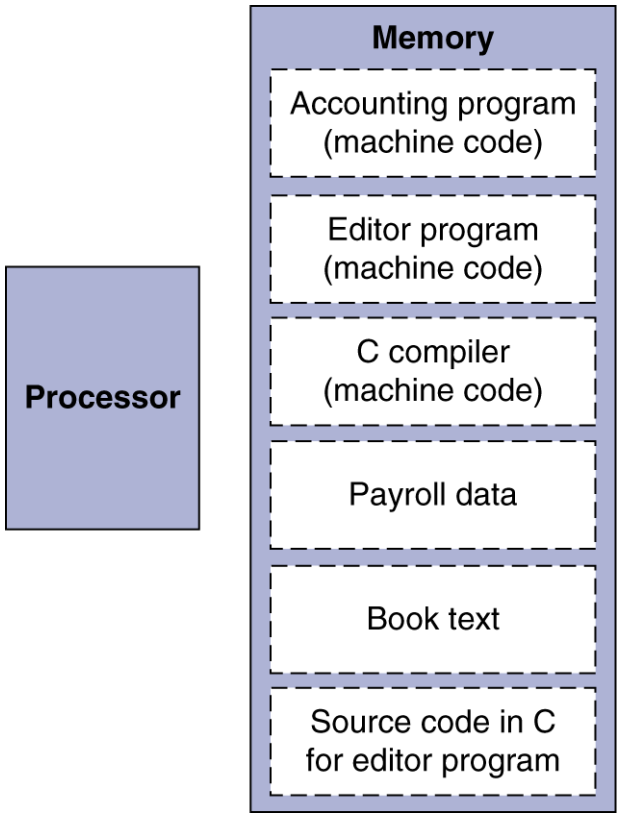
- `nor $rd, $rs, $rt` `# $rd = $rs nor $rt`
- Useful to invert bits in a word, a NOR b == NOT ( a OR b ), nor $t0, $t1, $zero does NOT operation

**Shift left/right logical variable**

- `sllv $rd, $rt, $rs` `# $rd = $rt << $rs`
- `srlv $rd, $rt, $rs` `# $rd = $rt >> $rs`

# The Stored-Program Concept

## The BIG Picture




- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

# DEALING WITH “PROCEDURE”

# Supporting Procedures in Computer Hardware

- Procedures (also called subroutines) are necessary in any programming language
- They allow better structuring of programs
- Thus we need mechanisms that allow to jump to the procedure and to return from it

```
k = 0;
switch (k){
  case 0 : f = max(i,j);
           i = i + j;
           break;
  case 1:  f = max(g,h);
           i = i + j;
           break;
}
```

```
int max(int k, int l)
  if (k <= l)
       return l;
  else
       return k;
}
```

# Supporting Procedures in Computer Hardware

- **Necessary steps for executing a procedure:**
  1. Place the parameters in place where the procedure can get 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 caller can access it
  6. Return control to the point of origin, since a procedure can be called from several points in a program

# Registers 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

# Procedure Call Instructions

- **Procedure call: jump and link `jal`**
  - **`jal ProcedureLabel`**
  - Two things happen at the same time
    1. Address of following instruction put in `$ra`
    2. Jumps to target address specified by **`ProcedureLabel`**

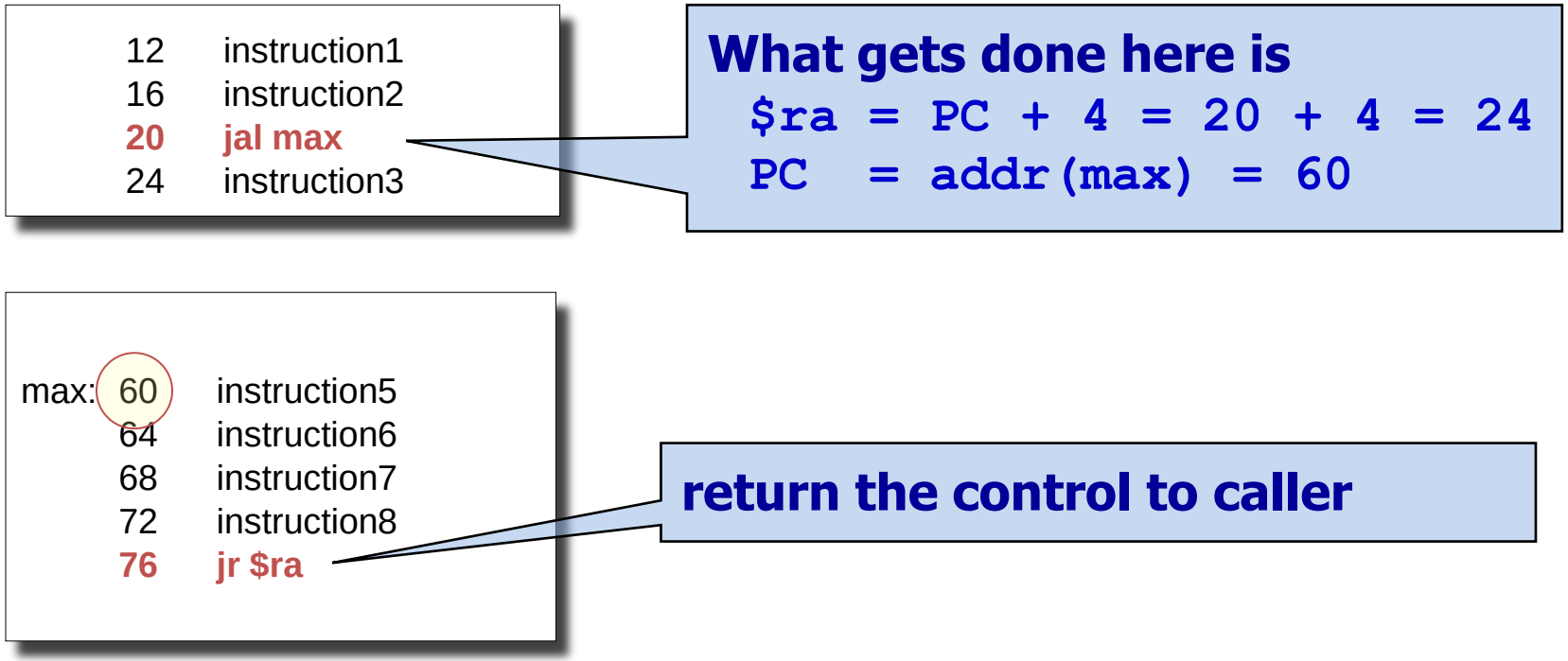
- **Procedure return: jump register `jr`**
  - **`jr $ra`**
  - Copies `$ra` to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

# 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`**

# `jal` and `jr` Example



# Problem with Nested Procedures

```
    12    instruction1
    16    instruction2
    20    jal max
    24    instruction3
```

**What gets done here is**

```
$ra = PC + 4 = 20 + 4 = 24
PC  = addr(max) = 60
```

```
max: 60    instruction5
     64    instruction6
     68    jal proc
     72    instruction8
     76    jr $ra
```

**What if we replace instruction 7 by another procedure call, say jal proc?**

```
$ra = 72
PC  = addr(proc) = 80
```

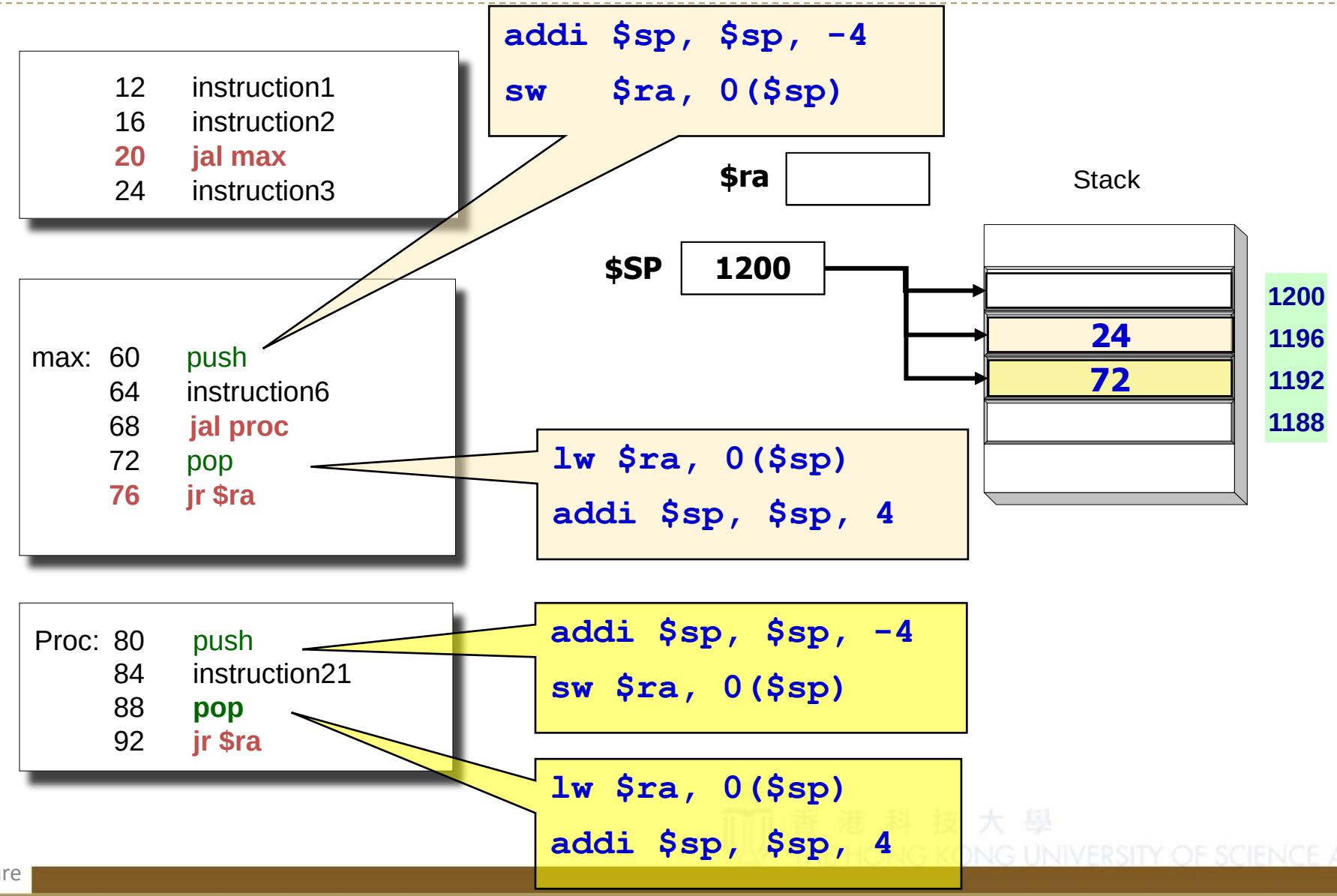
```
Proc: 80    instruction20
      84    instruction21
      88    jr $ra
```

**Oops! $ra = 72!!!!**
**Can't return to line 24**

# Stack: Supporting Procedures in MIPS

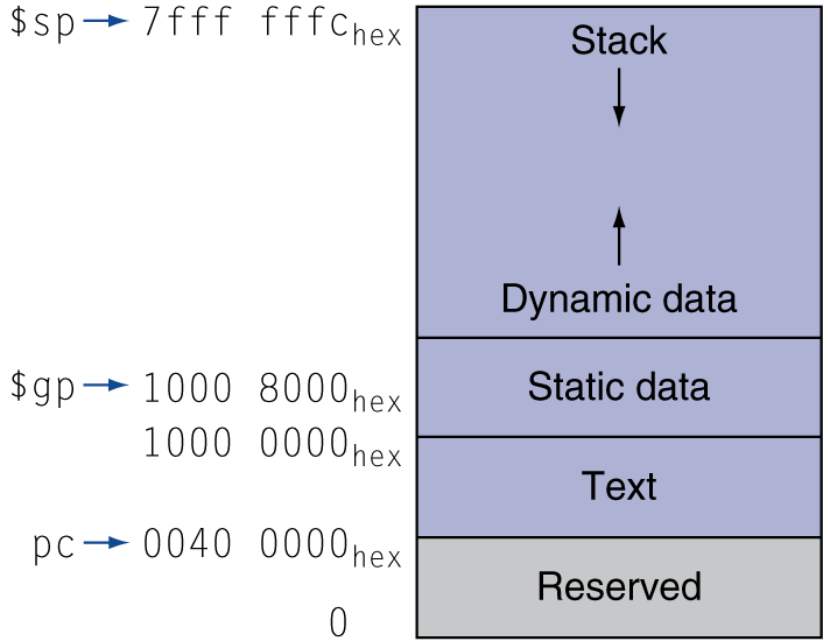
- **Since procedures are like small programs themselves, they may need to use the registers, and they may also call other procedures (nested calls)**
  - If we don't save some of the values stored in the registers, they will be wiped each time we call a new procedure
  - e.g. $ra was wiped out in previous example in max(), and we have no way to return from nested procedure calls
- **In MIPS, we need to save the registers by ourselves (some other ISAs would do it on your behalf)**
  - The perfect place for this is called a **stack**
    - a memory accessible only from the top (Last In First Out, LIFO)
    - placing things on the stack is called **push**
    - removing them is called **pop**
  - **push** and **pop** are simply **storing** and **loading** words to and from a specific location in the memory pointed to by **the stack pointer $sp** which always points to top of the stack

# Using Stack to Deal with Nested Procedure



# MIPS Memory Layout

- **Text: program code**
- **Static data: global variables**
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- **Dynamic data: heap**
  - E.g., malloc() in C, new in Java
- **Stack: automatic storage**



# Summary of MIPS Architecture Revealed So Far

- **MIPS operands:**
  - 32 registers (32 bits each)
  - $2^{30}$ memory word locations (32 bits each)

- **MIPS instructions:**
  - Arithmetic: **`add`**, **`sub`**, **`addi`**
  - Data transfer: **`lw`**, **`sw`**
  - Logical: **`and`**, **`or`**, **`nor`**, **`andi`**, **`ori`**, **`sll`**, **`srl`**
  - Conditional branch: **`beq`**, **`bne`**, **`slt`**
  - Unconditional jump: **`j`**, **`jr`**, **`jal`**

- **MIPS instruction formats:**
  - R-format, I-format, J-format (used by **`j`** and **`jal`**; to be explained later)

# MIPS Register Conventions

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| `$zero` | 0 | constant value 0 | n.a. |
| `$at` | 1 | reserved for assembler | n.a. |
| `$v0-$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 operating system kernel | n.a. |
| `$gp` | 28 | pointer to global area | yes |
| `$sp` | 29 | stack pointer | yes |
| `$fp` | 30 | frame pointer | yes |
| `$ra` | 31 | return address | yes |

Preserved on call means, the value of those registers should remain the same before and after the procedure is called

If any of those registers are modified inside the procedure, you should put them into stack before the procedure is actually executed

# Communication with People

- **Most computers use 8-bit (bytes) to represent characters**
  - ASCII: American Standard Code for Information Interchange
  - Example:

| ASCII value | Char-acter | ASCII value | Char-acter | ASCII value | Char-acter | ASCII value | Char-acter | ASCII value | Char-acter |
|---|---|---|---|---|---|---|---|---|---|
| 48 | 0 | 49 | 1 | 65 | A | 66 | B | 90 | Z |
| 97 | a | 98 | b | 32 | **Space** | 35 | # | 42 | * |

- **Notice that character "a" and "A" are assigned with different values!**

- **Operation with byte: `lb` (load byte), `sb` (store byte)**
  - Example

```
lb $t0, 0($sp)   # Read byte from source
sb $s0, 0($gp)   # Write byte to destination
```

# String Copy Example

- Give the MIPS code for procedure **`strcpy()`** in C++
- It copies string **`y`** to string **`x`** using the null byte termination convention

```
void strcpy (char x[], char y[] {
    int i=0;
    while ((x[i] = y[i])!='\0')  /* copy & test byte */
            i += 1;
}

i:                      $s0
address of array x[]:   $a0
address of array y[]:   $a1
address of y[i]:        $t1
y[i]:                   $t2
address of x[i]:        $t3
```

# Answer

```
strcpy:
        addi $sp, $sp, -4       # adjust stack for 1 item
        sw   $s0, 0($sp)        # save $s0
        add  $s0, $zero, $zero  # i = 0
Loop:
        add  $t1, $s0, $a1      # addr of y[i] in $t1
        lbu  $t2, 0($t1)        # $t2 = y[i]
        add  $t3, $s0, $a0      # addr of x[i] in $t3
        sb   $t2, 0($t3)        # x[i] = y[i]
        beq  $t2, $zero, exit   # exit loop if y[i] == 0
        addi $s0, $s0, 1        # i = i + 1
        j    Loop               # next iteration of loop
exit:
        lw   $s0, 0($sp)        # restore saved $s0
        addi $sp, $sp, 4        # pop 1 item from stack
        jr   $ra                # and return
```

# Dealing with 32-bit Immediate

- Constants are frequently short and fit into 16-bit field
- But sometimes they are bigger than 16 bits, e.g. 32-bit constant

**Problem:**

- With instruction learned so far, we cannot set registers' upper 16bits!

**Solution:**

- `lui` ("load upper immediate")
  - e.g. `lui reg, constant`
  - set the upper 16 bits of register `reg` to the 16-bit value specified in `constant`
  - **Set the lower 16 bits of register `reg` to zeros**
  - note that `constant` should not greater than $2^{16}$

# Example: Loading a 32-bit Constant

- How to load the 32-bit constant below into register `$s0`?

  0000 0000 0011 1101 0000 1001 0000 $0000_2$ (0x003D0900)

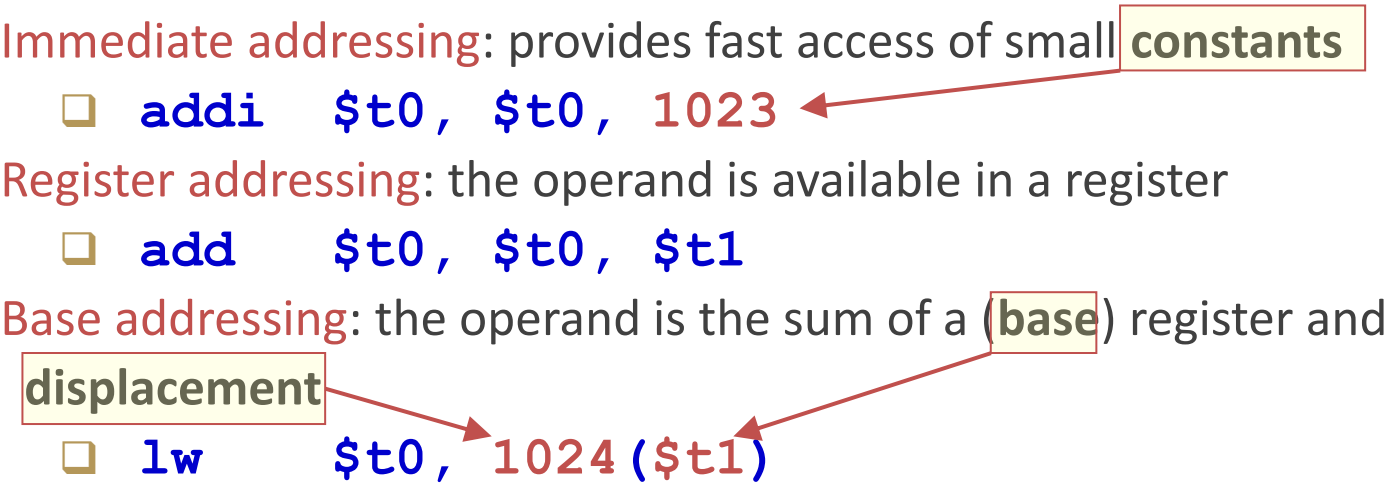
- Solution: (assuming the initial value in `$s0` is 0)

```
lui $s0, 61          # 61_10 = 0000 0000 0011 1101
# value of $s0 becomes 0000 0000 0011 1101 0000 0000 0000 0000
ori $s0, $s0, 2304   # 2304_10 = 0000 1001 0000 0000
# now, we get the value desired into the register
```

# ADDRESSING MODES

# MIPS Addressing Modes

- Addressing takes care of where to find **data** and **instruction**
- We have seen, so far three addressing modes of MIPS **to find data**:
  - Immediate addressing: provides fast access of small **constants**
    - `addi $t0, $t0, 1023`
  - Register addressing: the operand is available in a register
    - `add $t0, $t0, $t1`
  - Base addressing: the operand is the sum of a (**base**) register and a **displacement**
    - `lw $t0, 1024($t1)`



- MIPS architecture provides two more ways of addressing **to find instruction**

# Where is the next instruction?

- **Sequential execution**
  - PC holds the address of the instruction to be executed
  - The instruction is loaded to processor and starts to execute
  - PC is always updated to PC+4 while executing the instruction (assuming sequential execution of the program)
- **Conditional branch? Un-conditional jump?**
  - If the next instruction is not sequentially executed, PC will be flushed to hold the correct jump target address.

# Addressing in Conditional Branches

**We know that**

- **Conditional branch instructions (e.g., `beq`, `bne`) use I-format**
- **I-format can only specify 16-bit addresses**

**How to branch?**

- **PC-relative addressing**
  - A branch offset is added to (PC+4) to obtain address to branch to
    - Branch offset is described in number of **words**.
  - Branching within $2^{15}$ words before or after the current instruction is possible
  - This is good enough since conditional branches tend to branch to a nearby instruction

# Example: Branch Offset in Machine Language

| Address | Instruction |
|---|---|
| 40000008 | Instruction 1 |
| 4000000C | beq $zero, $s0, label |
| 40000010 | Instruction 3 |
| 40000014 | Instruction 4 |
| 40000018 | label: Instruction 5 |
| 4000001C | Instruction 6 |
| 40000020 | etc… |

Machine code to `beq` is `0x10100002`, which means 2 instructions from the next instruction

PC = `0x4000000C`

PC+4 = `0x40000010`

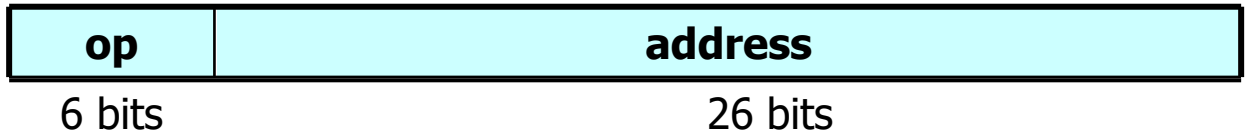
Add 4*2 = `0x00000008`

Target = `0x40000018`

| op | rs | rt | const or address |
|---|---|---|---|
| 000100 | 00000 | 10000 | 0000000000000010 |

# J-type Instruction Format

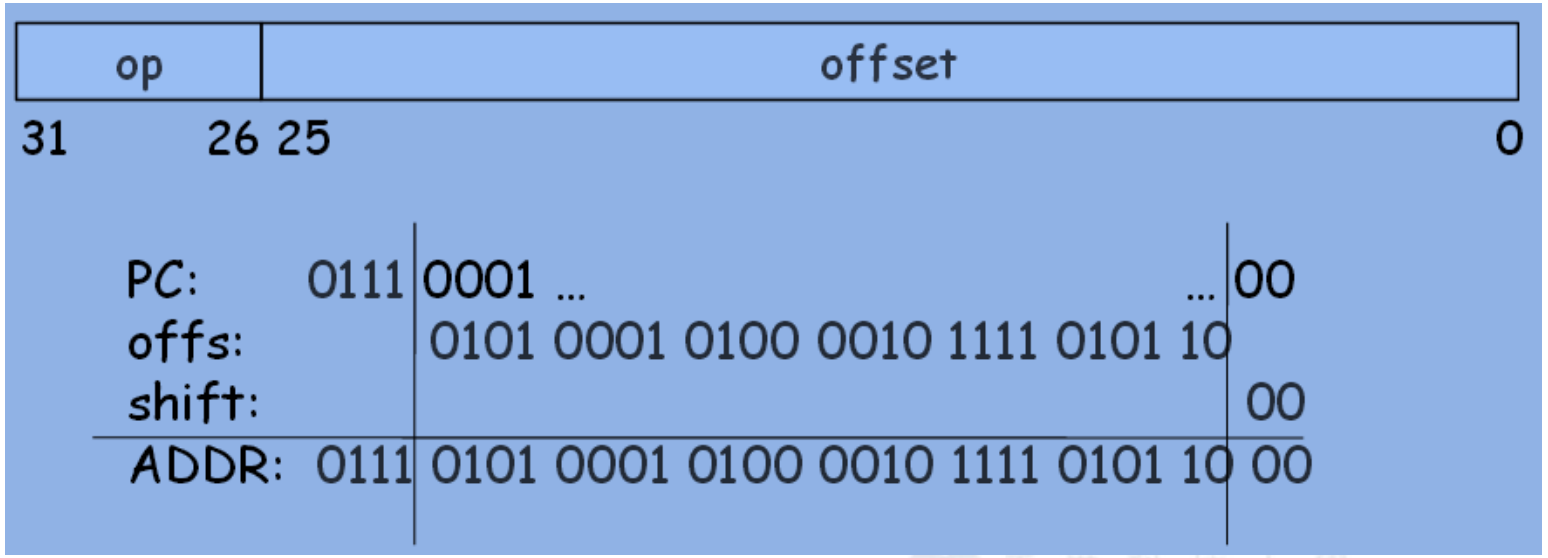
## J-type or J-format




- Used by instructions such as `j` ('jump') and `jal` ('jump and link')
  - e.g. `j  L1  # go to instruction labeled L1`

# Addressing in Un-conditional Jumps

- Direct Addressing: the address is 'the immediate'. 32-bit address cannot be embedded in a 32-bit instruction
- **Psuedo-direct Addressing**: 26 bits of the address is embedded as the immediate
- Example: `j Label`



# Example

```
Loop:  slt   $t1,$zero,$a1        # t1=9,a1=5
       beq   $t1,$zero,Exit       # no=>Exit
       add   $t0,$t0,$a0          # t0=8,a0=4
       subi  $a1,$a1,1            # a1=5
       j     Loop                 # goto Loop
Exit:  add $v0,$t0,$zero          # v0=2,t0=8
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 40000008 (slt) | 0 | 0 | 5 | 9 | 0 | 42 |
| 4000000C (beq) | 4 | 9 | 0 | 3 | | |
| 40000010 (add) | 0 | 8 | 4 | 8 | 0 | 32 |
| 40000014 (subi) | 8 | 5 | 5 | -1 | | |
| 40000018 (j) | 2 | 0000 0000 0000 0000 0000 0000 10 (in decimal, it's 2) | | | | |
| 4000001C (add) | 0 | 8 | 0 | 2 | 0 | 32 |

# Branching Far Away

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

```
beq $s0, $s1, L1   # L1 = 16-bit address
```

  - with this

```
bne $s0, $s1, L2   # L2 = 16-bit address
j   L1             # L1 = 26-bit address
L2:
```

# Stretching the Maximum Possible Distance

Because,

- All MIPS instructions are 4 bytes long

So,

- A branch target or offset can refer to number of words instead of number of bytes
- essentially stretch the maximum possible branching distance by 4x

Questions:

- What is the range a ‘j’ and ‘jal’ can jump to?
  - Within 256MB
- What if we want to jump beyond 256MB?

# Stretching the Maximum Possible Distance

| | |
|---|---|
| 0x0 0000000 | |
| … | |
| 0x0 FFFFFFF | |
| 0x1 0000000 | |
| … | |
| 0x1 FFFFFFF | |
| 0x2 0000000 | `J L1` |
| … | |
| 0x2 FFFFFFF | `L1: …` |
| … | |
| 0x7 0000000 | |
| … | |
| 0x7 FFFFFFF | |

# Stretching the Maximum Possible Distance (cont.)

- What if the Jump target is more than 256 MB away?

| | |
|---|---|
| 0x0 0000000 | |
| … | |
| 0x0 FFFFFFF | |
| 0x1 0000000 | |
| … | |
| 0x1 FFFFFFF | |
| 0x2 0000000 | |
| … | |
| 0x2 FFFFFFF | |
| … | |
| 0x7 0000000 | |
| … | |
| 0x7 FFFFFFF | |

```
J L1


L1: …
```

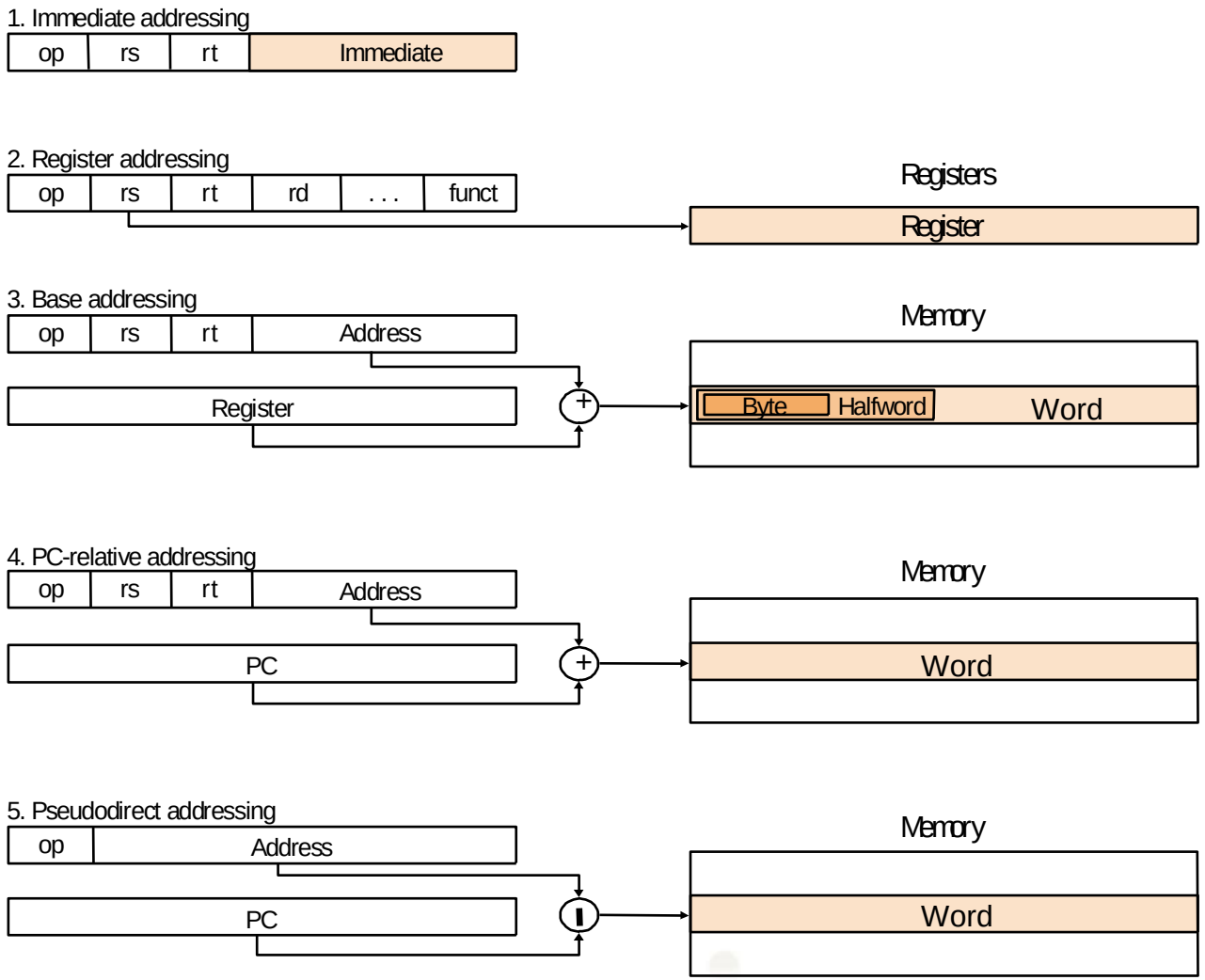
# Jump Register

- Set the register content as the target address
- Then simply `jr`

# Summary of MIPS Addressing Modes

1. **Immediate addressing**
   - The operand is a constant within the instruction itself
2. **Register addressing**
   - The operand is a register
3. **Base addressing or displacement addressing**
   - The operand is at the memory location with address
     = (register) + constant
4. **PC-relative addressing**
   - The address is = (PC) + 4 + constant
5. **Pseudodirect addressing**
   - The jump address is a constant in the instruction concatenated with the upper 4 bits of the PC

# Summary of MIPS Addressing Modes (cont'd)



# OTHER ISSUES (OPTIONAL)

# RISC vs. CISC Design (optional)

- **MIPS is an example of RISC**
  - **R**educed **I**nstruction **S**et **C**omputer
  - Each instruction does one simple thing
  - Most existing processors are RISC since it is more promising
- **Another approach is CISC**
  - **C**omplex **I**nstruction **S**et **C**omputer
  - One instruction may do multiple things, e.g. Intel's instruction set

| | RISC | CISC |
|---|---|---|
| Number of instructions in a program | (–) more | (+) less |
| Time to execute the program | (+) usually less | (–) usually more |
| Hardware design | (+) simple | (–) complex |

(+) means advantage, (–) means disadvantage

# Comparing RISC and CISC in Details (optional)

| RISC | CISC |
|---|---|
| Through quantitative measurements, choose only the most useful instructions and addressing modes. | Choose instructions and addressing modes that make the translation of high-level languages to assembly language simpler. |
| With few instructions and addressing modes, we can directly execute them in hardware. | Since we can have many instructions and addressing modes, we need a **microcode** (or **microprogrammed control**) to execute them in hardware. |
| A lot of chip space can be left for a large number of registers and cache memory. | We can have only few registers and small cache memory. |
| Compilers are more difficult to write. | Compilers are easier to write. |
| Assembly language programs are more difficult to write. | Assembly language programs are easier to write. |

# Assembler Pseudoinstructions

## Pseudoinstructions

- Assembly language instructions that do not have corresponding machine instructions (i.e., they need not be implemented directly in hardware)

## Why Pseudoinstructions?

- Their appearance in assembly language simplifies programming and translation, giving MIPS a richer set of assembly language instructions than those implemented by the hardware.

## Cost of supporting Pesudoinstructions

- The only cost is reserving one register, `$at`, for use by the assembler

# Examples of Pseudoinstructions

- **`move`:**
  - `move $t0, $t1  #$t0 gets value of $t1`
  - The assembler converts this pseudoinstruction into the machine language equivalent of the following instruction:
  - `add  $t0, $zero, $t1  #$t0 gets 0 + value of $t1`

- **Others:**
  - `blt` (‘branch on less than’)
  - `ble` (‘branch on less than or equal’)
  - `bgt` (‘branch on greater than’)
  - `bge` (‘branch on greater than or equal’)

# Concluding Remarks

- The stored-program concept underlies today's digital computers
- Registers are fast temporary storage inside the processor
- An instruction specifies an operation and its corresponding operand(s)
- All MIPS instructions are 32 bits in length
  - To simplify the instruction set architecture
  - But, multiple instruction formats are supported
- Four design principles for ISA
  - Simplicity favors regularity
  - Smaller is faster
  - Make common case fast
  - Good design demands good compromises

# Concluding Remarks (cont'd)

- **Program counter is a special register**
  - Pointing to the current instruction to be fetched and executed
- **Branch/jump instructions often require branch address calculation**
- **MIPS supports different addressing modes**
  1. Register
  2. Displacement
  3. Immediate
  4. PC-relative
  5. Pseudodirect
- **Pseudoinstructions extend the MIPS instruction set**
  - To facilitate program development
- **RISC and CISC are two very different design philosophies**