



# COMPUTER ARCHITECTURE

Code: CT173

## *Part III: Instruction Set Architecture Design*

---

MSc. NGUYEN Huu Van LONG

Department of Computer Networking and Communication,

College of Information & Communication Technology,

CanTho University

# Agenda

- **Instruction Set Architecture**

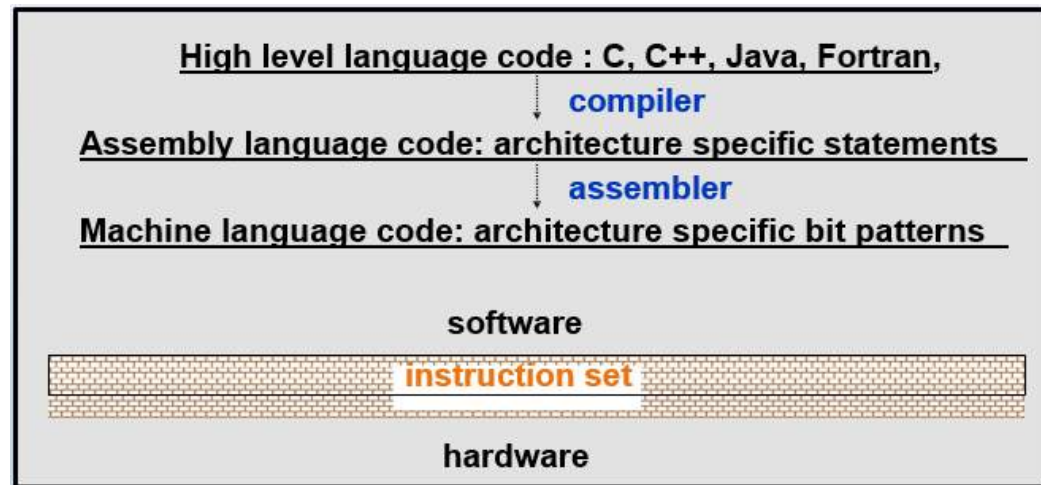
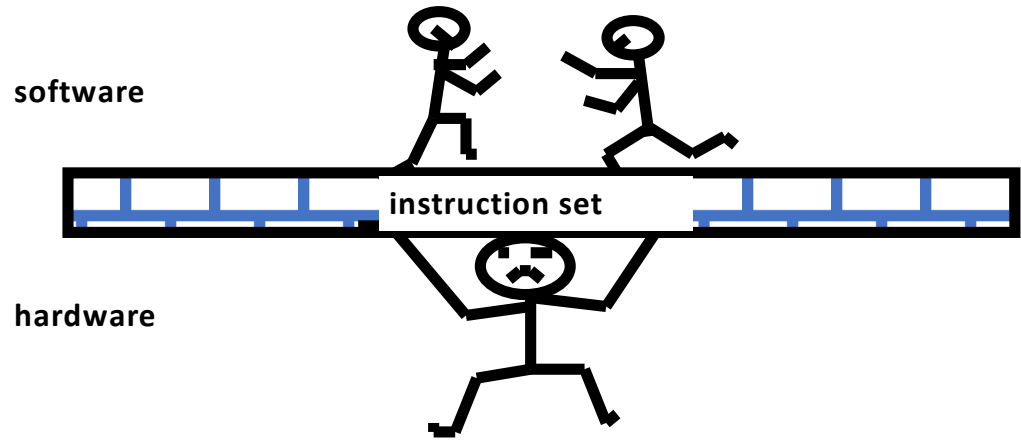
- Principal concepts
- How to be a good ISA
- ISA requirements
- Operands Storage
  - Stack architecture
  - Accumulator register architecture
  - General purpose register architecture
    - Register - Register: RISC machine
    - Register - Memory: CISC machine
- Memory addressing mode
- Type and size of operands
- Operation classification
- Control flow instructions
- Encoding format

- **The role of compilers**

- Typical RISC machines (**My simple RISC machine**, MIPS64)
- **Teach-Sim software** to learn yourself

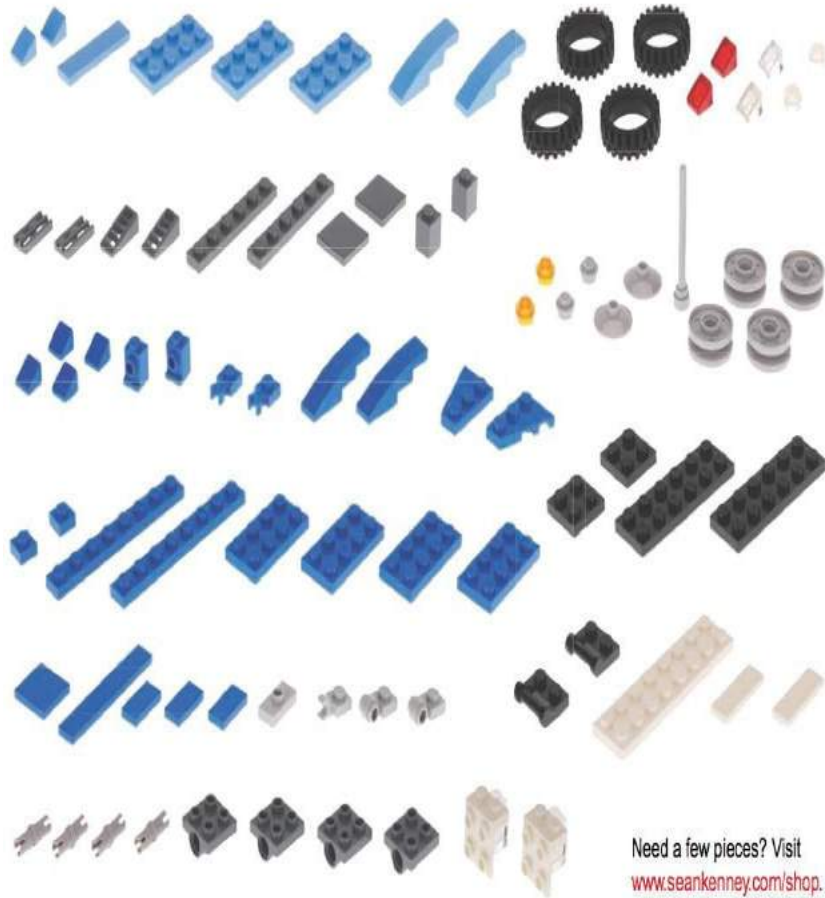
# ISA: Principal Concepts

- ISA serves as an **interface** between software and hardware to provides a mechanism by which the software **tells the hardware what should be done**
- To use the hardware of a computer, we must speak its (computer) language which are called **Instructions**, and its vocabulary is called an **Instruction Set**



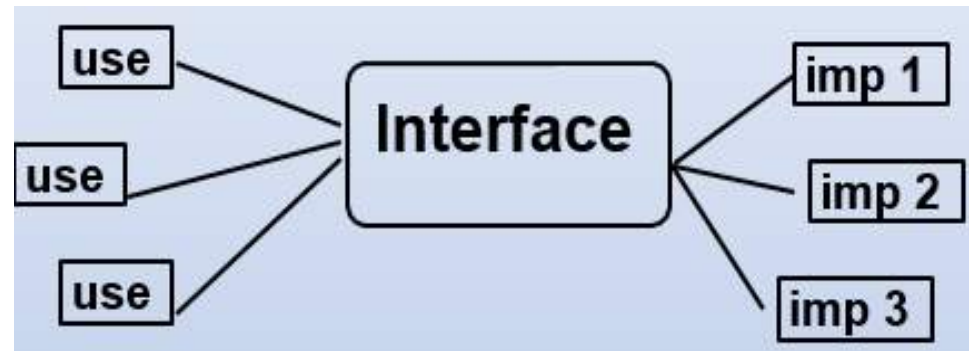
# ISA: Design Approach

## What is a good ISA design?



## A good ISA should be

- Lasts through many implementations (portability, compatibility)
- Can be used in many different ways (generality)
- Provides sufficient functionality to higher levels
- Permits an efficient implementation at lower levels



# ISA: Essentially requirements

- **Where are operands stored?**
  - registers, memory, stack, accumulator
- **How many explicit operands are there?**
  - 0,1, 2, or 3
- **How is the operand location specified?**
  - register, immediate, indirect...
- **What type & size of operands are supported?**
  - byte, int, float, double, string, vector...
- **What operations are supported?**
  - add, sub, mul, move, compare...
- **How is the operation flow controlled?**
  - branches, jumps, procedure calls...
- **What is the encoding format?**
  - fixed, variable, hybrid...

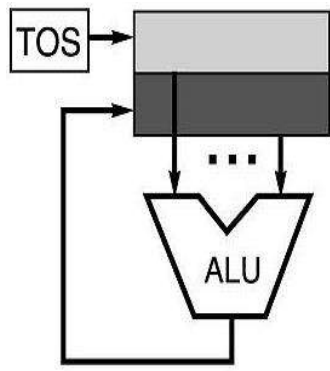
*These choices  
critically affect  
number of  
Instructions, CPI,  
and CPU cycle time*

# ISA: Operands Storage

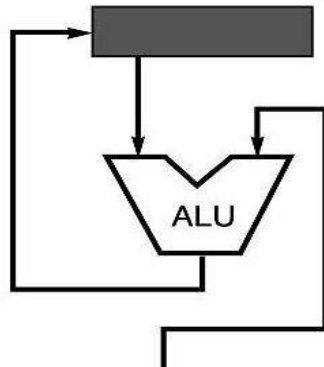
- There are two methods to store the operands: *explicitly* or *implicitly*
- Type of ISA could be classified by the way operands are stored in CPU
- Major choices:
  - In an **Accumulator architecture** one operand is *implicitly* the accumulator => similar to calculator
  - The operands in a **Stack architecture** are *implicitly* on the top of the stack
  - The **General-purpose register architectures** have only *explicit* operands - either registers or memory location

# ISA: Operand Storage $C := A + B$

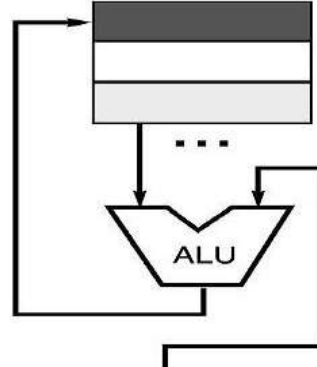
Stack	Accumulator	GPR (register-memory)	GPR (load- store)
<b>Push</b> <b>A</b> <b>Push</b> <b>B</b> <b>Add</b> <b>Pop C</b>	<b>Load A</b> <b>Add B</b> <b>Store C</b>	<b>Load R1, A</b> <b>Add R1, B</b> <b>Store C, R1</b>	<b>Load R1, A</b> <b>Load R2, B</b> <b>Add R3, R1, R2</b> <b>Store C, R3</b>



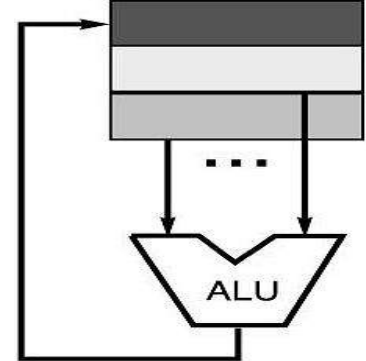
**Stack**



**Accumulator**  
 $acc = acc + mem[B]$




**Register - Memory**  
 $R1 = R1 + mem[B]$



**Register - Register**  
 $R3 = R1 + R2$

# ISA: Several types

ISA Type	Examples	Explicit operands per ALU inst.	Result Destination	Operand access method
<b>Stack</b>	B5500, B6500 HP 3000/70	0	Stack	Push & Pop Stack
<b>Accumulator</b>	Motorola 6809 + ancient ones	1	Accumulator	$\text{Acc} = \text{Acc} + \text{mem}[\text{A}]$
<b>General Purpose Register GPR</b>	Set IBM 360 DEC VAX + all modern micro's	2 or 3	Registers or Memory	$\text{Rx} = \text{Ry} + \text{mem}[\text{A}]$ $\text{Rx} = \text{Rx} + \text{Ry}$ (2) $\text{Rx} = \text{Rx} + \text{Rz}$ (3)



Register-register, register-memory,  
and memory-memory (gone) options

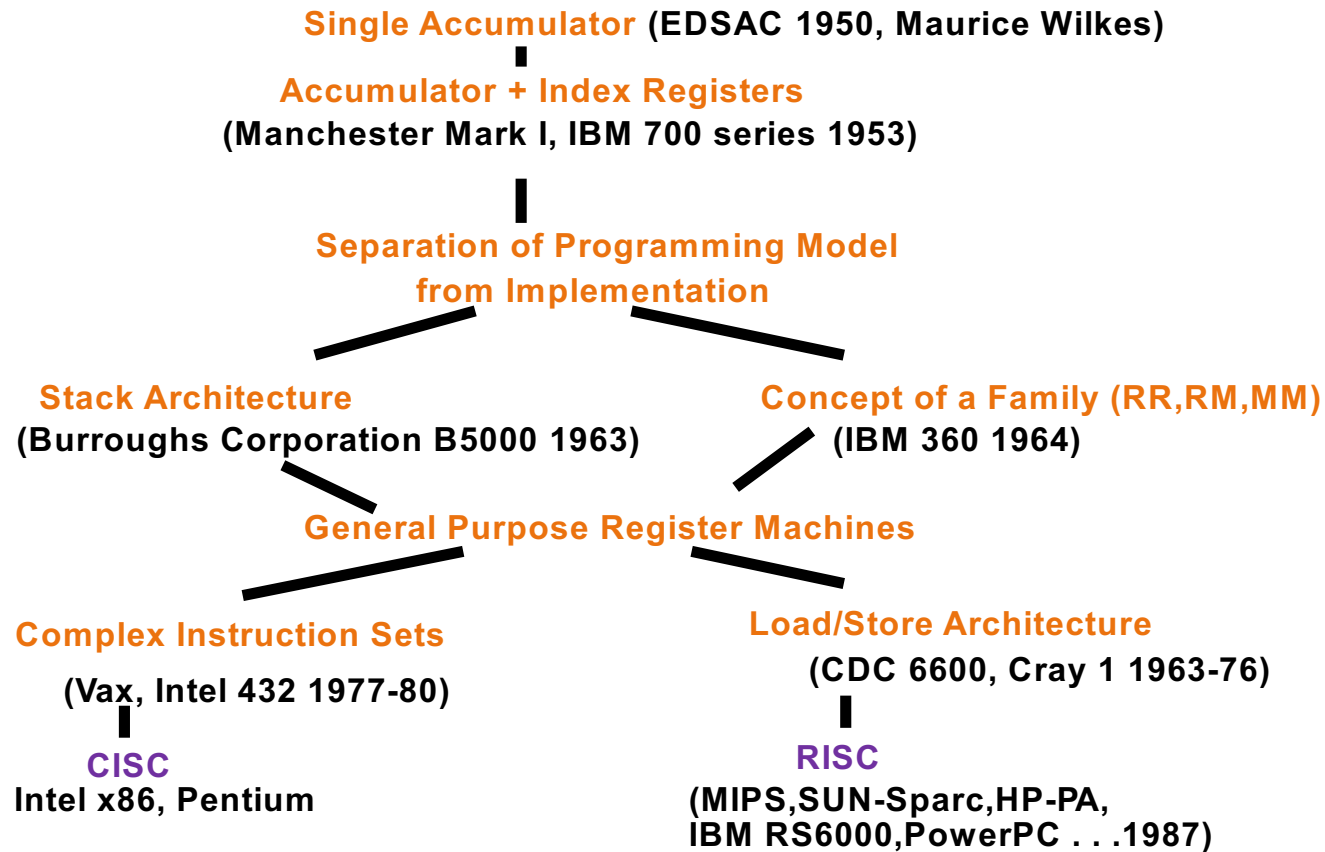


# ISA: Stack - Accumulator - GPR

ISA Type	Advantages	Disadvantages
<b>Stack</b>	<ul style="list-style-type: none"><li>• Simple effective address</li><li>• Short instructions</li><li>• Good code density</li></ul>	<ul style="list-style-type: none"><li>• Lack of random access</li><li>• Efficient code is difficult to generate</li><li>• Stack is often a bottleneck</li></ul>
<b>Accumulator</b>	<ul style="list-style-type: none"><li>• Minimal internal state</li><li>• Fast context switch</li><li>• Short instructions</li></ul>	<ul style="list-style-type: none"><li>• Very high memory traffic</li></ul>
<b>Register</b>	<ul style="list-style-type: none"><li>• Registers are faster than memory</li><li>• Registers can be used to hold variables<ul style="list-style-type: none"><li>+ reduce memory traffic</li><li>+ speed up programs</li></ul></li><li>• Registers are more efficient for a compiler to use than other forms of internal storage</li></ul>	<ul style="list-style-type: none"><li>• Longer instructions</li><li>• Possibly complex effective address generation</li><li>• Size and structure of register set has many options</li></ul>

**\*\*\*\*\* Register is the class that won out! \*\*\*\*\***

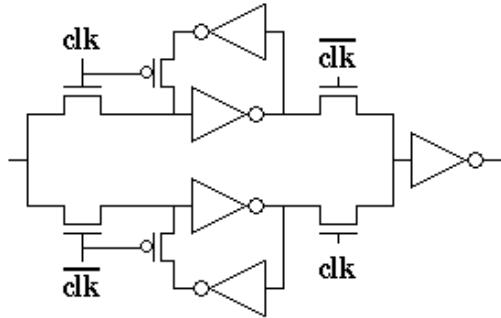
# ISA: Evolution hierarchy



# General purpose register - GPR

- **Registers are much faster than memory (even cache)**
  - Register values are available “immediately”
  - When memory isn’t ready, processor must wait or **stall**
- **Registers are convenient for variable storage**
  - Compiler assigns some variables (especially frequently used ones) just to registers
  - More compact code since small fields specify registers (compared to memory addresses)
- **Disadvantages**
  - Higher instruction count (load/store)
  - Dependent on good compiler (Reg. assignment)
  - Higher hardware cost (comparing to MEM)

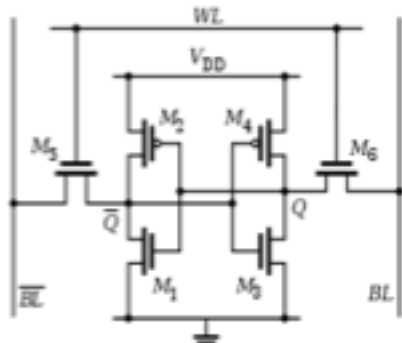
# GPR: Complex in register design



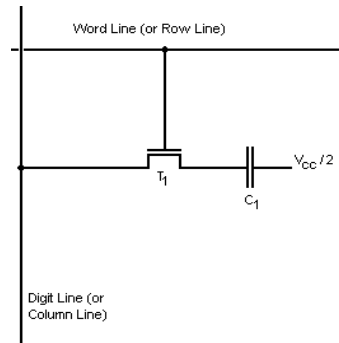
**Register (DFF) Cell (16T)**

	Register Bank	Memory
Size	256*4Byte	1K*4Byte
Area	0.14mm <sup>2</sup>	0.04mm <sup>2</sup>
Density	7KB/mm <sup>2</sup>	100KB/mm <sup>2</sup>

## Reg v.s. Mem (65nm CMOS)



**SRAM Cell (6T)**



**DRAM Cell (1T)**

# Register Machines

- How many registers are sufficient?
- **General-purpose registers vs. Special-purpose registers**
  - *Compiler flexibility and hand-optimization*
- Two major concerns for arithmetic and logical instructions (ALU)
  - Two or three operands
    - $X + Y \rightarrow X$
    - $X + Y \rightarrow Z$
- How many of the operands may be memory addresses (0 - 3)

Number of memory addresses	Max number of operands allowed	Examples
0	3	Alpha, ARM, MIPS, PowerPC, Sparc, Trimedia TM5200
1	2	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	VAX, PDP-1, National 32x32, IBM 360SS
3	3	VAX

**Conclusion: Register classification by (Number of memory addresses; Number of operands allowed)**

# Register Machine: Re - Re (0;3)

- ALU is **Register to Register** - also known as ***pure Reduced Instruction Set Computer (RISC)***
- **Advantages**
  - Simple fixed length instruction encoding
  - Decode is simple since instruction types are small
  - Simple code generation model
  - Instruction CPI tends to be very uniform
    - Except for memory instructions of course, but there are only 2 of them - load and store
- **Disadvantages**
  - Instruction count tends to be higher
  - Some instructions are short - wasting instruction word bits

# Register Machine: Re - Mem (1;2)

- **Evolved RISC and also old CISC**

- New RISC machines capable of doing *speculative loads*
- Predicated and/or *deferred loads* are also possible

- **Advantages**

- Data access to ALU immediate without loading first
- Instruction format is relatively simple to encode
- Code density is improved over Register (0, 3) model

- **Disadvantages**

- Operands are not equivalent - source operand may be destroyed
- Need for memory address field may limit # of registers
- CPI will vary
  - if memory target is in L0 cache then not so bad
  - if not - life gets miserable

# Register Machine: Mem - Mem (2;2) (3;3)

- **True and most complex CISC model**

- currently extinct and likely to remain so
- more complex memory actions are likely to appear but not
- directly linked to the ALU

- **Advantages**

- most compact code
- doesn't waste registers for temporary values
  - good idea for use once data - e.g. streaming media

- **Disadvantages**

- large variation in instruction size - may need a shoe-horn
- large variation in CPI - i.e. work per instruction
- exacerbates the infamous memory bottleneck
  - register file reduces memory accesses if reused

- **NOT USED TODAY AND MAYBE REMAIN SO!!**



# Register Machine: RISC vs CISC

MULT 2:3, 5:2

## CISC MODEL

- Emphasis on hardware
- Includes multi-clock complex instructions
- Memory-to-memory: “LOAD” and “STORE” incorporated in instructions
- Small code sizes
- Less memory access

LOAD A 2:3

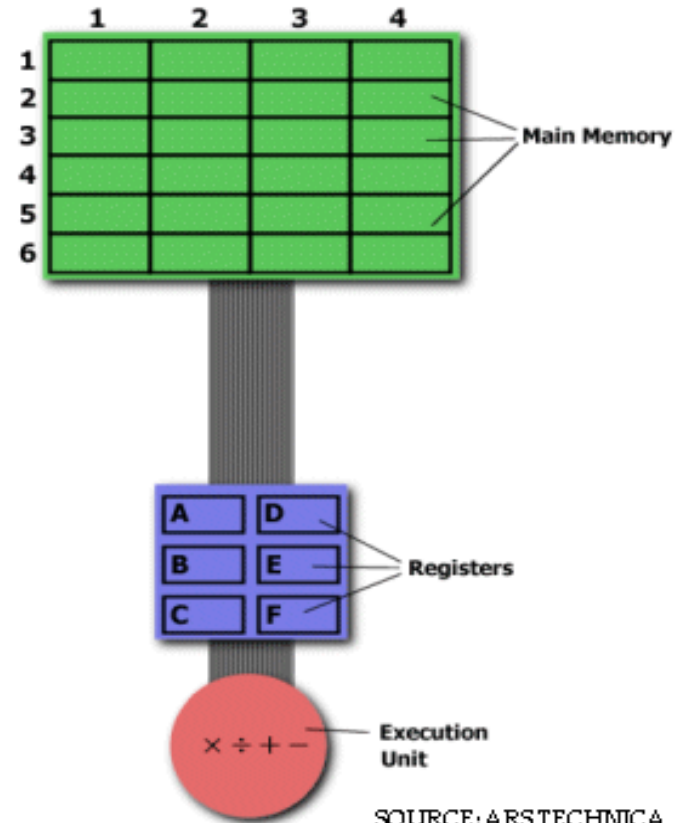
LOAD B 5:2

PROD A,B

STORE 2:3, A

## RISC MODEL

- Emphasis on software
- “Single”-clock, reduced instruction only
- Register to register: “LOAD” and “STORE” are independent instructions



# Register Machine: RISC vs CISC

Instructions CISC	Signification	Nb de cycles	Codage Nb octets	Instructions RISC	Signification	Nb de cycles	Codage Nb octets
MOVE Ri, Rj	Ri := Rj	3	4	MOVE Ri, Rj	Ri := Rj	1	4
MOVE (Ri), Rj	(Ri) := Rj	5	6	MOVE (Ri), Rj	(Rj) := Ri	2	4
MOVE Ri, (Rj)	Ri := (Rj)	5	6	MOVE Ri, (Rj)	Ri := (Rj)	2	4
MOVE (Ri), (Rj)	(Ri) := (Rj)	8	8	Non disponible			
MOVE Ri, val	Ri := val	3	6	MOVE Ri, val	Ri := val	1	4
OP Ri, Rj, Rk	Ri := Rj op Rk	3	4	OP Ri, Rj, Rk	Ri := Rj op Rk	1	4
OP Ri, Rj, val	Ri := Rj op val	3	6	OP Ri, Rj, val	Ri := Rj op val	1	4
OP Ri, Rj, (Rk)	Ri := Rj op (Rk)	5	6	Non disponible			
OP Ri, (Rj), Rk	Ri := (Rj) op Rk	5	6	Non disponible			
OP Ri, (Rj), (Rk)	Ri := (Rj) op (Rk)	8	6	Non disponible			
OP (Ri), Rj, Rk	(Ri) := Rj op Rk	5	6	Non disponible			
OP (Ri), Rj, (Rk)	(Ri) := Rj op (Rk)	8	6	Non disponible			
OP (Ri), (Rj), Rk	(Ri) := (Rj) op Rk	8	6	Non disponible			
OP (Ri), (Rj), (Rk)	(Ri) := (Rj) op (Rk)	10	6	Non disponible			
BRA adr	Saut a adr	3	4	BRA adr	Saut a adr	1	4
BEQ adr	Saut si z=1	3	4	BEQ adr	Saut si z=1	1	4
BNE adr	Saut si z=0	3	4	BNE adr	Saut si z=0	1	4

**Available operations for both processors are the followings**

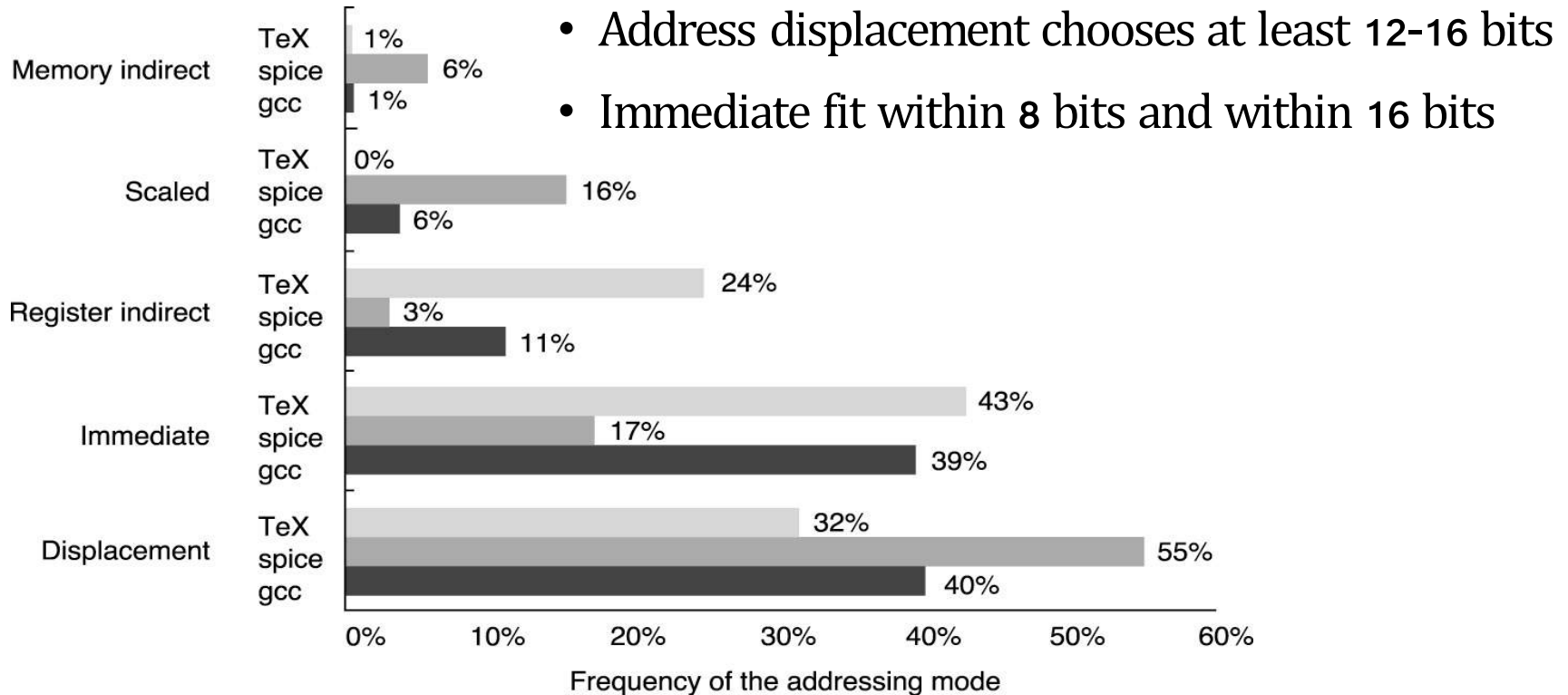
- **CISC** : OP = ADD, MULT, DIV, SOUS, SQRT, SIN, COS, TAN, ARCTAN, ARCOS, and logical operations;
- **RISC** : OP = ADD, MULT, SOUS, DIV, and logical operations.

# ISA: Memory Addressing Mode

*How do architectures specify the address of an object they will access?*

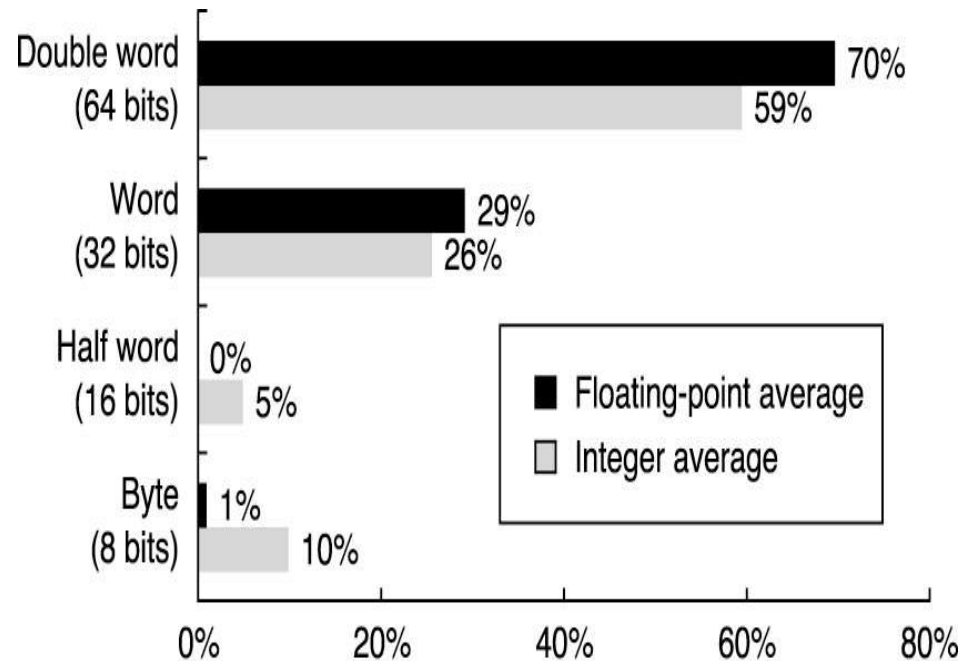
Addressing Mode	Example	Action
<i>Register Direct</i>	Add R4, R3	$R4 \leftarrow R4 + R3$
<i>Immediate</i>	Add R4, #3	$R4 \leftarrow R4 + 3$
<i>Displacement</i>	Add R4, 100(R1)	$R4 \leftarrow R4 + M[100 + R1]$
<i>Register indirect</i>	Add R4, (R1)	$R4 \leftarrow R4 + M[R1]$
<i>Indexed</i>	Add R4, (R1 + R2)	$R4 \leftarrow R4 + M[R1 + R2]$
<i>Direct</i>	Add R4, (1000)	$R4 \leftarrow R4 + M[1000]$
<i>Memory Indirect</i>	Add R4, @(R3)	$R4 \leftarrow R4 + M[M[R3]]$
<i>Auto-increment</i>	Add R4, (R2)+	$R4 \leftarrow R4 + M[R2]$ $R2 \leftarrow R2 + d$
<i>Auto-decrement</i>	Add R4, -(R2)	$R2 \leftarrow R2 - d$ $R4 \leftarrow R4 + M[R2]$
<i>Scaled</i>	Add R4, 100(R2)[R3]	$R4 \leftarrow R4 + M[100 + R2 + R3*d]$

# ISA: Memory Addressing Mode



# ISA: Type And Size of Operands

- *How is the type of an operand designated?*
  - usually encoded in the **OPCODE**, i.e. LDB - load byte; LDW - load word
- *What are the operand types: (imply their sizes)*
  - Character (8 bits or 1 byte)
  - Half word (16 bits or 2 bytes)
  - Word (32 bits or 4 bytes)
  - Double word (64 bits or 8 bytes)
  - Single precision floating point (4 bytes or 1 word)
  - Double precision floating point (8 bytes or 2 words)



# ISA: Operation Classification

Operator type	Description	Instruction examples
1. Arithmetic and Logical	Integer arithmetic and logical operations	<i>ADD, AND, OR, SUB, MUL, DIV</i>
2. Data transfer	Loads - Stores (move instructions on computers with memory addressing)	<i>LD, STR, LDW, STRW</i>
3. Control	Branch, jump, procedure call and return, traps	<i>JMP, BNR, CALL</i>
4. System	Operating system call, virtual memory management instructions	<i>OS CALL, VM</i>
5. Floating point	Floating point operations	<i>ADDF, MULF, DIVF</i>
6. Decimal	Decimal add, decimal multiply, decimal to characters conversions	<i>ADDD, CONVERT</i>
7. String	String move, compare, search	<i>MOVE, COMPARE, SEARCH</i>
8. Graphics	Pixel and vertex operations, compression/decompression	<i>COMPRESS, DECOMPRESS</i>

- All computers generally provide **a full set of operations for the first three categories**
- All computers must have **some instruction support for basic system functions**
- **Graphics** instructions **typically operate on many smaller data items in parallel**

# ISA: Control Flow Instructions

- ***Control flow instructions change the flow of control***
  - Instead of executing the next instruction, the program branches to the address specified in the branching instructions
- ***They are a big deal***
  - Primarily because they are difficult to optimize out
  - AND they are frequent
- ***Four types of control instructions***
  - *Conditional branches*
  - *Jumps - unconditional transfer*
  - *Procedure calls*
  - *Procedure returns*
- **Issues**
  - *Where is the target address? How to specify it?*
  - *Where is return address kept? How are the arguments passed? (calls)*
  - *Where is return address? How are the results passed? (returns)*

# ISA: Control Flow Instructions

- **PC-relative (Program Counter)**

- Supply a displacement added to the PC
  - Known at compile time for jumps, branches, and calls (specified within the instruction)
- The target is often near the current instruction
  - requiring fewer bits
  - independently of where it is loaded (position independence)

- **Register indirect addressing - dynamic addressing**

- The target address may not be known at compile time
- Naming a register that contains the target address
  - Case or switch statements
  - Virtual functions or methods
  - High-order functions or function pointers
  - Dynamically shared libraries



# ISA: Encoding Format

Operation and no. of operands	Address specifier 1	Address field 1	...	Address specifier	Address field
----------------------------------	------------------------	--------------------	-----	----------------------	------------------

(a) Variable (e.g., VAX, Intel 80x86)

Operation	Address field 1	Address field 2	Address field 3
-----------	--------------------	--------------------	--------------------

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field
-----------	----------------------	------------------

Operation	Address specifier 1	Address specifier 2	Address field
-----------	------------------------	------------------------	------------------

Operation	Address specifier	Address field 1	Address field 2
-----------	----------------------	--------------------	--------------------

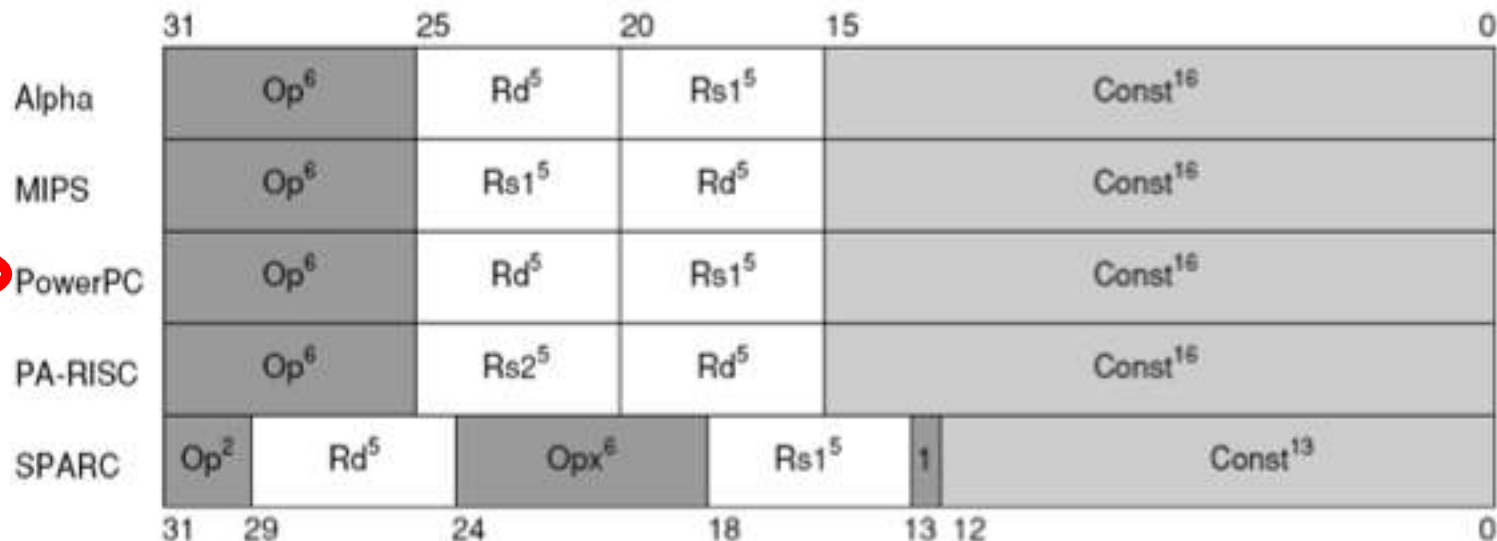
(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

- **Opcode (Operation)**: specifying the operation
- **Address Specifier**: tells what addressing mode is used
- **Encoding issues**
  - The desire to have as many registers and addressing modes as possible
  - The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size
  - A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation

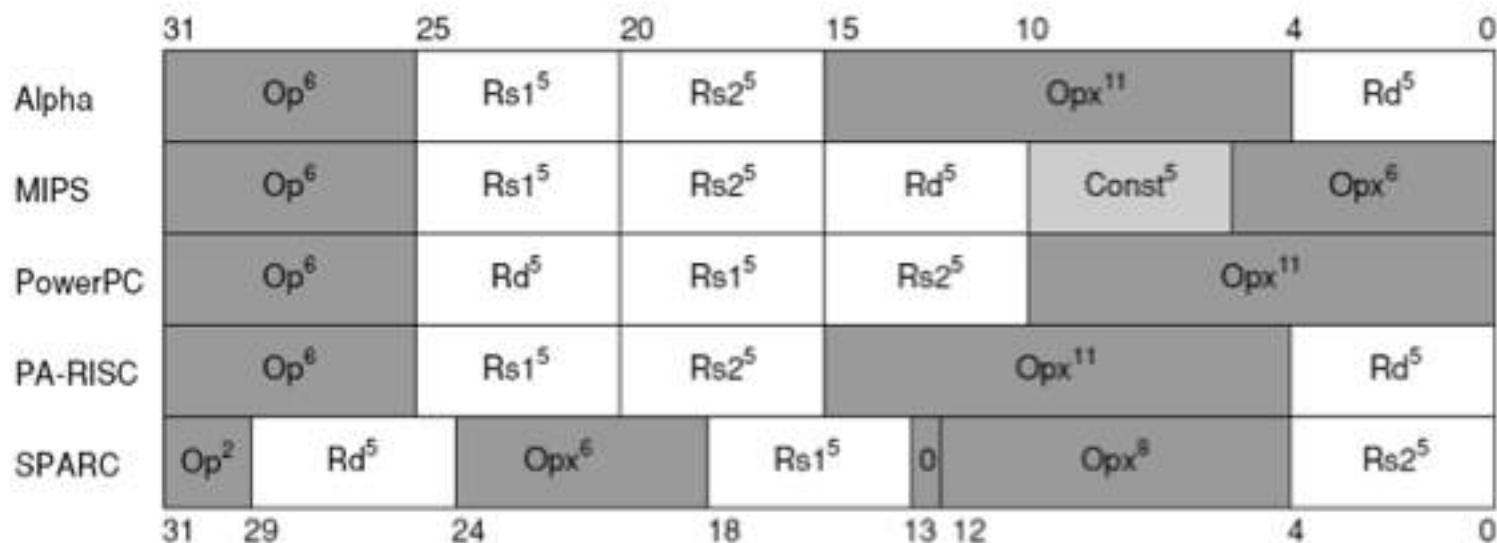
- Encoding format could be variable (a), fixed (b) or hybrid (c)
- CISC usually use (a)
- RISC usually use (b)

# ISA: Types of Instruction Format

## Register-immediate

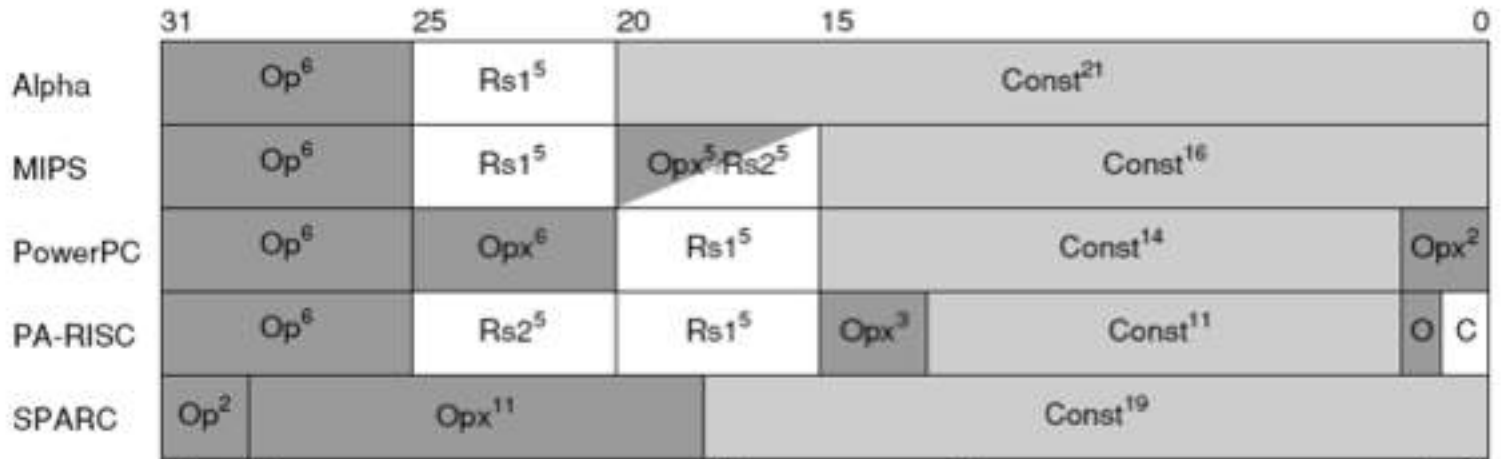


## Register-register

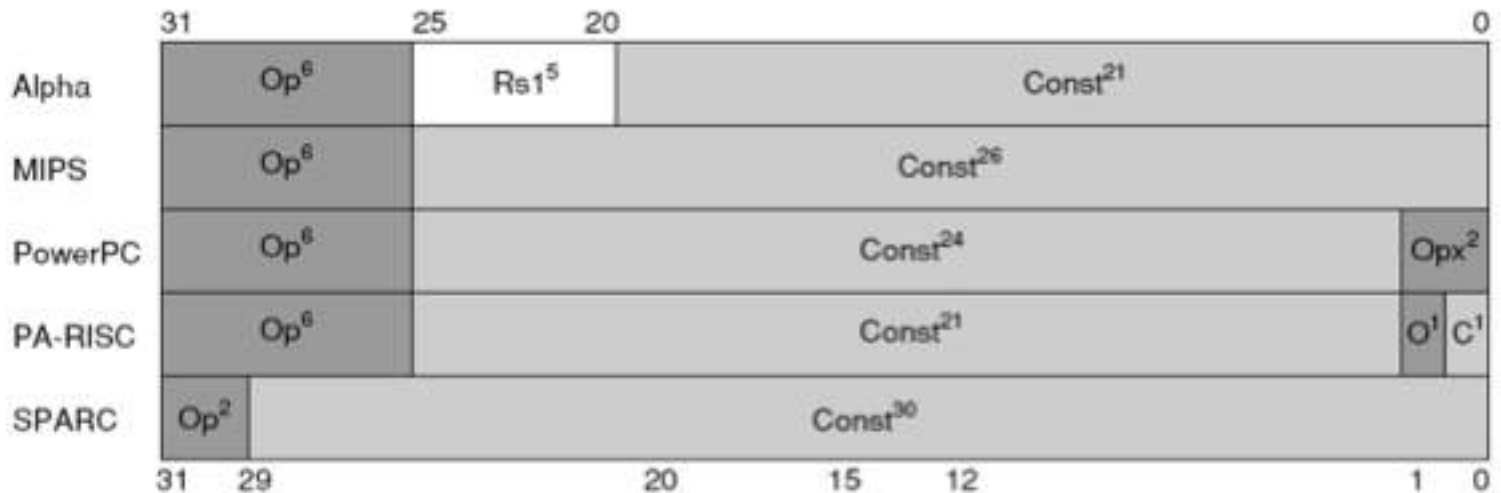


# ISA: Types of Instruction Format

Branch



Jump/call



Op<sub>x</sub> Opcode

Rs Register

Const Constant

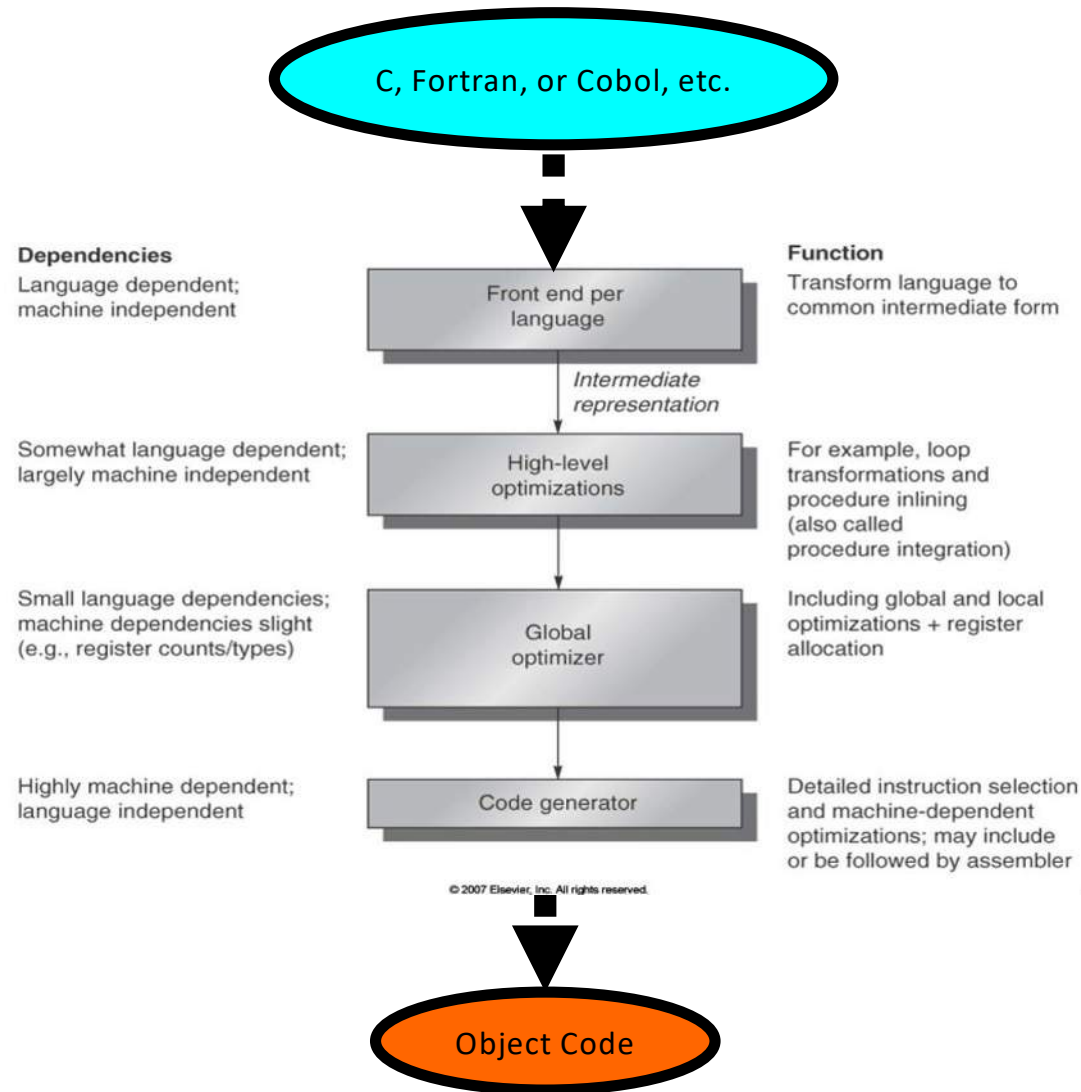
# Role of Compiler

- **Why Compiler has important role in enhance computer performance?**

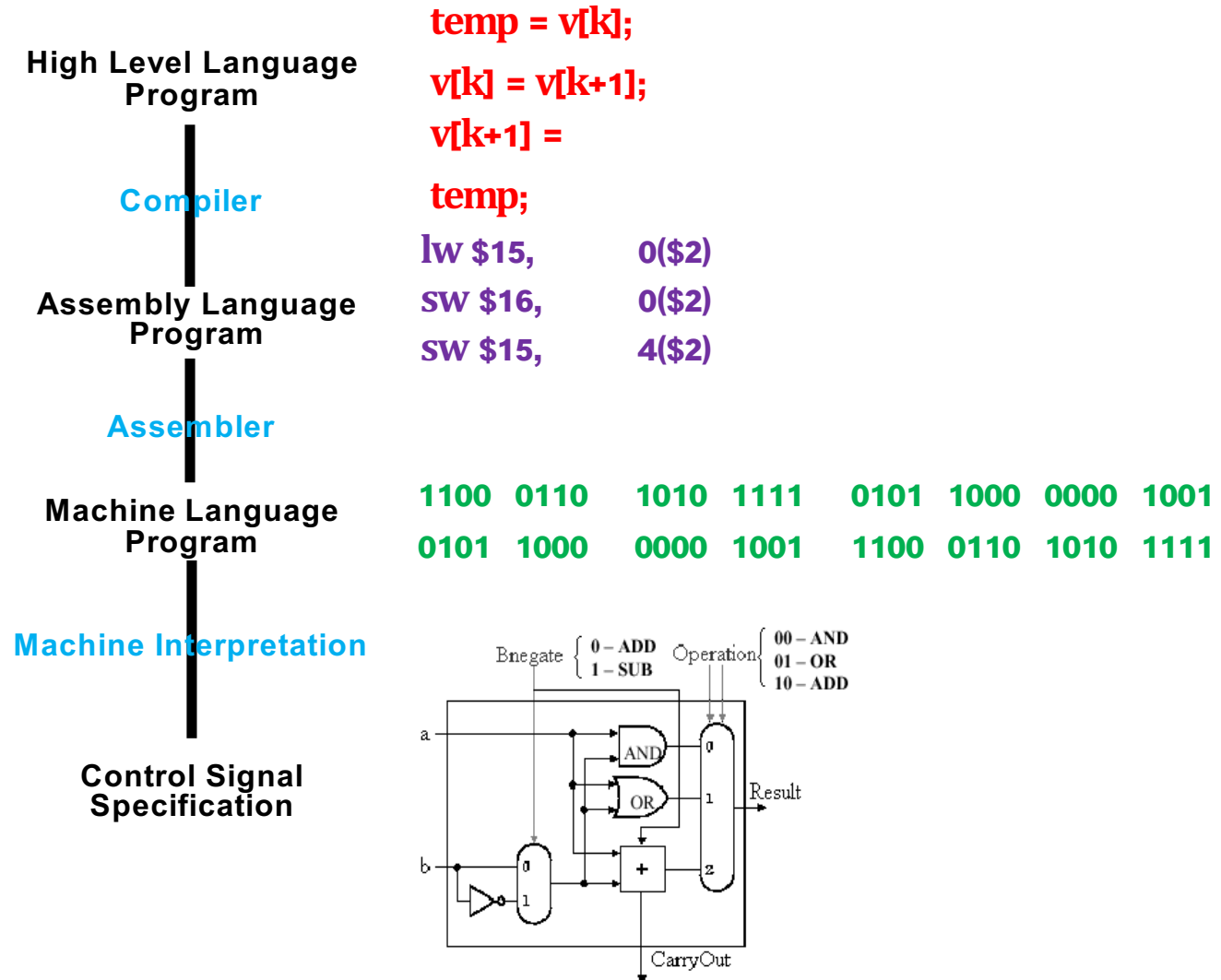
- Today almost programs or applications are done in *high-level languages* → an ISA is essentially a compiler target.
- Understanding compiler technology today is critical to designing and efficiently implementing an ISA.

- **Global goals of a typical compiler**

- All correct programs execute correctly
- Most compiled programs execute fast (optimizations)
- Fast compilation
- Debugging support



# Role of Compiler



# Role of Compiler: Optimization

- **Code improvements made by the compiler are called optimizations and can be classified**
  - High-order transformations: procedure inline
  - Optimizations: dead code elimination
  - Constant propagation
  - Common sub-expression elimination
  - Loop-unrolling
  - Register allocation (almost most important)
  - Machine-dependent optimizations: takes advantage of specific architectural features
- **Almost all of these optimizations are easier to do if there are many general registers available!**
  - Common sub/expression elimination stores temporary value into a register
  - Loop-unrolling
  - Procedure inline

# Role of Compiler: Optimization

```
int pred(int x) {  
    if (x == 0)  
        return 0;  
    else  
        return x - 1;  
}
```

**Procedure inlining**

```
int f(int y) {  
    return pred(y) + pred(0) + pred(y+1);  
}
```

```
int f(int y) {  
    int temp;  
    if (y == 0) temp = 0; else temp = y - 1;  
    if (0 == 0) temp += 0; else temp += 0 - 1;  
    if (y+1 == 0) temp += 0; else temp += (y + 1) - 1;  
    return temp;  
}
```

```
int x = 14;  
int y = 7 - x / 2;  
return y * (28 / x + 2);
```

**Constant propagation**

```
int x = 14;  
int y = 7 - 14 / 2;  
return y * (28 / 14 + 2);
```

```
int x = 14;  
int y = 0;  
return 0;
```

```
int foo(void)  
{  
    int a = 24;  
    int b = 25; /* Assignment to dead variable */  
    int c;  
    c = a << 2;  
    return c;  
    b = 24; /* Unreachable code */  
    return 0;  
}
```

**Dead code  
elimination**

```
a = b * c + g;  
d = b * c * e;
```

**Common sub expression  
elimination**

```
tmp = b * c;  
a = tmp + g;  
d = tmp * e;
```

# My ISA: Operator types

Operator type	Instructions	Using
Integer arithmetic	ADD, SUB, MUL, DIV	ADD R1, R2, R3; SUB R4, R5, R6
Floating point arithmetic	ADDF, SUBF, MULF, DIVF	ADDF F1, F2, F3
Logic operation	AND, OR, XOR, NEG, NAND, NOR	AND R1, R2, R3
Data transfer	LOAD, STORE	LOAD R4, @100; STORE R4,@100
Logic or arithmetic shift	SRL, SLL, SRA, SLA	SRL R5, 2; SLA R5, 3
Rotate	ROL, ROR	ROL R6
Comparison	CMP	CMP R3, R1, R2
Control (Condition)	JMP, BGT, BRA, BLT, MGT, MLT	JMP R5; BRA +2; BGT R5, +2;



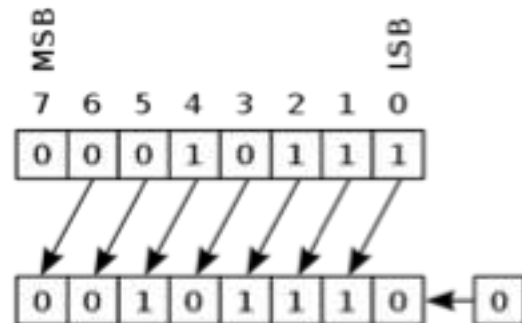
# My ISA: Shift instructions

- **Shift left/right logic and Shift right logic**

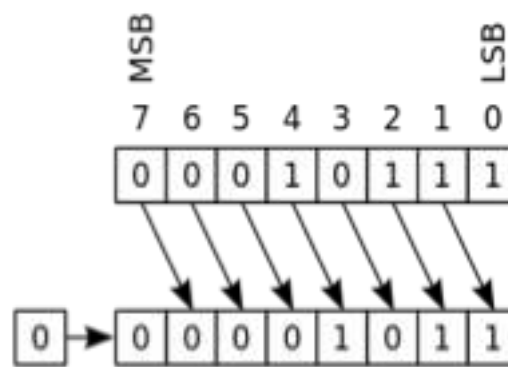
- Operand is *treated as bit sequence*, not a number
- Efficient way to perform multiplying/dividing of *Unsigned integers* by  $2^n$
- Shifting left  $n$  bits = multiplying by  $2^n$ ; Shifting right  $n$  bits = dividing by  $2^n$

- **Shift left/right arithmetic or signed shifted**

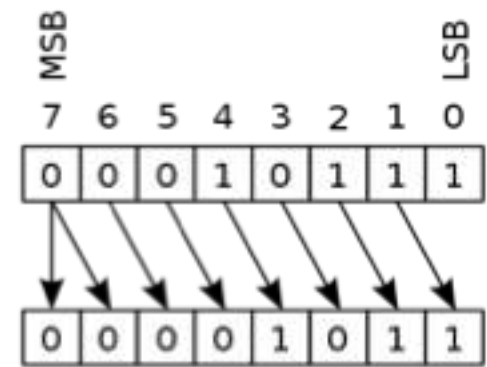
- Operand is *treated as a number*
- Efficient way to perform multiplication of *Signed/Unsigned integers* by  $2^n$
- Shifting left  $n$  bits (signed/unsigned) = multiplying by  $2^n$ ; Shifting right  $n$  bits (unsigned) = dividing by  $2^n$ ; Shifting right  $n$  bits (2's complement)  $\approx$  dividing by  $2^n$



Left shift logic (Unsigned) and Left shift arithmetic (Unsigned/Signed)



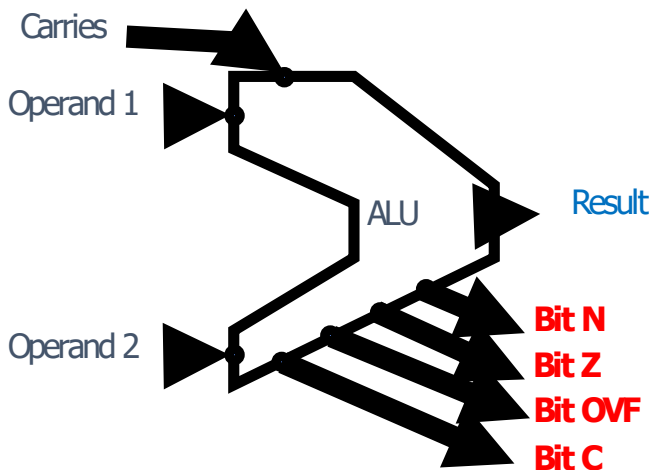
Right shift logic (Unsigned)



Right shift arithmetic (Signed/Unsigned)

# My ISA: Condition instructions

- Condition instructions (BGT, BLT...) using the result from Comparison instruction (CMP) to change the flow of data path.
- Using of CMP instruction: **CMP Ri, Rj, Rk** with Ri stores the Status Bits instead of storing the calculation result.
- There are two ways to store the Status Bits: Using a General Purpose Register or Using a Status Register.



## Status bits (Flags):

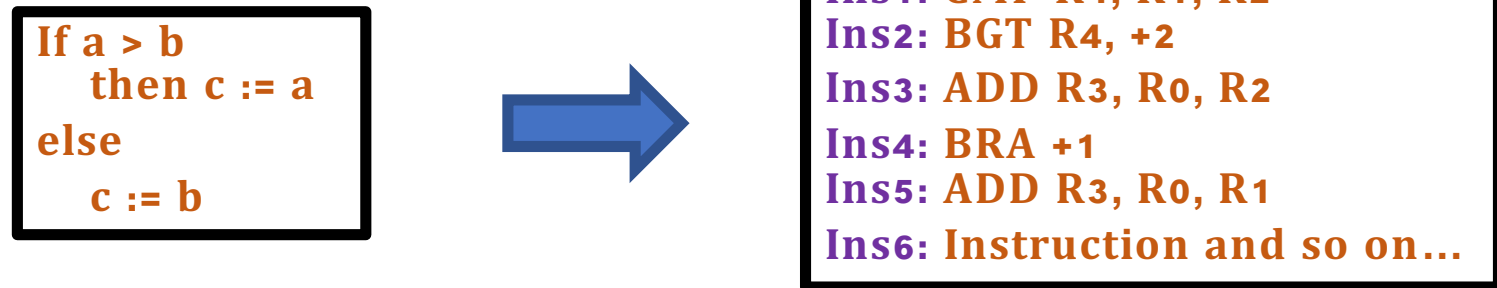
- **N (Negative):** 1 if result is negative.
- **Z (Zero):** 1 if result is zero
- **OVF (Overflow):** 1 if overflow in calculation
- **C (Carry):** (1 if carry out is 1)
- **Others:** P flag, I flag, S flag

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x	R = Readable bit W= Writable bit U = Unimplemented bit, read as '0' -n= Value at <u>POR</u> reset
IRP	RPI	RP0	TO	PD	Z	DC	C	
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

The Status Register of **PIC - Peripheral Interface Controller**, a family of **RISC microcontrollers**

# My ISA: Condition instructions

- To change the flow of data path, the PC register needs to change with (BRA) or without a condition (BGT, BLT), popularly called Branch instructions



- Branch instructions could extremely harm the modern CPUs which are all implemented parallel processing techniques*, i.e. pipelining or super pipelining. Therefore, the assembly programmer **should using Move instruction (with condition) instead of Branch instruction.**

```
Ins1: CMP R4, R1, R2
Ins2: ADD R3, R0, R2
Ins3: MGT R4, R3, R1
```

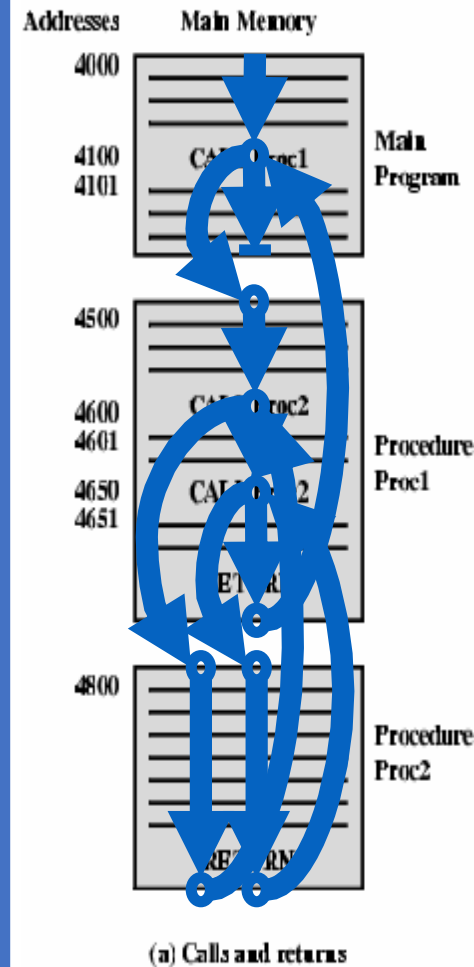
# My ISA: Condition instructions

- A program could have one or more Procs (Funcs) to complete its tasks.
- The Procs (Funcs) could be called whenever and wherever by **CALL** command from system. This command makes an temporally interrupt at the Procs (Funcs) calling instruction, indicated by address  $PA_1$ . PC register pointed to the address of Procs (Funcs)  $PAP_1$ :  $PC \leftarrow PAP_1$
- After finishing the Procs (Funcs) , the system returns back to  $PA_1$  by **RETURN** command.
- **R31** is selected to store the recently  $PA_i$  and **Stack structure** is used to store the entire list of PA. For Stack, **R30** is the stack pointer.
- To make a **CALL**: **JMPL  $R_i$**  which  $R_i$  is used to store  $PA_i$
- To make a **RETURN**: **JMP R31**

# My ISA: Condition instructions

- $R_5 = @4500, R_6 = @4800$
- At **Main Program**, call **Proc1**, interrupt at @4101
  - **JMPL**  $R_5$  which  $R_5 = @4500$ 
    - $R_{31} \leftarrow PC$  ( $R_{31} = @4101$ )
    - $PC \leftarrow R_5$  ( $PC = @4500$ )
- At **Proc1**, call **Proc2**, interrupt at @4601
  - **ADDI**  $R_{30}, R_{30}, 4$
  - **STORE**  $R_{31}, (R_{30})$  ( $R_{30} = @4101$ )
  - **JMPL**  $R_6$ 
    - $R_{31} \leftarrow PC$  ( $R_{31} = @4601$ )
    - $PC \leftarrow R_6$  ( $PC = @4800$ )
- At **Proc2**, return **Proc1** at @4601
  - **JMP**  $R_{31}$ 
    - $PC \leftarrow R_{31}$  ( $PC = @4601$ )

- At **Proc1**, call **Proc2**, interrupt at @4651
  - **JMPL**  $R_6$ 
    - $R_{31} \leftarrow PC$  ( $R_{31} = @4651$ )
    - $PC \leftarrow R_6$  ( $PC = @4800$ )
- At **Proc2**, return **Proc1** at @4651
  - **JMP**  $R_{31}$ 
    - $PC \leftarrow R_{31}$  ( $PC = @4651$ )
- At **Proc1**, return **Main Program**
  - **LOAD**  $R_{31}, (R_{30})$  ( $R_{31} = @4101$ )
  - **SUBI**  $R_{30}, R_{30}, 4$
  - **JMP**  $R_{31}$ 
    - $PC := R_{31}$  ( $PC = @4101$ )



# MIPS64 Architecture

- *A typical General-Purpose Load-Store Architecture*
- *Registers*
  - 32 - 64 bit Integer registers (R0:R31)
    - R0 is always equal to 0 and the rest are general purpose
  - 32 Floating point registers (F0:F31)
    - Contains a single precision 32-bit or a double precision 64-bit
- *Data Types*
  - byte, half word, word, double word
- *Instructions*
  - Fixed length: 32-bit (4 bytes)
  - 3 simple instruction types ==> easy decode ==> pipeline implementing ==> performance enhance

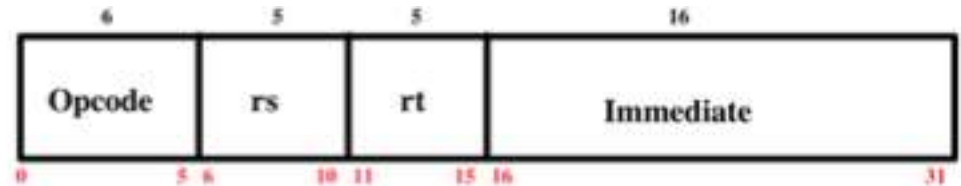
# MIPS64 Operations

- *Addressing modes for data*
  - Displacement (size 12-16 bits)
  - Immediate (size 8-16 bits)
  - Register indirect: placing 0 in the 16-bit displacement field
  - Absolute addressing: using R0 as the base register
- *Conditional branches*
  - Test the register source for zero or nonzero (compare equal)
  - Or compare 2 registers (compare less)
    - If satisfied, then jump  $PC + 4 + \text{immediate}$  (size 8 bits long)
  - jump, call, and return
- *Floats operations*
  - .S for single precision (32-bit)
  - .D for double precision (64-bit)

# MIP64 Instruction Format

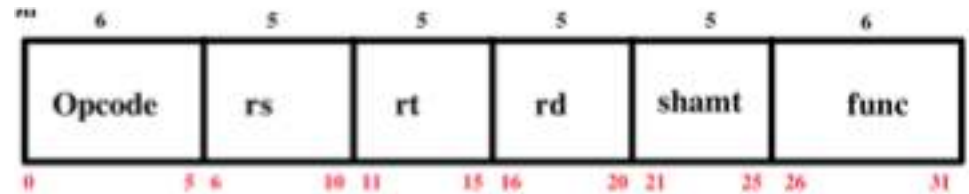
- **R instructions:** used when all data values are located in registers

- Using: **OP** rd, rs, rt
- Description:  $rd \leftarrow rs + rt$
- Example: **ADD** \$s1, \$s2, \$s3



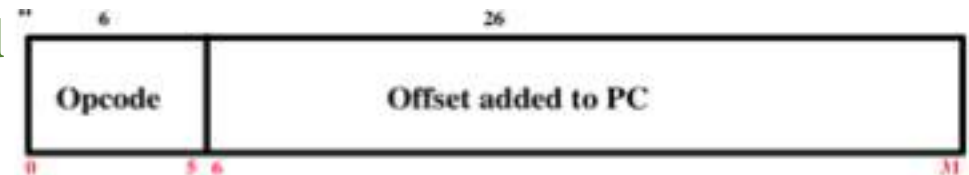
- **I instructions:** used when operating on an immediate value and register value

- Using: **OP** rt, rs, value
- Description:  $rt \leftarrow rs + \text{value}$
- Example: **ADDI** \$s1, \$s2, 100



- **J instructions:** used when a jump need to be performed

- Using: **OP** target\_add
- Description:  $PC \leftarrow \text{target\_add}$
- Example: **J** @1000





# MIPS64 Operations: Load and Store

Example instruction	Instruction name	Meaning
LD R1, 30(R2)	Load double word	$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[30 + \text{Regs}[R2]]$
LD R1, 1000(R0)	Load double word	$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[1000 + 0]$
LW R1, 30(R2)	Load word	$\text{Regs}[R1] \leftarrow 64 (\text{Mem}[30 + \text{Regs}[R2]]_0)^{32} \text{ ##}$ $\text{Mem}[30 + \text{Regs}[R2]]$
LB R1, 30(R2)	Load byte	$\text{Regs}[R1] \leftarrow 64 (\text{Mem}[30 + \text{Regs}[R2]]_0)^{56} \text{ ##}$ $\text{Mem}[30 + \text{Regs}[R2]]$
L.S F0, 30(R2)	Load FP single	$\text{Regs}[F0] \leftarrow 64 \text{ Mem}[30 + \text{Regs}[R2]] \text{ ##} 0^{32}$
L.D F0, 30(R2)	Load FP double	$\text{Regs}[F0] \leftarrow 64 \text{ Mem}[30 + \text{Regs}[R2]]$
SD R3, 500(R4)	Store double word	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow 64 \text{ Regs}[R3]$
SW R3, 500(R4)	Store word	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow 32 \text{ Regs}[R3]$
SH R3, 500(R4)	Store half word	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow 16 \text{ Regs}[R3]_{48..63}$
SB R3, 500(R4)	Store byte	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow 8 \text{ Regs}[R3]_{56..63}$
S.S F0, 500(R4)	Store FP single	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow 32 \text{ Regs}[F0]_{0..31}$
S.D F0, 500(R4)	Store FP double	$\text{Mem}[500 + \text{Regs}[R4]] \leftarrow 16 \text{ Regs}[F0]$

# MIPS64 Operations: A/L, Shift

Example instruction	Instruction name	Meaning
DADDU R1, R2, R3	Add unsigned	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + \text{Regs}[\text{R3}]$
DADDI R1, R1, #-3	Add immediate	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R1}] - 3$
LUI R1, #88	Load upper immediate	$\text{Regs}[\text{R1}] \leftarrow 032 \text{ \#\# } 88 \text{ \#\# } 016$
DSLL R1, R2, #3	Shift left logic	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] \ll 3$
DSLT R1, R2, R3	Set less than	If ( $\text{Regs}[\text{R2}] < \text{Regs}[\text{R3}]$ ) $\text{Regs}[\text{R1}] \leftarrow 1$ Else $\text{Regs}[\text{R1}] \leftarrow 0$

# MIPS64 Operations: Flow control

Example instruction	Instruction name	Meaning
J     name	Jump	$PC \leftarrow \text{name}; ((PC+4) \cdot 2^{25}) \leq \text{name} < ((PC+4) + 2^{25})$
JAL   name	Jump and link	$\text{Regs}[\text{R31}] \leftarrow PC+4; PC \leftarrow \text{name}; ((PC+4) \cdot 2^{25}) \leq \text{name} < ((PC+4) + 2^{25})$
JALR   R2	Jump and link register	$\text{Regs}[\text{R31}] \leftarrow PC+4; PC \leftarrow \text{Regs}[\text{R2}]$
JR     R3	Jump register	$PC \leftarrow \text{Regs}[\text{R3}]$
BEQZ   R4, name	Branch equal zero	If $(\text{Regs}[\text{R4}] == 0)$ $PC \leftarrow \text{name}; ((PC+4) \cdot 2^{15}) \leq \text{name} < ((PC+4) + 2^{15})$
BNE    R3, R4, name	Branch not equal zero	If $(\text{Regs}[\text{R3}] != \text{Regs}[\text{R4}])$ $PC \leftarrow \text{name}; ((PC+4) \cdot 2^{15}) \leq \text{name} < ((PC+4) + 2^{15})$
MOVZ R1, R2, R3	Conditional move if zero	If $(\text{Regs}[\text{R3}] == 0)$ $\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}]$



# Question?