

Instruction Set Architecture

Case Study: MIPS-R3000

Chapter 2 (3^{ed} or 4th Edition)

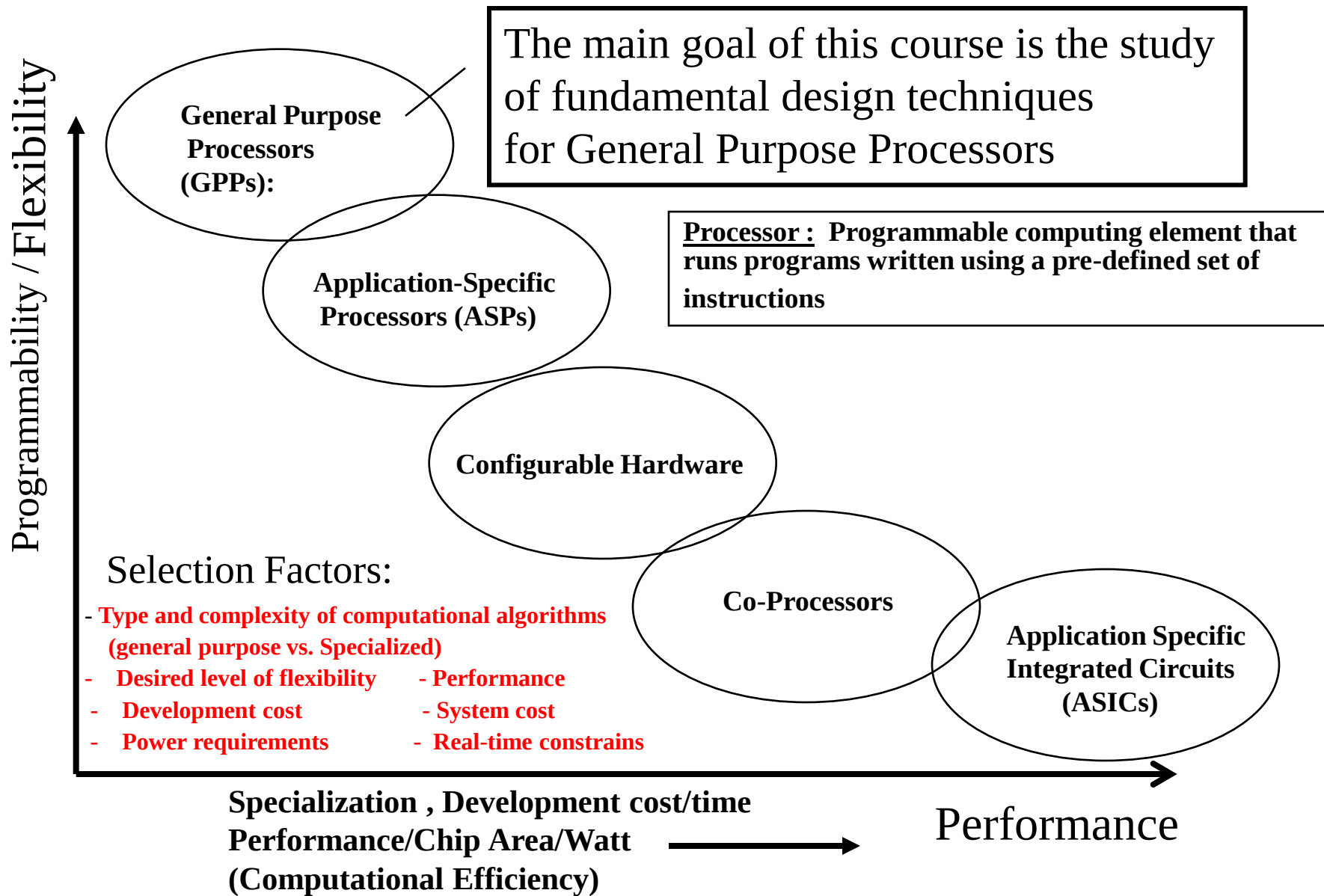
Outline

- ❖ Instruction Set Architecture Design
- ❖ CISC ver. RISC
- ❖ Overview of the MIPS Processor
- ❖ MIPS Assembly Language Programming

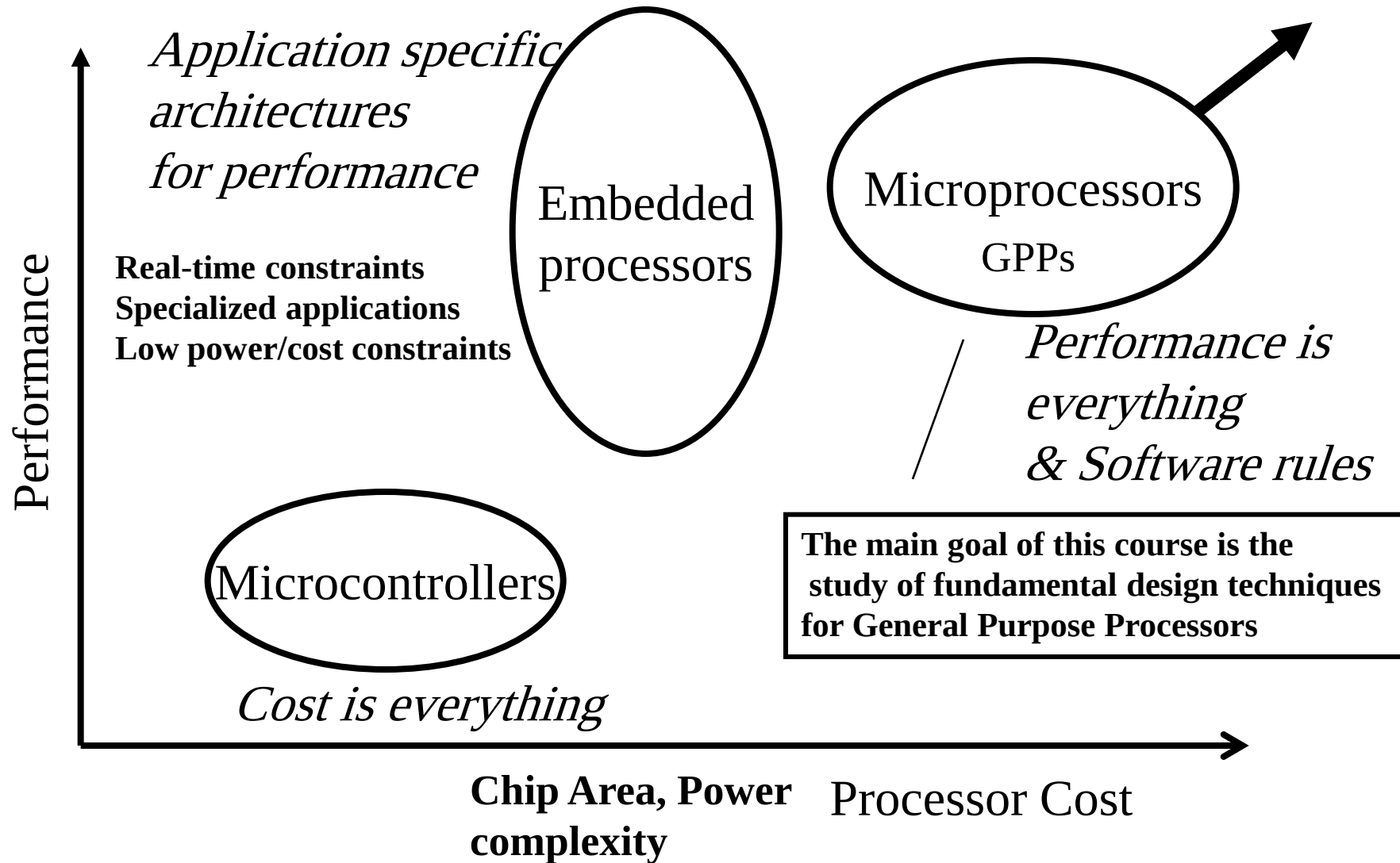
Computing Element Choices

- ❖ General Purpose Processors (GPPs): Intended for general purpose computing (desktops, servers, clusters..)
- ❖ Application-Specific Processors (ASPs): Processors with ISAs and architectural features tailored towards specific application domains
 - ✧ E.g Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs), Vector Processors??? ...
- ❖ Co-Processors: A hardware (hardwired) implementation of specific algorithms with limited programming interface (augment GPPs or ASPs)
- ❖ Configurable Hardware:
 - ✧ Field Programmable Gate Arrays (FPGAs)
 - ✧ Configurable array of simple processing elements
- ❖ Application Specific Integrated Circuits (ASICs): A custom VLSI hardware solution for a specific computational task
- ❖ The choice of one or more depends on a number of factors including:
 - Type and complexity of computational algorithm
(general purpose vs. Specialized)
 - Desired level of flexibility/
programmability
 - Development cost/time
 - Power requirements
 - Performance requirements
 - System cost
 - Real-time constraints

Computing Element Choices



The Processor Design Space



Processor = Programmable computing element that runs programs written using a pre-defined set of instructions

General Purpose Processor/Computer System Generations

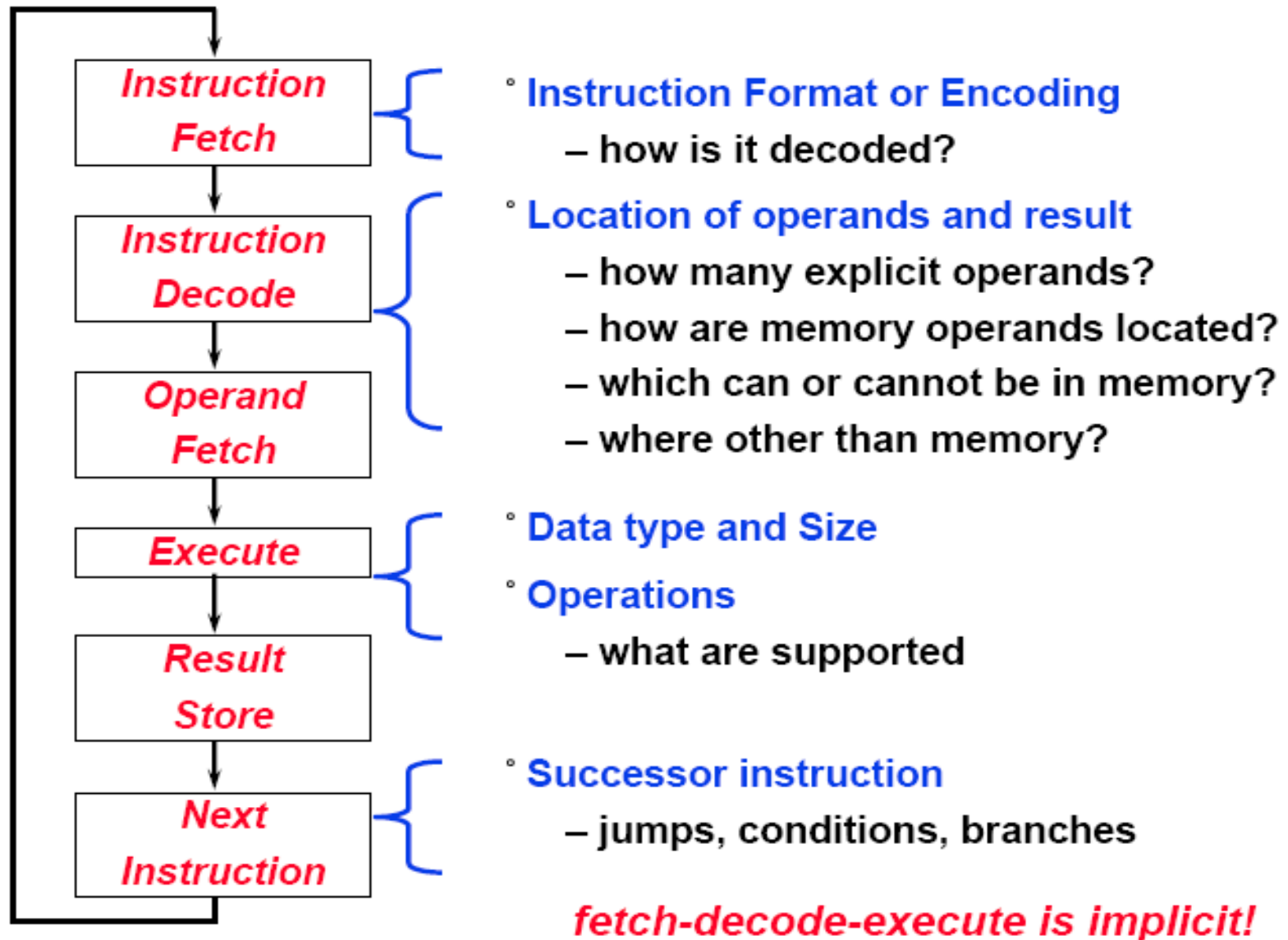
Classified according to implementation technology:

- ❖ The First Generation, 1946-59: Vacuum Tubes, Relays, Mercury Delay Lines:
 - ✧ ENIAC (Electronic Numerical Integrator and Computer): First electronic computer, 18000 vacuum tubes, 1500 relays, 5000 additions/sec (1944).
 - ✧ First stored program computer: EDSAC (Electronic Delay Storage Automatic Calculator), 1949.
- ❖ The Second Generation, 1959-64: Discrete Transistors.
 - ✧ e.g. IBM Main frames
- ❖ The Third Generation, 1964-75: Small and Medium-Scale Integrated (MSI) Circuits.
 - ✧ e.g. Main frames (IBM 360) , mini computers (DEC PDP-8, PDP-11).
- ❖ The Fourth Generation, 1975-Present: The Microcomputer. VLSI-based Microprocessors (single-chip processor)
 - ✧ First microprocessor: Intel's 4-bit 4004 (2300 transistors), 1970.
 - ✧ Personal Computer (PCs), laptops, PDAs, servers, clusters ...
 - ✧ Reduced Instruction Set Computer (RISC) 1984

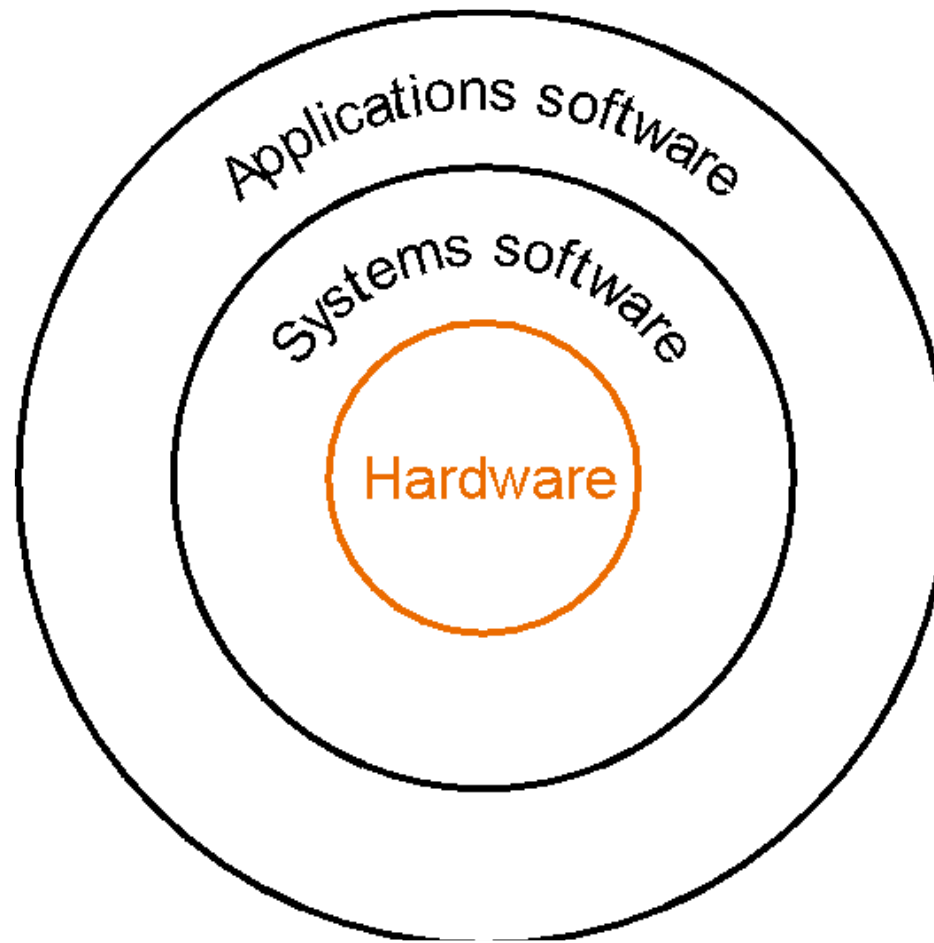
Common factor among all generations:

All target the The Von Neumann Computer Model or paradigm

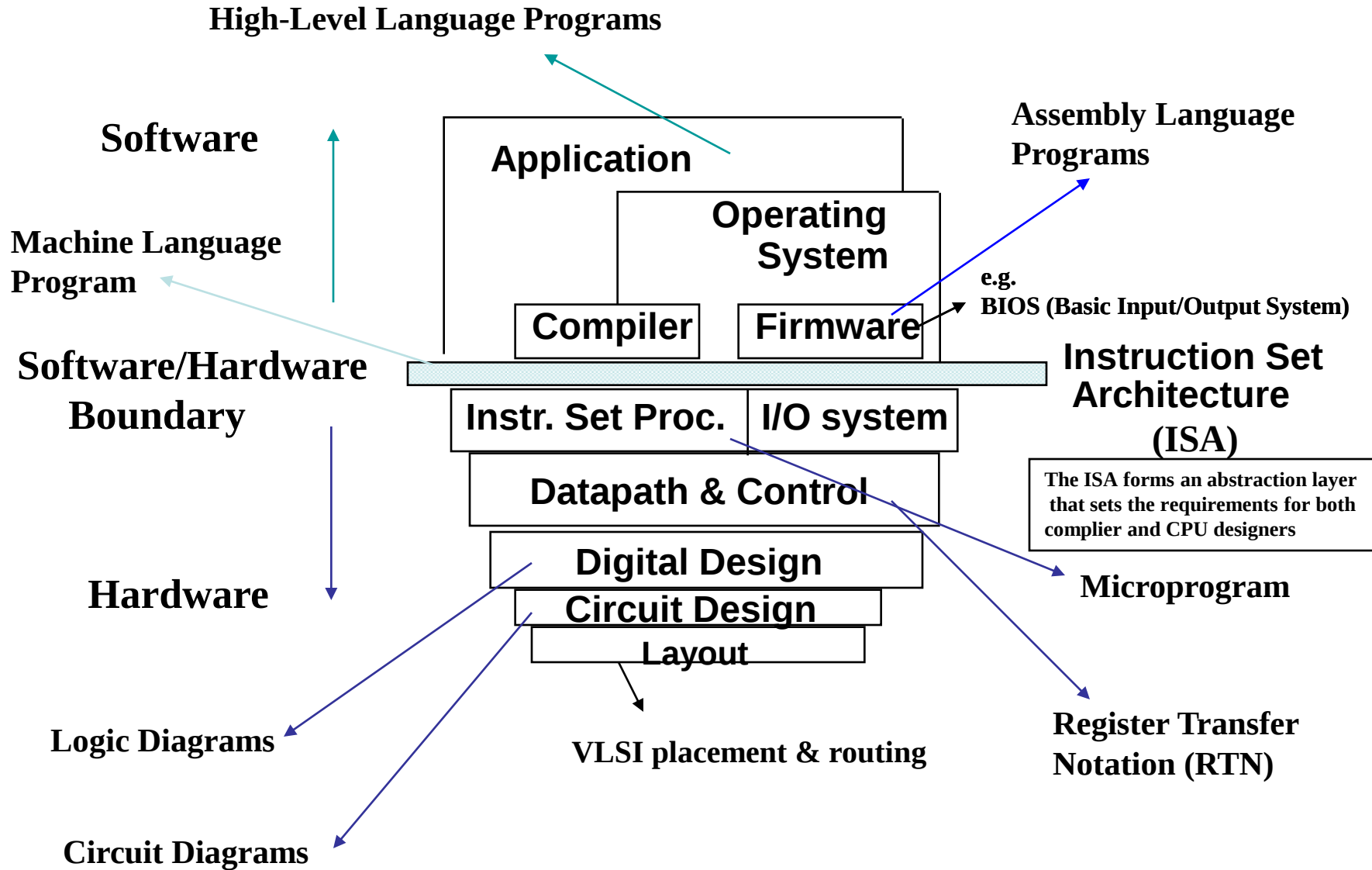
What Must be Specified?



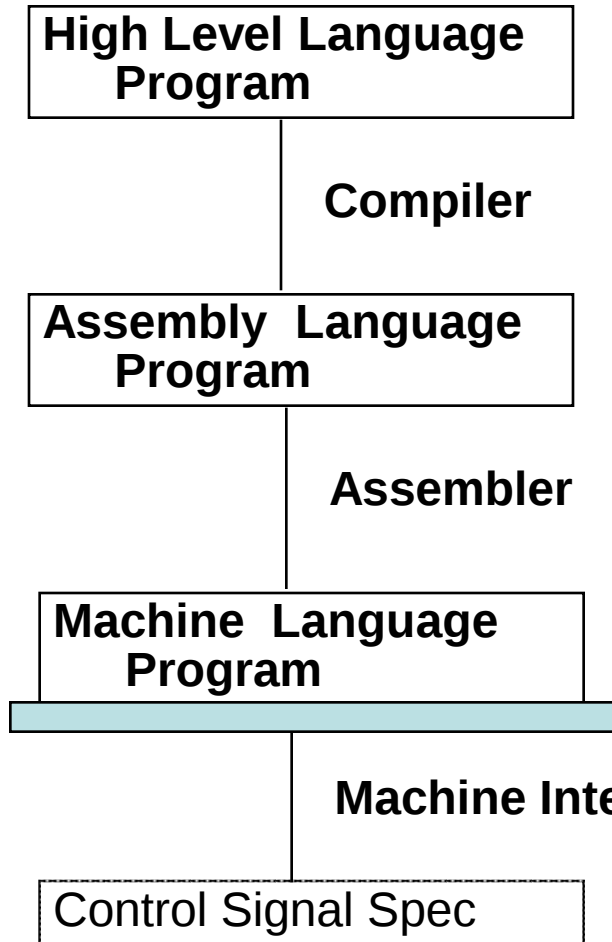
A Simplified View of The Software/Hardware Hierarchical Layers



Hierarchy of Computer Architecture



How to Speak Computer



```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)
```

```
10001100011000100000000000000000  
10001100111100100000000000000100  
10101100111100100000000000000000  
10101100011000100000000000000100
```

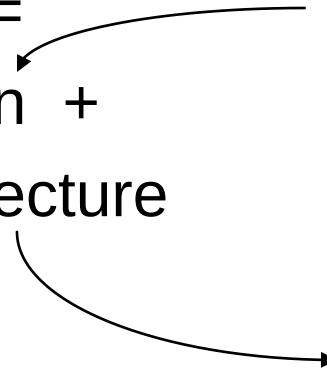
```
ALUOP[0:3] <= InstReg[9:11] & MASK
```

Need translation from application to physics

What is Computer Architecture?

Computer Architecture =
Machine Organization +
Instruction Set Architecture

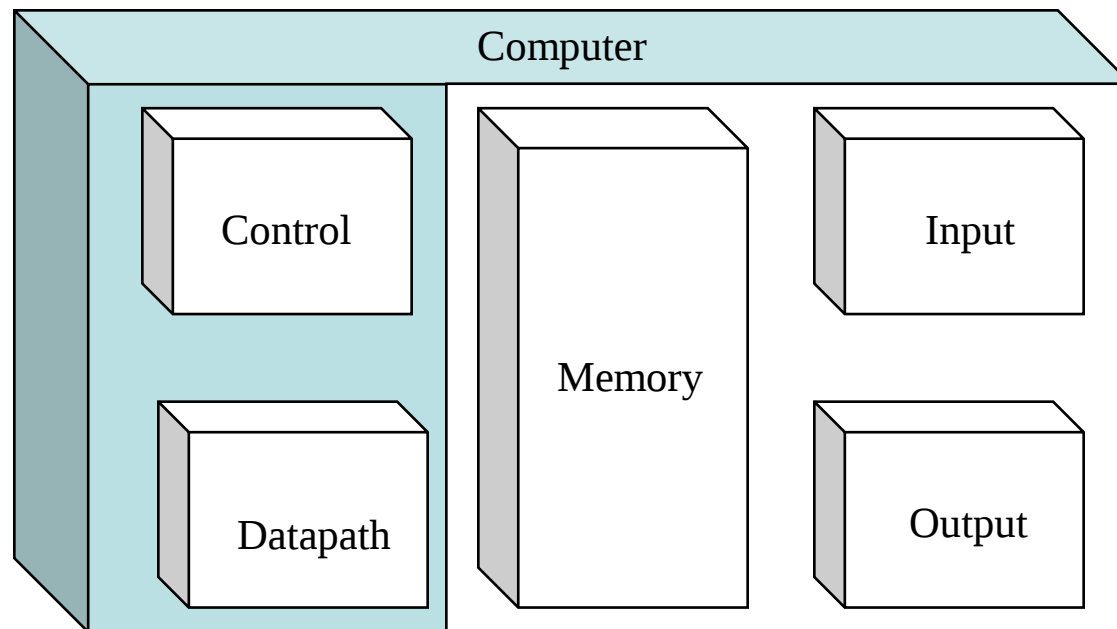
*What the machine
looks like*

A diagram consisting of two curved arrows. The first arrow starts from the text 'What the machine looks like' and points to the '+' sign in the equation. The second arrow starts from the text 'How you talk to the machine' and points to the 'Instruction Set Architecture' part of the equation.

How you talk to the machine

Computer Organization

- ❖ Once you have decided on an ISA, you must decide how to design the hardware to execute those programs written in the ISA as fast as possible (or as cheaply as possible, or using as little power as possible, ...).
- ❖ This must be done every time a new implementation of the architecture is released, with typically very different technological constraints



Instruction Set Architecture (ISA)

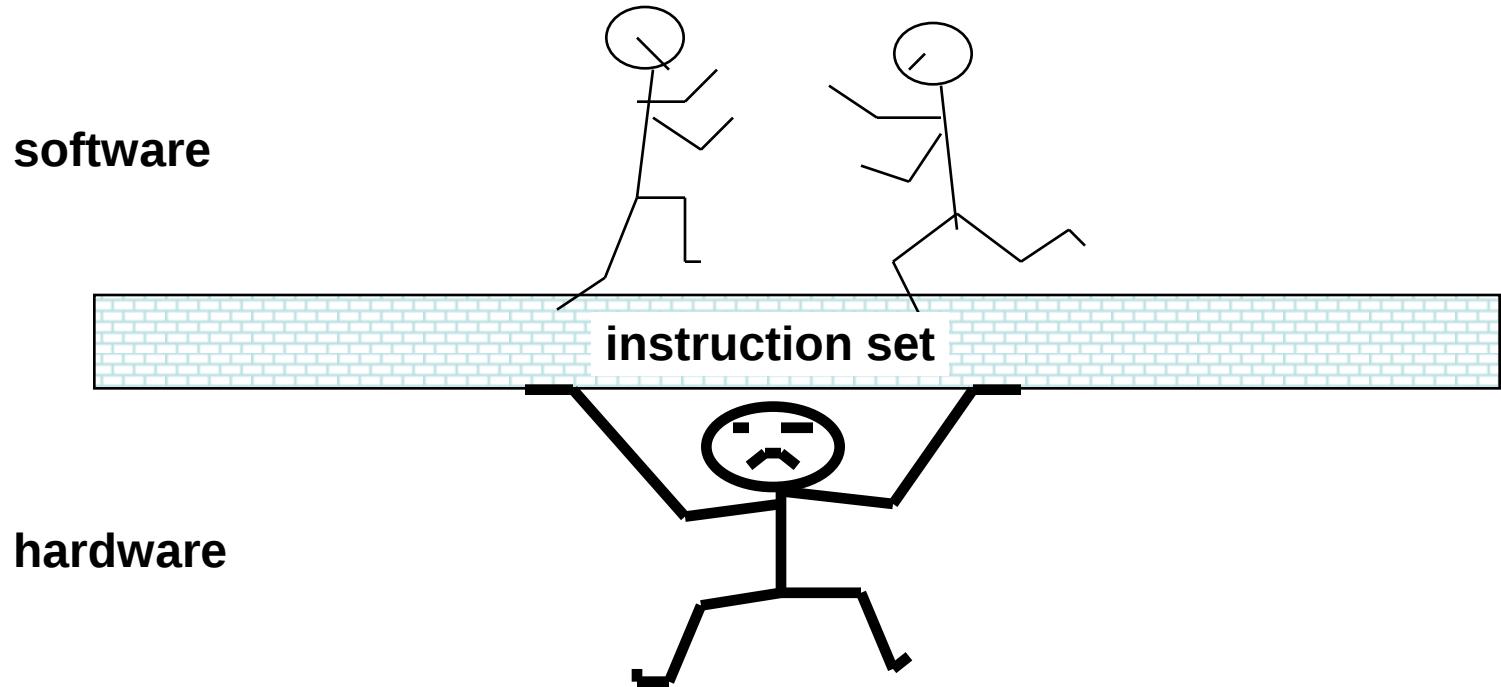
- ❖ Complete set of instructions used by a machine
- ❖ Abstract interface between the HW and lowest-level SW.
- ❖ An ISA includes the following ...
 - ✧ Instructions and Instruction Formats
 - Data Types, Encodings, and Representations
 - Programmable Storage: Registers and Memory
 - Addressing Modes: to address Instructions and Data
 - Handling Exceptional Conditions (like division by zero)

❖ Examples	(Versions)	First Introduced in
✧ Intel	(8086, 80386, Pentium, ...)	1978
✧ MIPS	(MIPS I, II, III, IV, V)	1986
✧ PowerPC	(601, 604, ...)	1993

The Instruction Set Architecture

- ❖ ISA is considered part of the SW
- ❖ Must be designed to survive changes in hardware technology, software technology, and application characteristic.
 - ✧ Is the agreed-upon interface between all the software that runs on the machine and the hardware that executes it.
- ❖ Advantages:
 - ✧ Different implementations of the same architecture
 - ✧ Easier to change than HW
 - ✧ Standardizes instructions, machine language bit patterns, etc.
- ❖ Disadvantage:
 - ✧ Sometimes prevents using new innovations

Instruction Set Architecture: Critical Interface



- ❖ Properties of a good abstraction
 - ✧ Lasts through many generations (portability)
 - ✧ Used in many different ways (generality)
 - ✧ Provides **convenient** functionality to higher levels
 - ✧ Permits an **efficient** implementation at lower levels

Basic ISA Classes

Accumulator:

1 address	add A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$
1+x address	addx A	$\text{acc} \leftarrow \text{acc} + \text{mem}[A + x]$

Stack:

0 address	add	$\text{tos} \leftarrow \text{tos} + \text{next}$
-----------	-----	--

General Purpose Register:

2 address	add A B	$\text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B)$
3 address	add A B C	$\text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C)$

Load/Store: (a modified form of GPR design)

3 address	load Ra Rb	$\text{Ra} \leftarrow \text{mem}[\text{Rb}]$
	add Ra Rb Rc	$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$
	store Ra Rb	$\text{mem}[\text{Rb}] \leftarrow \text{Ra}$

How to compare?

Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Number of Instructions

- Code sequence for $C = A + B$ for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C,R1	Add R3,R1,R2
Pop C			Store C,R3
Inst : 4	3	3	4

Stack machine: no general purpose registers - all operations are performed using the stack

Load-store machine: only load and store instructions reference memory
All ALU operations are performed using registers only.

Comparing Number of Bytes

Assumptions: 1 byte OP code; 4 byte memory address; 1 byte Reg. No.

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A 5	Load A 5	Load R1,A 6	Load R1,A 6
Push B 5	Add B 5	Add R1,B 6	Load R2,B 6
Add 1	Store C 5	Store C,R1 6	Add R3,R1,R2 4
Pop C 5			Store C,R3 6
Bytes: 16	15	18	22

Comparing Number of Memory Access

Assumptions: 1 memory access for OP code; 1 access per memory address

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A 2	Load A 2	Load R1,A 2	Load R1,A 2
Push B 2	Add B 2	Add R1,B 2	Load R2,B 2
Add 1	Store C 2	Store C,R1 2	Add R3,R1,R2 1
Pop C 2			Store C,R3 2
Total: 7	6	6	7

Now repeat the exercises (coding and comparisons) for a longer sequence:

$$A = (A + B * C) / (B^2 + C^2)$$

General Purpose Registers Dominate

- * Since 1975 all machines use general purpose registers
- * **Advantages of registers**
 - * registers are **faster** than memory
 - * registers are **easier for a compiler** to use

e.g., $(A*B) - (C*D) - (E*F)$ can do multiplies in any order
Stack is the most restrictive one
 - * registers can **hold variables**
 - **memory traffic is reduced**, so program is sped up
(registers are also faster than memory)
 - **code density improves** (since register named with fewer bits than memory location)
- * **Stack machines:** Burroughs B55/5700 mainframe, HP3000, Transputer, HP pocket calculators. With new Java machines, stack machines may make a comeback.

Addressing Modes: how data is accessed?

Addressing mode	Example	Meaning
Register	Add R4,R3	$R4 \leftarrow R4 + R3$
Immediate	Add R4,#3	$R4 \leftarrow R4 + 3$
Register indirect	Add R4,(R1)	$R4 \leftarrow R4 + \text{Mem}[R1]$
Displacement	Add R4,100(R1)	$R4 \leftarrow R4 + \text{Mem}[100 + R1]$
Indexed	Add R3,(R1+R2)	$R3 \leftarrow R3 + \text{Mem}[R1 + R2]$
Direct or absolute	Add R1,(100)	$R1 \leftarrow R1 + \text{Mem}[100]$
Memory indirect	Add R1,@(R3)	$R1 \leftarrow R1 + \text{Mem}[\text{Mem}[R3]]$
Auto-increment	Add R1,(R2)+	$R1 \leftarrow R1 + \text{Mem}[R2]; R2 \leftarrow R2 + d$
Auto-decrement	Add R1,-(R2)	$R2 \leftarrow R2 - d; R1 \leftarrow R1 + \text{Mem}[R2]$
Scaled	Add R1,d,100(R2)[R3]	$R1 \leftarrow R1 + \text{Mem}[100 + R2 + R3 * d]$

Typical Operations

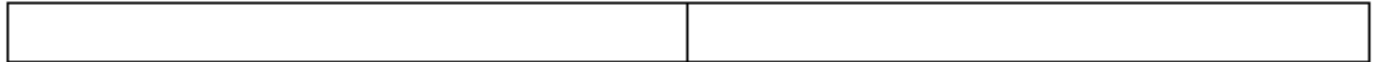
Data Movement	Load (from memory) memory-to-memory move input (from I/O device) push, pop (to/from stack)	Store (to memory) register-to-register move output (to I/O device)
Arithmetic	Data Types: (signed & unsigned) Integer (binary + decimal) (signed & unsigned) Floating Point Numbers Operations: Add, Subtract, Multiply, Divide	
Logical	Not, and, or, set, clear	
Shift	Arithmetic (& Logical) shift (left/right), rotate (left/right)	
Control (Jump/Branch)	unconditional, conditional	
Subroutine Linkage	call, return	
Interrupt	trap, return	
Synchronisation	test & set (atomic r-m-w)	
String	search, compare, translate	

Generic Examples of Instruction Formats

Variable:



Fixed:



Hybrid:



Key ISA decisions

❖ operations

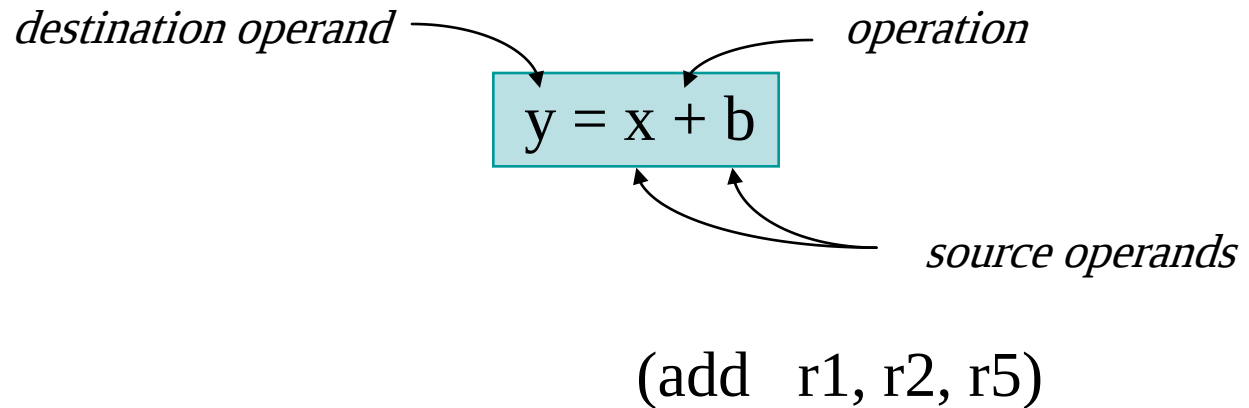
- how many?
- which ones

❖ operands

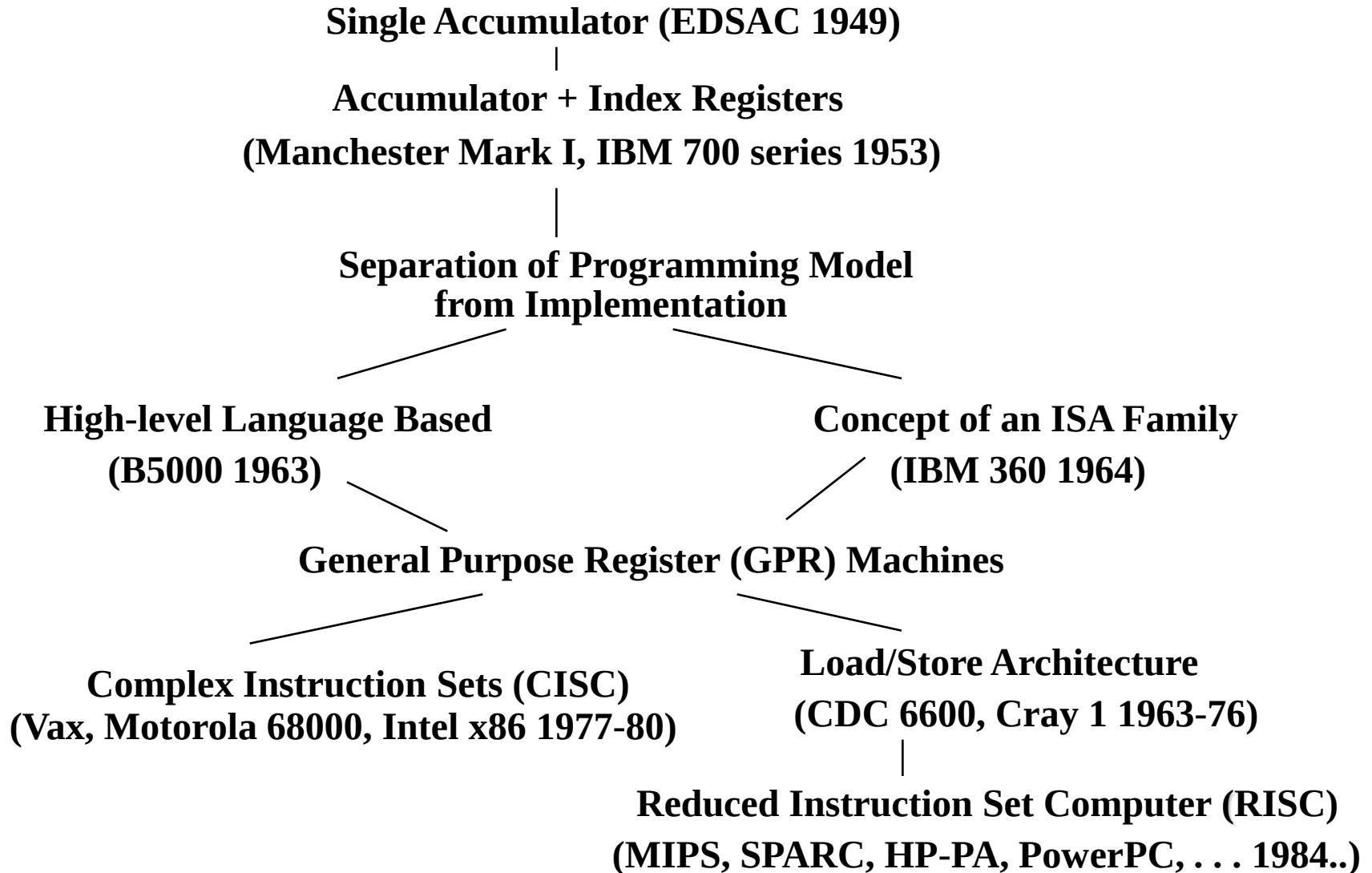
- how many?
- location
- types
- how to specify?

❖ instruction format

- size
- how many formats?
- **how many registers?**



Evolution of Instruction Set Architectures



What is CISC?

- ❖ CISC is an acronym for Complex Instruction Set Computer and are chips that are easy to program and which make efficient use of memory.
 - ✧ Since the earliest machines were programmed in assembly language and memory was slow and expensive, the CISC philosophy made sense
- ❖ Most common microprocessor designs such as the Intel 80x86 and Motorola 68K series followed the CISC philosophy.
- ❖ CISC was developed to make compiler development simpler
- ❖ CISC instructions sets some common attributes:
 - ✧ A 2-operand format, where instructions have a source and a destination. Register to register, register to memory, and memory to register commands. Multiple addressing modes for memory, including specialized modes for indexing through arrays
 - ✧ Variable length instructions where the length often varies according to the addressing mode
 - ✧ Instructions which require multiple clock cycles to execute.

CISC Characteristics

- ❖ Most CISC hardware architectures have several characteristics in common:
 - ✧ Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.
 - ✧ A small number of general purpose registers. This is the direct result of having instructions which can operate directly on memory and the limited amount of chip space not dedicated to instruction decoding, execution, and microcode storage.
 - ✧ Several special purpose registers. Many CISC designs set aside special registers for the stack pointer, interrupt handling, and so on. This can simplify the hardware design somewhat, at the expense of making the instruction set more complex.
 - ✧ Micro-programming is as easy as assembly language to implement, and much less expensive than hardwiring a control unit.
 - ✧ As each instruction became more capable, fewer instructions could be used to implement a given task. This made more efficient use of the relatively slow main memory.
 - ✧ Because micro-program instruction sets can be written to match the constructs of high-level languages, the compiler does not have to be as complicated.

CISC Disadvantages

- ❖ Designers soon realized that the CISC philosophy had its own problems, including:
 - ✧ Earlier generations of a processor family generally were contained as a subset in every new version - so instruction set & chip hardware become more complex with each generation of computers.
 - ✧ So that as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length - this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.
 - ✧ Many specialized instructions aren't used frequently enough to justify their existence -approximately 20% of the available instructions are used in a typical program.

Example CISC ISAs Motorola 680X0

18 addressing modes:

- ❖ Data register direct.
- ❖ Address register direct.
- ❖ Immediate.
- ❖ Absolute short.
- ❖ Absolute long.
- ❖ Address register indirect.
- ❖ Address register indirect with postincrement.
- ❖ Address register indirect with predecrement.
- ❖ Address register indirect with displacement.
- ❖ Address register indirect with index (8-bit).
- ❖ Address register indirect with index (base).
- ❖ Memory indirect postindexed.
- ❖ Memory indirect preindexed.
- ❖ Program counter indirect with index (8-bit).
- ❖ Program counter indirect with index (base).
- ❖ Program counter indirect with displacement.
- ❖ Program counter memory indirect postindexed.
- ❖ Program counter memory indirect preindexed.

Operand size:

- Range from 1 to 32 bits

Instruction Encoding:

- Instructions are stored in 16-bit words.
- the smallest instruction is 2- bytes (one word).
- The longest instruction is 5 words (10 bytes) in length.

Example CISC ISA: Intel 80386

12 addressing modes:

- ❖ Register.
- ❖ Immediate.
- ❖ Direct.
- ❖ Base.
- ❖ Base + Displacement.
- ❖ Index + Displacement.
- ❖ Scaled Index + Displacement.
- ❖ Based Index.
- ❖ Based Scaled Index.
- ❖ Based Index + Displacement.
- ❖ Based Scaled Index + Displacement.
- ❖ Relative.

Operand sizes:

- Can be 8, 16, 32, 48, 64, or 80 bits long.
 - Also supports string operations.
-

Instruction Encoding:

- The smallest instruction is **one** byte.
- The longest instruction is **12** bytes long.
- The first bytes generally contain the opcode, mode specifiers, and register fields.
- The remainder bytes are for address displacement and immediate data.

What is RISC?

❖ **RISC?**

RISC, or Reduced Instruction Set Computer. is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

❖ **Certain design features have been characteristic of most RISC processors:**

- ✧ one cycle execution time: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called PIPELINING
- ✧ pipelining: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- ✧ large number of registers: the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

RISC Attributes

- ❖ The main characteristics of CISC microprocessors are:
 - ✧ Extensive instructions.
 - ✧ Complex and efficient machine instructions.
 - ✧ Microencoding of the machine instructions.
 - ✧ Extensive addressing capabilities for memory operations.
 - ✧ Relatively few registers.

- ❖ In comparison, RISC processors are more or less the opposite of the above:
 - ✧ Reduced instruction set.
 - ✧ Less complex, simple instructions.
 - ✧ Hardwired control unit and machine instructions.
 - ✧ Few addressing schemes for memory operands with only two basic instructions, LOAD and STORE
 - ✧ Many symmetric registers which are organised into a register file.

RISC Disadvantages

- ❖ There is still considerable controversy among experts about the ultimate value of RISC architectures. Its proponents argue that RISC machines are both cheaper and faster, and are therefore the machines of the future.
- ❖ However, by making the hardware simpler, RISC architectures put a greater burden on the software. Is this worth the trouble because conventional microprocessors are becoming increasingly fast and cheap anyway?

Example RISC ISA: PowerPC

8 addressing modes:

- ❖ Register direct.
- ❖ Immediate.
- ❖ Register indirect.
- ❖ Register indirect with immediate index (loads and stores).
- ❖ Register indirect with register index (loads and stores).
- ❖ Absolute (jumps).
- ❖ Link register indirect (calls).
- ❖ Count register indirect (branches).

Operand sizes:

- Four operand sizes: 1, 2, 4 or 8 bytes.

Instruction Encoding:

- Instruction set has 15 different formats with many minor variations.
- All are **32** bits in length.

Example RISC ISA: SPARC

5 addressing modes:

- ❖ Register indirect with immediate displacement.
- ❖ Register indirect indexed by another register.
- ❖ Register direct.
- ❖ Immediate.
- ❖ PC relative.

Operand sizes:

- Four operand sizes: 1, 2, 4 or 8 bytes.
-

Instruction Encoding:

- Instruction set has 3 basic instruction formats with 3 minor variations.
- All are **32** bits in length.

CISC versus RISC Summary

CISC

Emphasis on hardware

Includes multi-clock
complex instructions

Memory-to-memory:
"LOAD" and "STORE"
incorporated in instructions

Small code sizes,
high cycles per second

Transistors used for storing
complex instructions

RISC

Emphasis on software

Single-clock,
reduced instruction only

Register to register:
"LOAD" and "STORE"
are independent instructions

Low cycles per second,
large code sizes

Spends more transistors
on memory registers

Summary of Design Principles

1. Simplicity favors regularity

- ✧ Simple instructions dominate the instruction frequency
 - So design them to be simple and regular, and make them fast
 - Use general-purpose registers uniformly across instructions
- ✧ Fix the size of instructions (simplifies fetching & decoding)
- ✧ Fix the number of operands per instruction
 - Three operands is the natural number for a typical instruction

2. Smaller is faster

- ✧ Limit the number of registers for faster access (typically 32)

3. Make the common case fast

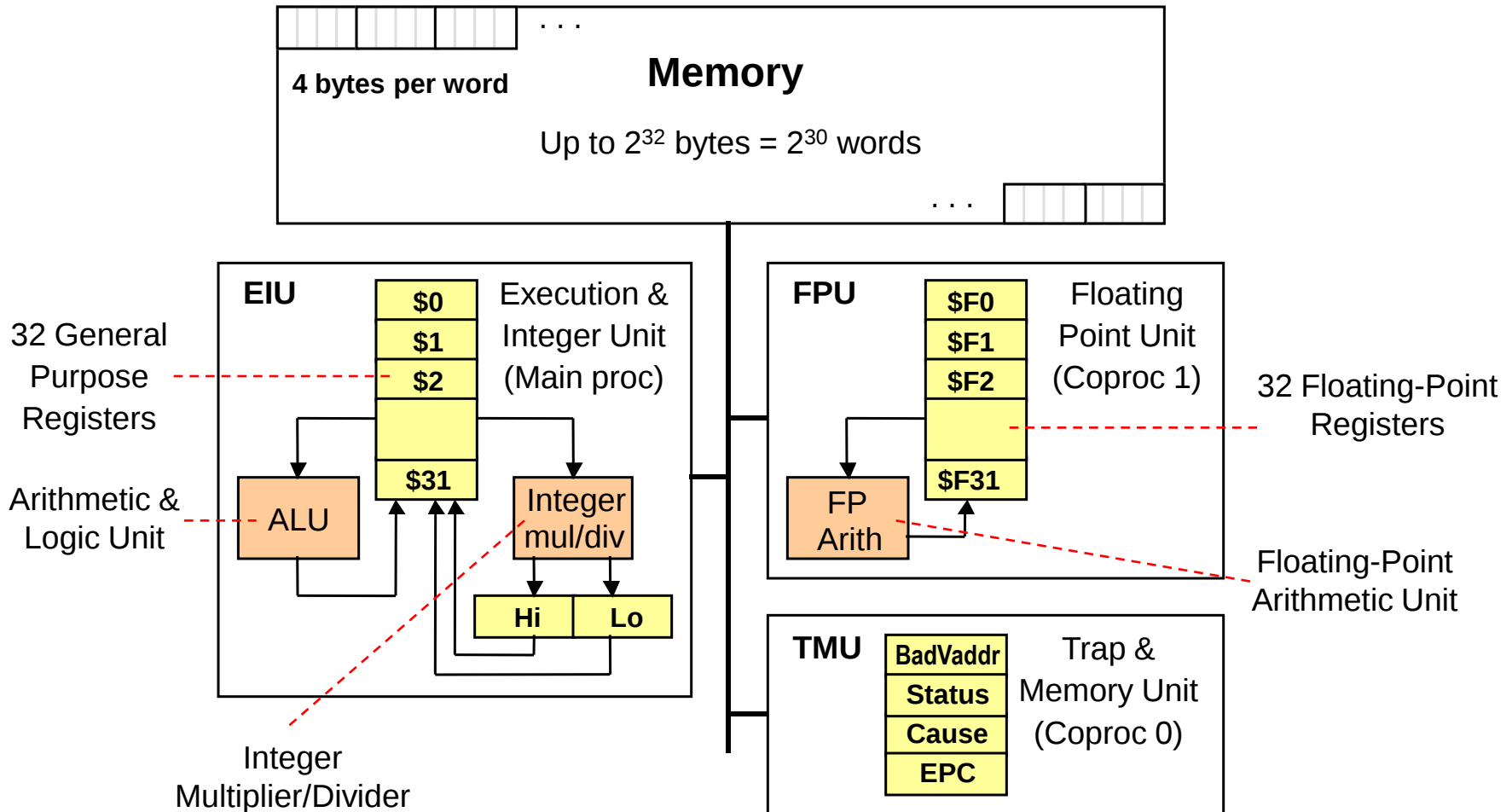
- ✧ Include constants inside instructions (faster than loading them)
- ✧ Design most instructions to be register-to-register

4. Good design demands good compromises

- ✧ Having one-size formats is better than variable-size formats, even though it limits the size of the immediate constants

Overview of the MIPS Processor

Logical View of the MIPS Processor



Overview of the MIPS Registers

❖ 32 General Purpose Registers (GPRs)

- ✧ 32-bit registers are used in MIPS32
- ✧ Register 0 is always zero
- ✧ Any value written to R0 is discarded

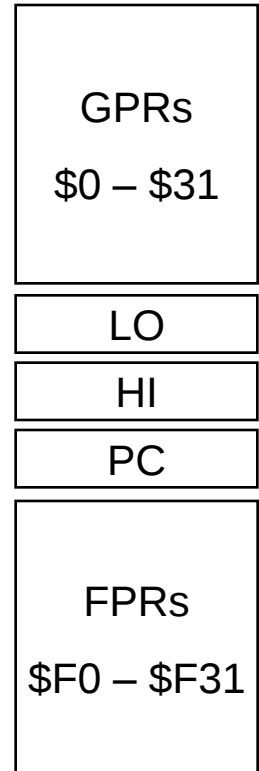
❖ Special-purpose registers LO and HI

- ✧ Hold results of integer multiply and divide

❖ Special-purpose program counter PC

❖ 32 Floating Point Registers (FPRs)

- ✧ Floating Point registers can be either 32-bit or 64-bit
- ✧ A pair of registers is used for double-precision floating-point



MIPS General-Purpose Registers

❖ 32 General Purpose Registers (GPRs)

- ✧ Assembler uses the dollar notation to name registers
 - \$0 is register 0, \$1 is register 1, ..., and \$31 is register 31
- ✧ All registers are 32-bit wide in MIPS32
- ✧ Register \$0 is always zero
 - Any value written to \$0 is discarded

❖ Software conventions

- ✧ Software defines names to all registers
 - To standardize their use in programs
- ✧ \$8 - \$15 are called \$t0 - \$t7
 - Used for **temporary** values
- ✧ \$16 - \$23 are called \$s0 - \$s7

\$0 = \$zero	\$16 = \$s0
\$1 = \$at	\$17 = \$s1
\$2 = \$v0	\$18 = \$s2
\$3 = \$v1	\$19 = \$s3
\$4 = \$a0	\$20 = \$s4
\$5 = \$a1	\$21 = \$s5
\$6 = \$a2	\$22 = \$s6
\$7 = \$a3	\$23 = \$s7
\$8 = \$t0	\$24 = \$t8
\$9 = \$t1	\$25 = \$t9
\$10 = \$t2	\$26 = \$k0
\$11 = \$t3	\$27 = \$k1
\$12 = \$t4	\$28 = \$gp
\$13 = \$t5	\$29 = \$sp
\$14 = \$t6	\$30 = \$fp
\$15 = \$t7	\$31 = \$ra

MIPS Register Conventions

- ❖ Assembler can refer to registers by name or by number
 - ✧ It is easier for you to remember registers by name
 - ✧ Assembler converts register name to its corresponding number

Name	Register	Usage
\$zero	\$0	Always 0 (forced by hardware)
\$at	\$1	Reserved for assembler use
\$v0 – \$v1	\$2 – \$3	Result values of a function
\$a0 – \$a3	\$4 – \$7	Arguments of a function
\$t0 – \$t7	\$8 – \$15	Temporary Values
\$s0 – \$s7	\$16 – \$23	Saved registers (preserved across call)
\$t8 – \$t9	\$24 – \$25	More temporaries
\$k0 – \$k1	\$26 – \$27	Reserved for OS kernel
\$gp	\$28	Global pointer (points to global data)
\$sp	\$29	Stack pointer (points to top of stack)
\$fp	\$30	Frame pointer (points to stack frame)
\$ra	\$31	Return address (used by jal for function call)

Instruction Formats

❖ All instructions are 32-bit wide, Three instruction formats:

❖ Register (R-Type)

✧ Register-to-register instructions

✧ Op: operation code specifies the format of the instruction



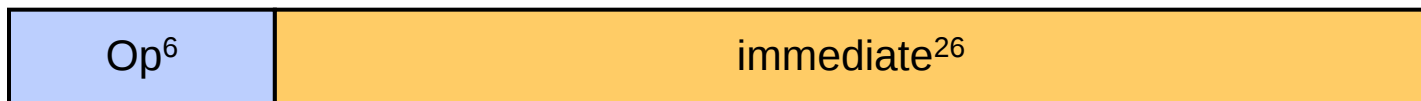
❖ Immediate (I-Type)

✧ 16-bit immediate constant is part in the instruction

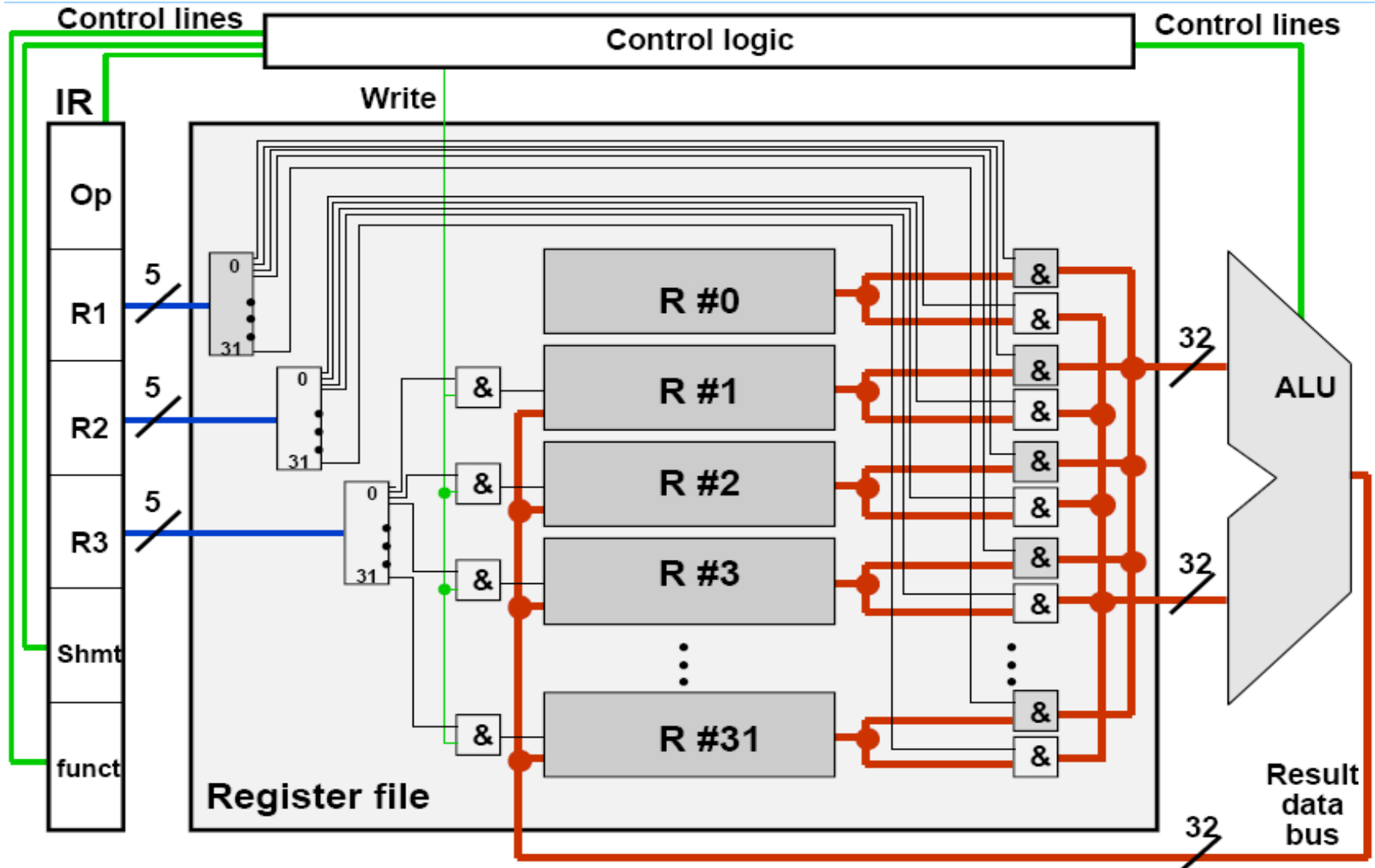


❖ Jump (J-Type)

✧ Used by jump instructions



Register File



MIPS Five Addressing Modes

1 Register Addressing:

Where the operand is a register (R-Type)

2 Immediate Addressing:

Where the operand is a constant in the instruction (I-Type, ALU)

3 Base or Displacement Addressing:

Where the operand is at the memory location whose address is the sum of a register and a constant in the instruction (I-Type, load/store)

4 PC-Relative Addressing:

Where the address is the sum of the PC and the 16-address field in the instruction shifted left 2 bits. (I-Type, branches)

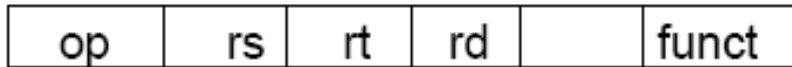
5 Pseudodirect Addressing:

Where the jump address is the 26-bit jump target from the instruction shifted left 2 bits concatenated with the 4 upper bits of the PC (J-Type)

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

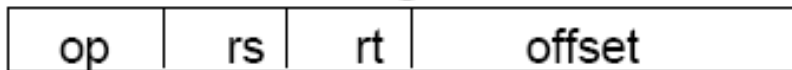
1. Register addressing



Register

word **operand**

2. Base addressing

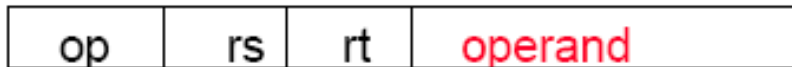


Memory

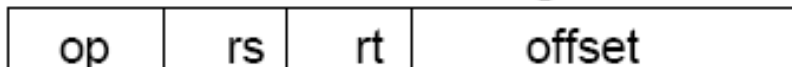
word or byte **operand**

base register

3. Immediate addressing



4. PC-relative addressing

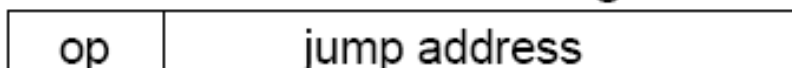


Memory

branch destination **instruction**

Program Counter (PC)

5. Pseudo-direct addressing



Memory

jump destination **instruction**

Program Counter (PC)

MIPS R-Type (ALU) Instruction Fields

R-Type: All ALU instructions that use three registers

	1st operand	2nd operand	Destination		
OP	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

❖ **op:** Opcode, basic operation of the instruction.

✧ For R-Type $op = 0$

Rs, rt , rd
are register specifier fields

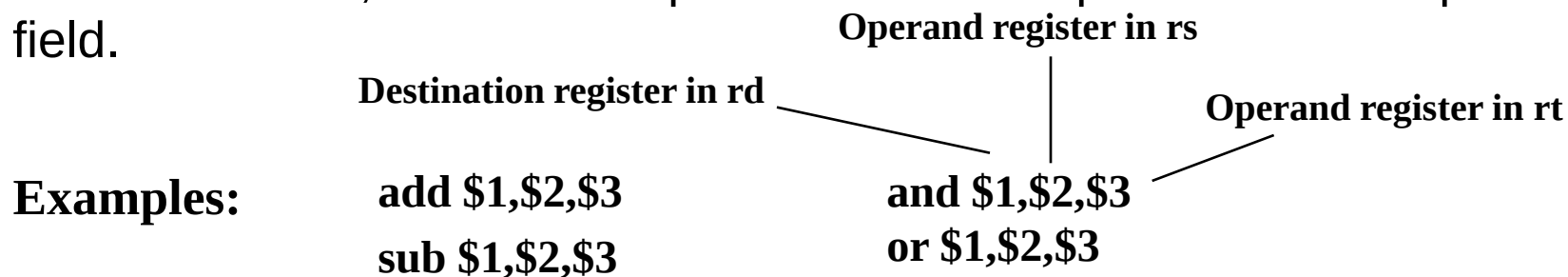
❖ **rs:** The first register source operand.

❖ **rt:** The second register source operand.

❖ **rd:** The register destination operand.

❖ **shamt:** Shift amount used in constant shift operations.

❖ **funct:** Function, selects the specific variant of operation in the op field.



R-Type = Register Type
Register Addressing used (Mode 1)

MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value
I-Type is also used for Loads/stores, conditional branches.

	1st operand	Destination	2nd operand
OP	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- ❖ **op**: Opcode, operation of the instruction.
- ❖ **rs**: The register source operand.
- ❖ **rt**: The result destination register.
- ❖ **immediate**: Constant second operand for ALU instruction.

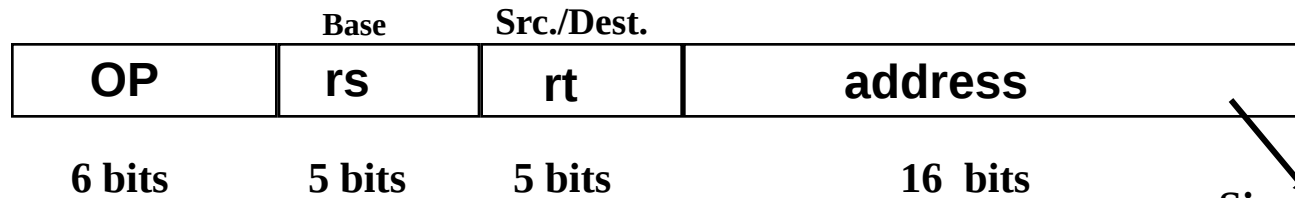
Examples:

add immediate:	addi \$1,\$2,100	
and immediate	andi \$1,\$2,10	

Result register in rt Source operand register in rs Constant operand in immediate

I-Type = Immediate Type
Immediate Addressing used (Mode 2)

MIPS Load/Store I-Type Instruction Fields



Signed address
offset in bytes

- ❖ **op**: Opcode, operation of the instruction.
 - ✧ For load op = 35, for store op = 43.
- ❖ **rs**: The register containing memory base address.
- ❖ **rt**: For loads, the destination register. For stores, the source register of value to be stored.
- ❖ **address**: 16-bit memory address offset in bytes added to base register.

Examples:

Store word:

sw \$3, 500(\$4)

Load word:

lw \$1, 32(\$2)

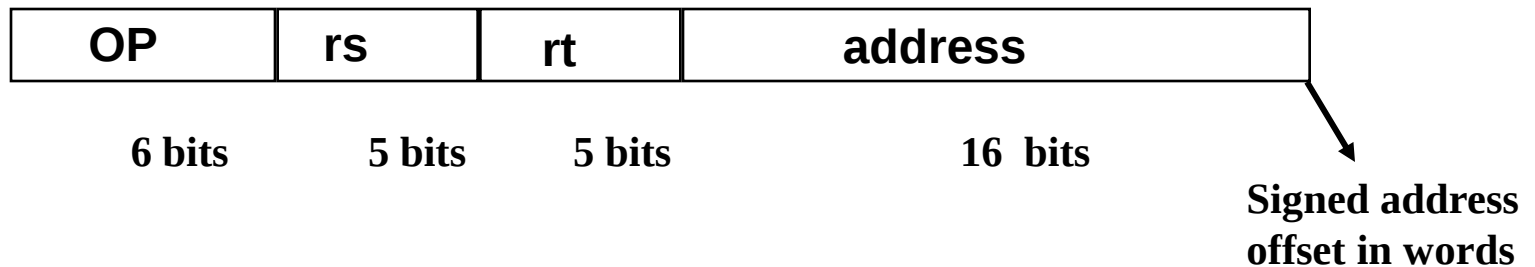
Destination register in rt

Offset

base register in rs

Base or Displacement Addressing used (Mode 3)

MIPS Branch I-Type Instruction Fields



- ❖ **op**: Opcode, operation of the instruction.
- ❖ **rs**: The first register being compared
- ❖ **rt**: The second register being compared.
- ❖ **address**: 16-bit memory address branch target offset in words added to PC to form branch address.

Examples:

Branch on equal	beq \$1,\$2,100
Branch on not equal	bne \$1,\$2,100

Register in rs Register in rt offset in bytes equal to instruction address field x 4

Added to PC to form branch target

PC-Relative Addressing used (Mode 4)

MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal



6 bits

26 bits

Jump target
in words

❖ **op:** Opcode, operation of the instruction.

✧ Jump j op = 2

✧ Jump and link jal op = 3

❖ **jump target:** jump memory address in words.

Jump memory address in bytes equal to
instruction field jump target x 4

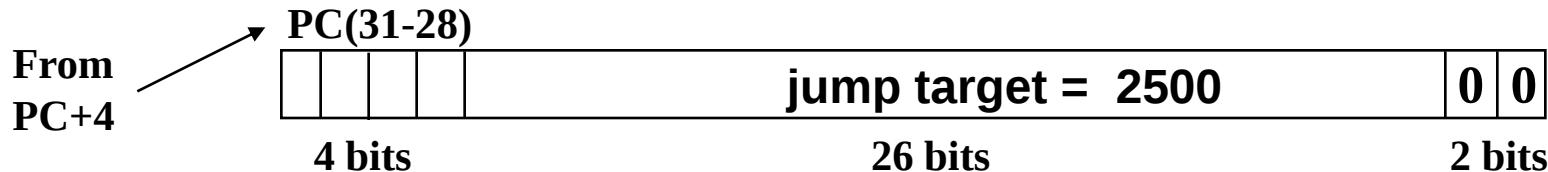
Examples: Jump

j 10000

Jump and link

jal 10000

Effective 32-bit jump address: PC(31-28),jump_target,00



J-Type = Jump Type

Pseudodirect Addressing used (Mode 5)

Instruction Categories

❖ Integer Arithmetic

- ✧ Arithmetic, logical, and shift instructions

❖ Data Transfer

- ✧ Load and store instructions that access memory
- ✧ Data movement and conversions

❖ Jump and Branch

- ✧ Flow-control instructions that alter the sequential sequence

❖ Floating Point Arithmetic

- ✧ Instructions that operate on floating-point registers

❖ Miscellaneous

- ✧ Instructions that transfer control to/from exception handlers
- ✧ Memory management instructions

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Alternative Architecture

R-Type Format



❖ **Op**: operation code (opcode)

- ✧ Specifies the operation of the instruction
- ✧ Also specifies the format of the instruction

❖ **funct**: function code – extends the opcode

- ✧ Up to $2^6 = 64$ functions can be defined for the same opcode
- ✧ MIPS uses opcode 0 to define R-type instructions

❖ Three Register Operands (common to many instructions)

- ✧ **Rs**, **Rt**: first and second source operands
- ✧ **Rd**: destination operand
- ✧ **sa**: the shift amount used by shift instructions

Integer Add / Subtract Instructions

Instruction	Meaning	R-Type Format					
add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x20
addu \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x21
sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x22
subu \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x23

❖ **add & sub:** overflow causes an **arithmetic exception**

✧ In case of overflow, result is not written to destination register

❖ **addu & subu:** same operation as **add & sub**

✧ However, no arithmetic exception can occur

✧ **Overflow is ignored**

❖ Many programming languages ignore overflow

✧ The **+** operator is translated into **addu**

✧ The **−** operator is translated into **subu**

Addition/Subtraction Example

- ❖ Consider the translation of: $f = (g+h) - (i+j)$
- ❖ Compiler allocates registers to variables
 - ✧ Assume that f , g , h , i , and j are allocated registers $\$s0$ thru $\$s4$
 - ✧ Called the **saved** registers: $\$s0 = \16 , $\$s1 = \17 , ..., $\$s7 = \23

- ❖ Translation of: $f = (g+h) - (i+j)$

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

- ✧ Temporary results are stored in $\$t0 = \8 and $\$t1 = \9
- ❖ Translate: **addu \$t0, \$s1, \$s2** to binary code

- ❖ Solution:

op	rs = \$s1	rt = \$s2	rd = \$t0	sa	func
000000	10001	10010	01000	00000	100001

Logical Bitwise Operations

❖ Logical bitwise operations: **and**, **or**, **xor**, **nor**

x	y	$x \text{ and } y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \text{ or } y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \text{ xor } y$
0	0	0
0	1	1
1	0	1
1	1	0

x	y	$x \text{ nor } y$
0	0	1
0	1	0
1	0	0
1	1	0

- ❖ AND instruction is used to clear bits: **$x \text{ and } 0 = 0$**
- ❖ OR instruction is used to set bits: **$x \text{ or } 1 = 1$**
- ❖ XOR instruction is used to toggle bits: **$x \text{ xor } 1 = \text{not } x$**
- ❖ NOR instruction can be used as a NOT, how?

✧ **`nor $s1,$s2,$s2`** is equivalent to **`not $s1,$s2`**

Logical Bitwise Instructions

Instruction	Meaning	R-Type Format					
and \$s1, \$s2, \$s3	$\$s1 = \$s2 \& \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x24
or \$s1, \$s2, \$s3	$\$s1 = \$s2 \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x25
xor \$s1, \$s2, \$s3	$\$s1 = \$s2 \wedge \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x26
nor \$s1, \$s2, \$s3	$\$s1 = \sim(\$s2 \$s3)$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x27

❖ Examples:

Assume $\$s1 = 0xabcd1234$ and $\$s2 = 0xffff0000$

and \$s0, \$s1, \$s2 # \$s0 = 0xabcd0000

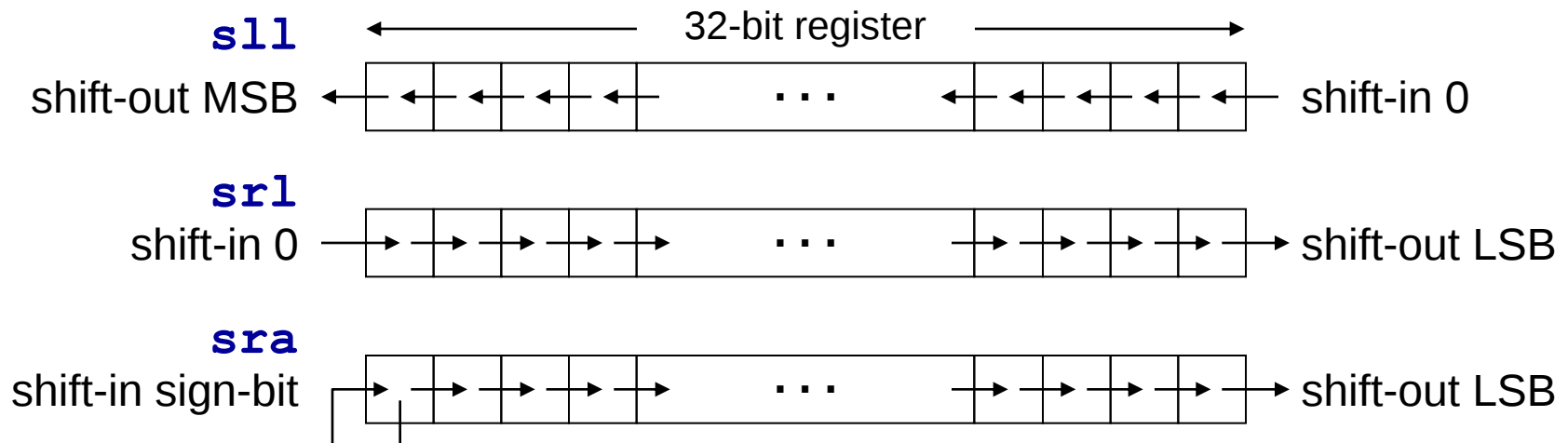
or \$s0, \$s1, \$s2 # \$s0 = 0xffff1234

xor \$s0, \$s1, \$s2 # \$s0 = 0x54321234

nor \$s0, \$s1, \$s2 # \$s0 = 0x0000edcb

Shift Operations

- ❖ Shifting is to move all the bits in a register left or right
- ❖ Shifts by a **constant** amount: **sll**, **srl**, **sra**
 - ✧ **sll/srl** mean **shift left/right logical** by a constant amount
 - ✧ The **5-bit shift amount** field is used by these instructions
 - ✧ **sra** means **shift right arithmetic** by a constant amount
 - ✧ The **sign-bit** (rather than 0) is shifted from the left



Shift Instructions

Instruction		Meaning	R-Type Format					
sll	\$s1,\$s2,10	\$s1 = \$s2 << 10	op = 0	rs = 0	rt = \$s2	rd = \$s1	sa = 10	f = 0
srl	\$s1,\$s2,10	\$s1 = \$s2 >>> 10	op = 0	rs = 0	rt = \$s2	rd = \$s1	sa = 10	f = 2
sra	\$s1, \$s2, 10	\$s1 = \$s2 >> 10	op = 0	rs = 0	rt = \$s2	rd = \$s1	sa = 10	f = 3
sllv	\$s1,\$s2,\$s3	\$s1 = \$s2 << \$s3	op = 0	rs = \$s3	rt = \$s2	rd = \$s1	sa = 0	f = 4
srlv	\$s1,\$s2,\$s3	\$s1 = \$s2 >>> \$s3	op = 0	rs = \$s3	rt = \$s2	rd = \$s1	sa = 0	f = 6
srav	\$s1,\$s2,\$s3	\$s1 = \$s2 >> \$s3	op = 0	rs = \$s3	rt = \$s2	rd = \$s1	sa = 0	f = 7

❖ Shifts by a **variable** amount: **sllv**, **srlv**, **srav**

✧ Same as **sll**, **srl**, **sra**, but a register is used for shift amount

❖ Examples: assume that \$s2 = 0xabcd1234, \$s3 = 16

sll \$s1,\$s2,8 \$s1 = \$s2<<8 \$s1 = 0xcd123400

sra \$s1,\$s2,4 \$s1 = \$s2>>4 \$s1 = 0xfabcd123

srlv \$s1,\$s2,\$s3 \$s1 = \$s2>>>\$s3 \$s1 = 0x0000abcd



op=000000	rs=\$s3=10011	rt=\$s2=10010	rd=\$s1=10001	sa=00000	f=000110
-----------	---------------	---------------	---------------	----------	----------

Binary Multiplication

- ❖ Shift-left (**sll**) instruction can perform multiplication
 - ✧ When the multiplier is a power of 2
- ❖ You can factor any binary number into powers of 2
 - ✧ Example: multiply **\$s1** by 36
 - Factor 36 into (4 + 32) and use distributive property of multiplication
 - ✧ $\$s2 = \$s1 * 36 = \$s1 * (4 + 32) = \$s1 * 4 + \$s1 * 32$

```
sll    $t0, $s1, 2      ; $t0 = $s1 * 4
sll    $t1, $s1, 5      ; $t1 = $s1 * 32
addu   $s2, $t0, $t1    ; $s2 = $s1 * 36
```

Your Turn . . .

Multiply \$s1 by 26, using shift and add instructions

Hint: $26 = 2 + 8 + 16$

```
sll    $t0, $s1, 1        ; $t0 = $s1 * 2
sll    $t1, $s1, 3        ; $t1 = $s1 * 8
addu   $s2, $t0, $t1      ; $s2 = $s1 * 10
sll    $t0, $s1, 4        ; $t0 = $s1 * 16
addu   $s2, $s2, $t0      ; $s2 = $s1 * 26
```

Multiply \$s1 by 31, Hint: $31 = 32 - 1$

```
sll    $s2, $s1, 5        ; $s2 = $s1 * 32
subu   $s2, $s2, $s1      ; $s2 = $s1 * 31
```

Integer Multiplication & Division

❖ Consider $a \times b$ and a/b where a and b are in $\$s1$ and $\$s2$

✧ Signed multiplication: `mult $s1,$s2`

✧ Unsigned multiplication: `multu $s1,$s2`

✧ Signed division: `div $s1,$s2`

✧ Unsigned division: `divu $s1,$s2`

❖ For multiplication, result is 64 bits

✧ LO = low-order 32-bit and HI = high-order 32-bit

❖ For division

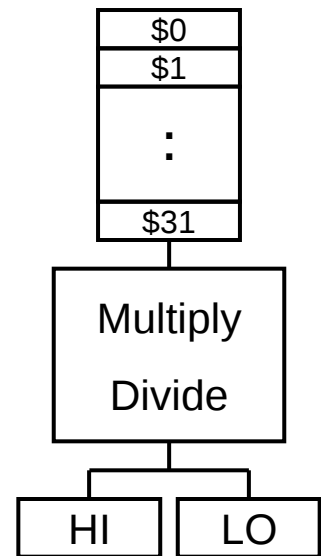
✧ LO = 32-bit quotient and HI = 32-bit remainder

✧ If divisor is 0 then result is unpredictable

❖ Moving data

✧ `mflo rd` (move from LO to rd), `mfhi rd` (move from HI to rd)

✧ `mtlo rs` (move to LO from rs), `mthi rs` (move to HI from rs)



Integer Multiply/Divide Instructions

Instruction	Meaning	Format					
mult rs, rt	hi, lo = $rs \times rt$	$op^6 = 0$	rs^5	rt^5	0	0	0x18
multu rs, rt	hi, lo = $rs \times rt$	$op^6 = 0$	rs^5	rt^5	0	0	0x19
div rs, rt	hi, lo = rs / rt	$op^6 = 0$	rs^5	rt^5	0	0	0x1a
divu rs, rt	hi, lo = rs / rt	$op^6 = 0$	rs^5	rt^5	0	0	0x1b
mfhi rd	rd = hi	$op^6 = 0$	0	0	rd^5	0	0x10
mflo rd	rd = lo	$op^6 = 0$	0	0	rd^5	0	0x12
mthi rs	hi = rs	$op^6 = 0$	rs^5	0	0	0	0x11
mtlo rs	lo = rs	$op^6 = 0$	rs^5	0	0	0	0x13

- ❖ Signed arithmetic: **mult**, **div** (rs and rt are signed)
 - ✧ LO = 32-bit low-order and HI = 32-bit high-order of multiplication
 - ✧ LO = 32-bit quotient and HI = 32-bit remainder of division
- ❖ Unsigned arithmetic: **multu**, **divu** (rs and rt are unsigned)
- ❖ NO arithmetic exception can occur

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Alternative Architecture

I-Type Format

- ❖ Constants are used quite frequently in programs
 - ✧ The R-type shift instructions have a **5-bit shift amount constant**
 - ✧ What about other instructions that need a constant?

- ❖ I-Type: Instructions with Immediate Operands



- ❖ 16-bit immediate constant is stored inside the instruction
 - ✧ Rs is the source register number
 - ✧ Rt is now the **destination** register number (for R-type it was Rd)
- ❖ Examples of I-Type ALU Instructions:
 - ✧ Add immediate: `addi $s1, $s2, 5` # `$s1 = $s2 + 5`
 - ✧ OR immediate: `ori $s1, $s2, 5` # `$s1 = $s2 | 5`

I-Type ALU Instructions

Instruction	Meaning	I-Type Format				
addi \$s1, \$s2, 10	$\$s1 = \$s2 + 10$	op = 0x8	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
addiu \$s1, \$s2, 10	$\$s1 = \$s2 + 10$	op = 0x9	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
andi \$s1, \$s2, 10	$\$s1 = \$s2 \& 10$	op = 0xc	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
ori \$s1, \$s2, 10	$\$s1 = \$s2 10$	op = 0xd	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
xori \$s1, \$s2, 10	$\$s1 = \$s2 \wedge 10$	op = 0xe	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
lui \$s1, 10	$\$s1 = 10 \ll 16$	op = 0xf	0	rt = \$s1	imm ¹⁶ = 10	

❖ **addi**: overflow causes an **arithmetic exception**

✧ In case of overflow, result is not written to destination register

❖ **addiu**: same operation as **addi** but **overflow is ignored**

❖ Immediate constant for **addi** and **addiu** is **signed**

✧ No need for **subi** or **subiu** instructions

❖ Immediate constant for **andi**, **ori**, **xori** is **unsigned**

Examples: I-Type ALU Instructions

❖ **Examples:** assume **A, B, C** are allocated **\$s0, \$s1, \$s2**

A = B+5; translated as **addiu \$s0,\$s1,5**

C = B-1; translated as **addiu \$s2,\$s1,-1**



op=001001	rs=\$s1=10001	rt=\$s2=10010	imm = -1 = 1111111111111111
-----------	---------------	---------------	-----------------------------

A = B&0xf; translated as **andi \$s0,\$s1,0xf**

C = B|0xf; translated as **ori \$s2,\$s1,0xf**

C = 5; translated as **ori \$s2,\$zero,5**

A = B; translated as **ori \$s0,\$s1,0**

❖ No need for **subi**, because **addi** has **signed immediate**

❖ Register 0 (**\$zero**) has always the value 0

32-bit Constants

- ❖ I-Type instructions can have only 16-bit constants



- ❖ What if we want to load a 32-bit constant into a register?

- ❖ Can't have a 32-bit constant in I-Type instructions ☹

- ✧ We have already fixed the sizes of all instructions to 32 bits

- ❖ **Solution: use two instructions instead of one** 😊

- ✧ Suppose we want: **\$s1=0xAC5165D9** (32-bit constant)

- ✧ **lui: load upper immediate**

lui \$s1, 0xAC51

\$s1=\$17

**load upper
16 bits**

0xAC51

**clear lower
16 bits**

0x0000

ori \$s1, \$s1, 0x65D9

\$s1=\$17

0xAC51

0x65D9

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ **Jump and Branch Instructions**
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Alternative Architecture

J-Type Format



- ❖ J-type format is used for unconditional jump instruction:

`j    label    # jump to label`

`. . .`

`label:`

- ❖ 26-bit immediate value is stored in the instruction
 - ✧ Immediate constant specifies address of target instruction
- ❖ Program Counter (PC) is modified as follows:

✧ Next PC = least-significant
2 bits are 00

The diagram shows the calculation of the next PC. It consists of a green box labeled 'PC⁴', followed by an orange box labeled 'immediate²⁶', and finally a green box labeled '00'.

- ✧ Upper 4 most significant bits of PC are unchanged

Conditional Branch Instructions

❖ MIPS **compare and branch** instructions:

beq *Rs*,*Rt*,*label* branch to **label** if (*Rs* == *Rt*)

bne *Rs*,*Rt*,*label* branch to **label** if (*Rs* != *Rt*)

❖ MIPS **compare to zero & branch** instructions

Compare to zero is used frequently and implemented efficiently

bltz *Rs*,*label* branch to **label** if (*Rs* < 0)

bgtz *Rs*,*label* branch to **label** if (*Rs* > 0)

blez *Rs*,*label* branch to **label** if (*Rs* <= 0)

bgez *Rs*,*label* branch to **label** if (*Rs* >= 0)

❖ No need for **beqz** and **bnez** instructions. Why?

Set on Less Than Instructions

❖ MIPS also provides **set on less than** instructions

slt **rd,rs,rt** if ($rs < rt$) $rd = 1$ else $rd = 0$

sltu **rd,rs,rt** **unsigned <**

slti **rt,rs,im¹⁶** if ($rs < im^{16}$) $rt = 1$ else $rt = 0$

sltiu **rt,rs,im¹⁶** **unsigned <**

❖ **Signed / Unsigned Comparisons**

Can produce **different** results

Assume **$\$s0 = 1$** and **$\$s1 = -1 = 0xffffffff$**

slt **$\$t0, \$s0, \$s1$** results in **$\$t0 = 0$**

sltu **$\$t0, \$s0, \$s1$** results in **$\$t0 = 1$**

More on Branch Instructions

❖ MIPS hardware does NOT provide instructions for ...

blt, bltu	branch if less than	(signed/unsigned)
ble, bleu	branch if less or equal	(signed/unsigned)
bgt, bgtu	branch if greater than	(signed/unsigned)
bge, bgeu	branch if greater or equal	(signed/unsigned)

Can be achieved with a **sequence of 2 instructions**

❖ How to implement:

❖ Solution:



```
blt $s0,$s1,label  
slt $at,$s0,$s1  
bne $at,$zero,label
```

❖ How to implement:

❖ Solution:



```
ble $s2,$s3,label  
slt $at,$s3,$s2  
beq $at,$zero,label
```

Pseudo-Instructions

- ❖ Introduced by assembler as if they were real instructions
 - ✧ To facilitate assembly language programming

Pseudo-Instructions	Conversion to Real Instructions
<code>move \$s1, \$s2</code>	<code>addu \$s1, \$s2, \$zero</code>
<code>not \$s1, \$s2</code>	<code>nor \$s1, \$s2, \$s2</code>
<code>li \$s1, 0xabcd</code>	<code>ori \$s1, \$zero, 0xabcd</code>
<code>li \$s1, 0xabcd1234</code>	<code>lui \$s1, 0xabcd</code> <code>ori \$s1, \$s1, 0x1234</code>
<code>sgt \$s1, \$s2, \$s3</code>	<code>slt \$s1, \$s3, \$s2</code>
<code>blt \$s1, \$s2, label</code>	<code>slt \$at, \$s1, \$s2</code> <code>bne \$at, \$zero, label</code>

- ❖ Assembler reserves `$at = $1` for its own use
 - ✧ `$at` is called the **assembler temporary** register

Jump, Branch, and SLT Instructions

Instruction		Meaning	Format			
j	label	jump to label	$op^6 = 2$	imm^{26}		
beq	rs, rt, label	branch if (rs == rt)	$op^6 = 4$	rs^5	rt^5	imm^{16}
bne	rs, rt, label	branch if (rs != rt)	$op^6 = 5$	rs^5	rt^5	imm^{16}
blez	rs, label	branch if (rs ≤ 0)	$op^6 = 6$	rs^5	0	imm^{16}
bgtz	rs, label	branch if (rs > 0)	$op^6 = 7$	rs^5	0	imm^{16}
bltz	rs, label	branch if (rs < 0)	$op^6 = 1$	rs^5	0	imm^{16}
bgez	rs, label	branch if (rs ≥ 0)	$op^6 = 1$	rs^5	1	imm^{16}

Instruction		Meaning	Format					
slt	rd, rs, rt	$rd = (rs < rt ? 1 : 0)$	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
sltu	rd, rs, rt	$rd = (rs < rt ? 1 : 0)$	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2b
slti	rt, rs, imm^{16}	$rt = (rs < imm ? 1 : 0)$	0xa	rs^5	rt^5	imm^{16}		
sltiu	rt, rs, imm^{16}	$rt = (rs < imm ? 1 : 0)$	0xb	rs^5	rt^5	imm^{16}		

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Alternative Architecture

Translating an IF Statement

❖ Consider the following IF statement:

```
if (a == b) c = d + e; else c = d - e;
```

Assume that *a*, *b*, *c*, *d*, *e* are in *\$s0*, ..., *\$s4* respectively

❖ How to translate the above IF statement?

```
        bne    $s0, $s1, else
        addu   $s2, $s3, $s4
        j      exit
else:    subu   $s2, $s3, $s4
exit:    . . .
```

Compound Expression with AND

- ❖ Programming languages use **short-circuit evaluation**
- ❖ If first expression is **false**, second expression is **skipped**

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

One Possible Implementation ...

```
    bgtz    $s1, L1      # first expression
    j       next        # skip if false
L1: bltz    $s2, L2      # second expression
    j       next        # skip if false
L2: addiu   $s3,$s3,1    # both are true
next:
```

Better Implementation for AND

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

The following implementation uses less code

Reverse the relational operator

Allow the program to **fall through** to the second expression

Number of instructions is reduced from 5 to 3

```
# Better Implementation ...
    blez    $s1, next    # skip if false
    bgez    $s2, next    # skip if false
    addiu   $s3,$s3,1     # both are true
next:
```

Compound Expression with OR

- ❖ Short-circuit evaluation for logical OR
- ❖ If first expression is **true**, second expression is **skipped**

```
if (($s1 > $s2) || ($s2 > $s3)) {$s4 = 1;}
```

- ❖ Use **fall-through** to keep the code as short as possible

```
    bgt $s1, $s2, L1      # yes, execute if part
    ble $s2, $s3, next    # no: skip if part
L1:  li  $s4, 1           # set $s4 to 1
next:
```

- ❖ **bgt**, **ble**, and **li** are **pseudo-instructions**

✧ Translated by the assembler to real instructions

Your Turn . . .

- ❖ Translate the IF statement to assembly language
- ❖ \$s1 and \$s2 values are **unsigned**

```
if( $s1 <= $s2 ) {  
    $s3 = $s4  
}
```

```
bgtu $s1, $s2, next  
move $s3, $s4  
next:
```

- ❖ \$s3, \$s4, and \$s5 values are **signed**

```
if ( ($s3 <= $s4) &&  
    ($s4 > $s5) ) {  
    $s3 = $s4 + $s5  
}
```

```
bgt $s3, $s4, next  
ble $s4, $s5, next  
addu $s3, $s4, $s5  
next:
```

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Alternative Architecture

Load and Store Instructions

- ❖ Instructions that transfer data between memory & registers
- ❖ Programs include variables such as arrays and objects
- ❖ Such variables are stored in memory

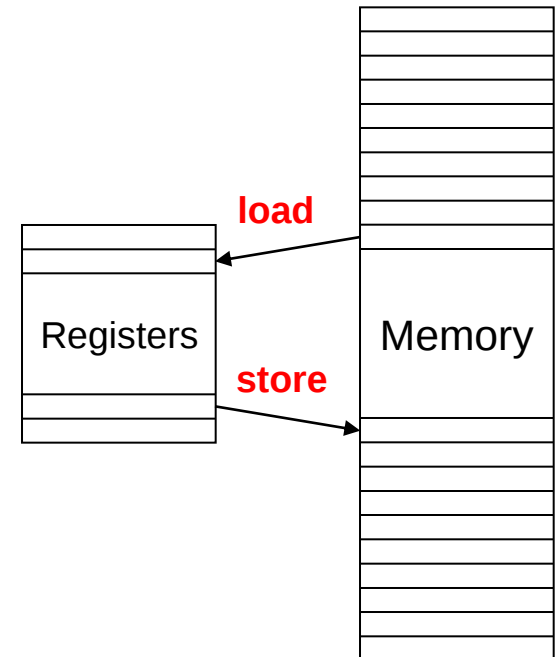
- ❖ **Load** Instruction:

- ✧ Transfers data from memory to a register

- ❖ **Store** Instruction:

- ✧ Transfers data from a register to memory

- ❖ **Memory address** must be specified by load and store



Load and Store Word

❖ Load Word Instruction (Word = 4 bytes in MIPS)

lw Rt, imm¹⁶(Rs) # Rt = MEMORY[Rs+imm¹⁶]

❖ Store Word Instruction

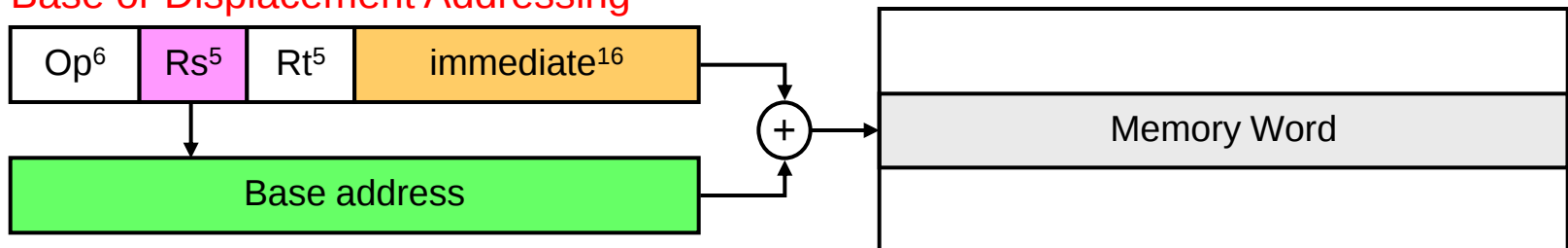
sw Rt, imm¹⁶(Rs) # MEMORY[Rs+imm¹⁶] = Rt

❖ **Base or Displacement addressing** is used

✧ Memory Address = Rs (**base**) + Immediate¹⁶ (**displacement**)

✧ Immediate¹⁶ is **sign-extended** to have a signed displacement

Base or Displacement Addressing



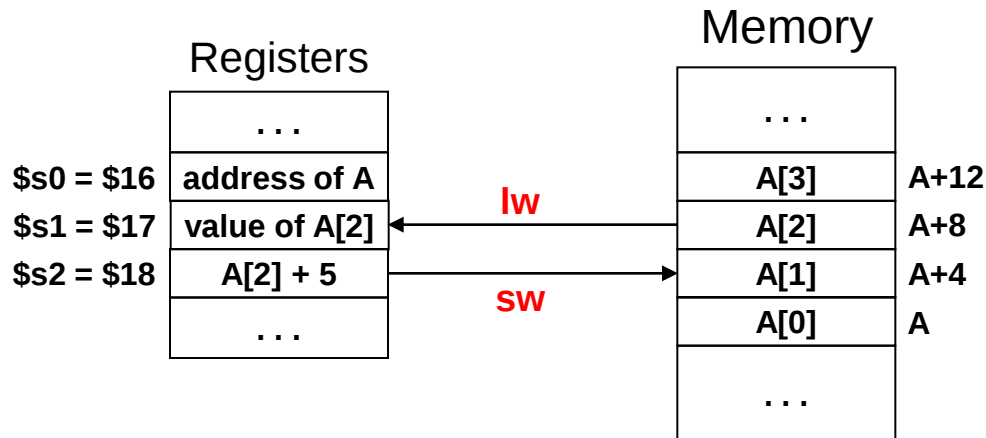
Example on Load & Store

❖ Translate $A[1] = A[2] + 5$ (A is an array of words)

✧ Assume that address of array A is stored in register \$s0

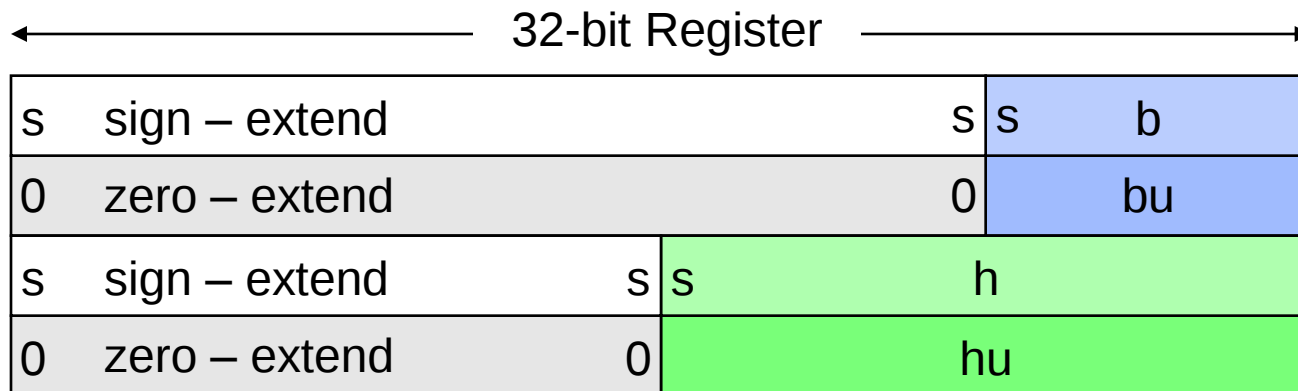
```
lw      $s1, 8($s0)      # $s1 = A[2]
addiu   $s2, $s1, 5      # $s2 = A[2] + 5
sw      $s2, 4($s0)      # A[1] = $s2
```

❖ Index of $a[2]$ and $a[1]$ should be multiplied by 4. Why?



Load and Store Byte and Halfword

- ❖ The MIPS processor supports the following data formats:
 - ✧ Byte = 8 bits, Halfword = 16 bits, Word = 32 bits
- ❖ Load & store instructions for bytes and halfwords
 - ✧ lb = load byte, lbu = load byte unsigned, sb = store byte
 - ✧ lh = load half, lhu = load half unsigned, sh = store halfword
- ❖ Load **expands** a memory data to fit into a 32-bit register
- ❖ Store **reduces** a 32-bit register to fit in memory



Load and Store Instructions

Instruction		Meaning	I-Type Format			
lb	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x20	rs ⁵	rt ⁵	imm ¹⁶
lh	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x21	rs ⁵	rt ⁵	imm ¹⁶
lw	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x23	rs ⁵	rt ⁵	imm ¹⁶
lbu	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x24	rs ⁵	rt ⁵	imm ¹⁶
lhu	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x25	rs ⁵	rt ⁵	imm ¹⁶
sb	rt, imm ¹⁶ (rs)	MEM[rs+imm ¹⁶] = rt	0x28	rs ⁵	rt ⁵	imm ¹⁶
sh	rt, imm ¹⁶ (rs)	MEM[rs+imm ¹⁶] = rt	0x29	rs ⁵	rt ⁵	imm ¹⁶
sw	rt, imm ¹⁶ (rs)	MEM[rs+imm ¹⁶] = rt	0x2b	rs ⁵	rt ⁵	imm ¹⁶

❖ **Base or Displacement Addressing** is used

✧ Memory Address = Rs (**base**) + Immediate¹⁶ (**displacement**)

❖ Two variations on base addressing

✧ If Rs = \$zero = 0 then Address = Immediate¹⁶ (**absolute**)

✧ If Immediate¹⁶ = 0 then Address = Rs (**register indirect**)

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ Alternative Architecture

Translating a WHILE Loop

❖ Consider the following WHILE statement:

```
i = 0; while (A[i] != k) i = i+1;
```

Where *A* is an array of integers (4 bytes per element)

Assume address *A*, *i*, *k* in *\$s0*, *\$s1*, *\$s2*, respectively

Memory

...	
A[i]	A+4×i
...	
A[2]	A+8
A[1]	A+4
A[0]	A
...	

❖ How to translate above WHILE statement?

```
        xor    $s1, $s1, $s1    # i = 0
        move   $t0, $s0         # $t0 = address A
loop:    lw     $t1, 0($t0)      # $t1 = A[i]
        beq    $t1, $s2, exit    # exit if (A[i]== k)
        addiu  $s1, $s1, 1      # i = i+1
        sll    $t0, $s1, 2      # $t0 = 4*i
        addu   $t0, $s0, $t0    # $t0 = address A[i]
        j      loop
exit:    . . .
```

Using Pointers to Traverse Arrays

- ❖ Consider the same WHILE loop:

```
i = 0; while (A[i] != k) i = i+1;
```

Where address of A, i, k are in \$s0, \$s1, \$s2, respectively

- ❖ We can use a **pointer** to traverse array A

Pointer is incremented by 4 (faster than indexing)

```
        move    $t0, $s0           # $t0 = $s0 = addr A
        j       cond              # test condition
loop:    addiu   $s1, $s1, 1        # i = i+1
        addiu   $t0, $t0, 4        # point to next
cond:    lw      $t1, 0($t0)        # $t1 = A[i]
        bne     $t1, $s2, loop     # loop if A[i] != k
```

- ❖ Only 4 instructions (rather than 6) in loop body

Copying a String

The following code copies source string to target string

Address of source in \$s0 and address of target in \$s1

Strings are terminated with a null character (C strings)

```
i = 0;  
do {target[i]=source[i]; i++;} while (source[i]!=0);
```

```
        move    $t0, $s0           # $t0 = pointer to source  
        move    $t1, $s1           # $t1 = pointer to target  
L1:  lb      $t2, 0($t0)           # load byte into $t2  
        sb      $t2, 0($t1)         # store byte into target  
        addiu   $t0, $t0, 1         # increment source pointer  
        addiu   $t1, $t1, 1         # increment target pointer  
        bne     $t2, $zero, L1      # loop until NULL char
```

Summing an Integer Array

```
sum = 0;  
for (i=0; i<n; i++) sum = sum + A[i];
```

Assume \$s0 = array address, \$s1 = array length = n

```
move    $t0, $s0           # $t0 = address A[i]  
xor     $t1, $t1, $t1      # $t1 = i = 0  
xor     $s2, $s2, $s2      # $s2 = sum = 0  
L1: lw   $t2, 0($t0)       # $t2 = A[i]  
addu    $s2, $s2, $t2      # sum = sum + A[i]  
addiu   $t0, $t0, 4        # point to next A[i]  
addiu   $t1, $t1, 1        # i++  
bne     $t1, $s1, L1       # loop if (i != n)
```

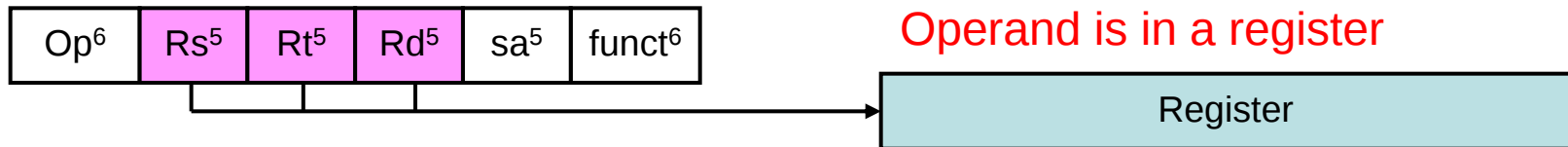
Addressing Modes

- ❖ Where are the operands?
- ❖ How memory addresses are computed?

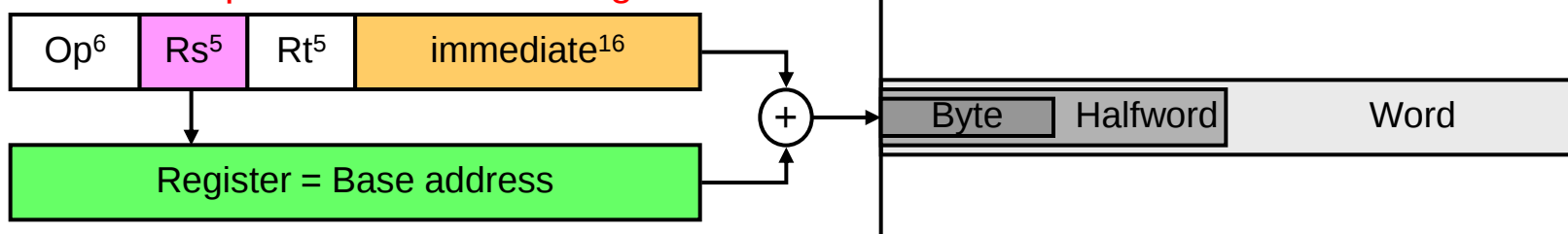
Immediate Addressing



Register Addressing

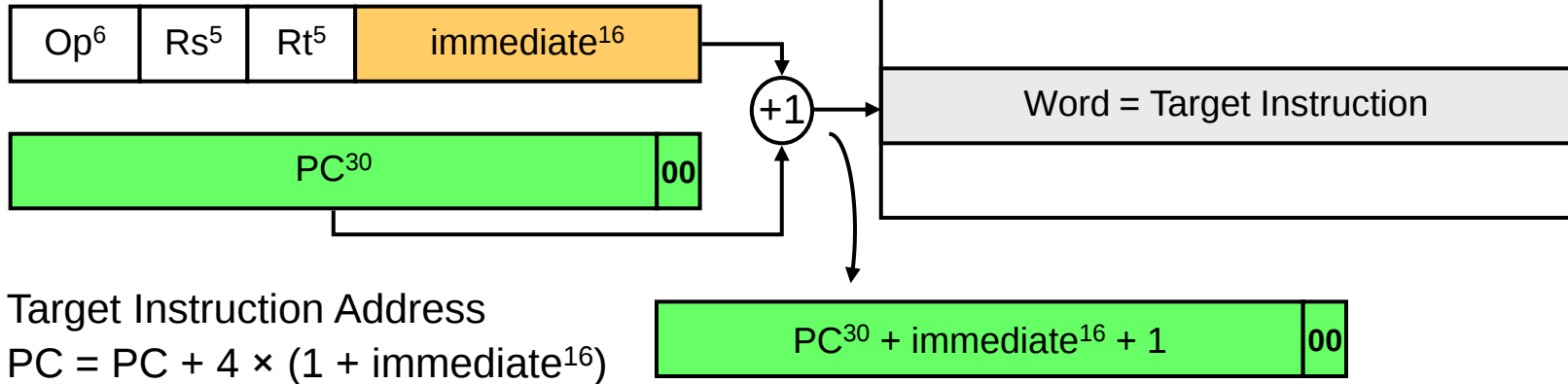


Base or Displacement Addressing

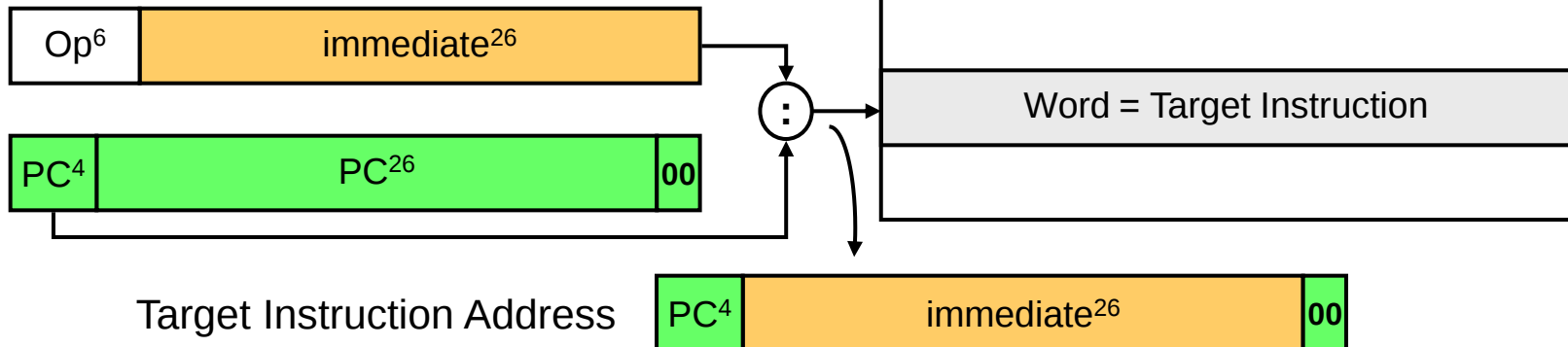


Branch / Jump Addressing Modes

PC-Relative Addressing



Pseudo-direct Addressing



Jump and Branch Limits

❖ Jump Address Boundary = 2^{26} instructions = 256 MB

✧ Text segment cannot exceed 2^{26} instructions or 256 MB

✧ Upper 4 bits of PC are unchanged

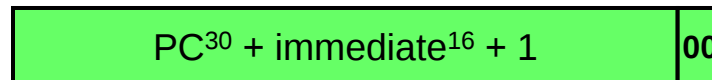
Target Instruction Address



❖ Branch Address Boundary

✧ Branch instructions use I-Type format (16-bit immediate constant)

✧ PC-relative addressing:



- Target instruction address = $PC + 4 \times (1 + \text{immediate}^{16})$
- Count number of instructions to branch from next instruction
- **Positive constant** => **Forward** Branch, **Negative** => **Backward** branch
- At most $\pm 2^{15}$ instructions to branch (most branches are near)

Next . . .

- ❖ Instruction Set Architecture
- ❖ Overview of the MIPS Processor
- ❖ R-Type Arithmetic, Logical, and Shift Instructions
- ❖ I-Type Format and Immediate Constants
- ❖ Jump and Branch Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Load and Store Instructions
- ❖ Translating Loops and Traversing Arrays
- ❖ **Alternative Architecture**

MIPS Assembly Language Programming

Assembly Language Statements

❖ Three types of statements in assembly language

- ✧ Typically, one statement should appear on a line

1. Executable Instructions

- ✧ Generate machine code for the processor to execute at runtime
- ✧ Instructions tell the processor what to do

2. Pseudo-Instructions and Macros

- ✧ Translated by the assembler into real instructions
- ✧ Simplify the programmer task

3. Assembler Directives

- ✧ Provide information to the assembler while translating a program
- ✧ Used to define segments, allocate memory variables, etc.
- ✧ Non-executable: directives are not part of the instruction set

Instructions

❖ Assembly language instructions have the format:

`[label:]      mnemonic      [operands]      [#comment]`

❖ Label: (optional)

- ✧ Marks the address of a memory location, must have a colon
- ✧ Typically appear in data and text segments

❖ Mnemonic

- ✧ Identifies the operation (e.g. **add**, **sub**, etc.)

❖ Operands

- ✧ Specify the data required by the operation
- ✧ Operands can be registers, memory variables, or constants
- ✧ Most instructions have three operands

`L1:      addiu $t0, $t0, 1                      #increment $t0`

Comments

❖ Comments are very important!

- ✧ Explain the program's purpose
- ✧ When it was written, revised, and by whom
- ✧ Explain data used in the program, input, and output
- ✧ Explain instruction sequences and algorithms used
- ✧ Comments are also required at the beginning of every procedure
 - Indicate input parameters and results of a procedure
 - Describe what the procedure does

❖ Single-line comment

- ✧ Begins with a hash symbol **#** and terminates at end of line

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Program Template

```
# Title:                               Filename:
# Author:                             Date:
# Description:
# Input:
# Output:
##### Data segment #####
.data
    . . .
##### Code segment #####
.text
.globl main
main:                                # main program entry
    . . .
li $v0, 10                          # Exit program
syscall
```

.DATA, .TEXT, & .GLOBL Directives

❖ .DATA directive

- ✧ Defines the **data segment** of a program containing data
- ✧ The program's variables should be defined under this directive
- ✧ Assembler will allocate and initialize the storage of variables

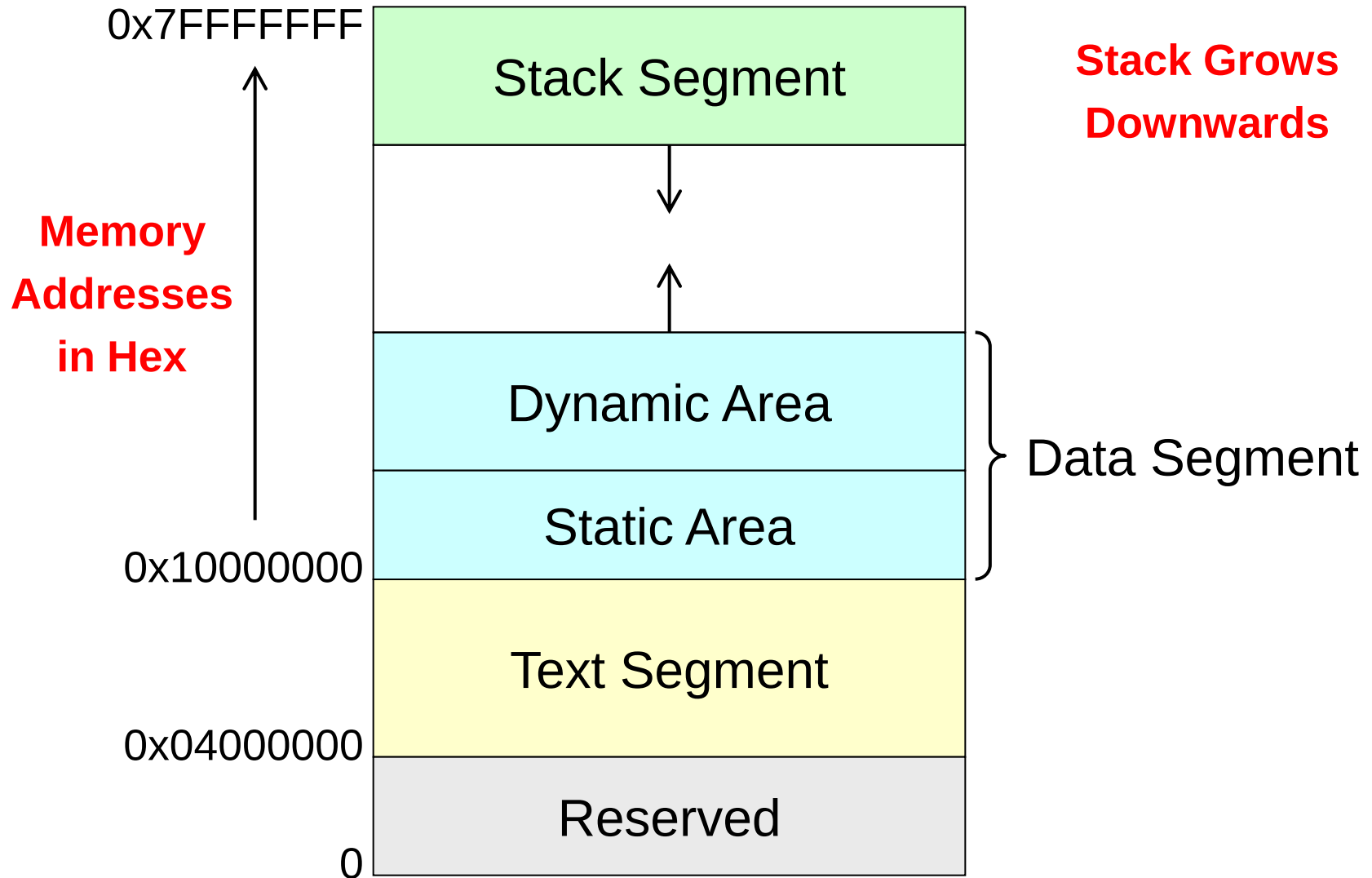
❖ .TEXT directive

- ✧ Defines the **code segment** of a program containing instructions

❖ .GLOBL directive

- ✧ Declares a symbol as **global**
- ✧ Global symbols can be referenced from other files
- ✧ We use this directive to declare *main* procedure of a program

Layout of a Program in Memory



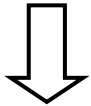
Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ **Defining Data**
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Data Definition Statement

- ❖ Sets aside storage in memory for a variable
- ❖ May optionally assign a name (label) to the data
- ❖ Syntax:

[name:] directive initializer [, initializer] . . .



var1: **.WORD** **10**

- ❖ All initializers become binary data in memory

Data Directives

❖ **.BYTE** Directive

- ✧ Stores the list of values as 8-bit bytes

❖ **.HALF** Directive

- ✧ Stores the list as 16-bit values aligned on half-word boundary

❖ **.WORD** Directive

- ✧ Stores the list as 32-bit values aligned on a word boundary

❖ **.FLOAT** Directive

- ✧ Stores the listed values as single-precision floating point

❖ **.DOUBLE** Directive

- ✧ Stores the listed values as double-precision floating point

String Directives

❖ **.ASCII** Directive

- ✧ Allocates a sequence of bytes for an ASCII string

❖ **.ASCIIZ** Directive

- ✧ Same as **.ASCII** directive, but adds a NULL char at end of string
- ✧ Strings are null-terminated, as in the C programming language

❖ **.SPACE** Directive

- ✧ Allocates space of n uninitialized bytes in the data segment

Examples of Data Definitions

.DATA

var1: .BYTE 'A', 'E', 127, -1, '\n'

var2: .HALF -10, 0xffff

var3: .WORD 0x12345678:100 ← **Array of 100 words**

var4: .FLOAT 12.3, -0.1

var5: .DOUBLE 1.5e-10

str1: .ASCII "A String\n"

str2: .ASCIIZ "NULL Terminated String"

array: .SPACE 100 ← **100 bytes (not initialized)**

MARS Assembler and Simulator Tool

C:\Documents and Settings\Muhamed Mudawar\My Documents\COE 308\Tools\MARS\Data.asm - MARS 3.4.1

File Edit Run Settings Tools Help

0101

Edit Execute

```
1 .data
2  var1:  .byte   3, -2, 'A'
3  var2:  .half   1, 256, 0xffff
4  var3:  .word   0x3delc74, 0xff
5  .align 3
6  str1:   .asciiz "COE 308"
7  str2:   .asciiz "Section 1"
8
9  .align 2
10 fibs:  .space 400  #space for 100 integers
11
12 .text
13 .globl main
14 main:
15  li $a0, 50
16  la $t0, fibs
17  li $t1, 1
18  li $t2, 1
```

Line: 6 Column: 24 ☒ Show Line Numbers

Mars Messages Run I/O

Clear

Registers		Coproc 1	Coproc 0
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x00000000	
\$v0	2	0x00000000	
\$v1	3	0x00000000	
\$a0	4	0x00000000	
\$a1	5	0x00000000	
\$a2	6	0x00000000	
\$a3	7	0x00000000	
\$t0	8	0x00000000	
\$t1	9	0x00000000	
\$t2	10	0x00000000	
\$t3	11	0x00000000	
\$t4	12	0x00000000	
\$t5	13	0x00000000	
\$t6	14	0x00000000	
\$t7	15	0x00000000	
\$s0	16	0x00000000	
\$s1	17	0x00000000	
\$s2	18	0x00000000	
\$s3	19	0x00000000	
\$s4	20	0x00000000	
\$s5	21	0x00000000	

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Memory Alignment

❖ Memory is viewed as an **array of bytes** with addresses

✧ **Byte Addressing**: address points to a byte in memory

❖ Words occupy 4 consecutive bytes in memory

✧ MIPS instructions and integers occupy 4 bytes

❖ **Alignment: address is a multiple of size**

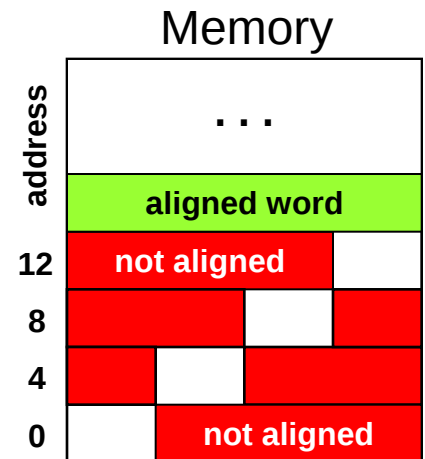
✧ Word address should be a multiple of **4**

▪ Least significant 2 bits of address should be **00**

✧ Halfword address should be a multiple of **2**

❖ **.ALIGN n** directive

✧ Aligns the next data definition on a 2^n byte boundary



Symbol Table

- ❖ Assembler builds a **symbol table** for labels (variables)
 - ✧ Assembler computes the address of each label in data segment

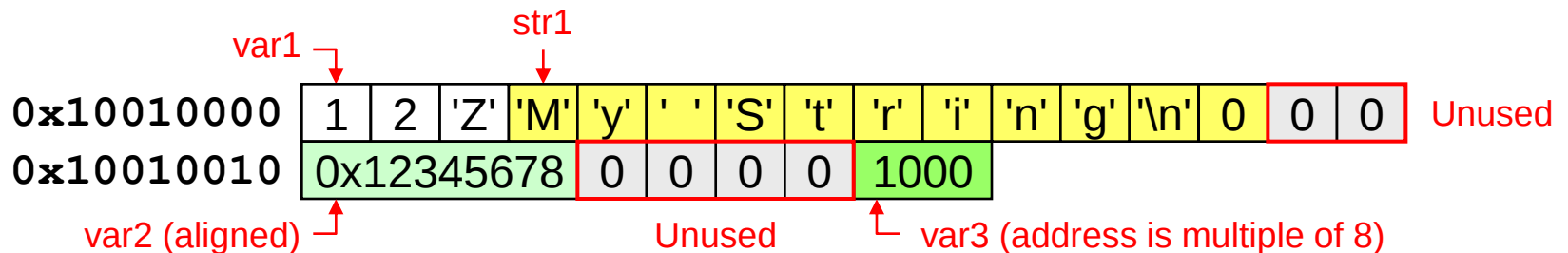
❖ Example

.DATA

```
var1:  .BYTE    1, 2, 'Z'
str1:  .ASCIIZ  "My String\n"
var2:  .WORD    0x12345678
.ALIGN  3
var3:  .HALF    1000
```

Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018



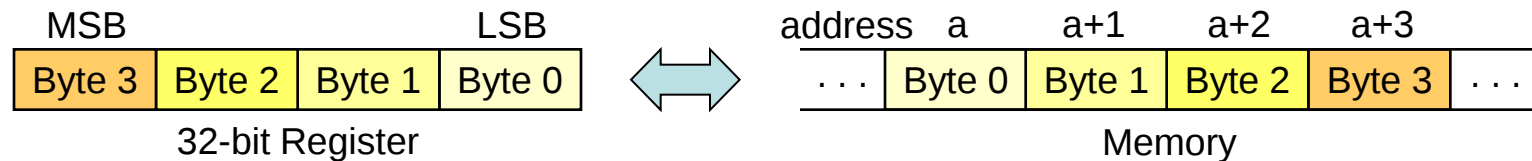
Byte Ordering and Endianness

❖ Processors can order bytes within a word in two ways

❖ Little Endian Byte Ordering

✧ Memory address = Address of **least significant byte**

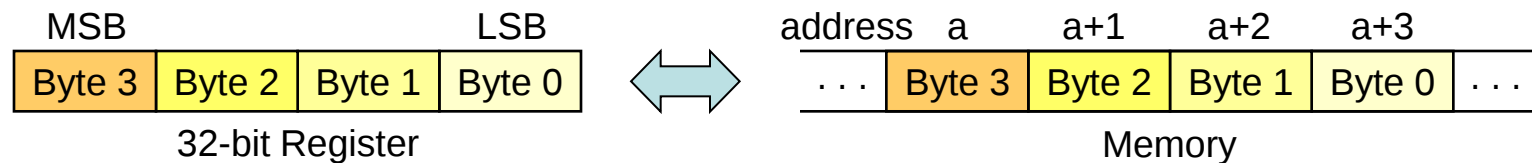
✧ Example: Intel IA-32, Alpha



❖ Big Endian Byte Ordering

✧ Memory address = Address of **most significant byte**

✧ Example: SPARC, PA-RISC



❖ MIPS can operate with both byte orderings

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

System Calls

- ❖ Programs do input/output through system calls
- ❖ MIPS provides a special **syscall** instruction
 - ✧ To obtain services from the operating system
 - ✧ Many services are provided in the SPIM and MARS simulators
- ❖ Using the **syscall** system services
 - ✧ Load the service number in register **\$v0**
 - ✧ Load argument values, if any, in registers **\$a0**, **\$a1**, etc.
 - ✧ Issue the **syscall** instruction
 - ✧ Retrieve return values, if any, from result registers

Syscall Services

Service	\$v0	Arguments / Result
Print Integer	1	\$a0 = integer value to print
Print Float	2	\$f12 = float value to print
Print Double	3	\$f12 = double value to print
Print String	4	\$a0 = address of null-terminated string
Read Integer	5	\$v0 = integer read
Read Float	6	\$f0 = float read
Read Double	7	\$f0 = double read
Read String	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	\$a0 = character to print
Read Char	12	\$a0 = character read

Supported by MARS

Reading and Printing an Integer

```
##### Code segment #####  
.text  
.globl main  
main:                                # main program entry  
    li    $v0, 5                     # Read integer  
    syscall                          # $v0 = value read  
  
    move  $a0, $v0                   # $a0 = value to print  
    li    $v0, 1                     # Print integer  
    syscall  
  
    li    $v0, 10                    # Exit program  
    syscall
```

Reading and Printing a String

```
##### Data segment #####  
.data  
    str: .space 10          # array of 10 bytes  
##### Code segment #####  
.text  
.globl main  
main:                        # main program entry  
    la    $a0, str          # $a0 = address of str  
    li    $a1, 10           # $a1 = max string length  
    li    $v0, 8            # read string  
    syscall  
    li    $v0, 4            # Print string str  
    syscall  
    li    $v0, 10           # Exit program  
    syscall
```

Program 1: Sum of Three Integers

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#   Input: Requests three numbers.
#   Output: Outputs the sum.
##### Data segment #####
.data
prompt:  .asciiz      "Please enter three numbers: \n"
sum_msg: .asciiz      "The sum is: "
##### Code segment #####
.text
.globl main
main:
    la    $a0,prompt      # display prompt string
    li    $v0,4
    syscall
    li    $v0,5            # read 1st integer into $t0
    syscall
    move  $t0,$v0
```

Sum of Three Integers - Slide 2 of 2

```
li    $v0,5                # read 2nd integer into $t1
syscall
move  $t1,$v0

li    $v0,5                # read 3rd integer into $t2
syscall
move  $t2,$v0

addu  $t0,$t0,$t1          # accumulate the sum
addu  $t0,$t0,$t2

la    $a0,sum_msg          # write sum message
li    $v0,4
syscall

move  $a0,$t0              # output sum
li    $v0,1
syscall

li    $v0,10               # exit
syscall
```

Program 2: Case Conversion

```
# Objective: Convert lowercase letters to uppercase
# Input: Requests a character string from the user.
# Output: Prints the input string in uppercase.
##### Data segment #####
.data
name_prompt: .asciiz      "Please type your name: "
out_msg:      .asciiz      "Your name in capitals is: "
in_name:      .space 31    # space for input string
##### Code segment #####
.text
.globl main
main:
    la    $a0,name_prompt  # print prompt string
    li    $v0,4
    syscall
    la    $a0,in_name      # read the input string
    li    $a1,31            # at most 30 chars + 1 null char
    li    $v0,8
    syscall
```

Case Conversion - Slide 2 of 2

```
    la    $a0,out_msg      # write output message
    li    $v0,4
    syscall
    la    $t0,in_name
loop:
    lb     $t1,($t0)
    beqz   $t1,exit_loop    # if NULL, we are done
    blt    $t1,'a',no_change
    bgt    $t1,'z',no_change
    addiu  $t1,$t1,-32       # convert to uppercase: 'A'-'a'=-32
    sb     $t1,($t0)
no_change:
    addiu  $t0,$t0,1        # increment pointer
    j      loop
exit_loop:
    la     $a0,in_name      # output converted string
    li     $v0,4
    syscall
    li     $v0,10           # exit
    syscall
```

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Procedures

- ❖ Consider the following **swap** procedure (written in C)
- ❖ Translate this procedure to MIPS assembly language

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

Parameters:

\$a0 = Address of **v[]**

\$a1 = **k**, and

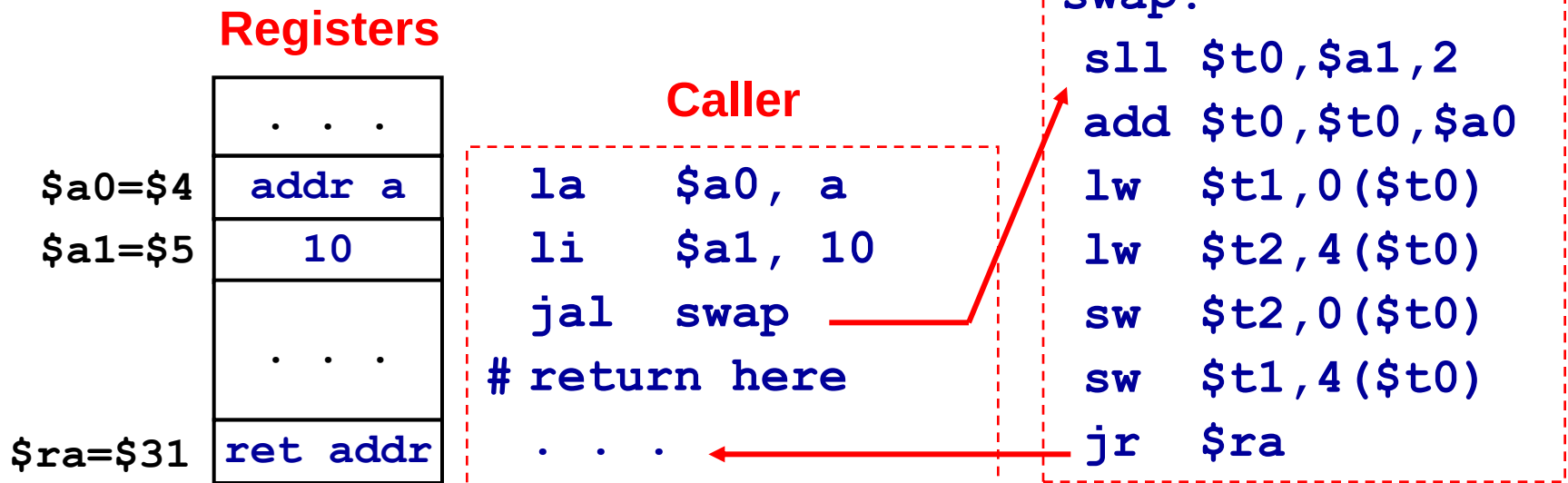
Return address is in **\$ra**

swap:

```
sll $t0,$a1,2      # $t0=k*4
add $t0,$t0,$a0     # $t0=v+k*4
lw  $t1,0($t0)      # $t1=v[k]
lw  $t2,4($t0)      # $t2=v[k+1]
sw  $t2,0($t0)      # v[k]=$t2
sw  $t1,4($t0)      # v[k+1]=$t1
jr  $ra             # return
```

Call / Return Sequence

- ❖ Suppose we call procedure swap as: **swap (a, 10)**
 - ✧ Pass **address** of array **a** and **10** as arguments
 - ✧ Call the procedure swap saving **return address** in **\$31 = \$ra**
 - ✧ Execute procedure swap
 - ✧ Return control to the point of origin (return address)



Details of JAL and JR

Address	Instructions	Assembly Language
---------	--------------	-------------------

00400020	lui \$1, 0x1001	la \$a0, a
00400024	ori \$4, \$1, 0	
00400028	ori \$5, \$0, 10	ori \$a1, \$0, 10
0040002C	jal 0x10000f	jal swap
00400030	...	# return here

Pseudo-Direct Addressing

PC = imm26 << 2
 0x10000f << 2
 = 0x0040003C

0040003C	sll \$8, \$5, 2	swap:
00400040	add \$8, \$8, \$4	sll \$t0, \$a1, 2
00400044	lw \$9, 0(\$8)	add \$t0, \$t0, \$a0
00400048	lw \$10, 4(\$8)	lw \$t1, 0(\$t0)
0040004C	sw \$10, 0(\$8)	lw \$t2, 4(\$t0)
00400050	sw \$9, 4(\$8)	sw \$t2, 0(\$t0)
00400054	jr \$31	sw \$t1, 4(\$t0)
		jr \$ra

\$31

0x00400030

Register \$31
 is the return
 address register

Instructions for Procedures

- ❖ JAL (**Jump-and-Link**) used as the call instruction
 - ✧ Save return address in **\$ra = PC+4** and jump to procedure
 - ✧ Register **\$ra = \$31** is used by **JAL** as the **return address**
- ❖ JR (**Jump Register**) used to return from a procedure
 - ✧ Jump to instruction whose address is in register Rs (PC = Rs)
- ❖ JALR (**Jump-and-Link Register**)
 - ✧ Save return address in Rd = PC+4, and
 - ✧ Jump to procedure whose address is in register Rs (PC = Rs)
 - ✧ Can be used to call methods (addresses known only at runtime)

Instruction		Meaning	Format					
jal	label	$\$31 = PC + 4$, jump	$op^6 = 3$	imm^{26}				
jr	Rs	$PC = Rs$	$op^6 = 0$	rs^5	0	0	0	8
jalr	Rd, Rs	$Rd = PC + 4$, $PC = Rs$	$op^6 = 0$	rs^5	0	rd^5	0	9

Next . . .

- ❖ Assembly Language Statements
- ❖ Assembly Language Program Template
- ❖ Defining Data
- ❖ Memory Alignment and Byte Ordering
- ❖ System Calls
- ❖ Procedures
- ❖ Parameter Passing and the Runtime Stack

Parameter Passing

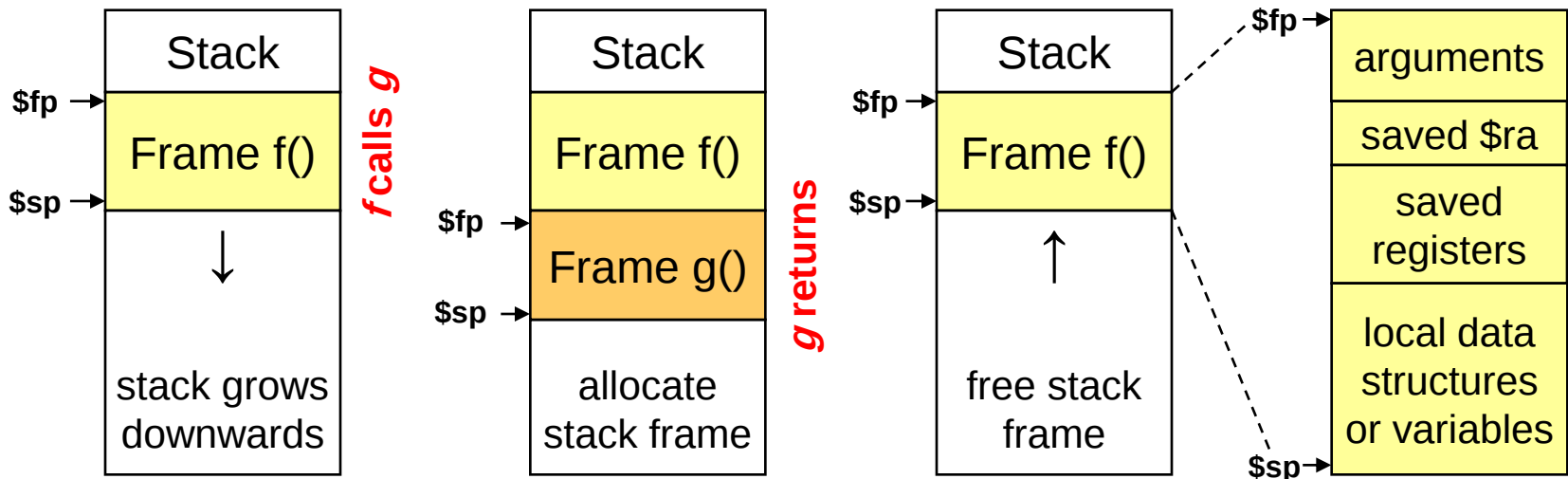
- ❖ Parameter passing in assembly language is different
 - ✧ More complicated than that used in a high-level language
- ❖ In assembly language
 - ✧ Place all required parameters in an accessible storage area
 - ✧ Then call the procedure
- ❖ Two types of storage areas used
 - ✧ Registers: general-purpose registers are used (**register method**)
 - ✧ Memory: stack is used (**stack method**)
- ❖ Two common mechanisms of parameter passing
 - ✧ Pass-by-value: parameter **value** is passed
 - ✧ Pass-by-reference: **address** of parameter is passed

Parameter Passing - cont'd

- ❖ By convention, registers are used for parameter passing
 - ✧ $\$a0 = \$4 \dots \$a3 = \7 are used for **passing arguments**
 - ✧ $\$v0 = \$2 \dots \$v1 = \3 are used for **result values**
- ❖ Additional arguments/results can be placed on the stack
- ❖ Runtime stack is also needed to ...
 - ✧ Store variables / data structures when they cannot fit in registers
 - ✧ Save and restore registers across procedure calls
 - ✧ Implement recursion
- ❖ Runtime stack is implemented via software convention
 - ✧ The **stack pointer** $\$sp = \29 (points to top of stack)
 - ✧ The **frame pointer** $\$fp = \30 (points to a procedure frame)

Stack Frame

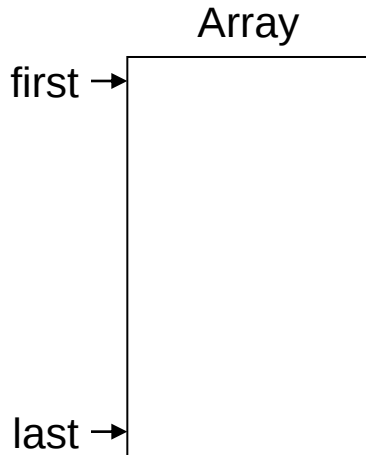
- ❖ **Stack frame** is the segment of the stack containing ...
 - ✧ Saved arguments, registers, and local data structures (if any)
- ❖ Called also the **activation frame** or **activation record**
- ❖ Frames are pushed and popped by adjusting ...
 - ✧ Stack pointer **\$sp** = **\$29** and Frame pointer **\$fp** = **R30**
 - ✧ Decrement **\$sp** to allocate stack frame, and increment to free



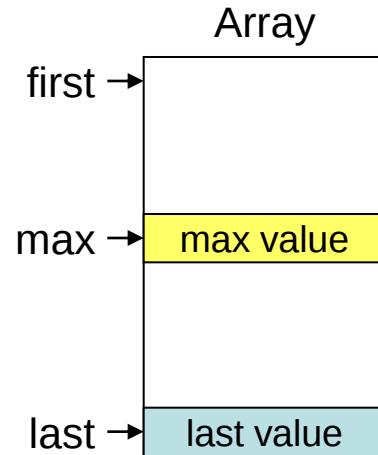
Preserving Registers

- ❖ Need to preserve registers across a procedure call
 - ✧ Stack can be used to preserve register values
- ❖ Which registers should be saved?
 - ✧ Registers modified by the called procedure, and
 - ✧ Still used by the calling procedure
- ❖ Who should preserve the registers?
 - ✧ **Called Procedure**: preferred method for modular code
 - Register preservation is done inside the called procedure
 - ✧ By convention, registers **\$s0**, **\$s1**, ..., **\$s7** should be preserved
 - ✧ Also, registers **\$sp**, **\$fp**, and **\$ra** should also be preserved

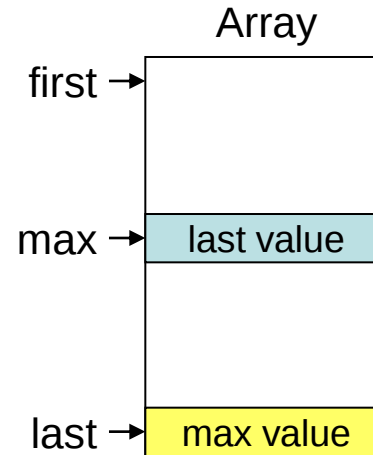
Selection Sort



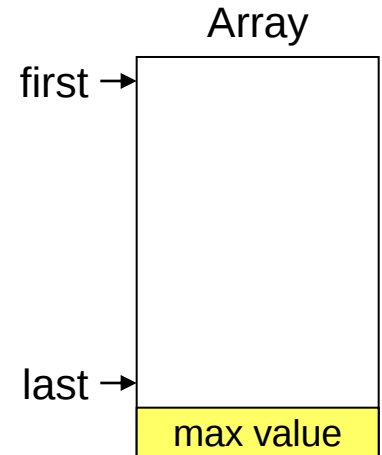
Unsorted



Locate
Max

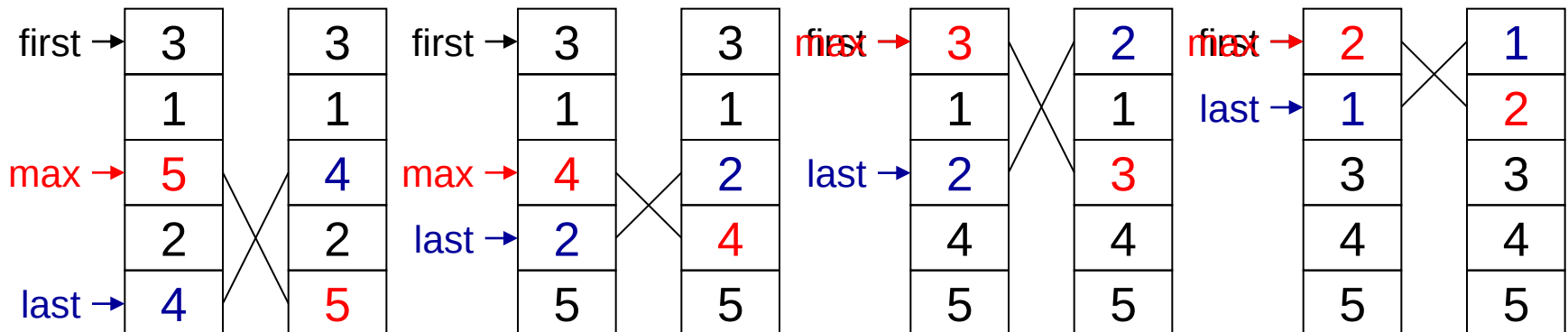


Swap Max
with Last



Decrement
Last

❖ Example



Selection Sort Procedure

```
# Objective: Sort array using selection sort algorithm
#   Input: $a0 = pointer to first, $a1 = pointer to last
#   Output: array is sorted in place
#####
sort: addiu $sp, $sp, -4      # allocate one word on stack
      sw     $ra, 0($sp)     # save return address on stack
top:  jal    max             # call max procedure
      lw     $t0, 0($a1)     # $t0 = last value
      sw     $t0, 0($v0)     # swap last and max values
      sw     $v1, 0($a1)
      addiu  $a1, $a1, -4    # decrement pointer to last
      bne    $a0, $a1, top   # more elements to sort
      lw     $ra, 0($sp)     # pop return address
      addiu  $sp, $sp, 4
      jr     $ra            # return to caller
```

Max Procedure

```
# Objective: Find the address and value of maximum element
#   Input: $a0 = pointer to first, $a1 = pointer to last
#   Output: $v0 = pointer to max, $v1 = value of max
#####
max:  move    $v0, $a0          # max pointer = first pointer
      lw      $v1, 0($v0)      # $v1 = first value
      beq     $a0, $a1, ret     # if (first == last) return
      move    $t0, $a0         # $t0 = array pointer
loop: addi    $t0, $t0, 4       # point to next array element
      lw      $t1, 0($t0)      # $t1 = value of A[i]
      ble     $t1, $v1, skip    # if (A[i] <= max) then skip
      move    $v0, $t0         # found new maximum
      move    $v1, $t1
skip: bne     $t0, $a1, loop    # loop back if more elements
ret:  jr      $ra
```

Example of a Recursive Procedure

```
int fact(int n) { if (n<2) return 1; else return (n*fact(n-1)); }
```

```
fact:  slti      $t0,$a0,2      # (n<2)?
        beq      $t0,$0,else    # if false branch to else
        li       $v0,1         # $v0 = 1
        jr       $ra           # return to caller

else:  addiu     $sp,$sp,-8     # allocate 2 words on stack
        sw       $a0,4($sp)    # save argument n
        sw       $ra,0($sp)    # save return address
        addiu    $a0,$a0,-1    # argument = n-1
        jal      fact         # call fact(n-1)
        lw       $a0,4($sp)    # restore argument
        lw       $ra,0($sp)    # restore return address
        mul      $v0,$a0,$v0   # $v0 = n*fact(n-1)
        addi     $sp,$sp,8     # free stack frame
        jr       $ra          # return to caller
```

You are going to enhance a machine and there are two types of possible improvements: either (i) make multiply instructions run 4 times faster, or (ii) make memory access instructions run two times faster than before. You repeatedly run a program that takes 100 seconds to execute (on the original machine) and find that of this time 25% is used for multiplication, 50% for memory access instructions, and 25% for other tasks.

1. What will the speedup be if you improve both multiplication and memory access?
2. Assume the program you run has 10 billions instructions and runs on the machine that has a clock rate of 1GHz. Calculate the CPI for this machine. Assume further that the CPI for multiplication instructions is 20 cycles and the CPI for memory access instructions is 6 cycles. Compute the CPI for all other instructions.
3. What is the CPI for the improved machine when improvements on both multiplication and memory access instructions are made?