

° Bölüm 3

° MIPS ARCHITECTURE

° MIPS INSTRUCTION SET

Instruction Set Architecture (ISA)

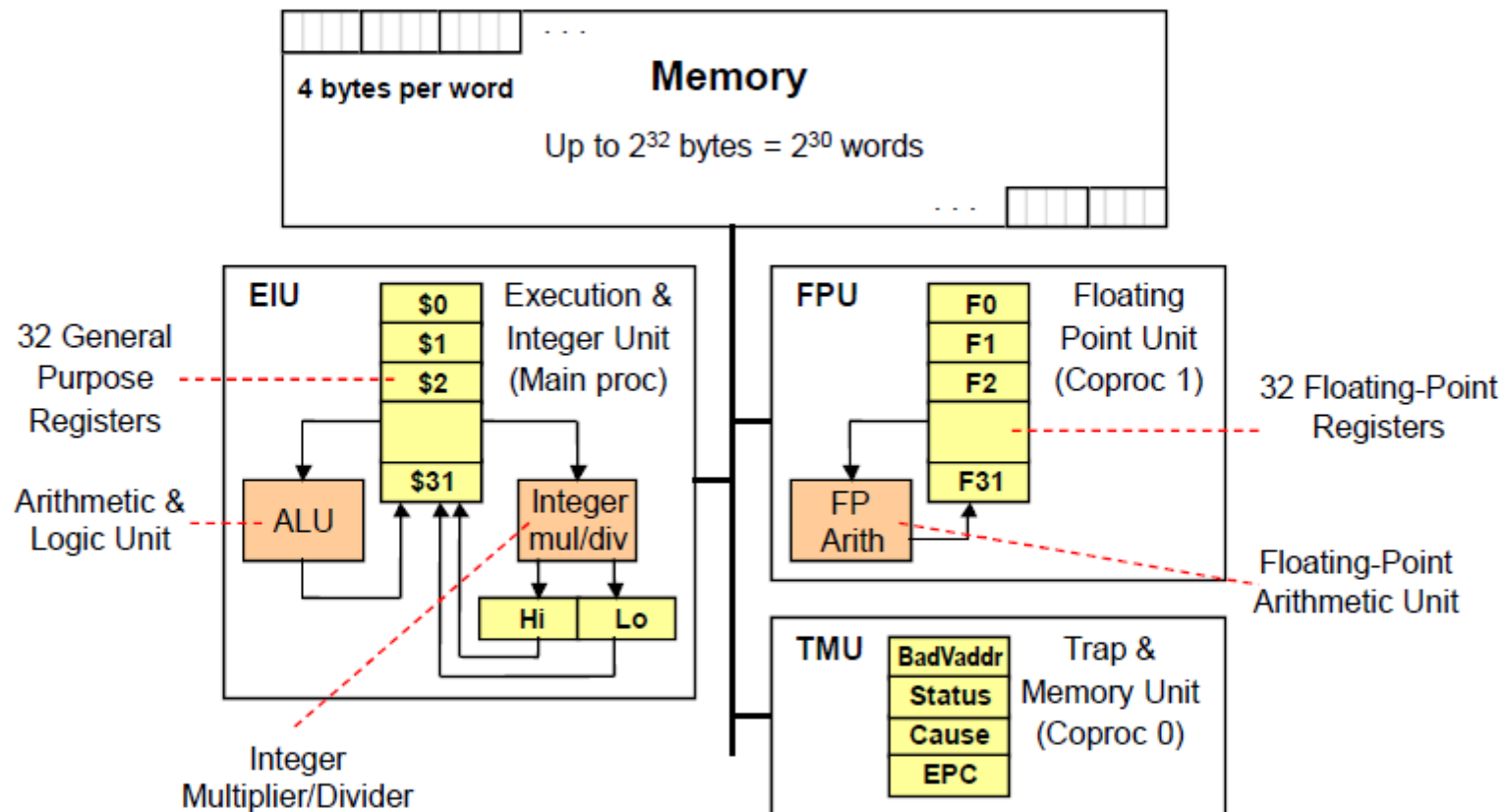
- ❖ Critical Interface between hardware and software
- ❖ An ISA includes the following ...
 - ✧ Instructions and Instruction Formats
 - ✧ Data Types, Encodings, and Representations
 - ✧ Programmable Storage: Registers and Memory
 - ✧ Addressing Modes: to address Instructions and Data
 - ✧ Handling Exceptional Conditions (like division by zero)
- ❖ Examples (Versions) Introduced in
 - ✧ Intel (8086, 80386, Pentium, ...) 1978
 - ✧ MIPS (MIPS I, II, III, IV, V) 1986
 - ✧ PowerPC (601, 604, ...) 1993

Instructions

- ❖ Instructions are the language of the machine
- ❖ We will study the MIPS instruction set architecture
 - ✧ Known as **Reduced Instruction Set Computer (RISC)**
 - ✧ Elegant and relatively simple design
 - ✧ Similar to RISC architectures developed in mid-1980's and 90's
 - ✧ Very popular, used in many products
 - Silicon Graphics, ATI, Cisco, Sony, etc.
 - ✧ Comes next in sales after Intel IA-32 processors
 - Almost 100 million MIPS processors sold in 2002 (and increasing)
- ❖ Alternative design: Intel IA-32
 - ✧ Known as **Complex Instruction Set Computer (CISC)**

- **The simplicity makes the MIPS architecture a favorite choice among universities and colleges for their introduction to computer architecture classes.**
- **This simplicity also makes the MIPS architecture very attractive to the embedded microprocessor market as it enables very cost-effective implementations**
- **load-store instruction set**
- **designed for pipelining efficiency**
- **fixed instruction set encoding**
- **efficient compiler target**
- **used in:**
 - **Nintendo 64**
 - **laser printers from HP and lexmark**
 - **advanced set-top boxes from Motorola and Sony**

Overview of the MIPS Architecture



Exceptions in MIPS

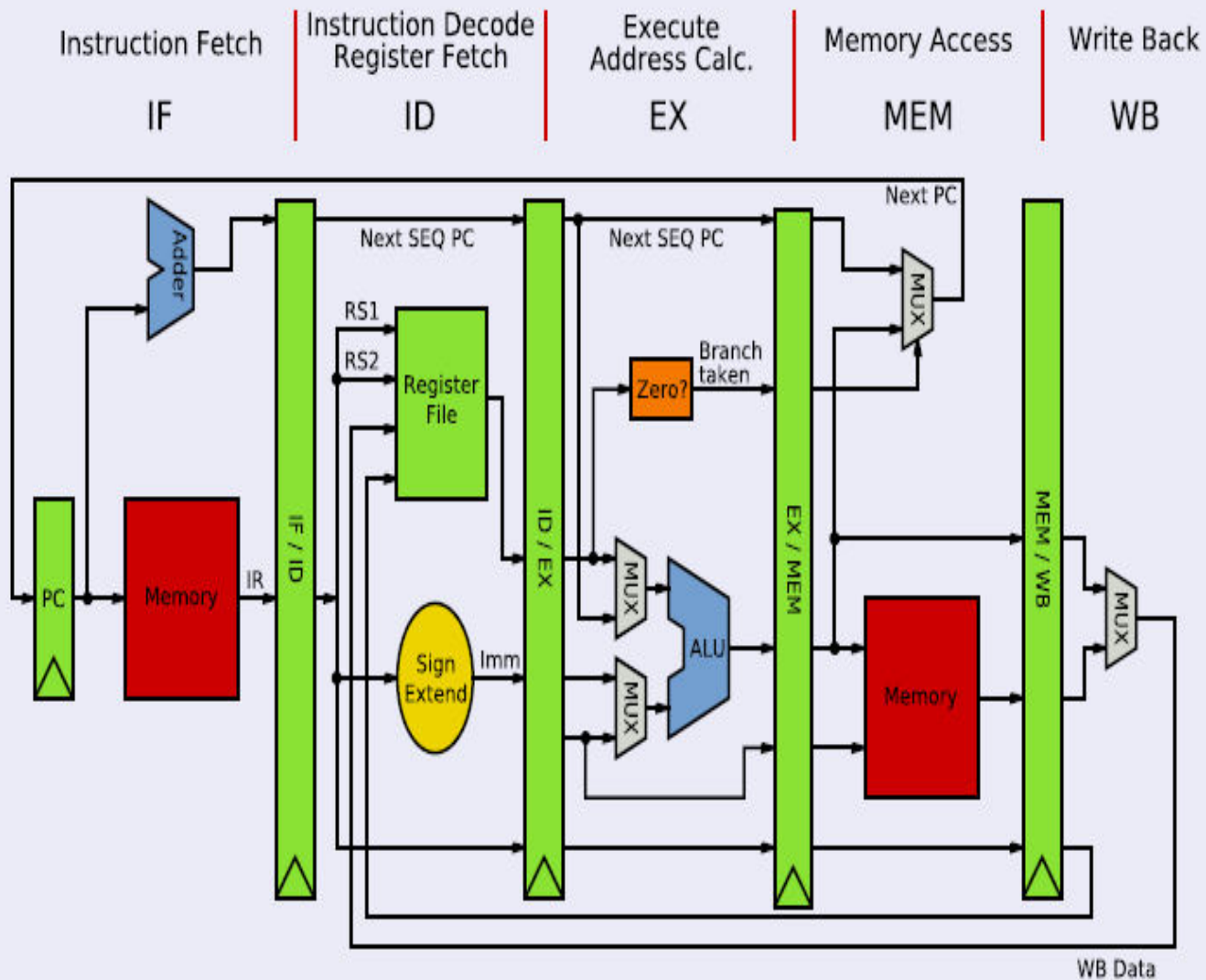
- An *interrupt* is an asynchronous exception. Synchronous exceptions, resulting directly from the execution of the program, are called *traps*.
- The CPU operates in one of the two possible modes, **user** and **kernel**.
- User programs run in user mode.
- The CPU enters the kernel mode when an exception happens.
- Coprocessor 0 can only be used in kernel mode.
- The whole upper half of the memory space is reserved for the kernel mode: it can not be accessed in user mode.

The relevant registers for the exception handling, in coprocessor 0 are

Exception handling registers in coprocessor 0

Register Number	Register Name	Usage
8	BadVAddr	Memory address where exception occurred
12	Status	Interrupt mask, enable bits, and status when exception occurred
13	Cause	Type of exception and pending interrupt bits
14	EPC	Address of instruction that caused exception

Microarchitecture



The Datapath Diagram

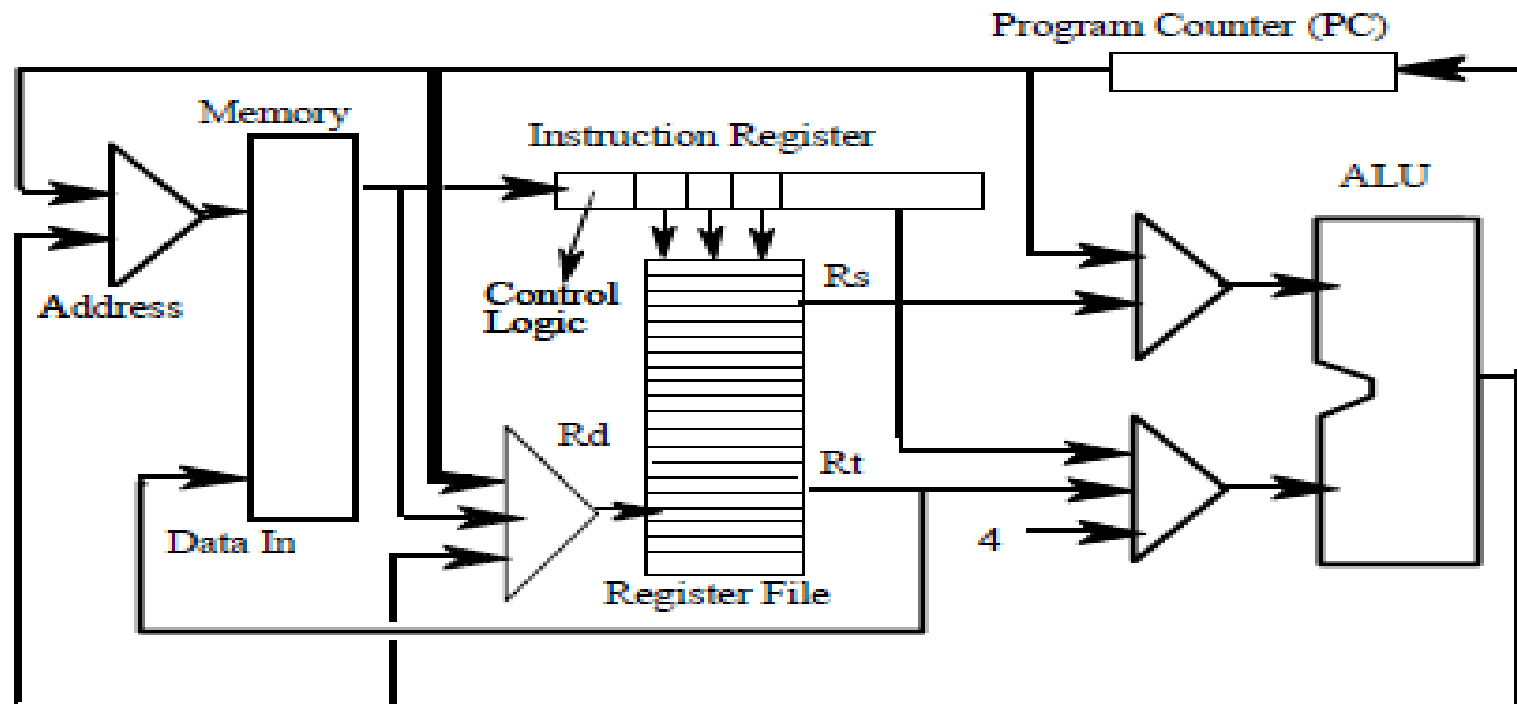


Figure 1.1 MIPS Simplified Datapath Diagram

How a computer executes a program

Fetch-decode-execute cycle (FDX)

1. **fetch** the next instruction from memory
2. **decode** the instruction
3. **execute** the instruction

Decode determines:

- operation to execute
- arguments to use
- where the result will be stored

Execute:

- performs the operation
- determines next instruction to fetch (by default, next one)

Datapath and control unit

Datapath

Major hardware components of the FDX cycle

- path of instructions and data through the processor
- components connected by **buses**

Bus – parallel path for transmitting values

- in MIPS, usually 32 bits wide

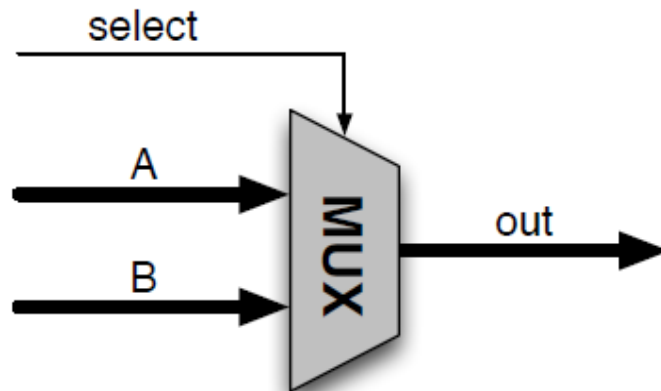
Datapath and control unit

Control unit

Controls the components of the datapath

- determines how data moves through the datapath
- receives **condition signals** from the components
- sends **control signals** to the components
- switches between buses with **multiplexers**

Multiplexer – component for choosing between buses

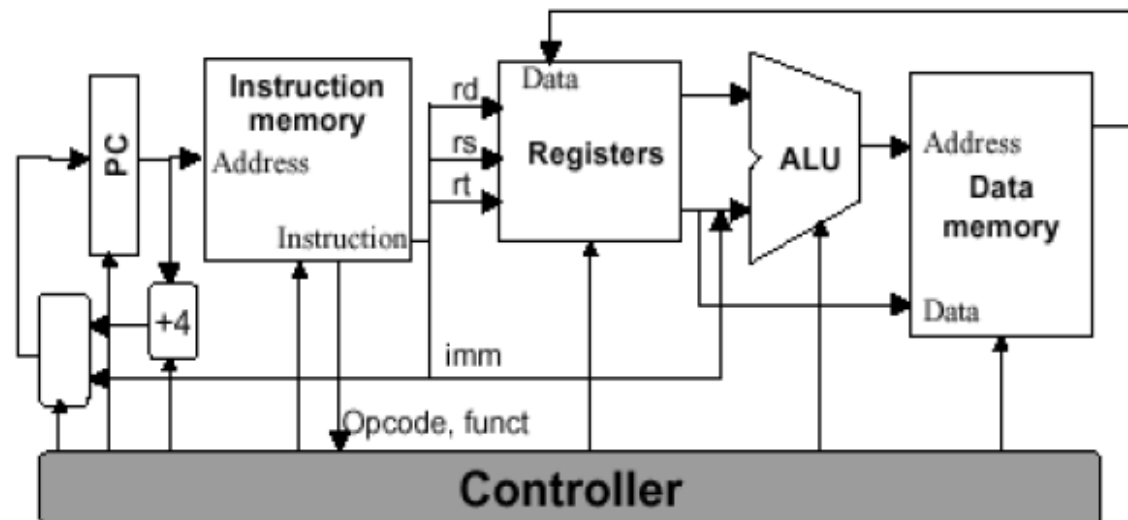


Components of the MIPS architecture

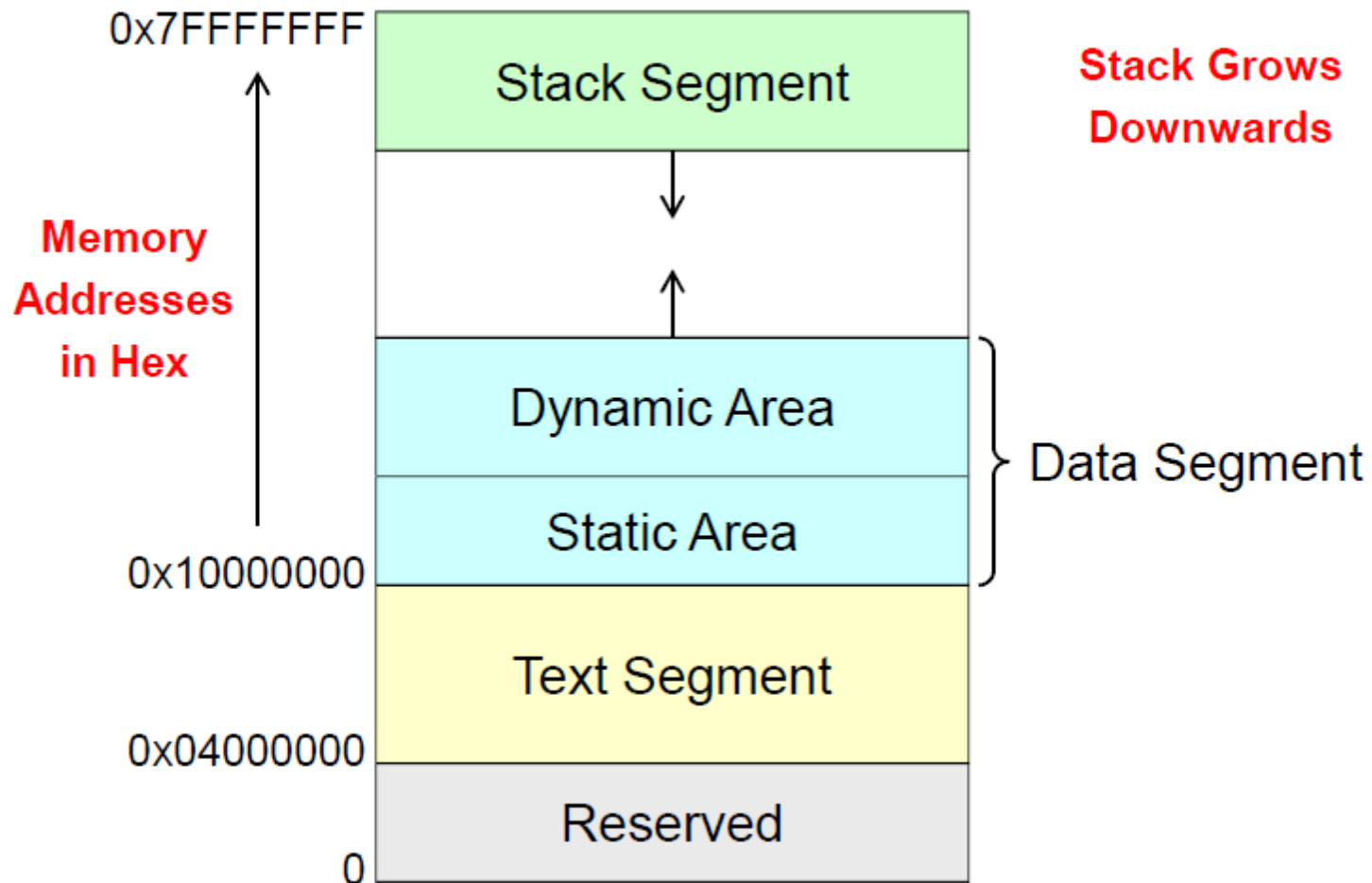
Major components of the datapath:

- program counter (PC)
- instruction register (IR)
- register file
- arithmetic and logic unit (ALU)
- memory

Control unit



Layout of a Program in Memory



Memory: text segment vs. data segment

In MIPS, programs are separated from data in memory

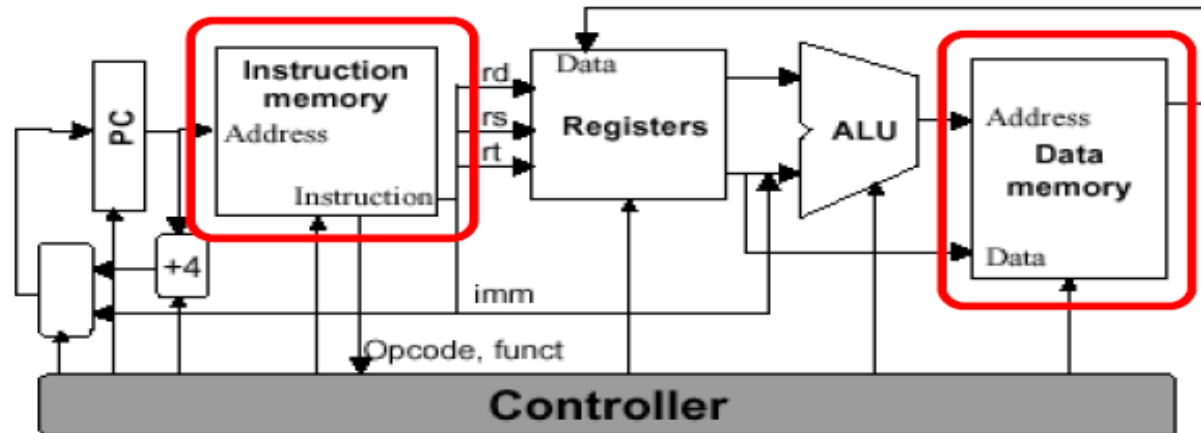
Text segment

- “instruction memory”
- part of memory that stores the program (machine code)
- **read only**

Data segment

- “data memory”
- part of memory that stores data manipulated by program
- **read/write**

Memory: text segment vs. data segment



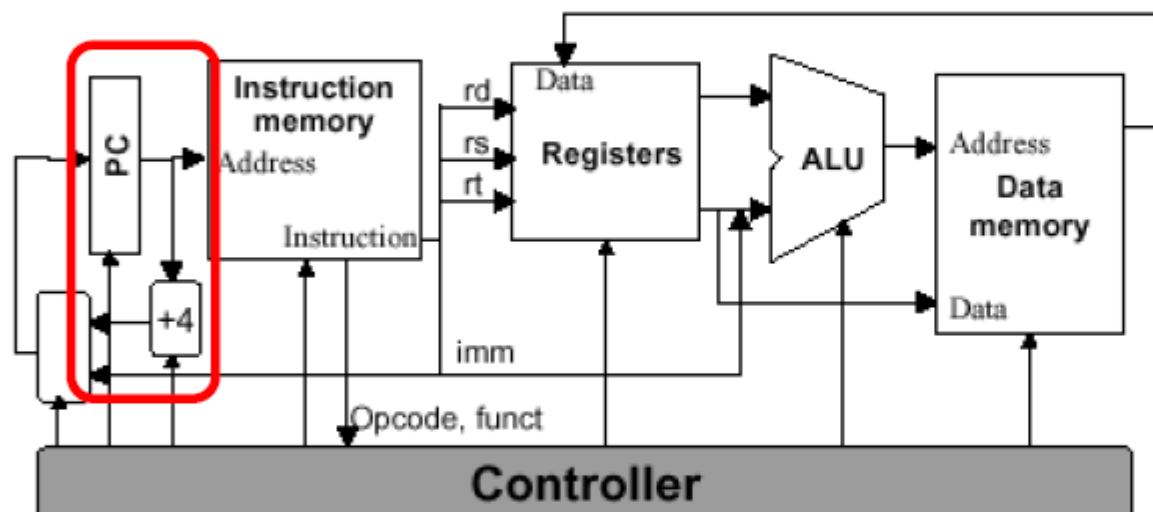
Distinction may or may not be reflected in the hardware:

- von Neumann architecture – single, shared memory
- Harvard architecture – physically separate memories

Program counter (PC)

Program: a sequence of machine instructions in the text segment

```
0x8d0b0000  
0x8d0c0004  
0x016c5020  
0xad0a0008  
0x21080004  
0x2129ffff  
0x1d20fff9
```



Program counter

Register that stores the **address** of the next instruction to fetch

- also called the instruction pointer (IP)

Incrementing the PC

In MIPS, each instruction is exactly 32-bits long

What is the address of the next instruction?

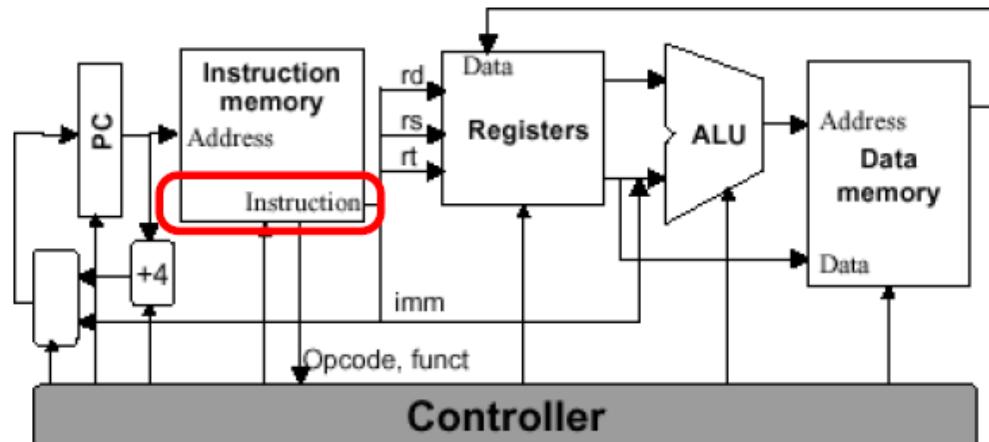
PC+4

(Each address refers to one byte, and $32/8 = 4$)

Instruction register (IR)

Instruction register

Register that holds the instruction currently being decoded

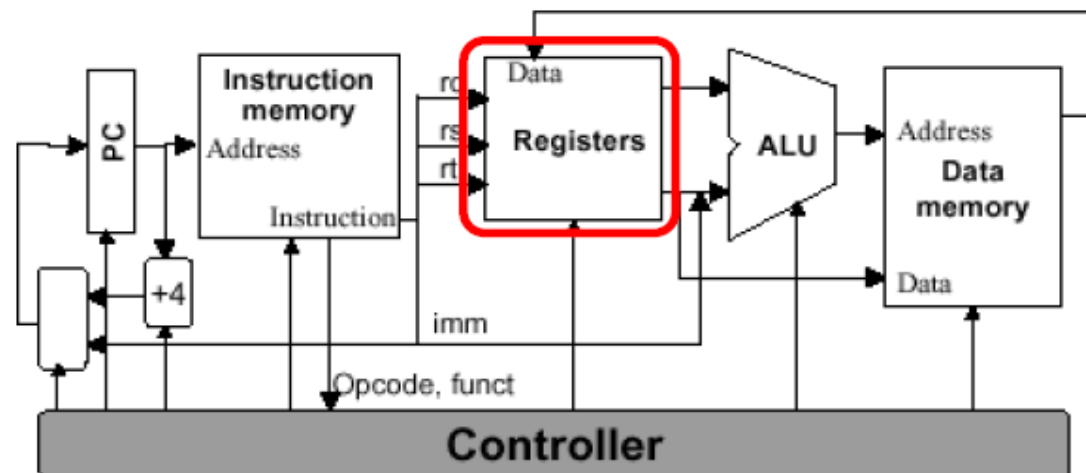


Note condition signals from the IR to the control unit!

Register file

Register: component that stores a 32-bit value

MIPS register file contains 32 registers



Arithmetic and logic unit (ALU)

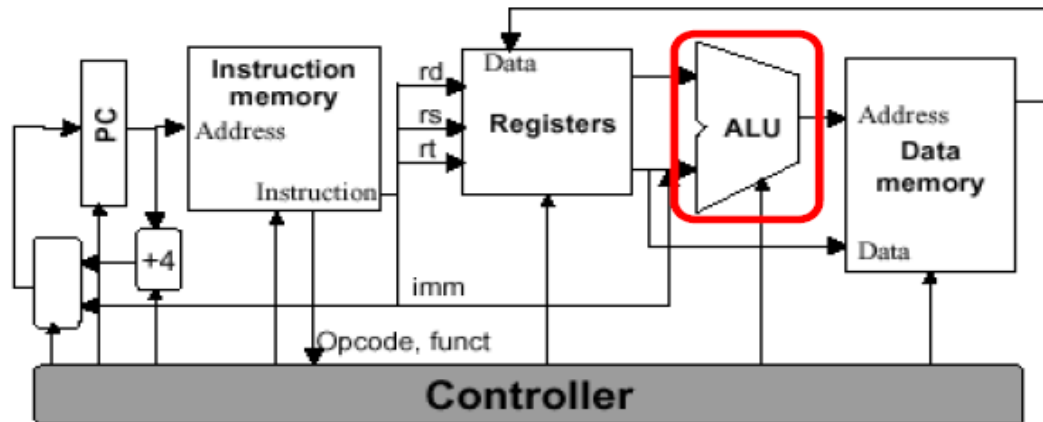
Implements binary arithmetic and logic operations

Inputs:

- operands – 2×32 -bit
- operation – control signal

Outputs:

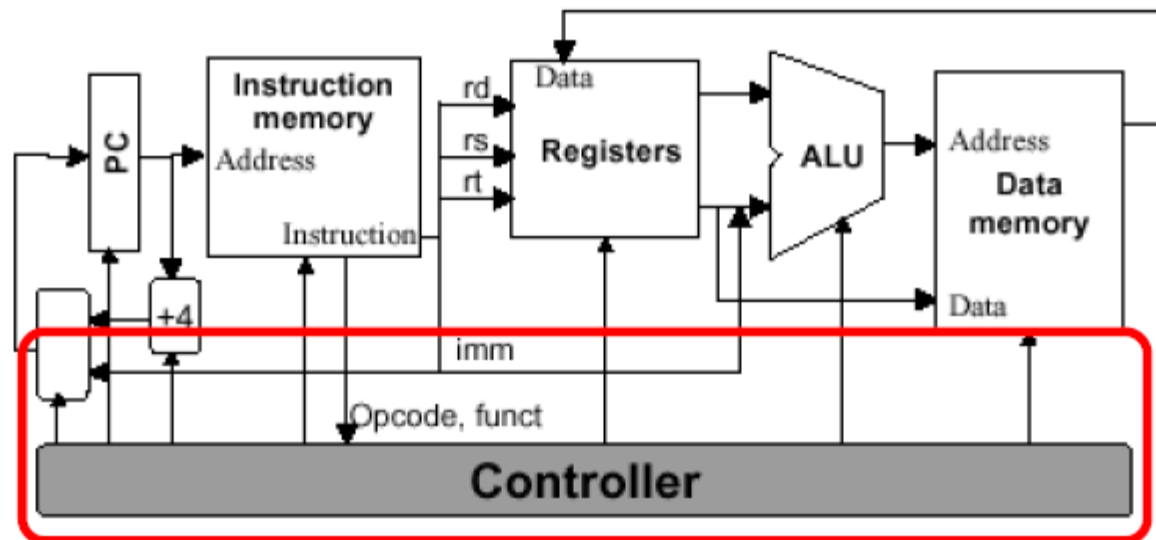
- result – 1×64 -bit
(usually just use 32 bits of this)
- status – condition signals



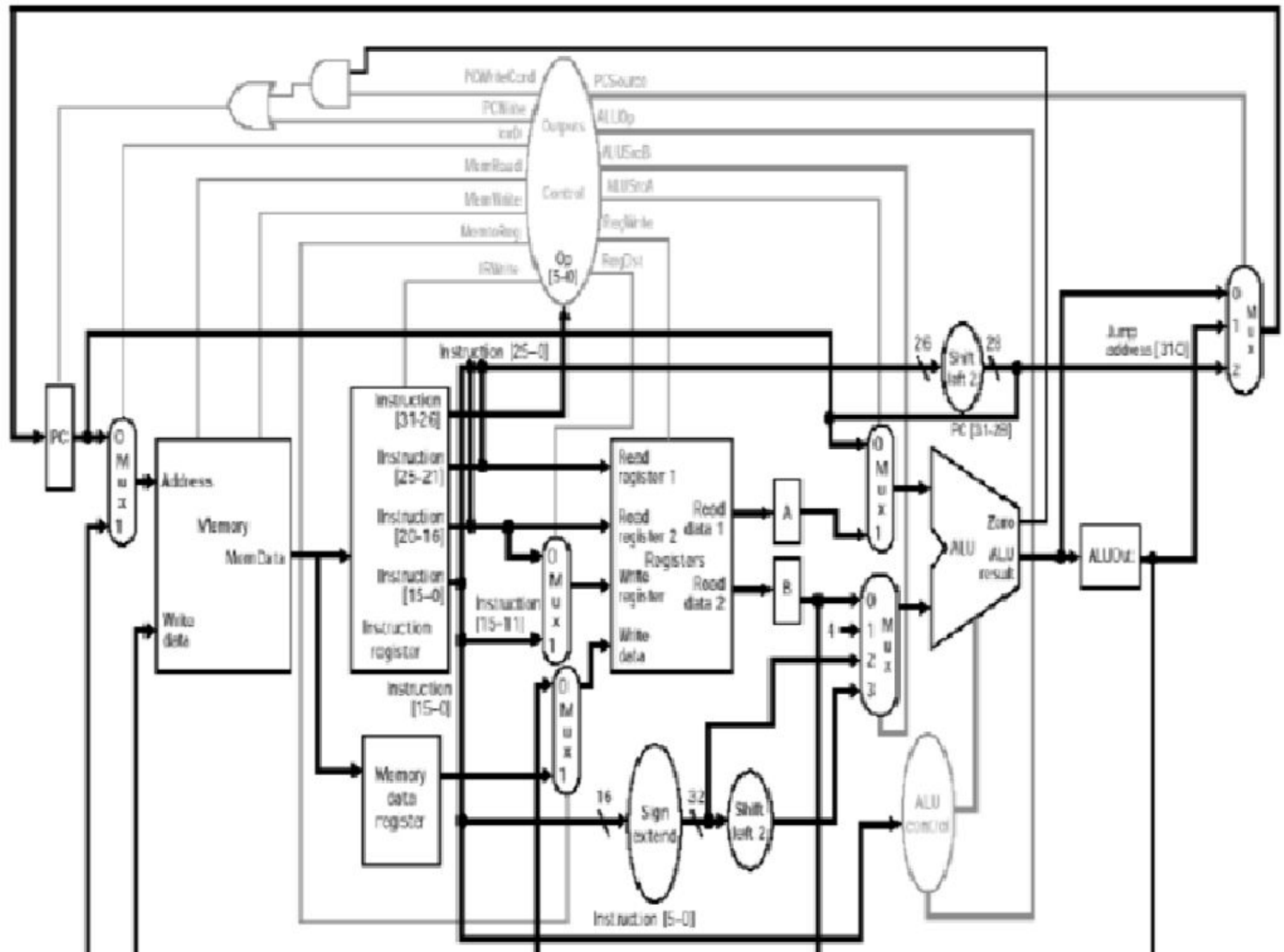
Control unit

Controls components of datapath to implement FDX cycle

- **Inputs:** condition signals
- **Outputs:** control signals



Implemented as a finite state machine



Control unit

Condition signals

- from IR – decode operation, arguments, result location
- from ALU – overflow, divide-by-zero, ...

Control signals

- to multiplexors – buses to select
- to each register – load new value
- to ALU – operation to perform
- to all – **clock signal**

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

- ✧ There are many registers (32)
- ✧ Software defines names to all registers
 - To standardize their use in programs
- ✧ Example: \$8 - \$15 are called \$t0 - \$t7
 - Used for **temporary** values

\$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



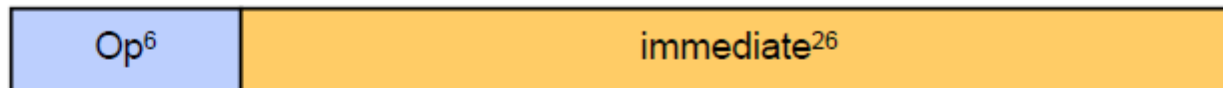
❖ Immediate (I-Type)

- ❖ 16-bit immediate constant is part in the instruction



❖ Jump (J-Type)

- ❖ Used by jump instructions



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

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, \dots, \$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 and y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x or y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

- ❖ AND instruction is used to clear bits: **x and 0 = 0**
- ❖ OR instruction is used to set bits: **x or 1 = 1**
- ❖ XOR instruction is used to toggle bits: **x xor 1 = 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 ^ \$s3	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x26
nor \$s1, \$s2, \$s3	\$s1 = ~(\$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

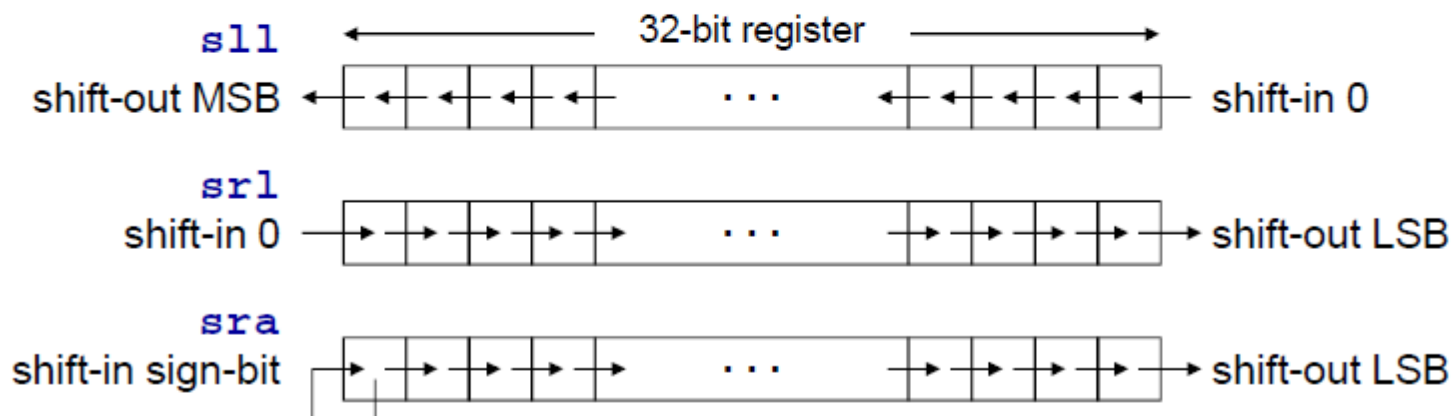
or \$s0, \$s1, \$s2 # \$s0 = 0xffff1234

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 = 0xabc1234, \$s3 = 16

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

sra \$s1,\$s2,4 \$s1 = \$s2>>4 \$s1 = 0xfabc123

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
```

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
```

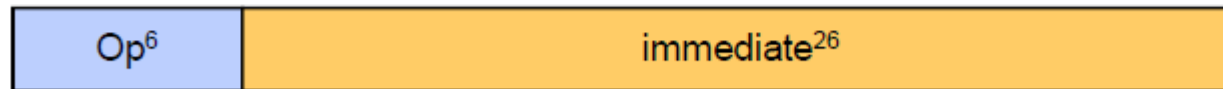
`$s1=$17`

load upper 16 bits	clear lower 16 bits
0xAC51	0x0000
0xAC51	0x65D9

```
ori $s1, $s1, 0x65D9
```

`$s1=$17`

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 =

PC ⁴	immediate ²⁶	00
-----------------	-------------------------	----

least-significant
2 bits are 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 $\$s1 = 1$ and $\$s0 = -1 = 0xffffffff$

`slt $t0,$s0,$s1` results in $\$t0 = 1$

`sltu $t0,$s0,$s1` results in $\$t0 = 0$

More on Branch Instructions


- ❖ MIPS hardware does NOT provide instructions for ...

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

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

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

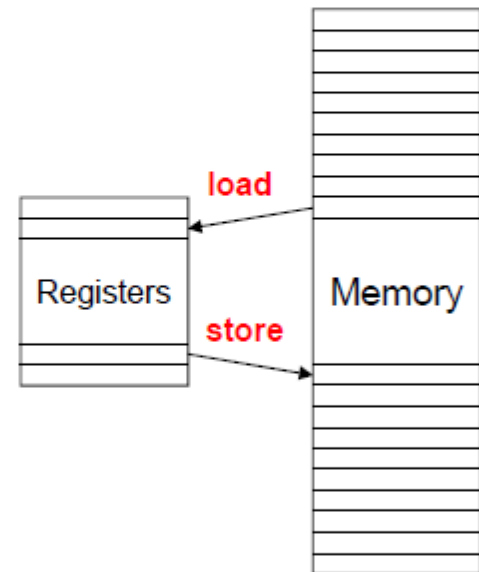
- ❖ **Load** Instruction:

- ✧ 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

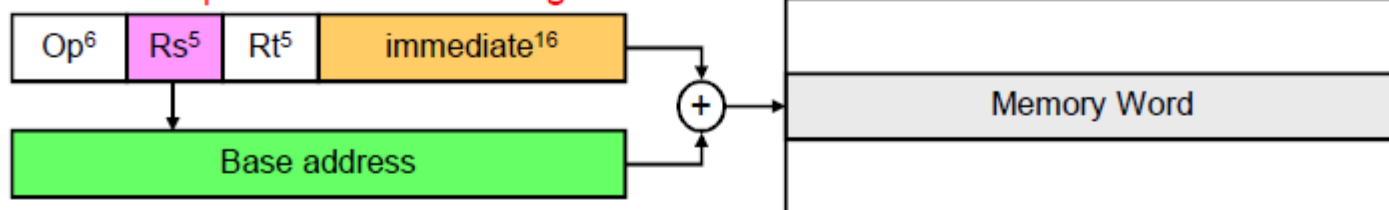
`sw Rt, imm16(Rs) # Rt → MEMORY[Rs+imm16]`

- ❖ 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



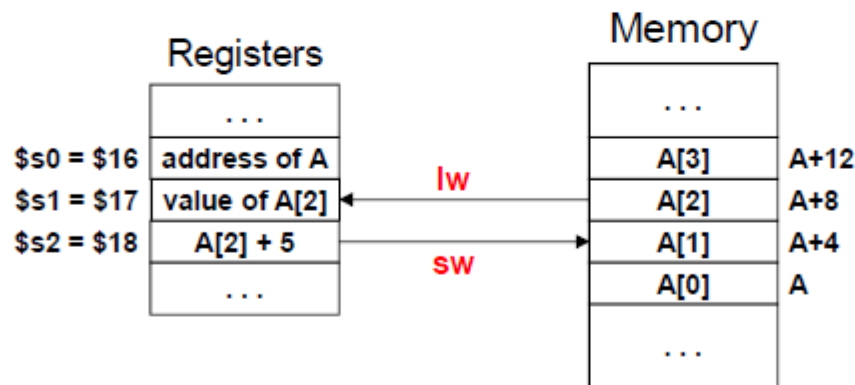
Example on Load & Store

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

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

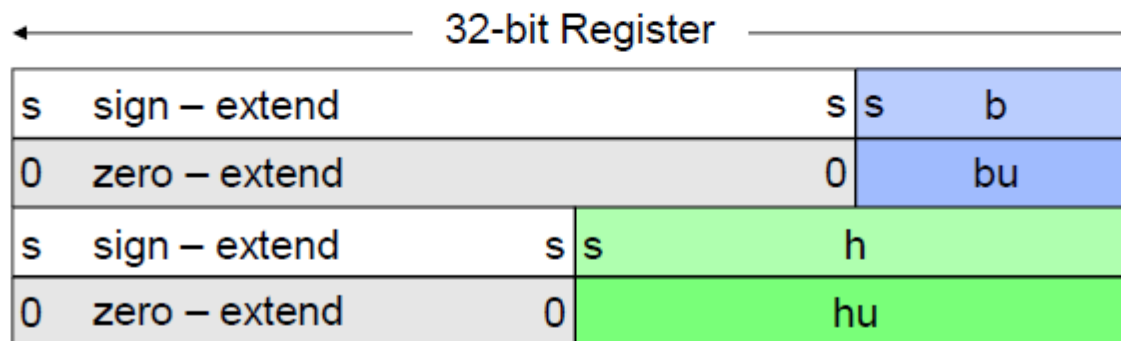
```
lw      $s1, 8($s0)      # $s1 = A[2]
addiu   $s2, $s1, 5      # $s2 = A[2] + 5
sw      $s2, 4($s0)      # A[1] = $s2
```

❖ 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)

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

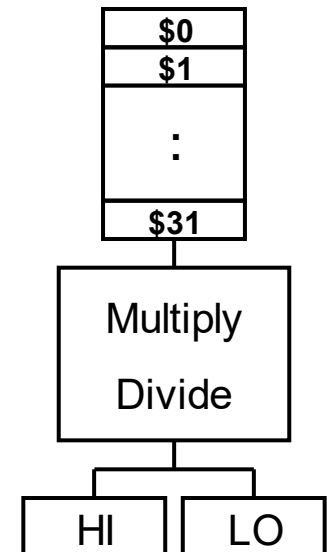
```
sum = 0;  
for (i=0; i<n; i++) sum = sum + A[i];
```

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)
```

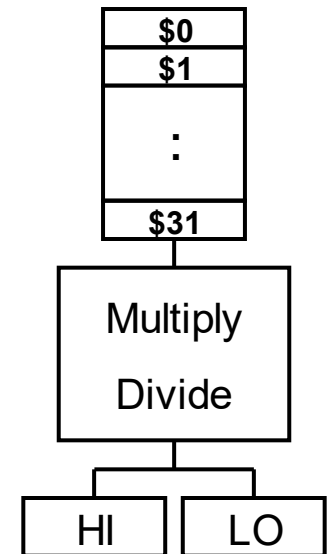
Integer Multiplication in MIPS

- Multiply instructions
 - `mult $s1,$s2` **Signed multiplication**
 - `multu $s1,$s2` **Unsigned multiplication**
- 32-bit multiplication produces a 64-bit Product
- Separate pair of 32-bit registers
 - **HI = high-order 32-bit of product**
 - **LO = low-order 32-bit of product**
- MIPS also has a special `mul` instruction
 - `mul $s0,$s1,$s2` **$\$s0 = \$s1 \times \$s2$**
 - **Put low-order 32 bits into destination register**
 - **HI & LO are undefined**



Integer Division in MIPS

- Divide instructions
 - `div $s1,$s2` **Signed division**
 - `divu $s1,$s2` **Unsigned division**
- Division produces quotient and remainder
- Separate pair of 32-bit registers
 - **HI = 32-bit remainder**
 - **LO = 32-bit quotient**
 - If divisor is 0 then result is **unpredictable**
- Moving data from HI/LO to MIPS registers
 - `mfhi Rd` (move from HI to Rd)
 - `mflo Rd` (move from LO to Rd)



Integer Multiply/Divide Instructions

Instruction	Meaning	Format						
mult Rs, Rt	Hi, Lo = Rs × Rt	op ⁶ = 0	Rs ⁵	Rt ⁵	0	0	0	0x18
multu Rs, Rt	Hi, Lo = Rs × Rt	op ⁶ = 0	Rs ⁵	Rt ⁵	0	0	0	0x19
mul Rd, Rs, Rt	Rd = Rs × Rt	0x1c	Rs ⁵	Rt ⁵	Rd ⁵	0	0	0x02
div Rs, Rt	Hi, Lo = Rs / Rt	op ⁶ = 0	Rs ⁵	Rt ⁵	0	0	0	0x1a
divu Rs, Rt	Hi, Lo = Rs / Rt	op ⁶ = 0	Rs ⁵	Rt ⁵	0	0	0	0x1b
mfhi Rd	Rd = Hi	op ⁶ = 0	0	0	Rd ⁵	0	0	0x10
mflo Rd	Rd = Lo	op ⁶ = 0	0	0	Rd ⁵	0	0	0x12

- ❖ 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

Integer to String Conversion

- Objective: convert an unsigned 32-bit integer to a string
- How to obtain the decimal digits of the number?
 - Divide the number by 10, Remainder = decimal digit (0 to 9)
 - Convert decimal digit into its ASCII representation ('0' to '9')
 - Repeat the division until the quotient becomes zero
 - Digits are computed **backwards** from least to most significant
- Example: convert 2037 to a string
 - Divide 2037/10 quotient = 203 remainder = 7 char = '7'
 - Divide 203/10 quotient = 20 remainder = 3 char = '3'
 - Divide 20/10 quotient = 2 remainder = 0 char = '0'
 - Divide 2/10 quotient = 0 remainder = 2 char = '2'

Integer to String Procedure

```
#-----
# int2str:  Converts an unsigned integer into a string
# Input:    $a0 = unsigned integer
# In/Out:   $a1 = address of string buffer (12 bytes)
#-----
int2str:
    move     $t0, $a0          # $t0 = dividend = unsigned integer
    li       $t1, 10           # $t1 = divisor = 10
    addiu    $a1, $a1, 11      # start at end of string buffer
    sb       $zero, 0($a1)     # store a NULL byte
convert:
    divu     $t0, $t1          # LO = quotient, HI = remainder
    mflo     $t0               # $t0 = quotient
    mfhi     $t2               # $t2 = remainder
    addiu    $t2, $t2, 0x30     # convert digit to a character
    addiu    $a1, $a1, -1      # point to previous byte
    sb       $t2, 0($a1)       # store digit character
    bnez     $t0, convert       # loop if quotient is not 0
    jr       $ra               # return to caller
```

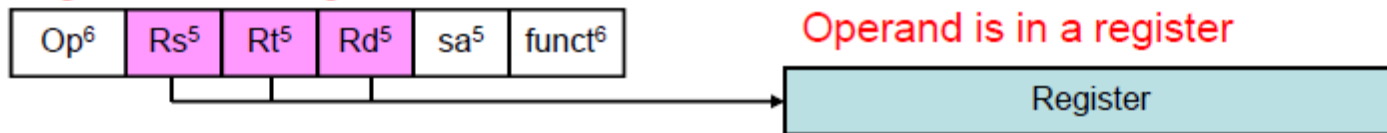
Addressing Modes

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

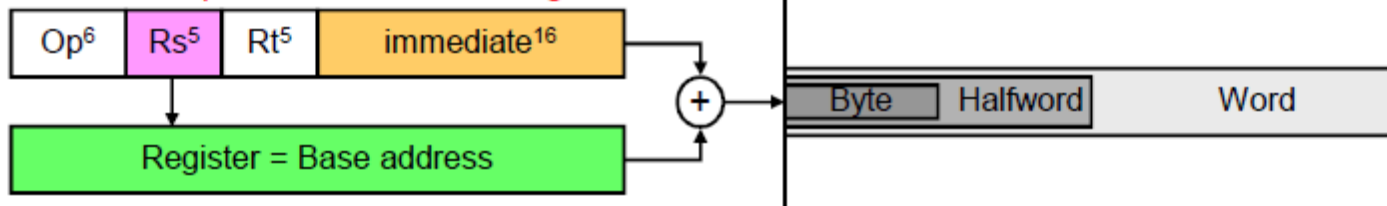
Immediate Addressing



Register Addressing



Base or Displacement Addressing

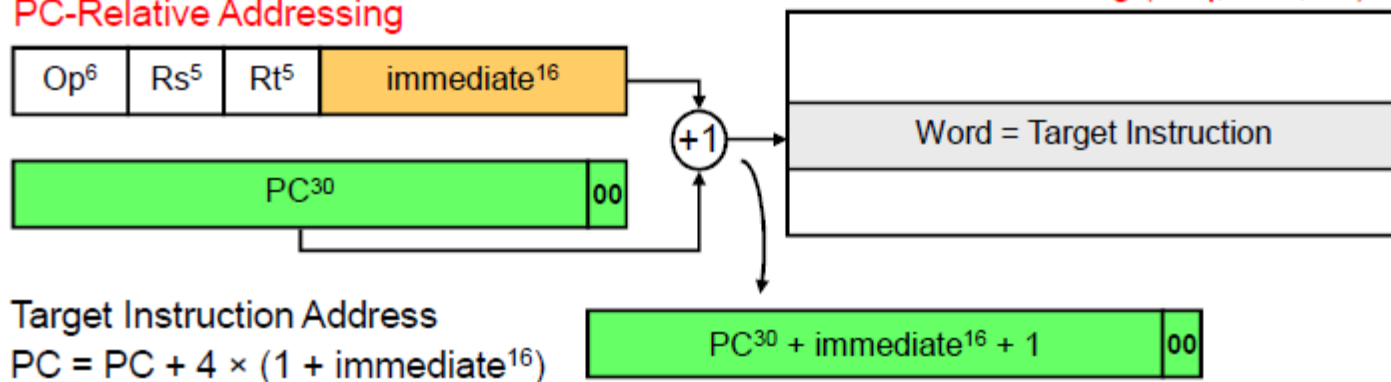


Operand is in memory (load/store)

Branch / Jump Addressing Modes

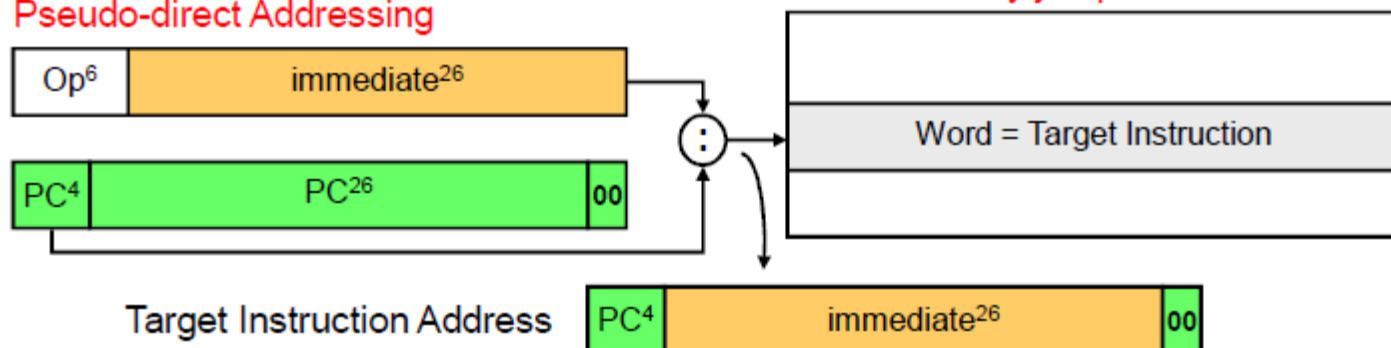
PC-Relative Addressing

Used for branching (beq, bne, ...)



Pseudo-direct Addressing

Used by jump instruction

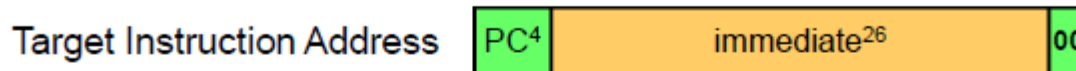


Jump and Branch Limits

❖ 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



❖ Branch Address Boundary

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

✧ PC-relative addressing:

$PC^{30} + \text{immediate}^{16} + 1$	00
---------------------------------------	----

- 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)

Quick Reference

Integer Instruction Set

Name	Syntax
Add: _____	add Rd, Rs, Rt
Add Immediate: _____	addi Rt, Rs, Imm
Add Immediate Unsigned: _____	addiu Rt, Rs, Imm
Add Unsigned: _____	addu Rd, Rs, Rt
And: _____	and Rd, Rs, Rt
And Immediate: _____	andi Rt, Rs, Imm
Branch if Equal: _____	beq Rs, Rt, Label
Branch if Greater Than or Equal to Zero: _____	bgez Rs, Label
Branch if Greater Than or Equal to Zero and Link: _____	bgezal Rs, Label
Branch if Greater Than Zero: _____	bgtz Rs, Label
Branch if Less Than or Equal to Zero: _____	blez Rs, Label
Branch if Less Than Zero and Link: _____	bltzal Rs, Label
Branch if Less Than Zero: _____	bltz Rs, Label
Branch if Not Equal: _____	bne Rs, Rt, Label
Divide: _____	div Rs, Rt
Divide Unsigned: _____	divu Rs, Rt
Jump: _____	j Label
Jump and Link: _____	jal Label
Jump and Link Register: _____	jalr Rd, Rs
Jump Register: _____	jr Rs
Load Byte: _____	lb Rt, offset(Rs)
Load Byte Unsigned: _____	lbu Rt, offset(Rs)
Load Halfword: _____	lh Rt, offset(Rs)
Load Halfword Unsigned: _____	lhu Rt, offset(Rs)
Load Upper Immediate: _____	lui Rt, Imm
Load Word: _____	lw Rt, offset(Rs)
Load Word Left: _____	lwl Rt, offset(Rs)
Load Word Right: _____	lwr Rt, offset(Rs)
Move From High: _____	mfhi Rd
Move From Low: _____	mflo Rd
Move to High: _____	mthi Rs
Move to Low: _____	mtlo Rs
Multiply: _____	mult Rs, Rt
Multiply Unsigned: _____	multu Rs, Rt
NOR: _____	nor Rd, Rs, Rt
OR: _____	or Rd, Rs, Rt
OR Immediate: _____	ori Rt, Rs, Imm

Store Byte:_____	sb	Rt, offset(Rs)
Store Halfword:_____	sh	Rt, offset(Rs)
Shift Left Logical:_____	sll	Rd, Rt, sa
Shift Left Logical Variable:_____	sllv	Rd, Rt, Rs
Set on Less Than:_____	slt	Rd, Rt, Rs
Set on Less Than Immediate:_____	slti	Rt, Rs, Imm
Set on Less Than Immediate Unsigned:_____	sltiu	Rt, Rs, Imm
Set on Less Than Unsigned:_____	sltu	Rd, Rt, Rs
Shift Right Arithmetic:_____	sra	Rd, Rt, sa
Shift Right Arithmetic Variable:_____	srav	Rd, Rt, Rs
Shift Right Logical:_____	srl	Rd, Rt, sa
Shift Right Logical Variable:_____	srlv	Rd, Rt, Rs
Subtract:_____	sub	Rd, Rs, Rt
Subtract Unsigned:_____	subu	Rd, Rs, Rt
Store Word:_____	sw	Rt, offset(Rs)
Store Word Left:_____	swl	Rt, offset(Rs)
Store Right:_____	swr	Rt, offset(Rs)
System Call:_____	syscall	
Exclusive OR:_____	xor	Rd, Rs, Rt
Exclusive OR Immediate:_____	xori	Rt, Rs, Imm

Macro instructions

Name	Syntax	
Absolute Value:_____	abs	Rd, Rs
Branch if Equal to Zero:_____	beqz	Rs, Label
Branch if Greater Than or Equal :	bge	Rs, Rt, Label
Branch if Greater Than or Equal Unsigned:_____	bgeu	Rs, Rt, Label
Branch if Greater Than:_____	bgt	Rs, Rt, Label
Branch if Greater Than Unsigned:_____	bgtu	Rs, Rt, Label
Branch if Less Than or Equal:_____	ble	Rs, Rt, Label
Branch if Less Than or Equal Unsigned:_____	bleu	Rs, Rt, Label
Branch if Less Than:_____	blt	Rs, Rt, Label
Branch if Less Than Unsigned:_____	bltu	Rs, Rt, Label
Branch if Not Equal to Zero:_____	bnez	Rs, Label
Branch Unconditional:_____	b	Label
Divide:_____	div	Rd, Rs, Rt
Divide Unsigned:_____	divu	Rd, Rs, Rt
Load Address:_____	la	Rd, Label
Load Immediate:_____	li	Rd, value
Move:_____	move	Rd, Rs
Multiply:_____	mul	Rd, Rs, Rt
Multiply (with overflow exception):_____	mulo	Rd, Rs, Rt
Multiply Unsigned (with overflow exception):_____	mulou	Rd, Rs, Rt
Negate:_____	neg	Rd, Rs
Negate Unsigned:_____	negu	Rd, Rs
Nop:_____	nop	

Not:_____	not	Rd, Rs
Remainder Unsigned:_____	remu	Rd, Rs, Rt
Rotate Left Variable:_____	rol	Rd, Rs, Rt
Rotate Right Variable:_____	ror	Rd, Rs, Rt
Remainder:_____	rem	Rd, Rs, Rt
Rotate Left Constant:_____	rol	Rd, Rs, sa
Rotate Right Constant:_____	ror	Rd, Rs, sa
Set if Equal:_____	seq	Rd, Rs, Rt
Set if Greater Than or Equal:_____	sge	Rd, Rs, Rt
Set if Greater Than or Equal Unsigned:_____	sgeu	Rd, Rs, Rt
Set if Greater Than:_____	sgt	Rd, Rs, Rt
Set if Greater Than Unsigned:_____	sgtu	Rd, Rs, Rt
Set if Less Than or Equal:_____	sle	Rd, Rs, Rt
Set if Less Than or Equal Unsigned:_____	sleu	Rd, Rs, Rt
Set if Not Equal:_____	sne	Rd, Rs, Rt
Unaligned Load Halfword Unsigned:_____	ulh	Rd, n(Rs)
Unaligned Load Halfword:_____	ulhu	Rd, n(Rs)
Unaligned Load Word:_____	ulw	Rd, n(Rs)
Unaligned Store Halfword:_____	ush	Rd, n(Rs)
Unaligned Store Word:_____	usw	Rd, n(Rs)