Instruction Set Architecture

Outline

- 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
- Addressing Modes

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) First Introduced in
  - Intel (8086, 80386, Pentium, ...) 1978
  - MIPS (MIPS I, II, III, IV, V) 1986
  - PowerPC (601, 604, …) 1993

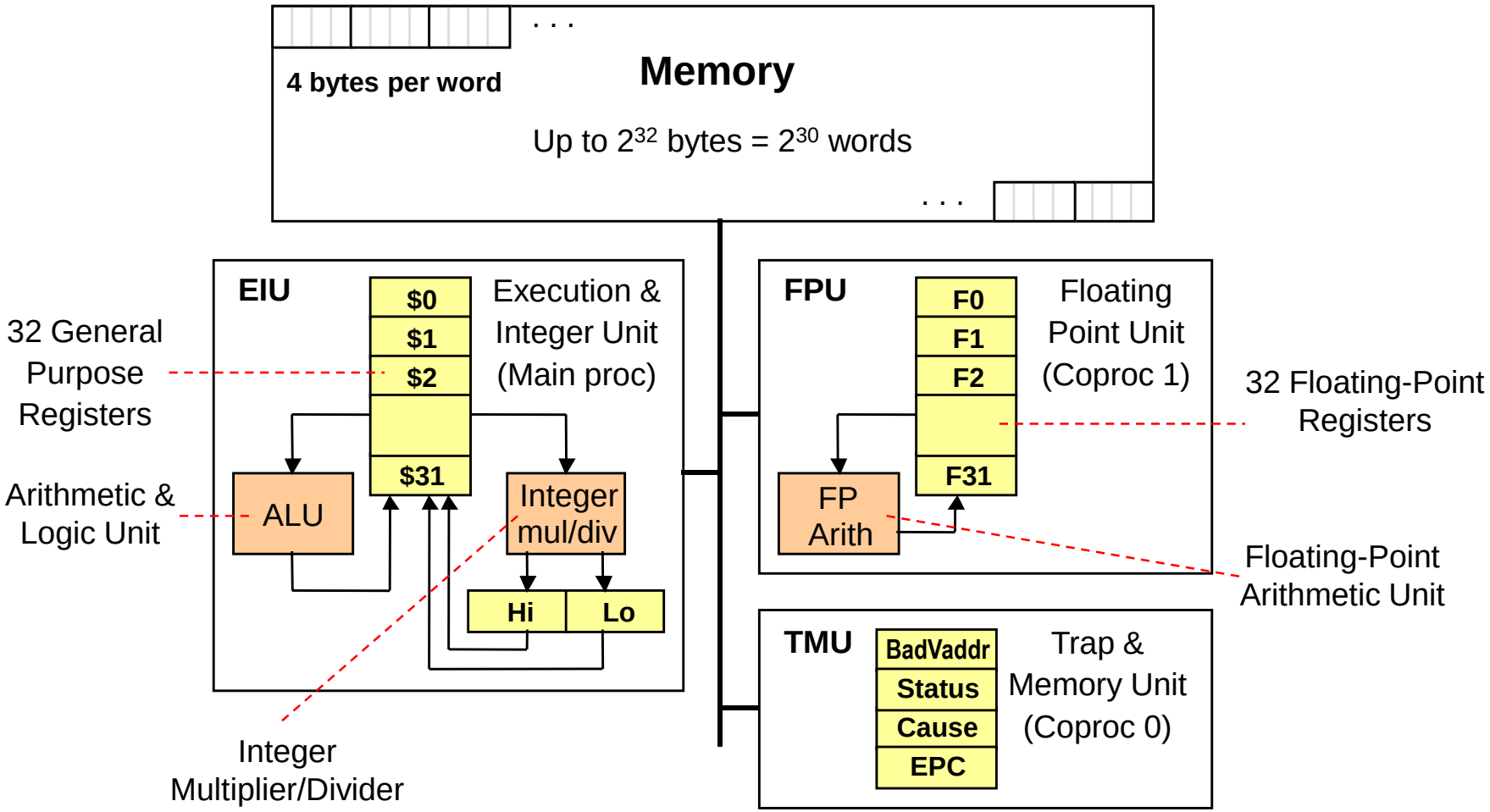
Instructions

- 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)**

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
- Addressing Modes

Overview of the MIPS Processor



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

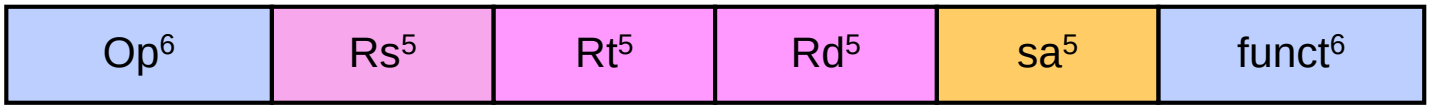
MIPS Register Conventions

- 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

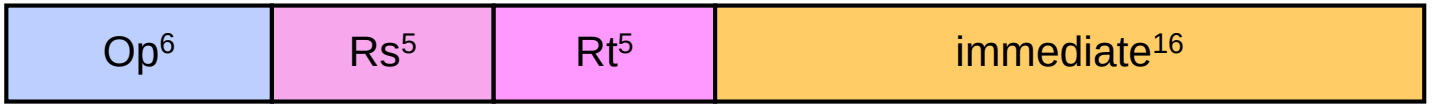
Name	Register	Usage
`$zero`	`$0`	Always 0 (forced by hardware)
`$at`	`$1`	Reserved for assembler use
`$v0 – $v1`	`$2 – $3`	Result values of a function
`$a0 – $a3`	`$4 – $7`	Arguments of a function
`$t0 – $t7`	`$8 – $15`	Temporary Values
`$s0 – $s7`	`$16 – $23`	Saved registers (preserved across call)
`$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

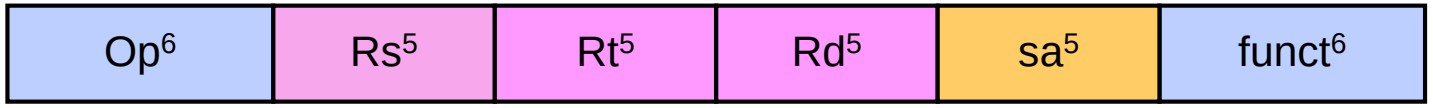
Instruction Categories

- 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

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
- Addressing Modes

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* 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

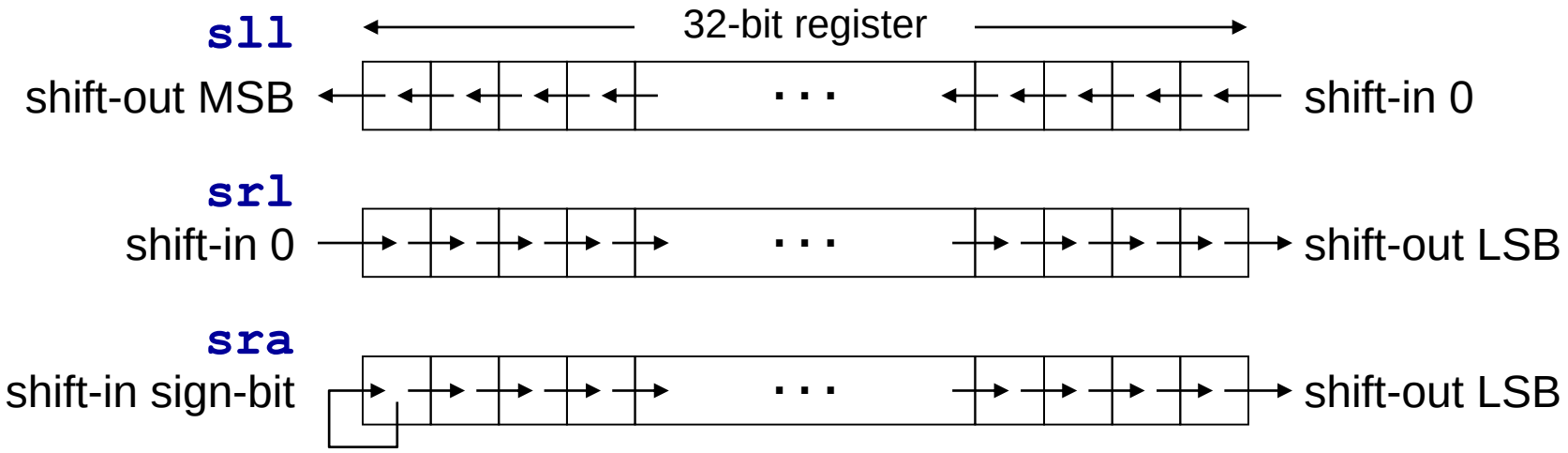
- Examples:

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 = 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

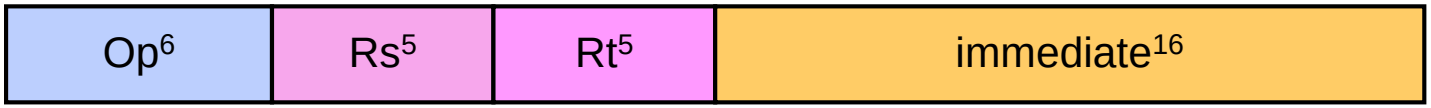
```
sll   $s2, $s1, 5          ; $s2 = $s1 * 32
subu  $s2, $s2, $s1        ; $s2 = $s1 * 31
```

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
- Addressing Modes

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 ^ 10	op = 0xe	rs = $s2	rt = $s1	imm^{16} = 10
lui $s1, 10	$s1 = 10 << 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
```

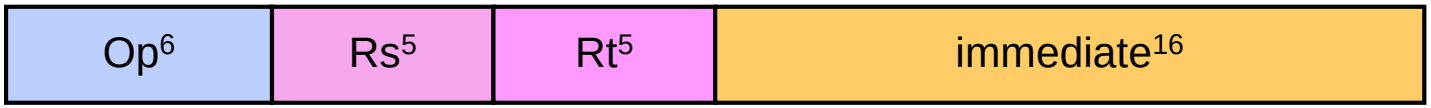
op=001001	rs=$s1=10001	rt=$s2=10010	imm = -1 = 1111111111111111

```
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**

		load upper 16 bits	clear lower 16 bits
`lui $s1,0xAC51`	`$s1=$17`	`0xAC51`	`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
- Addressing Modes

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
  - 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 < im16) 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 = 1`
`sltu`	`$t0,$s0,$s1`	results in	`$t0 = 0`

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
`move $s1, $s2`	`addu $s1, $zero, $s2`
`not $s1, $s2`	`nor $s1, $s2, $zero`
`li $s1, 0xabcd`	`ori $s1, $zero, 0xabcd`
`li $s1, 0xabcd1234`	`lui $at, 0xabcd` `ori $s1, $at, 0x1234`
`sgt $s1, $s2, $s3`	`slt $s1, $s3, $s2`
`blt $s1, $s2, label`	`slt $at, $s1, $s2` `bne $at, $zero, label`

- 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
- Addressing Modes

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
}
```

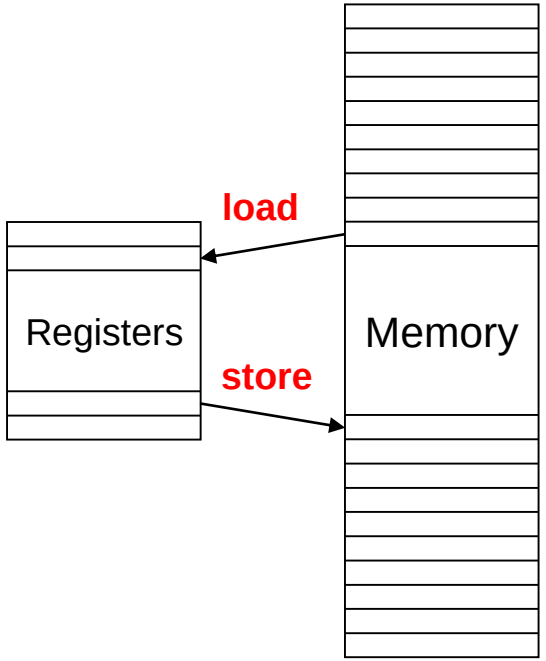
```
    bgt  $s3, $s4, next
    ble  $s4, $s5, next
    addu $s3, $s4, $s5
next:
```

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
- Addressing Modes

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)

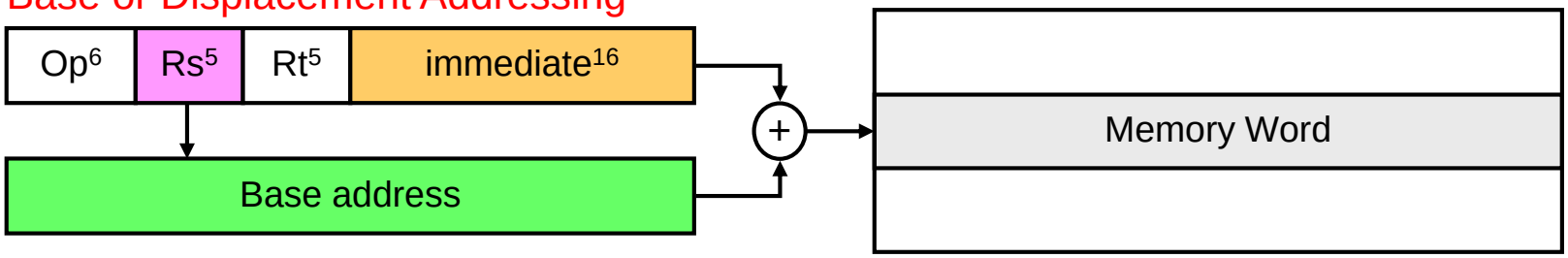
```
lw Rt, imm16(Rs)   # Rt = MEMORY[Rs+imm16]
```

- Store Word Instruction

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

- Base or Displacement addressing is used
  - Memory Address = Rs (base) + Immediate^{16} (displacement)
  - Immediate^{16} is sign-extended to have a signed displacement

Base or Displacement Addressing

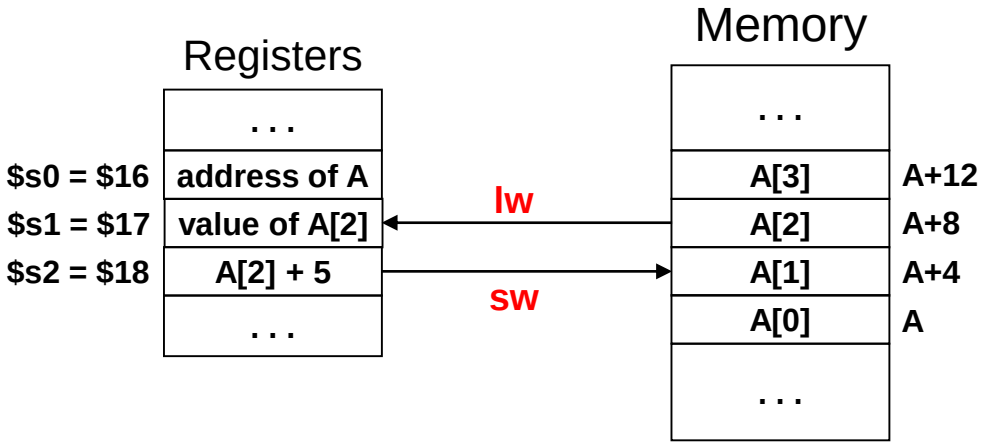


Example on Load & Store

- Translate A[1] = A[2] + 5 (A is an array of words)
  - Assume that address of array A is stored in register $s0

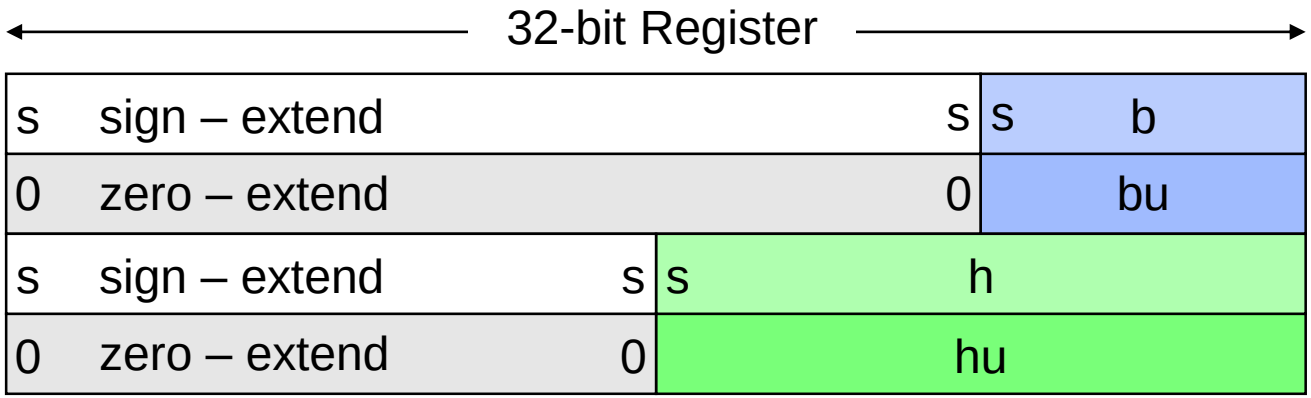
```
lw      $s1, 8($s0)        # $s1 = A[2]
addiu   $s2, $s1, 5        # $s2 = A[2] + 5
sw      $s2, 4($s0)        # A[1] = $s2
```

- 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) + $Immediate^{16}$ (displacement)
- Two variations on base addressing
  - If Rs = $zero = 0 then Address = $Immediate^{16}$ (absolute)
  - If $Immediate^{16}$ = 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
- Addressing Modes

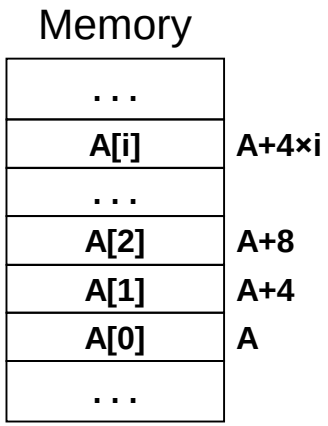
Translating a WHILE Loop

- 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



- 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

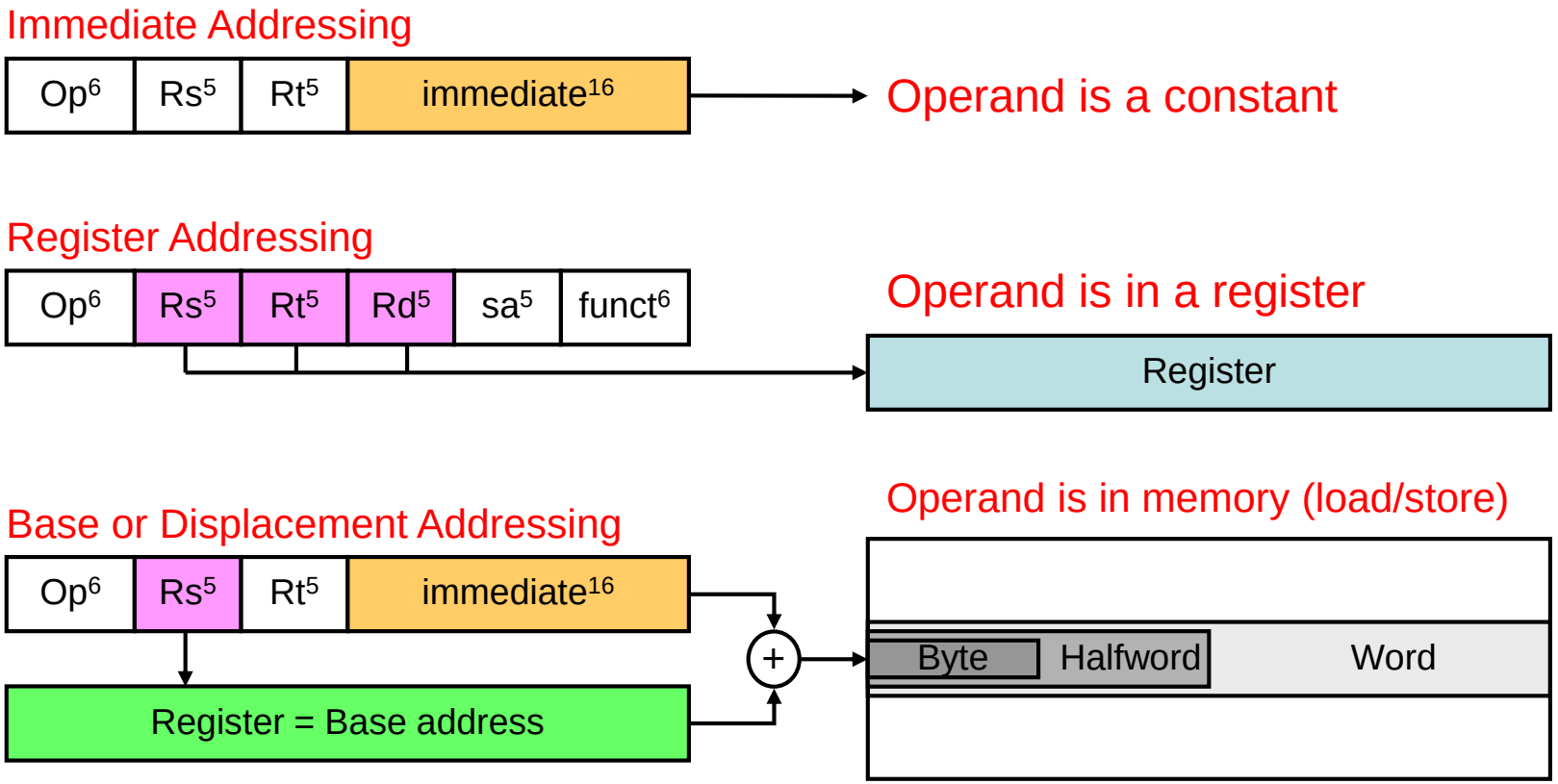
```
    move   $t0, $s0          # $t0 = address A[i]
    xor    $t1, $t1, $t1     # $t1 = i = 0
    xor    $s2, $s2, $s2     # $s2 = sum = 0
L1: lw     $t2, 0($t0)       # $t2 = A[i]
    addu   $s2, $s2, $t2     # sum = sum + A[i]
    addiu  $t0, $t0, 4       # point to next A[i]
    addiu  $t1, $t1,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

Next . . .

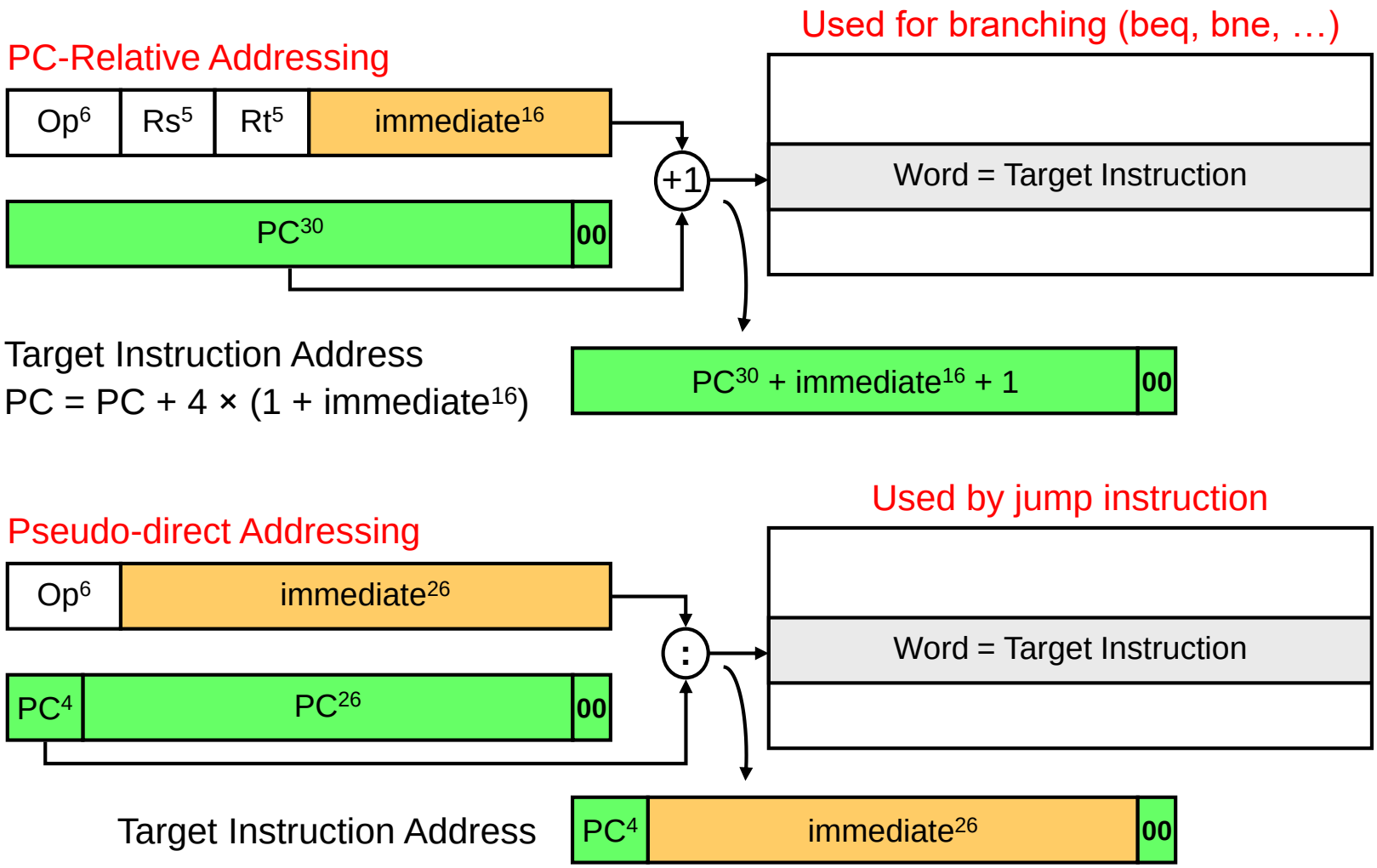
- Instruction Set Architecture
- Overview of the MIPS Processor
- R-Type Arithmetic, Logical, and Shift Instructions
- I-Type Format and Immediate Constants
- Jump and Branch Instructions
- Translating If Statements and Boolean Expressions
- Load and Store Instructions
- Translating Loops and Traversing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

Addressing Modes

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



Branch / Jump Addressing Modes



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×(1 + immediate^{16})
    - During assembly: immediate=(Target address – PC)/4, where PC contains address of next instruction

Jump and Branch Limits

- During execution, PC contains the address of current instruction (thus we add 1 to immediate[16]).
- Maximum branch limit is -2^{15} to $+2^{15}-1$ instructions.
- If immediate is positive => Forward Jump
- If immediate is negative => Backward Jump

❖ Example

```
      0
Again:4
      8   beq $s1,$s2 Next
      12
      16  bne $s1,$zero Again
Next: 20
```

Forward Jump
During assembly:
Immediate=(Next-PC)/4=(20-12)/4=2
During execution:
PC=PC+4*(immediate+1)=8+4*(3)=20

Backward Jump
During assembly:
Immediate=(Again-PC)/4=(4-20)/4=-4
During execution:
PC=PC+4*(immediate+1)=16+4*(-3)=4

Summary of RISC Design

- All instructions are typically of one size
- Few instruction formats
- All operations on data are register to register
  - Operands are read from registers
  - Result is stored in a register
- General purpose integer and floating point registers
  - Typically, 32 integer and 32 floating-point registers
- Memory access only via load and store instructions
  - Load and store: bytes, half words, words, and double words
- Few simple addressing modes

Four Design Principles

1. Simplicity favors regularity
   - ✧ 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
   - ✧ Fixed-size instructions compromise the size of constants