University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

MIPS ISA

References:

1) Textbook
2) Mark Redekopp's slide series

Shahin Nazarian

Spring 2010

Components of an ISA

1. Data and Address Size
   - 8-, 16-, 32-, 64-bit
2. Which instructions does the processor support
   - SUBtract instruc. vs. NEGate + ADD instrucs.
3. Registers accessible to the instructions
   - Faster than accessing data from memory
4. Addressing Modes
   - How instructions can specify location of data operands
5. Length and format of instructions
   - How is the operation and operands represented with 1's and 0's

MIPS ISA

- **MIPS** (originally an acronym for Microprocessor without Interlocked Pipeline Stages) is a RISC (Reduced Instruction Set Computing) ISA developed by MIPS Computer Systems (now **MIPS Technologies**)
- The early MIPS architectures were 32-bit, and later versions were 64-bit
- Computer architecture courses in universities and technical schools often study the MIPS architecture. The architecture greatly influenced later RISC architectures
- MIPS implementations are currently primarily used in many **embedded systems** such as the Series2 TiVo, Windows CE devices, Cisco routers, residential gateways, and video game consoles like the Nintendo 64 and Sony PlayStation, PlayStation 2, and PlayStation Portable handheld system. Until late 2006, they were also used in many of **SGI** (Silicon Graphics, Inc.)'s computer products. MIPS implementations were also used by **Digital Equipment Corporation**, **NEC**, **Pyramid Technology**, **Siemens**, **Nixdorf**, **Tandem Comput**ers and others during the late 1980s and 1990s

MIPS ISA we study here!

- RISC Style
- 32-bit internal / 32-bit external data size
  - Registers and ALU are 32-bits wide
  - Memory bus is logically 32-bits wide (though may be physically wider)
- Registers
  - 32 General Purpose Registers (GPR's)
    - For integer and address values
    - A few are used for specific tasks/values
  - 32 Floating point registers
- Fixed size instructions
  - All instructions encoded as a single 32-bit word
  - Three operand instruction format (dest, src1, src2)
  - Load/store architecture (all data operands must be in registers and thus loaded from and stored to memory explicitly)

MIPS Data Sizes

Integer

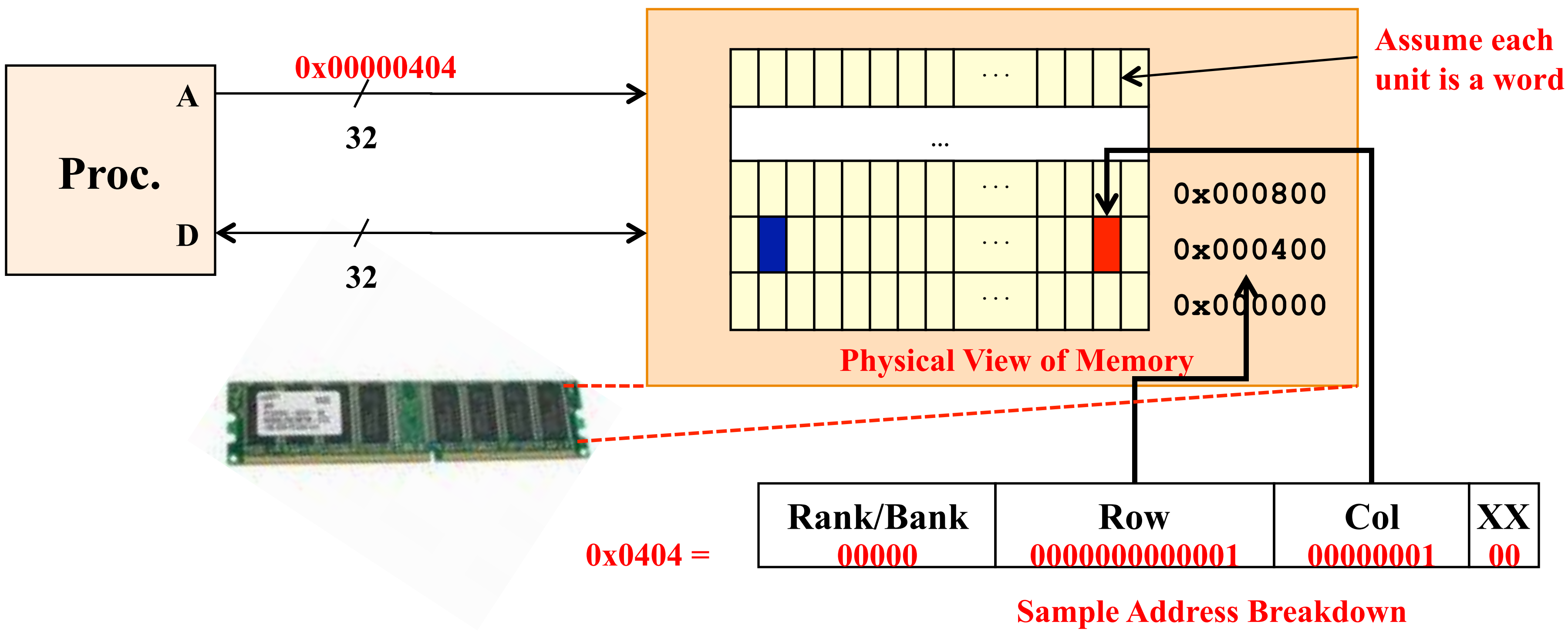
- 3 Sizes Defined
  - Byte (B)
    - 8-bits
  - Half(word) (H)
    - 16-bits = 2 bytes
  - Word (W)
    - 32-bits = 4 bytes

Floating Point

- 3 Sizes Defined
  - Single (S)
    - 32-bits = 4 bytes
  - Double (D)
    - 64-bits = 8 bytes
    - (For a 32-bit data bus, a double would be accessed from memory in 2 reads)

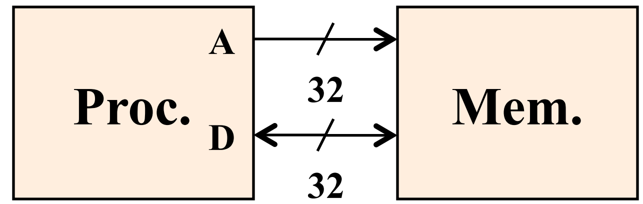
Physical Memory Organization

- Physical view of memory as large 2-D array of bytes (8K rows by 1KB columns) per chip (and several chips)
- Address is broken into fields of bits that are used to identify where in the array the desired 32-bit word is
  - Processor always accesses memory chunks the size of the data bus, selecting only the desired bytes as specified by the instruction



MIPS Memory Organization

- We can logically picture memory in the units (sizes) that we actually access them
- Most processors are *byte-addressable*
  - Every byte (8-bits) has a unique address
  - 32-bit address bus => 4 GB address space
- However, 32-bit logical data bus allows us to access 4-bytes of data at a time
- Logical view of memory arranged in rows of 4-bytes
  - Still with separate addresses for each byte



...	
F8	0x000002
13	0x000001
5A	0x000000

Logical Byte-Oriented View of Mem.

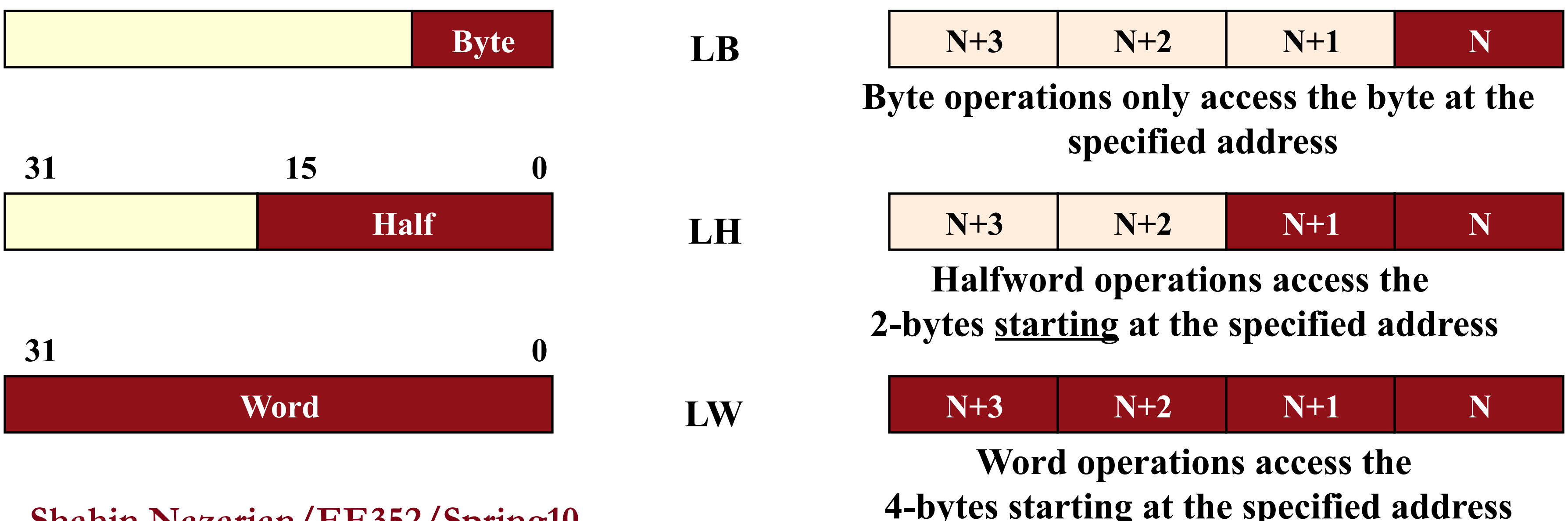
...				0x000008
8E	AD	33	29	0x000004
7C	F8	13	5A	0x000000

Logical Word-Oriented View

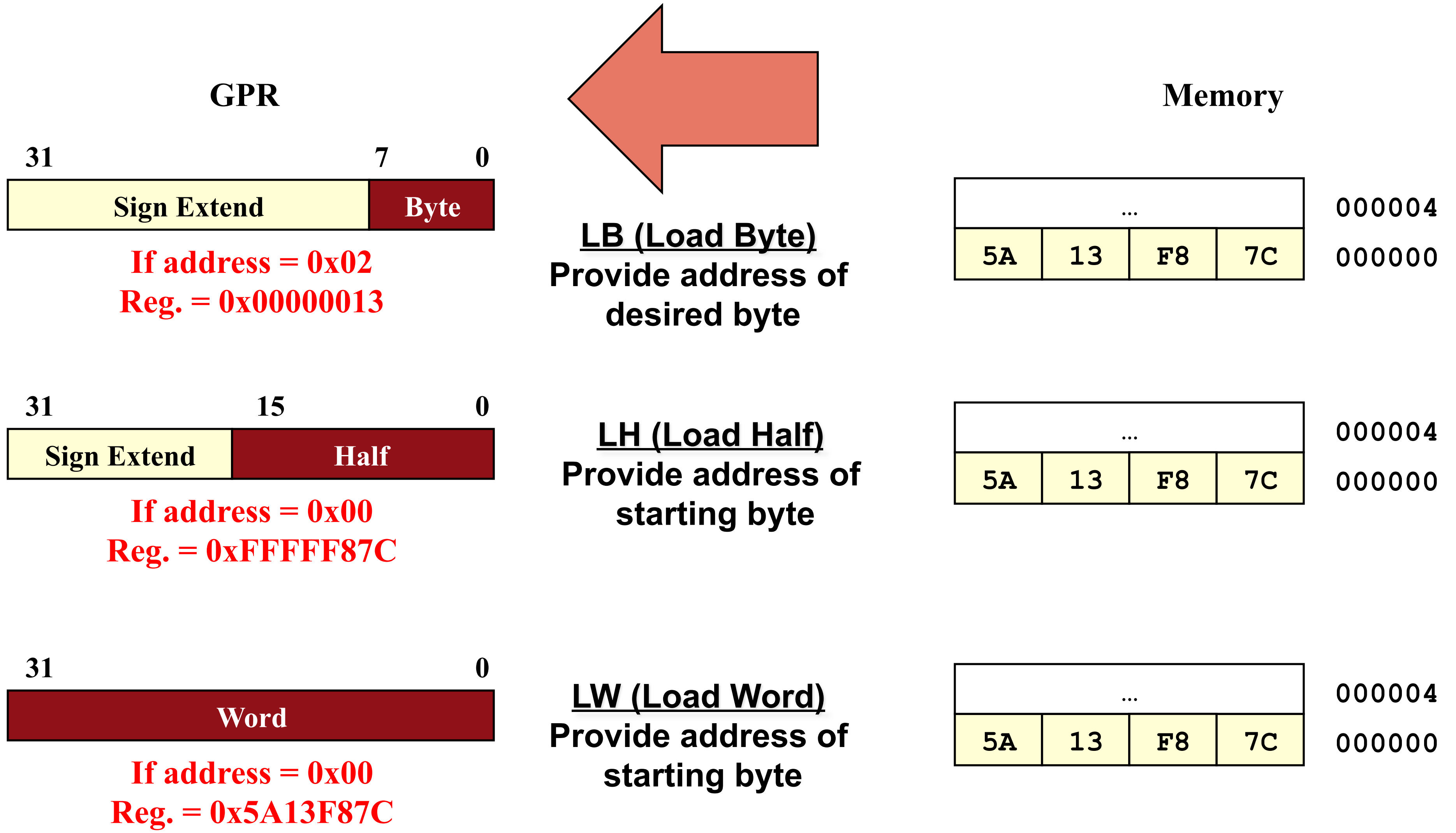
Memory & Data Size

- Little-endian memory can be thought of as right justified
- Always provide the LS-Byte address of the desired data
- Size is explicitly defined by the instruction used
- Memory Access Rules
  - Halfword or Word access **must** start on an address that is a multiple of that data size (i.e. half = multiple of 2, word = multiple of 4)

(Assume start address = N)



Memory Read Instructions (Signed)



Memory Read Instructions (Unsigned)

GPR

Memory

31		7	0
Zero Extend		Byte	

If address = 0x01
Reg. = 0x000000F8

LBU (Load Byte)
Provide address of desired byte

...				000004
5A	13	F8	7C	000000

31	15	0
Zero Extend	Half	

If address = 0x00
Reg. = 0x0000F87C

LHU (Load Half)
Provide address of starting byte

...				000004
5A	13	F8	7C	000000

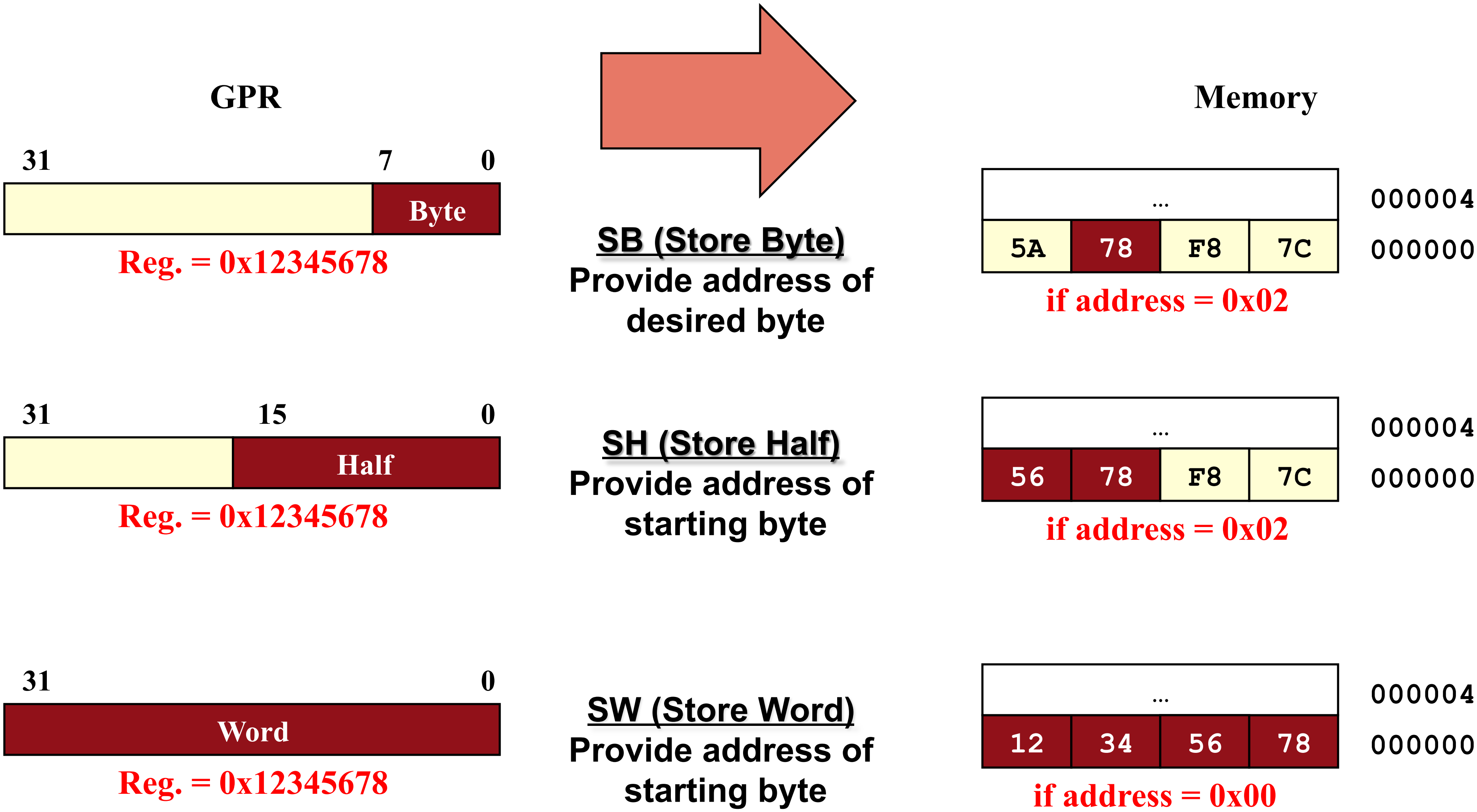
31	0
Word	

If address = 0x00
Reg. = 0x5A13F87C

LW (Load Word)
Provide address of starting byte

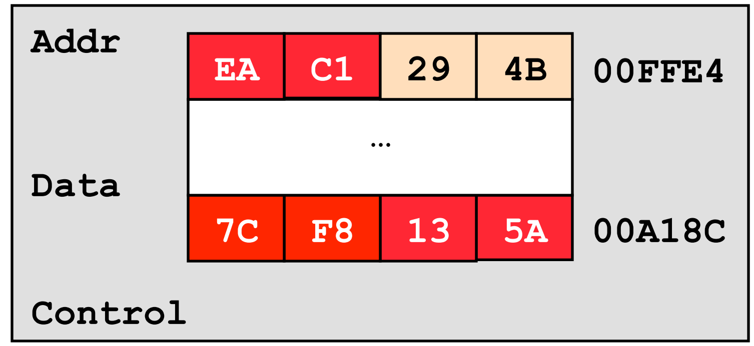
...				000004
5A	13	F8	7C	000000

Memory Write Instructions

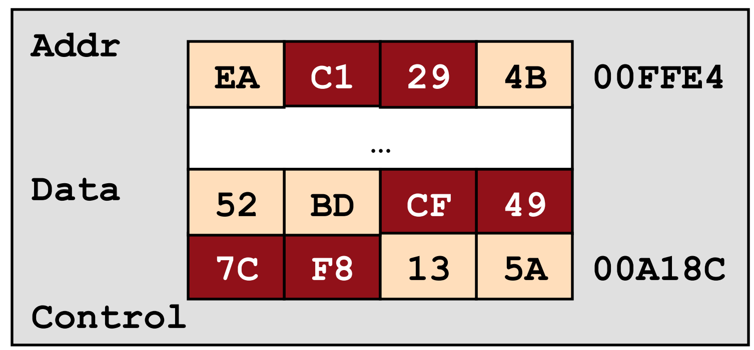


MIPS Memory Alignment Limitations

- Bytes can start at any address
- Halfwords must start on an even address
- Words must start on an address that is a multiple of 4
- Examples:
  - Word @ A18C – valid (multiple of 4)
  - Halfword @ FFE6 – valid (even)
  - Word @ A18E – invalid (non-multiple of 4)
  - Halfword @ FFE5 – invalid (odd)



Valid Accesses



Invalid Accesses

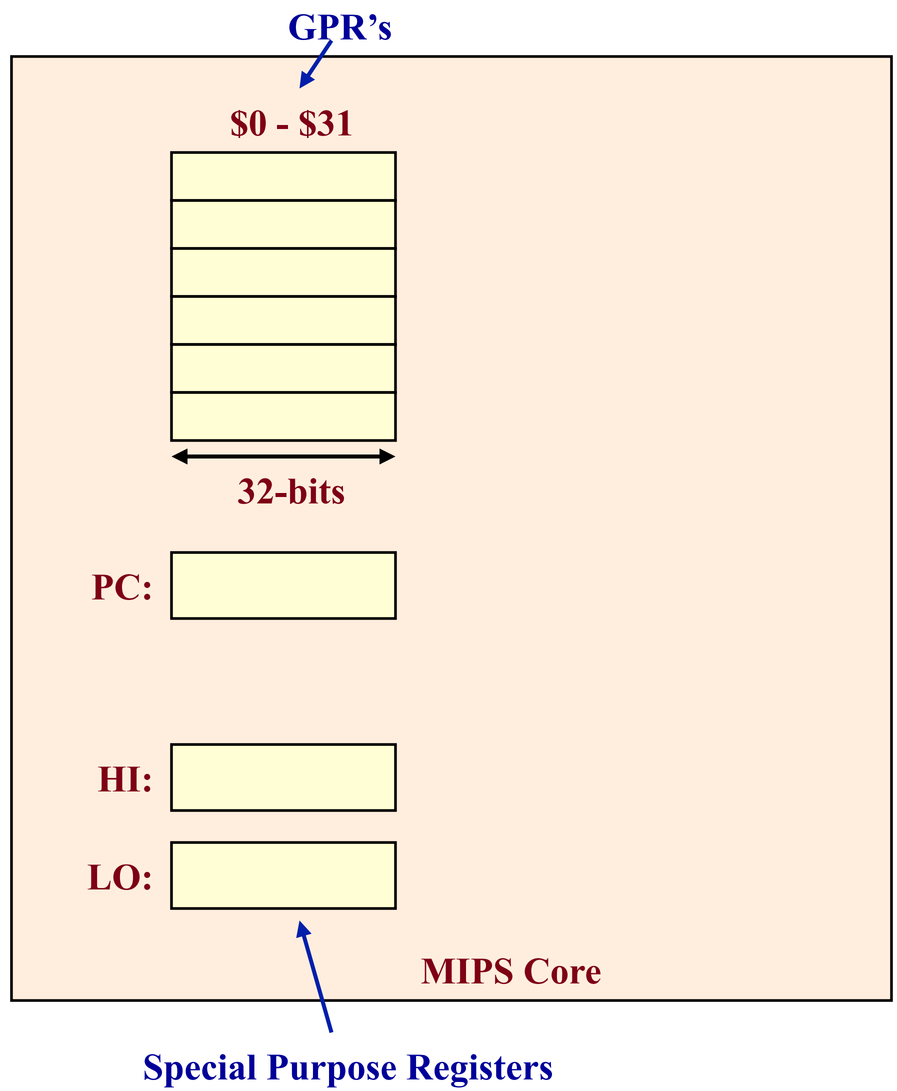
MIPS GPR's

$0-$31

Assembler Name	Reg. Number	Description
$zero	$0	Constant 0 value
$at	$1	Assembler temporary
$v0-$v1	$2-$3	Procedure return values or expression evaluation
$a0-$a3	$4-$7	Arguments/parameters
$t0-$t7	$8-$15	Temporaries
$s0-$s7	$16-$23	Saved Temporaries
$t8-$t9	$24-$25	Temporaries
$k0-$k1	$26-$27	Reserved for OS kernel
$gp	$28	Global Pointer (Global and static variables/data)
$sp	$29	Stack Pointer
$fp	$30	Frame Pointer
$ra	$31	Return address for current procedure

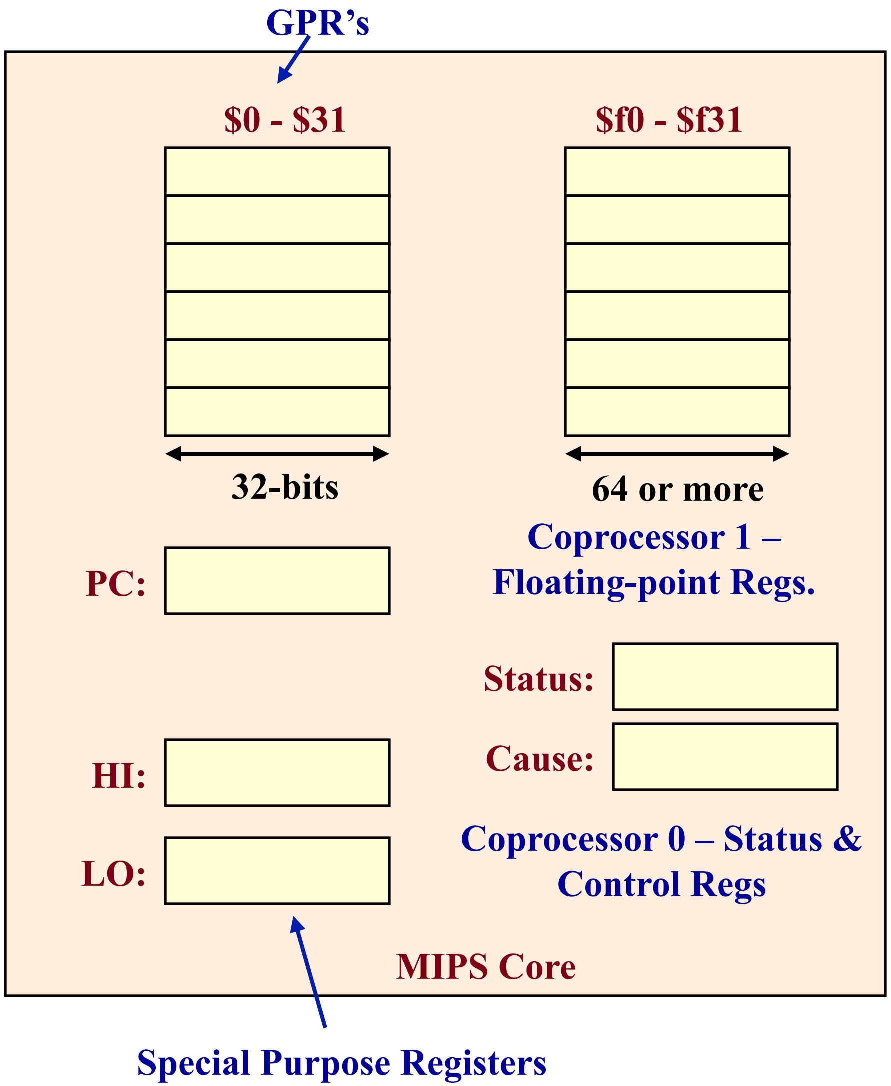
MIPS Programmer-Visible Registers

- General Purpose Registers (GPR's)
  - Hold data operands or addresses (pointers) to data stored in memory
- Special Purpose Registers
  - PC: Program Counter (32-bits)
    - Holds the address of the next instruction to be fetched from memory & executed
  - HI: Hi-Half Reg. (32-bits)
    - For MUL, holds 32 MSB's of result. For DIV, holds 32-bit remainder
  - LO: Lo-Half Reg. (32-bits)
    - For MUL, holds 32 LSB's of result. For DIV, holds 32-bit quotient



MIPS Programmer-Visible Registers

- Coprocessor 0 Registers
  - Status Register
    - Holds various control bits for processor modes, handling interrupts, etc.
  - Cause Register
    - Holds information about exception (error) conditions
- Coprocessor 1 Registers
  - Floating-point registers
  - Can be used for single or double-precision (i.e. at least 64-bits wides)



General Instruction Format Issues

- 3 Operand Format
  - Example:

  ADD $t0, $t1, $t2 ($t0 = $t1 + $t2)
- Fixed Size instructions = All instructions are a 32-bit (long)word
  - Bits describing the opcode, source/dest. registers and immediates/absolute addresses/ displacements must fit in a single 32-bit value

Other Instruction Formats

- Different instruction sets specify these differently
  - 3 operand instruction set (MIPS, PPC)
    - Similar to example on previous page
    - Format: ADD DST, SRC1, SRC2 (DST = SRC1 + SRC2)
  - 2 operand instructions (Intel / Motorola 68K)
    - Second operand doubles as source and destination
    - Format: ADD SRC1, S2/D (S2/D = SRC1 + S2/D)
  - 1 operand instructions (Old Intel FP, Low-End Embedded)
    - Implicit operand to every instruction usually known as the Accumulator (or ACC) register
    - Format: ADD SRC1 (ACC = ACC + SRC1)

General Instruction Format Issues

- Consider the pros and cons of each format when performing the set of operations
  - $F = X + Y - Z$
  - $G = A + B$
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruction size
- Modern high performance processors use 2- & 3-operand formats

Single-Operand	Two-Operand	Three-Operand

LOAD X
ADD Y
SUB Z
STORE F
LOAD A
ADD B
STORE G
	MOVE F,X
	ADD F,Y
	SUB F,Z
	MOVE G,A
	ADD G,B
		ADD F,X,Y
		SUB F,F,Z
		ADD G,A,B

(+) Smaller size to encode each instruction (-) Higher instruction count to load and store ACC value	Compromise of two extremes	(+) More natural program style (+) Smaller instruction count (-) Larger size to encode each instruction

ALU (R-Type) Instructions

Memory Access, Branch, & Immediate (I-Type) Instructions

MIPS INSTRUCTIONS

Shorthand Notation

- Shorthand notation for describing processor / memory (assembly) operations
- R[*n*] = value of GPR *n*
- F[*n*] = value of FP reg. *n*
- M[*n*] = value of memory at address *n*
- Examples
  - R[1] = R[2] + R[3]
  - R[5] = M[40 + R[1]]
    - Use the value of R1 + 40 as the address to read memory & place data in R5

MIPS Instructions & Addressing Modes

- Types of instructions
  - R-Type
    - Instructions with 3 register operands
    - Arithmetic and logic instructions
  - I-Type
    - Instructions with an immediate (constant) value
    - Memory load and store instructions
    - Arithmetic and logical immediate instructions
    - Branch instructions
- Addressing Modes: Methods for specifying the location of operands

Mode	Syntax	RTL	Description
Reg. Direct	$n	R[n]	Contents of given reg.
Reg. Indirect w/ Offset	off($n)	M[off+R[n]]	Contents of memory at address R[n] + offset
Immediate	Const	const	Operand is the constant

R-Type Instructions

- Format

6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
opcode	rs (src1)	rt (src2)	rd (dest)	shamt	function

  - rs, rt, rd are 5-bit fields for register numbers
  - shamt = shift amount and is used for shift instructions indicating # of places to shift bits
  - op (traditionally called the opcode) specifies the basic operation of the instruction
  - func (often called the function code) selects the specific variant of the operation in the op field
- Example:
  - ADD $5, $24, $17

opcode	rs	rt	rd	shamt	func
000000	11000	10001	00101	00000	100000
Arith. Inst.	$24	$17	$5	unused	ADD

R-Type Arithmetic/Logic Instructions

<table>
<tr><th>C operator</th><th>Assembly</th><th>Notes</th></tr>
<tr><td>+</td><td>ADD Rd, Rs, Rt</td><td></td></tr>
<tr><td>-</td><td>SUB Rd, Rs, Rt</td><td>Order: R[s] – R[t]. SUBU for unsigned</td></tr>
<tr><td>*</td><td>MULT Rs, Rt
MULTU Rs, Rt</td><td>Result in HI/LO. Use mfhi and mflo instruction to move results</td></tr>
<tr><td>*</td><td>MUL Rd, Rs, Rt</td><td>If multiply won’t overflow 32-bit result</td></tr>
<tr><td>/</td><td>DIV Rs, Rt
DIVU Rs, Rt</td><td>R[s] / R[t].
Remainder in HI, quotient in LO</td></tr>
<tr><td>&</td><td>AND Rd, Rs, Rt</td><td></td></tr>
<tr><td>|</td><td>OR Rd, Rs, Rt</td><td></td></tr>
<tr><td>^</td><td>XOR Rd, Rs, Rt</td><td></td></tr>
<tr><td>~(|)</td><td>NOR Rd, Rs, Rt</td><td>Can be used for bitwise-NOT (~)</td></tr>
<tr><td><<</td><td>SLL Rd, Rs, shamt
SLLV Rd, Rs, Rt</td><td>Shifts R[s] left by shamt (shift amount) or R [t] bits</td></tr>
<tr><td>>> (signed)</td><td>SRA Rd, Rs, shamt
SRAV Rd, Rs, Rt</td><td>Shifts R[s] right by shamt or R[t] bits replicating sign bit to maintain sign</td></tr>
<tr><td>>> (unsigned)</td><td>SRL Rd, Rs, shamt
SRLV Rd, Rs, Rt</td><td>Shifts R[s] left by shamt or R[t] bits shifting in 0’s</td></tr>
<tr><td><, >, <=, >=</td><td>SLT Rd, Rs, Rt
SLTU Rd, Rs, Rt</td><td>Order: R[s] – R[t]. Sets R[d]=1 if R[s] < R[t], 0 otherwise</td></tr>
</table>

Logical Operations

- Logic operations are usually performed on a pair of bits

X1	X2	AND
0	0	0
0	1	0
1	0	0
1	1	1

X1	X2	OR
0	0	0
0	1	1
1	0	1
1	1	1

X1	X2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

X1	NOT
0	1
1	0

AND – Output is true if both inputs are true

0 AND x = 0
1 AND x = x
x AND x = x

OR – Output is true if any input is true

0 OR x = x
1 OR x = 1
x OR x = x

XOR – Output is true if exactly one input is true

0 XOR x = x
1 XOR x = NOT x
x XOR x = 0

NOT – Output is inverse of input

Logical Operations

- **Logic operations on numbers means performing the operation on each pair of bits**

Initial Conditions: R[1]= 0xF0, R[2] = 0x3C

1 AND $2,$1,$2	→	0xF0	→	1111 0000
		AND 0x3C		AND 0011 1100
R[2] = 0x30	←	0x30	←	0011 0000
2 OR $2,$1,$2	→	$F0	→	1111 0000
		OR $3C		OR 0011 1100
R[2] = 0xFC	←	$FC	←	1111 1100
3 XOR $2,$1,$2	→	0xF0	→	1111 0000
		XOR 0x3C		XOR 0011 1100
R[2] = 0xCC	←	0xCC	←	1100 1100

Logical Operations

- **Logic operations on numbers means performing the operation on each pair of bits**

Initial Conditions: R[1]= 0xF0, R[2] = 0x3C

4 NOR \$2,\$1,\$2	→	0xF0	→	1111 0000
		NOR 0x3C		NOR 0011 1100
R[2] = 0x03	←	0x03	←	0000 0011

Bitwise NOT operation can be performed by NOR'ing register with itself

NOR \$2,\$1,\$1	→	0xF0	→	1111 0000
		NOR 0xF0		NOR 1111 0000
R[2] = 0x0F	←	0x0F	←	0000 1111

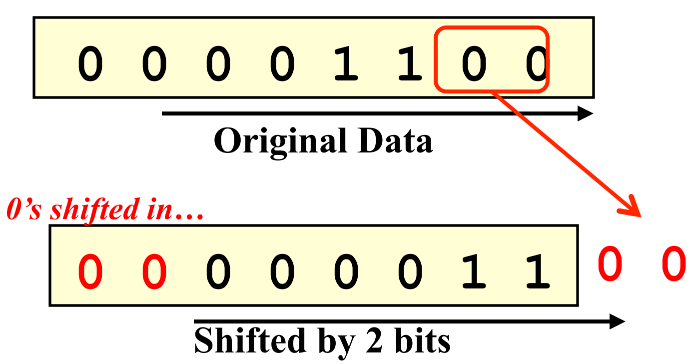
Logical Operations

- Logic operations are often used for "bit" fiddling
  - Change the value of 1-bit in a number w/o affecting other bits
  - C operators: & = AND, | = OR, ^ = XOR, ~ = NOT
- Examples (Assume an 8-bit variable, v)
  - Set the LSB to '0' w/o affecting other bits
    - v = v & 0xfe;
  - Check if the MSB = '1' regardless of other bit values
    - if(v & 0x80) { code }
  - Set the MSB to '1' w/o affecting other bits
    - v = v | 0x80;
  - Flip the LS 4-bits w/o affecting other bits
    - v = v ^ 0x0f;

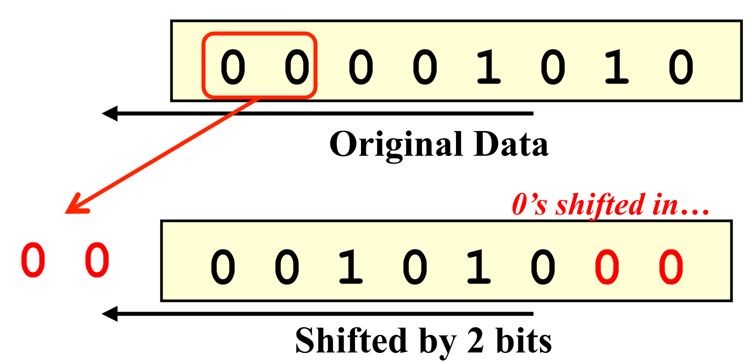
Shift Operations

- Shifts data bits either left or right
- Bits shifted out and dropped on one side
- Usually (but not always) 0's are shifted in on the other side
- Shifting is equivalent to multiplying or dividing by powers of 2
- 2 kinds of shifts
  - Logical shifts (used for unsigned numbers)
  - Arithmetic shifts (used for signed numbers)

Right Shift by 2 bits:

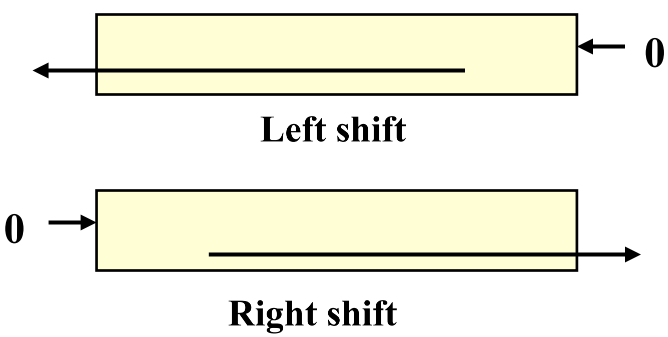


Left Shift by 2 bits:

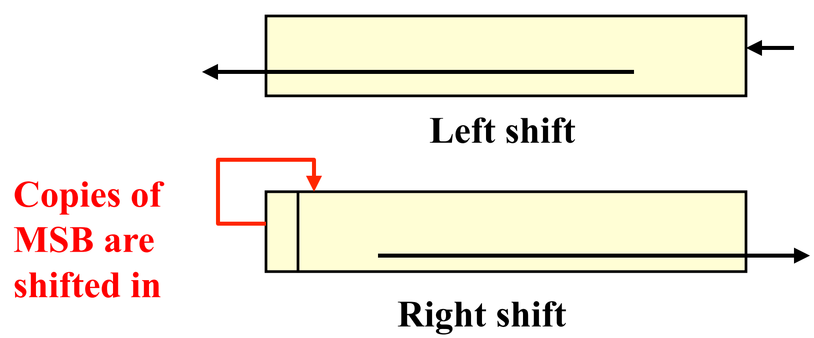


Logical Shift vs. Arithmetic Shift

- Logical Shift
  - Use for unsigned or non-numeric data
  - Will always shift in 0's whether it be a left or right shift

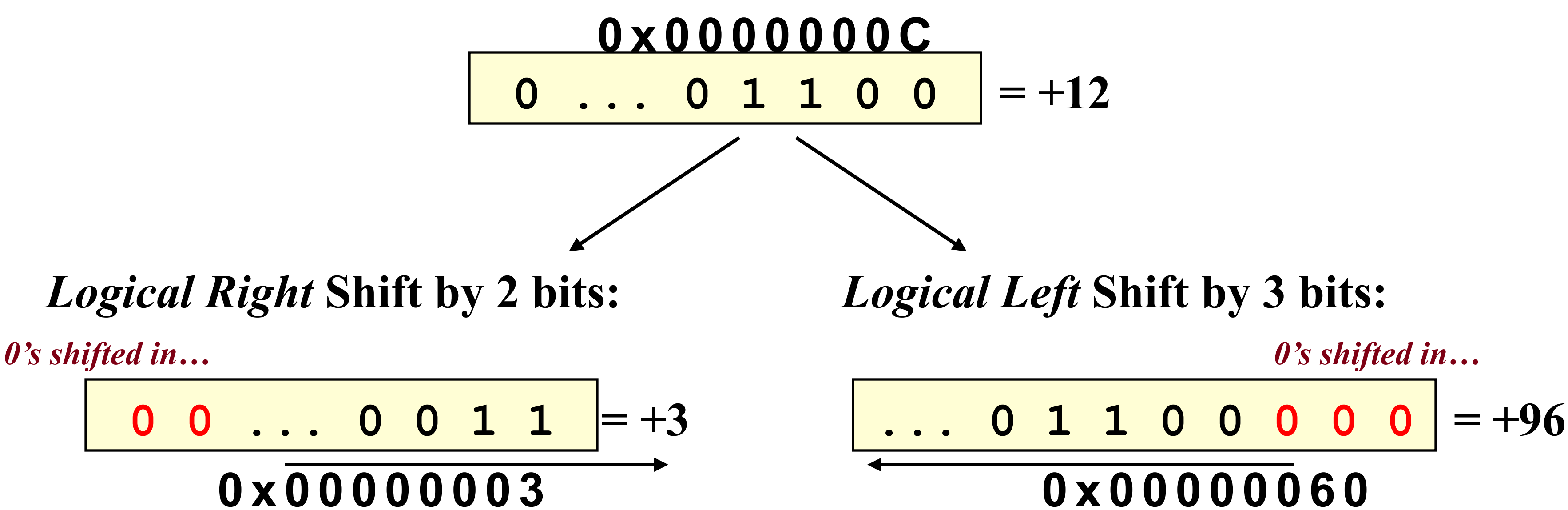


- Arithmetic Shift
  - Use for signed data
  - Left shift will shift in 0's
  - Right shift will sign extend (replicate the sign bit) rather than shift in 0's
    - If negative number …stays negative by shifting in 1's
    - If positive…stays positive by shifting in 0's



Logical Shift

- 0's shifted in
- Only use for operations on *unsigned* data
  - Right shift by n-bits = Dividing by 2^n
  - Left shift by n-bits = Multiplying by 2^n



Arithmetic Shift

- Use for operations on *signed* data
- Arithmetic Right Shift – replicate MSB
  - Right shift by n-bits = Dividing by 2^n
- Arithmetic Left Shift – shifts in 0's
  - Left shift by n-bits = Multiplying by 2^n

0xFFFFFFFC

`1 1 ... 1 1 0 0` = -4

Arithmetic Right **Shift by 2 bits:**

MSB replicated and shifted in…

`1 1 1 ... 1 1 1` = -1

0xFFFFFFFF

Notice if we shifted in 0's (like a logical right shift) our result would be a positive number and the division wouldn't work

Arithmetic Left **Shift by 2 bits:**

0's shifted in…

`1 ... 1 0 0 0 0` = -16

0xFFFFFFF0

Notice there is no difference between an arithmetic and logical left shift. We always shift in 0's

Shift Instructions

- Logical Shifts (SLL, SRL) and Arithmetic (SRA)
- Format:
  - Sxx rd, rt, shamt
  - SxxV rd, rt, rs
- Notes:
  - shamt limited to a 5-bit value (0-31)
  - SxxV shifts data in rt by number of places specified in rs
- Examples
  - SRA $5, $12, 7

opcode	rs	rt	rd	shamt	func
000000	00000	10001	00101	00111	000011
Arith. Inst.	unused	$12	$5	7	SRA

  - SRAV $5, $12, $20

000000	10100	10001	00101	00000	000111
Arith. Inst.	$20	$12	$5	unused	SRAV

I-Type Instructions

- Format

6-bits	5-bits	5-bits	16-bits
opcode	rs (src1)	rt (src/dst)	immediate

  - rs, rt are 5-bit fields for register numbers
  - immediate is a 16-bit constant
  - opcode identifies actual operation

- Example:
  - ADDI $5, $24,

opcode	rs	rt	immediate
001000	11000	00101	0000 0000 0000 0001
ADDI	$24	$5	20

  - LW $5, -8($3)

010111	00011	00101	1111 1111 1111 1000
LW	$3	$5	-8

Immediate Operands

- Most ALU instructions also have an immediate form to be used when one operand is a constant value
- Syntax: ADDI Rs, Rt, imm
  - Because immediates is limited to 16-bits, it must be extended to a full 32-bits when used the by the processor
  - Arithmetic instructions always **sign-extend** to a full 32-bits even for unsigned instructions (addiu)
  - Logical instructions always **zero-extend** to a full 32-bits
- Examples:
  - ADDI $4, $5, -1 // R[4] = R[5] + 0xFFFFFFFF
  - ORI $10, $14, -4 // R[10] = R[14] | 0x0000FFFC

Arithmetic	Logical
ADDI	ANDI
ADDIU	ORI
SLTI	XORI
SLTIU	

Note: SUBI is unnecessary since we can use ADDI with a negative immediate value

Load Format (LW)

- LW Rt, offset(Rs)
  - Rt = Destination register
  - offset(Rs) = Address of desired data
  - RTL: R[t] = M[offset + R[s]]
  - offset limited to 16-bit signed number
- Examples
  - LW $2, 0x40($3) // R[2] = 0xF8BE97CD
  - LW $2, 0xFFFC($4) // R[2] = 0x5A12C5B7

Register	Value
R[2]	old val.
R[3]	00002000
R[4]	0000204C

Address	Value
0x002040	F8BE97CD
0x002044	134982FE
0x002048	5A12C5B7

More LOAD Examples

- Examples
  - LB $2,0x45($3) // R[2] = 0xFFFFFF82
  - LH $2,-6($4) // R[2] = 0x00001349
  - LHU $2, -2($4) // R[2] = 0x0000F8BE

R[2]	old val.
R[3]	00002000
R[4]	0000204C

F8BE97CD	0x002048
134982FE	0x002044
5A12C5B7	0x002040

Store Format (SW)

- SW Rt, offset(Rs)
  - Rt = Source register
  - offset(Rs) = Address to store data
  - RTL: M[offset + R[s]] = R[t]
  - offset limited to 16-bit signed number
- Examples
  - SW $2, 0x40($3)
  - SW $2, 0xFFF8($4)

R[2]	123489AB
R[3]	00002000
R[4]	0000204C

0x002040	123489AB
0x002044	123489AB
0x002048	00000000

Loading an Immediate

- If immediate (constant) 16-bits or less
  - Use ORI or ADDI instruction with $0 register
  - Examples
    - ADDI $2, $0, 1 // R[2] = 0 + 1 = 1
    - ORI $2, $0, 0xF110 // R[2] = 0 | 0xF110 = 0xF110
- If immediate more than 16-bits
  - Immediate is limited to 16-bits so we must load constant with a 2 instruction sequence using the special **LUI** (Load Upper Immediate) instruction
  - To load $2 with 0x12345678
    - LUI $2,0x1234
    - ORI $2,$2,0x5678

R[2]	12340000	LUI
	OR 00005678	
R[2]	12345678	ORI

Translating HLL to Assembly

C operator	Assembly	Notes

int x,y,z;
…
x = y + z;
	LUI $8, 0x1000
	ORI $8, $8, 0x0004
	LW $9, 4($8)
	LW $10, 8($8)
	ADD $9,$9,$10
	SW $9, 0($8)
		Assume x @ 0x10000004
		& y @ 0x10000008
		& z @ 0x1000000C
		
		$9 is loaded with y
		$10 is loaded with z
		Add result if saved in where
		$8 is pointing at (x)

Pseudo-instructions

- "Macros" translated by the assembler to instructions actually supported by the HW
- Simplifies writing code in assembly
- Example – LI (Load-immediate) pseudo-instruction translated by assembler to 2 instruction sequence (LUI & ORI)

```
...
li    $2, 0x12345678
...
```

With pseudo-instruction

```
...
lui   $2, 0x1234
ori   $2, $2, 0x5678
...
```

After assembler…

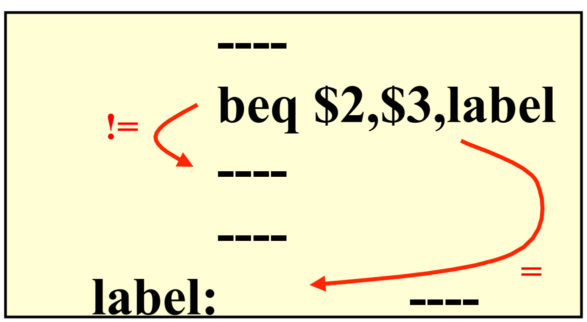
Pseudo-instructions

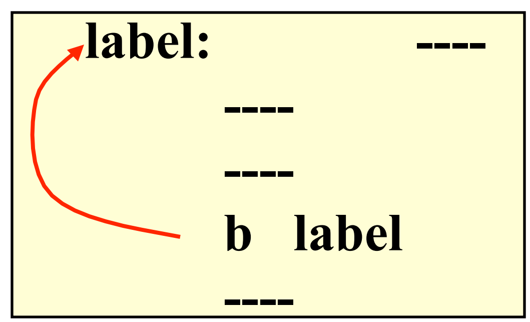
Pseudo-instruction	Actual Assembly
NOT Rd,Rs	NOR Rd,Rs,$0
NEG Rd,Rs	SUB Rd,$0,Rs
LI Rt, immed. # Load Immediate	LUI Rt, {immediate[31:16], 16'b0} ORI Rt, {16'b0, immediate[15:0]}
LA Rt, label # Load Address	LUI Rt, {immediate[31:16], 16'b0} ORI Rt, {16'b0, immediate[15:0]}
BLT Rs,Rt,Label	SLT $1,Rs,Rt BNE $1,$0,Label

Note: Pseudoinstructions are assembler-dependent

Branch Instructions

- Conditional Branches
  - Branches only if a particular condition is true
  - Fundamental Instrucs.: BEQ (if equal), BNE (not equal)
  - Syntax: `BNE/BEQ Rs, Rt, label`
    - Compares Rs, Rt and if EQ/NE, branch to label, else continue
- Unconditional Branches
  - Always branches to a new location in the code
  - Instruction: `BEQ $0,$0,label`
  - Pseudo-instruction: `B label`





Two-Operand Compare & Branches

- Two-operand comparison is accomplished using the SLT/SLTI/SLTU (Set If Less-than) instruction
  - Syntax: SLT Rd,Rs,Rt or SLT Rd,Rs,imm
    - If Rs < Rt then Rd = 1, else Rd = 0
  - Use appropriate BNE/BEQ instruction to infer relationship

Branch if…	SLT	BNE/BEQ
\$2 < \$3	SLT \$1,\$2,\$3	BNE \$1,\$0,label
\$2 ≤ \$3	SLT \$1,\$3,\$2	BEQ \$1,\$0,label
\$2 > \$3	SLT \$1,\$3,\$2	BNE \$1,\$0,label
\$2 ≥ \$3	SLT \$1,\$2,\$3	BEQ \$1,\$0,label

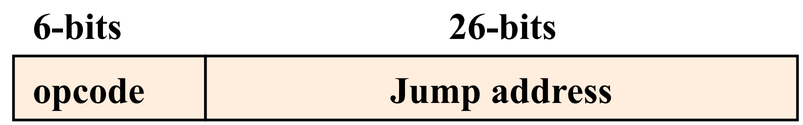
Translating HLL to Assembly

C operator	Assembly

int dat[4],x=0;
for(i=0;i<4;i++)
x += dat[i];
	DAT: .space 16
	X: .long 0
	LA $8, DAT
	ADDI $9,$0,4
	ADD $10,$0,$0
	LP: LW $11,0($8)
	ADD $10,$10,$11
	ADDI $8,$8,4
	ADDI $9,$9,-1
	BNE $9,$0,LP
	LA $8,X
	SW $10,0($8)

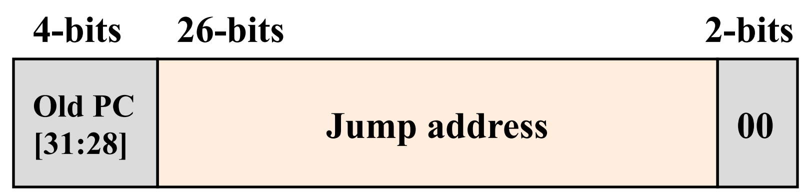
Jump Instructions

- Jumps provide method of branching beyond range of 16-bit displacement
- Syntax: J *label/address*
  - Operation: PC = address
  - Address is appended with two 0's just like branch displacement yielding a 28-bit address with upper 4-bits of PC unaffected
- New instruction format: J-Type



Sample Jump instruction

PC before execution of Jump



New PC after execution of Jump

Jump Register

- 'jr' instruction can be used if a full 32-bit jump is needed or variable jump address is needed
- Syntax: JR rs
  - Operation: PC = R[s]
  - R-Type machine code format
- Usage:
  - Can load rs with an immediate address
  - Can calculate rs for a variable jump (class member functions, switch statements, etc.)