

EE 457 Unit 3

Instruction Sets

With Focus on our Case Study: MIPS

INSTRUCTION SET OVERVIEW

Instruction Sets

- Defines the software interface of the processor and memory system
- Instruction set is the vocabulary the HW can understand and the SW is composed with
- Most assembly/machine instructions fall into one of three categories
 - _____
 - _____
 - _____

Instruction Set Architecture (ISA)

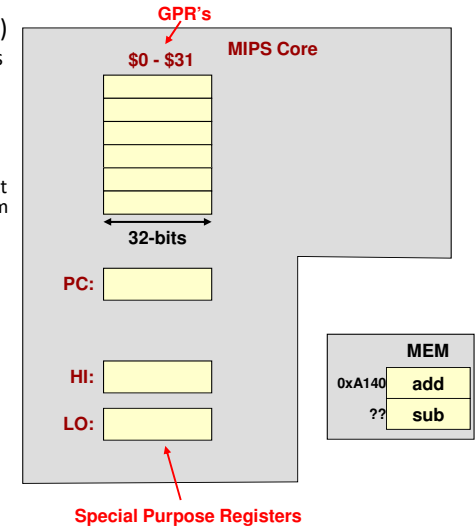
- 2 approaches
 - CISC = Complex instruction set computer
 - Large, rich vocabulary
 - More work per instruction, slower clock cycle
 - RISC = Reduced instruction set computer
 - Small, basic, but *sufficient* vocabulary
 - Less work per instruction, faster clock cycle
 - Usually a simple and small set of instructions with regular format facilitates building faster processors

MIPS ISA

- 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)

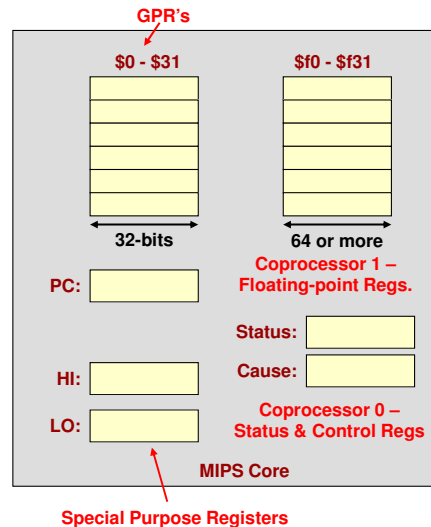
MIPS Programmer-Visible Registers

- General Purpose Registers (GPR's)
 - Hold data operands or addresses (pointers) to data stored in memory
- Special Purpose Registers
 - PC: _____ (32-bits)
 - Holds the _____ of the next 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 wide)



MIPS GPR's

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

General Instruction Format Issues

- Instructions must specify three things:
 - _____
 - _____
 - _____
- Example: ADD \$3, \$1, \$2 (\$3 = \$1 + \$2)
- Binary (machine-code) representation broken into fields of bits for each part

OpCode	Src. 1	Src. 2	Dest.	Shift Amount	Function
000000	00001	00010	00011	00000	100000
Arith.	\$1	\$2	\$3	Unused	Add

Historical Instruction Format Options

- Different instruction sets specify these differently
 - 3 operand instruction set (MIPS, PPC)
 - Usually all 3 operands in registers
 - 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 (Low-End Embedded, Java Virtual Machine)
 - Implicit operand to every instruction usually known as the _____ register
 - Format: ADD SRC1 (ACC = ACC + SRC1)
 - 0 operand instructions / _____ architecture
 - Push operands on a stack: PUSH X, PUSH Y
 - ALU operation: ADD (Implicitly adds top two items on stack: X + Y & replaces them with the sum)

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 instruc. size
- Modern, high performance processors use 2- and 3-operand formats

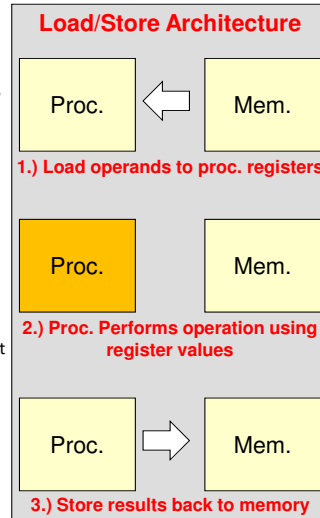
Stack Arch.	Single-Operand	Two-Operand	Three-Operand
	LOAD X	MOVE F,X ADD F,Y SUB F,Z MOVE G,A ADD G,B	ADD F,X,Y SUB F,F,Z ADD G,A,B
(+) Smaller size to encode each instruction			(+) More natural program style (+) Smaller instruction count

Addressing Modes

- Addressing modes refers to how an instruction specifies _____ the operands are
 - Can be in a _____, _____, or in the machine code of the instruction (immediate value)
- MIPS: All data operands for arithmetic instructions must be in a register
- But what about something like: \$8 = \$8 + A[i]
 - Intel instructions would allow: ADD \$8,A[i]
 - A[i] is in memory
 - MIPS require a _____ to read data from memory into a register
 - _____
 - ADD \$8,\$8,\$9

Operand Addressing

- Load/Store architecture
 - Load operands from memory into a register
 - Perform operations on registers and put results back into other registers
 - Store results back to memory
 - Because ALU instructions only access registers, the CPU design can be simpler and thus faster
- Most modern processors follow this approach
- Older designs
 - Register/Memory Architecture (Intel)
 - 1 operand of a ALU instruc. can be in a reg. or mem. but the other must be in a register
 - Memory/Memory Architecture (DEC VAX)
 - Operands of ALU instruc. can be in any combination of memory or registers
 - ADD addrDst, addrSrc1, addrSrc2



Load/Store Addressing

- When we load or store from/to memory how do we specify the address to use? Do we need sophisticated/exotic address modes (auto-increment, base+scaled index?)
- Option 1: Direct Addressing
 - Constant address: LW \$8, 0xA140
 - Insufficient!
 - Would have to translate to:
 - LW \$8, 0xA140
 - _____
 - _____

```
i = 0;
While(i < MAX)
x = x + A[i++];
```

	MEM
A[0] @ 0xA140	00
A[1] @ 0xA144	00
A[2] @ 0xA148	00
A[3] @ 0xA14C	00

Load/Store Addressing

- Option 2: _____
 - Put address in a register: \$9 = 0xA140
 - LW uses variable address in reg.: LW \$8, _____
 - Increment address via normal ADD instruc. (ADD \$9,\$9, _____)
 - Sufficient!
- Option 3: _____
 - Sums a constant offset with variable address in register
 - Put address in a register: \$9 = 0xA140
 - LW uses variable address in reg.: LW \$8, 0 (\$9) [_____]
 - LW uses variable address in reg.: LW \$8, 4 (\$9) [_____]
 - Sufficient!

```
i = 0;
While(i < MAX)
x = x + A[i++];
```

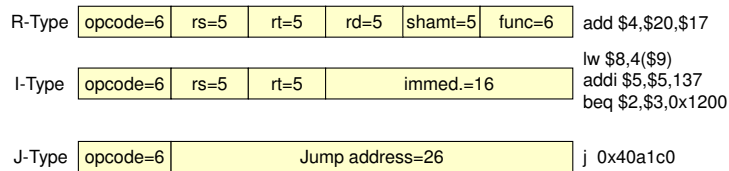
	MEM
A[0] @ 0xA140	00
A[1] @ 0xA144	00
A[2] @ 0xA148	00
A[3] @ 0xA14C	00

Immediate Addressing

- Suppose you want to increment a variable (register)
 - \$8 = \$8 + 1
 - Where do we get the 1 from?
- Could have compiler/loader _____ and then load it from memory _____
- Constant usage is very common, so instruction sets usually support a constant to be directly placed _____
- Known as immediate value because it is immediately available with the instruction machine code itself
- Example: ADDI \$8,\$8,1

MIPS Instruction Format

- CISC and other older architectures use a variable size instruction to match the varying operand specifications (memory addresses, etc.)
 - 1 to 8 bytes
- MIPS uses a FIXED-length instruction as do most RISC-style instruction sets
 - Every instruction is 32-bits (4-bytes)
 - One format (field breakdown) is not possible to support all the different instructions
 - MIPS supports 3 instruction formats: R-Type, I-Type, J-Type

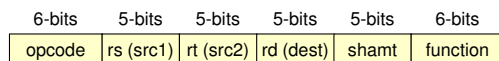


ALU (R-Type) Instructions
Memory Access, Branch, & Immediate (I-Type) Instructions

MIPS INSTRUCTIONS

R-Type Instructions

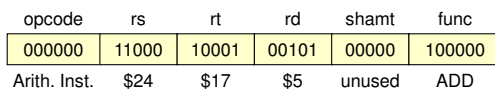
- Format



- 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
- opcode and func identify actual operation

- Example:

– ADD \$5, \$24, \$17



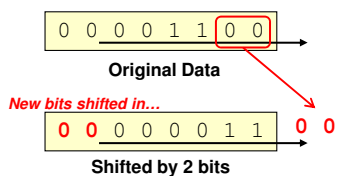
R-Type Arithmetic/Logic Instructions

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

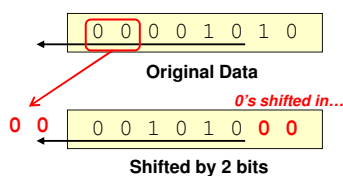
Shift Operations

- 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:

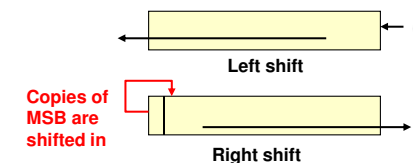
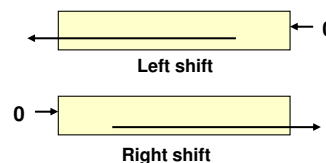


Left Shift by 2 bits:



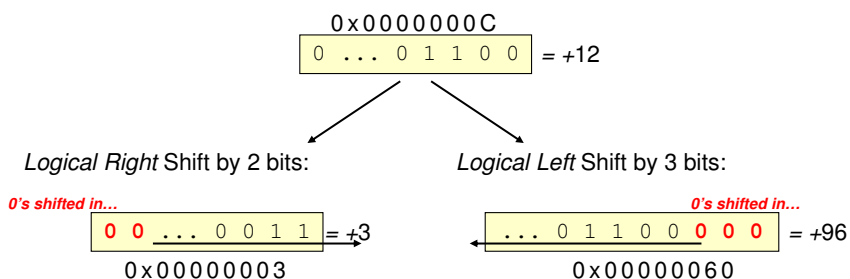
Logical Shift vs. Arithmetic Shift

- 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



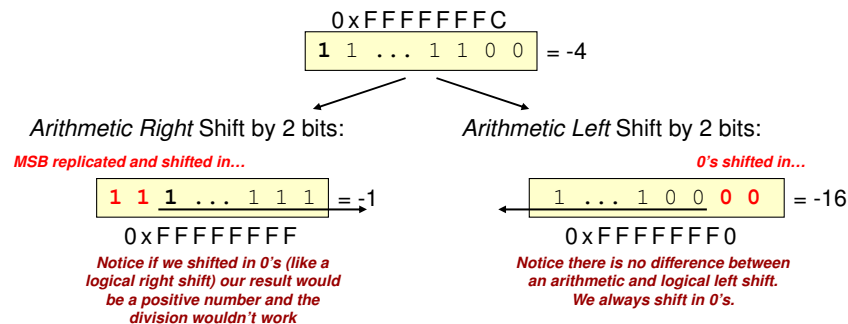
Logical Shift

- 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



Logical Shift Instructions

- SRL instruction – Shift Right Logical
- SLL instruction – Shift Left Logical
- Format:
 - SxL rd, rt, shamt
 - SxLV rd, rt, rs
- Notes:
 - shamt limited to a 5-bit value (0-31)
 - SxLV shifts data in rt by number of places specified in rs
- Examples
 - SRL \$5, \$12, 7
 - SLLV \$5, \$12, \$20

opcode	rs	rt	rd	shamt	func
000000	00000	10001	00101	00111	000010
Arith. Inst.	unused	\$12	\$5	7	SRL
000000	10100	10001	00101	00000	000100
Arith. Inst.	\$20	\$12	\$5	unused	SLLV

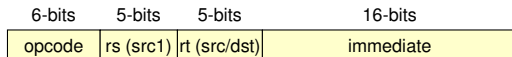
Arithmetic Shift Instructions

- SRA instruction – Shift Right Arithmetic
- Use SLL for arithmetic left shift
- Format:
 - SRA rd, rt, shamt
 - SRAV rd, rt, rs
- Notes:
 - shamt limited to a 5-bit value (0-31)
 - SRAV shifts data in rt by number of places specified in rs
- Examples
 - SRA \$5, \$12, 7
 - SRAV \$5, \$12, \$20

opcode	rs	rt	rd	shamt	func
000000	00000	10001	00101	00111	000011
Arith. Inst.	unused	\$12	\$5	7	SRA
000000	10100	10001	00101	00000	000111
Arith. Inst.	\$20	\$12	\$5	unused	SRAV

I-Type Instructions

- Format



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

- ADDI \$5, \$24, 1
- LW \$5, -8(\$3)

opcode	rs	rt	immediate
001000	11000	00101	0000 0000 0000 0001
ADDI	\$24	\$5	20
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 are limited to 16-bits, they must be extended to a full 32-bits when used by the processor
 - Arithmetic instructions always _____ to a full 32-bits even for unsigned instructions (addiu)
 - Logical instructions always _____ to a full 32-bits
- Examples:
 - ADDI \$4, \$5, -1 // R[4] = R[5] + _____
 - ORI \$10, \$14, -4 // R[10] = R[14] | _____

Arithmetic	Logical
ADDI	ANDI
ADDIU	ORI
SLTI	XORI
SLTIU	

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

Bytes, Half-words, Words, Double-words, yikes!

MEMORY ORGANIZATION

MIPS Data Sizes

Integer

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

Floating Point

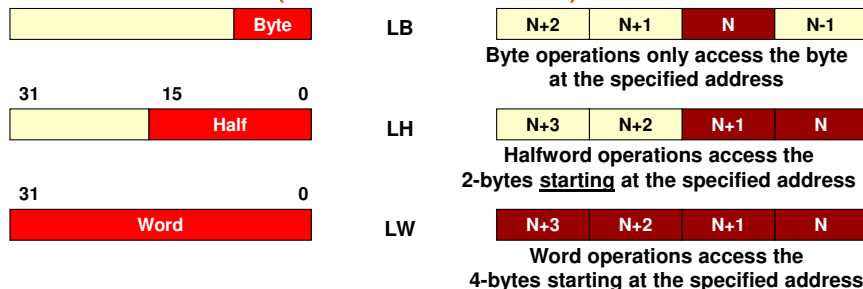
- 3 Sizes Defined
 - Single (S)
 - 32-bits = 4 bytes
 - Double (D)
 - 64-bits = 8 bytes
 - (For a 32-bit data bus, a double would be accessed from memory in 2 reads)

In MIPS, size matters to memory access instructions, but ALU instructions always perform operation on full 32-bit register values

Memory & Data Size

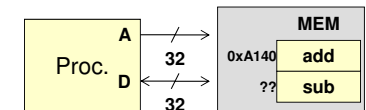
- Little-endian memory can be thought of as right justified
- Always provide the _____ of the desired data
- Size is explicitly defined by the instruction used
- Memory Access Rules
 - Halfword or Word access **must** start on an address that is a multiple of that data size (i.e. half = multiple of 2,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

(Assume start address = N)



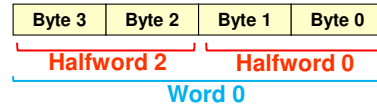
MIPS Memory Organization

- In a 32-bit system, addresses are 32-bits which means we can make $2^{32} = 4$ Giga (~billion) addresses
 - Recall: $2^{10} = 1K$ ($\sim 10^3$), $2^{20} = 1M$ ($\sim 10^6$), $2^{30} = 1G$ ($\sim 10^9$)
- If the current instruction is at address 0x0000A140, what address does the next instruction occupy (Recall instructions = 32-bits)
- But how much data does each address correspond to...
 - A byte (8-bits), a half-word (16-bits), or a word (32-bits)
- Most systems are byte-addressable meaning each byte receives a unique address
 - ASCII characters = 1-byte
 - Pixels in an image = 1-byte



Memory & Word Size

- If each byte has its own address, which address should we use for half-words (2-byte chunks) or words (4-byte chunks)?
 - Start address = Smallest byte address within the larger chunk
- If we provide the start address (say 0x4000) to memory, how does it know whether we want the byte, halfword, or word at address 0x4000?
 - Other control signals indicate how many bytes to access (1=byte, 2=half, or 4=word)



	Byte Address	
Word 0x4000	0x4000	...
	0x4001	
	0x4002	
	0x4003	
Word 0x4004	0x4004	...
	0x4005	
	0x4006	
	0x4007	

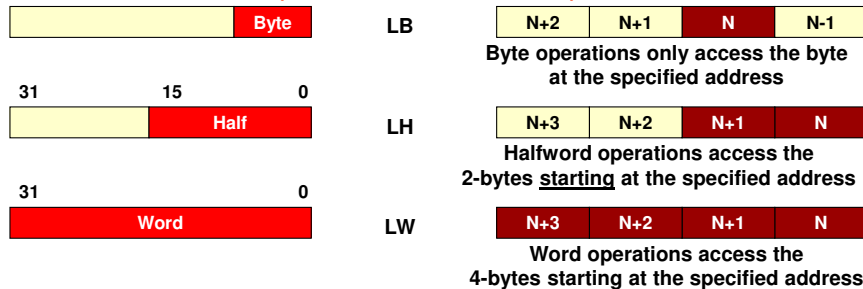
Getting data in and out of the processor

LOAD/STORE INSTRUCTIONS

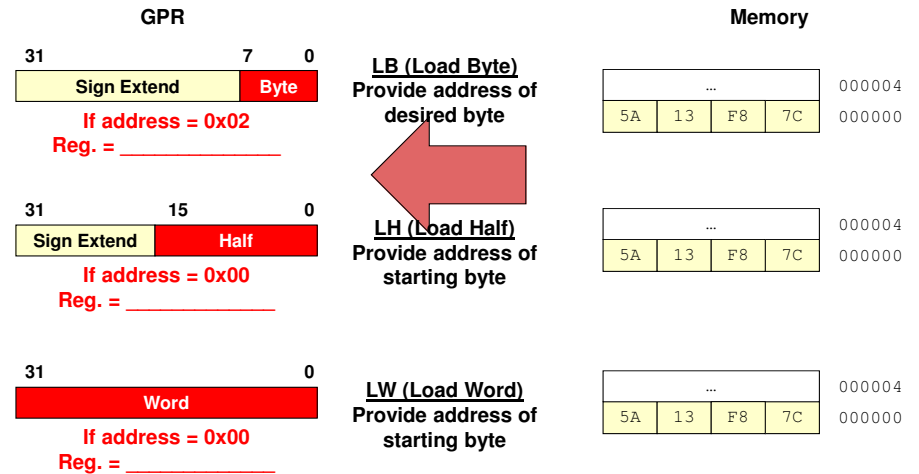
Memory & Data Size

- Little-endian memory can be thought of as right justified
- Always provide the _____ 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)

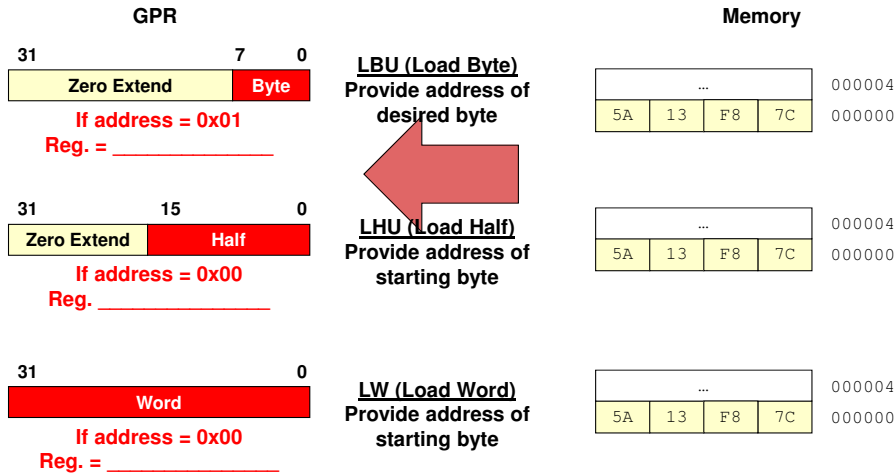
(Assume start address = N)



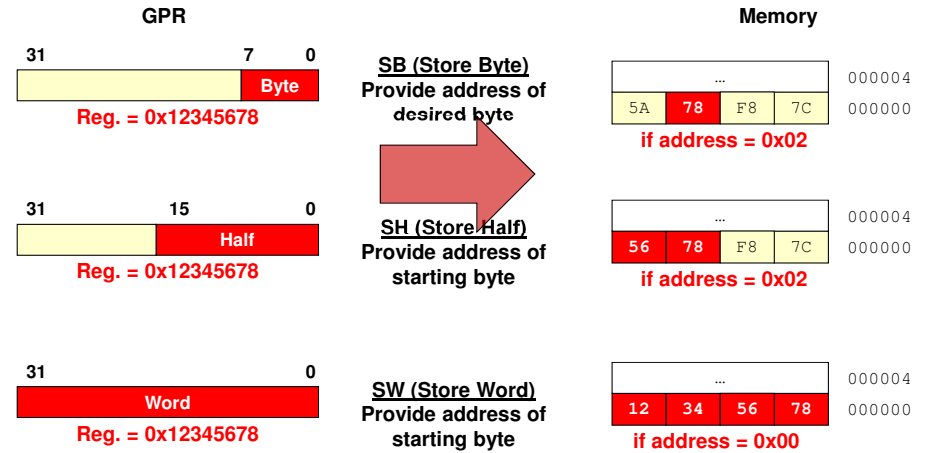
Memory Read Instructions (Signed)



Memory Read Instructions (Unsigned)

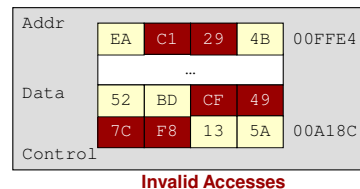
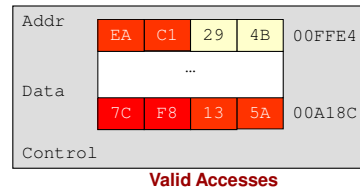


Memory Write Instructions



MIPS Memory Alignment Limitations

- Bytes can start at any address
- Halfwords must start on an _____ address
- Words must start on an address that is a _____
- Examples:
 - Word @ A18C -
 - Halfword @ FFE6 -
 - Word @ A18E -
 - Halfword @ FFE5 -



Load Format (LW, LH, LB)

- LW Rt, offset(Rs)
 - Rt = Destination register
 - offset(Rs) = Address of desired data
 - RTL: $R[t] = M[\text{offset} + R[s]]$
 - offset limited to 16-bit signed number

- Examples
 - LW \$2, 0x40(\$3) // R[2] = _____
 - LBU \$2, -1(\$4) // R[2] = _____
 - LH \$2, 0xFFFC(\$4) // R[2] = _____

R[2]	old val.	F8BE97CD	0x002048
R[3]	00002000	134982FE	0x002044
R[4]	0000204C	5A12C5B7	0x002040

More LOAD Examples

- Examples

- LB \$2,0x45(\$3) // R[2] = _____
- LH \$2,-6(\$4) // R[2] = _____
- LHU \$2,-2(\$4) // R[2] = _____

R[2]	old val.	F8BE97CD	0x002048
R[3]	00002000	134982FE	0x002044
R[4]	0000204C	5A12C5B7	0x002040

Store Format (SW,SH,SB)

- SW Rt, offset(Rs)

- Rt = Source register
- offset(Rs) = Address to store data
- RTL: $M[\text{offset} + R[s]] = R[t]$
- offset limited to 16-bit signed number

- Examples

- SW \$2, 0x40(\$3)
- SB \$2, -5(\$4)
- SH \$2, 0xFFFE(\$4)

R[2]	123489AB	89AB97CD	0x002048
R[3]	00002000	AB4982FE	0x002044
R[4]	0000204C	123489AB	0x002040

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

- immediates 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

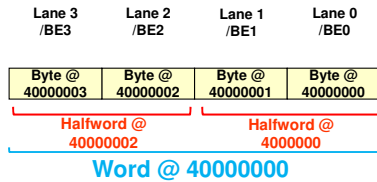
- _____
- _____

R[2]	
R[2]	12345678

PROC/MEM PHYSICAL INTERFACE

Memory & Word Size

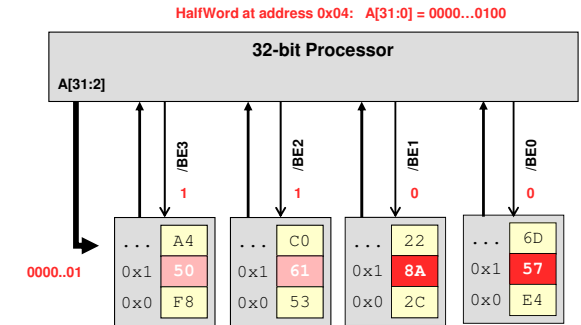
- What are the control signals that indicate access size?
- Though we may have a 32-bit address bus A[31:0], physically the processor will convert the lower 2 address bits A[1:0] and the size information into 4 separate _____
[/*BE3*../*BE0*]



Desired Memory	Internal A[31:0]	Physical Addr. Bus A[31:2]	/BE3	/BE2	/BE1	/BE0
Word @ 0x40000000	0100...00	0100...00				
Half @ 0x40000002	0100...00	0100...00				
Byte @ 0x40000002	0100...00	0100...00				
Byte @ 0x40000001	0100...00	0100...00				
Half @ 0x40000004	0100...01	0100...01				

Address & Data Bus Connections

- Organize memory into several byte-size memories running in parallel (sometimes known as “banks”)
- Convert lower address bits into bank enables to selectively enable each bank
- A[31:2] is provided to all memory banks specifying the same internal location

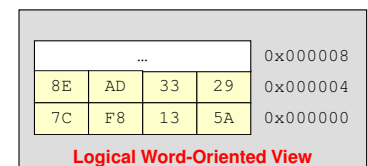
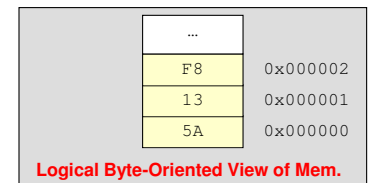
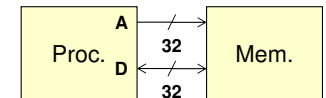


Byte Addressable Processors

Proc.	External Data Bus	Address Pin-Out	Min. # of Banks	Shift in Address
8088 (8-bit proc.)	D[7:0]	A[19:___]		
8086 (16-bit proc.)	D[15:0]	A[19:___]		
80386 (32-bit proc.)	D[31:0]	A[23:___]		
Core Series (64-bit proc.)	D[63:0]	A[35:___]		

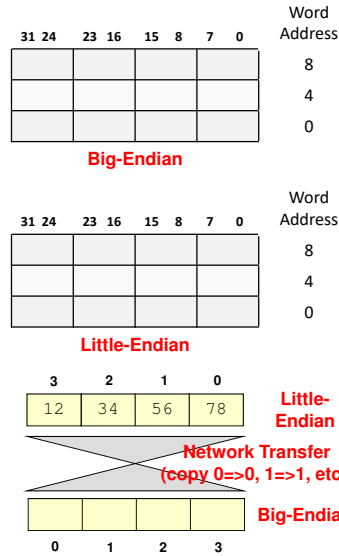
MIPS Memory Organization

- 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
- Logical view of memory arranged in rows of largest access size (word)
 - Still with separate addresses for each byte
 - Can get halfword or bytes



Little- vs. Big-Endian Organization

- Refers to ordering of bytes w/in a larger chunk
- Big-Endian
 - Byte '0' is at the _____ of a word
 - PPC, Sparc
- Little-Endian
 - Byte '0' is at the _____ of a word
 - Intel, _____
- MIPS can be configured either way
- Issues arise when moving smaller pieces within a large chunk across different endian-systems (e.g. TCP/IP transfer from little-endian machine to big-endian machine)



Program Flow Control

BRANCH INSTRUCTIONS

Instruction Boundaries

- If the current instruction is at address 0xA140, what address does the next instruction occupy?
 - Each instruction is 32-bits = 4-bytes
 - The next instruction is located @ 0xA144
- We see then that instructions always lie on an addresses that are multiples of 4
- Fact 1:** The PC register in the processor stores the address of the next instruction to be fetched
- Fact 2:** Registers are needed when we want to store *variable* bits
- Fact 3:** Addresses are 32-bits in MIPS
- Do we need a 32-bit register for the PC?

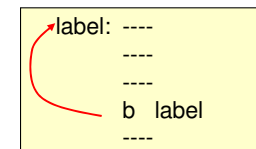
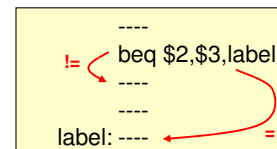
	MEM
0xA140	add
0xA144	sub
0xA148	bne

XX00 = 00000
XX04 = _____
XX08 = _____
XX0c = _____
XX10 = _____

Multiples of 4 in hex and binary

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: _____
 - 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

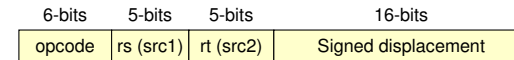
- If $R_s < R_t$ then $R_d = 1$, else $R_d = 0$

– Use appropriate BNE/BEQ instruction to infer relationship

Branch if...	SLT	BNE/BEQ
$\$2 < \3	SLT $\$1, \$2, \$3$	___ $\$1, \$0, label$
$\$2 \leq \3	SLT $\$1, \$3, \$2$	___ $\$1, \$0, label$
$\$2 > \3	SLT $\$1, \$3, \$2$	___ $\$1, \$0, label$
$\$2 \geq \3	SLT $\$1, \$2, \$3$	___ $\$1, \$0, label$

Branch Machine Code Format

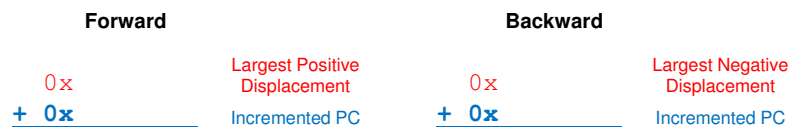
- Branch instructions use the I-Type Format



- Operation: $PC = PC + \{disp., 2'b00\}$
- Displacement notes
 - Displacement is the value that should be added to the PC so that it now points to the desired branch location
 - Processor appends two 0's to end of disp. since all instructions are 4-byte words
 - Essentially, displacement is in units of words

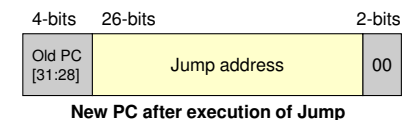
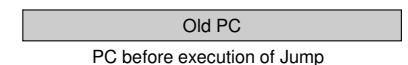
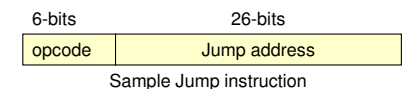
Range of Branching

- Suppose a branch instruction is located at $0x80000000$, how far away can you branch?
 - Displacement is 16-bits concatenated with two 0's
 - Largest positive 16-bit number: _____
 - Largest negative 16-bit number: _____



Jump Instructions

- Instruction format: J-Type
- 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



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: _____ = 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.)

SUPPORT FOR SUBROUTINES

Implementing Subroutines

- To implement subroutines in assembly we need to be able to:
 - Branch to the subroutine code (JAL / JALR)
 - Know where to return to when we finish the subroutine (JR \$ra)

C code:

```
...
res = avg(x,4);
...

int avg(int a, int b)
{ ... }
```



Assembly:

```
.text
...
jal  AVG
...

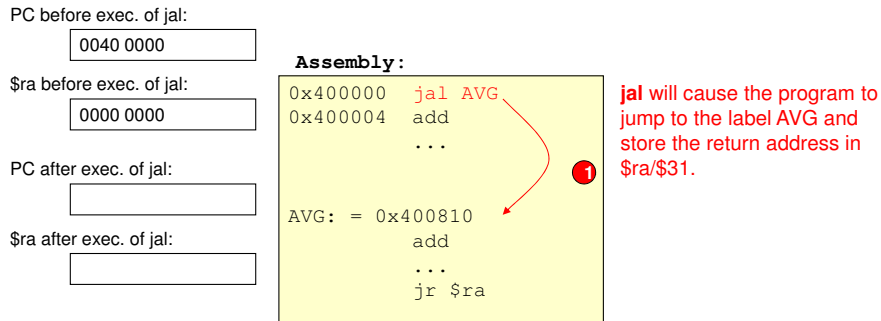
AVG:  ...
     jr  $ra
```

Jumping to a Subroutine

- JAL instruction (Jump And Link)
 - Format: **jal Address/Label**
 - Similar to jump where we load an address into the PC [e.g. PC = addr]
 - Same limitations (26-bit address) as jump instruction
 - Addr is usually specified by a label
- JALR instruction (Jump And Link Register)
 - Format: **jalr \$rs**
 - Jumps to address specified by \$rs
- In addition to jumping, JAL/JALR stores the _____ into R[31]=\$ra (= return address) to be used as a link to return to after the subroutine completes

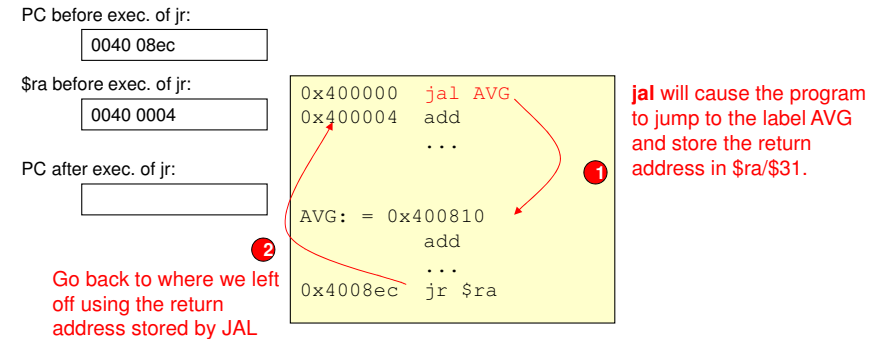
Jumping to a Subroutine

- Use the JAL instruction to jump execution to the subroutine and leave a link to the following instruction



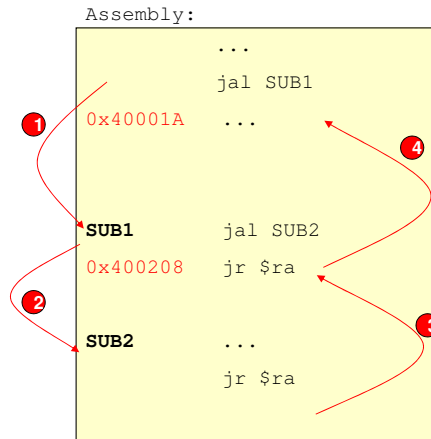
Returning from a Subroutine

- Use a JR with the \$ra register to return to the instruction after the JAL that called this subroutine



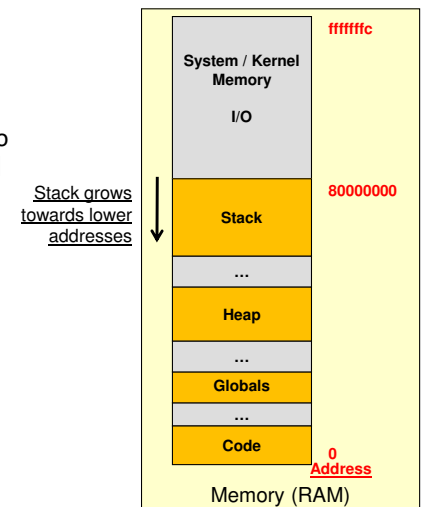
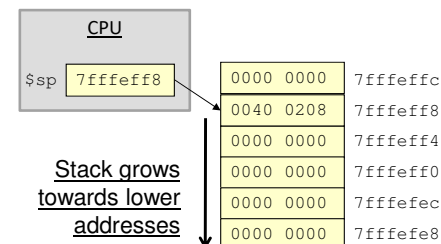
Dealing with Return Addresses

- Multiple return addresses can be spilled to memory
 - “Always” have enough memory
- Note: Return addresses will be accessed in reverse order as they are stored
 - 0x400208 is the second RA to be stored but should be the first one used to return
 - A stack/LIFO is appropriate!



Subroutines & Stacks

- Stack is a reserved area in memory
- Subroutines require a link (_____ address) to be saved on the stack
- Processors usually dedicate a register to point to the top of the stack (\$sp=R[29] = stack pointer)



Stack Facts

- Stack grows in the direction of:
 - Decreasing Addresses
 - Increasing address
- Stack is a (LIFO / FIFO) data structure.
- Stack Pointer points to the (top/bottom) of the stack
- Stack Pointer Register points to the
 - Top-most FILLED location
 - Next FREE location above the top-most filled location

Stack Facts

- When you push do you...
 - Increment the SP
 - Decrement the SP
- When you pop, first you _____ then you _____
- When you push do you
 - First update the SP and then place data
 - Place data then update SP

Stack Balancing

- Stack shall be balanced:
 - _____ number of push and pops
 - Pops shall be performed in _____ order as corresponding pushes

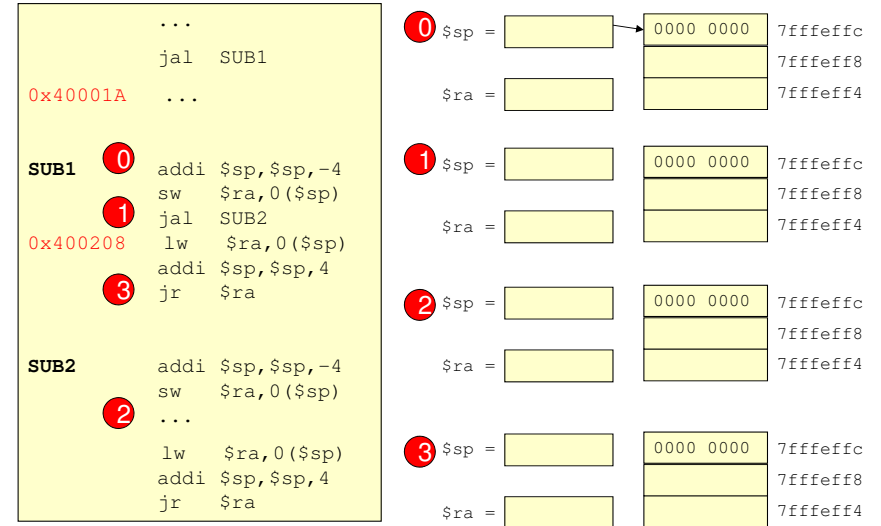
Subroutines Calling Subroutines

- Nested subroutines make the stack (grow / shrink) because more (stack pointer values / return addresses) are stored on the stack
- Recursive subroutines make the stack (grow / shrink)

Subroutines and the Stack

- When writing native assembly, programmer must add code to manage return addresses and the stack
- At the beginning of a routine (PREAMBLE)
 - Push \$ra (produced by 'jal') onto the stack
- Execute subroutine which can now freely call other routines
- At the end of a routine (POSTAMBLE)
 - Pop/restore \$ra from the stack

Subroutines and the Stack



Translating HLL to Assembly

- HLL variables are simply locations in memory
 - A variable name really translates to an address in 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
char a[100]; ... a[1]--;	LUI \$8, 0x1000 ORI \$8, \$8, 0x000C LB \$9, 1(\$8) ADDI \$9,\$9,-1 SB \$9,1(\$8)	Assume array 'a' starts @ 0x1000000C

Translating HLL to Assembly

C operator	Assembly	Notes
int dat[4],x; ... x = dat[0]; x += dat[1];	LUI \$8, 0x1000 ORI \$8, \$8, 0x0010 LW \$9, 0(\$8) LW \$10, 4(\$8) ADD \$9,\$9,\$10 SW \$9, 16(\$8)	Assume dat @ 0x10000010 & x @ 0x10000020
unsigned int y; short z; y = y / 4; z = z << 3;	LUI \$8, 0x1000 ORI \$8, \$8, 0x0010 LW \$9, 0(\$8) SRL \$9, \$9, 2 SW \$9, 0(\$8) LH \$9, 4(\$8) SLA \$9, \$9, 3 SH \$9, 4(\$8)	Assume y @ 0x10000010 & z @ 0x10000014

Translating HLL to Assembly

C operator	Assembly
int dat[4],x=0;	DAT: .space 16
for(i=0;i<4;i++)	X: .long 0
x += dat[i];	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)

Branch Example 1

C Code

```
if A > B (&A in $t0)
  A = A + B (&B in $t1)
else
  A = 1
```

MIPS Assembly

```
.text
LW $t2,0($t0)
LW $t3,0($t1)
SLT $1,$t3,$t2
BEQ $1,$0,ELSE
ADD $t2,$t2,$t3
B NEXT
ELSE: ADDI $t2,$0,1
NEXT: SW $t2,0($t0)
----
```

Could use pseudo-inst. "BLE \$4,\$5,ELSE"

This branch skips over the "else" portion. This is a pseudo-instruction and is translated to BEQ \$0,\$0,next

Branch Example 2

C Code

```
for(i=0;i < 10;i++) ($t0=i)
  j = j + i; ($t1=j)
```

MIPS Assembly

```
.text
ADDI $t0,$0,$0
LOOP: SLTI $1,$t0,10
      BEQ $1,$0,NEXT
      ADD $t1,$t1,$t0
      ADD $t0,$t0,1
      B LOOP
NEXT: ----
```

Branches if i is not less than 10

Loops back to the comparison check

Another Branch Example

C Code

```
int dat[10];
for(i=0;i < 10;i++) ($t1=i)
  data[i] = 5;
```

MIPS Assembly

```
.data
dat: .space 40
.text
la $t0,dat
addi $t1,$zero,10
addi $t2,$zero,5
LOOP: sw $t2,0($t0)
      addi $t0,$t0,4
      addi $t1,$t1,-1
      bnez $t1,$zero,LOOP
NEXT: ----
```

A Final Example

C Code

```
char A[] = "hello world";
char B[50];
// strcpy(B,A);
i=0;
while(A[i] != 0){
    B[i] = A[i]; i++;
}
B[i] = 0;
```

MIPS Assembly

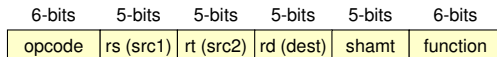
```
.data
A: .asciiz "hello world"
B: .space 50

.text
la $t0,A
la $t1,B
LOOP: lb $t2,0($t0)
      beq $t2,$zero,NEXT
      sb $t2,0($t1)
      addi $t0,$t0,1
      addi $t1,$t1,1
      b LOOP
NEXT: sb $t2,0($t1)
```

REFERENCE

R-Type Instructions

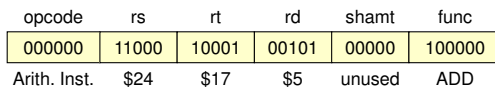
- Format



- 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
- opcode and func identify actual operation

- Example:

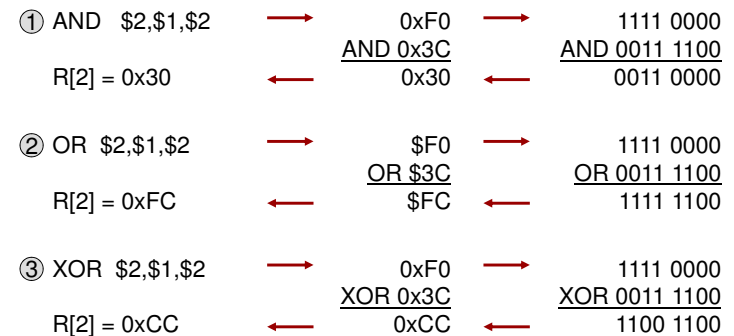
– ADD \$5, \$24, \$17



Logical Operations

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

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



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

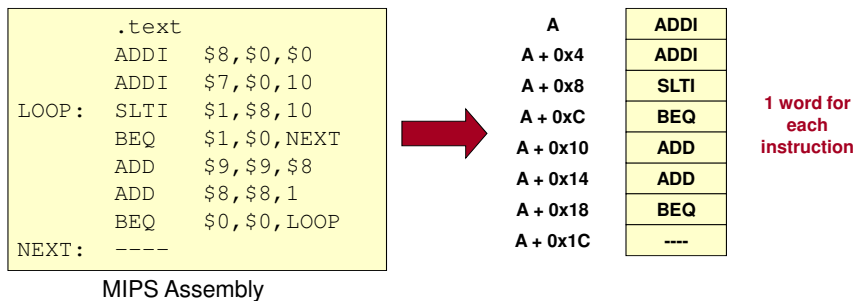
NOR \$2,\$1,\$1 → 0xF0 → 1111 0000
 NOR 0xF0 NOR 1111 0000
 R[2] = 0x0F ← 0x0F ← 0000 1111

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
 - $v = v ^ 0x0f;$

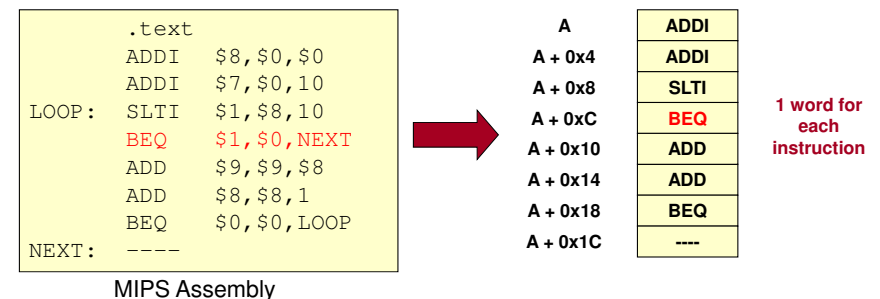
Calculating Branch Displacements

- To calculate displacement you must know where instructions are stored in memory (relative to each other)
 - Don't worry, assembler finds displacement for you...you just use the label



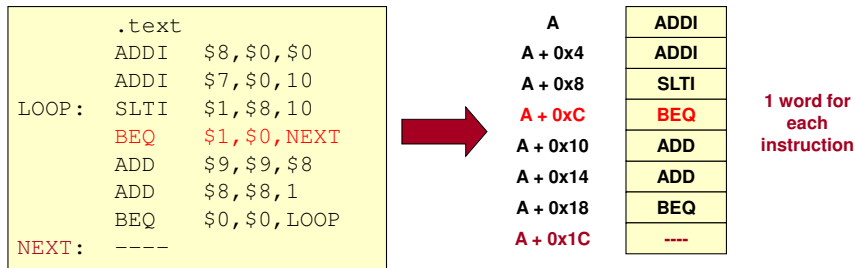
Calculating Displacements

- Disp. = [(Addr. of Target) – (Addr. of Branch + 4)] / 4
 - Constant 4 is due to the fact that by the time the branch executes the PC will be pointing at the instruction after it (i.e. plus 4 bytes)
- Following slides will show displacement calculation for BEQ \$1,\$0,NEXT



Calculating Displacements

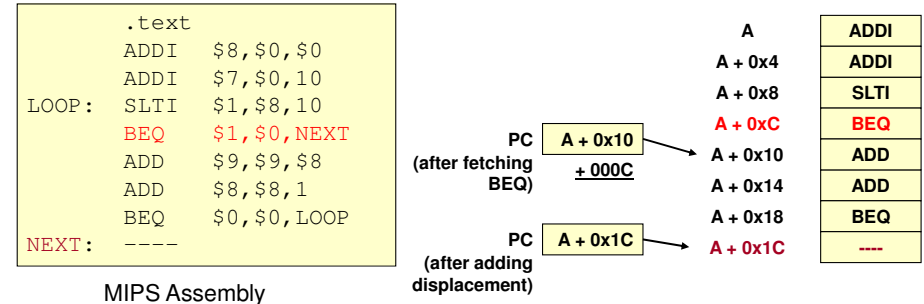
- $Disp. = [(Addr. of Target) - (Addr. of Branch + 4)] / 4$
- $Disp. = (A+0x1C) - (A+0x0C + 4) = 0x1C - 0x10 = 0x0C / 4 = 0x03$



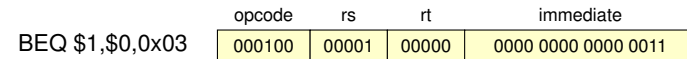
MIPS Assembly

Calculating Displacements

- If the BEQ does in fact branch, it will add the displacement ({0x03, 00} = 0x000C) to the PC (A+0x10) and thus point to the MOVE instruction (A+0x1C)

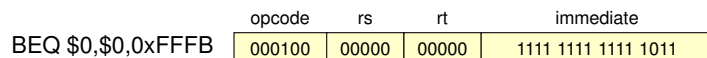
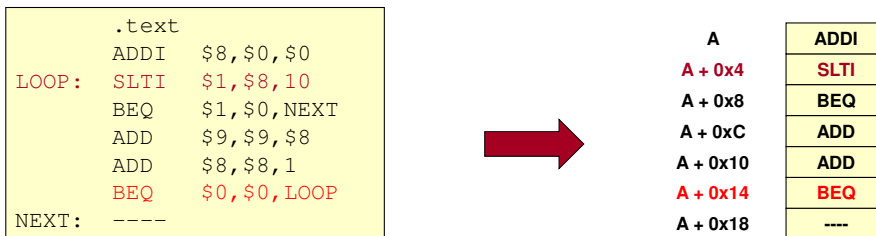


MIPS Assembly



Another Example

- $Disp. = [(Addr. of Label) - (Addr. of Branch + 4)] / 4$
- $Disp. = (A+0x04) - (A+0x14 + 4) = 0x04 - 0x18 = 0xFFEC / 4 = 0xFFFFB$



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 are limited to 16-bits, they must be extended to a full 32-bits when used 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

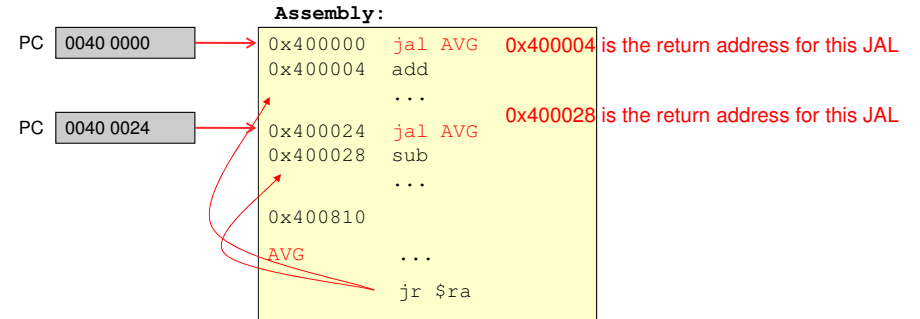
Loading an Immediate

- 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
 - Immediates 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

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

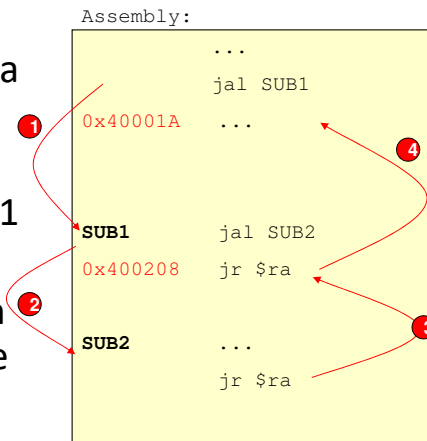
Return Addresses

- No single return address for a subroutine since AVG may be called many times from many places in the code
- JAL always stores the address of the instruction after it (i.e. PC of 'jal' + 4)



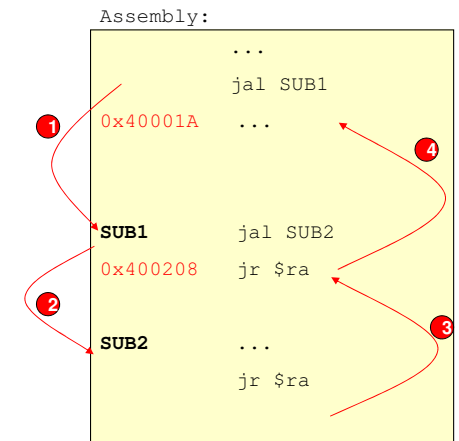
Return Addresses

- A further complication is nested subroutines (a subroutine calling another subroutine)
- Main routine calls SUB1 which calls SUB2
- Must store both return addresses but only one \$ra register



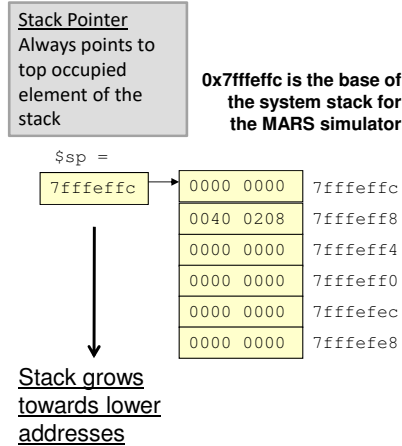
Dealing with Return Addresses

- Multiple return addresses can be spilled to memory
 - "Always" have enough memory
- Note: Return addresses will be accessed in reverse order as they are stored
 - 0x400208 is the second RA to be stored but should be the first one used to return
 - A stack is appropriate!



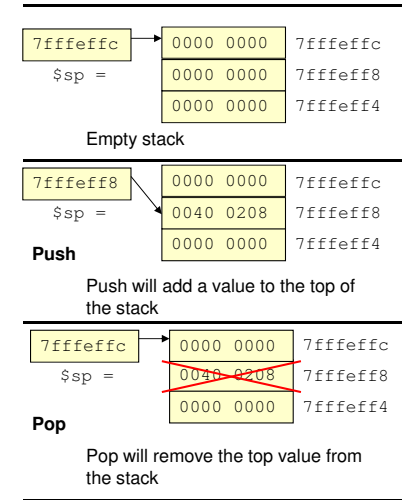
Stacks

- Stack is a data structure where data is accessed in reverse order as it is stored
- Use a stack to store the return addresses and other data
- System stack defined as growing towards smaller addresses
 - MARS starts stack at 0x7fffffc
 - Normal MIPS starts stack at 0x80000000
- Top of stack is accessed and maintained using \$sp=R[29] (stack pointer)
 - \$sp points at top **occupied** location of the stack



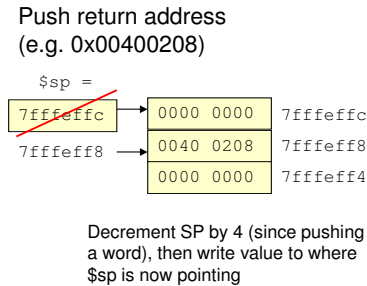
Stacks

- 2 Operations on stack
 - Push: Put new data on top of stack
 - Decrement \$sp
 - Write value to where \$sp points
 - Pop: Retrieves and "removes" data from top of stack
 - Read value from where \$sp points
 - Increment \$sp to effectively "delete" top value



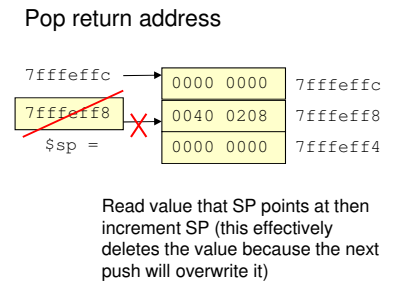
Push Operation

- Push: Put new data on top of stack
 - Decrement SP
 - addi \$sp,\$sp,-4
 - Always decrement by 4 since addresses are always stored as words (32-bits)
 - Write return address (\$ra) to where SP points
 - sw \$ra, 0(\$sp)



Pop Operation

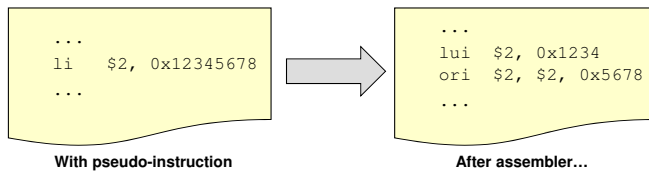
- Pop: Retrieves and "removes" data from top of stack
 - Read value from where SP points
 - lw \$ra, 0(\$sp)
 - Increment SP to effectively "delete" top value
 - addi \$sp,\$sp,4
 - Always increment by 4 when popping addresses



Warning: Because the stack grows towards lower addresses, when you push something on the stack you subtract 4 from the SP and when you pop, you add 4 to the SP.

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)



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. See MARS Help for more details.

Credits

- These slides were derived from Gandhi Puvvada's EE 457 Class Notes