EE 357 Unit 11

MIPS ISA

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

- 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
  - Halfword (H)
    - 16-bits = 2 bytes
  - Word (W)
    - 32-bits = 4 bytes

Floating Point

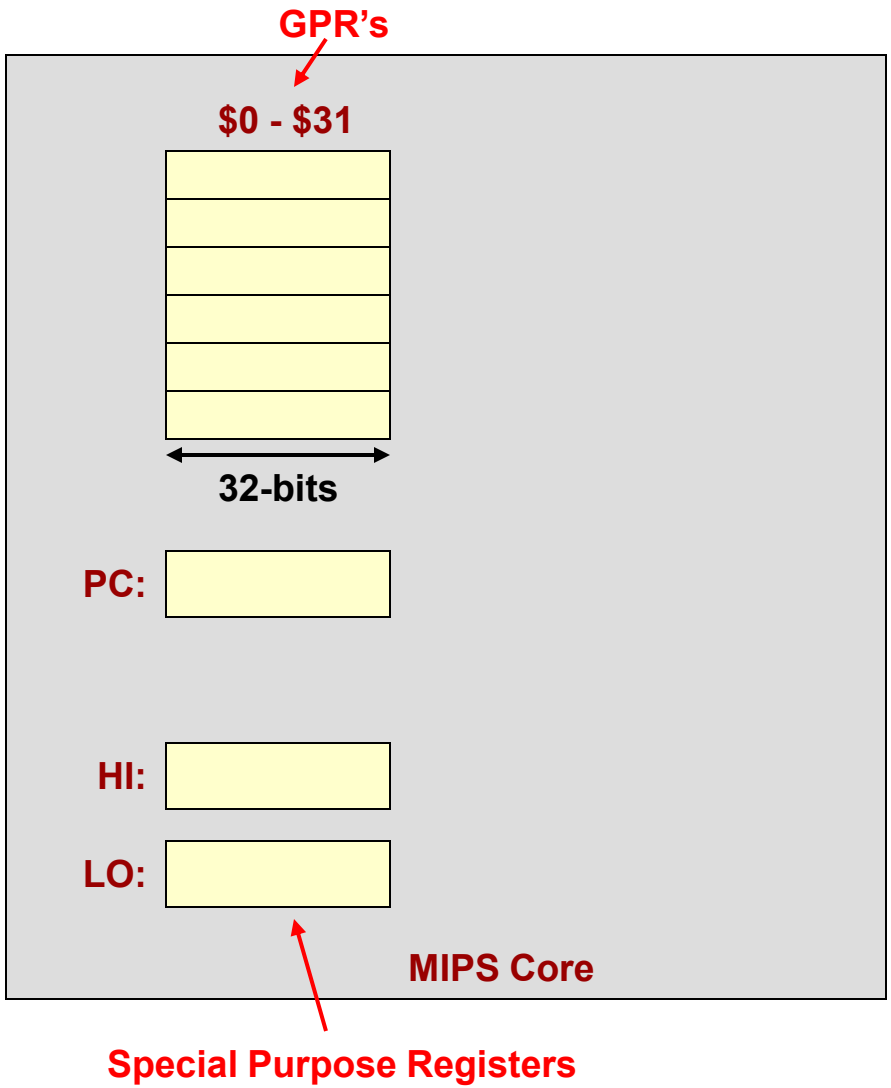
- 3 Sizes Defined
  - Single (S)
    - 32-bits = 4 bytes
  - Double (D)
    - 64-bits = 8 bytes
    - (For a 32-bit data bus, a double would be accessed from memory in 2 reads)

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

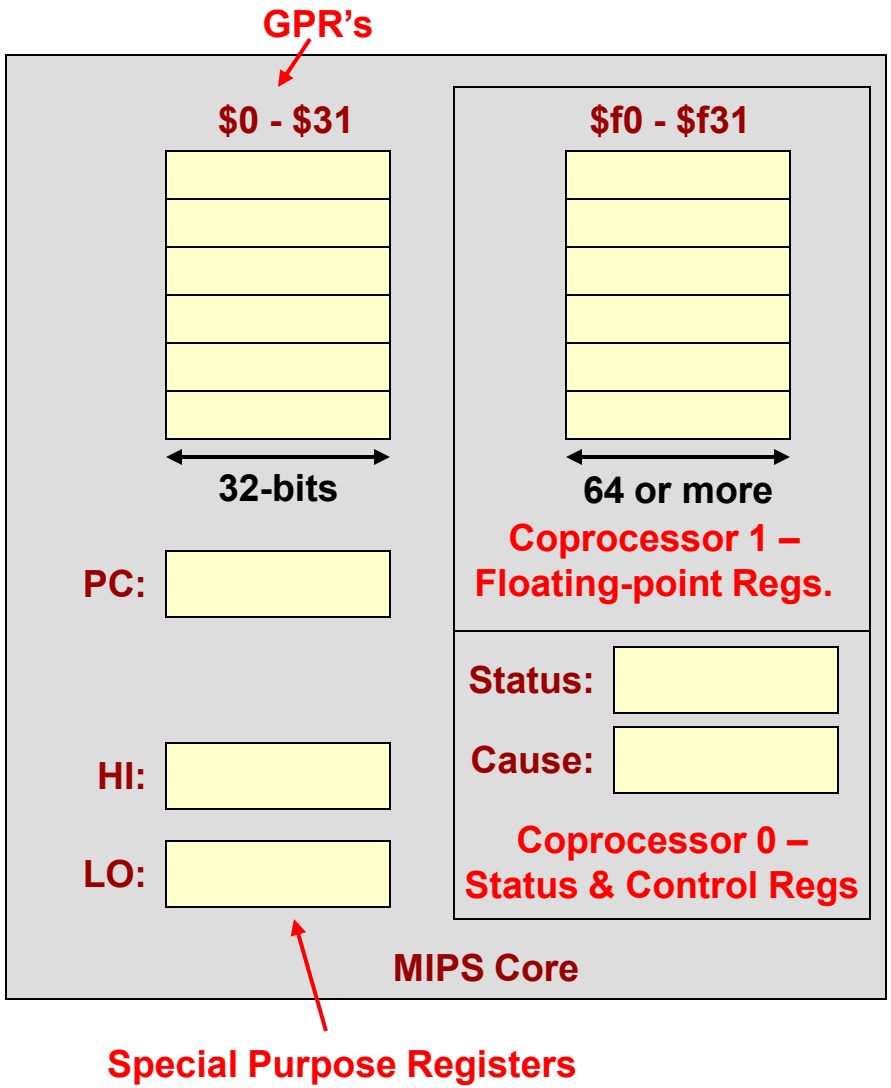
MIPS Programmer-Visible Registers

- 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

ALU (R-Type) Instructions

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

MIPS INSTRUCTIONS

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	Shorthand	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
  - opcode and func identify actual operation
- 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

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	Order: R[s] – R[t]. Sets R[d]=1 if R[s] < R[t], 0 otherwise

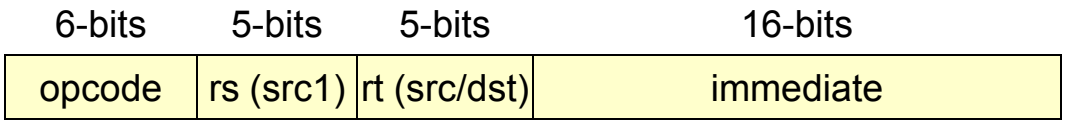
Shift Instructions

- 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
  - 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

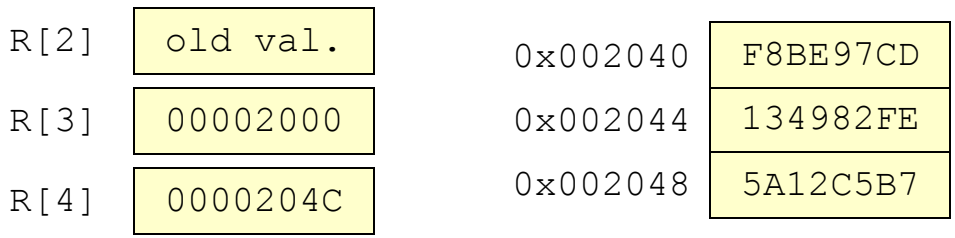
- Most ALU instructions also have an immediate form to be used when one operand is a constant value
- Syntax: ADDI Rs, Rt, imm
  - Because immediates are limited to 16-bits, they must be extended to a full 32-bits when used the by the processor
  - Arithmetic instructions always **sign-extend** to a 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

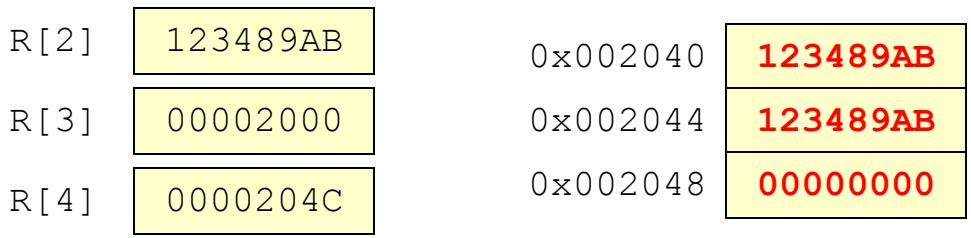
Load Format (LW)

- LW Rt, offset(Rs)
  - Rt = Destination register
  - offset(Rs) = Address of desired data
  - Shorthand: 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



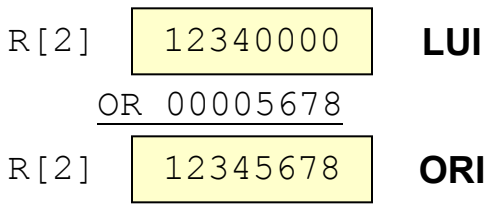
Store Format (SW)

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



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
    - LUI $2,0x1234
    - ORI $2,$2,0x5678



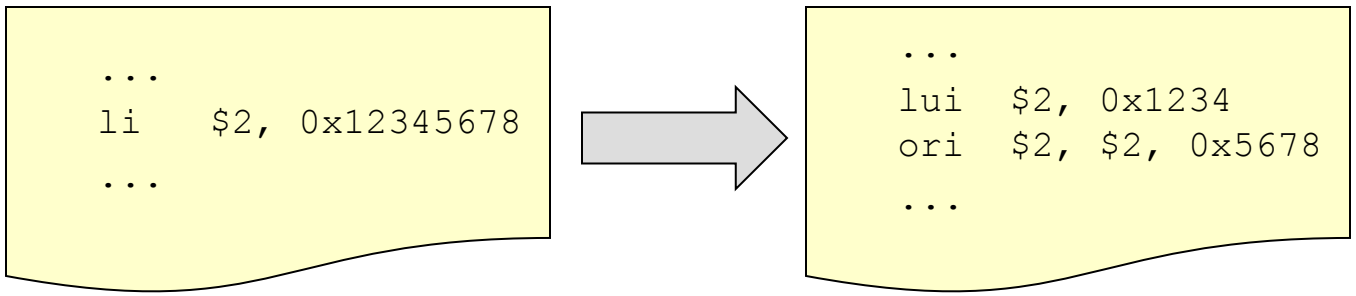
Translating HLL to Assembly

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

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)



With pseudo-instruction

After assembler…

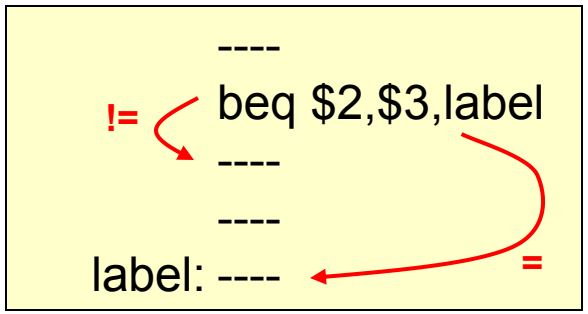
Pseudo-instructions

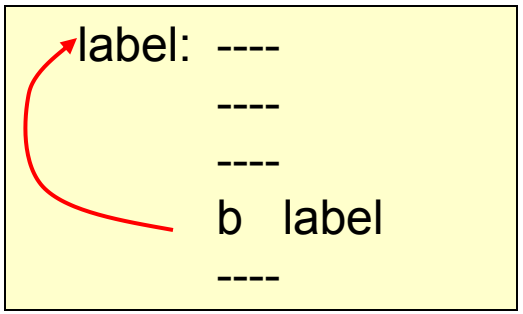
Pseudo-instruction	Actual Assembly
NOT Rd,Rs	NOR Rd,Rs,$0
NEG Rd,Rs	SUB Rd,$0,Rs
LI Rt, immed. # Load Immediate	LUI Rt, {immediate[31:16], 16'b0} ORI Rt, {16'b0, immediate[15:0]}
LA Rt, label # Load Address	LUI Rt, {immediate[31:16], 16'b0} ORI Rt, {16'b0, immediate[15:0]}
BLT Rs,Rt,Label	SLT $1,Rs,Rt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

Note: Pseudoinstructions are assembler-dependent. See MARS Help for more details.

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

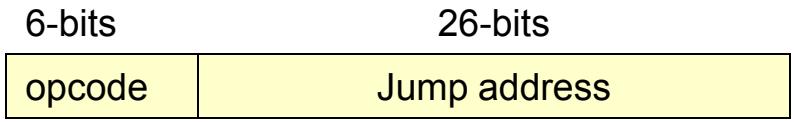
Translating HLL to Assembly

C operator	Assembly

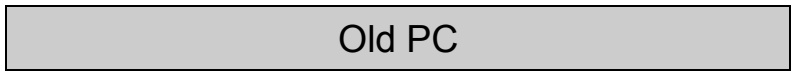
int dat[4],x=0;
for(i=0;i<4;i++)
x += dat[i];
	DAT: .space 16
	X: .long 0
	LA $8, DAT
	ADDI $9,$0,4
	ADD $10,$0,$0
	LP: LW $11,0($8)
	ADD $10,$10,$11
	ADDI $8,$8,4
	ADDI $9,$9,-1
	BNE $9,$0,LP
	LA $8,X
	SW $10,0($8)

Jump Instructions

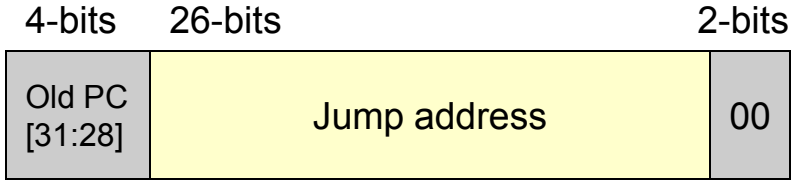
- Jumps provide method of branching beyond range of 16-bit displacement
- Syntax: J *label/address*
  - Operation: PC = address
  - Address is appended with two 0's just like branch displacement yielding a 28-bit address with upper 4-bits of PC unaffected
- New instruction format: J-Type



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.)