COMPUTER
ARCHITECTURE

CS 10 Computer Architecture and Organization
More Instruction Set Architecture

Foothill College
Computer Science Department

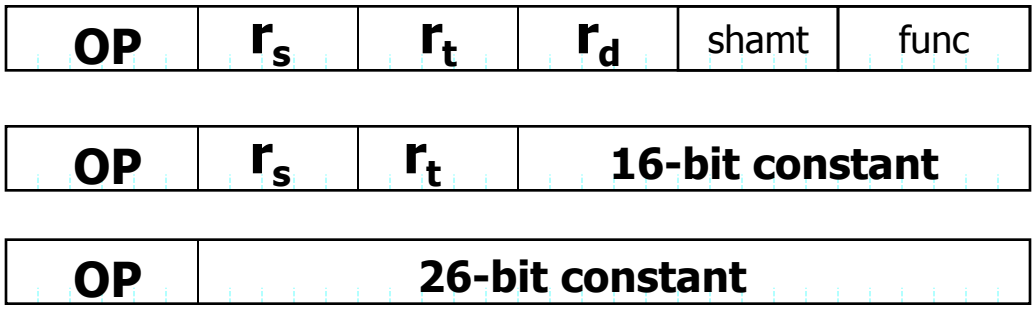
More MIPS Instructions

- MIPS instructions
  - signed vs. unsigned instructions
  - larger constants
  - accessing memory
  - branches and jumps
  - multiply, divide
  - comparisons
  - logical instructions

Recap: MIPS Instruction Formats

- All MIPS instructions fit into a single 32-bit word
- Every instruction includes various "fields":
  - a 6-bit operation or "OPCODE"
    - specifies which operation to execute (fewer than 64)
  - up to three 5-bit OPERAND fields
    - each specifies a register (one of 32) as source/destination
  - embedded constants
    - also called "literals" or "immediates"
    - 16-bits, 5-bits or 26-bits long
    - sometimes treated as signed values, sometimes unsigned
- There are three basic instruction formats:

- R-type, 3 register operands (2 sources, destination)
- I-type, 2 register operands, 16-bit constant
- J-type, no register operands, 26-bit constant



Working with Constants

- Immediate instructions allow constants to be specified within the instruction
  - Examples
    - add 2000 to register $5

```
addi $5, $5, 2000
```

    - subtract 60 from register $5

```
addi $5, $5, -60
```

      - … no **`subi`** instruction!
    - logically AND $5 with 0x8723 and put the result in $7

```
andi $7, $5, 0x8723
```

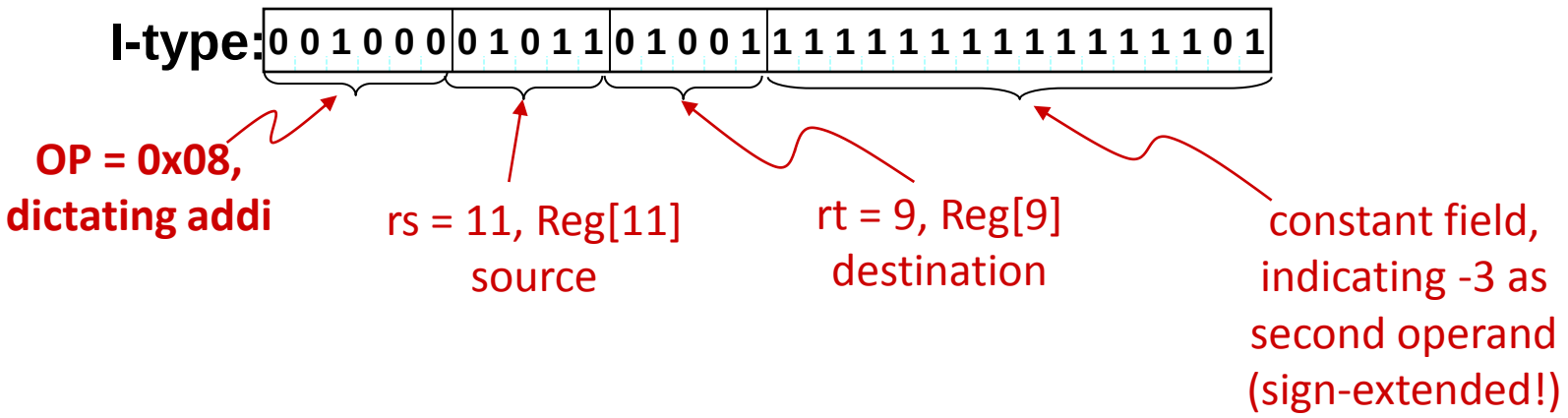
    - put the number 1234 in $10

```
addi $10, $0, 1234
```

  - But…
    - these constants are limited to 16 bits only!
      - Range is [-32768…32767] if signed, or [0…65535] if unsigned

Recap: ADDI

addi instruction: adds register contents, signed-constant:



Symbolic version: `addi $9, $11, -3`

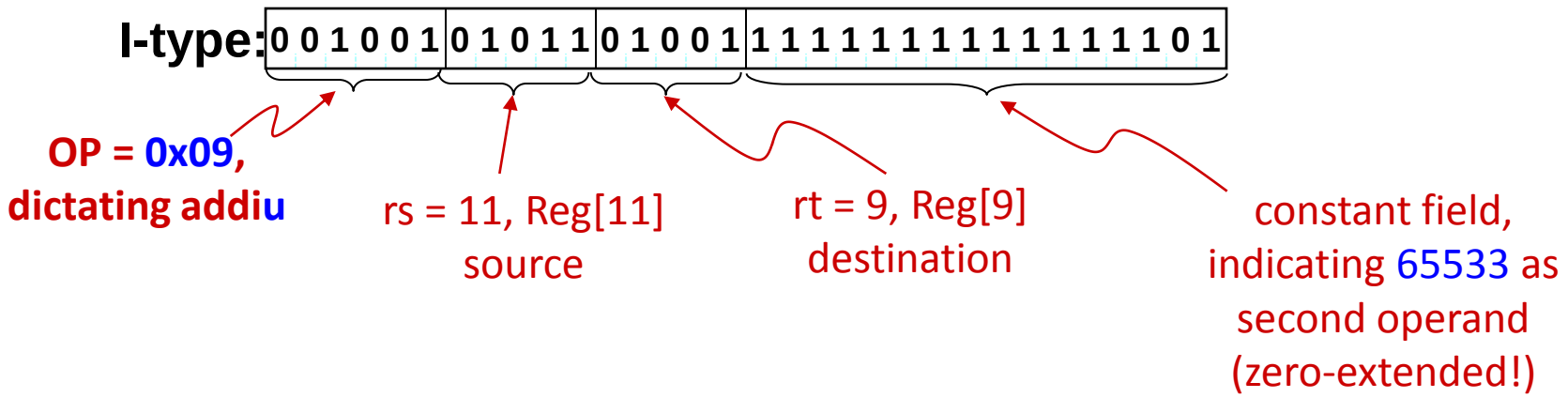
`addi rt, rs, imm`:

Reg[rt] = Reg[rs] + sxt(imm)

"Add the contents of rs to const; store result in rt"

ADDIU: Signed vs. Unsigned Constants

addiu instruction: adds register to unsigned-constant:



Symbolic version: `addiu $9, $11, 65533`

`addiu rt, rs, imm`:

Reg[rt] = Reg[rs] + (imm)

"Add the contents of rs to const; store result in rt"

Logical operations are always "unsigned", so always zero-extended

How About Larger Constants?

- Problem: How do we work with bigger constants?
  - Example: Put the 32-bit value 0x5678ABCD in $5
- One Solution:
  - put the upper half (0x5678) into $5
  - then shift it left by 16 positions (0x5678 0000)
  - now "add" the lower half to it (0x5678 0000 + 0xABCD)

```
addi $5, $0, 0x5678
sll  $5, $5, 16
addi $5, $5, 0xABCD
```

- One minor problem with this:
  - **addi** can mess up by treating the constants are signed
  - use **addiu** or **ori** instead

How About Larger Constants?

- Observation: This sequence is very common!
  - so, a special instruction was introduced to make it shorter
  - the first two (**addi + sll**) combo is performed by

lui

"load upper immediate"

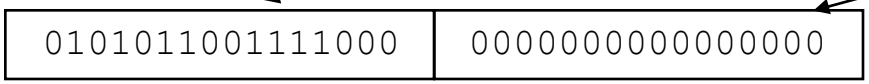
- puts the *16-bit* immediate into the *upper* half of a register

  - Example: Put the 32-bit value 0x5678ABCD in $5

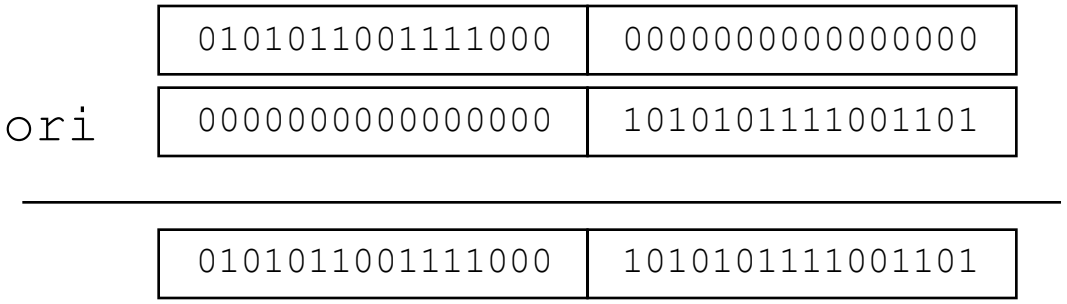
```
lui $5, 0x5678
ori $5, $5, 0xABCD
```

How About Larger Constants?

- Look at this in more detail:
  - "load upper immediate"
    lui $5, 0x5678 // 0101 0110 0111 1000 in binary



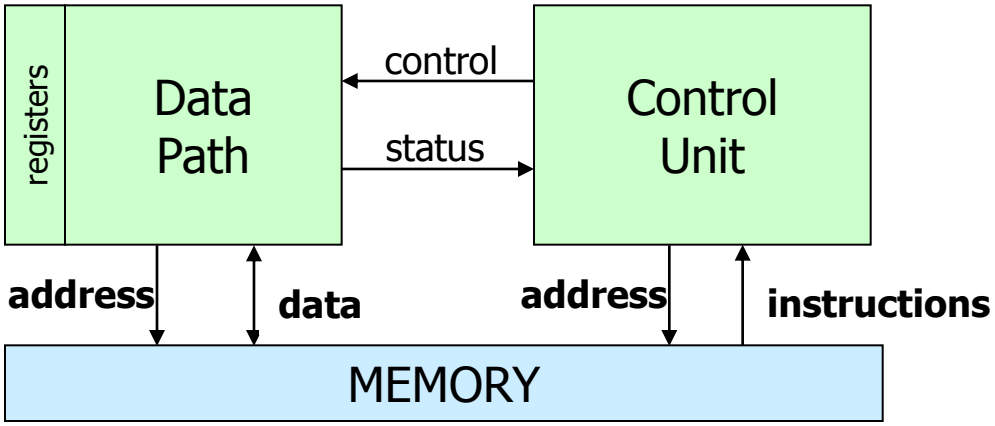
- Then must get the lower order bits right
  - ori $5, $5, 0xABCD // 1010 1011 1100 1101



Reminder: In MIPS, Logical Immediate instructions (ANDI, ORI, XORI) do not sign-extend their constant operand

Accessing Memory

- MIPS is a “load-store” architecture
  - all operands for ALU instructions are in registers or immediate
  - cannot directly add values residing in memory
    - must first bring values into registers from memory (called LOAD)
    - must store result of computation back into memory (called STORE)



MIPS Load Instruction

- Load instruction is I-type

`lw rt, imm(rs)`

Meaning: Reg[rt] = Mem[Reg[rs] + sign-ext(imm)]

Abbreviation: `lw rt,imm` for `lw rt, imm($0)`

  - Does the following:
    - takes the value stored in register $rs
    - adds to it the immediate value (signed)
    - this is the address where memory is looked up
    - value found at this address in memory is brought in and stored in register $rt

MIPS Store Instruction

- Store instruction is also I-type

`sw rt, imm(rs)`

Meaning: Mem[Reg[rs] + sign-ext(imm)] = Reg[rt]

Abbreviation: `sw rt,imm` for `sw rt, imm($0)`

  - Does the following:
    - takes the value stored in register $rs
    - adds to it the immediate value (signed)
    - this is the address where memory is accessed
    - reads the value from register $rt and writes it into the memory at the address computed

MIPS Memory Addresses

- **`lw`** and **`sw`** read whole 32-bit words
  - so, addresses computed must be multiples of 4
    - Reg[rs] + sign-ext(imm) must end in "00" in binary
  - otherwise: runtime exception

- There are also byte-sized flavors of these instructions
  - **`lb`** (load byte)
  - **`sb`** (store byte)
    - work the same way, but their addresses do not have to be multiples of 4

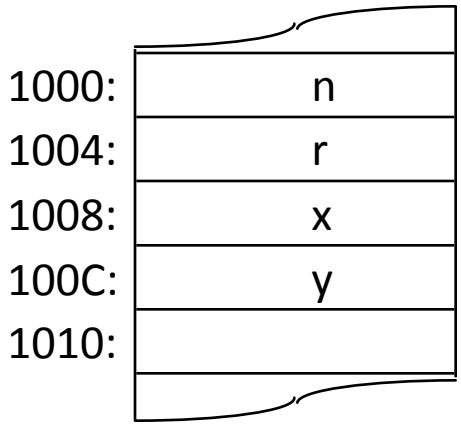
Storage Conventions

Addr assigned at compile time

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

```
int x, y;
y = x + 37;
```

Compilation approach:
LOAD, COMPUTE, STORE



translates to

```
lw      $t0, 0x1008($0)
addiu   $t0, $t0, 37
sw      $t0, 0x100C($0)
```

or, more humanely, to

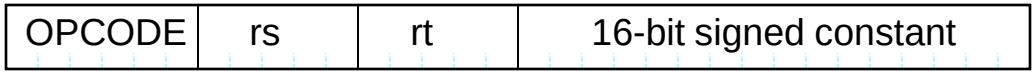
```
x=0x1008
y=0x100C
lw      $t0, x($0)
addiu   $t0, $t0, 37
sw      $t0, y
```

rs defaults to $0

MIPS Branch Instructions

MIPS *branch instructions* provide a way of conditionally changing the PC to some nearby location...

I-type:



`beq rs, rt, label` # Branch if equal

```
if (REG[RS] == REG[RT])
 {
   PC = PC + 4 + 4*offset;
 }
```

`bne rs, rt, label` # Branch if not equal

```
if (REG[RS] != REG[RT])
 {
   PC = PC + 4 + 4*offset;
 }
```

Notice on memory references offsets are multiplied by 4, so that branch targets are restricted to word boundaries.

Branch targets are specified relative to the next instruction (which would be fetched by default). The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset's value. The size of the constant field (16-bits) limits the range of branches.

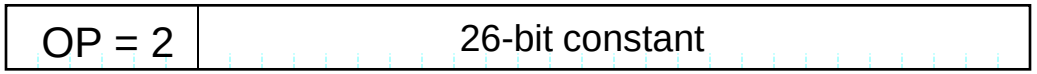
MIPS Jumps

- The range of MIPS branch instructions is limited to approximately ± 32K instructions (± 128K bytes) from the branch instruction.
- To branch farther: an unconditional jump instruction is used.

- Instructions:

```
j label          # jump to label (PC = PC[31-28] ||  CONST[25:0]*4)
jal label        # jump to label and store PC+4 in $31
jr $t0           # jump to address specified by register's contents
jalr $t0, $ra    # jump to address specified by first register's contents
                   and store PC+4 in second register
```

- Formats:
  - J-type: used for j



  - J-type: used for jal

  - R-type, used for jr

  - R-type, used for jalr

Multiply and Divide

- Slightly more complicated than add/subtract
  - multiply: product is twice as long!
    - if A, B are 32-bit long, A * B is how many bits?
  - divide: dividing integer A by B gives two results!
    - quotient and remainder
- Solution: two new special-purpose registers
  - “Hi” and “Lo”

Multiply

- MULT instruction
  - **`mult rs, rt`**
  - Meaning: multiply contents of registers $rs and $rt, and store the (64-bit result) in the pair of special registers **`{hi, lo}`**

    `hi:lo = $rs * $rt`
  - upper 32 bits go into **`hi`**, lower 32 bits go into **`lo`**
- To access result, use two new instructions
  - **`mfhi:`** move from hi

    `mfhi rd`
    - move the 32-bit half result from **`hi`** to $rd
  - **`mflo:`** move from lo

    `mflo rd`
    - move the 32-bit half result from **`lo`** to $rd

Divide

- DIV instruction
  - **`div rs, rt`**
  - Meaning: divide contents of register $rs by $rt, and store the quotient in **`lo`**, and remainder in **`hi`**

```
lo = $rs / $rt
hi = $rs % $rt
```

- To access result, use **`mfhi`** and **`mflo`**

- NOTE: There are also unsigned versions
  - **`multu`**
  - **`divu`**

Now we can do a real program: Factorial...

Synopsis (in C):

- Input in n, output in ans
- r1, r2 used for temporaries
- assume n is small

```
int n, ans, r1, r2;
r1 = 1;
r2 = n;
while (r2 != 0) {
   r1 = r1 * r2;
   r2 = r2 - 1;
}
ans = r1;
```

MIPS code, in assembly language:

```
n:      .word   123
ans:    .word   0
        ...
        addi    $t0, $0, 1          # t0 = 1
        lw      $t1, n($0)          # t1 = n
loop:   beq     $t1, $0, done       # while (t1 != 0)
        mult    $t0, $t1            # hi:lo = t0 * t1
        mflo    $t0                 # t0 = t0 * t1
        addi    $t1, $t1, -1        # t1 = t1 - 1
        j       loop                # Always loop back
done:   sw      $t0, ans($0)        # ans = r1
```

Comparison: `slt, slti`

- slt = set-if-less-than
  - **`slt rd, rs, rt`**

    $rd = ($rs < $rt) // “1” if true and “0” if false

- slti = set-if-less-than-immediate
  - **`slt rt, rs, imm`**

    $rt = ($rs < sign-ext(imm))

- also unsigned flavors
  - **`sltu`**
  - **`sltiu`**

Logical Instructions

- Boolean operations: bitwise on all 32 bits
  - AND, OR, NOR, XOR
  - **`and, andi`**
  - **`or, ori`**
  - **`nor`** // Note: There is no **`nori`**
  - **`xor, xori`**

- Examples:
  - **`and $1, $2, $3`**
    $1 = $2 & $3
  - **`xori $1, $2, 0xFF12`**
    $1 = $2 ^ 0x0000FF12
  - See all in textbook!

Summary - 1

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

Summary - 2

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional branch	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call