ICS 233
Computer Architecture & Assembly Language

MIPS PROCESSOR INSTRUCTION SET

ICS 233
Computer Architecture & Assembly Language

Lecture 5

Lecture Outline

- **MIPS Instruction Classes**
- **MIPS Instruction R-Type Format**
- **MIPS Arithmetic & Logical Instructions**

MIPS Instruction Set

- **MIPS Instructions can be classified into following categories**
  - **Integer Arithmetic**
    - **Arithmetic, logical, and shift instructions**
  - **Data Transfer**
    - **Load and store instructions that access memory**
    - **Data movement and conversions**
  - **Jump and Branch**
    - **Flow-control instructions that alter the sequential sequence**
  - **Floating Point Arithmetic**
    - **Instructions that operate on floating-point registers**
  - **Miscellaneous**
    - **Instructions that transfer control to/from exception handlers**
    - **Memory management instructions**

R-Type Format

- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction
- **funct**: function code – extends the opcode
  - Up to $2^6 = 64$ functions can be defined for the same opcode
  - MIPS uses opcode 0 to define R-type instructions
- Three Register Operands (common to many instructions)
  - **Rs**, **Rt**: first and second source operands
  - **Rd**: destination operand
  - **sa**: the shift amount used by shift instructions

Integer Add /Subtract Instructions

Instruction	Meaning	R-Type Format					
add $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x20
addu $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x21
sub $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x22
subu $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x23

- add & sub: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- addu & subu: same operation as add & sub
  - However, no arithmetic exception can occur
  - **Overflow is ignored**
- Many programming languages ignore overflow
  - The + operator is translated into **addu**
  - The **–** operator is translated into **subu**

MIPS - Arithmetic Instructions

- **add** (add registers signed)
  - **Instruction Mnemonic :**
    `add rd, rs, rt` ;where rs, rt, rd are registers, ;overflow detected
  - **Meaning :**
    rd ← rs + rt ; signed add operation
  - **Example :**
    `add $s1, $s2, $s3` ; $s1← $s2 + $s3

- **addu** (add registers unsigned)
  - **Instruction Mnemonic :**
    `addu rd, rs, rt` ;where rs, rt, rd are registers, ; overflow not detected
  - **Meaning :**
    rd ← rs + rt ; unsigned add operation
  - **Example :**
    `addu $s1, $s2, $s3` ; $s1← $s2 + $s3

MIPS - Arithmetic Instructions

- **sub** (subtract registers signed)
  - **Instruction Mnemonic :**
    `sub rd, rs, rt` ;where rs, rt, rd are registers, ;overflow detected
  - **Meaning :**
    rd ← rs - rt ; signed subtraction
  - **Example :**
    `sub $s1, $s2, $s3` ; $s1← $s2 - $s3

- **subu** (subtract registers unsigned)
  - **Instruction Mnemonic :**
    `subu rd, rs, rt` ;where rs, rt, rd are registers, ;overflow not detected
  - **Meaning :**
    rd ← rs - rt ; unsigned subtraction
  - **Example :**
    `subu $s1, $s2, $s3` ; $s1← $s2 - $s3

Addition/Subtraction Example

- **Consider the translation of: f = (g+h) – (i+j)**

- **Compiler allocates registers to variables**
  - Assume that *f*, *g*, *h*, *i*, and *j* are allocated registers $s0($16), $s1($17), $s2($18), $s3($19), $s4($20)
    i.e., $s0 ← f, $s1 ← g, $s2 ← h, $s3 ← i, $s4 ← j

- **Translation of: f = (g+h) – (i+j)**

```
addu $t0, $s1, $s2    # $t0 = g + h
addu $t1, $s3, $s4    # $t1 = i + j
subu $s0, $t0, $t1    # f = (g+h)–(i+j)
```

  - Temporary results are stored in $t0($8) and $t1($9)
  - Final result, i.e., computed f value is stored in $s0

- **Translate: `addu $t0,$s1,$s2` to binary code**
  - **Solution:**

op	rs = $s1	rt = $s2	rd = $t0	sa	func
000000	10001	10010	01000	00000	100001

Logical Bitwise Operations

- **Logical bitwise operations: and, or, xor, nor**

x	*y*	*x* and *y*
0	0	0
0	1	0
1	0	0
1	1	1

x	*y*	*x* or *y*
0	0	0
0	1	1
1	0	1
1	1	1

x	*y*	*x* xor *y*
0	0	0
0	1	1
1	0	1
1	1	0

x	*y*	*x* nor *y*
0	0	1
0	1	0
1	0	0
1	1	0

- **and** instruction is used to clear bits: ***x* and 0 = 0**
- **or** instruction is used to set bits: ***x* or 1 = 1**
- **xor** instruction is used to toggle bits: ***x* xor 1 = not *x***
- **nor** instruction can be used as a **not**, how?
  - **`nor $s1,$s2,$s2`** is equivalent to **`not $s1,$s2`**

Logical Bitwise Instructions

Instruction	Meaning	R-Type Format					
and $s1, $s2, $s3	$s1 = $s2 & $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x24
or $s1, $s2, $s3	$s1 = $s2 | $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x25
xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x26
nor $s1, $s2, $s3	$s1 = ~($s2|$s3)	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x27

- Examples:

Assume `$s1 = 0xabcd1234` and `$s2 = 0xffff0000`

```
and $s0,$s1,$s2     # $s0 = 0xabcd0000
or  $s0,$s1,$s2     # $s0 = 0xffff1234
xor $s0,$s1,$s2     # $s0 = 0x54321234
nor $s0,$s1,$s2     # $s0 = 0x0000edcb
```


MIPS - Logical Instructions

- **and (logical and)**
  - **Instruction Mnemonic :**

    and rd, rs, rt where rs, rt, rd are registers,
  - **Meaning :**

    rd ← rs & rt
  - **Example :**

    and $s1, $s2, $s3 ; $s1← $s2 & $s3

MIPS - Logical Instructions

❑or (logical or)

- **Instruction Mnemonic :**

  or rd, rs, rt ;where rs, rt, rd are registers,

- **Meaning :**

  rd ← rs | rt

- **Example :**

  or $s1, $s2, $s3 ; $s1← $s2 | $s3

MIPS - Logical Instructions

❑xor (logical Exclusive-or)

- **Instruction Mnemonic :**

  xor rd, rs, rt ;where rs, rt, rd are registers,

- **Meaning :**

  rd ← rs ⊕ rt

- **Example :**

  xor $s1, $s2, $s3 ; $s1← $s2 ⊕ $s3

MIPS - Logical Instructions

❑ **nor (logical nor)**

➢ **Instruction Mnemonic :**

nor rd, rs, rt **;where rs, rt, rd are registers,**

➢ **Meaning :**

rd ← rs nor rt

➢ **Example :**

nor $s1, $s2, $s3 **; $s1← $s2 nor $s3**