ICS 233
Computer Architecture & Assembly Language

MIPS PROCESSOR INSTRUCTION SET

ICS 233
Computer Architecture & Assembly Language

Lecture 7

Lecture Outline

- **MIPS Instruction I-Type Format**
- **MIPS I-type ALU Instructions**
- **MIPS I-type Data Transfer Instructions (Load & Store Instructions)**

I-Type Format

- Constants are used quite frequently in programs
  - The R-type shift instructions have a 5-bit shift amount constant
  - **What about other instructions that need a constant?**
- **I-Type: Instructions with Immediate Operands**

- 16-bit immediate constant is specified immediately in the instruction
  - Rs is the source register number
  - Rt is now the destination register number (for R-type it was Rd)
- **Examples of I-Type ALU Instructions:**
  - Add immediate: `addi $s1, $s2, 5  # $s1 = $s2 + 5`
  - OR immediate: `ori $s1, $s2, 5  # $s1 = $s2 | 5`

I-Type ALU Instructions

Instruction	Meaning	I-Type Format			
addi $s1, $s2, 10	$s1 = $s2 + 10	op = 0x8	rs = $s2	rt = $s1	imm^{16} = 10
addiu $s1, $s2, 10	$s1 = $s2 + 10	op = 0x9	rs = $s2	rt = $s1	imm^{16} = 10
andi $s1, $s2, 10	$s1 = $s2 & 10	op = 0xc	rs = $s2	rt = $s1	imm^{16} = 10
ori $s1, $s2, 10	$s1 = $s2 | 10	op = 0xd	rs = $s2	rt = $s1	imm^{16} = 10
xori $s1, $s2, 10	$s1 = $s2 ^ 10	op = 0xe	rs = $s2	rt = $s1	imm^{16} = 10
lui $s1, 10	$s1 = 10 << 16	op = 0xf	0	rt = $s1	imm^{16} = 10

- **addi: overflow causes an arithmetic exception**
  - In case of overflow, result is not written to destination register
- **addiu: same operation as addi but overflow is ignored**
- Immediate constant for addi and addiu is signed
  - No need for **subi** or **subiu** instructions
- Immediate constant for andi, ori, xori is unsigned

MIPS - Arithmetic Instructions

❑ **addi (add immediate signed)**

➢ **Instruction Mnemonic :**

```
addi  rd, rs, const        ;where rs,  rd are registers,
                           ; const is a 16-bit constant value
                           ;overflow detected
```

➢ **Meaning :**

rd ← rs + const

➢ **Example :**

```
addi $s1, $s2, 100         ; $s1← $s2 + 100
```

❑ **addiu (add immediate unsigned)**

➢ **Instruction Mnemonic :**

```
addiu  rd, rs, const       ;where rs,  rd are registers,
                           ; const is a 16-bit constant value
                           ;overflow not detected
```

➢ **Meaning :**

rd ← rs + const

➢ **Example :**

```
addiu $s1, $s2, 100        ; $s1← $s2 + 100
```

MIPS - Logical Instructions

❑ **andi (logical and immediate)**

➢ **Instruction Mnemonic :**

andi rd, rs, const ;where rs, rd are registers,
; const is a 16-bit constant

➢ **Meaning :**

rd ← rs & const

➢ **Example :**

andi $s1, $s2, 100 ; $s1← $s2 & 100

MIPS - Logical Instructions

❑ **ori (logical or immediate)**

➢ **Instruction Mnemonic :**

ori rd, rs, const ; where rs, rd are registers,
; const is a 16-bit constant

➢ **Meaning :**

rd ← rs | const

➢ **Example :**

ori $s1, $s2, 100 ; $s1← $s2 I 100

MIPS - Logical Instructions

❑ **xori (logical Exclusive-or immediate)**

➢ **Instruction Mnemonic :**

xori rd, rs, const ;where rs, rd are registers,
; const is a 16-bit value

➢ **Meaning :**

rd ← rs const

➢ **Example :**

xori $s1, $s2, 100 ; $s1← $s2 ⊕ 100

MIPS – Data Transfer Instructions

❑ **lui (Load upper immediate)**

➢ **Instruction Mnemonic :**

lui rd, const ;where rd is a register,
;const is a 16-bit number

➢ **Meaning :**

rd ← const*2^{16} **; load constant in upper 16 bits of register**

➢ **Example :**

lui $s1, 100 **; $s1 ← 100 * 2^{16}**

32-bit Constants

- I-Type instructions can have only 16-bit constants

- What if we want to load a 32-bit constant into a register?
- Can't have a 32-bit constant in I-Type instructions ☹
  - We have already fixed the sizes of all instructions to 32 bits
- Solution: use two instructions instead of one ☺
  - Suppose we want: **`$s1=0xAC5165D9`** (32-bit constant)
  - **`lui`**: **load upper immediate**

		load upper 16 bits	clear lower 16 bits
`lui $s1,0xAC51`	`$s1($17)`	`0xAC51`	`0x0000`
`ori $s1,$s1,0x65D9`	`$s1($17)`	`0xAC51`	`0x65D9`

Examples: I-Type ALU Instructions

- Examples: assume A, B, C are allocated $s0, $s1, $s2

```
A = B+5;      translated as   addiu $s0,$s1,5
C = B-1;      translated as   addiu $s2,$s1,-1
```

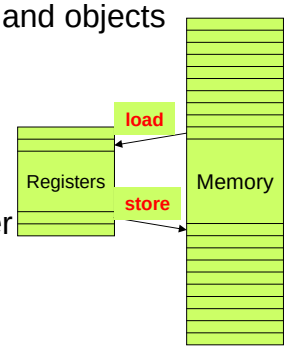
op=001001	rs=$s1=10001	rt=$s2=10010	imm = -1 = 1111111111111111

```
A = B&0xf;    translated as   andi  $s0,$s1,0xf
C = B|0xf;    translated as   ori   $s2,$s1,0xf
C = 5;        translated as   ori   $s2,$zero,5
A = B;        translated as   ori   $s0,$s1,0
```

- No need for **`subi`**, because **`addi`** has signed immediate
- Register 0 (**`$zero`**) has always the value 0

Load and Store Instructions

- Instructions that transfer data between memory & registers
- Programs include variables such as arrays and objects
- Such variables are stored in memory
- Load Instruction:
  - Transfers data from memory to a register
- Store Instruction:
  - Transfers data from a register to memory
- Memory address must be specified by load and store



Load and Store Word

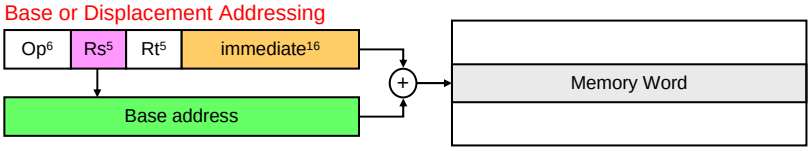
- Load Word Instruction (Word = 4 bytes in MIPS)

```
lw Rt, imm16(Rs)  # Rt = MEMORY[Rs+imm16]
```

- Store Word Instruction

```
sw Rt, imm16(Rs)  # MEMORY[Rs+imm16] = Rt
```

- Base or Displacement addressing is used
  - Memory Address = Rs (base) + Immediate^{16} (displacement)
  - Immediate^{16} is sign-extended to have a signed displacement

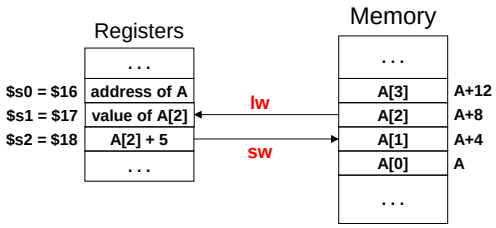


Example on Load & Store

- **Translate A[1] = A[2] + 5 (A is an array of words)**
  - Assume that address of array A is stored in register $s0

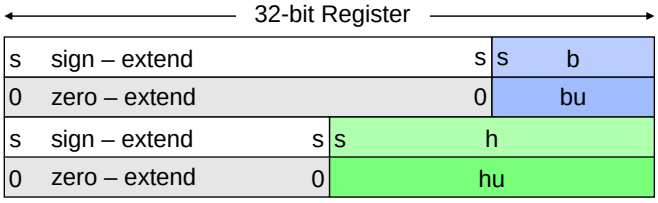
```
lw      $s1, 8($s0)        # $s1 = A[2]
addiu   $s2, $s1, 5        # $s2 = A[2] + 5
sw      $s2, 4($s0)        # A[1] = $s2
```

- Index of *a*[2] and *a*[1] should be multiplied by 4. Why?



Load and Store Byte and Halfword

- The MIPS processor supports the following data formats:
  - Byte = 8 bits, Halfword = 16 bits, Word = 32 bits
- Load & store instructions for bytes and halfwords
  - lb = load byte, lbu = load byte unsigned, sb = store byte
  - lh = load half, lhu = load half unsigned, sh = store halfword
- Load expands a memory data to fit into a 32-bit register
- Store reduces a 32-bit register to fit in memory



Load and Store Instructions

Instruction	Meaning	I-Type Format			
lb rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x20	rs^5	rt^5	imm^{16}
lh rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x21	rs^5	rt^5	imm^{16}
lw rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x23	rs^5	rt^5	imm^{16}
lbu rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x24	rs^5	rt^5	imm^{16}
lhu rt, imm^{16}(rs)	rt = MEM[rs+imm^{16}]	0x25	rs^5	rt^5	imm^{16}
sb rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x28	rs^5	rt^5	imm^{16}
sh rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x29	rs^5	rt^5	imm^{16}
sw rt, imm^{16}(rs)	MEM[rs+imm^{16}] = rt	0x2b	rs^5	rt^5	imm^{16}

- Base or Displacement Addressing is used
  - Memory Address = Rs (base) + $Immediate^{16}$ (displacement)
- Two variations on base addressing
  - If Rs = $zero = 0 then Address = $Immediate^{16}$ (absolute)
  - If $Immediate^{16}$ = 0 then Address = Rs (register indirect)

MIPS – Data Transfer Instructions

❑ **lw (Load word)**

➢ **Instruction Mnemonic :**

lw rd, const(rs) **;where rs, rd are registers, ;const is a 16-bit displacement**

lw rd, addr **; where addr is the label of the memory location to be accessed**

➢ **Meaning :**

rd ← Memory[rs + const] **; load word from memory to register**

➢ **Example :**

lw $s1, 100($s2) **; $s1 ← Memory[$s2+100]**

lw $s1, NUM1 **; load register $s1 with a word from memory location NUM1**

MIPS – Data Transfer Instructions

- **lh (Load halfword with sign extension)**
  - **Instruction Mnemonic :**

```
lh  rd, const(rs)      ;where rs, rd are registers,
                       ;const is a 16-bit displacement
lh rd, addr            ; where addr is the label of the memory
                         location to be accessed
```

  - **Meaning :**

```
rd ← Memory[rs + const]    ; load halfword from memory to register
```

  - **Example :**

```
lh  $s1, 100($s2)      ; $s1 ← Memory[$s2+100]
lh  $s1, NUM1          ; load register $s1 with a half-word from
                         memory location NUM1
```

- **lhu (Load halfword unsigned – without sign extension)**
  - **Instruction Mnemonic :**

```
lhu  rd, const(rs)     ;where rs, rd are registers,
                        ;const is a 16-bit displacement
lhu rd, addr           ; where addr is the label of the memory
                         location to be accessed
```

  - **Meaning :**

```
rd ← Memory[rs + const]    ; load halfword from memory to register
```

  - **Example :**

```
lhu  $s1, 100($s2)  ; $s1 ← Memory[$s2+100]
lhu  $s1, NUM1      ; load register $s1 with a half-word from
                      memory location NUM1
```


MIPS – Data Transfer Instructions

- **lb (Load byte with sign extension)**
  - **Instruction Mnemonic :**

```
lb rd, const(rs)       ;where rs, rd are registers,
                        ;const is a 16-bit displacement
lb rd, addr            ; where addr is the label of the memory
                         location to be accessed
```

  - **Meaning :**

```
rd ← Memory[rs + const]    ; load byte from memory to register
```

  - **Example :**

```
lb  $s1, 100($s2)      ; $s1 ← Memory[$s2+100]
lb  $s1, NUM1          ; load register $s1 with a byte from
                         memory location NUM1
```

- **lbu (Load byte unsigned – without sign extension)**
  - **Instruction Mnemonic :**

```
lbu  rd, const(rs)     ;where rs, rd are registers,
                       ;const is a 16-bit displacement
lbu rd, addr           ; where addr is the label of the memory
                         location to be accessed
```

  - **Meaning :**

```
rd ← Memory[rs + const]    ; load unsigned byte from memory to register
```

  - **Example :**

```
lbu  $s1, 100($s2)     ; $s1 ← Memory[$s2+100]
lbu  $s1, NUM1         ; load register $s1 with a byte from
                         memory location NUM1
```