3. Instructions: Encoding

Yang Song

UNIVERSITY OF SOUTH CAROLINA

College Of Engineering & Computing

R-Type Instruction

Four instruction encoding formats:

1. R-type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- op: operation: All R-type instructions use opcode 000000
- rs: first source operand
- rt: second source operand
- rd: destination operand
- shamt: shift amount- used only in shift operations
- funct: selects specific variant of the opcode

Syntax: $< op >$ \$rd, \$rs, \$rt

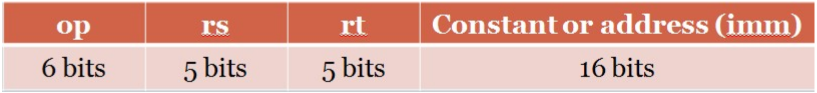
Example

```
add $t0, $s1, $s2
```

0	17	18	8	0	32

I-type Instruction

2. I-type



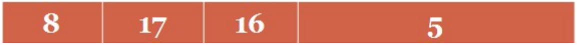
op	rs	rt	Constant or address (imm)
6 bits	5 bits	5 bits	16 bits

- All opcodes except 000000, 00001x, 0100xx are used for I-type instructions
- rs, imm are always as source

Syntax:

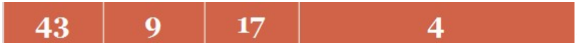
- $< op >$ \$rt, \$rs, imm
- $< op >$ \$rt, offset(\$rs)

addi \$s0, \$s1, 5



8	17	16	5

sw \$s1, 4(\$t1)



43	9	17	4

J-Type Instruction

3. J-type

op	pseudo-direct address
6 bits	26 bits

- J-type instructions use opcode 00001x.
- 26 bit pseudo-direct address

4. Co-processor Instruction
   MIPS processors all have two standard coprocessors, CP0 and CP1. CP0 processes various kinds of program exceptions. CP1 is a floating point processor.
   - All coprocessor instructions instructions use opcode 0100xx

Example

Reverse the instruction:

0xAD310004

to the corresponding MIPS Assembly code

Example Solution

1. Convert the hexadecimal representation to binary representation:

   0xAD310004

A	D	3	1	0	0	0	4
1010	1101	0011	0001	0000	0000	0000	0100

2. Look up the opcode and find its corresponding encoding

Opcode (6)	rs (5)	rt (5)	immediate (16)
101011	01001	10001	0000 0000 0000 0100
sw	$9	$17	4

Logical Operations

MIPS instructions for logical operations:

1. And (bitwise): and , andi
2. Or (bitwise): or, ori
3. Not (bitwise): nor
4. Shift left logical: sll

   ```
   sll $t2, $s0, 4
   ```

   - 00101001_{two} by $2_{ten} \rightarrow 10100100_{two}$
   - Multiply 41_{ten} by $2^2_{ten} = 164_{ten}$
5. Shift right logical: srl
   - 00101001_{two} by $2_{ten} \rightarrow 00001010_{two}$
   - Divide 41_{ten} by $2^2_{ten} (rounddown) = 10_{ten}$

MIPS Instruction Types

- MIPS uses three-address instructions for data manipulation: 1 destination, 2 sources
- MIPS is register-to-register architecture
  - Destination and source must be registers
  - Special instructions required to access main memory
- Types of MIPS instructions:
  1. Arithmetic/logical/shift/comparison
  2. Control instructions (branch and jump)
  3. Load/store
  4. Other (exception, register movement to/from register, etc.)

MIPS Registers

MIPS has 32 $\times$ 32-bit general purpose *integer* **registers**:

- Used for frequently accessed data
- Some have special purposes
- Numbered 0 to 31
- A 32-bit data called a ”*word*”
- Limiting to 32 registers speeds up register access time
  For memories, smaller is faster: influences clock cycle time

Program counter (**PC**) contains address of next instruction to be executed.

32 Registers

The 32 registers are the only registers the programmer can directly use.
Usage:

$RegisterNumber Or $RegisterName

```
$0 // constant 0 (cannot be overwritten )
$1 // $at (reserved for assembler)
$2,$3 // $v0,$v1 ( expression evaluation and results of a function )
$4−$7 // $a0−$a3 (arguments 1−4)
$8−$15 // $t0−$t7 (temporary registers )
$16−$23 // $s0−>$s7 (for saved local variables )
$24,$25 // $t8, $t9 (more temps)
$26,$27 // $k0, $k1 ( reserved for OS kernel)
$28 // $gp ( pointer to global area)
$29 // $sp ( stack pointer )
$30 // $fp (frame pointer )
$31 // $ra ( return address, for branch−and−links)
```

Register vs Memory

Why registers?

Registers are faster to access than memory

Register vs Memory

Why registers?

- Registers are faster to access than memory
- Operating on memory data always requires loads and stores

Register vs Memory

Why registers?

- Registers are faster to access than memory
- Operating on memory data always requires loads and stores
- Compiler must use registers for variables as much as possible

Register vs Memory

Why registers?

Registers are faster to access than memory

Operating on memory data always requires loads and stores

Compiler must use registers for variables as much as possible

- Only spill to memory for less frequently used variables
- Register optimization is important!

Memory Operands

Main memory used for composite data, such as arrays, structures, dynamic data.

- Memory is *byte addressed*. Each address identifies an 8-bit byte.
  - * 1 word = 4 bytes = 32 bits
  - * 1 halfword = 2 byte = 16 bits
  - * 1 byte = 8 bits
- Words are aligned in memory. Address must be a multiple of 4.
- MIPS is Big Endian
  Most-significant byte at least address of a word.
  c.f. Little Endian: least-significant byte at least address
  ▸ Example
  0x1234

Arithmetic Operations

- Add and subtract, three operands: 2 sources and 1 destination

```
add a, b, c  #a gets b + c
```

Arithmetic Example

C code:

```
f = (g + h) - (i + j);
```

Compiled MIPS code:

```
add $t0, $s1, $s2   # temp $t0 = g + h
add $t1, $s3, $s4   # temp $t1 = i + j
sub $s0, $t0, $t1  # f = $t0 - $t1
```

(the variables f, g, h, i, j are assigned to reg \$s0, \$s1, \$s2, \$s3, \$s4, respectively)

- All arithmetic operations have this form

Memory Operations Example: Load

Consider an assignment when an operand is in memory, the load operation is needed.
Assume *g* in register $s1, *h* in $s2, *A* is an array of 100 words, the base address of *A* is $s3.
Given the C code:

```
g = h + A[8];
```

Compiled MIPS code:
NOTE that index 8 requires offset of 32 since one word carries 4 bytes

```
lw  $t0, 32($s3) #load word at A[8] to temporary reg $t0
add $s1, $s2, $t0
```

Memory Operations Example: Save

Following the scenario above, suppose the destination of the assignment is $A[12]$.
So the C code turns to be:

```
A[12] = h + A[8];
```

And the compiled MIPS code turns to be:

```
lw   $t0, 32($s3) #load word at A[8] to temporary reg $t0
add $s1, $s2, $t0 #temporary reg $t0 gets h+A[8]
sw   $t0, 48($3)   #stores result back into A[12]
```

Instructions **lw** and **sw** are the load and save operation for word.
Similar instructions are:

- lh, lhu, lb, lbu
- sh, sb

Constant and Immediate Operands

Consider following MIPS code:

```
lw $t0, AddressOfConstant4($s1)  # $t0 gets constant 4
add $s0, $s3, $t0 #add $t0 to reg $s3 ($t0 == 4)
```

Immediate operand avoids a load instruction from memory to register. A quick add instruction with one constant operand is called ”*add immediate*”:

```
addi $s3, $s3, 4
```

Note: there is NO subtract immediate instruction (use a negative constant instead). E.g.:

```
addi $s2, $s1, −1
```

Question What does following code mean?

```
add $s2, $s1, $0
```

Binary Representation for Unsigned Integer

Generally, in any number base, the value of i^{th} digit d is:

$$d \times Base^i$$

- Suppose x is a binary number that consists of n digits:
  $x = x_{n-1}x_{n-2}...x_1x_0$
  therefore, it means
  $x = x_{n-1} \times 2^{n-1} + x_{n-2} \times 2^{n-2} + ... + x_1 \times 2^1 + x_0 \times 2^0$
- The range of n-digit binary number is from 0 to 2^{n-1}
  A 32-bit unsigned binary integer number range is 0 to +4,294,967,295

For example,
1101_{two}
represents:
$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 11_{ten}$

2s-Complement Representation for Signed Integers

For signed integer, the binary representation tells the sign of the number by the **MSB** (Most Significant Bit, the leftmost bit or bit 31 for MIPS processor). This convention is called "**two's complement**" representation:

1. MSB 0 means Positive
   e.g. 0000 0000 0000 0000 0000 0000 0000 0010 $= 2_{ten}$
2. MSB 1 means Negative
   e.g. 1111 1111 1111 1111 1111 1111 1111 1110 $= -2_{ten}$

The range for a n-digit binary signed integer is from -2^{n-1} to 2^{n-1}.

- For 32-bit MIPS processor, the range is -2,147,483,648 to +2,147,483,647.
  -2^{n-1} cant be represented
- For non-negative signed integer, its 2s complement representation is exactly the same as the unsigned binary representation

Sign Negation

Let x_{two} be the 2s-complement representation of an integer X_{ten} that is,

$x_{two} = x_{n-1}x_{n-2}...x_1x_0$

Two steps to find the 2s-complement representation of its negation:

Sign Negation

1. Complement: $1 \rightarrow 0$ or $0 \rightarrow 1$
2. Add 1

▸ Exercise

$1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 0110 \quad 1110 = ?_{ten}$

Sign Extension

- Representing a number using more bits but preserve the numeric value
- In MIPS instruction set
  - addi: extend immediate value
  - lb, lh: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  c.f. unsigned values: extend with 0s

Example

Extend an 8-bit number to 16-bit

2_{ten} : 0000 0010 $\rightarrow$ 0000 0000 0000 0010

-2_{ten} : 1111 1110 $\rightarrow$ 1111 1111 1111 1110