5118007-02 Computer Architecture

Ch. 2 Instructions: Language of the Computer

19 Mar 2024

Shin Hong

Bit-wise Operations

- Bit-wise operations are to operate on a part of word (e.g., 8 bits, 1 bit)
  - extract a third byte in a word
  - read a specific flag in a byte, and turn it on and off

- Bit-wise logical operators

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Shift

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

- Move all the bits in a word to the left (to most-significant bit) or to the right (least-significant bit)
  - `sll` <$d>, <$s>, <n> # $d = $s << n
  - `srl` <$d>, <$s>, <n> # $s = $s >> n
  - Ex.

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

Bit-wise AND

- Bit-by-bit operation that produces 1 if and only if both bits of the operands are 1
  - `and <$d> <$r1> <$r2>`
  - used for reading a specific bit with a bitmask
- Ex.
  - `$t2 :` 0000 0000 0000 0000 0000 **1101 1100** 0000
  - `$t1 :` 0000 0000 0000 0000 00**11 1100** 0000 0000
  - `$t1 AND $t2:` 0000 0000 0000 0000 0000 **1100** 0000 0000

Bit-wise OR

- Bit-by-bit operation that produces 1 if and only if either bits of the operands is 1
  - `or` <$d> <$r1> <$r2>
  - used for writing specific bits with a bitmask
- Ex.
  - `$t2 :` 0000 0000 0000 0000 0000 **1101 1100** 0000
  - `$t1 :` 0000 0000 0000 0000 00**11 1100** 0000 0000
  - `$t1 OR $t2:` 0000 0000 0000 0000 00**11 1101 1100** 0000

Jump for Branching and Looping

- A jump instruction is to determine which instruction will be executed in the next cycle

- Conditional jump (branching)
  - `beq <$r1>, <$r2>, L1    # if (r1 == r2) goto L1 ;`
  - `bne <$r1>, <$r2>, L1    # if (r1 != r2) goto L1 ;`

- Unconditional jump
  - `j L1                    # goto L1 ;`

Example

```
if (e == f) {
    a = b + c ;
}
else {
    a = b - c ;
}
```

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else:
sub $s0, $s1, $s2
Exit:
```

Example

```
int a[100] ;
(…)
while (a[i] == k) {
  i += 1 ;
}
```

```
# $s3:i, $s5:k, $s6:s
Loop:
sll $t1, $s3, 2
add $t1, $t1, $s6
lw $t0, 0($t1)
bne $t0, $s5, Exit
addi $s3, $s3, 1
j Loop
Exit:
```

Test for Inequality

- Set-on-less-than instruction
  - `slt <$t0>, <$s3>, <$s4>`
    - `$t0` will be 1 if `$s3` < `$s4`; 0, otherwise.
  - `slti <$t0>, <$s3>, <n>`
  - `stlu <$t0>, <$s3>, <$s4>`
    - `test $s3` < `$s4` as unsigned values
    - ex. `$s3 : 0000 0000 0000 0000 0000 1101 1100 0000`

      `$s4 : 1111 1111 1111 1111 1111 1111 1111 1111`

Procedure Call (1/2)

- A procedure (or function) is a sequence of instructions that provides a specific functionality

- Jump-and-link instruction
  - `jal <L1>`
    - jump to L1, the starting location of a target procedure while automatically storing the location of the next instruction (i.e., PC+4) at `$ra`
  - `jr $ra`
    - come back to the origin

Procedure Call (2/2)

1. put parameters in a place where the procedure accesses
   - `$a0-$a3` : four arguments
2. jump to the beginning of the procedure
3. acquire the storage resources needed for the procedure
4. perform the desired task
5. put the result value when the caller accesses
   - `$v0-$v1`: return values
6. return the control back to the origin

Stack

- Allocate memory for a procedure execution using a special region called *stack*
  - stack pointer (`$sp`): a register indicating the top
  - push : to place given data to the stack
  - pop: to remove stored data from the stack
- The stack grows downward
  - initially, `$sp` is assigned with the greatest address
  - decrease `$sp` by the number of bytes that a procedure uses
  - refer each local variable with `$sp` as base

Example

```
int leaf_example
(int g, int h, int i, int j)
{
      int f;

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

```
leaf_example:

addi $sp, $sp, -12  # adjust stack to make room for 3 items
sw  $t1, 8($sp)     # save register $t1 for use afterwards
sw  $t0, 4($sp)     # save register $t0 for use afterwards
sw  $s0, 0($sp)     # save register $s0 for use afterwards

add $t0,$a0,$a1 # register  $t0 contains g + h
add $t1,$a2,$a3 # register  $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)


lw $s0, 0($sp)  # restore register $s0 for caller
lw $t0, 4($sp)  # restore register $t0 for caller
lw $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items

jr   $ra    # jump back to calling routine
```