Logic operations

- Instructions that operate on fields of bits within a word.
- The packing and unpacking of bits into words.
- Quick summary of important logic instructions

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 left logical

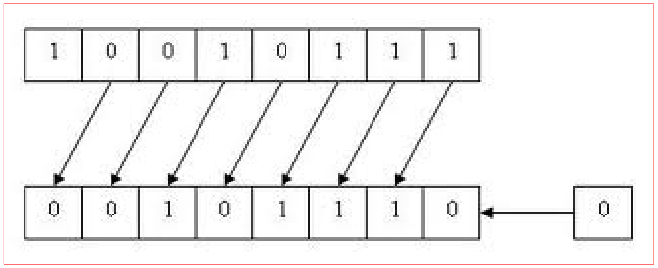
- Shift left: move all the bits in a word to the left
  - Example

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

Shift left by 4

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

  - Note: 144 = 9x16



- General rule:
  - Shift X left by 1: Y=2*X
  - Shift X left by 2: Y=4*X
  - Shift X left by n (n<32): $Y=2^n*X$

Example

- Assembly instruction:

```
sll  $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits
```

- Corresponding machine instruction

Shift amount

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

R-format

$s0

$t2

Shift left logical (sll)

- Shift right logical is also supported by MIPS.
  - The instruction format for srl is very similar to that of sll
  - funct=2

srl $t2, $s0, 1

Quick exercise

- Which of the following is the correct machine code for

srl $t2, $s0, 1

Note: $t2 <-> $10
$s0 <-> $16

- A.

op	rs	rt	rd	shamt	funct
0	0	16	10	1	0

- B.

op	rs	rt	rd	shamt	funct
0	0	10	16	1	0

- C.

op	rs	rt	rd	shamt	funct
0	0	16	10	1	2

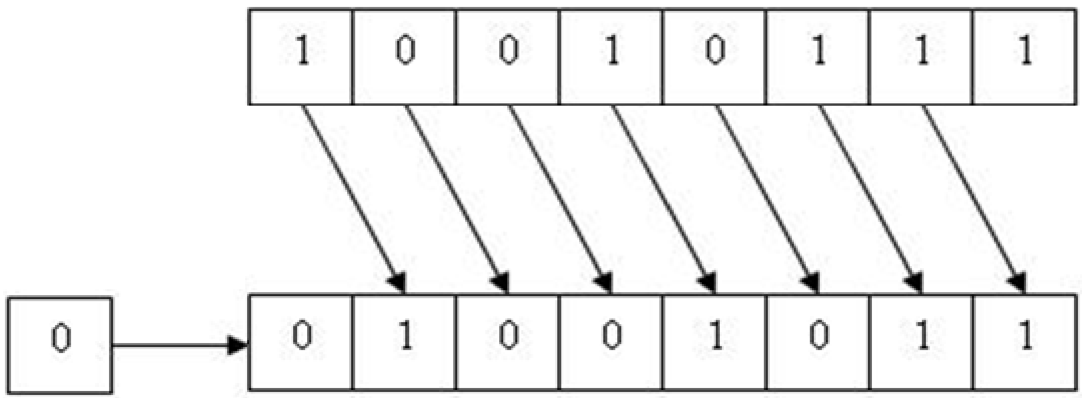
- D.

op	rs	rt	rd	shamt	funct
0	0	10	16	1	2

Shift right logical

- Move all bits in a word to the right

srl $t2, $s0, 1



$s0: 151

$t2: 75

- What would be the value if $s0 is shifted right by 2?
  - A. 30
  - B. 37
  - C. 42
  - D. 45

Logic AND

- AND: a logic bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in both operands.

```
a :   1 1 0 0 0 0 1 1
b :   0 1 0 1 0 1 1 1   &
      ---------------
c :   0 1 0 0 0 0 1 1
```

- MIPS instruction

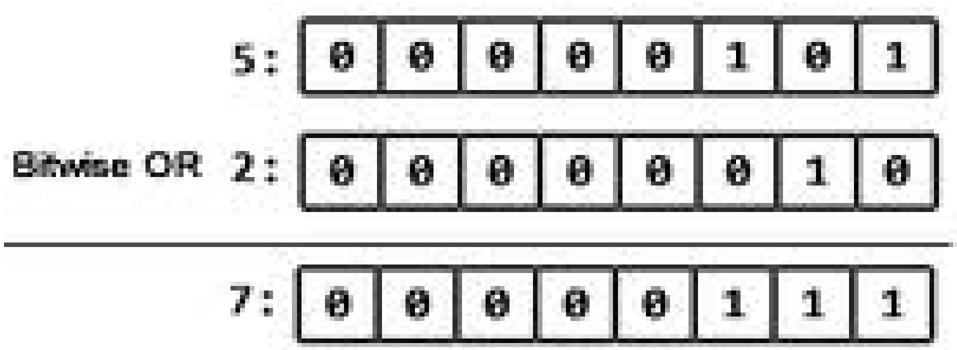
```
and $t0,$t1,$t2      # reg $t0 = reg $t1 & reg $t2
```

0	9	10	8	0	36
op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

AND

Logic OR

- OR: a logic bit-by-bit operation with operands that calculates a 1 if there is a 1 in either operand.



- MIPS instruction

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

OR

0	9	10	8	0	37
op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

Logic NOR

- NOR: a logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands.
  - funct=39

X	Y	O
0	0	1
0	1	0
1	0	0
1	1	0

Quick exercise

- How to invert every bit in $t1 with an NOR operation?
  - A. nor $t1, $t1, $zero
  - B. nor $t1, $t1, $0
  - C. nor $t1, $t1, $t1
  - D. None of the above

Immediate version

- Constants are useful in AND and OR logic operations
- MIPS provides immediate and (andi) and immediate or (ori)
  - andi: opcode="001100"=12_{ten}
  - ori: opcode="001101" =13_{ten}

- Question: which operations can isolate a field in a word?
  - A. and (or andi)
  - B. A shift left followed by a shift right

0	1	1	0	1	0	1	1

Code for isolating a field

- Code 1:

  andi $s1, $s1, 28 # 28="00011100"

  Result:

0	0	0	0	1	0	0	0

- Code 2:

  sll $s1, $s1, 3 # shift $s1 to the left by 3 bits

0	1	0	1	1	0	0	0

  srl $s1, $s1, 5 # shift $s1 to the right by 5 bits

  Result:

0	0	0	0	0	0	1	0

Instructions for making decisions

- What distinguishes a computer from a simple calculator is its ability to make decisions

- Conditional branch

  - Branch if equal:

```
beq register1, register2, L1
```

Offset with respect to the current location of execution

  - Branch if not equal:

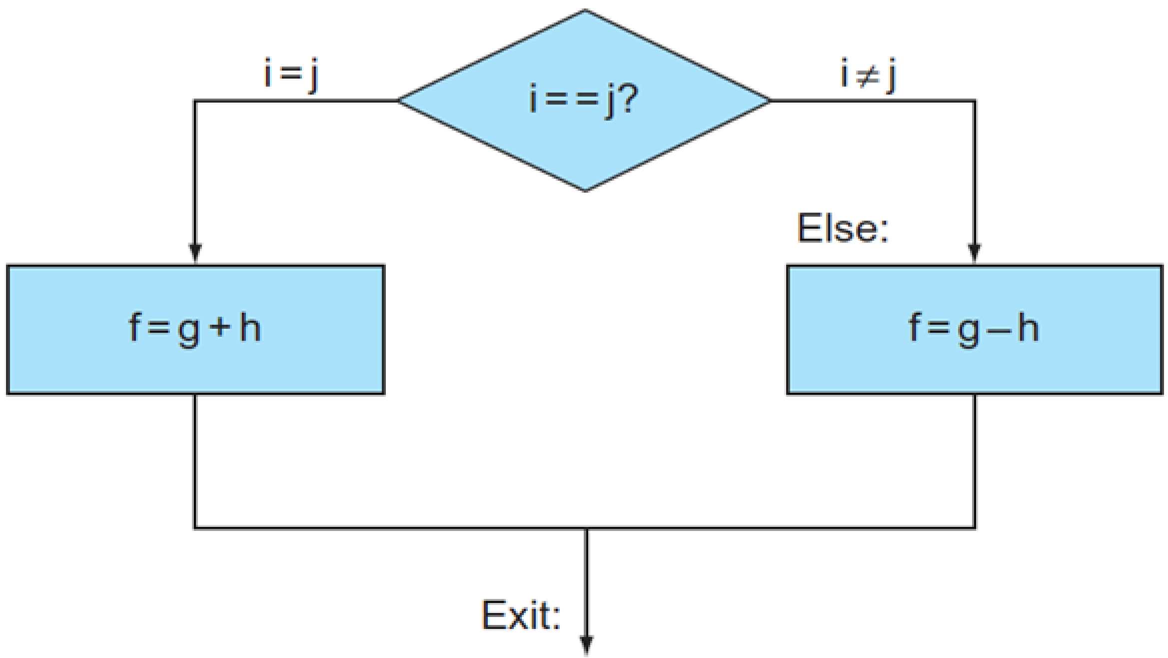
```
bne register1, register2, L1
```

Example

- Compiling *if-then-else* into conditional branches

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

- Draw the flow chart:



Build the MIPS code

- Determine register allocation
  - i: $s3
  - j: $s4
  - f: $s0
  - g: $s1
  - h: $s2

- Write the assembly code

```
      bne $s3,$s4,Else     # go to Else if i ≠ j
      add $s0,$s1,$s2      # f = g + h (skipped if i ≠ j)
      j Exit     # go to Exit
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```

Example

- Compile a *while* loop in c

```
while (save[i] == k)
      i += 1;
```

- Assume that
  - i: $s3
  - k: $s5
  - Base of array save: $s6

```
Loop: sll  $t1,$s3,2      # Temp reg $t1 = i * 4
      add $t1,$t1,$s6     # $t1 = address of save[i]
      lw  $t0,0($t1)      # Temp reg $t0 = save[i]
            ?             # go to Exit if save[i] ≠ k
      addi  $s3,$s3,1     # i = i + 1
      j     Loop          # go to Loop
   Exit:
```

- Which one below is the missing statement?
  - A. beq $s5, $t0, Exit
  - B. beq $s5, $t1, Exit
  - C. bne $s5, $t0, Exit
  - D. bne $s5, $t1, Exit

```
while (save[i] == k)
      i += 1;
```

- i: $s3
- k: $s5
- Base of array save: $s6

Understand hardware/software interface

- Compilers frequently create branches and labels where they do not appear in the programming language

- Basic block: a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning.
  - One of the first early phases of compilation is breaking the program into basic blocks

Instructions for comparisons

- Set on less than (slt)

```
slt    $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4
```

- Immediate version

```
slti   $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

- MIPS compilers use slt, slti, beq, bne, and $zero to create all relative conditions

- For simplicity, MIPS architecture does not include branch on less than.

A quick exercise

- Convert the c statement into MIPS code

```
if($s1 < $s2) goto L1;
```

- MIPS code:
  - A.
    ```
    slt $t0, $s1, $s2
    bne $t0, $zero, L1
    ```
  - B.
    ```
    slt $t0, $s2, $s1
    beq $t0, $zero, L1
    ```

Beware of signs

- slt and slti work with signed integers
  - Suppose that $s0 has the binary number

    -1

    1111 1111 1111 1111 1111 1111 1111 1111_{two}
  - $s1 has the binary number

    +1

    0000 0000 0000 0000 0000 0000 0000 0001_{two}

- What are the values of $t0 and $t1 after the two instructions:

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

Bounds check

- Suppose that your program needs to jump to a place called IndexOutOfBounds if
  - $s1 is greater than or equal to a positive bound $t2
  - Or $s1 is negative

- By treating the value of $s1 as unsigned numbers, we can use one condition to check both:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

  - Can you explain why?

Check yourself

- C has many statements for decisions and loops, while MIPS only has a few. Which of the following do or do not explain this imbalance? Why?
  - A. More decision statements make code easier to read and understand.
  - B. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
  - C. More decision statements mean fewer lines of code, which generally reduces coding time.
  - D. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

Write efficient programs

- It is important to understand how the computer sees a high-level language program such as a program written in c.

- Sometime we can make the program run more than 10 times faster, just by understanding how the computer sees things

- Knowledge of instruction set architecture helps you write efficient programs

Example

- A simple c program:

```
int func (int a, int b) {
      int sum = a + b;
      if ((a % 2) == 1 || (a % 2) == -1) //test odd
         sum++;
      return sum / 2;
}
```

1

20

20

2

Conditional branch 1

Conditional branch 1

20

- Approximate its efficiency:
  - \+ takes 1 clock cycle
  - % takes 20 clock cycles
  - / takes 20 clock cycles
  - == takes 1 clock cycle
  - Sum++ takes 2 clock cycles

65 clock cycles

Tips for optimization

- ++i is never slower than i++, always use ++i
- Modulo dominates, so let's focus on that
- Test for a odd or even
  - We only need to look at the last bit
  - 110010 is even and 110011 is odd
  - Bitwise AND looks per bit
    - a&1
    - Shows whether last bit is set
    - Works regardless of sign

Improvement

- Improved c program

```
int func (int a, int b) {
    int sum = a + b;
    if ((a & 1) == 1) //test odd
        ++sum;
    return sum / 2;
}
```

1

1

1

Conditional branch
1

20

- Approximate its efficiency:
  - \+ takes 1 clock cycle
  - & takes 1 clock cycle
  - == takes 1 clock cycle
  - ++sum takes 1 clock cycle
  - / takes 20 clock cycle

24 clock cycles

Further optimization tips

- Division dominates
- Division by multiple of 2 can be achieved through shift right logical
  - Divide sum by 2 -> shift sum to the right by 1
- Finding: division by x where x is a multiple of 2 can be reduced to right shift by Log_2x.
  - X=2 -> $Log_2X=1$ -> right shift by 1
  - X=4 -> $Log_2X=2$ -> right shift by 2
  - X=8 -> $Log_2X=3$ -> right shift by 3
  - …

Further improvement

- Further optimized c program

```
int func (int a, int b) {
    int sum = a + b;
    if ((a & 1) == 1) //test odd
        ++sum;
    return sum >> 1;
}
```

1

1

1

Conditional branch
1

1

- Approximate its efficiency:
  - \+ takes 1 clock cycle
  - & takes 1 clock cycle
  - == takes 1 clock cycle
  - ++sum takes 1 clock cycle
  - \>> takes 1 clock cycle

5 clock cycles

Check yourself

- Why does C provide two sets of operators for the logic AND operation, i.e. && and &?

Check yourself

- Why does C provide two sets of operators for the logic AND operation, i.e. && and &?

- Answer:
  - && and & are used for different purposes and therefore are implemented differently
  - & is implemented through the logic and operation in MIPS
  - && is implemented through the conditional branch instructions.