Computer Architecture (Practical Class)
C and Assembly: Bit-level Operations

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2021/2022

Bit-level Operations

Bit-level operations modify one or more bits at a time and are used to manipulate binary numbers (stored in memory or registers)

- Are very efficient operations, directly supported by the processor
- Can be used to:
  - Extract information from groups of bits
  - Perform multiplications and divisions by powers of two several times faster
  - Apply an operation to multiple data within a single variable (if the variables has groups of bits representing different data)

Two major groups of bit-level operations

- Boolean logic
  - Each bit is compared individually using the logic function specified
  - Logic functions: AND, OR, XOR, NOT
- Bit movements
  - Shift bits left/right (multiply/divide by powers of two)
  - Rotate bits left or right

Boolean Logic Operations: AND

`AND` (the " & " operator in C)

X	Y	AND
0	0	0
0	1	0
1	0	0
1	1	1

- Usage: `AND` *origin*, *destination*
  - operation: *destination* = *destination* AND *origin* (the result is placed in *destination*)
  - *origin* can be a memory address, a constant value or a register
  - *destination* can be a memory address or a register
  - the `AND` instruction can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

Boolean Logic Operations: OR

`OR` (the " | " operator in C)

X	Y	OR
0	0	0
0	1	1
1	0	1
1	1	1

- Usage: `OR` *origin* , *destination*
  - operation: *destination* = *destination* OR *origin* (the result is placed in *destination*)
  - *origin* can be a memory address, a constant value or a register
  - *destination* can be a memory address or a register
  - the `OR` instruction can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

Boolean Logic Operations: XOR

`XOR` (the " ^ " operator in C)

X	Y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

- Usage: `XOR` *origin* , *destination*
  - operation: *destination* = *destination* XOR *origin* (the result is placed in *destination*)
  - *origin* can be a memory address, a constant value or a register
  - *destination* can be a memory address or a register
  - the `XOR` instruction can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

`NOT` (the " ~ " operator in C)

X	NOT
0	1
1	0

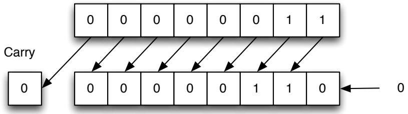
- Usage: `NOT` *destination*
  - operation: *destination* = NOT *destination* (the result is placed in *destination*)
  - *destination* can be a memory address or a register
  - the `NOT` instruction can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

Important notes

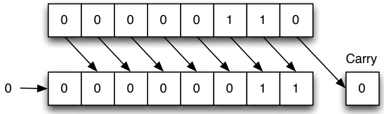
- The Assembly instruction `NEG` changes the sign of an integer
- The logic C operator "`!`" is not a bit-level operator; it considers the entire number as a logical value

y	NOT y	NEG y	!y
0	-1	0	1
-1	0	1	0
1	-2	-1	0

- Shifting bits left and right is a very easy operation to implement on a processor. It is often used as a fast way to implement multiplication/division
- Consider a binary number:
    - Shifting a digit left (entering a zero) corresponds to a multiplication by 2



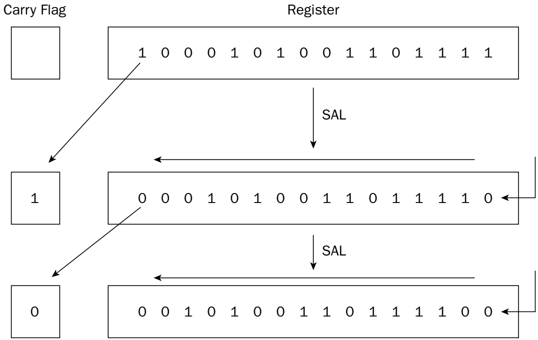
    - Sifting a digit right (losing the rightmost digit) corresponds to a division by 2



- This is true for any base *b* (shifting a digit left/right corresponds to multiply/divide by *b*)

`SHL/SAL` (the " << " operator in C)

- SHL and SAL are equivalent operations
  - They exist for consistency with the right shift (we will see why in a moment)
- Shifts bits to the left; zeros enter on the right and the last bit to exit left goes to the carry flag



`SHL/SAL` - Three formats

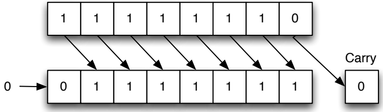
- `SHL` *destination* (or `SAL` *destination*)
  - Shifts the *destination* value left one position (equivalent to $destination *= 2$)
- `SHL` %cl, *destination* (or `SAL` %cl, *destination*)
  - Shifts the *destination* value left by the number of times specified in the CL register (equivalent to $destination *= 2^{CL}$)
- `SHL` $n, *destination* (or `SAL` $n, *destination*)
  - Shifts the *destination* value left by the number of times specified by a constant value n (equivalent to $destination *= 2^{n}$)

- In all formats, *destination* can be a memory address or a register
- The `SHL/SAL` instructions can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

Shifting Bits Right (1/5)

Notes about signed numbers

- Performing a right shift on a signed integer value may adversely affect the sign of the integer
- When shifted to the right, a negative number will lose its sign if we zero-fill the leading bits

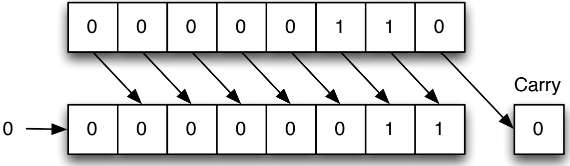


- To solve this problem there is a distinction between the right-shift instructions:
  - SHR - the **logic shift**t to the right *does not preserve the signal*;
  - SAR - the **arithmetic shift** to the right *preserves the signal*.

Shifting Bits Right (2/5)

SHR (the " >> " operator in C, when applied to unsigned integers)

- Logic shift to the right
- Shifts bits to the right; zeros enter on the left and the last bit to exit to the right goes to the carry flag (similar to the left shift)
- Therefore, *does not preserve the signal* of the number



SHR - Three formats

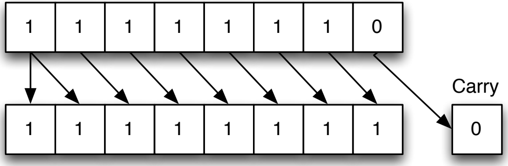
- `SHR` *destination*
  - Shifts the *destination* value right one position (equivalent to *destination* $/= 2$).
- `SHR` %cl, *destination*
  - Shifts the *destination* value right by the number of times specified in the CL register (equivalent to *destination* $/= 2^{CL}$).
- `SHR` $n, *destination*
  - Shifts the *destination* value right by the number of times specified by a constant value n (equivalent to *destination* $/= 2^{n}$)

- In all formats, *destination* can be a memory address or a register
- The `SHR` instruction can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

Shifting Bits Right (4/5)

SAR (the " >> " operator in C, when applied to signed integers)

- Arithmetic shift to the right
- Shifts bits to the right; either clears or sets the bits entered on the left, according to the sign of the integer. The last bit that exits to the right goes to the carry flag
- Therefore, *preserves the signal* of the number



Shifting Bits Right (5/5)

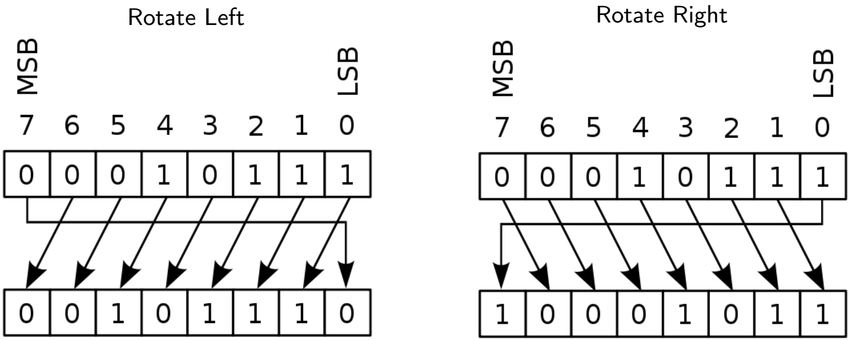
SAR - Three formats (similar to SHR):

- `SAR` *destination*
- `SAR` %cl, *destination*
- `SAR` $n, *destination*

- In all formats, *destination* can be a memory address or a register
- The `SAR` instruction can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

ROL/ROR (no equivalent operation in C)

- ROL - Bit rotation to the left
- ROR - Bit rotation to the right
- Perform just like the shift instructions, except the overflow bits are pushed back into the other end of the value instead of being dropped.



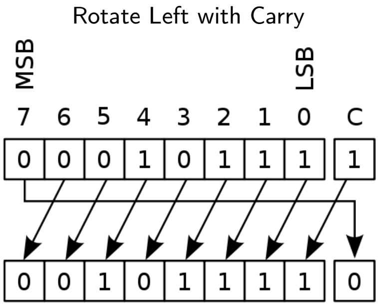
ROL/ROR - Three formats (similar to shift instructions):

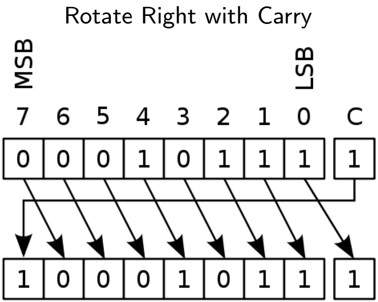
- `RO{L/R}` *destination*
- `RO{L/R}` %cl, *destination*
- `RO{L/R}` $n, *destination*

- In all formats, *destination* can be a memory address or a register
- The `ROL/ROR` instructions can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

`RCL/RCR` (no equivalent operation in C)

- RCL - Bit rotation to the left *with carry*
- RCR - Bit rotation to the right *with carry*
- Perform bit rotation, but the first entering bit comes from carry and overflow bits go to carry.





RCL/RCR - Three formats (similar to ROL/ROR):

- `RC{L/R}` *destination*
- `RC{L/R}` %cl, *destination*
- `RC{L/R}` $n, *destination*

- In all formats, *destination* can be a memory address or a register
- The `RCL/RCR` instructions can operate on numbers of 8(b), 16(w), 32(l), or 64(q) bits

Bit Masks (1/4)

Bit masks

- One common use of bit-level operations is to implement *masking* operations
- A *mask* is a bit pattern that indicates a selected set of bits within a number

For example, assume that `%al` has a binary value of 00101100

- How can we get the value of the 4 least significant bits?

To solve the problem we need to:

1. Determine the logic operator to use
2. Determine the mask, based on the selected operator

```
movb $0b00101100, %al
movb $0b00001111, %ah
andb %ah, %al          # %al = 0b00001100
```

Bit Masks (2/4)

Assume now we want to replace the 4 least significant bits of %al for the 4 least significant bits of %cl We need to perform three steps:

1. Set the 4 least significant bits of %al to zero using the `and` operation
2. Set the 4 most significant bits of %cl to zero using the `and` operation
3. Replace the bits in %al with the bits in %cl using the `or` operation

```
movb $0b00101100, %al
movb $0b01000011, %cl

movb $0b11110000, %ah
andb %ah, %al            # %al = 0b00100000
notb %ah                 # %ah = 0b00001111 (inverts the mask)
andb %ah, %cl            # %cl = 0b00000011

orb  %cl, %al            # %al = 0b00100011
```

Bit Masks (3/4)

- The `xor` operation can be used to set a number to zero or compare two numbers
  - The result of a `xor` operation between two equal numbers is zero

```
xorl %ebx, %ecx
jz is_equal
```

- The `xor` operation is also used to invert the bits of a register
  - Placing 0/1 on the mask to keep/invert the original bit

```
movb  $0b00101100, %al
movb  $0b00001111, %ah
xorb  %ah, %al            # %al = 0b00100011
```

Bit Masks - Summary (4/4)

- The **AND** operator can be used to:
  - Set bits to zero - use a bit mask with those bits set to zero
  - Get the value of a few bits - use a bit mask with those bits set to one
  - "Round" a number to a power of 2 (2^x) - use a bit mask with x least significant bits to zero
  - Get the remainder of the division by a power of two (2^x) - use a bit mask with x least significant bits to one
- The **OR** operator can be used to:
  - Set bits to one - use a bit mask with those bits set to one
  - Join the bits of two numbers - perform an **or** between the two numbers
- The **XOR** operator can be used to:
  - Check if two numbers are equal - if a **xor** between the two numbers is zero
  - Invert bits - use a bit mask with those bits set to one

```c
#define mask(n)         ((1<<(n))-1)
#define mask2(n1,n2)    (mask(n2-n1+1)<<(n1))

#include <stdio.h>
int main ()
{
  printf("%d\n", mask(2));
  printf("%d\n", mask(3));
  printf("%d\n", mask2(1,2));
  printf("%d\n", mask2(1,3));
  return 0;
}
```

- The above code prints...
  - A. the numbers 2, 3, 2, 3
  - B. the numbers 4, 8, 6, 9
  - C. the numbers 3, 7, 6, 14
  - D. None of the above.

Practice problem

Implement in C and in Assembly the function:

```
void sum_bytes(int a, int b, char *sum)
```

- Sums the most significant byte of the integer `a` with the least significant byte of `b`
- Writes the result in the memory address pointer by `sum`

Practice problem

Function sum_bytes in C

```
void sum_bytes(int a, int b, char *sum){
  char msb_a = (a & 0xFF000000) >> 24;
  char lsb_b = b & 0x000000FF;

  *sum = msb_a + lsb_b;
}
```

Function sum_bytes in Assembly

```
sum_bytes:
  # a in %edi, b in %esi, *sum in %rdx

  shrl $24, %edi        # MSB in %dil
  addb %dil, %sil
  movb %sil, (%rdx)     # *sum = %dil + %sil
  ret
```

Practice problem

Given the following C code to multiply two numbers using logic and bit movement operations, implement the equivalent in Assembly

Function multiply in C

```c
int multiply(int a, int b){
  int res = 0;

  while (b != 0){
    if ((b & 1) == 1)
      res = res + a;
    a = a << 1;
    b = (unsigned int)b >> 1;
  }

  return res;
}
```

Function multiply in Assemby

```
#int mult(int a, int b)
mult:
    # a in %edi, b in %esi
    movl $0, %eax         # res = 0

loop_mult:
    cmpl $0, %esi         # while(b!=0)
    je end
    pushq %rsi            # save b in stack
    andl $1, %esi         # b = b & 1
    cmpl $1, %esi
    jne next
    addl %edi, %eax       # res = res + a
next:
    popq %rsi             # get b from stack
    shll %edi             # a << 1
    shrl %esi             # b >> 1
    jmp loop_mult
end:
    ret
```