# Bit Operations

# Bit usage

- A bit can mean one of a pair of characteristics
- True or false
- Male or female
- Bit fields can represent larger classes
    - There are 64 squares on a chess board, 6 bits could specify a position
    - The exponent field of a float can be represented using a number of bits.
    - We could use a 3 bit field to store a color from black, red, green, blue, yellow, cyan, purple and white

# Bit operations

- Individual bits have values 0 and 1
- There are instructions to perform bit operations
- Using 1 as true and 0 as false
  - 1 and 1 = 1, or in C/C++, `1 && 1 = 1`
  - 1 and 0 = 0, or in C/C++, `1 && 0 = 0`
  - 1 or 0 = 1, or in C/C++, `1 || 0 = 1`
- We are interested in operations on more bits
  - `10101000b & 11110000b = 10100000b`
  - `10101000b | 00001010b = 10101010b`
- These are called "bit-wise" operations
- We will not use bit operations on single bits, though we will be able to test/set/reset individual bits

# The Not operation

- C/C++ uses `!` for a logical not
- C/C++ uses `~` for a bit-wise not

```
!0 == 1
!1 == 0
~(false) == true
~(true) == false
~10101010b == 01010101b
~0xff00 == 0x00ff
!1000000 == 0  (non-zero integer is seen as true in c/c++)
~0== ?
~1== ?
```

# The Not operation

- C/C++ uses `!` for a logical not
- C/C++ uses `~` for a bit-wise not

```
!0 == 1
!1 == 0
~(false) == true
~(true) == false
~10101010b == 01010101b
~0xff00 == 0x00ff
!1000000 == 0  (non-zero integer is see as true in c/c++)
~0== -1
~1== -2
```

# The Not instruction

- The `not` instruction flips all the bits of a number - one's complement
- The `not` operator does not affect any flags
- There is only a single operand which is the source and destination
- For memory operands you must include a size prefix
- The sizes are `byte`, `word`, `dword` and `qword`

```
not     rax         ; invert all bits of rax
not     dword [x]   ; invert double word at x
not     byte [x]    ; invert a byte at x
```

# And operation

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

- C/C++ uses && for a logical and
- C/C++ uses & for a bit-wise and

```
11001100b & 00001111b == 00001100b
11001100b & 11110000b == 11000000b
0xabcdefab & 0xff == 0xab
0x0123456789abcdef & 0xff00ff00ff00ff00 == 0x010045008900cd00
```

- Bit-wise **and** is a bit selector

# And instruction

- The **and** instruction performs a bit-wise **and**
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Both destination and source cannot be in memory
- The sign flag and zero flag are set (or cleared)

# And Example

- We wish to extract bits 0-3 and store them in `rbx`

```
mov     rax, 0x12345678
mov     rbx, rax
and     rbx, 0xf            ; rbx has the low nibble 0x8
```

- We wish to extract bits 4-7 and store them in `rax`

```
mov     rdx, 0              ; prepare to divide
mov     rcx, 16             ; by 16
idiv    rcx                 ; rax has 0x1234567
and     rax, 0xf            ; rax has the nibble 0x7
```

# Or operation

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

- C/C++ uses `||` for a logical or
- C/C++ uses `|` for a bit-wise or

```
11001100b | 00001111b == 11001111b
11001100b | 11110000b == 11111100b
0xabcdefab | 0xff == 0xabcdefff
0x0123456789abcdef | 0xff00ff00ff00ff00 == 0xff23ff67ffabffef
```

- **or** is a bit setter

# Or instruction

- The **or** instruction performs a bit-wise **or**
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Both destination and source cannot be in memory
- The sign flag and zero flag are set (or cleared)

# Or example

- Make a number odd

```
mov     rax, 0x1124
or      rax, 1          ; make the number odd
```

- Set bits 8-15.

```
mov     rax, 0x1000
or      rax, 0xff00     ; set bits 15-8
```

How would you make a number even?

# Exclusive or operation

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

- C/C++ uses ^ for exclusive or

```
00010001b ^ 00000001b == 00010000b
01010101b ^ 11111111b == 10101010b
01110111b ^ 00001111b == 01111000b
0xaaaaaaaa ^ 0xffffffff == 0x55555555
0x12345678 ^ 0x12345678 == 0x00000000
```

- Exclusive or is a bit flipper

# Exclusive or instruction

- The **xor** instruction performs a bit-wise **exclusive or**
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Both destination and source cannot be in memory
- The sign flag and zero flag are set (or cleared)
- `mov rax, 0` uses 7 bytes
- `xor rax, rax` uses 3 bytes
- `xor eax, eax` uses 2 bytes

# Exclusive or example

- Zero out a register.

```
mov     rax, 0x12345678
xor     eax, eax                    ; set rax to 0
```

- Flip bits 0-3

```
mov     rax, 0x1234
xor     rax, 0xf                    ; change to 0x123b
```

- Swap the value in two registers

```
xor rax, rbx
xor rbx, rax
xor rax, rbx
```

# Shift operations

- C/C++ uses << for shift left and >> for shift right
- Shifting left introduces low order 0 bits
- Shifting right propagates the sign bit in C++ for signed integers
- Shifting right introduces 0 bits in C++ for unsigned integers
- Shifting left is like multiplying by a power of 2
- Shifting right is like dividing by a power of 2

```
101010b >> 3 == 101b
111111b << 2 == 11111100b
125 << 2 == 500   (125=>1111101<<2==111110100=>500)
0xabcd >> 4 == 0xabc
```

# Shift instructions

- Shift left: `shl`
- Shift right: `shr`
- Shift arithmetic left: `sal`
- Shift arithmetic right: `sar`
- `shl` and `sal` are the same
- `shr` introduces 0 bits on the top end
- `sar` propagates the sign bit
- All the shifts use 2 operands
  - A destination register or memory
  - In immediate number of bits to shift
    - Or from old 16 bit asm the `cl` register can be used
- The sign and zero flags are set (or cleared)
- The carry flag is set to the last bit shifted out

# Extracting a bit field

- There are at least 2 ways to extract a bit field
- Shift right followed by an **And** operation
  - To extract bits $k$ to $m$ (inclusive) with $m \geq k$, shift right $k$ bits
  - And this value with a mask of $m - k + 1$ bits all set to 1

# Extracting a bit field with shift/and

Need to extract bits 9–3

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Shift right 3 bits

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

And with 0x7f

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

# Extracting a bit field

- The second way
- Shift left and then right
    - Shift left until bit $m$ is the highest bit
    - With 64 bit registers, shift left $63 - m$ bits
    - Shift right to get original bit $k$ in position 0
    - With 64 bit registers, shift right $63 - (m - k)$ bits

# Extracting a bit field with shift/shift

Need to extract bits 9–3

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Shift left 6 bits

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Shift right 9 bits

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

# Rotate instructions

- The `ror` instruction rotates the bits of a register or memory location to the right
  - Values from the low end start filling in the top bits
- The `rol` instruction rotates left
  - Values from the top end of the value start filling in the low order bits
- These are 2 operand instructions like the shift instructions
- The first operand is the source to rotate (and the destination)
- The second operand is the number of bits to rotate
- The second operand is either an immediate value or `cl`
- Assuming 16 bit rotates

```
1 ror 2 = 0100000000000000b
0xabcd ror 4 = 0xdabc
0x4321 rol 4 = 0x3214
```

# Filling a field

- There are at least 2 ways of filling in a field (with existing values)
- Use shifts and a mask.
  - Working with a 64 bit register, filling bits $k$ to $m$ (inclusive)
  - Prepare a mask of $m - k + 1$ bits all 1
  - Shift the new value and the mask left $k$ bits
  - Negate the mask
  - And the old value and the mask
  - Or in the new value for the field

# Filling a field 1

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | We want to replace bits 6-3 | | | | | | | | | | | |
| Original | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | | | | | | |
| | | | | | with | | | | | | | | | | | |
| Value | | | | | | | | | | | | | 1 | 1 | 0 | 1 |
| | create mask of length 6-3+1=4 | | | | | | | | | | | | | | | |
| Mask | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | Shift both by k=3 | | | | | | | | | | | | | | | |
| Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| | | | | | | | | | | | | | | | | |
| Mask | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| | Negate the mask | | | | | | | | | | | | | | | |
| Mask | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| | And with original | | | | | | | | | | | | | | | |
| Original | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| Mask | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | or value with result | | | | | | | | | | | | | | | |
| | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

# Filling a field

- Second method
- Use rotate and shift instructions and or in new value
  - Rotate the register right $k$ bits
  - Shift the register right $m - k + 1$ bits
  - Shift the register left $m - k + 1$ bits
  - Or in the new value
  - Rotate the register left $k$ bits

# Filling a field 2

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | We want to replace bits 6-3 | | | | | | | | | | | |
| Original | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | | | | | | |
| | | | | | with | | | | | | | | | | | |
| Value | | | | | | | | | | | | | 1 | 1 | 0 | 1 |
| | Rotate original right by k=3 | | | | | | | | | | | | | | | |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| | Shift right by m-k+1=6-3+1=4 | | | | | | | | | | | | | | | |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | shift left by 4 | | | | | | | | | | | | | | | |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | or with value | | | | | | | | | | | | | | | |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| | Rotate left by k=3 | | | | | | | | | | | | | | | |
| | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

# Bit testing and setting

- It takes a few instructions to extract or set bit fields
- The same technique could be used to test or set single bits
- It can be more efficient to use special instructions operating on a single bit
  - The `bt` instruction tests a bit
    - the CF flag gets set to the value of the tested bit
    - we can gain access to the flag using `setc cl` (for example)
  - `bts` tests a bit and sets it
    - tested bit gets set to 1
  - `btr` tests a bit and resets it
    - tested bit gets set to 0
  - `btc` tests a bit and flips it
    - tested bit gets complemented
- These are all 2 operand instructions
- The first operand is a register or memory location
- The second is the bit to work on, either an immediate value or a register

# Bit testing and setting example

- Checking if a number is odd

```
mov  rax, 101
bt   rax, 0
setc dl ;  1 will be stored in dl, i.e the number is odd
```

- Setting the 7th and 33rd bit of the qword A in memory to 1

```
bts qword [A],  7
bts qword [A],  33
```