Introduction to x86

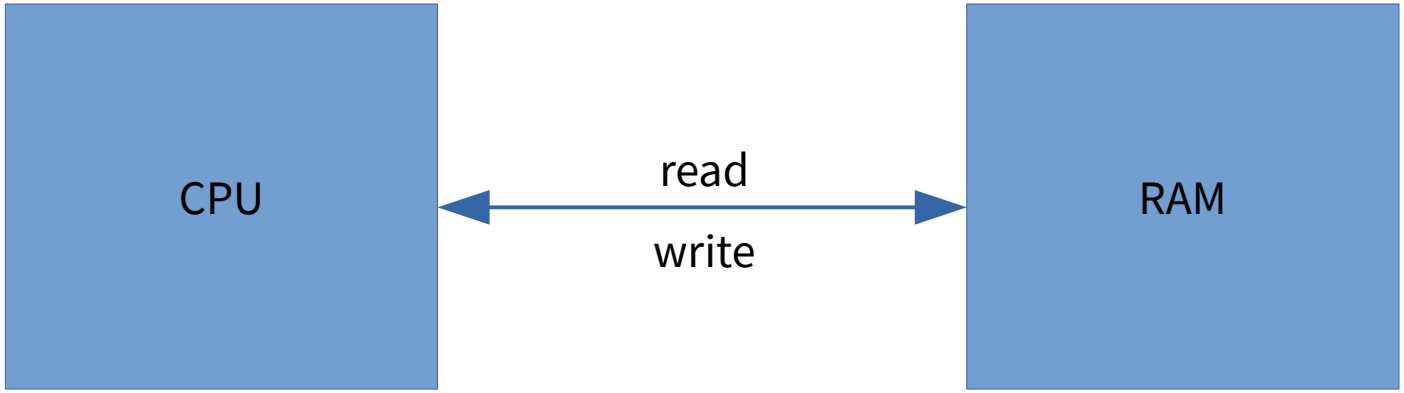
Ivan Sorokin

Computer Model

A real computer is a complicated piece of hardware with many intricate details. For teaching purposes we will leave out some unnecessary details. Initially we will discuss a simplified model suitable for teaching. Later we will refine our model to match real hardware more closely.

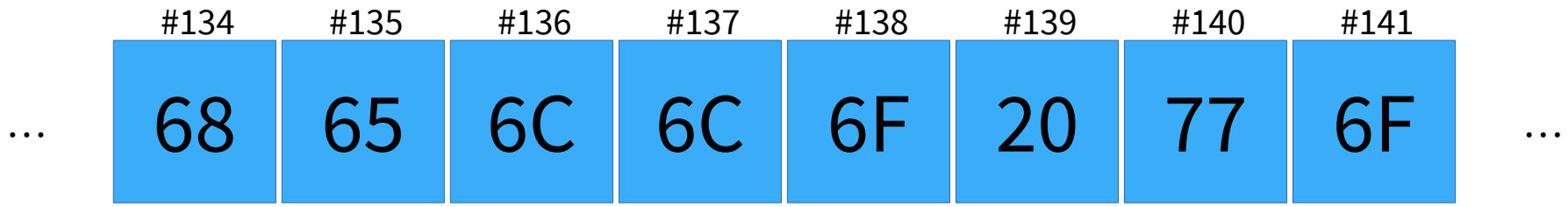
Computer Model

In a (highly) simplified model a computer consists of two components a CPU and RAM



RAM

RAM (Random Access Memory) is a numbered set of cells.

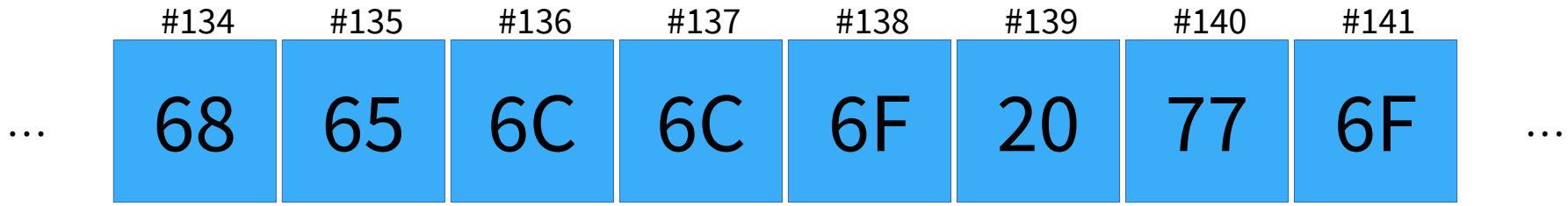


Numbered means that each cell has a number assigned to it.

The total number of cells determines the amount of RAM. As of 2016 computers typically have 8GB-32GB of RAM installed.

(TODO) In our model we will assume that cells are number from 0 to N. This is not the case in real world, where valid ranges can be non-continous.

RAM



RAM supports two operations: read and write.

- write, given a cell index and a value, changes the content of the specified cell to the specified value. Cell retain its content till the next write to the same cell
- read, given a cell index, retrieves the content of the specified cell

The index of a cell is called an address.

A cell can be modified only as a whole e.g. individual bits in a cell can not be modified independently.

RAM

In our model we will assume cell size to be 1 byte.

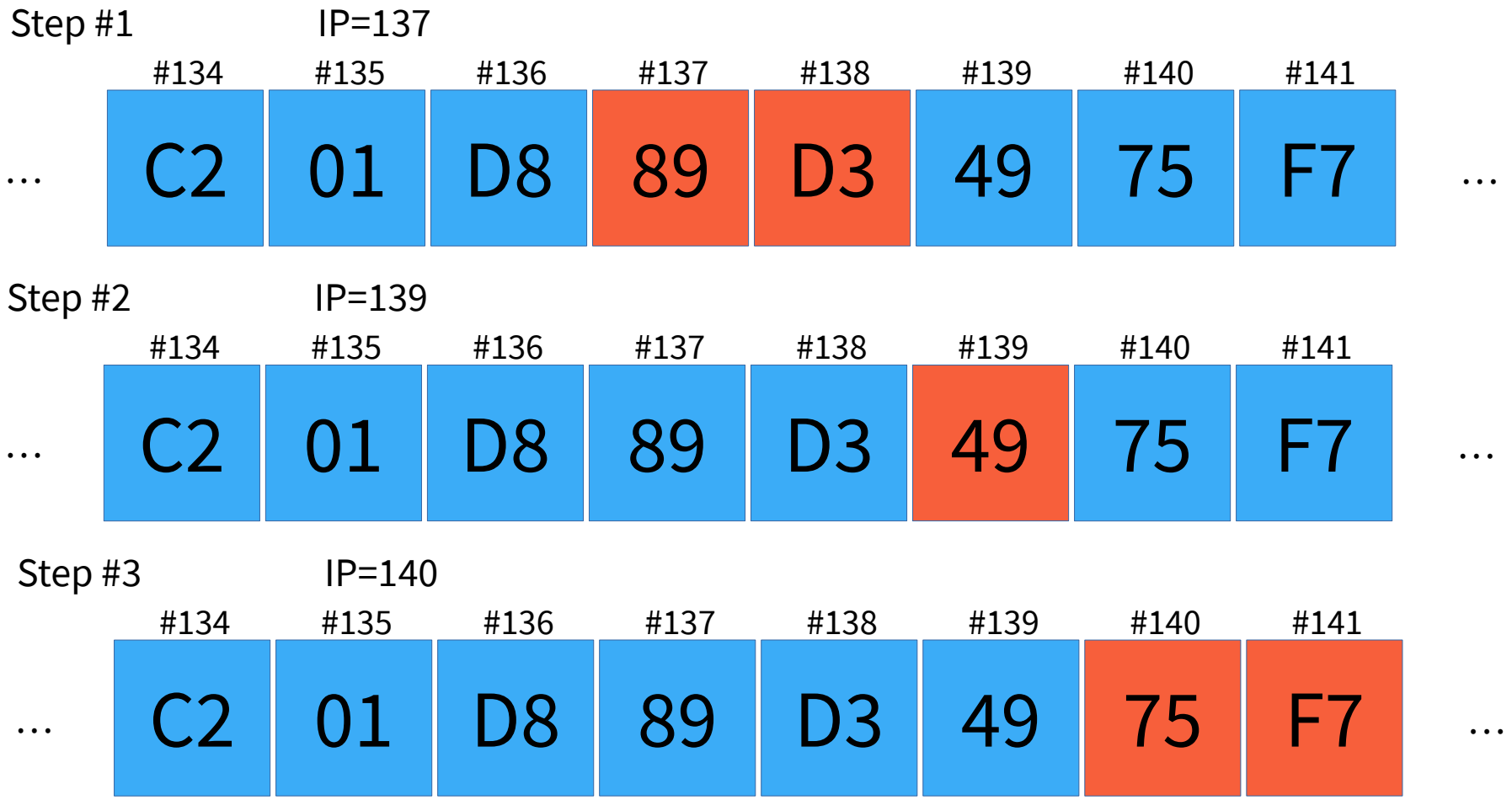
(sidenote) In the real world, data between a CPU and RAM is never transfered in bytes, as the overhead of transfering individual bytes gets prohibitely large. Modern RAM has a single addressable unit 64 bytes long which is of the same size as a cache line of modern CPUs. As the CPU maintains an illusion that memory can be byte-addressable we will ignore this detail for now.

CPU

A CPU executes programs.

A CPU keeps an internal number called register IP (instruction pointer). This register holds the address of the next instruction to be executed. On each step it reads a byte at address IP and possibly several following bytes. Each sequence of bytes is called an instruction and has a meaning assigned. CPU executes the instruction then add the length of the command to the register IP so the next instruction will be executed on the next step.

CPU



This process repeats billions times a second. Modern CPUs are able to execute up to 12 billion instructions a second.

CPU

For convenience instructions are typically written not in their memory encoding, but using a human-readable mnemonics. E.g.

```
89 C2          mov dx,ax
01 D8          add ax,bx
89 D3          mov bx,dx
49             dec cx
75 F7          jnz mylabel
```

The language of these mnemonics is called Assembly Language.

CPU

In addition to register IP, x86 CPU has 8 so-called GPRs (general purpose registers). Their names are:

AX, CX, DX, BX, SP, BP, SI, DI

These registers are 16-bit wide.

A register is a (very fast) memory cell located in a CPU. Most arithmetic operations operate on GPRs. GPRs are commonly used to keep intermediate results of computation.

Instruction MOV

The simplest and one of the most commonly used insturuction on x86 is MOV. MOV has two arguments source and destination. It copies the value from source to destination. Destination can be a register and source can be another register or an immediate value.

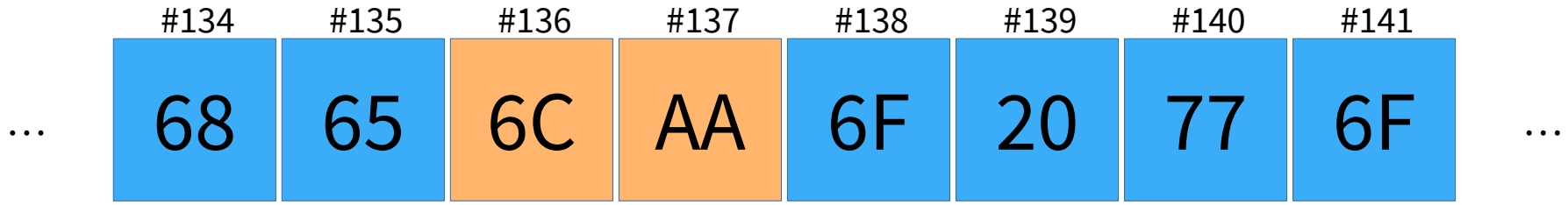
```
                MOV     dst, src        ; dst = src
B8 05 00        MOV     AX, 5           ; AX = 5
B9 0A 00        MOV     CX, 10          ; CX = 10
89 C8           MOV     AX, CX          ; AX = CX
89 D0           MOV     AX, DX          ; AX = DX
89 CA           MOV     DX, CX          ; DX = CX
```

Instruction MOV

MOV can be used to move values to/from memory. Brackets are used to refer to memory location.

```
; read 10th memory cell to register AX
A1 0A 00    MOV AX, [10]
; read the memory cell with index BX to AX
8B 07       MOV AX, [BX]
; write AX to the memory cell with index BX
89 07       MOV [BX], AX
```

Endian




```
MOV AX, [136]
```

What value will be stored in AX?

- `43628 (0xAA6C)` — little endian (x86)
- `27818 (0x6CAA)` — big endian

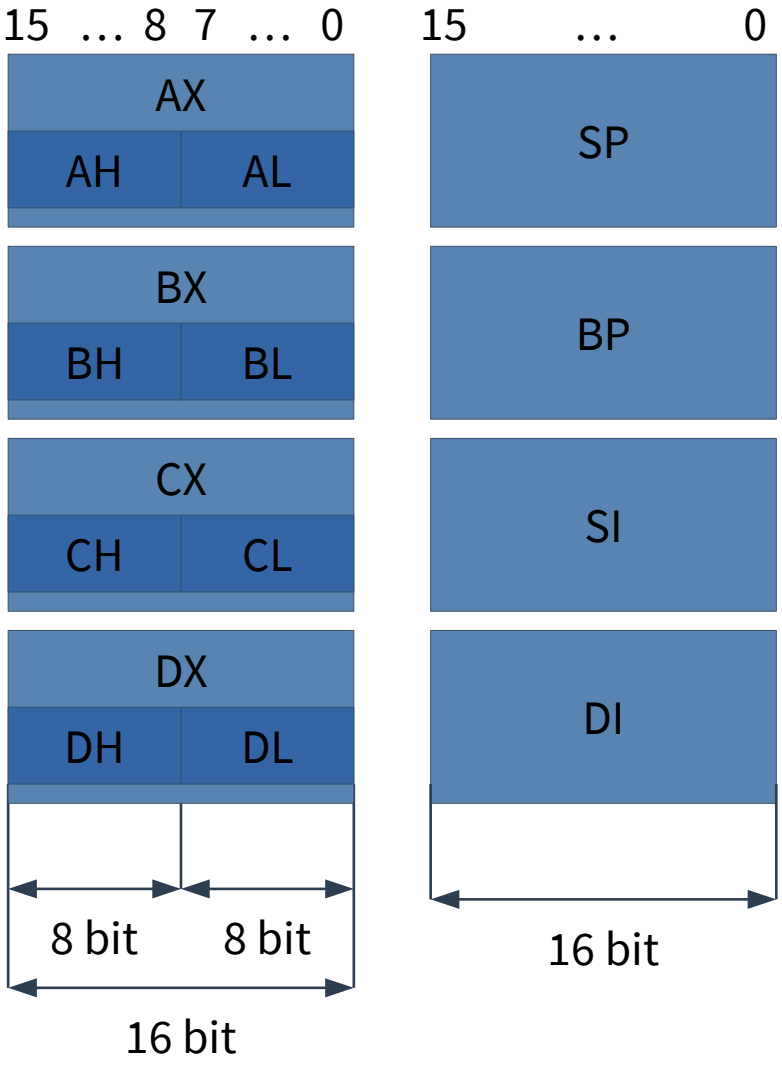
8-bit memory operations

One can read a single byte of memory by using 8-bit registers (AL, AH, BL, BH, CL, CH, DL, DH):

```
MOV AL, [136]
```

108 (0x6C) will be stored in **AL**.

Registers 16-bit



Instruction MOV

Not all combinations of sources and distinations are allowed. For example a single MOV instruction can not move data from memory to memory.

```
$ cat 1.asm
mov [ax], [bx]
$ nasm 1.asm
1.asm:1: error: invalid combination of opcode and operands
```

Instruction MOV

A set of valid combinations of sources and destinations was expanding over time. On modern CPUs it includes:

```
MOV reg, reg
MOV reg, imm
MOV reg, [imm]
MOV reg, [reg]
MOV [reg], reg
MOV [reg], imm
MOV [imm], reg
MOV [imm], imm
```

Basic Arithmetic Instructions

A set of basic arithmetic instructions includes instructions: ADD, SUB, AND, OR, XOR

```
; ADD writes to the destination the sum of the
; source and the destination
01 C8          ADD AX, CX       ; AX = AX + CX
; SUB writes the difference, ditto AND, OR, XOR
29 C8          SUB AX, CX       ; AX = AX - CX
21 C8          AND AX, CX       ; AX = AX & CX
09 C8          OR  AX, CX       ; AX = AX | CX
31 C8          XOR AX, CX       ; AX = AX ^ CX
```

Basic Arithmetic Instructions

ADD, SUB, AND, OR, XOR supports the same source/destination combinations as MOV:

```
21 D8           AND AX, BX
83 E0 05        AND AX, 5
23 06 05 00     AND AX, [5]
23 07           AND AX, [BX]
21 07           AND [BX], AX
```

INC, DEC

INC (increment), DEC (decrement) have only one argument:

```
40              INC AX
FE 07           INC byte [BX]
FF 07           INC word [BX]
48              DEC AX
```

NEG, NOT

NEG (negate), NOT (bit-wise not):

```
F7 D8           NEG AX
F6 1F           NEG byte [BX]
F7 1F           NEG word [BX]
F7 D0           NOT AX
```

MUL, DIV

The format of MUL and DIV instructions differs from the one of other arithmetic instructions. MUL has only one argument. It multiply AX by its argument and write the result to a pair DX:AX, where DX is high part and AX low part.

```
                MUL src             ; DX:AX = AX * src


F7 E3           MUL BX              ; DX:AX = AX * BX
F7 27           MUL WORD [BX]       ; DX:AX = AX * [BX]
```

There are two types of MUL instructions. One for unsigned value (MUL) and one for signed (IMUL).

```
F7 EB           IMUL BX             ; DX:AX = AX * BX
```

DIV

Division has a signed (IDIV) and an unsigned (DIV) forms. They divides a number represented by a pair of registers DX:AX, where DX is high part and AX is low part by the argument. The quotient is written to AX, the remainder to DX.

```
                DIV src         ; AX = DX:AX / src
                                ; DX = DX:AX % src

F7 F3           DIV BX          ; AX = DX:AX / BX
                                ; DX = DX:AX % BX
F7 FB           IDIV BX         ; AX = DX:AX / BX
                                ; DX = DX:AX % BX
```

CWD

In case a division of a 16-bit number by a 16-bit number is required, 16-bit divident need to be expanded to 32-bit pair DX:AX. For unsigned numbers we just need to zero out high half.

```
31 D2                XOR DX,DX   ; zero out dx
F7 F3                DIV BX
```

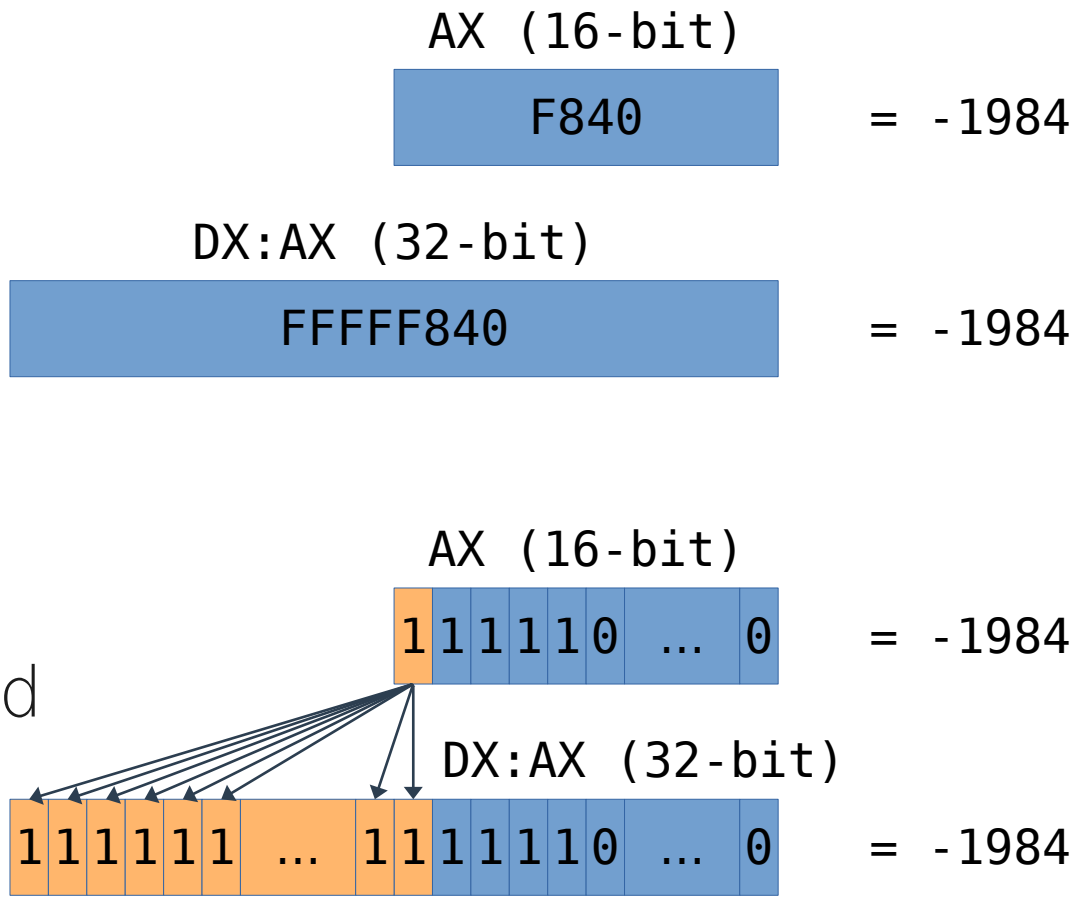
For signed special instruction CWD exists to copy the highest bit of AX to all bits of DX.

```
99                   CWD
F7 FB                IDIV BX
```

CWD

CWD converts 16-bit value in AX to 32-bit value in DX:AX.

```
99      CWD
F7 FB   IDIV BX
```



This operation is called “signed extension”.

DIV

In case a division by zero is requested. The execution of the program is interrupted and the control is transferred to the OS. It is up to the OS to decide what to do with the program next. The program is usually terminated. Most OSes provide a (OS-specific) way to handle the division by zero and to continue the execution.

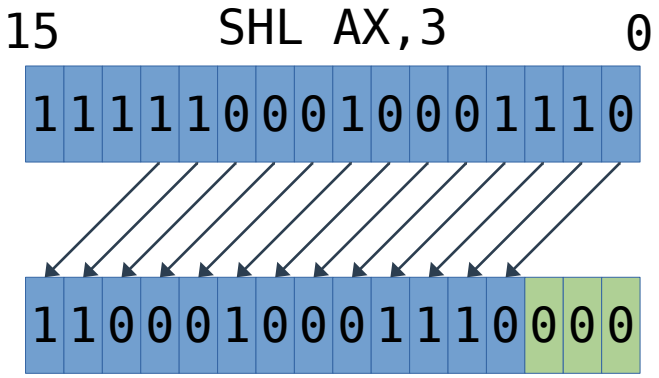
When the result of 32-bit by 16-bit division doesn't fit 16-bit register the same error as division by zero is reported.

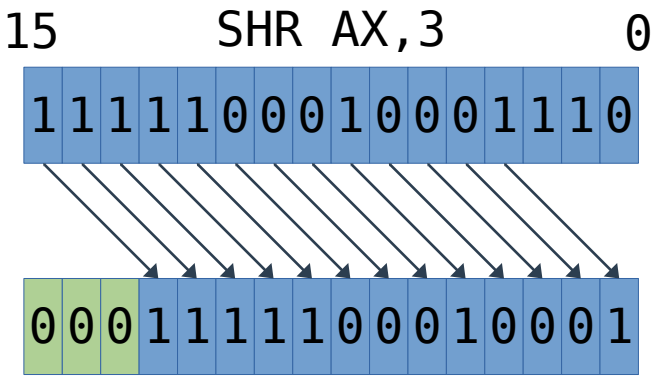
Shifts

For shifts there are three instructions available:

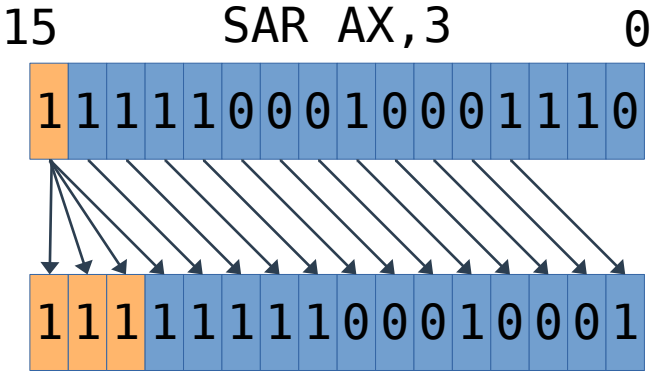
```
D3 E0          SHL AX,CL
D3 E8          SHR AX,CL
D3 F8          SAR AX,CL
```

Shifts





SHR is called logical shift. It is used for unsigned numbers.



SAR is called arithmetic shift. It is used for signed numbers.

Branches, JMP

Instruction JMP modify register IP, so the next instruction to be executed is not the next instruction after JMP, but the instruction at the address specified in the argument.

```
40                  loop:    INC AX
EB FD                        JMP loop
```

FD means -3. It is added to register IP after execution of JMP instruction. It means that targets of 2-byte JMP instruction must be within range -128..127 from the end of JMP instruction.

JMP

In case JMP target is further than -128..127 then longer form of JMP can be used.

```
E9 34 12        JMP label

... 0x1234 bytes of data

label:
```

Conditional Branches

To make a conditional branch a pair of instructions is required:

```
39 D8          cmp ax, bx   ; compare ax and bx
74 10          je label     ; jump if ax == bx

39 D8          cmp ax, bx   ; compare ax and bx
7F 10          jg label     ; jump if ax > bx
```

Conditional branches

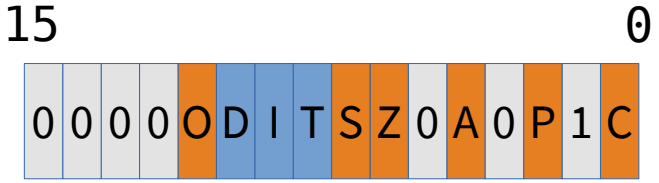
There are many types of conditional branches:

```
je, jne          jump if equal/not-equal
jg, jng          jump if greater (signed)
jl, jnl          jump if less (signed)
ja, jna          jump if above (unsigned)
jb, jnb          jump if below (unsigned)
```

FLAGS register

cmp instruction modifies the register called FLAGS.

jxx instructions reads register FLAGS and jump according to the condition.



Bits from this register have they own names:

- bit C is called carry flag
- bit Z is called zero flag
- bit S is called sign flag
- bit O is called overflow flag

Conditional branches

There are jxx instructions that checks the specific bits in FLAGS register.

```
jc/jnc          jump if carry flag is set
jz/jnz          jump if zero flag is set
js/jns          jump if sign flag is set
jo/jno          jump if overflow flag is set
```

FLAGS register

Register FLAGS is modified not only by instruction CMP, but also by most other arithmetic instructions (ADD, SUB, MUL, etc).

CMP modifies register FLAGS the same way SUB instruction does (CMP is SUB that doesn't write destination).

ADD/SUB modify FLAGS register in the following way:

- ZF (zero) is set when the result is zero.
- SF (sign) is set when the result is negative.
- CF (carry) is set when unsigned operation caused 16th bit to be carried over/borrowed from
- OF (overflow) is set when signed operation causes overflow.

Carry Flag vs Overflow Flag

```
          0000 + 0001 = 0001
signed       0 +    1 =    1
unsigned     0 +    1 =    1

OF = 0       CF = 0
```

```
                7FFF + 0001 =   8000
signed     32767 +    1 = -32768
unsigned   32767 +    1 =  32768

OF = 1         CF = 0
```

```
          FFFF  + 0001 = 0000
signed    -1    +    1 =    0
unsigned  65535 +    1 =    0

OF = 0       CF = 1
```

```
                 8000 + 8000   = 0000
signed     -32768 + -32768 =    0
unsigned    32768 + 32768  =    0

OF = 1       CF = 1
```

CMP vs SUB

```
cmp AX, BX
je label
```

Which flag je should check?

```
sub AX, BX
```

- ZF (zero) is set when the result is zero.
- SF (sign) is set when the result is negative.
- CF (carry) is set when unsigned operation caused 16th bit to be carried over/borrowed from
- OF (overflow) is set when signed operation causes overflow.

CMP vs SUB

```
cmp AX, BX
je label
```

Which flag je should check?

Answer: ZF.

```
74 10       je label
74 10       jz label
```

je and jz is the same instruction!

```
sub AX, BX
```

- ZF (zero) is set when the result is zero.
- SF (sign) is set when the result is negative.
- CF (carry) is set when unsigned operation caused 16^{th} bit to be carried over/borrowed from
- OF (overflow) is set when signed operation causes overflow.

Conditional branches

There are jxx instructions that checks the specific bits in FLAGS register.

```
je/jz          jump if ZF=1
jc             jump if CF=1
js             jump if SF=1
jo             jump if OF=1
```

CMP vs SUB

```
cmp AX, BX
ja/jb label
```

Which flag ja/jb should check?

```
sub AX, BX
```

- ZF (zero) is set when the result is zero.
- SF (sign) is set when the result is negative.
- CF (carry) is set when unsigned operation caused 16th bit to be carried over/borrowed from
- OF (overflow) is set when signed operation causes overflow.

Conditional branches

There are jxx instructions that checks the specific bits in FLAGS register.

```
je/jz          jump if ZF=1
jb/jc          jump if CF=1
ja             jump if CF=0 & ZF=0
js             jump if SF=1
jo             jump if OF=1
```

CMP vs SUB

```
cmp AX, BX
jg/jl label
```

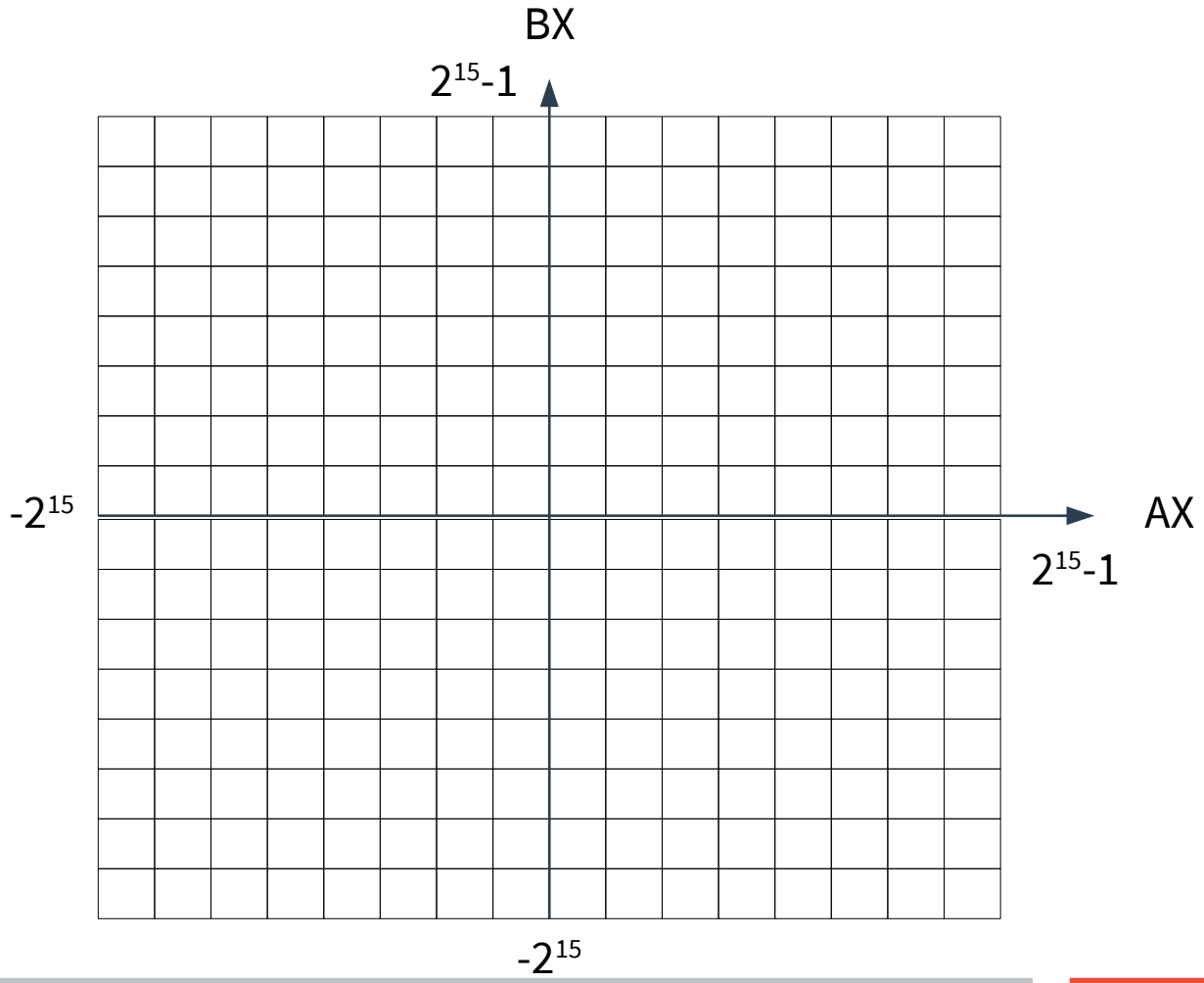
Which flag jg/jl should check?

```
sub AX, BX
```

- ZF (zero) is set when the result is zero.
- SF (sign) is set when the result is negative.
- CF (carry) is set when unsigned operation caused 16th bit to be carried over/borrowed from
- OF (overflow) is set when signed operation causes overflow.

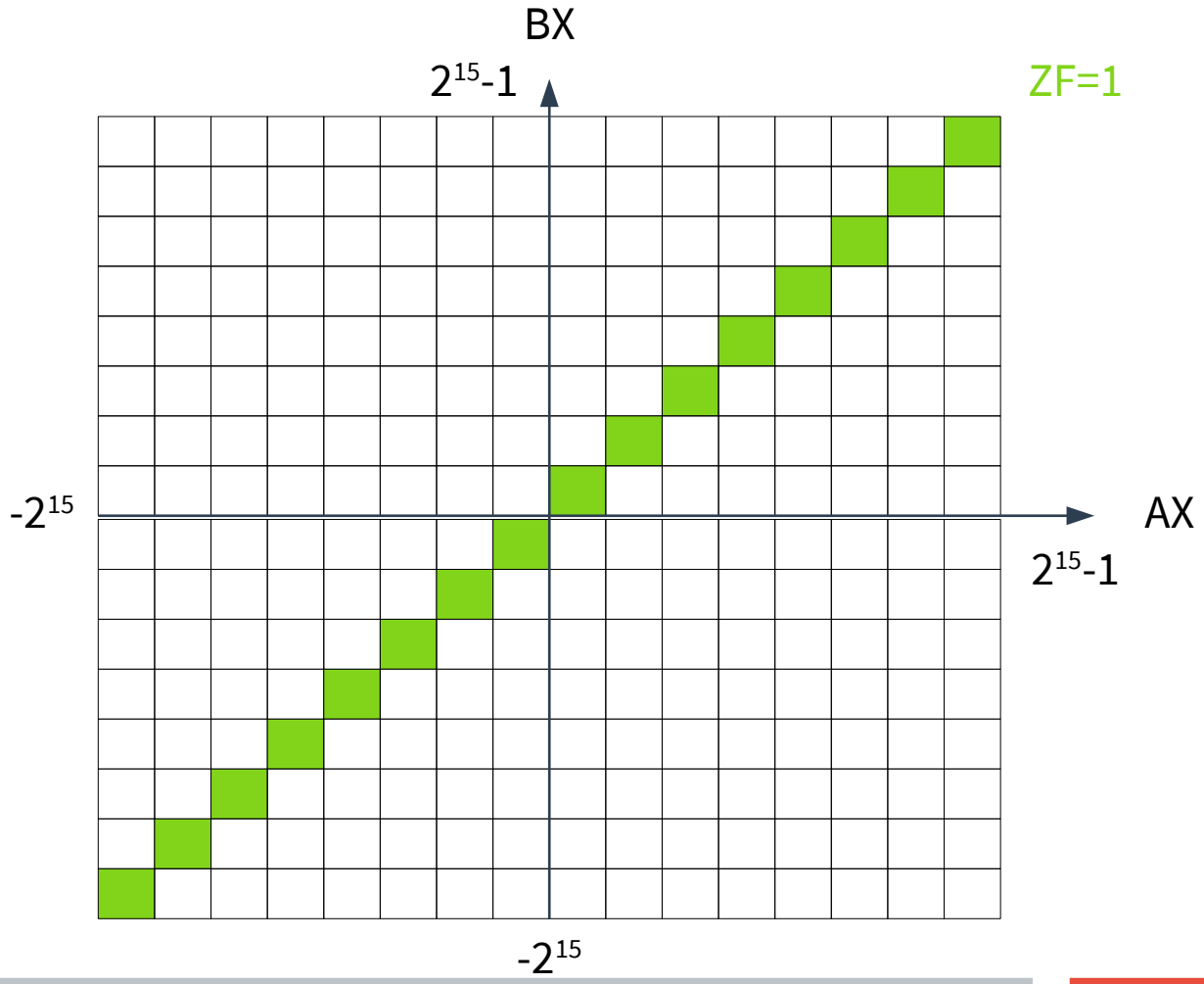
Flags Graph

CMP AX, BX



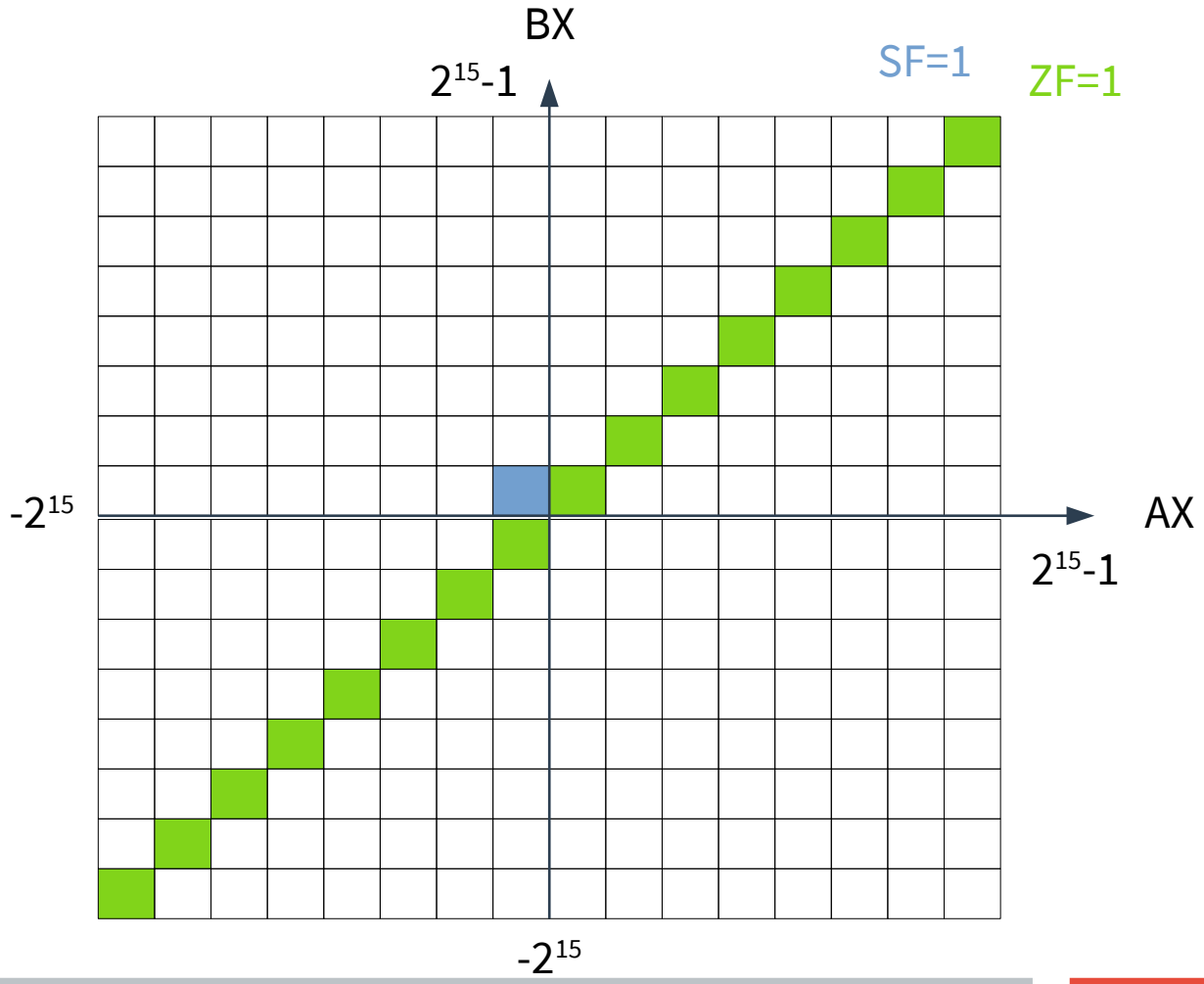
Flags Graph

CMP AX, BX



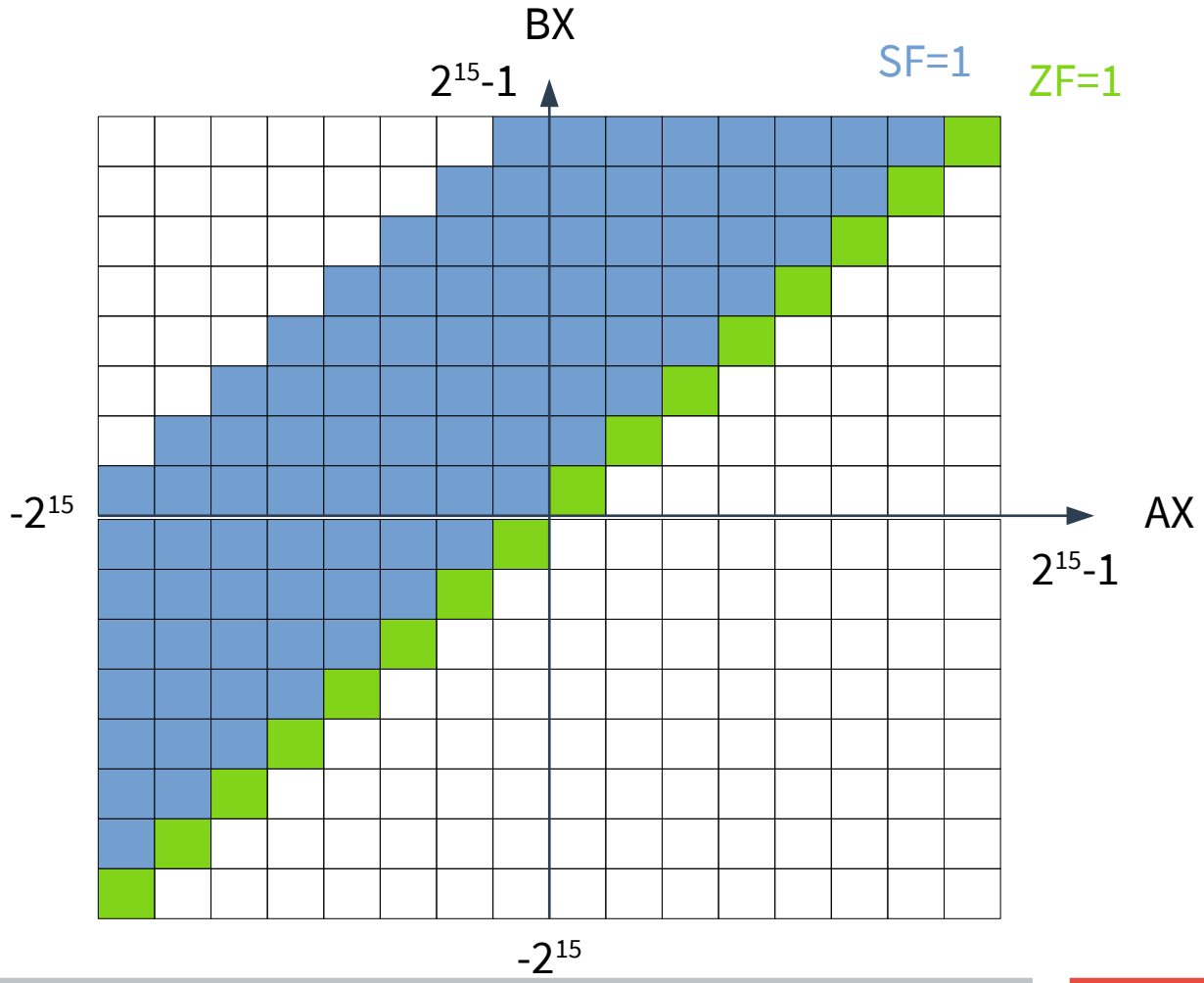
Flags Graph

CMP AX, BX



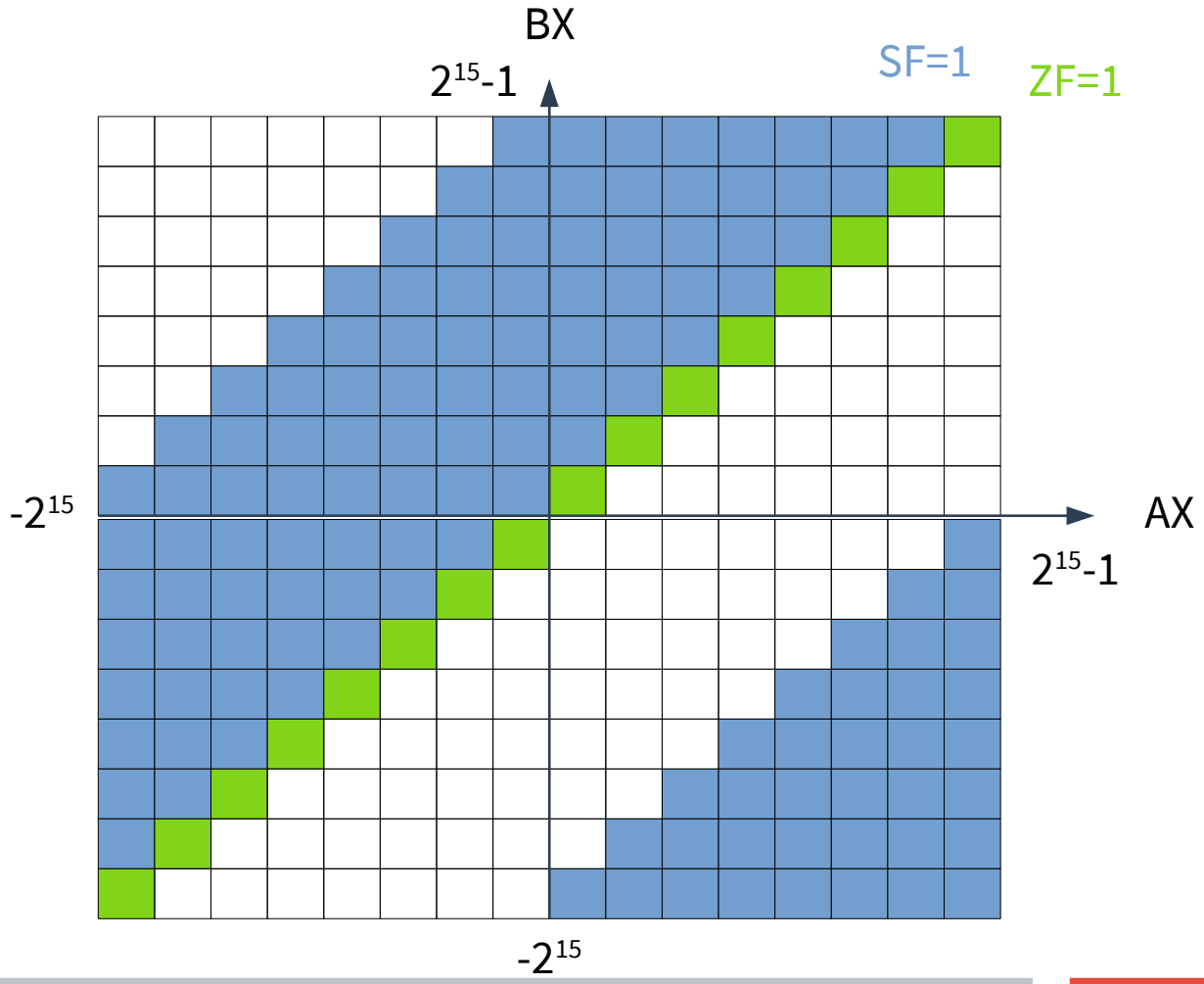
Flags Graph

CMP AX, BX



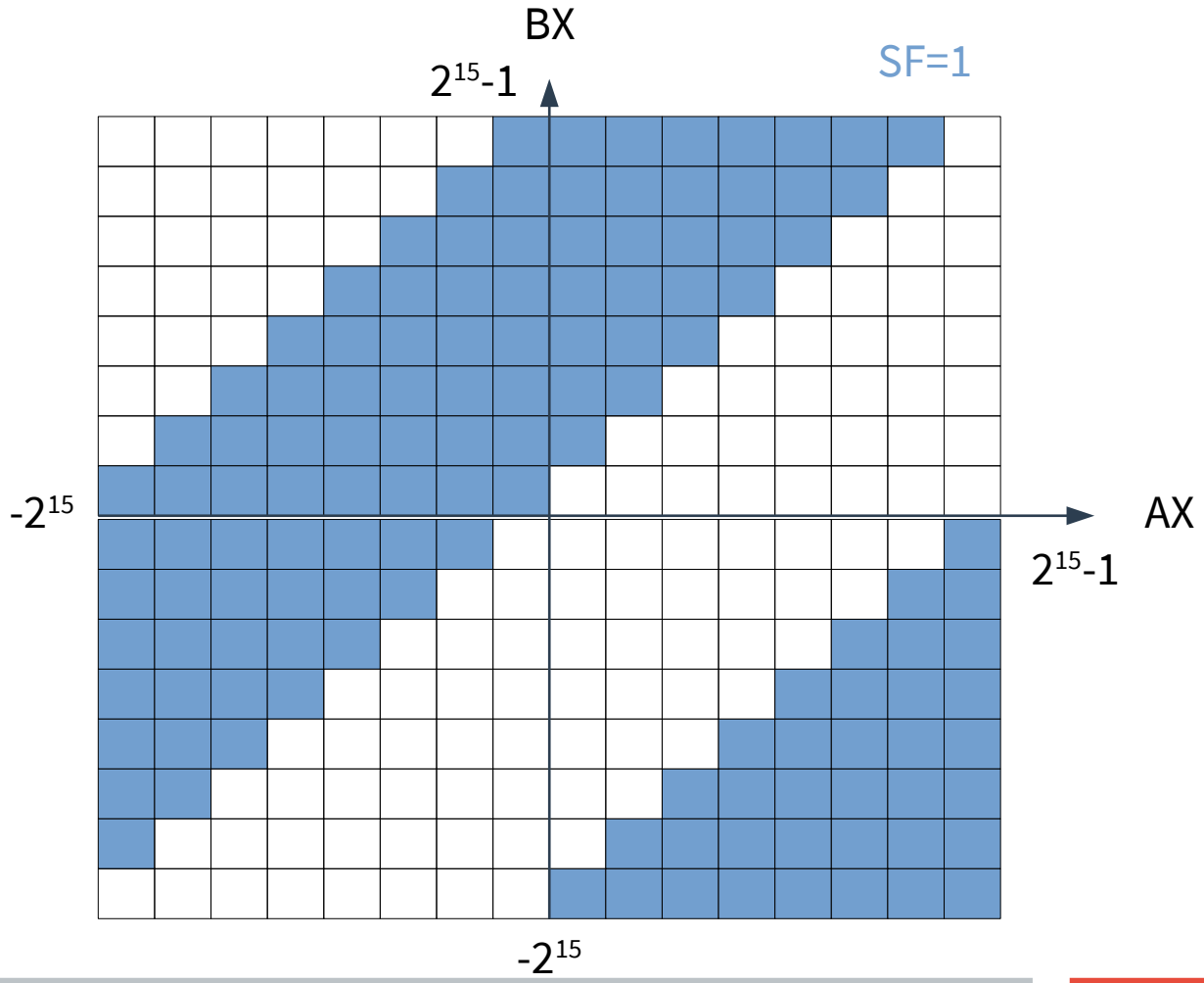
Flags Graph

CMP AX, BX



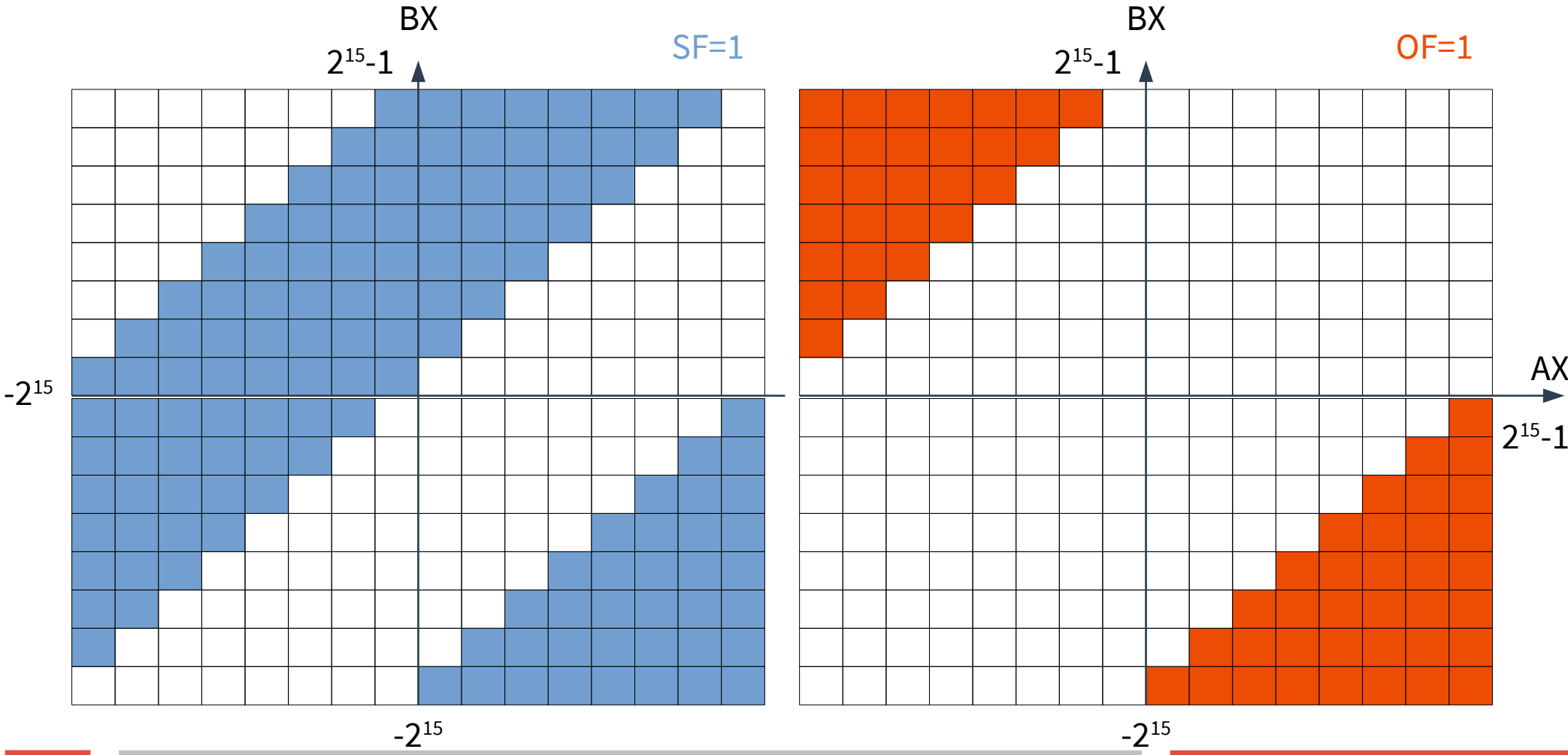
Flags Graph

CMP AX, BX



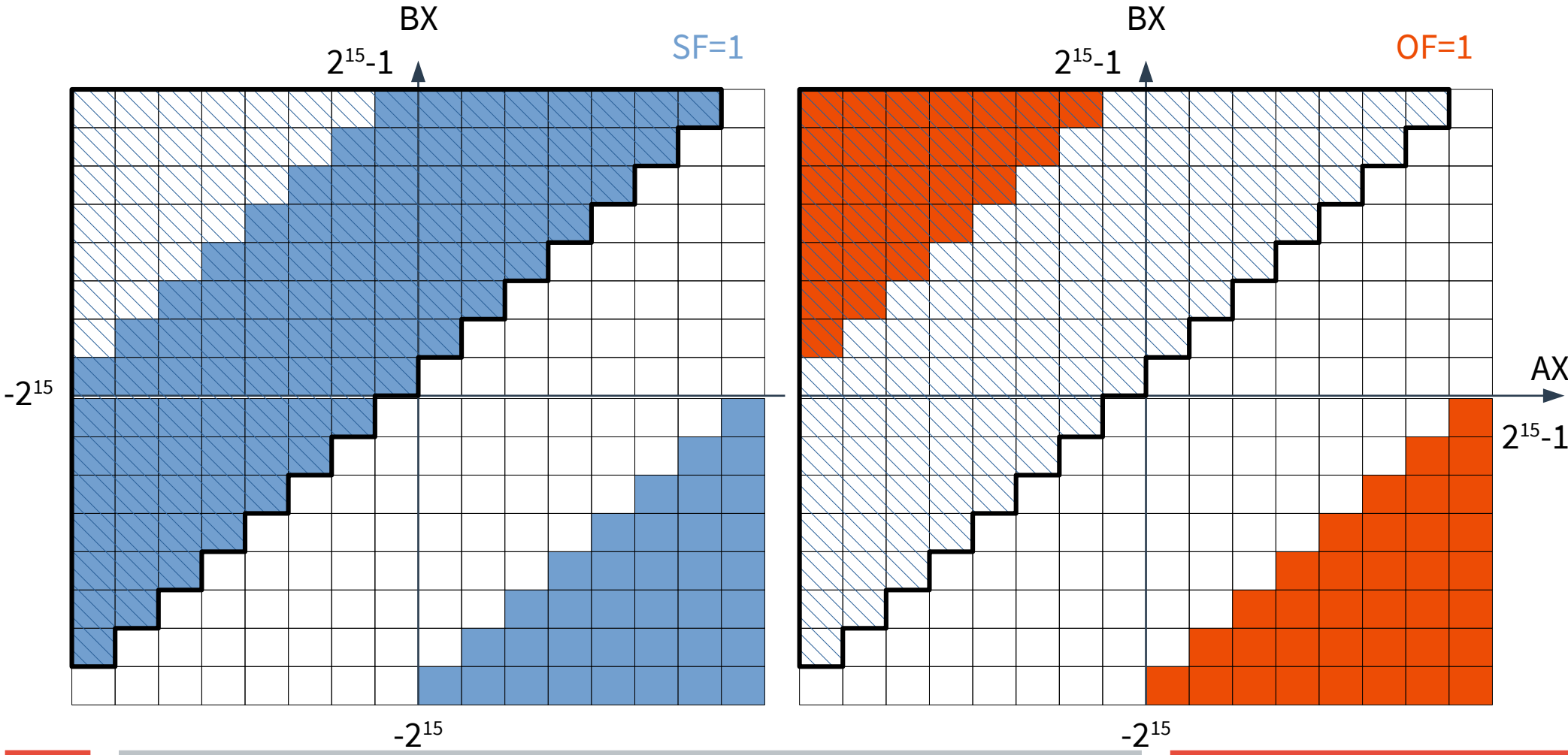
Flags Graph

CMP AX, BX



Flags Graph

AX < BX



Conditional branches

There are jxx instructions that checks the specific bits in FLAGS register.

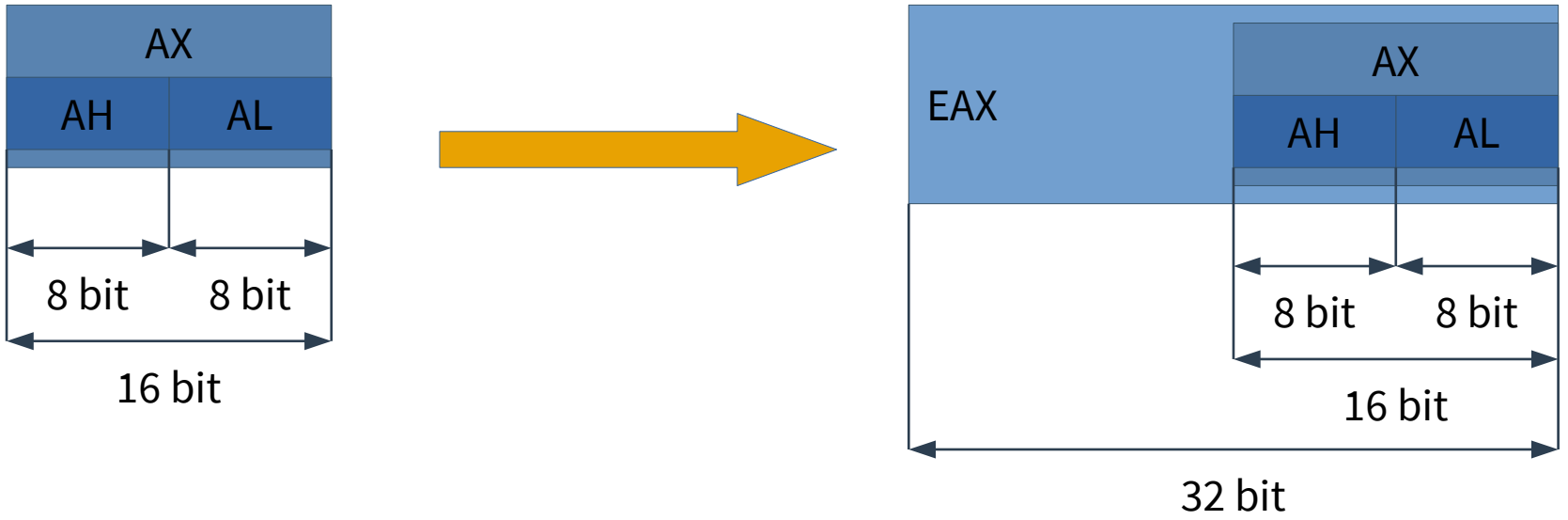
```
je/jz           jump if ZF=1
jb/jc           jump if CF=1
ja              jump if CF=0 & ZF=0
jl              jump if SF≠OF
jg              jump if SF=OF & ZF=0
js              jump if SF=1
jo              jump if OF=1
```

Branches

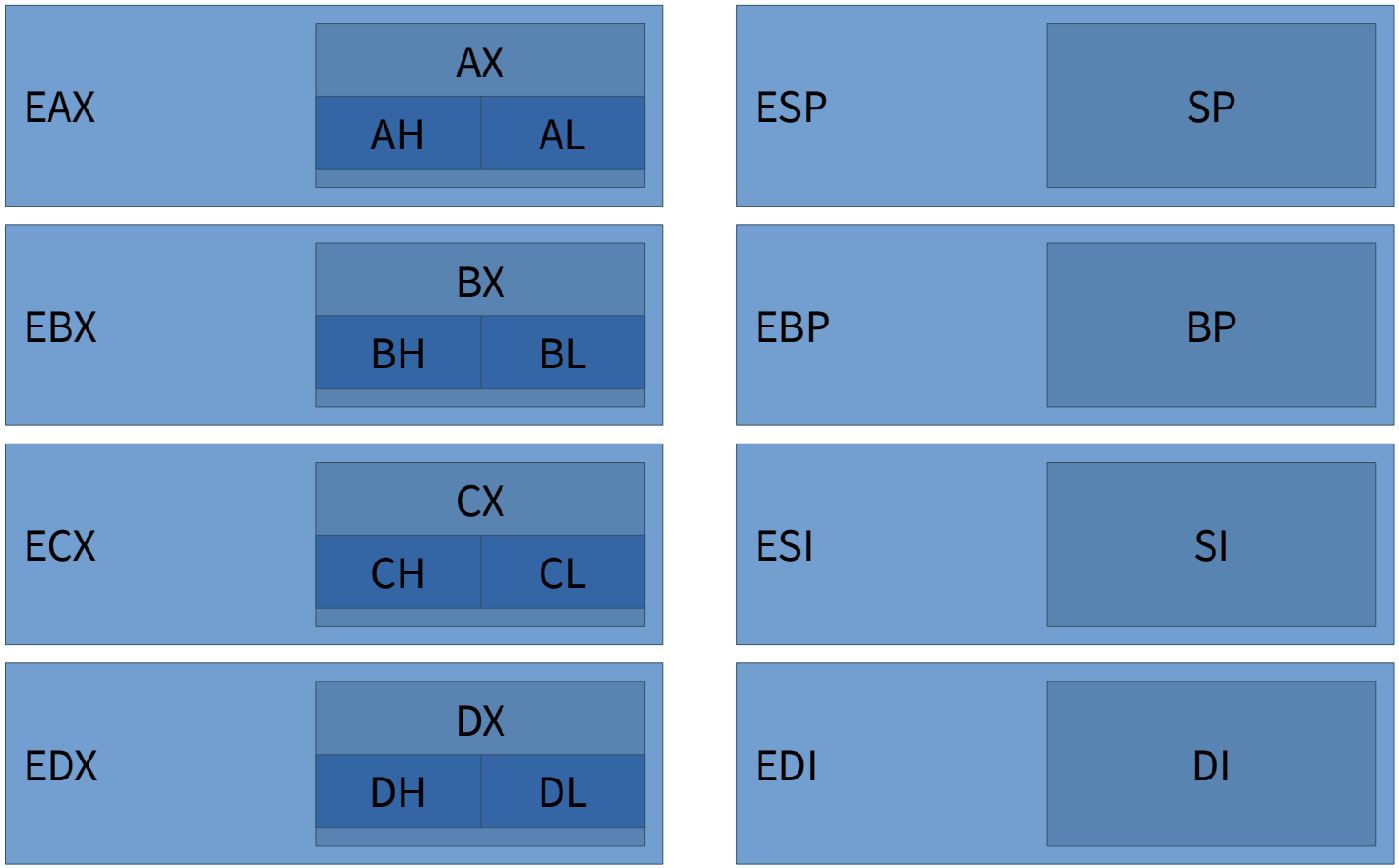
Many instructions update FLAGS register according to the result of the operation. Sometimes conditional branches can be made without cmp:

```
89 C2       loop:   mov dx,ax
01 D8               add ax,bx
89 D3               mov bx,dx
49                  dec cx
75 F7               jnz loop
```

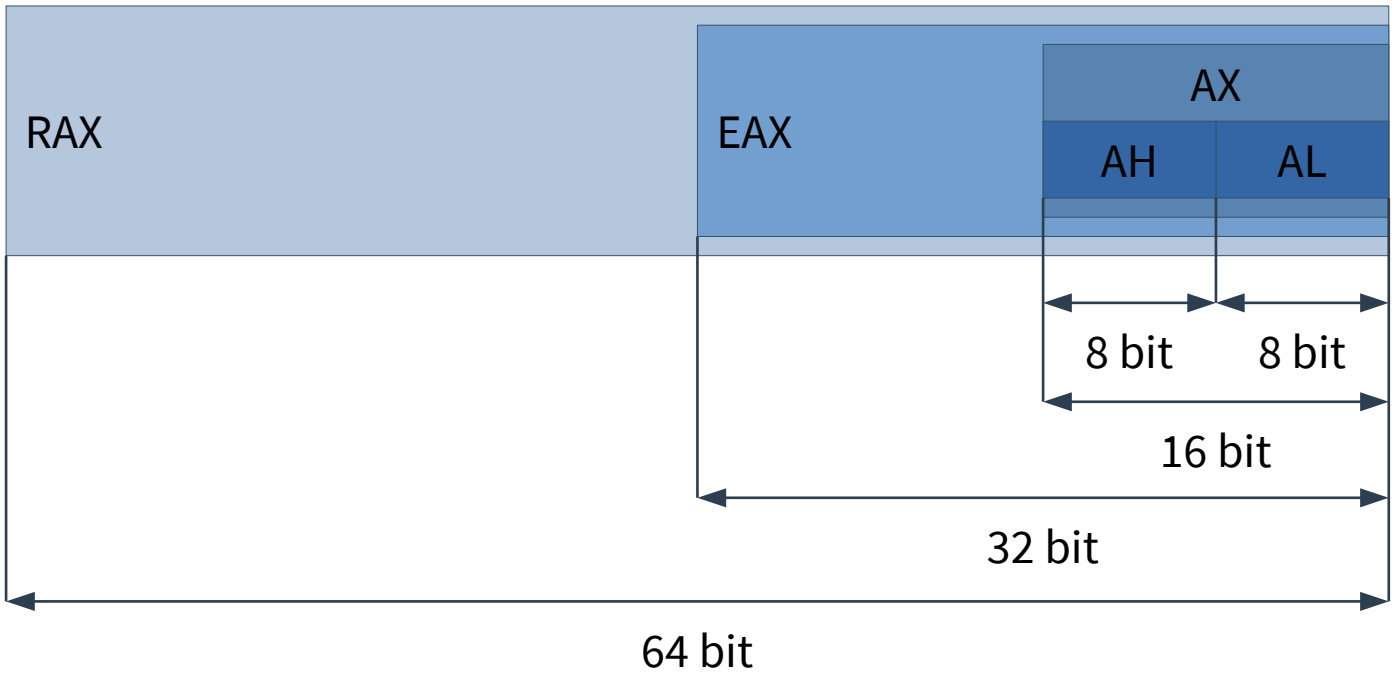
Registers 32-bit



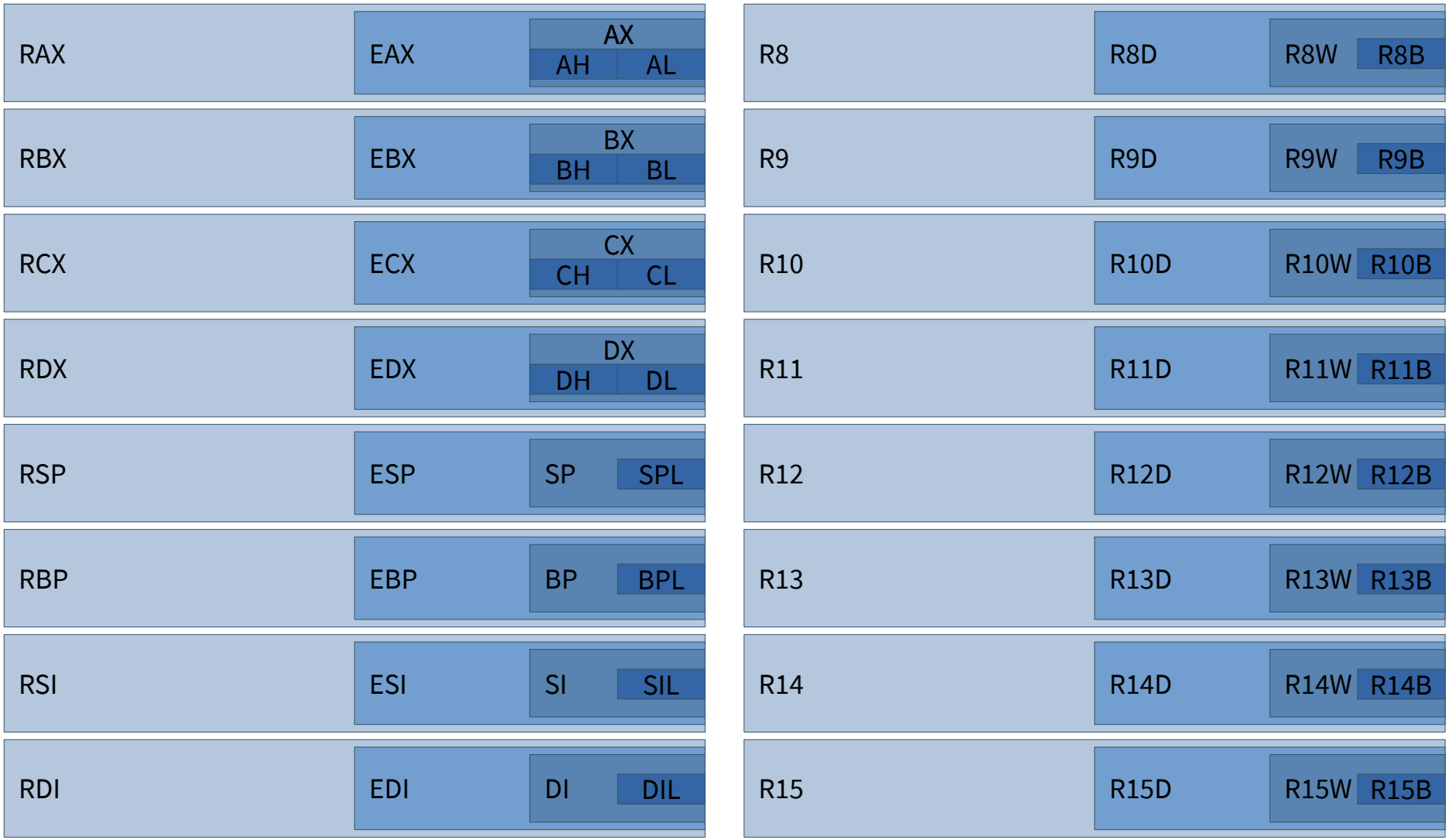
Registers 32-bit



Registers 64-bit



Registers 64-bit



Addressing modes

```
MOV reg, [reg + {1,2,4,8} * reg + imm]
```

```
48 8B 44 8B 01    MOV RAX, [RBX + RCX*4 + 1]
48 8D 44 8B 01    LEA RAX, [RBX + RCX*4 + 1]
```

ADC

```
ADD AX, BX

(17-bit) (16-bit) (16-bit)
 CF:AX  =   AX   +   BX

ADC AX, BX

(17-bit) (16-bit) (16-bit) (1-bit)
 CF:AX  =   AX   +   BX   + CF
```

SBB

```
SUB AX, BX

(17-bit) (16-bit) (16-bit)
 CF:AX  =   AX   -   BX

SBB AX, BX

(17-bit) (16-bit) (16-bit) (1-bit)
 CF:AX  =   AX   -   BX   - CF
```