

# Chapter 3: Introduction to the Assembler

# Topic

- Assembly programming and 68000 microprocessor architecture
  - Chapter 7 by Berger
  - Chapter 2 by Clements
  - Quick Reference:  
<http://www.easy68k.com/files/EASy68KQuickRef.pdf>

# From C/C++ to Machine Code

- **Write** your code in Visual Studio, Xcode, Vim, or ...
- **Compile** it using Visual Studio, Xcode, Gcc/G++, or ...
  - Cross-compiler: compile the code for a (micro)computer system with a processor of different architecture
  - *Compiling, linking, and assembling*
- **Run** the compiled program (binaries, images, etc.) on the same or a different computer with a processor of the same architecture
  - Run the cross-compiled program on a system with a processor of different architecture
  - *Loading and executing*
- The compiled program is called “**Machine Code**” or “**Machine Language**”

# Introduction to Assembly Language

- Every computer system has a **fundamental set of operations** it can perform
- **These operations** are defined by the *instruction set* of the processor
  - The instruction set is the **atomic element of the processor**
  - All the complex operations are achieved by building **sequences of these fundamental operations**
  - Called **Machine Language** or **Machine Code**
- Assembly language is the *human readable* form of the *machine language*

Instead of writing a program in machine language as:

```
00000412 307B7048
00000416 327B704A
0000041A 1080
0000041C B010
0000041E 67000008
00000422 1600
00000424 61000066
00000428 5248
0000042A B0C9
```

We write the program in assembly language as:

MOVEA.W	(TEST_S,PC,D7),A0	*We'll use address indirect
MOVEA.W	(TEST_E,PC,D7),A1	*Get the end address
MOVE.B	D0,(A0)	*Write the byte
CMP.B	(A0),D0	*Test it
BEQ	NEXT_LOCATION	*OK, keep going
MOVE.B	D0,D3	*Copy bad data
BSR	ERROR	*Bad byte
ADDQ.W	#01,A0	*Increment the address
CMPA.W	A1,A0	*Are we done?

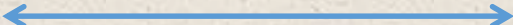
# Why Assembly Language?

- Computer programming depends upon a knowledge of the processor
  - Understanding assembly language is of **understanding the computing engine**
  - **Performance optimization**
  - Debugging
  - Kernel development
    - Mixed C and Assembly language programming in Linux kernel
- Haven't compilers made assembly language obsolete?
  - *Not all processor architectures have compilers to support*



# Assembly Language

- There is a **1:1 correspondence** between **assembly language** instructions and **machine language** instructions

**MOVE.B D1,D3**  **1601**

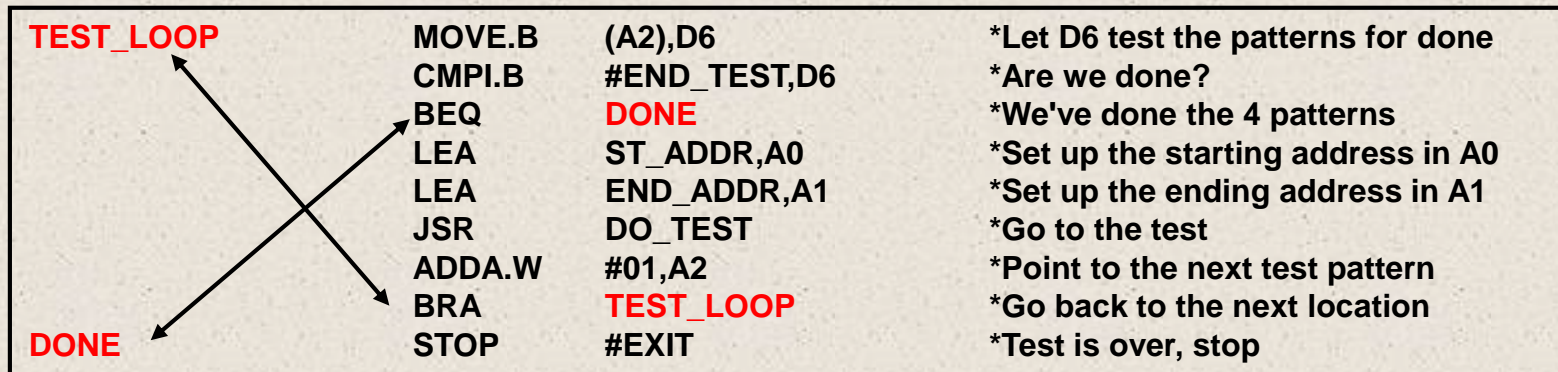
- The assembly language instructions (called ***mnemonics***) give a human readable indication of what the instruction does
- For example:
  - **MOVE.B** : Move (Copy) one byte of data
  - **CMP.B** : Compare two bytes of data
  - **BEQ** : Branch to a different instruction if the “result” is zero
  - **ADD.W** : Add two “word” values

# Format of the 68000 Instruction

Each instruction has a **label**, **op-code**, **operands** (0 to 2)

Label	Op-Code	Operand1	Operand2	*Comment
THIS	MOVE.B	\$1234	\$5678	*an example

- **Label**: a symbolic name, usually *refers to a memory address*
  - Label is a tool for a readable program



# Format of the 68000 Instruction

- **Op-code:** Instructions to the microprocessor
  - Example: MOVE, CMP
- **Pseudo Op-codes** (Assembler directives): Used to help *make the program readable*, instructions to the assembler program
  - Example: ORG, EQU, SET, REG, DC, DCB, DS, END
- **Operands:** (0 operand, 1 operand, 2 operands)
  - 0 operand: NOP (No **OP**eration, do nothing)
  - 1 operand: BRA FOO (**BR**anch **Al**ways)
  - 2 operands: ADD.W D0,D3
- Operand is one of Effective Addressing modes (where is the data?)

	ORG	\$400	*Start of code
	MOVE.B	Y,D0	*Get first operand
	ADDI.B	#24,D0	*Add constant
	MOVE.B	D0,Sum	*Store the result
	ORG	\$600	*Start of data area
Y	DC.B	27	*Store the constant 27 in memory
Sum	DS.B	1	*Reserve a byte for Sum



# Instruction Set Architecture

- In order to program in assembly language, we must be familiar with the **programmer's model of the processor**, which includes:

## *Instruction Set* and *Effective Addressing Modes*

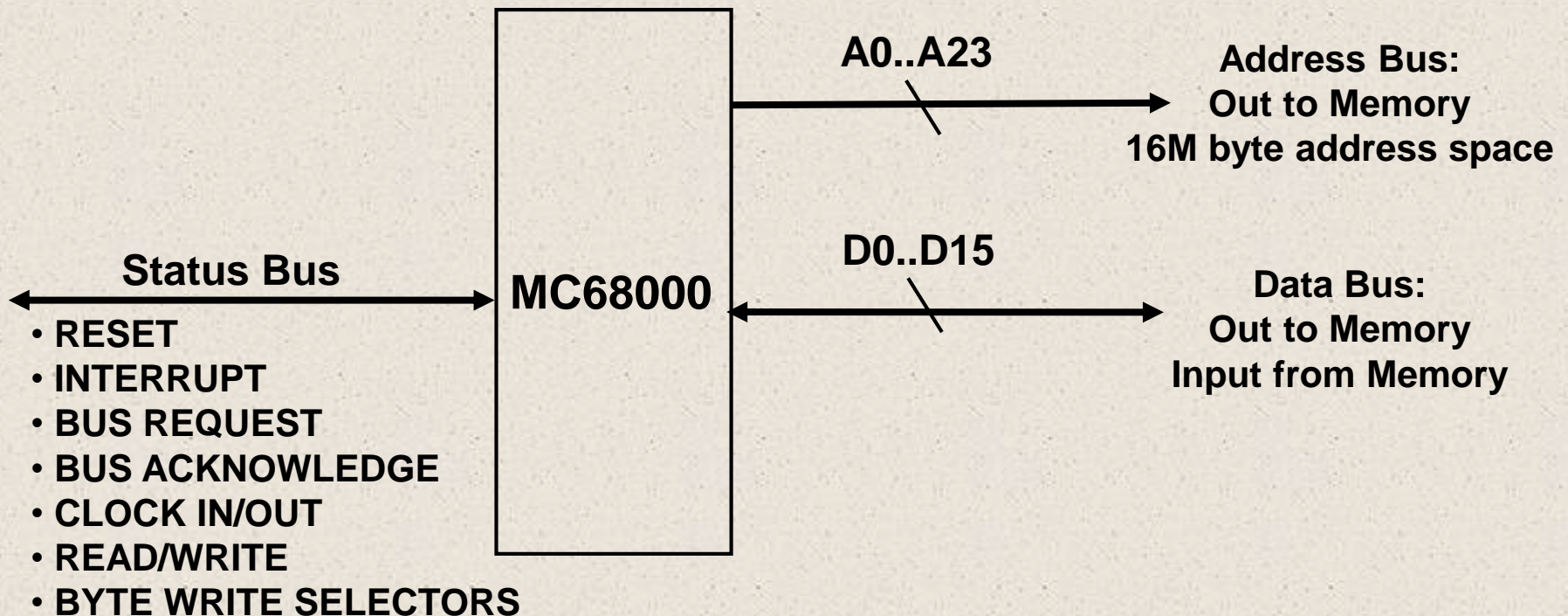
- **Instruction Set**
  - Tells the processor what to do (opcode)
- **Effective Addressing Modes**
  1. Describe how the processor **accesses the data** that the instruction will operate on, and
  2. Describe **what to do with the data** after the operation
- Before we can create an assembly language program, we need to understand the architecture of the machine we are programming for
  - Unlike C/C++, **assembly language is not portable between computers with processors of different architectures**
  - E.g., an assembly language program written for a Pentium processor will not run on an ARM7 architecture processor

# Effective Addressing Modes

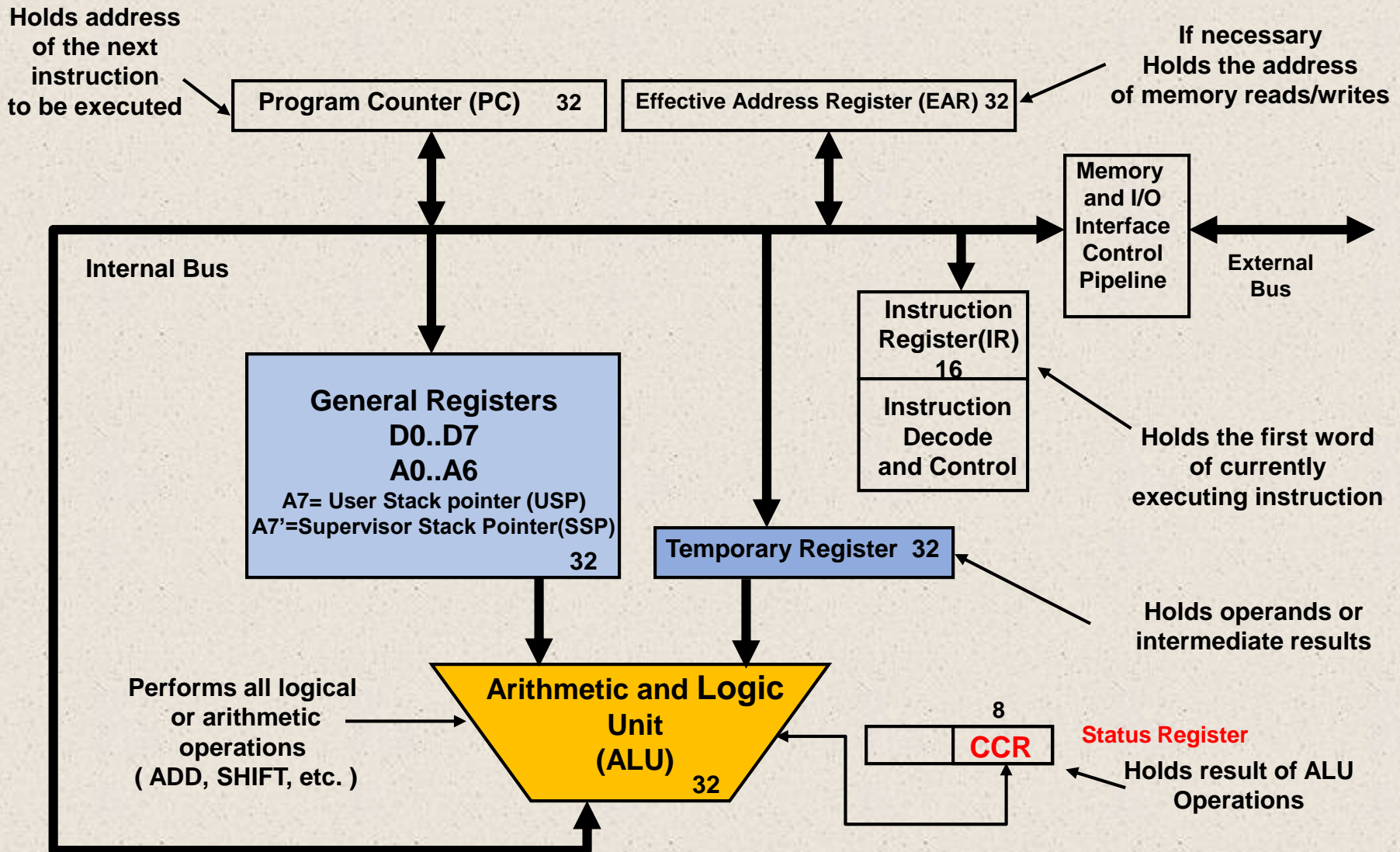
- In 68K manual, each instruction has different codes for each EA mode
  - **Dn**: data register direct: D0, D1, ..., D7
  - **An**: address register direct : A0, A1, ..., A6
  - **(An)**: address register indirect: (A0), (A1), ..., (A6)
  - **(An)+**: address register indirect with post-increment
  - **-(An)**: address register indirect with pre-decrement
  - **(xxx).W**: Absolute addressing (word)
  - **(xxx).L**: Absolute addressing (long-word)
  - **#<data>**: Immediate Addressing
  - **(d<sub>16</sub>, An)**: address register indirect with displacement ( $EA = (An) + d_{16}$ )
  - **(d<sub>8</sub>, An, Xn)**: address register indirect with index ( $EA = (An) + (Xn) + d_8$ )
  - **(d<sub>16</sub>, PC)**: Program counter with displacement ( $EA = (PC) + d_{16}$ )
  - **(d<sub>8</sub>, PC, Xn)**: Program counter with index ( $EA = (PC) + (Xn) + d_8$ )

# Microprocessor Component Systems

- Three major busses of an MC68000 microprocessor
  - **Address Bus:** Unidirectional, homogeneous (24 lines)
  - **Data Bus:** Bidirectional, homogeneous (16 lines)
  - **Status Bus:** Heterogeneous, additional control and housekeeping signals
  - Three-bus system, similar in other processors



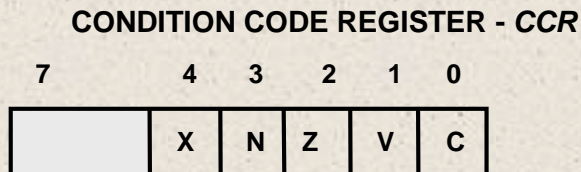
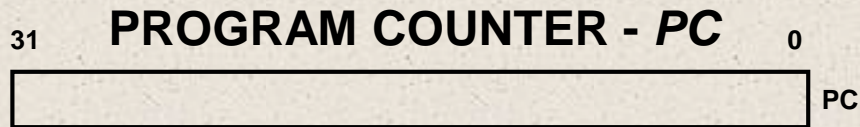
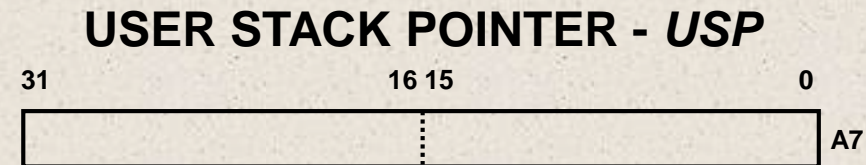
# Hardware Organization of the MC68000





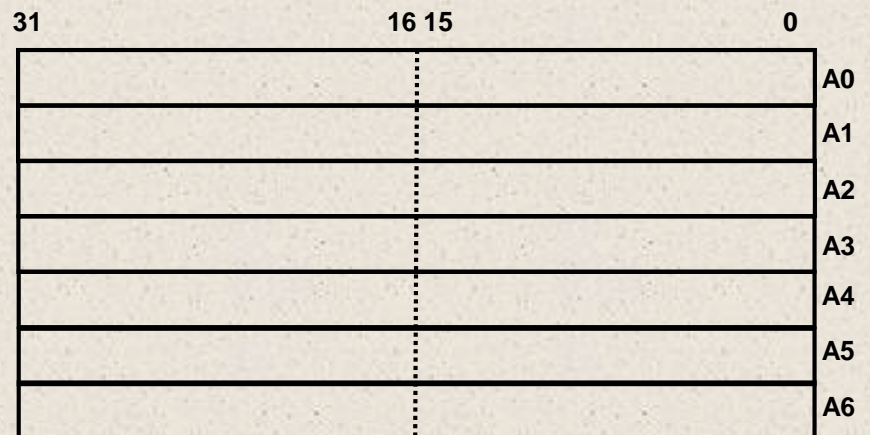
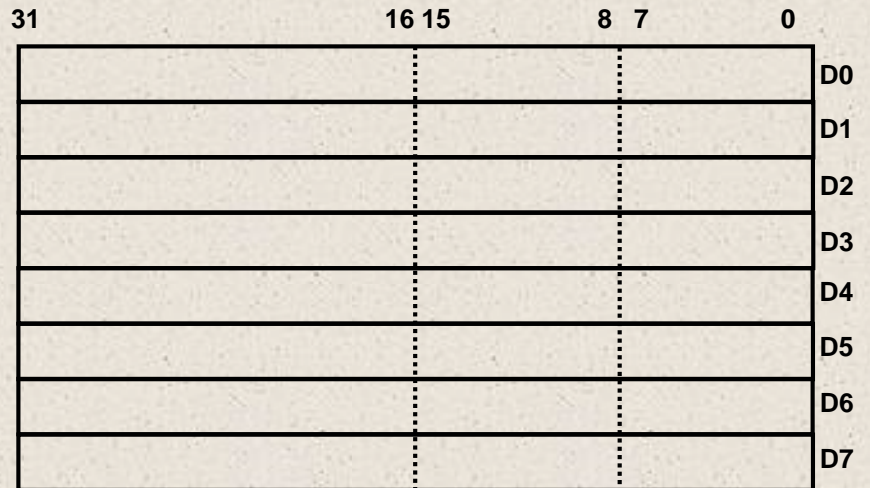
# User Programming Model

- For most applications the architectural model of the 68000 is the ***User Programmer's Model***



## CONDITION CODES (FLAGS)

- C = Carry Flag
- V = Overflow Flag
- Z = Zero Flag
- N = Negative Flag
- X = Extend Flag



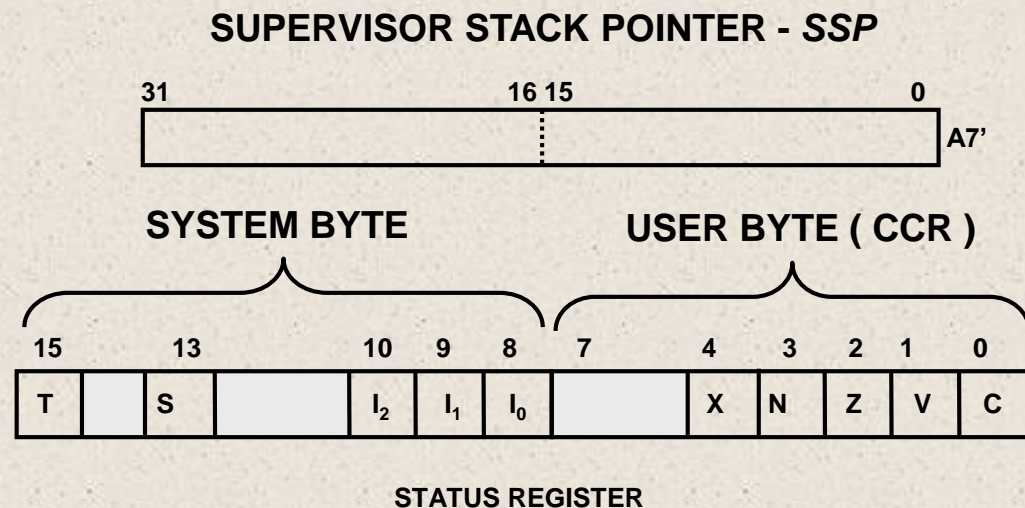


# Supervisor Programming Model

- In addition to the registers available in the **User Programming Model**, the **Supervisor Programming Model** includes *additional registers to support task (Operating System) management*
- An operating system normally accesses the **Supervisor Mode** to prevent individual software modules from interfering with each other
- If  $S=1$ , then in Supervisor Mode
  - Stack pointer is **A7'** when  $S=1$
  - **Application** programs *should not* change bits in **Status Register**

## SUPERVISOR STATUS CODES

- **T** = Trace bit
- **S** = Supervisor State
- $I_2 \ I_1 \ I_0$  = Interrupt Masks



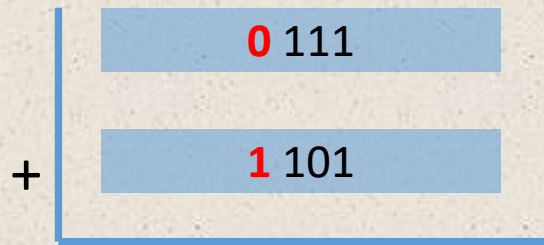
# Two's Complement Arithmetic

- Arithmetic **overflow**
  - Adding **two positive** number **results in a negative** number
  - Adding **two negative** number **results in a positive** number
- If the result is **out of range**, the computer results in an **incorrect answer**.
- How to detect the overflow? **Check V bit and C bit**
- 68K CCR register: XNZVC
  - **$V = 1$  when **overflow**  $\rightarrow$  out of range (**error**)**
  - **$C = 1$  when there is a **carry**  $\rightarrow$  got carry bit (if  $V=0$ , it is **not an error**)**

# Two's Complement Arithmetic

- In a 4-bit system (range -8 to 7)

1.  $7 + (-3) = 4$



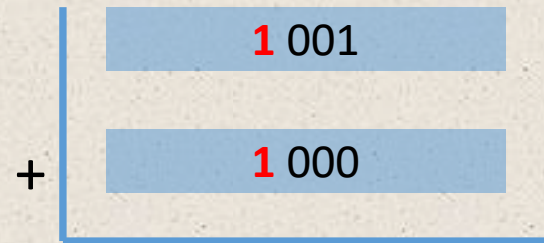
carry = 1    0 100

No overflow:  $V=0$ ,  $C=1$

carry-out bit is invisible.

Then the answer is 4 (correct)

2.  $(-7) + (-8)$



carry = 1    0 001

Overflow (sign bit changed to 0):  
 $V=1$ ,  $C=1$ .

***Incorrect result, error.***

# Introduction to Easy68K Simulator

- Step 1: Using an **ASCII-only TEXT EDITOR**, write your program as a series of instructions, line by line, then save the file (.X68)
- Step 2: Use the assembler program, Easy68K, to assemble (**not compile!**) the ASCII text file to an **object** file (.S68) and a **listing** file (.L68 )
- Step 3: Use the simulator program in Easy68K to run your program on your PC

We write the program in assembly language as:

MOVEA.W	(TEST_S,PC,D7),A0	*We'll use address indirect
MOVEA.W	(TEST_E,PC,D7),A1	*Get the end address
MOVE.B	D0,(A0)	*Write the byte
CMP.B	(A0),D0	*Test it
BEQ	NEXT_LOCATION	*OK, keep going
MOVE.B	D0,D3	*copy bad data
BSR	ERROR	*Bad byte
ADDQ.W	#01,A0	*increment the address
CMPA.W	A1,A0	*are we done?

**Source file**

machine language

00000412	307B7048
00000416	327B704A
0000041A	1080
0000041C	B010
0000041E	67000008
00000422	1600
00000424	61000066
00000428	5248
0000042A	B0C9

**Object file**