

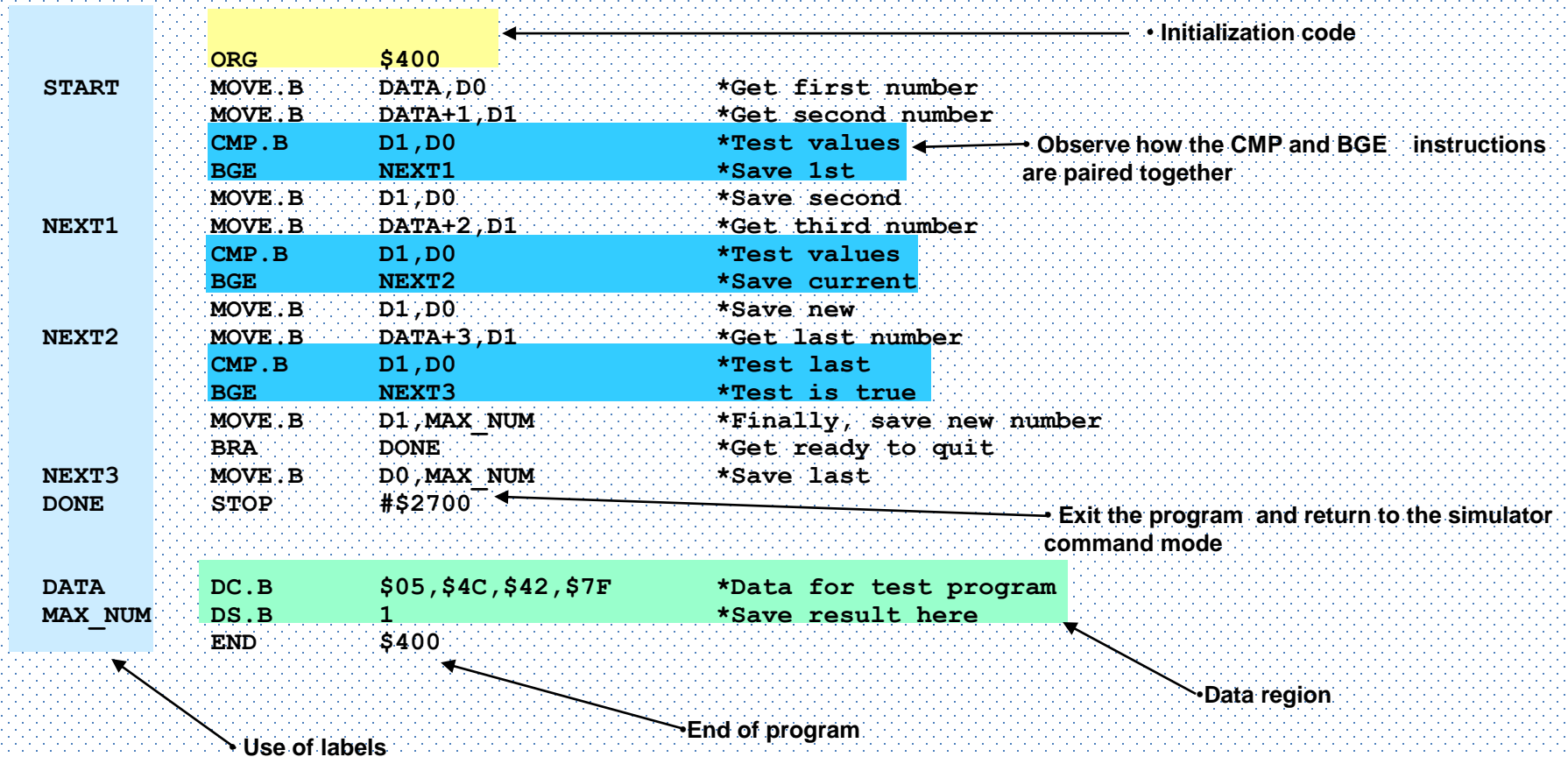
Chapter 4: Assembly Directives and Addressing Modes

Covered Topic

- 68000 Addressing Mode and Directives
 - Effective Addressing Mode
 - Directives
- 68K manual
- Chapter 8, 9 (Berger)
- Chapter 3, 4 (Clements)

Source Code for Sample Program

```
*****
*
* Program to compare 4 bytes of memory, D0 holds the largest value
*
*****
```



Assembly Language

- **Labels**

- Labels are equated **to the memory location** that defines the instruction or data they represent
- Labels start in column 1 and are usually **limited to 8-14 characters**
- First space after the label defines the beginning of the OP CODE field

- **OP Codes**

- **Operands**

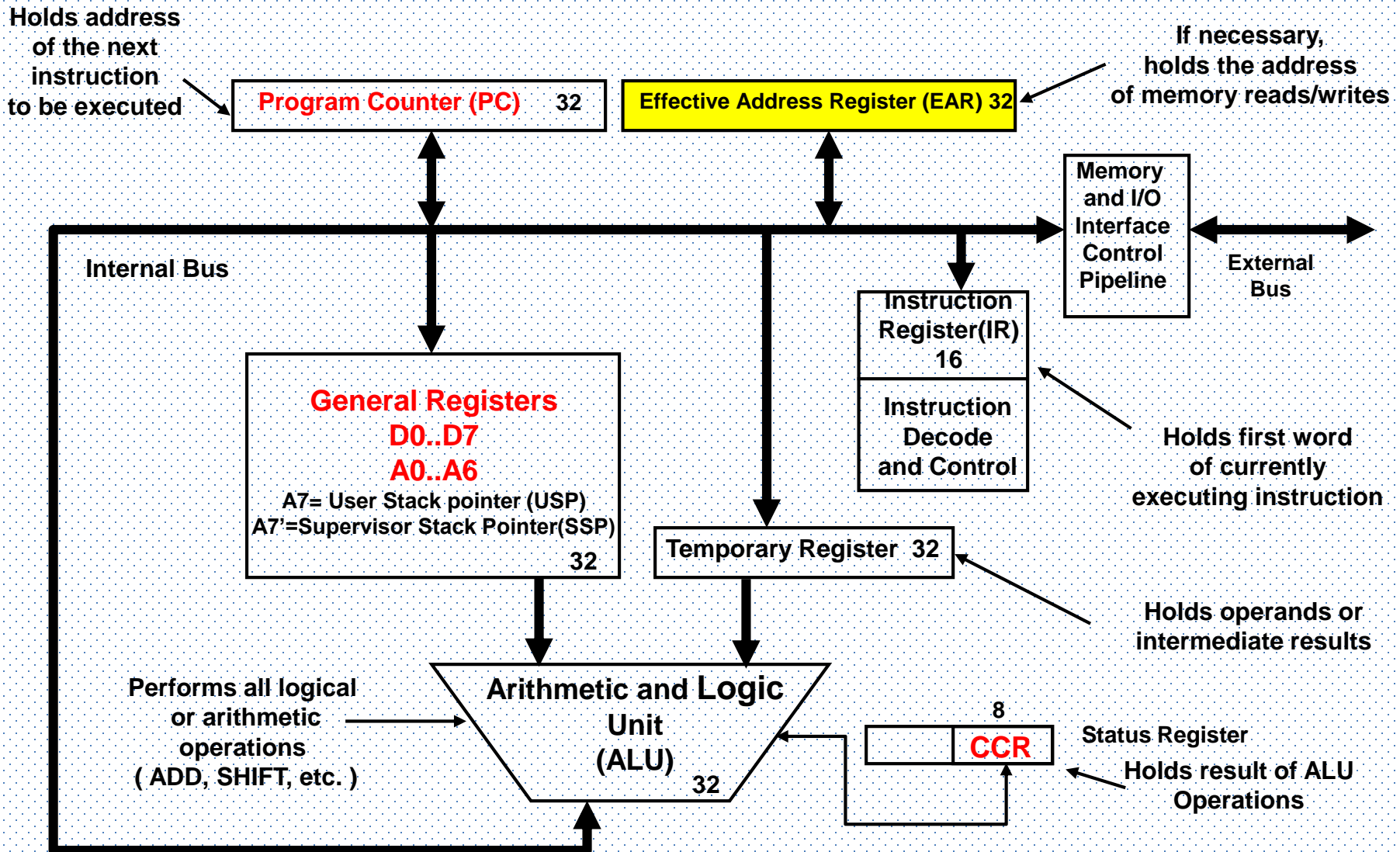
- **Comments**

- Preceded by a special character (*, ; or :) indicates the beginning of the comment field
- It is ***highly recommended*** that you comment liberally
 - 1 comment per instruction?

- **Pseudo OP Code**

- OPT, ORG, DC.B, etc.

Hardware Organization of the MC68000



Effective Addressing Modes

- In 68K manual, each instruction has different codes for different EA modes:
 - **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
 - **#<data>**: immediate addressing (direct number)
 - **(xxx).W**: absolute addressing (short/word)
 - **(xxx).L**: absolute addressing (long)
 - **(d₁₆, An)**: address register indirect with displacement
(EA = (An) + d₁₆)
 - **(d₈, An, Xn)**: address register indirect with index
(EA = (An) + (Xn) + d₈)
 - **(d₁₆, PC)**: Program counter with displacement (EA = (PC) + d₁₆)
 - **(d₈, PC, Xn)**: Program counter with index (EA = (PC) + (Xn) + d₈)

Primary Addressing Modes

- Mode 0: **Data register direct**
 - Source or destination is a data register (D0...D7)
- Mode 1: **Address register direct**
 - Source or destination is an address register (A0...A6)
- Mode 2: **Address register indirect**
 - The address register, A0...A6, contains the address of the source or destination of the effective address
 - This is the “pointer” in C++
 - The contents of the address register is the **address of the data**
- Mode 7, subclass 4: **Immediate addressing**
 - The source value (preceded by the **# sign**) is the data
- Mode 7, subclass 0: **Absolute addressing (word)**
 - The memory location is explicitly specified as a 16-bit
- Mode 7, subclass 1: **Absolute addressing (long)**
 - The memory location is explicitly specified as a 32-bit

“Data Register Indirect” mode does not exist in 68K processor!!!

Primary Addressing Modes – cont'd

- Mode 3: **Address register indirect with post-increment**
 - After the instruction is executed, the contents of the address register is incremented by one
 - ***The stack POP operation!***
- Mode 4: **Address register indirect with pre-decrement**
 - The address register, A0..A6, contains the address of the source or destination of the effective address
 - Before the instruction is executed, the contents of the address register is decremented by one
 - ***The stack PUSH operation!***

Register Direct Addressing

- **The simplest addressing mode**
 - The source **or** destination of an operand is in a **data register or** an **address register**
- Fast: external memory does not have to be accessed
- Used to hold variables that are frequently accessed
- Used by compilers to improve performance
 - *Have you used the register keyword in C or C++?*
- Examples

MOVE.B D0,D3 ;Copy the source operand in register D0 to register D3

SUB.L A0,D3 ;Subtract the source operand in register A0 from register D3

CMP.W D2,D0 ;Compare the values in register D2 and register D0

ADD.W D3,D4 ;Add the source operand in register D3 to register D4

Immediate Addressing

- The actual operand (literal value) forms part of the instruction
 - Immediate addressing can be used **only to specify a source operand**.
- Immediate addressing is indicated by a **'#' symbol** in front of the source operand.
 - MOVE.B #24, D0 → move decimal data 24 to D0
- **NOT Immediate addressing -> data is not indicated by a '#' symbol!**
 - MOVE.B 24, D0 → move **data in the address 24₁₀** to D0
 - MOVE.B \$24, D0 → move **data in the address 24₁₆** to D0
- **Important:** \$ is for hex, % is for binary, no symbol is for decimal
- Some operations on immediate numbers may be replaced to hard-coded into the instruction except MOVE (there is no MOVEI instruction)
 - Examples:

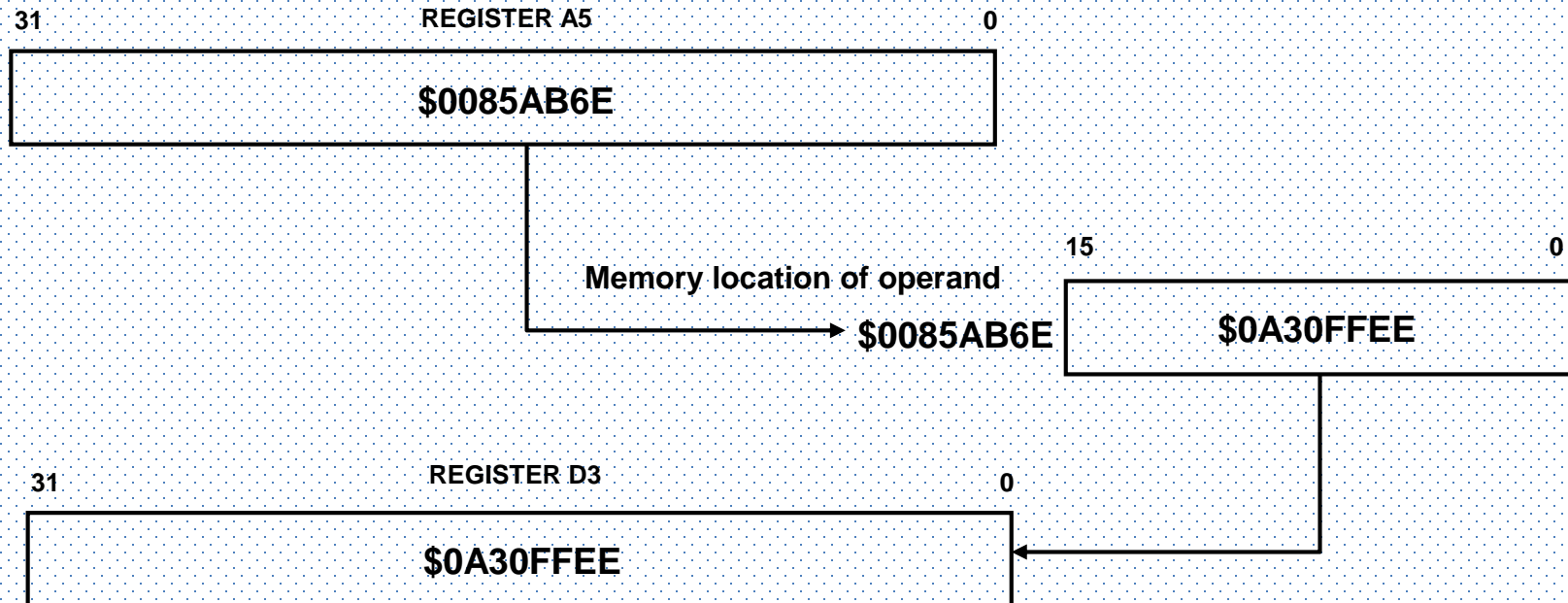
ADD.W	#\$5000,D6	→	ADDI.W	#\$5000,D6
SUB.W	#\$AAAA,D4	→	SUBI.W	#\$AAAA,D4
CMP.W	#\$BBBC,D5	→	CMPI.W	#\$BBBC,D5
OR.W	#\$CCCC,D2	→	ORI.W	#\$CCCC,D2

Comments on Immediate Addressing

- Immediate addressing is most frequently used to **initialize memory** and **variables** at boot-up
 - Even if you have data tables in memory, still need to initialize the pointers to these tables
- Immediate addressing is very important for **compiler optimizations**
 - It is an instruction that can be moved elsewhere in the program
 - Pipeline optimization
 - Since it does not access to memory to get the data, it is fast
- Immediate addressing can be wasteful if it's used continuously
 - Should only be used to load (initialize) a memory or register value

Address Register Indirect Addressing

- Instruction specifies one of the 68000's address registers
 - Example: **MOVE.L (A5),D3**
 - Load data register D3 with the contents of the memory location pointed to by address register A5
- The source address register contains the address of the operand
- The processor then accesses the operand ***pointed to*** by the address register
- Contents of the address register pointed to by A5 are copied to the data register



Address Register Indirect Addressing (2)

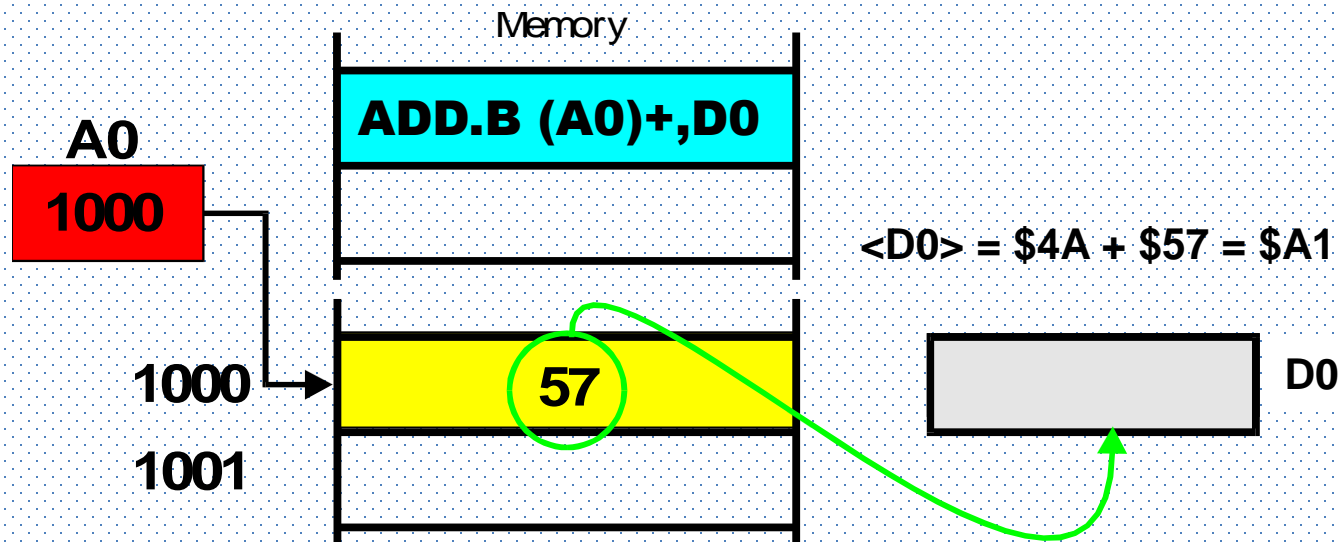
- Why use indirect addressing?
 - Allows us to **compute a memory location (pointer arithmetic)** during program execution instead of being fixed when the program is assembled
 - E.g., arrays, buffers, etc.
- What good is it?
 - If we compute an address, we can easily move up and down through tables and structures
 - “C++” pointers are indirect addresses

Address Register Indirect Addressing with Post-increment/Pre-decrement

- **Post-increment:** Automatically increment the value in the address register after operand is fetched from memory
- **Pre-decrement:** Automatically decrement the value before operand is fetched
 - *Increments/decrements by 1, 2, or 4 if operation is on bytes, words or longwords, respectively*
- Example: MOVE.L (A5)+,D3
 - <A5> = \$0085AB6E
 - <\$0085AB6E> = \$0A30FFEE
 - The opcode will cause \$0A30FFEE to be loaded into D3
 - The value in A5 will automatically be incremented to \$0085AB72 (*increment by a longword size*)
 - If this instruction was part of a loop, successive memory locations would be fetched and transferred to D3
- **Note that there is neither pre-increment nor post-decrement!**

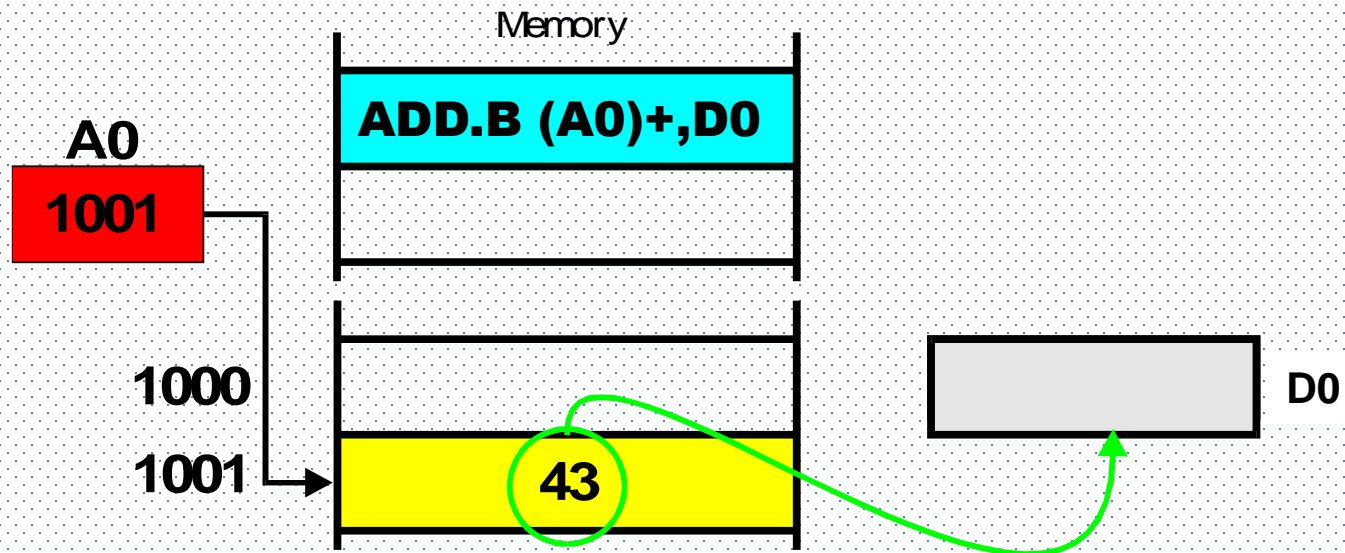
Auto-incrementing

- If the addressing mode is specified as (A0)+, the ***contents of the address register A0 are incremented after*** they have been used.
- Example: The address register contains 1000 and points at location 1000
- Address A0 register is used to access memory location 1000 and the contents of this location (i.e., 57) are added to D0



Auto-incrementing (2)

- After the instruction has been executed:
 - Content of A0 is incremented : $\langle A0 \rangle = 1001$
- If the instruction after **ADD.B (A0)+,D0** was a branch back instruction, then the result would keep adding the contents of successive memory locations to $\langle D0 \rangle$
- Auto-incrementing instructions are valuable because so much of memory is data in ordered lists



Example of Address Register Indirect Addressing

- The following fragment of code uses address register indirect addressing with post-incrementing to add together five numbers stored in consecutive memory locations.
- Note: **LEA (Load Effective Address)**
 - Puts an address into an address register

	MOVE.B	#5,D0	;Five numbers to add
	LEA	Table,A0	;A0 points at the numbers
	CLR.B	D1	;Clear the sum
Loop	ADD.B	(A0)+,D1	;REPEAT Add number to total
	SUB.B	#1,D0	
	BNE	Loop	;UNTIL all numbers added
	STOP	#\$2700	
*			
Table	DC.B	1,4,2,6,5	;Some data

Stack-based Operations

During a **PUSH** operation
the stack grows towards
lower memory addresses

Example:

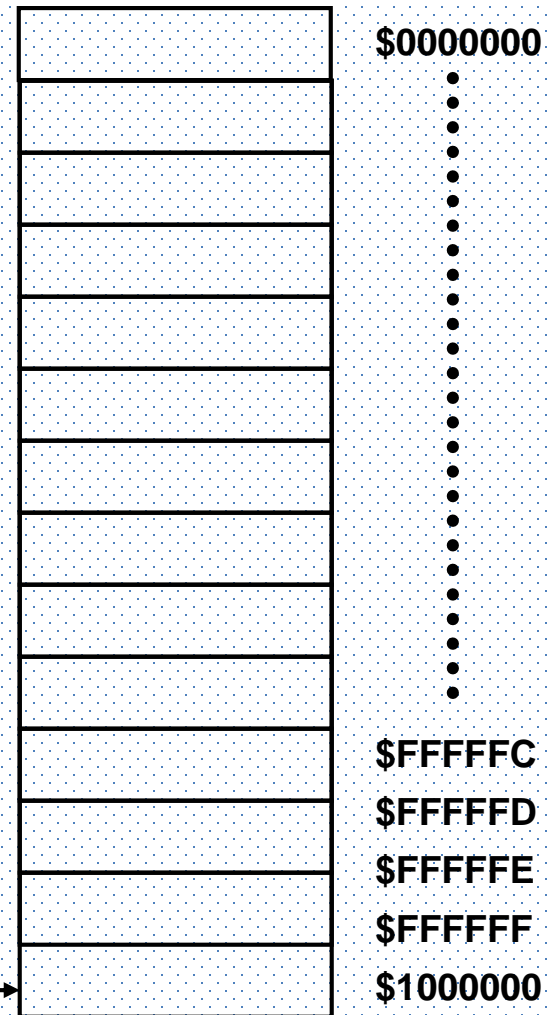
`MOVE.B D0,-(SP)`
will place a byte of data
on the stack.

During a **POP** operation
the stack shrinks towards
higher memory addresses

Example:

`MOVE.B (SP)+,D0`
will move a byte of data
from the stack to D0.

Initial value of the stack pointer, SP
The TOP of the stack



Top of Stack

Direct (Absolute) Addressing

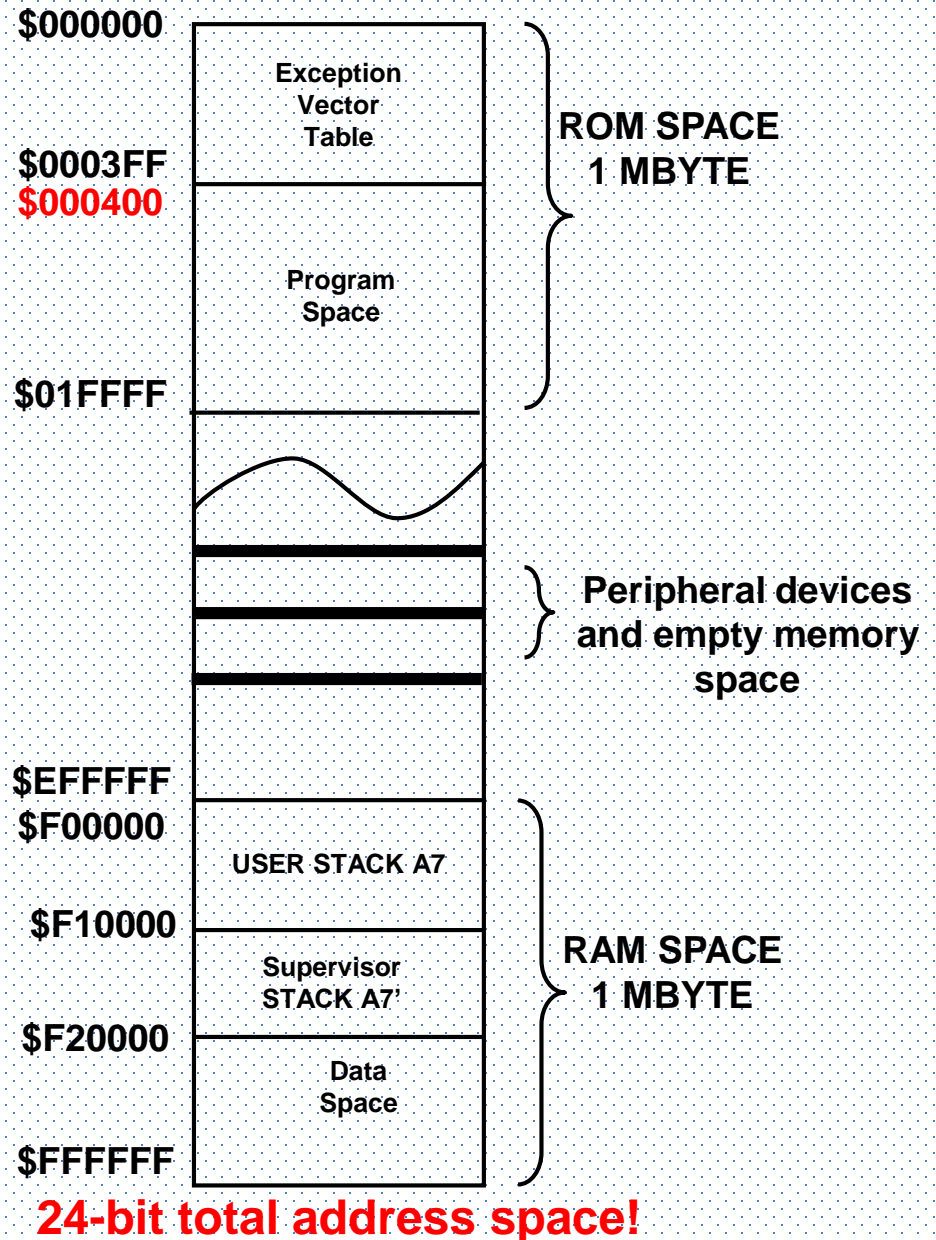
- Instruction provides *the memory address of the operand*
- Requires two memory accesses
 - 1) accessing the address and 2) accessing the actual operand (data)
- **Important:** \$ is for hex, % is for binary, no symbol is for decimal
- For example:
 - **CLR.B 1234** clears the contents of memory location 1234 (decimal)
- This mode is not commonly used in most programming situations
 - Prevents code from being relocated
 - Multiple memory accesses reduce performance
- 68000 allows for both word (16-bit) and long (32-bit) direct addressing

Before we start on Absolute Addressing!

- Two critical concepts
- **Address**: affects where is the data
 - Assembler can ***accept*** the specified address without a change
 - Assembler can ***change (by sign extension)*** the address by itself
- **Addressing Mode**: affects how the machine code is generated
 - Assembler can ***allow programmer to specify*** the Addressing Mode (fully or partially)
 - Assembler can also ***deduce*** the Address Mode by itself

Memory Map of a Typical 68000-based System

- Memory includes both ROM and RAM
- Your instructions should be in the memory
- 68000 hardware is designed to detect illegal memory accesses and cause a BUS ERROR exception to be taken



Absolute Addressing – Word

- **Mode 7: Absolute Address-Word: Mode 111, Subclass 000**
 - 68k has 24 bits for addressing, so technically it should have at least 24 bits for an address
 - 68k uses a 16-bit ***sign extended address***
 - Most Significant Bit (MSB) of the first extension word ***fills all the upper address bits from Bit 31 to Bit 15***

Absolute Addressing – Long

- **Mode 7: Absolute Addressing-Long: Mode 111, Subclass 001**
 - Actual address of the operand is contained in the **two words** following the instruction word
 - Data to be operated on is read from the 32-bit memory address formed by concatenating the high-order word and low-order word to form the full memory address
- **Problem of aliasing**
 - For the **68K processor**, the **address is 24-bit**, but it can accept **32-bit address! How???**
 - It might have an ***inadvertently duplicating address***
 - Where is the address \$AAFF00FF in the memory?
\$00FF00FF
 - So, try to keep address bits **Bit 31 to Bit 24 all zeros**

Absolute Addressing Range

- When assembling, the **range of an address** determines the actual **address mode**!

For example, **MOVE**

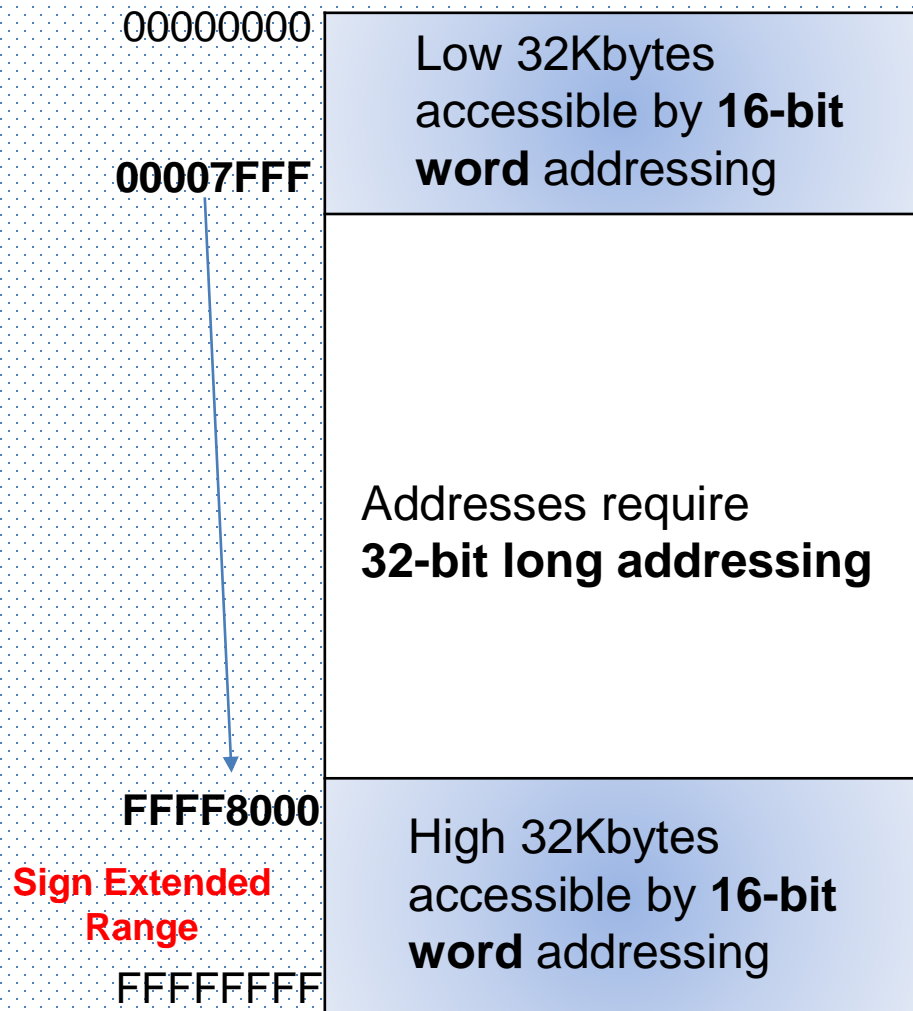
MOVE.B \$00004214, D5
→ Word addressing

MOVE.B \$FFFF4214, D5
→ Long addressing

MOVE.B \$0000A000, D5
→ Long addressing

MOVE.B \$A000, D5
→ Long addressing

MOVE.B \$FFFFA000, D5
→ Word addressing



Summary of Primary Addressing Modes

- **Register direct** addressing (**Data** or **Address** Register)
 - Used for variables that can be held in registers
- **Immediate** addressing
 - Used for constants that do not change
 - Initializing variables
- **Absolute** addressing (**Long** or **Word** Address)
 - Used for variables that reside in memory
 - Prevents programs from being relocated
- **Address register indirect** addressing
 - Most efficient way to address memory
 - Treats memory addresses as variable values
- **Address register indirect with post-increment or pre-decrement**
 - Used for sequential data manipulation
 - PUSH and POP operations to the stack

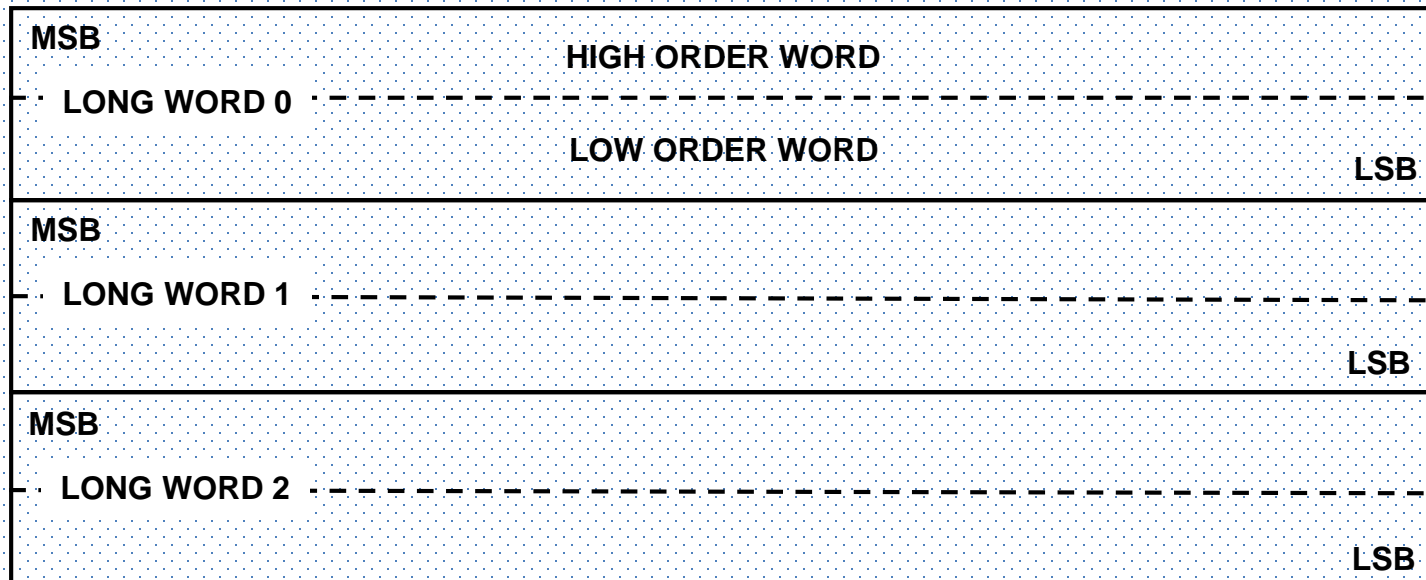
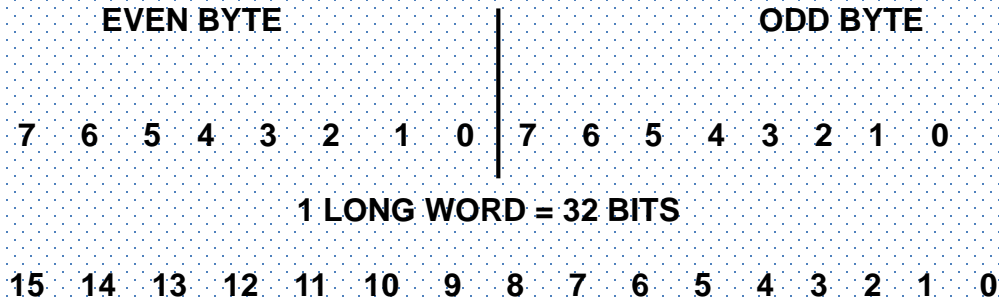
Non-aligned Accesses*

- For the 68000 microprocessor, only accesses to **even address** for ***word or long addressing operations*** are allowed
- For a misaligned transfer, more than one bus cycle may be required
- ***Instruction words*** must be aligned on word boundaries
- Misalignment of word or long-word operands can cause the microprocessor to perform **multiple bus cycles** for the operand transfer
 - Degrades processor performance
 - Means, you might not get a warning/error messages
- *Microprocessor's* performance is optimized when word and long-word memory operations are aligned on word and long-word boundaries

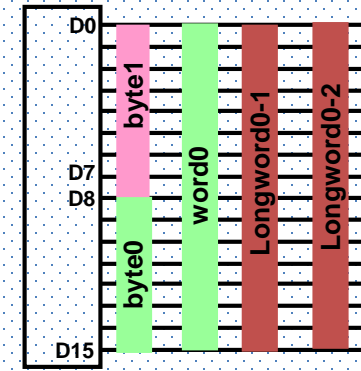
* Source: Freescale M68030 User's Manual

Memory Organization

- Storing 32-bit values in a 16-bit external memory

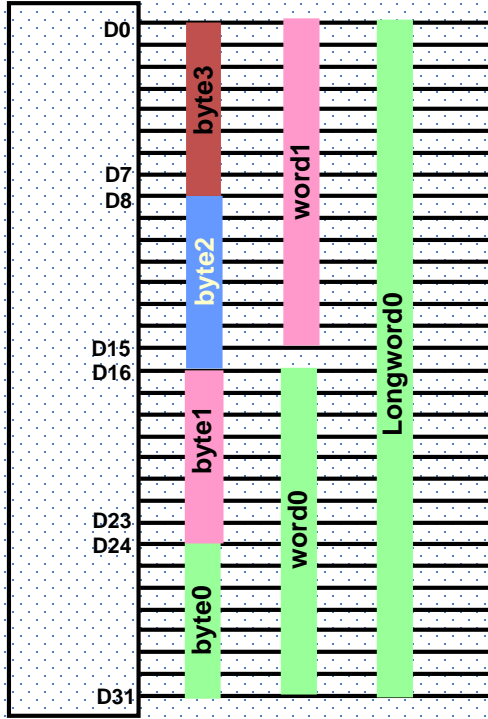


Memory Alignment

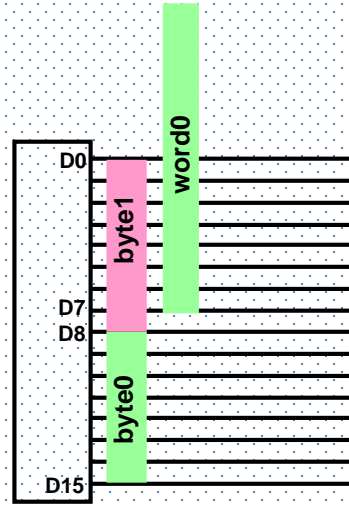


16-bit wide data bus

Aligned

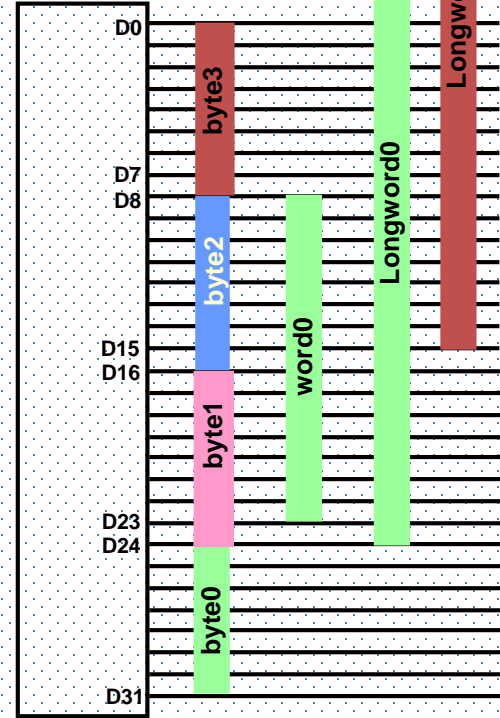


32-bit wide data bus



16-bit wide data bus

Non-aligned



32-bit wide data bus