Lab Reports

Reports should be in neat, concise form and turned in on time. The best reports will cover all vital information in detail without excessive verbiage on unnecessary details. Your report will typically follow the following format:

Title Page
Contains a typed one paragraph abstract and must be signed by you. The title page format you MUST use is on the following page.

Abstract (on the title page)
Explain in a few sentences what the program does. Describe how your program works and any special features of your program.

Check Out (on the title page - ONLY IF REQUIRED)
Selected labs may require you to demonstrate how your lab assignment works. If you are required to demonstrate your program this line will be used by a Teaching Assistant to indicate that it worked properly.

Program Description (separate page - ONLY IF REQUIRED)

- main body of your program
  Should include pseudo code or a flow chart (as appropriate, not all labs will need pseudo code or flow charts), input/output specifications, memory requirements, register and memory locations used, any algorithms used. Please cite any references you used for additional information. The program description should be concluded with a copy of your program listing; subroutine listings should be reserved for the description of each subroutine.

- any program subroutines
  Should include a flow chart (if appropriate), input/output specifications, memory requirements, register and memory locations used, any algorithms used. Please cite any references you used for additional information.

Questions (separate page - ONLY IF REQUIRED)
Rather than a formal lab report each lab assignment will be accompanied by a series of questions which you must answer on a separate sheet. These questions will be graded and will form a major part of the lab grade.

Program Listing (separate page - REQUIRED)
You MUST include a copy of your program with every lab assignment you turn in. This should be an assembler listing which includes the symbol table for your program.

EEAP 282 TITLE PAGE

Names:

_____
Typed

_____
SIgnature

No laboratory will be graded unless it is signed. By signing this page you are indicating that this lab and that the work described in the lab report is your own. Any complaints about grading should be directed to Prof. Merat or the lab T.A.'s.

TITLE:

_____

Abstract:

[ ] Checked by _____ _____

Basic computer operation and organization

From an engineering viewpoint a computer manipulates coded data and responds to events occurring in the external world. This is called a stored-program or von Neumann machine architecture.

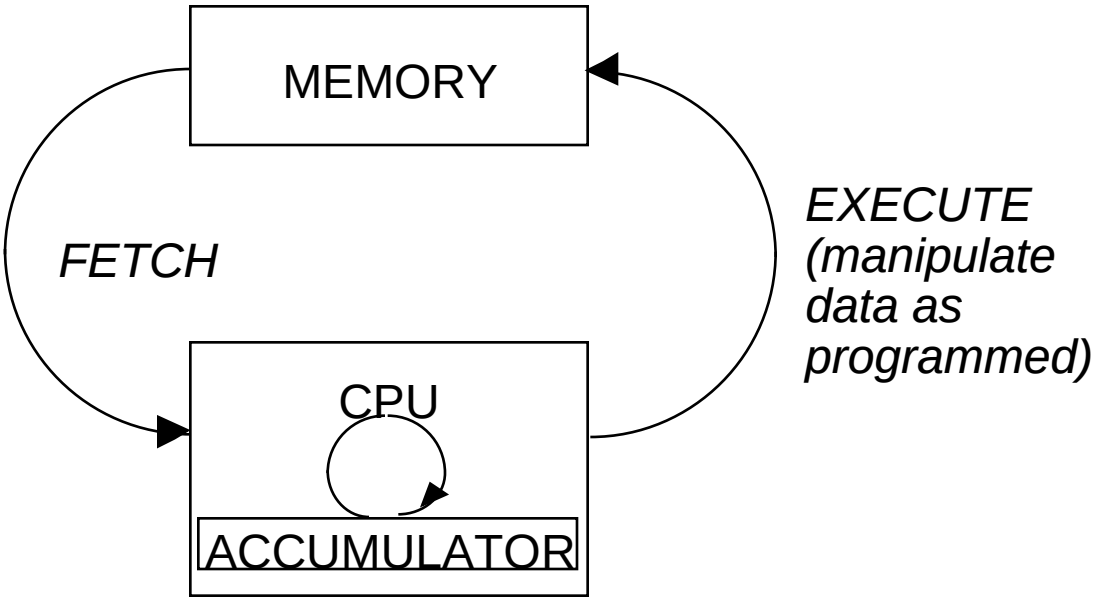
- memory - used to store both programs and data instructions (this is the core of the von Neumann architecture)
  - program instructions are coded data which tells the computer to do something, i.e. adding two numbers together
  - data is simply information to be used by the program, i.e. two numbers to be added together

    We need something to decode the memory and determine what represents instructions and what represents data
- central processing unit gets instructions and/or data from memory, decodes the instructions, and performs a sequence of programmed tasks

Nothing can occur simultaneously or instantaneously in a computer. Important operations are

- fetching instruction(s) from memory
- decoding the instruction(s)
- performing the indicated operations

This is the basic “fetch-execute” cycle



Problems with this diagram:

- what memory is being fetched?
- how does the computer tell program instructions from program data
- what happens when the program needs more than one piece of data?

Answers:

- put program instructions and data in separate areas of memory. These don't have to be well organized, but do need to be defined and can be intermixed. Usually program instructions are kept together.
- Internal pointers kept in special locations called registers in the CPU keep track of what data and program instructions are being referenced and/or fetched.
- The CPU has local storage in special locations called registers for temporary storage of data and/or instructions.

Keeping track of what instruction is being executed is so important that a special CPU register called the program counter is used to keep track of the address of the instruction to be executed.

Central Processing Unit

Control Unit	Arithmetic Logic Unit	Registers

Control unit:

- decodes the program instructions
- program counter which contains the location of the next instruction to be executed
- status register which monitors the execution of instructions and keeps track of overflows, carries, borrows, etc.

RISC reduced instruction set machine

- executes a simple set of instructions very fast

CISC complex instruction set (such as the 68000)

- has a powerful set of instructions which allows many complex operations to be represented as a single instruction

Arithmetic Logic Unit

- carries out the instructions decoded by the control unit

Older microprocessors had special registers called accumulators which had to be used for math calculations. Modern microprocessors such as the 68000 have general-purpose registers and do not have accumulators.

Memory

- ROM read-only memory

  non-volatile → all bits can be read, no bits can be changed
- RAM random access memory
  (not really a good name since almost all memory has random access capability)

  volatile → all bits can be read and/or written

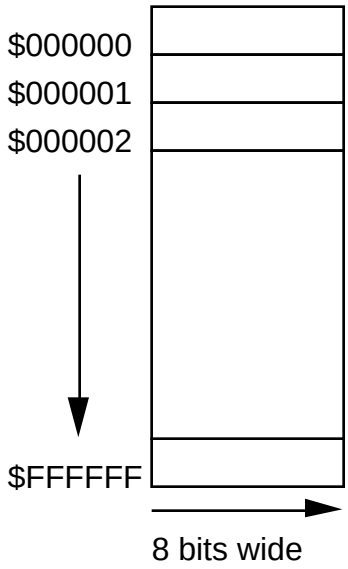
User concern with memory

- large computer usually don't even think about memory organization
- mini computer sequence of RAM, how much RAM, etc.
- micro computer memory constraints on RAM, ROM. Very limited address space.

Computer memory is always organized in a fixed manner:

- 16k x 1 bit
- 4k x 1 bit
- 8k x 8 bits typical of small microcomputers

Vertical grid organization of memory:



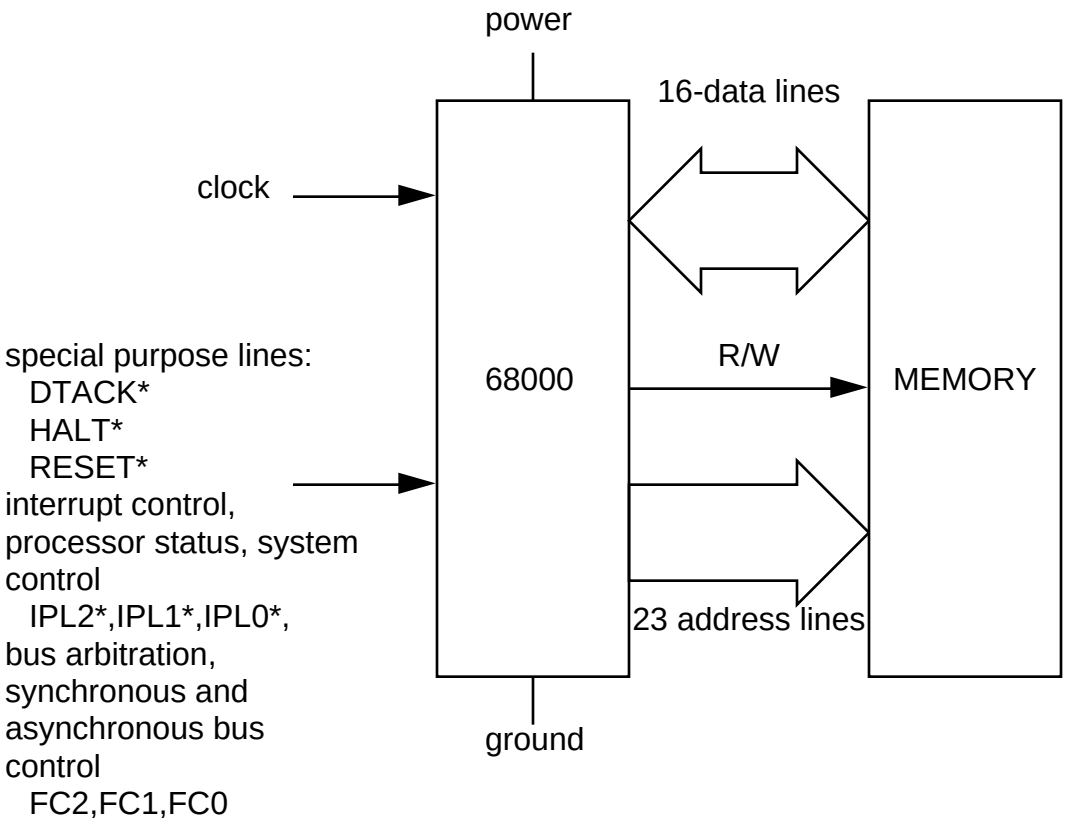
This diagram represents $FFFFFF = 2^{24} bytes of memory

- Any particular processor will have an x-bit address capacity:

64k	6502, Z-80, 8085, etc.
640k	8088
16MB	68000
lots	80386, 68030

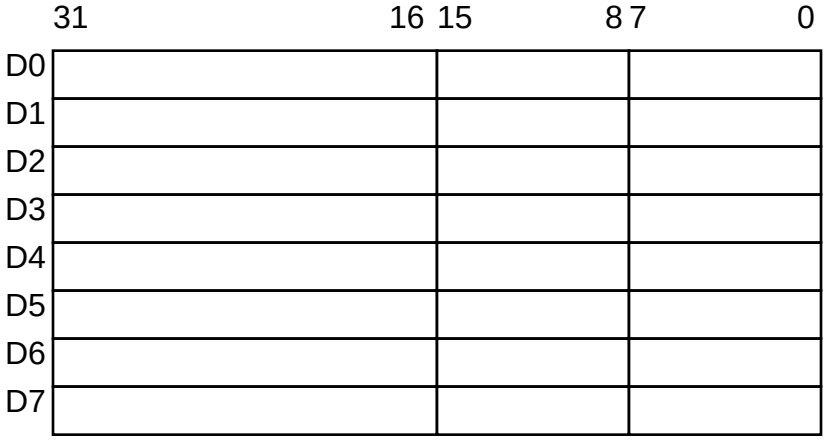
- memory addresses do not need to be contiguous
- ROM and RAM can be intermixed
- memory does not have to start at $000000 or end at $0FFFFF

68000 architecture:

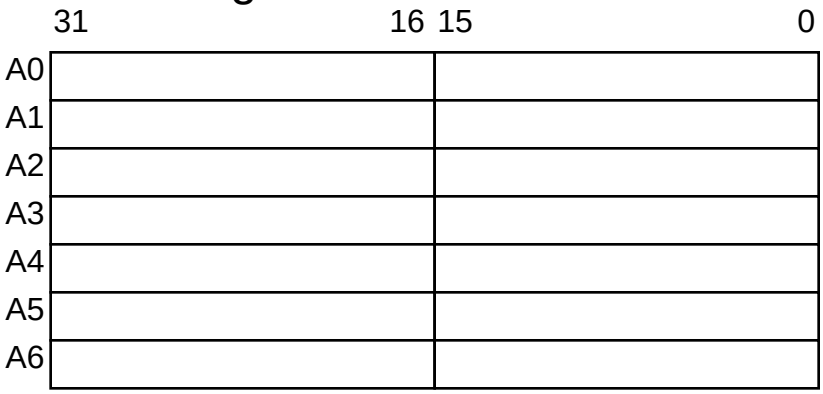


68000's internal register organization:

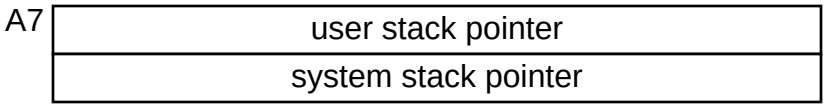
data registers:



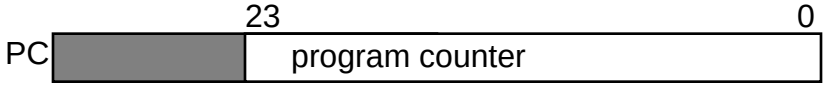
address registers:



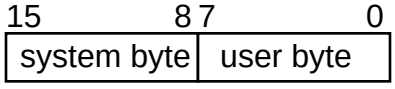
stack pointers:



program counter:

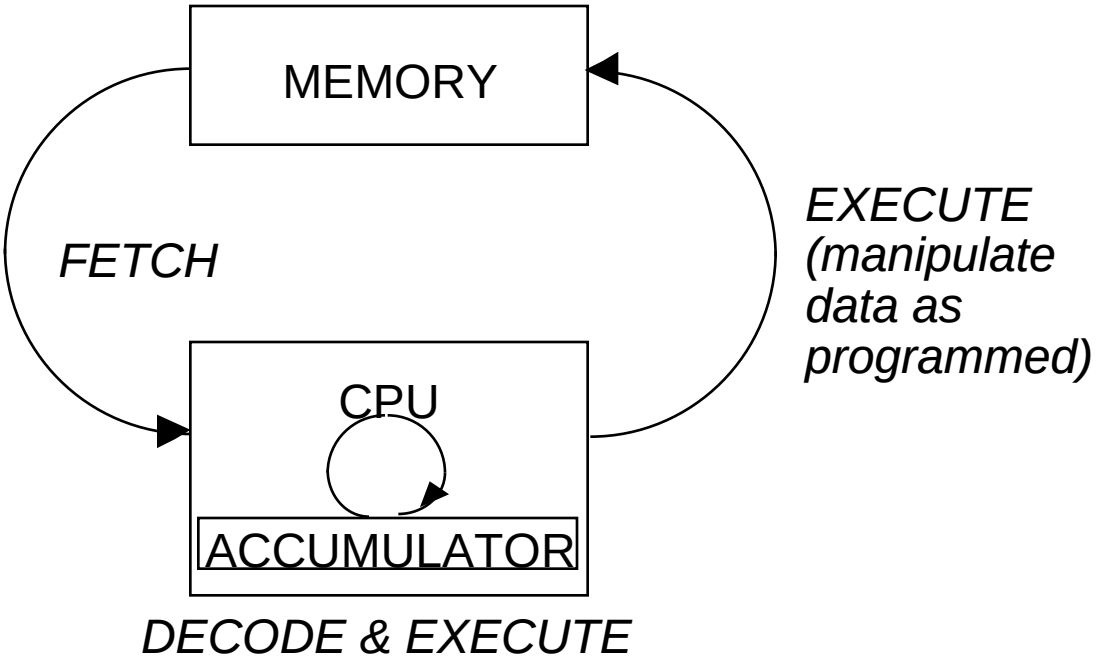


status register:



SOFTWARE ARCHITECTURE:

Fetch-execute revisited:



Basic fetch-execute cycle:

```
loop:   fetch_instruction
execute_instruction
goto loop.
```

Program counter:

Because the computer must keep track of instruction locations it uses the program counter to keep track of the address of the next instruction to be executed:

```
loop:   instruction_register = fetch_instruction(pc_address)
decode_instruction(instruction_register)
execute_instruction(instruction_register)
goto loop.
```

Instruction_register is an internal (to the processor) memory location (register) used to store coded data (instructions).

Instructions are coded data in the following format:

op_code source(s) destination next_instruction

The exact way this information is represented is different for every microcomputer. Present microcomputers typically code instructions in several somewhat simplified formats

op_code source1 source2/destination

or

op_code source/destination

where source, source1 and source2 identify where any needed data is to be obtained and the result (if any) is to be placed in a destination which is also the second source of needed data. For example, x:=x+y is represented in the first format as

ADD Y X

Add processing to decode instructions:

Our original fetch-execute cycle has now become more complex:

```
loop:   instruction_register = fetch_instruction(pc_address)
decode_instruction(instruction_register)
while extension_flag set do
    fetch additional information
    update pc_address
    end while.
execute_instruction(instruction_register)
update pc_address.
update status register.
goto loop.
```

Instructions which require more than one computer word to describe can be indicated by extension_flag and fetched until the instruction is complete.

The exact manner in which memory locations are represented is known as addressing and can directly effect the program execution speed of the computer.

Basic computer operation and organization

Use hex to represent memory locations as seen by the microcomputer. Memory can be organized as:

- bytes

address	memory
$0	byte 0
$1	byte 1
$2	byte 2
$3	byte 3
$4	byte 4
$5	byte 5

- words

address	memory		
$0	byte 0	byte 1	word 0
$2	byte 2	byte 3	word 1
$4	byte 4	byte 5	word 2
$6	byte 6	byte 7	word 3
$8			word 4
$A			word 5

- long words

address	memory			
$0	byte 0	byte 1	byte 2	byte 3
$4	byte 4	byte 5	byte 6	byte 7
$8	byte 8	byte 9	byte A	byte B
$C	byte C	byte D	byte E	byte F

Machine code (stored program execution)

z := x + yhigh level C or Pascal representation
where x,y,and z will represent words in memory

Data:

z is at memory address $1204
x is at memory address $1200
y is at memory address $1202

address	memory				contents
$1200	0001	0010	0011	0100	$1234
$1202	0100	0011	0010	0001	$4321
$1204	0000	0000	0000	0000	$0000

For some reason we decided to use words (16 bits) for all operations.

Instructions:

address	memory		meaning
$1000	3A	38	move a word from $1200
	12	00	to D5
$1004	DA	78	add the word at $1202 to
	12	02	the contents of D5
$1008	31	C5	move the contents of D5
	12	04	to $1204
$100C	4E	40	stop

Coding of an instruction. This is an opcode word as defined by Motorola for the 68000. See The MC68000 Programmer's reference Manual.

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
op code	size	destination	source	
		register	mode	mode	register
first four bits are the op code, indicates a move in this case	size code 01=byte 11=word 10=long word	Dn = data register Abs.w = absolute word Abs.L=absolute long word	how to manipulate data:	Dn = data register Abs.w = absolute word Abs.L=absolute long word

In this case the $3A38 instruction at $1000 would be interpreted as a MOVE instruction. (see p.4-116 of the Programmer's Reference Manual, current edition)

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

first four bits are the op code for a move	word length	D5 data register	put data into register	get data from memory address (word length) which follows EXTENSION WORD	word length address

Disassembly is the interpretation of coded instructions
The instructions we just used are interpreted as:

$3A38 1200

0011 1010 0011 1000		rewrite as binary
0011 101 000 111 000		regroup into the appropriate fields: op code, destination, and source
op code	00XX	indicates a MOVE, move data from source to destination
size	11	indicates word length
destination	101 000	
mode	000	indicates to a data register
register	101	indicates to register D5
source	111 000	
mode	111	indicates one of several possible modes: absolute and PC relative
register	000	indicates that Abs.W is being used requiring a 16-bit extension word

instruction is a word length move of the contents of $1200 to D5

$DA78 1202

1101 1010 0111 1000 rewrite as binary

The form of this instruction is different from that of the MOVE.

The 1101 op code indicates that this is an ADD.

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	register	Op-mode	Effective Address mode	register
op code, indicates an ADD in this case				specifies one of the eight data registers			

1101 101 001 111 000 regroup into the appropriate fields: op code, register, op-mode, and effective address

op code 1101 indicates an ADD, add binary

register 101 indicates register D5

op-mode 001 indicates word length add of the form (<Dn>) + (<ea>) → <Dn>

parentheses are used to indicate the contents of

effective address

mode 111 indicates absolute or PC relative addressing

register 000 indicates that Abs.W is being used

instruction is a word length add of the contents of $1202 to the contents of D5 with the result being put into D5

$31C5 1204

0011 0001 1100 0101		rewrite as binary
0011 000 111 000 101		regroup into the appropriate fields: op code, destination, and source
op code	00XX	indicates a MOVE, move data from source to destination
size	11	indicates word length
destination	000 111	
mode	111	indicates one of several possible modes: absolute and PC relative
register	000	indicates that Abs.W is being used
source	000 101	
mode	000	indicates from a data register
register	101	indicates from register D5

instruction is a word length move of the contents of D5 to $1204

Your textbook (p.54-55) lists several common instructions:

MOVE

copy 16-bit word specified by the source into the location specified by the destination operand

303C <number>	MOVE.W	#N,D0
33FC <number>,<address>	MOVE.W	#N,<address>
3039 <address>	MOVE.W	<address>,D0
33C0 <address>	MOVE.W	D0,<address>

ADD

adds the 16-bit word specified by the source and the 16-bit contents of the destination. The result is stored in the destination:

(<source>) + (<destination>) → <destination>

0640 <number>	ADDI.W	#N,D0
0679 <number>,<address>	ADDI.W	#N,<address>
D079 <address>	ADD.W	<address>,D0
D179 <address>	ADD.W	D0,<address>

SUB

subtracts the 16-bit word specified by the source from the 16-bit contents of the destination. The result is stored in the destination:

(<destination>) - (<source>) → <destination>

0440 <number>	SUBI.W	#N,D0
0479 <number>,<address>	SUBI.W	#N,<address>
9079 <address>	SUB.W	<address>,D0
9179 <address>	SUB.W	D0,<address>

Example: Chapter 3, problem 25

```
00010A      303C      000A
00010E      33C0      0000 020A
000114      0679      00C3 0000 020C
00011C      9079      0000 020C
000122      0679      0AF3 0000 020A
00012A      0640      00F8
...

00020A      0036
00020C      03FA
```

All numbers are in hex. Each line indicates an individual instruction.

Initially, (D0) = 0000 003B, ($020A)=$0036, ($020C)=$03FA

The disassembled program:

```
MOVE.W    #10,D0          put $A into D0
MOVE.W    D0,$020A        put the contents of D0
                          ($A) into address $20A
ADDI.W    #$C3,$020C      add $C3 to the contents
                          of $020C ($3FA) and put
                          the result into $20C
SUB.W     $20C,D0         subtract what's in $20C
                          from the contents of D0
                          ($A) and put the result in
                          D0
ADDI.W    #$0AF3,$020A    add $0AF3 to the
                          contents of $20A
ADDI.W    #$F8,D0         add $F8 to the contents
                          of D0
```

The detailed disassembly:

303C 000A

0011 0000 0011 1100		rewrite as binary
0011 000 000 111 100		regroup
op code	00XX	indicates a MOVE
size	11	indicates word length MOVE
destination	000 000	
mode	000	indicates data register
register	000	register D0
source	111 100	
mode	111	any of several modes
register	100	indicates immediate mode, designated as Imm, i.e. a constant contained in an extension word

MOVE.W #10,D0

33C0 0000 020A

0011 0011 1100 0000		rewrite as binary
0011 001 111 000 000		regroup
op code	00XX	indicates a MOVE
size	11	indicates word length MOVE
destination	001 111	
mode	111	indicates any of several modes
register	001	indicates Abs.L, a long word address requiring two extension words
source	000 000	
mode	000	indicates a data register
register	000	register D0

MOVE.W D0, $0000 020A

0679 00C3 0000 020C

0000 0110 0111 1001 rewrite as binary
0000 0110 01 111 001 regroup

op code 0000 0110 indicates an ADDI
size 01 word operation, i.e. one 16-bit extension word

effective address
mode 111
register 001 indicates Abs.L, requires a 32-bit longword address, i.e. two 16-bit extension words

ADDI.W #$C3, $0000 020C

9079 0000 020C

1001 0000 0111 1001 rewrite as binary
1001 000 001 111 001 regroup

op code 1001 indicates a SUB
register 000 indicates D0
op-mode 001 word operation, i.e. (<Dn>)-(<ea>)→<Dn>

effective address
mode 111
register 001 indicates Abs.L, requires a 32-bit longword address, i.e. two 16-bit extension words

SUB.W $0000 020C, D0

0679 0AF3 0000 020A

0000 0110 0111 1001 rewrite as binary
0000 0110 01 111 001 regroup

op code 0000 0110 indicates an ADDI
size 01 word operation, i.e. one 16-bit extension word
effective address
mode 111
register 001 indicates Abs.L, requires a 32-bit longword address, i.e. two 16-bit extension words

ADDI.W #$0AF3, $0000 020A

0640 00F8

0000 0110 0100 0000 rewrite as binary
0000 0110 01 000 000 regroup

op code 0000 0110 indicates an ADDI
size 01 word operation, i.e. one 16-bit extension word
effective address
mode 000 indicates data register
register 000 indicates D0

ADDI.W #$F8, D0

The final program is then
If (D0)=$3B, ($20A) = $36, ($20C) = $3FA

MOVE.W	#10,D0	(D0)=$A
MOVE.W	D0,$020A	($20A)=$000A
ADDI.W	#$C3,$020C	($020C) = ($020C)+$C3 = $3FA + $C3 = $4BD
SUB.W	$20C,D0	(D0) = (D0)-($20C) = $A-$4BD = $ FB4D
ADDI.W	#$0AF3,$020A	($20A) = ($20A)+$0AF3 = $A + $0AF3 = $0AFD
ADDI.W	#$F8,D0	(D0) = (D0)+$F8= $FB4D + $F8 = $FC45

Math:

000A	0000 0000 0000 1010
04BD	0000 0100 1011 1101
complement	1111 1011 0100 0010
add 1	1111 1011 0100 0011
000A	0000 0000 0000 1010
-04BD	1111 1011 0100 0011
FB4D	1111 1011 0100 1101
+00F8	0000 0000 1111 1000
FC45	1111 1100 0100 0101

ADD — Add Binary — ADD

Operation: (Source)+(Destination)→Destination

Assembler Syntax ADD <ea>,Dn
ADD Dn,<ea>

Attributes: Size=(Byte,Word,Long)

Description: Add the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.
X Set the same as the carry bit.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	Register	Op-Mode	Effective Address Mode	Register

Instruction Fields:

Register field — Specifies any of the eight data registers.
Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>)+(<ea>)→<Dn>
100	101	110	(<ea>)+(<Dn>)→<ea>

Effective Address field — Determines addressing mode:

a. If the location specified is a source operand, then all addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	register number
An*	001	register number
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	111	010
d(PC,Xi)	111	011
Imm	111	100

* Word and Long only.

b. If the location specified is a destination operand, then only alterable memory addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	—	—
d(PC,Xi)	—	—
Imm	—	—

Notes: 1. If the destination is a data register, then it cannot be specified by using the destination <ea> mode, but must use the destination Dn mode instead.
2. ADDA is used when the destination is an address register. ADDI and ADDQ are used when the source is immediate data. Most assemblers automatically make this decision.

MOVE — Move Data From Source to Destination — MOVE

Operation: (Source) → Destination

Assembler Syntax MOVE <ea>,<ea>

Attributes: Size=(Byte,Word,Long)

Description: Move the content of the source to the destination location. The data is examined as it is moved, and the condition codes set accordingly. The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.
X Not affected.

Instruction Format:

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
0	0	Size	Destination Register	Destination Mode	Source Mode	Source Register

Instruction Fields:

Size field — Specifies the size of the operand to be moved.
- 01 — byte operation.
- 11 — word operation.
- 10 — long operation.

Destination Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	register number
An	—	—
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	—	—
d(PC,Xi)	—	—
Imm	—	—

Source Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	register number
An*	001	register number
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	111	010
d(PC,Xi)	111	011
Imm	111	100

* For byte size operation, address register direct is not allowed.

Notes: 1. MOVEA is used when the destination is an address register. Most assemblers automatically make this distinction.
2. MOVEQ can also be used for certain operations on data registers.

ADDI — Add Immediate — ADDI

Operation: Immediate Data + (Destination)→Destination

Assembler Syntax ADD #<data>,<ea>

Attributes: Size=(Byte,Word,Long)

Description: Add the immediate data to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word, or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.
X Set the same as the carry bit.

Instruction Format:

15	14	13	12	11	10	9	8	7 6	5 4 3	2 1 0
0	0	0	0	0	1	1	0	Size	Effective Address Mode	Register
Word Data (16 bits)								Byte Data (8 bits)		
Long Data (32 bits, including previous word)										

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	register number
An	—	—
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	—	—
d(PC,Xi)	—	—
Imm	—	—

Immediate field — (Data immediately following the instruction):
If size=00, then the data is the low order byte of the immediate word.
If size=01, then the data is the entire immediate word.
If size=10, then the data is the next two immediate words.

SUB — Subtract Binary — SUB

Operation: (Destination)-(Source)→Destination

Assembler Syntax SUB <ea>,Dn
SUB Dn,<ea>

Attributes: Size=(Byte,Word,Long)

Description: Subtract the source operand from the destination operand, and store the result in the destination. The size of the operation may be specified to be byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.
X Set the same as the carry bit.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	1	Register	Op-Mode	Effective Address Mode	Register

Instruction Fields:

Register field — Specifies any of the eight data registers.
Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>)-(<ea>)→<Dn>
100	101	110	(<ea>)-(<Dn>)→<ea>

Effective Address field — Determines addressing mode:

a. If the location specified is a source operand, then all addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	register number
An*	001	register number
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	111	010
d(PC,Xi)	111	011
Imm	111	100

* For byte size operation, address register direct is not allowed

b. If the location specified is a destination operand, then only alterable memory addressing modes are allowed as shown:

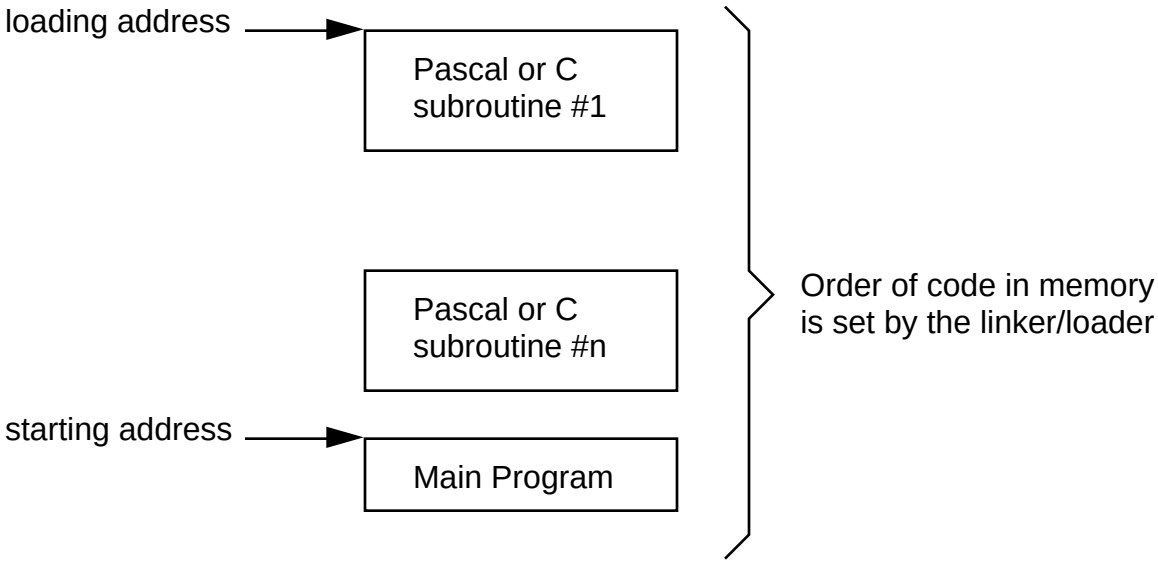
Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	register number
(An)+	011	register number
-(An)	100	register number
d(An)	101	register number

Addressing Mode	Mode	Register
d(An,Xi)	110	register number
Abs.W	111	000
Abs.L	111	001
d(PC)	—	—
d(PC,Xi)	—	—
Imm	—	—

Notes: 1. If the destination is a data register, then it cannot be specified by using the destination <ea> mode, but must use the destination Dn mode instead.
2. SUBA is used when the destination is an address register. SUBI and SUBQ are used when the source is immediate data. Most assemblers automatically make this distinction.

Running Programs (See Section 3.5.2 of your text)

starting address set by the assembler or the linker.
loading address set by the linker.



The Program Counter (PC) MUST be set before you can run a program.

You can do this in the debugger in several ways:

1. **M**emory **R**egister @PC=1000h
   <You cannot use $1000 in the debugger>

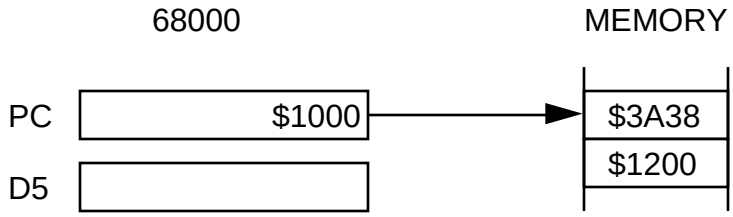
2. **P**rogram **S**tep **F**rom 1000h
   Program **S**tep

You can also automatically set the PC in the assembler

```
<label>   <your code begins here>
          rest of your program
          end      <label>
```

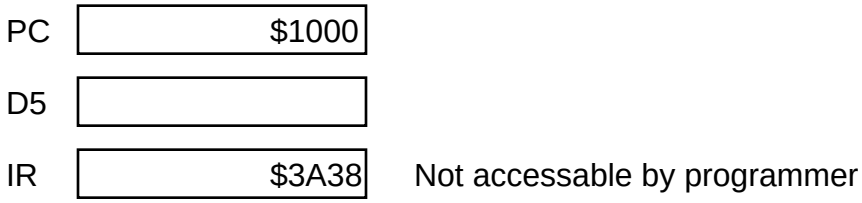
where <label> is any name you want. It will be used to set the initial PC value.

Fetch and execute for a simple example:
Initially:

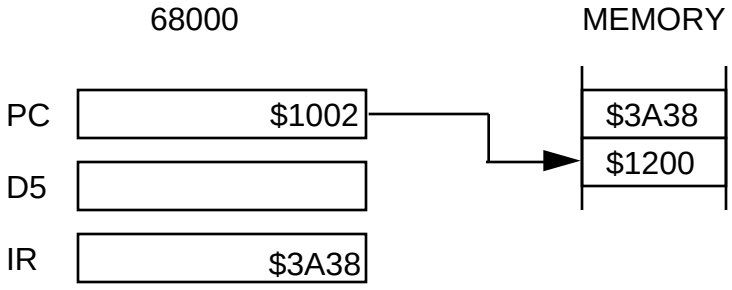


Fetch:

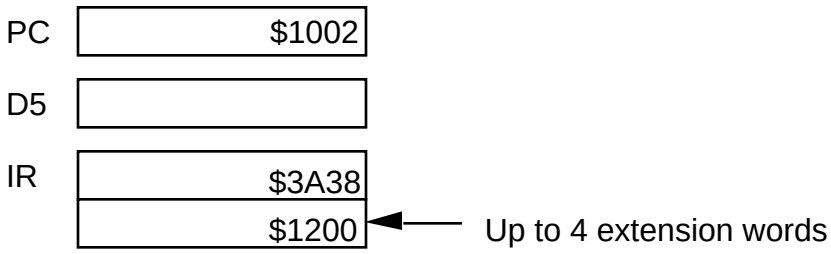
- Fetch instruction pointed to by PC and move into internal instruction register (not user accessable)



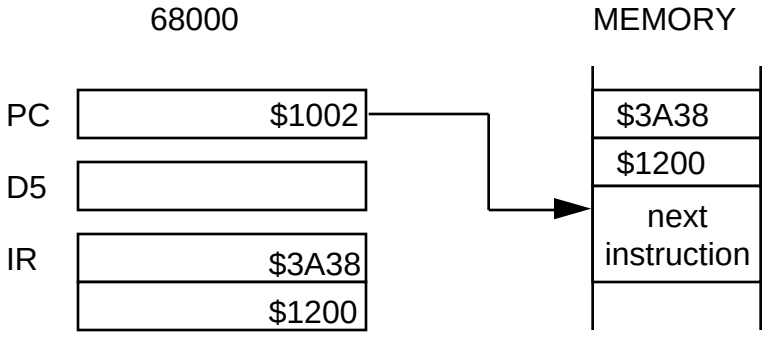
- Increment the PC by two bytes (one word due to bus width)



- Decode the instruction. Effective address field indicates an extension so fetch it.



- Increment the PC by two bytes



- Execute the instruction
  uses address in extension word to fetch ($1200)
- Repeat

You can look up how long it takes instructions to execute:

- MOVE.W $1200,D5
  This has one extension word, addressing modes are xxx.W and Dn
  From Table D.2 in Programmer's Reference Manual, Appendix D

Source	Destination	Clock periods (read/writes)
(xxx).W	Dn	16(4/0)

- ADD.W $1202,D5
  From Table D.4 in Programmer's Reference Manual, Appendix D

Instruction	Size	op <ea>,Dn
ADD	word	4(1/0)+

  Now use Table D.1 to compute the cycles required to compute the effective address and execute any fetches

(xxx).W	Absolute short	word=8[2/0]

- MOVE.W D5,$1204
  Now use Table D.2 to compute the cycles required to compute the effective address and execute any fetches

Dn	(xxx).W	12(2/1)

More detailed example:

Assume PC=$100

instruction	address	machine code	mnemonics
1	000100	3039 0000 2000	MOVE.W $2000,D0
2	000106	0679 0012 0000 2004	ADDI.W #18,$2004

program execution

read cycle	put (PC) on address bus, (CPU) put 3039 on data bus (memory)
	decode 3039, increment PC to 102
read cycle	put 102 on address bus read 0000 from memory, PC → 104
read cycle	put 104 on address bus read 2000 from memory, PC → 106
read cycle	put 2000 on address bus read ($2000) pc stays at 106

This is instruction:
MOVE.W source xxx.L destination Dn

From Table D.2, it takes 16 clock cycles (4 reads/0 writes) to execute.

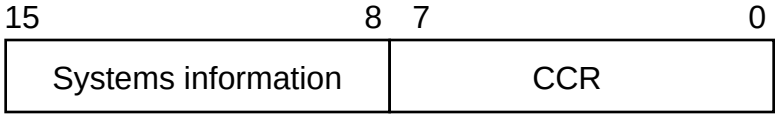
YOU WANT FAST INSTRUCTIONS WHENEVER POSSIBLE, i.e. NO extension words.

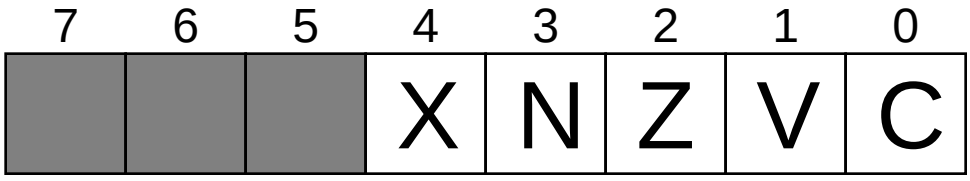
Example:
MOVEQ does not use an extension word.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1				0	source							
op code				Dn = data register				8 bit constant. -128 to +127							

Section 5.1 The Condition Code Register (CCR)

16-bit status register





bits	function
7,6,5	not used
4	extend bit retains carry bit for multi-word arithmetic
3	negative set to 1 if instruction result is negative, set to 0 if positive
2	zero set to 1 if result is 0
1	overflow set if signmed overflow occurs
0	carry/borrow

NOTE:

MOVE	<ea>,CCR	only effects CCR
MOVE	<ea>,SR	effects entire SR
MOVE	CCR,	only effects CCR (upper byte is set to all 0's)
MOVE	SR,	entire SR

These are word length instructions.

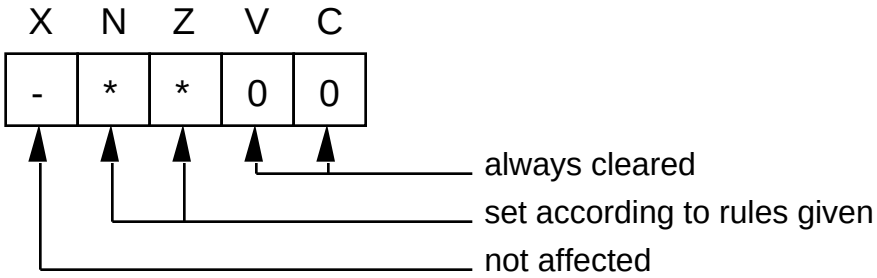
The System Part of the SR

bits	function
11,12,14	not used
8,9,10	interrupt mask a priority scheme to determine who has control of the computer
13	supervisor set to 0 if user, set to 1 if supervisor
15	trace set to 1 if program is to be single stepped

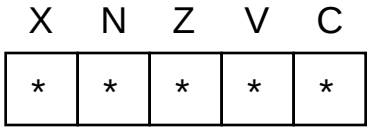
Any unused (reserved) bit is always set to zero!
You can always read the entire SR, but you can only modify the system byte of the SR in supervisor mode.

Examples:

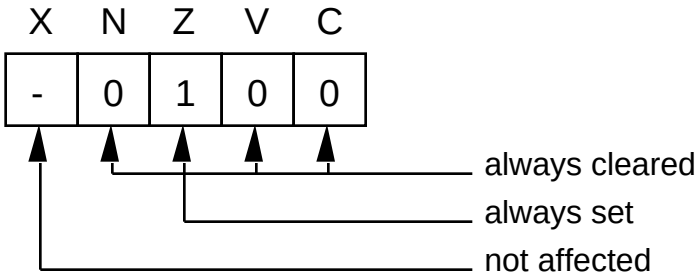
MOVE



ADD



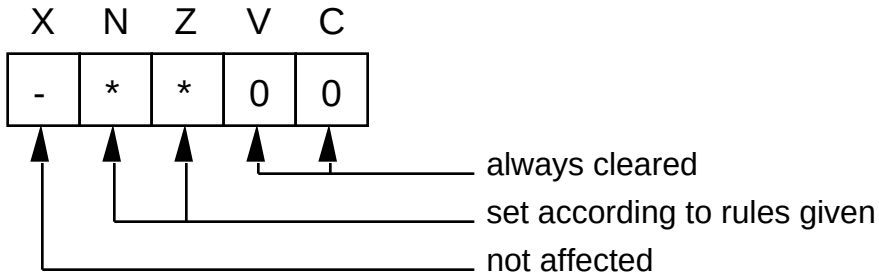
CLR



Even a MOVE instruction effects the status register

overflow	V → 0	
carry	C → 0	
negative	N → *	Depends on number being moved
zero	Z → *	Depends on number being moved
extend	X → -	Not changed

MOVE



Examples:

MOVE.W LENGTH,D1

if (LENGTH)=0 then (Z) → 1

MOVE.B #$FF,D1

(Z) → 0, (N) → 1

How an instruction effects the SR is shown in the Programmer's Reference Manual and on the Programmer's Reference Card

Examples of status flags (all word length)

Consider an add instruction of the form
ADD.W D0,D3

15_{10}	addition of	$0000\ 0000\ 0000\ 1111_2$	Overflow
15_{10}	two	$0000\ 0000\ 0000\ 1111_2$	V=0
30_{10}	positive signed numbers	$0000\ 0000\ 0001\ 1110_2$	Carry C=0

126_{10}	addition of	$0000\ 0000\ 0111\ 1110_2$	Overflow
3_{10}	two	$0000\ 0000\ 0000\ 0011_2$	V=1
129_{10}	positive signed numbers	$1000\ 0000\ 0000\ 0001_2$ (this is -127_{10})	Carry C=0

The result 129_{10} is out of range for a signed 16-bit number. As a result, the sign of the result does not match that of the operands and signed overflow occurrs.

-2_{10}	addition of	$1111\ 1111\ 1111\ 1110_2$	Overflow
-3_{10}	two	$1111\ 1111\ 1111\ 1101_2$	V=0
-5_{10}	negative integers with no overflow	$1111\ 1111\ 1111\ 1101_2$ with a carry	Carry C=1

The signs match so no signed overflow occurred.

-127_{10}	addition of	$1000\ 0000\ 0000\ 0001_2$	Overflow
-5_{10}	two	$1111\ 1111\ 1111\ 1011_2$	V=1
-132_{10}	negative integers with overflow	$0111\ 1111\ 1111\ 1011_2$ with a carry	Carry C=1

The result -132_{10} is out of range for a signed 16-bit so the signs don't match and signed overflow occurred. In addition a carry occurred.

128_{10}	addition of	$1000\ 0000\ 0000\ 0000_2$	Overflow
15_{10}	two	$0000\ 0000\ 0000\ 1111_2$	V=0
143_{10}	positive	$1000\ 0000\ 0000\ 1111_2$	Carry
	unsigned integers		C=0

128_{10}	addition of	$1000\ 0000\ 0000\ 0000_2$	Overflow
143_{10}	two	$1000\ 0000\ 0000\ 1111_2$	V=1
271_{10}	positive	$0000\ 0000\ 0000\ 1111_2$	Carry
	unsigned integers	with a carry	C=1

The same analysis can be applied to subtraction:

SUB.W D0,$1200
where (D0)=$0F13 and ($1200)=$01C8

456_{10}	subtraction of	$ 01 C8	Overflow V=0
-3859_{10}	two signed	+ $ F0 ED	Carry C=0
-3403_{10}	numbers	$F2 B5	

Note that since the result is negative this would be sign extended if to long word if the instruction length were .L

Simple assembly language example:

PROGRAM 4.1 of text

```
field#1   field#2       field#3
          ORG           $1000        ;start program at this
                                     memory location
* CODED   INSTRUCTIONS
MAIN:     MOVE          DATA,D5      ;get first number, use
                                     symbol for it
          ADD           NEXT,D5      ;add NEXT to D5
          MOVE          D5,ANSWER    ;save result
          TRAP          #0           ;this will stop the
                                     program, but will not do
                                     what it is supposed to
                                     do

          ORG           $1200
* DATA    DECLARATIONS
DATA:     DC            $1235        ;put $1235 into location
NEXT:     DC            $4321        ;put $4321 into location
ANSWER:   DS            1            ;reserves one word of
                                     memory
                                     ; could also have used
                                     DC.W $0

          END           MAIN         ;stop assembler
```

NOTES:

1. Program in text uses HEXIN and HEXOUT. These do not work in our debugger. You will be introduced to their equivalent in Lab #3.
2. Use of symbols in programs is highly recommended to make them more readable.
3. Symbol table contains a symbol field, type field, and a value field.
4. Use of colons (:) following labels is optional if the label's name begins in column 1.
5. Use of the semi-colon to begin a comment is also optional.

You can write programs in machine code but that is:

- tedious
- slow
- prone to errors

So, use programs to make process more efficient

source program	→	assembler	→	linker/loader
(uses mnemonics for machine code)		translates mnemonics into machine code; calculates addresses, etc.		references any system calls; loads program into memory

Cross-assemblying is when you assemble on another machine, say an 80286, using a program to generate 68000 machine code.

Down-line loading is when you transfer object code between machines. When you transfer your code to the in-circuit emulator you are downloading.

Example of mnemonic instruction:

MOVE	.W	D0,	D3
op code mnemonic for a move	word length	from D0	to D3

It would take a great deal of effort to calculate addresses all the time so a good assembler allows you to assign names to program locations and constants.

For example,

```
MOVE.W    D5, DATA
```

instead of

```
MOVE.W    D5,$1200
```

Section 4.2 of textbook describes program organization

Labels are implied if they precede a valid instruction code and begin in column 1. Labels are defined if they are followed by a colon.

```
implied:
LOOP    MOVE.W      D5,DATA
defined:
LOOP:   MOVE.W      D5,DATA
```

Comments are implied if they follow a valid instruction on a line. In some assemblers they must be preceded by a semi-colon (;) or asterisk (*). Comments are defined if they begin with a "*" in column 1.

Assembler directives tell the assembler to perform a support task such as beginning the program at a certain memory location.

ORG	tells the assembler where that section of the program is to go in memory
END	end of entire program (including data). Put the starting label after the END for automatic loading of the starting PC.
DC	puts a set of data into meory (define constant)
DS	reserves specified memory locations

Many assembler directives and instructions can operate on bytes, words or long words. What is to be acted on is indicated by the suffix:

.B	byte length operations
.W	word length operations (almost always assumed)
.L	long word operations
$	indicates a hex number, decimal is assumed otherwise (Does not work in debugger.)
h	follows number in debugger to indicate hex. Hex constants in debugger must begin with a number.
#	preceded an immediate constant
D0-D7	data registers
A0-A7	address registers

Some assemblers will print out a symbol table which will list all variables, including labels, and their values.

The EQU directive (F&T, Section 6.3.2)
Directly puts something in the symbol table. Such a symbol is NOT a label, but a constant! Use EQU to define often-used constants.

```
LENGTH    EQU    $8
MASK      EQU    $000F
DEVICE    EQU    $3FF01
```

can also use the format

```
LABEL     EQU    *
```

which enters the current value of the PC as its value

SET is the same as EQU but you can re-define the value of the variable later in your program.

XREF tells the assembler/linker that the following symbol(s) are defined in another program module (file)

XDEF tells the assembler/linker that the following symbol(s) are defined in this program module for use (reference) by another program module. Described on p.204-205 of F&T.

```
DATA      EQU     $6000
PROGRAM   EQU     $4000

ORG       DATA
* TABLE OF FACTORIALS
FTABLE    DC      1           0!=1
          DC      1           1!=1
test      DC      2           2!=2
          DC      6           3!=6
          DC      24          4!=24
          DC      120         5!=120
          DC      720         6!=720
          DC      5040        7!=5040
VALUE     DS.B    1           input to factorial function
          DS.B    1           align on word boundary
RESULT    DS.W    1           result of factorial



ORG       PROGRAM
main
*                             PUT TABLE BASE ADDRESS IN A0
          NOP
          NOP
          MOVEA.W #FTABLE,A0  gets $6000
          MOVEA.W FTABLE,A1   gets $1
          MOVE.W  #FTABLE,A2  gets $6000
          MOVE.W  #FTABLE,D0  gets $6000
          MOVE.W  FTABLE,D1   gets $1

          MOVE.W  test(A0),D3 test displacement


          MOVE.W  #5,VALUE    inputto fact is 5

fact      MOVE.W  VALUE,D5    get input
          ADD.W   D5,D5       double for word offset
          LEA     FTABLE,A3   get base address
          MOVE.W  0(A3,D5),D6 get result
          MOVE.W  D6,RESULT   output

          END main
```

How to run your program:

as68k Example1

Assumes a file with the full name Example1.s is present. Produces an output Example1.o

This is a two-pass assembler. The first pass reads the entire program, computes all instruction addresses, and assigns addresses to labels. The second pass converts all instructions into machine code using the label addresses.

ld68k -o Example1 Example1

The first file name following the -o is the output file which will automatically be named Example1.x; the second file name is the input which is assumed to be Example1.o

db68k Example1

You must set the PC in the debugger to run your program. You can do this in the debugger in several ways:

1. **M**emory **R**egister @PC=1000h
   <You cannot use $1000 in the debugger>

2. **P**rogram **S**tep **F**rom 1000h
   Program **S**tep

You can also automatically set the PC in the assembler

```
<label>   <your code begins here>
          rest of your program
          end     <label>
```

where <label> is any name you want. It will be used to set the initial PC value.

SOME USEFUL DEBUGGER COMMANDS ARE:

Debugger **Q**uit **Y**es <return>	Quits the debugger.
Window **A**ctive **A**ssembly **R**egisters <return>	Removes the journal window and shows the Status Register.
Program **S**tep **F**rom **1000h** <return>	Resets the code window to $1000 and executes the instruction at $1000. Note that only one instruction is executed.
Program **S**tep <return>	Executes the instruction currently highlighted. This command following the initial Program Step From 1000h would execute the instruction at $1006.
Memory **R**egister @**PC=1000h** <return>	Sets the current value of the PC to $1000, i.e. this is the next instruction to be executed.
Memory **R**egister @**A3=1000h** <return>	Sets the current contents of A3 to $1000. Can be used for all registers including SR.
Expression **M**onitor **V**alue @**A1**	Continuously displays the value of A1 in the monitor window.

NOTE: The @ indicates a reserved symbol such as the name of a data or address register, the PC or the SR.

COMMENTS ON MC68000 INSTRUCTIONS IN LAB#2

* These instructions operate on data registers

```
        MOVE.W      #$FFFE,D0       ;you will get different
                                    results if you use .L
                                    instructions
        ADD.W       #1,D0
        ADD.W       #1,D0
        ADD.W       #$FFFE,D0
        ADD.W       #2,D0
```

* These instructions operate on address registers

```
        LEA         $2000,A0
        MOVE        #$2000,A1       ;this is not an
                                    allowed instruction,
                                    assembler will
                                    automatically
                                    convert to MOVEA
        MOVE        D0,(A0)         ;address register
                                    indirect
```

If you look at the MOVE instruction, An is not allowed. You must use a MOVEA which can only have an address register as a destination. The instruction
MOVEA <ea>,A1
is the only form of the MOVE that can put data into an address register. The size of the operator can be .W or .L Word size operands are sign extended to 32 bits before any operations are done.

The LEA instruction is subtly different than a MOVEA—it computes <effective address> and puts that into An. Only a long form of the instruction is allowed.

MOVEA	converts addresses into constants
LEA	generates position independent code using PC relative address modes; better for position independent code
MOVE D0,(A0)	moves contents of D0 into address location stored in A0