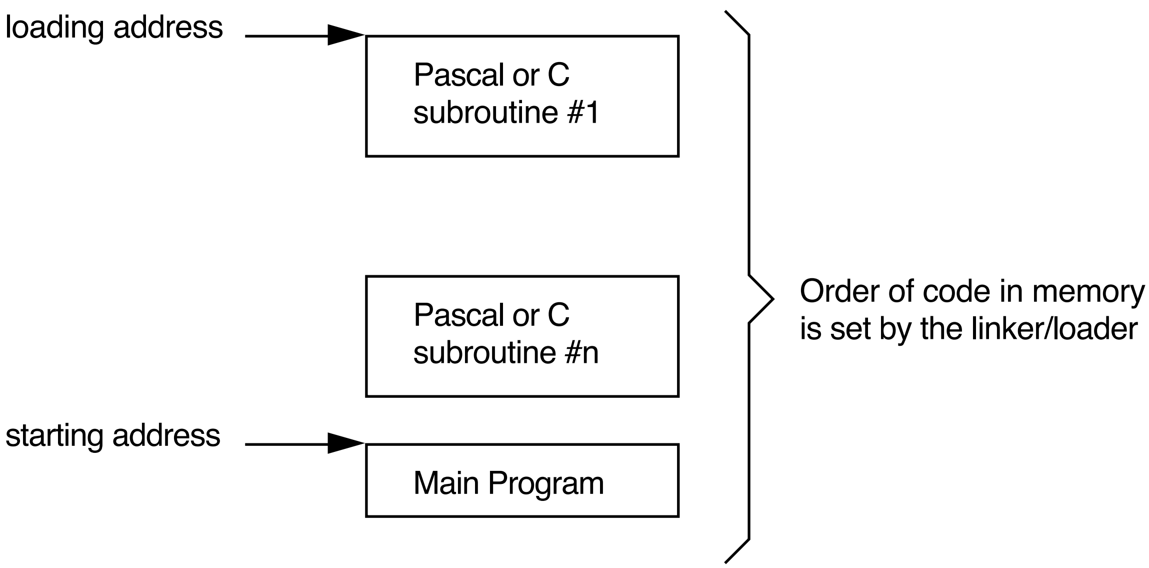
Running Programs (See Section 3.5.2 of your text)

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>

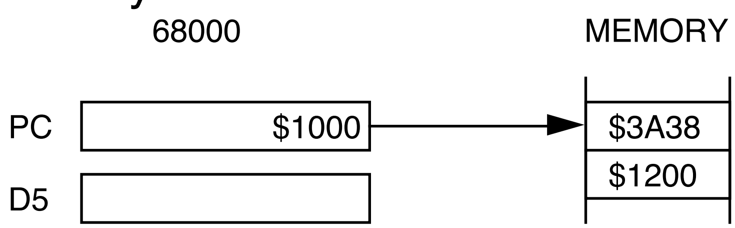
2. **P**rogram **S**tep **F**rom 1000h
   Program **S**tep

You can also automatically set the PC in the assembler

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

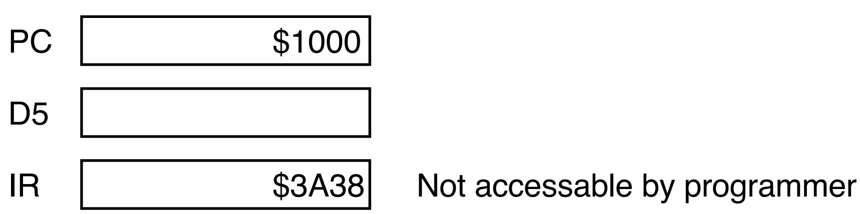
where <label> is any name you want. It will be used to set the initial PC value.

Fetch and execute for a simple example:
Initially:

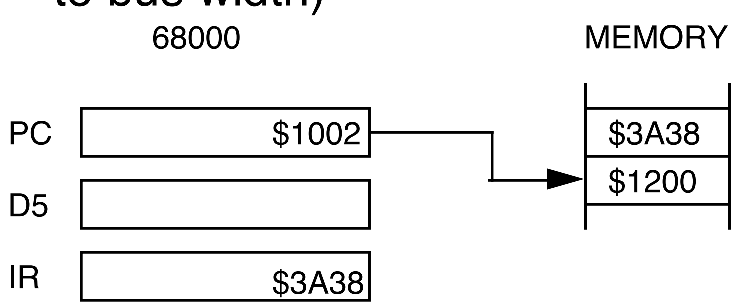


Fetch:

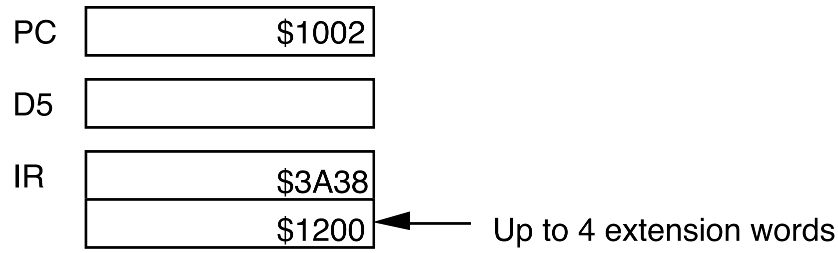
- Fetch instruction pointed to by PC and move into internal instruction register (not user accessable)



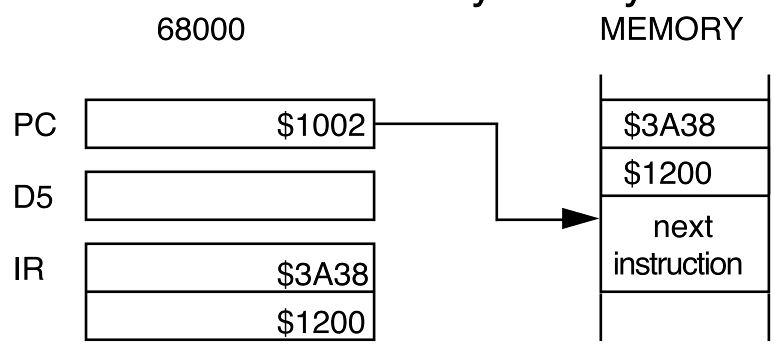
- Increment the PC by two bytes (one word due to bus width)



- 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.

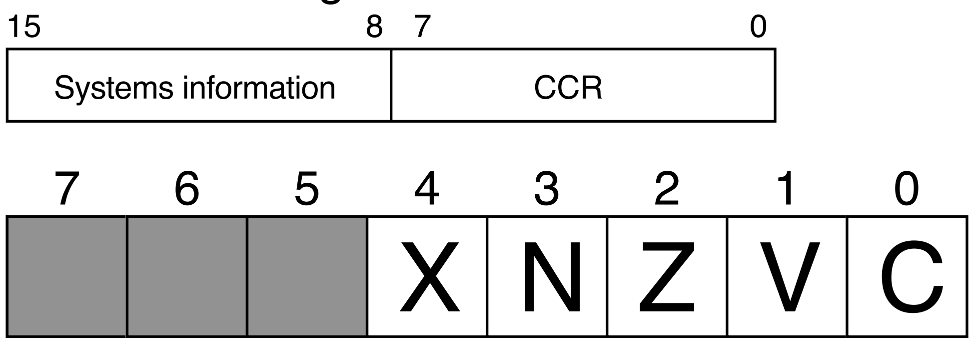
YOU WANT FAST INSTRUCTIONS WHENEVER POSSIBLE, i.e. NO extension words.

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	<u>extend bit</u> retains carry bit for multi-word arithmetic
3	<u>negative</u> set to 1 if instruction result is negative, set to 0 if positive
2	<u>zero</u> set to 1 if result is 0
1	<u>overflow</u> set if signmed overflow occurs
0	<u>carry/borrow</u>

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

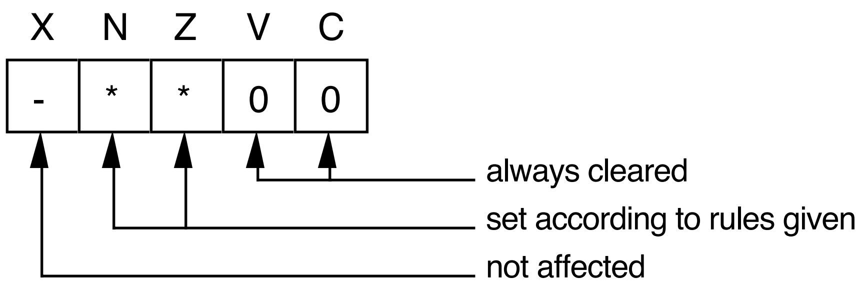
15	14	13	12	11	10	9	8
T		S			I_2	I_1	I_0

bits	function
11,12,14	not used
8,9,10	<u>interrupt mask</u> a priority scheme to determine who has control of the computer
13	<u>supervisor</u> set to 0 if user, set to 1 if supervisor
15	<u>trace</u> 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:

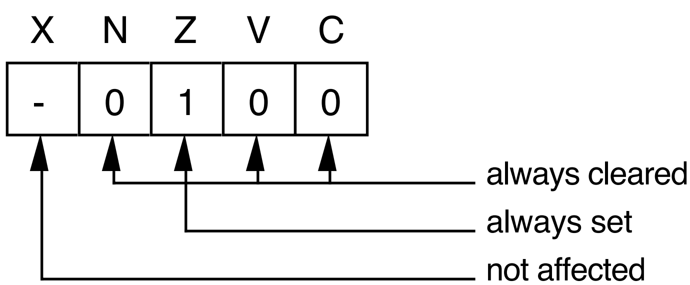
MOVE



ADD

X	N	Z	V	C
*	*	*	*	*

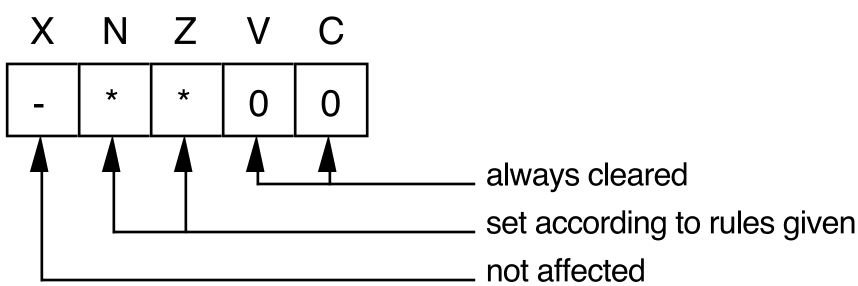
CLR



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 byte length operations)

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

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

126_{10}	addition of two	$0111\ 1110_2$	Carry
$\underline{3_{10}}$	positive <u>signed</u>	$\underline{0000\ 0011}_2$	C=1
129_{10}	numbers	$10000\ 0001_2$	Overflow
		(this is actually -127_{10})	V=0

The result 129_{10} is out of range for a signed 8-bit number. As a result, a carry occurs. However, the sign of the result matches that of the operands so signed overflow does not occur.

-2_{10}	addition of two	$1111\ 1110_2$	Carry
$\underline{-3_{10}}$	negative integers	$\underline{1111\ 1101}_2$	C=1
-5_{10}	with no overflow	$11111\ 1011_2$	Overflow
		with a carry	V=0

The signs match so no signed overflow occurred.

-127_{10}	addition of two	$1000\ 0001_2$	Carry
$\underline{-5_{10}}$	negative integers	$\underline{1111\ 1011}_2$	C=1
-132_{10}	with overflow	$10111\ 1100_2$	Overflow
		is actually +124	V=1

The decimal result -132_{10} is out of range for a signed 8-bit number so the sign of the result doesn't match and signed overflow occurs. In addition a carry occurred.

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

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

The same analysis can be applied to subtraction:

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

456_{10}	subtraction of	$ 01 C8	Carry C=0
$\underline{-3859_{10}}$	two signed	<u>+ $ F0 ED</u>	Overflow V=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.

<u>The EQU directive (F&T, Section 6.3.2)</u>
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