more with hgre } instructions.

M68000

M68000 Assembly Programming:

3

- Format of assembler instruction (mnemonic).

  <label> <op code> <operands> <comment>

- Label:

  (1) Start with letter, followed by letters and/or digits.

  (Some assemblers allows special symbols in label, eg _).

  (2) Most assemblers restrict # chars in label to 8.

  (3) Can be defined in 2 ways:

  (a) Start at column 1 followed by space.

  (b) Start in any column, terminate by colon (:)

- Opcode: cannot begin at column 1 because it will be considered as label.

- Operands:

  (1) separate opcode and operands by at least one blank space.

  (2) When 2 operands present, separate them with comma (no space).

  Some assemblers do not allow blank space between operands. (ours didn't !!!)

- Comment:

  (1) separate comment from operands with at least one space.

  (2) Some assemblers requires special character to start a comment - usually ; - not as 455

  (3) Illegal: <label> <comment>.
  - first word of comment is taken as opcode.

  (4) If column 1 contains "*", the entire line is treated as comment.

Instruction encoding in M68000

Skip first

- The M68000 uses multi length instruction encoding i.e., different instructions are encoded by different number of words.

The encoding format is always the same as is as follows:

(1) The first word (16 bits) in the instruction is the opcode word.

The opcode word specifies:

(1) the operation
(2) size of the operands.
(3) location of the source & destination operands.

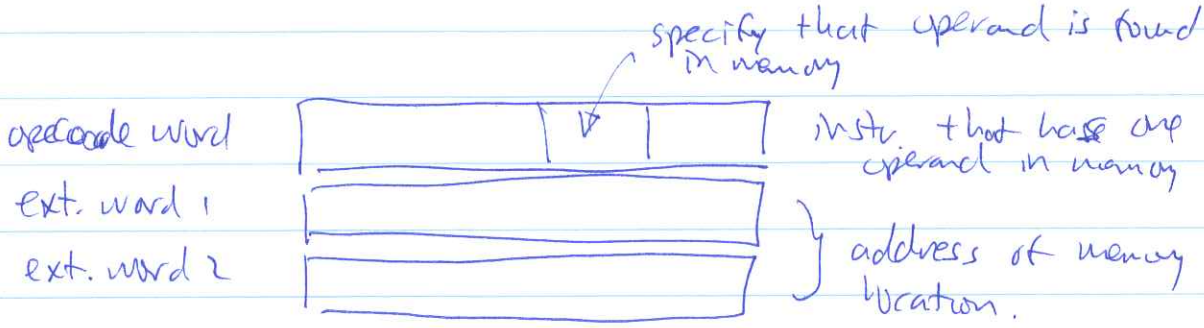
The location specified is not complete in some cases, in which case the extension words are needed to complete the specification.

(2) One or more extension words may follow the opcode word to complete the address of the source & destination operands.

- Note: extension words are used for example for instructions that has an operand in memory.

To identify the operand in memory, the address of the memory location containing the operand is specified. This requires 32 bits.

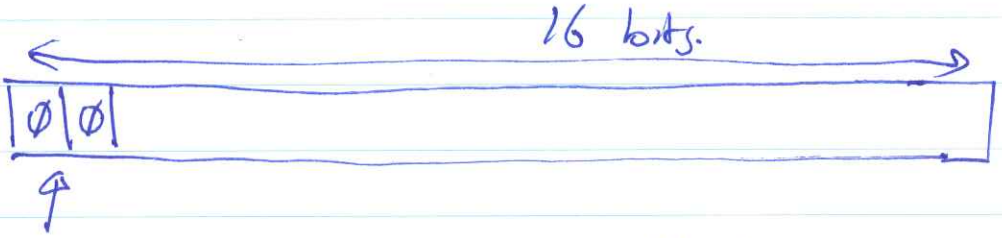
The address of the operand will be placed in 2 words immediately following the opcode word of the instruction.



Example encoding:

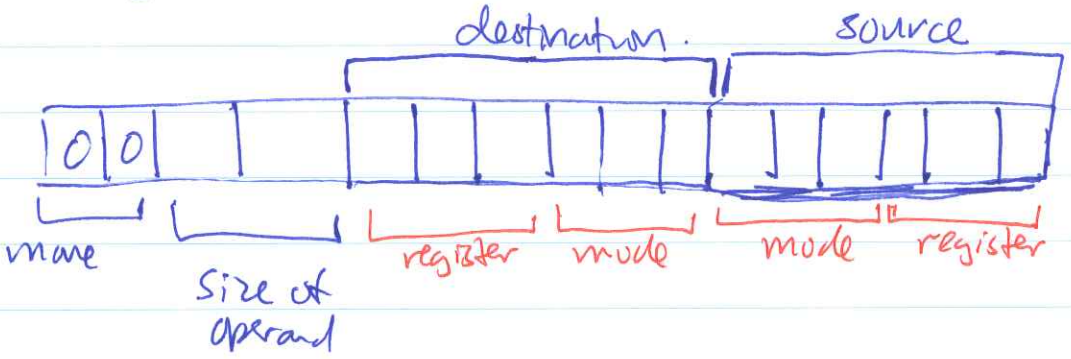
The move instruction in M68000 is used to transfer data from one place to another.

Format (encoding):



indicates that instruction is MOVE.

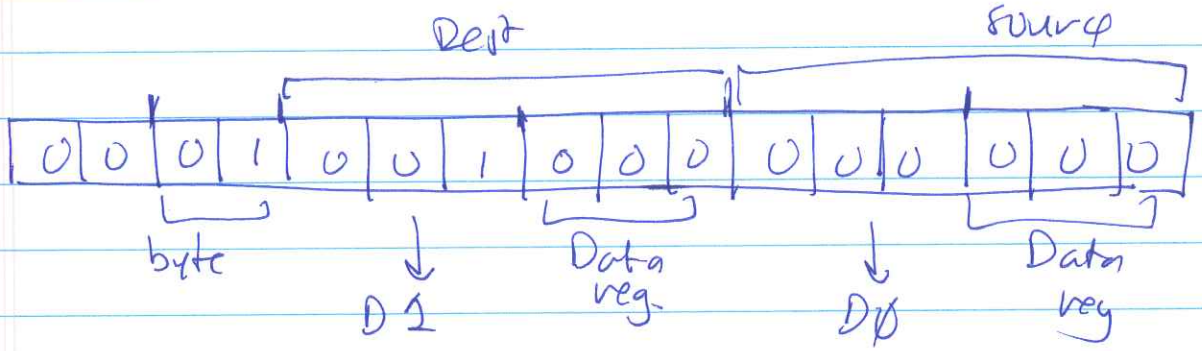
Remaining fields:



00 = byte
11 = word
10 = long word

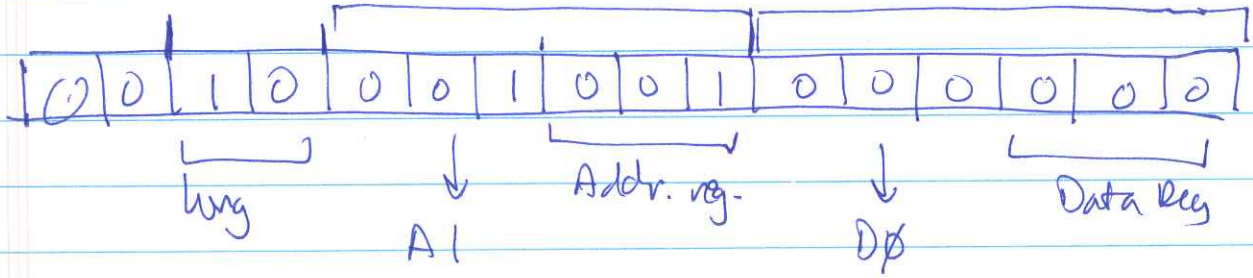
mode = indicates addressing mode (later)
reg = ~~indicate~~ identifies register if mode is a register mode.

Example:



MOVE.B D0, D1. → 1200 Hex.

0001	0010	0000	0000
1	2	0	0



move.L D0, A1

0010	0010	0100	0000
2	2	4	0

Hex.

Example:

mode = 000 indicates "Data register mode".

Then reg =		
	000	D0
	001	D1
	010	D2
	011	D3
	100	D4
	101	D5
	110	D6
	111	D7.

mode = 001 indicate "Address register mode"

Then: reg =			
	000	=	A0
	001	=	A1

etc.

mode = 111 indicates "memory"
reg = 001 indicates long address

Show intro-asm1.s

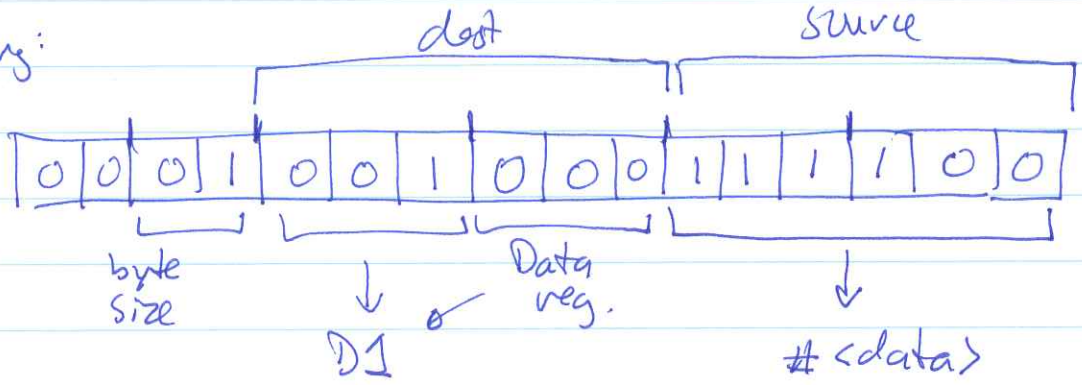
interpret cycles.

Demo intro-asm1.s

Example:

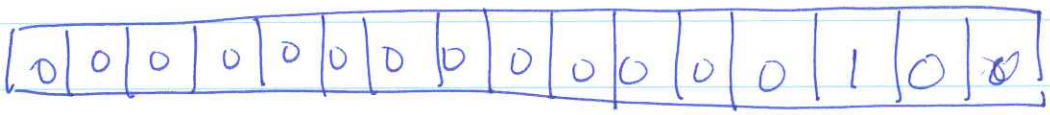
move.b #4, D# move the byte size operand 4 into D#.

Encoding:



meaning: operand ~~data follows instruction~~! Stored in extension words.

next word of the instruction contains the data:



"4" in binary (2's cmpl).

M68000 Instruction set

- The M68000 instructions can be subdivided into 5 broad classes:

  (1) Data movement (copy!)
  (2) arithmetic
  (3) logic
  (4) branch & jump (includes subroutine call) and compare
  (5) control - system functions (not covered in this course)

- Instructions can have 0, 1 or 2 operands.

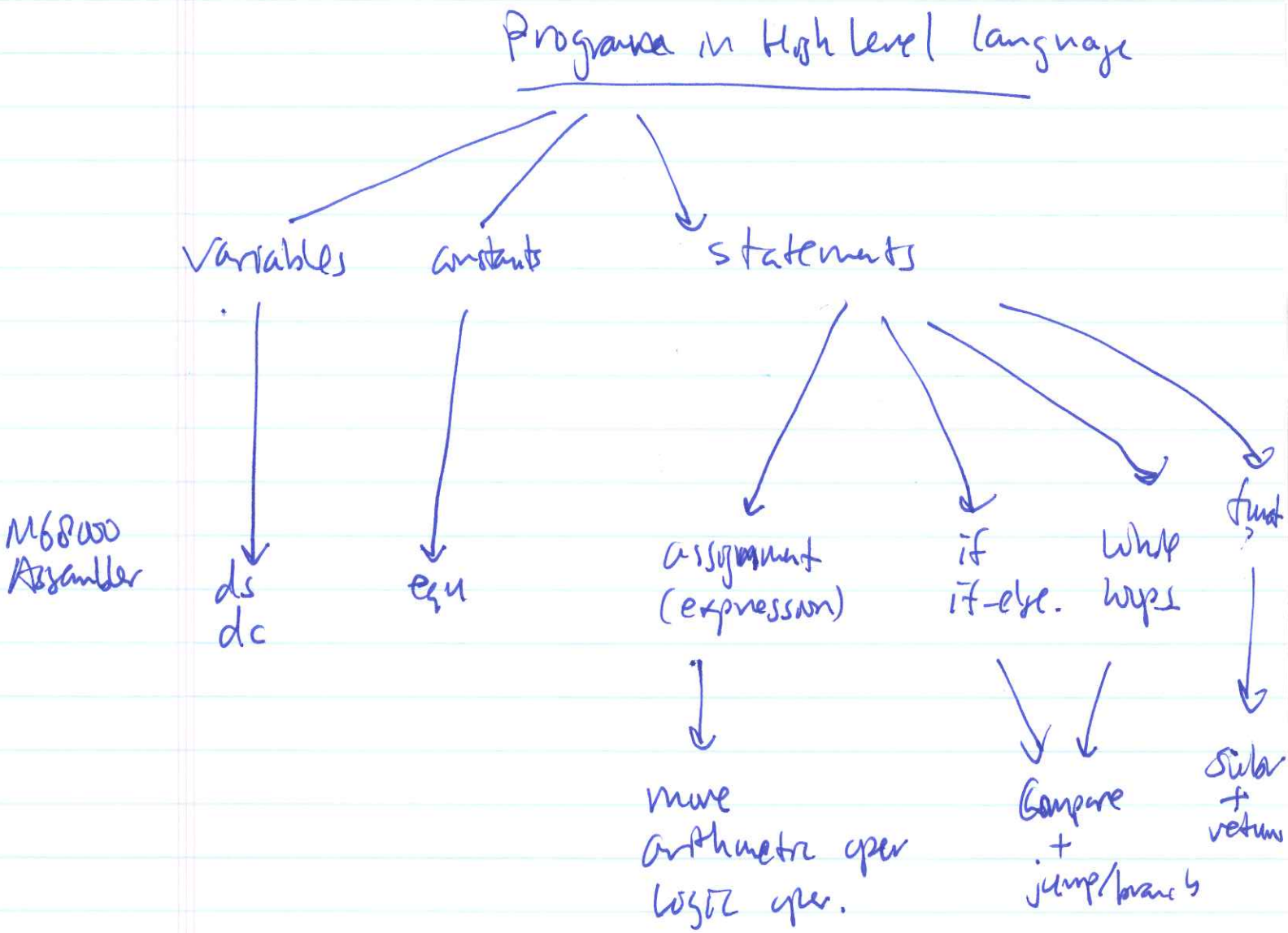
  All M68000 dual operand instructions are written in the form:

  no space!!!

  [label:] <operation> <source>,<destination>

  <source>: location of the source operand
  <destination>: location of the destination operand.

- Some operations that requires 2 source operands (eg. ADD, SUB.. etc), will use the <destination> as the second source operand.


Programs in High Level language
Variables
constants
statements
M68000 Assembler
ds
dc
equ
assignment (expression)
if
if-else.
while loops
func
move
arithmetic oper
logic oper.
Compare + jump/branch
Subr + return

MOVE instruction — actually: COPY

- The move instruction is used to copy data from source to destination.

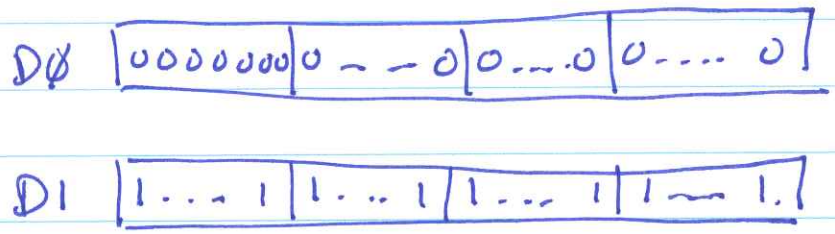
Syntax: `MOVE[.s] <source>, <destination>`

- One can specify the amount of data to be copied by the size of the operation:

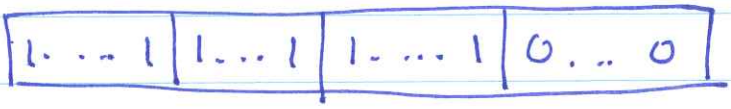
MOVE.B	will copy	8 bits. (byte size operand)
MOVE.W	will copy	16 bits (word size ...)
MOVE.L	will copy	32 bits. (long word size)

eg. `MOVE.B D0, D1` copies lower 8 bits of D0 into D1.

eg



move.B D0, D1



D0 ↘ { Move.W D0, D1
MOVE.L D0, D1.

- The condition flags (N, Z, V, C) are set according to the result of the move.

- MOVE can be used with:

  any source (Dn, An, memory etc).
  any destination, EXCEPT address registers.

  The move to addr. registers is special because it does NOT affect cond. flags

  (Recall: operations affecting addr. reg's do not change N, Z, V, C flags).

- To distinguish this special case, M68000 has another move instruction:

  movea[.s] <source>, An

  Need time to explain to students!

  Note: s = word or long only, not byte (error!).

  → if destination of move is an ADDR reg, you need to write move instruction as:

  movea (make sure you realise that this instruction does not update N, Z, V, C flags).

- "Quick" operations:

Story: frequently, programs use initialization statements like this: i = 0 i++ j = 1 j--

M68000 tries to optimize things. ↑ small constant.

M68000 has several "quick" instructions:

move-quick, add-quick, subtract-quick.

syntax: `moveq #<constant>, Dn`

- Compare:

machine code:

```
move.l  #4, d0      203C
                    0000
                    0004

moveq   #4, d0      7004
```

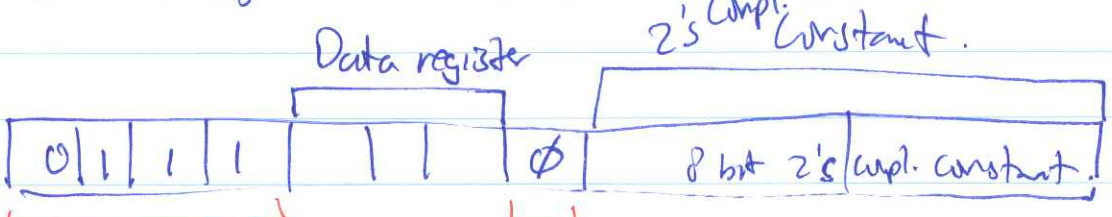
intro-casuals

↑ only take (long) operands. – only LONG !!!

- "Quick" means: less memory access.

moveq is executed 3 times faster than move.l !!!

MOVEQ encoding:



identifies moveq instruction.

↑ will be sign-extended to 32 bits and moved to register.

How do you tell what is allowed in source & dest?

- Ideally, you don't want to burden the assembler programmer will extra rules.

But simple fact of life is, we have to limit ~~implementation~~ flexibility to cut cost.

(any CPU)
So: M68000 ideally would allow pwgrammer to use any source & any dest. in any assembler instruction.

But: that's not so.

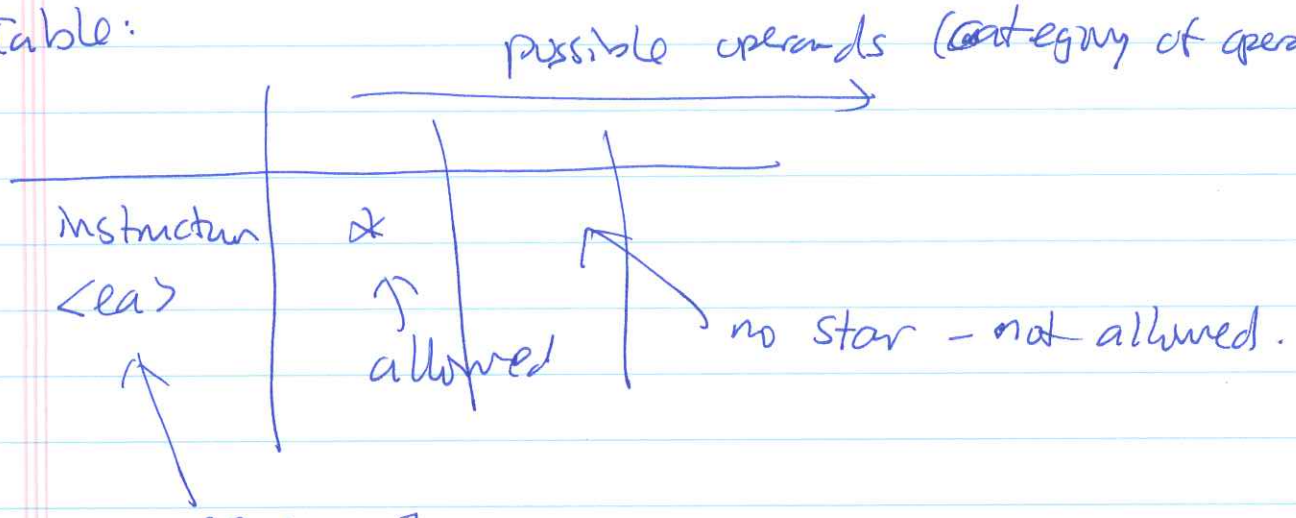
For each assembler instruction, a specific set of source operands and a specific set of dest. operands are allowed.

- Don't bother to remember for all. Just practice and learn.

- Keep a table handy for reference.

Hand out tables!!!

Table:



Example:

move <ea>, <ea>

→ all source operands are allowed

→ dest. operands cannot be:

#	(later).
An	(address register).
d(PC)	(not covered)
d(PC, Ri)	(not covered)

Stop Here
Do addr. mode FIRST !!! — addressing mode (next set of notes).

Basic M68000 instructions

Instruction syntax	Description.
Data movement.	
CLR [.s] <ea>	∅ → Destination
MOVE [.s] <sea>, <dea>	Source → destination
MOVEA [.s] <sea>, An	Source → An. note: s = W or L only.
MOVEQ #<const data>, Dn	<const data> → Dn note: size is always Lng! note: constant ∈ [-128, 127]
Arithmetic	
ADD [.s] <ea>, Dn ADD [.s] Dn, <dea>	Dn + Source → Dn Dest + Dn → Dest
SUB [.s] <sea>, Dn SUB [.s] Dn, <dea>	Dn − source → Dn Dest − Dn → Dest

!!! must use "A" if addr. reg. is destination

★ ADDA [.S] <ea>, An — An + Source → An

note: (1) S = W or L only
(2) flags unchanged!

★ SUBA [.S] <ea>, An — An − Source → An

ADDI [.S] #<const>, <ea> — Dest + <const> → Dest

SUBI [.S] #<const>, <ea> — Dest − <const> → Dest

Quick versions of ADDI & SUBI:

ADDQ [.S] #<const>, <ea> — Dest + <const> → Dest

size = B, W, L !!!

SUBQ [.S] #<const>, <ea>

note: <const> ∈ [1, 8].

"Quick" because less time in fetching instruction!!!

demo: quicks

- The notation:

  ~~<ea> means "effective address",~~

  <sea> = source ~~effective~~ address
  <dea> = destination effective address.

- Whenever you see , <sea> or <dea>, it means that the instruction can accept various different types of operands:

  eg:

  MOVEQ #<const> , Dn

  this instruction only accept ~~the~~ one form:

  eg: MOVEQ #1, D~~0~~
  MOVEQ #0, D7

  But:

  ~~MOVEQ~~
  MOVEA [.S] <ea>, An

  ↑ accepts a number of ~~any~~ effective address formats.
  Work ones? → look them up!

- In order to understand how computer obtain the operands for an instruction, we have to talk about: addressing modes.

This is one of the most difficult topic in Assembler programming. (next to recursion).

- In assembler, there are ~~different~~ specific ways to tell the assembler where to get the operands.

Operands can be:

(1) constants encoded inside the instruction.

(2) data in registers.

(3) data in memory, in this case, the address is given inside the instruction.

- The data in registers are specified by the name of the register. (holding the data).

- The data in memory is specified by the location (address) in memory.