Representing ~~chara~~ Alphanumerical data & Instruction encoding.

- Alphanumerical datum is a "character" from a character set.

- Different character sets have different number (and type) of characters.

  eg: Chinese char. set has over 20,000 diff. characters.

- Most widely used character set is the Alphabet, augmented with numbers & lexicographical symbols.

  → A, B, C, D, ... Z
  a, b, c, d, ... z
  1, 2, 3, ... 9, Ø
  (SPACE), #, $, &, %, (,), [,],
  +, −, *, /, :, ;, <, >, =, etc.

- ~~At one time, different computers use dif~~.

- The representation is again through encoding:

  we use a small integer to represent a character (diff. characters have diff. "integer value".)

- Example encoding:

show them an octal dump of a file:

```
vi myfile
```

```
~~od -c myfile~~
```

printascii

```
printascii < myfile
```

printascii also works interactively.

or even from SAME manuf.!
from diff. manufacturer

- At one time, diff. computers use diff. character encoding schemes.

What would be the result ?

suppose you are edit a file on your PC. made by manufacturer X.

You bring your file to school and try to read it on a PC by manuf. Y.

You file may ~~not~~ be recognizable !

Why ?

PC X	
A	1000.0001
B	1000 0010
C	1000 0011
D	1000 0100

PC Y	
:	
U	1000.0001
V	1000.0010
W	1000.0011
X	1000.0100

BAD =

10000010
10000001
1000.0100 → VUX

- A standard encoding was establish called

ASCII

/

American Standard ~~for~~ Code for Information Interchange.

and it quickly became standard to represent alphanum. data in <u>all</u> modern computers.

- The ASCII code ~~rep~~ represents 128 characters. Some are "non-printable"

eg: 127 - delete.

- ~~Ask tom~~

- What is the code for character A ?

$$\rightarrow A = (41)_{16}$$

$$= (0100.0001)_2.$$

$$= 65$$

- To distinguish the character codes, we enclosed then between ' ' in C:

'A' = (Ascii) code (a small integer) representing character A.

- Because characters are repr. by small integers, we can do calculations with them:

'B' - 'A' = 1 !!!

- A terminal reads in data in ASCII code.

If the data is to represent an integer, conversion must take place.

eg: 12 is entered in ascii as:

'1' '2' '\n'

or:

00110001	00110010	00001010
(49)	(50)	(10)

But 12 as integer is : (8 bits)

00001100

How do we do the conversion?
↳ program!

(that's what scanf do for you automatically)

- What computer can do:

(1) assign a known value

representation ↓

Best known value : $\emptyset$ = 000000000000

$\emptyset$ ↑ value

(2) Do computations in 2's complement
eg: +, −, *, /

(3) negate values :

2	→	−2	(subtract from 10000000)
00000010		11111110	

⇒ with these capabilities, we can build representation with a progra

Computer can do the following:

use these op's to convert a string repr. to bin repr!

- (1) initialize an int variable to any value
- (2) add binary values, sub, mult, div. etc.
- (3) negate a binary value.

atoi:

input: ascii string representing an integer. (S[])

output: internal (2's compl) repr. of integer. (value)

```
if (S[0] == '-')
   { sign = -1;  startpos = 1; }
else
   { sign = 1;  startpos = 0; }

value = 0;

for (i = startpos;  i < strlen(s) ; i++)
  { value = (s[i] - '0') + 10 * value;
  }

if (sign == -1)
    value = - value;
```

255/demo

atoi.cc

example: 12 '1' '2'

(00110001) − (00110000) = 00000001

'1' − '0' = 49 − 48 = 1

2 points ~~first~~:

S.charAt(i) = char at position i

(int) s.charAt(i)
= ascii code value of character.

(char)(int) s.charAt(i)
= character

Java atoi:

```
int startPos, sign;
int value
int i;
```

Input: String s.
Output: value = 2's compl. repr. for digit string s.

Alg:

```
                                  (ascii code 2D = 00101101
    if ( s.charAt (Ø) == '-' )
        sign = -1               ( Sign = 11111111111111111111111111111111)
        start Pos = 1
    else
        sign = 1                ( sign = Ø0000.....          0001)
        startPos = Ø

    value = Ø                   ( value = 0000....0 )
    for (i = Ø ; i < s.length(); i++)
        value = value *10 + (s.charAt(i) - 'Ø');


    if (sign == -1)
        value = -value;          ( 2's compl. negation! )


Output value & 2's compl. repr.
```

Try examples :

6

~6

12

~12

Example : s[] = '-' '1' '2' (-12)
(strlen(s) = 3).

s[Ø] == '-' is true → sign = -1
startpos = 1.

value = Ø (0000000)

iter 1: i = startpos
= 1;
value = (s[1] - 'Ø') + 10 * value
= ('1' - 'Ø') + 10 * Ø
= 1 + Ø
= 1
i++
i = 2.

use 2's compl. representation !!!

iter 2: i = 2
value = (s[2] - 'Ø') + 10 * value
= ('2' - 'Ø') + 10 * 1
= 2 + 10
= 12

i++
i = 3.
done (i = strlen(s) = 3)

sign == -1 is true → value = -value
= -12.

result: value = -12.

~e255/demo

itoa.c

itoa:

input : value n in two's compl. repr.

output : char s[ ] - ascii repr. of n.
(\0 terminated string).

```
int length, sign, i;
char s[100], helpstring[100];

    ~~[illegible]~~

    if (n < 0)
      { sign = -1; n = -n; }
    else
      { sign = 1; }
    ________________________________ now: n ≥ 0

    length = 0;                    (# chars to repr. n).

    do
     { helpstring[length] = '0' + (n % 10);
       length++;
       n = n / 10;
     } while (n > 0);

    if (sign == -1)
      { helpstring[length] = '-'; length++ }
    Reverse String helpstring into s.
```

Java itoa :

```
int sign, i, j
String help, result
char next_char;

   if (value == ∅)               ( value = 0000.....0)
      return ("∅")                ( ASCII code for char ∅ )

   if ( value < ∅ )              ( value = 1......)
     { Sign = -1;
        Value = - value;         ( 2's compl negate! )
     }
   else
      { sign = 1;
      }

   help = ""                    ( no char codes).

                     00110000
   do
     { next_char = '∅' + (value % 10);
                   ASCII char. code for next digit !

        help = help + next_char      (concatenate!)
        value = value /10;
     }  while (value > ∅);
  flip string help
```

Better, concatenate at start!!!

Example:

n = -12

n < Ø → sign = -1 ,

n = 12

iter 1: length = Ø

helpstring[Ø] = 'Ø' + (12 % 10)

= 'Ø' + 2

= '2'

length = 1

n = n / 10

= 1

iter 2: length = 1

helpstring[1] = 'Ø' + (1 % 10)

= 'Ø' + 1

= '1'

length = 2

n = n / 10

= Ø

stop.

now: helpstring[] = '2' '1' (12 in reverse order).

length = 2
↓
sign == -1 thus: helpstring[2] = '-'

Result:

helpstring[] = '2' '1' '-' '[illegible]' (-12 in reverse order).
↑

Reverse helpstring into s:

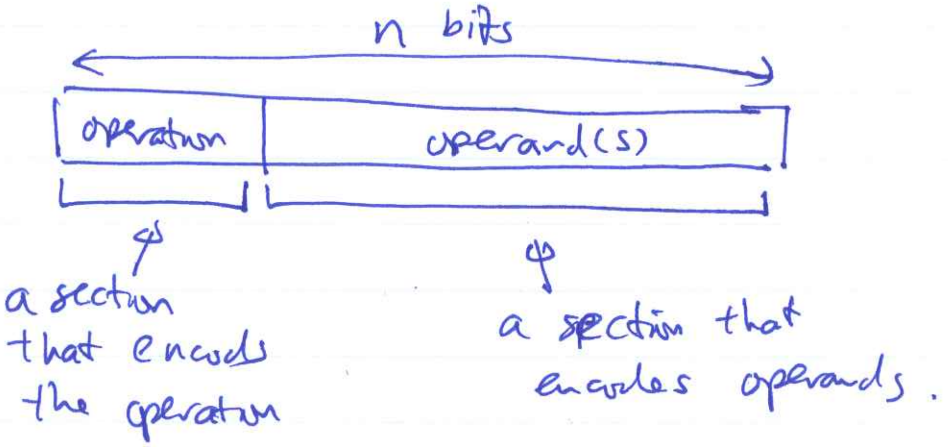
s[] = '-' '1' '2' '\0'
↑

C represent strings as null terminated strings.

Representing Computer Instructions

- category of operations performed by computer
  - 0 operands → (HALT) commands
  - 1 operand → unary operations. Eg: negate
  - 2 operands → binary operations. Eg: add

- Format of computer instructions:


n bits
operation
operand(s)
a section that encods the operation
a section that encodes operands.

Note: n can be fixed or variable.

- Types of ~~operation~~ instruction formats

fixed length
(n bits, n constant)

↓

Every instruction executed by computer has same # bytes

(SPARC, MIPS, PowerPC)

Variable length.
(n variable).

↓

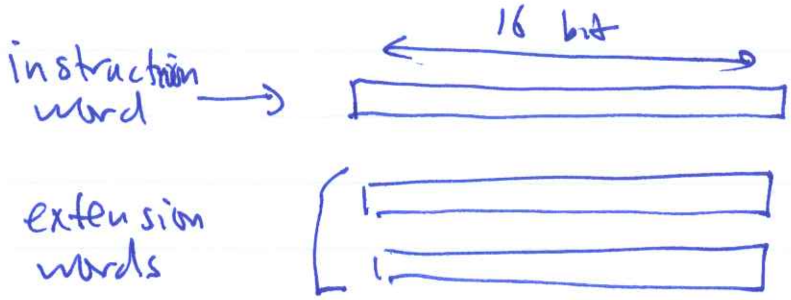
diff. instructions may ~~be~~ be encoded using diff. # bytes.

(Intel
M68000).

- Typically: modern computers use fixed length instruction format.

  – Intel still uses variable length to maintain compatibility with 8080 invented in 1978.

M68000 instruction encoding:

- 

- M68000 is an example of variable length instruction encoding, popular ~~before~~ in CPU's developed before 1984 or so.

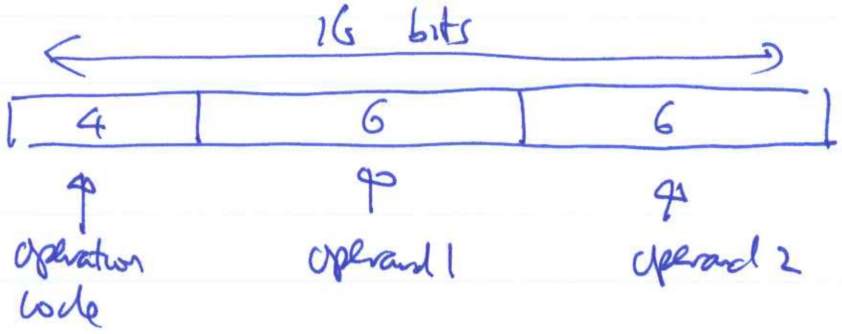
- M68000 Instruction consists of:

  (1) one instruction word of 16 bits.

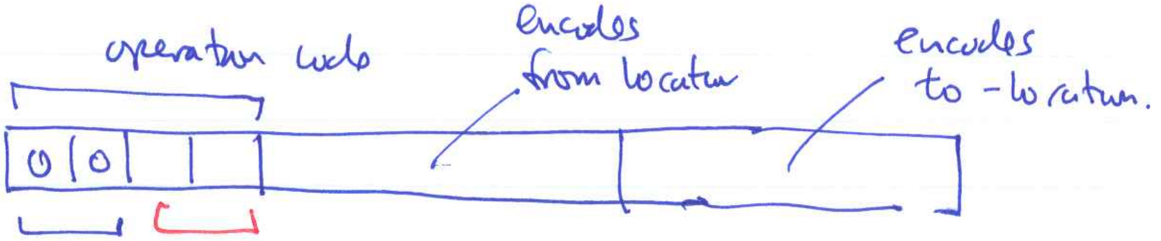
  (2) possibly followed by one or more (upto 4) extension words (each 16 bits) ~~used to~~

Note: bits inside the instruction word will convey information on how many extension words will follow.

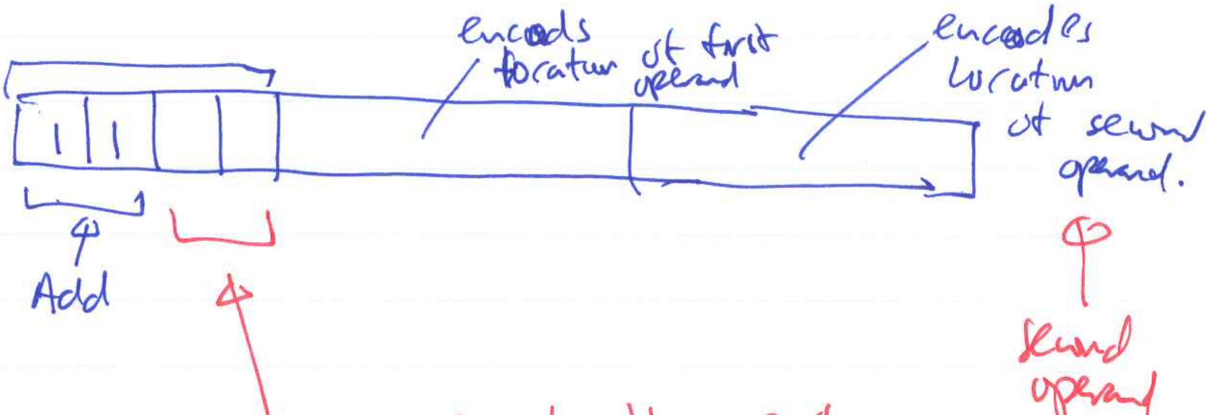
- Typical format for a M68000 instruction word: (first word of instruction) encoding a binary instruction.



Example:



move — transfer data from one place
(copy) to another.

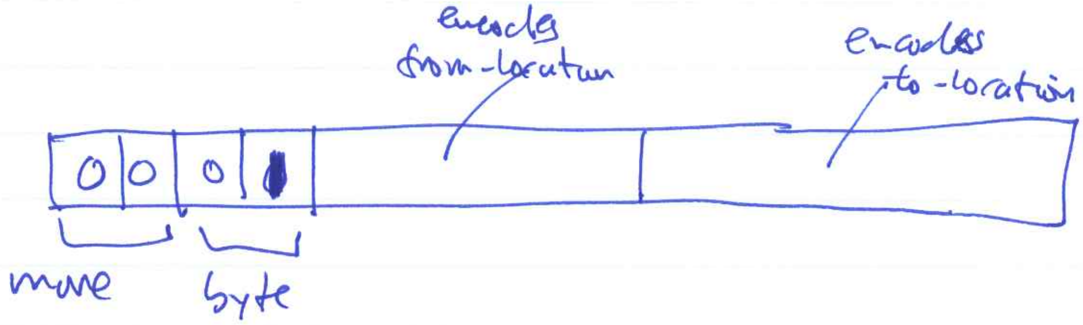

encods location of first operand
encodes location of second operand.
1
1
Add

encodes the length ~~ope~~ of the operands.

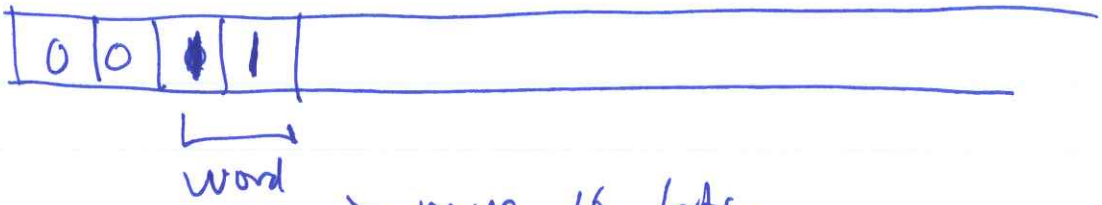
second operand doubles as destination

- 3rd & 4th bit in instruction word encodes the length of the operands

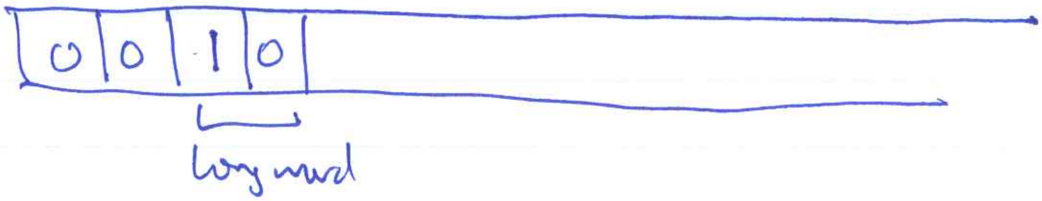
eg:



means: move 1 byte from from-location to to-location.



~ move 16 bits.



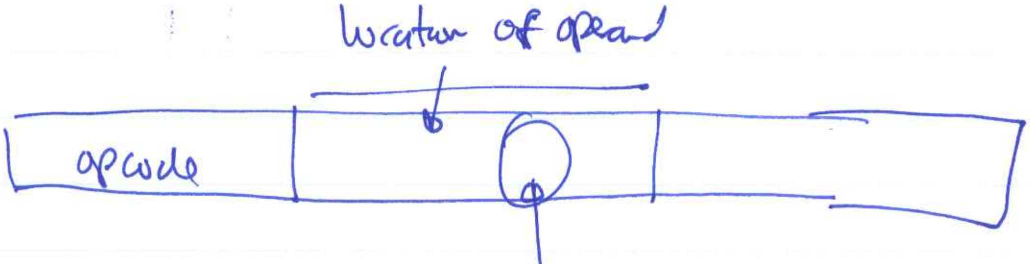
~ move 32 bits

- Extension words are needed when operand are located in memory.

Reason:

- Operand in memory is identified by an address.
- ~~Add~~ Addresses in M68000 are 32 bits long.
- No way to fit 32 bits into 6 bits available.

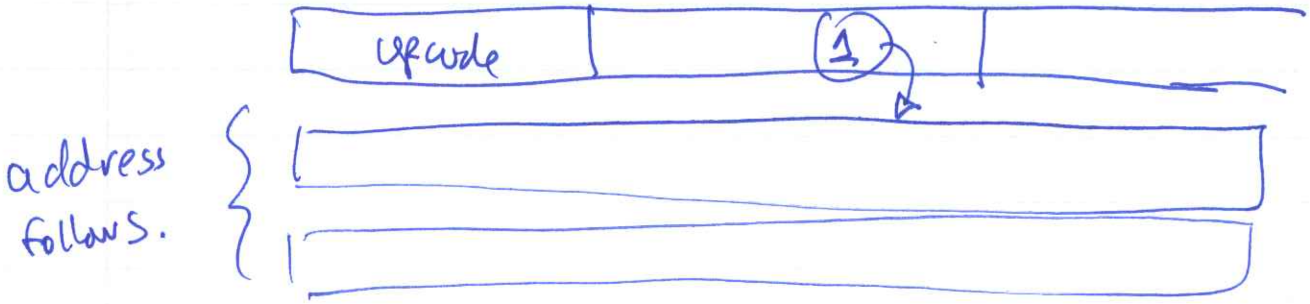
- So:



some bit will indicate that operand is in memory.

eg: ∅ : not in memory
1 : in memory

Then:



MOVE

Move Data from Source to Destination
(M68000 Family)

Operation: Source → Destination

Assembler Syntax: MOVE ⟨ea⟩,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Moves the data at the source to the destination location, and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			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	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

MC68020, MC68030, AND MC68040 ONLY

(bd,An,Xn)*	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)*	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Can be used with CPU32.

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

MC68020, MC68030, AND MC68040 ONLY

(bd,An,Xn)**	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)**	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*For byte size operation, address register direct is not allowed.
**Can be used with CPU32.

Notes:

1. Most assemblers use MOVEA when the destination is an address register.
2. MOVEQ can be used to move an immediate 8-bit value to a data register.

ABCD

R/M field — Specifies the operand addressing mode:
0 — the operation is data register to data register
1 — the operation is memory to memory

Register Ry field — Specifies the source register:
If R/M = 0, specifies a data register
If R/M = 1, specifies an address register for the predecrement addressing mode

ADD

Add
(M68000 Family)

Operation: Source + Destination ▶ Destination

Assembler Syntax: ADD ⟨ea⟩,Dn
ADD Dn,⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Adds the source operand to the destination operand using binary addition, and stores the result in the destination location. The size of the operation may be specified as 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
*	*	*	*	*

X Set the same as the carry bit.
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.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
1	1	0	1	REGISTER			OPMODE			MODE			REGISTER		

ADD

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	$\langle ea\rangle + \langle Dn\rangle \rightarrow \langle Dn\rangle$
100	101	110	$\langle Dn\rangle + \langle ea\rangle \rightarrow \langle ea\rangle$

Effective Address Field — Determines addressing mode:

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

MC68020, MC68030, AND MC68040 ONLY

(bd,An,Xn)**	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)**	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Word and Long only.

**Can be used with CPU32.

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
−(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

MC68020, MC68030, AND MC68040 ONLY

(bd,An,Xn)*	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)*	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Can be used with CPU32.

ILLEGAL

Description: Forces an illegal instruction exception, vector number 4. All other illegal instruction bit patterns are reserved for future extension of the instruction set and should not be used to force an exception.

Condition Codes:
Not affected

Instruction Format:

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

JMP

Jump
(M68000 Family)

Operation: Destination Address → PC

Assembler Syntax: JMP ⟨ea⟩

Attributes: Unsized

Description: Program execution continues at the effective address specified by the instruction. The addressing mode for the effective address must be a control addressing mode.

Condition Codes:
Not affected.

Instruction Format:

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

Instruction Fields:
Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed as shown:

JMP

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

MC68020, MC68030, AND MC68040 ONLY

(bd,An,Xn)*	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)*	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*Can be used with CPU32.

JSR

Jump to Subroutine
(M68000 Family)

Operation: SP − 4 → Sp; PC → (SP)
Destination Address → PC

Assembler Syntax: JSR ⟨ea⟩

Attributes: Unsized

Description: Pushes the long-word address of the instruction immediately following the JSR instruction onto the system stack. Program execution then continues at the address specified in the instruction.

Condition Codes:
Not affected.

Instruction Format:

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

SPARC instruction encoding:

32 bits.

	5	6	5	1	8	5
~~op~~	Destin	opcode	source1	∅	...	source2

type

or:

	5	6	5	1	13
op	Destin	opcode	source1	1	binary constant

type

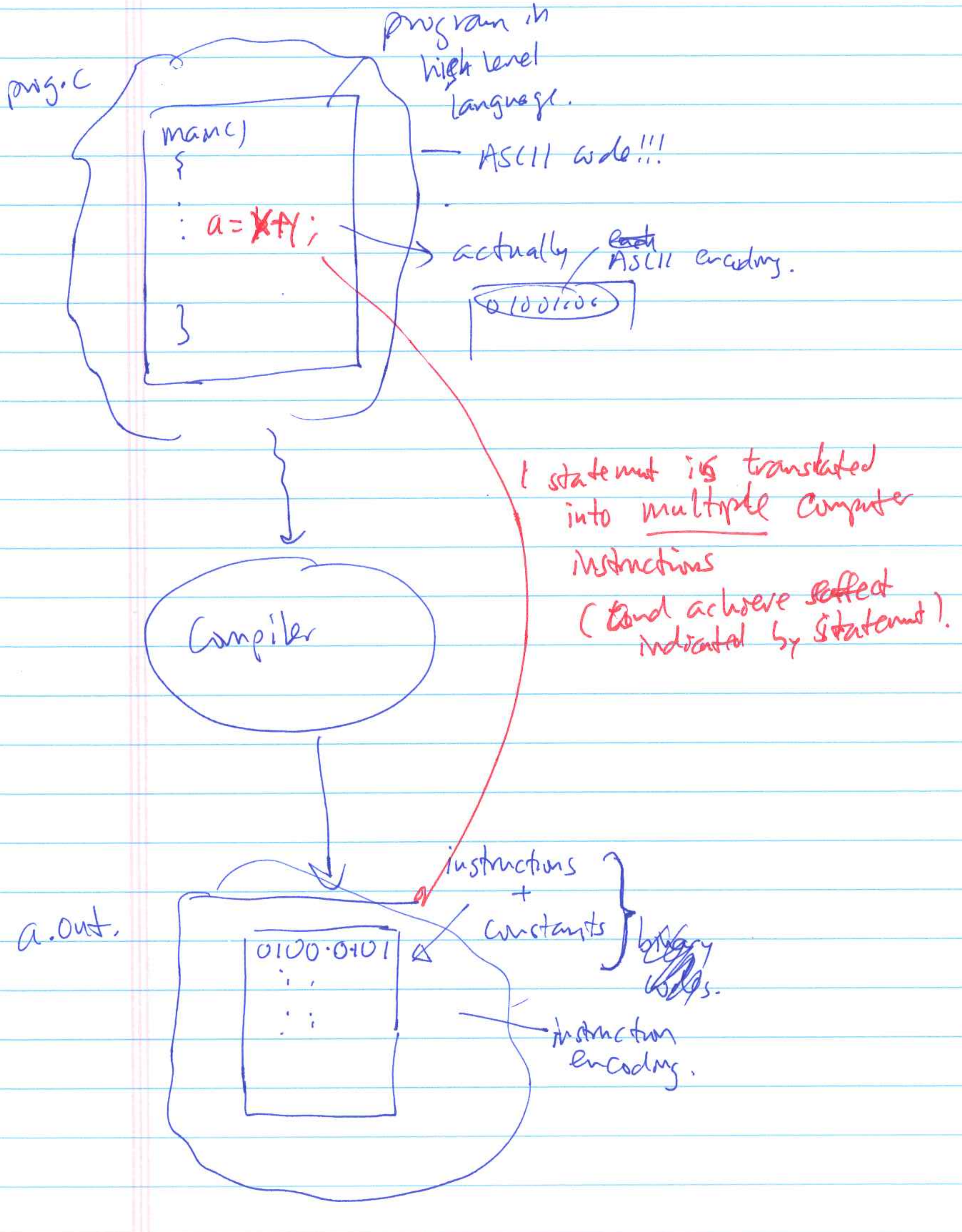
(type + opcode) determines the operation.

- Fixed length : each SPARC instruction is 32 bits.
- Regular : each field in the instruction has a well-defined use.

This allows the computer manufacturer to simplify the hardware.
pipeline the hardware is easy too.

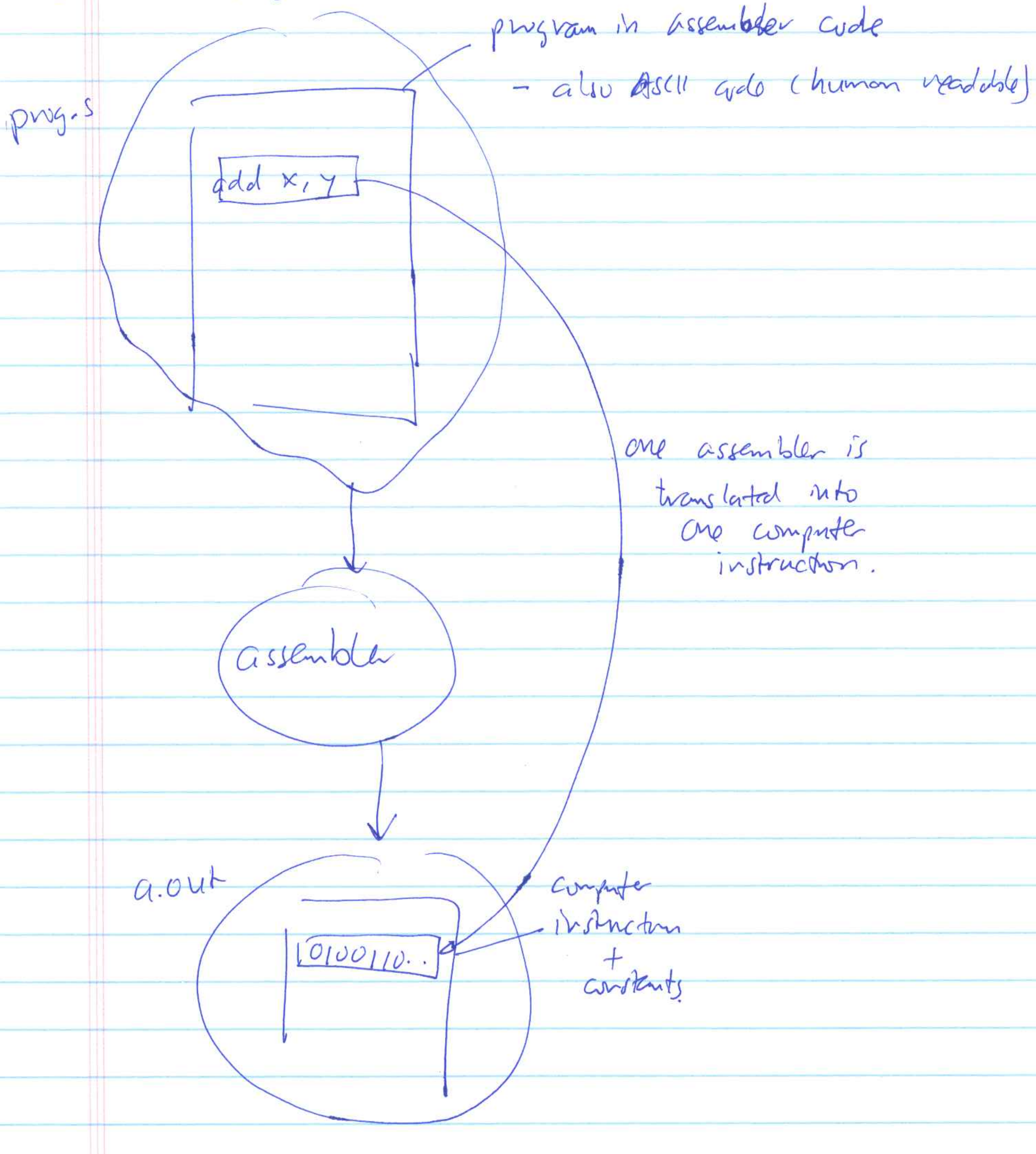
- This is because MEMORY SPEED ≈ Register speed !!! (otherwise CISC is faster!).

Transforming human readable programs into computer executable programs



1 statement is translated into multiple computer instructions (and achieve ~~effect~~ effect indicated by statement).

assembler program


program in assembler code
- also ASCII code (human readable)
prog.s
add x, y
assembler
a.out
10100110. .
computer instruction + constants

one assembler is translated into one computer instruction.