The Elements of Computing Systems

– Nand 2 Tetris

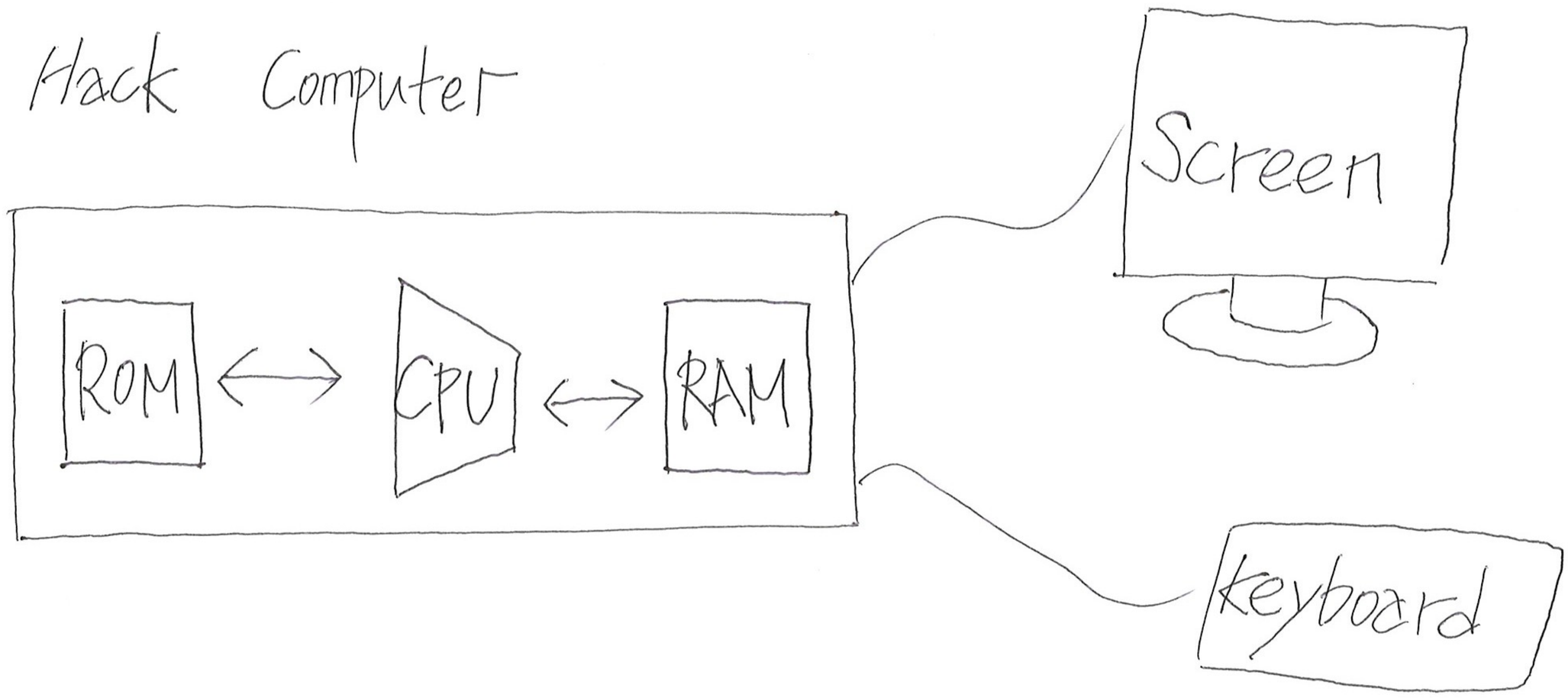
Part 0. Hello, World Below

- Mental tool - Abstraction

Focus on WHAT : Abstraction (exact specification, interface)
not HOW : Implementation

* Separate concerns and get this mental saving

- Hack Computer



Part 1. Boolean Logic

- Boolean Operations
  - Basic operations

returns the opposite of it.

X	Y	AND (∧)	OR (∨)	X	¬X
0	0			0	1
0	1		1	1	0
1	0		1		
1	1	1	1		

returns 1 only if both of them were 1

returns 1 if any one of them is 1

  - Secondary operations

X	Y	$(\neg X \vee Y)$ $X \to Y$	(XOR) $X \oplus Y$	(IFF, biconditional) $X \leftrightarrow Y$, $X \equiv Y$
0	0	1		1
0	1	1	1	
1	0		1	
1	1	1		1

returns 1 only if either X is 0 or Y is 1.

returns 1 only if one of them is 1

returns 1 only if both are equal

- Boolean Identities
  - Commutative Laws

    $X \wedge Y = Y \wedge X$

    $X \vee Y = Y \vee X$
  - Associative Laws

    $X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z$

    $X \vee (Y \vee Z) = (X \vee Y) \vee Z$
  - Distributive Laws

    $X \wedge (Y \vee Z) = (X \wedge Y) \vee (X \wedge Z)$

    $X \vee (Y \wedge Z) = (X \vee Y) \wedge (X \vee Z)$
  - De Morgan Laws

    $\neg(X \wedge Y) = \neg X \vee \neg Y$

    $\neg(X \vee Y) = \neg X \wedge \neg Y$

Don't ask about Y

$\boxed{\neg X \wedge \neg Z}$ $\boxed{\vee}$

$\boxed{X \wedge \neg Y \wedge \neg Z}$

- Truth Table to Boolean Expression

X	Y	Z	
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

$\boxed{\neg X \wedge \neg Y \wedge \neg Z}$ $\boxed{\vee}$

$\boxed{\neg X \wedge Y \wedge \neg Z}$ $\boxed{\vee}$

$\boxed{X \wedge \neg Y \wedge \neg Z}$

can combine into one clause

an expression that gets a value of 1 only at this row

Any Boolean function can be represented using an expression containing AND and NOT operations.
(X OR Y = NOT (NOT(X) AND NOT(Y)))

- NAND

X	Y	
0	0	
0	1	
1	0	
1	1	0

$X \text{ NAND } Y = \neg(X \wedge Y)$

returns 0 only if both of them were 1

i) $\neg X = X \text{ NAND } X$

ii) $X \wedge Y = \neg(X \text{ NAND } Y)$

- Any Boolean function can be represented using an expression containing only NAND operations.
- "Not every" Real number function can be represented using arithmetic operations.

Boolean is finite world.
And that's what exactly gives us the basic power to construct computers "only with NAND gate"

- Gate Logic
  : A technique for implementing Boolean functions using logic gates
  - Logic gates = Elementary + Composite
    (Nand, And, Or, Not) (Mux, Adder, ⋮)

- Project 1

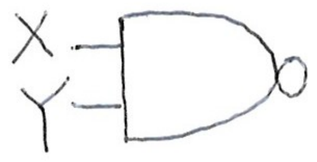
  Given: Nand

  Goal: Build the following gates

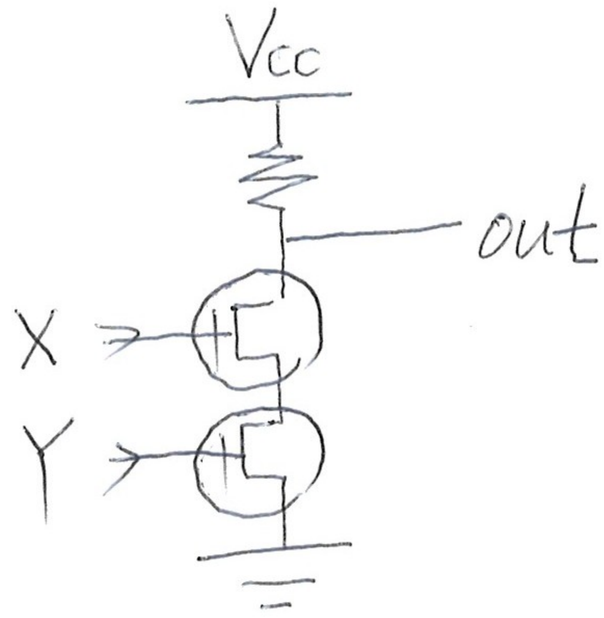
Elementary logic gates	Not And Or Xor	Mux Dmux
16-bit variants	Not 16 And 16 Or 16	Mux 16
Multi-way variants	Or 8Way	Mux 4Way 16 Mux 8Way 16 DMux 4Way DMux 8Way

• NAND to Elementary Logic Gates

– NAND

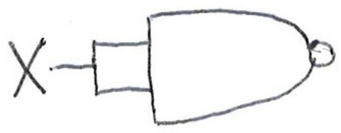


X	Y	
0	0	1
0	1	1
1	0	1
1	1	0



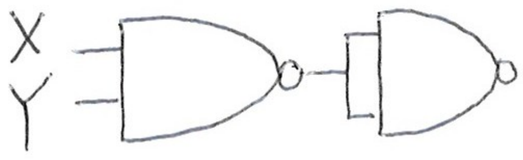
<NMOS NAND>

– NOT

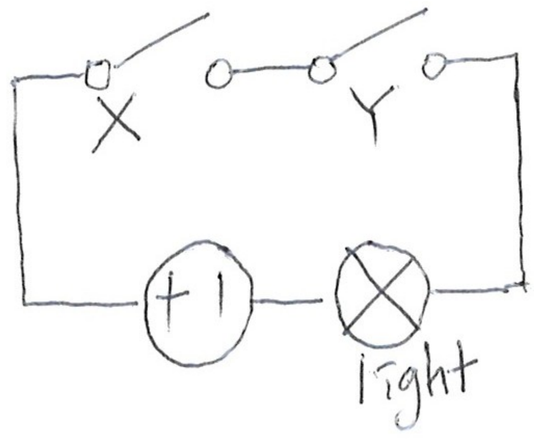


X	
0	1
1	0

– AND



X	Y	
0	0	0
0	1	0
1	0	0
1	1	1

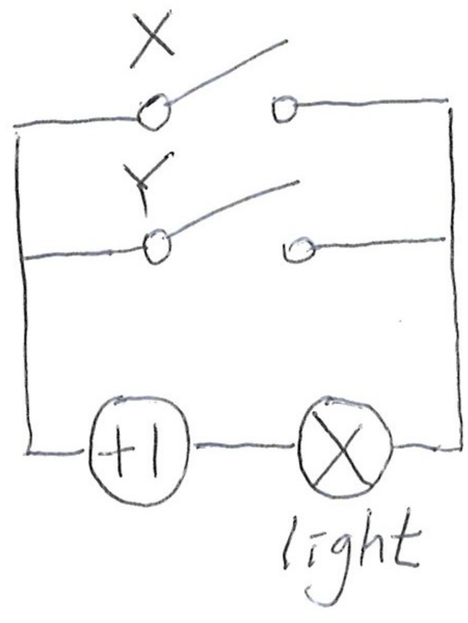


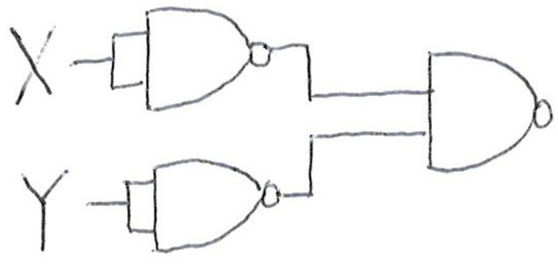
– OR

$\neg(X \vee Y) = (\neg X) \wedge (\neg Y)$: De Morgan Laws

$(X \vee Y) = \neg(\neg X \wedge \neg Y)$

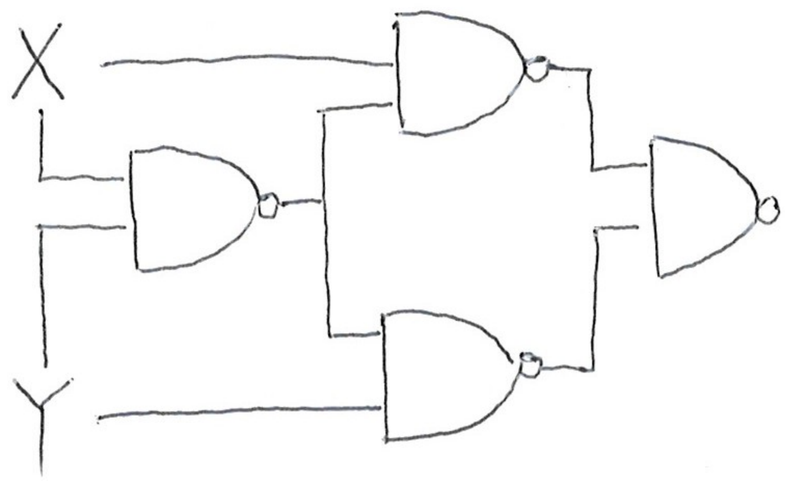
$= \neg X \text{ NAND } \neg Y$





X	Y	
0	0	0
0	1	1
1	0	1
1	1	1

– XOR



X	Y	
0	0	0
0	1	1
1	0	1
1	1	0

XOR

$$X \oplus Y = (X \wedge \neg Y) \vee (\neg X \wedge Y)$$

$$= \neg\neg \{ (X \wedge \neg Y) \vee (\neg X \wedge Y) \}$$

$$= \neg \{ \neg (X \wedge \neg Y) \wedge \neg (\neg X \wedge Y) \}$$

$$= \neg \{ (\neg X \vee Y) \wedge (X \vee \neg Y) \}$$

$$= (\neg X \vee Y) \text{ NAND } (\neg Y \vee X)$$

$$= \{ X \text{ NAND } (X \text{ NAND } Y) \} \text{ NAND } \{ Y \text{ NAND } (X \text{ NAND } Y) \}$$

$$\neg X \vee Y = (\neg X \vee X) \wedge (\neg X \vee Y)$$

$$= \neg X \vee (X \wedge Y)$$

$$= \neg \{ X \wedge \neg (X \wedge Y) \}$$

$$= X \text{ NAND } (X \text{ NAND } Y)$$

$$\neg Y \vee X = Y \text{ NAND } (X \text{ NAND } Y)$$

- MUX

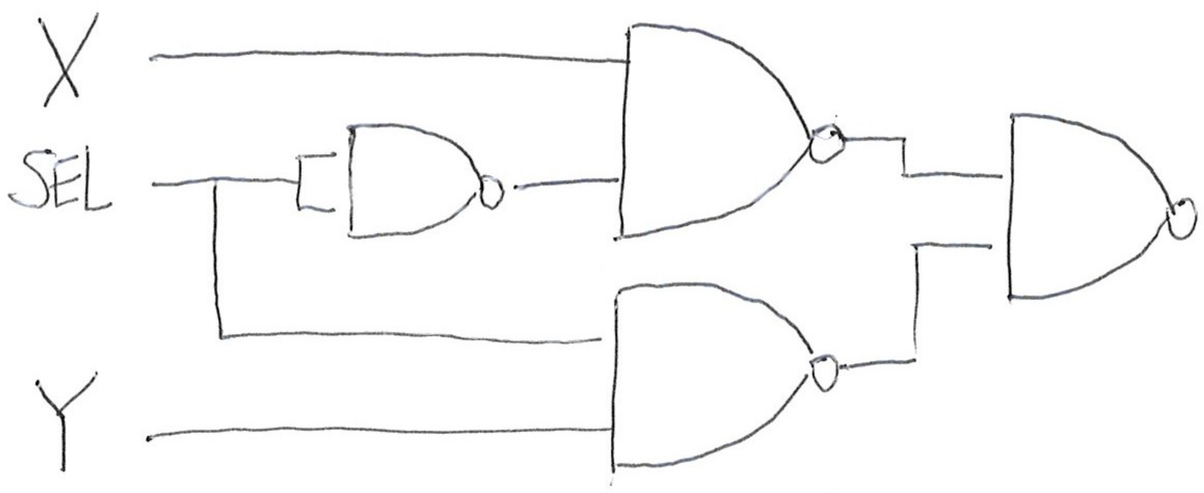
$(\neg sel \wedge X) \vee (sel \wedge Y)$

$= \neg\neg\{(\neg sel \wedge X) \vee (sel \wedge Y)\}$

$= \neg\{\neg(\neg sel \wedge X) \wedge \neg(sel \wedge Y)\}$

$= (\neg sel \text{ NAND } X) \text{ NAND } (sel \text{ NAND } Y)$

X	Y	SEL	
0	0	0	X
0	1		
1	0		
1	1		
0	0	1	Y
0	1		
1	0		
1	1		

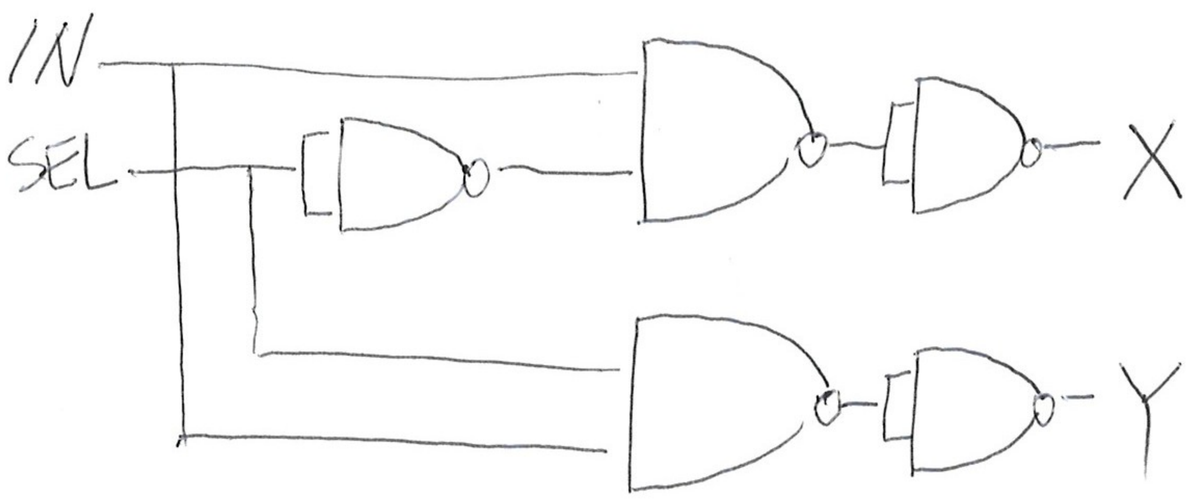


- DMUX

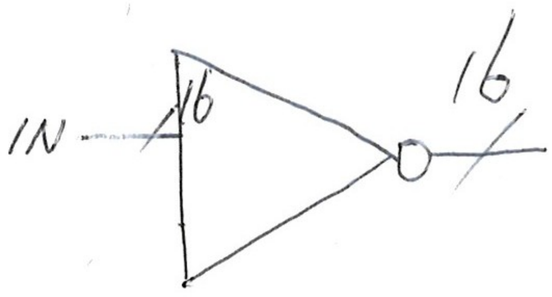
$X = IN \wedge \neg sel$

$Y = IN \wedge sel$

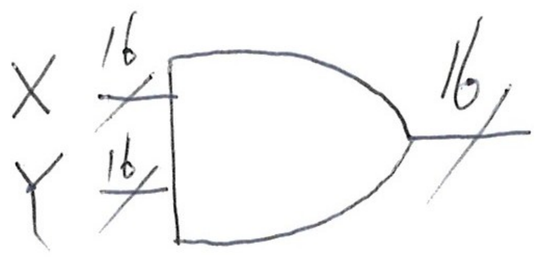
SEL	X	Y
0	IN	0
1	0	IN



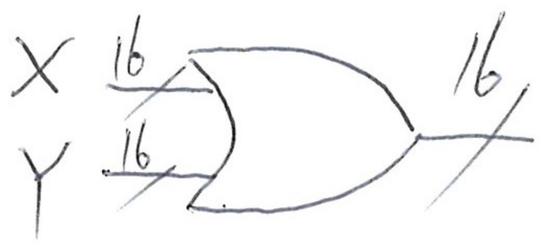
- 16-bit Variants
  - Not 16



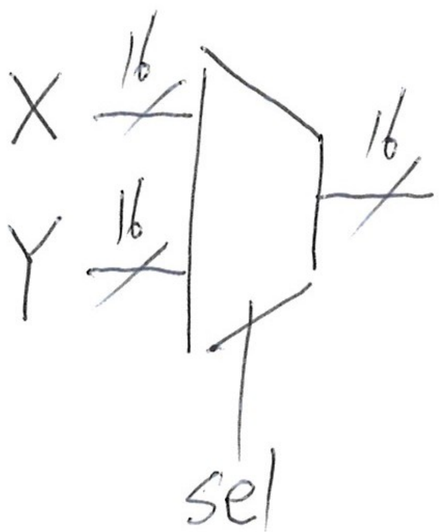
  - And 16



  - Or 16

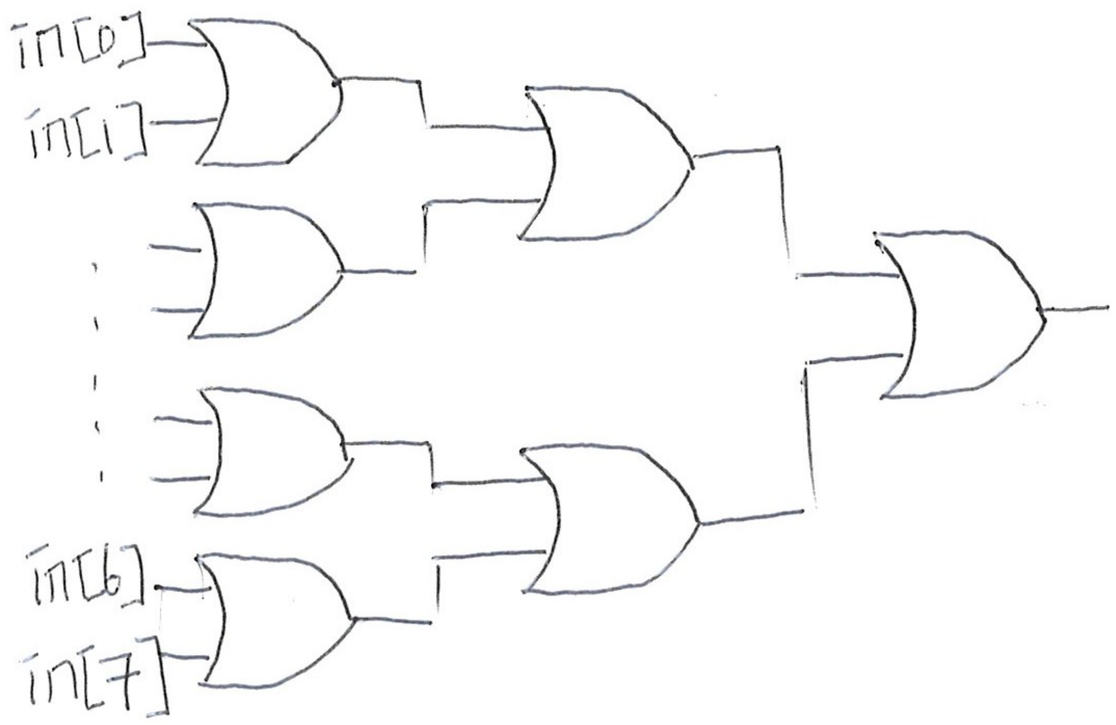


  - Mux 16

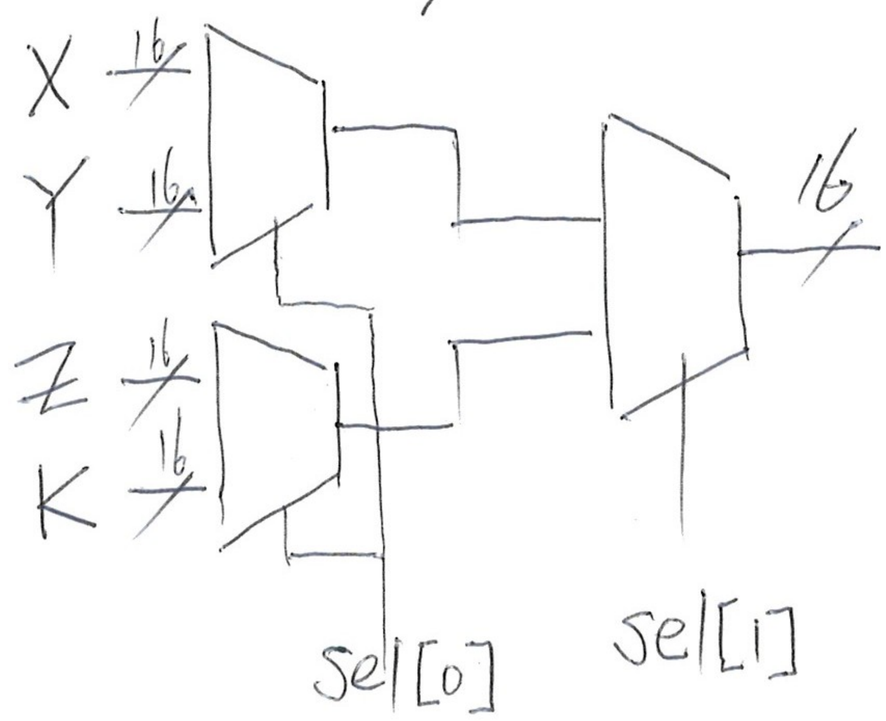


- Multi-way Variants

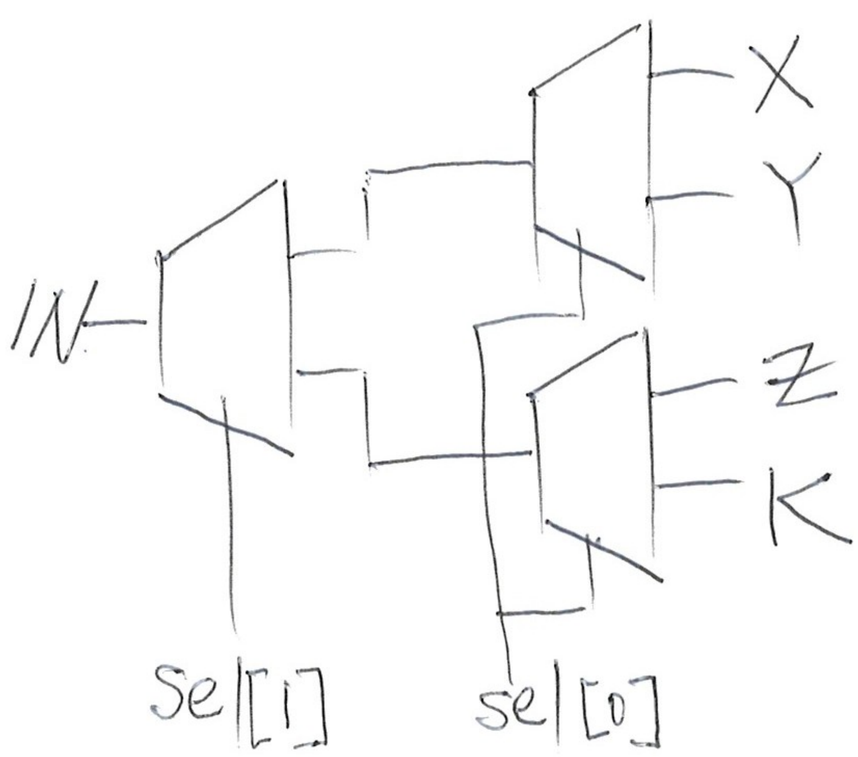
- Or8way



- Mux4way16



- DMux4way



Part 2. Boolean Arithmetic

- Binary → Decimal

$$\boxed{(X_n X_{n-1} \cdots X_0)_{base} = \sum_{i=0}^{n} X_i \times base^i}$$: general

$$\boxed{(b_n b_{n-1} \cdots b_0)_2 = \sum_{i=0}^{n} b_i \cdot 2^i}$$: binary

* Maximum with k bits

: $2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} = 2^k - 1$

- Decimal → Binary

Sol 1)

$$\begin{array}{r|l} 2 & 87 \\ \hline 2 & 43 \cdots 1 \\ \hline 2 & 21 \cdots 1 \\ \hline 2 & 10 \cdots 1 \\ \hline 2 & 5 \cdots 0 \\ \hline 2 & 2 \cdots 1 \\ \hline & 1 \cdots 0 \end{array}$$

$1010111_{(2)}$

Sol 2)

128	64	32	16	8	4	2	1
1	0	1	0	1	1	1	

$_{(2)}$

$87 - 64 = 23$

$23 - 16 = 7$

$7 - 4 = 3 = 2 + 1$

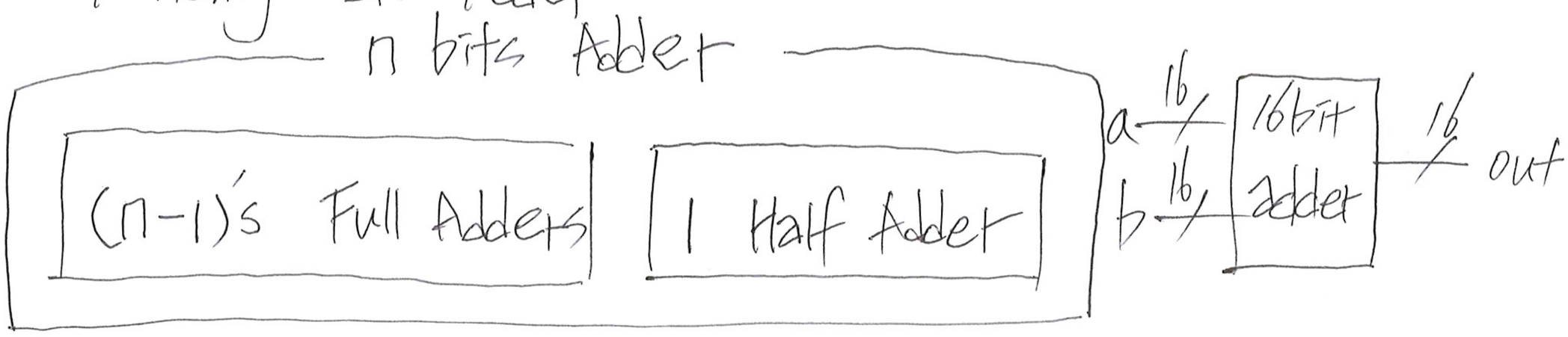
$3 - 2 = 1$

$1 - 1 = 0$

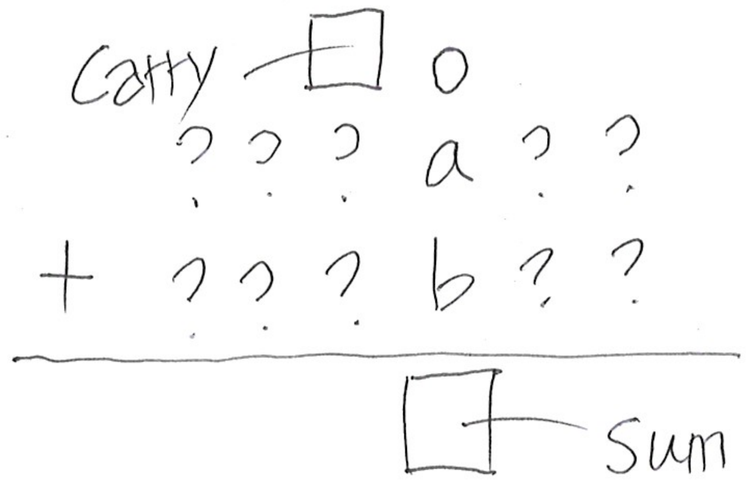
- Manipulating Binary Numbers

Division Multiplication	} software layer
Greater Subtraction	} get for free
Addition	implement

- Building an Adder

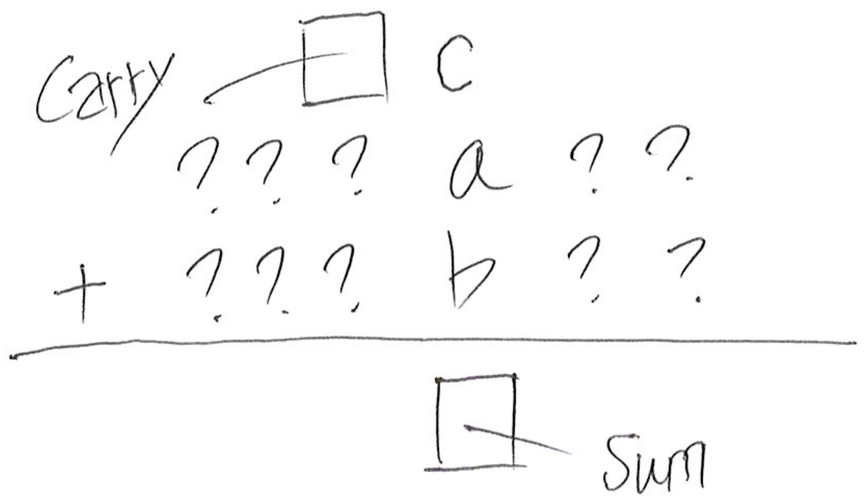


- Half Adder



a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
		XOR	AND

- Full Adder



a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$sum = a \text{ XOR } b \text{ XOR } c$

$carry = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$

$= (a \wedge b) \vee \{(a \text{ XOR } b) \wedge c\}$

- 2's complement

4bits	(15)	−1
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	(8)	−8
1001	(9)	−7
1010	(10)	−6
1011	(11)	−5
1100	(12)	−4
1101	(13)	−3
1110	(14)	−2
1111	(15)	−1
0000	0	
0001	1	
0010	2	
⋮		

$$-X \simeq (2^n - X) = (?)$$

$$2^n - (?) = X$$

- λ to $-\lambda$

$$-\lambda \simeq (2^n - \lambda) = 1 + \underbrace{(2^n - 1)}_{1111_{(2)}} - \lambda = \begin{array}{r} 1111_{(2)} \\ -\ \lambda \\ \hline +\ 1 \\ \hline \end{array}$$

- $\boxed{-x = !x + 1}$

- $x \to x-1$

$-(x-1) = !(x-1) + 1$

$\boxed{-x = !(x-1)}$

- $-x = !x+1$

$-x-2 = !x-1$

$!(-x-2) = !(!x-1)$

$(!x = -x-1)$이므로

$-(-x-2)-1 = !(!x-1)$

$\boxed{x+1 = !(!x-1)}$

- $-x = !x+1$

$-x+y-1 = !x+y$

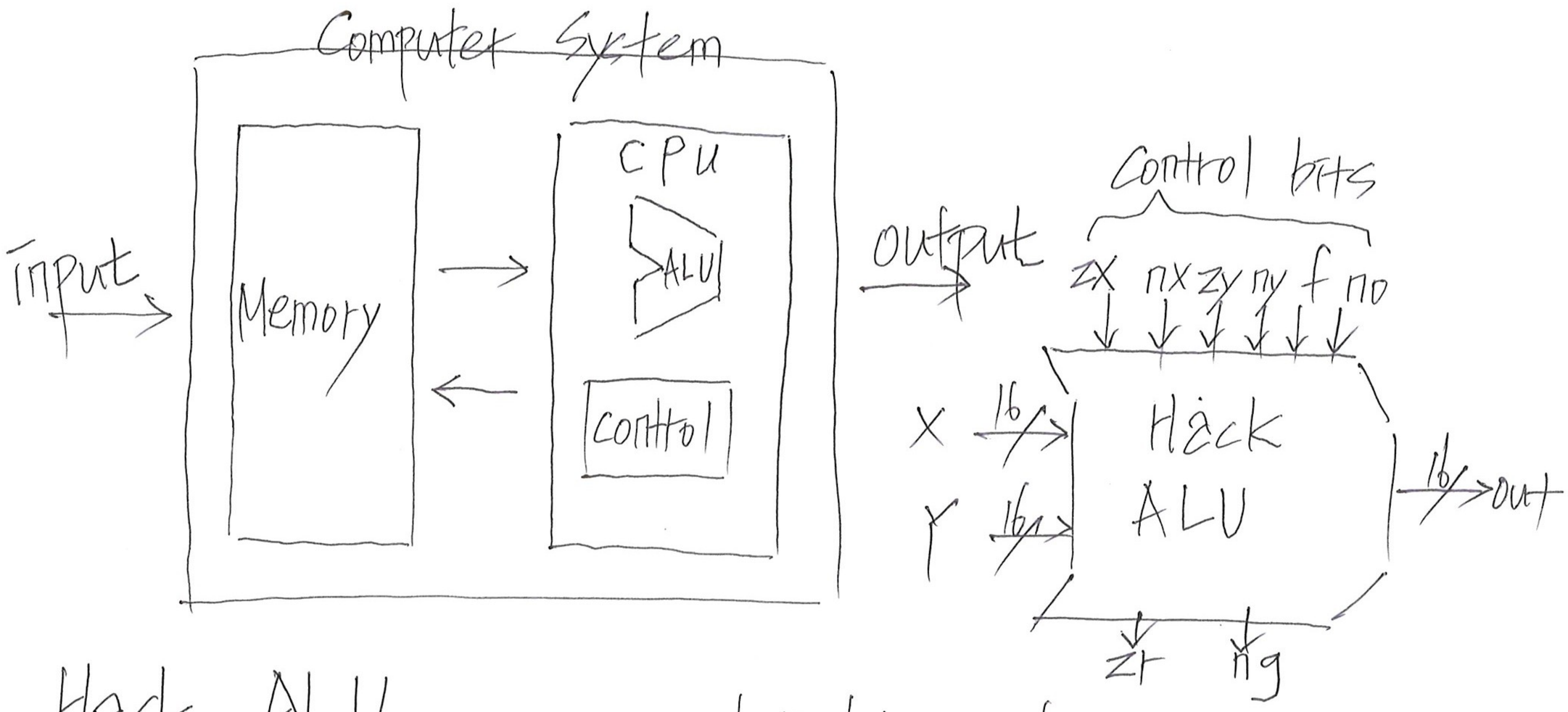
$!(-x+y-1) = !(!x+y)$

$(!x = -x-1)$이므로

$-(-x+y-1)-1 = !(!x+y)$

$\boxed{x-y = !(!x+y)}$

- Von Neumann Architecture



- Hack ALU

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then $\lambda=0$	if nx then $x=!\lambda$	if zy then $y=0$	if ny then $y=!y$	if f then out=$\lambda+y$ else out=$\lambda \& y$	if no then out=!out	out(λ,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!λ
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	λ-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

if out==0 then zr=1, else zr=0
if out<0 then ng=1, else ng=0

Hack ALU

zx	nx	x	zy	ny	y	f	f(x,y)	no	out
if zx then x=0	if nx then x=!x		if zy then y=0	if ny then y=!y		if f then out=x+y else out=x&y		if no then out=!out	out(x,y)
1	0	0000	1	0	0000	1	0000	0	0
1	1	1111	1	1	1111	1	1110	1	1
1	1	1111	1	0	0000	1	1111	0	-1
0	0	x	1	1	1111	0	x	0	x
1	1	1111	0	0	y	0	y	0	y
0	0	x	1	1	1111	0	x	1	!x
1	1	1111	0	0	y	0	y	1	!y
0	0	x	1	1	1111	1	x-1	1	-x
1	1	1111	0	0	y	1	y-1	1	-y
0	1	!x	1	1	1111	1	!x-1	1	x+1
1	1	1111	0	1	!y	1	!y-1	1	y+1
0	0	x	1	1	1111	1	x-1	0	x-1
1	1	1111	0	0	y	1	y-1	0	y-1
0	0	x	0	0	y	1	x+y	0	x+y
0	1	!x	0	0	y	1	!x+y	1	x-y
0	0	x	0	1	!y	1	x+!y	1	y-x
0	0	x	0	0	y	0	x&y	0	x&y
0	1	!x	0	1	!y	0		1	x|y

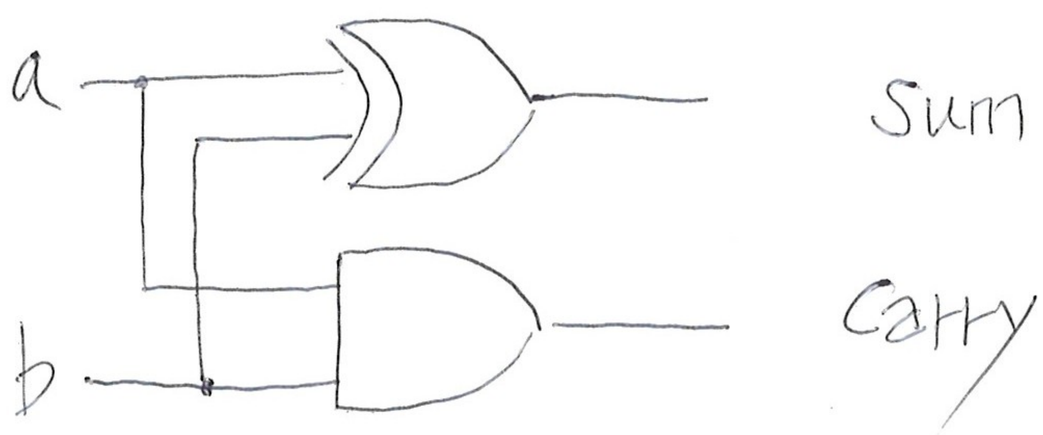
Project 2

Given : All the chips built in Project 1

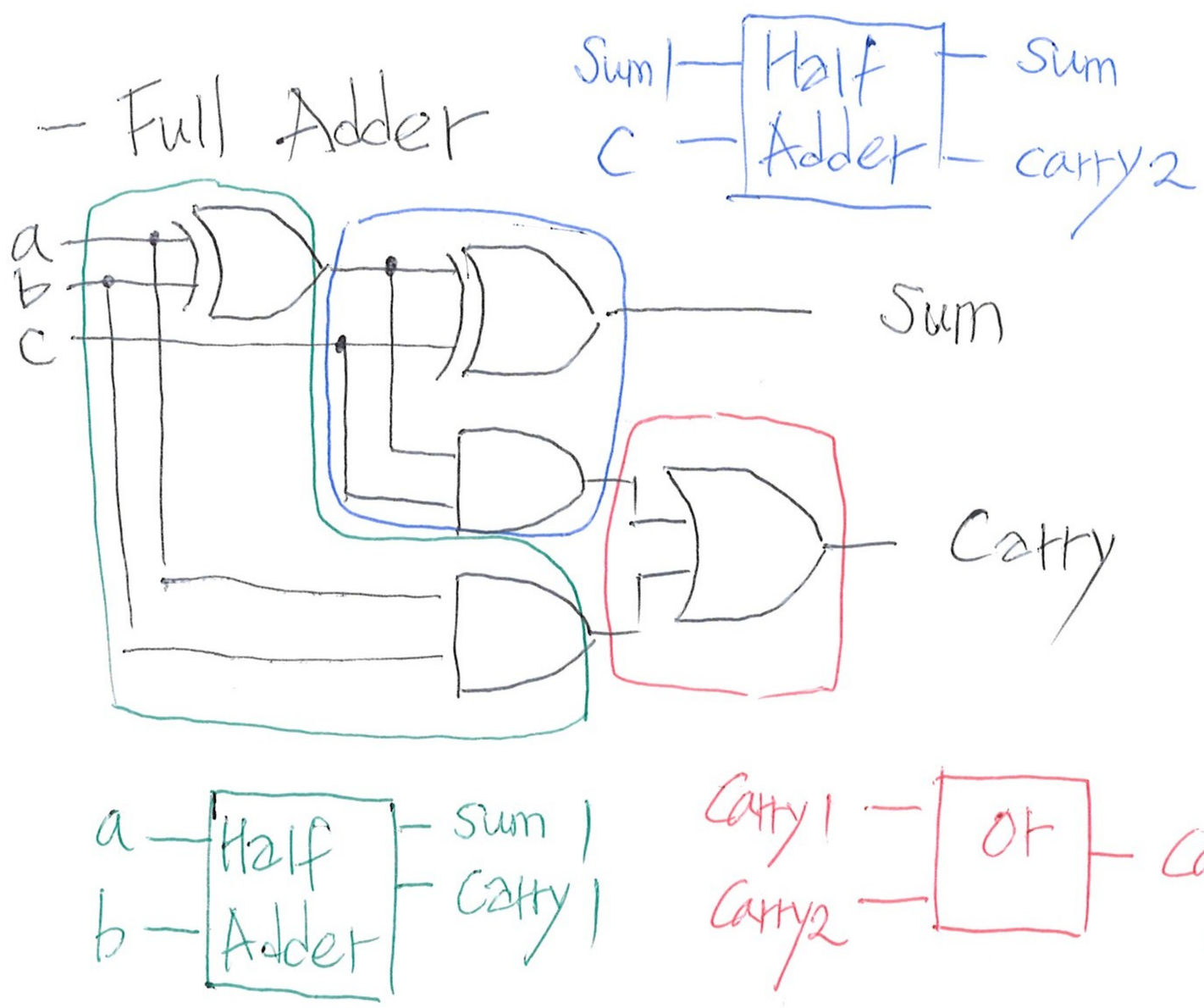
Goal : Build the following chips

Half Adder
Full Adder
Add 16
Inc 16
ALU

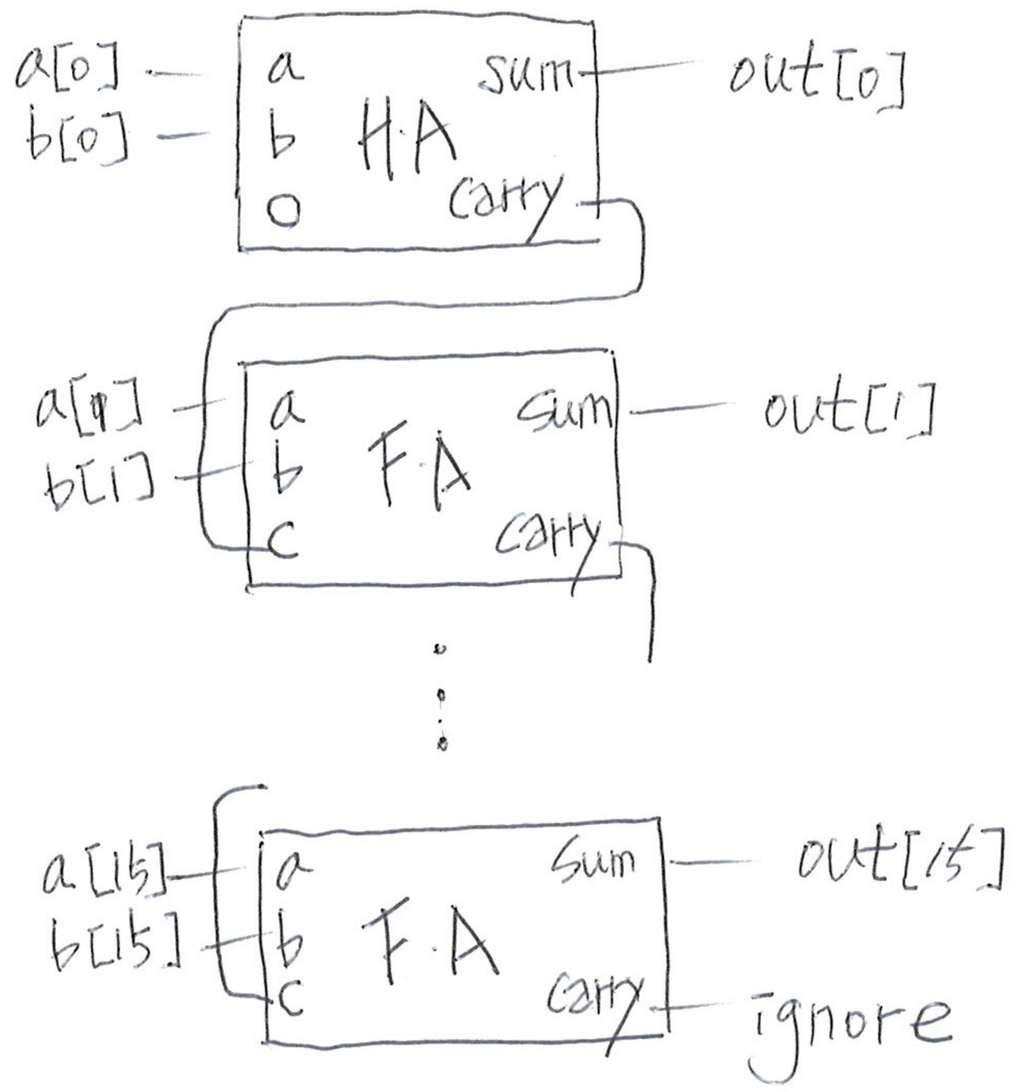
- Half Adder



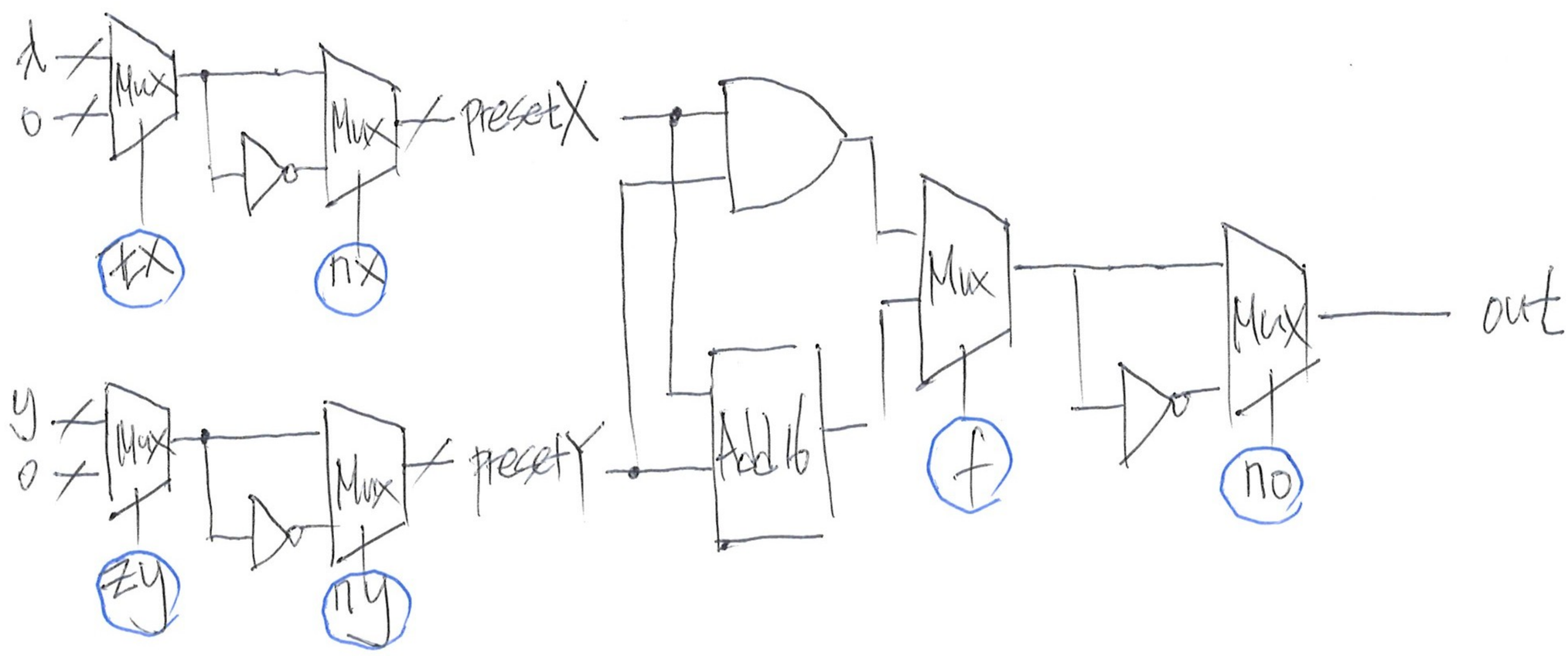
- Full Adder

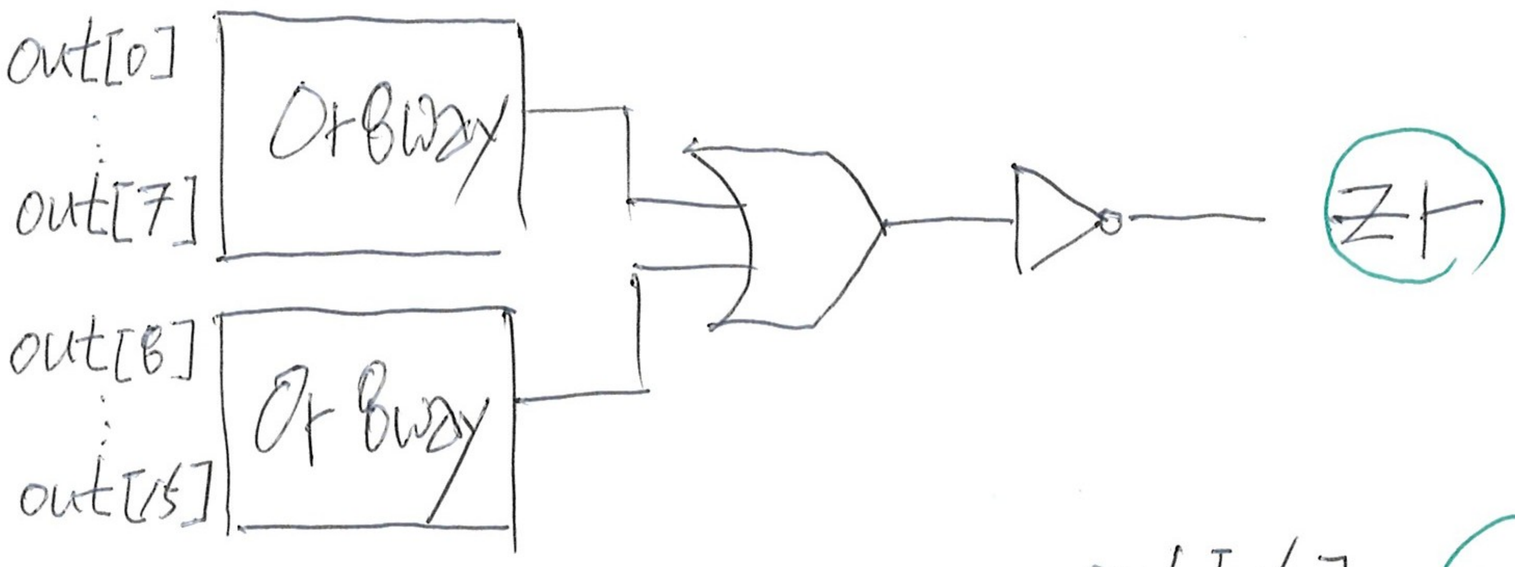


- Add 16



- ALU



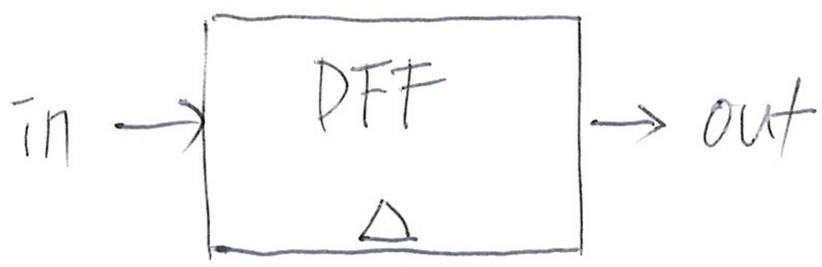


out[15] = ng

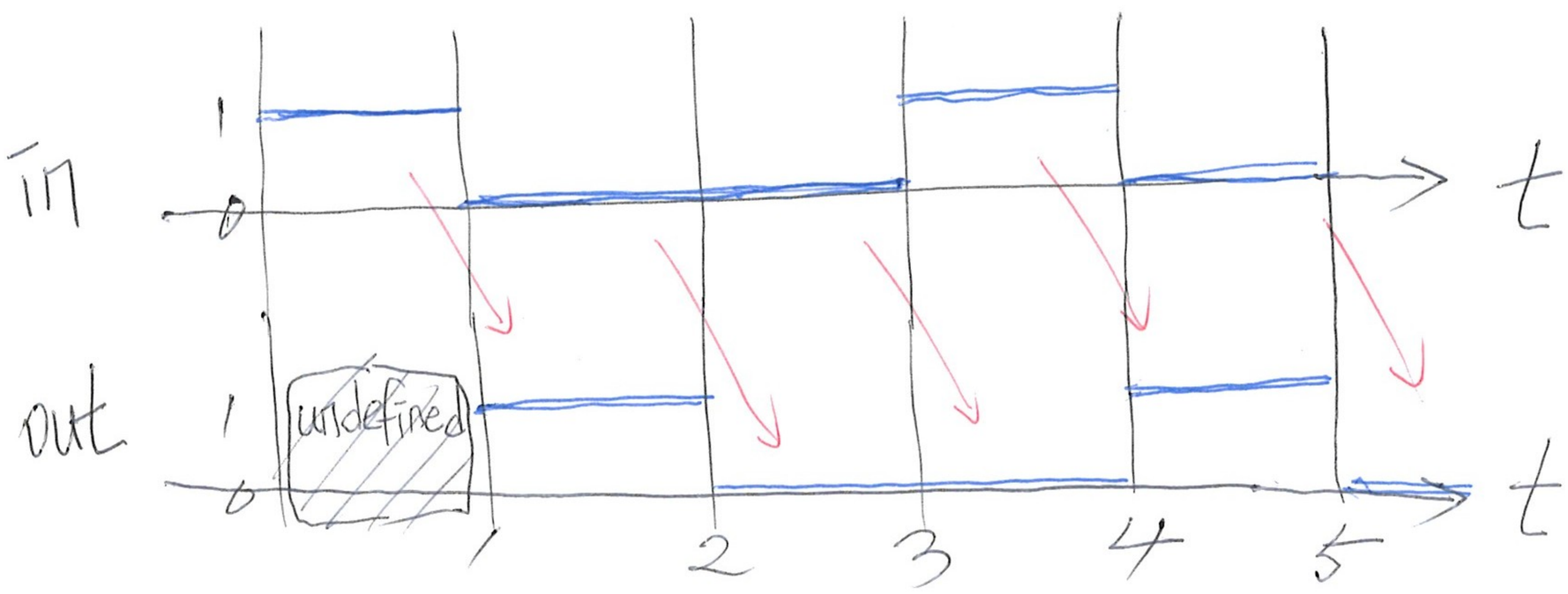
Part 3. Memory

- Combinatorial Logic : out[t] = function(in[t])
- Sequential Logic : state[t] = function(state[t-1])

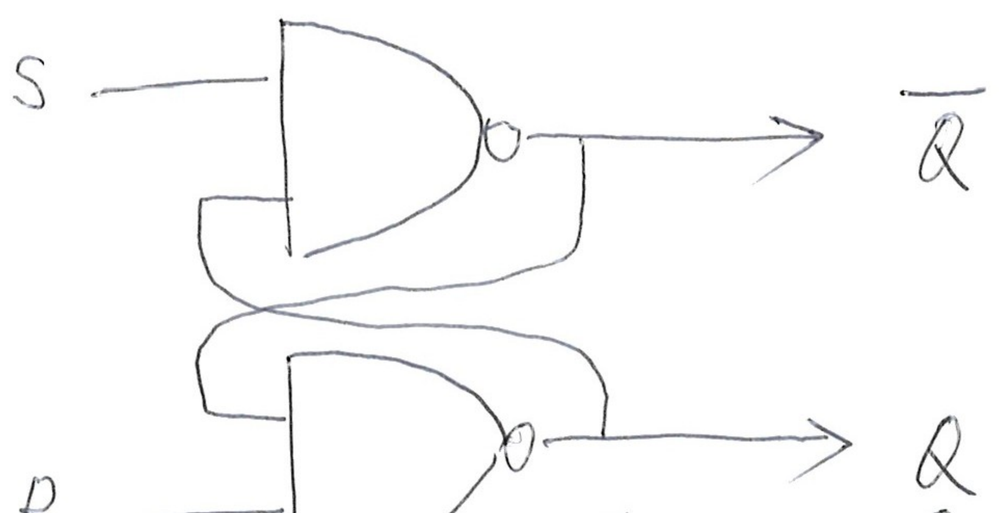
- Flip Flops



out[t] = in[t-1]



- SR Latch



	Q_n		Q_{n+1}		
S R	Q	$\overline{Q}$	Q	$\overline{Q}$	
0 0	0 /	/ 0	0 /	/ 0	No Change
0 1	0 /	/ 0	0 /	0 /	Reset
1 0	0 /	/ 0	/ 0	/ 0	Set
1 1	0 /	/ 0			Indeterminate

Project 3

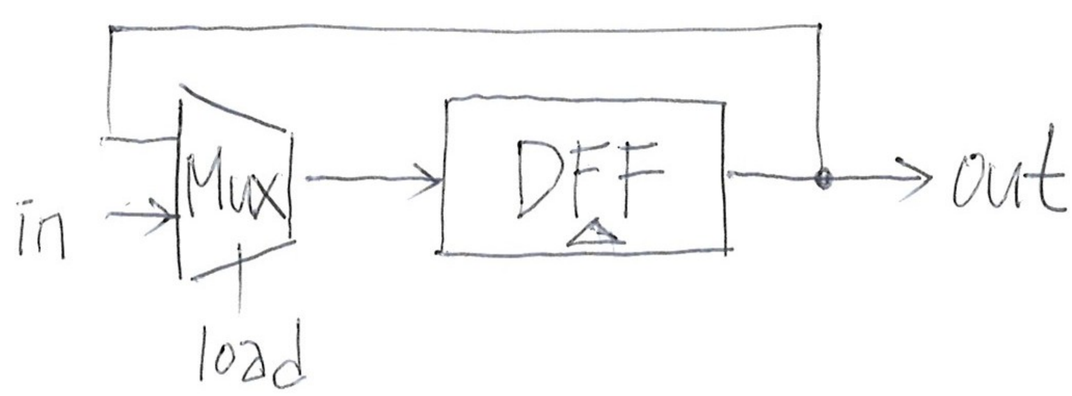
Given

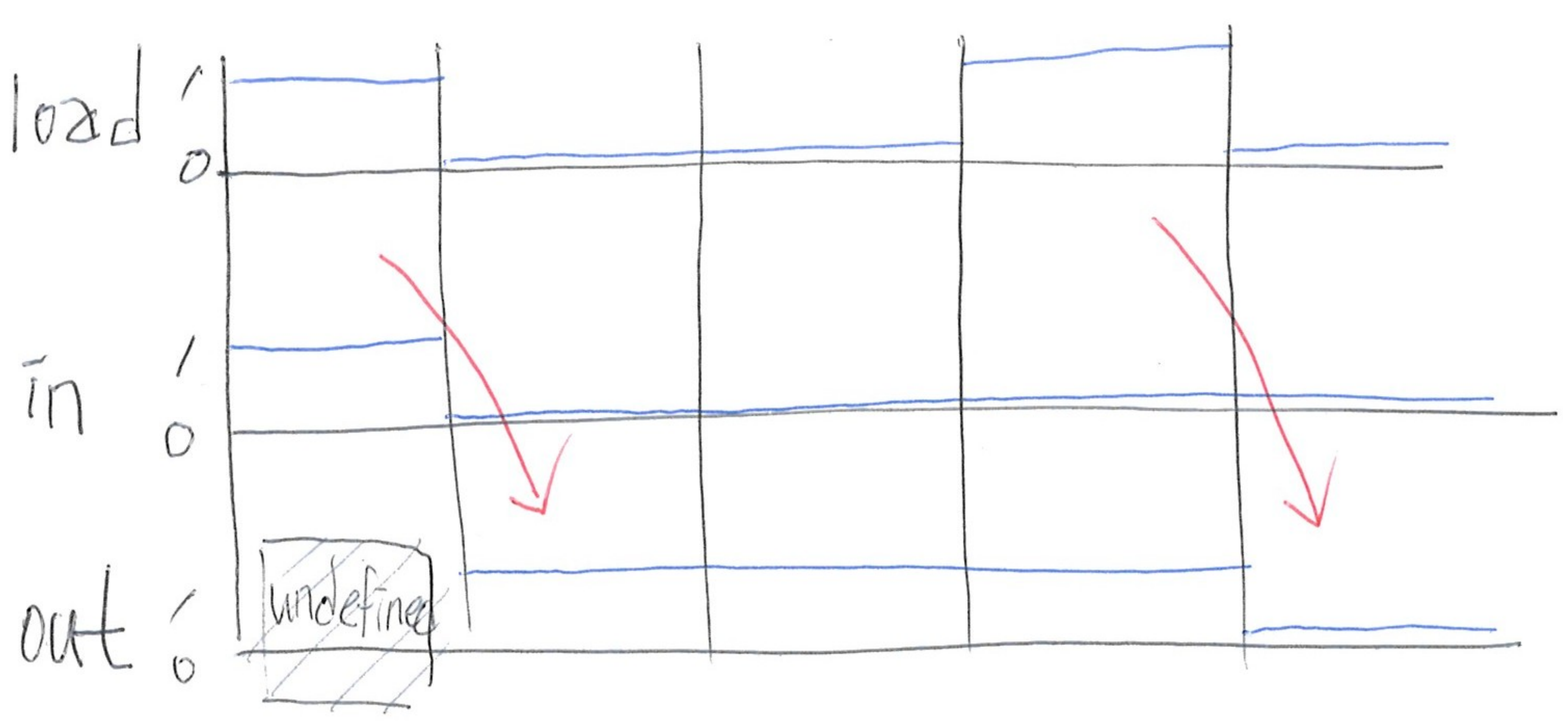
- All the chips built in Projects 1 and 2
- Data Flip-Flop (DFF gate)

Goal: Build the following chips

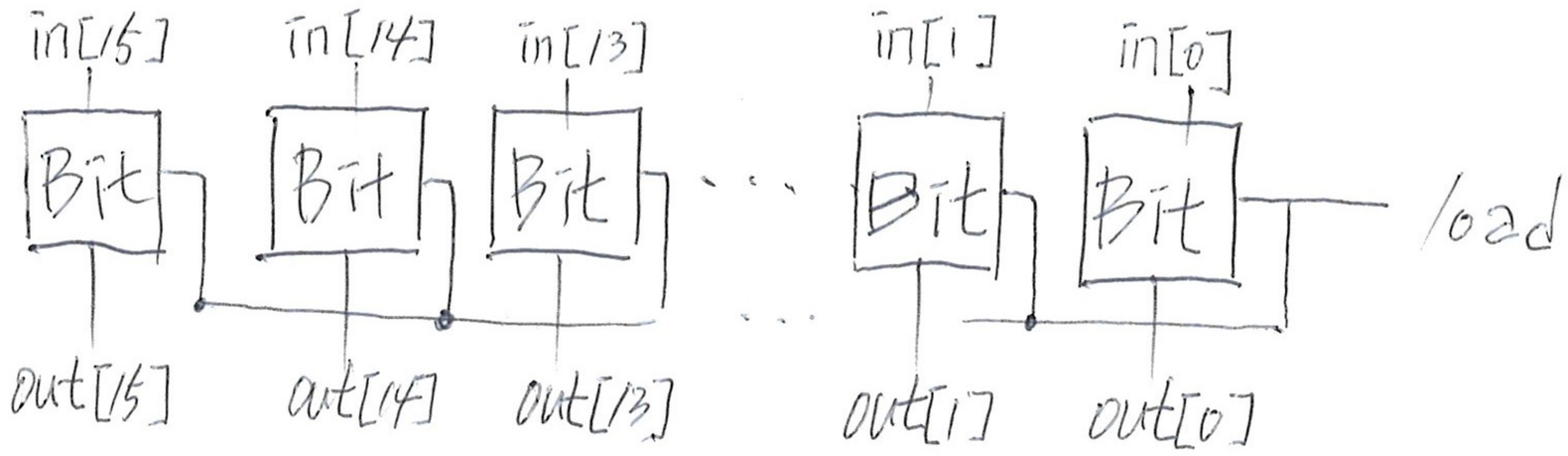
Bit
Register
RAM8
RAM64
RAM512
RAM 4K
RAM 16K
PC

- 1- Bit Register

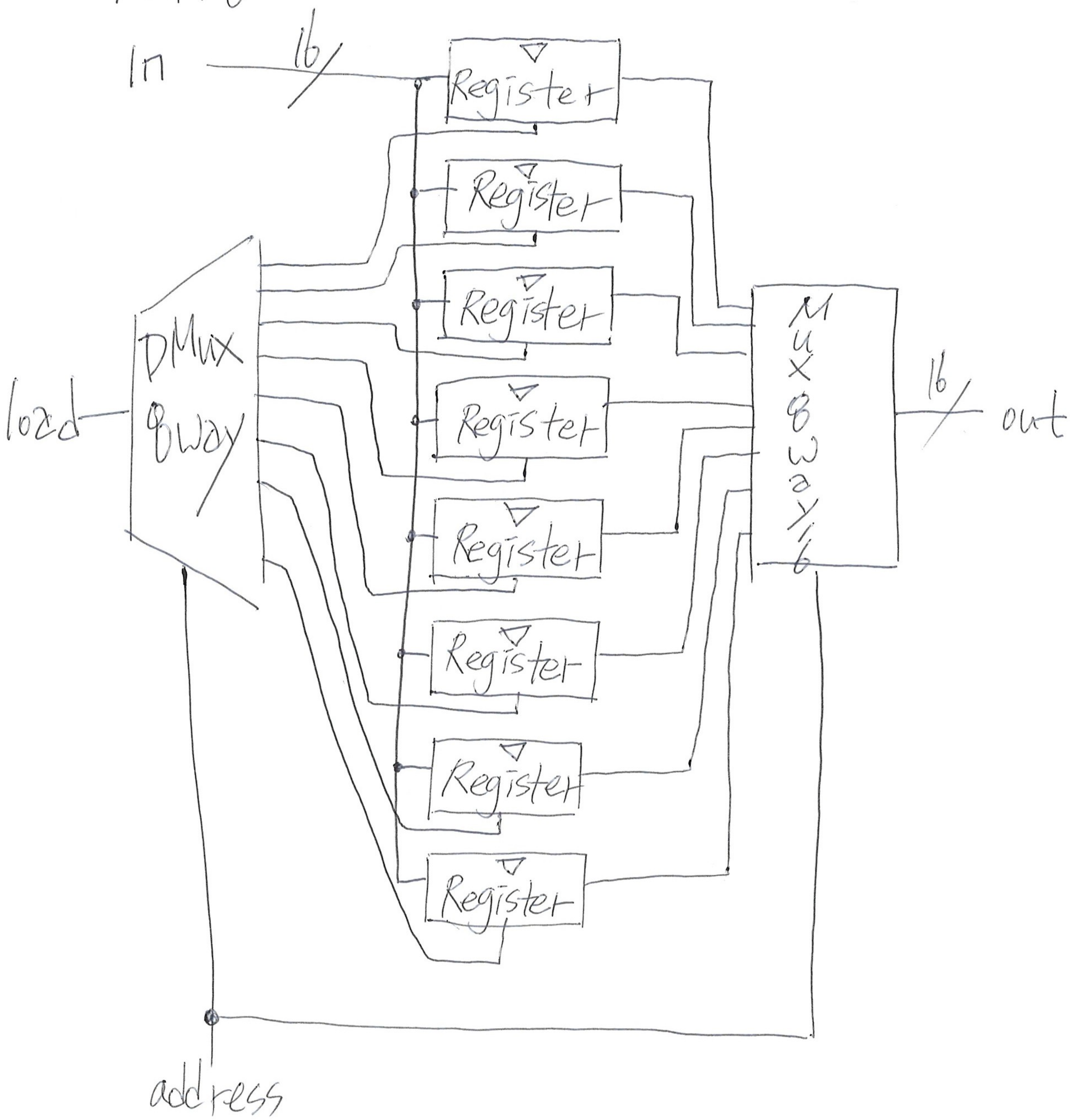




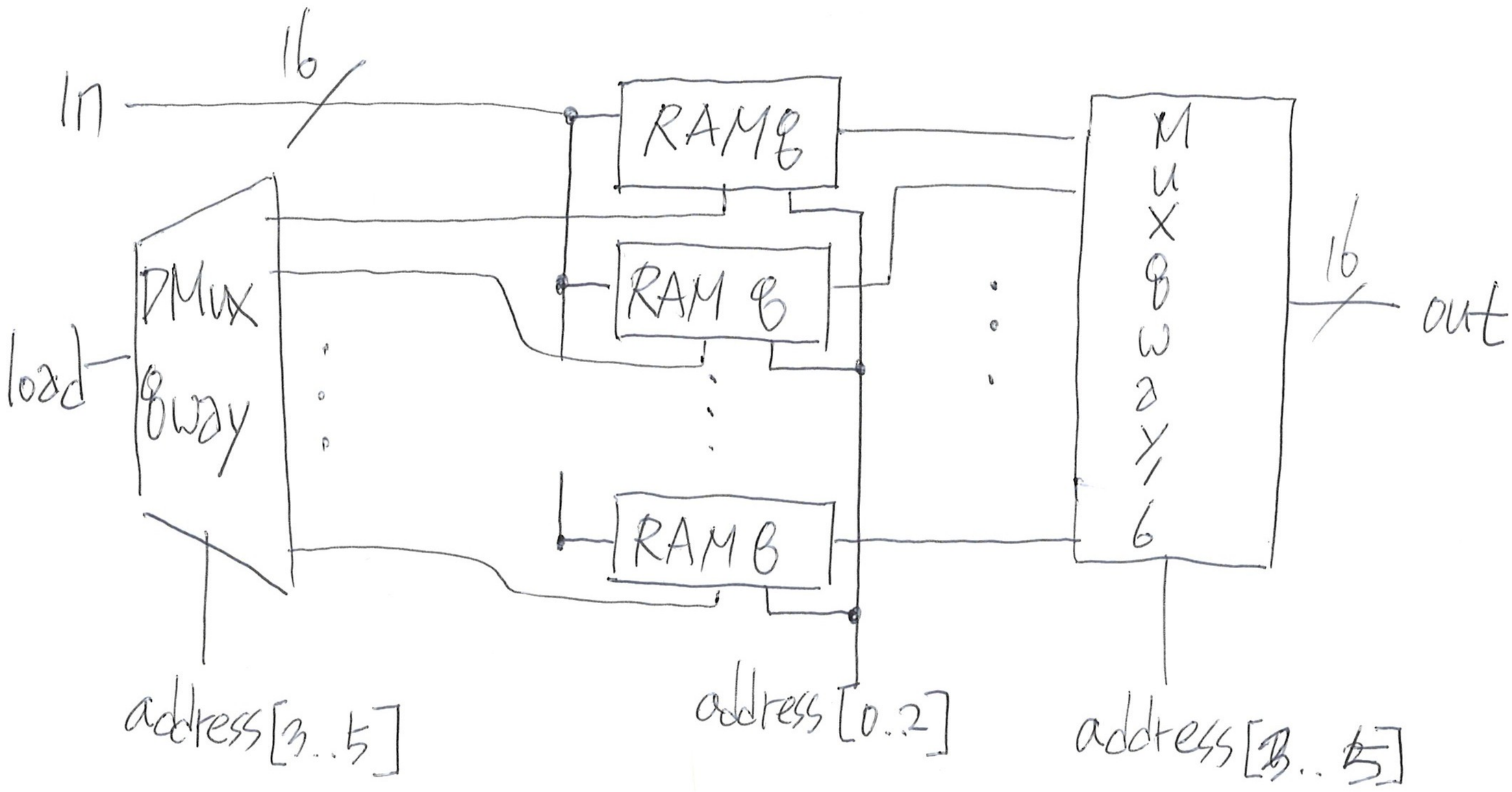
- 16-bit register



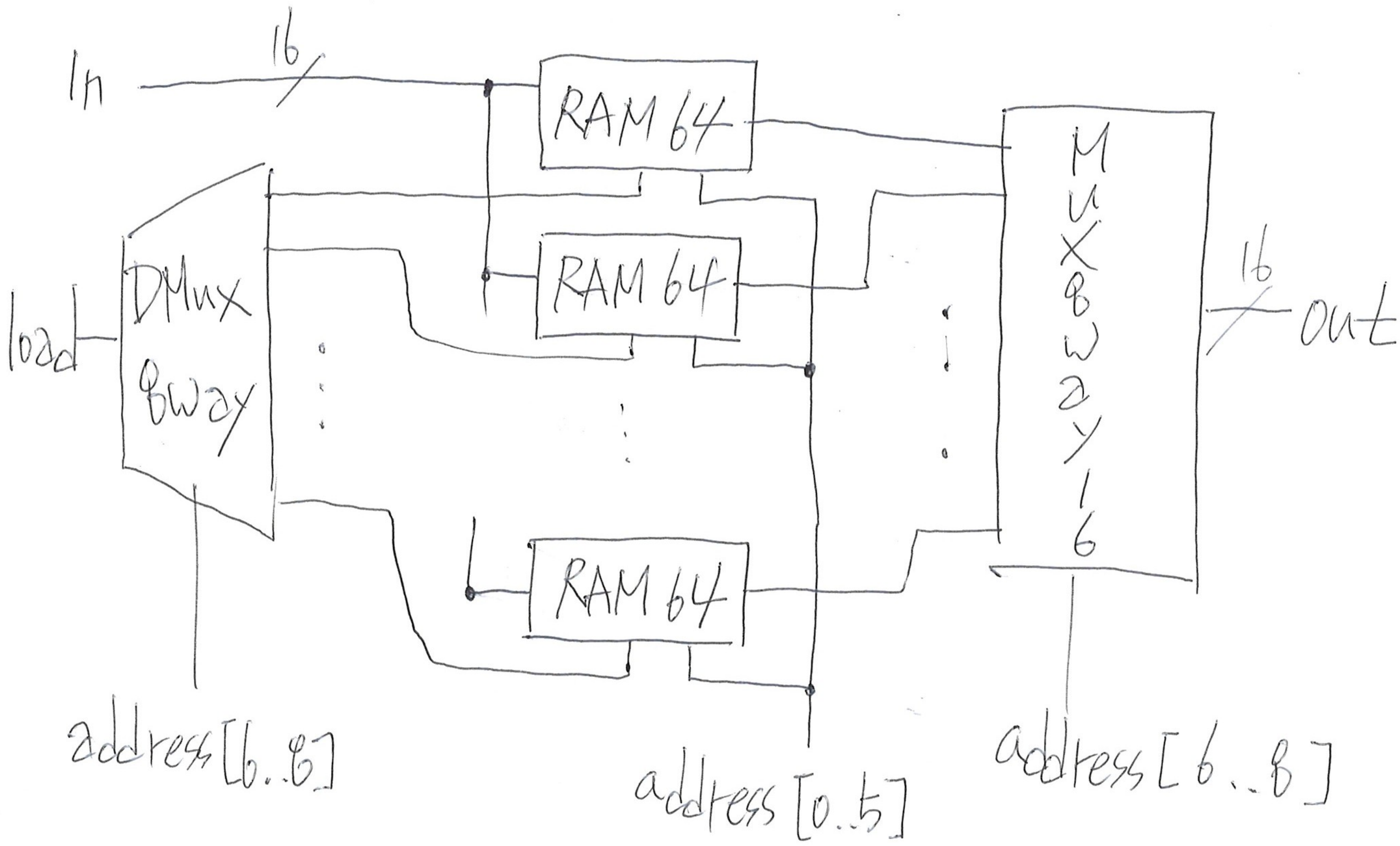
- RAM8



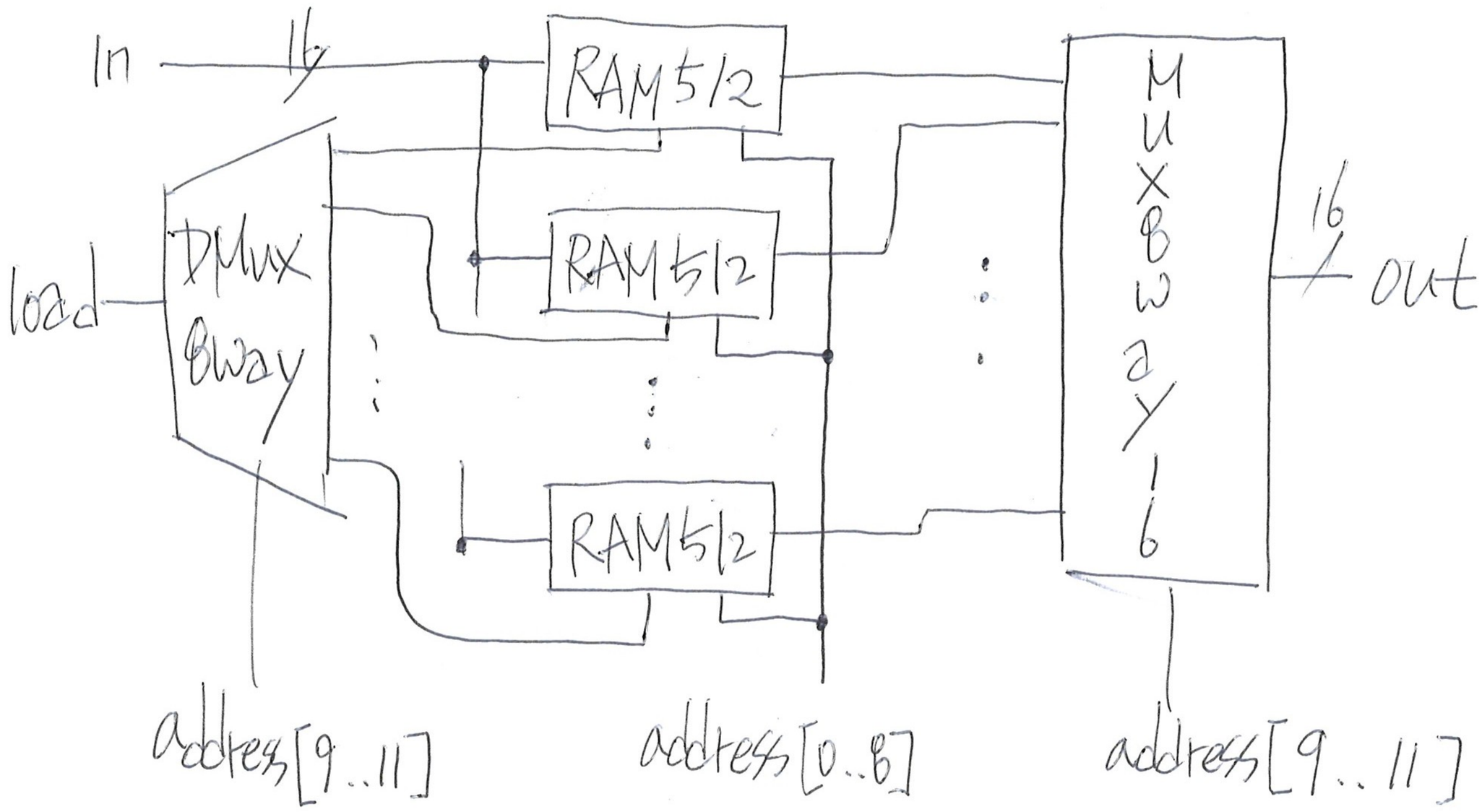
- RAM64



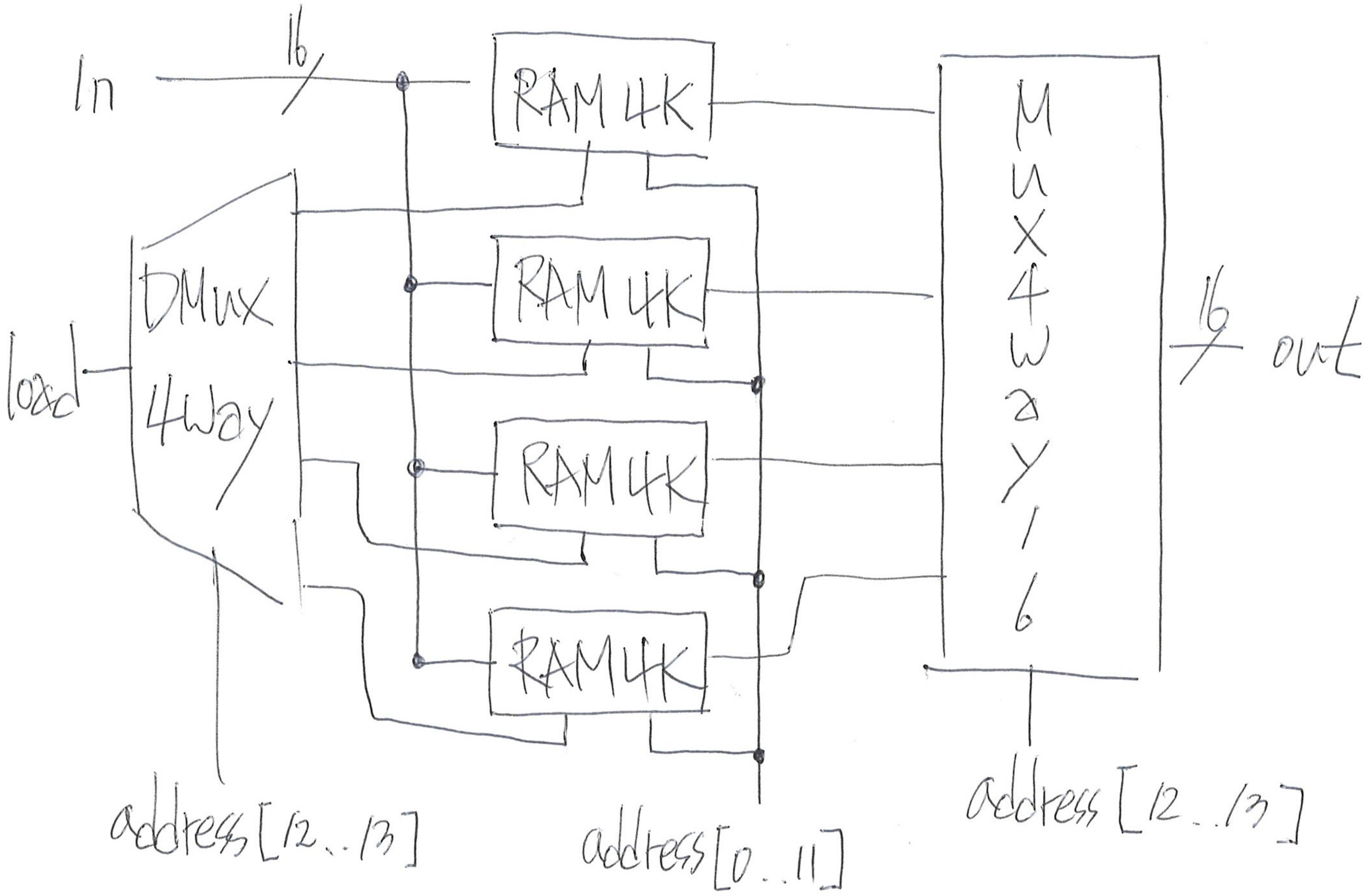
- RAM512



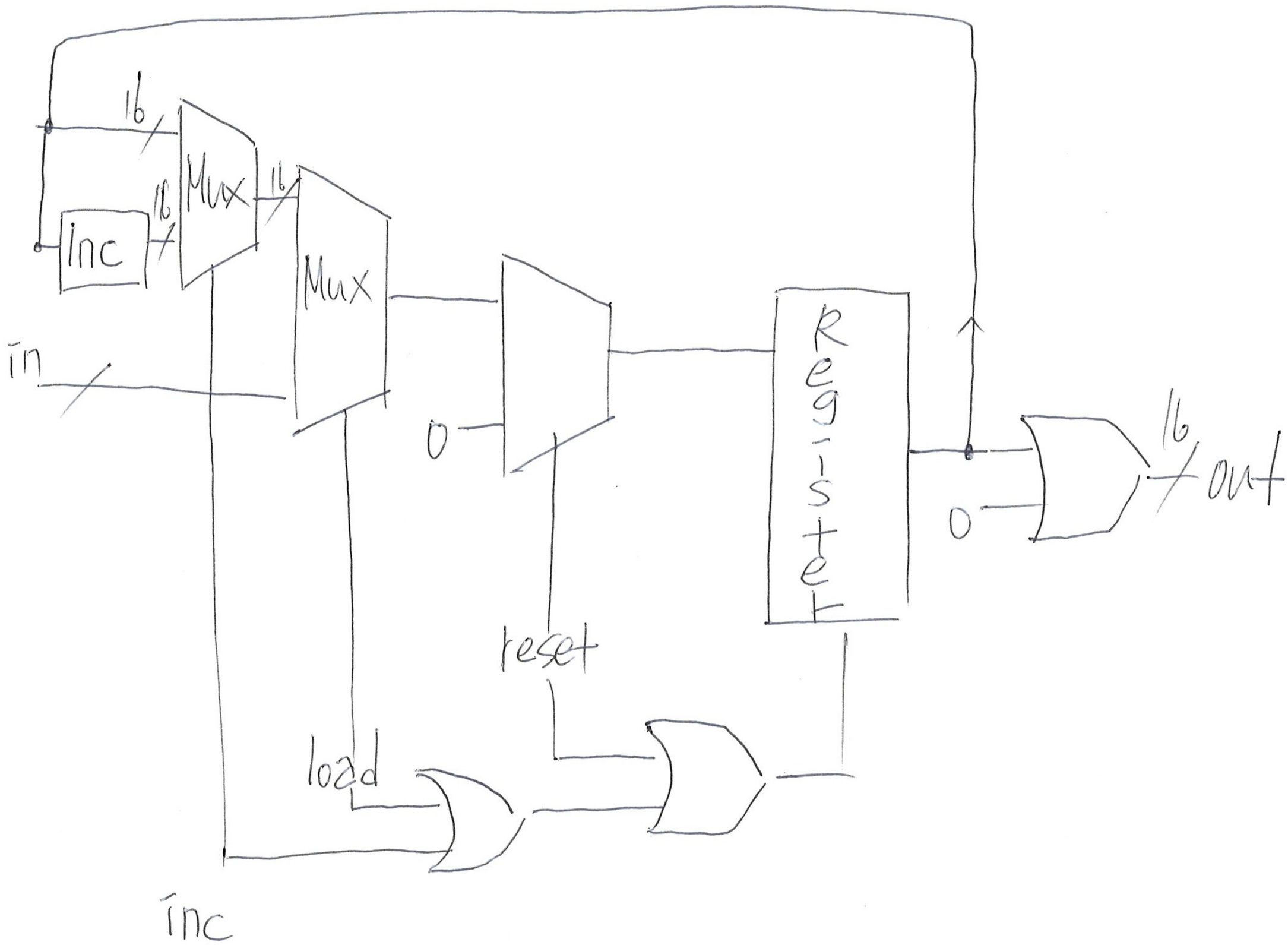
– RAM4K



– RAM16K

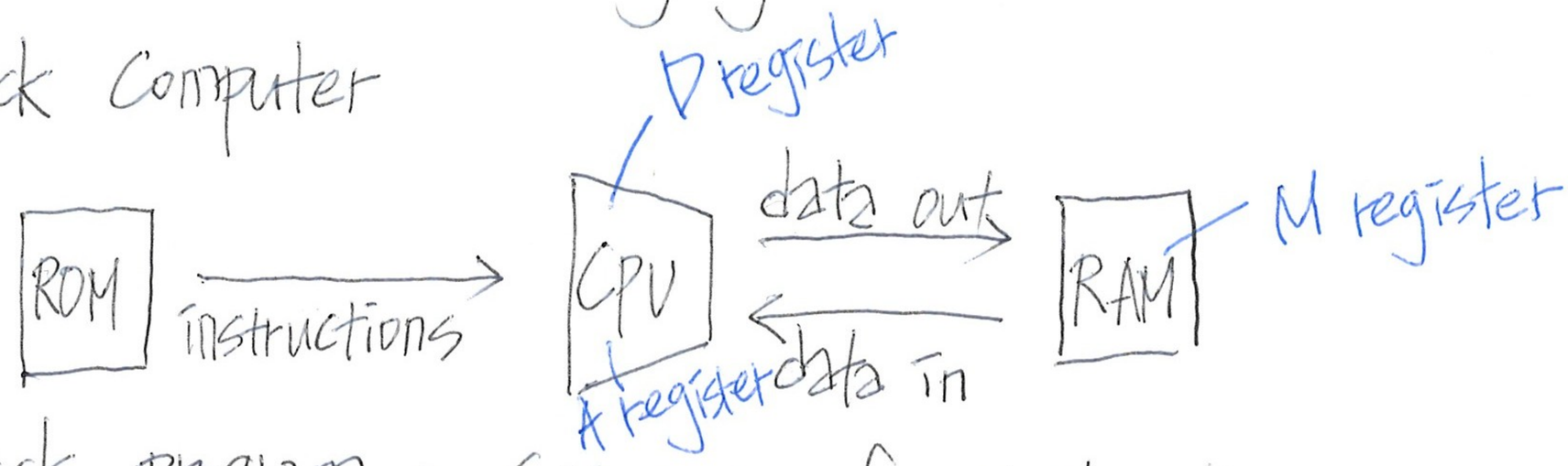


- PC



Part 4. Machine Language

- Hack Computer



- Hack program : sequence of instructions
- Hack machine language
  - A instructions
  - C instructions

- A instruction
  Symbolic Syntax : @ value
  Binary Syntax : | 0 | 15 bit binary # |
  op code

- C instruction
  Symbolic Syntax : dest = comp ; jump
  Binary Syntax : | 1 | 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3
  op code

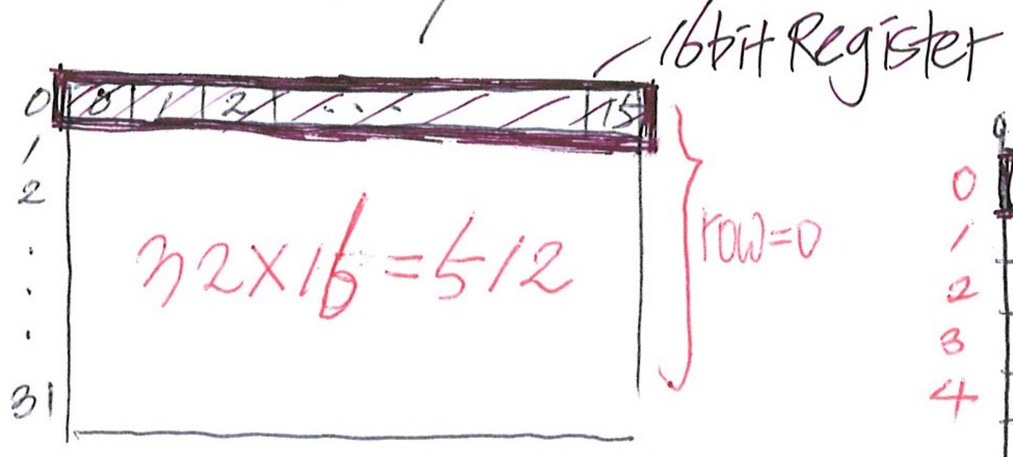
comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

dest	d1	d2	d3	effect: value is stored in
null	0	0	0	x
M	0	0	1	RAM[A]
D	0	1	0	D Register
MD	0	1	1	RAM[A], D Register
A	1	0	0	A Register
AM	1	0	1	A Register, RAM[A]
AD	1	1	0	A,D Register
AMD	1	1	1	A,D Register, RAM[A]

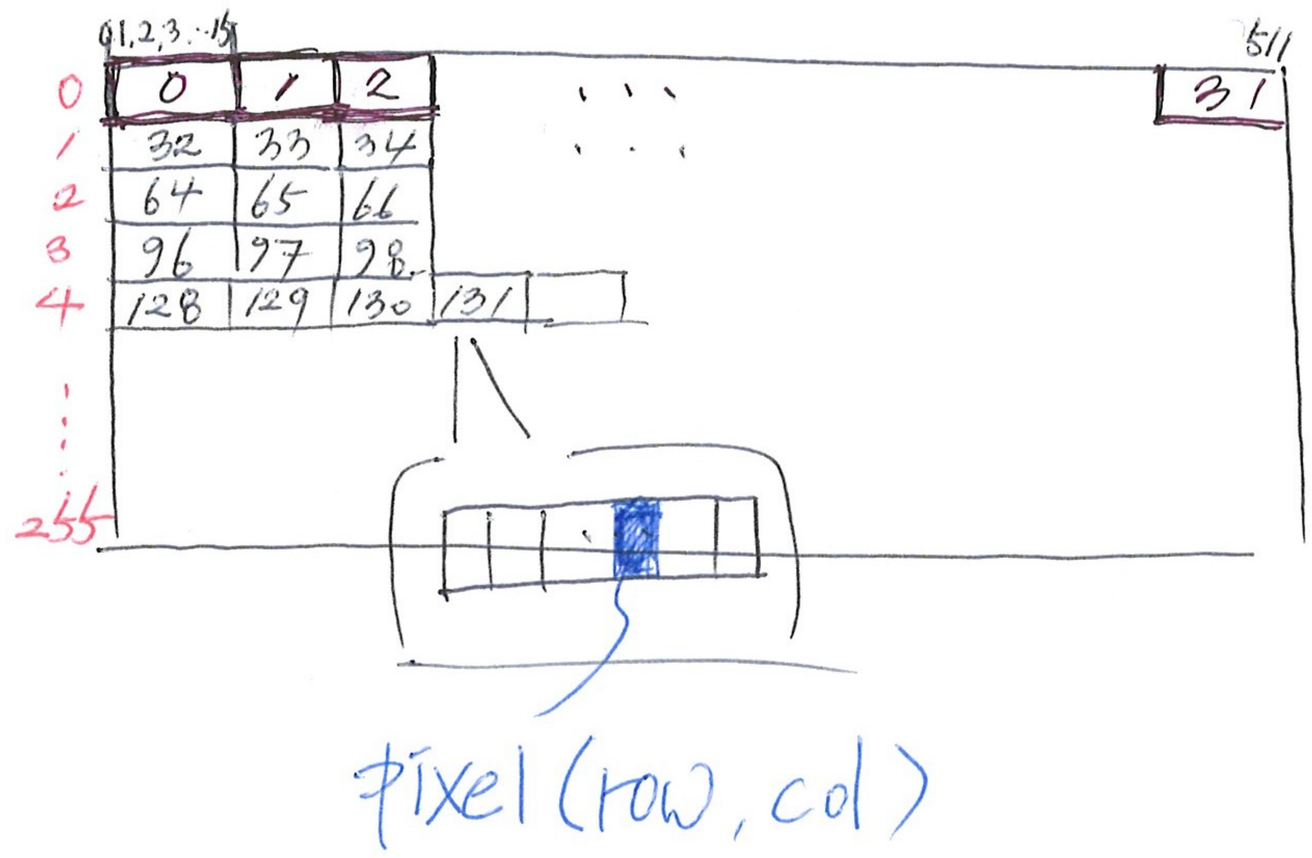
jump	j1	j2	j3	effect
null	0	0	0	x
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	out = 0
JGE	0	1	1	out ≥ 0
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	out ≠ 0
JLE	1	1	0	out ≤ 0
JMP	1	1	1	jump

- Screen memory map

<memory>



<512 × 256 Screen>



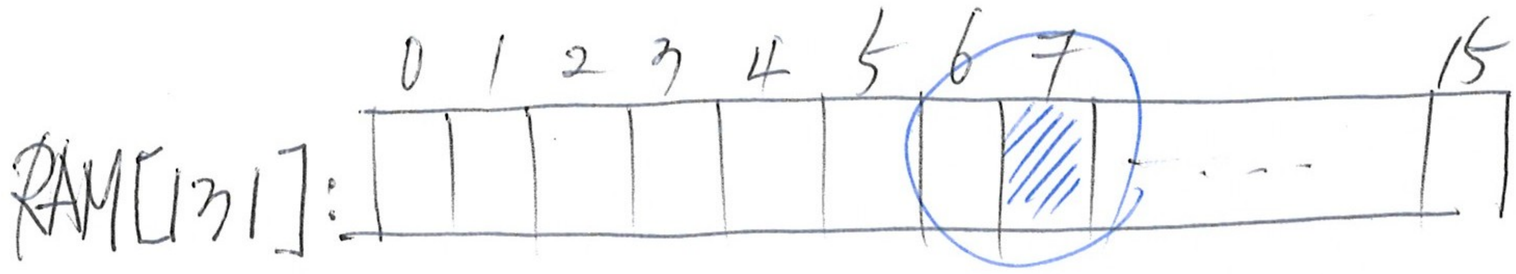
step1: word = RAM[32Row + col/16]

step2: (col % 16)th bit of word.

ex) pixel(4, 55) ??

step1: word = RAM[128 + 3] = RAM[131]

step2: 7th bit of RAM[131]



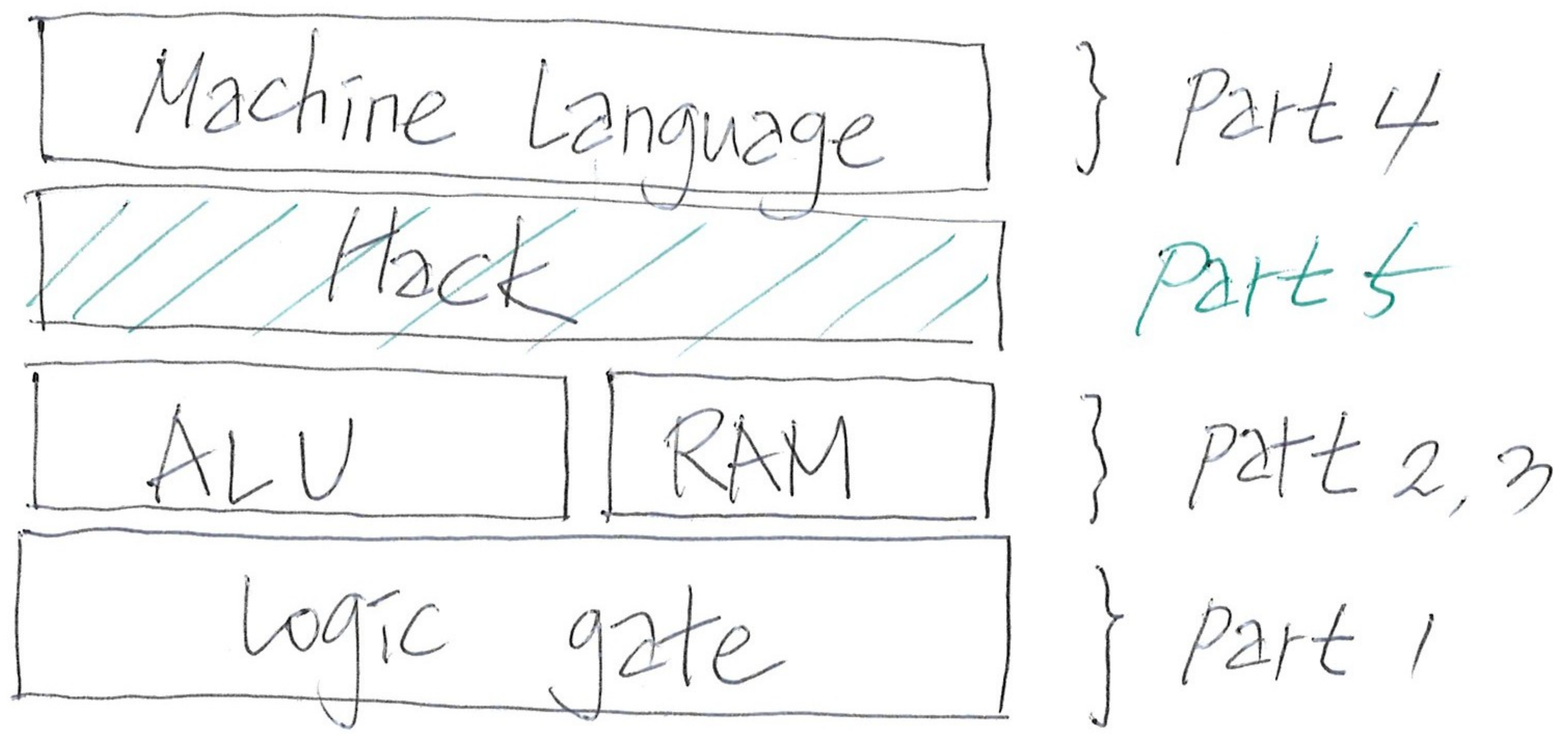
- Built-in symbols

symbol	value
R0	0
R1	1
R2	2
⋮	⋮
R15	15
SCREEN	16384
KBD	24576

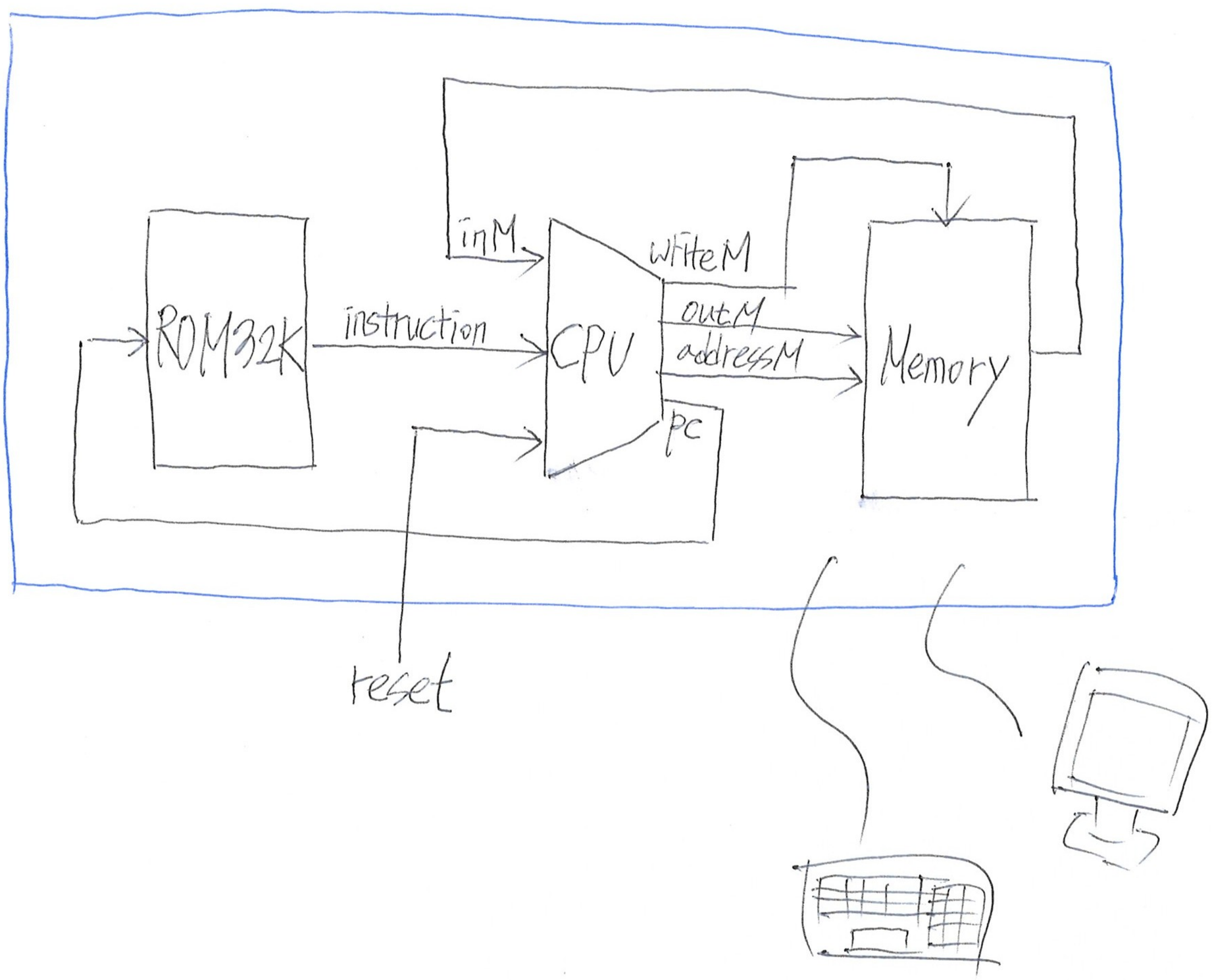
Symbol	value
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

- Virtual registers : R0 ~ R15
- Base address of I/O Memory maps : SCREEN, KBD
- Nand2Tetris Part2 : SP, LCL, ARG, THIS, THAT

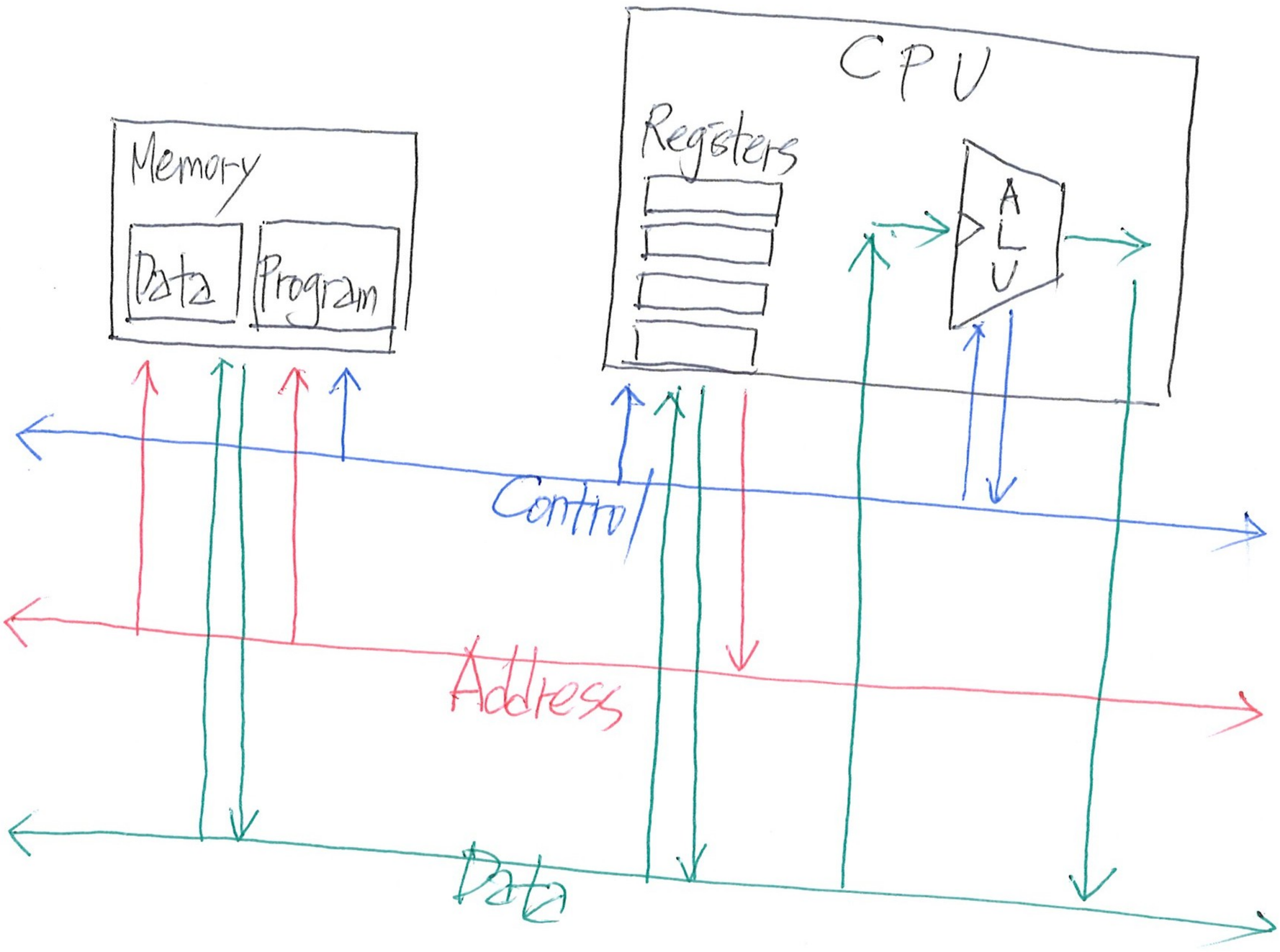
Part 5 Computer Architecture



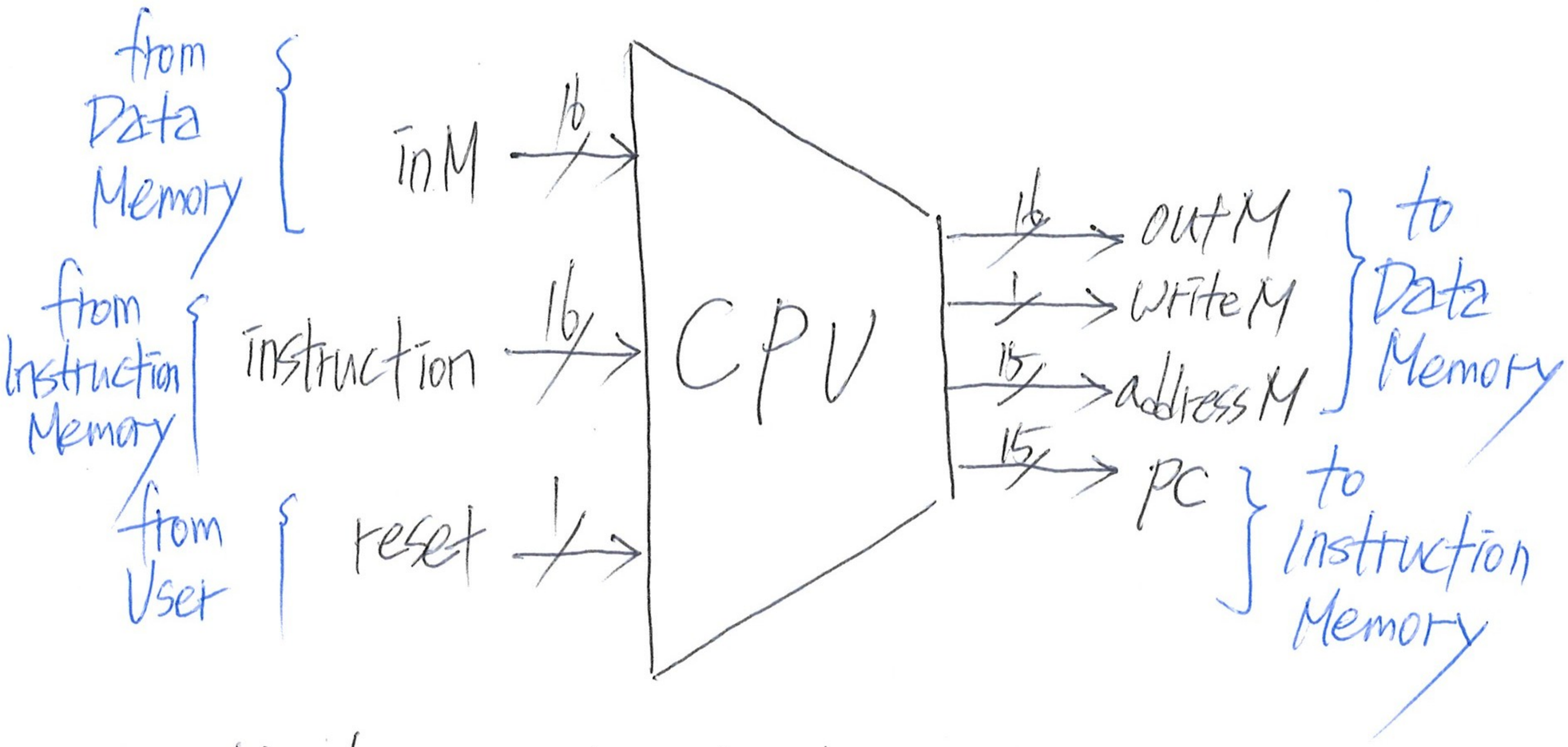
- Hack Computer implementation



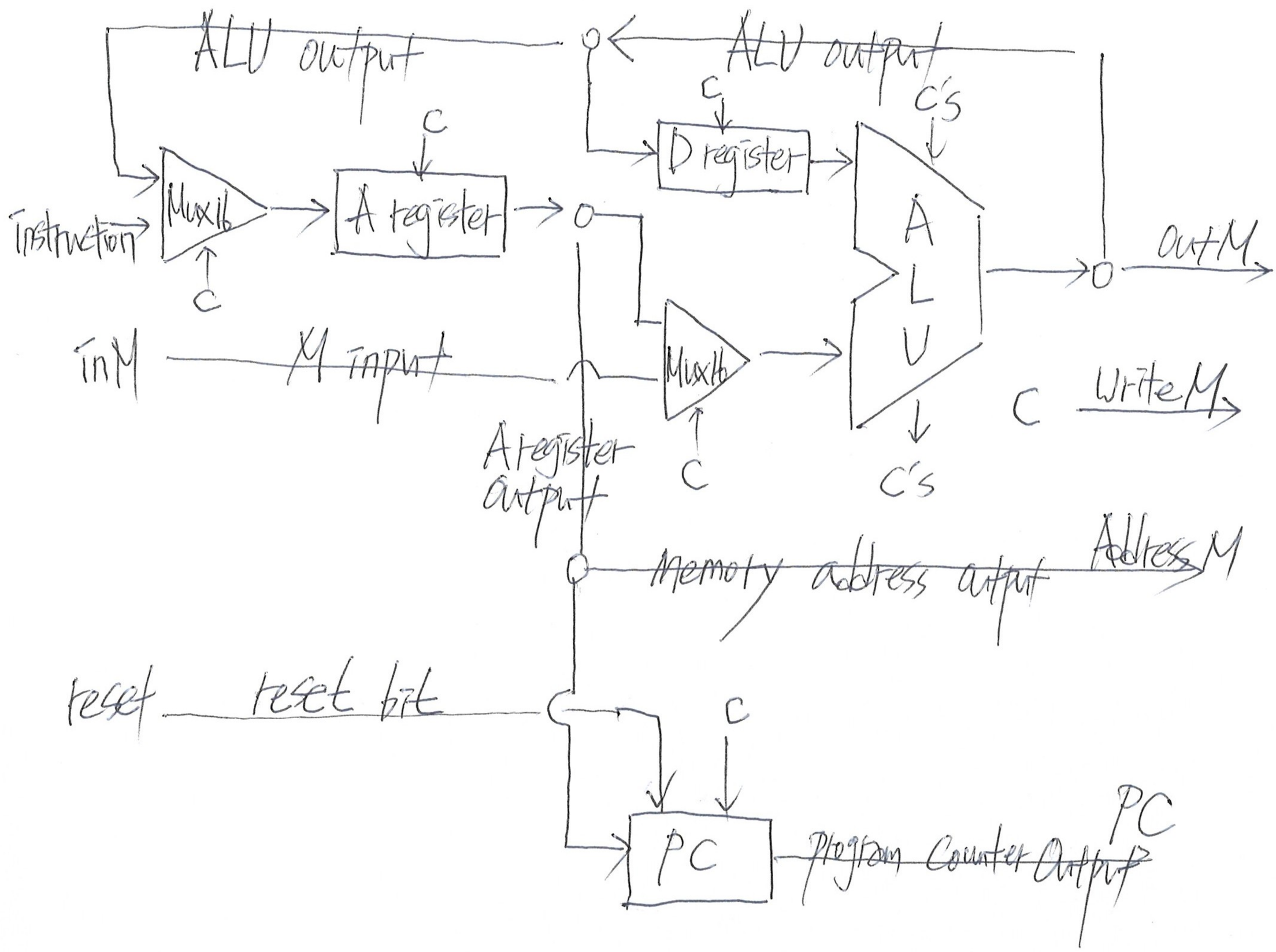
- Information Flows



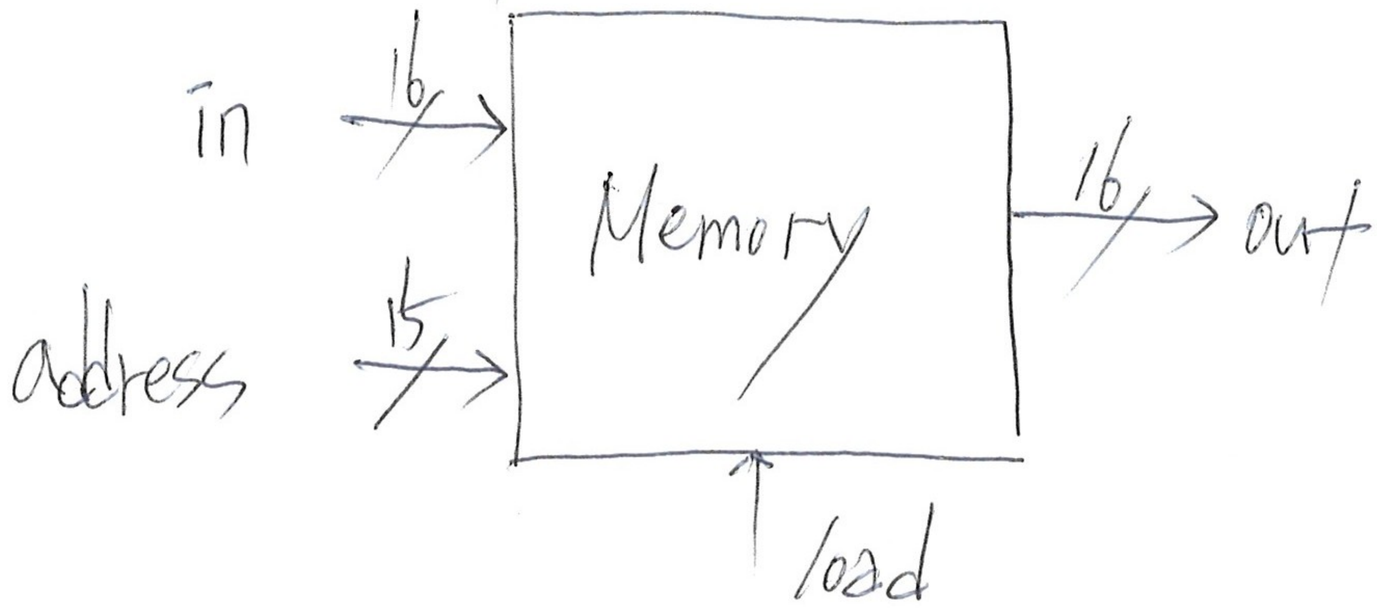
- Hack CPU Interface



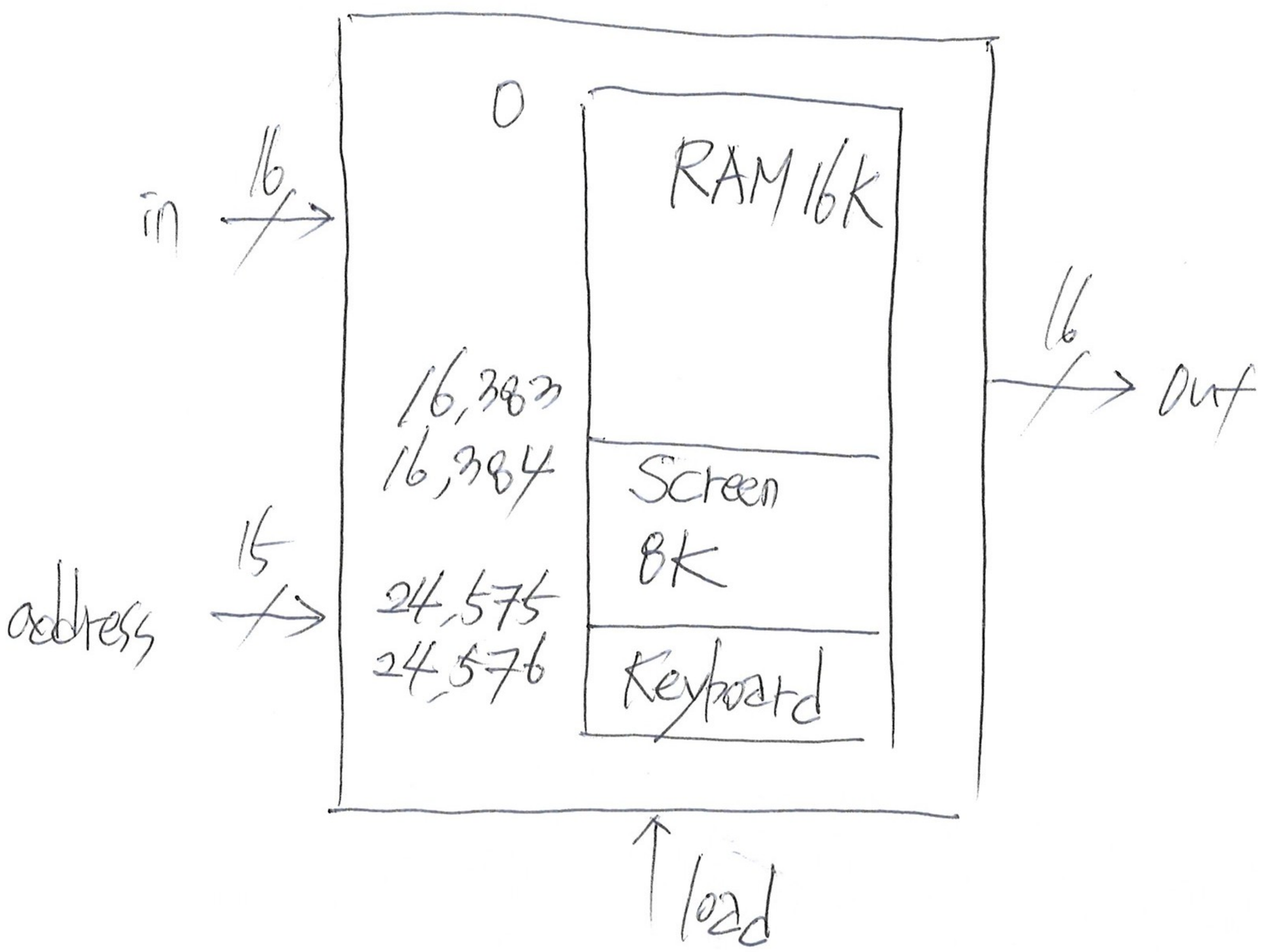
- Hack CPU Implementation



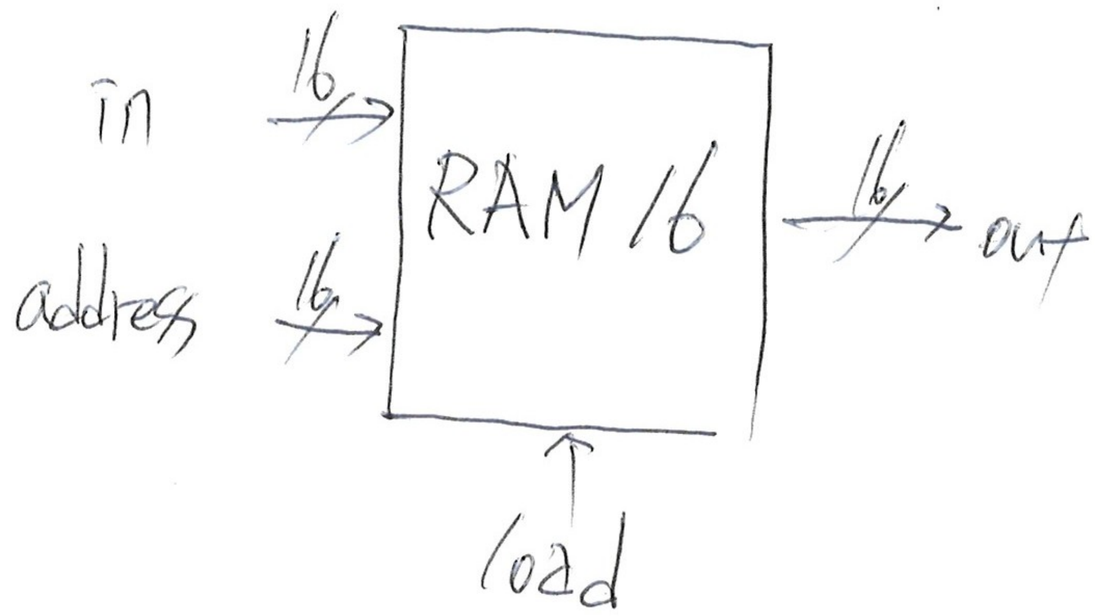
- Hack Memory Interface



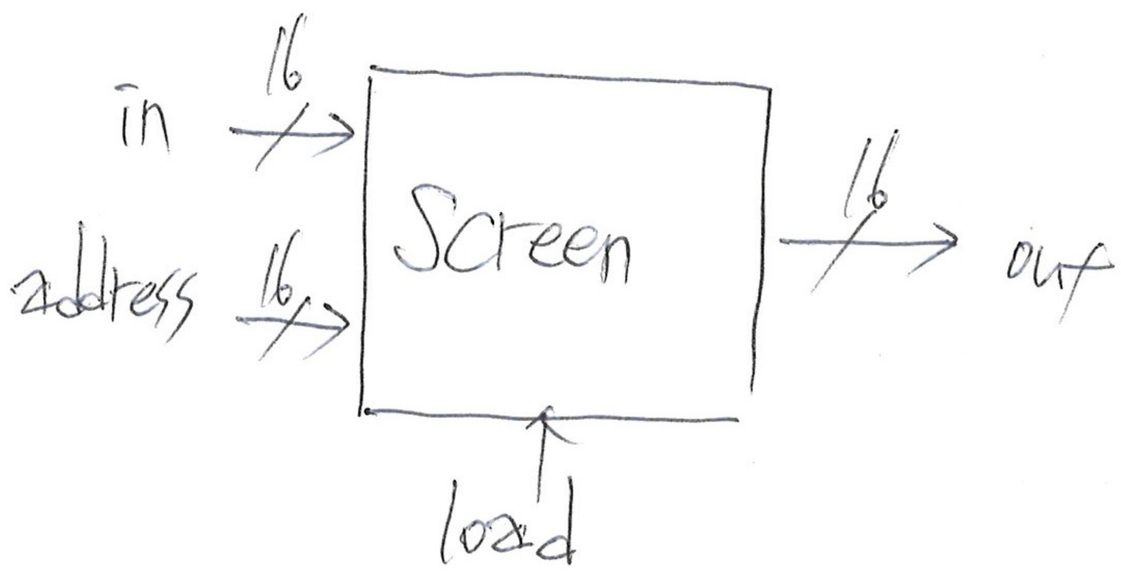
- Hack Memory Implementation



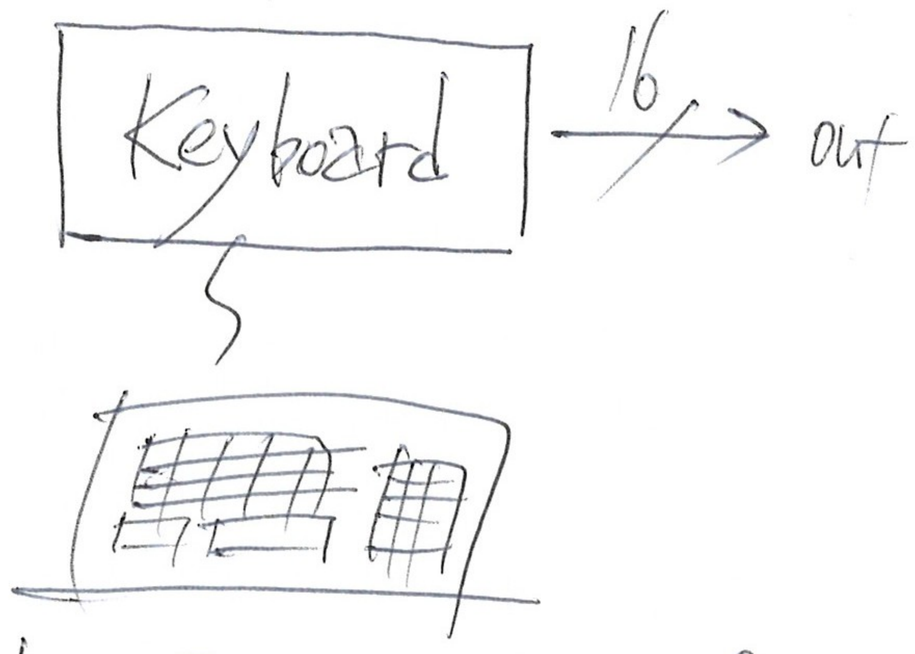
- Hack RAM Interface



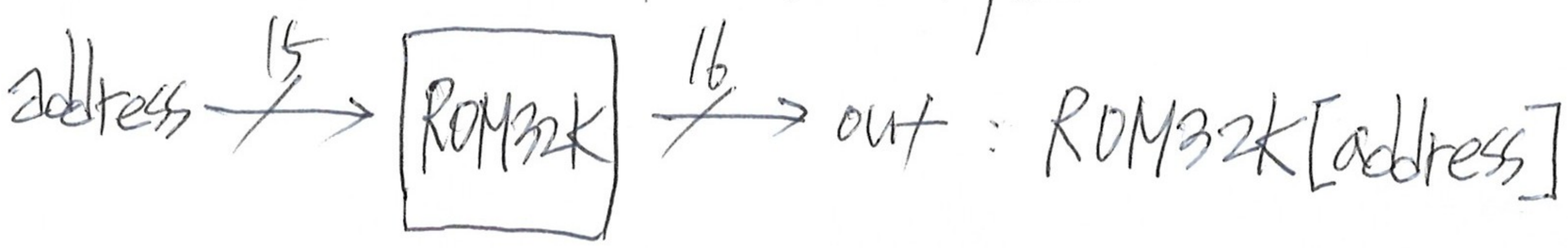
- Hack Screen Interface



- Hack Keyboard Interface



- Hack ROM Interface



Part 6. Assembler

Assembly Language $\xrightarrow{\text{assembler}}$ Machine Language

- Sub Tasks of a Hack Assembler

1. Reading & Parsing Commands (Parser)
2. Converting mnemoics → code (Code)
3. Handling Symbols (Symbol Table)

- Parser: Breaks command into its components.
  Code: Translates each component to its binary format.
  Symbol Table: Add symbols (pre-defined, labels, variables) to the table.

- Overall logic

1. Initialization
   - Of Parser
   - Of Symbol table

2. First Pass

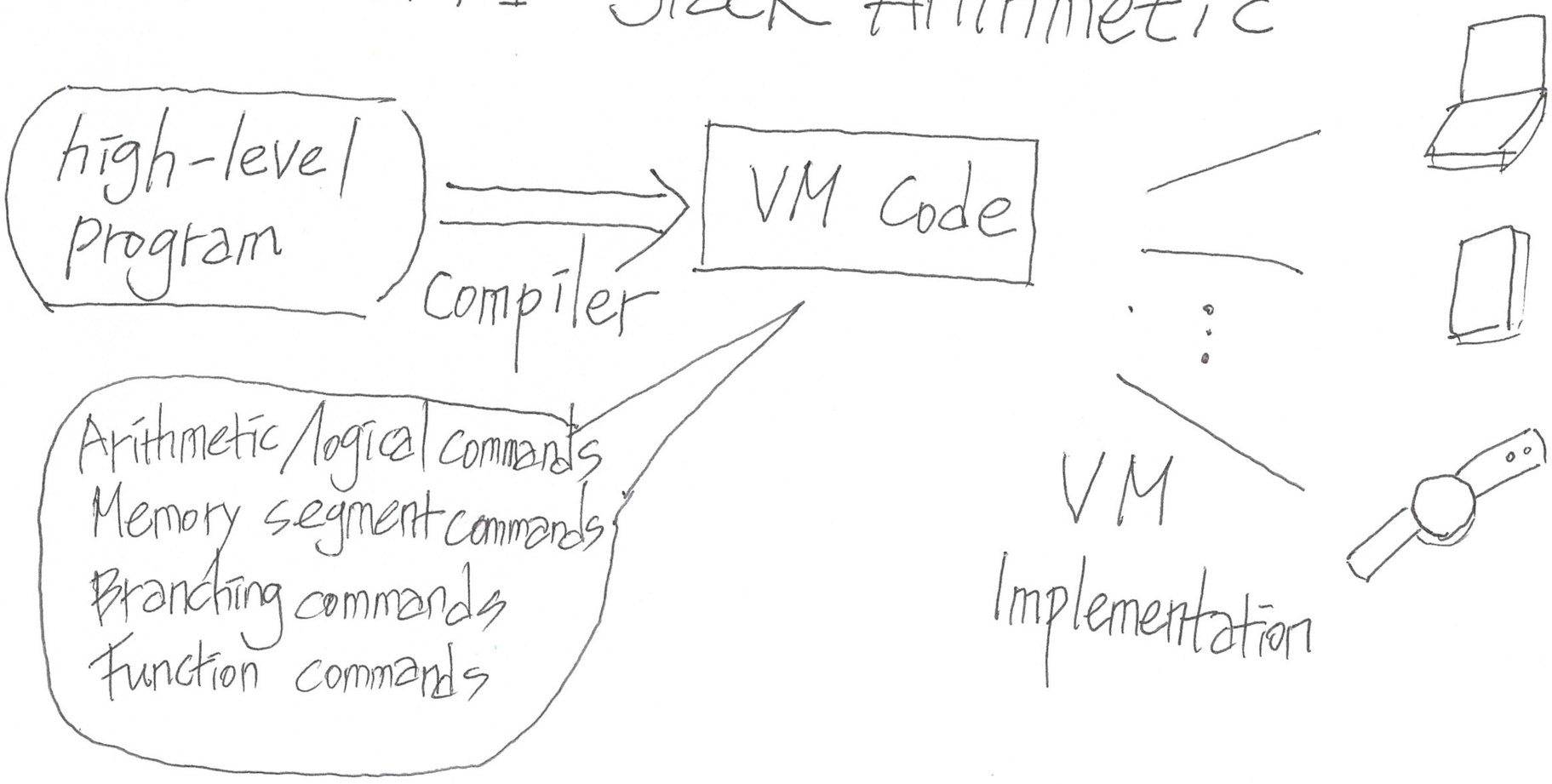
   Read all commands, only paying attention to labels and updating the symbol table.

3. Restart reading and translating commands

4. Main loop

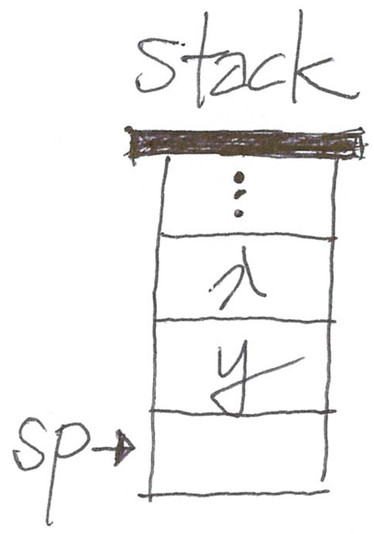
   - Get the next assembly language command & parse it.
   - A instruction : Translate symbols to binary address
   - C instruction : Get code for each part & put them together
   - Output the resulting machine language command.

Part 7. VM I Stack Arithmetic



- Arithmetic/Logical commands

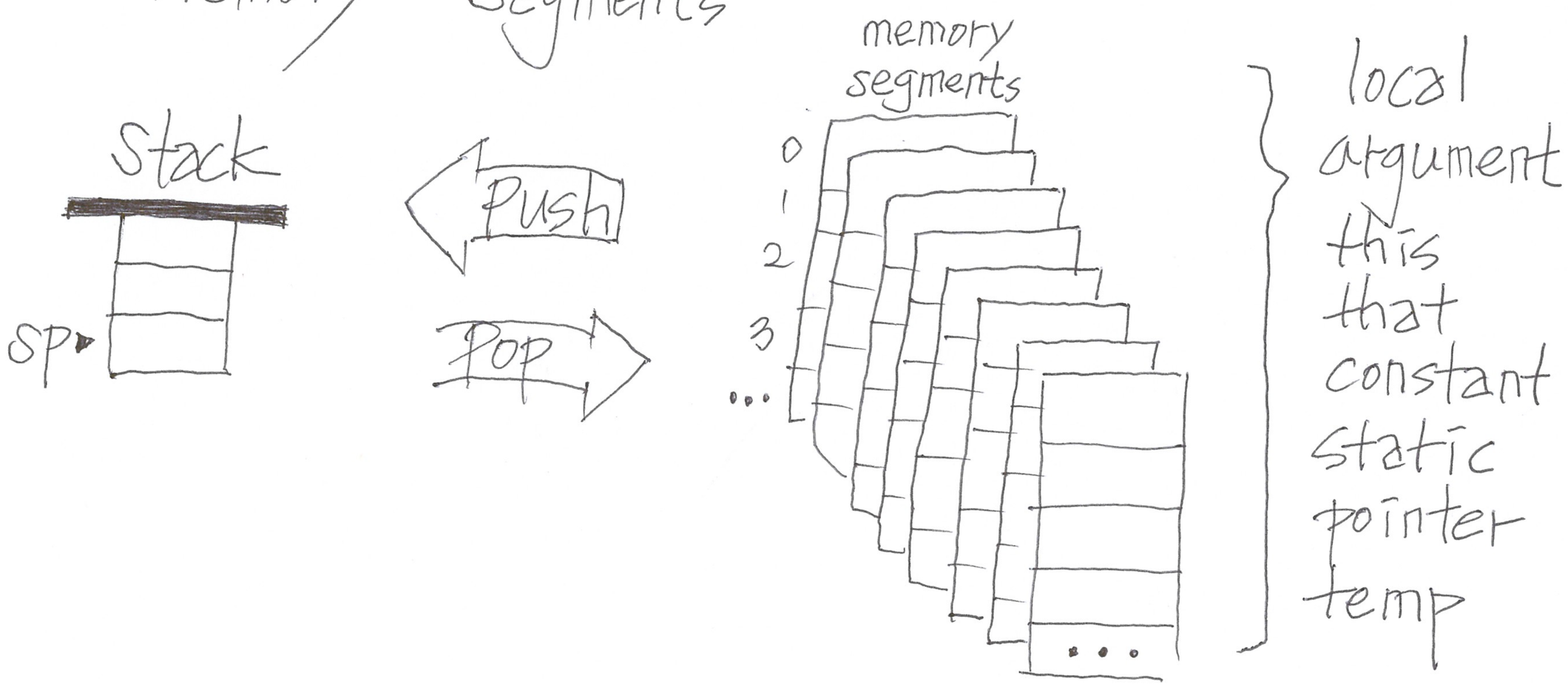
Command	Return Value	Return Value
add	$x+y$	integer
sub	$x-y$	
neg	$-y$	
eq	$x==y$	boolean
gt	$x>y$	
lt	$x<y$	
and	x and y	
or	x or y	
not	not y	



- Memory segment Commands

pop segment i
push segment i

- Memory Segments



Syntax: push/pop segment i

- Pointer manipulation

```
D=*p          // D becomes 23
```

Hack
@p
A=M
D=M

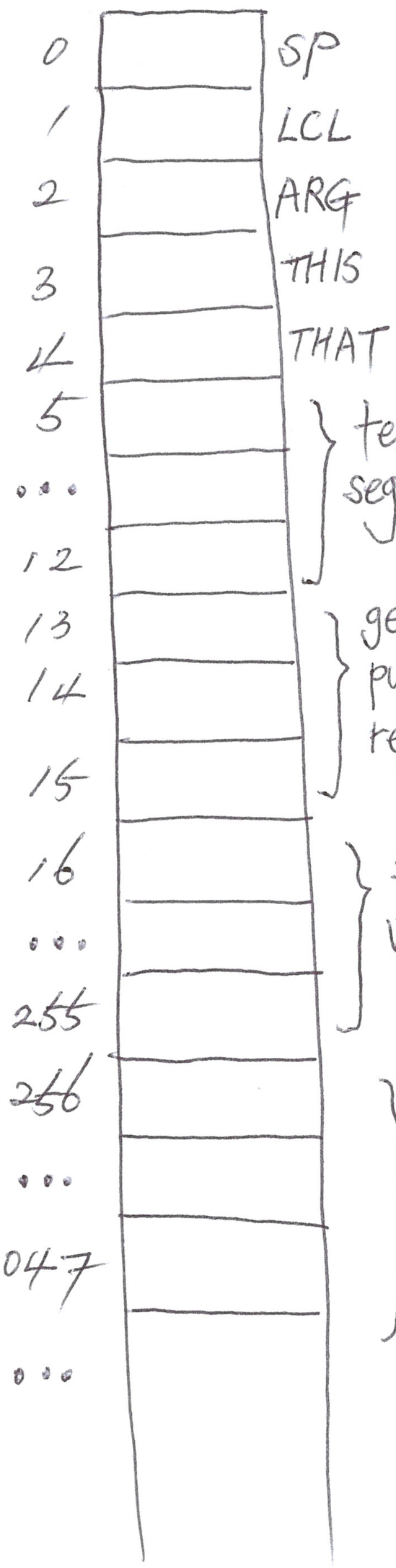
```
p--           //RAM[0] becomes 256
D=*p          // D becomes 19
*q=9          //RAM[1024] becomes 9
q++           // RAM[1] becomes 1025
```

RAM

0	257	p
1	1024	q
2	1765	
	...	
256	19	
257	23	
258	903	
	...	
1024	5	
1025	12	
1026	-3	

```
*p // the memory location that p points at
*q // the memory location that q points at
p++ // increment    p=p+1
q-- // decrement    q=q-1
```

- Standard VM mapping on the Hack platform

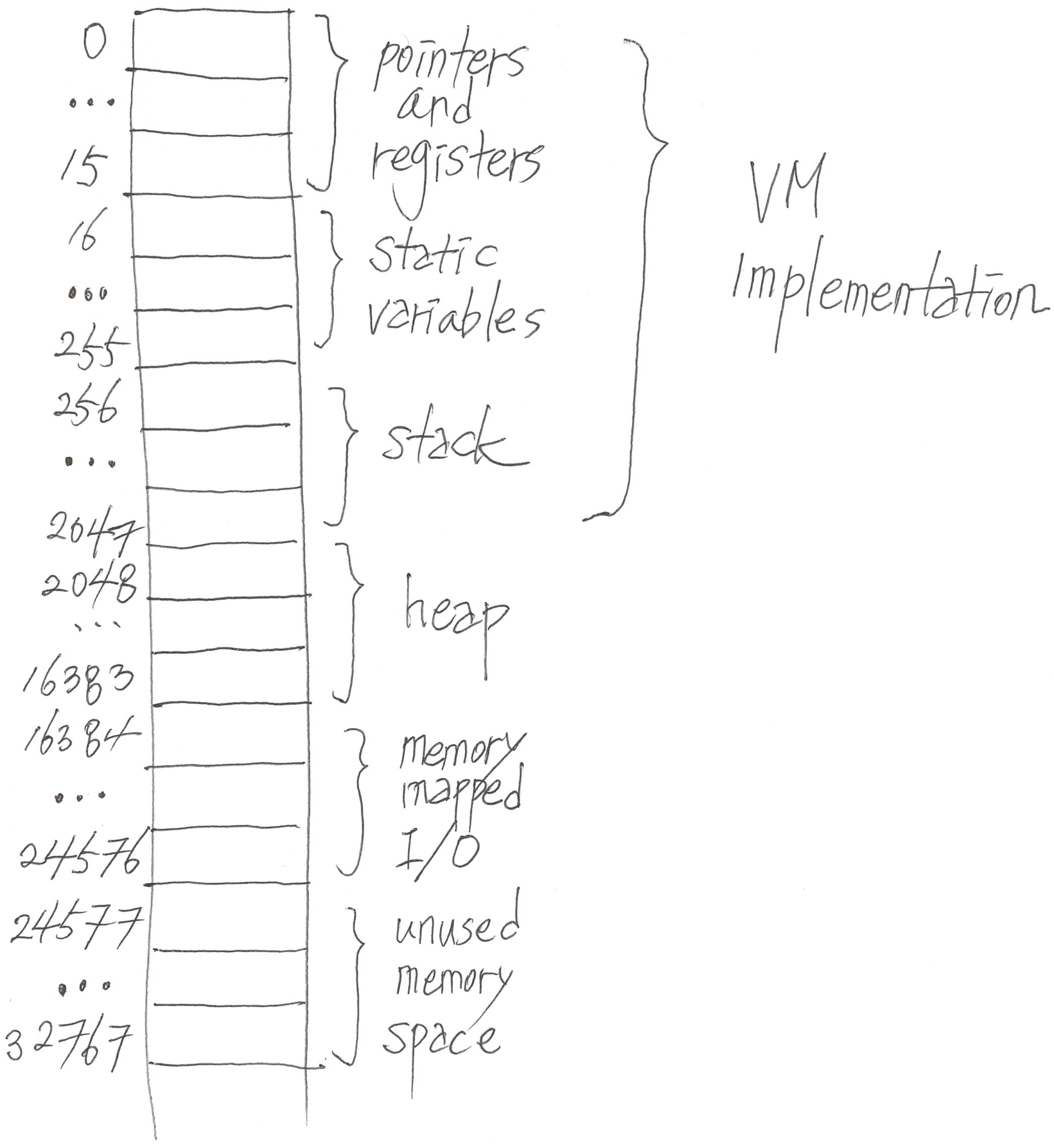


SP	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value
LCL, ARG, THIS, THAT	These predefined symbol point, respectively, to the base addresses within the host RAM of the virtual segments local, argument, this, and that of the currently running VM function
R13–R15	These predefined symbols can be used for any purpose
Xxx.i symbols	The static segment is implemented as follows: each static variable i in file Xxx.vm is translated into the assembly symbol Xxx.i In the subsequent assembly process, these symbolic variables will be allocated to the RAM by the Hack Assembler.
functionName$label	Let foo be a function within a VM file Xxx. Each label bar command within foo should generate and insert into the assembly code stream a symbol Xxx.foo$bar
functionName	Each function foo command within a VM file Xxx should generate and insert into the asembly code stream a symbol Xxx.foo that labels the entry point to the function's code. In the subsequent assembly process, the asembler will translate this symbol into the physical memory address where the function code starts
functionName$ret.i	Let foo be a function within a VM file Xxx.

(...)

- Standard mapping of the VM on the Hack platform

Hack RAM



Part 8. VM II. Program Control

• Branching Commands

High-level program

```
// Returns x*y
int mult(int x, int y) {
    int sum = 0;
    int n = 1;
    // sum = sum+x, y times
    while !(n > y) {
        sum += x;
        n++;
    }
    return sum;
}
```

Pseudo VM code

```
function mult(x,y)
    push 0
    pop sum
    push 1
    pop n
label LOOP
    push n
    push y
    gt
    if-goto ENDLOOP
    push sum
    push x
    add
    pop sum
    push n
    push 1
    add
    pop n
    goto LOOP
label ENDLOOP
    push sum
    return
```

label *label* : label declaration command
goto *label* : jump to execute the command just after *label*
if-goto *label* : if cond jump to execute the command just after *label*

- Function Commands : function

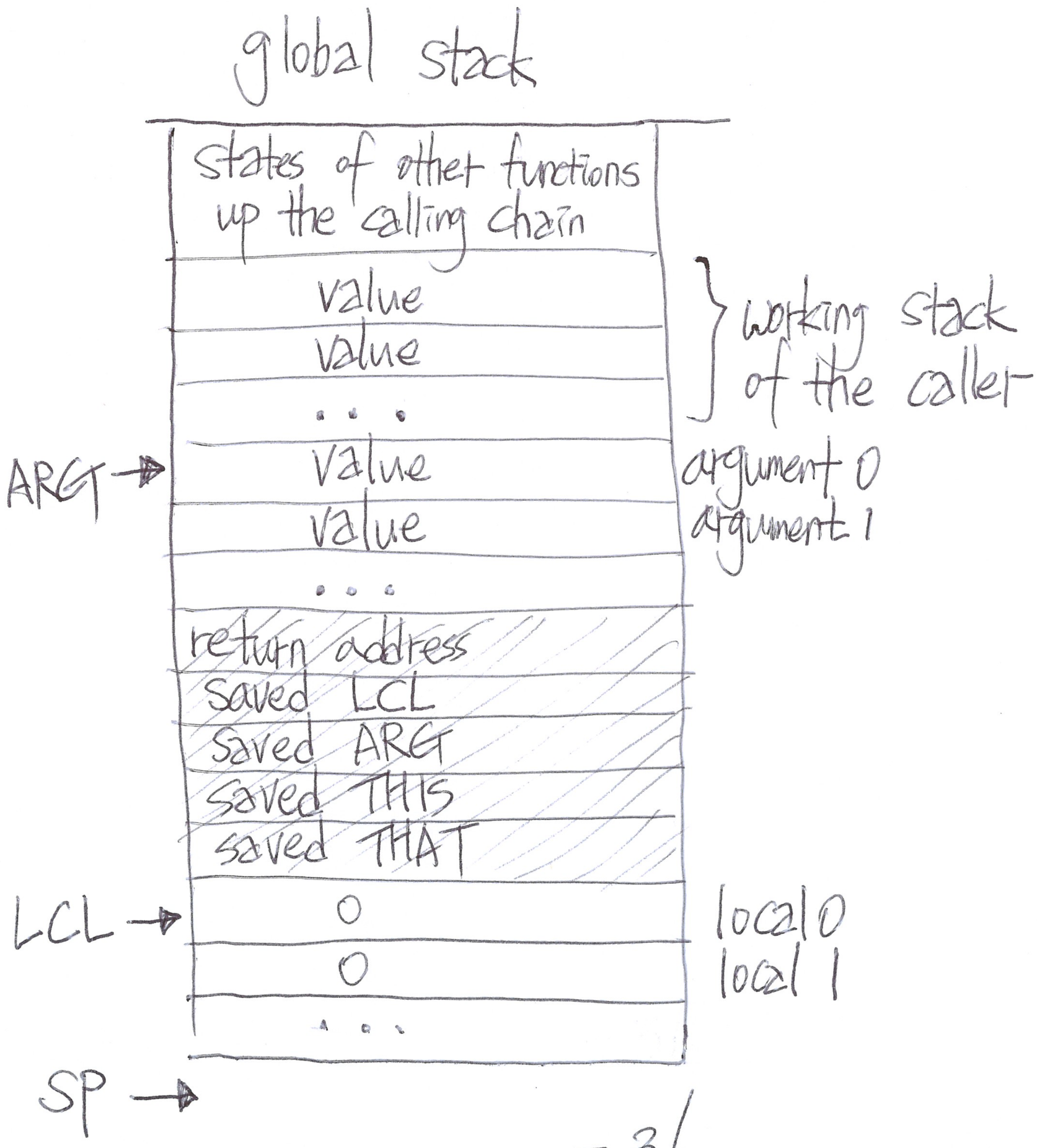
VM Command: function functionName nVars

translate

(here starts a function that has nVars local variables)

Assembly code

```
(functionName)
    repeat nVars times:   // nVars = number of local variables
    push 0                // initializes the local variables to 0
```



- Function Commands : call

VM Command : call functionName nArgs

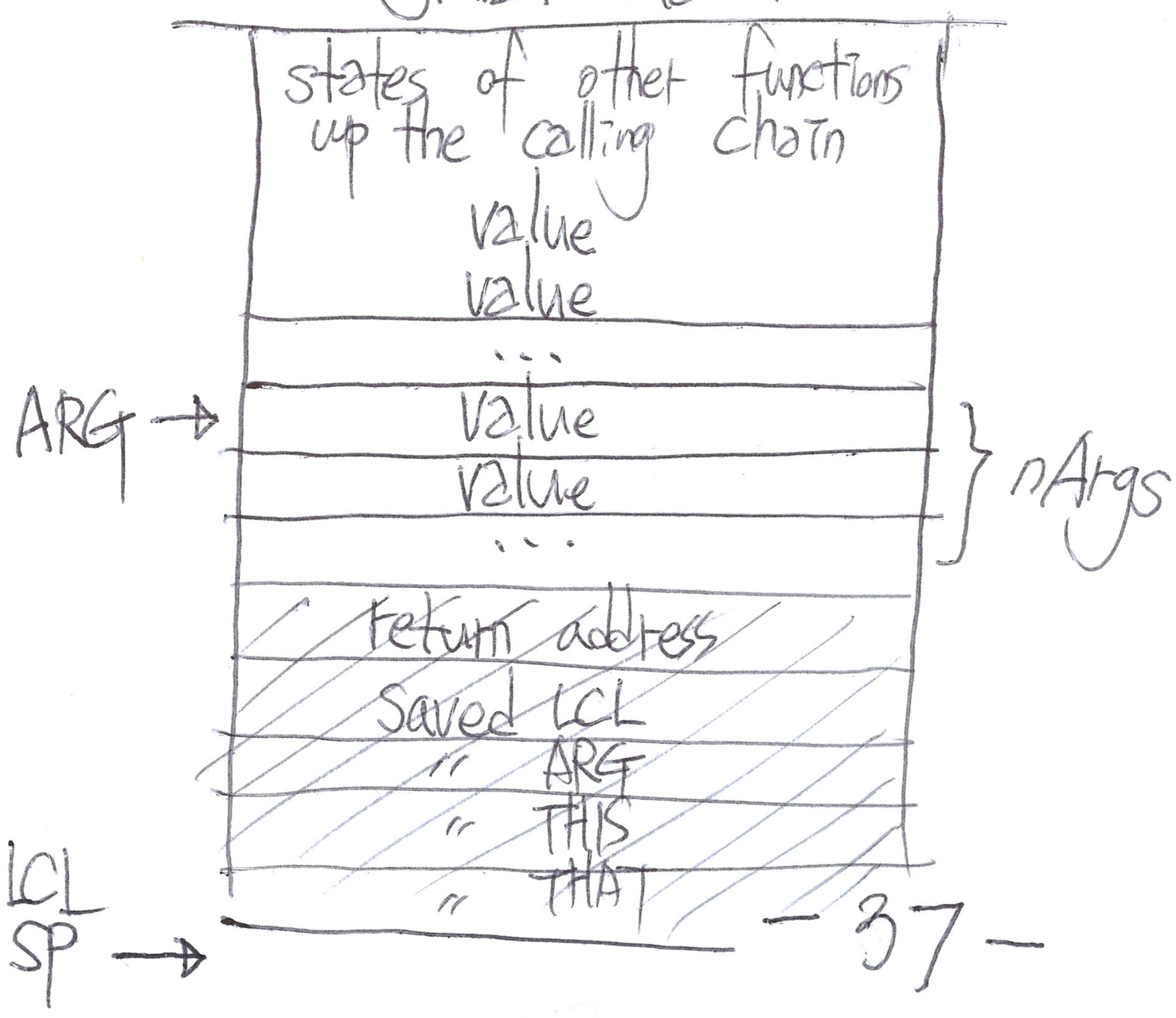
(calls the function, informing that nArgs arguments have been pushed onto the stack)

translate

Assembly code

```
push returnAddress
push LCL                    // Saves LCL of the caller
push ARG                       ARG
push THIS                      THIS
push THAT                      THAT
ARG = SP-5-nArgs            // Repositions ARG
LCL = SP                    // Repositions LCL
goto functionName
(returnAddress)
```

global stack



- Function Command: return

VM Command: return

Assembly Code

```
endFrame = LCL
retAddr = *(endFrame - 5)
*ARG = pop()
SP = ARG + 1
THAT = *(endFrame - 1)
THIS = *(endFrame - 2)
ARG = *(endFrame - 3)
LCL = *(endFrame - 4)
goto retAddr
```

- Bootstrap Code

VM Code

```
SP = 256
call Sys.init
```

Assembly Code

```
@256
D=A
@SP
M=D
```

"SP=256"

1. Arithmetic Commands

1) Binary

add | sub | and | or

Assembly

```
@SP
AM=M-1      // SP--; A=SP; RAM[A]=y
D=M         // D=y
A=A-1       // A=SP-1
M={ D+M     // add
  { M-D     // sub
  { D&M     // and
  { D|M     // or
```

2) Unary

neg | not

Assembly

```
@SP
A=M-1
M={ -M      // neg
  { !M      // not
```

3) Compare

eq | gt | lt

Assembly

```
@SP
AM=M-1
D=M
A=A-1
D=M-D
@truelabel
D;{ JEQ     // eq
  { JGT     // gt
  { JLT     // lt

@SP
A=M-1
M=0         // false
@endlabel
0;JMP
(truelabel)
@SP
A=M-1
M=-1        // true
(endlabel)
```