



# 計算機結構

( 習題：Nand2tetris 硬體部分 )

陳鍾誠

2017 年 9 月 2 日

# Chapter 0 – Background

- Digital Logic
- Hardware Description Language

# Digital Logic

# Truth Table

$x$	$y$	$z$	$f(x, y, z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

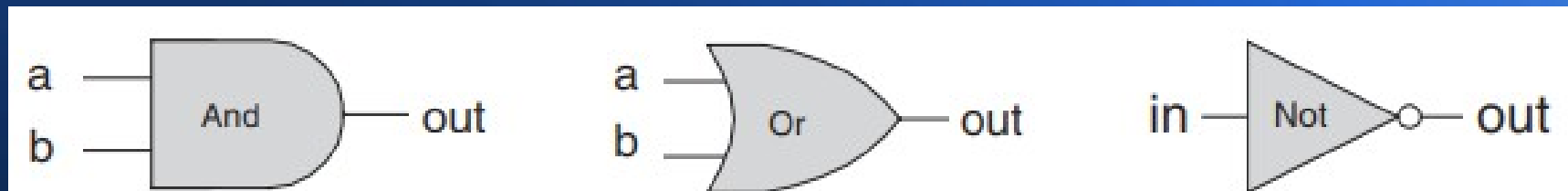
**Figure 1.1** Truth table representation of a Boolean function (example).

# Logic Expression

Function	$x$	0	0	1	1
	$y$	0	1	0	1
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
$x$ And Not $y$	$x \cdot \bar{y}$	0	0	1	0
$x$	$x$	0	0	1	1
Not $x$ And $y$	$\bar{x} \cdot y$	0	1	0	0
$y$	$y$	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x + y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not $y$	$\bar{y}$	1	0	1	0
If $y$ then $x$	$x + \bar{y}$	1	0	1	1
Not $x$	$\bar{x}$	1	1	0	0
If $x$ then $y$	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

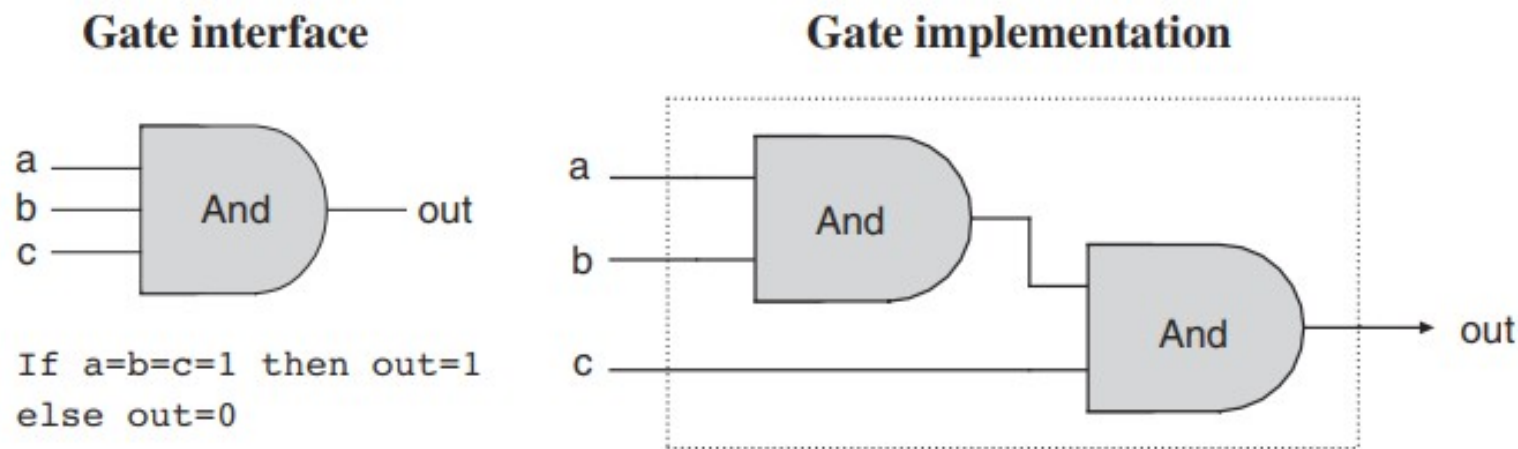
**Figure 1.2** All the Boolean functions of two variables.

# Gates



**Figure 1.3** Standard symbolic notation of some elementary logic gates.

# Composite Gates



**Figure 1.4** Composite implementation of a three-way And gate. The rectangle on the right defines the conceptual boundaries of the gate interface.

# HDL

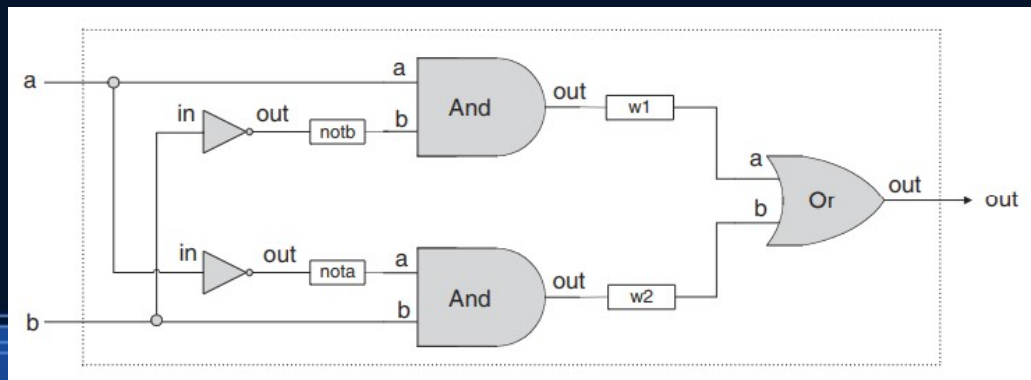
## (Hardware Description Language)

```
/** Checks if two 3-bit input buses are equal */
CHIP EQ3 {
    IN  a[3], b[3];
    OUT out; // True iff a=b
    PARTS:
        Xor(a=a[0], b=b[0], out=c0);
        Xor(a=a[1], b=b[1], out=c1);
        Xor(a=a[2], b=b[2], out=c2);
        Or(a=c0, b=c1, out=c01);
        Or(a=c01, b=c2, out=neq);
        Not(in=neq, out=out);
}
```

**Figure A.1** HDL program example.



# HDL



## *HDL program* (Xor.hdl)

```

/* Xor (exclusive or) gate:
   If a<>b out=1 else out=0. */
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
    Not(in=a, out=nota);
    Not(in=b, out=notb);
    And(a=a, b=notb, out=w1);
    And(a=nota, b=b, out=w2);
    Or(a=w1, b=w2, out=out);
}

```

## *Test script* (Xor.tst)

```

load Xor.hdl,
output-list a, b, out;
set a 0, set b 0,
eval, output;
set a 0, set b 1,
eval, output;
set a 1, set b 0,
eval, output;
set a 1, set b 1,
eval, output;

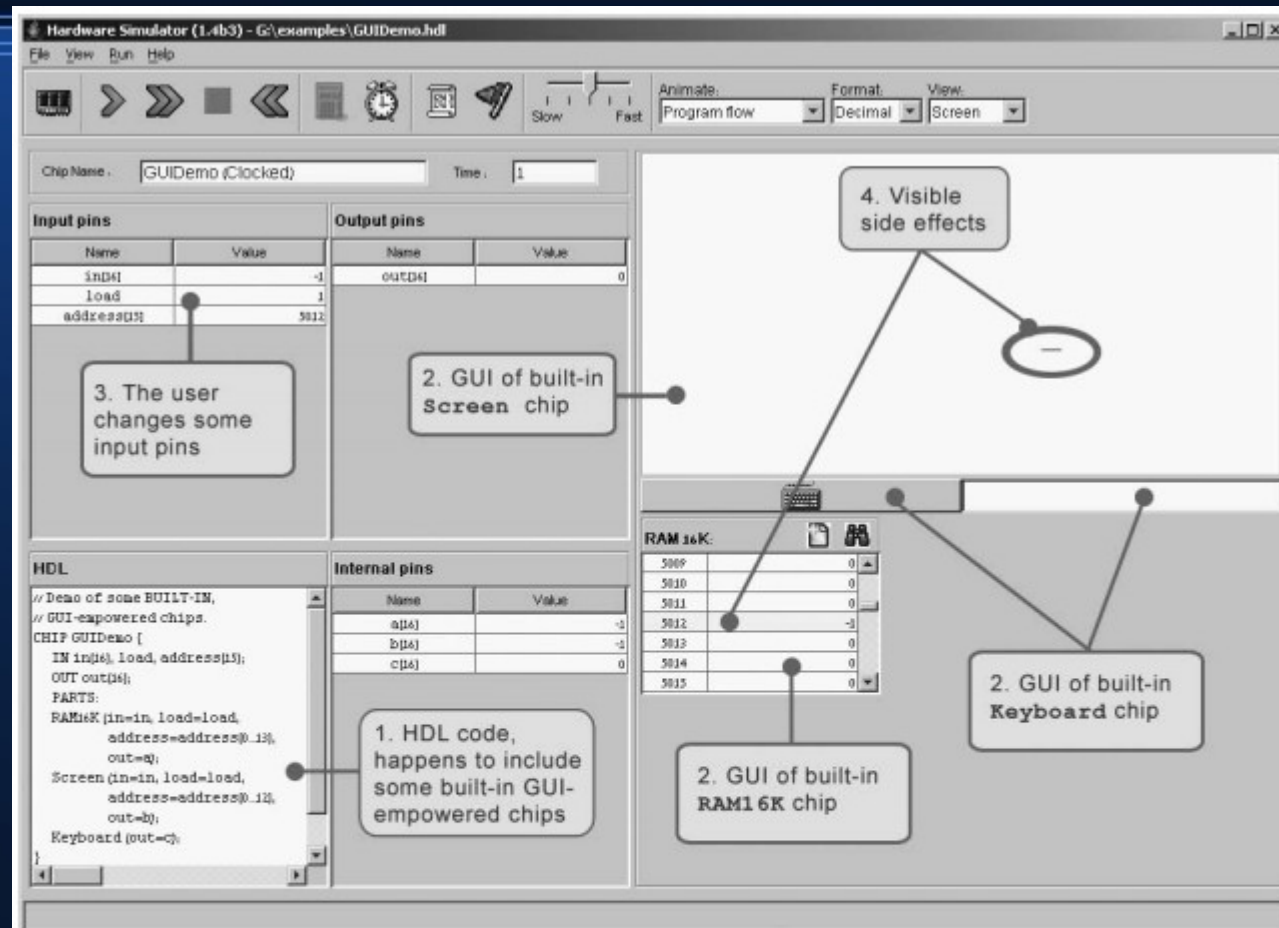
```

## *Output file* (Xor.out)

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

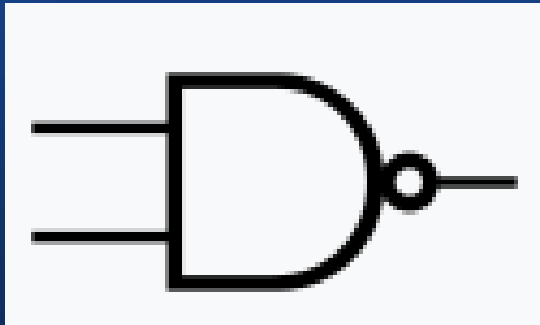
**Figure 1.6** HDL implementation of a Xor gate.

# Hardware Simulator



**Figure A.5** GUI-empowered chips. Since the loaded HDL program uses GUI-empowered chips as internal parts (step 1), the simulator draws their respective GUI images (step 2). When the user changes the values of the chip input pins (step 3), the simulator reflects these changes in the respective GUIs (step 4). The circled horizontal line is the visual side effect of storing -1 in memory location 5012. Since the 16-bit 2's complement binary code of -1 is 1111111111111111, the computer draws 16 pixels starting at the 320th column of row 156, which happen to be the screen coordinates associated with address 5012 of the screen memory map (the exact memory-to-screen mapping is given in chapter 4).

# NAND Gate



$$Y = \overline{A B}$$

<i>a</i>	<i>b</i>	Nand( <i>a</i> , <i>b</i> )
0	0	1
0	1	1
1	0	1
1	1	0

**Chip name:** Nand

**Inputs:** a, b

**Outputs:** out

**Function:** If a=b=1 then out=0 else out=1

**Comment:** This gate is considered primitive and thus there is no need to implement it.

# Karnaugh map (1)

2 变量卡诺图：

A \ B	0	1	
	$\bar{B}$	$B$	
0	$m_0$ $\bar{A}\bar{B}$	$m_1$ $\bar{A}B$	$\bar{A}$ ----- $A$
1	$m_2$ $A\bar{B}$	$m_3$ $AB$	

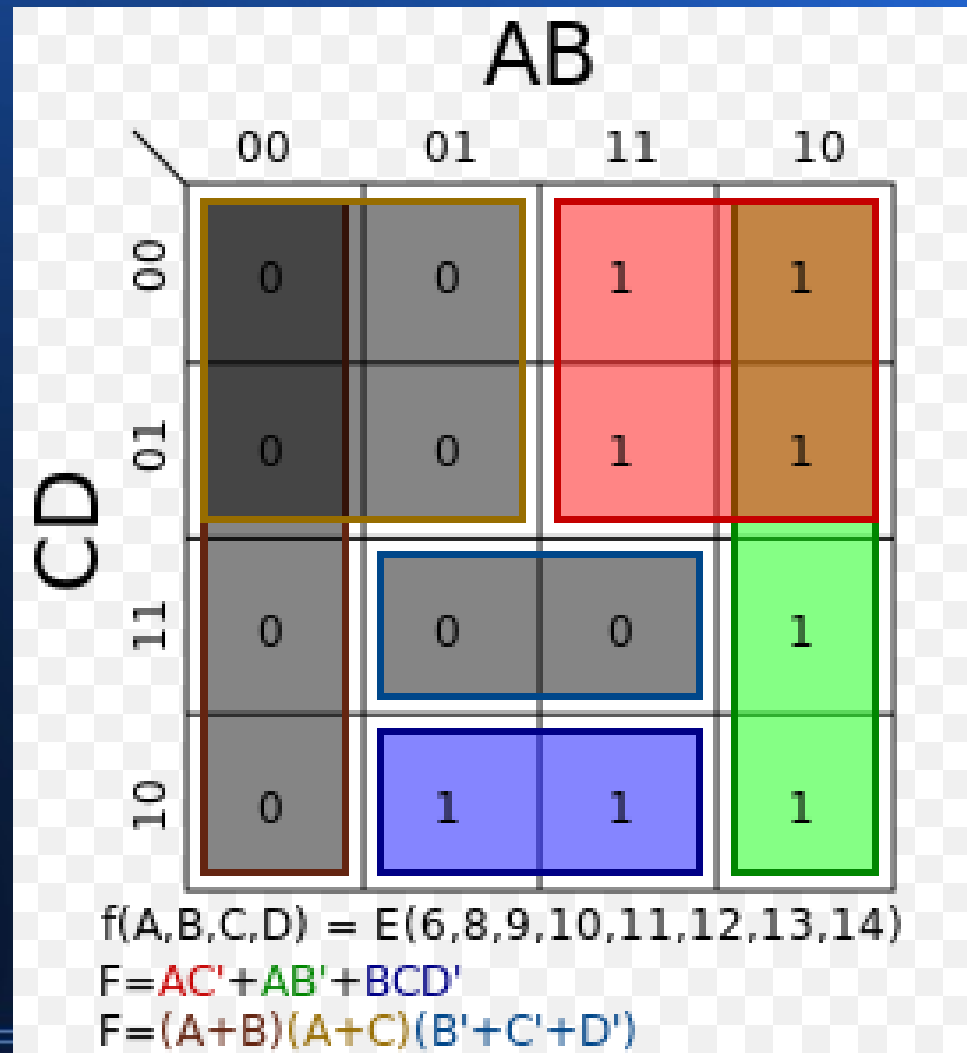
3 变量卡诺图：

A \ BC	00	01	11	10	
	$\bar{B}\bar{C}$	$\bar{B}C$	$BC$	$B\bar{C}$	
0	$m_0$ $\bar{A}\bar{B}\bar{C}$	$m_1$ $\bar{A}\bar{B}C$	$m_3$ $\bar{A}BC$	$m_2$ $\bar{A}B\bar{C}$	$\bar{A}$ ----- $A$
1	$m_4$ $A\bar{B}\bar{C}$	$m_5$ $A\bar{B}C$	$m_7$ $ABC$	$m_6$ $AB\bar{C}$	

4 变量卡诺图：

AB \ CD	00	01	11	10
	$\bar{C}\bar{D}$	$\bar{C}D$	$CD$	$C\bar{D}$
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

# Karnaugh map (2)



# Karnaugh map (3)

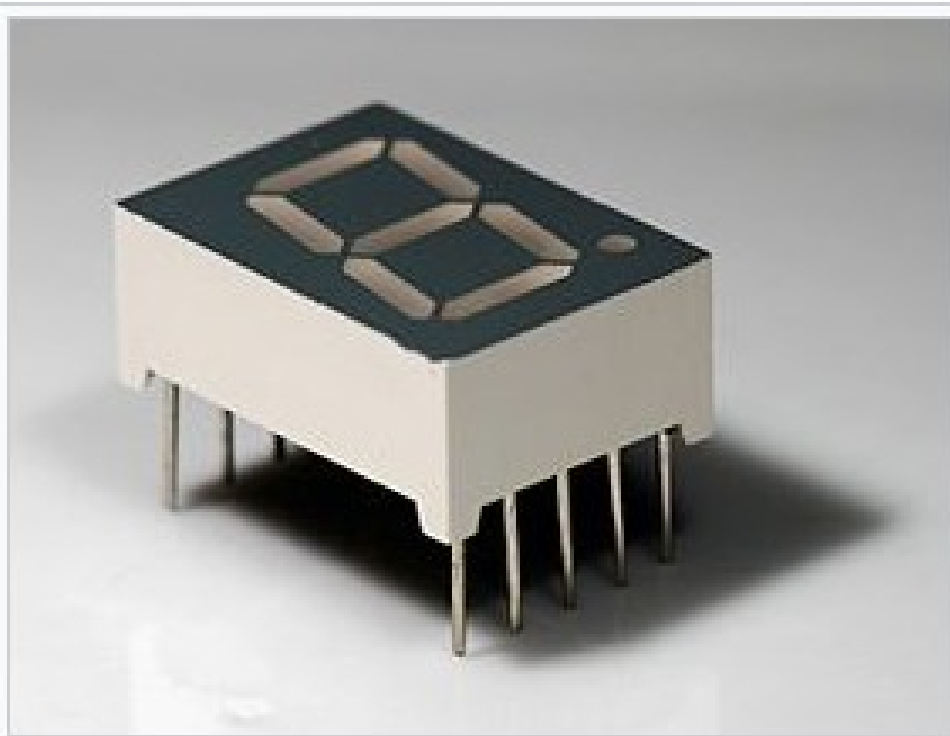
例 用卡诺图法将函数化简为最简与非式。

$$F = \sum m^4(0,1,3,5,7,14,15)$$

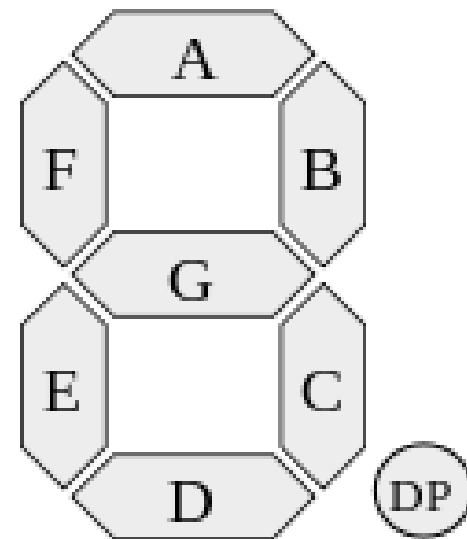
$AB \backslash CD$		$AB$			
		$00$	$01$	$11$	$10$
$00$	$1$	$0$	$0$	$0$	
$01$	$1$	$1$	$0$	$0$	
$11$	$1$	$1$	$1$	$0$	
$10$	$0$	$0$	$1$	$0$	

$$\begin{aligned} F &= \overline{A}\overline{B}\overline{C} + ABC + \overline{A}D \\ &= \overline{\overline{A}\overline{B}\overline{C}} + ABC + \overline{A}D \\ &= \overline{\overline{A}\overline{B}\overline{C}} \bullet \overline{\overline{A}\overline{B}\overline{C}} \bullet \overline{\overline{A}D} \end{aligned}$$

# Exercise (1)

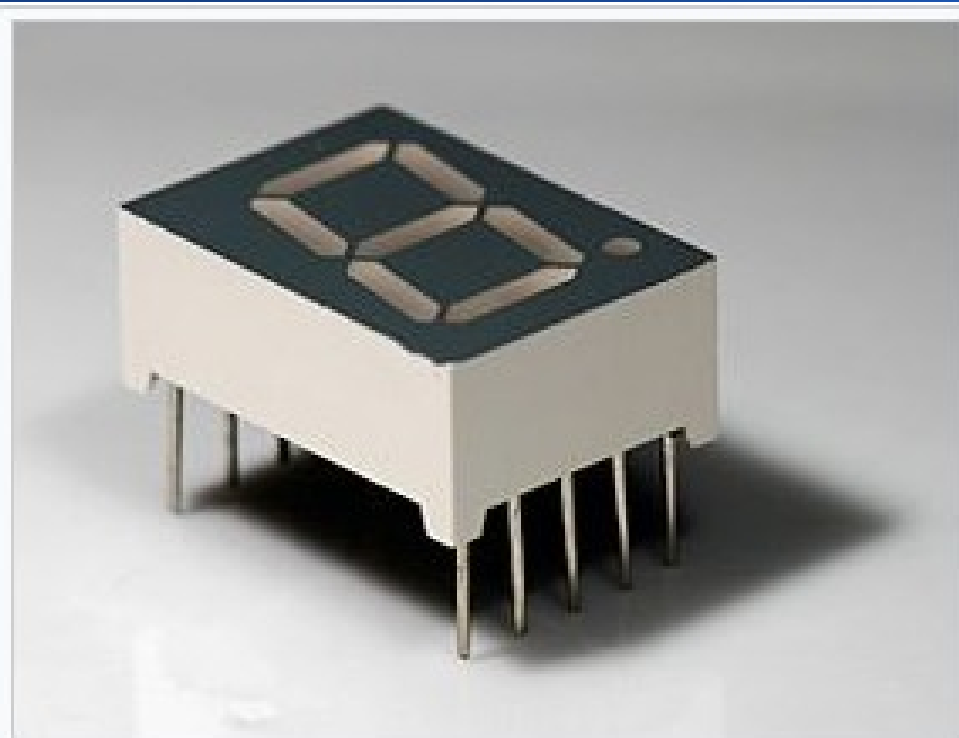


A typical 7-segment LED display component, with decimal point

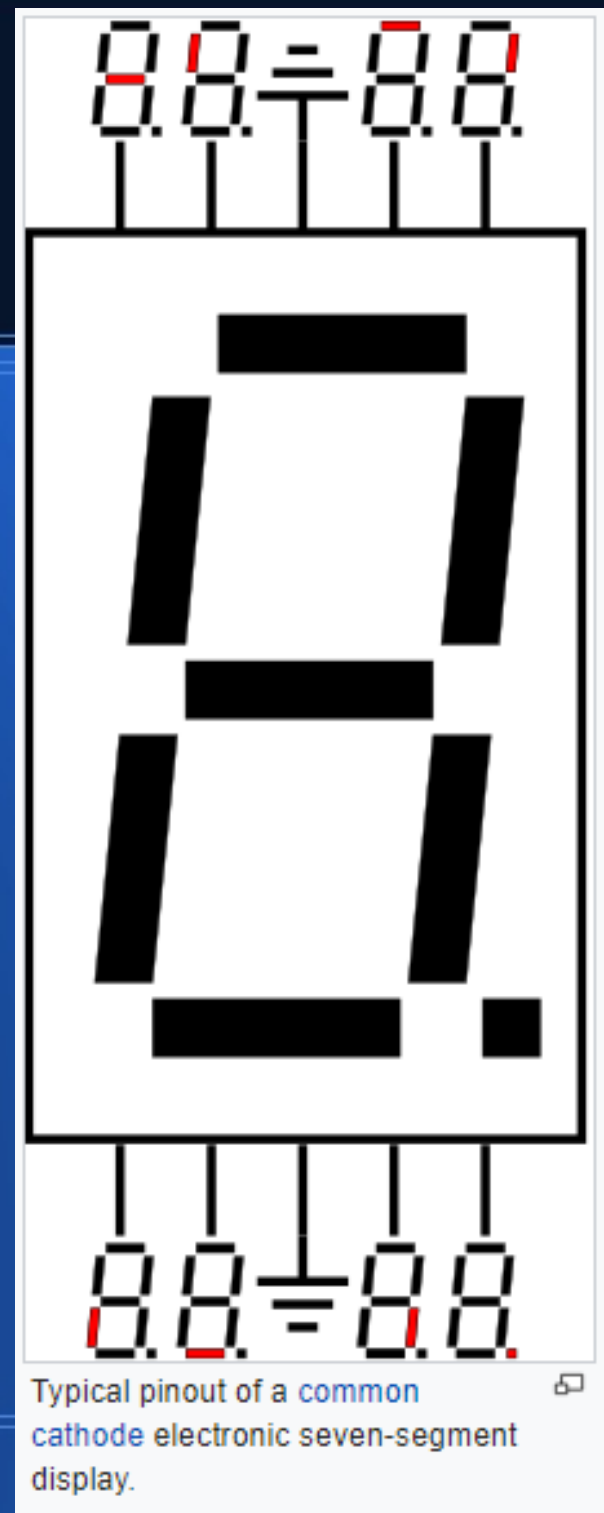


The individual segments of a seven-segment display

# Exercise (2)



A typical 7-segment LED display component, with decimal point



Typical pinout of a common cathode electronic seven-segment display.



# Exercise (3)

Digit	Display	gfedcba	abcdefg	a	b	c	d	e	f	g
0	0	0x3F	0x7E	on	on	on	on	on	on	off
1	1	0x06	0x30	off	on	on	off	off	off	off
2	2	0x5B	0x6D	on	on	off	on	on	off	on
3	3	0x4F	0x79	on	on	on	on	off	off	on
4	4	0x66	0x33	off	on	on	off	off	on	on
5	5	0x6D	0x5B	on	off	on	on	off	on	on
6	6	0x7D	0x5F	on	off	on	on	on	on	on
7	7	0x07	0x70	on	on	on	off	off	off	off
8	8	0x7F	0x7F	on	on	on	on	on	on	on
9	9	0x6F	0x7B	on	on	on	on	off	on	on
A	A	0x77	0x77	on	on	on	off	on	on	on
b	b	0x7C	0x1F	off	off	on	on	on	on	on
C	C	0x39	0x4E	on	off	off	on	on	on	off
d	d	0x5E	0x3D	off	on	on	on	on	off	on
E	E	0x79	0x4F	on	off	off	on	on	on	on
F	F	0x71	0x47	on	off	off	off	on	on	on

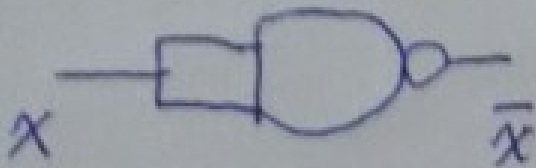
# Chapter 1 – Boolean Logic

# Project 1: Elementary Logic Gates

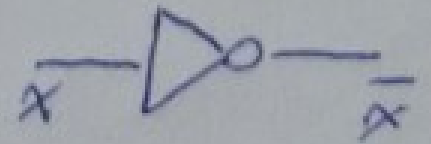
Chip (HDL)	Description
Nand	Nand gate (primitive)
Not	Not gate
And	And gate
Or	Or gate
Xor	Xor gate
Mux	Mux gate
DMux	DMux gate
Not16	16-bit Not
And16	16-bit And
Or16	16-bit Or
Mux16	16-bit multiplexor
Or8Way	Or(in0,in1,...,in7)
Mux4Way16	16-bit/4-way mux
Mux8Way16	16-bit/8-way mux
DMux4Way	4-way demultiplexor
DMux8Way	8-way demultiplexor

# Not

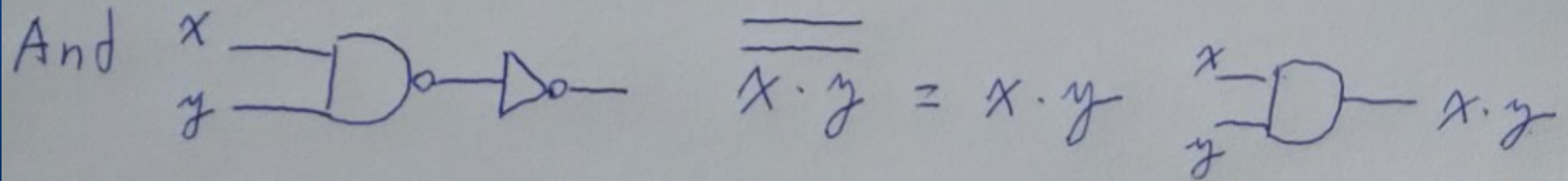
Not



$$\bar{x} = \overline{x \cdot x}$$



# And



**And** The And function returns 1 when both its inputs are 1, and 0 otherwise.

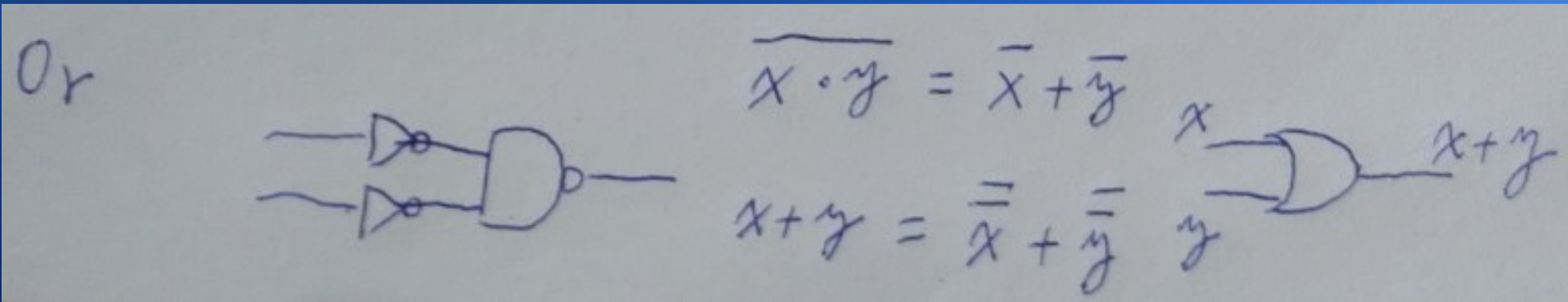
**Chip name:** And

**Inputs:** a, b

**Outputs:** out

**Function:** If  $a=b=1$  then  $out=1$  else  $out=0$ .

# Or



**Or** The Or function returns 1 when at least one of its inputs is 1, and 0 otherwise.

**Chip name:** Or

**Inputs:** a, b

**Outputs:** out

**Function:** If  $a=b=0$  then  $out=0$  else  $out=1$ .

# Xor (1)

**Xor** The Xor function, also known as “exclusive or,” returns 1 when its two inputs have opposing values, and 0 otherwise.

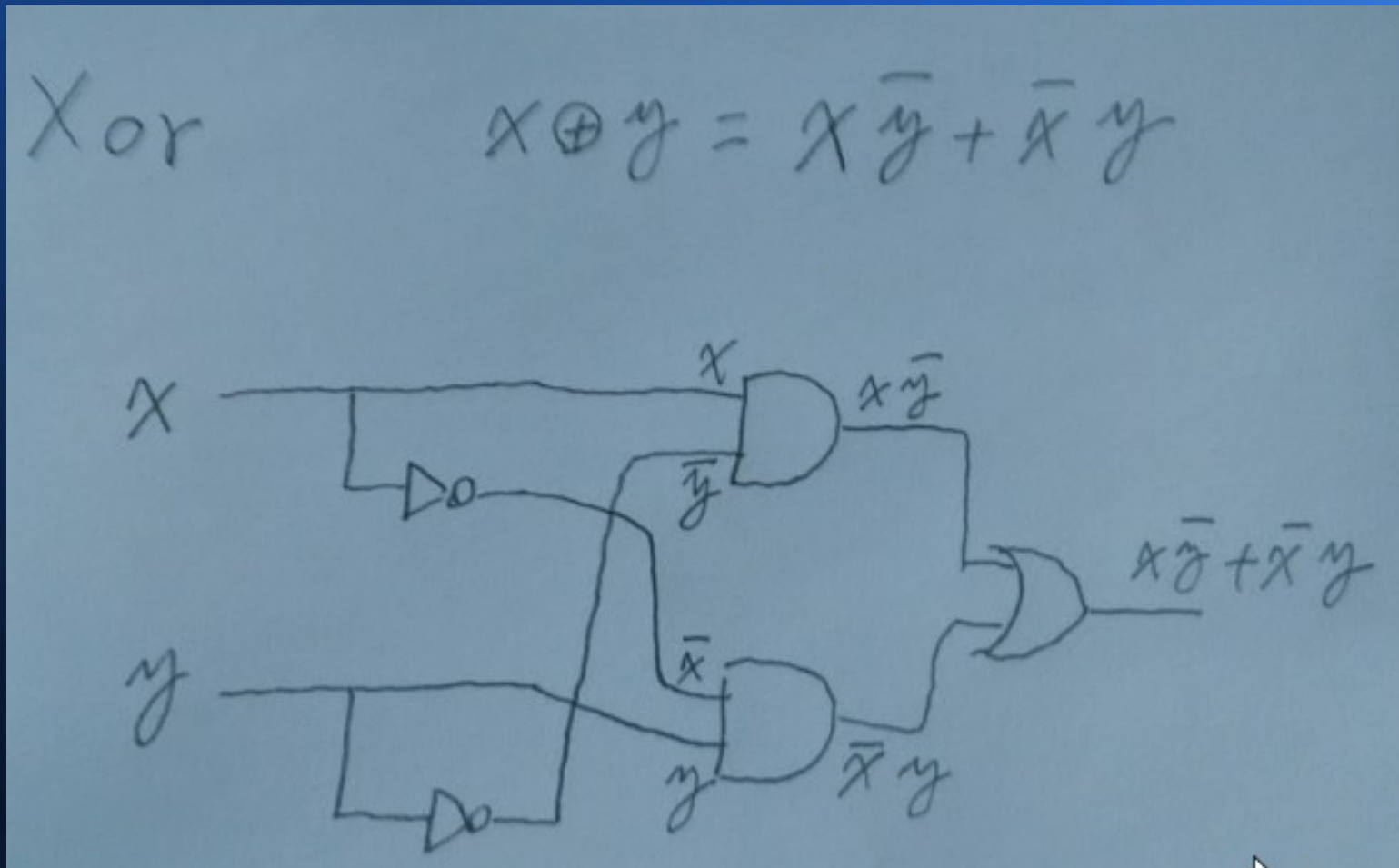
**Chip name:** Xor

**Inputs:** a, b

**Outputs:** out

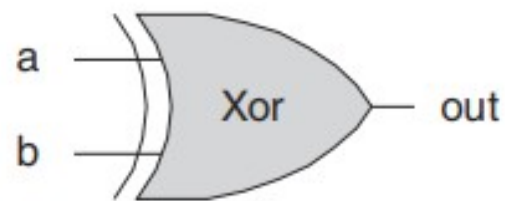
**Function:** If  $a \neq b$  then  $out=1$  else  $out=0$ .

# Xor (2)

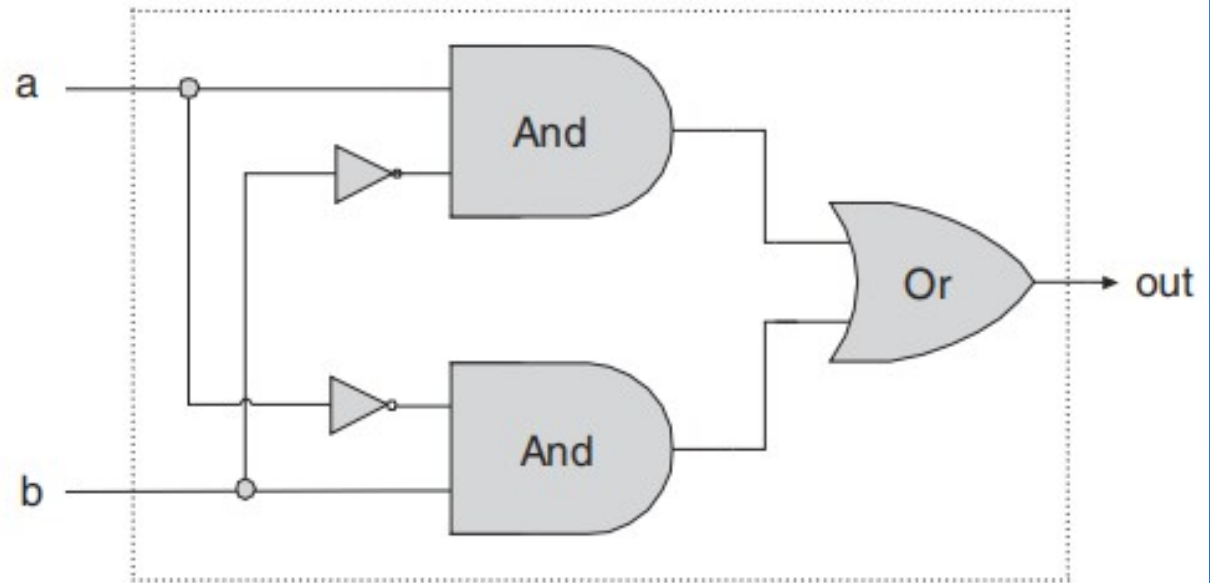




# Xor (3)



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0



**Figure 1.5** Xor gate, along with a possible implementation.

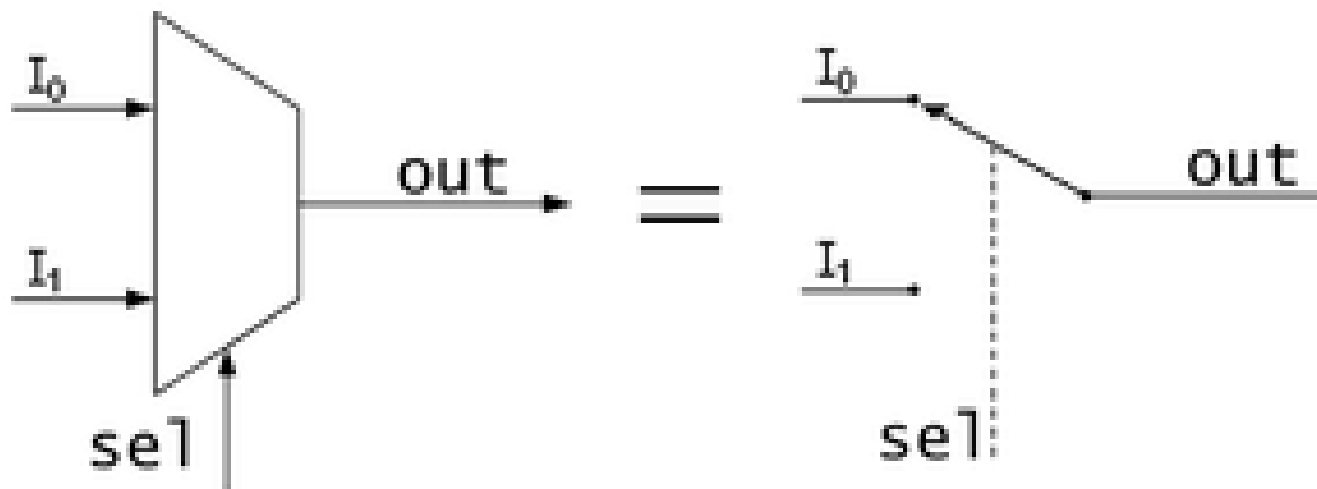
# Mux (1)

**Chip name:** Mux

**Inputs:** a, b, sel

**Outputs:** out

**Function:** If sel=0 then out=a else out=b.

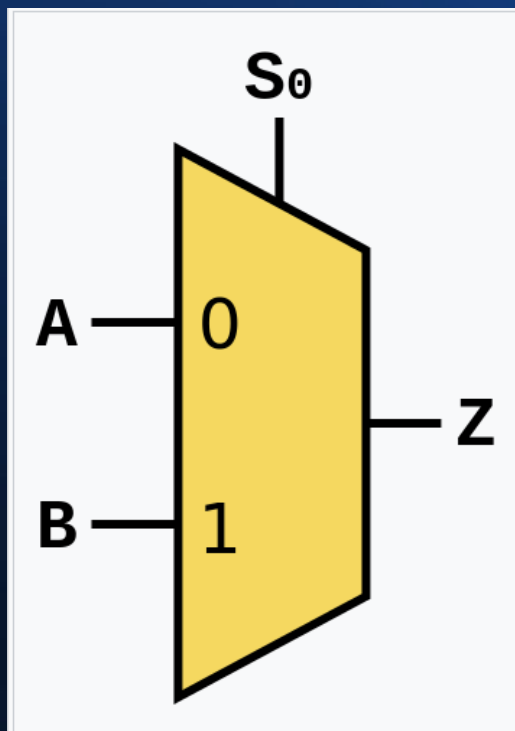


2選1數據多工器的結構簡圖，其功能類似一個雙擲的開關。



# Mux (2)

$$Z = (A \cdot \bar{S}) + (B \cdot S)$$



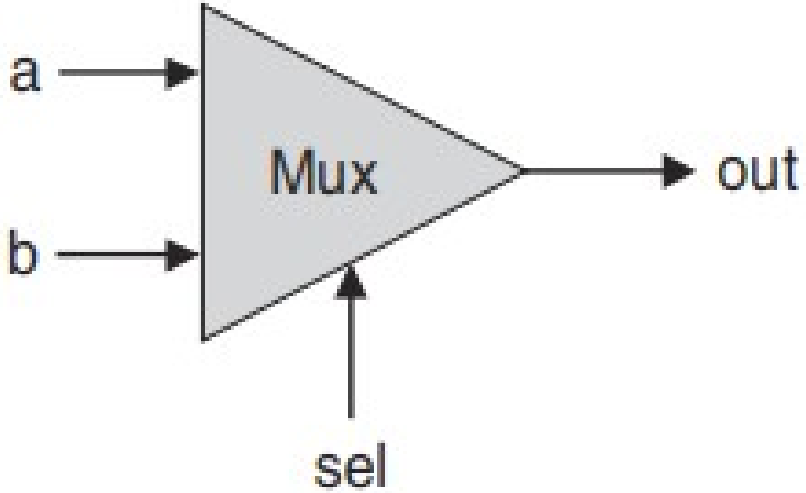
一個2選1數據多工器，A，B，S和Z分別表示兩個輸入訊號、選擇訊號和輸出訊號。

$S$	$A$	$B$	$Z$
0	1	1	1
	1	0	1
	0	1	0
	0	0	0
1	1	1	1
	1	0	0
	0	1	1
	0	0	0

# MUX (3)

a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

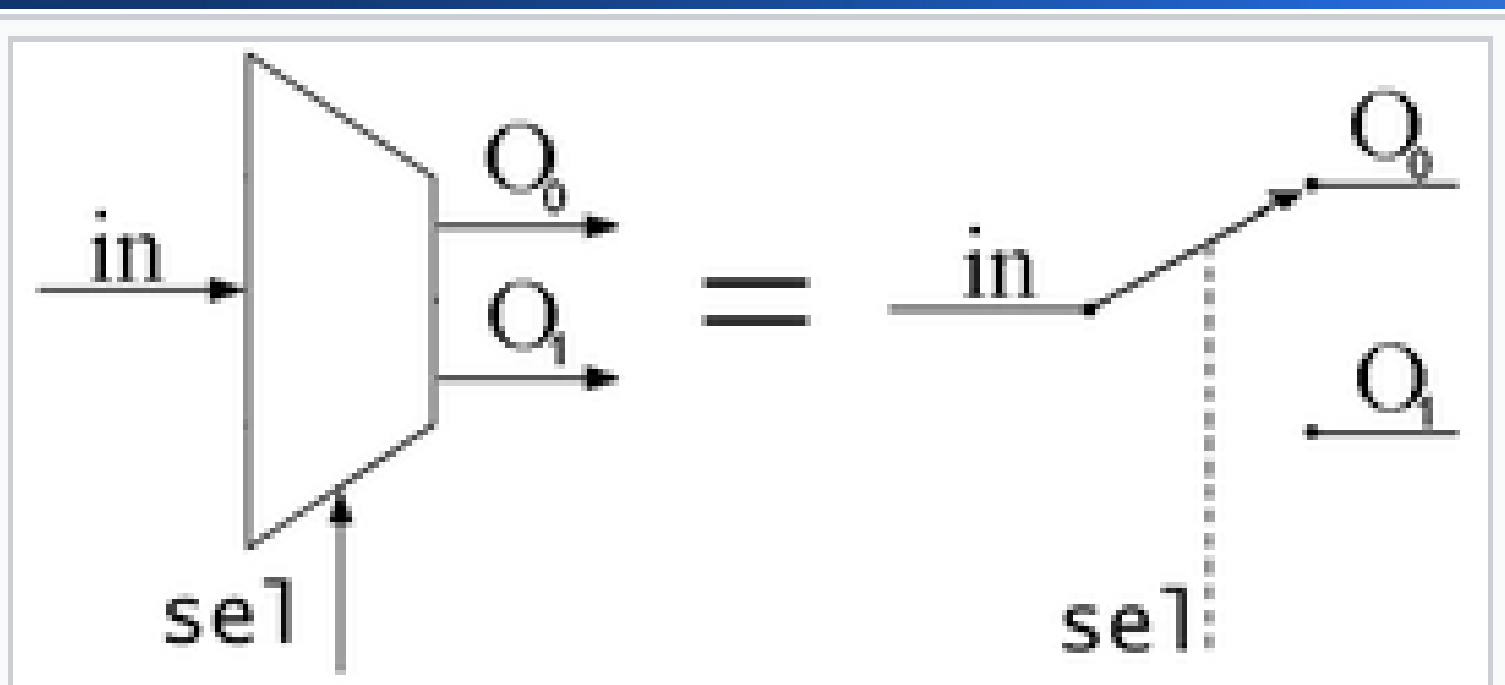


# MUX Chip

7400系列有若干種積體電路具有數據多工器功能，列表如下<sup>[6]</sup>：

	IC 晶片代號	功能	輸出狀態
1	74157	四2選1數據多工器	輸出原變量
2	74158	四2選1數據多工器	輸出反變量
3	74153	雙4選1數據多工器	輸出原變量
4	74352	雙4選1數據多工器	輸出反變量
5	74151A	8選1數據多工器	輸出原變量/反變量
6	74151	8選1數據多工器	輸出反變量
7	74150	16選1數據多工器	輸出反變量

# DMUX (1)



1線—2線數據分配器。像數據多工器一樣，它可以等同於一個控制開關。



# DMUX (2)

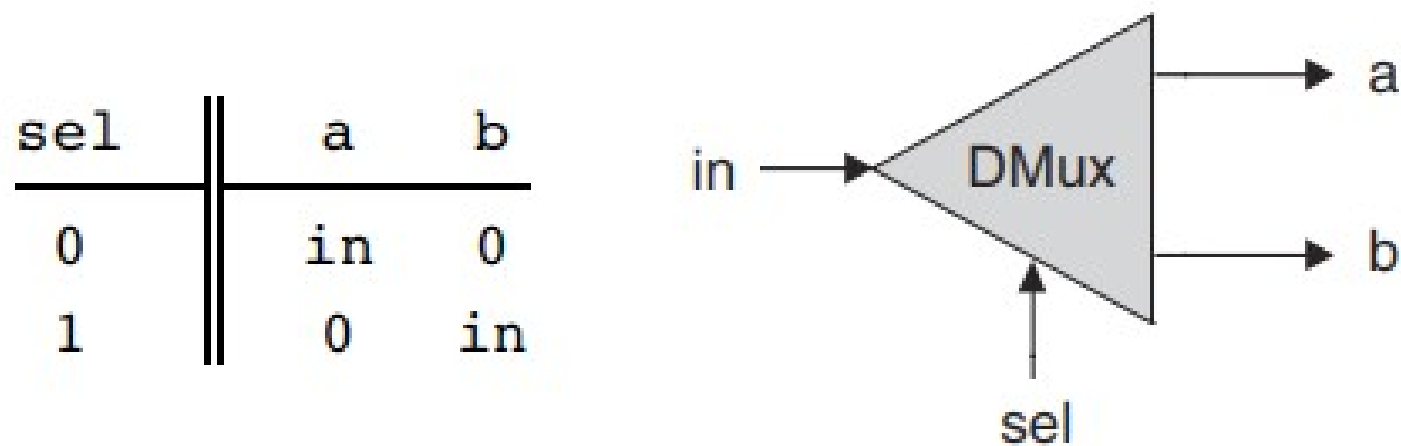
**Chip name:** DMux

**Inputs:** in, sel

**Outputs:** a, b

**Function:** If sel=0 then {a=in, b=0} else {a=0, b=in}.

# DMUX (3)



**Figure 1.9** Demultiplexor.



# Not16

**Multi-Bit Not** An  $n$ -bit Not gate applies the Boolean operation Not to every one of the bits in its  $n$ -bit input bus:

```
Chip name: Not16
Inputs:    in[16] // a 16-bit pin
Outputs:   out[16]
Function:   For i=0..15 out[i]=Not(in[i]).
```

# And16

**Multi-Bit And** An  $n$ -bit And gate applies the Boolean operation And to every one of the  $n$  bit-pairs arrayed in its two  $n$ -bit input buses:

```
Chip name: And16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:   For i=0..15 out[i]=And(a[i],b[i]).
```

# Or16

**Multi-Bit Or** An  $n$ -bit Or gate applies the Boolean operation Or to every one of the  $n$  bit-pairs arrayed in its two  $n$ -bit input buses:

**Chip name:** Or16

**Inputs:**  $a[16]$ ,  $b[16]$

**Outputs:**  $out[16]$

**Function:** For  $i=0..15$   $out[i]=Or(a[i],b[i])$ .

# Mux16

**Multi-Bit Multiplexor** An  $n$ -bit multiplexor is exactly the same as the binary multiplexor described in figure 1.8, except that the two inputs are each  $n$ -bit wide; the selector is a single bit.

**Chip name:** Mux16

**Inputs:**  $a[16]$ ,  $b[16]$ ,  $sel$

**Outputs:**  $out[16]$

**Function:** If  $sel=0$  then for  $i=0..15$   $out[i]=a[i]$   
else for  $i=0..15$   $out[i]=b[i]$ .

# Or8Way

**Multi-Way Or** An  $n$ -way Or gate outputs 1 when at least one of its  $n$  bit inputs is 1, and 0 otherwise. Here is the 8-way variant of this gate:

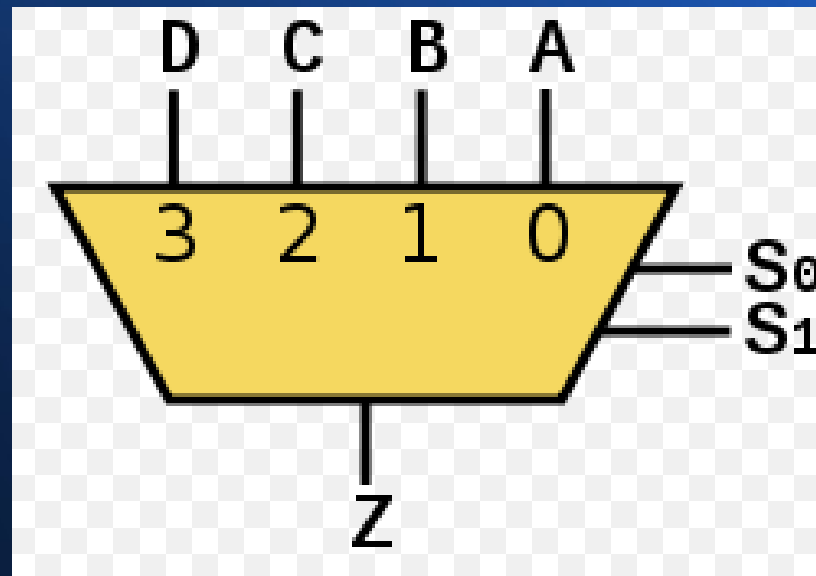
**Chip name:** Or8Way

**Inputs:** in[8]

**Outputs:** out

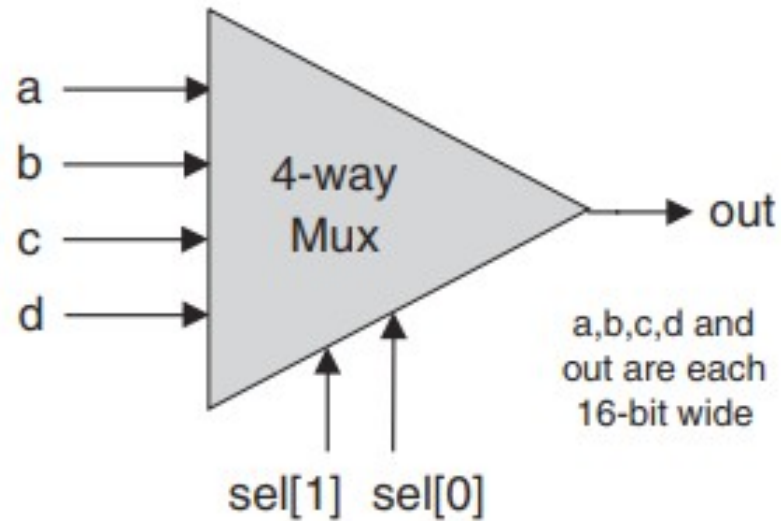
**Function:** out=Or(in[0],in[1],...,in[7]).

# Mux4Way (1)



# Mux4Way (2)

sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d



**Figure 1.10** 4-way multiplexor. The width of the input and output buses may vary.



# Mux4Way16

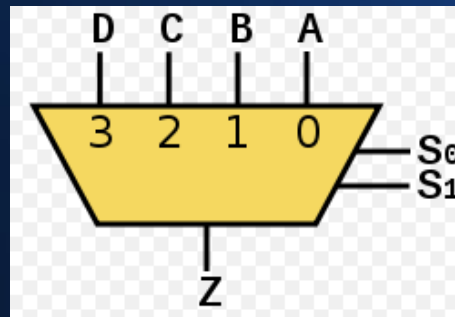
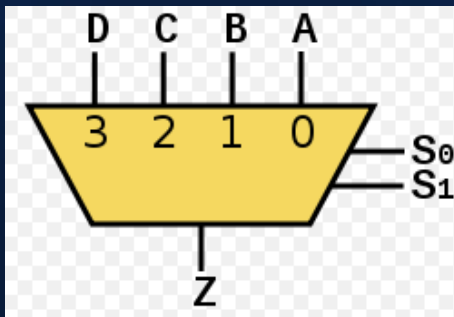
**Chip name:** Mux4Way16

**Inputs:** a[16], b[16], c[16], d[16], sel[2]

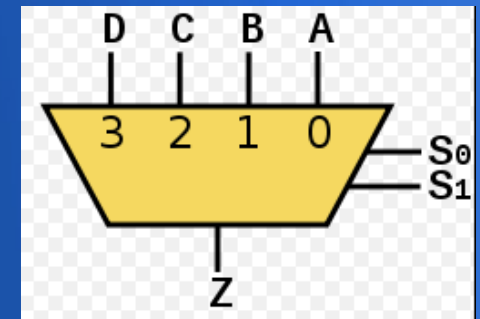
**Outputs:** out[16]

**Function:** If sel=00 then out=a else if sel=01 then out=b  
else if sel=10 then out=c else if sel=11 then out=d

**Comment:** The assignment operations mentioned above are all 16-bit. For example, "out=a" means "for i=0..15 out[i]=a[i]".

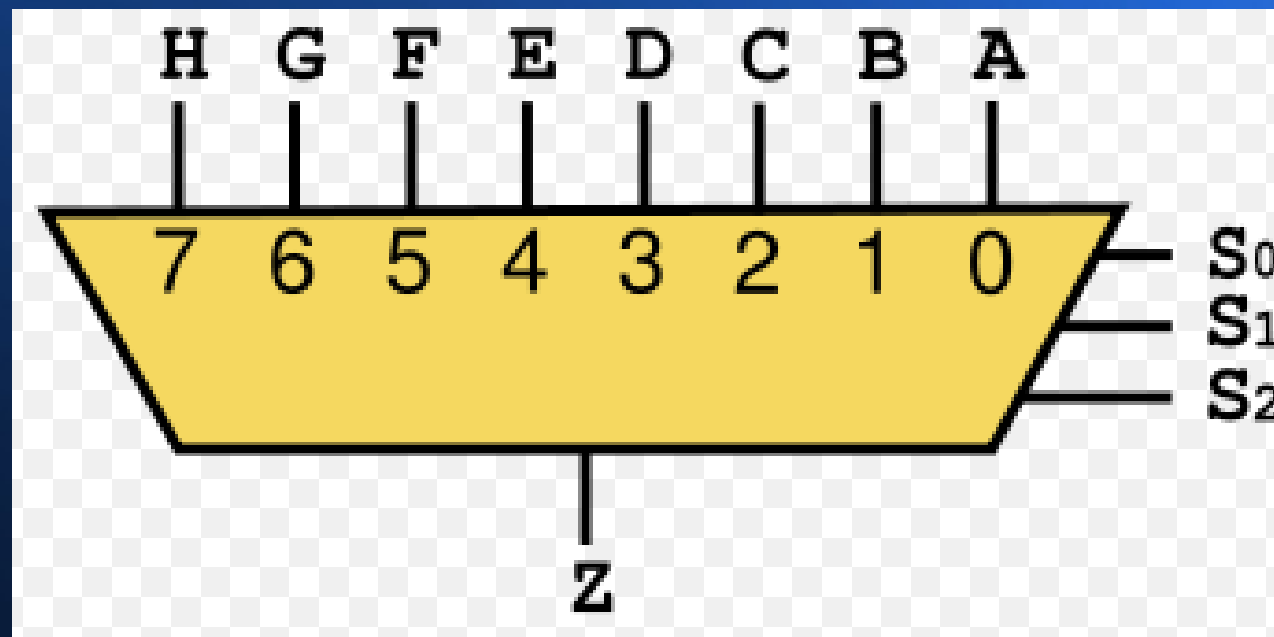


...





# Mux8Way



# Mux8Way16

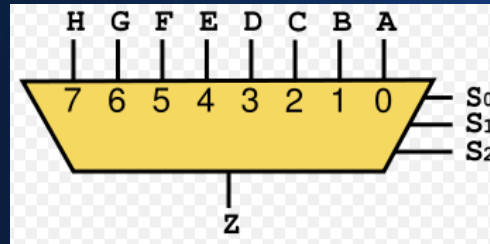
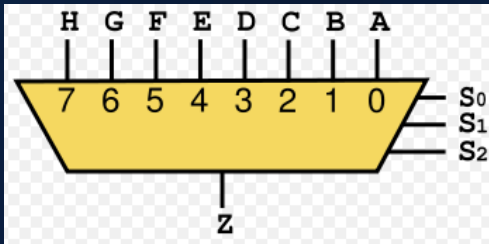
**Chip name:** Mux8Way16

**Inputs:** a[16], b[16], c[16], d[16], e[16], f[16], g[16], h[16],  
sel[3]

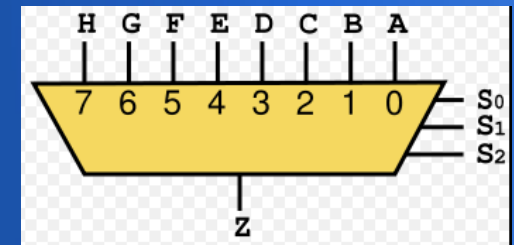
**Outputs:** out[16]

**Function:** If sel=000 then out=a else if sel=001 then out=b  
else if sel=010 out=c ... else if sel=111 then out=h

**Comment:** The assignment operations mentioned above are all  
16-bit. For example, "out=a" means "for i=0..15  
out[i]=a[i]".



■ ■ ■



# DMux4Way (1)

sel[1]	sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

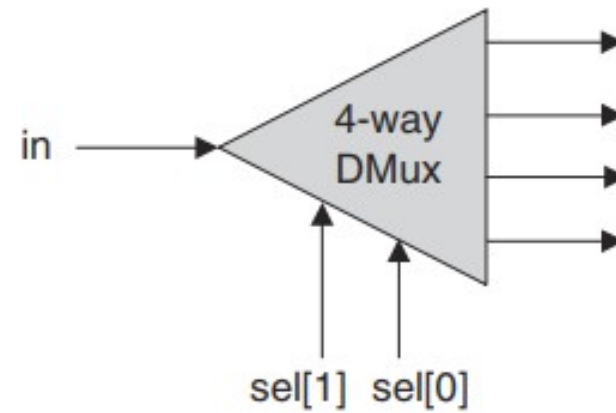


Figure 1.11 4-way demultiplexor.

**Chip name:** DMux4Way

**Inputs:** in, sel[2]

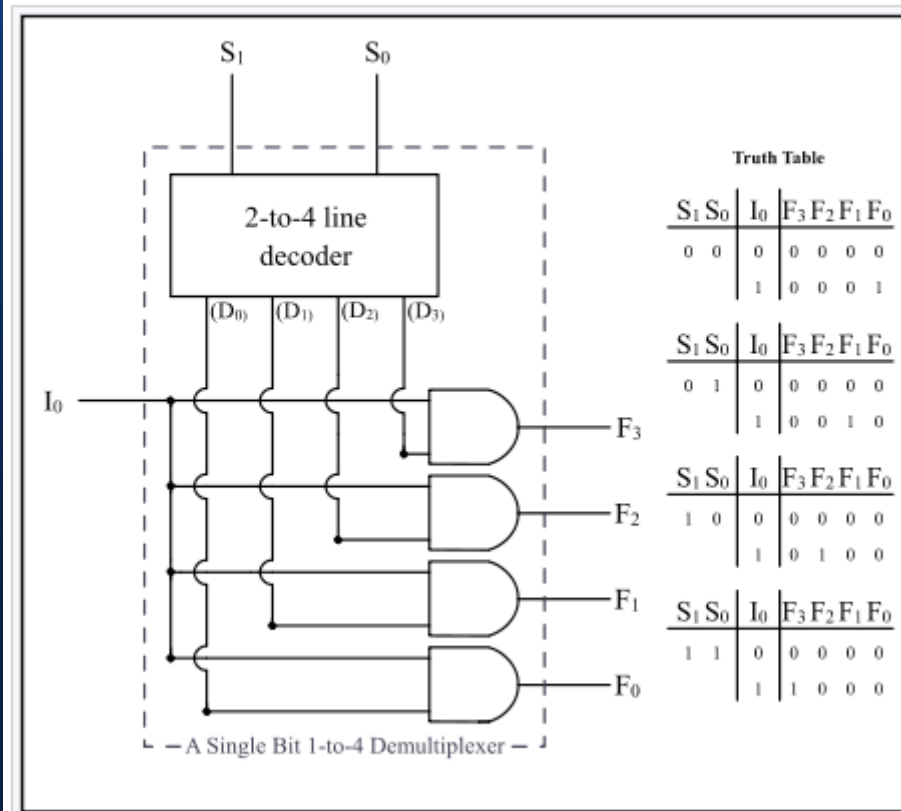
**Outputs:** a, b, c, d

**Function:** If sel=00 then {a=in, b=c=d=0}  
else if sel=01 then {b=in, a=c=d=0}  
else if sel=10 then {c=in, a=b=d=0}  
else if sel=11 then {d=in, a=b=c=0}.

# DMux4Way (2)

$$A = (X \cdot \overline{S})$$

$$B = (X \cdot S)$$



Example: A Single Bit 1-to-4 Line Demultiplexer



# DMux8Way

**Chip name:** DMux8Way

**Inputs:** in, sel[3]

**Outputs:** a, b, c, d, e, f, g, h

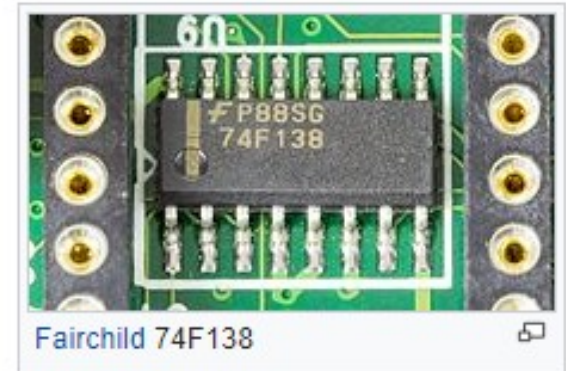
**Function:** If sel=000 then {a=in, b=c=d=e=f=g=h=0}  
else if sel=001 then {b=in, a=c=d=e=f=g=h=0}  
else if sel=010 ...  
...  
else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.

# DMUX Chip

## List of ICs which provide demultiplexing [\[ edit \]](#)

The 7400 series has several ICs that contain demultiplexer(s):

IC No. (7400)	IC No. (4000)	Function	Output State
74139		Dual 1:4 demux.	Output is inverted input
74156		Dual 1:4 demux.	Output is open collector
74138		1:8 demux.	Output is inverted input
74238		1:8 demux.	
74154		1:16 demux.	Output is inverted input
74159	CD4514/15	1:16 demux.	Output is open collector and same as input



# Chapter 2 – Boolean Arithmetic

# Project 2: Combinational Chips

Chip (HDL)	Description
HalfAdder	Half Adder
FullAdder	Full Adder
Add16	16-bit Adder
Inc16	16-bit incrementer
ALU	Arithmetic Logic Unit (without handling of status outputs)
ALU	Arithmetic Logic Unit (complete)



# Binary Addition

0	0	0	1		(carry)
	1	0	0	1	$x$
+	0	1	0	1	$y$
<hr/>					
0	1	1	1	0	$x + y$
no overflow					

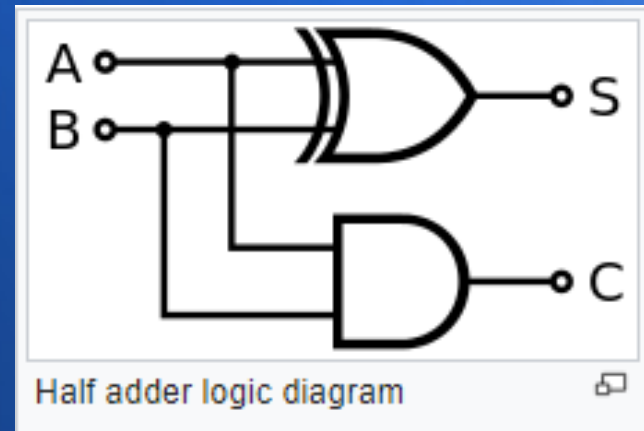
1	1	1	1	
	1	0	1	1
+	0	1	1	1
<hr/>				
1	0	0	1	0
overflow				

# 2's complement

**Figure 2.1** 2's complement representation of signed numbers in a 4-bit binary system.

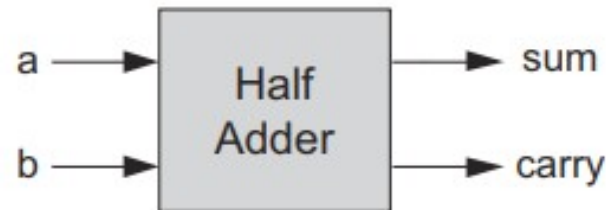
Positive numbers		Negative numbers	
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

# Half Adder (1)



# Half Adder (2)

Inputs		Outputs	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



**Chip name:** HalfAdder

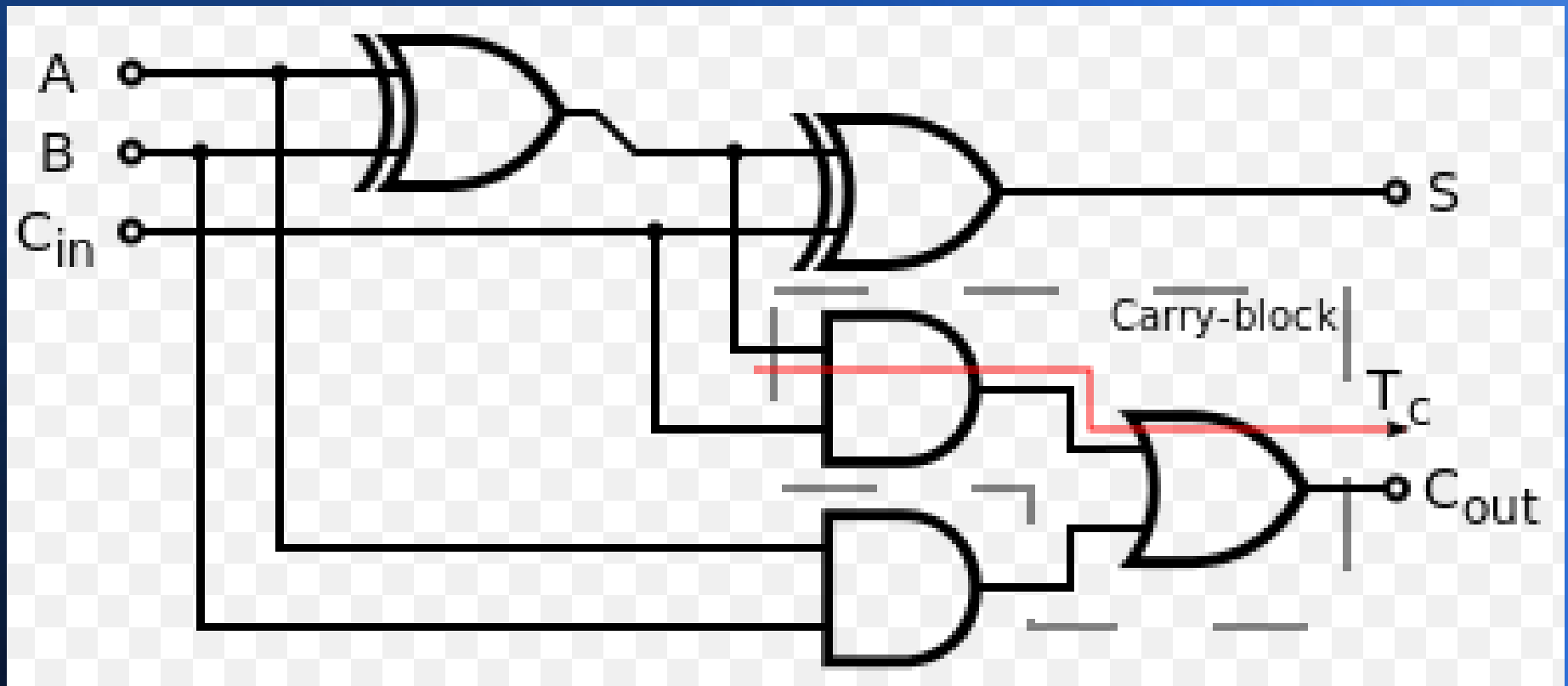
**Inputs:** a, b

**Outputs:** sum, carry

**Function:** sum = LSB of  $a + b$   
carry = MSB of  $a + b$

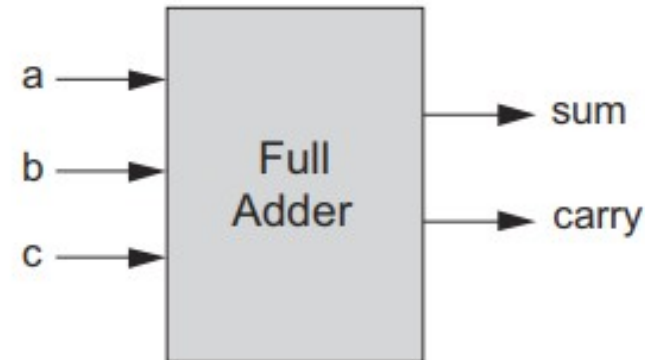
**Figure 2.2** Half-adder, designed to add 2 bits.

# Full Adder (1)



# Full Adder (2)

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



**Chip name:** FullAdder

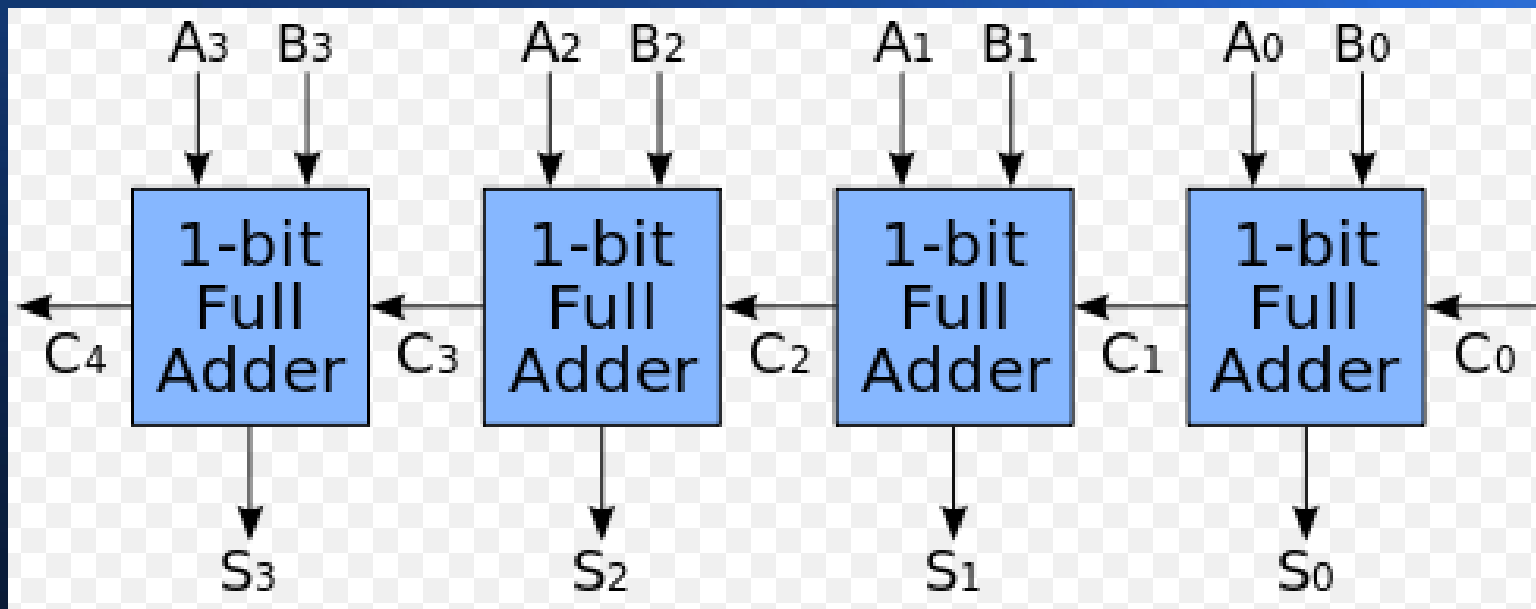
**Inputs:** a, b, c

**Outputs:** sum, carry

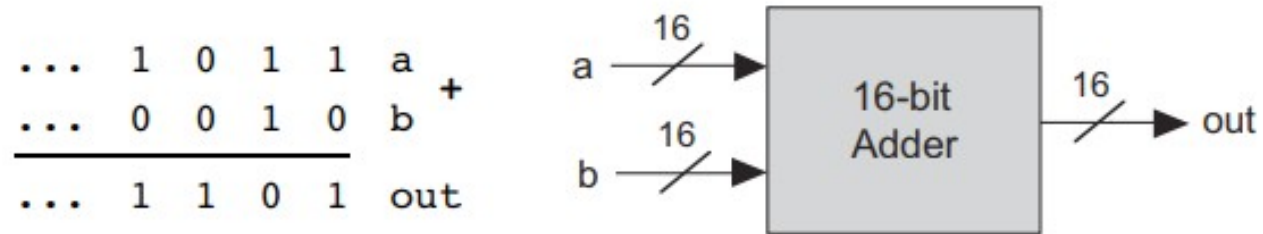
**Function:**  $\text{sum} = \text{LSB of } a + b + c$

$\text{carry} = \text{MSB of } a + b + c$

# Add16 (1)



# Add16 (2)



**Chip name:** Add16

**Inputs:** a[16], b[16]

**Outputs:** out[16]

**Function:** out = a + b

**Comment:** Integer 2's complement addition.  
Overflow is neither detected nor handled.

**Figure 2.4** 16-bit adder. Addition of two  $n$ -bit binary numbers for any  $n$  is “more of the same.”



# Inc16

- $\text{Inc16} = \text{Add16}(\text{in}, 1)$

**Chip name:** Inc16

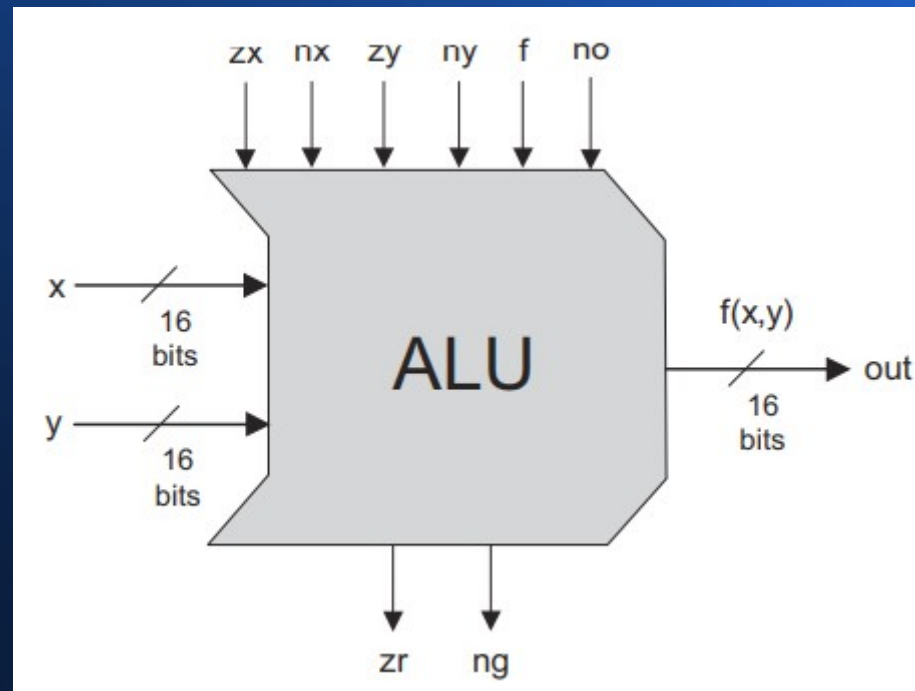
**Inputs:** in[16]

**Outputs:** out[16]

**Function:** out=in+1

**Comment:** Integer 2's complement addition.  
Overflow is neither detected nor handled.

# ALU (1)



# ALU (2)

**Chip name:** ALU

**Inputs:** x[16], y[16], // Two 16-bit data inputs  
zx, // Zero the x input  
nx, // Negate the x input  
zy, // Zero the y input  
ny, // Negate the y input  
f, // Function code: 1 for Add, 0 for And  
no // Negate the out output

**Outputs:** out[16], // 16-bit output  
zr, // True iff out=0  
ng // True iff out<0

**Function:** if zx then x = 0 // 16-bit zero constant  
if nx then x = !x // Bit-wise negation  
if zy then y = 0 // 16-bit zero constant  
if ny then y = !y // Bit-wise negation  
if f then out = x + y // Integer 2's complement addition  
else out = x & y // Bit-wise And  
if no then out = !out // Bit-wise negation  
if out=0 then zr = 1 else zr = 0 // 16-bit eq. comparison  
if out<0 then ng = 1 else ng = 0 // 16-bit neg. comparison

**Comment:** Overflow is neither detected nor handled.

# ALU

- without handling of status outputs

These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
zx	nx	zy	ny	f	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	f(x,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	!x
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	x-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

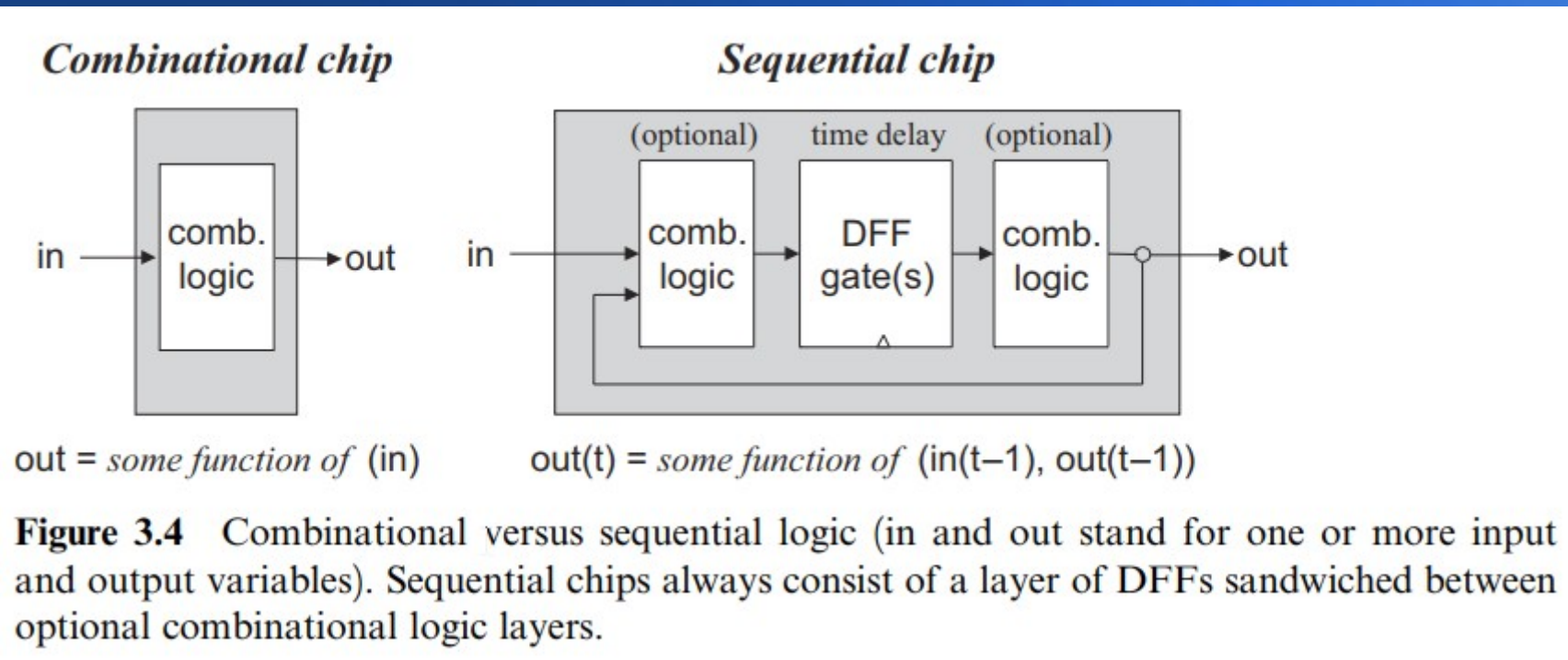
**Figure 2.6** The ALU truth table. Taken together, the binary operations coded by the first six columns effect the function listed in the right column (we use the symbols !, &, and | to represent the operators Not, And, and Or, respectively, performed bit-wise). The complete ALU truth table consists of sixty-four rows, of which only the eighteen presented here are of interest.

# ALU

- complete

```
* if the ALU output == 0, zr is set to 1; otherwise zr is set to 0;  
* if the ALU output < 0, ng is set to 1; otherwise ng is set to 0.  
*/  
  
// Implementation: the ALU logic manipulates the x and y inputs  
// and operates on the resulting values, as follows:  
// if (zx == 1) set x = 0          // 16-bit constant  
// if (nx == 1) set x = !x        // bitwise not  
// if (zy == 1) set y = 0          // 16-bit constant  
// if (ny == 1) set y = !y        // bitwise not  
// if (f == 1) set out = x + y    // integer 2's complement addition  
// if (f == 0) set out = x & y    // bitwise and  
// if (no == 1) set out = !out    // bitwise not  
// if (out == 0) set zr = 1  
// if (out < 0) set ng = 1
```

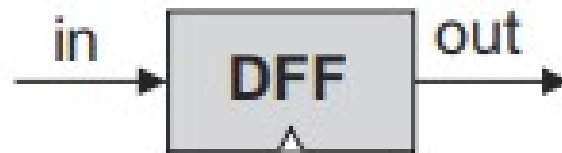
# Chapter 3 – Sequential Logic



# Project 3: Sequential Chips

Chip (HDL)	Description
DFF	Data Flip-Flop (primitive)
Bit	1-bit register
Register	16-bit register
RAM8	16-bit / 8-register memory
RAM64	16-bit / 64-register memory
RAM512	16-bit / 512-register memory
RAM4K	16-bit / 4096-register memory
RAM16K	16-bit / 16384-register memory
PC	16-bit program counter

# DFF : D Flip-Flop

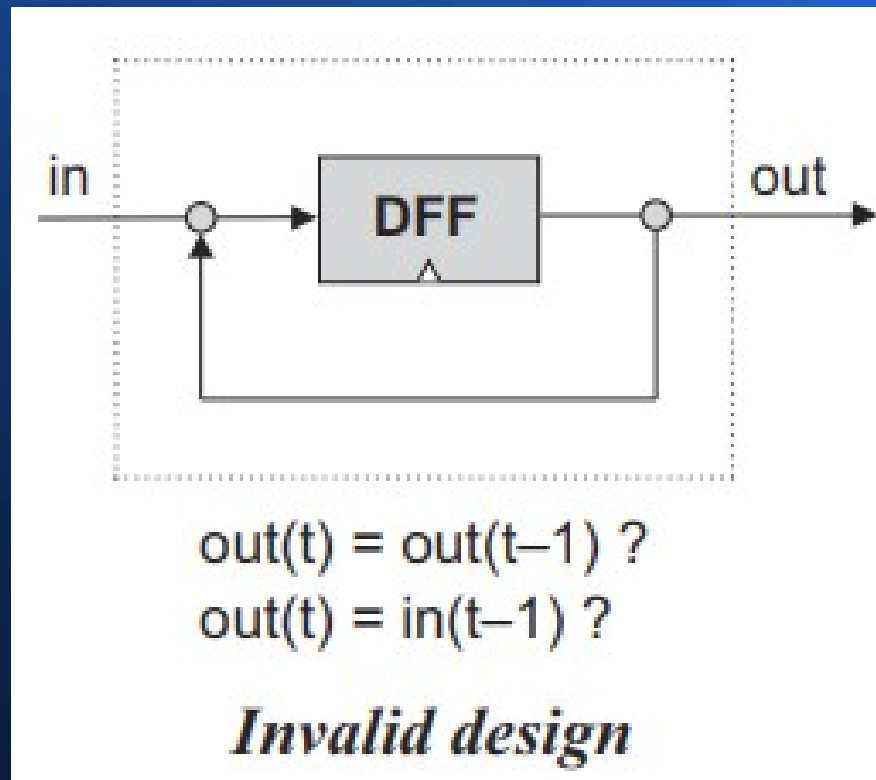


$$\text{out}(t) = \text{in}(t-1)$$

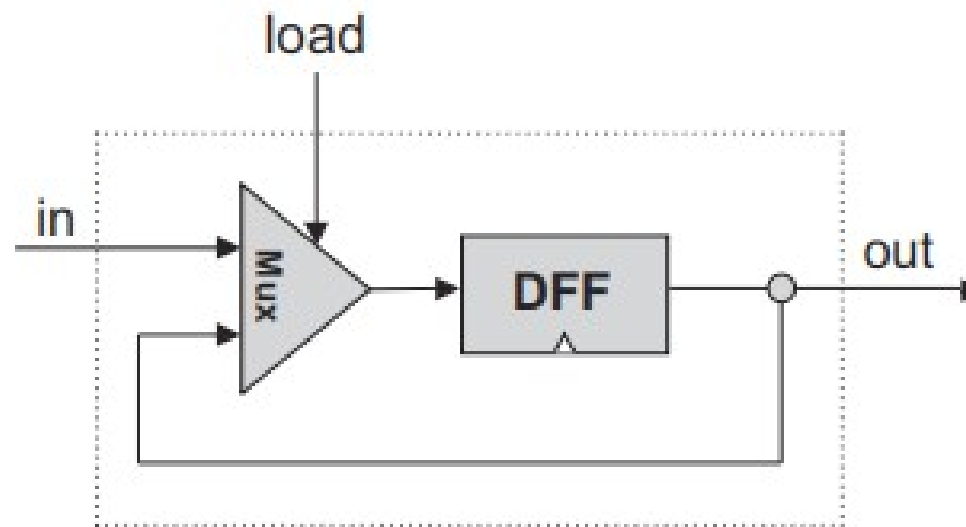
*Flip-flop*



# Bit ?



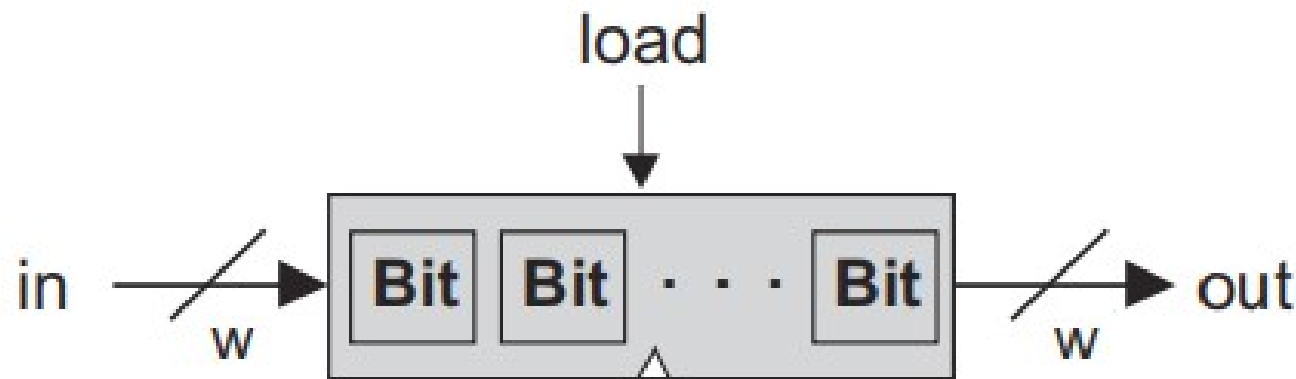
# Bit



if  $\text{load}(t-1)$  then  $\text{out}(t) = \text{in}(t-1)$   
else  $\text{out}(t) = \text{out}(t-1)$

*1-bit register (Bit)*

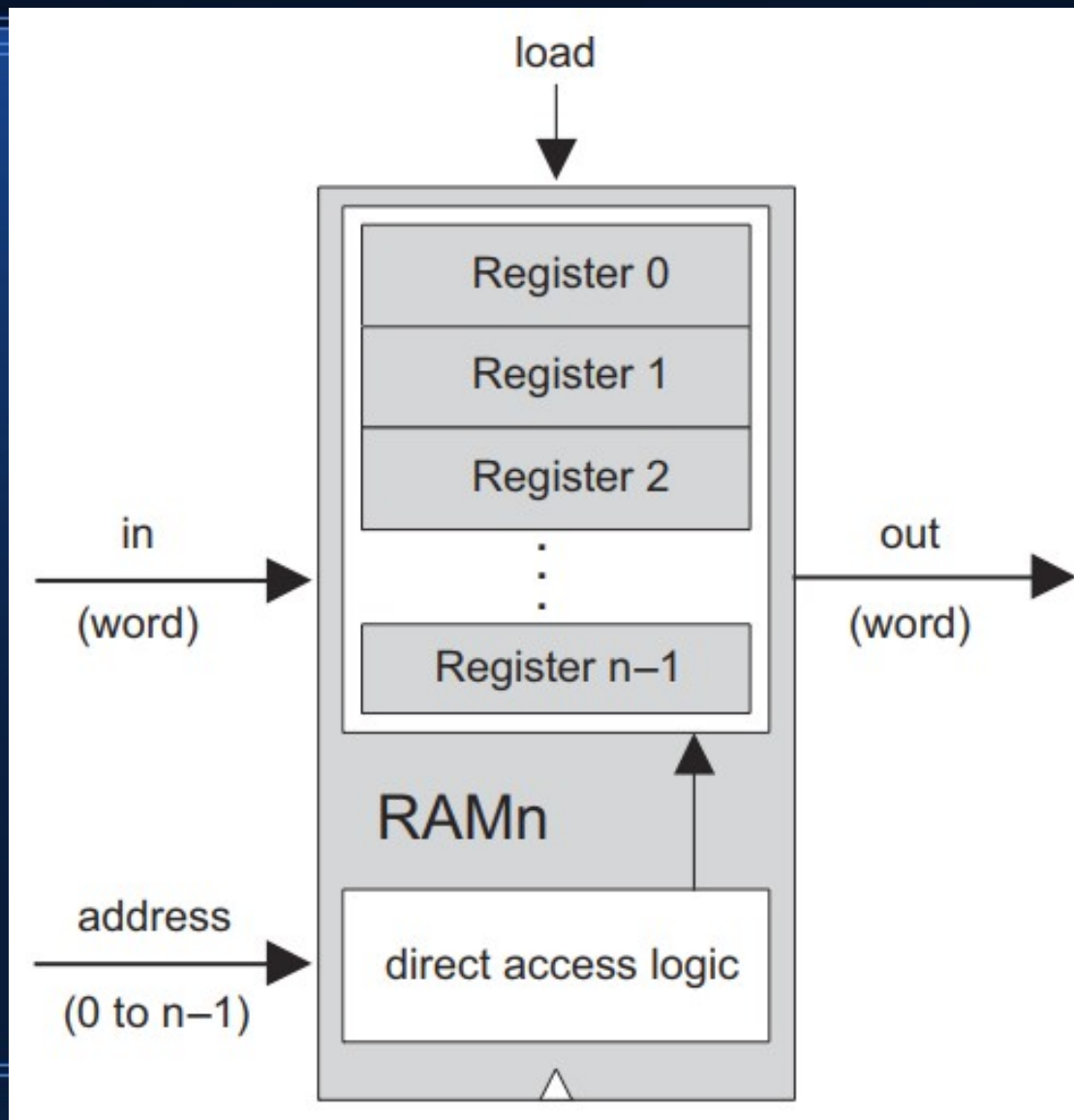
# Register



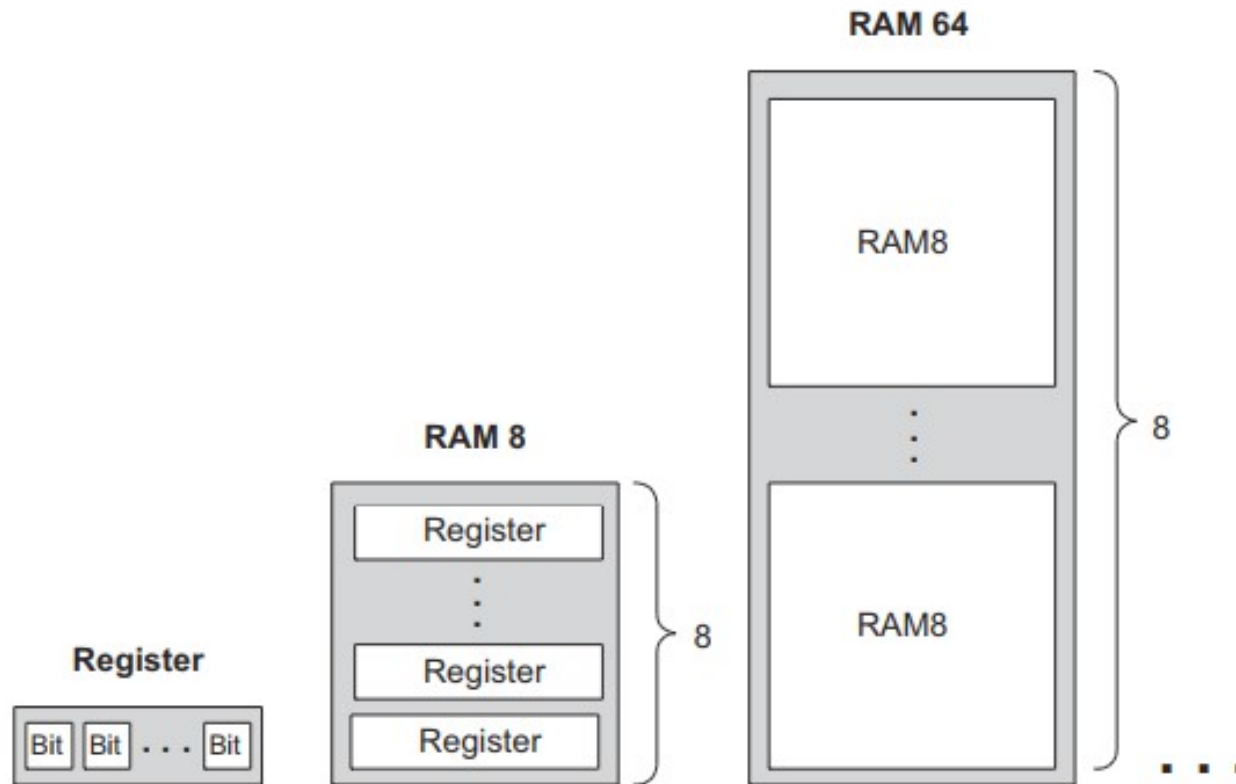
if  $\text{load}(t-1)$  then  $\text{out}(t) = \text{in}(t-1)$   
else  $\text{out}(t) = \text{out}(t-1)$

*$w$ -bit register*

# RAM

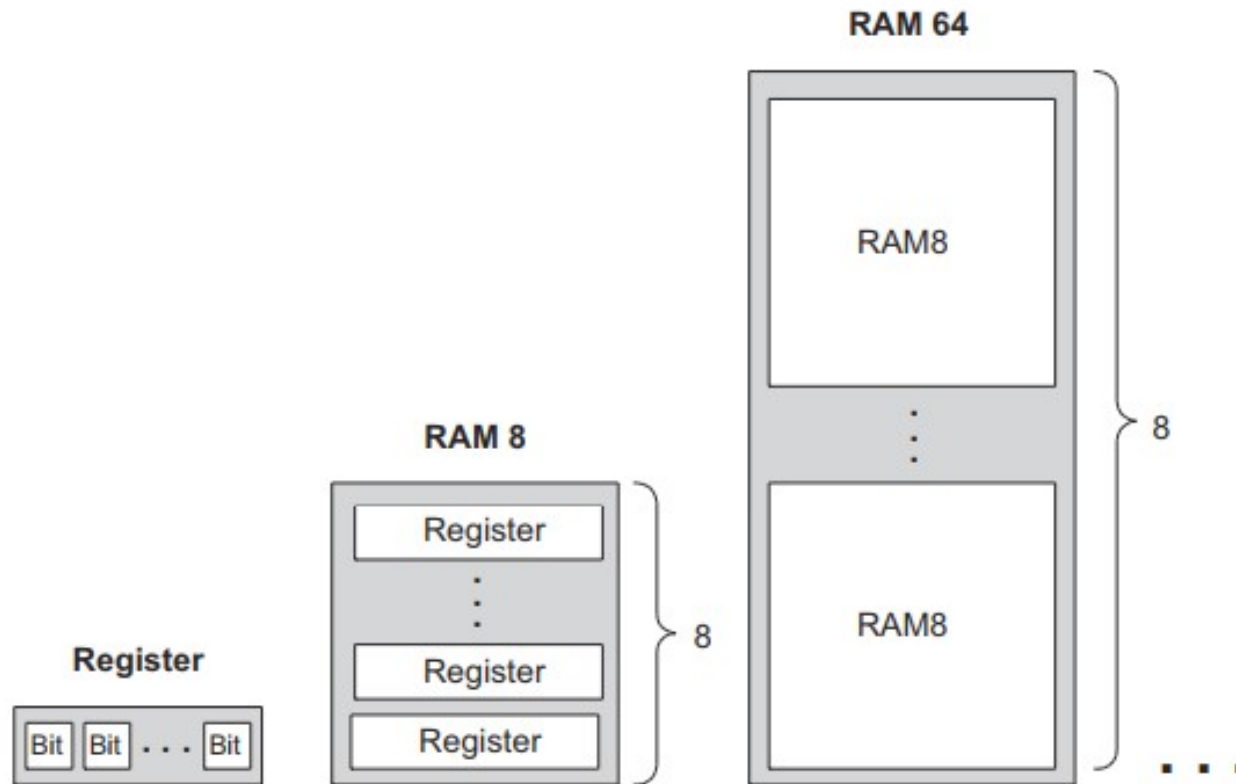


# RAM8



**Figure 3.6** Gradual construction of memory banks by recursive ascent. A  $w$ -bit register is an array of  $w$  binary cells, an 8-register RAM is an array of eight  $w$ -bit registers, a 64-register RAM is an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build a 16K RAM chip.

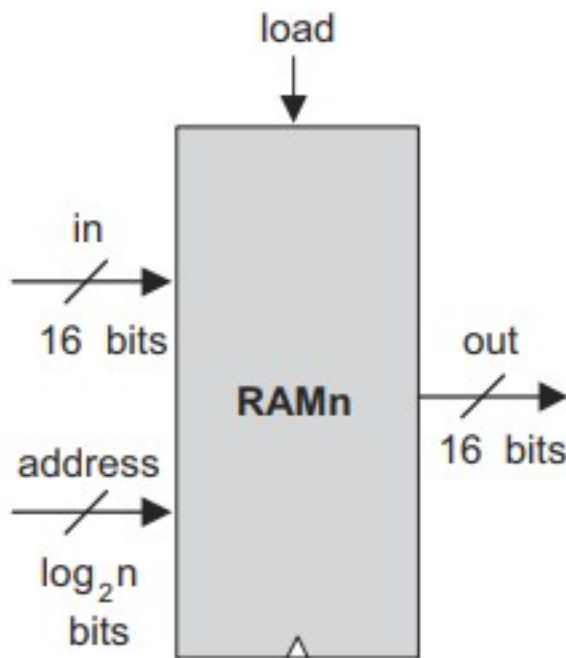
# RAM64



**Figure 3.6** Gradual construction of memory banks by recursive ascent. A  $w$ -bit register is an array of  $w$  binary cells, an 8-register RAM is an array of eight  $w$ -bit registers, a 64-register RAM is an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build a 16K RAM chip.

# RAM512 、 RAM4K 、 RAM16k ...

- Just the same way ...



**Chip name:** RAMn // n and k are listed below

**Inputs:** in[16], address[k], load

**Outputs:** out[16]

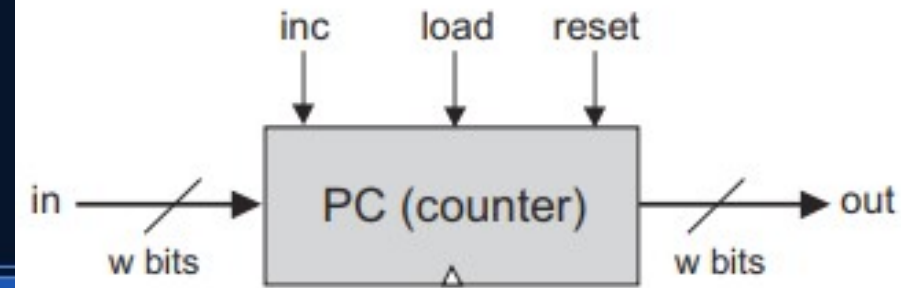
**Function:**  $out(t) = RAM[address(t)](t)$   
If load(t-1) then  
 $RAM[address(t-1)](t) = in(t-1)$

**Comment:** "=" is a 16-bit operation.

The specific RAM chips needed for the Hack platform are:

Chip name	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

# PC (Program Counter)



```
/**
 * A 16-bit counter with load and reset control bits.
 * if      (reset[t] == 1) out[t+1] = 0
 * else if (load[t] == 1)  out[t+1] = in[t]
 * else if (inc[t] == 1)   out[t+1] = out[t] + 1 (integer addition)
 * else                   out[t+1] = out[t]
 */

CHIP PC {
    IN in[16], load, inc, reset;
    OUT out[16];

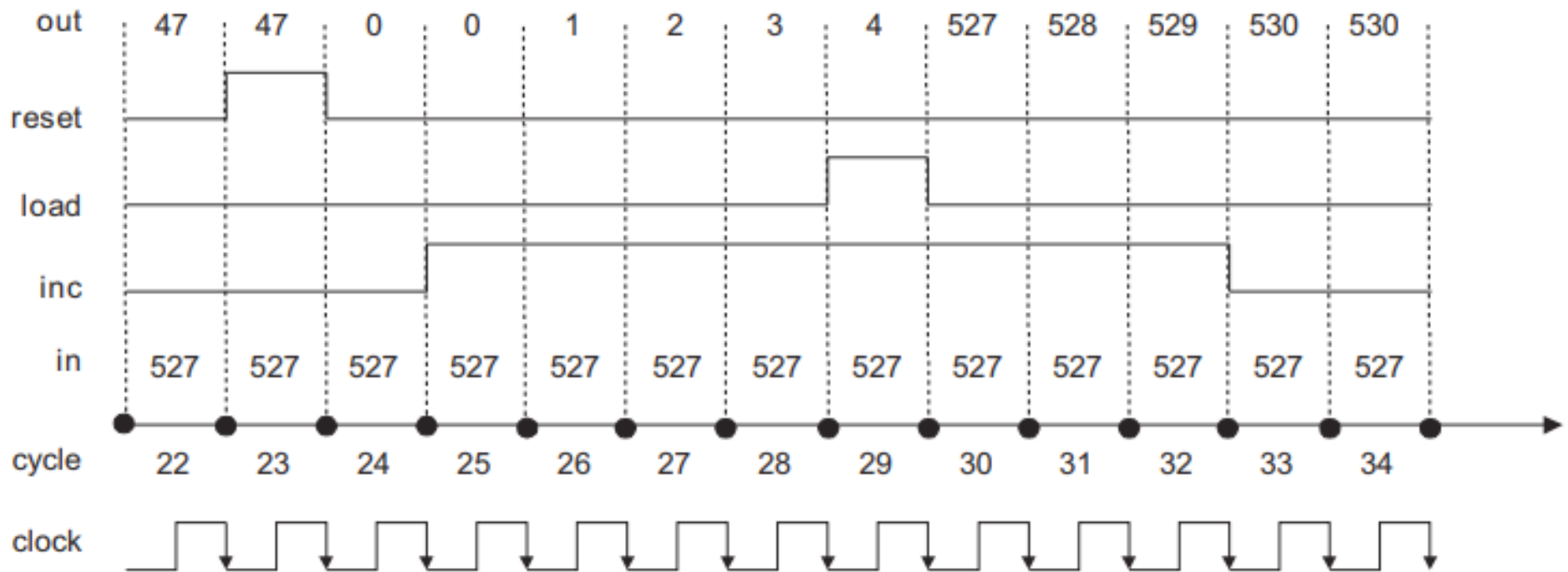
    PARTS:
        // Put your code here:
}
```



# PC (2)

```
Chip name: PC // 16-bit counter
Inputs:    in[16], inc, load, reset
Outputs:   out[16]
Function:  If reset(t-1) then out(t)=0
           else if load(t-1) then out(t)=in(t-1)
           else if inc(t-1) then out(t)=out(t-1)+1
           else out(t)=out(t-1)
Comment:   "=" is 16-bit assignment.
           "+" is 16-bit arithmetic addition.
```

# PC (3)



# Chapter 4 – Machine Language

# Project 4: Machine Language Programming

Program	Description	Comments / Tests
Mult.asm	<p><b>Multiplication:</b> in the Hack computer, the top 16 RAM words (<math>\text{RAM}[0] \dots \text{RAM}[15]</math>) are also referred to as the so-called <i>virtual registers</i> <math>R0 \dots R15</math>.</p> <p>With this terminology in mind, this program computes the value <math>R0 * R1</math> and stores the result in <math>R2</math>.</p> <p>Note that in the context of this program, we assume that <math>R0 \geq 0</math>, <math>R1 \geq 0</math>, and <math>R0 * R1 &lt; 32768</math> (you are welcome to ponder where this limiting value comes from). Your program need not test these conditions, but rather assume that they hold.</p>	<p>Start by using the supplied assembler to translate your <code>Mult.asm</code> program into a <code>Mult.hack</code> file.</p> <p>To test your program, load <code>Mult.hack</code> into the ROM. Next, put some values in <math>R0</math> and <math>R1</math>, run the code, and inspect <math>R2</math>.</p> <p>Alternatively, use the supplied <code>Mult.tst</code> script and <code>Mult.cmp</code> compare file (that's how we test your program). These supplied files are designed to test your program by running it on several representative data values.</p>
Fill.asm	<p><b>I/O handling:</b> this program illustrates low-level handling of the screen and keyboard devices, as follows.</p> <p>The program runs an infinite loop that listens to the keyboard input. When a key is pressed (any key), the program blackens the screen, i.e. writes "black" in every pixel; the screen should remain fully black as long as the key is pressed.</p> <p>When no key is pressed, the program clears the screen, i.e. writes "white" in every pixel; the screen should remain fully clear as long as no key is pressed.</p>	<p>Start by using the supplied assembler to translate your <code>Fill.asm</code> program into a <code>Fill.hack</code> file. Implementation note: your program may blacken and clear the screen's pixels in any spatial/visual order, as long as pressing a key continuously for long enough results in a fully blackened screen, and not pressing any key for long enough results in a fully cleared screen.</p> <p>The simple <code>Fill.tst</code> script, which comes with no compare file, is designed to do two things: (i) load the <code>Fill.hack</code> program, and (ii) remind you to select 'no animation', and then test the program interactively by pressing and releasing some keyboard keys.</p> <p>The <code>FillAutomatic.tst</code> script, along with the compare file <code>FillAutomatic.cmp</code>, are designed to test the Fill program automatically, as described by the test script documentation.</p> <p>For completeness of testing, it is recommended to test the Fill program both interactively and automatically.</p>

# Assembly (1)

## The A-instruction

```
@value    // A ← value
```

Where *value* is either a number or a symbol referring to some number.

### Used for:

- Entering a constant value  
(A = value)
- Selecting a RAM location  
(register = RAM[A])
- Selecting a ROM location  
(PC = A)

### Coding example:

```
@17    // A = 17  
D = A   // D = 17
```

```
@17    // A = 17  
D = M   // D = RAM[17]
```

```
@17    // A = 17  
JMP     // fetch the instruction  
        // stored in ROM[17]
```

Later

# Assembly (2)

## IF logic – Hack style

High level:

```
if condition {  
    code block 1}  
else {  
    code block 2}  
code block 3
```

Hack convention:

- ❑ True is represented by -1
- ❑ False is represented by 0

Hack:

```
D ← not condition  
@IF_TRUE  
D;JEQ  
code block 2  
@END  
0;JMP  
(IF_TRUE)  
code block 1  
(END)  
code block 3
```

# Assembly (3)

## WHILE logic – Hack style

High level:

```
while condition {  
    code block 1  
}  
Code block 2
```

Hack:

```
(LOOP)  
    D ← not condition)  
    @END  
    D;JEQ  
    code block 1  
    @LOOP  
    0;JMP  
(END)  
    code block 2
```

Hack convention:

- ❑ True is represented by -1
- ❑ False is represented by 0



# Assembly (4)

## Complete program example

### C language code:

```
// Adds 1+...+100.  
into i = 1;  
into sum = 0;  
while (i <= 100){  
    sum += i;  
    i++;  
}
```

### Hack assembly code:

```
// Adds 1+...+100.  
@i      // i refers to some RAM location  
M=1     // i=1  
@sum    // sum refers to some RAM location  
M=0     // sum=0  
(LOOP)  
@i  
D=M     // D = i  
@100  
D=D-A   // D = i - 100  
@END  
D;JGT   // If (i-100) > 0 goto END  
@i  
D=M     // D = i  
@sum  
M=D+M   // sum += i  
@i  
M=M+1   // i++  
@LOOP  
0;JMP   // Got LOOP  
(END)  
@END  
0;JMP   // Infinite loop
```

### Hack assembly convention:

- Variables: lower-case
- Labels: upper-case
- Commands: upper-case

Demo  
CPU emulator



# Mult.asm

Program	Description	Comments / Tests
Mult.asm	<p><b>Multiplication:</b> in the Hack computer, the top 16 RAM words (RAM[0] ... RAM[15]) are also referred to as the so-called <i>virtual registers</i> R0 ... R15.</p> <p>With this terminology in mind, this program computes the value <math>R0 \cdot R1</math> and stores the result in R2.</p> <p>Note that in the context of this program, we assume that <math>R0 \geq 0</math>, <math>R1 \geq 0</math>, and <math>R0 \cdot R1 &lt; 32768</math> (you are welcome to ponder where this limiting value comes from). Your program need not test these conditions, but rather assume that they hold.</p>	<p>Start by using the supplied assembler to translate your Mult.asm program into a Mult.hack file.</p> <p>To test your program, load Mult.hack into the ROM. Next, put some values in R0 and R1, run the code, and inspect R2.</p> <p>Alternatively, use the supplied Mult.tst script and Mult.cmp compare file (that's how we test your program). These supplied files are designed to test your program by running it on several representative data values.</p>

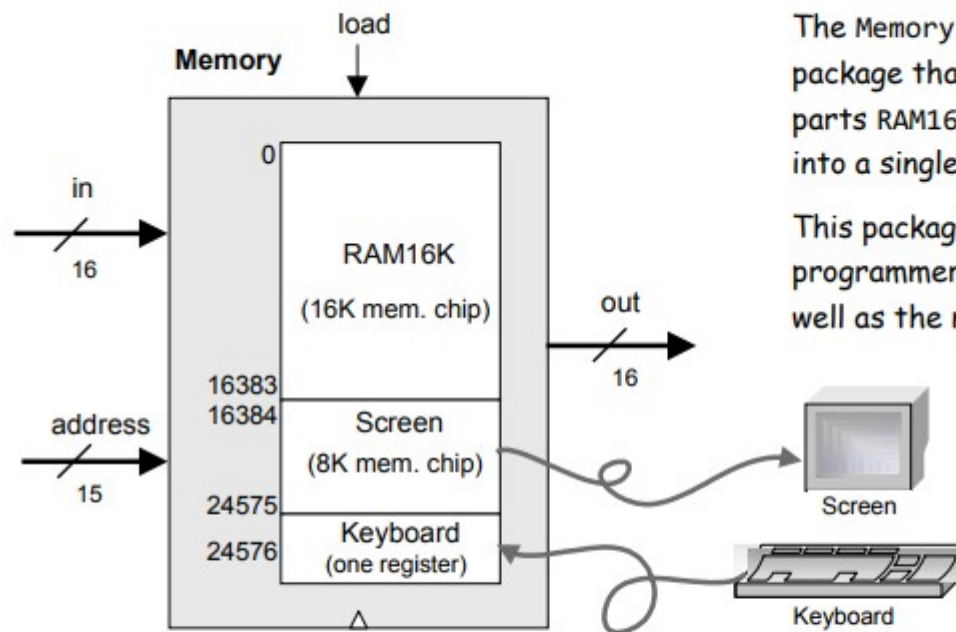
# Assembly (I/O)

- Memory-Mapped I/O

**I/O pointers:** The symbols `SCREEN` and `KBD` are "automatically" predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the Hack platform's *screen* and *keyboard* memory maps)

# Memory (mapped to device)

## Memory: physical implementation



The Memory chip is essentially a package that integrates the three chip-parts RAM16K, Screen, and Keyboard into a single, contiguous address space.

This packaging effects the programmer's view of the memory, as well as the necessary I/O side-effects.

### Access logic:

- ❑ Access to any address from 0 to 16,383 results in accessing the RAM16K chip-part
- ❑ Access to any address from 16,384 to 24,575 results in accessing the Screen chip-part
- ❑ Access to address 24,576 results in accessing the keyboard chip-part
- ❑ Access to any other address is invalid.

# Fill.asm

## Fill.asm

**I/O handling:** this program illustrates low-level handling of the screen and keyboard devices, as follows.

The program runs an infinite loop that listens to the keyboard input. When a key is pressed (any key), the program blackens the screen, i.e. writes "black" in every pixel; the screen should remain fully black as long as the key is pressed.

When no key is pressed, the program clears the screen, i.e. writes "white" in every pixel; the screen should remain fully clear as long as no key is pressed.

Start by using the supplied assembler to translate your Fill.asm program into a Fill.hack file. Implementation note: your program may blacken and clear the screen's pixels in any spatial/visual order, as long as pressing a key continuously for long enough results in a fully blackened screen, and not pressing any key for long enough results in a fully cleared screen.

The simple Fill.tst script, which comes with no compare file, is designed to do two things: (i) load the Fill.hack program, and (ii) remind you to select 'no animation', and then test the program interactively by pressing and releasing some keyboard keys.

The FillAutomatic.tst script, along with the compare file FillAutomatic.cmp, are designed to test the Fill program automatically, as described by the test script documentation.

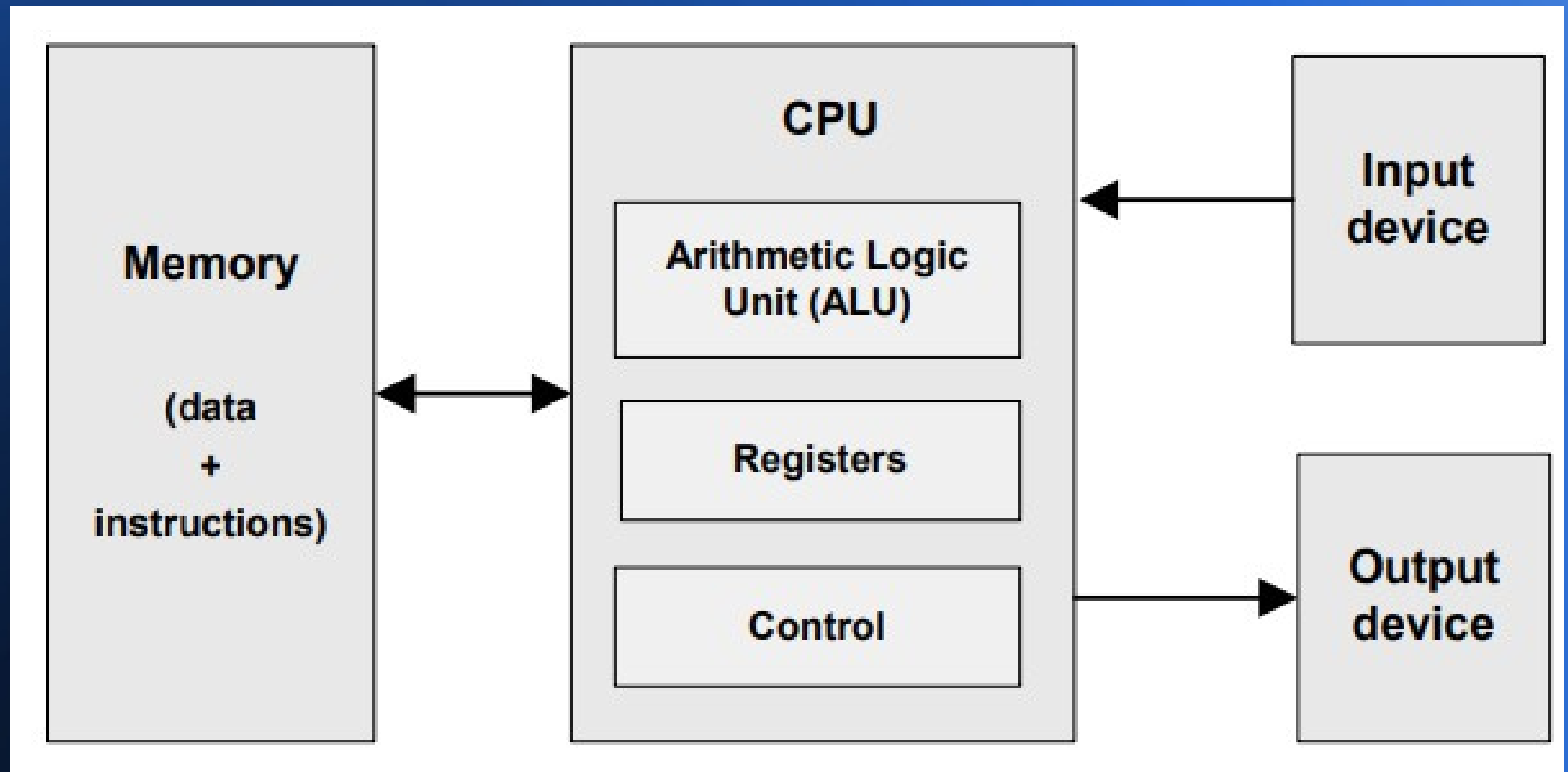
For completeness of testing, it is recommended to test the Fill program both interactively and automatically.

# Chapter 5 – Computer Architecture

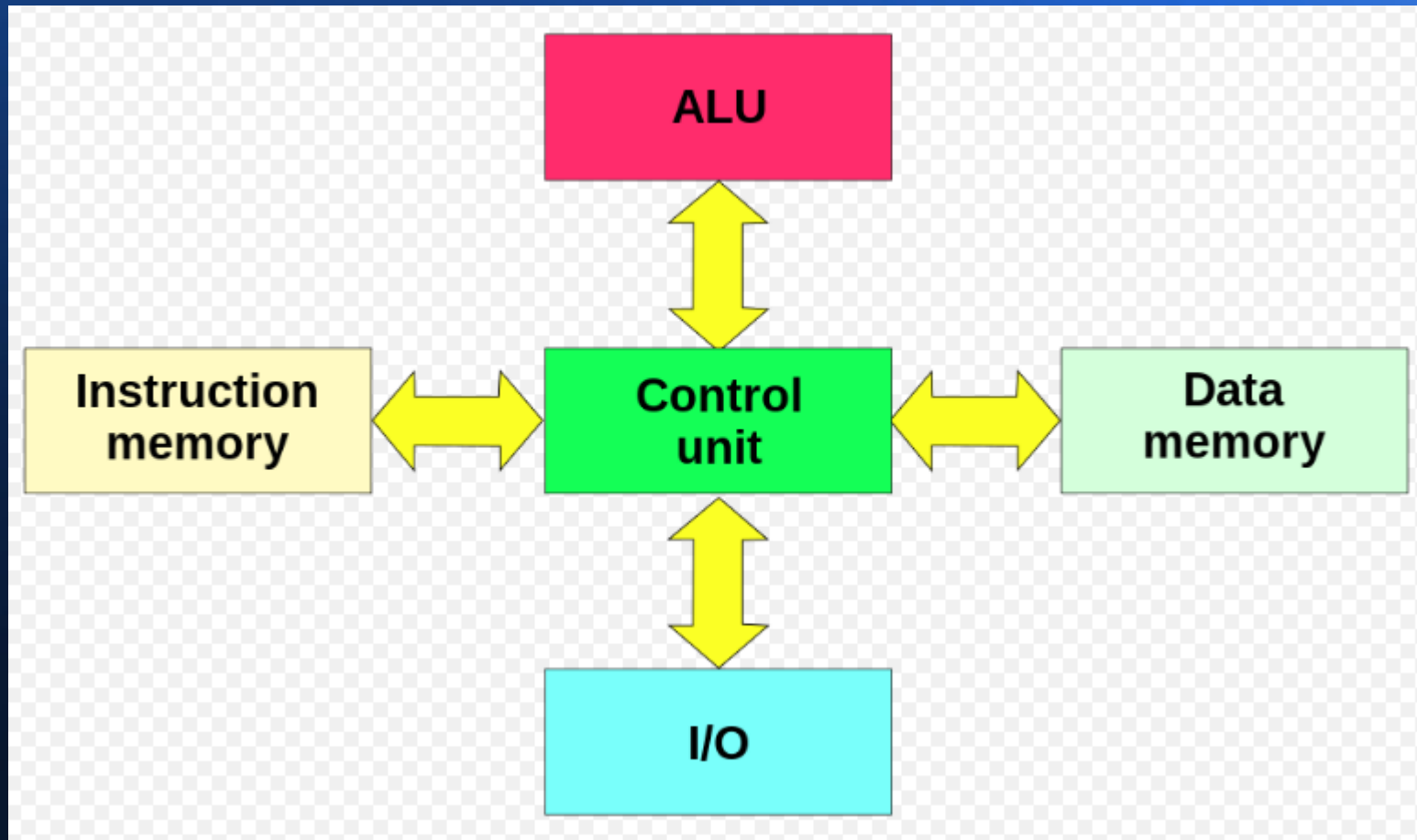
# Project 5: Computer Architecture

Chip (HDL)	Description
<i>Memory.hdl</i>	Entire RAM address space
<i>CPU.hdl</i>	The Hack CPU
<i>Computer.hdl</i>	The platform's top-most chip

# Von Neumann Architecture

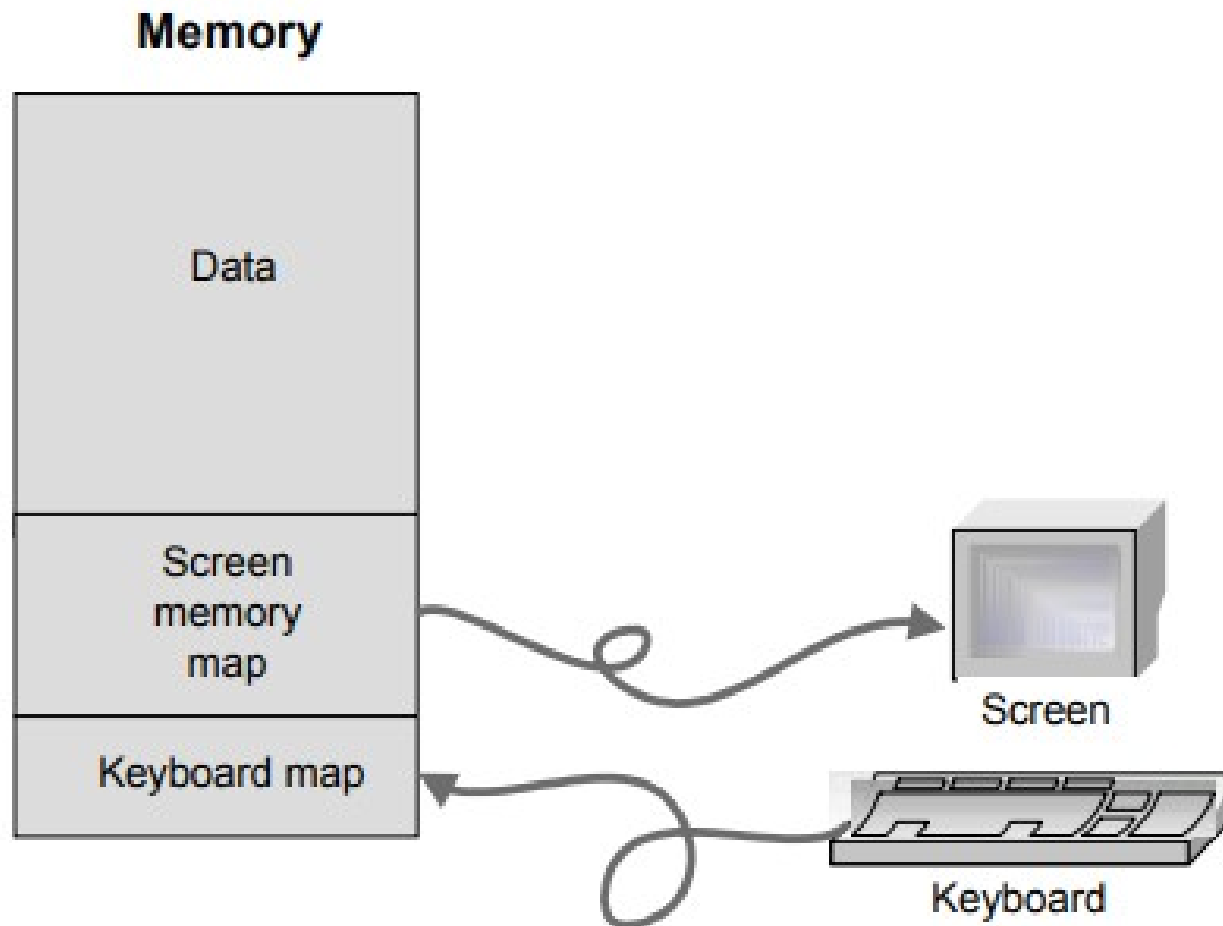


# Harvard architecture





# Memory (1)



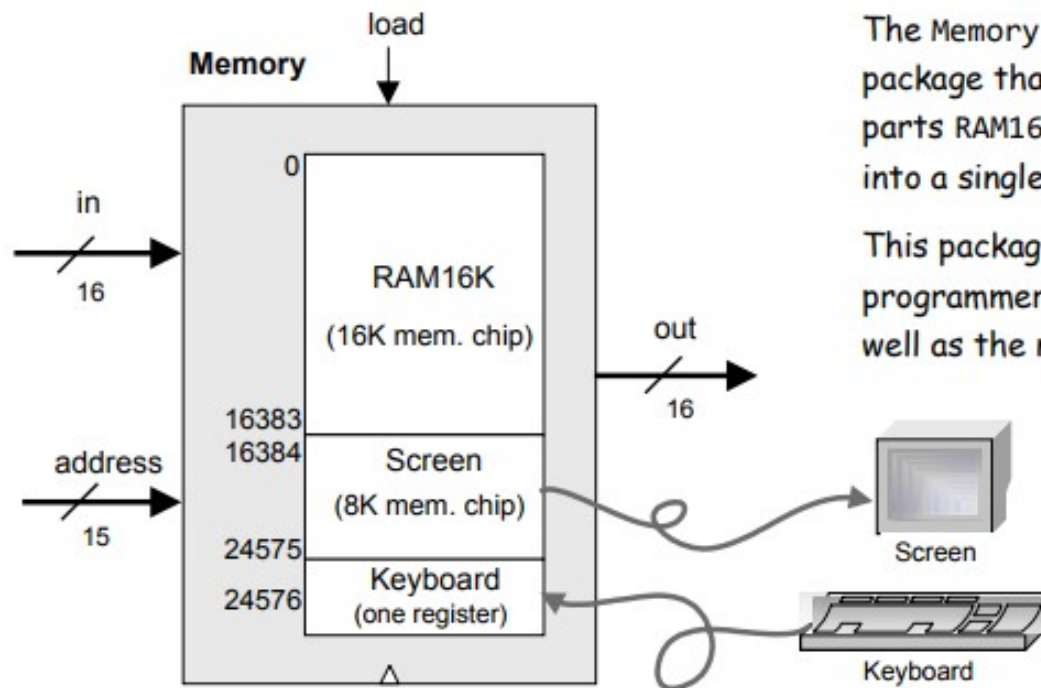
# Memory (2)

```
* Only the upper 16K+8K+1 words of the Memory chip are used.  
* Access to address>0x6000 is invalid. Access to any address in  
* the range 0x4000-0x5FFF results in accessing the screen memory  
* map. Access to address 0x6000 results in accessing the keyboard  
* memory map. The behavior in these addresses is described in the  
* Screen and Keyboard chip specifications given in the book.  
*/
```

```
CHIP Memory {  
    IN in[16], load, address[15];  
    OUT out[16];  
  
    PARTS:  
    // Put your code here:  
}
```

# Memory (3)

## Memory: physical implementation



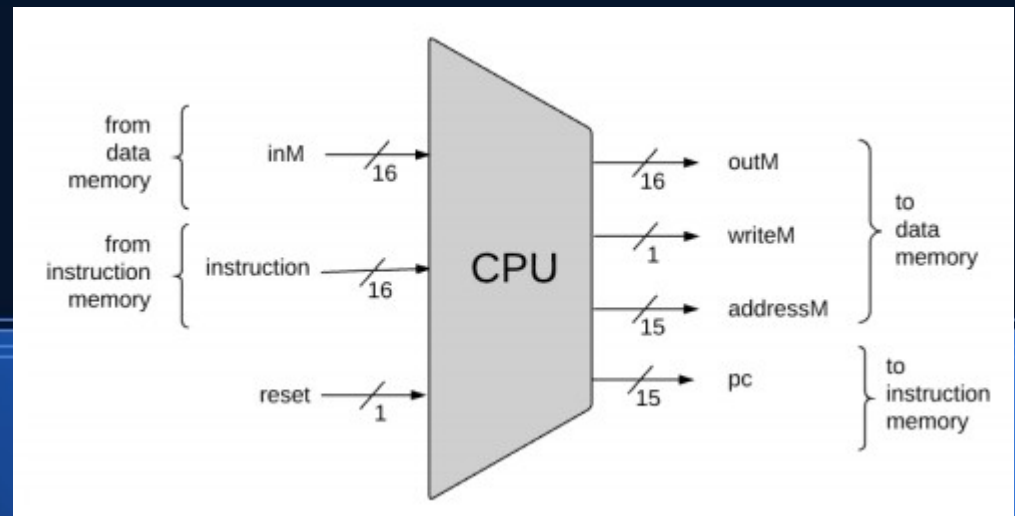
The Memory chip is essentially a package that integrates the three chip-parts RAM16K, Screen, and Keyboard into a single, contiguous address space.

This packaging effects the programmer's view of the memory, as well as the necessary I/O side-effects.

### Access logic:

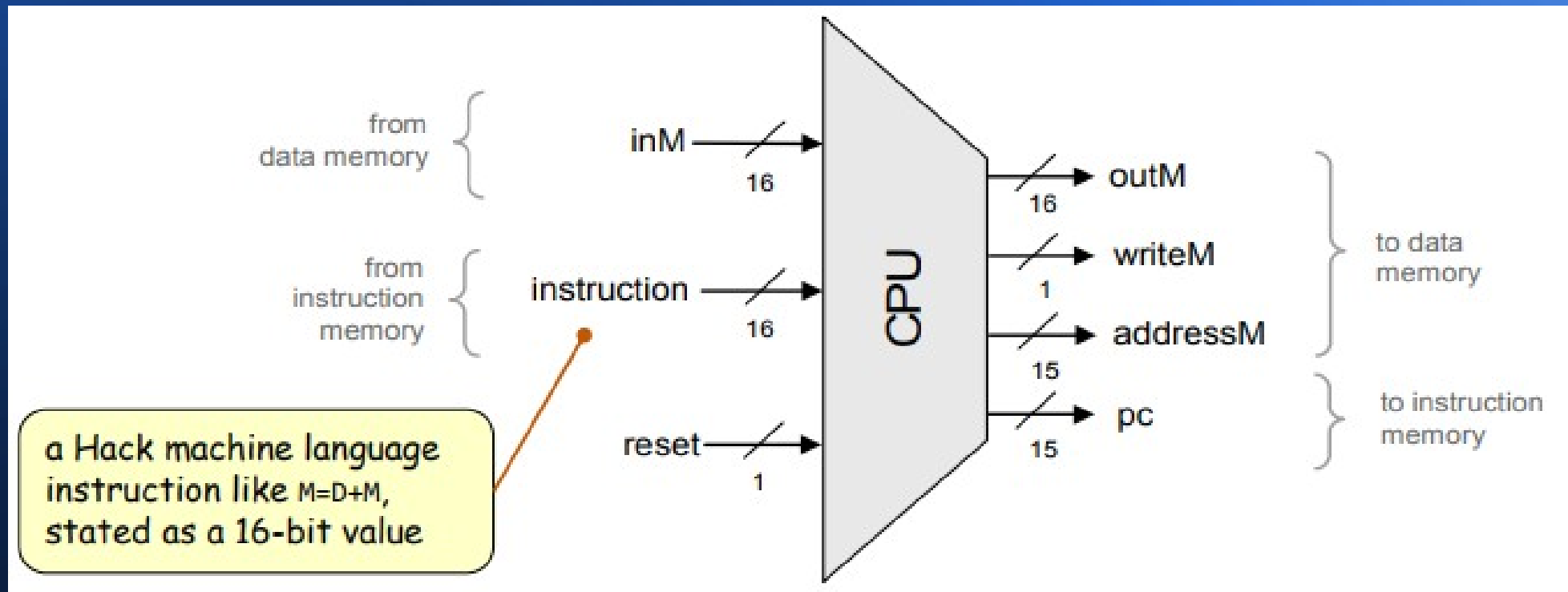
- ❑ Access to any address from 0 to 16,383 results in accessing the RAM16K chip-part
- ❑ Access to any address from 16,384 to 24,575 results in accessing the Screen chip-part
- ❑ Access to address 24,576 results in accessing the keyboard chip-part
- ❑ Access to any other address is invalid.

# CPU (1)



```
CHIP CPU {  
  
    IN  inM[16],           // M value input  (M = contents of RAM[A])  
        instruction[16],  // Instruction for execution  
        reset;           // Signals whether to re-start the current  
                          // program (reset==1) or continue executing  
                          // the current program (reset==0).  
  
    OUT outM[16],          // M value output  
        writeM,           // Write to M?  
        addressM[15],     // Address in data memory (of M)  
        pc[15];           // address of next instruction  
  
    PARTS:  
    // Put your code here:  
}
```

# CPU (2)



# CPU (3)

*dest = comp; jump*

*comp*

*dest*

*jump*

binary: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

(when a=0) <i>comp</i>	c1	c2	c3	c4	c5	c6	(when a=1) <i>comp</i>	d1	d2	d3	Mnemonic	Destination (where to store the computed value)
0	1	0	1	0	1	0		0	0	0	null	The value is not stored anywhere
1	1	1	1	1	1	1		0	0	1	M	Memory[A] (memory register addressed by A)
-1	1	1	1	0	1	0		0	1	0	D	D register
D	0	0	1	1	0	0		0	1	1	MD	Memory[A] and D register
A	1	1	0	0	0	0	M	1	0	0	A	A register
!D	0	0	1	1	0	1		1	0	1	AM	A register and Memory[A]
!A	1	1	0	0	0	1	!M	1	1	0	AD	A register and D register
-D	0	0	1	1	1	1		1	1	1	AMD	A register, Memory[A], and D register
-A	1	1	0	0	1	1	-M					
D+1	0	1	1	1	1	1						
A+1	1	1	0	1	1	1	M+1					
D-1	0	0	1	1	1	0						
A-1	1	1	0	0	1	0	M-1					
D+A	0	0	0	0	1	0	D+M					
D-A	0	1	0	0	1	1	D-M					
A-D	0	0	0	1	1	1	M-D					
D&A	0	0	0	0	0	0	D&M					
D A	0	1	0	1	0	1	D M					

j1 (out < 0)	j2 (out = 0)	j3 (out > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If out > 0 jump
0	1	0	JEQ	If out = 0 jump
0	1	1	JGE	If out ≥ 0 jump
1	0	0	JLT	If out < 0 jump
1	0	1	JNE	If out ≠ 0 jump
1	1	0	JLE	If out ≤ 0 jump
1	1	1	JMP	Jump

# CPU (4)

*dest = comp; jump*

binary:

1

1

1

a

*comp*

c1

c2

c3

c4

*dest*

c5

c6

d1

d2

*jump*

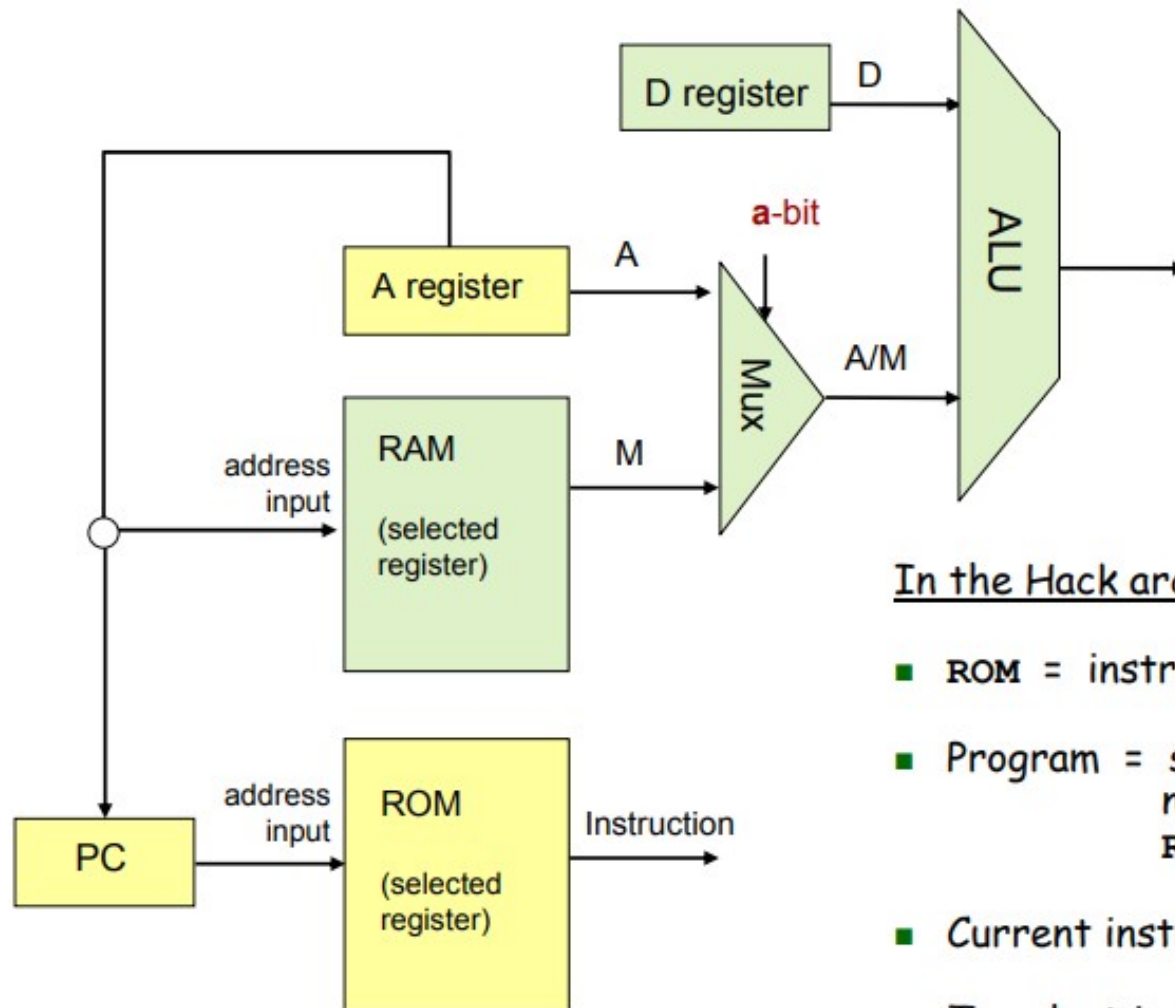
d3

j1

j2

j3

## Control (focus on the yellow chips only)



### In the Hack architecture:

- ROM = instruction memory
- Program = sequence of 16-bit numbers, starting at ROM[0]
- Current instruction = ROM[PC]
- To select instruction  $n$  from the ROM, we set A to  $n$ , using the instruction @ $n$

# CPU (4)

*dest = comp; jump*

*comp*

*dest*

*jump*

binary:

1

1

1

a

c1

c2

c3

c4

c5

c6

d1

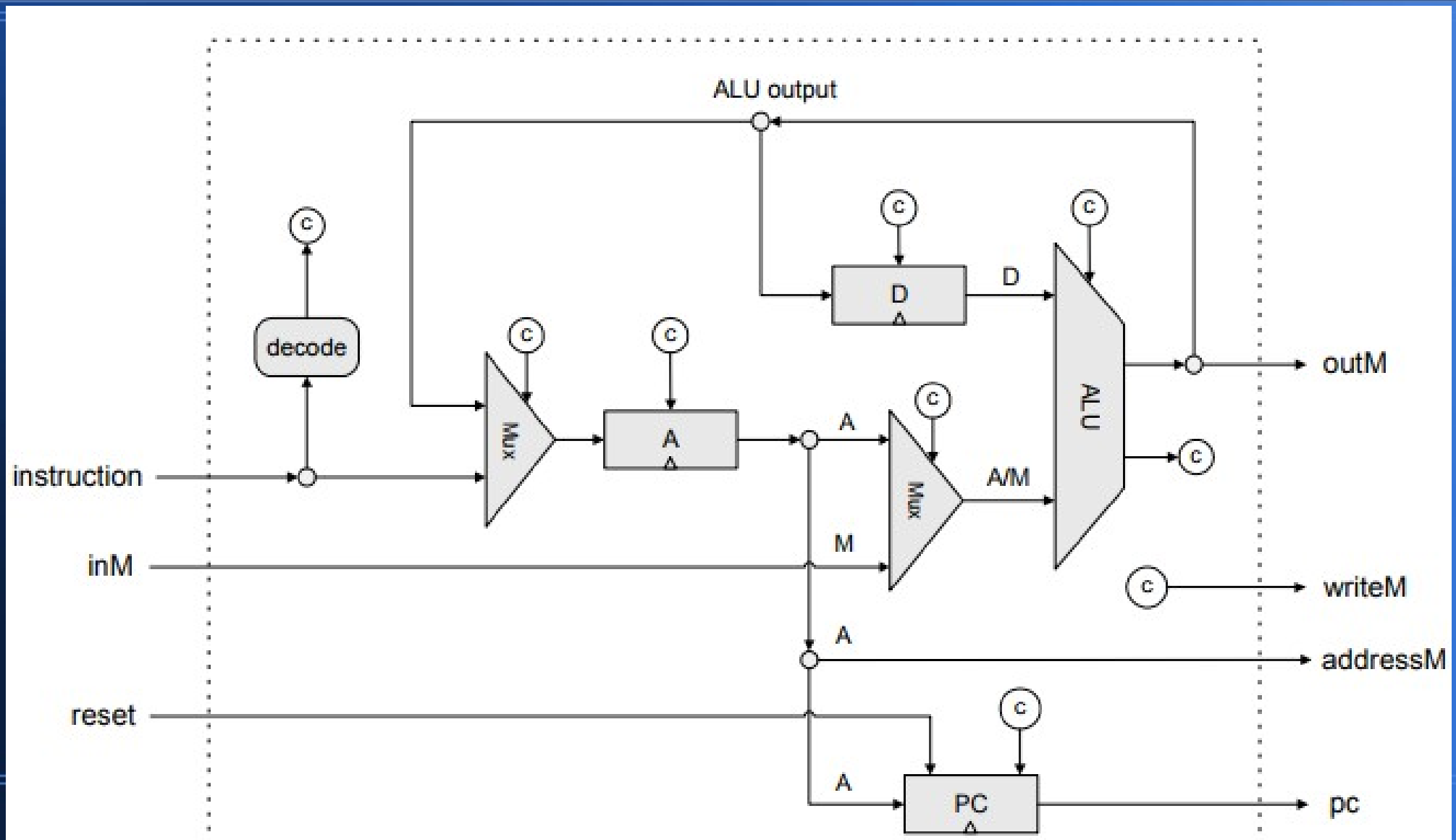
d2

d3

j1

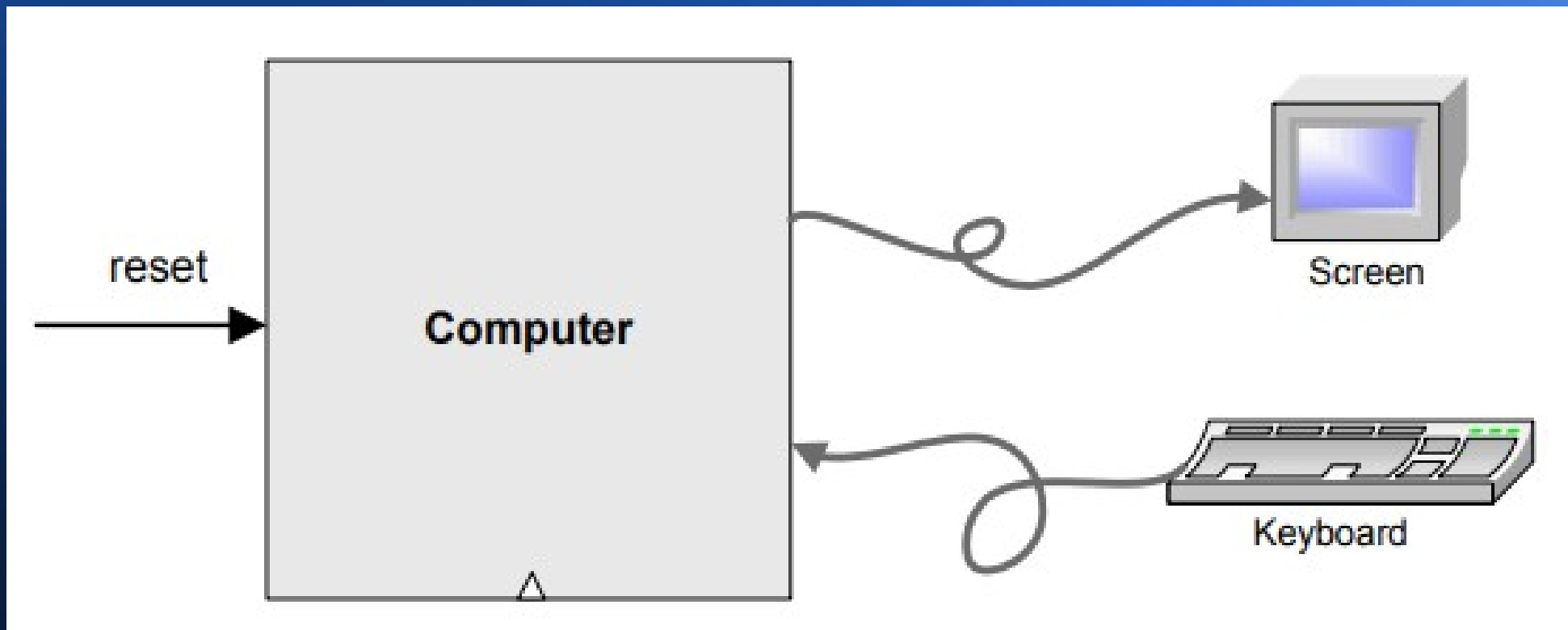
j2

j3

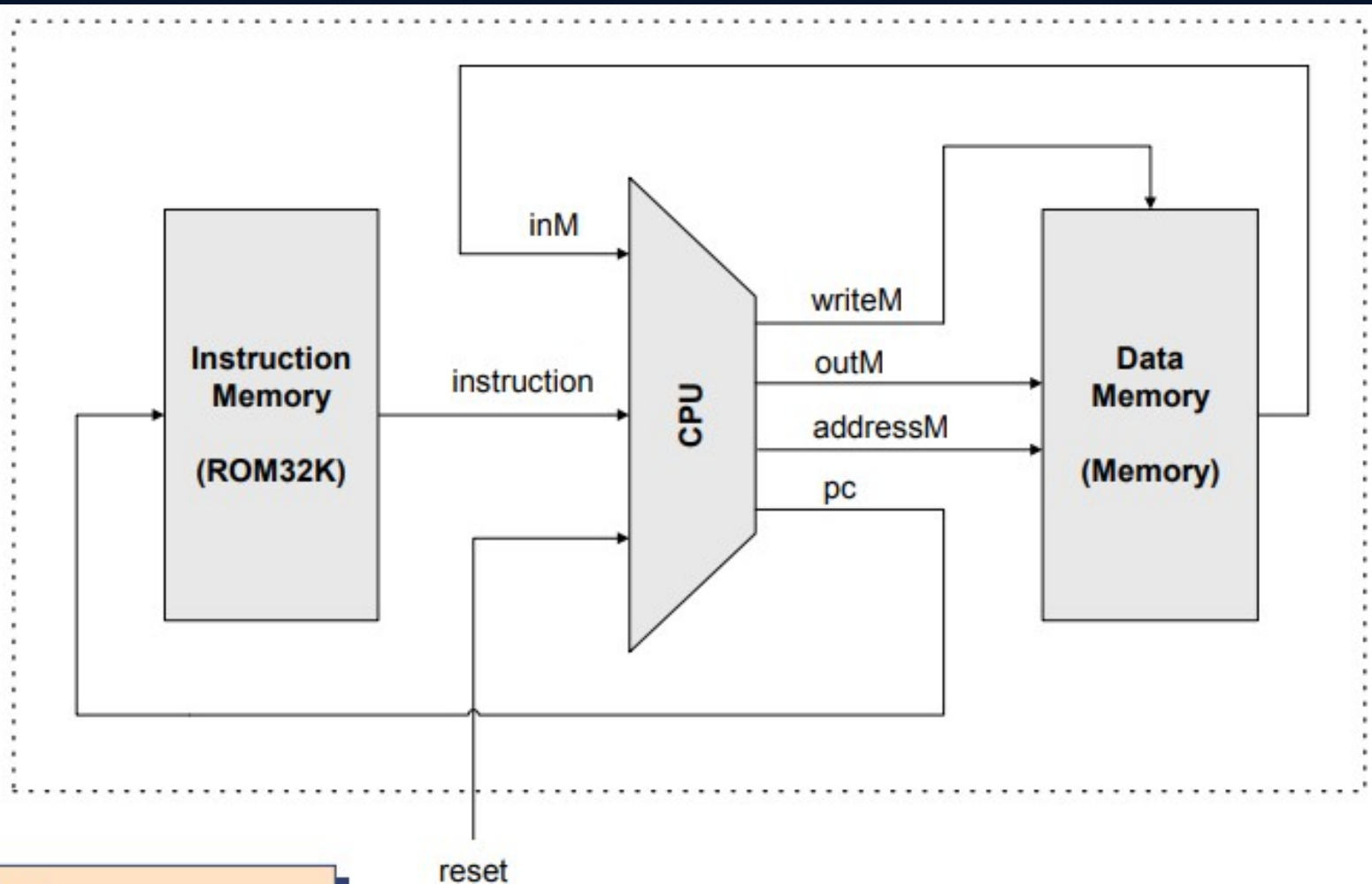




# Computer (1)



# Computer (2)



```
CHIP Computer {  
  IN reset;  
  PARTS:  
    // implementation missing  
}
```

## Implementation:

Simple, the chip-parts do all the hard work.

# Computer (3)


```
/**
 * The HACK computer, including CPU, ROM and RAM.
 * When reset is 0, the program stored in the computer's ROM executes.
 * When reset is 1, the execution of the program restarts.
 * Thus, to start a program's execution, reset must be pushed "up" (1)
 * and "down" (0). From this point onward the user is at the mercy of
 * the software. In particular, depending on the program's code, the
 * screen may show some output and the user may be able to interact
 * with the computer via the keyboard.
 */

CHIP Computer {

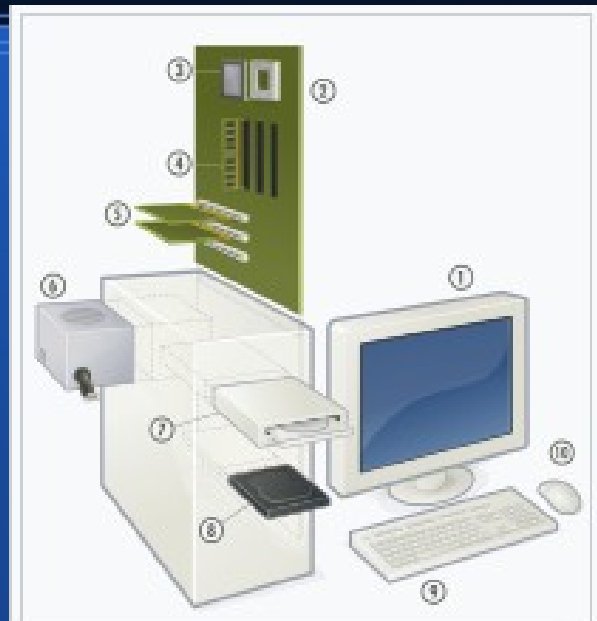
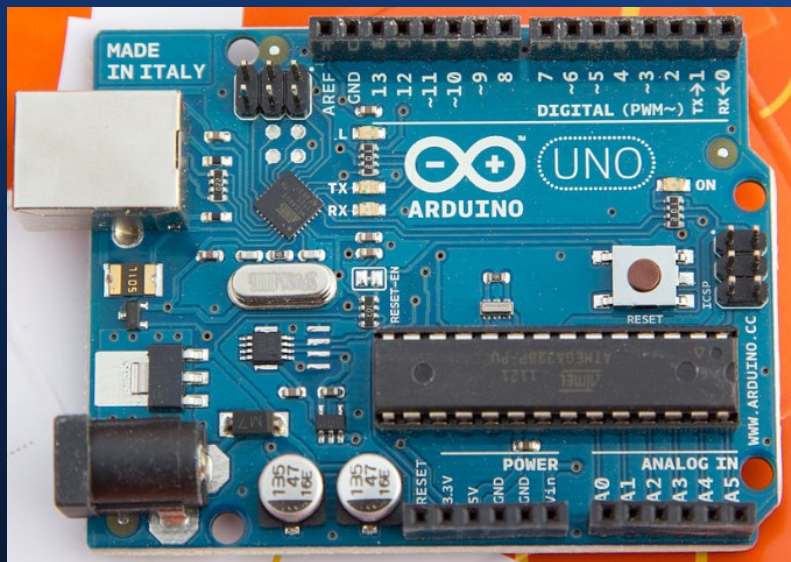
    IN reset;

    PARTS:
    // Put your code here:

}
```



# Computer (4)



個人電腦的主要結構：

1. 螢幕
2. 主機板
3. 中央處理器 (微處理器)
4. 記憶體
5. 介面卡 (如音效卡、網路卡)
6. 電源供應器
7. 軟碟機 / 光碟機
8. 硬碟
9. 鍵盤
10. 滑鼠

# It's your turn

- Do it yourself
- You can make it !