Nand2Tetris - The Elements of Computing Systems

Noam Nisan and Shimon Schocken

16th September 2013

Abstract

The book's software suite:

Simulators *HardSimulator, CPUEmulator, VMEmulator.*
(supplied: build hardware platforms and execute programs)

Translators *Assembler, Jack Compiler.*
(I built but also supplied: translate from high-level to low-level)

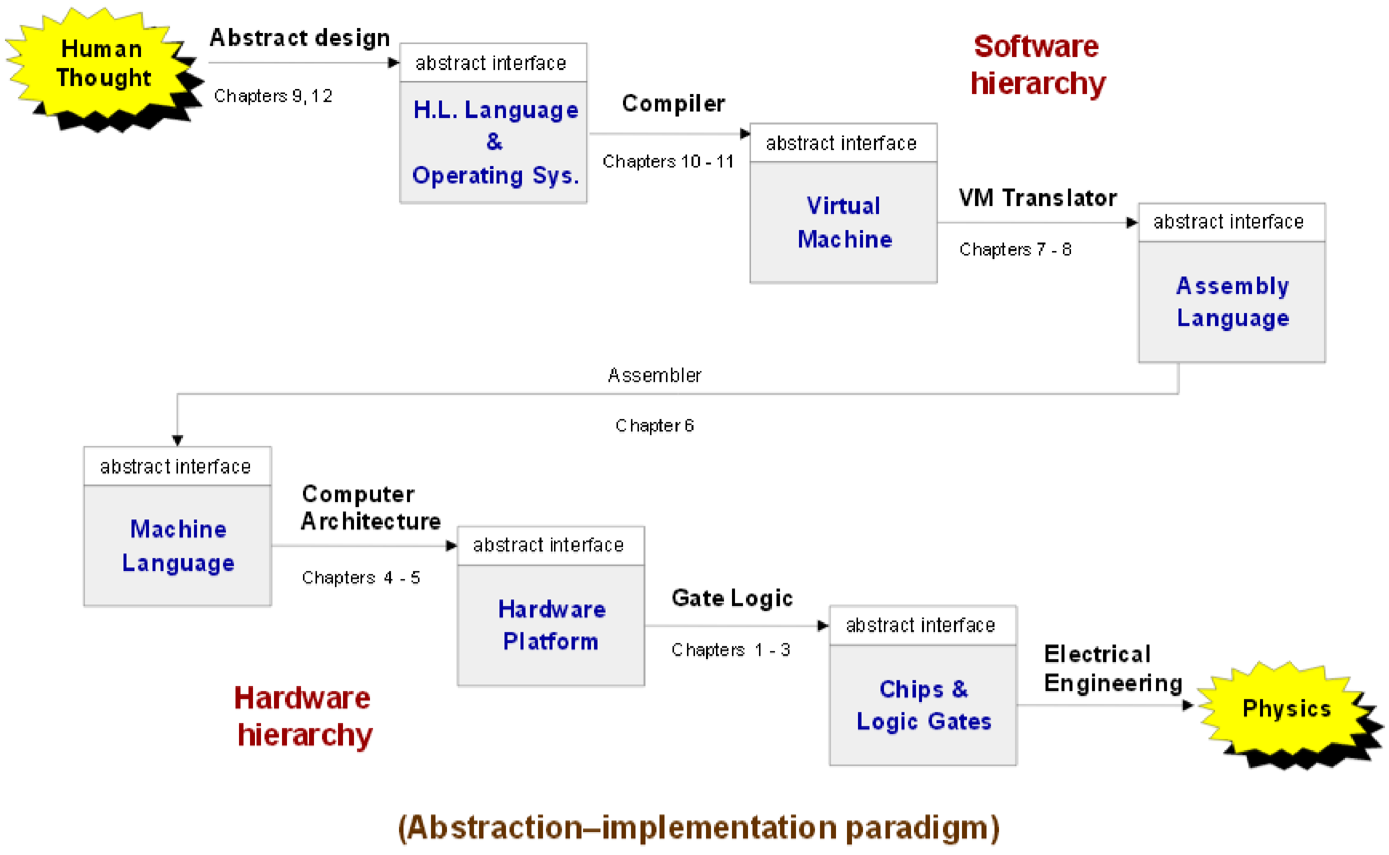


Figure 0.1: Course theme and structure.

Contents

1 Boolean Logic

1.1 Background

1.1.1 Boolean (Binary) Algebra [p8]

Boolean functions operate on binary inputs, returning binary outputs; use *truth tables* to represent functions.

Notation	Boolean Expression
$x \cdot y = xy$	x AND y
$x + y$	x OR y
$\overline{x}$	NOT x

Table 1.1: Boolean expressions created from the Boolean operators "AND", "OR" and "NOT".

Boolean Expressions

Canonical Representation *Every Boolean function can be expressed using one Boolean expression.*
Method: (see E1.1)

1. Mark every row for which the function has value 1.
2. AND together the *literals* (the variable itself or it's negation) for each of these rows.
3. OR these terms.

Corollary 1.1. *Every Boolean function can be expressed using three Boolean operators AND, OR, NOT.*

Note. The number of Boolean functions that can be defined over n binary variable is 2^{2^n}.

Example (E1.1)**.** Consider the Boolean Expression $f(x, y, z) = (x + y) \cdot \overline{z}$.

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

Canonical Form: $f(x, y, z) = \overline{x}y\overline{z} + x\overline{y}\overline{z} + xy\overline{z}$. Observe can be factored further, yet remains clearer regarding the truth table in this form.

Remark. The NAND function can construct each one of the AND, OR, and NOT operations, and since these three operations can express every Boolean expression it follows:

Theorem 1.2. *Every Boolean function can be constructed from NAND alone.*

Operation	Construction from NAND
AND	Nand (Nand (x, y) , Nand (x, y))
OR	Nand (Nand (x, x) , Nand (y, y))
NOT	Nand (x, x)
NOR	"We've already created NOT and OR." NOT (OR (x, y))
XOR	Similarly: OR (AND (x, NOT (y)) , AND (NOT (x) , y))

1.1.2 Gate Logic [p11]

Chip/Gate A physical device that implements a Boolean function.
If a Boolean function f operates on n variables and returns m binary results, the gate that implements f will have n input pins and m output pins.

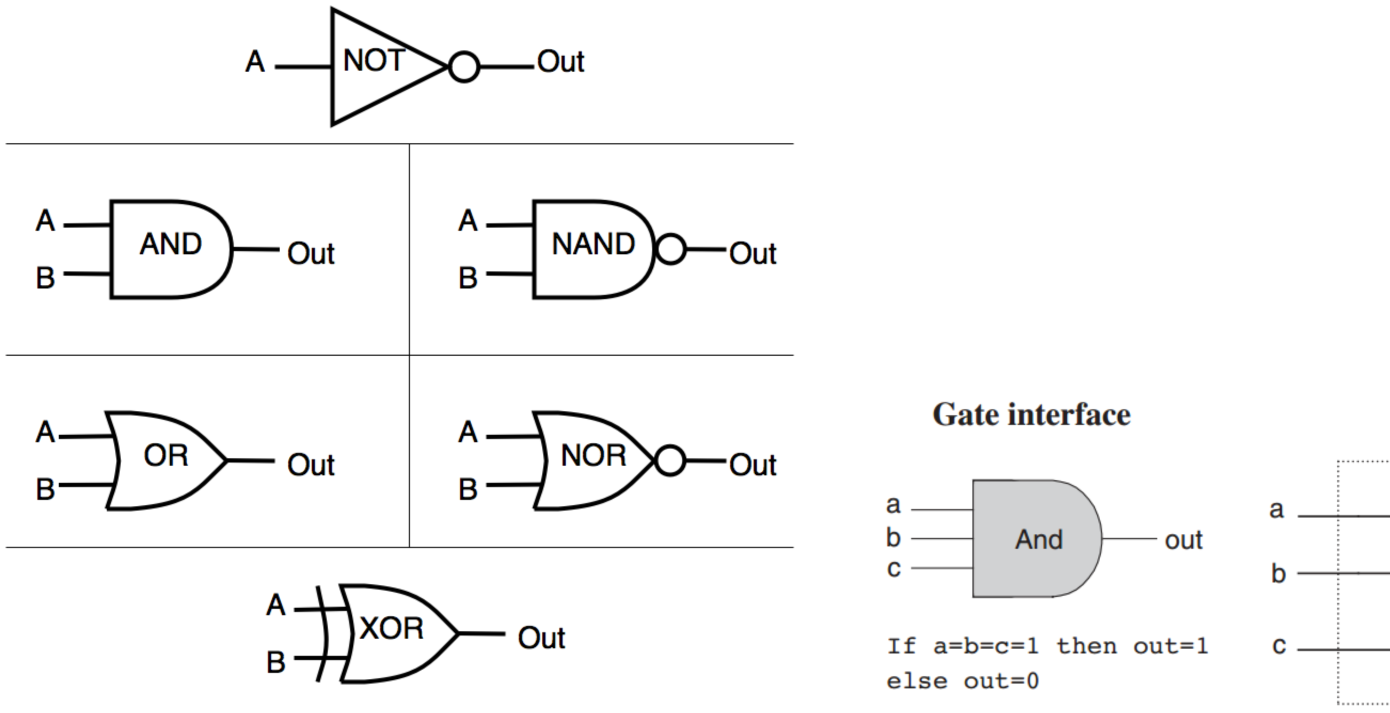


Figure 1.1: Left: Standard logic gate symbols. Right: **WHAT** a three-way AND gate is doing, and **HOW** it operates. Observe *the gate interface is unique,* whilst *there exists many different implementations*. The art of logic design is: *Given a gate specification, find an efficient way to implement it using other gates that were **already** implemented.*

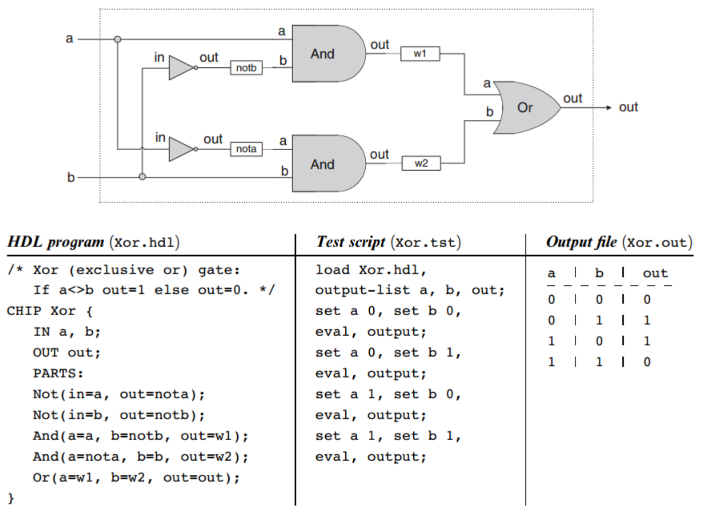
1.1.3 Actual Hardware Construction [p13]

Gets Messy!

1.1.4 (Virtual) Hardware Description Language, (V)HDL [p14]

- Hardware designers specify the chip's structure by writing an *HDL program* (c.f. a programming language).
- The designs are then subjected to rigorous testing using a *hardware simulator* (c.f. a compiler).

Example (E1.2).



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
```

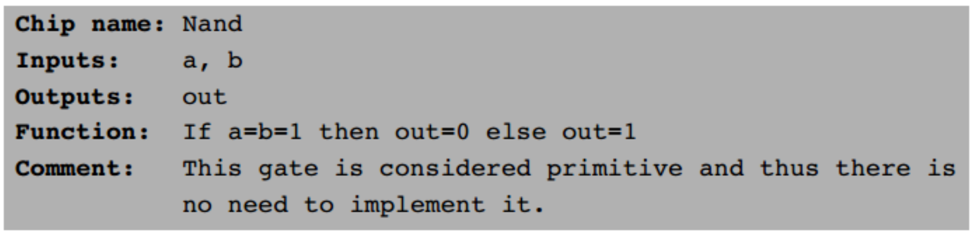
Figure 1.2: Designing a XOR gate using HDL, with test and output file.

1.2 Specification

1.2.1 The NAND Gate [p19]

a	b	NAND (a, b)
0	0	1
0	1	1
1	0	1
1	1	0

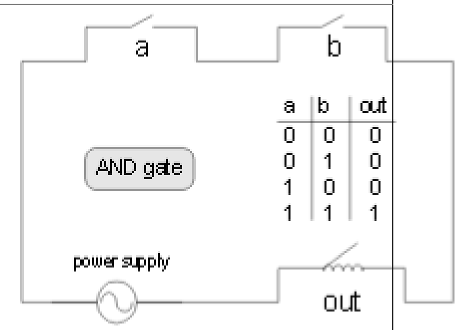
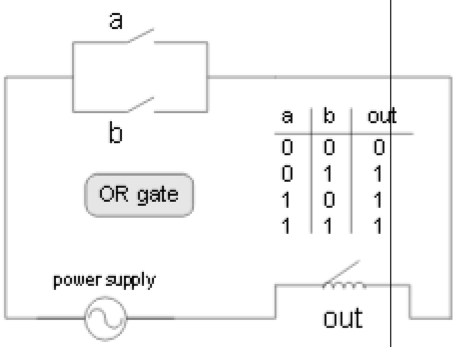
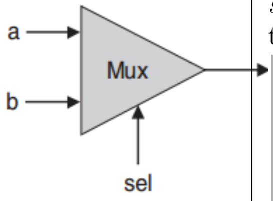
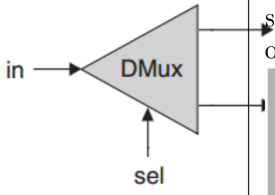
Remark. Throughout this course, we use *chip API boxes* to specify chips:



```
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.
```

Figure 1.3: *Chip API box* for a NAND chip.

1.2.2 Basic Logic Gates [p19]

Gate	Chip API Box	Circuit Implementations
NOT	**Chip name:** Not **Inputs:** in **Outputs:** out **Function:** If in=0 then out=1 else out=0.	
AND	**Chip name:** And **Inputs:** a, b **Outputs:** out **Function:** If a=b=1 then out=1 else out=0	 a b AND gate power supply out a b out 0 0 0 0 1 0 1 0 0 1 1 1
OR	**Chip name:** Or **Inputs:** a, b **Outputs:** out **Function:** If a=b=0 then out=0 else out=1	 a b OR gate power supply out a b out 0 0 0 0 1 1 1 0 1 1 1 1
XOR	**Chip name:** Xor **Inputs:** a, b **Outputs:** out **Function:** If a≠b then out=1 else out=0.	
MULTIPLEXOR 	A three-input gate that uses one of the inputs, *selection bit* to select and output one of the other two inputs, *data bits.* **Chip name:** Mux **Inputs:** a, b, sel **Outputs:** out **Function:** If sel=0 then out=a else out=b.	
DEMULTIPLEXOR 	The opposite function of a multiplexor: takes a single input and channels it to one of two possible outputs, specified by a selector bit. **Chip name:** DMux **Inputs:** in, sel **Outputs:** a, b **Function:** If sel=0 then {a=in, b=0} else {a=0, b=in}.	

1.2.3 Multi-Bit Versions of Basic Gates [p21]

- Computer hardware is typically designed to operate on multi-bit arrays called *buses.*
- When referring to individual bits in a bus (suppose we are dealing with a 16-bit bus named `data`), it is common to use the array syntax `data[0]`, `data[1]`, ..., `data[15]`.

Remark (Backwards Conveyor Belt). Arrays are specified from right-to-left in the HDL used.

n-bit Gate	Chip API Box (16-bit gate)
MULTI-BIT NOT	**Chip name:** Not16 **Inputs:** in[16] // a 16-bit pin **Outputs:** out[16] **Function:** For i=0..15 out[i]=Not(in[i]).
MULTI-BIT AND	**Chip name:** And16 **Inputs:** a[16], b[16] **Outputs:** out[16] **Function:** For i=0..15 out[i]=And(a[i],b[i]).
MULTI-BIT OR	**Chip name:** Or16 **Inputs:** a[16], b[16] **Outputs:** out[16] **Function:** For i=0..15 out[i]=Or(a[i],b[i]).
MULTI-BIT MULTIPLEXOR	**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].

1.2.4 Multi-Way Versions of Basic Gates [p23]

- **Idea:** Evaluating more than 2 inputs.

n-way Gate	Chip API Box (8-way gate)
MULTI-BIT OR	**Chip name:** Or8Way **Inputs:** in[8] **Outputs:** out **Function:** out=Or(in[0],in[1],...,in[7]).
MULTI-WAY/MULTI-BIT MULTIPLEXOR	An m-way, n-bit multiplexor selects one of m n-bit input buses and outputs it to a single n-bit output bus. The selection is specified by a set of $k = \log_2 m$ control bits. **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]". **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]".

MULTI-WAY/MULTI-BIT DEMULTIPLEXOR	An m-way n-bit demultiplexor channels a single n-bit input into one of m possible n-bit outputs. The selection is specified by a set of k control bits, where $k = \log_2 m$.
	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}.
	
	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}.

1.3 Implementation [p25]

Primitive Gates provide a set of elementary building blocks form which everything else can be built.

Remark (C.f. Axioms in mathematics). We use NAND as our basic building block, yet other ones are possible (e.g. NOR, or a combination of AND,OR and NOT); just as all theorems in geometry can be founded on different sets of axioms.

1.4 Project [p27]

1.5 Perspective [p26]

2 Boolean Arithmetic [p29]

2.1 Background [p30]

2.1.1 Binary numbers and addition [p30]

When we press the keyboard keys labelled **1**, **9** and **Enter**, the equivalent 32-bit binary code (if we are working on a 32-bit machine) **00000000000000000000000000010011** ends up in the register of the computer's memory.

LSB (Least Significant Bits) the right-most digits of a binary number.

MSB (Most Significant Bits) the left-most digits of a binary number.

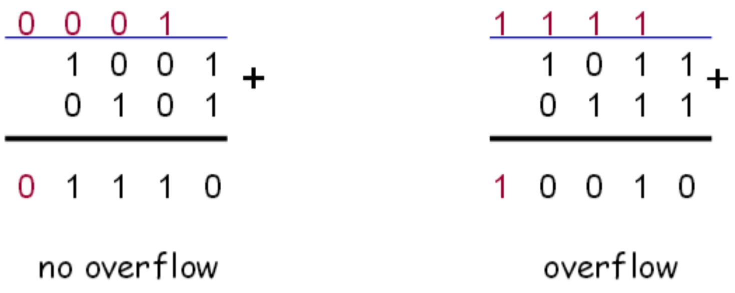


Figure 2.1: *Add digit by digit from right to left.* Observe, computer hardware for binary addition of 2 n-bit numbers can be built from logic gates designed to calculate the **sum of 3 bits** (pair of bits plus carry bit) - hence the **full adder**.

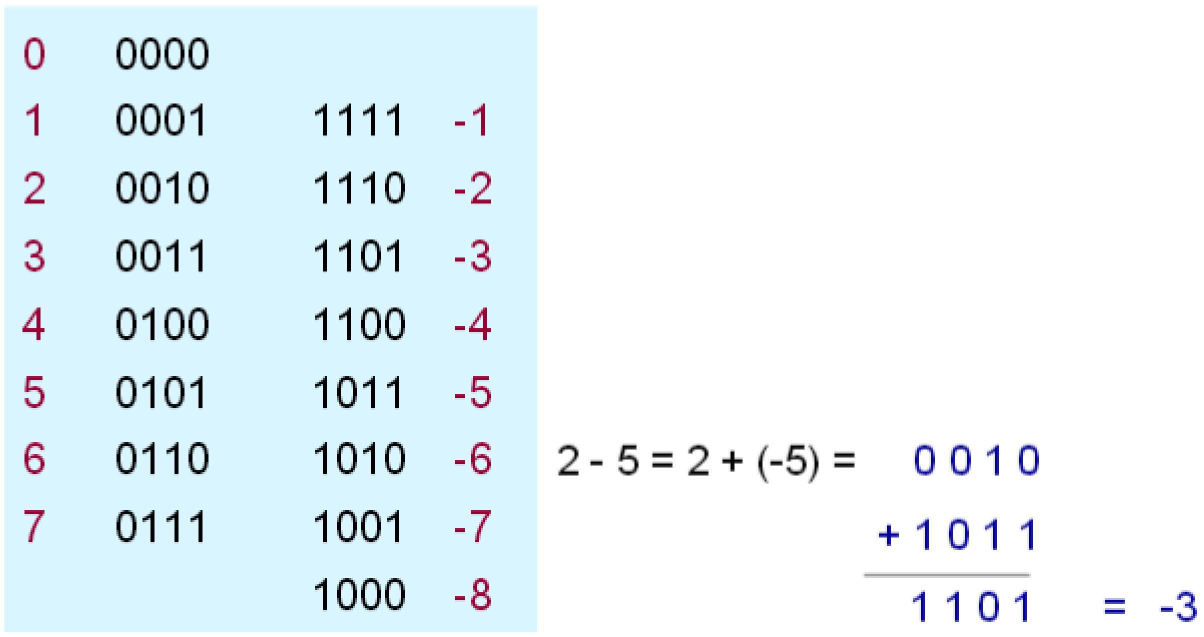
2.1.2 Signed Binary Numbers [p31]

A binary system with n digits can generate a set of 2^n different bit patterns; hence if we need to represent positive and negative numbers, we spilt these arrangements into 2 equal subsets (one for the positive numbers, the other for the negative numbers).

Definition 2.1. The *2's (or radix) complement method* of a number x is

$$\overline{x} = \begin{cases} 2^n - x & \text{if } x \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

This is equivalent to *inverting each digit and adding 1.*



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

$$2 - 5 = 2 + (-5) = \begin{array}{r} 0\,0\,1\,0 \\ +\,1\,0\,1\,1 \\ \hline 1\,1\,0\,1 \end{array} = -3$$

Figure 2.2: 2's complement representation of signed number in a 4-bit binary system. Observe the total number of numbers represent is 2^n, with 2^{n-1} is each subset. Also, the addition of a number an its inverse is 0000 (e.g. $1 + (-1) = 0001 + 1111 = (1)\,0000$, where the leading 1 is omitted because we're working in a 4-bit binary system. This representation of negative numbers makes subtraction very easy - as shown right. We conclude that all basic arithmetic and logical operators can be perform by a single chip (**ALU**).

Example (E2.1)**.**

ALU (Arithmetic Logical Unit) the centrepiece chip or the CPU (which is the centrepiece of a computer) that executes all arithmetic and logical operations.

Remark. All positive number begin with **0**, and all negative numbers begin with **1**.

2.2 Specification [p32]

One such ALU is the *adder* chip.

2.2.1 Adders

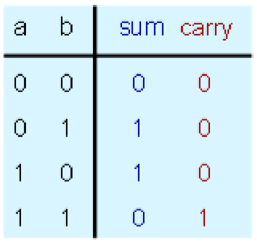
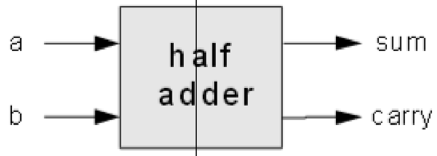
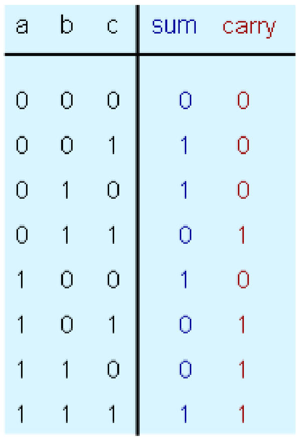
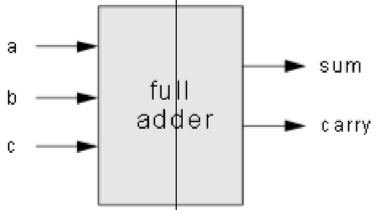
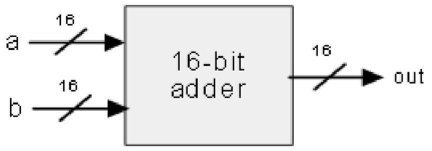
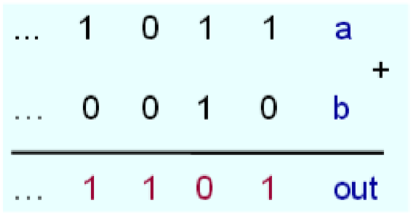
Adder	Implementation
Half-adder: designed to add 2 bits. Based on the XOR and AND gates	 
Full-adder: designed to add 3 bits. Can be based on half-adder gates.	 
Multi-bit Adder: designed to add two n-bit numbers ($n \in \{16, 32, 64, ...\}$. Array of full-adder gates.	  16-bit adder

Table 2.1: Hierarchy of three adders.

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

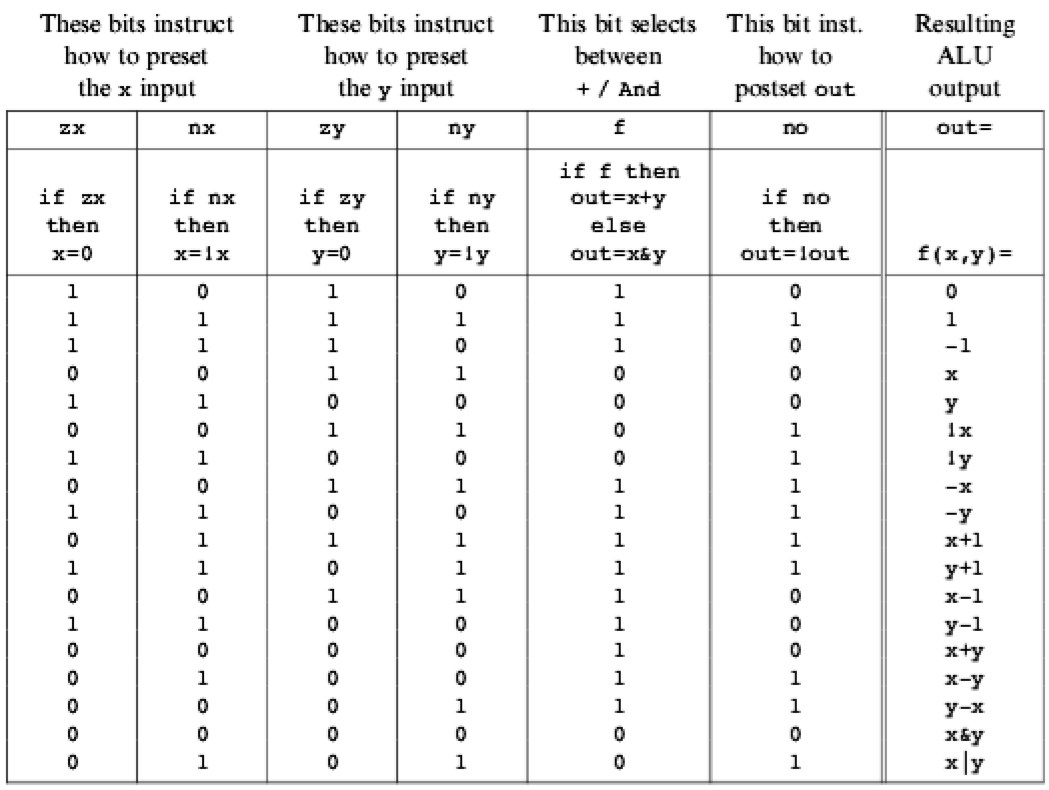
a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
...  1  0  1  1   a
                    +
...  0  0  1  0   b
-------------------
...  1  1  0  1   out
```

Incrementer: it is convenient to have a chip dedicated to adding the constant 1 to a given number (e.g. for calculating negative numbers).

2.2.2 The Arithmetic Logic Unit (ALU) [p35]

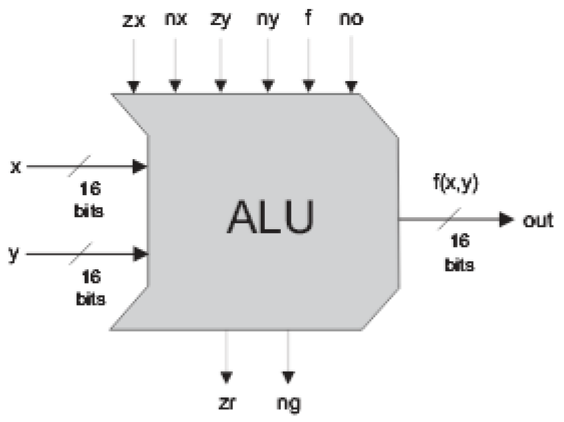
This subsubsection describes an ALU that will become the centerpiece of our computer platform *Hack*. The Hack ALU computes a fixed set of function $out = f_i(x, y)$ where x, y are two 16-bit inputs, out a 16-bit ouptut and f_i is an arithmetic or logical function selected from a fixed set of eighteen possible functions:



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 2.3: The ALU truth table working with **16-bit** inputs/output (so if $zy = 1$, y would zeroed $y = (000...00)_2$ and in general $0 = (000...00)_2$, $1 = (111...11)_2$; note !=Not, &=And and |=Or (performed **bit-wise**). We designed the ALU by defining which functions were desired and worked backwards to figure out how x, y and *out* can be manipulated by binary operations to achieve these results. We have included the 6 control bits, each using a straightforward binary operation.

Remark. We instruct the ALU which function to compute by setting six input bits, called *control bits*; hence we have $2^6 = 64$ different functions (the function can either be included or not). 18 are of interest to us.



```
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.
```

Figure 2.4: ALU specification (not particular efficient; we have chose to specify an ALU hardware with limited functionality and implement as many operations as possible in software - operating system).

Note. The overall functionally of the hardware/software platform is delivered jointly by the **ALU** and the **operating system**.

3 Sequential Logic [p41]

3.1 Background [p42]

3.1.1 Sequential VS. Combinational Logic

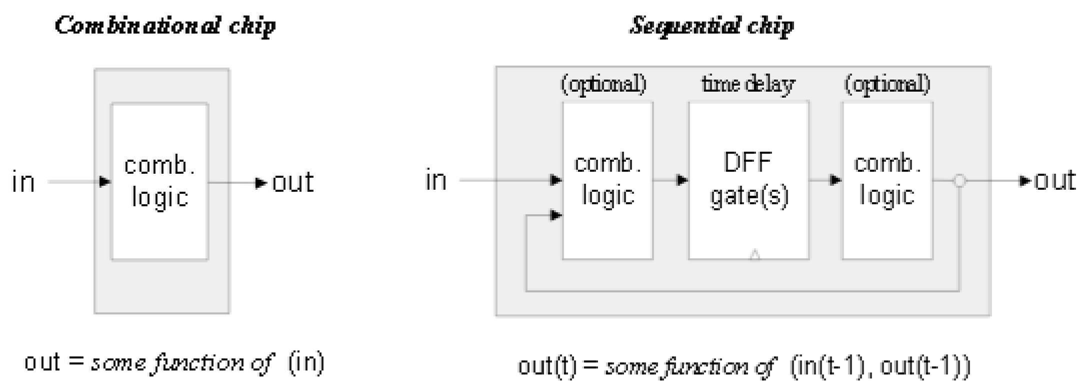
All the chips discussed and built so far are combinational chips:

Combinational devices operate on data only; compute functions that depend solely on *combinations of their values.* This type of chip cannot store and recall values.

Sequential devices operate on data and a clock signal; can be *state-aware* and provide storage and synchronisation services.

The low-level behaviour of sequential gates is **tricky**; although

Theorem 3.1. *Every sequential chip can be based upon the* ***"data flip flop"*** *or* ***DFF*** *sequential gate.*



3.1.2 Memory [p42]

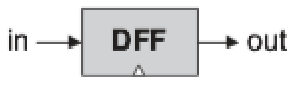
The act of "remembering something" is *time-dependent* (you remember now what has been committed to memory before). So for chip to "remember", we require s means for representing the progression of time.

Hierarchy of memory chips:

- Flip-flop gates.
- Binary cells.
- Registers.
- RAM.

The Clock Most computers have a master clock that delivers a continuous train of alternating signals between two phases labelled "0" and "1". The elapsed time between "0" and "1" is the *cycle*, and each clock cycle represent on discrete time unit.

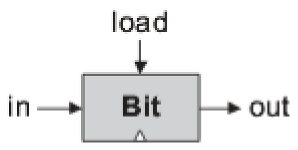
Flip-Flops We consider a variant of a flip-flop called the *data flip-flop*, whose interface consists of a single-bit data input/output, and a *clock* (continuously changing) according to the master clock's signal: $out(t) = in(t-1)$ where in/out are the input/output values and t is the current clock cycle.



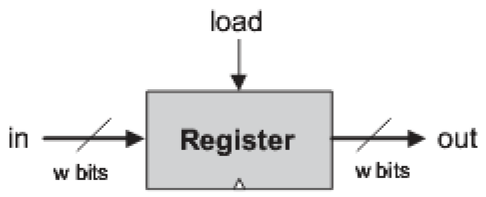
```
Chip name: DFF
Inputs:    in
Outputs:   out
Function:  out(t)=in(t-1)
Comment:   This clocked gate has a built-in
           implementation and thus there is
           no need to implement it.
```

Figure 3.1: API for a *data flip-flop* gate; c.f. Nand gate regarding primitive nature.

Registers a storage device that can "remember" a value over time: $out\,(t) = out\,(t-1)$; note, a DFF can only output its previous input (i.e. $out\,(t) = in\,(t-1)$).



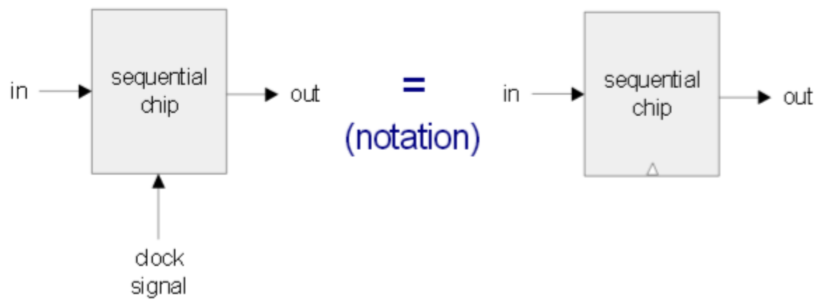
```
Chip name: Bit
Inputs:    in, load
Outputs:   out
Function:  If load(t-1) then out(t)=in(t-1)
           else out(t)=out(t-1)
```



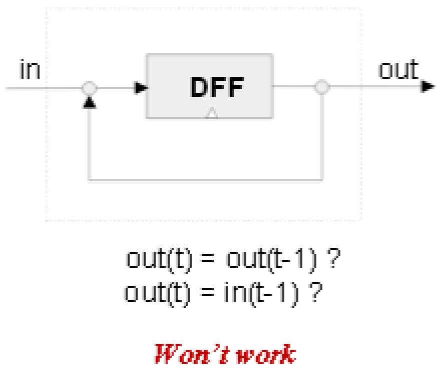
```
Chip name: Register
Inputs:    in[16], load
Outputs:   out[16]
Function:  If load(t-1) then out(t)=in(t-1)
           else out(t)=out(t-1)
Comment:   "=" is a 16-bit operation.
```

Figure 3.2: Top: API for a single-bit register (a *Bit*, or *binary cell*) chip.
Bottom: API for a 16-bit register: to **read**, we identify the output; to **write**, we write a new data value $in = d$ into the register and assert (set to 1) the *load* input, and in the next clock cycle the register commits to the new data (outputting d).

Notation 3.2. Represent the clock signal of a sequential chip:



Remark (WARNING:). The following 1-bit register description from a DFF is invalid:



It is not clear how we'll never load new data into this because should we draw input from the *in* or *out* wire? More generally, chips design dictates that *internal pins must have a fan-in of 1*.

Once we have the mechanism for remembering a single bit with time, we construct n-bit wide registers by forming an array of as many single-bit registers as needed.

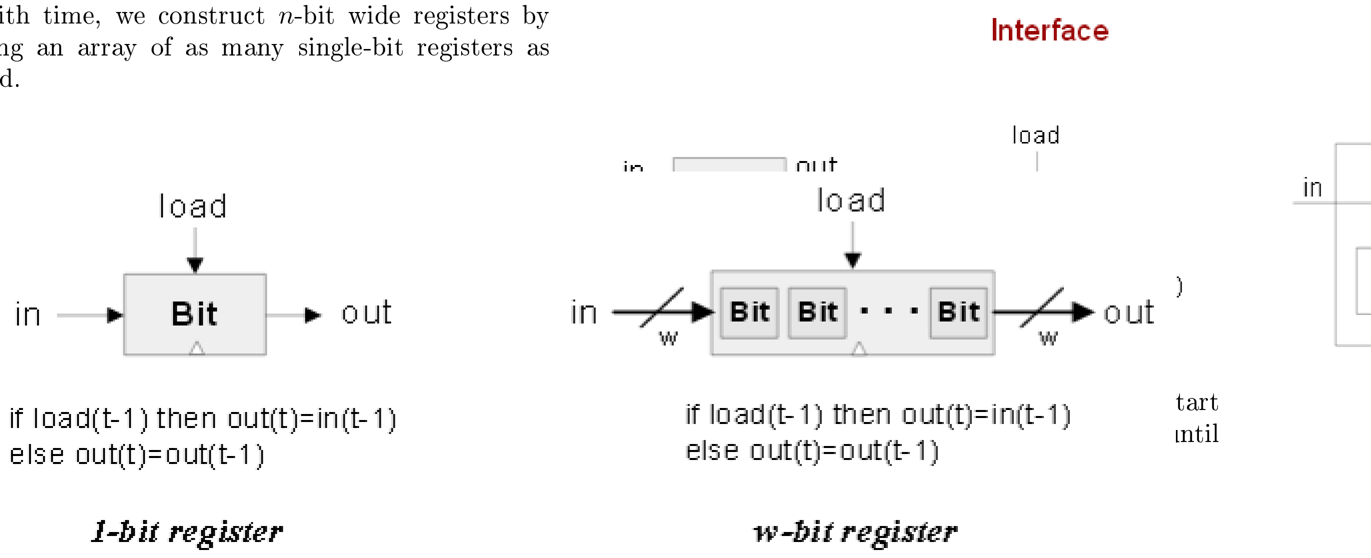


Figure 3.4: From single-bit to multi-bit registers.

The *width* of a register is the number of bits it holds (e.g. 16,32,64, ...). The multi-bit contents of a register are the *words*.

Memories Stacking many register form a *Random Access Memory (RAM)* unit; this name is derived from the requirement that read/write operations of any randomly chosen word on RAM should be accessed directly (in equal speed, irrespective of it physical location). Hence, a classical RAM device accepts three inputs: *data*, *address*, and *load*. *Address* specifies which RAM register to access in the current time unit. If $load = 0$ (a read operation) the RAM's $out = value\,of\,selected\,register$; otherwise (a write operation, $load = 1$) the selected memory register commits to the input value.

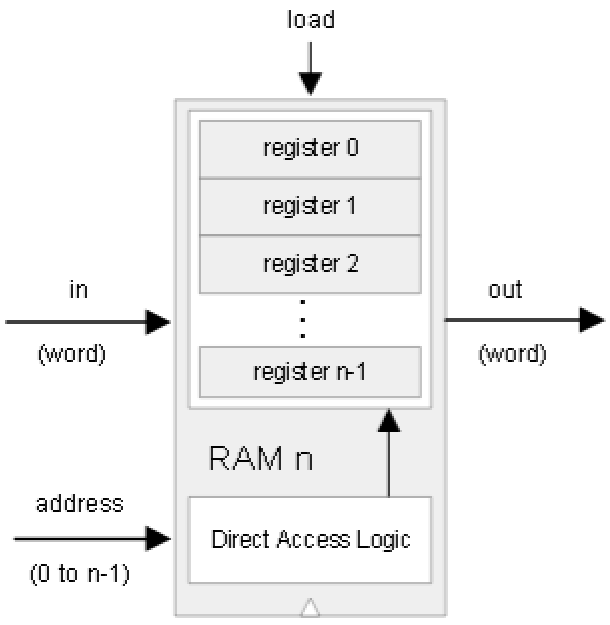
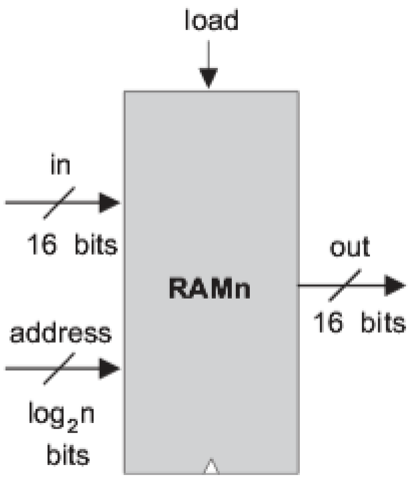
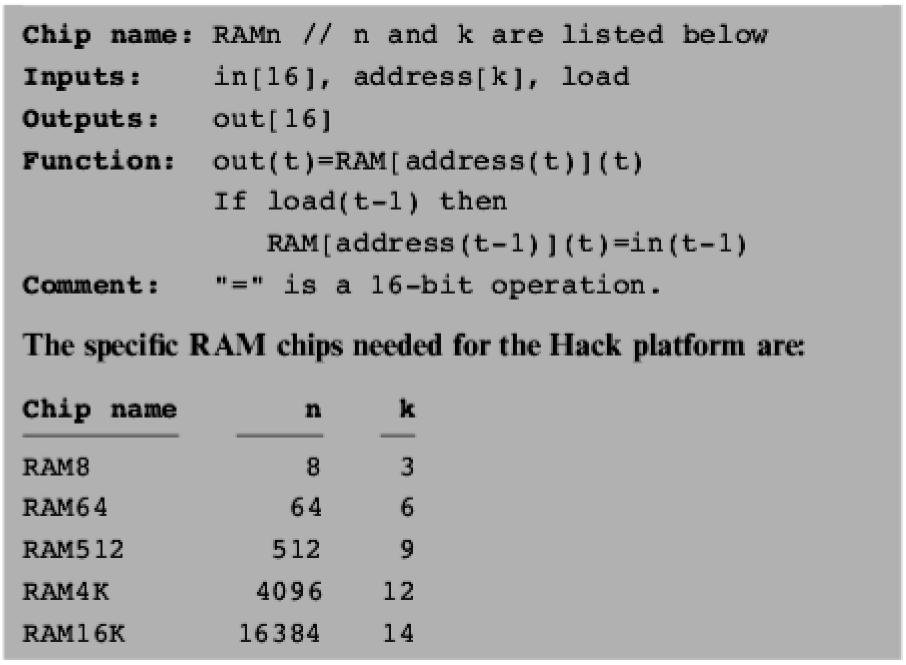


Figure 3.5: RAM.

The *width* of RAM of one of its registers (32- or 64-bit wide RAM).
The *size* of RAM is the number of registers in the RAM.





```
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

Figure 3.6: API for16-bits wide RAM with various sizes (RAM8, RAM64, RAM512, RAM4K and RAM16”).
Read: to read register number m, input *address* $= m$ outputting the value of this register (combinational operation independtent of the clock).
Write: to write new data d to register number m, input *address* $= m$,*in* $= d$ and asser the *load* input. In the next clock cycle, the selected register commits the new value d, as well as the RAM outputting this.

Counters a sequential chip whose state is $c \in \mathbb{Z}$ (typically 1) that increments every time unit, effecting $out\,(t) = out\,(t-1) + c$. Use for tasks that require such a measure.

3.2 Specification [p47]

Hierarchy of sequential chips:

- Data Flip-flops (DFFs).
- Registers (based on DFFs).
- Memory banks (based on registers).
- Coutner chips (based on registers).

3.3 Implementation [p50]

3.4 Perspective [p52]

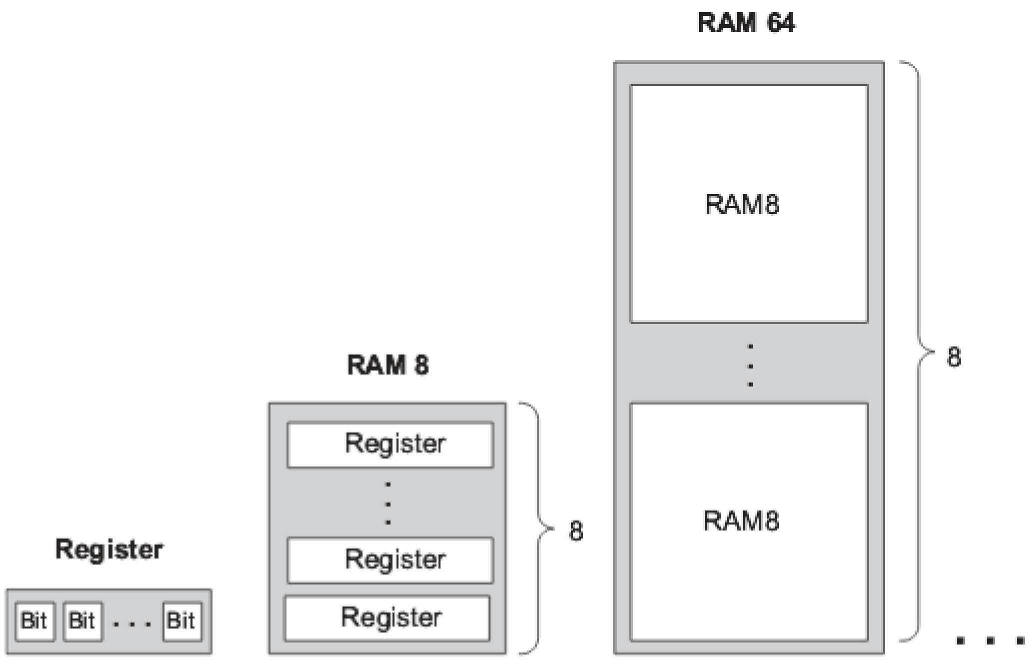
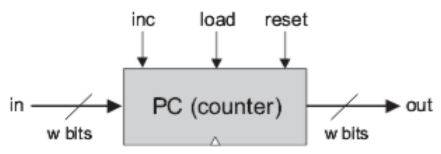
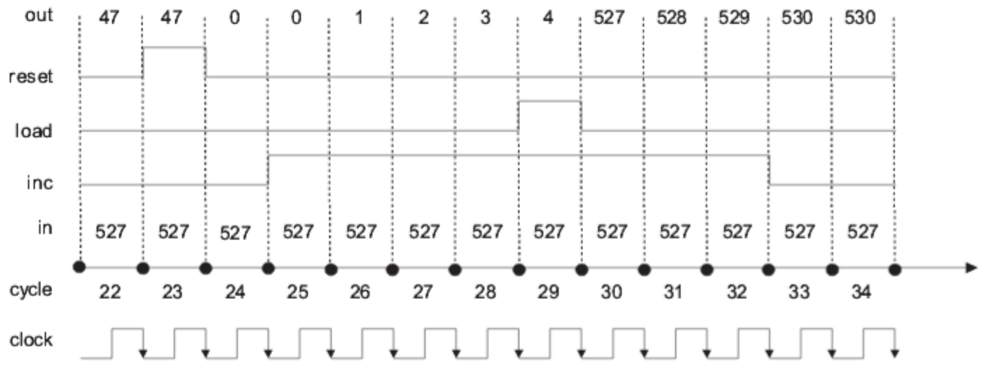


Figure 3.8: Construction of memory banks by recursive ascent.

4 Machine Language [p57]



```
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.
```



We assume that we start tracking the counter in time unit 22, when its input and output happen to be 527 and 47, respectively. We also assume that the counter's control bits (reset, load, inc) start at 0—all arbitrary assumptions.

Figure 3.7: API for a counter chip; similar to register with two extra inputs:
$inc = 1 \Rightarrow$increment the counter's state every clock cycle, outputting the result.
$reset = 1 \Rightarrow$reset counter to 0.