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.

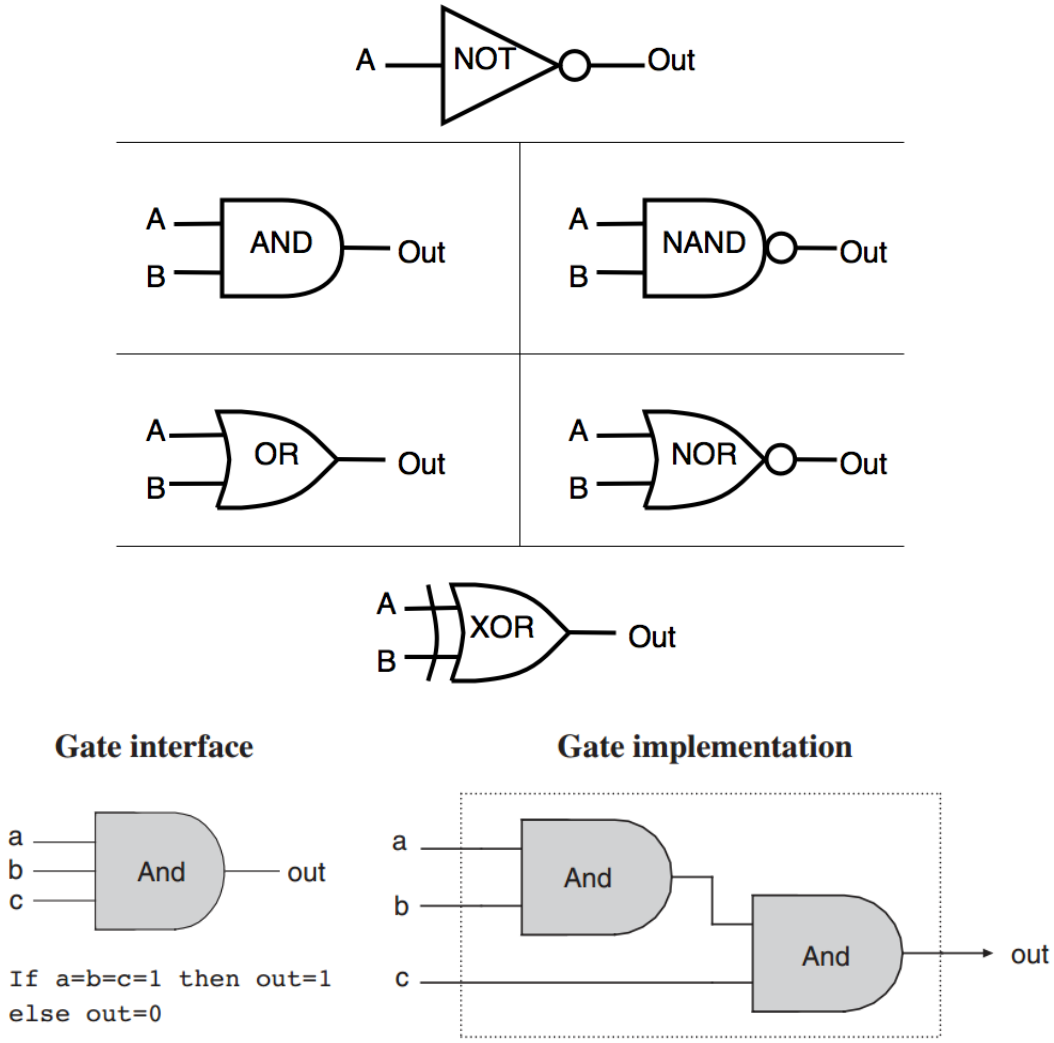


Figure 1.1: Left: Standard logic gate symbols. Right: **WHAT** a three-way AND gate is doing, and **HOW** it operates. Observe *the gate interface is unique*, whilst *there exists many different implementations*. The art of logic design is: *Given a gate specification, find an efficient way to implement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

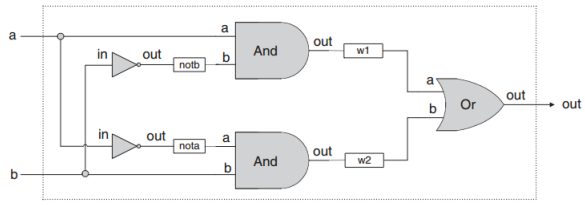
1.1.3 Actual Hardware Construction [p13]

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]

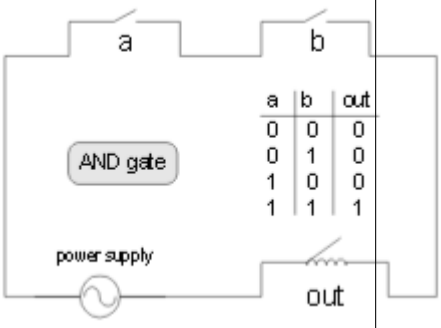
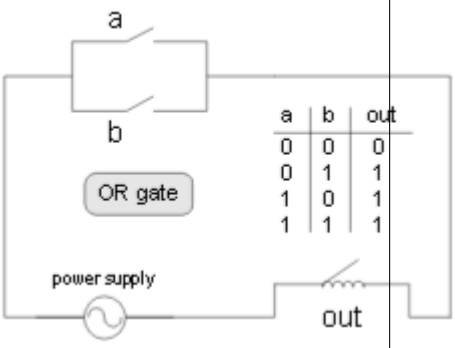
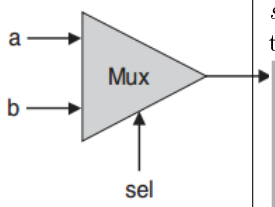
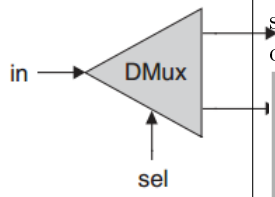
a	b	NAND (a, b)
0	0	1
0	1	1
1	0	1
1	1	0

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	
OR	**Chip name:** Or **Inputs:** a, b **Outputs:** out **Function:** If a=b=0 then out=0 else out=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]