

Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
(csdsp@bristol.ac.uk)

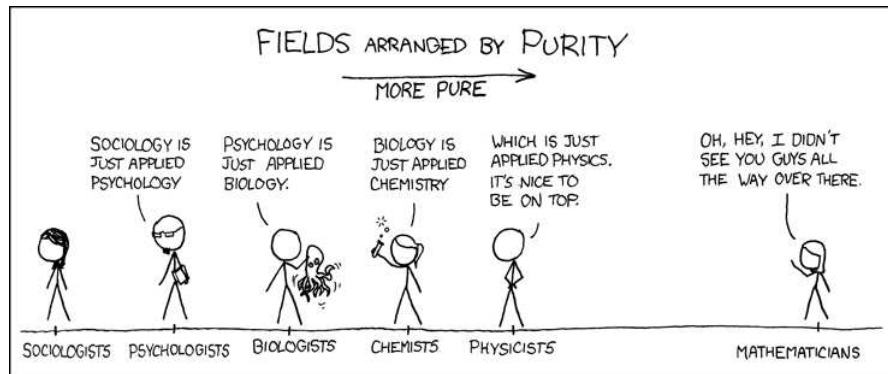
September 17, 2026

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
 - ▶ the associated notes page may be pre-populated with extra, written explanation of material covered in lecture(s), plus
 - ▶ anything with a “grey’ed out” header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:



<https://xkcd.com/435>

© Daniel Page (computer architecture)
Computer Architecture

University of
BRISTOL

git # 0073a8663 @ 2026-09-15

COMS10015 lecture: week #1

► Agenda: an introduction to

1. **propositional logic**,
2. **Boolean algebra**, and
3. application of, i.e., use-cases and rationale for the above within the context of COMS10015.

Notes:

Notes:

Part 1: propositional logic (1)

- **Concept:** a **proposition** is basically a statement

the temperature is 20°C

this statement is false

the temperature is too hot

which adheres to some simple rules.

Notes:

Part 1: propositional logic (1)

- **Concept:** a **proposition** is basically a statement

the temperature is 20°C

~~this statement is false~~

the temperature is too hot

which adheres to some simple rules, i.e., whose meaning

1. can be **evaluated** to produce a **truth value**, i.e., **false** or **true**.

Notes:

Part 1: propositional logic (1)

- **Concept:** a **proposition** is basically a statement

the temperature is 20°C

~~this statement is false~~

~~the temperature is too hot~~

which adheres to some simple rules, i.e., whose meaning

1. can be **evaluated** to produce a **truth value**, i.e., **false** or **true**,
2. must be unambiguous.

Notes:

Part 1: propositional logic (1)

- **Concept:** a **proposition** is basically a statement

the temperature is 20°C

the temperature is x °C

~~this statement is false~~

~~the temperature is too hot~~

which adheres to some simple rules, i.e., whose meaning

1. can be **evaluated** to produce a **truth value**, i.e., **false** or **true**,
2. must be unambiguous,
3. can include free **variables**, which must be bound to concrete values before evaluation.

Notes:

Part 1: propositional logic (1)

- **Concept:** a **proposition** is basically a statement

f = the temperature is 20°C
 $g(x)$ = the temperature is $x^{\circ}\text{C}$
~~this statement is false~~
~~the temperature is too hot~~

which adheres to some simple rules, i.e., whose meaning

1. can be **evaluated** to produce a **truth value**, i.e., **false** or **true**,
2. must be unambiguous,
3. can include free **variables**, which must be bound to concrete values before evaluation, and
4. can be represented using a short-hand variable or function.

Notes:

Part 1: propositional logic (2)

- **Question:** how can such statements be *represented*?
- **Answer #1:** using a *expression*-style form.
 - single statements can be combined using various **connectives**, e.g.,
the temperature is not 20°C
 - so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
 $\neg(\text{the temperature is } 20^{\circ}\text{C})$
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
the temperature is 20°C and it is sunny
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
(the temperature is 20°C) $\wedge$ (it is sunny)
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
the temperature is 20°C or it is sunny
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
(the temperature is 20°C) $\vee$ (it is sunny)
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
either the temperature is 20°C or it is sunny, but not both
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,
 4. “ x or y but not x and y ” is denoted $x \oplus y$, and usually called exclusive-or,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
(the temperature is 20°C) $\oplus$ (it is sunny)
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,
 4. “ x or y but not x and y ” is denoted $x \oplus y$, and usually called exclusive-or,

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
the temperature being 20°C implies that it is sunny
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,
 4. “ x or y but not x and y ” is denoted $x \oplus y$, and usually called exclusive-or,
 5. “ x implies y ” is denoted $x \Rightarrow y$, and sometimes written “if x then y ”, and

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
 $(\text{the temperature is } 20^{\circ}\text{C}) \Rightarrow (\text{it is sunny})$
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,
 4. “ x or y but not x and y ” is denoted $x \oplus y$, and usually called exclusive-or,
 5. “ x implies y ” is denoted $x \Rightarrow y$, and sometimes written “if x then y ”, and

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
the temperature is 20°C is equivalent to it being sunny
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,
 4. “ x or y but not x and y ” is denoted $x \oplus y$, and usually called exclusive-or,
 5. “ x implies y ” is denoted $x \Rightarrow y$, and sometimes written “if x then y ”, and
 6. “ x is equivalent to y ” is denoted $x \equiv y$, and sometimes written “ x if and only if y ” or “ x iff. y ”.

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
 - ▶ single statements can be combined using various **connectives**, e.g.,
(the temperature is 20°C) $\equiv$ (it is sunny)
 - ▶ so that, adding parentheses where needed to add clarity, we get
 1. “not x ” is denoted $\neg x$,
 2. “ x and y ” is denoted $x \wedge y$,
 3. “ x or y ” is denoted $x \vee y$, and usually called inclusive-or,
 4. “ x or y but not x and y ” is denoted $x \oplus y$, and usually called exclusive-or,
 5. “ x implies y ” is denoted $x \Rightarrow y$, and sometimes written “if x then y ”, and
 6. “ x is equivalent to y ” is denoted $x \equiv y$, and sometimes written “ x if and only if y ” or “ x iff. y ”.

Notes:

Part 1: propositional logic (2)

- ▶ **Question:** how can such statements be *represented*?
- ▶ **Answer #1:** using a *expression*-style form.
- ▶ you *might* see more formal terms or different notation for the *same* connectives:
 - ▶ $\neg$ is often termed logical **compliment** (or **negation**),
 - ▶ $\wedge$ is often termed logical **conjunction**,
 - ▶ $\vee$ is often termed logical (inclusive) **disjunction**,
 - ▶ $\oplus$ is often termed logical (exclusive) **disjunction**,
 - ▶ $\Rightarrow$ is often termed logical **implication**, and
 - ▶ $\equiv$ is often termed logical **equivalence**.

Notes:

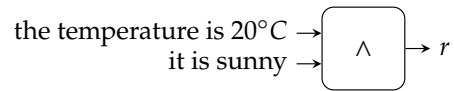
Part 1: propositional logic (3)

► **Question:** how can such statements be *represented*?

► **Answer #2:** using a *diagrammatic*-style form, e.g.,

$$r = (\text{the temperature is } 20^{\circ}\text{C}) \wedge (\text{it is sunny})$$

$\equiv$



which, more formally, could be termed a **data-flow diagram**.

Notes:

Part 1: propositional logic (4)

► **Question:** how can such statements be *evaluated*?

► **Answer:** in the general case,

- we know a given statement will evaluate to a truth value, i.e., **false** or **true**,
- we may want to evaluate for *one*, but can enumerate *all* input combinations,
- the result is termed a **truth table**, e.g., if

$$\begin{aligned} x &= \text{the temperature is } 20^{\circ}\text{C} \\ y &= \text{it is sunny} \\ r &= (\text{the temperature is } 20^{\circ}\text{C}) \wedge (\text{it is sunny}) \end{aligned}$$

then

inputs		output
x	y	r
false	false	false
false	true	false
true	false	false
true	true	true

noting that

1. each row details the output(s) associated with a given assignment to the inputs,
2. if there are n inputs, the truth table will have 2^n rows,
3. one can view the outcome as a form of specification.

Notes:

Part 1: propositional logic (5)

- **Question:** how can such statements be *evaluated*?
- **Answer:** in some specific cases, e.g., those for connectives such as

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \Rightarrow y$	$x \equiv y$
false	false	true	false	false	false	true	true
false	true	true	false	true	true	true	false
true	false	false	false	true	true	false	false
true	true	false	true	true	false	true	true

we already know what the associated truth table is.

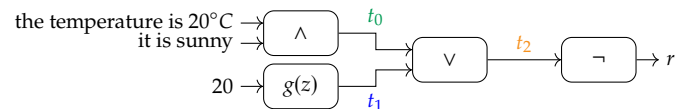
Notes:

Part 1: propositional logic (6)

- **Question:** how can such statements be *evaluated*?
- **Example:**
 - imagine we have

x = the temperature is 20°C
 y = it is sunny
 $g(z)$ = the temperature is z°C
 r = $\neg(((\text{the temperature is } 20^\circ\text{C}) \wedge (\text{it is sunny})) \vee (\text{the temperature is } z^\circ\text{C}))$

which could be translated into the diagrammatic form



- an example evaluation might be as follows:

inputs		intermediates			output
x	y	t_0	t_1	t_2	r
false	false	false	false	false	true
false	true	false	false	false	true
true	false	false	true	true	false
true	true	true	true	true	false

Notes:

Part 2: Boolean algebra (1)

► Concept:

1. in **elementary algebra**, for some number x we have that

$$x + 0 = x$$

and

$$x \cdot 1 = x,$$

2. in **set theory**, for some set x we have that

$$x \cup \emptyset = x$$

and

$$x \cap \mathcal{U} = x,$$

3. in **propositional logic**, for some truth value x we have that

$$x \vee \mathbf{false} = x$$

and

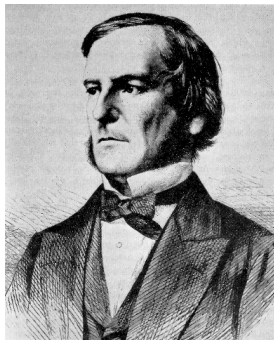
$$x \wedge \mathbf{true} = x.$$

Notes:

Part 2: Boolean algebra (2)

Thou must

1. work with the set $\mathbb{B} = \{0, 1\}$ of **binary** digits, using 0 and 1 instead of **false** and **true**,
 2. shorten every statement into either a **variable** *or* **function**,
 3. use **unary operators**, e.g., $\neg$ (or NOT), and **binary operators**, e.g., $\wedge$ and $\vee$ (or AND and OR), to form **expressions**,
 4. manipulate said expressions according to some axioms (or rules),
- then call the result **Boolean algebra**.



Notes:

Part 2: Boolean algebra (3)

► **Concept:** put more concretely, we now have

1. a set of operators specified by

Definition

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \Rightarrow y$	$x \equiv y$
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

Notes:

Part 2: Boolean algebra (3)

► **Concept:** put more concretely, we now have

2. a set of axioms that allow manipulation of expressions comprised of said operators, i.e.,

Definition

Name	Axiom(s)	Name	Axiom(s)
commutativity	$x \wedge y \equiv y \wedge x$	commutativity	$x \vee y \equiv y \vee x$
association	$(x \wedge y) \wedge z \equiv x \wedge (y \wedge z)$	association	$(x \vee y) \vee z \equiv x \vee (y \vee z)$
distribution	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	distribution	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$

plus other rules such as **precedence** (to deal with ambiguity in the absence of parentheses).

Notes:

Part 2: Boolean algebra (3)

► **Concept:** put more concretely, we now have

2. a set of axioms that allow manipulation of expressions comprised of said operators, i.e.,

Definition			
Name	Axiom(s)	Name	Axiom(s)
identity	$x \wedge 1 \equiv x$	identity	$x \vee 0 \equiv x$
null	$x \wedge 0 \equiv 0$	null	$x \vee 1 \equiv 1$
idempotency	$x \wedge x \equiv x$	idempotency	$x \vee x \equiv x$
inverse	$x \wedge \neg x \equiv 0$	inverse	$x \vee \neg x \equiv 1$

plus other rules such as **precedence** (to deal with ambiguity in the absence of parentheses).

Notes:

Part 2: Boolean algebra (3)

► **Concept:** put more concretely, we now have

2. a set of axioms that allow manipulation of expressions comprised of said operators, i.e.,

Definition			
Name	Axiom(s)	Name	Axiom(s)
absorption	$x \wedge (x \vee y) \equiv x$	absorption	$x \vee (x \wedge y) \equiv x$
de Morgan	$\neg(x \wedge y) \equiv \neg x \vee \neg y$	de Morgan	$\neg(x \vee y) \equiv \neg x \wedge \neg y$

plus other rules such as **precedence** (to deal with ambiguity in the absence of parentheses).

Notes:

► **Concept:** put more concretely, we now have

2. a set of axioms that allow manipulation of expressions comprised of said operators, i.e.,

Definition		
Name	Axiom(s)	
equivalence	$x \equiv y$	$\equiv (x \Rightarrow y) \wedge (y \Rightarrow x)$
implication	$x \Rightarrow y$	$\equiv \neg x \vee y$
involution	$\neg \neg x$	$\equiv x$

plus other rules such as **precedence** (to deal with ambiguity in the absence of parentheses).

Notes:

Definition
The fact there are AND and OR forms of most axioms hints at a more general underlying principle. Consider a Boolean expression e: the principle of duality states that the dual expression e^D is formed by 1. leaving each variable as is, 2. swapping each $\wedge$ with $\vee$ and vice versa, and 3. swapping each 0 with 1 and vice versa. Of course e and e^D are different expressions, and clearly not equivalent; if we start with some $e \equiv f$ however, then we do still get $e^D \equiv f^D$.

Example
As an example, consider axioms for 1. distribution, e.g., if $e = x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$ then $e^D = x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$ and 2. identity, e.g., if $e = x \wedge 1 \equiv x$ then $e^D = x \vee 0 \equiv x.$

Notes:

Definition

The de Morgan axiom can be turned into a more general principle. Consider a Boolean expression e : the **principle of complements** states that the **complement expression** $\neg e$ is formed by

1. swapping each variable x with the complement $\neg x$,
2. swapping each $\wedge$ with $\vee$ and vice versa, and
3. swapping each 0 with 1 and vice versa.

Example

As an example, consider that if

$$e = x \wedge y \wedge z,$$

then by the above we should find

$$f = \neg e = (\neg x) \vee (\neg y) \vee (\neg z).$$

Proof:

x	y	z	$\neg x$	$\neg y$	$\neg z$	e	f
0	0	0	1	1	1	0	1
0	0	1	1	1	0	0	1
0	1	0	1	0	1	0	1
0	1	1	1	0	0	0	1
1	0	0	0	1	1	0	1
1	0	1	0	1	0	0	1
1	1	0	0	0	1	0	1
1	1	1	0	0	0	1	0

Notes:

Part 2: Boolean algebra (6)

► **Concept:** we can define various **derived operators** in terms of NOT, AND, and OR.

► **Example:**

► “exclusive-OR” or **XOR**, such that

$$x \oplus y \equiv (\neg x \wedge y) \vee (x \wedge \neg y)$$

so

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Notes:

Part 2: Boolean algebra (6)

► **Concept:** we can define various **derived operators** in terms of NOT, AND, and OR.

► **Example:**

► “NOT-AND” or **NAND**, such that

$$x \overline{\wedge} y \equiv \neg(x \wedge y)$$

so

x	y	$x \overline{\wedge} y$
0	0	1
0	1	1
1	0	1
1	1	0

► “NOT-OR” or **NOR**, such that

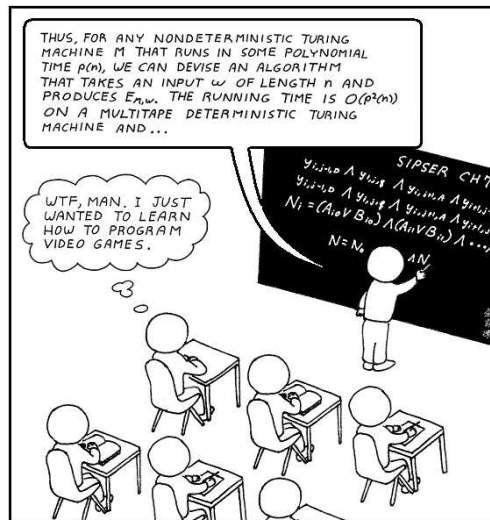
$$x \overline{\vee} y \equiv \neg(x \vee y)$$

so

x	y	$x \overline{\vee} y$
0	0	1
0	1	0
1	0	0
1	1	0

Notes:

Part 3: towards practical use-cases (1)



Notes:

- **Concept:**
 - Boolean algebra plays a role in *many* contexts, e.g.,

$r \text{ is } x$	$\equiv$	$r = x$	$\cong$	$\mathbf{r} == \mathbf{x}$	$\cong$	$\mathbf{r} = \mathbf{x}$
$r \text{ is NOT } x$	$\equiv$	$r = \neg x$	$\cong$	$\mathbf{r} != \mathbf{x}$	$\cong$	$\mathbf{r} = \sim \mathbf{x}$
$r \text{ is } x \text{ NAND } y$	$\equiv$	$r = x \overline{\wedge} y$	$\cong$	$\mathbf{r} != \mathbf{x} \ \&\& \ \mathbf{y}$	$\cong$	$\mathbf{r} = \sim(\ \mathbf{x} \ \& \ \mathbf{y} \)$
$r \text{ is } x \text{ NOR } y$	$\equiv$	$r = x \overline{\vee} y$	$\cong$	$\mathbf{r} != \mathbf{x} \ \ \mathbf{y}$	$\cong$	$\mathbf{r} = \sim(\ \mathbf{x} \ \ \mathbf{y} \)$
$r \text{ is } x \text{ AND } y$	$\equiv$	$r = x \wedge y$	$\cong$	$\mathbf{r} == \mathbf{x} \ \&\& \ \mathbf{y}$	$\cong$	$\mathbf{r} = \mathbf{x} \ \& \ \mathbf{y}$
$r \text{ is } x \text{ OR } y$	$\equiv$	$r = x \vee y$	$\cong$	$\mathbf{r} == \mathbf{x} \ \ \mathbf{y}$	$\cong$	$\mathbf{r} = \mathbf{x} \ \ \mathbf{y}$
$r \text{ is } x \text{ XOR } y$	$\equiv$	$r = x \oplus y$	$\cong$		$\cong$	$\mathbf{r} = \mathbf{x} \ \wedge \ \mathbf{y}$

- note that the latter columns capture
 - **decisional** operators, e.g., $\&\& : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, and
 - **computational** operators, e.g., $\& : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^n$.

Notes:

Listing

```
1 void condition_v1( int a, int b, int c, int d ) {
2   if( ( ( a + b ) > c ) && ( ( d % 2 ) == 0 ) ) {
3     //      ( a + b is    > c ) and      ( d is    even )
4
5     printf( "if -> true   : %d %d %d %d\n", a, b, c, d );
6   }
7   else {
8     //      not ( ( a + b is    > c ) and      ( d is    even ) )
9     //
10    // = not   ( a + b is    > c ) or not ( d is    even )
11    // =      ( a + b isn't > c ) or   ( d isn't even )
12    // =      ( a + b is    <= c ) or   ( d is    odd  )
13
14    printf( "if -> false  : %d %d %d %d\n", a, b, c, d );
15  }
16 }
```

Notes:

Listing

```
1 void condition_v2( int a, int b, int c, int d ) {  
2   if( ( ( a + b ) > c ) && ( d = 3 ) ) {  
3     printf( "if -> true : %d %d %d %d\n", a, b, c, d );  
4   }  
5   else {  
6     printf( "if -> false : %d %d %d %d\n", a, b, c, d );  
7   }  
8 }
```

Notes:

Part 3: a (more) practical perspective (4)

Formalisation $\leadsto$ optimisation

► **Question:** simplify the Boolean expression

$$f = (\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$

into a form that contains the fewest operators possible.

Notes:

- **Question:** simplify the Boolean expression

$$f = (\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$

into a form that contains the fewest operators possible.

- **Solution #1:** less steps.

$$\begin{aligned} f &= (\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b) \\ &= \neg(a \vee b) \vee (\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \quad (\text{commutativity}) \\ &= \neg(a \vee b) \quad (\text{absorption}) \end{aligned}$$

Notes:

- **Question:** simplify the Boolean expression

$$f = (\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$

into a form that contains the fewest operators possible.

- **Solution #2:** more steps.

$$\begin{aligned} f &= (\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b) \\ &= ((\neg a \wedge \neg b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b) \quad (\text{de Morgan}) \\ &= ((\neg a \wedge \neg b) \wedge (\neg c \wedge \neg d \wedge \neg e)) \vee \neg(a \vee b) \quad (\text{de Morgan}) \\ &= ((\neg a \wedge \neg b) \wedge (\neg c \wedge \neg d \wedge \neg e)) \vee (\neg a \wedge \neg b) \quad (\text{de Morgan}) \\ &= (\neg a \wedge \neg b) \vee ((\neg a \wedge \neg b) \wedge (\neg c \wedge \neg d \wedge \neg e)) \quad (\text{commutativity}) \\ &= (\neg a \wedge \neg b) \quad (\text{absorption}) \\ &= \neg(a \vee b) \quad (\text{de Morgan}) \end{aligned}$$

Notes:

- **Question:** simplify the Boolean expression

$$g = (a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c)$$

into a form that contains the fewest operators possible.

Notes:

- **Question:** simplify the Boolean expression

$$g = (a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c)$$

into a form that contains the fewest operators possible.

- **Solution:**

$$\begin{aligned} g &= (a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c) \\ &= (a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c) \vee (\neg a \wedge b) && \text{(commutativity)} \\ &= (a \wedge b) \wedge (c \vee \neg c) \vee (\neg a \wedge b) && \text{(distribution)} \\ &= (a \wedge b) \wedge 1 \vee (\neg a \wedge b) && \text{(inverse)} \\ &= (a \wedge b) \vee (\neg a \wedge b) && \text{(identity)} \\ &= b \wedge (a \vee \neg a) && \text{(distribution)} \\ &= b \wedge 1 && \text{(inverse)} \\ &= b && \text{(identity)} \end{aligned}$$

Notes:

► **Question:** simplify the Boolean expression

$$g = (a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c)$$

into a form that contains the fewest operators possible.

► **Proof:**

<i>a</i>	<i>b</i>	<i>c</i>	$\neg a$	$\neg b$	$\neg c$	<i>t</i> ₀	<i>t</i> ₁	<i>t</i> ₂	<i>g</i>
0	0	0	1	1	1	0	0	0	0
0	0	1	1	1	0	0	0	0	0
0	1	0	1	0	1	0	1	0	1
0	1	1	1	0	0	0	1	0	1
1	0	0	0	1	1	0	0	0	0
1	0	1	0	1	0	0	0	0	0
1	1	0	0	0	1	0	0	1	1
1	1	1	0	0	0	1	0	0	1

where

$$\begin{aligned} t_0 &= a \wedge b \wedge c \\ t_1 &= \neg a \wedge b \\ t_2 &= a \wedge b \wedge \neg c \end{aligned}$$

such that $g = t_0 \vee t_1 \vee t_2$.

Quote

If I designed a computer with 200 chips, I tried to design it with 150. And then I would try to design it with 100. I just tried to find every trick I could in life to design things real tiny.

– Wozniak

Quote

So I took 20 chips off their board; I bypassed 20 of their chips.

– Wozniak

Notes:

Notes:

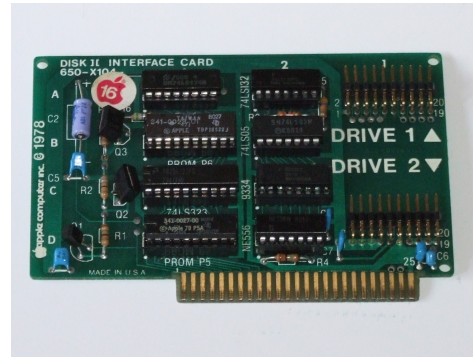
- The quotes relate to design and implementation of a (floppy) disk controller for the Apple II computer (circa 1977); there is an obvious focus on efficiency, which is credited as allowing the controller to be commercially viable. A detailed overview of the overarching anecdote is available via

https://en.wikipedia.org/wiki/Disk_II

or

<https://apple2history.org/history/ah05/>

The moral is that, in reality, “it works”, while important, may not be good enough: meeting various other (market-driven) quality metrics (e.g., efficiency, physical size, power consumption, etc.) is often vital rather than simply attractive.



Notes:

https://en.wikipedia.org/wiki/File:Shugart_SA400.jpg

https://en.wikipedia.org/wiki/File:Interface_Card_-_Disk_II_Interface_Apple2.jpg

Definition

Consider a Boolean expression:

1. When the expression is written as a sum (i.e., OR) of terms which each comprise the product (i.e., AND) of variables, e.g.,

$$\underbrace{(a \wedge b \wedge c) \vee (d \wedge e \wedge f)}_{\text{minterm}},$$

it is said to be in **disjunctive normal form** or **Sum of Products (SoP)** form; the terms are called the **minterms**. Note that each variable can exist as-is *or* complemented using NOT, meaning

$$\underbrace{(\neg a \wedge b \wedge c) \vee (d \wedge \neg e \wedge f)}_{\text{minterm}},$$

is also a valid SoP expression.

2. When the expression is written as a product (i.e., AND) of terms which each comprise the sum (i.e., OR) of variables, e.g.,

$$\underbrace{(a \vee b \vee c) \wedge (d \vee e \vee f)}_{\text{maxterm}},$$

it is said to be in **conjunctive normal form** or **Product of Sums (PoS)** form; the terms are called the **maxterms**. As above each variable can exist as-is *or* complemented using NOT.

Notes:

- **Concept: Electronic Design Automation (EDA)** [3], applied to tasks such as
 - manipulation,
 - translation,
 - simulation,
 - verification,
 - ...
- of functionality related to and explained by Boolean algebra.

Notes:

- Seymour Cray (a *supercomputer* architect) produced some interesting anecdotes about use of Computer-Aided Design (CAD) tools, which one could view as a more general case of EDA, in his design process: see for example

https://en.wikipedia.org/wiki/Seymour_Cray

Listing

```
1 from sympy import *
2 from sympy.logic.boolalg import simplify_logic
3
4 a, b, c, d, e = symbols("a b c d e")
5
6 f = (~ (a | b) & ~(c | d | e)) | ~(a | b)
7
8 print(f)
9 print(simplify_logic(f))
10
11 g = (a & b & c) | (~a & b) | (a & b & ~c)
12
13 print(g)
14 print(simplify_logic(g))
```

Output

```
1 ~(a | b) | ~(a | b) & ~(c | d | e)
2 ~a & ~b
3 (b & ~a) | (a & b & c) | (a & b & ~c)
4 b
```

Notes:

- Note that this example can be explored online via

<https://live.sympy.org>

- **Question:** imagine we have a Boolean expression (or circuit)

$$f(x_0, x_1, \dots x_{n-1})$$

and manipulate it into a different form

$$g(x_0, x_1, \dots x_{n-1})$$

e.g., optimise it: we want to know if there is a bug in g .

Notes:

- **Question:** imagine we have a Boolean expression (or circuit)

$$f(x_0, x_1, \dots x_{n-1})$$

and manipulate it into a different form

$$g(x_0, x_1, \dots x_{n-1})$$

e.g., optimise it: we want to know if there is a bug in g .

- **Solution(s):**

1. *prove* that f and g are equivalent,
2. enumerate all 2^n possible input combinations, and see if f ever differs from g , or
3. use a (carefully defined) instance of **SAT**.

Notes:

Definition

The **Boolean satisfiability problem** (or **SAT**) is a decision problem. Consider some Boolean function

$$f(x_0, x_1, \dots, x_{n-1}).$$

which defines a **SAT instance**. The problem is to decide whether or not an assignment to said variables (i.e., $x_i \in \{0, 1\}$ for $0 \leq i < n$) exists st. f is **satisfiable** (i.e., we have $f(x_0, x_1, \dots, x_{n-1}) = 1$).

Notes:

- SAT is NP-complete [8, 9] meaning that
 - no known algorithm will efficiently solve every SAT instance, but
 - other NP problems can be expressed as SAT instances.

► **Question:** decide whether

$$f(a, b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$\equiv$

$$g(a, b) = (a \vee b) \wedge \neg(a \wedge b).$$

Notes:

- The idea of the SAT instance is to use how XOR works: what we're saying is that if h is satisfiable, then we have an assignment to the variables st. the result of f and g *differ* (XOR produces 1 as output if one (and only one) of the inputs are 1, and 0 otherwise).
- One underlying idea here is that each approach is valid, but also might be advantageous or disadvantageous when used in a particular context: the use of axiomatic manipulation does not *tell* us anything (e.g., why f and g differ, if they do), for example, and brute-force enumeration will clearly become intractable very quickly (as n grows). The other is that greater understanding of the relationship between theory and practice allows better results in both: here, in a rough sense,
 - in theory, SAT helps us understand *what* we can compute, while
 - in practice, SAT helps us understand *how* we compute it.

► **Question:** decide whether

$$f(a, b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$\equiv$

$$g(a, b) = (a \vee b) \wedge \neg(a \wedge b).$$

► **Solution #1:** manipulation via axioms, e.g.,

$$\begin{aligned} f(a, b) &= (b \wedge \neg a) && \vee && (a \wedge \neg b) \\ &= (b \wedge \neg a) \vee 0 && \vee && (a \wedge \neg b) \vee 0 && \text{(identity)} \\ &= (b \wedge \neg a) \vee (b \wedge \neg b) && \vee && (a \wedge \neg b) \vee (a \wedge \neg a) && \text{(inverse)} \\ &= (b \wedge (\neg a \vee \neg b)) && \vee && (a \wedge (\neg b \vee \neg a)) && \text{(distribution)} \\ &= ((\neg a \vee \neg b) \wedge b) && \vee && ((\neg a \vee \neg b) \wedge a) && \text{(commutativity)} \\ &= (\neg a \vee \neg b) \wedge (a \vee b) && && && \text{(distribution)} \\ &= (a \vee b) \wedge (\neg a \vee \neg b) && && && \text{(commutativity)} \\ &= (a \vee b) \wedge \neg(a \wedge b) && && && \text{(de Morgan)} \\ &= g(a, b) \end{aligned}$$

Notes:

- The idea of the SAT instance is to use how XOR works: what we're saying is that if h is satisfiable, then we have an assignment to the variables st. the result of f and g differ (XOR produces 1 as output if one (and only one) of the inputs are 1, and 0 otherwise).
- One underlying idea here is that each approach is valid, but also might be advantageous or disadvantageous when used in a particular context: the use of axiomatic manipulation does not *tell* us anything (e.g., why f and g differ, if they do), for example, and brute-force enumeration will clearly become intractable very quickly (as n grows). The other is that greater understanding of the relationship between theory and practice allows better results in both: here, in a rough sense,
 - in theory, SAT helps us understand *what* we can compute, while
 - in practice, SAT helps us understand *how* we compute it.

► **Question:** decide whether

$$f(a, b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$\equiv$

$$g(a, b) = (a \vee b) \wedge \neg(a \wedge b).$$

► **Solution #2:** brute-force enumeration, i.e.,

a	b	$b \wedge \neg a$	$a \wedge \neg b$	$f(a, b)$	$a \vee b$	$\neg(a \wedge b)$	$g(a, b)$
0	0	0	0	0	0	1	0
0	1	1	0	1	1	1	1
1	0	0	1	1	1	1	1
1	1	0	0	0	1	0	0

Notes:

- The idea of the SAT instance is to use how XOR works: what we're saying is that if h is satisfiable, then we have an assignment to the variables st. the result of f and g differ (XOR produces 1 as output if one (and only one) of the inputs are 1, and 0 otherwise).
- One underlying idea here is that each approach is valid, but also might be advantageous or disadvantageous when used in a particular context: the use of axiomatic manipulation does not *tell* us anything (e.g., why f and g differ, if they do), for example, and brute-force enumeration will clearly become intractable very quickly (as n grows). The other is that greater understanding of the relationship between theory and practice allows better results in both: here, in a rough sense,
 - in theory, SAT helps us understand *what* we can compute, while
 - in practice, SAT helps us understand *how* we compute it.

► **Question:** decide whether

$$f(a, b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$\equiv$

$$g(a, b) = (a \vee b) \wedge \neg(a \wedge b).$$

► **Solution #3:** define

$$h(x_0, x_1, \dots, x_{n-1}) = f(x_0, x_1, \dots, x_{n-1}) \oplus g(x_0, x_1, \dots, x_{n-1})$$

and use this as a SAT instance: if the SAT instance is satisfiable, this yields a **test vector** we can use to debug g .

Notes:

- The idea of the SAT instance is to use how XOR works: what we're saying is that if h is satisfiable, then we have an assignment to the variables st. the result of f and g differ (XOR produces 1 as output if one (and only one) of the inputs are 1, and 0 otherwise).
- One underlying idea here is that each approach is valid, but also might be advantageous or disadvantageous when used in a particular context: the use of axiomatic manipulation does not *tell* us anything (e.g., why f and g differ, if they do), for example, and brute-force enumeration will clearly become intractable very quickly (as n grows). The other is that greater understanding of the relationship between theory and practice allows better results in both: here, in a rough sense,
 - in theory, SAT helps us understand *what* we can compute, while
 - in practice, SAT helps us understand *how* we compute it.

Listing

```
1 from sympy import *
2 from sympy.logic.inference import satisfiable
3
4 a, b, c, d, e = symbols("a b c d e")
5
6 f = (b & ~a) | (a & ~b)
7 g = (a | b) & ~(a & b)
8
9 print(f)
10 print(g)
11 print(satisfiable(f ^ g))
```

Output

```
1 (a & ~b) | (b & ~a)
2 (a | b) & ~(a & b)
3 False
```

Notes:

- Note that this example can be explored online via <https://live.sympy.org>

- **Concept:** truth tables can accommodate **don't care** entries, e.g.,

x	y	r
?	0	1
0	1	?
1	1	0

such that

- a ? (rather than 0 or 1) means we “don't care” ($\neq$ “don't know”),
- on the LHS, for an *input*,
 - ? is a wildcard (or short-hand),
 - it means 0 *and* 1,
 - we've compressed two truth table rows into one.
- on the RHS, for an *output*,
 - ? is a choice,
 - it means 0 *or* 1,
 - we can select which one to, e.g., optimise the associated expression.

Notes:

Part 3: a (more) practical perspective (16)

Constraints $\rightsquigarrow$ utilisation

- **Concept:** NAND and NOR are **universal** (or **functionally complete**), e.g.,

$$\begin{aligned}\neg x &\equiv x \bar{\wedge} x \\ x \wedge y &\equiv (x \bar{\wedge} y) \bar{\wedge} (x \bar{\wedge} y) \\ x \vee y &\equiv \neg x \bar{\wedge} \neg y \equiv (x \bar{\wedge} x) \bar{\wedge} (y \bar{\wedge} y)\end{aligned}$$

which we can prove via

x	y	$x \bar{\wedge} y$	$x \bar{\wedge} x$	$y \bar{\wedge} y$	$(x \bar{\wedge} y) \bar{\wedge} (x \bar{\wedge} y)$	$(x \bar{\wedge} x) \bar{\wedge} (y \bar{\wedge} y)$
0	0	1	1	1	0	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	0	0	0	1	1

$\therefore$ *any* Boolean function can be expressed using a *single* operator.

Notes:

► **Question:** translate

$$x \wedge (y \vee z)$$

into a version using NAND only.

Notes:

► **Question:** translate

$$x \wedge (y \vee z)$$

into a version using NAND only.

► **Solution #1:** apply the identities *naively* to get

$$\begin{aligned} & x \wedge (y \vee z) \\ = & x \wedge ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z)) \\ = & (x \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z))) \overline{\wedge} (x \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z))) \end{aligned}$$

Notes:

► **Question:** translate

$$x \wedge (y \vee z)$$

into a version using NAND only.

► **Solution #2:** apply the identities *intelligently* to get

$$\begin{aligned} & x \wedge (y \vee z) \\ = & x \wedge ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z)) \\ = & t \overline{\wedge} t \end{aligned}$$

where $t = x \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z))$ is a common sub-expression [2].

Notes:

Conclusions

► **Take away points:**

1. Boolean algebra is a (somewhat) cosmetic extension of what you already know.
2. The design of computational devices, e.g., micro-processors, *isn't* ad hoc: Boolean algebra offers a theoretical basis for reasoning about computational devices (and computation).
 - *any* Boolean function f which can be specified using a truth table can be computed using an associated Boolean expression,
 - a Boolean expression is composed of Boolean operators,
 - *if* we (physically) implement the Boolean operators, we can implement the Boolean expression and hence compute f .

Notes:

Conclusions

► Take away points:

3. We'll focus on *application* (i.e., *use*) versus *theory* (e.g., *study*) of Boolean algebra from here on.
4. Keep in mind that
 - “theory $\neq$ practice”,
 - “a solution” $\neq$ “the solution”; “it works” $\neq$ “it works *well*”,
 - worse-is-better > do-the-right-thing [4] $\implies$ “get the job done”,
 - ...

Notes:

Additional Reading

- *Wikipedia: Boolean algebra*. URL: https://en.wikipedia.org/wiki/Boolean_algebra.
- D. Page. “Chapter 1: Mathematical preliminaries”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009.
- W. Stallings. “Chapter 11: Digital logic”. In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013.
- A.S. Tanenbaum and T. Austin. “Section 3.1: Gates and Boolean algebra”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.

Notes:

References

- [1] [Wikipedia: Boolean algebra](https://en.wikipedia.org/wiki/Boolean_algebra). URL: https://en.wikipedia.org/wiki/Boolean_algebra (see p. 135).
- [2] [Wikipedia: Common sub-expression elimination](https://en.wikipedia.org/wiki/Common_subexpression_elimination). URL: https://en.wikipedia.org/wiki/Common_subexpression_elimination (see pp. 127, 129).
- [3] [Wikipedia: Electronic Design Automation \(EDA\)](https://en.wikipedia.org/wiki/Electronic_design_automation). URL: https://en.wikipedia.org/wiki/Electronic_design_automation (see p. 101).
- [4] [Wikipedia: Worse is better](https://en.wikipedia.org/wiki/Worse_is_better). URL: https://en.wikipedia.org/wiki/Worse_is_better (see pp. 131, 133).
- [5] D. Page. “Chapter 1: Mathematical preliminaries”. In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer, 2009 (see p. 135).
- [6] W. Stallings. “Chapter 11: Digital logic”. In: *Computer Organisation and Architecture*. 9th ed. Prentice Hall, 2013 (see p. 135).
- [7] A.S. Tanenbaum and T. Austin. “Section 3.1: Gates and Boolean algebra”. In: *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012 (see p. 135).
- [8] S. Cook. “The complexity of theorem proving procedures”. In: *ACM Symposium on Theory of Computing (STOC)*. 1971, pp. 151–158 (see p. 110).
- [9] L. Levin. “Universal search problems”. In: *Problems of Information Transmission* 9:3 (1973), pp. 265–266 (see p. 110).

Notes: