Intro. to Computer Architecture

Daniel Page

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

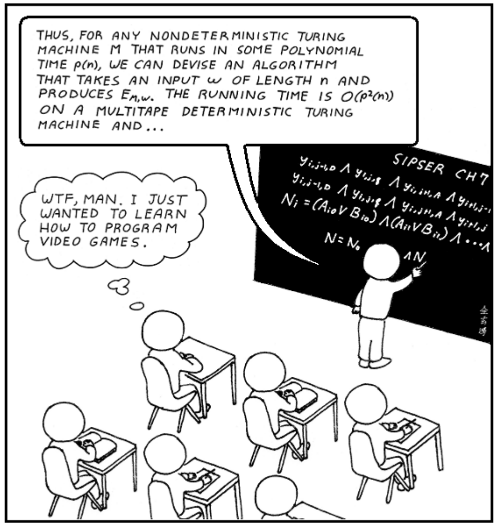
January 9, 2018

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 explaination 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:



http://abstrusegoose.com/206

Notes:

COMS12200 lecture: week #1

- (Fairly) reasonable question(s):
  1. “I thought this was CS, not Maths!”, and
  2. “why does this unit duplicate material in *other* units?”.

Notes:

- (Fairly) reasonable question(s):
  1. "I thought this was CS, not Maths!", and
  2. "why does this unit duplicate material in *other* units?".
- Answer: it isn't, and it doesn't (well, not *too* much) ... note
  - theoretical concepts, e.g.,

$$\text{axiomatic manipulation} \rightsquigarrow \begin{cases} \textbf{optimisation} \\ \textbf{universality} \end{cases}$$

    often have significant practical motivations or implications,
  - it's perfectly reasonable to utilise **Electronic Design Automation (EDA)** [3] tools, and
  - Boolean algebra has wider application than hardware design.

Notes:

Boolean algebra: theory ≠ practice (1)

Axiomatic manipulation $\rightsquigarrow$ optimisation

- Question: simplify the Boolean expression

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

into a form that contains the fewest operators possible.

Notes:

Boolean algebra: theory ≠ practice (1)

Axiomatic manipulation $\rightsquigarrow$ optimisation

- Question: simplify the Boolean expression

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

into a form that contains the fewest operators possible.

- Solution #1: less steps.

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

Notes:

Boolean algebra: theory ≠ practice (1)

Axiomatic manipulation $\rightsquigarrow$ optimisation

- Question: simplify the Boolean expression

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

into a form that contains the fewest operators possible.

- Solution #2: more steps.

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

Notes:

Boolean algebra: theory $\neq$ practice (2)

Axiomatic manipulation $\leadsto$ optimisation

- **Question**: simplify the Boolean expression

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

into a form that contains the fewest operators possible.

Notes:

Boolean algebra: theory $\neq$ practice (2)

Axiomatic manipulation $\leadsto$ optimisation

- **Question**: simplify the Boolean expression

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

into a form that contains the fewest operators possible.

- **Solution**:

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

Notes:

Boolean algebra: theory ≠ practice (3)

Axiomatic manipulation ⇝ optimisation

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:

- 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

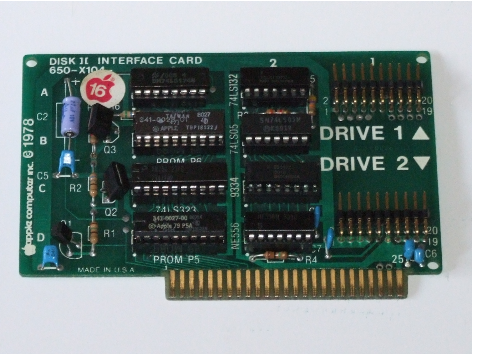
  or

  http://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.

Boolean algebra: theory ≠ practice (4)

Axiomatic manipulation ⇝ optimisation



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

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

Notes:

Boolean algebra: theory ≠ practice (5)

Axiomatic manipulation $\rightsquigarrow$ universality

- We *can* add to our existing suite of operators (the result is often termed a **derived operator**), e.g.,
  - "NOT-AND" or **NAND**, st.

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

so

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

and

  - "NOT-OR" or **NOR**, st.

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

so

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

Notes:

Boolean algebra: theory ≠ practice (6)

Axiomatic manipulation $\rightsquigarrow$ universality

- Question: haven't we already got enough ... this is *already* quite difficult!
- Answer: NAND and NOR turn out to be **functionally complete** (or **universal**), e.g.,

$$\begin{array}{lcl} \neg x & \equiv & x \overline{\wedge} x \\ x \wedge y & \equiv & (x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y) \\ x \vee y \equiv \neg x \overline{\wedge} \neg y & \equiv & (x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y) \end{array}$$

which we can prove via

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

Notes:

Boolean algebra: theory $\neq$ practice (6)

Axiomatic manipulation $\rightsquigarrow$ universality

- Question: haven't we already got enough ... this is *already* quite difficult!
- Answer: NAND and NOR turn out to be **functionally complete** (or **universal**), e.g.,

$$
\begin{array}{lclcl}
\neg x & & & \equiv & x \overline{\wedge} x \\
x \wedge y & & & \equiv & (x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y) \\
x \vee y & \equiv & \neg x \overline{\wedge} \neg y & \equiv & (x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)
\end{array}
$$

which we can prove via

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

- Eureka: computation of *any* Boolean function can be expressed using *one* simple building block component.

Notes:

Boolean algebra: theory $\neq$ practice (7)

Axiomatic manipulation $\rightsquigarrow$ universality

- Question: translate

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

into a version using NAND only.

Notes:

Boolean algebra: theory ≠ practice (7)

Axiomatic manipulation ⇝ universality

- Question: translate

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

into a version using NAND only.

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

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

Notes:

Boolean algebra: theory ≠ practice (7)

Axiomatic manipulation ⇝ universality

- Question: translate

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

into a version using NAND only.

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

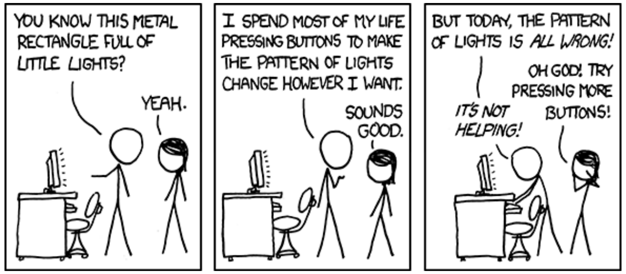
$$\begin{array}{ll} & x \wedge (y \vee z) \\ = & x \wedge ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z)) \\ = & t \overline{\wedge} t \end{array}$$

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

Notes:

Boolean algebra: theory ≠ practice (8)
Axiomatic manipulation ⇝ universality

- ... which leads neatly to lab. worksheet #1:



http://xkcd.com/722/

Notes:

Boolean algebra: theory ≠ practice (9)
Utilising design automation tools

Listing

```
from sympy                 import *
from sympy.logic.boolalg import simplify_logic

a, b, c, d, e = symbols( "a b c d e" )

f = ( ~( a | b ) & ~( c | d | e ) ) | ~( a | b )

print f
print simplify_logic( f )
print

f = ( a & b & c ) | ( ~a & b ) | ( a & b & ~c )

print f
print simplify_logic( f )
print
```

Notes:

- Note that this example can be explored online via
  http://live.sympy.org
- The underlying point here is that it is often fine to use a computer to help, but only if you really understand the underlying theory: for more complex designs, this becomes a more important of course.
- Seymour Cray (a *super*computer architect) produced some interesting anecdotes about use of CAD tools in his design process: see for example
  http://en.wikipedia.org/wiki/Seymour_Cray

Definition

Boolean algebra plays a role in many programming constructs:

r is x	$\equiv$	$r = x$	$\cong$	`r == x`	$\cong$	`r = x`
r is NOT x	$\equiv$	$r = \neg x$	$\cong$	`r != x`	$\cong$	`r = ~x`
r is x NAND y	$\equiv$	$r = x \overline{\wedge} y$	$\cong$	`r != x && y`	$\cong$	`r = ~( x & y )`
r is x NOR y	$\equiv$	$r = x \overline{\vee} y$	$\cong$	`r != x || y`	$\cong$	`r = ~( x | y )`
r is x AND y	$\equiv$	$r = x \wedge y$	$\cong$	`r == x && y`	$\cong$	`r = x & y`
r is x OR y	$\equiv$	$r = x \vee y$	$\cong$	`r == x || y`	$\cong$	`r = x | y`
r is x XOR y	$\equiv$	$r = x \oplus y$			$\cong$	`r = x ^ 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

```
void condition_v1( int a, int b, int c, int d ) {
  if( ( ( a + b ) > c ) && ( ( d % 2 ) == 0 ) ) {
    //         ( a + b is    >  c ) and     ( d is    even )

    printf( "if -> true  : %d %d %d %d\n", a, b, c, d );
  }
  else {
    //   not ( ( a + b is    >  c ) and     ( d is    even ) )
    //
    // = not   ( a + b is    >  c ) or  not ( d is    even )
    // =       ( a + b isn't >  c ) or      ( d isn't even )
    // =       ( a + b is    <= c ) or      ( d is    odd  )

    printf( "if -> false : %d %d %d %d\n", a, b, c, d );
  }
}
```

Notes:

Boolean algebra: theory ≠ practice (11)

Boolean algebra ⇝ *soft*ware design

Listing

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

Notes:

Boolean algebra: theory ≠ practice (12)

Boolean algebra ⇝ test and verification

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

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

  and manipulate it into a different form

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

  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)

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

  and manipulate it into a different form

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

  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. try all 2^n possible assignments, 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

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

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

Notes:

- SAT is NP-complete [7, 8] meaning that
  1. no known algorithm will efficiently solve every SAT instance, but
  2. other NP problems can be expressed as SAT instances.

Boolean algebra: theory ≠ practice (14)

Boolean algebra ⇝ test and verification

- **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.

Boolean algebra: theory ≠ practice (14)

Boolean algebra ⇝ test and verification

- **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.,

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

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.

Boolean algebra: theory ≠ practice (14)

- **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.

Boolean algebra: theory ≠ practice (14)

- **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

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

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.

Boolean algebra: theory ≠ practice (15)

Boolean algebra ⇝ test and verification

Listing

```
from sympy                  import *
from sympy.logic.inference import satisfiable

a, b, c, d, e = symbols( "a b c d e" )

f = ( b & ~a ) | ( a & ~b )
g = ( a | b ) & ~( a & b )

print f
print g
print satisfiable( f ^ g )
print
```

Notes:

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

Conclusions

Quote

In theory, there is no difference between theory and practice. But, in practice, there is.

– van de Snepscheut (http://en.wikiquote.org/wiki/Jan_L._A._van_de_Snepscheut)

- Take away points: being pragmatic, it should be clear
  1. "it works" ≠ "it works *well*",
  2. using automation is fine *iff.* you know the underlying theory,
  3. using brute-force is fine *iff.* you know the underlying theory,
  4. Boolean algebra > Boolean axioms: concepts that *seem* of interest in theory alone, *can* be important if/when applied in practice,
  5. computer architecture > hardware: Boolean algebra can also explain/support concepts in software.

Notes:

Additional Reading

- *Wikipedia: Boolean algebra.* URL: `http://en.wikipedia.org/wiki/Boolean_algebra`.
- D. Page. "Chapter 1: Mathematical preliminaries". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 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.* URL: `http://en.wikipedia.org/wiki/Boolean_algebra` (see p. 65).

[2] *Wikipedia: Common sub-expression elimination.* URL: `http://en.wikipedia.org/wiki/Common_subexpression_elimination` (see pp. 33, 35).

[3] *Wikipedia: Electronic Design Automation (EDA).* URL: `http://en.wikipedia.org/wiki/Electronic_design_automation` (see pp. 7, 9).

[4] D. Page. "Chapter 1: Mathematical preliminaries". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 65).

[5] W. Stallings. "Chapter 11: Digital logic". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013 (see p. 65).

[6] A.S. Tanenbaum and T. Austin. "Section 3.1: Gates and Boolean algebra". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 65).

[7] S. Cook. "The complexity of theorem proving procedures". In: *ACM Symposium on Theory of Computing (STOC)*. 1971, pp. 151–158 (see p. 52).

[8] L. Levin. "Universal search problems". In: *Problems of Information Transmission* 9.3 (1973), pp. 265–266 (see p. 52).

Notes: