CS/COE0447: Computer Organization and Assembly Language

Logic Design Introduction (Brief?)

Appendix B: The Basics of Logic Design

Dept. of Computer Science
University of Pittsburgh
Logic design?

- Digital hardware is implemented by way of *logic design*
- Digital circuits process and produce two discrete values: 0 and 1
- Example: 1-bit full adder (FA)
Layered design approach

- Logic design is done using **logic gates**
- Often we design a desired hardware function using high-level languages (HDLs) and somewhat higher level than logic gates
- Two approaches in design
  - Top down
  - Bottom up

We’ll do logic bottom up
Transistor as a switch

“N”-type TR

“P”-type TR

“ON”

G=1

G=0

“OFF”

G=0

G=1
An inverter

“A”-type TR

“A”-type TR

“A”-type TR

“A”-type TR
When $A = 1$
When $A = 0$

- **"P"-type TR**
  - A=0
  - "ON"
  - "0"

- **"N"-type TR**
  - "0"
  - "OFF"

- Y=1

"1"
Abstraction

“P”-type TR

A

“1”

“N”-type TR

“0”

A

Y

A

Y
Logic gates

2-input AND

2-input OR

2-input NAND

2-input NOR
Describing a function

- Output_A = F(Input_0, Input_1, …, Input_{N-1})
- Output_B = F'(Input_0, Input_1, …, Input_{N-1})
- Output_C = F''(Input_0, Input_1, …, Input_{N-1})
- ...

Methods
- Truth table
- Sum of products
- Product of sums
### Truth Table

#### Input

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C(_{in})</th>
<th>S</th>
<th>C(_{out})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Diagram

[Diagram of a simple circuit with inputs A and B, and outputs S and C\(_{out}\).]
Sum of products

\[
S = A'B'C_{in} + A'BC_{in} + AB'C_{in}' + ABC_{in}
\]

\[
C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}
\]

“Minterm”
NOT(A) AND NOT(B) AND Cin

OR two minterms
Combinational vs. sequential logic

- **Combinational logic** = *function*
  - A function whose outputs are dependent only on the current inputs
  - As soon as inputs are known, outputs can be determined

- **Sequential logic** = *combinational logic + memory*
  - Some memory elements (i.e., “state”)
  - Outputs are dependent on the current state and the current inputs
  - Next state is dependent on the current state and the current inputs
Combinational logic

Outputs are uniquely determined by the inputs at any moment
Combinational logic

Outputs are uniquely determined by the inputs at any moment
Sequential logic

delay (it takes time to compute, matched to clock)

Outputs are determined by current & past inputs (past is “state”)
Sequential logic

Outputs are determined by current & past inputs (past is “state”)
Combinational logic

- Any combinational logic can be implemented using sum of products (OR-AND) or product of sums (AND-OR)

- Input-output relationship can be defined in a truth table format
- From truth table, derive each output function
- And then we can derive a circuit!! Let’s try it!
  - Example: Write circuit for an 1-bit ADDER

- Boolean expressions can be further manipulated (e.g., to reduce cost) using various Boolean algebraic rules
Boolean algebra

- Boole, George (1815~1864): mathematician and philosopher; inventor of Boolean Algebra, the basis of all computer arithmetic

- Binary values: \{0,1\}
- Two binary operations: AND (\times/\cdot), OR (+)
- One unary operation: NOT (\sim)
Boolean algebra

- Binary operations: AND (×/·), OR (+)
  - Idempotent
    - \( a \cdot a = a + a = a \)
  - Commutative
    - \( a \cdot b = b \cdot a \)
    - \( a + b = b + a \)
  - Associative
    - \( a \cdot (b \cdot c) = (a \cdot b) \cdot c \)
    - \( a + (b + c) = (a + b) + c \)
  - Distributive
    - \( a \cdot (b + c) = a \cdot b + a \cdot c \)
    - \( a + (b \cdot c) = (a + b) \cdot (a + c) \)
Boolean algebra

- De Morgan’s laws
  - \( \neg(a \cdot b) = \neg a + \neg b \)
  - \( \neg(a+b) = \neg a \cdot \neg b \)

- More…
  - \( a+(a\cdot b) = a \)
  - \( a\cdot(a+b) = a \)
  - \( \neg\neg a = a \)
  - \( a+\neg a = 1 \)
  - \( a\cdot(\neg a) = 0 \)

It is not true I ate the sandwich and the soup.

same as:
I didn’t eat the sandwich or I didn’t eat the soup.

It is not true that I went to the store or the library.

same as:
I didn’t go to the store and I didn’t go to the library.
Expressive power

- With AND/OR/NOT, we can express any function in Boolean algebra
  - Sum (+) of products (·)

- What if we have NAND/NOR/NOT?
- What if we have NAND only?
- What if we have NOR only?
Using NAND only

\[ \neg(A \land A) = \neg A \]

\[ \neg(\neg(A \land B)) = A \land B \]

\[ \neg(\neg A \land \neg B) = A + B \]
Using NOR only (your turn)

- Can you do it?
- NOR is \( \neg(A + B) \)
  - I.e., We need to write NOT, AND, and OR in terms of NOR

<table>
<thead>
<tr>
<th>NOT</th>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg(A + A) )</td>
<td>( \neg(\neg(A + A) + \neg(B + B)) )</td>
<td>( \neg(\neg(A + B) + \neg(A + B)) )</td>
</tr>
<tr>
<td>( \neg A \land \neg A )</td>
<td>( \neg (\neg A \land \neg A + \neg B \land \neg B) )</td>
<td>( (A + B) \land (A + B) )</td>
</tr>
<tr>
<td>( \neg A )</td>
<td>( \neg(\neg A \land \neg B) )</td>
<td>( A + B )</td>
</tr>
<tr>
<td>( \neg(\neg A) \land \neg(\neg B) )</td>
<td>( A \land B )</td>
<td></td>
</tr>
</tbody>
</table>
Using NOR only (your turn)
Now, it’s really your turn….

- How about XOR?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ C = A'B + AB' \]
Now, it’s really your turn….

- How about XOR?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ C = A'B + AB' \]
Simplifying expressions

- \( C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in} \)
- \( C_{out} = BC_{in} + AC_{in} + AB \)

- Simplification reduces complexity: faster, smaller circuit!
Karnaugh map

\[ C_{out} = A'B'C_{in} + AB'C_{in} + ABC_{in}' + ABC_{in} \]

A “tool” to help simplify boolean expressions
Like a “slide rule”: Useful but limited

A truth table listing “minterms”
Minterms written in Gray code order
One var value changes between col/row

Build from the initial boolean expression
Put a “1” where a minterm is true
E.g., \( AB'C_{in} \) has a 1

Now, to simplify:
Look for adjacent max rectangular groups with power of 2 elements.
In such a group, some var is \( \{0,1\} \)
Eliminate that variable

Here’s another one!
Groups can be vertical too.
They can even “wrap around”
They can also overlap
Diagonals aren’t allowed
- $C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}$
- $S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$

$$\begin{array}{c|c|c}
A & BC_{in} & C_{out} \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}$$

$$\begin{array}{c|c|c}
A & BC_{in} & S \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}$$
Four (or more?) Variables

Can you minimize this one?

In AB: B is both \{0,1\}
In CD: C is both \{0,1\}

Eliminate B, C
Thus, we have just AD

Can you minimize this one?

C,D both have \{0,1\}
A has \{0,1\}

Eliminate A,C,D
Thus, we have just B
Four (or more?) Variables

<table>
<thead>
<tr>
<th>AB</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Can you minimize this one?

Combine on top row
Combine on bottom row

A’B’D’
AB’D’

These terms can now combine
Thus, we have B’D’

Karnaugh Maps (K-Maps) are a simple calculation tool.

In practice, sophisticated logic synthesis algorithms/tools are used.
In-class Example

- A device called a “7 segment LED digit”
- There are 8 LEDs – one for seven “segments” of a numeral and 1 for a decimal point

Problem
- Given a 3-bit number, draw the corresponding numeral
- E.g., 000 is the numeral 0, 001 is numeral 1 and so forth

Solution
- Create a Boolean function for each segment. Ignore the decimal point.
- Boolean function over three inputs for the 3-bit number.

Let’s try it!!
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 0
0 1 1 1 0 1 1 1
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 1
0 0 0 1 0 1 0 0

d0  d7
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 2
1 0 1 1 0 0 1 1
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 3
1 0 1 1 0 1 1 0

d0  d7
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 4
1 1 0 1 0 1 0 0

d0    d7
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 5
1 1 1 0 0 1 1 0

d0       d7
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 6
1 1 0 0 0 1 1 1
Hex Digit LED
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
“Draw” numbers 0 to 9

Numeral 7
0 0 1 1 0 1 0 0
In-class Example

- Create a truth table
- Inputs are numbered i0 to i2 (3 bits)
- Outputs are numbered d0 to d7, corresponding to segments
- “Draw” the numerals by setting d0 to d7 to 1s or 0s.
### Inputs

<table>
<thead>
<tr>
<th>i2</th>
<th>i1</th>
<th>i0</th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Input:** 3-bit number

**Outputs:** Segments for the LED hex digit
Fill in the truth table for each numeral
Numerals 0 to 2 are shown.
*Can you complete 3 to 7?*
Completed truth table
Now, write down the *minimal* (simplified) Boolean functions
Use a K-map to minimize each one!
<table>
<thead>
<tr>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2</td>
<td>i1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Completed truth table
Now, write down the minimal (simplified) Boolean functions
Use a K-map to minimize each one!
Use a K-map for each output function – d0 to d7

Let’s start with d0
We’ll only do a few – d0, d3 and d5

*Can you do the rest on your own??*
Function $d_0$

<table>
<thead>
<tr>
<th>$i_2$</th>
<th>$i_1$, $i_0$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i_2$</th>
<th>$i_1$</th>
<th>$i_0$</th>
<th>$d_0$</th>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
<th>$d_6$</th>
<th>$d_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Function $d_0$

$$d_0 = i_2' i_1 + i_2 i_1' + i_2 i_0$$
**Function d3**

<table>
<thead>
<tr>
<th>i2</th>
<th>i1, i0</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i2</th>
<th>i1</th>
<th>i0</th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

CS/CoE1541: Intro. to Computer Architecture
### Function $d_3$

<table>
<thead>
<tr>
<th>$i_2$</th>
<th>$i_1$</th>
<th>$i_0$</th>
<th>$d_0$</th>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
<th>$d_6$</th>
<th>$d_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$$d_3 = \overline{i_2} + \overline{i_1i_0} + i_1i_0$$
### Function $d_5$

<table>
<thead>
<tr>
<th>i2</th>
<th>i1, i0</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i2</th>
<th>i1</th>
<th>i0</th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Function $d_5$

$$d_5 = \overline{i_1} + i_0 + i_2$$
Completed Circuit with all functions d0 to d7

Outputs to the LED hex digit

See example: LEDhexdigit.circ
Multiplexor (aka MUX)
An example, yet VERY useful circuit!

\[
Y = \begin{cases} 
S \cdot A + S \cdot B & \\
S = \emptyset & 
\end{cases}
\]

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

when S =
0: output A
1: output B
A 32-bit MUX

Use 32 1-bit muxes
Each mux selects 1 bit
S is connected to each mux

a. A 32-bit wide 2-to-1 multiplexor
b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors
Building a 1-bit ALU

- ALU = arithmetic logic unit = arithmetic unit + logic unit
Building a 32-bit ALU
Implementing “sub”

Binvert=1
CarryIn=1 for 1st 1-bit ALU
Operation=2
Implementing NAND and NOR

NOR:
NOT (A OR B)
by DeMorgan’s Law:
(NOT A) AND (NOT B)

Thus,
Operation=0,
Ainvert=1,
Binvert=1

And, NAND???
Implementing SLT (set-less-than)

1-bit ALU for bits 0~30

1-bit ALU for bit 31
Implementing SLT (set-less-than)

SLT uses subtraction
slt $t0,$t1,$t2
$t1<$t2: $t1$-$t2$ gives negative result
set is 1 when negative

Setting the control
perform subtraction (Cin=1,Binvert=1)
select Less as output (Operation=3)
ALU31’s Set connected to ALU0 Less

Consider
Suppose $t1=10$ and $t2=11$

$t1$ - $t2$ = -1 = 1111...1 binary
$t0$ = 0000...1
Implementing SLT (set-less-than)

SLT uses subtraction

\[ \text{slt } \$t0, \$t1, \$t2 \]

\$t1<\$t2: \$t1-\$t2 \text{ gives negative result}
set is 1 when negative

Setting the control

perform subtraction (Cin=1, Binvert=1)
select Less as output (Operation=3)
ALU31’s Set connected to ALU0 Less

Why do we need Set? Could we use just the Result31?
Supporting BEQ and BNE

BEQ uses subtraction
beq $t0,$t1,LBL
perform $t0$-$t1
result=0 $\rightarrow$ equality

Setting the control
subtract (Cin=1,Binvert=1)
select result (operation=2)
detect zero result

“zero detector”
Abstracting ALU

- Note that ALU is a combinational logic