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 desired function using high-level languages and somewhat higher level than logic gates
- Two approaches in design
  - Top down
  - Bottom up

Transistor as a switch

- “N”-type TR
- “P”-type TR
An inverter

When A = 1

"P"-type TR

A

"1"

"0"

"N"-type TR

A=Y

ON

When A = 0

"P"-type TR

A=0

"1"

"0"

"N"-type TR

A=Y

OFF

Abstraction

"P"-type TR

A

"1"

"0"

"N"-type TR

A=Y

Y

Y

"0"
Logic gates

2-input AND
\[ Y = A \land B \]

2-input OR
\[ Y = A \lor B \]

2-input NAND
\[ Y = \overline{A \land B} \]

2-input NOR
\[ Y = \overline{A \lor B} \]

Describing a function

- \( \text{Output}_A = F(\text{Input}_0, \text{Input}_1, ..., \text{Input}_{N-1}) \)
- \( \text{Output}_B = F'(\text{Input}_0, \text{Input}_1, ..., \text{Input}_{N-1}) \)
- \( \text{Output}_C = F''(\text{Input}_0, \text{Input}_1, ..., \text{Input}_{N-1}) \)
- ...

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

Truth table

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</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>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</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>

Sum of products

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

Sequential logic

- Any combinational logic can be implemented using sum of products or product of sums
- Input-output relationship can be defined in the truth table format
- From the truth table, each output function can be derived
- 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 (~)

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

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

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

Expressive power

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

- What if we have NAND/NOR/NOT?
- What if we have NAND only?
- What if we have NOR only?
**Multiplexor (aka MUX)**

\[
Y = (S) \? B : A;
\]

**A 32-bit MUX**

- A 32-bit MUX

**Simplifying expressions**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>(B)</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>

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

**Karnaugh map**

\[
\begin{array}{c|c|c|c}
\hline
BC_{in} & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 1 & 1 \\
\hline
\end{array}
\]

- \(C_{out} = BC_{in} + AB + AC_{in}\)
Building a 1-bit ALU

- ALU = arithmetic logic unit = arithmetic unit + logic unit

Building a 32-bit ALU

Implementing “sub”

Implementing NAND and NOR
Implementing SLT (set-less-than)

1-bit ALU for bits 0~30

1-bit ALU for bit 31

Supporting BEQ and BNE

Abstracting ALU

- Note that ALU is a combinational logic

ALU operation
RS latch

- Beware of the feedback!

When R=0, S=1

- When R=1, S=0

- When R=0, S=0, and Q was 0
When $R=0$, $S=0$, and $Q$ was 1

What happens if $R=S=1$

Note that we have an RS latch in the back-end of this design

Note that $R$, $S$ inputs always get opposite values when $C=1$

When $C=0$, $S=R=0 \Rightarrow$ RS latch remembers the previous value
**D latch**

- **Latched mode**
  - Truth table:
    | C | D | Q(t) |
    |---|---|------|
    | 0 | 0 | Q(t-1) |
    | 0 | 1 | Q(t-1) |
    | 1 | 0 | 0 |
    | 1 | 1 | 1 |

- **Transparent mode**

**D flip-flop (D-FF)**

- Two cascaded D latches; C input of the second is inverted
- This is a negative edge triggered D-FF

**D latch**

**D flip-flop**
Finite state machine (FSM)

Traffic light control example

- Two states
  - NSlite: green light on North-South road
  - EWlite: green light on East-West road

- Current state goes for 30 seconds, then
  - Switch to the other state if there is a car waiting
  - Current state goes for another 30 seconds if not

- We use 1/30 Hz clock

Traffic light control example

<table>
<thead>
<tr>
<th>Current state</th>
<th>Input NSlite</th>
<th>Input EWlite</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSgreen</td>
<td>0</td>
<td>0</td>
<td>NSgreen</td>
</tr>
<tr>
<td>NSgreen</td>
<td>0</td>
<td>1</td>
<td>EWgreen</td>
</tr>
<tr>
<td>EWgreen</td>
<td>1</td>
<td>0</td>
<td>NSgreen</td>
</tr>
<tr>
<td>EWgreen</td>
<td>1</td>
<td>1</td>
<td>EWgreen</td>
</tr>
<tr>
<td>EWgreen</td>
<td>0</td>
<td>0</td>
<td>EWgreen</td>
</tr>
<tr>
<td>EWgreen</td>
<td>1</td>
<td>1</td>
<td>NSgreen</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current state</th>
<th>Output NSlite</th>
<th>Output EWlite</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSgreen</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>EWgreen</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Traffic light control example

- Let’s assign “0” to NSlite and “1” to EWlite initially
- NextState = CurrentState’-EWcar + CurrentState-NScarc
- NSlite = CurrentState’
- EWlite = CurrentState

How do I implement a logic design?

- Various ways to implement your design
  - Use 74* chips (you may need many of those for even a very simple design)
  - Use PLA (Programming Logic Array)
  - Use FPGA (Field Programmable Gate Array)
  - ASIC (Application Specific Integrated Circuits)
    - Most capable approach (in terms of design size and performance); however, there is a large fixed development cost.
    - This approach is justified only for volume products

Using 74* chips

Using 74* chips

“Breadboard”+74* chips
Using a PLA

- It’s a programmable device
  - You can implement your own (complex) combinational logic (=function) using a single PLA

- A PLA device is much more capable than individual 74* chips
  - Much denser (that means you can implement many functions)

- Internally, PLA implements a function of the “sum-of-product” format
  - “AND” plane – for product terms
  - “OR” plane – for summing products

FPGA

- FPGA is even more capable (millions of gates)
  - Combinational logic
  - Sequential logic
  - Memory arrays
  - Even some embedded microprocessor cores
  - I/O support such as Ethernet MAC (media access control) block

- Best of all, it’s field programmable and affordable

To wrap up

- In digital logic, transistors are used as simple switches
- Logic gates are an abstraction of a transistor network

- A combinational logic block has inputs and outputs whose values are immediately determined as inputs become known
- A sequential logic block is composed of a combinational logic block and memory elements
To wrap up

- Boolean algebra provides a theoretical foundation for digital logic
- Starting from two transistors (N-type and P-type), we’ve built logic gates and more complex structures (bottom up)
- An ALU for the MIPS architecture has been built!

To wrap up

- Flip-flops (FFs) were used as a memory element
- A finite state machine (FSM) can be implemented using FFs and some combinational logic