Combinational Circuits: From Boolean Algebra to Gates

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.
6.004 Outline

- Module 1: RISC-V Assembly language programming (4 lectures)
- Module 2: (Digital) Logic design (8 lectures)
  - Boolean Algebra
  - Combinational circuits
  - (Clocked) Sequential circuits
  - Expressing logic designs in Bluespec
  - Logic synthesis:
    - From Bluespec to Logic circuits
    - From Logic circuits to standard gate libraries
  - Concurrency issues
- Module 3: RISC-V processors (6+1 lectures)
- Module 4: OS, I/O, Virtual memory (3 lectures)
- Module 5: Multicores (2 lectures)
A combinational circuit has binary inputs and outputs.

It represents a pure function:
\[ f: \text{Bool} \times \text{Bool} \times \text{Bool} \rightarrow \text{Bool} \times \text{Bool} \]

It has no memory or state, i.e., given the same input it produces the same output.
Three simple combinational circuits

- **NOT**
  
  \[
  \begin{array}{c|c}
  a & s \\
  \hline
  0 & 1 \\
  1 & 0 \\
  \end{array}
  \]

  \[s = \sim a\]

- **AND**
  
  \[
  \begin{array}{c|c|c|c}
  a & b & s \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
  \end{array}
  \]

  \[s = a \cdot b\]

- **OR**
  
  \[
  \begin{array}{c|c|c|c}
  a & b & s \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
  \end{array}
  \]

  \[s = a + b\]
Some other famous gates

- **NAND**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>

\[ s = \neg(a \cdot b) \]

- **NOR**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ s = \neg(a + b) \]

The functionality of these gates can be expressed using NOT, AND, and OR gates.
Exclusive OR (XOR): another famous gate

- XOR

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</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>

From the Truth Table XOR produces a 1 when either \((a=0)\) AND \((b=1)\) or \((a=1)\) AND \((b=0)\). Hence, \(a \oplus b = \neg a.b + a.\neg b\)

You can express XOR using NOT, AND, and OR gates
Nomenclature

- We use the words in each of the following categories interchangeably
  - combinational circuits, Boolean expressions, Boolean circuits
  - gate, Boolean operator

- We use variables to name wires in a combinational circuit
Truth Table

- The functionality of any combinational circuit can be defined using a Truth Table
- It lists all possible combinations of values for inputs and specifies the output for each input

What is the size of the truth table for an n input and m output function?

2^n m-bit rows

Truth Tables are not suitable for describing functions with large number of inputs
There is a Boolean expression corresponding to every truth table

- Write a *product term* using only AND and NOT corresponding to each row with a 1
- The final Boolean expression is the *sum* of all such product terms

\[ s_1 = (\neg a) \cdot (\neg b) \cdot (\neg c) + a \cdot (\neg b) \cdot (\neg c) + a \cdot (\neg b) \cdot c \]
\[ s_2 = (\neg a) \cdot b \cdot c + a \cdot (\neg b) \cdot (\neg c) + a \cdot b \cdot (\neg c) \]

**sum-of-products (SOP) representation**
A set of gates is considered *universal* if any Boolean function can be described using only those gates. Examples:

- \{AND, OR, NOT\}
- \{NAND\}
- \{NOR\}

Proof: Any Truth Table can be represented as a Sum of Product.
Binary Addition

- Addition in base 2 is performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>1110</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

Let us build a hardware adder
Half Adder

- Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>s</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean equations

\[
\begin{align*}
    s &= \neg a \cdot b + a \cdot \neg b \\
    &= a \oplus b \\
    c &= a \cdot b
\end{align*}
\]
Combinational Logic for an adder

- Full adder (FA): adds two one-bit numbers and a carry-in bit, and produces a sum bit and a carry-out bit; can be built using two HAs.

- Cascade FAs to perform binary addition

---

February 21, 2019
Describing a 32-bit adder alternatives

- Truth Table with $2^{64}$ rows and 33 columns (sum(32)+carry(1))!
- 32 sets of Boolean equations, where each set describes a FA
- Some notation to describe recurrences
  - $t_k = a_k \oplus b_k$
  - $s_k = t_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + c_k \cdot t_k$
- Circuit diagrams - tedious to draw

Such representations are too verbose and not useful when we want computers to simulate the behavior of the circuit, i.e., determine the output given an input.

We will use a programming language called Bluespec System Verilog (BSV) to express all circuits.
**Half Adder in Bluespec**

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

### Boolean equations

\[
\begin{align*}
    s &= \sim a \cdot b + a \cdot (\sim b) \\
    &= a \oplus b \\
    c &= a \cdot b
\end{align*}
\]

### Bluespec function

```plaintext
function ha(a, b);
    s = a ^ b;
    c = a & b;
    return {c, s};
endfunction
```

Not quite correct - needs type annotations
Half Adder \textit{corrected}

\begin{verbatim}
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
    Bit#(1) s;     a type declaration that says
    Bit#(1) c;     s is one bit wide
    Bit#(2) result; says that result is a two
    s = a ^ b;     bit vector
    c = a & b;
    result[0] = s; says that the zeroth bit of
    result[1] = c; result is the same as s
    return result;
endfunction
\end{verbatim}
More convenient syntax

function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
    Bit#(1) s = a ^ b;
    Bit#(1) c = a & b;
    return {c, s};
endfunction

- How big is \{c,s\}?
  2 bits

- Using \{c,s\} notation avoids the need to name intermediate results

A type of a variable can be declared at the same place where we define it. \{...\} can be used to define a bit vector.
Full Adder using HAs

```plaintext
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
    Bit#(2) ab = ha(a, b);
    Bit#(2) abc = ha(ab[0], c_in);
    Bit#(1) c_out = ab[1] | abc[1];
    return {c_out, abc[0]};
endfunction
```

- `ha` is being used as a black-box;
- `fa` code is simply a wiring diagram

Extracts the sum bit

Extracts the carry bit
2-bit Ripple-Carry Adder
cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
Bit#(2) s = 2b’00;
Bit#(3) c = 3b’000;
c[0] = 0;
Bit#(2) cs0 = fa(x[0], y[0], c[0]);
s[0] = cs0[0];
Bit#(2) cs1 = fa(x[1], y[1], c[1]);
s[1] = cs1[0];
return {c[2],s};
endfunction

Use fa as a black-box

s has two wires, initially each s wire is zero
wire s[0] is updated
wire s[1] is updated
The same as writing
{c[2],s[1],s[0]};
Selectors and Multiplexers
Selecting a wire: $x[i]$

- **Constant selector:** e.g., $x[2]$

  - No hardware; $x[2]$ is just the name of a wire

- **Dynamic selector:** $x[i]$

  - 4-way mux
A 2-way multiplexer

A mux is a simple conditional expression

Bluespec  
\[(p)? b : a ;\]

Python  
```
return b if p else a
```

True is treated as a 1 and False as a 0

Gate-level implementation

If \(a\) and \(b\) are \(n\)-bit wide then this structure is replicated \(n\) times; \(p\) is the same input for all the replicated structures
A 4-way multiplexer

```python
def mux(a, b, c, d, s):
    if s == 0:
        return a
    elif s == 1:
        return b
    elif s == 2:
        return c
    else:
        return d
```

The syntax for the multiplexer can be defined as:

```plaintext
case ({s1, s0})
  2'b00 : a;
  2'b01 : b;
  2'b10 : c;
  2'b11 : d;
endcase
```

This can also be implemented using `n-1` two-way multiplexers:

```
(s1==0) & (s0==1),
which is the same writing ~s1 & s0
```

This is because a `n-way` multiplexer can be implemented using `n-1` two-way multiplexers.
Next lecture

- More simple-combinational circuits
- Logic synthesis and circuit optimization
Take Home

1. Provide a truth table for a full adder (fa)
2. Specify a Sum of Products (SOP) representation for each of the full adder outputs.