Describing Combinational circuits in BSV

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.
Boolean Algebra and Arithmetic

- Numbers can be represented in binary (base 2) notation and arithmetic can be performed on binary numbers
- Binary arithmetic has one-to-one correspondence with Boolean algebra, i.e., operations on Boolean expressions
- Thus, all arithmetic operations can be expressed as Boolean or combinational circuits!
Encoding Positive Integers

- Bit $i$ in a binary representation (in the right-to-left order) is assigned the weight $2^i$

- Value of an $N$-bit number is given by the formula

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

- What value does $011111010000$ encode?

$$V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + \ldots$$
$$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$$

Smallest number? 0  Largest number? $2^{N-1}$
Binary Addition

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

Base 10

\[ 14 + 7 = 21 \]

Base 2

\[ 1110 + 111 = 10101 \]

Let us build a hardware adder
Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
2. Full adder (FA): adds two one-bit numbers and a carry, and produces a sum bit and a carry bit
   Can be built using two HAs
3. Cascade FAs to perform binary addition

\[
\begin{align*}
    a_3 &\rightarrow b_3 & a_2 &\rightarrow b_2 & a_1 &\rightarrow b_1 & a_0 &\rightarrow b_0 \\
    c_4 &\rightarrow & c_3 &\rightarrow & c_2 &\rightarrow & c_1 &\rightarrow & c_0 (=0)
\end{align*}
\]
Describing a 32-bit adder alternatives

- Truth Table with $2^{32}$ rows and two columns (sum and carry)!
- 32 sets of Boolean equations, where each set describes a FA
- Some notation to describe recurrences
  \[
  \begin{align*}
  t_k &= a_k \oplus b_k \\
  s_k &= t_k \oplus c_k \\
  c_{k+1} &= a_k \cdot b_k + c_k \cdot t_k
  \end{align*}
  \]
  \[
  0 \leq k \leq 31
  \]
- 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 BSV

<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:

\[
s = \neg a \cdot b + a \cdot \neg b = a \oplus b \\
c = a \cdot b
\]

```groovy
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 ^ b;
    Bit#(1) c = a & b;
    return \{c,s\};
endfunction
\end{verbatim}

- "Bit#(1) a" type declaration says that a is one bit wide
- \{c,s\} represents bit concatenation
- How big is \{c,s\}?
  \begin{itemize}
    \item 2 bits
  \end{itemize}
Suppose we write \( t = ha(a, b) \) then \( t \) is a two bit quantity representing \( s \) and \( c \) values.

- We can recover \( s \) and \( c \) values from \( t \) by writing \( t[0] \) and \( t[1] \), respectively.

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

\( ha \) can be used as a black-box as long as we understand its type signature.
Full Adder using HAs

\[
\text{function } \text{Bit}(2) \ fa(\text{Bit}(1) \ a, \ \text{Bit}(1) \ b, \ \text{Bit}(1) \ c_{\text{in}});
\]

\[
\begin{align*}
\text{Bit}(2) \ ab &= \ ha(a, \ b); \\
\text{Bit}(2) \ abc &= \ ha(ab[0], \ c_{\text{in}}); \\
\text{Bit}(1) \ c_{\text{out}} &= ab[1] \mid abc[1]; \\
\text{return} \ \{c_{\text{out}}, \ abc[0]\};
\end{align*}
\]

\text{endfunction}

ha is being used as a black-box; fa code is simply a wiring diagram
The "let" syntax

No need to write the type if the compiler can deduce it
2-bit Ripple-Carry Adder
cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);

Bit#(2) s = 0;
Bit#(3) c = 0;
c[0] = 0;
let cs0 = fa(x[0], y[0], c[0]);
s[0] = cs0[0];
c[1] = cs0[1];
let cs1 = fa(x[1], y[1], c[1]);
s[1] = cs1[0];
c[2] = cs1[1];
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
Assigning to Multi-Bit Words

```plaintext
Bit #(3) c = 0;

c0 = c[0];
```

- Means `c` is three bits wide and each element is set to zero

- `c0` is assigned to element 0 of `c`, but the values of the rest of the elements are not affected

- Each bit in a multi-bit word must have an initial value
  - An attempt to use uninitialized bits will raise a compiler warning and result in unexpected behavior
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

- We can also select multiple bits: e.g., $x[2:1]$ means $\{x[2], x[1]\}$

- **Dynamic selector:** $x[i]$
  - 4-way mux
A 2-way multiplexer

A mux is a simple conditional expression

**BSV**  
\[(s==0)? a : b ;\]

**Python**  
\[a if s == 0 else b\]

Gate-level implementation

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

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

n-way mux can be implemented using n-1 two-way muxes
Shift operators
Logical right shift by 2

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately
- Rotates and arithmetic shifts are similar

Rotate: $a \ b \ c \ d$

Arithmetic: $a \ b \ c \ d$

useful for multiplication and division by $2^n$
Logical right shift by \( n \)

- Suppose we want to build a shifter which shifts a value \( x \) by \( n \) where \( n \) is between 0 and 31
- One way to do this is by connecting 31 different shifters via a mux

How many 2-way one-bit muxes are needed to implement this structure?

\[ n \times (n-1) \]

Can we do better?
Logical right shift by $n$

- Shift $n$ can be broken down into log $n$ steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 ($=2+1$) by doing shifts of size 2 and 1
  - Shift 5 ($=4+1$) by doing shifts of sizes 4 and 1
  - Shift 21 ($=16+4+1$) by doings shifts of sizes 16, 4 and 1
- For a 32-bit number, a 5-bit $n$ can specify all the needed shifts
  - $3_{10} = 00011_2$, $5_{10} = 00101_2$, $21_{10} = 10101_2$
- The bit encoding of $n$ tells us which shifters are needed; if the value of the $i^{th}$ (least significant) bit is 1 then we need to shift by $2^i$ bits
Conditional operation: shift versus no-shift

- We need a mux to select the appropriate wires: if \( s \) is one the mux will select the wires on the left otherwise it would select wires on the right

\[
(s==0)?\{a,b,c,d\}:\{2'b0,a,b\};
\]
Logical right shift circuit

- Define \( \log n \) shifters of sizes 1, 2, 4, ...
- Define \( \log n \) muxes to perform a particular size shift
- Suppose \( s = \{s_1,s_0\} \) is a two bit number. Shift circuit to shift a number by \( s \) can be expressed as two nested conditionals expressions
Lecture L04: Binary Arithmetic
Lecture L05: Complex combinational circuits in Bluespec