Constructive Computer Architecture

Combinational circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
A combinational circuit is a pure function; given the same input, it produces the same output.

It can be described using a Truth Table though it is not practical to do so for a large number of inputs.

- Size of truth table for a 32-bit adder? $2^{32+32}$ rows

We will use a programming language called Bluespec System Verilog (BSV) to express all ckt's.
Half Adder

<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**

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

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

"Bit#(1) a" type declaration says that a is one bit wide

\{c, s\} represents bit concatenation

How big is \{c, s\}?

2 bits
\end{verbatim}
BSV notes

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

Suppose we write \( t = ha(a, b) \) then \( t \) is a two bit quantity representing \( c \) and \( s \) values.

We can recover \( c \) and \( s \) values from \( t \) by writing \( t[1] \) and \( t[0] \), respectively.
Full Adder
1-bit adder with a carry-in input

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
The “let” syntax

No need to write the type if the compiler can deduce it

```verilog
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);

let ab = ha(a, b);
let abc = ha(ab[0], c_in);
let c_out = ab[1] | abc[1];
return {c_out, abc[0]};
endfunction
```
Types

A type is a grouping of values:

- Integer: 1, 2, 3, ...
- Bool: True, False
- Bit: 0, 1
- A pair of Integers: Tuple2#(Integer, Integer)
- A function $fname$ from Integers to Integers:
  
  ```
  function Integer $fname$ (Integer arg)
  ```

Every expression in a BSV program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler.

Thus, we say an expression has a type or belongs to a type.

The type of each expression is unique.
Parameterized types: #

- A type declaration itself can be parameterized by other types.
- Parameters are indicated by using the syntax `#`
  - For example, `Bit#(n)` represents n bits and can be instantiated by specifying a value of n.
    - `Bit#(1), Bit#(32), Bit#(8), ...`
Type synonyms

```haskell
typedef bit [7:0] Byte;
typedef Bit#(8) Byte;

typedef Bit#(32) Word;

typedef Tuple2#(a,a) Pair#(type a);

typedef Int#(n) MyInt#(numeric type n);

In some special cases one can just write:

typedef Int#(n) MyInt#(type n);
```
The programmer writes down types of some expressions in a program and the compiler deduces the types of the rest of expressions. If the type deduction cannot be performed or the type declarations are inconsistent then the compiler complains.

```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#(2) c_out = ab[1] | abc[1];
return {c_out, abc[0]};
endfunction
```

Type checking prevents lots of silly mistakes.
2-bit Ripple-Carry Adder

cascading full adders

\[
\begin{align*}
&\text{function } \text{Bit\#(3) } \text{add}(\text{Bit\#(2) } x, \text{ Bit\#(2) } y, \\
&\quad \text{Bit\#(1) } c0); \\
&\quad \text{Bit\#(2) } s = 0; \quad \text{Bit\#(3) } c = 0; \quad c[0] = c0; \\
&\quad \text{let } cs0 = \text{fa}(x[0], y[0], c[0]); \\
&\quad \quad c[1] = cs0[1]; \quad s[0] = cs0[0]; \\
&\quad \text{let } cs1 = \text{fa}(x[1], y[1], c[1]); \\
&\quad \quad c[2] = cs1[1]; \quad s[1] = cs1[0]; \\
&\quad \text{return } \{c[2], s\}; \\
&\text{endfunction}
\end{align*}
\]

Initially

\(s\) wires are zero

Use \text{fa} as a black-box

Initially

\(s[0]\) wires are zero

wire \(s[0]\) is updated

wire \(s[1]\) is updated
Assigning to Vector elements

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

- Element 0 of $c$ is connected to $c_0$ but the value of the rest of the elements is not affected

- Initial value of a vector must be set; we use "?" if we don't know the initial value
An w-bit Ripple-Carry Adder

- For a w-bit adder, unless we know the value of w, we cannot write a straight-line program as we did for 2-bit adder.
- Use loops!

```plaintext
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);
    Bit#(w) s; Bit#(w+1) c=0; c[0] = c0;
    for(Integer i=0; i<w; i=i+1)
        begin
            let cs = fa(x[i],y[i],c[i]);
            c[i+1] = cs[1]; s[i] = cs[0];
        end
    return {c[w],s};
endfunction
```

There are some subtle type errors in this program but before we fix them, you may wonder what is the meaning of a loop in terms of gates?
All loops are unfolded by the compiler!

```plaintext
for (Integer i=0; i<w; i=i+1)
begin
  let cs = fa(x[i], y[i], c[i]);
  c[i+1] = cs[1]; s[i] = cs[0];
end

Can be done only when the value of w is known

cs0 = fa(x[0], y[0], c[0]); c[1]=cs0[1]; s[0]=cs0[0];
cs1 = fa(x[1], y[1], c[1]); c[2]=cs1[1]; s[1]=cs1[0];
...
csw = fa(x[valw-1], y[valw-1], c[valw-1]);
c[valw] = csw[1]; s[valw-1] = csw[0];
```
Loops to gates

Unfolded loop defines an acyclic wiring diagram

\[
\begin{align*}
\text{cs0} &= \text{fa}(x[0], y[0], c[0]); \quad c[1]=\text{cs0}[1]; \quad s[0]=\text{cs0}[0]; \\
\text{cs1} &= \text{fa}(x[1], y[1], c[1]); \quad c[2]=\text{cs1}[1]; \quad s[1]=\text{cs1}[0]; \\
&\vdots \\
\text{csw} &= \text{fa}(x[\text{valw}-1], y[\text{valw}-1], c[\text{valw}-1]); \\
\text{c[valw]} &= \text{csw}[1]; \quad \text{s[valw-1]} = \text{csw}[0];
\end{align*}
\]
Instantiating the parametric Adder

```verbatim
defunction Bit#(w+1) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);```

How do we define a `add32`, `add3` ... using `addN`?

```verbatim
// concrete instances of addN!
defunction Bit#(33) add32(Bit#(32) x, Bit#(32) y, Bit#(1) c0) = addN(x, y, c0);
```

The numeric type `w` on the RHS implicitly gets instantiated to 32 because of the LHS declaration

```verbatim
defunction Bit#(4) add3(Bit#(3) x, Bit#(3) y, Bit#(1) c0) = addN(x, y, c0);
```
Fixing the type errors

\texttt{valueOf}(w) \textit{versus} w

- Each expression has a type and a value, and these two come from entirely disjoint worlds.
- \texttt{w in Bit#(w)} resides in the types world.
- Sometimes we need to use values from the types world into actual computation. The function \texttt{valueOf} allows us to do that.
  - Thus
    
    \[
    i < \text{w} \text{ is not type correct}
    \]
    
    \[
    i < \text{valueOf}(w) \text{ is type correct}
    \]
Fixing the type errors

**TAdd\#(w, 1) versus w+1**

- Sometimes we need to perform operations in the types world that are very similar to the operations in the value world
  - Examples: `Add`, `Mul`, `Log`

- We define a few special operators in the types world for such operations
  - Examples: `TAdd\#(m, n)`, `TMul\#(m, n)`, ...
Fixing the type errors

**Integer versus Int#(32)**

- In mathematics integers are unbounded but in computer systems integers always have a fixed size.
- BSV allows us to express both types of integers, though unbounded integers are used only as a programming convenience.

```haskell
for (Integer i=0; i<valw; i=i+1)
begin
    let cs = fa(x[i],y[i],c[i]);
    c[i+1] = cs[1]; s[i] = cs[0];
end
```
A w-bit Ripple-Carry Adder

corrected

```haskell
function Bit#(TAdd#(w, 1)) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);

Bit#(w) s; Bit#(TAdd#(w, 1)) c; c[0] = c0;
let valw = valueOf(w);
for (Integer i = 0; i < valw; i = i+1)
begin
    let cs = fa(x[i], y[i], c[i]);
    c[i+1] = cs[1]; s[i] = cs[0];
end
return {c[valw], s};
endfunction
```

Structural interpretation of a loop – unfold it to generate an acyclic graph
BSV Compiling phases

- **Type checking**: Ensures that type of each expression can be determined uniquely; Otherwise the program is rejected

- **Static elaboration**: Compiler eliminates all constructs which have no direct hardware meaning
  - Loops are unfolded
  - Functions are in-lined; even recursive functions can be used as long as all the recursion can be gotten rid of at compile time
  - After this stage the program does not contain any Integers because Integers are unbounded in BSV

- Gates are generated (actually Verilog)
Takeaway

- Once we define a combinational ckt, we can use it repeatedly to build larger ckts.
- The BSV compiler, because of the type signatures of functions, prevents us from connecting them in obviously illegal ways.
- We can write parameterized ckts in BSV, for example an n-bit adder. Once n is specified, the correct ckt is automatically generated.
- Even though we use loop constructs and functions to express combinational ckts, all loops are unfolded and functions are inlined during the compilation phase.