Review: Representations for CL

- Use this diagram and techniques we learned to transform from one to another

In practice, useful only for small blocks.
- Cut block into smaller pieces (hierarchy)
- Rely on Computer Aided Design Tools (logic synthesis).
Arithmetic and Logic Unit

- Most processors contain a special logic block called “Arithmetic and Logic Unit” (ALU)
- We’ll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR

\[ \text{when } S=00, \ R=A+B \]
\[ \text{when } S=01, \ R=A-B \]
\[ \text{when } S=10, \ R=A \text{ AND } B \]
\[ \text{when } S=11, \ R=A \text{ OR } B \]
Adder/Subtractor Design -- how?
• Truth-table, then determine canonical form, then minimize and implement.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>000...0</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>000...1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000...00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000...01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 Functions with $2^{64}$ rows!</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>111...1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111...10</td>
</tr>
</tbody>
</table>

° Look at breaking the problem down into smaller pieces that we can assemble together. An adder has a natural structure to its design.

Adder/Subtractor – One-bit adder

LSB...

\[
\begin{array}{ccc}
a_3 & a_2 & a_1 \\
+ & b_3 & b_2 & b_1 \\
s_3 & s_2 & s_1 & s_0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
a_0 & b_0 & s_0 & c_1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

\[
s_0 = \\
c_1 =
\]
Adder/Subtractor – One-bit adder

(1/2)...

<table>
<thead>
<tr>
<th>a_i</th>
<th>b_i</th>
<th>c_i</th>
<th>s_i</th>
<th>c_{i+1}</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>

s_i =

C_{i+1} =

Adder/Subtractor – One-bit adder

(2/2)...

\[ s_i = \text{XOR}(a_i, b_i, c_i) \]
\[ c_{i+1} = \text{MAJ}(a_i, b_i, c_i) - a_i b_i + a_i c_i + b_i c_i \]
N 1-bit adders ® 1 N-bit adder

What about overflow? Would be nice/useful/mandatory to know when overflow occurs.

Overflow detection?
• Overflow never when A and B both have different sign.
• Adding two positive numbers:
  • Can never result in a carry out of the MSB position
  • A carry into the MSB stage turns the n-bit representation into a negative number ® result is too big and overflow occurred.
• Adding two negative numbers:
  • Always results in a carry out of the MSB position
  • No carry into the MSB stage turns the n-bit representation into a positive number ® overflow

° In either case, if carry in to the MSB stage is different from carry out, then overflow has occurred.

overflow = c_n xor c_{n-1}
Subtraction Based on Adder Circuit:

\[ A - B = A + (-B) = A + \sim B + 1 \]

XOR is a “Conditional” inversion, \( c_0 = 1 \) + 1

Verilog and Boolean Equations

• Besides the two major styles, structural and behavioral, a third style (somewhat in between) called dataflow

• Continuous Assignment statements

```verilog
// y = ab + ac + bc;
wire a, b, c, y;
assign y = a & b | a & c | b & c;
```

• Not like a normal assignment in programming languages. This assign happens continuously. Whenever anything on the RHS changes, the LHS is updated.
Verilog Continuous Assign

• The Boolean operators are defined to be bitwise:

```verilog
// Y = AB;
wire A[3:0], B[3:0];
assign Y = A & B;
// Or equivalently
assign Y[0] = A[0] & B[0];
```

Verilog Bitwise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>OR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>XOR</td>
<td>Bitwise Exclusive OR</td>
</tr>
<tr>
<td>XNOR</td>
<td>Bitwise Exclusive NOR</td>
</tr>
<tr>
<td>NOT</td>
<td>Inversion</td>
</tr>
</tbody>
</table>
### Concatenation in Verilog

wire A[7:0], B[7:0];
wire C[3:0], D[11:0];
wire Y[15:0];
assign Y = {A, B};
// Makes a longer bit vector from A with B

- **Concatenation on the LHS**

assign {C, D} = Y;
// Splits up Y into pieces

assign {C, D} = {A, B};
// Also works

assign {OPCODE, RS, RT, IMMED} = INSTRUCTION;
// Might be useful

### Replication

n{m}

- **Replicates m n-times.**

- **Example:**

wire X[31:0];
wire IMMED[15:0];
assign X = {16{IMMED[15]}, IMMED};
// Sign-extends IMMEDIATE into X

- **Concatenation, replication operations don’t generate any combinational logic, then only generate wires.**
**Conditional Operator**

When select=0, out=in0
When select=1, out=in1

assign out = select ? in1 : in0;
// if select is false (all 0)
// set out to in0, else set out to in1

- Works with bit vectors or expressions
- Note, conditional operator not bitwise.