Lecture 7 – Adders and Multipliers
Ripple Carry Adder

Key observations, the value of the carry into any stage of a multi-cell adder depends only on:

- The data bit of previous stage
- The carry into the 1st stage
- When both inputs 0, no carry
- When one is 0, the other is 1, propagate carry input
- When both are 1, then generate a carry

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>cin</th>
<th>cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Kill
- Propagate
- Generate
Carry-lookahead adder

- **Generate**
  \[ G_i = a_i \times b_i \]

- **Propagate**
  \[ P_i = a_i \oplus b_i; \text{ or } P_i = a_i + b_i \]

- **Pi and Gi are mutually exclusive**
  - Because \( P_i \) is asserted when \((a_i, b_i) = \{(1,0), (0,1)\}\);
  - \( G_i \) is asserted when \((a_i, b_i) = (1,1)\)

- \( S_i = a_i \oplus b_i \oplus c_i = P_i \oplus c_i \)

- \( c(i+1) = (a_i+b_i) \times c_i + a_i \times b_i = (a_i \oplus b_i) \times c_i + a_i \times b_i = P_i \times c_i + G_i \)

- **Write carry out as function of preceding G, P, and cout**
  \[ c_1 = G_0 + P_0 \times c_0 \]
  \[ c_2 = G_1 + P_1 \times c_1 \]
  \[ c_3 = G_2 + P_2 \times c_2 \]
  \[ c_4 = G_3 + P_3 \times c_3 \]
Reducing the complexity

- \( c_1 = G_0 + (P_0 \times c_0) \)
- \( c_2 = G_1 + (P_1 \times [G_0 + P_0 \times c_0]) \)
  \[ = G_1 + (P_1 \times G_0) + (P_1 \times P_0 \times c_0) \]
- \( c_3 = G_2 + (P_2 \times G_1) + (P_2 \times P_1 \times G_0) + (P_2 \times P_1 \times P_0 \times c_0) \)

That is \( c_i \) can only use \( c_0 \) and \( P(i-1) \) ... \( P_0 \) and \( G(i-1) \) ... \( G_0 \)

Increase speed at what cost?

Can you illustrate how to build a 32-bit adder with carry look ahead?
Carry Look Ahead Adder

\[
c_0 = \text{Cin}
\]

\[
c_1 = G_0 + c_0 \cdot P_0
\]

\[
c_2 = G_1 + G_0 \cdot P_1 + c_0 \cdot P_0 \cdot P_1
\]

\[
c_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + c_0 \cdot P_0 \cdot P_1 \cdot P_2
\]

\[
c_4 = \ldots
\]

**Truth Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Cin</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Cin</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

"kill"  
"propagate"  
"propagate"  
"generate"

\[
G = A \text{ and } B
\]

\[
P = A \text{ xor } B
\]
Multiply Overview

- Binary multiplication is just a *bunch* of left shifts and adds

A diagram is shown illustrating the process:

- The multiplicand and multiplier are aligned.
- The partial products are formed in parallel and added in parallel for faster multiplication.
- The double precision product is the result.
Division Overview

- Division is just a bunch of quotient digit guesses and right shifts and subtracts.
More complicated than addition
• accomplished via shifting and addition

More time and more area
• \( m \) bits \( \times \) \( n \) bits = \( m+n \) bit product

Let's look at 3 (unsigned) versions of multiplication designs in the next few slides
Unisigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit Adder, 64-bit Product reg, 32-bit multiplier reg

Multiplier = datapath + control
Multiply Algorithm Version 1

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to product & place the result in Product register
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit.
3. Shift the Multiplier register right 1 bit.

32nd repetition?
- No: < 32 repetitions
- Yes: 32 repetitions

Done

---

<table>
<thead>
<tr>
<th>Product</th>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0011</td>
<td>0000 0010</td>
</tr>
<tr>
<td>1: 0000 0010</td>
<td>0011</td>
<td>0000 0010</td>
</tr>
<tr>
<td>2: 0000 0010</td>
<td>0011</td>
<td>0000 0100</td>
</tr>
<tr>
<td>3: 0000 0010</td>
<td>0001</td>
<td>0000 0100</td>
</tr>
<tr>
<td>1: 0000 0110</td>
<td>0001</td>
<td>0000 0100</td>
</tr>
<tr>
<td>2: 0000 0110</td>
<td>0001</td>
<td>0000 1000</td>
</tr>
<tr>
<td>3: 0000 0110</td>
<td>0000</td>
<td>0000 1000</td>
</tr>
<tr>
<td>0000 0110</td>
<td>0000</td>
<td>0000 1000</td>
</tr>
</tbody>
</table>
Observations on Multiply Version 1

- 1 clock per cycle => \( \approx 100 \) clocks per multiply because of 32 repetitions, 3 steps in one repetition
  - Ratio of add/sub to multiply is from 5:1 to 100:1
  - Slow

- 0’s inserted in the rightmost bit of multiplicand as shifting left
  => least significant bits of product never changed once formed

- 1/2 bits in multiplicand always 0
  - MSB are 0s at the beginning
  - 0 is inserted in LSB as multiplicand shifting left

  => 64-bit multiplicand register is wasted
  => 64-bit adder is wasted

- Instead of shifting multiplicand to left, let’s shift \textit{product} to right
MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg
Multiply Algorithm Version 2

1. Test Multiplier0
   - Multiplier0 = 1
     - 1a. Add multiplicand to the left half of product & place the result in the left half of Product register
   - Multiplier0 = 0

2. Shift the Product register right 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?
- No: < 32 repetition
- Yes: 32 repetitions

Done
Still more wasted space in Version 2

1. Test Multiplier
   - Multiplier0 = 1
     - 1a. Add multiplicand to the left half of product & place the result in the left half of Product register
   - Multiplier0 = 0

2. Shift the Product register right 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?
- No: < 32 repetition
- Yes: 32 repetitions
  - Done

Product | Multiplier | Multiplicand
--------|------------|------------
0000 0000 | 0011 | 0010
1: 0010 0000 | 0011 | 0010
2: 0001 0000 | 0011 | 0010
3: 0001 0000 | 0001 | 0010
1: 0011 0000 | 0001 | 0010
2: 0001 1000 | 0001 | 0010
3: 0001 1000 | 0000 | 0010
1: 0001 1000 | 0000 | 0010
2: 0000 1100 | 0000 | 0010
3: 0000 1100 | 0000 | 0010
1: 0000 1100 | 0000 | 0010
2: 0000 0110 | 0000 | 0010
3: 0000 0110 | 0000 | 0010
0000 0110 | 0000 | 0010

Product   Multiplier   Multiplicand
Observations on Multiply Version 2

- Product register *wastes space that exactly matches size of multiplier*
  
=> combine Multiplier register and Product register
MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)
Multiply Algorithm Version 3

Start

1. Test Product0

Product0 = 1

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

2. Shift the Product register right 1 bit

32nd repetition?

Yes: 32 repetitions

Done

No: < 32 repetition

Product   Multiplicand
0000 0011  0010
1: 0010 0011  0010
2: 0001 0001  0010
1: 0011 0000  0010
2: 0001 1000  0010
1: 0001 1000  0010
2: 0000 1100  0010
1: 0000 1100  0010
2: 0000 0110  0010

0000 0110  0010
Multiplier - Datapath and Control

- Controller
  - Start
  - Control signals: Ready, Multiplicand, Multiplier
  - Data signals: Load_regs, Shift_regs, Add_regs, Decr_P
  - Internal signals: Zero, clock, reset
  - Outputs: Q[0], Product

- Datapath
  - Registers: A, B, Q, C, P

- Register A (Sum)
  - Bits: 16, 15, 8, 8, 7, 0
  - Values: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1

- Register B (Multiplicand)
  - Bits: 7, 8, 0
  - Values: 1, 1, 1, 0, 1, 0, 1, 1, 1

- Register P (Counter)
  - Bits: 0
  - Values: 1, 0, 0, 0
ASMD Chart for Binary Multiplier

- **Start**: Reset counter
- **S_idle**: $A \leq 0$, $C \leq 0$, $B \leq \text{Multiplier}$, $Q \leq \text{Multiplier}$, $P \leq n$

- **S_add**: $P \leq P-1$, Decrement counter, $[C, A] \leq A + B$, Add multiplicand to shifted sum

- **S_shift**: $[C, A, Q] \leq [C, A, Q] >> 1$, 17-bit register shifts to the right by one bit

- **Loadregs**: $A \leq 0$, $C \leq 0$, $B \leq \text{Multiplier}$, $Q \leq \text{Multiplier}$, $P \leq n$

- **S_add Decr_P**: $P \leq P-1$, $[C, A] \leq A + B$

- **Add_regs**: $[C, A, Q] \leq [C, A, Q] >> 1$

- **Zero**
Control Specification

(a) State Transition Diagram

<table>
<thead>
<tr>
<th>State Transition</th>
<th>Register Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
</tr>
<tr>
<td>$S_{idle}$</td>
<td>$S_{add}$</td>
</tr>
<tr>
<td>$S_{idle}$</td>
<td>$S_{add}$</td>
</tr>
<tr>
<td>$S_{add}$</td>
<td>$S_{shift}$</td>
</tr>
<tr>
<td>$S_{shift}$</td>
<td></td>
</tr>
</tbody>
</table>

- Initial state
- $A \leq 0, C \leq 0, P \leq dp\_width$
- $P \leq P - 1$
- if $(Q[0])$ then $(A \leq A + B, C \leq C_{out})$
- shift right $[CAQ], C \leq 0$
module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);
// Default configuration: five-bit datapath
parameter dp_width = 5; // Set to width of datapath
output [2*dp_width -1: 0] Product;
output Ready;
input [dp_width -1: 0] Multiplicand, Multiplier;
input Start, clock, reset_b;

parameter BC_size = 3; // Size of bit counter
parameter S_idle = 3'b001, // one-hot code
    S_add = 3'b010,
    S_shift = 3'b100;
reg [2: 0] state, next_state;
reg [dp_width -1: 0] A, B, Q; // Sized for datapath
reg C;
reg [BC_size -1: 0] P;
reg Load_regs, Decr_P, Add_regs, Shift_regs;
// Miscellaneous combinational logic
assign Product = {A, Q};
wire Zero = (P == 0); // counter is zero // Zero = ~|P; // alternative
wire Ready = (state == S_idle); // controller status
// control unit
always @ (posedge clock, negedge reset_b)
if (~reset_b) state <= S_idle; else state <= next_state;

always @ (state, Start, Q[0], Zero) begin
next_state = S_idle;
Load_regs = 0;
Decr_P = 0;
Add_regs = 0;
Shift_regs = 0;
case (state)
  S_idle: begin if (Start) next_state = S_add;
           Load_regs = 1; end
  S_add: begin next_state = S_shift; Decr_P = 1;
           if (Q[0]) Add_regs = 1; end
  S_shift: begin Shift_regs = 1;
            if (Zero) next_state = S_idle;
            else next_state = S_add; end
  default : next_state = S_idle;
endcase
end

// datapath unit
always @ (posedge clock) begin
if (Load_regs) begin
  P <= dp_width;
  A <= 0;
  C <= 0;
  B <= Multiplicand;
  Q <= Multiplier;
end
if (Add_regs) {C, A} <= A + B;
if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
if (Decr_P) P <= P -1;
end
endmodule