### Computing components from ALU1

**What is the worst case propagation delay from B to out?** (consider both arithmetic and logic operations)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 120ps</td>
<td>B: 150ps</td>
<td>C: 160ps</td>
</tr>
</tbody>
</table>

**XOR gate**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,B</td>
<td>out</td>
<td>30ps</td>
</tr>
</tbody>
</table>

**Full Adder**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,B</td>
<td>Sum</td>
<td>60ps</td>
</tr>
<tr>
<td>Cin</td>
<td>Sum</td>
<td>30ps</td>
</tr>
<tr>
<td>A,B</td>
<td>Cout</td>
<td>90ps</td>
</tr>
<tr>
<td>Cin</td>
<td>Cout</td>
<td>60ps</td>
</tr>
</tbody>
</table>

**Logic Unit**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,B</td>
<td>out</td>
<td>110ps</td>
</tr>
<tr>
<td>R</td>
<td>out</td>
<td>10ps</td>
</tr>
</tbody>
</table>

**2-to-1 Multiplexer**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>out</td>
<td>60ps</td>
</tr>
<tr>
<td>S</td>
<td>out</td>
<td>80ps</td>
</tr>
</tbody>
</table>
More Finite State Machines: Multiplication as an Example

Lab 4

Exam 1
Learning Objectives

- Design a Finite State Machine that controls a datapath
- Use Register Transfer Language to describe how state is modified each clock cycle
Step 1: Understand your problem
We can perform binary multiplication using the same algorithm as decimal multiplication.

\[
\begin{array}{c}
1101 \ (13) \\
\times \ 1010 \ (10)
\end{array}
\]

\[
\begin{array}{cccccc}
& & & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & & \\
\downarrow & & & & & \\
0 & 0 & 0 & & & \\
\downarrow & & & & & \\
1 & 1 & 0 & 1 & & \\
\downarrow & & & & & \\
& & & & & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 \ (130)
\end{array}
\]

- a) Add 0000
- b) Add 1101
- c) Add 1010
Multiplying two unsigned N-bit integers produces at most a 2N-bit integer

\[
\begin{array}{c}
  \text{1111 (15)} \\
  \times \text{1111 (15)} \\
  \hline
  \text{1111} \\
  \text{1111} \\
  \text{1111} \\
  \text{+1111} \\
  \hline
  \text{11100001 (225)}
\end{array}
\]
Multiplication can be described as a series of shift and add operations:

\[
\begin{array}{c}
00001101 \quad (13) \\
\times \quad 1010 \quad (10) \\
0000 \quad \Rightarrow 0 \\
00011010 \\
0000 \\
+01101000 \\
10000010 \quad (130)
\end{array}
\]

\[P = (A \ll 1) + (A \ll 3)\]
We determine whether to add each iteration by one-bit from the multiplicand

\[ 1101 \ (13) \]
\[ \times \underline{1010} \ (10) \]
\[ \underline{0000} \]
\[ 1101 \]
\[ \underline{0000} \]
\[ +1101 \]
\[ 10000010 \ (130) \]
We can use bitshifting and bitmasks to extract one bit from the multiplicand for control.

\[
\begin{array}{c}
1101 \ (13) \\
\times \ 1010 \ (10)
\end{array}
\]

\[
\begin{array}{c}
\phantom{\times} \\
0000 \\
1101 \\
0000 \\
1010 \\
\end{array}
\]

\[
\begin{array}{c}
B \\
B \& 1 \\
B \gg 1 \\
B \& 1 \\
B \gg 1
\end{array}
\]

\[
\begin{array}{c}
0000 \\
0000 \\
0101 \\
0001 \\
0010
\end{array}
\]
You try another example

\[ 1001 \times 0011 \]
You try another example

\[
\begin{array}{c}
1001 \ (9) \\
\times \ 0011 \ (3)
\end{array}
\]

\[
\begin{array}{c}
1001 \\
1001 \\
1001 \\
0000 \\
0000 \\
0000 \\
0000
\end{array}
\]

\[
00011011 \ (27)
\]
Write some C code that can perform the shift-add multiply algorithm

\[
\begin{array}{c}
1101 \ (13) \\
\times \ 1010 \ (10) \\
\hline
0000 \\
1101 \\
0000 \\
+1101 \\
\hline
10000010 \ (130)
\end{array}
\]

```c
int shift_add_multiply(unsigned short X,
                       unsigned short Y) {
    unsigned int A = (unsigned int) X;
    unsigned short B = Y;
    unsigned int P = 0;
    return P;
}
```
int shift_add_multiply(unsigned short X, unsigned short Y) {
    unsigned int A = (unsigned int) X;
    unsigned short B = Y;
    unsigned int P = 0;

    while (B != 0) {
        if ((B & 1) == 1) {
            P = P + A;
        }
        B = B >> 1;
        A = A << 1;
    }

    return P;
}
Step 2: Understand the datapath
Identify needed state components from variables (what state is stored?)

```c
int shift_add_multiply(...) {
    unsigned int A = (unsigned int) X;
    unsigned short B = Y;
    unsigned int P = 0;
    return P;
}
```
Identify how we modify that state

while (B != 0) {
    if ((B & 1) == 1) {
        P = P + A;
    }
    B = B >> 1;
    A = A << 1;
}
Define specification for control FSM

while (B != 0) {
    if ((B & 1) == 1) {
        P = P + A;
    }
    B = B >> 1;
    A = A << 1;
}
Control FSM has 3 inputs and 7 outputs

- B is zero (Z)
- B_Q[0] is 1 (B[0])
- START

- Load A (LA)
- Shift A (SA)
- Load B (LB)
- Shift B (SB)
- Add when 1/Zero when 0 (OP)
- Load P (LP)
- DONE
Control and Datapath

Control FSM

X Y
ALU
OP

AB
A (2N-bit reg) EN

B (N-bit reg) EN

P (2N-bit reg) EN

START

DONE

<<1

D

Q

>>1

D

Q

=0?

B_Q[0]

A

B

ALU

OP

D

Q

D

Q
Step 3: Design your FSM (don’t forget timing!)
int shift_add_multiply(...) {
    unsigned int A = (unsigned int) X;
    unsigned short B = Y;
    unsigned int P = 0;

    while (B != 0) {
        if ((B & 1) == 1) {
            P = P + A;
        }
        B = B >> 1;
        A = A << 1;
    }
    return P;
}
Register transfers show source state elements, operations, and destination state elements

A = X
B = Y
P = 0
A = A << 1
B = B >> 1
P = P + A
Wait
A = X, B = Y, P = 0
Do
Nothing
Shift
A = A << 1, B = B >> 1
Shift-Add
A = A << 1, B = B >> 1, P = P + A
Done
DONE = 1

A

B

C

D

E
Translate register transfers into control signals

A (2N-bit reg) EN

B (N-bit reg) EN

ALU

Control FSM

Wait
A = X, B = Y, P = 0

Wait
sa' _LA, sa' _LB, op', lp'

Done

Shift
A = A << 1, B = B >> 1

Shift
sa' _LA, sa' _LB, op', lp'

Done

Do Nothing

Do Nothing

P (2N-bit reg) EN

D Q

Wait

0 1

SA = 1

LA = 1

Q

D Q

LP = 0

DONE = 0

START

Y

SB = 1

B_Q[0]

<<1

=0?

>>1

0 1

SA = 1

LA = 1

Q

D Q

LP = 0

DONE = 0

A = X, B = Y, P = 0

sa' _LA, sa' _LB, op', lp'

Done

Shift
A = A << 1, B = B >> 1

Shift
sa' _LA, sa' _LB, op', lp'

Done

Do Nothing

Do Nothing

Wait
Wait
A = X, B = Y, P = 0

Do
Nothing

Shift
A = A << 1, B = B >> 1

Shift-Add
A = A << 1, B = B >> 1, P = P + A

Done
DONE = 1