Chapter 4
Digital Circuits

Signals: On or off for 1 or 0.

Digital devices compute output signals from input signals.

Build larger boxes from smaller ones.
1. **AND gate** \((c = a \cdot b)\)

\[
\begin{array}{ccc}
  a & b & c \\
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
\end{array}
\]

2. **OR gate** \((c = a + b)\)

\[
\begin{array}{ccc}
  a & b & c \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
\end{array}
\]

3. **Inverter** \((c = \overline{a})\)

\[
\begin{array}{c}
  a \quad c = \overline{a} \\
  0 \quad 1 \\
  1 \quad 0 \\
\end{array}
\]

4. **Multiplexor**
   (if \(d = 0\), \(c = a\);
   else \(c = b\))

\[
\begin{array}{ccc}
  a & b & c \\
  0 & 0 & a \\
  1 & 0 & b \\
\end{array}
\]
More Gates

\[ A \lor B = (A + B) \ \overline{AB} \]

\[ A \land B = \overline{(A + B)} \]

\[ A \oplus B = (A + B) \ \overline{AB} \]
Combine Gates for Larger Computations

\[
A \quad \quad B \quad \quad C \quad \quad D
\]

\[
\overline{AB} (\overline{C + D})
\]
Constructing the Multiplexor
Goal: An ALU

- ALU operation
- a → ALU → Zero → Result → Overflow → CarryOut
- b → ALU
One-Bit, Two Function ALU

Operation

Result

a

b

0

1
One-Bit Adder

```
  +
 /|
a b
```

- **CarryIn**
- **Sum**
- **CarryOut**
One-Bit Adder in Context

The one-bit adder does the work done for one column of a binary addition.

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
\hline
+ & 0 & 0 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
\hline
0 & 1 & 1 & 1 \\
\hline
0 & 0 & 0 & 1 \\
\end{array}
\]

Cin  
Sum  
Cout
Building the One-Bit Adder

\[ \text{Sum} = a \oplus b \oplus \text{CarryIn} \]
\[ \text{CarryOut} = (a \land b) \lor (\text{CarryIn} \land (a \oplus b)) \]
Building a Larger Adder

```
a0  a1  a2  a3
  |    |    |
  v    v    v
   a   b   a   b
   |   |   |   |   |
   v   v   v   v
  Sum Sum Sum Sum
  b   a

0

s0  s1  s2  s3
```

Oflow

CinCout

ab

Sum

CinCout

ab

Sum

CinCout

ab

Sum

CinCout

ab

Sum

CinCout

ab

Sum

CinCout

ab

Sum
One-Bit, Three-Function ALU

Operation

CarryIn

Result

CarryOut

a

b
Building a Larger ALU
Getting the Adder to Subtract

Invert b, set first carryin to 1

Diagram:

- **a**
- **b**
- **Binvert**
- **Operation**
- **CarryIn**
- **Result**
- **CarryOut**
ALU Code 3

The ALU should compute the function for set on less than.

- On subtract, the high bit means $a < b$.

- The multiplexor is extended to copy a third input Less on code three.
Final ALU
ALU Function Code

The Bnegate symbol can be considered part of the ALU control signal.

0 00 and
0 01 or
0 10 add
1 10 subtract
1 11 less than
Gate Delays

The time to compute a circuit is estimated by the number of gates on the longest path.

Three
Ripple-Carry Addition

Longest path goes through every bit.
Carries

For the $i$-th bit, denote:

$a_i, b_i$: Data inputs
$c_i$: Carry input

Carries are computed:

$c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i)$

Each must wait for the previous one.
Computing Carry-Ins From the Input Bits

\[ c_1 = b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0 \]

\[ c_2 = b_1 \cdot c_1 + a_1 \cdot c_1 + a_1 \cdot b_1 \]

\[ = b_1 \cdot (b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0) + a_1 \cdot (b_0 \cdot c_0 + a_0 \cdot c_0 + a_0 \cdot b_0) + a_1 \cdot b_1 \]

\[ = (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) + (a_1 \cdot b_0 \cdot c_0) + (b_1 \cdot a_0 \cdot b_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot b_0 \cdot c_0) + (a_1 \cdot b_1) \]

\[ c_3 = b_2 \cdot c_2 + a_2 \cdot c_2 + a_2 \cdot b_2 \]

Substitute \( c_2 \) and you’re on your way!
This finds $c_2$ with two gate delays rather than four.

In principle, we can find $c_{32}$ in two gate delays rather than 64.

Would you like to find the expression for $c_3$? Would you like to build the circuit? How about $c_{32}$?
Re-Write the Problem

Define:

\[ g_i = a_i \cdot b_i \]

*Bit* \( i \) *generates a carry when both input bits are one.*

\[ p_i = a_i + b_i \]

*Bit* \( i \) *propagates a carry when either input bit is one.*

Then:

\[ c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) = \]
\[ (b_i + a_i) \cdot c_i + (a_i \cdot b_i) = \]
\[ g_i + (p_i \cdot c_i) \]
Compute the Carries Again

\[ c_1 = g_0 + (p_0 \cdot c_0) \]

\[ c_2 = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0) \]

\[ c_3 = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0) \]

\[ c_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0) \]

*Delay of three instead of two, but much more manageable.*
Carry-Lookahead Adder
Combine Using Ripple-Carry
Combine Four Fours

We can compute super generate and propagate for each group of four:

\[ P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0 \quad \ldots \quad P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12} \]

\[ G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \]

\[ \ldots \]

\[ G_3 = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}) \]

Then

\[ C_1 = G_0 + (P_0 \cdot c_0) \]

\[ C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0) \]

\[ C_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot c_0) \]

\[ C_4 = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0) \]
Carry-Lookahead Adder

CarryIn

a0  a1  a2  a3  a4  a5  a6  a7  b0  b1  b2  b3  b4  b5  b6  b7

ALU0 P0 G0

C1  \(c_i + 1\)

ALU1 P1 G1

a8  a9  a10  a11  b8  b9  b10  b11

ALU2 P2 G2

C2  \(c_i + 2\)

ALU3 P3 G3

a12  a13  a14  a15  b12  b13  b14  b15

C3  \(c_i + 3\)

C4  \(c_i + 4\)

Result0–3

Result4–7

Result8–11

Result12–15

Carry-out
How to Multiply in Decimal

\[
\begin{array}{c}
3142 \\
\times 1032 \\
\hline
6284 \\
9426 \\
+ 3142 \\
\hline
3242544
\end{array}
\]
How to Multiply in Binary

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 0 \\
\times & & & & & & & \\
1 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
+ & 0 & 1 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]
Re-Organized Binary Multiply

\[ 010110 \times 110101 \]

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]
## Running Total

\[010110 \times 110101 = 10010001110\]

<table>
<thead>
<tr>
<th>terms</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>010110</td>
<td>000000000101101101</td>
</tr>
<tr>
<td>0000000</td>
<td>000000000101101100</td>
</tr>
<tr>
<td>010110</td>
<td>000001101110111011</td>
</tr>
<tr>
<td>0000000</td>
<td>000001101110111000</td>
</tr>
<tr>
<td>010110</td>
<td>000111001110111011</td>
</tr>
<tr>
<td>010110</td>
<td>010010001110111011</td>
</tr>
</tbody>
</table>
Motion is Relative

*Shift the sum left instead of the terms right.*

\[010110 \times 110101 = 10010001110\]

<table>
<thead>
<tr>
<th>terms</th>
<th>sum (shift each step)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 0</td>
<td>0 1 0 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>0 1 1 0 1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>0 1 1 1 0 0 1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>1 0 0 1 0 0 0 1 1 1 0</td>
</tr>
<tr>
<td>0 1 0 1 1 0</td>
<td>1 0 0 1 0 0 0 1 1 1 0</td>
</tr>
</tbody>
</table>

CSc 314  •  T W Bennet  •  Mississippi College
Conditional Addition

*Perform addition when the control bit is 1.*

010110 \times 110101 = 10010001110

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

010110 \times 110101 = 10010001110
**Shift Together**

\[ 010110 \times 110101 = 10010001110 \]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
\end{array}
\]
Multiplication Hardware

32-bit ALU

Multiplicand

32 bits

Product

Shift right

Write

64 bits

Control test
Multiplication Algorithm

1. Test Product0
   - Product0 = 1
   - Product0 = 0

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

2. Shift the Product register right 1 bit

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

Done
Multiplication, Combined Register

$010110 \times 110101 = 010010001110$

0 1 0 1 1 0  

First Arg

↓ Test Bit

0 0 0 0 0 0 | 1 1 0 1 0 1 0 1  

Second Arg

0 1 0 1 1 0 | 1 1 0 1 0 1 0 1  

Test 1, Add

0 0 1 0 1 1 0 | 1 1 0 1 0 1 0  

Shift Right

0 0 1 0 1 1 0 | 1 1 0 1 0 1 0  

Test 0, No Op

0 0 0 1 0 1 1 0 | 1 1 0 1  

Shift Right

0 1 1 0 1 1 1 0 | 1 1 0 1  

Test 1, Add
Continued, Repeating first line

0 1 0 1 1 0  First Arg

↓ Test Bit

0 1 1 0 1 1 1 0|1 1 0 1  Test 1, Add
0 0 1 1 0 1 1 1 0|1 1 0  Shift Right
0 0 1 1 0 1 1 1 0|1 1 0  Test 0, No Op
0 0 0 1 1 0 1 1 1 0|1 1  Shift Right
0 1 1 1 0 0 1 1 1 0|1 1  Test 1, Add
0 0 1 1 1 0 0 1 1 1 0|1 1  Shift Right
1 0 0 1 0 0 0 1 1 1 0|1  Test 1, Add
0 1 0 0 1 0 0 0 1 1 1 0  Shift Right
And Carry, Too

The shift must bring in the carryout from the addition.

\[ 101100 \times 001011 = 000111100100 \]
Final Multiplication Hardware

- Multiplicand
- 32-bit ALU
- Product
- Control test
- Shift right
- Write
- Shift In
- 32 bits
- 64 bits
- Carry Out
- Test
Decimal Division

\[
\begin{array}{c}
42 \overline{5638} \\
42 \\
143 \\
126 \\
178 \\
168 \\
10
\end{array}
\]
Binary Division

\[
\begin{array}{c|c}
1 & 1 0 0 0 1 \\
\hline
0 & 1 0 1 1 1 \\
0 & 1 0 1 1 1 \\
0 & 0 1 1 0 1 1 \\
0 & 1 0 1 1 1 \\
0 & 0 0 1 0 0 0 \\
0 & 0 0 0 0 0 \\
0 & 0 1 0 0 0 1 \\
0 & 0 0 0 0 0 \\
0 & 1 0 0 0 1 0 \\
0 & 1 0 1 1 1 \\
0 & 0 1 0 1 1 
\end{array}
\]
Hardware Division

Shift the dividend left, accumulate the quotient in the bits deserted by the dividend.

\[
\begin{array}{c}
\frac{586}{23} = 25, \text{ Remainder 11}
\end{array}
\]

0000010111 \quad \text{Divisor}

000000000001001010 \quad \text{Init}

0000000001001010100 \quad \text{Shift}

\ldots

0000010010010101000000 \quad \text{Shift}

0000100100101010000000 \quad \text{Shift}

00000011011010000001 \quad \text{Sub/Set}
Hardware Division Continued

\[
\frac{586}{23} = 25, \text{ Remainder } 11
\]

```
0 0 0 0 0 1 0 1 1 1

Divisor

...```

```
0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1
  Sub/Set

0 0 0 0 0 1 1 0 1 1 0 1 0 0 0 0 0 1 0
  Shift

0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 1
  Sub/Set

0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0
  Shift

0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0
  Shift

0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0
  Shift

0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 1
  Sub/Set

Remainder  Quotient
```
Division Algorithm

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

Start

Remainder = 0

Test Remainder

Remainder < 0

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done. Shift left half of Remainder right 1 bit
Floating Point Addition Algorithm

1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.

2. Add the significands.

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.

4. Round the significand to the appropriate number of bits.

Overflow or underflow?

Yes

No

Still normalized?

Yes

Done

No

Exception
Floating Point Addition Hardware
Floating Point Multiplication Algorithm

1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent

2. Multiply the significands

3. Normalize the product if necessary, shifting it right and incrementing the exponent

4. Round the significand to the appropriate number of bits

5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative

Done