Arithmetic Circuits & Multipliers

- Addition, subtraction
- Performance issues
  - ripple carry
  - carry bypass
  - carry skip
  - carry lookahead
- Combinational multiplier
- Two’s complement multiplier
- Smaller multipliers, faster multipliers

Signed integers: 2's complement

- \(-2^{N-1}\) to \(2^{N-1} - 1\)

8-bit 2's complement example:
11010110 = \(-2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42\)

If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

By moving the implicit location of "decimal" point, we can represent fractions too:
1101.0110 = \(-2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625\)

Sign extension

Consider the 8-bit 2's complement representation of:
42 = 00101010
-5 = \(-00000101 + 1\)
  = 11111010 + 1
  = 11111011

What is their 16-bit 2's complement representation?
42 = 0000000000101010
-5 = 1111111111111011

Extend the MSB (aka the "sign bit") into the higher-order bit positions

Using Signed Arithmetic in Verilog

```
wire signed [7:0] total;
wires [3:0] counter; // max value 15, counting widgets off the mfg line
wire signed [5:0] available;

assign total = available + counter; // does this give the correct answer?
NO! counter = 4'b1111 is treated as -1. Need to "append" a leading zero
assign total = available + {1'b0, counter}; // or use $unsigned()
assign total = available + $unsigned(counter);
```
**Adder: a circuit that does addition**

Here’s an example of binary addition as one might do it by “hand”:

\[
\begin{array}{c}
1101 \\
+ 0101 \\
10010
\end{array}
\]

Carries from previous column

Adding two N-bit numbers produces an (N+1)-bit result

If we build a circuit that implements one column:

we can quickly build a circuit to add two 4-bit numbers...

**“Full Adder” building block**

- S = \( A \oplus B \oplus C \)
- \( CO = \overline{A}BC + \overline{A}BC + AB\overline{C} + ABC \)
- \( = (\overline{A} + A)BC + (\overline{B} + B)AC + AB\overline{(C + C)} \)
- \( = BC + AC + AB \)

**Subtraction: A - B = A + (-B)**

Using 2’s complement representation: \(-B = \overline{B} + 1\)

So let’s build an arithmetic unit that does both addition and subtraction. Operation selected by control input:

**Condition Codes**

Besides the sum, one often wants four other bits of information from an arithmetic unit:

- \( Z \) (zero): result is = 0
- \( N \) (negative): result is < 0
- \( C \) (carry): indicates an add in the most significant position produced a carry, e.g., 1111 + 0001 from last FA
- \( V \) (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., 0111 + 0111

To compare A and B, perform A-B and use condition codes:

**Signed comparison:**

- LT \( \overline{N} \oplus V \)
- LE \( Z + (\overline{N} \oplus V) \)
- EQ Z
- NE \( \sim Z \)
- GE \( (\overline{N} \oplus V) \)
- GT \( (Z + (\overline{N} \oplus V)) \)

**Unsigned comparison:**

- LTU C
- LEU \( C + Z \)
- GEU \( \sim C \)
- GTU \( \sim (C + Z) \)

**To compare A and B:**

- \( \overline{A}N_{\text{c1}} \oplus B_{\text{c1}} + S_{\text{c1}}N_{\text{c1}} - 1 \)
- \( \overline{A}N_{\text{c2}} \oplus B_{\text{c2}} + S_{\text{c2}}N_{\text{c2}} - 1 \)
- \( \overline{A}N_{\text{c3}} \oplus B_{\text{c3}} + S_{\text{c3}}N_{\text{c3}} - 1 \)
- \( \overline{A}N_{\text{c4}} \oplus B_{\text{c4}} + S_{\text{c4}}N_{\text{c4}} - 1 \)
Condition Codes in Verilog

- **Z** (zero): result is = 0
- **N** (negative): result is < 0

**C** (carry): indicates an add in the most significant position produced a carry, e.g., 1111 + 0001

- **V** (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., 0111 + 0111

```verilog
wire signed [31:0] a, b, s;
wire z, n, v, c;
assign {c, s} = a + b;
assign z = ~s;
assign n = s[31];
assign v = a[31] & b[31] & s[31] & c;
```

Might be better to use sum-of-products formula for V from previous slide if using LUT implementation (only 3 variables instead of 4).

Modular Arithmetic

The Verilog arithmetic operators (+,-, *) all produce full-precision results, e.g., adding two 8-bit numbers produces a 9-bit result.

In many designs one chooses a “word size” (many computers use 32 or 64 bits) and all arithmetic results are truncated to that number of bits, i.e., arithmetic is performed modulo 2^word size.

Using a fixed word size can lead to overflow, e.g., when the operation produces a result that’s too large to fit in the word size. One can:

- Avoid overflow: choose a sufficiently large word size
- Detect overflow: have the hardware remember if an operation produced an overflow - trap or check status at end
- Embrace overflow: sometimes this is exactly what you want, e.g., when doing index arithmetic for circular buffers of size 2^N.

- “Correct” overflow: replace result with most positive or most negative number as appropriate, aka saturating arithmetic. Good for digital signal processing.

Speed: \( t_{PD} \) of Ripple-carry Adder

- \( C_0 = AB + AC_I + BC_I \)

Worst-case path: carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001.

\[
t_{PD} = (N-1) \times (t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)
\]

\( t_{adder} = (N-1)t_{carry} + t_{sum} \)

How about the \( t_{PD} \) of this circuit?

- \( t_{PD} \) of this circuit is not \( 2 \times t_{PD,N-BIT RIPPLE} \)

\( t_{PD} \) of this circuit is: \( t_{PD,N-BIT RIPPLE} + t_{PD,FA} \)

\( \Theta(N) \) is read "order N": means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

Timing analysis is tricky!
Alternate Adder Logic Formulation

How to Speed up the Critical (Carry) Path? (How to Build a Fast Adder?)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Ci</th>
<th>S</th>
<th>Co</th>
<th>Carry status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>generate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>generate</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Full Adder} & \quad C_{in} \rightarrow C_{o} \\
\text{Generate (G)} & = AB \\
\text{Propagate (P)} & = A \oplus B \\
C_{o}(G, P) & = G + P C_i \\
S(G, P) & = P \oplus C_i
\end{align*}
\]

Note: can also use \( P = A + B \) for \( C_o \)

Faster carry logic

Let’s see if we can improve the speed by rewriting the equations for \( C_{OUT} \):

\[
C_{OUT} = AB + AC_{IN} + BC_{IN} = AB + (A + B)C_{IN}
\]

\[
C_{OUT} = G + P C_{IN}
\]

generate  propagate

Actually, \( P \) is usually defined as \( P = A \land B \) which won’t change \( C_{OUT} \) but will allow us to express \( S \) as a simple function:

\[
S = P \lor C_{IN}
\]

Virtex II Adder Implementation

1 half-Slice = 1-bit adder

1 CLB = 4 Slices = 2, 4-bit adders

64-bit Adder: 16 CLBs

Virtex II Carry Chain

CLBs must be in same column
**Carry Bypass Adder**

Can compute $P$, $G$ in parallel for all bits

**Key Idea:** if $(P_0 \ P_1 \ P_2 \ P_3)$ then $C_{0,3} = C_{i,0}$

**Critical Path Analysis**

For the second stage, is the critical path:

$BP_2 = 0$ or $BP_2 = 1$?

Message: Timing analysis is very tricky - Must carefully consider data dependencies for false paths

**16-bit Carry Bypass Adder**

What is the worst case propagation delay for the 16-bit adder?

Assume the following for delay each gate:

- $P$, $G$ from $A$, $B$: 1 delay unit
- $P$, $G$, $C_i$ to $C_o$ or Sum for a $C/S$: 1 delay unit
- 2:1 mux delay: 1 delay unit

**Carry Bypass vs Ripple Carry**

Ripple Carry: $t_{adder} = (N-1) \ t_{carry} + t_{sum}$

Carry Bypass: $t_{adder} = 2(M-1) \ t_{carry} + t_{sum} + (N/M-1) \ t_{bypass}$

$t_{adder}$

ripple adder

bypass adder

$M = \text{bypass word size}$

$N = \text{number of bits being added}$
Carry Lookahead Adder (CLA)

• Recall that \( C_{OUT} = G \cdot P \cdot C_{IN} \) where \( G = A \& B \) and \( P = A \oplus B \)

• For adding two \( N \)-bit numbers:

\[
C_N = G_{N-1} + P_{N-1}C_{N-1} \\
= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-2} \\
= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-3} + \ldots + P_{N-1} \ldots P_0C_{IN}
\]

\( C_N \) in only 3 gate delays*: 
1 for \( P/G \) generation, 1 for ANDs, 1 for final OR

*assuming gates with \( N \) inputs

• Idea: pre-compute all carry bits as \( f(Gs, Ps, C_{IN}) \)

### The 74182 Carry Lookahead Unit

- high speed carry lookahead generator
- used with 74181 to extend carry lookahead beyond 4 bits
- correctly handles the carry polarity of the 181

\[ C_{OUT} = \sum \left( G \cdot P \right) \cdot C_{IN} \]

G and \( P \) can be computed for groups of bits (instead of just for individual bits). This allows us to choose the maximum fan-in we want for our logic gates and then build a hierarchical carry chain using these equations:

\[
C_{J+1} = G_{J+1,K} + P_{J+1,K}C_J \\
G_{IK} = G_{J+1,K} + P_{J+1,K} G_{J} \\
P_{IK} = P_{J+1,K} P_{J} \\
\]

“generate a carry from bits \( I \) thru \( K \) if it is generated in the high-order \((J+1,K)\) part of the block or if it is generated in the low-order \((I,J)\) part of the block and then propagated thru the high part”

Block Generate and Propagate

where \( I < J \) and \( J+1 < K \)

Hierarchical building block

P/G generation

1st level of lookahead
8-bit CLA (P/G generation)

Log₂(N)

8-bit CLA (carry generation)

Log₂(N)

8-bit CLA (complete)

tₚD = Θ(log(N))

Unsigned Multiplication

AB, called a "partial product"  →  A₃B₀ A₂B₀ A₁B₀ A₀B₀
A₃B₁ A₂B₁ A₁B₁ A₀B₁
A₃B₂ A₂B₂ A₁B₂ A₀B₂
+ A₃B₃ A₂B₃ A₁B₃ A₀B₃

Multiplying N-bit number by M-bit number gives (N+M)-bit result

Easy part: forming partial products
(just an AND gate since Bᵢ is either 0 or 1)
Hard part: adding M N-bit partial products
2's Complement Multiplication
(Baugh-Wooley)

Step 1: two's complement operands so high
order bit is \(-2^{n-1}\). Must sign extend partial
products and subtract the last one

\[ X = -2^x x_3 + \sum x_i 2^i \]
\[ Y = -2^y y_3 + \sum y_i 2^i \]

The product of \( X \) and \( Y \) is:

\[ XY = x_3 y_3 2^6 + \sum x_i y_3 2^{i+3} + \sum x_3 y_j 2^{j+3} + \sum \sum x_i y_j 2^{i+j} \]

For two's complement, the following is true:

\[ -\Sigma x_i 2^i = -2^x + \Sigma x_i 2^i + 1 \]

The product then becomes:

\[ XY = x_3 y_3 2^6 + x_3 x_y 2^{i+3} + 2^3 - 2^6 + \Sigma x_3 y_j 2^{j+3} + 2^3 - 2^6 + \Sigma x_3 y_j 2^{j+3} + \Sigma x_3 y_j 2^{j+3} + \Sigma \Sigma x_i y_j 2^{i+j} + 2^4 - 2^7 \]

The product then becomes:

\[ XY = x_3 y_3 2^6 + x_3 x_y 2^{i+3} + 2^3 - 2^6 + \Sigma x_3 y_j 2^{j+3} + 2^3 - 2^6 + \Sigma x_3 y_j 2^{j+3} + \Sigma x_3 y_j 2^{j+3} + \Sigma \Sigma x_i y_j 2^{i+j} + 2^4 - 2^7 \]

Result: multiplying 2's complement operands
takes just about same amount of hardware as
multiplying unsigned operands!
## 2’s Complement Multiplication

### Multiplication in Verilog

You can use the "*" operator to multiply two numbers:

```verilog
wire [9:0] a,b;
wire [19:0] result = a*b; // unsigned multiplication!
```

If you want Verilog to treat your operands as signed two’s complement numbers, add the keyword `signed` to your `wire` or `reg` declaration:

```verilog
wire signed [9:0] a,b;
wire signed [19:0] result = a*b; // signed multiplication!
```

Remember: unlike addition and subtraction, you need different circuitry if your multiplication operands are signed vs. unsigned. The same is true of the `>>>` (arithmetic right shift) operator. To get signed operations all operands must be signed.

To make a signed constant: `10'sh37C`

## Multiplication on the FPGA

In the XC2V6000: 6 columns of multipliers, 24 in each column = 144 multipliers

Hardware multiplier block: two 18-bit two’s complement (signed) operands

Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit M times:

### Sequential Multiplier

Init: $P \leftarrow 0$, load A and B

Repeat M times:

- $P \leftarrow P + (B_{LSB} == 1 \ ? \ A : 0)$
- shift P/B right one bit

Done: (N+M)-bit result in P/B
Bit-Serial Multiplication

Init: \( P = 0 \); Load \( A, B \)

Repeat \( M \) times {
  Repeat \( N \) times {
    shift \( A, P \):
    \( \text{Amsb} = \text{Alsb} \)
    \( \text{Pmsb} = \text{Plsb} + \text{Alsb} \times \text{Blsb} + C/0 \)
  }
  shift \( P, B \): \( \text{Pmsb} = C \), \( \text{Bmsb} = \text{Plsb} \)
}

\((N+M)\)-bit result in \( P/B \)

Combinational Multiplier (unsigned)

\[
\begin{array}{cccccc}
X_3 & X_2 & X_1 & X_0 & \times & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
+ & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\
+ & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\
+ & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
\]

Useful building block: Carry-Save Adder

Good for pipelining: delay through each partial product (except the last) is just \( T_{PD,\text{AND}} + T_{PD,\text{FA}} \).

No carry propagation time!

Wallace Tree Multiplier

This is called a 3:2 counter by multiplier hackers: counts number of 1’s on the 3 inputs, outputs 2-bit result.

Wallace Tree:
Combine groups of three bits at a time

Higher fan-in adders can be used to further reduce delays for large \( M \).

4:2 compressors and 5:3 counters are popular building blocks.
**Multiplication by a constant**

- If one of the operands is a constant, make it the multiplier (B in the earlier examples). For each “1” bit in the constant we get a partial product (PP) – may be noticeably fewer PPs than in the general case.
  - For example, in general multiplying two 4-bit operands generates four PPs (3 rows of full adders). If the multiplier is say, 12 (4'b1100), then there are only two PPs: 8*A+4*A (only 1 row of full adders).
  - But lots of “1”s means lots of PPs... can we improve on this?
- If we allow ourselves to subtract PPs as well as adding them (the hardware cost is virtually the same), we can re-encode arbitrarily long contiguous runs of “1” bits in the multiplier to produce just two PPs.

\[ \cdots011110\cdots = \cdots100000\cdots - \cdots000010\cdots = \cdots0100010\cdots \]

where T indicates subtracting a PP instead of adding it. Thus we’ve re-encoded the multiplier using 1,0,-1 digits – aka *canonical signed digit* – greatly reducing the number of additions required.

---

**Booth Recoding: Higher-radix mult.**

Idea: If we could use, say, 2 bits of the multiplier in generating each partial product we would halve the number of columns and halve the latency of the multiplier!

\[
\begin{array}{cccccccc}
A_{N-1} & A_{N-2} & \ldots & A_4 & A_3 & A_2 & A_1 & A_0 \\
B_{M-1} & B_{M-2} & \ldots & B_3 & B_2 & B_1 & B_0 \\
\end{array}
\]

Booth’s insight: rewrite 2*A and 3*A cases, leave 4*A for next partial product to do!

---

**Booth recoding**

On-the-fly canonical signed digit encoding!

<table>
<thead>
<tr>
<th>(B_{K+1})</th>
<th>(B_K)</th>
<th>(B_{K-1})</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>add 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>add A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>add 2*A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sub 2*A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>sub A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sub A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>add 0</td>
</tr>
</tbody>
</table>

A “1” in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!

---

**Summary**

- Performance of arithmetic blocks dictate the performance of a digital system
- Architectural and logic transformations can enable significant speed up (e.g., adder delay from \(O(N)\) to \(O(\log_2(N))\))
- Similar concepts and formulation can be applied at the system level
- Timing analysis is tricky: watch out for false paths!
- Area-Delay trade-offs (serial vs. parallel implementations)