

Arithmetic Logic Units

EE685, Fall 2024

Hank Dietz

<http://aggregate.org/hankd/>

Integer Representations

- **Unsigned binary**; bit k has value 2^k
- Signed binary; MSB is 1 if negative:
 - **2's complement**: $-x$ is $(\sim x)+1$
 - Same add/sub circuit as unsigned
 - One more negative value than positive
 - **1's complement**: $-x$ is $\sim x$
 - Both $+0$ and -0 , or use -0 as **NaN**
 - **Sign + magnitude**: $-x$ is $x^{\wedge}MSB$
 - Like 1's comp.; used for float mantissa

Negate Operations in Verilog

```
// 2's complement
module twosneg(d, s);
parameter BITS=8; output [BITS-1:0] d; input [BITS-1:0] s;
assign d = (~s) + 1;
Endmodule
```

```
// 1's complement
module onesneg(d, s);
parameter BITS=8; output [BITS-1:0] d; input [BITS-1:0] s;
assign d = ~s;
Endmodule
```

```
// sign + magnitude
module smneg(d, s);
parameter BITS=8; output [BITS-1:0] d; input [BITS-1:0] s;
assign d = { ~s[BITS-1], s[BITS-2:0] };
endmodule
```

Negate with NaN in Verilog

```
// NaN-preserving 1's complement
module onesneg(d, s);
parameter BITS=8; output [BITS-1:0] d; input [BITS-1:0] s;
assign d = (((&s) || (!s)) ? s : ~s);
endmodule
```

```
// NaN-preserving sign + magnitude
module smneg(d, s);
parameter BITS=8; output [BITS-1:0] d; input [BITS-1:0] s;
assign d = ((!s[BITS-2:0]) ? s :
            {~s[BITS-1], s[BITS-2:0]});
endmodule
```

More Integer Representations

- Various other notations (less common):
 - **Negabinary**; no sign, bit k has value -2^k
- **BCD (binary coded decimal)**
 - Each decimal digit is 4 bits, carry is adjusted
 - **ASCII** is BCD with top 4 bits 0011 (i.e., +48)
- **Gray code**; add/sub 1 changes only 1 bit
 - Race free, great for encoders
 - Awkward for math
- **Saturation** as opposed to **modular** arithmetic
 - v becomes $\min(\text{maxvalue}, \max(v, \text{minvalue}))$

Gray Code Conversions

```
// Binary to Gray
module bin2gray(gray, bin);
parameter BITS=8;
output [BITS-1:0] gray; input [BITS-1:0] bin;
assign gray = (bin ^ (bin >> 1));
endmodule
```

```
// Gray to Binary
module gray2bin(bin, gray);
parameter BITS=8;
output [BITS-1:0] bin; input [BITS-1:0] gray;
wire [BITS-1:0] t[BITS:0]; // temporaries, not all used
genvar i;
generate for (i=BITS>>1; i>=1; i=i>>1) begin
    assign t[i>>1] = (t[i] ^ (t[i] >> i));
end endgenerate
assign t[BITS>>1] = gray; // initial bit pattern
assign bin = t[0]; // final bit pattern
endmodule
```

Running The Verilog Code

- Uses the standard Gray coding
 - Binary to Gray is just $(\text{Binary} \wedge (\text{Binary} \gg 1))$
 - Gray to Binary takes $\log_2(\text{BITS})$ gates using algorithm from Magic Algorithms page
- Can run it here:
<http://aggregate.org/EE380/alugray.html>
- Notice that this is an **exhaustive test**

Recursive Gray Conversion

```
// Gray to Binary module gray2bin(bin, gray);
parameter BITS=8;
parameter R=BITS/2;
output [BITS-1:0] bin;
input [BITS-1:0] gray;

genvar i;
generate
  if (R < 2) begin
    assign bin = (gray ^ (gray >> 1));
  end else begin
    gray2bin #(BITS,R>>1) recur(bin, (gray ^ (gray >> R)));
  end
endgenerate
endmodule
```


Running Recursive Version

- Uses the standard Gray coding
 - Binary to Gray is just $(\text{Binary} \wedge (\text{Binary} \gg 1))$
 - Gray to Binary takes $\log_2(\text{BITS})$ gates and is implemented recursively!
- Can run it here:
<http://aggregate.org/EE380/alugrayrecur.html>
- Notice that this is an **exhaustive test**

2's Comp. Add / Subtract

- There are many approaches... we'll discuss:
 - **Ripple carry** (remember this from CPE282?)
 - **Carry lookahead** (also from CPE282)
 - **Carry select**
 - **Speculative carry**
- Pick the *most suitable* implementation
- No matter what implementation, **$a - b$** is always **$a + (\sim b) + 1$** , with a circuit using **$a_k + (b_k \wedge \text{sub})$** and an initial carry in of *sub*

A 1-Bit Half Adder (HA)

```
// Half Adder
module ha(sum, cout, a, b);
output sum, cout;
input a, b;
assign sum = a ^ b;      // sum
assign cout = a & b;    // cout
endmodule
```

A 1-Bit Half Adder (HA)

```
// Half Adder
module ha(sum, cout, a, b);
output sum, cout;
input a, b;
xor(sum, a, b);           // sum
and(cout, a, b);         // cout
endmodule
```

A 1-Bit Full Adder (FA)

```
// Full Adder
module fa(sum, cout, a, b, cin);
output sum, cout;
input a, b, cin;
wire aorb, gen, prop;
xor(sum, a, b, cin); // sum
and(gen, a, b); // generate
or(aorb, a, b); // propagate
and(prop, aorb, cin);
or(cout, gen, prop); // cout
endmodule
```

An 8-Bit Ripple Carry Adder

```
// Ripple carry addition, 8-bit
module add8(sum, cout, a, b, cin);
output [7:0] sum;
output cout;
input [7:0] a, b;
input cin;
wire [6:0] lcout;
fa fa0(sum[0], lcout[0], a[0], b[0], cin),
   fa1(sum[1], lcout[1], a[1], b[1], lcout[0]),
   fa2(sum[2], lcout[2], a[2], b[2], lcout[1]),
   fa3(sum[3], lcout[3], a[3], b[3], lcout[2]),
   fa4(sum[4], lcout[4], a[4], b[4], lcout[3]),
   fa5(sum[5], lcout[5], a[5], b[5], lcout[4]),
   fa6(sum[6], lcout[6], a[6], b[6], lcout[5]),
   fa7(sum[7], cout, a[7], b[7], lcout[6]);
endmodule
```

Let's Test It: testbench

```
module testbench;
reg [7:0] a, b, refsum; wire [7:0] sum;
reg cin, refcout; wire cout; integer tested=0, wrong=0;

add8 uut(sum, cout, a, b, cin);          // unit under test

initial begin a=0; repeat (256) begin b=0;
  repeat (256) begin cin=0; repeat (2) #1 begin
    {refcout, refsum} = a + b + cin;    // oracle
    tested=tested+1;
    if ((refcout != cout) || (refsum != sum)) begin
$display("Wrong: %d+%d+%d is {%d,%d}, but got {%d,%d}",
  a, b, cin, refcout, refsum, cout, sum);
    wrong=wrong+1;
  end cin=1; end b = b + 1; end a = a + 1; end
$display("%d cases tested, %d wrong", tested, wrong);
end endmodule
```

Let's Really Test It...

- Can run it here:
<http://aggregate.org/EE380/aluripple8.html>
- Notice that this is an **exhaustive test**
 - All 131,072 cases are tried ($2^8 \times 2^8 \times 2$)
 - Exhaustive testing quickly becomes less feasible as number of input bits grows

A Parametric Ripple Adder

```
// Ripple carry addition, BITS-bit
module add(sum, cout, a, b, cin);
parameter BITS=8;
output [BITS-1:0] sum;
output cout;
input [BITS-1:0] a, b;
input cin;
wire [BITS:0] c; // temporary (local) wires

genvar i;
generate for (i=0; i<BITS; i=i+1) begin:fas
    // full adders named fas[i].myfa
    fa myfa(sum[i], c[i+1], a[i], b[i], c[i]);
end endgenerate

assign c[0] = cin; // first carry in
assign cout = c[BITS]; // last carry out
endmodule
```

Let's Test That One Too...

- Can run it here:
<http://aggregate.org/EE380/alurippleBITS.html>
- Notice that the entire Verilog program is **just one line longer** than the 8-bit-only version
- This version essentially **generates the exact same gates**; there's no hardware cost to being parametric

A Parametric Ripple Subtract

```
// Ripple carry subtraction, BITS-bit
module sub(result, borrow, a, b);
parameter BITS=8;
output [BITS-1:0] result;
output borrow;
input [BITS-1:0] a, b;
wire [BITS-1:0] not b;

// 2's complement  $a - b$  is  $a + (-b)$  is  $a + ((\sim b) + 1)$ 
add (#BITS) adder(result, borrow, a, notb, 1'b1);

assign notb = ~b;

endmodule
```

Combined Add/Subtract

```
// Ripple carry addition, BITS-bit
module addsub(result, carry, a, b, dosub);
parameter BITS=8;
output [BITS-1:0] result;
output carry;
input [BITS-1:0] a, b;
input dosub;
wire [BITS:0] c; // temporary (local) wires

genvar i;
generate for (i=0; i<BITS; i=i+1) begin:fas
    // full adders named fas[i].myfa
    fa myfa(result[i], c[i+1], a[i], b[i]^dosub, c[i]);
end endgenerate

assign c[0] = dosub; // first carry in
assign carry = c[BITS]; // last carry out
endmodule
```

A Carry Lookahead Adder

- Carry lookahead parallelizes carry computations

a	b	cin	cout	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	propagate cin
1	0	0	0	
1	0	1	1	propagate cin
1	1	0	1	generate carry
1	1	1	1	both generate and propagate

$$\text{cout}_k = g_{k-1} \mid (g_{k-2} \ \& \ p_{k-1}) \mid (g_{k-3} \ \& \ p_{k-2} \ \& \ p_{k-1}) \mid \dots$$

$$\text{cout}_k = \text{OR}(g_{k-1} , \text{AND}_{k-2..k-1}, \text{AND}_{k-3..k-1}, \dots)$$

Make The Wide AND/OR

```
module look(cout, prop, gen);
parameter BITS=8;
output cout; input [BITS-1:0] prop; input [BITS:0] gen;
wire [BITS:0] ands;

genvar i;
generate
    // ands[i] = gen[i] & prop[i] & prop[i+1] & ... &
prop[BITS-1]
    assign ands[BITS] = gen[BITS];
    for (i=BITS-1; i>=0; i=i-1) begin
        assign ands[i] = gen[i] & &prop[BITS-1:i]; // wide AND
    end
endgenerate

// cout = OR( ands[] )
assign cout = |ands[BITS:0]; // wide OR
endmodule
```

A Carry Lookahead Adder

```
wire [BITS:0] gen, lcin;
wire [BITS-1:0] prop;

genvar i, j;
generate
    assign prop = a | b;
    assign gen = { (a & b), cin };

    // make p and g
    assign lcin[0] = cin;
    for (i=1; i<BITS+1; i=i+1) begin:looks
        look #(i) mylook(lcin[i], prop[i-1:0], gen[i:0]);
    end
endgenerate

assign sum = a ^ b ^ lcin;
assign cout = lcin[BITS];
```

And Let's Test It!

- Can run it here:
<http://aggregate.org/EE380/aluclookBITS.html>
- This isn't quite as fast as it looks because most logic can't have fast k-input AND and OR gates
 - e.g., $\text{AND}(a, b, c, d)$ becomes $\text{AND}(\text{AND}(a, b), \text{AND}(c, d))$
 - Thus, time is actually $O(\log_i(\text{BITS}))$ where i is the maximum number of inputs per gate

A Carry Select Adder

- Carry select simply cracks the problem in half
 - The low bits [BITS/2:0] are computed directly
 - Compute hi bits [BITS-1:BITS/2] **twice**:
assuming **carry in of 0** and **carry in of 1**
 - Select the correct high bits

```
// Carry select
module cselect(sum, cout, a, b, cin);
parameter BITS=8;
output [BITS-1:0] sum;
output cout;
input [BITS-1:0] a, b;
input cin;
```

A Carry Select Adder

```
generate
  if (BITS < 2) begin
    fa onebit(sum, cout, a, b, cin);
  end else begin
    wire [BITS-(BITS/2)-1:0] sumhi0, sumhi1;
    wire coutlo, couthi0, couthi1;
    cselect #(BITS/2) recurlo(sum[(BITS/2)-1:0], coutlo,
      a[(BITS/2)-1:0], b[(BITS/2)-1:0], cin);
    cselect #(BITS-(BITS/2)) recurhi0(sumhi0, couthi0,
      a[BITS-1:BITS/2], b[BITS-1:BITS/2], 0);
    cselect #(BITS-(BITS/2)) recurhi1(sumhi1, couthi1,
      a[BITS-1:BITS/2], b[BITS-1:BITS/2], 1);
    assign sum[BITS-1:BITS/2] = (coutlo ? sumhi1 : sumhi0);
    assign cout = (coutlo ? couthi1 : couthi0);
  end
endgenerate
endmodule
```

And Let's Test It!

- Can run it here:
<http://aggregate.org/EE380/alucselectBITS.html>
- Pretty obviously $O(\log_2(\text{BITS}))$
- Notice that this is generating a recursive decomposition that ends in a full adder; alternatively, could stop at any level and use a ripple-carry adder

A Speculative Carry Adder

- Carry select computes the hi bits both ways... but **they're not equiprobable!**

- In practice, **carry in of 0 is more likely**

a	b	carryout	
0	0	0	< for small positive ints, most common
0	1	carryin	< "" carryin is usually 0
1	0	carryin	< "" carryin is usually 0
1	1	1	

- Good guess: **a & b**

a	b	carryout	guess	
0	0	0	0	< correct 100% of the time
0	1	carryin	0	< correct at least 50%
1	0	carryin	0	< correct at least 50%
1	1	1	1	< correct 100% of the time

A Speculative Carry Adder

```
// Speculative carry
module specc(sum, cout, ok, a, b, cin, guess);
parameter BITS=8;
output [BITS-1:0] sum;
output cout, ok;
input [BITS-1:0] a, b;
input cin, guess;
wire coutlo;

// don't really care how we do the halves....
assign { coutlo, sum[(BITS/2)-1:0] } = a[(BITS/2)-1:0] +
    b[(BITS/2)-1:0] + cin;
assign { cout, sum[BITS-1:BITS/2] } = a[BITS-1:BITS/2] +
    b[BITS-1:BITS/2] + guess;
assign ok = (coutlo == guess);
endmodule
```

Test This State Machine

- Can run it here:
<http://aggregate.org/EE380/aluspeccBITS.html>
- Notice that the combinatorial module is NOT always going to generate a correct result...
 - 1st clock cycle, try `guess`; if right, we're done
 - 2nd clock cycle, `~guess` must be right
- With better guess, originally used in Pentium 4 to save power with double-speed adder; fewer gates & power usually off for 2nd cycle

Adder Choices

- Not only can you use any of those or others, but **you can make hybrids**, e.g.:
 - 32-bit speculative on 16-bit chunks
 - 16-bit carry select on 8-bit chunks
 - 8-bit carry select on 4-bit chunks
 - 4-bit ripple carry
- There's no such thing as a "best" adder... it's all about **picking the engineering tradeoffs to optimize for the current situation**
- **Verilog builds whatever is convenient for +**

Multiply (**fast, big**...**slow, small**)

- **Random logic**: build the truth table and find the minimum circuit to implement it
- **Lookup table**: table[{a, b}] holds $a*b$
- **Repetitive addition**: $a*b$ is sum a copies of b
- **Shift & add loop**: like in gradeschool, but we don't need to do it in base 10

Shift & Add Multiply

- Like in gradeschool, but “digits” are base 2... it’s an AND gate: $0*0=0$; $0*1=0$; $1*0=0$; $1*1=1$;
- At each step, shift and add to sum:

```

          0 0 1 1      3
        * 1 1 0 1    *13
          0 0 1 1      9
          0 0 0 0  _   3_
            0 0 1 1  _
              0 0 1 1  _
                0 0 1 1 1

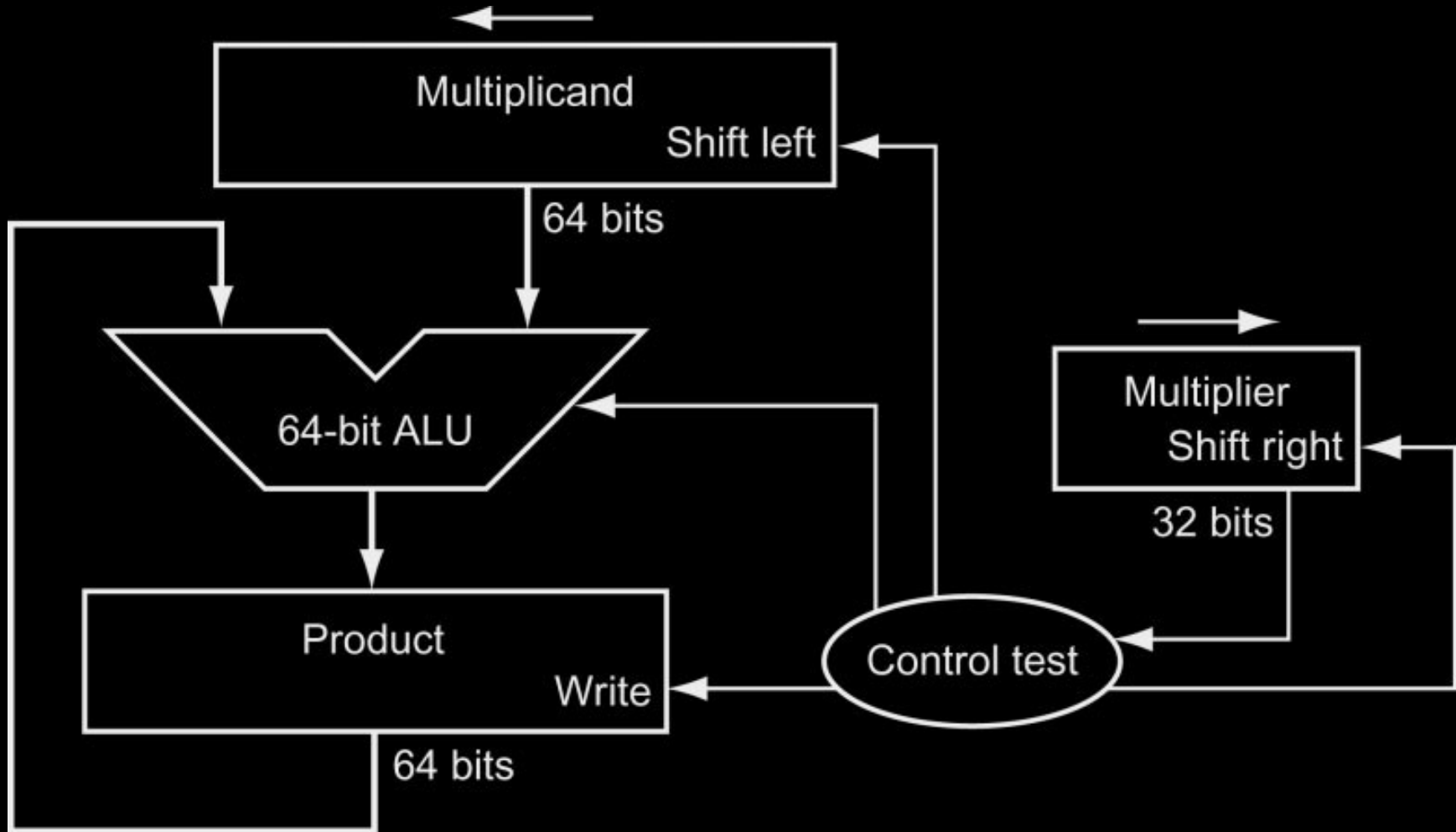
```

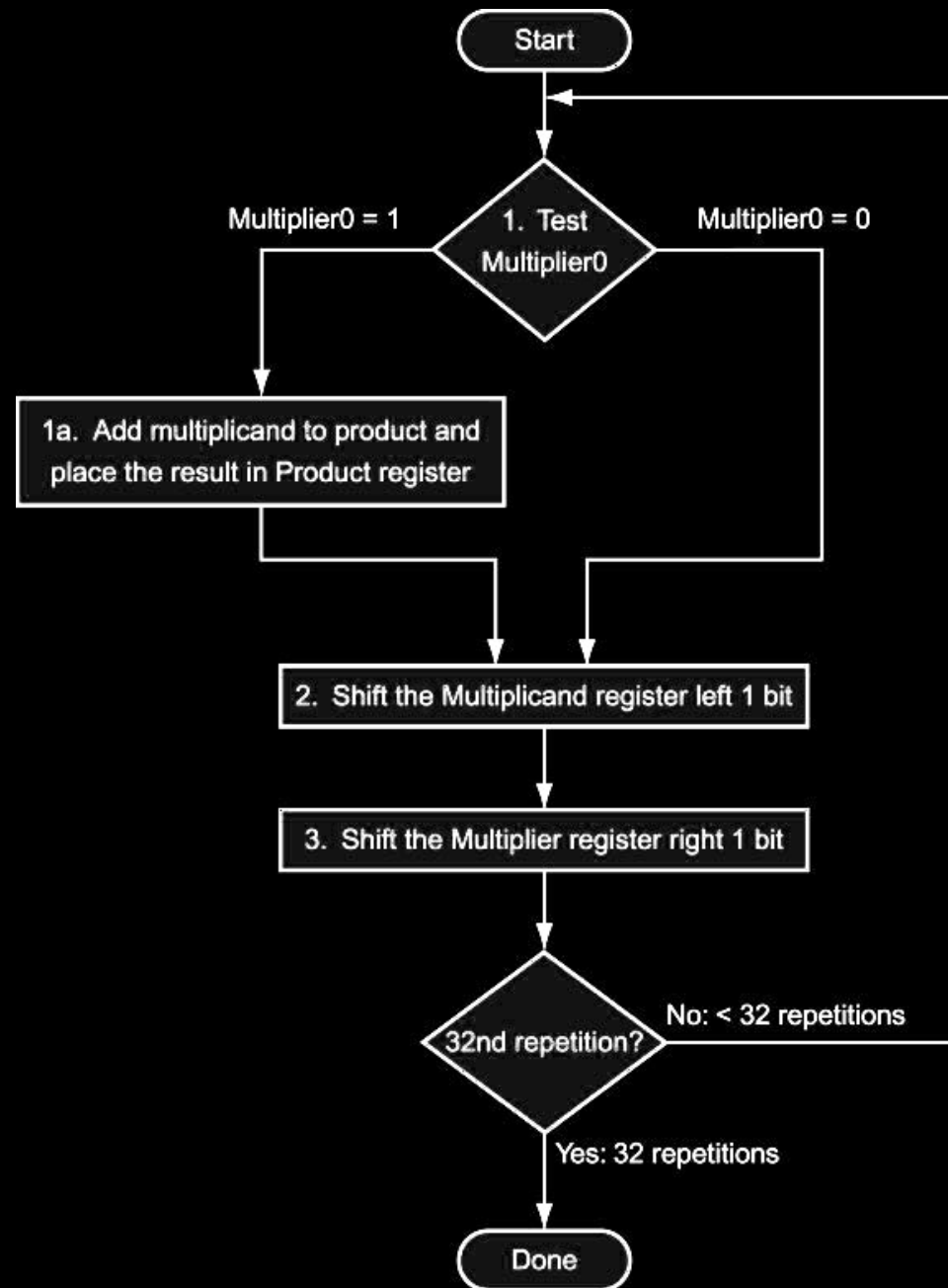
Shift & Add Multiply

- At each step, shift and add to sum:

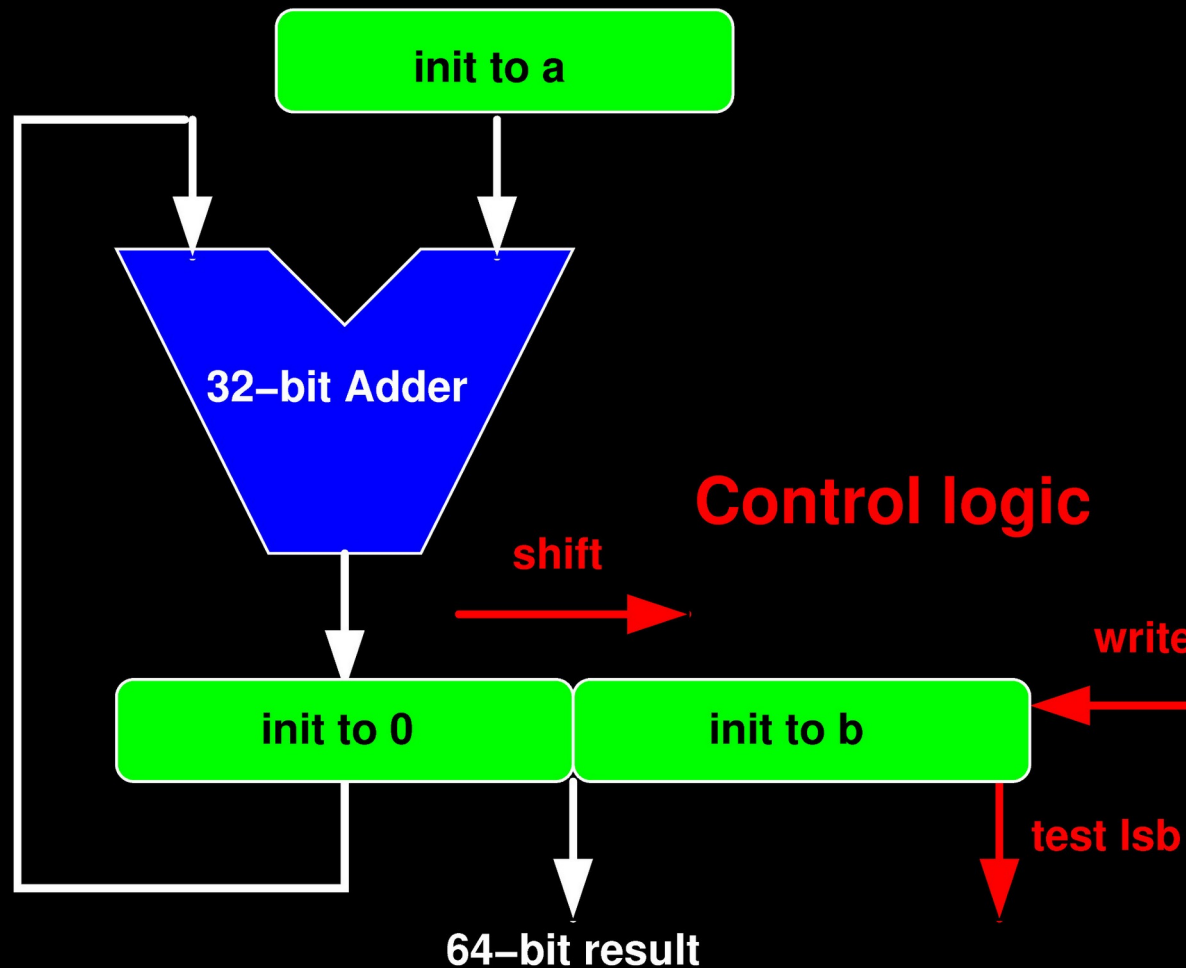
					0	0	1	1	
				*	1	1	0	1	
=	0	0	0	0	0	0	0	0	these bits not yet active
+					0	0	1	1	
=	<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>	0	0	1	1	
+				0	0	0	0		this didn't change anything
=	<u>0</u>	<u>0</u>	<u>0</u>	0	0	0	1	<u>1</u>	
+			0	0	1	1			
=	<u>0</u>	<u>0</u>	0	0	1	1	<u>1</u>	<u>1</u>	
+		0	0	1	1				
=	0	0	1	0	0	1	1	1	these bits no longer active

An Obvious Multiply Design





A Smarter Multiply Design

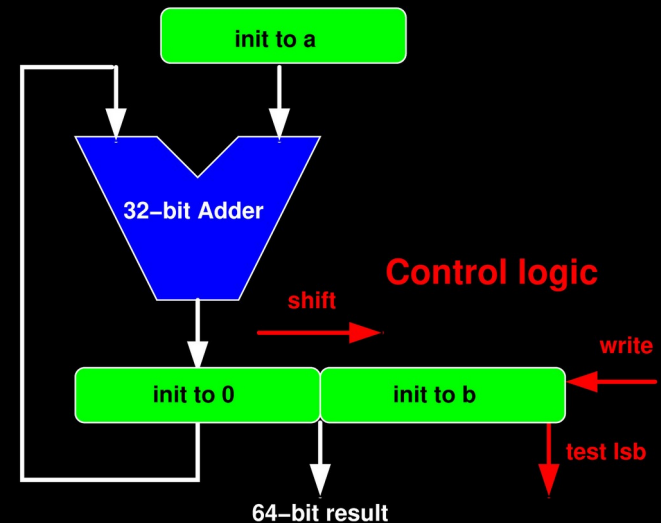


Multiplier in Verilog

```
module mul(ready, prod, a, b, reset, clk);
parameter BITS = 32;
input [BITS-1:0] a, b;
input reset, clk;
output reg [BITS*2-1:0] prod;
output reg ready;
reg [BITS-1:0] d;
reg [BITS-1:0] state;
reg [BITS:0] sum;
```

Multiplier in Verilog

```
always @(posedge clk or posedge reset) begin
  if (reset) begin
    ready <= 0;
    state <= 1;
    d <= a;
    prod <= {{BITS{1'b0}}, b};
  end else begin
    if (state) begin
      sum = prod[BITS*2-1:BITS] + d;
      prod <= (prod[0] ?
        {sum, prod[BITS-1:1]} :
        (prod >> 1));
      state <= {state[BITS-2:0], 1'b0};
    end else begin
      ready <= 1;
    end
  end
end
end
endmodule
```



Let's Test It!

- Can run it here:
<http://aggregate.org/EE380/alumulBITS.html>
- This is a state machine, so there's a reset signal and a clock input
- Exhaustive testing is still feasible...
for modest numbers of bits
 - 8x8 multiply has 65536 test cases
 - 16x16 multiply has 4294967296 test cases
 - 32x32 multiply has 18446744073709551616

Better Shift & Add Multiply

- Can **skip 0 bits in b**
- **Swap a, b if a has more 0s than b**
- **Booth's Algorithm:**
 - Convert a run of 1s into just two 1s by using add and subtract: $a*7 = a+a*2+a*4 = (-a)+a*8$
 - A run of 1s becomes subtract, skip the run of 1s, and then add what was the next 0

Counting Trailing Zeros

- A binary search for the least significant 1 bit

```
// 32-bit trailing 0 counter
module trail0(d, s);
output [5:0] d; input [31:0] s;
wire [4:0] r; wire [15:0] s16; wire [7:0] s8; wire [3:0] s4;

assign {r[4],s16} = ((|s[15:0]) ? {1'b0,s[15:0]} :
                    {1'b1,s[31:16]});
assign {r[3],s8} = ((|s16[7:0]) ? {1'b0,s16[7:0]} :
                    {1'b1,s16[15:8]});
assign {r[2],s4} = ((|s8[3:0]) ? {1'b0,s8[3:0]} :
                    {1'b1,s8[7:4]});
assign {r[1],r[0]} = ((|s4[1:0]) ? {1'b0,!s4[0]} :
                      {1'b1,!s4[2]});

// force 32 if s is 0
assign d = ((|s) ? {1'b0,r} : 32);
endmodule
```


Best Multiply?

- As for add, there is no “best”...
- Many processors do the following:
 - Have an **8x8 multiply lookup table**
 - Use the standard shift & add sequence, but compute using **8-bit base 256 “digits”**
 - E.g., many AMD processors did 4-cycle 32x32 integer multiplies this way

Divide?

- A lot like multiply... but most methods produce both divide result and remainder (modulus)
- Shift & subtract algorithms often used
 - More complex because subtract should only happen when value \geq divisor
 - Tricks for skipping steps in multiply don't directly apply here; often 32 cycles for 32/32

Divide in Verilog

- Remainder and quotient are paired in a single register during the shift & subtract sequence

```
module div(ready, q, r, n, d, reset, clk);
parameter BITS=8;
input [BITS-1:0] n, d;
input reset, clk;
output wire [BITS-1:0] q, r;
output reg ready;
reg [2*BITS-1:0] rq;
reg [BITS-1:0] state, s;
reg lsb;
```

Divide in Verilog

```
assign q = rq[BITS-1:0];
assign r = rq[2*BITS-1:BITS];
always @(posedge clk or posedge reset) begin
    if (reset) begin
        ready <= 0; state <= 1; rq <= { {BITS{1'b0}}, n };
    end else begin
        if (state) begin if (d == 0) begin
            rq <= 0; read <= 1; // really divide by 0 error
        end else begin
            if (rq[2*BITS-2:BITS-1] >= d) begin
                s = rq[2*BITS-2:BITS-1] - d; lsb = 1; end
            else begin s = rq[2*BITS-2:BITS-1]; lsb = 0; end
            rq <= { s, rq[BITS-2:0], lsb };
            state <= {state[BITS-2:0], 1'b0};
        end end else begin ready <= 1;
    end end end endmodule
```

Let's Test It!

- Can run it here:
<http://aggregate.org/EE380/aludivBITS.html>
- This is a state machine, so there's a reset signal and a clock input
- Exhaustive testing like multiply...

Floating Point

- Trades precision & accuracy for dynamic range
- Not the only way to do this...
 - **LNS**: Log Number Systems
 - **Unums**: universal numbers, **Posits**
- Different machines used to have incompatible floating-point formats, different accuracy, etc. ...
but then came the **IEEE 754 standard**

<https://ieeexplore-ieee-org.ezproxy.uky.edu/document/8766229>

IEEE 754 Basics

- Each float value is a pair of signed integers:
 - Exponent is 2's comp., but adds a bias
 - Mantissa (fraction) is sign + magnitude

$$(-1)^{\text{sign}} * 2^{\text{exponent-bias}} * (1 + 2^{1-\text{precision}} * \text{fraction})$$

- **Sign** is the MSB, 1 for negative values
- **Exponent** gets bias = maximum exponent (that makes minimum look like all 0s)
- **Precision** is one greater than field width (a leading 1 is implied for normal floats)

Print it out, read it in?

- Is there a problem with decimal fractions?

```
volatile float a, b, asum, bsum;
main()
{
    a = 0.1; b = 1.0; asum = 0; bsum = 0;
    for (int i=0; i<1000; ++i) { asum += a; bsum += b; }

    printf("%f, %f\n", a, asum);
    printf("%f, %f\n", b, bsum);
}
```

```
0.100000, 99.999046
1.000000, 1000.000000
```

IEEE 754 Details

- Predictive infinities and NaNs
 - Gracefully overflow with $+/- \infty$
 - **NaNs** really about *when to handle errors*
- **Denormalized** numbers
 - **Values near 0 don't get normalized**
 - Often simplified to just 0 as a special case
- **Rounding modes**
 - Can specify rounding up, down, toward 0...
 - Extra “**guard**” bits used to preserve accuracy

Common Float Formats

Precision	Word	Sign	Exp	Fraction
Half	16	1	5	10
bfloat16	16	1	8	7
Single	32	1	8	23
Double	64	1	11	52
Extended	80	1	15	64
Quad	128	1	15	113

bfloat16 is really 16 MSBs of Single
Extended used inside Intel 8087

Let's Keep It Simple Here...

- We'll just detail how **bfloat16** works
 - No ∞ nor NaN, and 0 is the only subnormal
 - No rounding, guard bits (**lousy accuracy**)

<code>float16[15]</code>	sign bit, 1 means negative
<code>float16[14:7]</code>	exponent, +bias
<code>float16[6:0]</code>	normalized mantissa

<http://super.ece.engr.uky.edu:8088/cgi-bin/float16.cgi>

<http://aggregate.org/EE480/floaty.html>

Can Use 32-bit float To Check

- Just implements the **top 16 bits of 32-bit float** so just look at those bits...

```
typedef unsigned short float16;  
float16 f16; unsigned int i;
```

```
i = (f16 << 16);  
... *((float *) &i) ...
```

```
f16 = (i >> 16);
```

```
Sign = ((i & 0x8000) ? 1 : 0);
```

```
Exp = ((i >> 7) & 0xff);
```

```
Frac = ((i & 0x7f) + (f16 ? 0x80 : 0));
```

Addition Algorithm: $r = a + b$

- **Denormalize** so that $a \cdot 2^{\text{EXP}} == b \cdot 2^{\text{EXP}}$
(make smaller equal the larger of the two)
- Add/subtract fractions, depending on signs
- Set sign of result
- **Normalize**

Note: **can lose accuracy!**

- Denormalize loses $a \cdot 2^{\text{EXP}} - b \cdot 2^{\text{EXP}}$ bits
- **Catastrophic cancellation** if $a \approx -b$

Associativity?

- Is floating-point add associative:

`float a,b,c; IS (a+(b+c)) == ((a+b)+c) ?`

```
volatile float a, b, c;
main() {
    a = 1000000000.0; b = -a; c = 1.0;
    printf("(a+(b+c)) = %f\n", a, b, c, (a+(b+c)));
    printf("((a+b)+c) = %f\n", a, b, c, ((a+b)+c));
}
```

```
(1000000000.000000+(-1000000000.000000+1.000000)) = 0.000000
((1000000000.000000+-1000000000.000000)+1.000000) = 1.000000
```

Loop Index Not Incrementing?

```
volatile float a, b, bold; volatile int ia, ib;
main() {
    ia = 1000000000; a = ia; bold = -1;

    for (ib=0; ib<ia; ++ib) ;
    printf("ib = %d\n", ib);
    for (b=0; b<a; ++b) {
        if (b == bold) {
            printf("stuck at b = %f\n", b); exit(1);
        }
        bold = b;
    }
    printf("b = %f\n", b);
}
```

```
ib = 1000000000
stuck at b = 16777216.000000
```

Normalization Issues

- Normalization requires shifting until 1 in MSB
 - Need to count leading zeros
 - Barrel shift to the left (multiply by 2^k)
- For addition, denormalization requires shifting
 - Pick smaller exponent, compute difference
 - Barrel shift to the right (divide by 2^k)
- You want to do this combinatorially...

Barrel Shifter

- Simple trick: \log_2 decomposition

```
// 32-bit barrel shift right logical (0 fill)
module srl(d, s, sh);
output [31:0] d; input [31:0] s, sh;
wire [31:0] by1, by2, by4, by8, by16;
assign by1 = (sh[0] ? {1'b0, s[31:1]} :s);
assign by2 = (sh[1] ? {2'b0, by1[31:2]} :by1);
assign by4 = (sh[2] ? {4'b0, by2[31:4]} :by2);
assign by8 = (sh[3] ? {8'b0, by4[31:8]} :by4);
assign by16 = (sh[4] ? {16'b0, by8[31:16]} :by8);
assign d = ((|sh[31:5]) ? 0 : by16);
endmodule
```

Let's Test It!

- Can run it here:
<http://aggregate.org/EE380/alubarrel.html>
- Purely combinatorial, but this isn't an exhaustive test because there are too many cases
- Shifting the other direction is just as easy

Counting Leading Zeros

- A lot like counting trailing zeroes...

```
// 32-bit leading 0 counter
module lead0(d, s);
output [5:0] d; input [31:0] s;
wire [4:0] r; wire [15:0] s16; wire [7:0] s8; wire [3:0] s4;

assign {r[4],s16} = ((|s[31:16]) ? {1'b0,s[31:16]} :
                    {1'b1,s[15:0]});
assign {r[3],s8} = ((|s16[15:8]) ? {1'b0,s16[15:8]} :
                    {1'b1,s16[7:0]});
assign {r[2],s4} = ((|s8[7:4]) ? {1'b0,s8[7:4]} :
                    {1'b1,s8[3:0]});
assign {r[1],r[0]} = ((|s4[3:2]) ? {1'b0,!s4[3]} :
                      {1'b1,!s4[1]});

// force 32 if s is 0
assign d = ((|s) ? {1'b0,r} : 32);
endmodule
```

Multiplication Algorithm:

$$r = a * b$$

- Set sign of result
- Add exponents
- Multiply fractions (8 bit * 8 bit); keep high bits
- Normalize

Note: doesn't really lose accuracy...

Floating-Point Multiply

- Just like the algorithm, but **normalize** is easy

```
// Floating-point multiply, 16-bit r=a*b
module fmul(r, a, b);
output wire `FLOAT r;
input wire `FLOAT a, b;
wire [15:0] m; // double the bits in a fraction, we need high bits
wire [7:0] e;
wire s;
assign s = (a `FSIGN ^ b `FSIGN);
assign m = ({1'b1, (a `FFRAC)} * {1'b1, (b `FFRAC)});
assign e = (((a `FEXP) + (b `FEXP)) - 127 + m[15]);
assign r = (((a == 0) || (b == 0)) ? 0 :
            (m[15] ? {s, e, m[14:8]} : {s, e, m[13:7]}));
endmodule
```


Divide Algorithm: $r=a/b$

- Easier to compute reciprocal, $t=1/b$
- $r = a * (1/b)$

Reciprocal Algorithm: $r=1.0/x$

- Guess & iteratively refine guess
- Note that 2.0f in our format is 0x4080

```
typedef union { float f; int i; } fi_t;
float recip(float x)
{
    fi_t t;
    t.f = guess(x);
    t.f *= (2.0f - (t.f * x)); // 1st iter
    t.f *= (2.0f - (t.f * x)); // 2nd iter
    return(t.f);
}
```

Reciprocal Algorithm: $r=1.0/x$

- A really sneaky way to guess, using the fact that $1.0/2^n$ is 2^{-n} , which can be computed by int sub...

```
typedef union { float f; int i; } fi_t;
float recip(float x)
{
    fi_t t;
    t.f = x;
    t.i = magic - t.i; // guess
    t.f *= (2.0f - (t.f * x)); // 1st iter
    t.f *= (2.0f - (t.f * x)); // 2nd iter
    return(t.f);
}
```

Reciprocal Algorithm: $r=1.0/x$

- Try all; best magic is **0x7eea**
average **3.98 bits bad** without iterations!
- Min max error is 7 bits, using 0x7f00

```
typedef union { float f; int i; } fi_t;
float recip(float x)
{
    fi_t t;
    t.f = x;
    t.i = magic - t.i; // guess
    return(t.f);
}
```

A Better Reciprocal Guess

- Can actually do better quite easily using a **lookup table (ROM) to invert the mantissa**
 - Low 7 bits of mantissa replaced by lookup
 - Exponent is either:
 - 254 – *exp iff low mantissa bits were 0*
 - 253 – *exp otherwise*
- Note that subnormals are still special cases; **1/0 should produce NaN**, but we'll allow 0 here
- Iteratively improve if more mantissa bits needed

Reciprocal Lookup Table

- Here's the 7-bit mantissa reciprocal table:

00, 7e, 7c, 7a, 78, 76, 74, 72, 70, 6f, 6d, 6b, 6a, 68, 66, 65,
63, 61, 60, 5e, 5d, 5b, 5a, 59, 57, 56, 54, 53, 52, 50, 4f, 4e,
4c, 4b, 4a, 49, 47, 46, 45, 44, 43, 41, 40, 3f, 3e, 3d, 3c, 3b,
3a, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 2f, 2e, 2d, 2c, 2b,
2a, 29, 28, 28, 27, 26, 25, 24, 23, 23, 22, 21, 20, 1f, 1f, 1e,
1d, 1c, 1c, 1b, 1a, 19, 19, 18, 17, 17, 16, 15, 14, 14, 13, 12,
12, 11, 10, 10, 0f, 0f, 0e, 0d, 0d, 0c, 0c, 0b, 0a, 0a, 09, 09,
08, 07, 07, 06, 06, 05, 05, 04, 04, 03, 03, 02, 02, 01, 01, 00

Other Arithmetic

- Square root can be computed as $x * 1/\sqrt{x}$ using method similar to $1/x$ for $1/\sqrt{x}$
- Some things are easy:
 - Absolute value; $\text{abs}(x)$ is $(x \ \& \ \sim\text{sign})$
 - Shifts just add/subtract to/from exponent
- Transcendentals are hard because they can't be expressed as a finite polynomial

Transcendental Functions

- Sin, cos, tan, arctan, tanh, log, etc.
- Evaluate a truncated Taylor series
 - Evaluated using only +, -, *, and 1/x
 - Slow and error can accumulate
- CoRDiC: Coordinate Rotation Digital Computer
 - Essentially a binary search for value
 - Rotations by constants, scaled so multiply can be implemented by a shift
 - Get approx. 1 bit of result per step

float/int Conversions

- **An integer is a denormalized float...**
- 16-bit int to float:
 - Make int positive, set sign
 - Take most significant 1 + 7 more bits
 - Set exponent to normalize result
- float to 16-bit int:
 - Take (positive) 8-bit fraction part
 - Barrel shift integer appropriately
 - Negate if sign was set

SIMD Within A Register

- A lot of machines have SIMD/vector ALUs that employ multiple ALUs with the same control
- **SWAR** recognizes that it's even easier to carve existing registers/datapaths into fields, e.g.:

```
reg [31:0] a, b, c; ... a = b + c;
```

could be 4 8-bit adds by simply *breaking the carry chain* in the right spots:

```
a[31:24]=b[31:24]+c[31:24]; a[23:16]=b[23:16]+c[23:16];  
a[15: 8]=b[15: 8]+c[15: 8]; a[ 7: 0]=b[ 7: 0]+c[ 7: 0];
```

Summary

- There are lots of ways to do arithmetic
 - Different representations
 - Different algorithms
 - You've **seen Verilog code** for many things
 - Engineering tradeoffs decide best choice
- Computer arithmetic doesn't follow math rules:
 - `float a,b,c; is (a+(b+c)) == ((a+b)+c) ?`
 - `float a; for (a=0; a<1000000000.0; ++a); ?`
 - `0.1` is a repeating fraction in base 2 float