



Chapter 3

Arithmetic for Computers Implementation

Today

- Review representations (252/352 recap)
- **Floating point**
- Addition: Ripple carry adder, carry-look-ahead adder
- Barrel Shifter
- Multiplication: Simple and Radix-4
- Division: Simple and Newton-Raphson
- Floating point: Addition, multiplication

Unsigned integer representation

- With n bits, max value that can be represented: $2^n - 1$

Binary to Decimal conversion

1	1	0	1	0	0
2^5	2^4	2^3	2^2	2^1	2^0

32 + 16 + 4
=52

6 bits, so max number possible is $2^6-1 = 63$

Binary to Decimal conversion

0	0	1	1	0	1
2^5	2^4	2^3	2^2	2^1	2^0
<hr/>					
		8	+ 4		+ 1

=13

6 bits, so max number possible is $2^6 - 1 = 63$

Binary to Decimal conversion (8 bits)

1	1	0	0	1	1	0	1					
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0					
<hr/>												
128	+	64	+			8	+	4			+	1
=205												

8 bits, so max number possible is $2^8 - 1 = 255$

Binary to Decimal conversion (8 bits)

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1
= 255

8 bits, so max number possible is $2^8 - 1 = 255$

Decimal to Binary

1. Find number of bits required

$$\text{Floor}(\log_2 \text{number}) + 1$$

2. For each bit-position, starting from highest, Repeatedly check if number greater or equal to $2^{\text{bit-position}}$, and set bit to 0 or 1 accordingly

Number of bits required

- **52:** $\log_2(52) = 5.7$; $\text{floor}(5.7) = 5$; # bits = 6
 - Check 1: $2^6 - 1 \leq 52$; $63 < 52$ (YES)
 - Check 2: $2^5 - 1 > 52$; $32 > 52$ (YES)
- **102:** $\log_2(102) = 6.67$; $\text{floor}(6.67) = 6$; # bits = 7
 - Check 1: $2^7 - 1 \leq 102$; $127 < 107$ (YES)
 - Check 2: $2^6 - 1 > 102$; $63 > 107$ (YES)
- **276:** $\log_2(276) = 8.10$; $\text{floor}(8.10) = 8$; # bits = 9
 - Check 1: $2^9 - 1 \leq 276$; $511 < 276$ (YES)
 - Check 2: $2^8 - 1 > 276$; $255 > 276$ (YES)

Number of bits required

- **64:** $\log_2(64) = 6.0$; $\text{floor}(6.0) = 6$; # bits = 7
 - Check 1: $2^7 - 1 \leq 64$; $127 < 64$ (YES)
 - Check 2: $2^6 - 1 > 64$; $63 > 64$ (YES)

Decimal to Binary

52; # bits = 6

Bit positions start at 0, so 6 bits means 2^0 to 2^5

Bit position	Power of 2	Number	>=	Remainder	Bit value
5	32	52	Yes	20	1
4	16				
3	8				
2	4				
1	2				
0	1				

We can check

1	1	0	1	0	0
2^5	2^4	2^3	2^2	2^1	2^0

32 + 16 + 4
=52

Decimal to Binary

37; # bits = 6

Bit positions start at 0

Bit position	Power of 2	Number	>=	Remainder	Bit value
5	32	37	Yes	5	1
4	16				
3	8				
2	4				
1	2				
0	1				

We can check

1	0	0	1	0	1
2^5	2^4	2^3	2^2	2^1	2^0
<hr/>					
32			+ 4		+1
=37					

Decimal to Binary

37; # bits = 6

Bit positions start at 0

Bit position	Power of 2	Number	>=	Remainder	Bit value
5	32	37	Yes	5	1
4	16				
3	8				
2	4				
1	2				
0	1				

2's complement representation

- Allows representing negative numbers
- **Arithmetic operations can be done by operating on individual bits**
- 0 is always all bits 0
- Most negative number: -2^{n-1}
- Most positive number: $+2^{n-1} - 1$

2's complement range

1 bit	This is weird: -1 to 0		
2 bits	-2	To	+1
3 bits	-4	To	+3
4 bits	-8	To	7
5 bits	-16	To	15
6 bits	-32	To	31
7 bits	-64	To	63
8 bits	-128	To	127
9 bits	-256	To	255
10 bits	-512	To	511

Decimal to Binary (2's comp)

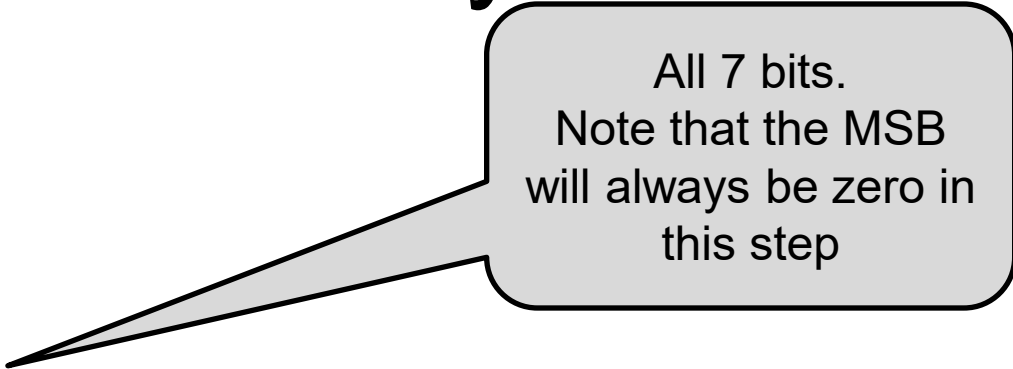
- First get number of bits
$$\text{Floor}(\log_2 \text{abs}(\text{number})) + 2$$
- If positive number, then use process we developed before and you are done
- If negative number,
 - First get representation of the absolute value
 - Then invert all bits
 - Then add +1 to the inverted bits

Decimal to Binary

- 52

1.# bits = 7

2.Positive number; 0110100



All 7 bits.
Note that the MSB
will always be zero in
this step

Decimal to Binary

- -52

1. # bits = 7

2. Negative number

a) Representation of +52 = 0110100

All 7 bits.
Note that the **MSB will always be zero in this intermediate step**

b) Invert all bits:

c) Add +1:

```
1001011
+0000001
=1001100
```

All 7 bits. Note that the MSB will always be **ONE for negative numbers** at the very end

Decimal to Binary

- -101

1. # bits = 8

2. Negative number

a) Representation of +101 = 01100101

All 7 bits.
Note that the MSB
will always be zero in
this step

b) Invert all bits:

c) Add +1:

```
10011010
+00000001
=10011011
```

All 7 bits. Note that the MSB will
always be **ONE for negative
numbers** at the very end

Decimal to Binary

- -64

1. # bits = 7

2. Negative number

a) Representation of +64 = 1000000

All 7 bits.
Note that the **MSB will always be zero in this intermediate step**

b) Invert all bits:

c) Add +1:

```
0111111
+0000001
=1000000
```

All 7 bits. Note that the MSB will always be **ONE for negative numbers** at the very end

2's complement binary to decimal

- If MSB is 0, same as unsigned
- If MSB is 1, reverse steps:
 - a) Invert all bits
 - b) Add +1
 - c) Now determine magnitude
Remember it is a negative number

2's complement Binary to decimal

- 1001100
- MSB is 1

a) Invert all bits: 0110011

b) Add +1: +0000001

0110100

$2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0$

$$= 32 + 16 + 4 = 52$$

c) -52

2's complement Binary to decimal

- 10011011
- MSB is 1

a) Invert all bits: 01100100

b) Add +1: +00000001

01100101

$2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0$

$$= 64 + 32 + 4 + 1 = 101$$

c) -101

2's complement arithmetic

It's bitwise addition!

- $52 + (-101) = -49$

$$\begin{array}{r} 00110100 \\ +10011011 \\ \hline 11001111 \end{array}$$

- Let's check what this value is

Check value

- 11001111
- MSB is 1

a) Invert all bits: 00110000

b) Add +1: +00000001


00110001

$2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0$

$$= 32 + 16 + 1 = 49$$

c) -49

2's complement extension

- -52 in 7 bits
 - -52 in 8 bits
 - -52 in 9 bits
 - -52 in 10 bits
- 
- -52
 1. # bits = 7
 2. Negative number
 - a) Representation of +52 = 0110100
 - b) Invert all bits: 1001011
 - c) Add +1: +0000001
=1001100

2's complement extension

- 52 7 bits
- a) Representation of +52 = 0110100
- b) Invert all bits: 1001011
- c) Add +1:
- +0000001
=1001100

Extension rule: 2s complement

1001100 7 bits

11001100 8 bits

111001100 9 bits

- To take a number represented in X bits can get its representation in Y bits, ($Y > X$), copy the **MSB** into the “new” bit positions

Fixed point

- After the decimal point negative powers of 2
- 0.001

1	.	1	0	1	0
2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}

$$2.625$$

Conversion from binary to decimal

- 0.43

Power of 2	Weight	Number	>=	Remainder	Bit value
2	0.5	0.43	No	0.43	0
4	0.25	0.43	Yes	0.18	1
8	0.125	0.18	Yes	0.055	1
16	0.0625	0.055	No	0.055	0
32	0.03125	0.055	Yes	0.02375	1
64	0.015625	0.02375	Yes	0.008125	1
128	0.0078125	0.008125	Yes	0.0003125	1
Represented value	0.4296875				

Power of 2	Weight	Number	>=	Remainder	Bit value
2	0.5	0.17	No	0.17	0
4	0.25	0.17	No	0.17	0
8	0.125	0.17	Yes	0.045	1
16	0.0625	0.045	No	0.045	0
32	0.03125	0.045	Yes	0.01375	1
64	0.015625	0.01375	No	0.01375	0
128	0.0078125	0.01375	Yes	0.0059375	1
Represented value	0.1640625				

Power of 2	Weight	Number	>=	Remainder	Bit value
2	0.5	0.14	No	0.14	0
4	0.25	0.14	No	0.14	0
8	0.125	0.14	Yes	0.015	1
16	0.0625	0.015	No	0.015	0
32	0.03125	0.015	No	0.015	0
64	0.015625	0.015	No	0.015	0
128	0.0078125	0.015	Yes	0.0071875	1
Represented value	0.1328125				

Power of 2	Weight	Number	>=	Remainder	Bit value
2	0.5	0.09	No	0.09	0
4	0.25	0.09	No	0.09	0
8	0.125	0.09	No	0.09	0
16	0.0625	0.09	Yes	0.0275	1
32	0.03125	0.0275	No	0.0275	0
64	0.015625	0.0275	Yes	0.011875	1
128	0.0078125	0.011875	Yes	0.0040625	1
Represented value	0.0859375				

Power of 2	Weight	Number	>=	Remainder	Bit value
2	0.5	0.01	No	0.01	0
4	0.25	0.01	No	0.01	0
8	0.125	0.01	No	0.01	0
16	0.0625	0.01	No	0.01	0
32	0.03125	0.01	No	0.01	0
64	0.015625	0.01	No	0.01	0
128	0.0078125	0.01	Yes	0.0021875	1
Represented value	0.0078125				

Power of 2	Weight	Number	>=	Remainder	Bit value
2	0.5	0.01	No	0.01	0
4	0.25	0.01	No	0.01	0
8	0.125	0.01	No	0.01	0
16	0.0625	0.01	No	0.01	0
32	0.03125	0.01	No	0.01	0
64	0.015625	0.01	No	0.01	0
128	0.0078125	0.01	Yes	0.0021875	1
256	0.00390625	0.002188	No	0.0021875	0
512	0.001953125	0.002188	Yes	0.000234375	1
Represented value	0.009765625				

Various conversions

- 1001001
- Decimal value interpreted as unsigned representation?
- Decimal value interpreted as 2's complement representation?
- $0.1001001 = ?$

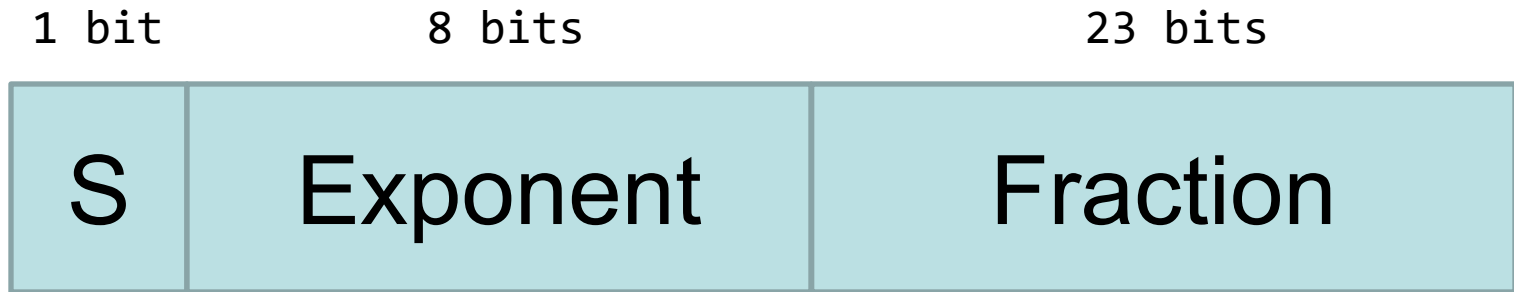
Floating point

- Allows representing very large and very small numbers
- Standard used in all machines today
 - Much interesting theory behind it
- It is equivalent to scientific notation but done in binary.
- Think:
 - $0.145 = +1.45 * 10^{-2}$
 - $0.0090897 = +9.0897 * 10^{-3}$
 - $-145 = -1.45 * 10^{+2}$
- How do we do this with binary?

Floating Point Standard

IEEE-754 Standard

Single Precision Representation



- $N = -1^S * 1.fraction * 2^{exponent-127}$
when $1 \leq exponent \leq 254$
- $N = -1^S * 0.fraction * 2^{-126}$
when $exponent == 0$

Example

1100000011001000100000000000000000

1 10000001 10010000000000000000000000000000

$S = 1$; therefore negative number
exponent = 129

fraction = 1001000000000000000000000000

$$N = -1^1 * 1.1001 * 2^{129-127}$$

$$N = -1^1 * 1.1001 * 2^2$$

$$N = -1^1 * 110.01$$

$$N = 6.25$$

CORRECTION: N should be -6.25

Example 2

010000**1**011001000100000000000000000000

0 10000**1**01 100100000000000000000000000000

S = 1 ; therefore negative number
exponent = 133

fraction = 100100000000000000000000000000

$$N = -1^1 * 1.1001 * 2^{133-127}$$

$$N = -1^1 * 1.1001 * 2^5$$

$$N = -1^1 * 110010$$

$$N = 50$$

CORRECTION: S = 0; therefore positive number

All the -1^1 should be -1^0

Example 3

743.5

0010 1110 0111.10

= 1.01110 0111.10 * 2^9

1) *fraction* = 011100111

2) Exponent-127=9 \Rightarrow Exponent = 136

10001000

3) Final representation

0 10001000 0111 0011 1000 0000 0000 000

CORRECTION: fraction is missing another 1 at the right

Special cases

<http://blogs.msdn.com/b/premk/archive/2006/02/25/539198.aspx>

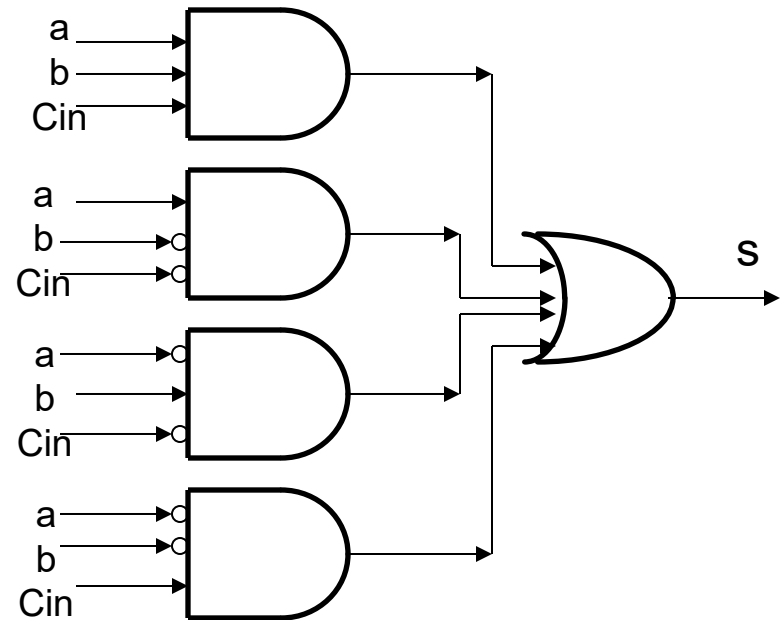
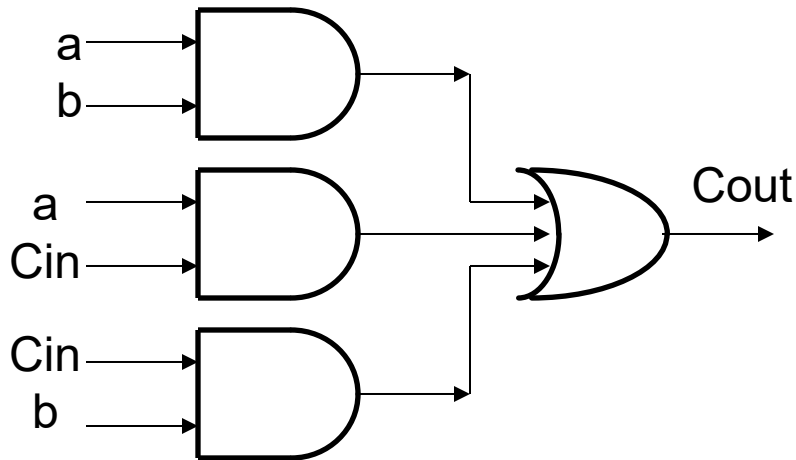
Sign(s)	Exponent (e)	Mantissa (m)	Range for Single Precision values in binary	Range Name
1	11..11	10..00 : 11.11	—	QNaN
1	11..11	00..01 : 01..11	—	SNaN
1	11..11	00..00	$< -(2-2^{-23}) \times 2^{127}$	-Infinity (Negative Overflow)
1	11..10 : 00..01	11..11 : 00..00	$-(2-2^{-23}) \times 2^{127}$: -2^{-126}	Negative Normalized $-1.m \times 2^{(e-b)}$
1	00..00	11..11 : 00..01	$-(1-2^{-23}) \times 2^{-126}$: -2^{-149}	Negative Denormalized $-0.m \times 2^{(-b+1)}$
—	—	—	-2^{-150} : < -0	Negative Underflow
1	00..00	00..00	-0	-0
0	00..00	00..00	+0	+0
—	—	—	$> +0$: 2^{-150}	Positive Underflow

Today

- Review representations (252/352 recap)
- Floating point
- **Addition: Ripple carry adder, carry-look-ahead adder**
- Barrel Shifter
- Multiplication: Simple and Radix-4
- Division: Simple and Newton-Raphson
- Floating point: Addition, multiplication

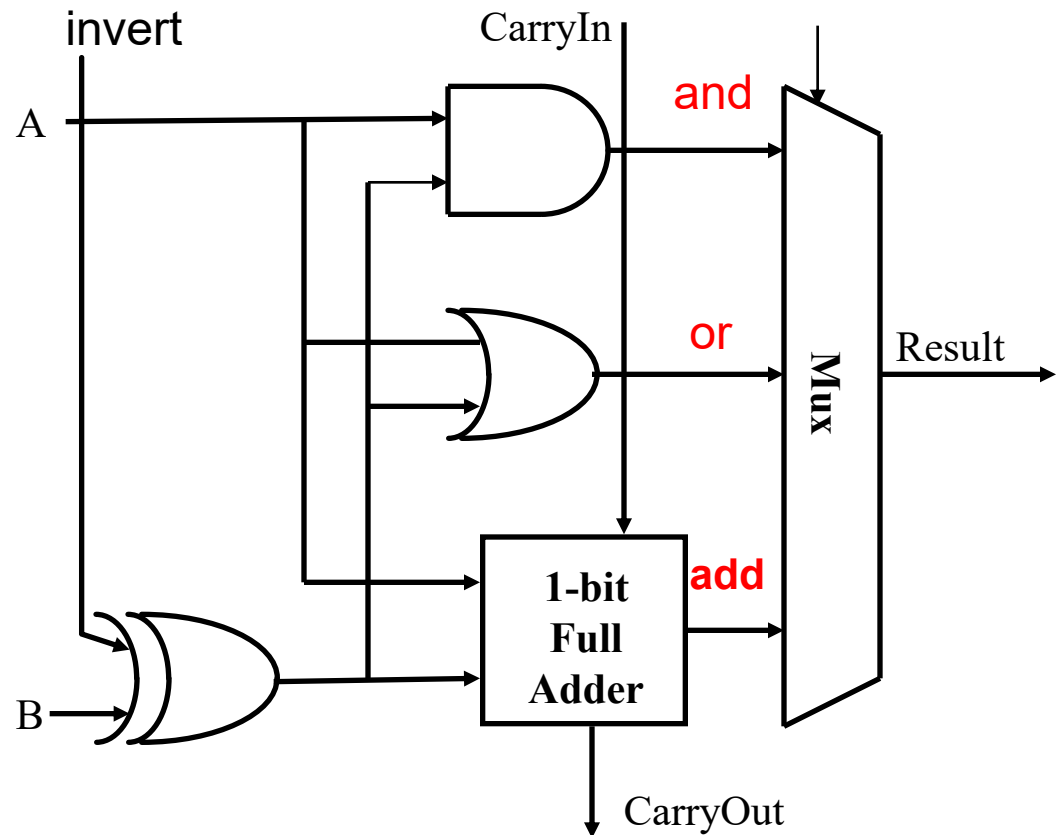
Full adder

- Three inputs and two outputs
- $C_{out}, s = F(a, b, C_{in})$
 - C_{out} : only if **at least two** inputs are set
 - S : only if **exactly one** input or **all three** inputs are set
- Logic?

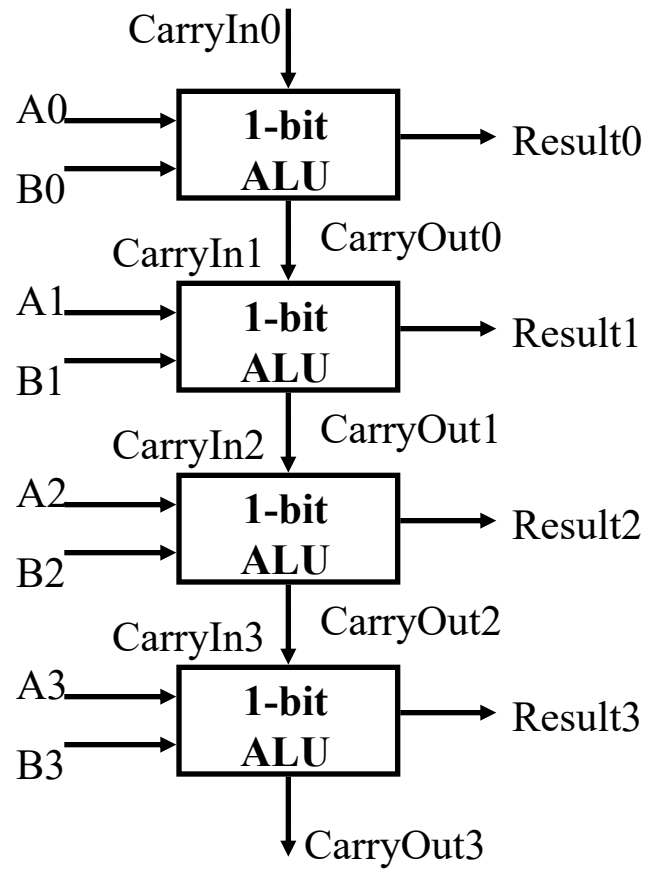


Subtract

- $A - B = A + (-B)$
 - form two complement by invert and add one



Ripple-carry adder



Problem : Slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
 - Delay = 32x CP(Fast adder) + XOR
- Is there more than one way to do addition?
 - two extremes: ripple carry and sum-of-products
 - Flatten expressions to two levels

Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$

$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$

$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$

$$c_2 = b_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1b_1$$

$$c_2 = b_1b_0c_0 + b_1a_0c_0 + b_1a_0b_0 + a_1b_0c_0 + a_1a_0c_0 + a_1a_0b_0 + a_1b_1$$

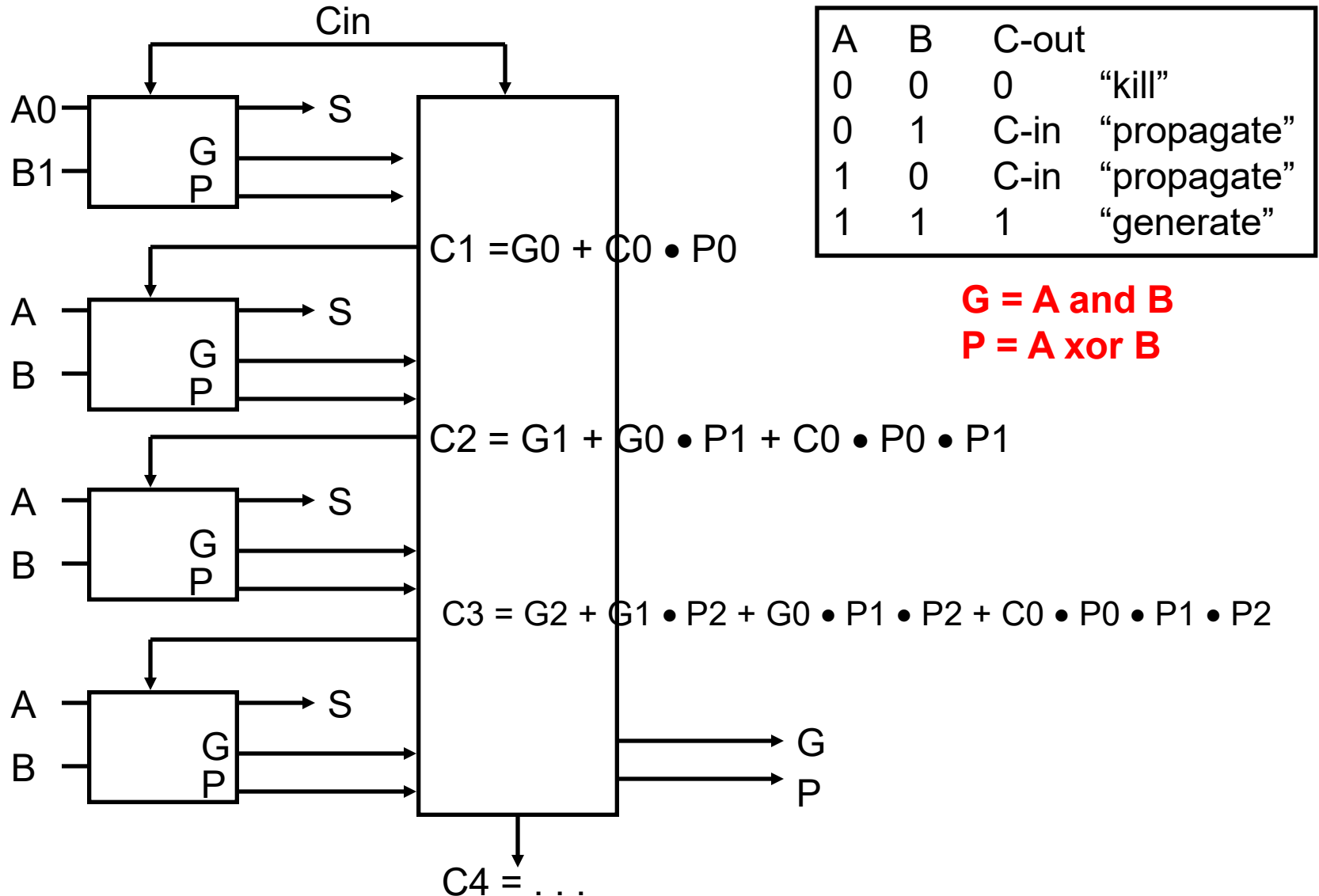
$$c_3 = ?$$

$$c_{31} = ? \text{ Not feasible! Why? Exponential fanin}$$

Carry look-ahead

- An approach in-between our two extremes
- Motivation:
 - If we didn't know the value of carry-in, what could we do?
 - When would we always **generate** a carry?
 - $g_i = a_i b_i$
 - When would we **propagate** the carry?
 - $p_i = a_i + b_i$
- Did we get rid of the ripple?

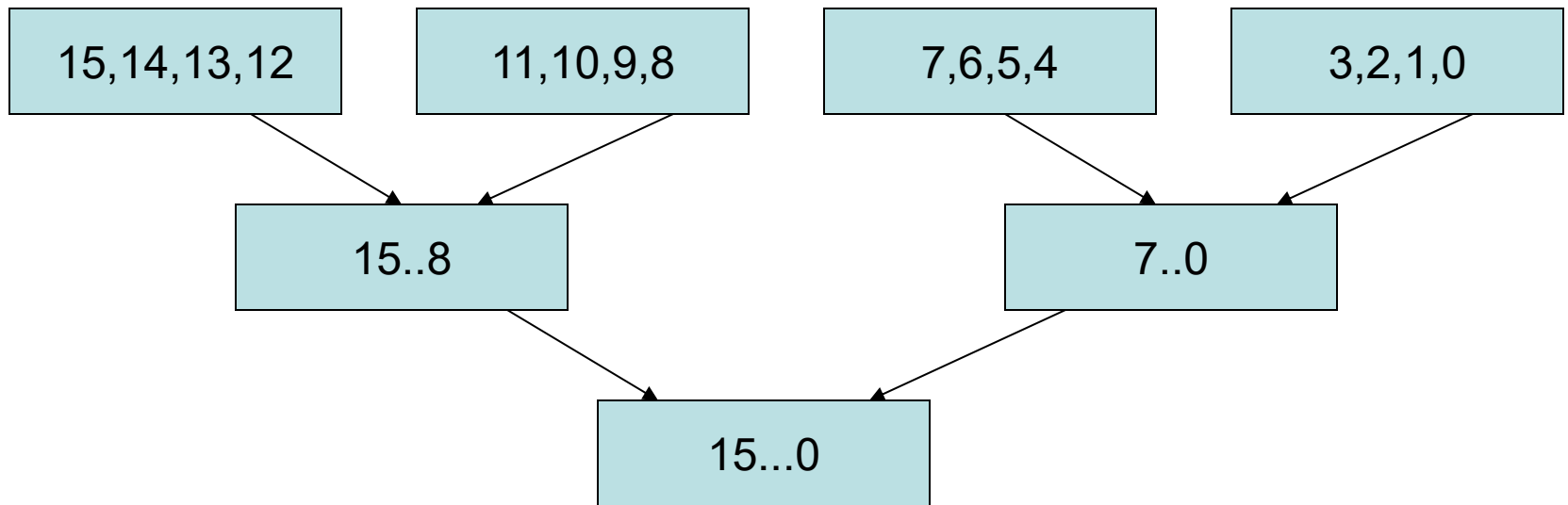
Carry-lookahead adder



Carry-Lookahead Adder

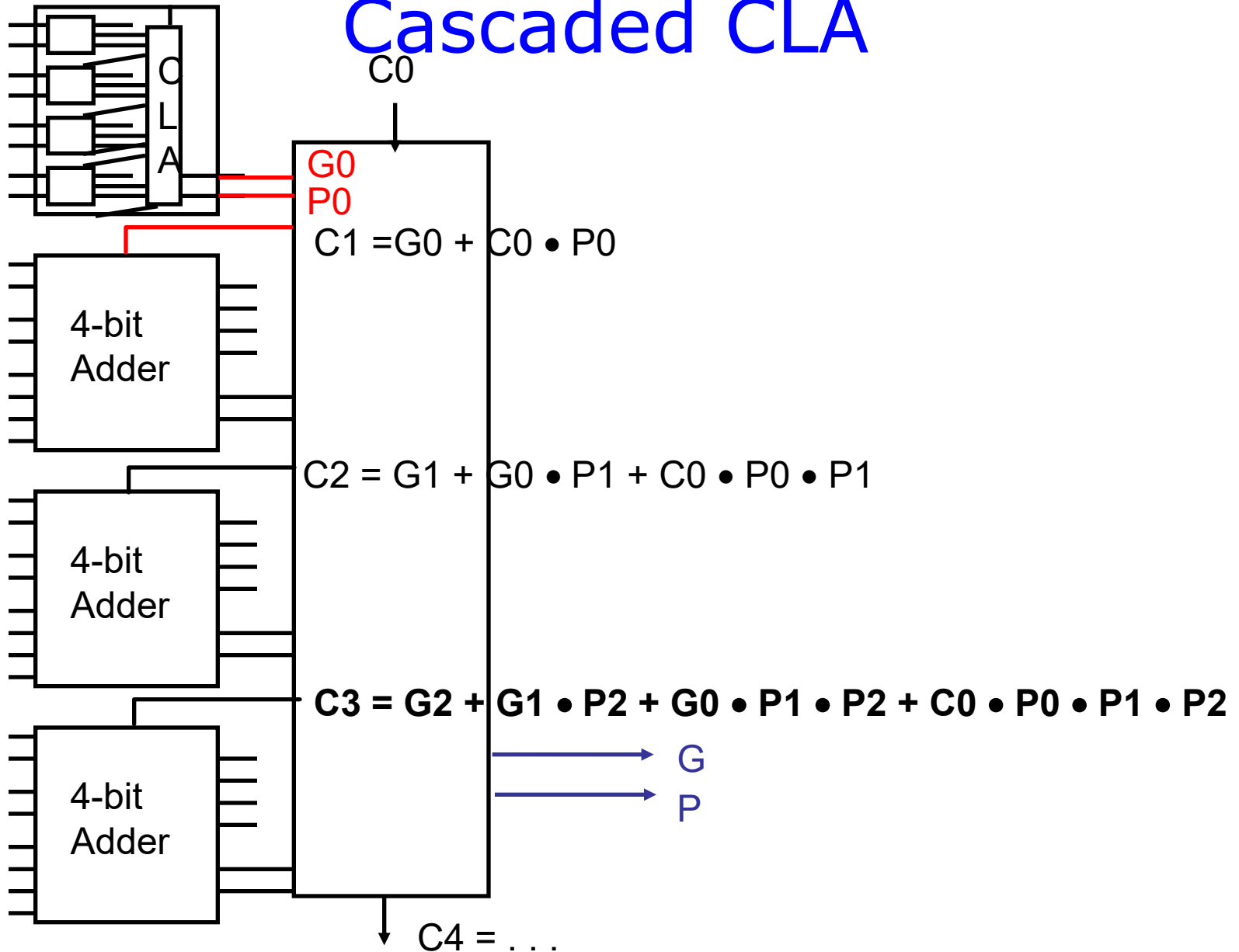
- Wait a minute!
 - Nothing has changed
 - Fanin problems if you flatten!
 - Linear fanin, not exponential
 - Ripple problem if you don't!
- Enables divide-and-conquer
- Figure out Generate and Propagate for 4-bits together
- Compute hierarchically

Carry Lookahead adder



- Height of tree = $O(\lg(n))$
- 32 bit addition : $k * \lg(32) = k * 5$

Cascaded CLA



Carry-Lookahead Adder

- Hierarchy

- $G_{i,k} = G_{j+1,k} + P_{j+1,k} * G_{i,j} \quad (\text{assume } i < j + 1 < k)$

- $P_{i,k} = P_{i,j} * P_{j+1,k}$

- $G_{0,7} = G_{4,7} + P_{4,7} * G_{0,3}$

- $P_{0,7} = P_{0,3} * P_{4,7}$

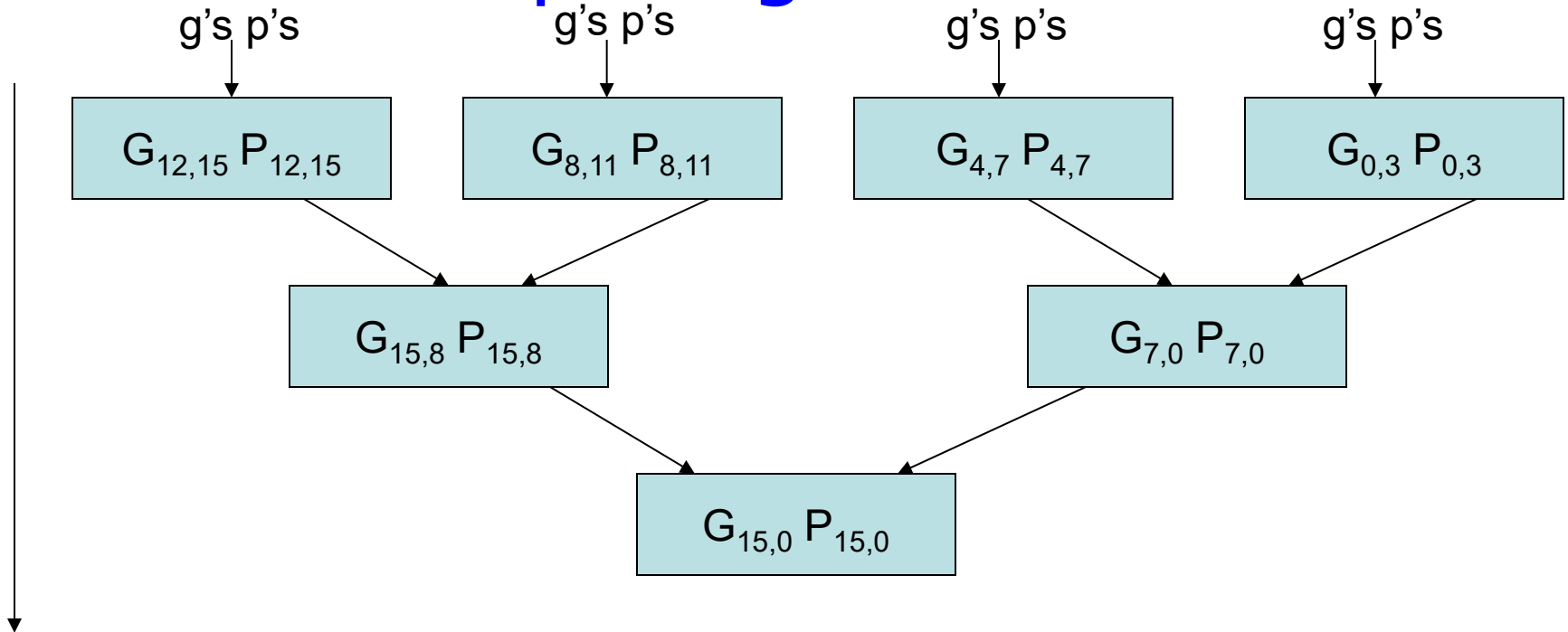
- $G_{8,15} = G_{12,15} + P_{12,15} * G_{8,11}$

- $P_{8,15} = P_{8,11} * P_{12,15}$

- $G_{0,15} = G_{8,15} + P_{8,15} * G_{0,7}$

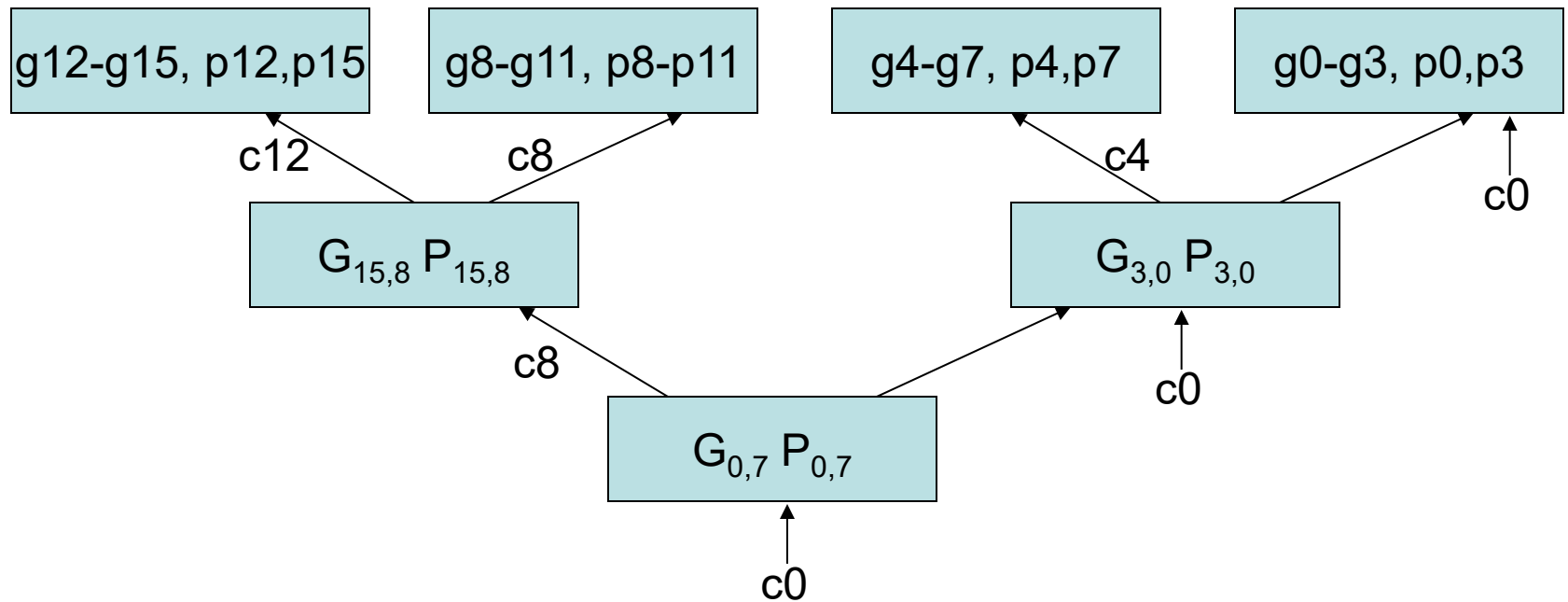
- $P_{0,15} = P_{0,7} * P_{8,15}$

Computing G's and P's



- Parallel algorithm in hardware

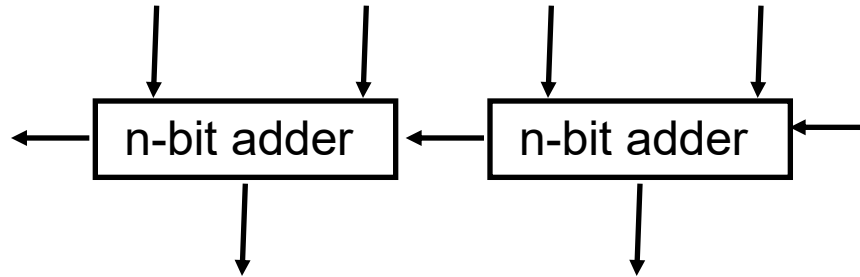
Computing C's



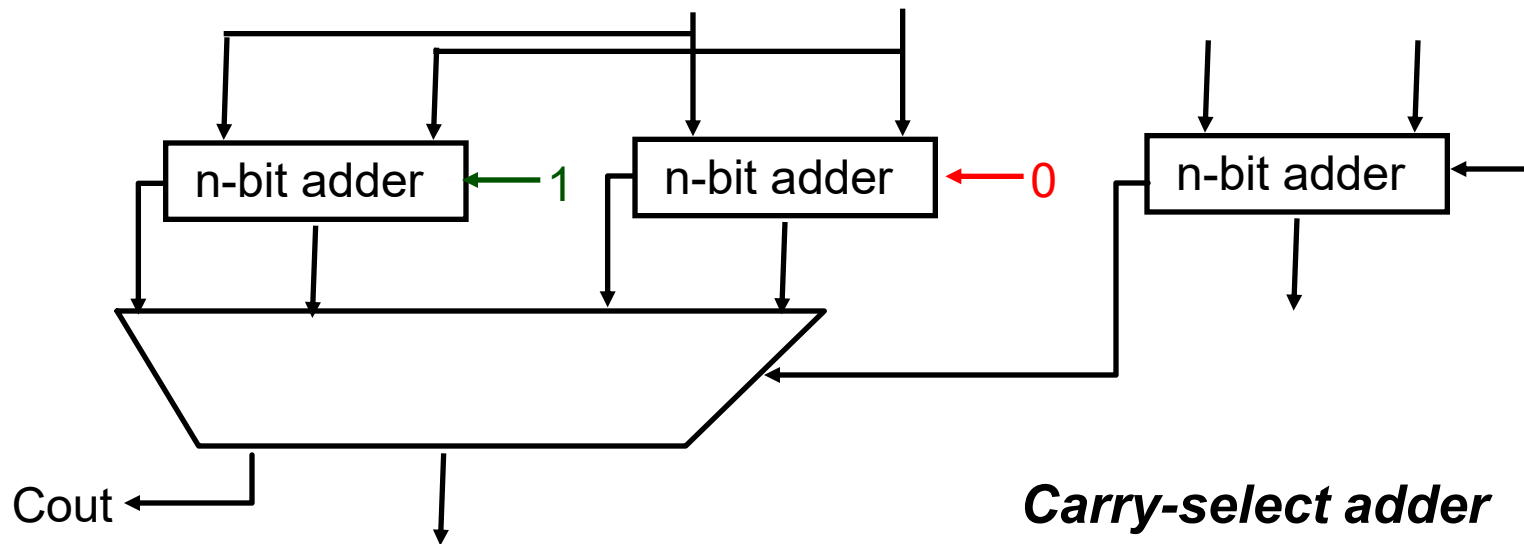
- Different tree.
- Note propagation order
- Worth spending some time to think about the intricacies

Carry-selection: Guess

$$CP(2n) = 2 * CP(n)$$

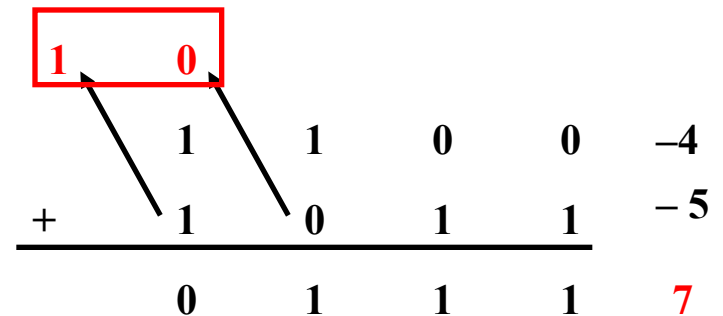
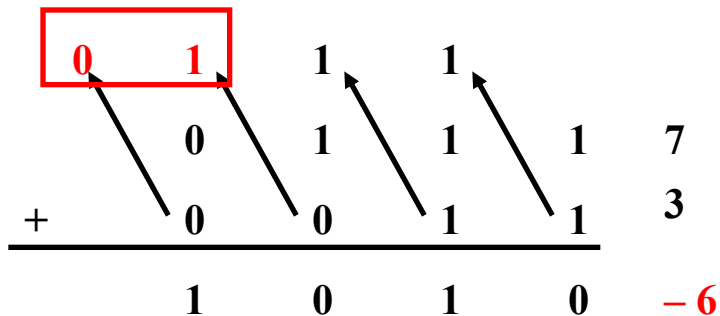


$$CP(2n) = CP(n) + CP(\text{mux})$$



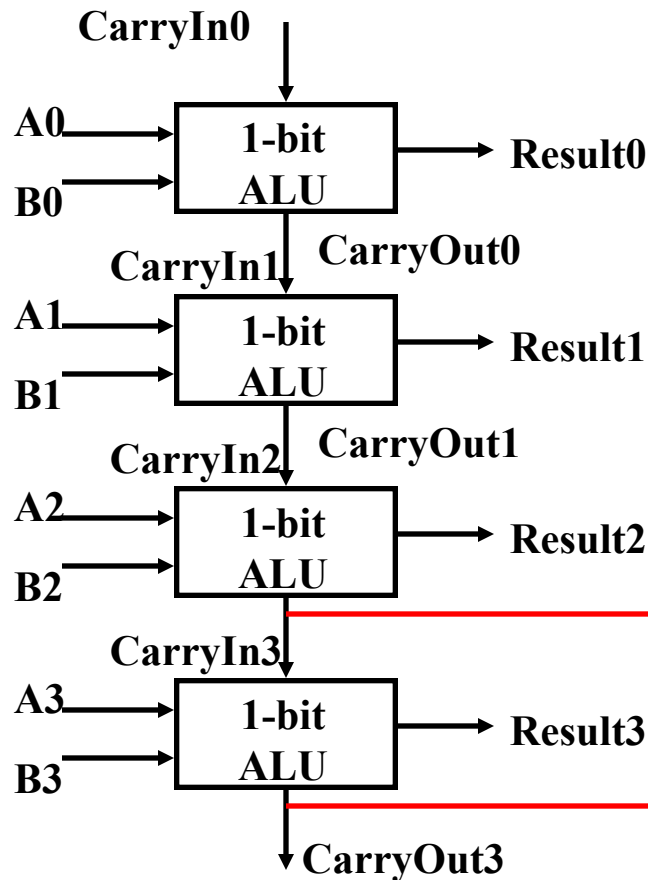
Overflow

- Overflow: the result is too large (or too small) to represent properly
 - Example: $-8 < \text{4-bit binary number} \leq 7$
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
 - 2 positive numbers and the sum is negative
 - 2 negative numbers and the sum is positive
- On your own: Prove you can detect overflow by:
 - Carry into MSB \oplus Carry out of MSB

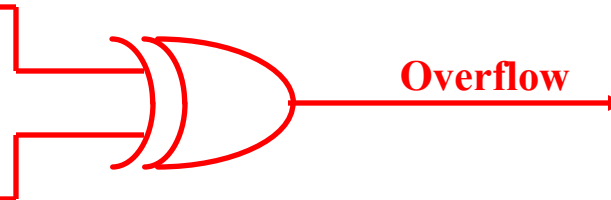


Overflow detection

- Carry into MSB \oplus Carry out of MSB
 - For N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0



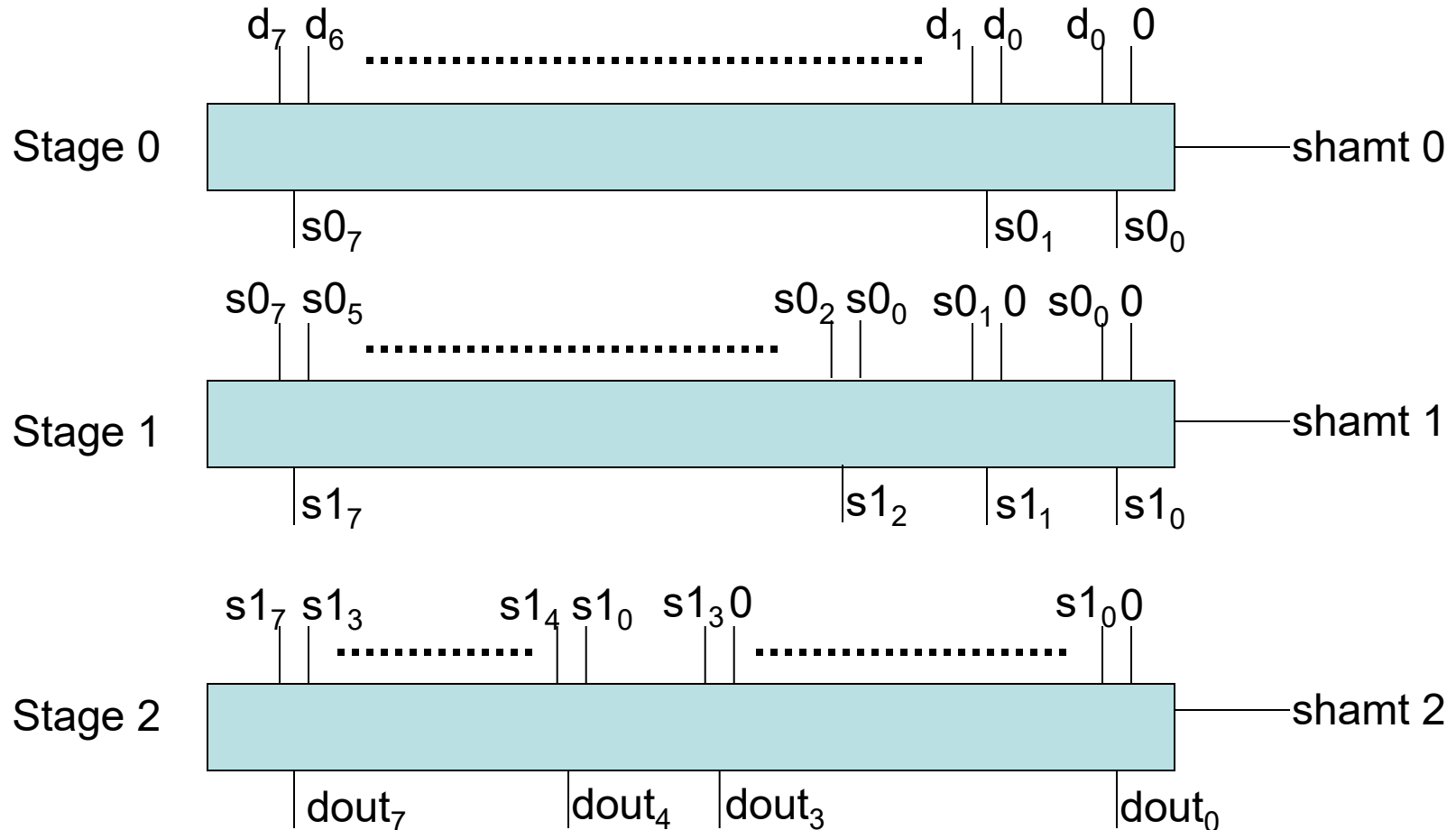
Negative, Zero

- Required for conditional branches
- Zero
 - How?
 - NOR all 32 bits
 - Avoid 33rd bit (carry out)
- Negative may be required on overflow
 - If (a<b) jump : jump taken if a-b is negative
- Tempting to consider MSB
 - E.g. if (-5 < 4) branch
 - Branch should be taken, but (-5-4) computation results in overflow... so MSB is 0
 - E.g. if (7 < -3) branch
 - Branch should not be taken but (7- (-3)) results in overflow... so MSB is 1.

Shift

- E.g., Shift left logical for $d\langle 7:0 \rangle$ and $shamt\langle 2:0 \rangle$
 - Using 2-1 muxes called $Mux(select, in0, in1)$
 - $stage0\langle 7:0 \rangle = Mux(shamt\langle 0 \rangle, d\langle 7:0 \rangle, 0 \parallel d\langle 7:1 \rangle)$
 - $stage1\langle 7:0 \rangle = Mux(shamt\langle 1 \rangle, stage0\langle 7:0 \rangle, 00 \parallel stage0\langle 6:2 \rangle)$
 - $dout\langle 7:0 \rangle = Mux(shamt\langle 2 \rangle, stage1\langle 7:0 \rangle, 0000 \parallel stage1\langle 3:0 \rangle)$
- Other operations
 - Right shift
 - Arithmetic shifts
 - Rotate

Barrel Shifter



MK

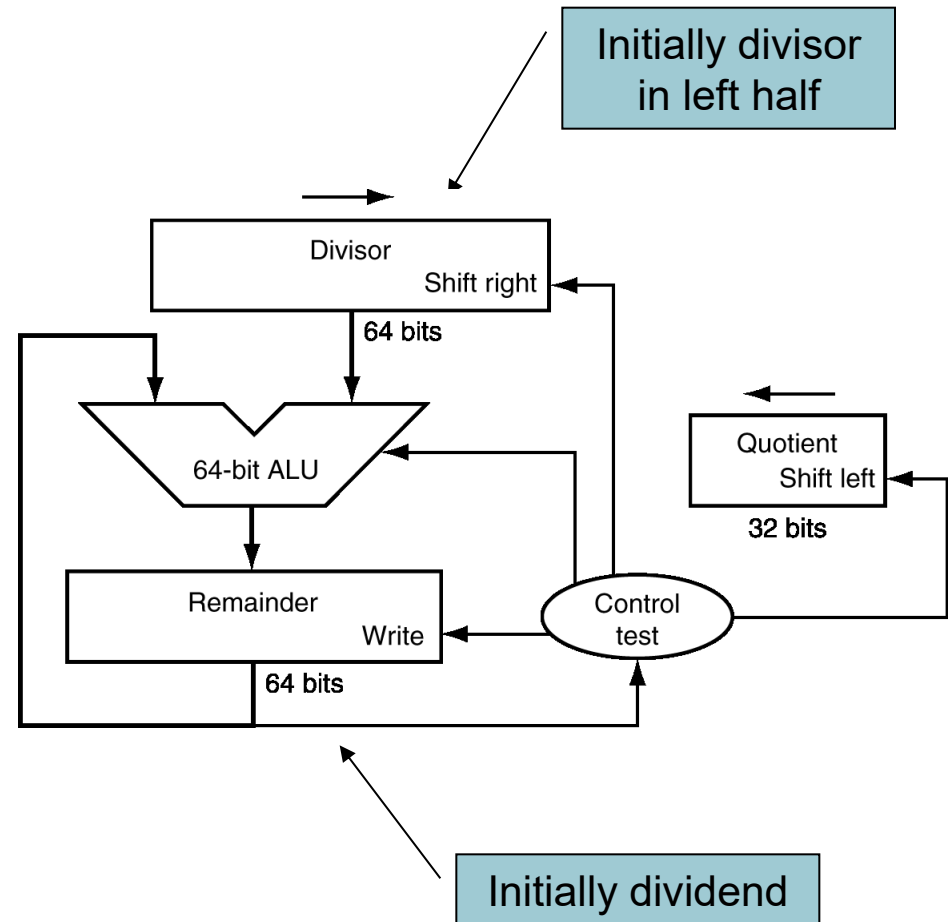
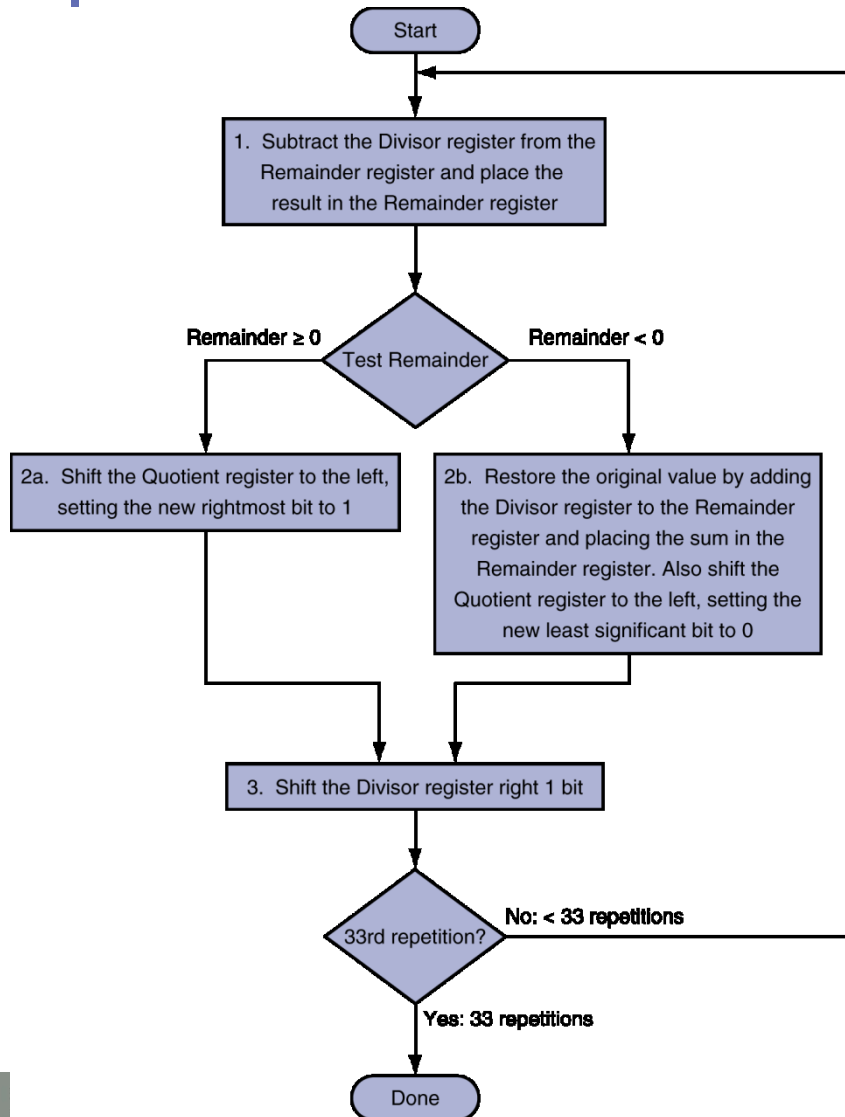


Chapter 3 — Arithmetic for Computers — 64

Multiplication: Radix-4

- Basic idea: look at more bits at any one time
- Entire courses on this.
- This is a great set of slides:
https://www.ece.ucsb.edu/~parhami/pres_f_older/f31-book-arith-pre-pt3.ppt
UCSB.

Division Hardware



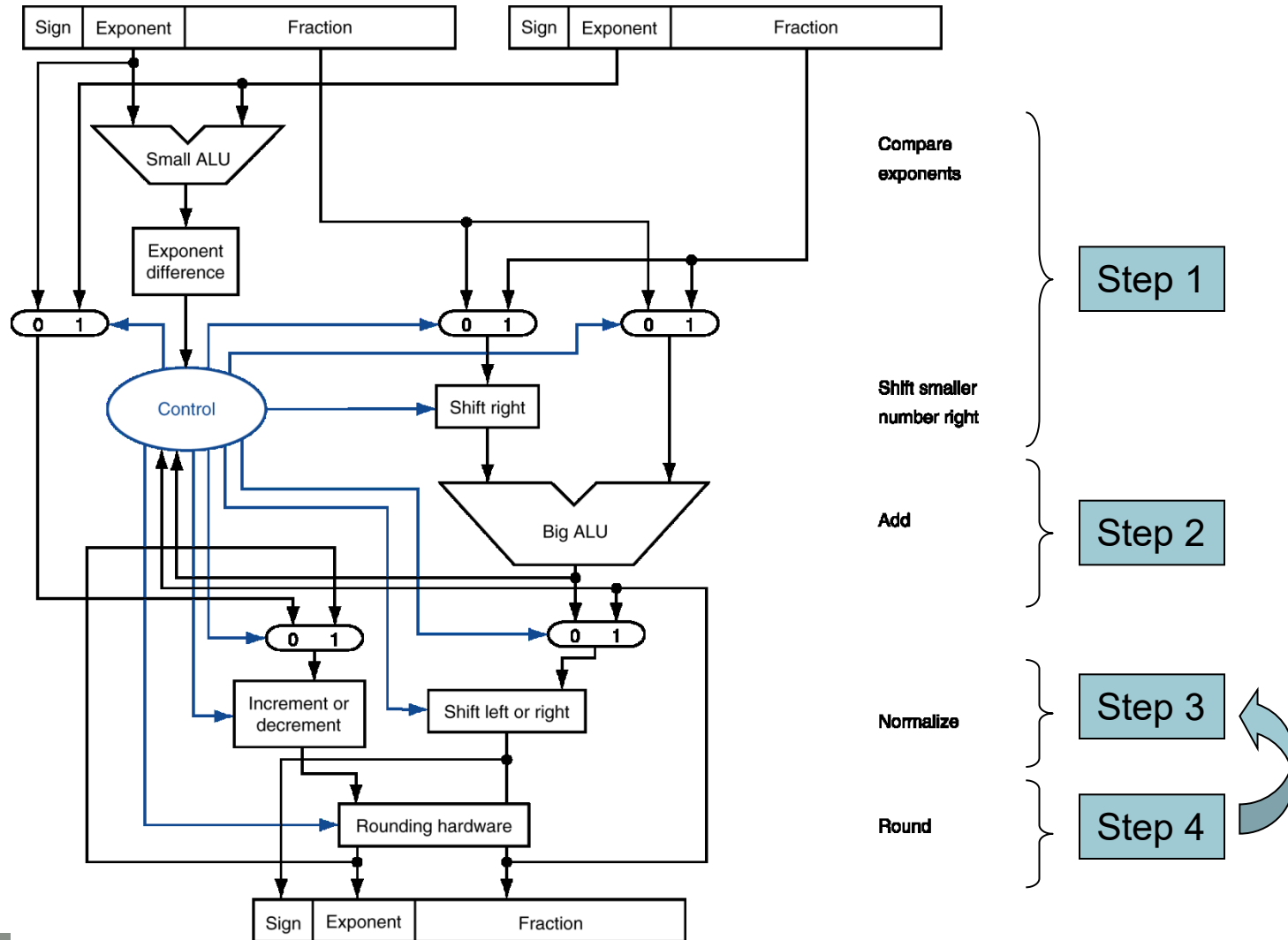
Division complex

- Newton-Raphson adaption
- SRT division
- See slides at very end of this slide deck

Floating point Addition

- Subtract exponents ($d = E_x - E_y$)
- Align significands
 - Shift right d positions the significand of the operand with the smaller exponent
 - Select as the exponent of the result the largest exponent
- Add significands and produce sign of result
- Normalization
 - Already normalized – no action
 - Overflow (for addition): shift right the significand one bit position, and increment exponents by one
 - Underflow (for subtraction): shift left significand by the number of leading zeroes, decrement the exponent by same number
- Round (according to rounding mode)
- Determine exception and special flags

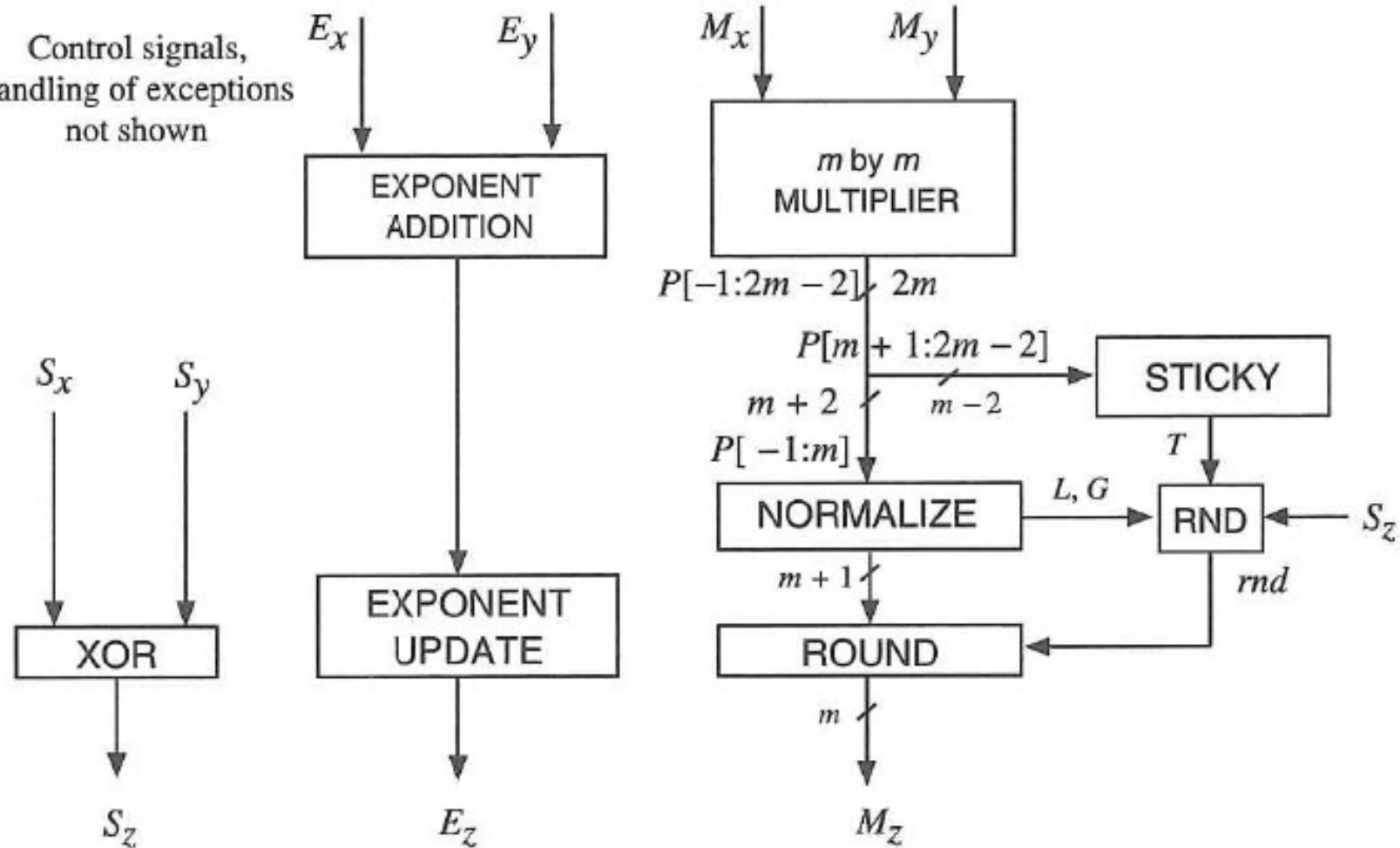
FP Adder Hardware



Floating Point Multiplication

- Multiplication of magnitudes, addition of exponents and generation of sign
- Normalization
- Rounding modes

Control signals,
handling of exceptions
not shown



Today

- Review representations (252/352 recap)
- **Floating point**
- **Addition: Ripple carry adder, carry-look-ahead adder**
- **Barrel Shifter**
- **Multiplication: Simple and Radix-4**
- **Division: Simple and Newton-Raphson**
- **Floating point: Addition, multiplication**

Division in detail

- All this material is completely optional

Goldschmidt Division And the AMD K7 Implementation



TONY NOWATZKI
CS/ECE 755

Outline



1. Motivation
2. Algorithm Fundamentals (with Demo)
3. Benefits and Drawbacks
4. Implementation
 - a. AMD K7 and Existing Multiplier
 - b. Augmentations to the Multiplier
 - c. Reciprocal Tables
5. Additional Optimizations
 - a. Multiply under Division
 - b. Early Completion

Is Division Important?



- Spec 92: (Assume: 3 Cycle Add/Mult, and 20 Cycle Divide)

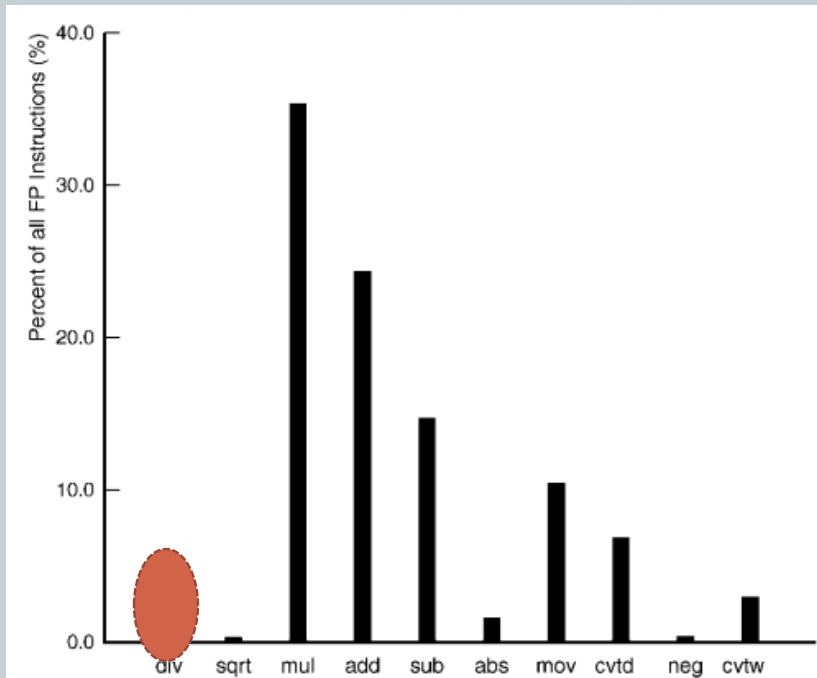


Fig. 1. Distribution of floating point instructions.

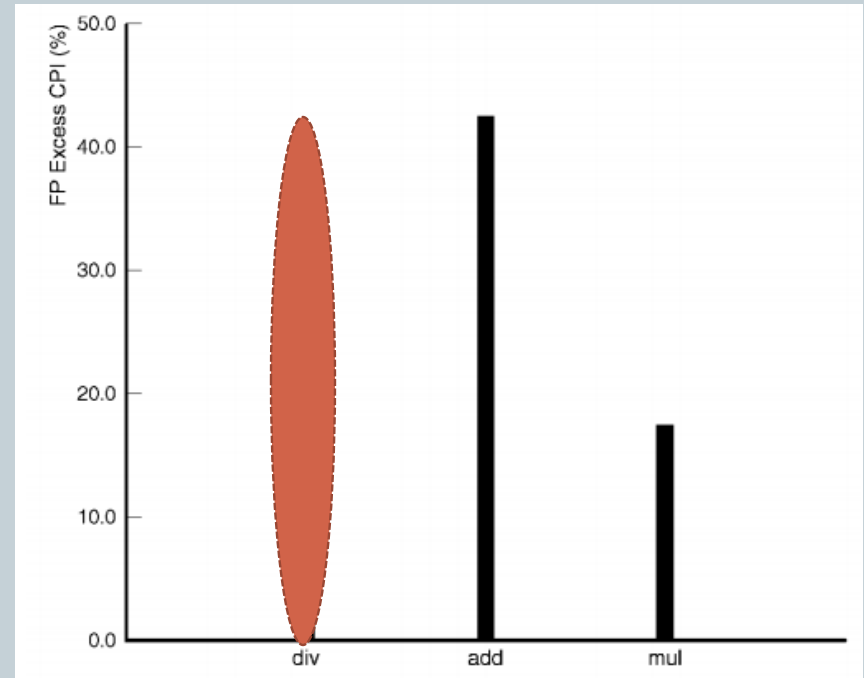


Fig. 2. Distribution of functional unit stall time.

- [Source: S.F. Oberman and M.J. Flynn, “Design issues in division and other floating-point operations,” IEEE Trans.Computers, vol.46, no.2, pp.154–161, Feb. 1997.]

Goldschmidt: The Idea



- Principle 1:

- Any factor that multiplies both the numerator and denominator will leave the quotient unchanged.

$$\frac{N \times X}{D \times X} = \frac{N}{D}$$

- Principle 2:

- If, through successive multiplication, you can drive the denominator to 1, then the new numerator will be the quotient.

$$\frac{N \times X_1 \times X_2 \times X_3}{D \times X_1 \times X_2 \times X_3} = \frac{N / D}{1}$$

Goldschmidt: The Idea (2)



- How do we determine X_i ?
 - (by complex Mathematical Proof)
- Initial Value is reciprocal estimate

$$X_0 = \text{estimate} \left(\frac{1}{D} \right)$$

- Subsequent Values:

$$X_i = 2 - D_i$$

Goldschmidt: Summary



- Initialization:

$$X_0 = \text{estimate}(1/D) \quad N_0 = N \quad D_0 = D$$

- Each Iteration:

$$N_i = N_{i-1} \times X_{i-1}$$

$$D_i = D_{i-1} \times X_{i-1}$$

$$X_i = 2 - D_i$$

- Finalization:

$$N_f = N_{f-1} \times X_{f-1}$$

- Demo!

Properties of Goldschmidt



The Good:

- 😊 Quadratic Convergence
- 😊 Only 2 Multiplies Per Iteration
- 😊 Highly Parallel
 - 😊 Both Multiplies are independent.

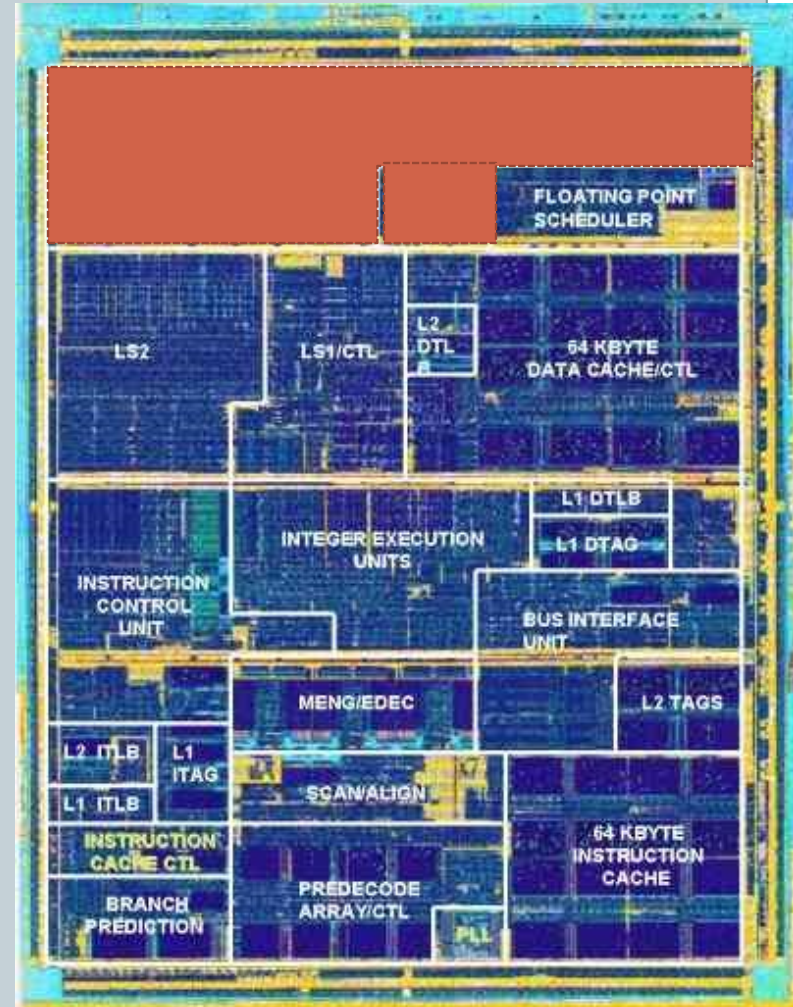
The Bad:

- 😞 Need Accurate Reciprocal Lookup
 - 😞 Algorithm may fail to converge if guess is $<75\%$ or $>150\%$ of what it should be
- 😞 Goldschmidt Is Not Self Error-Correcting

AMD K7



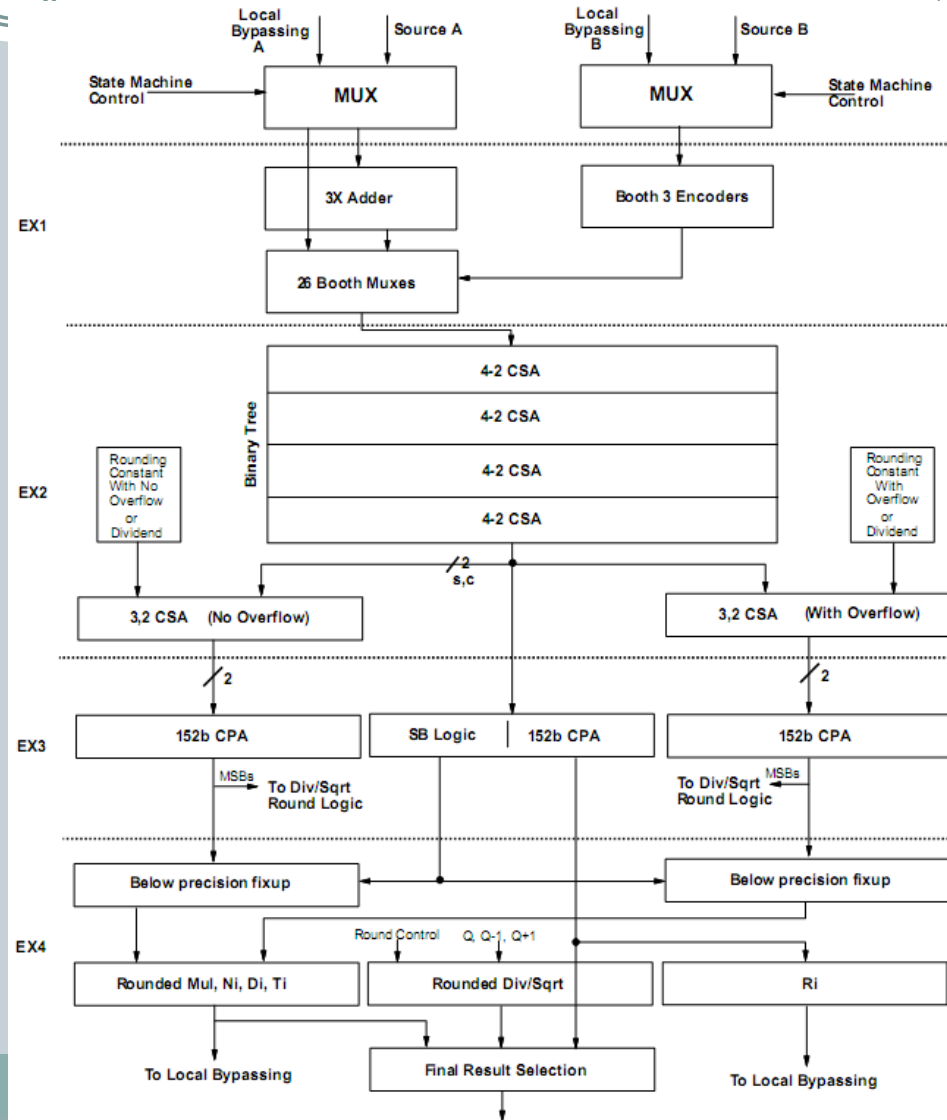
- Released in 1999
- Relatively Modern Design
 - 3-Way Superscalar
 - Out of Order Processing
 - 15 Stage Pipeline
 - 500+ MHZ
 - Separate Floating Point / Integer Scheduler
- Parts of Chip Modified:
 - Floating Point Execution Units (just the multiplier)
 - Floating Point Control Unit



AMD K7 Multiplier

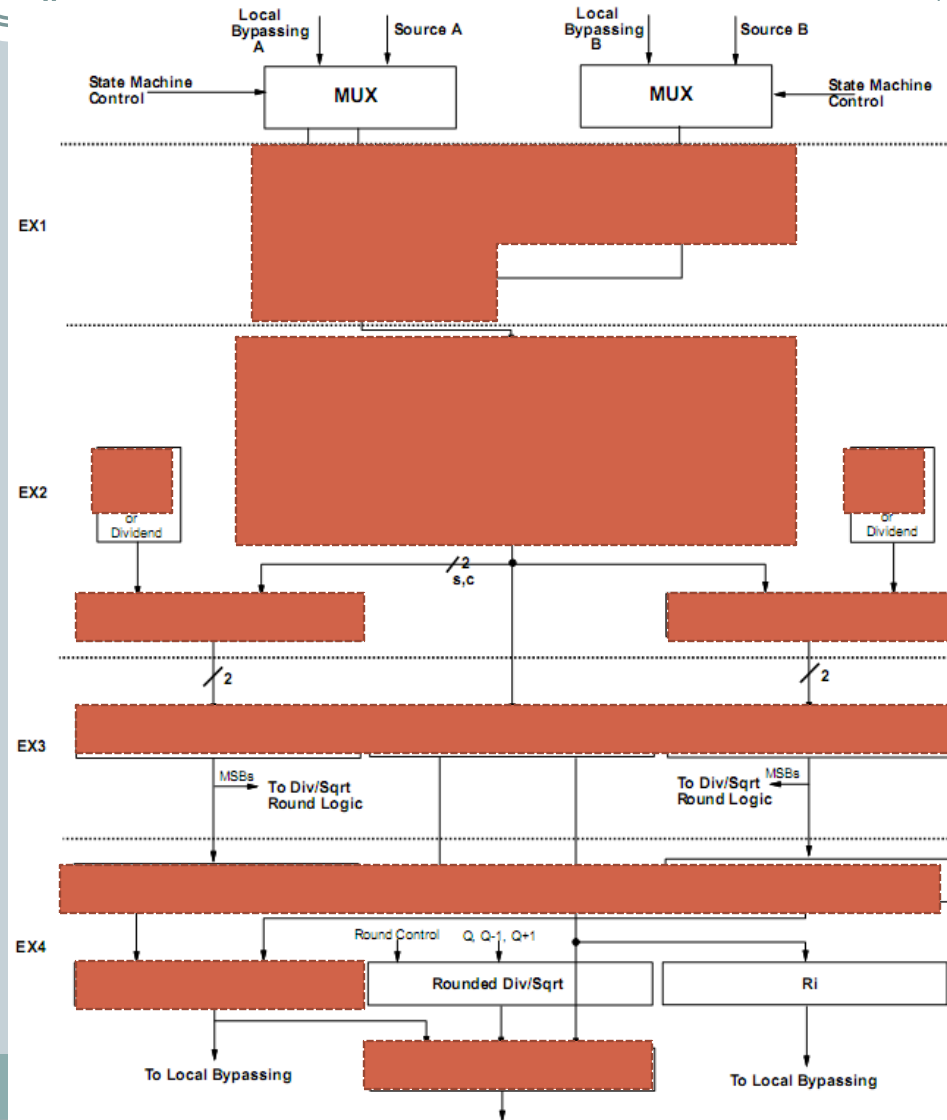
- Booth Style Multiplier
- Uses Static CMOS
- 4 Cycle Latency
- Fully Pipelined
- 76 Bit Width
- Local Bypassing
- X87 and IEEE Compliant

(Note: this pipeline is for the mantissa or 'fraction')



AMD K7 Multiplier (2)

- **EX1:**
 - Partial products generated by Booth 3 Algorithm
- **EX2:**
 - Carry Select Tree adds partial products
 - Rounding Constant Added
 - Normalized twice assuming overflow, or no overflow.
- **EX3:**
 - Final Unrounded Result Assimilated
- **EX4:**
 - X87 Compliant Rounding
 - Result Selection



State Machine



Get Estimate of (1/D)



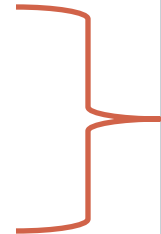
BEGIN_DIVISION: Input = (a, b, pc, rc) Output = (q_f)

$x_0 = \text{recip_estimate}(b)$

Perform Numerator and Denominator Multiplications

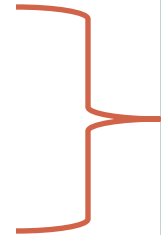


$d_0 = \text{ITERMUL}(x_0, b), r_0 = \text{comp1}(d_0)$
 $n_0 = \text{ITERMUL}(x_0, a)$
if (PC == SINGLE)
 $n_f = n_0, r_f = r_0$
 goto END_DIVISION



Iteration 1

$d_1 = \text{ITERMUL}(d_0, r_0), r_1 = \text{comp1}(d_1)$
 $n_1 = \text{ITERMUL}(n_0, r_0)$
if (PC == DOUBLE)
 $n_f = n_1, r_f = r_1$
 goto END_DIVISION

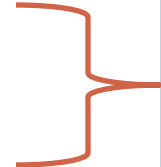


Iteration 2

Calculate Quotient With Correct Precision



$d_2 = \text{ITERMUL}(d_1, r_1), r_2 = \text{comp1}(d_2)$
 $n_2 = \text{ITERMUL}(n_1, r_1)$
 $n_f = n_2, r_f = r_2$



Iteration 3

END_DIVISION:

$q_i = \text{LASTMUL}(n_f, r_f, pc)$

Calculate Remainder



$rem = \text{BACKMUL}(q_i, b, a), q_f = \text{round}(q_i, rem, pc, rc)$

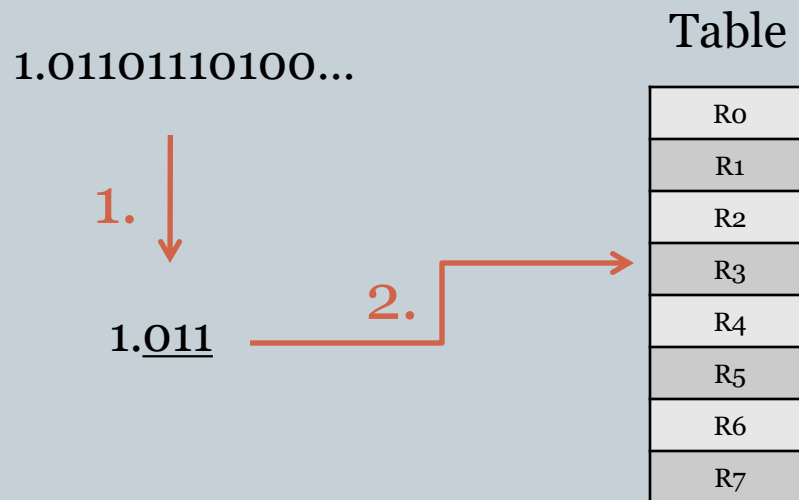


Perform Rounding

Standard Reciprocal Lookup



- Reciprocal Lookup typically follows this procedure:
 1. Truncate Input Bits to Size of Table
 2. Index Into Reciprocal Table

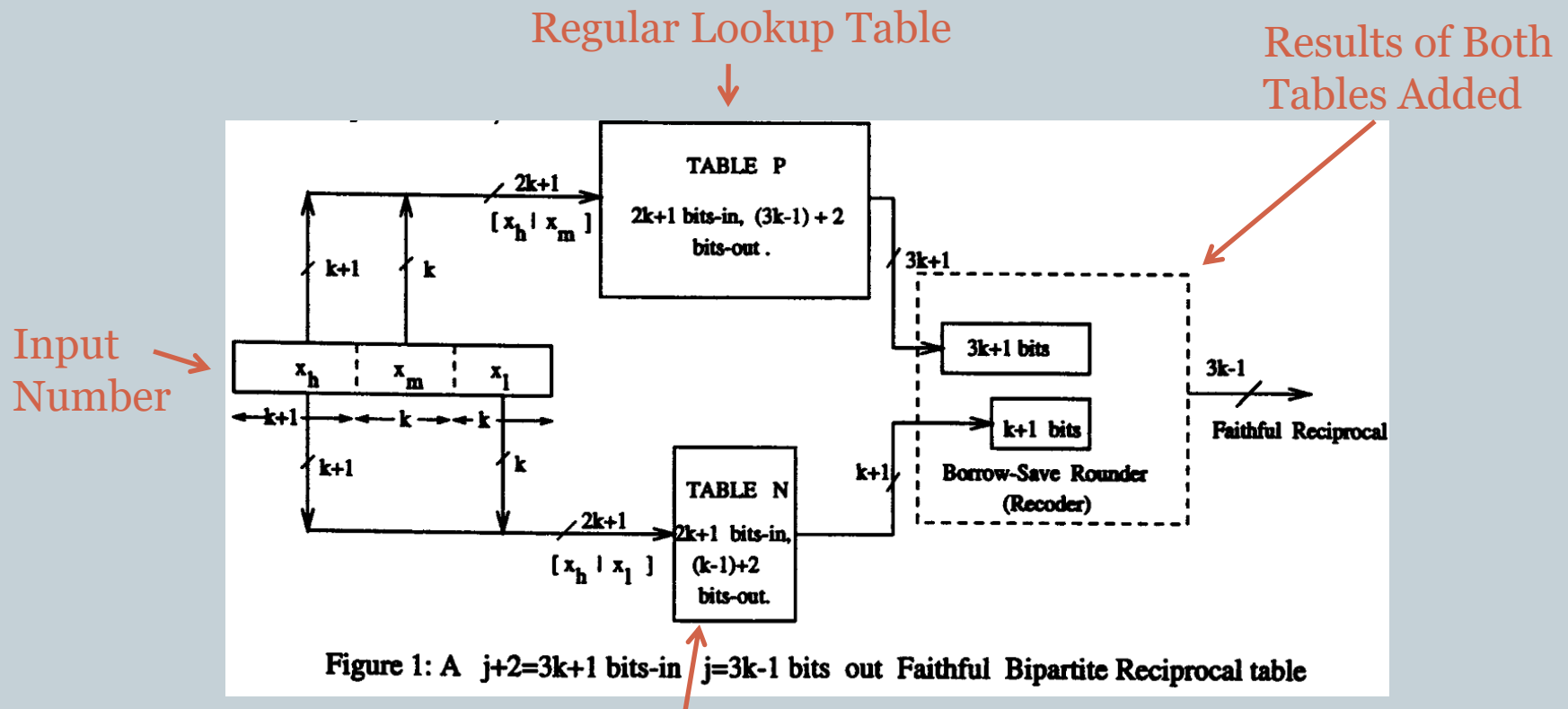


Dual Table Reciprocal Lookup



- Lookup table in the K7 is much cooler!
- Uses a Novel Compression Scheme to increase the precision/table size.
- Two Tables: Regular Table and Offset Table
- Basic Concept:
 - Parallel Lookup in Both Tables, then combine (add) the results.

Dual Table Lookup Implementation



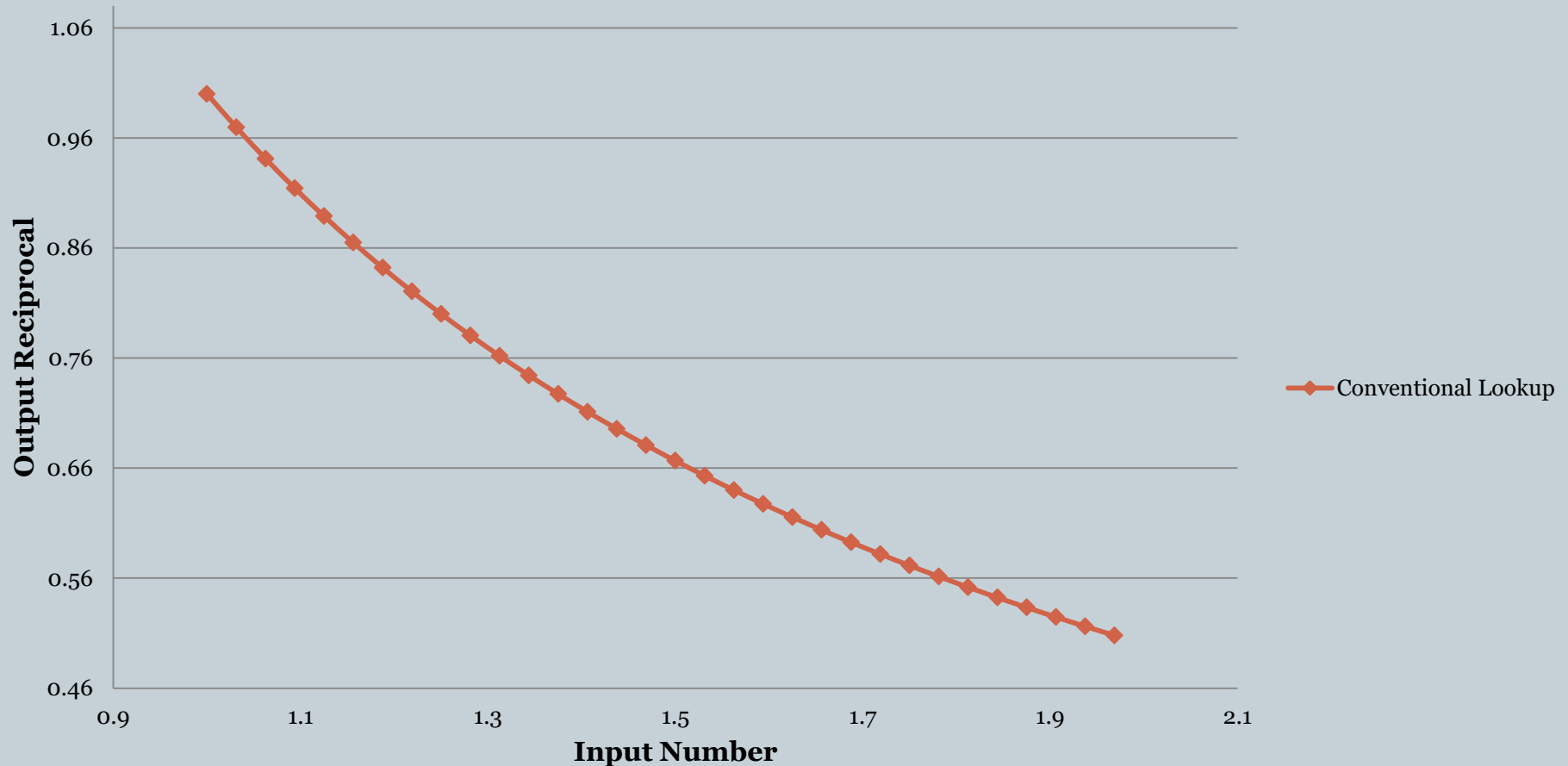
Offset Lookup Table

Question: How do we determine the values in the offset table?

Dual Table Explanation 1



Reciprocal Lookup

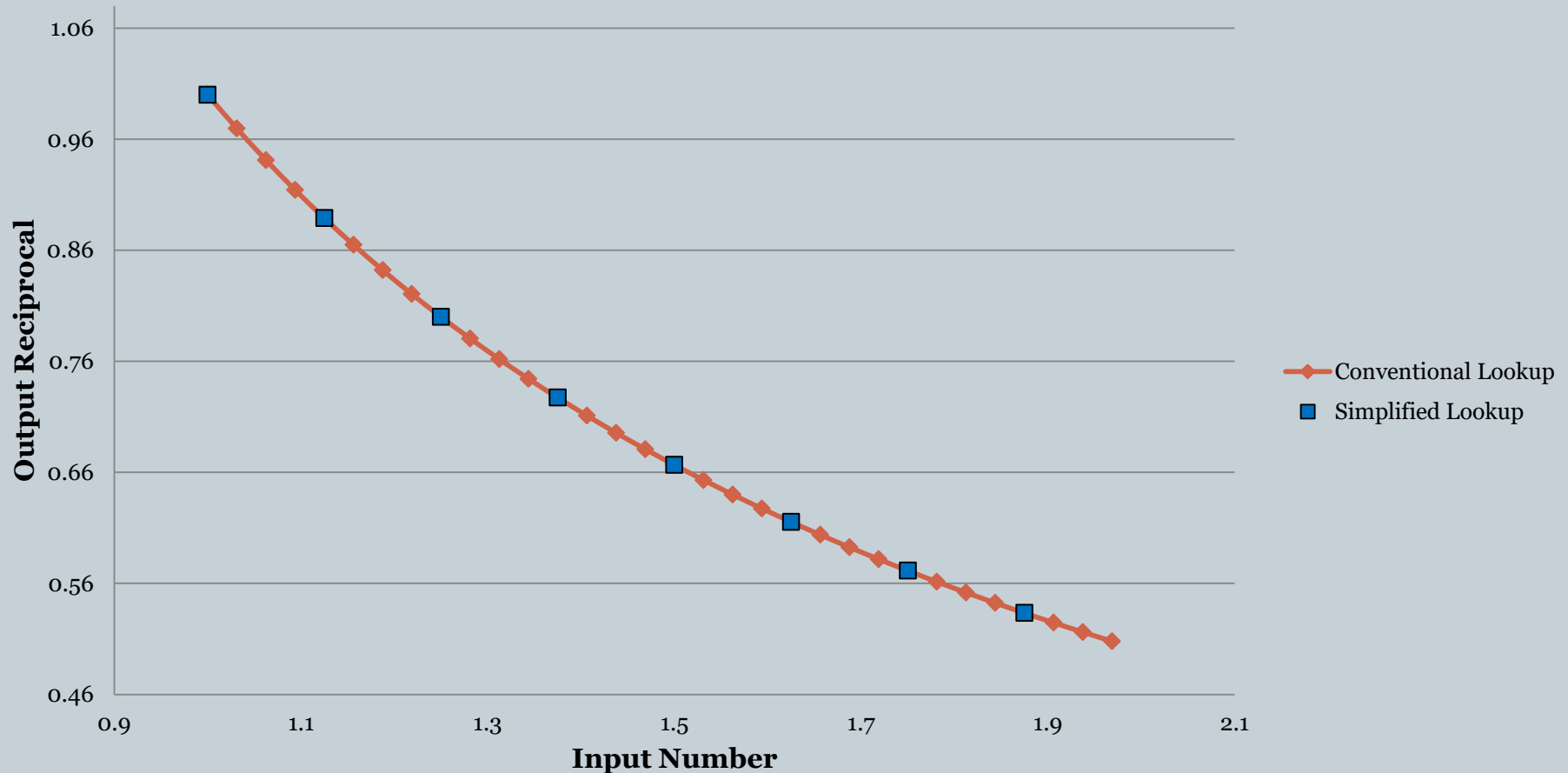


32 Values in my Lookup Table

Dual Table Explanation 2



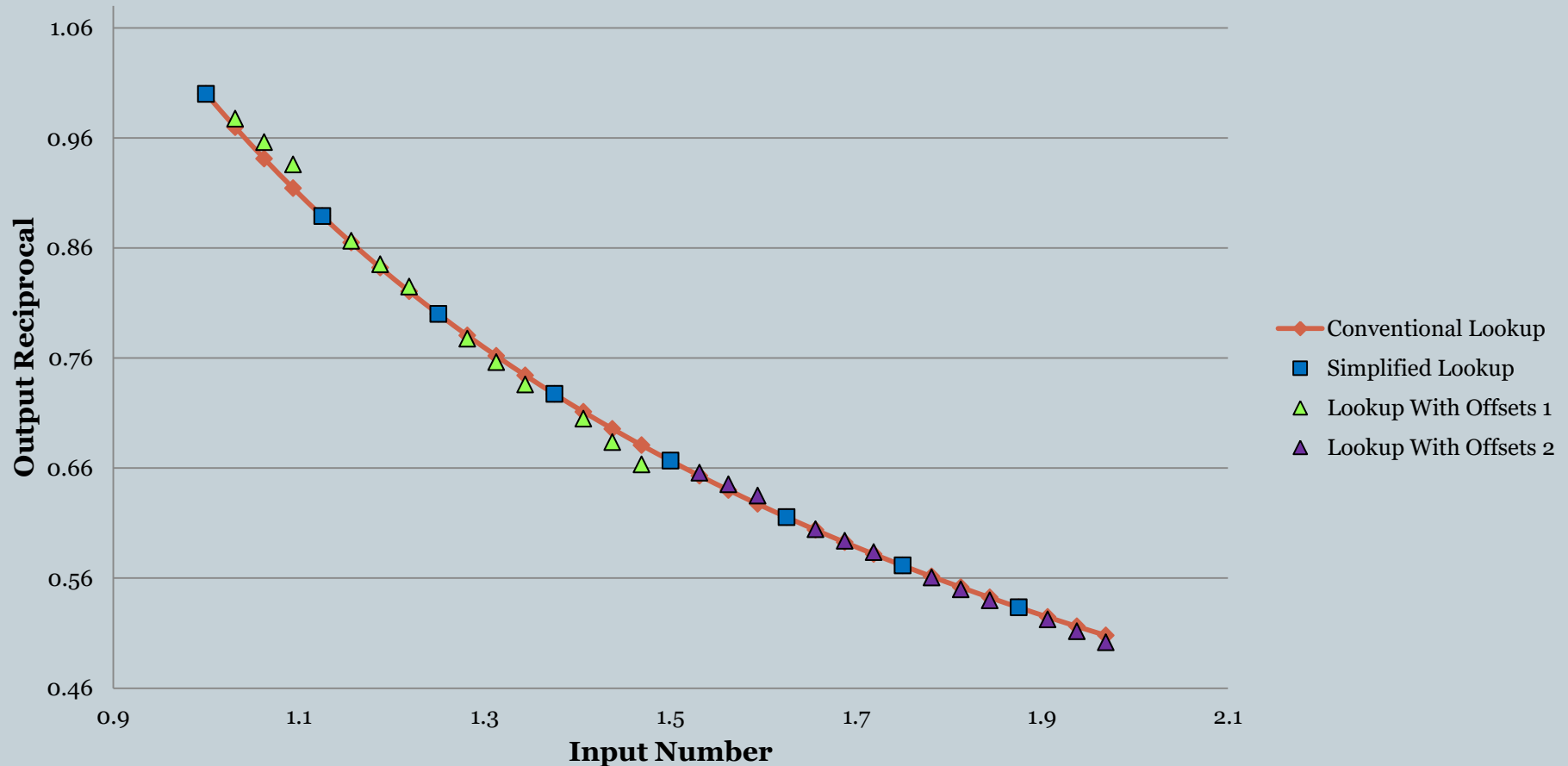
Reciprocal Lookup



Dual Table Explanation 3



Reciprocal Lookup



8 + 3 + 3 = 14 Values in my Table (much less)

Example Dual Reciprocal Table



Input Index Bits	Positive Part
1.00 00 xx	0.1 1111110 1
1.00 01 xx	0.1 1101111 1
1.00 10 xx	0.1 1100001 1
1.00 11 xx	0.1 1011000 1
1.01 00 xx	0.1 1001010 1
1.01 01 xx	0.1 1000000 1
1.01 10 xx	0.1 0111000 1
1.01 11 xx	0.1 0110000 1
1.10 00 xx	0.1 0101001 1
1.10 01 xx	0.1 0100011 1
1.10 10 xx	0.1 0011100 1
1.10 11 xx	0.1 0010111 1
1.11 00 xx	0.1 0010001 1
1.11 01 xx	0.1 0001101 1
1.11 10 xx	0.1 0001000 1
1.11 11 xx	0.1 0000100 1

Input Index Bits	Negative Part
1.00 xx 00	0.0000 0000 0
1.00 xx 01	0.0000 0011 0
1.00 xx 10	0.0000 0111 0
1.00 xx 11	0.0000 1010 0
1.01 xx 00	0.0000 0000 0
1.01 xx 01	0.0000 0010 0
1.01 xx 10	0.0000 0101 0
1.01 xx 11	0.0000 0111 0
1.10 xx 00	0.0000 0000 0
1.10 xx 01	0.0000 0010 0
1.10 xx 10	0.0000 0011 0
1.10 xx 11	0.0000 0101 0
1.11 xx 00	0.0000 0000 0
1.11 xx 01	0.0000 0010 0
1.11 xx 10	0.0000 0011 0
1.11 xx 11	0.0000 0100 0

Optimal 6 bits in, 5 bits out Bipartite Reciprocal Table

Reciprocal Tables in AMD K7



- **Dual Table**

- Regular Table has 2^{10} Entries and is 16 Bits Wide
- Offset Table has 2^{10} Entries and is 7 Bits Wide
- Total Size: 69Kbits
- 3 Cycle Latency
- Minimum Accuracy: 15.94 bits

Performance



- Latencies:

	Single (23 Bits)	Double (52 Bits)	Extended(68)
Iterations	1 Iterations	2 Iterations	3 Iterations
Cycles (latency)	16 Cycles	20 Cycles	24 Cycles
Cycles (throughput)	13 Cycles	17 Cycles	21 Cycles

- This is good, but we can do a couple optimizations.

Optimization 1: Multiply Under Division



- Problem: The Division Controller consumes the multiplier during a divide.
 - This could mean large latency penalties for multiplies, even if they are independent of the division.
- However, only two multiplies in pipeline at a time.
- Solution: Division Control Unit will notify the Floating Point Scheduler when there are free cycles available for multiplication.
- This approach attains 80% of the performance of a completely non-blocking Divider.

Optimization 2: Early Completion



- Problem: If the Denominator is a power of two, we are doing a lot of extra work.
 - In this case, we just need to modify the exponent. (with some attention to the x87 rounding rules)
- Solution:
 - Add detection logic for power-of-2 Denominators
 - Allow division operation to complete early
 - Notify scheduler many cycles in advance when completion will occur
- Using this method, all divisions by a power of two may complete in 11 cycles.

Conclusions



- Goldschmidt's algorithm gives us a fast Divider with little increase in multiplier area ($\sim 10\%$).
- Static power/Area/Design Time can be saved by not including a standalone Divider.
- Observed that small/accurate reciprocal rom tables can be implemented using dual table compression.

Backup Slides



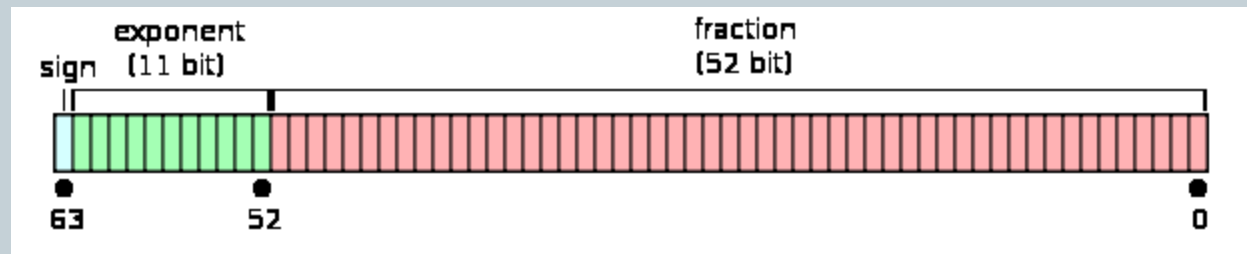
Floating Point Format



- Single:
[Bias = 127]



- Double:
[Bias = 1023]



- Value = $sign \times [1.Fraction] \times 2^{Exponent - Bias}$

Verification



- Multiplier was verified in 2 ways:
 1. Directed Random Vectors
 - C program generates inputs whose outputs lie near rounding boundries
 - 100,000+ Vectors tested at RTL and gate levels
 2. Formal Proof
 - Proof Based on a C language description of the Hardware
 - Each proof line coded into ACL2 Logic
 - Verified with ACL2 logic checker
 - 250 Definitions, 3000 lines

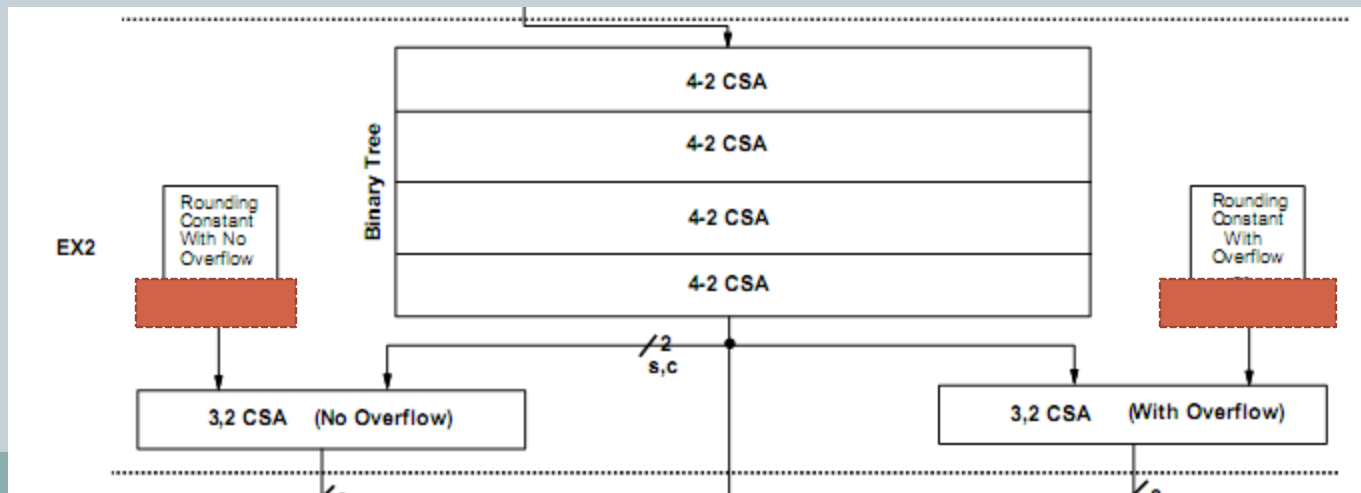
Determining the Remainder



- Remainder is found by multiplying the Denominator and the Quotient, then subtracting the Numerator:

$$\text{Remainder} = D \times Q - N$$

- (This is actually the inverse of the remainder, but oh well.)
- In the K7 multiplier, this can be produced in one multiplication.



Sources of Error in K7 Goldschmidt



- Error due to initial approximation
- Error due to use of one's complement, rather than true two's complement in the iterations
- Error due to using rounded results in the iteration multiplications

Rounding Details



- Rounding Modes

Guard Bit	Rem- ainder	RN	RP (+/-)	RM (+/-)	RZ
0	=0	trunc	trunc	trunc	trunc
0	-	trunc	trunc/dec	dec/trunc	dec
0	+	trunc	inc/trunc	trunc/inc	trunc
1	= 0	RNE	inc/trunc	trunc/inc	trunc
1	-	trunc	inc/trunc	trunc/inc	trunc
1	+	inc	inc/trunc	trunc/inc	trunc

Table 3. Action table for round function

- Rounding Constants

	SINGLE	DOUBLE	EXTENDED	INTERNAL
RN	(24'b0,1'b1,126'b0)	(53'b0,1'b1,97'b0)	(64'b0,1'b1,86'b0)	(68'b0,1'b1,82'b0)
RZ	151'b0	151'b0	151'b0	-
RM	(24'b0,(127(Sign)))	(53'b0,(98(Sign)))	(64'b0,(87(Sign)))	-
RP	(24'b0,(127(!Sign)))	(53'b0,(98(!Sign)))	(64'b0,(87(!Sign)))	-
LASTMUL	(25'b0,1'b1,125'b0)	(54'b0,1'b1,96'b0)	(65'b0,1'b1,85'b0)	(69'b0,1'b1,81'b0)
ITERMUL	(76'b0,1'b1,74'b0)			
BACKMUL	(!Dividend[67:0],(83(1'b1)))			