Lecture 5

Other Adder Issues

Mark Horowitz
Computer Systems Laboratory
Stanford University
horowitz@stanford.edu

Overview

- Reading
  - There are many papers on fast adder design. I have collected a few of them and have posted a web page that lists them under reading
  - You should read the Adder paper posted by Sam Naffziger. It should explain how to build Ling adder (useful for homework)
- Introduction:

  In the previous lecture we talked about different ways to organized adders, and showed a couple different ways to think about tree adders. This lecture will look at the Ling reformulation of the adder equations, and then look at some other issue that the designers of modern adders need to consider. If we have time, we will talk about the construction of floating point adder units.

Much of this material first appeared in a lecture from Brucek Khailany

Review: Basic Tree Adder Architecture

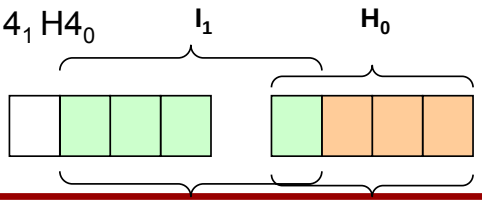
- Convert input operands into propagate and generate
  - $P_i = a_i + b_i$
  - $G_i = a_i * b_i$
- Carry calculation
  - Use P's & G's to compute carry to every bit (c_i)
  - Either build a combining tree for each bit
  - Or Fan-out results from tree(s) to each output
  - Remember that $C_{i+1} = G_{i:0} + P_{i:0}\ Cin$
    - Some times people try to combine Cin with the LSB of PG tree to remove the need for this gate (Make $G_0 = G_0 + P_0\ Cin$)
- Compute final sum
  - Si = ai⊕bi⊕ci

Ling Adder

- Ling adders is just a way to reformulate the Pg Gg equations
- The normal equation are:
  - $G4 = G_3 + P_3\ (G_2 + P_2\ (G_1 + P_1 G_0))$
  - $P4 = P_0 P_1 P_2 P_3$
- The equation for G4 requires a 4 stack since it is really
  - $G4 = G_3 + P_3\ G_2 + P_3\ P_2\ G_1 + P_3\ P_2\ P_1 G_0$
- Stacks are bad, so it is always nice to reduce them
  - Notice if P = A+B, then if G is true P is also true, so
    - $G4 = P_3\ G_3 + P_3\ G_2 + P_3\ P_2\ G_1 + P_3\ P_2\ P_1 G_0$
    - $G4 = P_3\ (G_3 + G_2 + P_2\ G_1 + P_2\ P_1 G_0)$
    - Define $H4 = (G_3 + G_2 + P_2\ G_1 + P_2\ P_1 G_0)$
  - H4 is easier to compute than G
- Can compute H4 directly from inputs w/o first computing P,G

Ling Adders cont'd

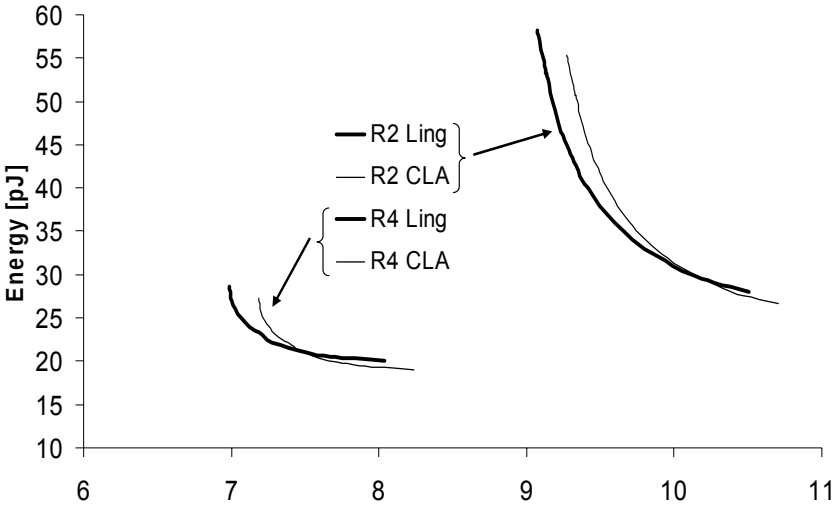
- But now the problem is how to use H?
  - Ultimately need G
- Want it to work in any tree type structure
  - Each group does not need P_{MSB}
  - But the next more significant group need it to use H
- Define pseudo P called I to replace Pg
  - $I4_0 = P_{-1} P_0 P_1 P_2$
  - $I4_1 = P_3 P_4 P_5 P_6$
- Then the combining formula looks just like it did before!
  - $H8_0 = H4_1 + I4_1 H4_0$



Who Cares?

- Normally this type of optimization would not matter much
  - Trick only works with P,G, and not Pg Gg
    - Gg does not imply that Pg is true
    - H does not imply I either
  - This means you get savings only at the first level of tree
  - But adders are carefully optimized, and every bit helps
- Ultimately need to add the missing P back to generate Carry
  - Put Cin into Ii_0 in the open slot for P_{-1}
  - When you generate C from H, I
    - $Cin_{i+1} = P_i (H_{i:0} + I_{i:0})$,
      - which is not much slower than normal Carry
    - In carry select adders, P_i can be added to the local chains

Ling vs. CLA

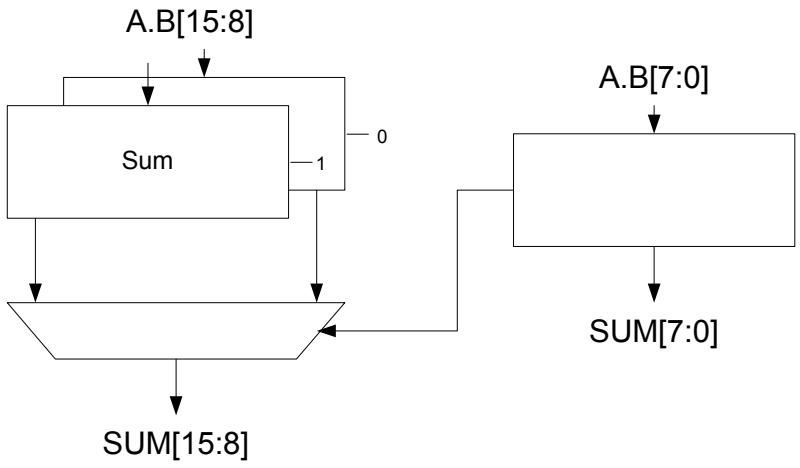


R. Zlatanovici, ESSCIRC'03

From Bora Nikolic

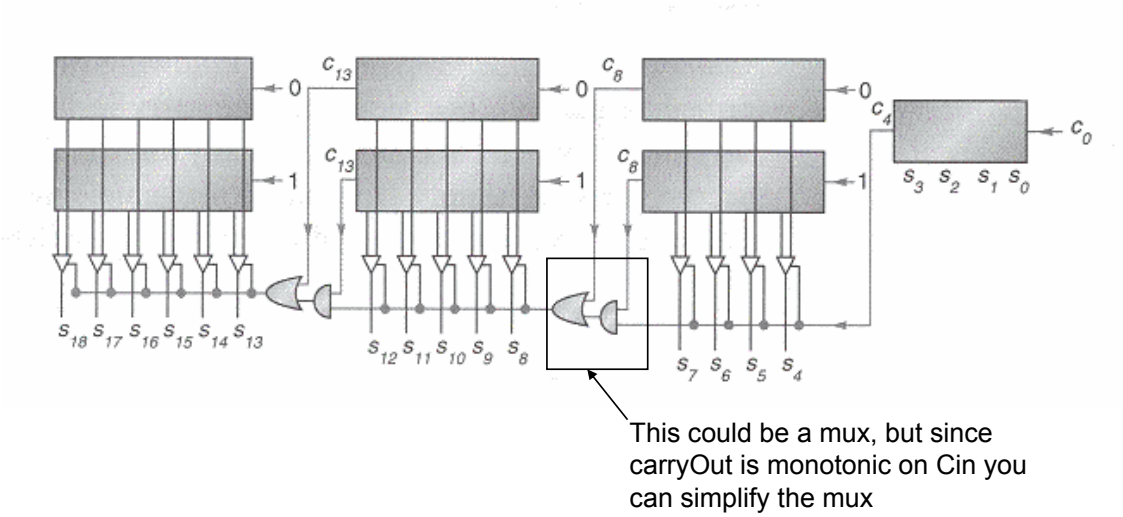
Review: Carry-Select Adders

- The Carry must fan-out to drive the width of the select group
  - This fanout must be accounted for



Simple Linear Carry-Select Adders

- Now ripple the carry through the select blocks
  - Critical path is linear with the number of blocks



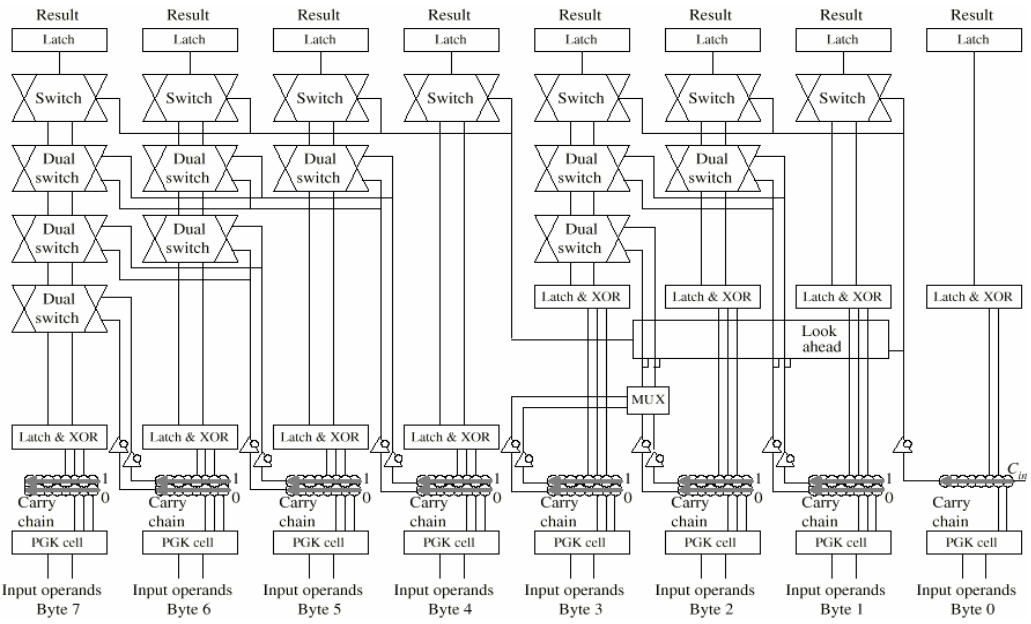
Select Trees: What To Do When C_{in} is Late

- For the upper ½ of an adder, Cin_{32} is usually late
  - If the use the architecture on the previous page, you then need to ripple the carry through 4 selects
  - Would be better to compute how Sum_i depends on Cin_{32} while we are computing Cin_{32}
- Assume you have 8 bit groups the algorithm is
  - Compute Sum1 (assume Cin to group is 1) and Sum0
  - Use Cout of previous group to create new Sum1* and Sum0*
    - $Sum0^*_{63\text{-}56}=(Cout0_{56}\ ?\ Sum1_{63\text{-}56}\ :\ Sum0_{63\text{-}56})$
    - $Sum1^*_{63\text{-}56}=(Cout1_{56}\ ?\ Sum1_{63\text{-}56}\ :\ Sum0_{63\text{-}56})$
- The result is
  - Sum1* is the correct sum if $carryOut_{47}$ is 1
  - Sum0* is the correct sum if $carryOut_{47}$ is 0
  - Effectively in one gate you have doubled the group size

Select Trees

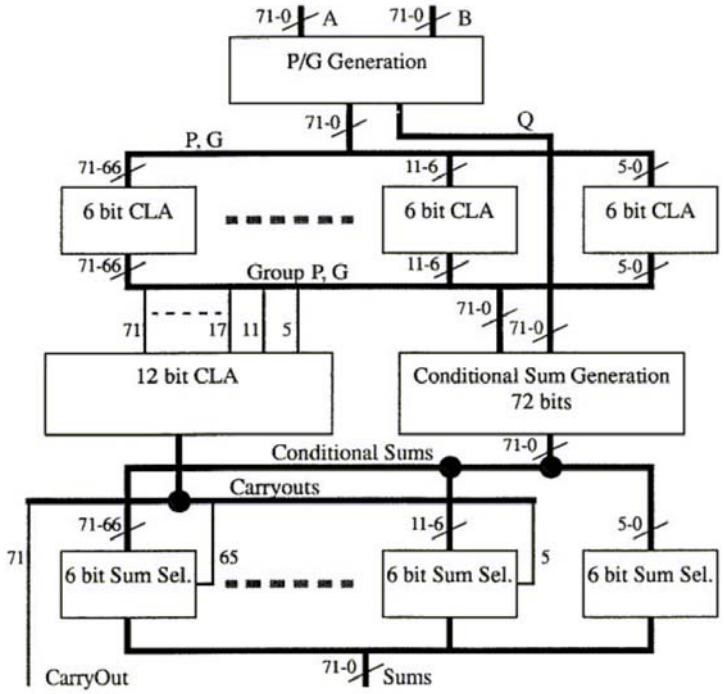
- Generally need to use them in fast adders
  - Otherwise can't generate the conditional sums fast enough
- But they are usually used only once
  - In radix 4 systems to move 4 bit groups to depend on C16
  - In binary tree systems to make final group size around 8
- But sometimes people go overboard
  - In the Alpha adder, C32 was slow enough
  - Rippled through 3 of these stages!
    - These were differential pass transistors stages
    - But still …

Alpha Adder

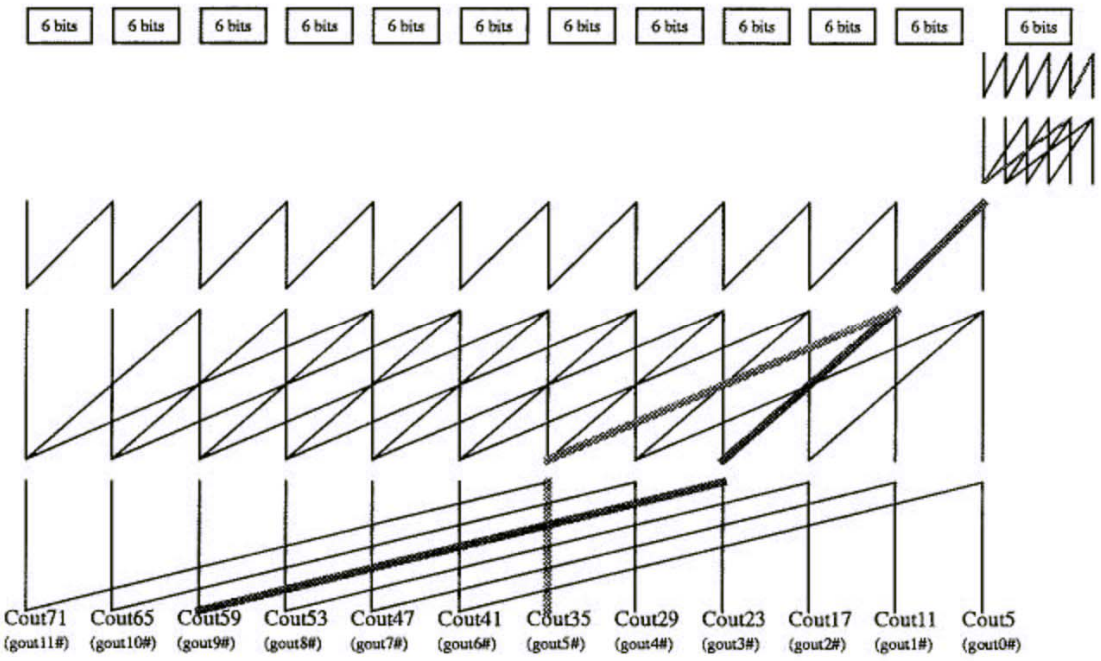


72-bit Pentium® II Adder

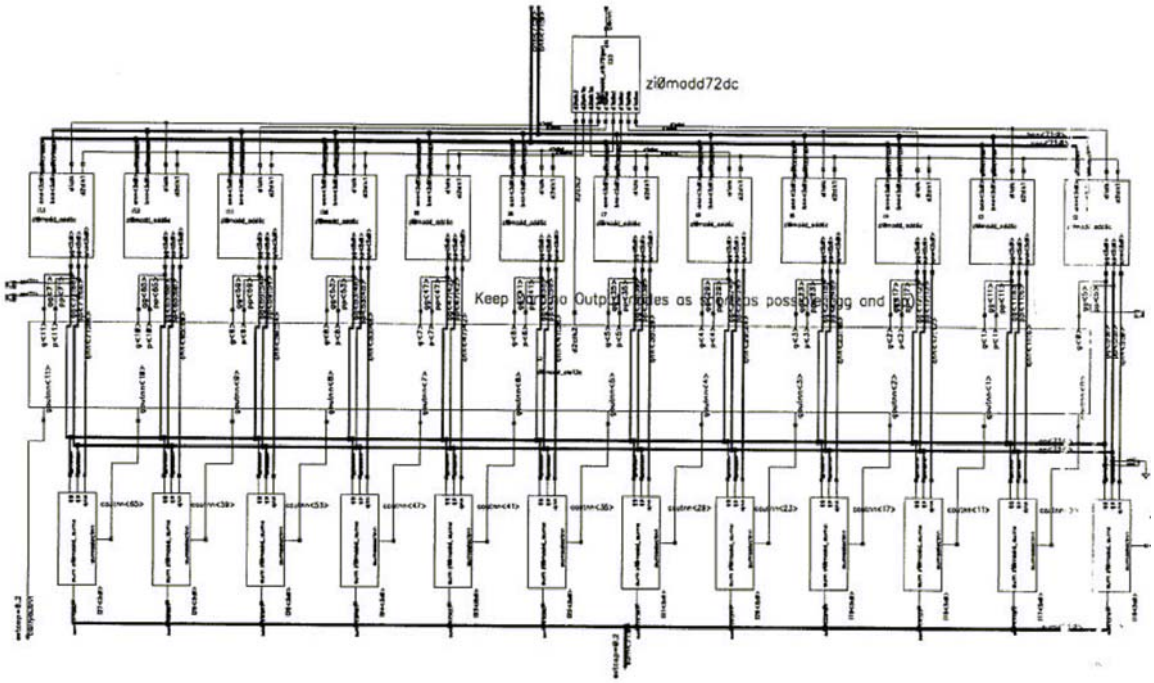
- 72-bit adder (Jason Stinson)
  - 0.35u process
  - Domino
  - Kogge-Stone
    - CLA+sumselect
  - Combines terms in both domino and CMOS stages



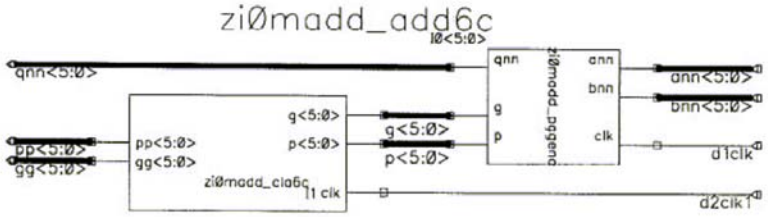
72-bit Pentium® II Adder

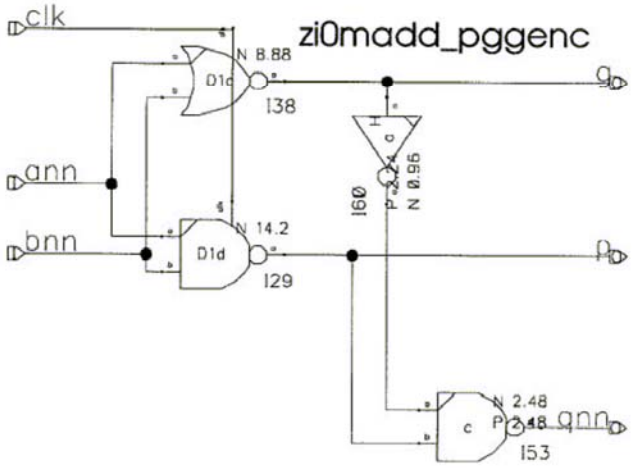


72-bit Pentium® II Adder

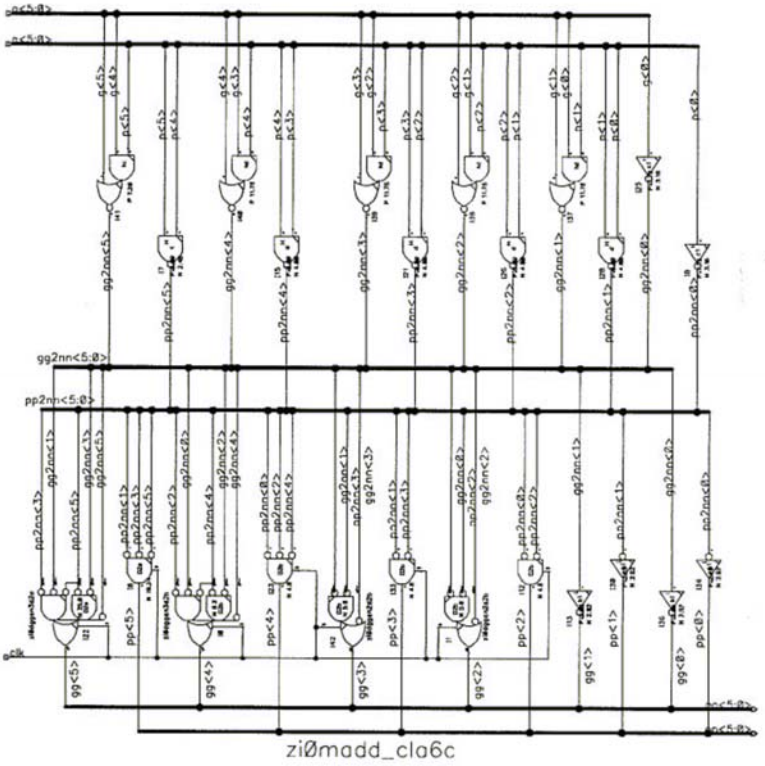


72-bit Pentium® II Adder

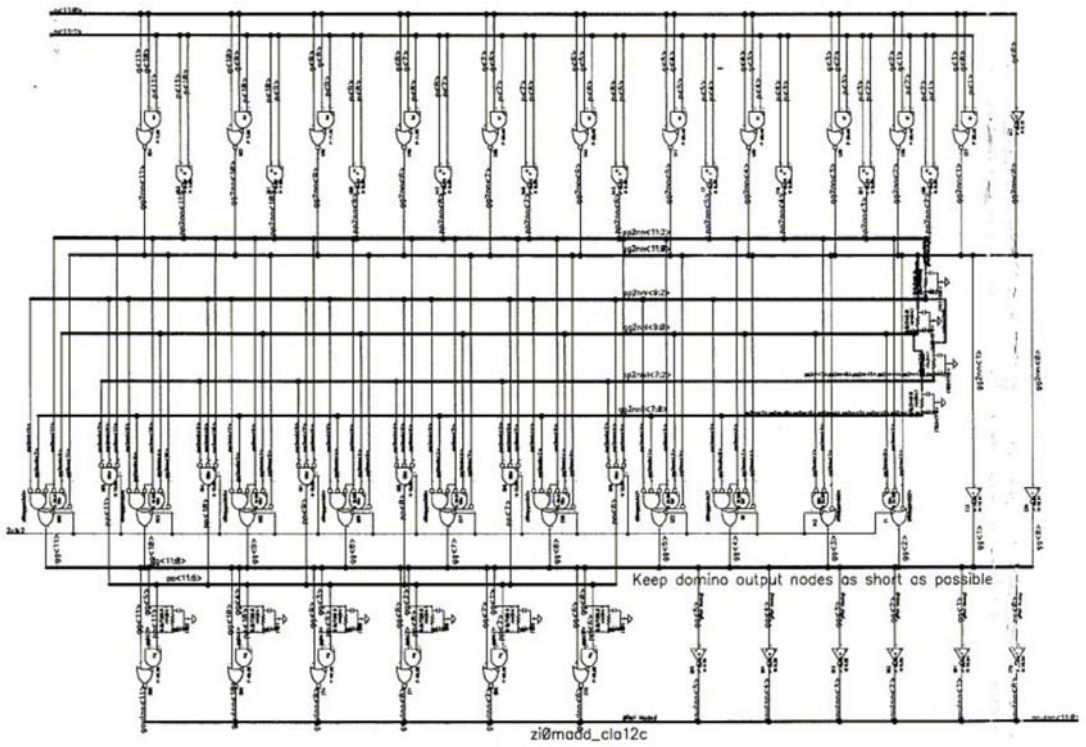




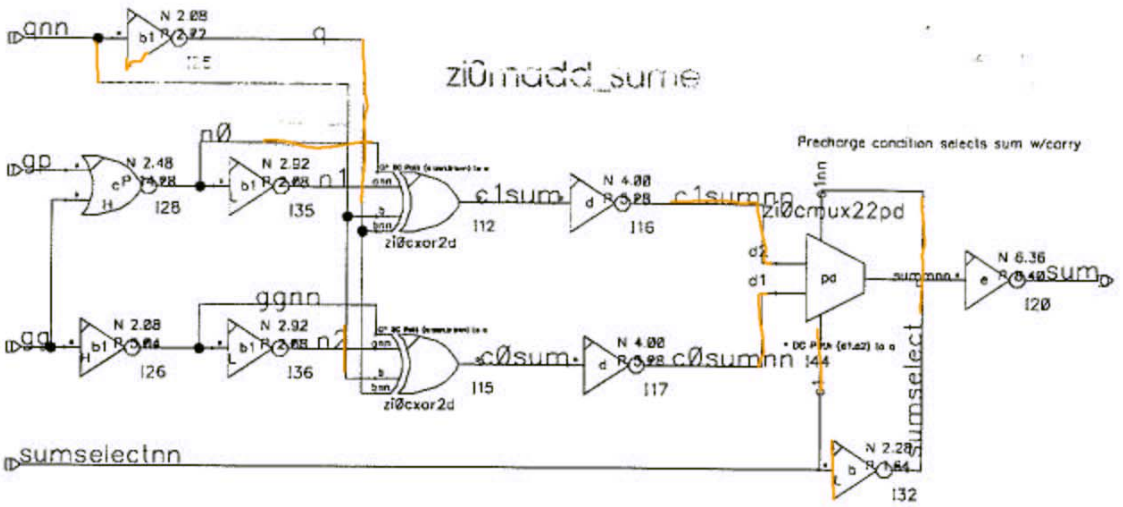
72-bit Pentium® II Adder



72-bit Pentium® II Adder



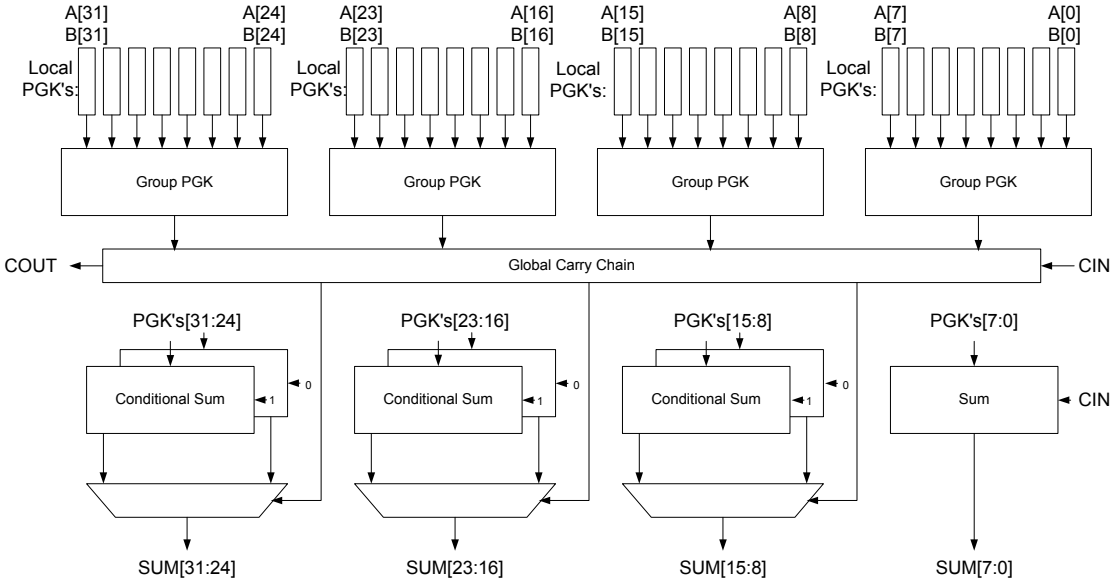
72-bit Pentium® II Adder



A Real-World Adder Design Example

- Part of Imagine
  - A high-performance media processor designed at Stanford
- 32-bit segmented integer adder
  - Two-level tree to compute global carries
  - Uses carry-select to compute final sums from global carries
- Static CMOS logic
  - Also pass gate logic
- Design constraints
  - Area
  - Design complexity (modularity)
  - Speed

Adder Architecture



This Adder Is A Trade-off

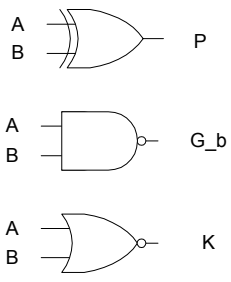
- It is balancing design/logic complexity and speed
  - It uses large groups which will ultimately limit performance
- It does use some tree structures
  - It does not ripple carries
  - But the group generation is a little slow
- Also uses large block sizes (8 bits)
  - Does not move the carry select input to lower significance
  - Need to worry about how outputs in block are generated

Segmented Adder: Details

- Local PGK's:
  - Convert input operands into Propagate (P), Generate (G), Kill(K)
- Group PGK's:
  - Determine P,G,K for groups of 8 bits
- Global carry chain:
  - Compute cin[8], cin[16], cin[24], cout(cin[32]) from group PGK's and cin
- Conditional sums:
  - Compute 8-bit sum for cin=0 and 8-bit sum for cin=1 as soon as PGK's are known
- Final Mux:
  - Use cin's from global carry chain to select conditional sums

Local PGK Logic

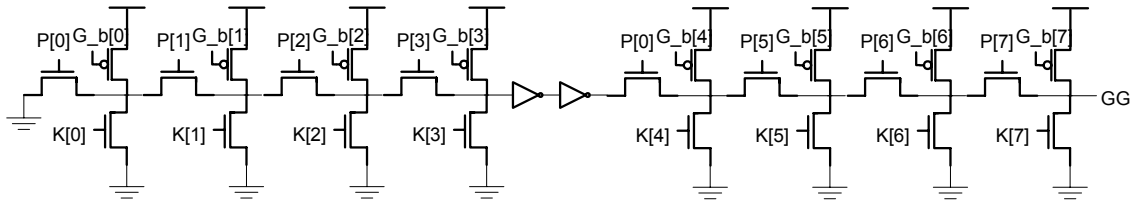
- Pre-computation necessary to do fast carry computation
  - P = a⊕b
  - G = ab
  - K = ~(a+b)



- Size gates to fan-out to four carry chains
- Note: To do A-B, use ~B here

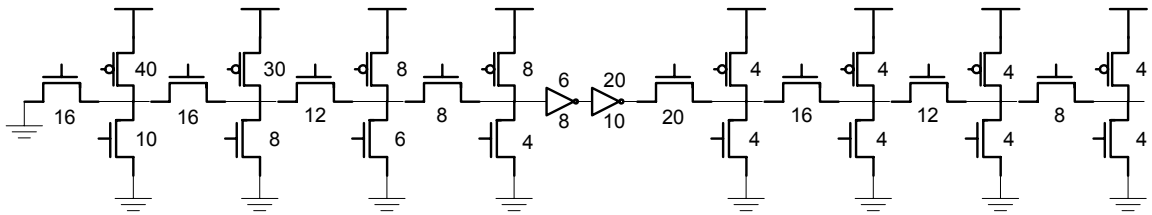
Compute Group PGK's: Use Manchester Carry Chains

- Usually dynamic, but still works with static logic
- Group PGK's:
  - GP = P[7].P[6].P[5].P[4].P[3].P[2].P[1].P[0]
  - GG = G[7] + G[6].P[7] + G[5].P[6].P[7]+…
  - GK = ~(GG+GP)
- Use Carry chains
- Example for GG (group generate):



Carry Chain Sizing

- Minimize size of transistors not on critical path
- Taper sizes along carry chain
  - Reduces diffusion capacitance



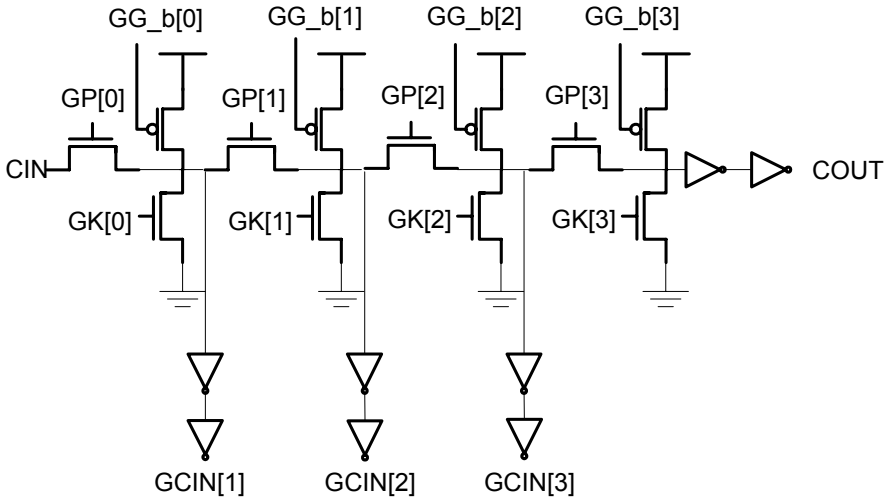
- What does tapering decrease?
  - It **increases** the LE of the critical input
- Why is the ratio of the first inverter funny

Static Carry Chains

- Sizing is to reduce the parasitic delay
  - This delay dominates in large fanin structures, since it grows proportional to n^2
  - Using geometric sizing (reducing each transistor along the chain by α) makes the parasitic delay linear
  - But still does not make them fast
- Even though this chain is static logic
  - Drive the carry chain both up (K) and down (G)
  - Output is degraded, since it uses nMOS only pass devices
    - Using CMOS transmission gates is usually slower because of the added parasitic capacitors

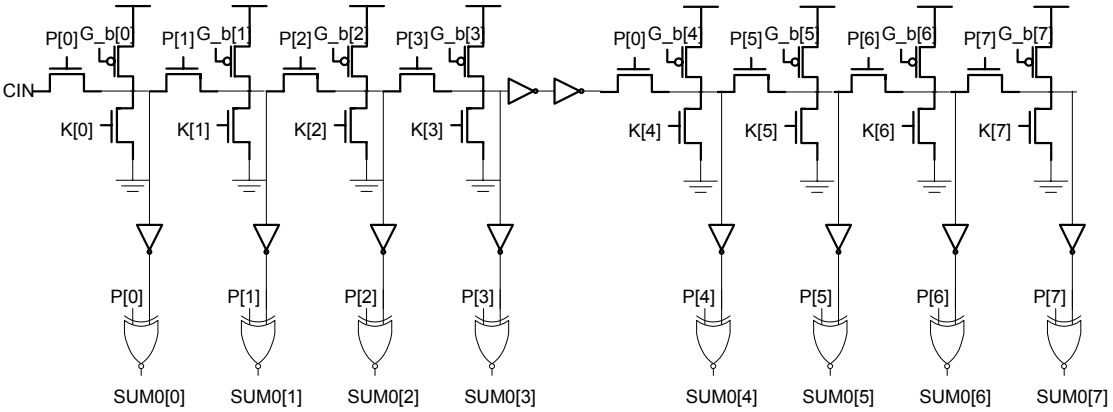
Global Carry Chain

- Must fan out GCIN[3:1] to 8 muxes
  - Added load capacitance slows down the chain



Conditional Sums

- Use same carry chain
- Do two of these (one for cin=0, one for cin=1):



- Mux SUM0[7:0] and SUM1[7:0] with output of global carry chain

Arithmetic Operations For Media Processing

- Used in Media processing
  - DSP's, multimedia extensions to instruction set architectures (MMX, VIS)
- Consider three variations of conventional arithmetic:
  - Segmented Arithmetic
    - Break carry chain
  - Arithmetic operations similar to add/subtract
    - Example: 4 parallel 8-bit unsigned absolute differences
  - Saturation
    - Don't wraparound on overflow

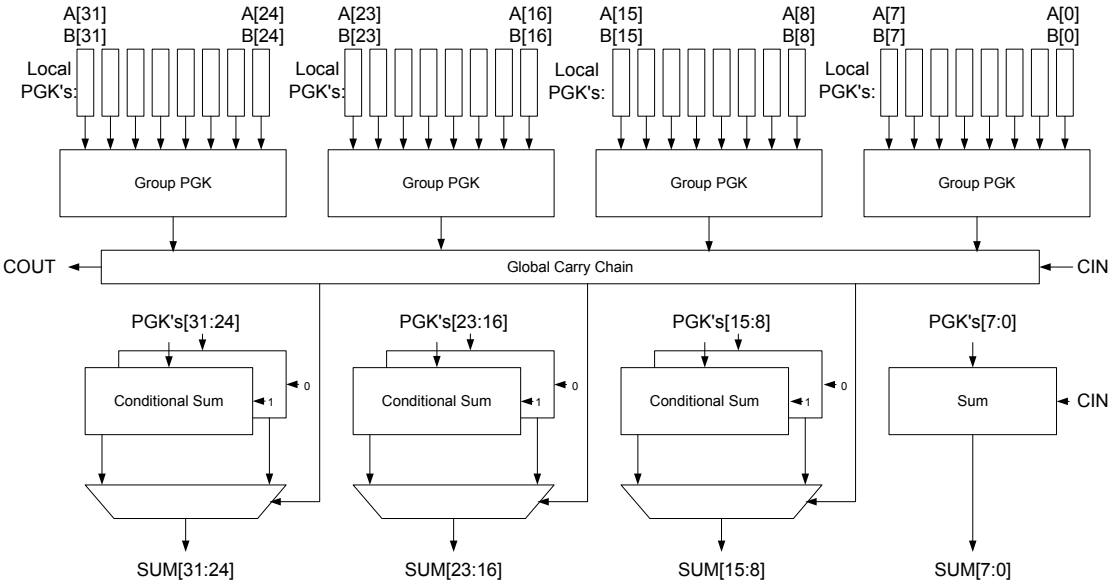
Segmented add operation

- Support 32-bit, dual half-word, or quad byte ops
- Example: 4 parallel byte additions
- Treat each byte as a separate 2's complement number
- Don't propagate the carries across byte boundaries

	1 4	0 2	F F	F F
+	F E	0 2	0 1	0 2
	1 2	0 4	0 0	0 1

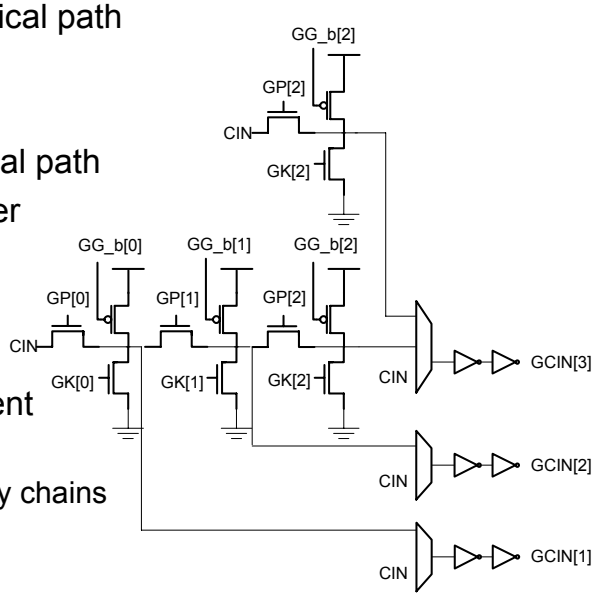
Modification For Segmentation

- Only modify carry propagation in global carry chain



Global Carry w/ Segment Support

- This method adds mux to critical path
- We can improve on this
  - Possible to move off critical path
  - By moving to start of adder
- If the op type is known early
  - For cells at start of segment
    - Change P, G definition
    - Change Cin to local carry chains



Absolute Difference

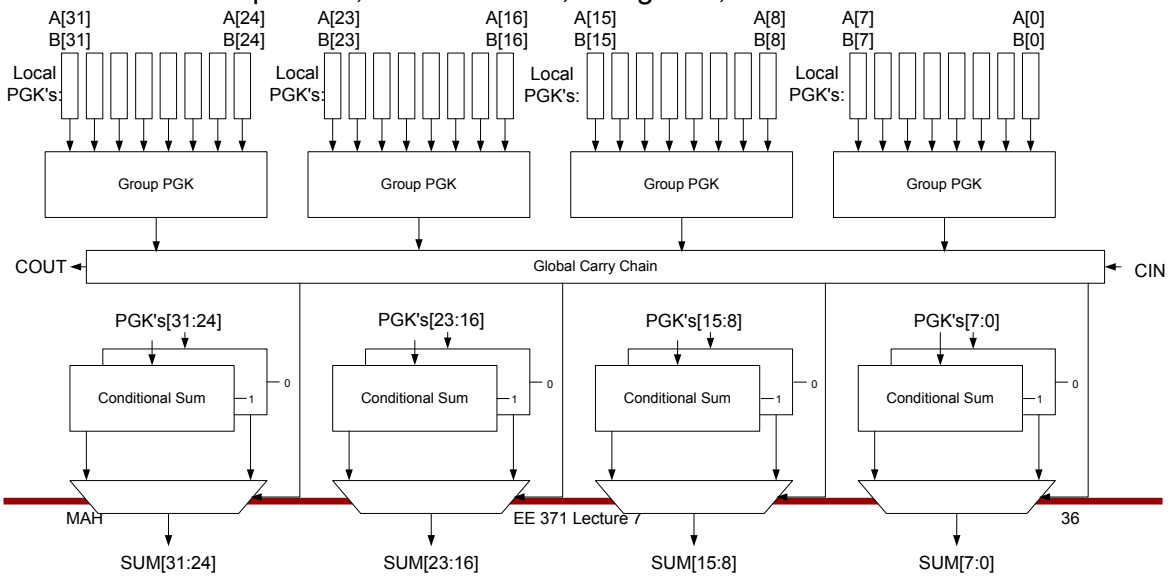
- Example: 4 parallel byte absolute differences
  - Important in MPEG encoding algorithm
- Algorithm:
  - Take two unsigned 8-bit numbers (between 0 and 255)
  - Compute |a-b|
  - Result is unsigned (between 0 and 255)

Absolute Difference

- How do we compute |a-b|?
  - We need to compute a-b and b-a and take the positive one
  - Remember that in 2's complement, -x = ~x + 1
  - The carry-select adder will compute a+~b+1 and a+~b
    a+~b+1 = a-b
    a+~b = a - b - 1
  - Note that
    ~(b - a) + 1 = a - b
    so
    ~(b - a) = a - b - 1
    or
    b - a = ~(a - b - 1)
- So, to compute |a-b|, just choose between SUM1 or ~SUM0 depending on the sign bit

Sum of Absolute Differences

  - Must do conditional sum for lower 8 bits
  - Must further modify global carry chain to look at sign bits
    - If positive, choose SUM1; If negative, choose ~SUM0



Saturation

- Often in media and signal processors, saturating arithmetic is supported:
  - Don't wraparound on overflow
  - Result should be largest (or smallest) value possible

- Examples:
  - 32-bit saturating integer add:
    - IADDS32(0x7FFFFFFF,0X00000001)=0x7FFFFFFF
  - 8-bit saturating unsigned subtract:
    - USUBS8(0x02FE02FE,0x03FE01FF)=0x00000100

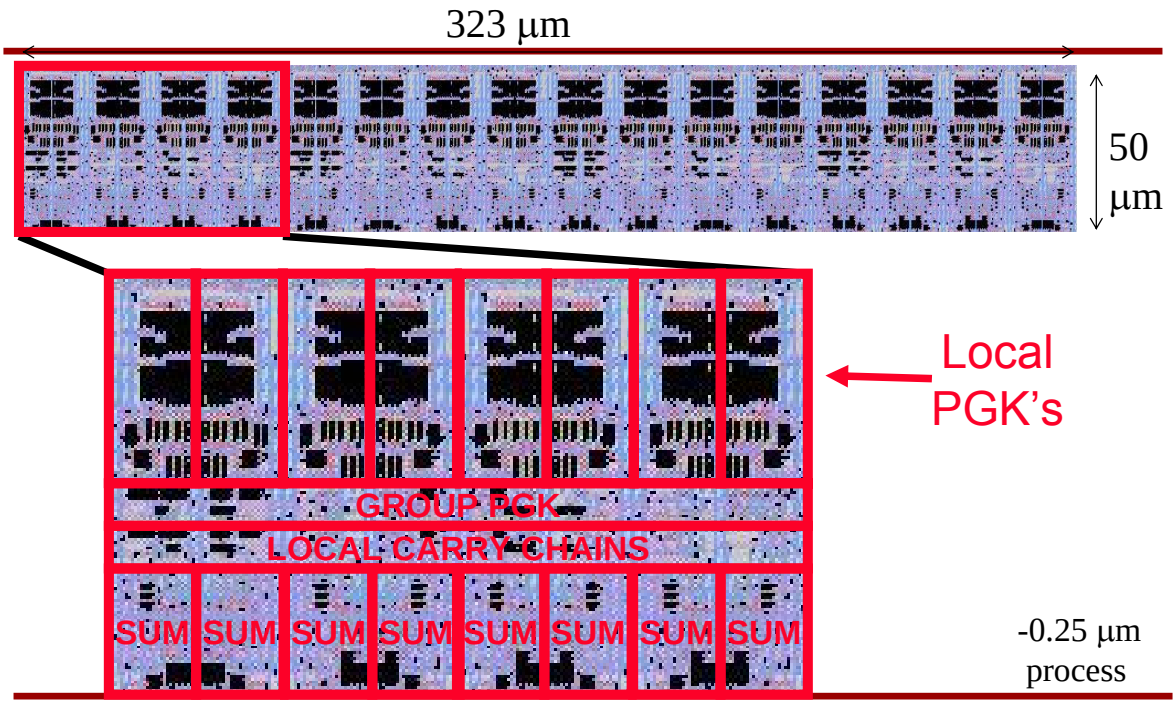
Hardware Support For Saturation

- Overflow detection
  - Example: signed addition
    - Can look at sign bits of inputs and outputs
    - Or can compute using ovf = cinmsb⊕coutmsb
- Overflow propagation
  - Similar to segmented, global carry chain, except for overflows
- Output muxing
  - Need a many-to-one mux for each byte to choose between: 0xff,0x00,0x7f,0x80 and the unsaturated value
- Methods for speeding up saturation
  - Could probably do "carry-select saturation detection"

Simulated Performance

- Two implementations:
  - Custom circuits (using circuits from these slides)
    - 15.1 FO4 delays through integer adder
    - 9.3 FO4 delays through overflow detection and saturation
    - ~3000λ x 500λ for adder only (excluding ovf det and saturation)
  - Standard cell implementation
    - ~23.5 FO4 delays through integer adder
    - ~10 FO4 delays through overflow detection and saturation
    - ~8000λ x 800λ for adder only (excluding ovf det and saturation)
  - Significant room for speed improvement through any of the following techniques:
    - Domino circuits
    - Faster carry-chain structures
      - e.g. carry-select on upper half of carry chains within each group pgk

Adder layout

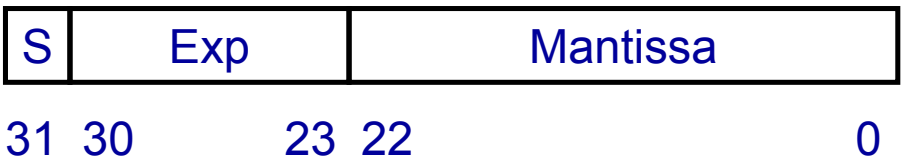


Adder Results From Previous Years

- 8 bit Manchester carry chains are slow, no matter how you size them. If you are going to use 8 bit groups, you probably need look-ahead in that group
- Using dynamic gates is much faster than static gates but
  - Need to worry a lot more about clock skew and noise margin issues. You also need to think about power
- It is possible to move segment overhead off the critical path
- Saturation is a pain since you need to know overflow condition before you can select the correct sum
  - But you can calculate overflow early if you spend hardware

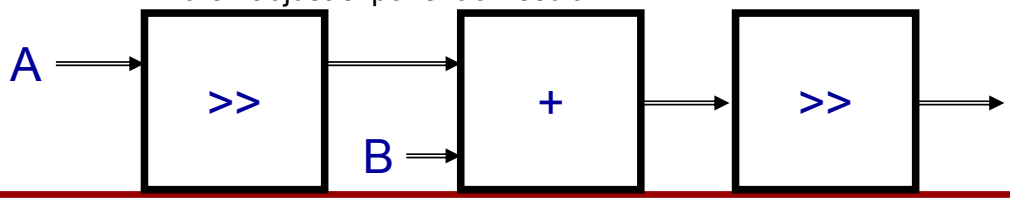
Floating-Point Number Representation

- IEEE Format Single-Precision Floating-Point numbers:
  - Contain Sign bit, exponent, and mantissa
  - Number represents (-1)S*(1.mantissa)*(2)Exp-127
  - Numbers are in sign magnitude form



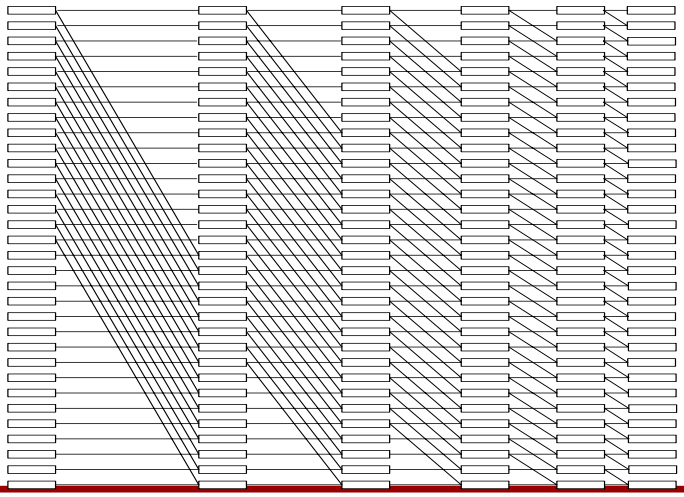
Floating-Point Addition

- Basic Algorithm:
  - Compare exponents of A & B
  - Perform alignment shift:
    - Whichever has smaller exponent, shift its mantissa right by |exp(A)-exp(B)|
  - Add unshifted mantissa with shifted mantissa
  - Round result
  - Perform normalize shift:
    - Shift output of add so that leading 1 lands in the right place, then adjust exponent of result



Alignment Shifter

- Use difference in exponents as shift amount
- Simplest shifter is 5 stages of 2:1 muxes
- Control wires for each stage come directly from exp diff:



Shifter Design

- Can improve on this:
  - Stage 1: 4:1 mux; shift by 0, 8, 16, or 24
  - Stage 2: 6:1 mux; shift by 0, 1, 2, 3, 4, 5, or 6
  - Must do pre-computation on shift amount to convert to 1-hot mux selects, but need buffer chain for fan-out anyway
- Tradeoffs between number of stages and logical effort of each stage are possible
- But shifters are not fast
  - Especially if the shift amount is not know in advance
  - For floating point shift amount depends on the operands

Aside: What About Segmented Shifts?

- Example: support 32-bit arithmetic shifts or 16-bit arithmetic shifts
  - Arithmetic shifts mean sign extend msb on right shifts
- Increases the number of mux inputs and/or stages of the shifter (at least for bits near the msb):
  - Stage 1: 5:1 mux:
    - 32-bit shift by 0, 8, 16, 24 or 16-bit shift by 8
  - Stage 2: 7:1 mux
    - 32-bit shift by 0, 2, 4, 6 or 16-bit shift by 2, 4, 6
  - Stage 3: 3:1 mux
    - 32-bit shift by 0, 1 or 16-bit shift by 1

Fast Floating Point Trick

- In general floating point requires two variable length shifts
  - To normalize the inputs mantissa's to equal significance
  - To renormalize the output, so mantissa's MSB is '1'
- The critical observation is that no operation requires two both
  - If the numbers are the same significance
    - Don't need a normalization shift greater than 1 bit
    - But you can get large cancellation, and need renormalization
  - If the numbers are difference significance
    - Need to normalize them
    - But they can't cancel, max renormalization is a shift of +/- 1
- But don't use this trick if throughput more important than latency

Floating Point Rounding

- In integer world, there is a right answer
  - You have all the bits to start with
  - Overflow is an error
- With floating point numbers, there is not one right answer
  - You loose bits all the time
    - Example is when you right shift for normalization
  - Need to decide what answer to give
    - IEEE standard gives many options
    - Most common is rounded
  - Rounding seems easy, but it is quite complex
    - Don't know the significance of result until you have it
    - But you want to add the ½ for rounding during the add
  - See Appendix A of Hennessy & Patterson for details

Normalize Shifter

- Shift mantissa so that leading one lands in the right place
- Adjust exponent if necessary
- If the input operands have very different exponents, the leading one will be in one of two places
- If the input operands have the same exponents or exponents that only differ by 1, the leading one could be anywhere
  - e.g. 1.00000 + -0.111101 = 0.000011

Floating-Point Adder Summary

- Basic operation: Shift, Add, Shift
- Carry-select adder important for rounding
  - compute two correctly rounded adds in parallel
  - choose the correct one based on overflow detection
- Can reuse same hardware for floating-point, integer, and segmented arithmetic operations
- Built 4-cycle pipelined floating-point adder
  - runs at 30 FO4 delays per cycle in standard cell implementation (5 FO4 from clocking overhead)
  - ~10,000λ x 3300λ