

# Floating Point Circuits

- **Topics**
  - **Addition and Subtraction**
    - » **Go for the hard one first**
  - **Multiply**
  - **Fused Multiply Add – FMA/MAF**
  - **Divide**
  - **Sqrt**



# Addition Algorithm

- **Basic algorithm for add**
  - **subtract exponents to see which one is bigger  $d = E_x - E_y$**
  - **swap values so biggest exponent addend is in a fixed register**
  - **alignment step**
    - » **shift smallest significand  $d$  positions to the right**
    - » **copy largest exponent into exponent field of the smallest**
  - **add or subtract significands**
    - » **add if signs equal – subtract if they aren't**
    - » **(Opposite for FP subtract (subtract if signs equal, add it not))**
  - **normalize result**
    - » **details next slide**
  - **round according to the specified mode**
  - **generate exceptions if they occur**



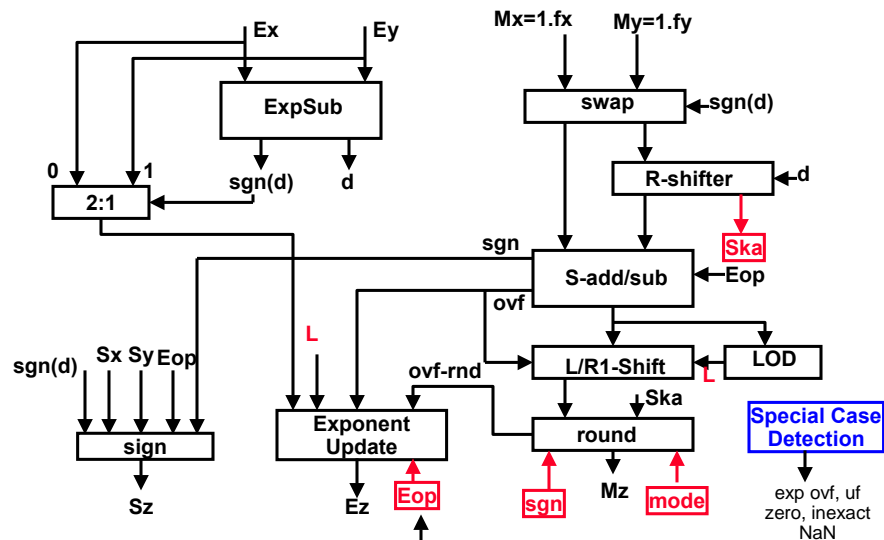
## Normalization Cases

- **Result already normalized**
  - no action needed
- **On an add**
  - you may have 2 leading bits before the “.”
  - hence significand shift right one & increment exponent
- **On a subtract**
  - the significand may have n leading zero’s
  - hence shift significand left by n and decrement exponent by n
  - note: common circuit is a LOD ::= leading 0 detector

$$\text{Value} = (-1)^S \times 1.F \times 2^{E-127}$$



## Basic Addition Circuit



## Devil is in the Details

- For now let's assume we're dealing with normals
- ExpSub
  - 2 8-bit unsigned numbers
    - » subtract can't generate an overflow
  - 2 choices
    - » unsigned subtract
      - borrow out becomes the  $\text{sgn}(d)$
    - » turn into 2's complement and add them
      - requires 9 bits → suboptimal choice
- Eop is simple
  - XOR of  $S_x$  and  $S_y$
- 2 mux stages
  - both are 2:1
    - » SWAP is 24 bits wide, and the 2:1 is 8 bits for the exponent
      - why 24?
      - in order to allow both normals and denormals



## R-Shift Alignment Step

- Again 2 options
  - simple shift mantissa and decrement  $d$ 
    - » problem – for large  $d$  this is too slow
  - barrel shift
    - » how many stages?
    - » note that  $d$  is an 8 bit unsigned number

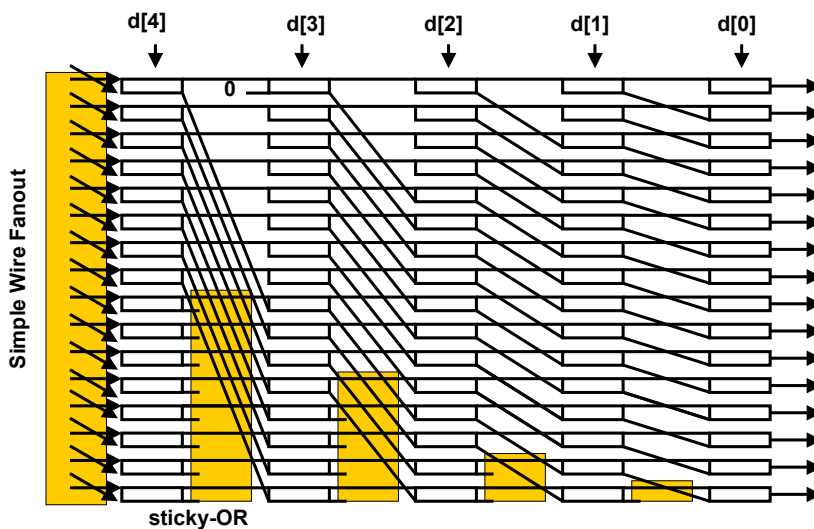


## R-Shift Alignment Step

- **Again 2 options**
  - simple shift mantissa and decrement  $d$ 
    - » problem – for large  $d$  this is too slow
  - barrel shift
    - » how many stages?
    - » note that  $d$  is an 8 bit unsigned number
- **Answer**
  - 5 stages + a conditioner + a sticky circuit
  - take advantage of the fact that 24 is the biggest shift that makes sense
  - hence OR the high order 3 bits of  $d$ 
    - » if 1: zero the fraction
      - sticky is an OR of the full 24 bit *fraction of the moment*
      - usually just a tree of NOR gates
    - » if 0: barrel shift based on the other 5 bits
      - each shift stage has a sticky NOR tree of the shift amount



## 5-stage Barrel Shifter (bottom half)



## Barrel Shifters Ain't Cheap

- Lots of 2:1 muxes and lots of wires
- Important trick
  - for any  $E_{op}$ 
    - » there is a max of one long shift
    - » and the other shift is at most 1
  - hence
    - » mux the barrel shifter where it's needed
- Note barrel shifter may get used twice
  - alignment when exponents differ significantly
  - on an effective subtract during normalization
    - » lots of leading zero's in the significand
  - so hefty structure gets amortized



## S-Add-Sub

- Add or subtract significands
  - what you do depends on the  $E_{op} = \text{XOR}(M_x, M_y)$
  - same as the integer world
    - » either build an adder subtractor
    - » or on an effective subtract – complement and add
- Note
  - we didn't do a magnitude compare on the significands
  - hence the result may be negative
    - » → sign of result must be kept
    - » influences the sign of the result NOT the result value
      - one minor advantage of floating point
      - no need to worry about calculating overflow in this step



## L0D

- **Detecting the number of leading order 0**
  - 24 places to look – need a 5 bit result
- **several methods**
  - 5 boolean functions of 24 variables
    - » it's not as bad as it looks
  - **priority encoder**
    - » if all higher order bits are 0 select a hardwired 5 bit code
    - » also not too bad but a bit slower
  - **table lookup**
    - » small table 24x5 bits
    - » the worst choice



## L/R1 Shifter

- **variable number of left shifts or 1 right shift**
  - **right shift 1 is easy**
    - » contributes to the sticky bit
  - **variable left shift**
    - » remember the guard bits
      - G + R are shifted
      - 0's injected from the right
      - sticky bit keeps its value
    - » if you implemented a barrel shifter for rounding
      - you probably want to re-use it rather than building 2 of them
    - » compensating for left vs. right
      - requires an additional mux at the front and back
      - to handle bit reversal chores



## Rounding

- **Add**
  - Add rnd to the 24 bit value based on the rounding mode
    - » unbiased:  $\text{rnd} = \text{G}(\text{L} + \text{R} + \text{S})$  or the add 1 to G and maybe zero L trick
    - » +inf:  $\text{rnd} = \text{sgn}'(\text{G} + \text{R} + \text{S})$
    - » -inf:  $\text{rnd} = \text{sgn}(\text{G} + \text{R} + \text{S})$
    - » 0 → truncate:  $\text{rnd} = 0$
  - simple boolean function of 7 variables
    - » 2 mode bits
    - » 3 guard bits
    - » sgn
    - » L
- **Shift**
  - if carry into high order bit of add
    - » shift result 1 bit to the right
    - » signal overflow to exponent update



## Exponent Update

- **Just a loadable saturating counter**
  - loaded with result of 2:1 exponent mux
- **w/ an associated subtracter**
  - L value during normalization is subtrahend
  - incremented if  $\text{ovf\_rnd}$  is signalled
  - confusion about  $\text{ovf}$  on a effective subtract???? Grr!!
- **Other tactics exist**
  - but these depend on a bunch of timing issues that we're ignoring at this point
- **Whew – at last something is really simple**



## Sign Calculation

- This one is a bit hairy
  - logic is simple – boolean function of 5 variables
    - » sign of the exponent subtract
    - » sign of the result
    - »  $S_x$ ,  $S_y$ , and  $O_p$ 
      - note this was the confusion in class (in the book as well)
      - $E_{op}$  can be figured out from  $S_x$  and  $S_y$  and  $O_p$
  - but getting it correct is hard
    - » getting the truth table right always makes me crazy
- Let
  - $E_{op} = 0 \rightarrow$  add
  - $S_x$  or  $S_y$  or  $S_s$  or  $sgn(d) = 0 \rightarrow$  positive (normal convention)
    - »  $sgn(d) = 0 \rightarrow E_x \geq E_y$
- Interactive phase begins



## Sign Function $sgn(d) = 0$

$sgn(d)??$	$S_x$	$S_y$	$O_p$	$S_s$	$S_z$
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	1	1

$sgn(d) = 0$   
 $\rightarrow E_x \geq E_y$   
 since possible = then  $S_s$  counts





## Sign Function $\text{sgn}(d) = 1$

$\text{sgn}(d) = 1$   
 $\rightarrow E_y < E_x$   
 no possible  
 =  
 then ignore  
 Ss

$\text{sgn}(d)$	Sx	Sy	Eop	Ss	Sz
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	0
1	1	1	1	1	0



## And the Answer Is

$$\text{Sign-of-Result} = \text{sgn} * \text{Sy}' * \text{op}' + \text{Sx} * \text{Sy}' * \text{op} + \text{sgn}' * \text{Sx} * \text{Ss} + \text{Sy}' * \text{Op}' * \text{Ss} + \text{Sgn} * \text{Sy}' * \text{op} + \text{Sy}' * \text{Op}' * \text{Ss}$$

Note: I'm pretty sure this is right  
 but send email to [ald@cs.utah.edu](mailto:ald@cs.utah.edu) if you  
 suspect an error – it's complicated and I  
 haven't simulated it yet



## Exceptions

- **Overflow**
  - **causes**
    - » exponent incremented during normalization or rounding overflow
  - **detect**
    - » when carry out of exponent update counter happens
      - note one of the operands could have been infinity
      - don't need to special case for an add
    - » OR when exponent is all 1's
  - **action**
    - » set result to  $\infty$ 
      - hence saturating counter
      - and carry out or all 1's  $\rightarrow$  0'ing Mz
      - sign takes care of itself
    - » set overflow flag



## Underflow

- **NOTE: AI's view and the book's differ**
- **Book:**
  - cause: if exponent decremented during normalization
  - result:  $E \leftarrow 0$ , fraction left un-normalized
- **My view:**
  - E goes to 0 or below for any reason

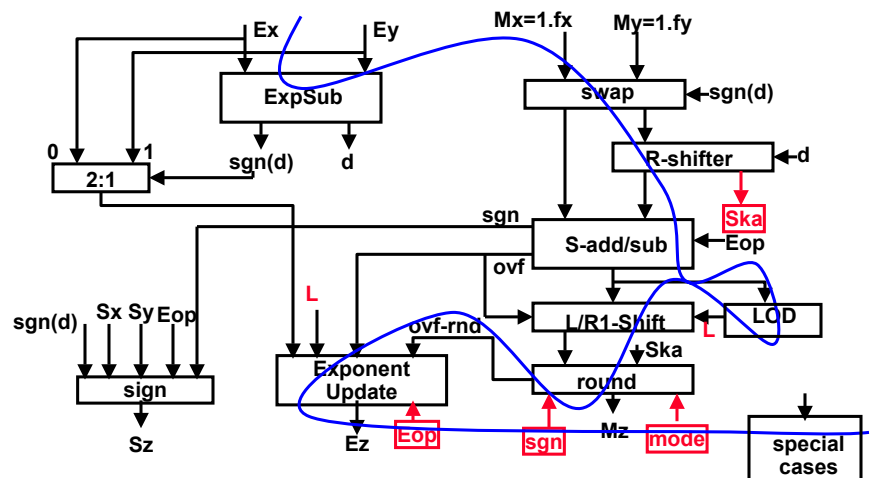


## Other Exceptions

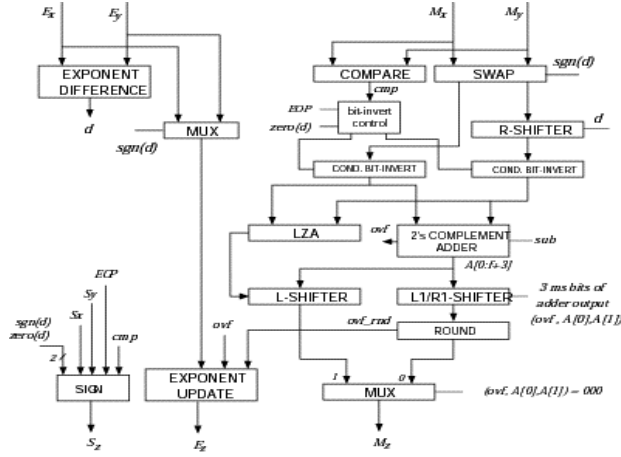
- Zero
  - cause
    - » significand (after rounding) goes to zero
  - action
    - » set E to 0, and set zero flag
- Inexact
  - set flag if prior to rounding  $G+R+S = 1$
- NaN
  - here's the weird one
  - must check X and Y operands
    - » if either is a NaN
    - » then set flag and force result to NaN

## Basic Implementation Analysis

- Worst case path analysis



## A Improved "Single Path" Implementation

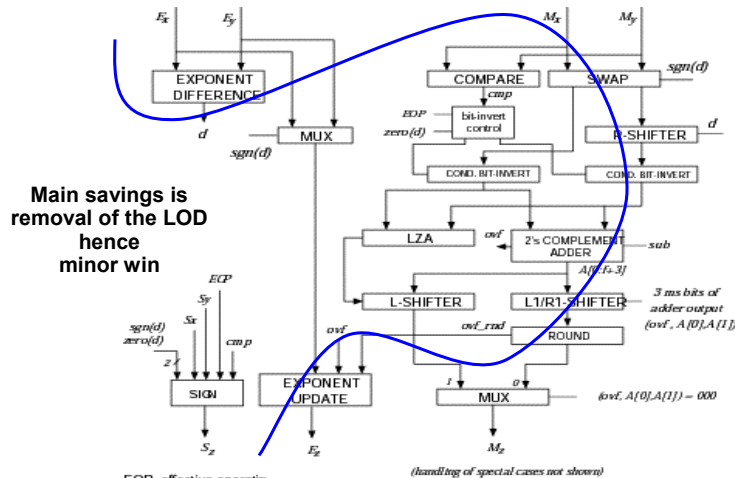


EOP: effective operatio  
R-SHIFTER: variable right shifte  
L-SHIFTER: variable left shifte  
L1/R1-SHIFTER: one position left/right shifte

figure 8.8 from text



## "Single Path" Worst Case



Main savings is  
removal of the LOD  
hence  
minor win

EOP: effective operatio  
R-SHIFTER: variable right shifte  
L-SHIFTER: variable left shifte  
L1/R1-SHIFTER: one position left/right shifte

figure 8.8 from text



## What Changed?

- **S-Add/Sub**
  - replaced by 2's complement adder
    - » on eff-sub complement subtrahend
      - bit invert and then put carry in to adder
    - » to avoid re-complementing the result
      - smallest operand is complemented → result positive
      - complicates the compare however
        - need to compare the exponents & significands
        - since exponents may be =
- **LZA – leading zero anticipation**
  - calculates the position of the leading 1
  - similar to the add in complexity but done in parallel



## More Changes

- **Round and Big (>3) left shift in parallel**
  - claim if big left shift occurs then  $G,R,S=0$  hence no rounding needed
    - » I claim this isn't quite true
      - you don't know how many bits were shifted right and there might be a 1 out there.
      - hence R-shift count would also be required to determine role of sticky bit



## Improving Further

- 2 paths
  - CLOSE – for subtraction and exponent difference of 0 or 1
  - FAR - for addition and subtraction when  $d > 1$
- However
  - path latencies are quite different
  - not substantially evil
    - » can always signal a ready bit
  - but this complicates the processor pipeline
    - » and makes forwarding super weird
  - can always fix with a non-laminar pipeline
    - » but it is non-laminar

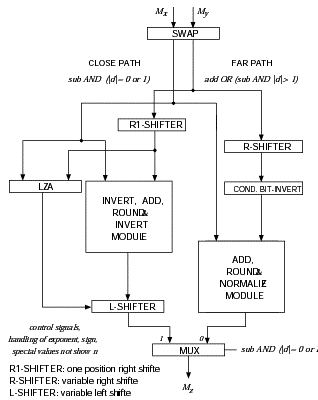
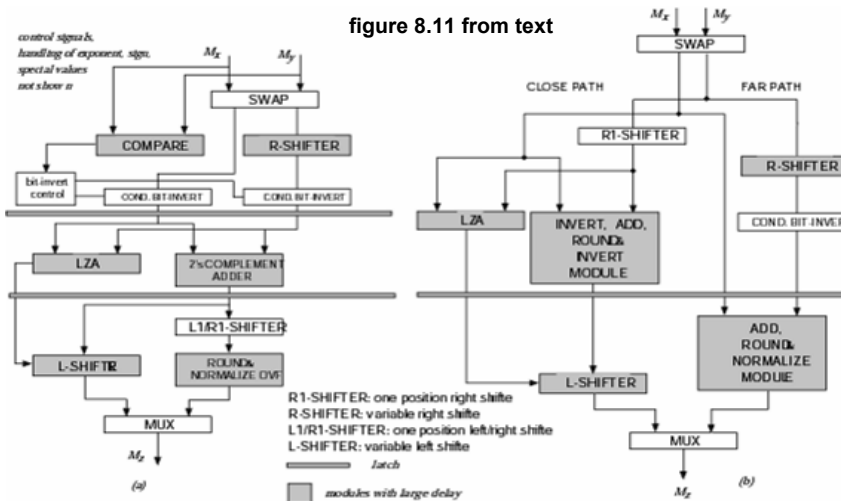


figure 8.10 from the text

## Pipelined Single and Double Path

figure 8.11 from text



## Comments on Text Pipeline

- Basically it depends where you are in the timing regime
  - for slow clock rates and a good process
    - » the previous pipeline model is fine
  - for high performance processors on a best process
    - » every non-trivial module will be pipelined
    - » Horowitz example
      - 4-cycle pipelined floating-point adder
      - runs at 30 FO4 delays per cycle in standard cell
      - implementation (5 FO4 from clocking overhead)
      - $\sim 10,000\lambda \times 3300\lambda$
  - however
    - » both area and frequency are hugely dependent on FO4 budget
    - » 15 FO4 designs exist with 20+ stages
      - these designs are very laminar
      - you have to be at 15 FO4



## Floating Point Multiplication

- Basic algorithm
  - multiply significands & add exponents
    - » exponent add
      - slightly tricky – why?
    - » multiply of  $m$  bits  $\rightarrow$   $2m$  bit result
      - only need to keep 2 bits from lower order half for rounding
        - G & Sticky
  - normalize result and update exponent
    - » exponent update needs to check for all 1's and overflow
  - round
  - checks for special values and set exception flags
    - » NaN in  $\rightarrow$  NaN out  $\rightarrow$  should be a qNaN
    - » Infinity
      - overflow on carry out  $\rightarrow \infty \rightarrow E = \text{all } 1\text{'s}, f = \text{all } 0\text{'s}$
      - exponent can still go to all 1's even with no overflow
      - hence a all 1's check circuit is required



## Exponent Addition

- **Biased representation**
  - $E = \text{actual value} + \text{bias}$ 
    - »  $E_x = V_x + B$
    - »  $E_x + E_y = V_x + V_y + 2B$
    - » → need to subtract the bias to get the proper representation
  - **0's and denormals**
    - » if  $E_x$  or  $E_y$  is 0 then must set carry in
    - » since actual  $V = 1$ -bias in this case
  - $E_z = E_x + E_y - B$
- **Mz overflow**
  - effectively need a 9 bit add/subtract
  - $M_x + M_y$  step can produce a carry out
    - » but on the bias subtract step the carry out bit may clear
    - » if not then the exponent must be set to all 1's
- **Sign of the result**
  - $S_z = \text{XOR}(S_x, S_y)$



## Normalization & Rounding

- **Normalization**
  - similar to what happened with addition except
    - » inputs in range 1:2 → result in range 1:4
    - » hence may need one right shift & increment exponent
      - right shift → update sticky
- **Rounding**
  - also similar to addition but with only 2 guard bits:  $G$  &  $S$ 
    - » let
      - $L$  = low order bit of mantissa (.....LGS)
      - $sgn$  is sign of the result
  - **unbiased**
    - »  $rnd = GS + GS'L = G(S+L)$
  - **toward 0**
    - » simple truncation:  $rnd = 0$
  - →  $+\infty$ 
    - »  $rnd = sgn'(G+T)$
  - →  $-\infty$ 
    - »  $rnd = sgn(G+T)$





## Basic Circuit

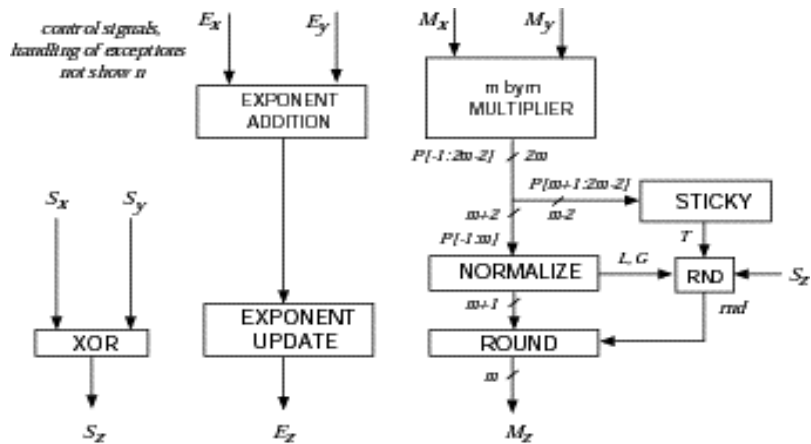


figure 8.12 from text

## Exceptions and Special Values

- **Exceptions (same as for addition)**
  - exponent overflow after normalization → set overflow flag
    - » and result is set to infinity
  - exponent = 0 → set underflow flag (zero or denormal)
  - zero flag set (2 options)
    - » check for 0 operand and other not infinity
      - OK since need to check for NaN's and infinity anyway
    - » check result
  - inexact set if  $G+T=1$
  - NaN set
    - » if one operand is 0 and the other is infinity
    - » or if one or both operands are NaN's
- **Denormals**
  - possible when one or both operands are denormals
    - » hence left shift during normalization and exponent subtract
  - also when exponent underflows the mantissa is shifted right
    - » creates denormal

## Denormal Conundrum

- **Whacky method**
  - normalization phase shifts left and decrements exponent
  - then if exponent underflows
    - » increment exponent and then right shift significand until exponent gets back to zero
  - can you say **SLOW!**
    - » one trick is to notice if an operand is denormal
    - » if not then this step won't happen
- **Alternative**
  - negative exponent → shift amount



## Improving on the Basic Algorithm

- **Multiplier is the slowest phase**
  - pipeline it and use the tactics you already know about
    - » output of multiplier's high half is in carry-save form
    - » then use row compressors to speed up partial product add
- **Overlap multiply with sticky bit computation**
  - basic method
    - » use conventional representation for low-half
      - → carry-propagate adders for partial product add
    - » then take bit-wise OR of the result and OR that to Sticky
  - improvement 1: use a trick
    - » number of trailing result 0's is the sum of the operand trailing 0's
      - if  $> 25$  (24 bit significand plus G) then  $S=0$  otherwise  $S=1$
  - improvement 2: use faster carry-save for low half as well
    - » determine sticky from carry-save representation of the low-half



## The Carry-Save Sticky

- **Basic idea**
  - **add -1 (all 1's in 2's complement) to partial product**
    - » effect: add one more row of partial products – e.g. -1
    - » if result would have been zero then result will be -1

```

S      ssssssss
C      cccccccc
-1     11111111
  
```

Note: I don't see the  
performance adv. here

```

-----
          zzzzzzzz
          tttttttt
 $Z_i = (S_i \text{ xor } C_i)'$ 
 $T_i = S_{i+1} + C_{i+1}$ 
 $W_i = Z_i \text{ xor } T_i$ 
Sticky = NAND( $W_i$ )
  
```



## Multiply-Add Fused

- **MAF advantages (note text views the glass as half full)**
  - **increased precision**
    - » single round and normalize as opposed to two
  - **common operation**
    - » hardware support for the common case principle
    - » benefit to the compiler as well
  - **simplifies forwarding/bypass logic**
    - » particularly important for long latency operations
  - **reduces register file pressure**
    - » savings in power and increases performance
      - one of the few times you can win on both fronts
  - **easy to use for either ADD or Multiply**
    - »  $X*Y+W$ 
      - Y set to 1 for an add
      - W set to 0 for a multiply



## Other FMA/MAF Issues (the book elides)

- IEEE 754 spec doesn't include MAF as an operation
  - Wedge it in as follows
    - » define new *super extended format*
      - allows doubles to be exactly represented
    - » define multiplication to silently cast operands to SEF and return exact result
    - » define addition to silently cast the W operand to SEF and return the result in the desired precision
  - SEF's added accuracy simplifies iterative divide and SQRT operations
  - Some serious software issues about when it should and shouldn't be used
    - » e.g.:  $\text{SQRT}(X*X-(Y*Y))$  when  $X==Y$ 
      - could return Zero, NaN, or a small positive number from MAF
      - non-MAF will return 0
      - oops!!



## MAF's and Compilers (also elided)

- Basic MAF facts
  - requires compiler support or custom assembly language
  - compilers are never forced to use MAF's
  - hence difficult in saying anything definitive about rounding behavior on systems with MAF hardware
  - compilers should have a switch that disables MAF code generation
- Register pressure
  - actually worse for an individual instruction
    - » 3 reads and 1 write for a MAF instruction
    - » → increase of register read ports may result
  - at algorithm level register pressure is less
    - » 3 reads and 1 write vs. 4 reads and 2 writes for non-MAF
- HW benefits
  - parallel partial product accumulation and addend alignment
  - add is done to product still in carry-save form
  - potential better support for denormals



## Basic MAF Algorithm

- $Z = X*Y+W$ 
  - $Mx * My; Ex+Ey = Exy$ 
    - » product must be kept in full double precision
      - since add may cancel the high-order half
    - » partial product adds can be in carry-save format
  - compare  $Exy$  and  $Ew$ 
    - » produces alignment shift
    - » shift addend significand
      - double precision result removes need to shift smaller significand
  - select  $\max(Exy, Ew)$  for exponent
  - add product and aligned addend
    - » result here needs to be in conventional form
  - normalize result and update exponent
  - round
  - determine exception flags and special values



## Alignment of W

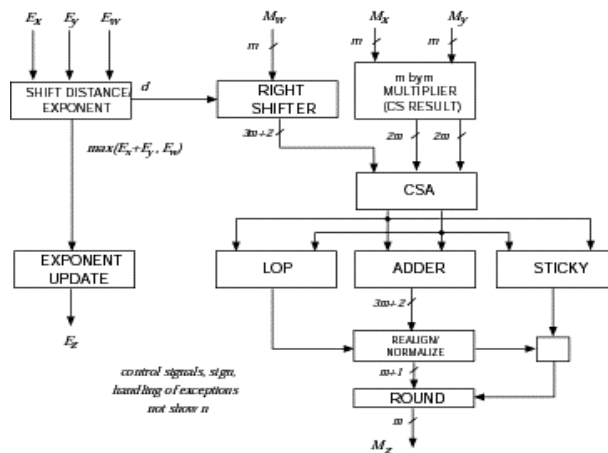
- Basic trick
  - By comparing  $Exy$  and  $Ew$  you can determine
    - » least significant bit of the product and the addend
  - However the distance between them can be enormous in either direction
    - » consider
      - large\*large+tiny OR tiny\*tiny+large
    - » need to avoid storing all the bits in between
    - » ideas?



## Alignment Cases

- **W is much smaller than X\*Y**
  - then W is crushed to sticky before being added
- **W is much larger than X\*Y**
  - then add it with a single 0 separator and crush X\*Y to sticky
- **W is smaller than X\*Y**
  - low-order part is crushed to sticky
  - high order part is added
- **W is larger than X\*Y**
  - simple align and add
  
- **Bottom line**
  - adder stage requires  $3m+2$  bits
    - »  $m$  bits for addend, separator,  $2m$  for product, and guard
  - the sticky bit is out there too

## Basic Implementation



text figure 8.19

## Devil is Still in the Details

- For biased exponent  $\max(E_x + E_y, E_w)$ 
  - $\rightarrow \max(E_bx + E_by - \text{bias}, E_bw)$
- Alignment of W w.r.t double precision product performed concurrently
  - since product isn't aligned
    - » left shift can be up to  $m+3$  positions
    - » right shift can be up to  $2m-1$  positions
  - avoid the need for bidirectional shift
    - » position addend  $m+3$  positions to the left of the product
    - » then shift right by  $d$ 
      - where  $d = E_x + E_y - E_w + m + 3$
      - which for a biased representation really means
        - $d = E_bx + E_by - E_bw - \text{bias} + m + 3$
    - » no shift is performed if  $d \leq 0$
    - » max shift is  $3m+1$



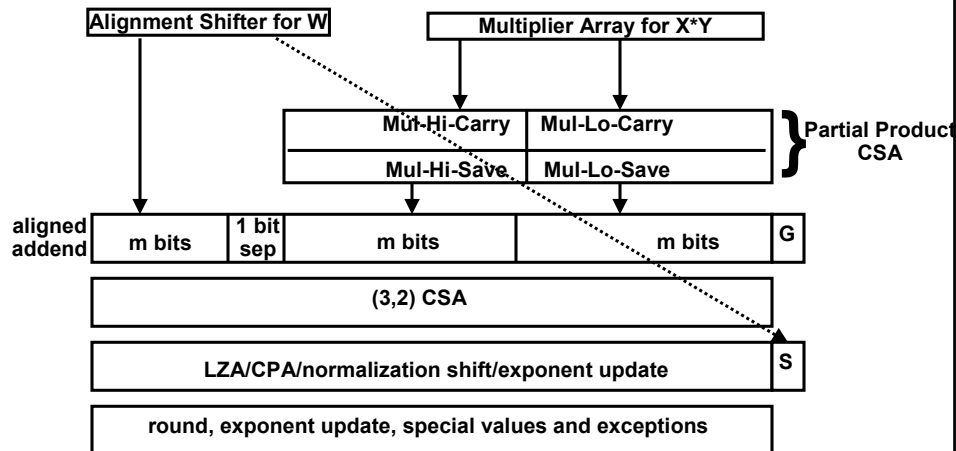
## More Devils

- Adder output may require realignment
  - since add may cancel high-order product bits
  - max left shift of up to  $2m$  bits may be required
  - fast method (same as with Fadd)
    - » leading one position (LOP)
      - note book terminology change – LOD for Fadd discussion
    - » replaced by LZA (leading zero anticipator)
      - same complexity as adder
    - » LZA and add step done in parallel
- Pipelining the design for higher throughput
  - not nearly as easy as the book would lead you to believe
  - a good pipeline is all about timing
    - » and timing is always a serious pain in the tuckus



## Alternative Implementation

- Source Intel FPCOE



## MAF Special Values and Exceptions

- Final operation is an ADD
  - hence all of this is the same as with Fadd
  - refer to slides 18, 19, 20

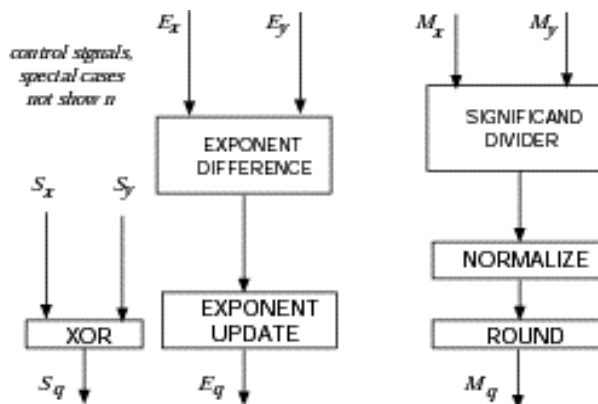


## Floating Point Division

- For  $q=x/d$
- Basic algorithm
  - divide the significands, subtract the exponents
    - »  $M_q = M_x/M_d$ 
      - use the methods you already know about
        - SRT (Sweeney, Robertson, Tocher developed this algo. independently)
      - the only nice thing about division is that  $q$  fits in the same  $m$  bits as  $x$  and  $d$ .
    - »  $E_q = E_x - E_d$ 
      - but we need to remember that the exponents are biased
      - hence:  $E_{bq} = E_{bx} - E_{by} + \text{bias}$
    - »  $S_q = \text{XOR}(S_x, S_d)$
  - normalize  $M_q$  and update exponent
  - round
  - determine exception flags and special values



## Divider Data Path



text figure 8.24



## Devil is STILL in the Details

- Normalization depends on range of significands
  - $x$  and  $d$  are between 1:2 if they are normals
  - hence  $q$  is between  $\frac{1}{2}$  and 1
    - » → a possible left shift of one position might be needed
    - » means the guard bit is needed as the shift-in value
- Rounding
  - things to notice
    - »  $G$  was used in normalization
    - » infinite number of bits might be needed for an exact result
      - as if FP ops are ever exact
    - » anything else?
      - yep – shows up in 2 slides
  - → need  $R$  and Sticky
    - » sticky is tricky (next slide)



## Tricky Stuff

- Sticky is tricky
  - Sticky bit is effectively
    - » 0 if the remainder is 0
    - » 1 if it isn't
  - Hence you need to check for the remainder = 0
- Rounding to Nearest has a trick you can exploit
  - the tie case can't happen
  - tie case →  $f+1$  bit exact quotient for an  $f$ -bit fraction
    - » →  $Mq = 1d_1 \dots d_f \cdot 1q_1 \dots q_f \cdot 1x2^{1-e} = 1x_1 \dots x_f \cdot 2^{f+2}$ 
      - $e$  is 0 or 1 since result may be normalized or not
        - remember the 1 bit shift
    - » LHS has an odd number of terms →  $< f+1$  leading 0's
    - » RHS has at least  $f+2$  leading 0's
    - » hence can't be true so tie can't happen



## Digit Recurrence Rounding

- **Digit recurrence division (text 5.2.2)**
  - do recurrent division
    - » need to do  $m+2+p$  steps
      - need  $m$  for the quotient
      - +2 for Guard and Round
      - $p$  is either 1 or 2 based on the redundant digit representation
    - then correct,
    - in the floating point case we then normalize and round
  - **What might happen**
    - if final residual is negative we may need to decrement the last bit of the quotient  $q_L$
    - rounding might then increment it again
  - **Opportunity to combine correct, normalize and round steps**



## Rounding to Nearest

- **$Q_1 \dots Q_m GR$** 
  - sign = sign of the residual
- **Correction**
  - $Q_m$ -sign
- **Rounding**
  - if quotient is normalized then add Sticky-sign to position  $G$
  - if quotient is not normalized then add Sticky-sign to position  $R$
- **Note book has a lot of notation (you've noticed)**
  - but this is the idea



## Floating Point SQRT

- $S = \text{Sqrt}(X)$
- **Basic algorithm**
  - $M_s = \text{SQRT}(M_x)$
  - $E_s = E_b/2$  (problem with this?)
    - » oops problem  $E_s$  may not be an integer
    - » if low order bit = 0
      - then shift exponent right 1
      - compute the square root of  $M_x$ 
        - done using iterative approximation methods (overview shortly)
    - » if low order bit = 1
      - then  $M_s = M_x/2$  and  $E_s = E_x + 1$
      - note this can happen at most once since what you care about is the real unbiased value of the exponent being even – if it was odd then this step fixes the problem but creates another
        - what is it?
  - **Normalize and update exponent**
  - **Round**
  - **Determine flags and special values**



## Ex low order bit test

- **Remember the bias**
  - $E_b = E_v + \text{bias}$
  - Hence  $E_b/2 = E_v/2 + \text{bias}/2 \neq E_v/2 + \text{bias} = \text{what we want}$ 
    - » therefore must add bias to  $E_b$  before the check
      - $(E_b + \text{bias})/2 = E_v/2 + \text{bias}/2 + \text{bias}/2 = E_v/2 + \text{bias}$
- **Trick**
  - do we need to do the whole addition to determine
    - » yes and no
      - yes if the  $E_b + \text{Bias}$  is even then we keep the value
      - no: if  $E_b + \text{Bias}$  is odd then we throw the sum away
    - » note bias is always odd in IEEE 754
      - hence if  $E_b$  is even then we don't do the add



## Other Issues

- **Normalization**
  - since  $x$  is between  $\frac{1}{2}$  and  $2$ 
    - » the  $\frac{1}{2}$  results from the adjustment step for an odd true exponent value
  - $\text{SQRT}(x)$  then has a range  $1/\text{SQRT}(2) : \text{SQRT}(2)$ 
    - » hence a max left shift of 1 is required for values  $< 1$
  - **Rounding modes**
    - » these are similar to division
      - $G$  could have been shifted
      - hence we need R and Sticky
        - Sticky is needed for rounding to  $\pm \infty$
        - Sticky is not needed for unbiased since the tie can't happen
        - rounding to zero is simple truncation



## Iterative Approximation

- **We didn't have time to cover Chapter 7**
  - serious *arithmites* should look at this
    - » useful for all weird stuff
      - Cordic algorithms for trigonometric functions
      - SQRT or iterative division
        - since both are somewhat rare
        - minimizing hardware maybe a better choice than fast hardware
- **Hence this is just an overview**
  - purpose = awareness
  - note: FMA/MAF circuits are very useful for this



## Newton's Method

- **note:** also called Newton-Raphson method
- **Idea**
  - use a Taylor's series to find solutions to an equation in the area of a suspected root
    - » problems – convergence and singularities if you guess wrong
  - Taylor's series of  $f(x)$  about a point  $x=x_0+e$ 
    - »  $f(x_0+e) = f(x_0)+f'(x_0)e+1/2f''(x_0)e^2+\dots$
    - » keep first order terms and set  $f(x_0+e)=0$
    - » solve for  $e=e_0$ 
      - $e_0 = - (f(x_0))/(f'(x_0))$
    - » eventually leads to a recurrence relation which says

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



## Iterative Divide w/ FMA's

- **Use Newton-Raphson**
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
- **For approximation to  $1/B$  set**
  - $f(x) = 1/(x-B)$
  - $f'(x) = -1/x^2$
- **Then**
  - $x_{i+1} = x_i + x_i^*(1-B*x_i)$ 
    - » requires 2 FMA's
- **Accuracy doubles until it reaches precision of calculation**
- **Important**
  - want to use unbiased rounding for all intermediate values
    - » to avoid accumulated error
  - final round needs to be to user specified mode



## Devil in the Details again

- **Need to pick a suitable value for  $x_0$** 
  - **lots of methods to do this**
    - » plus a lot of theory involved in the choice
  - **fortunately for division**
    - » trick is to not pick a value near 0
    - » given the range of the right answer is between 1/2 and 1
      - this shouldn't be hard
    - » picking somewhere in the middle makes sense
      - how about  $\frac{3}{4}$
  - **more practically**
    - » use high order bits as a table index to an initial value choice
    - » more index bits  $\rightarrow$  more rapid convergence
- **Stopping point**
  - when nothing changed right of the sticky bit



## Iterative SQRT with FMA's

### Also Newton-Raphson

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- **For approximation to  $1/(\text{Sqrt}(B))$  set**
  - $f(x) = 1/(x^2-A)$
  - $f'(x) = -2/x^3$
- **Then**
  - $x_{i+1} = x_i + x_i * (1/2 - (A/2 * x_i) * x_i)$ 
    - » requires 3 FMA's
- **Initial value**
  - know answer can be between  $1/\text{sqrt}(2)$  and  $\text{sqrt}(2)$ 
    - » pick  $\frac{3}{4}$  again or use index trick
- **Note**
  - inverse sqrt is easier than sqrt



## Whew

- That's it for floating point
- Bottom line
  - most of the hard stuff you already knew
    - » good algorithms for add/sub/mul/div
    - » they get used again
  - the school book algorithms are a start
  - but there is a lot of hair
    - » input operand checks
    - » result checks
    - » forcing special values
      - multiplexors and hardwired values selected conditionally
  - result of the hair
    - » floating point circuits tend to be larger and slower than their integer counterparts
- The hope
  - you got the basics
  - and you have a deeper appreciation of the dangers of using FP in your programs

