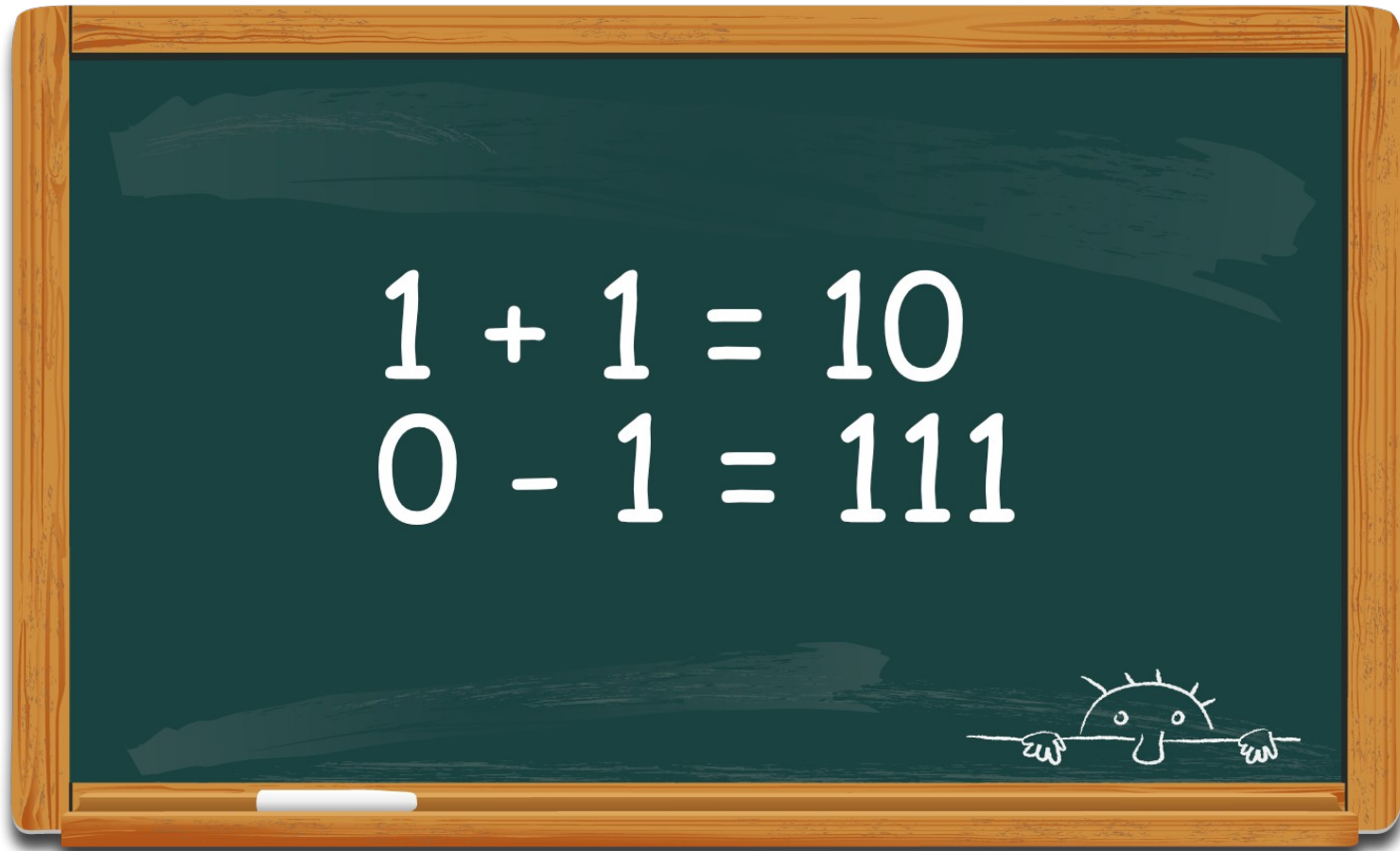


# FPGA Maths



by Iain Waugh

# Overview

- Number system basics
- Adding and subtracting
- Multiplying
- Using DSPs
- Dividing
- Overview of other math functions

# Main VHDL Number Types

- `natural`
  - 0 to  $2^{31} - 1$
  - 0 to 2,147,483,647
- `integer`
  - $-2^{31}$  to  $2^{31} - 1$
  - -2,147,483,648 to 2,147,483,647
- `unsigned`
  - Any size ('n' bits)
  - 0 to  $2^n - 1$
- `signed`
  - Any size ('n' bits + sign bit)
  - $-(2^n - 1)$  to  $2^n - 1$



# Natural and Integer VHDL Code

```
signal a          : natural := 2147483647;
--signal a2       : natural := 2**31-1; -- Fails
--signal a_plus_1 : natural := 2147483648; -- Fails
signal b          : integer := -2147483648;
--signal b_minus_1 : integer := -2147483649; -- Fails
signal c          : integer := 2147483647;
--signal c_plus_1  : integer := 2147483648; -- Fails
```

# Unsigned and Signed VHDL Code

- Unsigned data: every bit is part of the number
- Signed data: the top bit indicates '+' or '-'
- Signed data has a smaller 'max' value

```
signal a : unsigned(7 downto 0);  
-- Max is 255, Min is 0
```

7							0
n	n	n	n	n	n	n	n

```
signal b : signed(7 downto 0);  
-- Max is 127, Min is -128
```

+/-	n	n	n	n	n	n	n
-----	---	---	---	---	---	---	---

# Unsigned and Signed Verilog Code

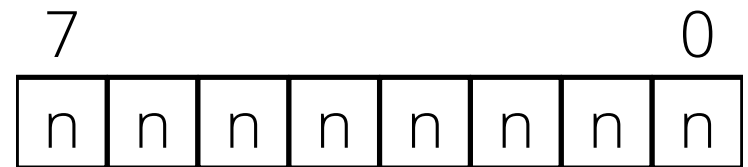
- Unsigned data: every bit is part of the number
- Signed data: the top bit indicates '+' or '-'
- Signed data has a smaller 'max' value

```
reg [7:0] a;
```

```
// Max is 255, Min is 0
```

```
reg signed [7:0] b;
```

```
// Max is 127, Min is -128
```



# Conversion Functions

- VHDL is strongly typed
- If you want to change between numbers and vectors, you need to use a conversion function
- If you want to treat them as '1's and '0's with no numerical meaning, you "cast" it to std\_logic\_vector

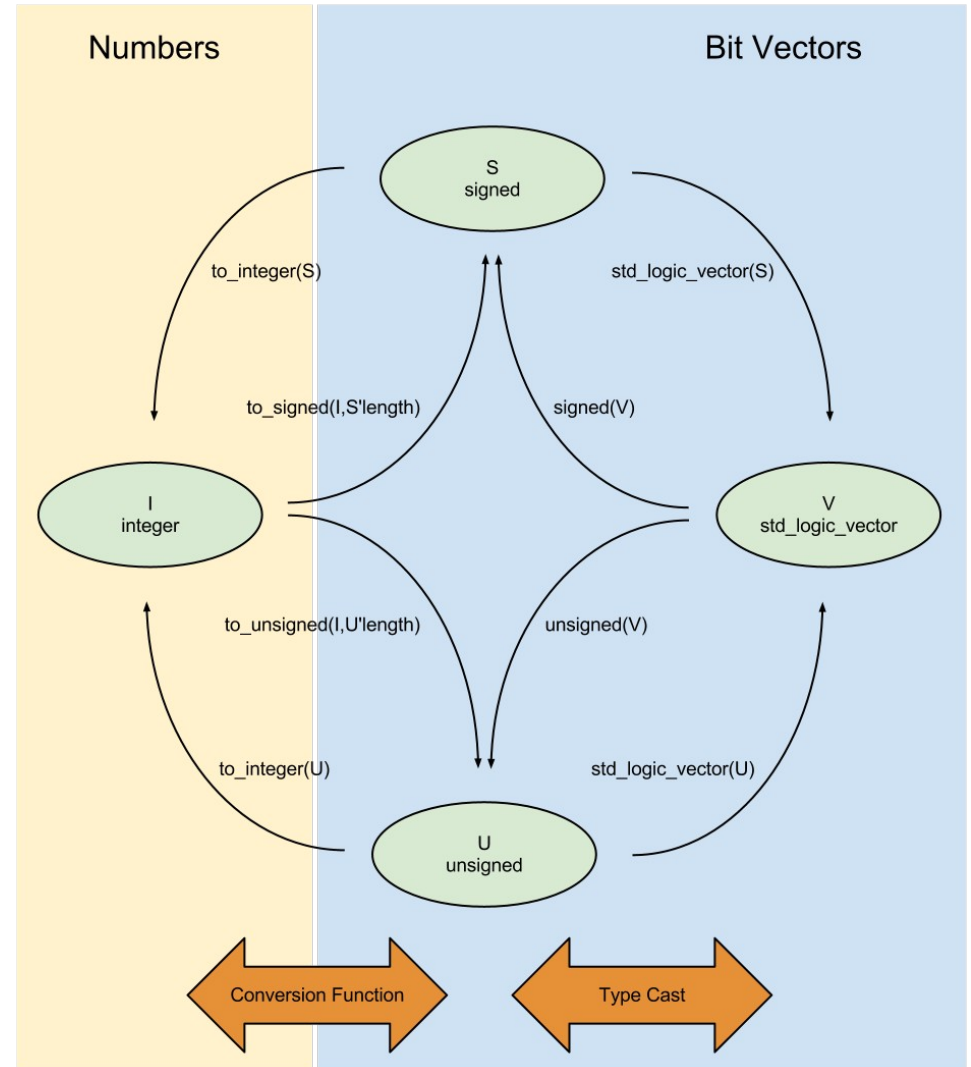


Image credit: Shannon Hilbert

# VHDL Types and Packages

- You have a choice of:
  - `ieee.numeric_std`
  - `ieee.std_logic_arith`
  - `ieee.std_logic_unsigned`
- Modern convention is:
  - Only use `numeric_std`
- The other packages have conflicting functions



# Adding: Theory

$$\boxed{0} + \boxed{0} = \boxed{0}$$

$$\boxed{0} + \boxed{1} = \boxed{1}$$

$$\boxed{1} + \boxed{0} = \boxed{1}$$

$$\boxed{1} + \boxed{1} = \boxed{0} \quad \boxed{1}$$

carry or  
overflow

- $a + b + \text{carry} = c$
- $c = a \text{ XOR } b$
- $\text{Carry} = a \text{ AND } b$
- The carry bit is why you have to use a longer vector for the result

# Adding: Practise

- Write it like you see it

`a = b + c;`

- Don't forget to have the right number of bits in the result ('a').

# Vector Sizes During Addition

- Two unsigned 4-bit numbers:  $a=1111$ ,  $b=1111$
- $a+b = 11110$  (a 5-bit number)

# Vector Sizes During Addition

- Two unsigned 4-bit numbers:  $a=1111$ ,  $b=1111$
- $a+b = 11110$  (a 5-bit number)
- When adding/subtracting any 2 values, the length of the result =  $\max(\text{len}(a), \text{len}(b)) + 1$

# Vector Sizes During Addition

- Two unsigned 4-bit numbers:  $a=1111$ ,  $b=1111$
- $a+b = 11110$  (a 5-bit number)
- When adding/subtracting any 2 values, the length of the result =  $\max(\text{len}(a), \text{len}(b)) + 1$
- $a+b+c = 2$  extra bits ( $15+15+15 = 45$ )
- $a+b+c+d = 2$ extra bits ( $15+15+15+15 = 60$ )
- Adding/subtracting 'n' values, add  $\log_2(n)$  bits

# Increasing Signed Lengths

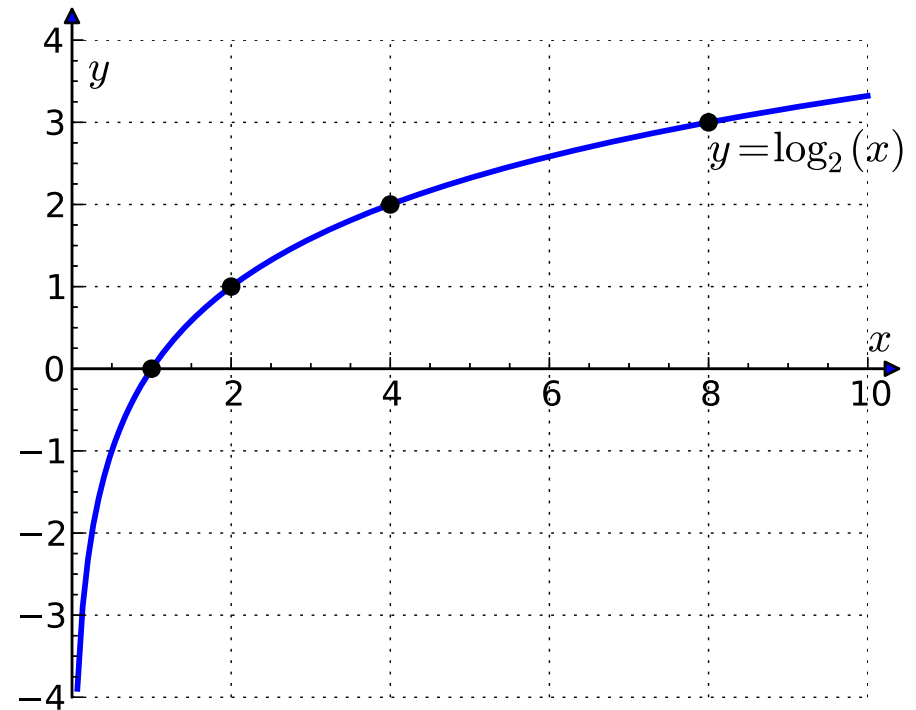
- If you want to put a signed value into a longer vector, you have to "sign extend" it.
- Copy the top-most bit's value into all the new extra top bits.

**1**001      **0**001

**11**1001      **00**0001

# $\log_2(n)$ ?

- If you want: "the number of bits that are needed to hold the number 'n' "
- $\log_2(x)$  isn't what you're after
  - $\log_2(0)$  isn't 1
  - $\log_2(1)$  is 0 (want 1)
  - $\text{ceil}(\log_2(n))$  isn't quite right either



# Which log2(n) Function for 'n' Bits?

- You need a custom function to return '1' when n=0
- [ieee.math\\_real](#) doesn't help
- See "test\_log.vhd" source  
vsim -c -do test\_log.do

```
function nbits(x : natural) return
natural is
    variable temp : natural := x;
    variable n    : natural := 1;
begin
    while temp > 1 loop
        temp := temp / 2;
        n    := n+1;
    end loop;
    return n;
end nbits;
```

Integer value	0	1	2	3	4	5	7	8	15	16
Needs this many bits:	1	1	2	2	3	3	3	4	4	5
math_real : log2(real(n))	n/a	0	1.00	1.58	2.00	2.32	2.81	3.00	3.91	4.00
integer(ceil(log2(real(n))))	n/a	0	1	2	2	3	3	3	4	4
'nbits' function	1	1	2	2	3	3	3	4	4	5



# Juggling With numeric\_std

- This is VHDL at its most verbose...
- Especially if you're operating across different bit-widths
- Even more so if you're mixing signed and unsigned types

# resize shift\_left Gotcha

- Q: What's wrong with this:

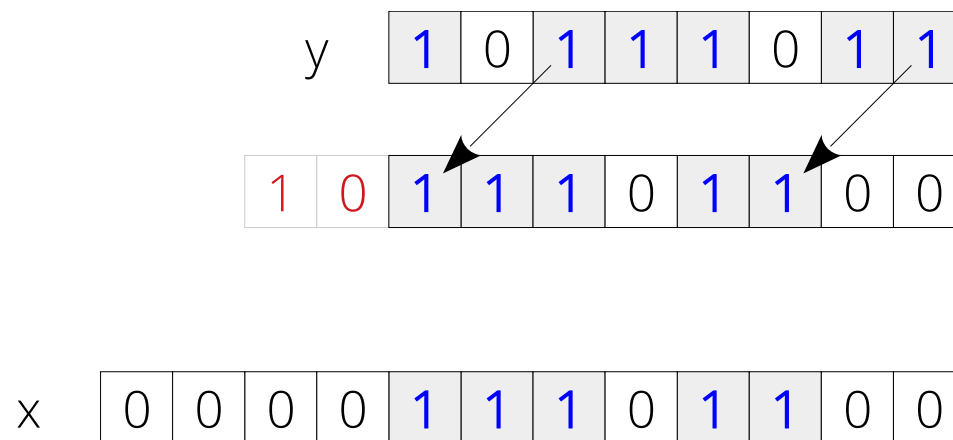
```
x <= resize(shift_left(y, 2), x'length);
```

# resize shift\_left Gotcha

- Q: What's wrong with this:

```
x <= resize(shift_left(y, 2), x'length);
```

- A: `y` is shifted left before the `resize`, so you may lose the top bits of it before it is assigned to `x`.



# resize shift\_left Gotcha

- Q: What's wrong with this:

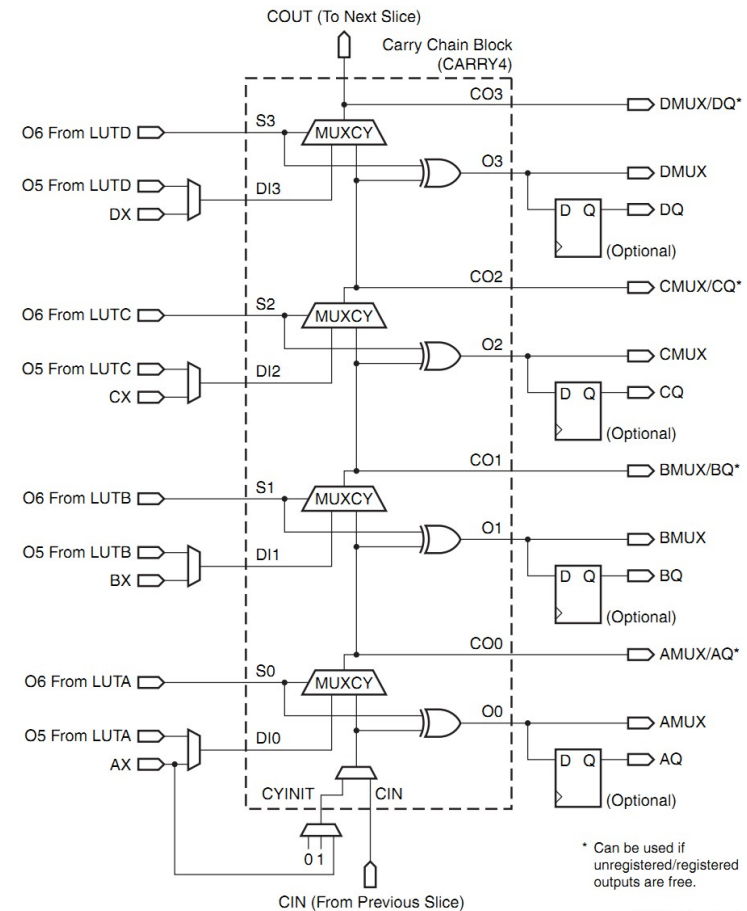
```
x <= resize(shift_left(y, 2), x'length);
```

- A: `y` is shifted left before the `resize`, so you may lose the top bits of it before it is assigned to `x`.
- Do this instead:

```
x <= shift_left(resize(y, x'length), 2);
```

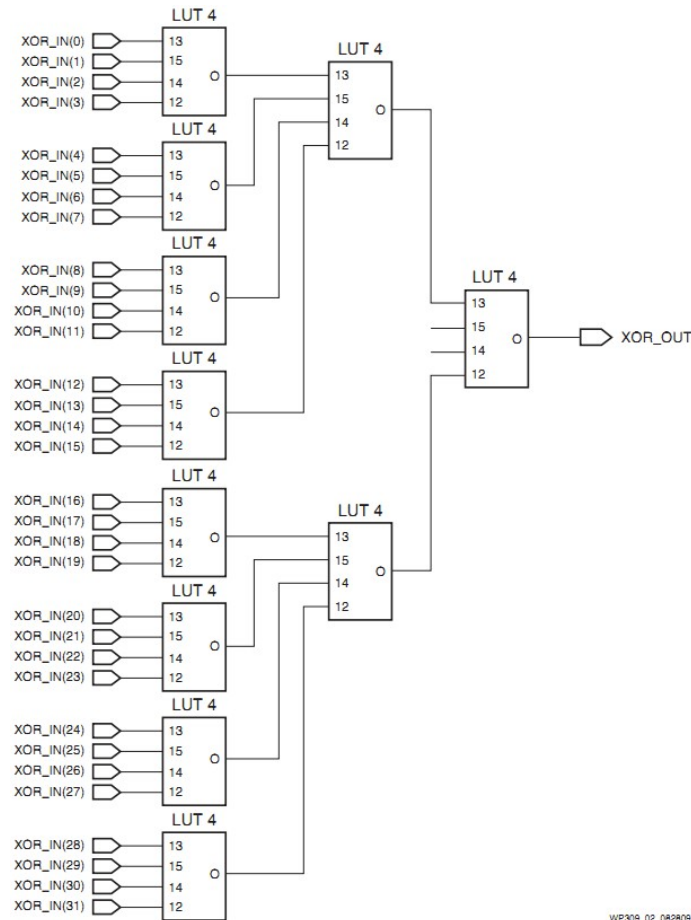
# FPGA Adders

- Xilinx slices have dedicated fast adder logic (vertically)
- For 7-Series FPGAs:
  - 1x CLB has 2x slices
  - 1x slice has 4x LUTs
  - ...and a fast carry chain for 4x LUTs



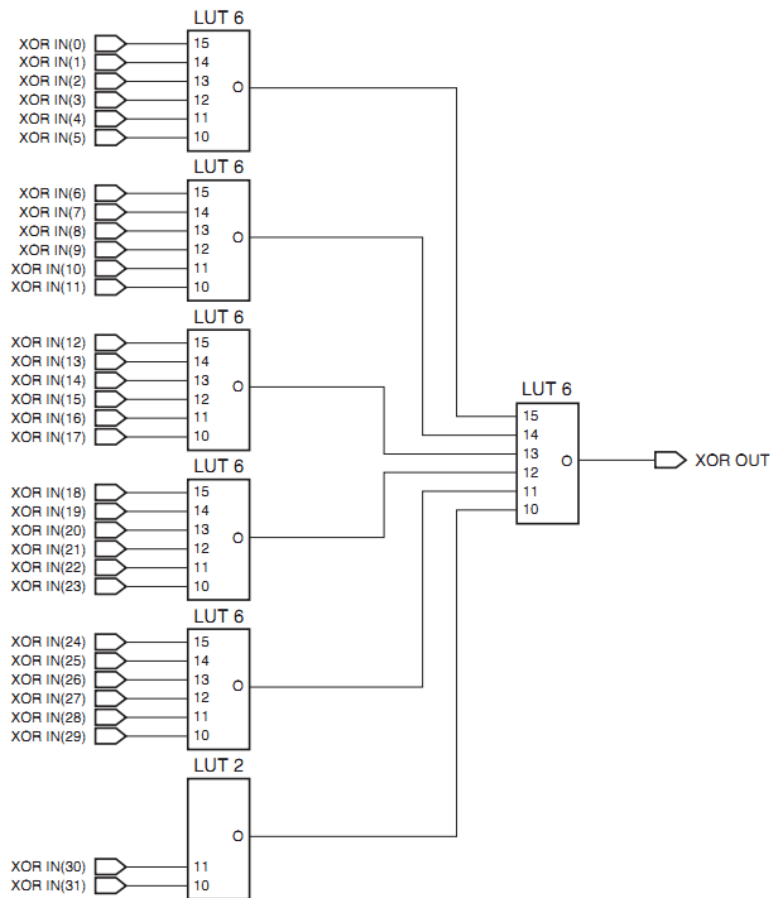
UG474\_c2\_23\_071813

# Note: Wide Operations Are Slower...



- 32-bit operation requires 3 levels of logic for 4-input LUTs

# ...for Some Architectures



WP309\_01\_082809

- 32-bit operation with 6-input LUTs requires 2 levels of logic

# Subtracting: Theory

- Convert the number to subtract (the subtrahend) to 2's complement
  - (invert the bits, then add '1')

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

 = 13

-

0	0	0	1	1	0	1	1
---	---	---	---	---	---	---	---

 = 27

Invert 27

1	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---

 = -28

add '1'

1	1	1	0	0	1	0	1
---	---	---	---	---	---	---	---

 = -27



# Subtracting: Theory

- Convert the number to subtract (the subtrahend) to 2's complement

- (invert the bits, then add '1')

- Add the two together

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array} = 13$$

$$- \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} = 27$$

$$\text{Invert 27} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array} = -28$$

$$\text{add '1'} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline \end{array} = -27$$

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array} = 13$$

$$+ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline \end{array} = -27$$

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ \hline \end{array} = -14$$

# Subtracting: Practise

- Write it like you see it

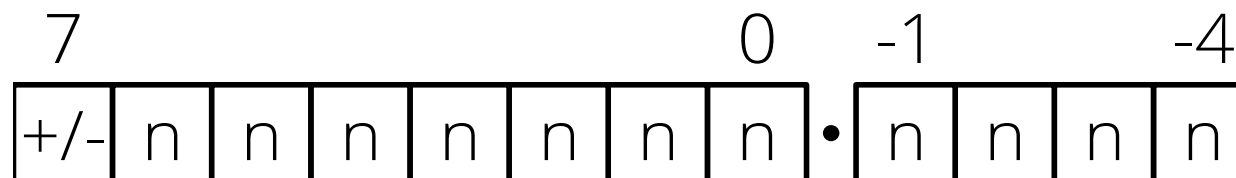
`a = b - c;`

- Don't forget to have the right number of bits in the result ('a').

# Other Number Systems

- Floating Point (won't be covered here)
- Fixed Point

```
signal a : signed(7 downto -4);
```



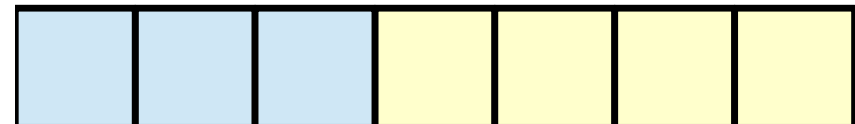
- Add and subtract, no problem
- Multiply, divide; take care where the '.' goes

# Multiplying in Logic

- Old-school maths
- Long multiplication
- The length of the result is:  
length of A +  
length of B

1	0	0	1
---	---	---	---

1	0	1
---	---	---



- Terminology:
  - The **Multiplier** is the smaller number
  - The **Multiplicand** is the bigger number



# Multiplying in Logic

- Use a shift+add technique
  1. Examine the bits of the multiplier from right to left
  2. If it's a '1', add the multiplicand to an accumulator
  3. Shift the multiplicand left
  4. Do this 3 times  
Here,  $nbits(multiplier) = 3$

$$\begin{array}{r}
 \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9 \\
 \times \boxed{1} \boxed{0} \boxed{1} = 5 \\
 \hline
 \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9
 \end{array}$$

$$\begin{array}{r}
 \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9 \ll 1 \\
 \times \boxed{1} \boxed{0} \boxed{1} = 5 \\
 \hline
 \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9
 \end{array}$$

$$\begin{array}{r}
 \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9 \ll 2 \\
 \times \boxed{1} \boxed{0} \boxed{1} = 5 \\
 \hline
 \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9 \ll 2 \\
 + \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{1} \boxed{0} \boxed{0} \boxed{1} = 9 \\
 \hline
 \boxed{0} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} = 45
 \end{array}$$

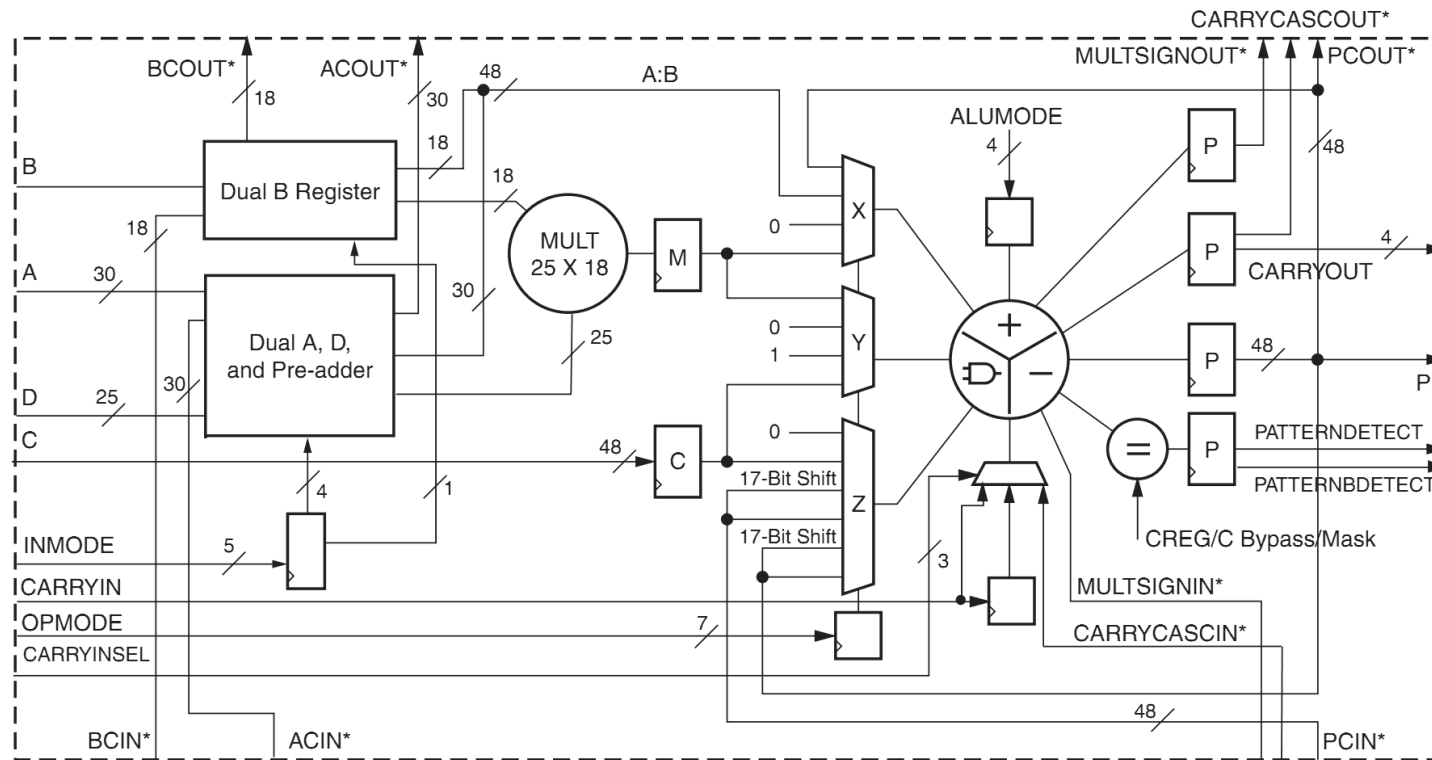


# Multiply in Logic Faster

- With 4 and 6-input LUTs, you can handle 2 bits per cycle with minimal penalty to fMax

# Working With DSPs

- Lots of maths functions in 1 block
- Xilinx DSP48E1 shown below

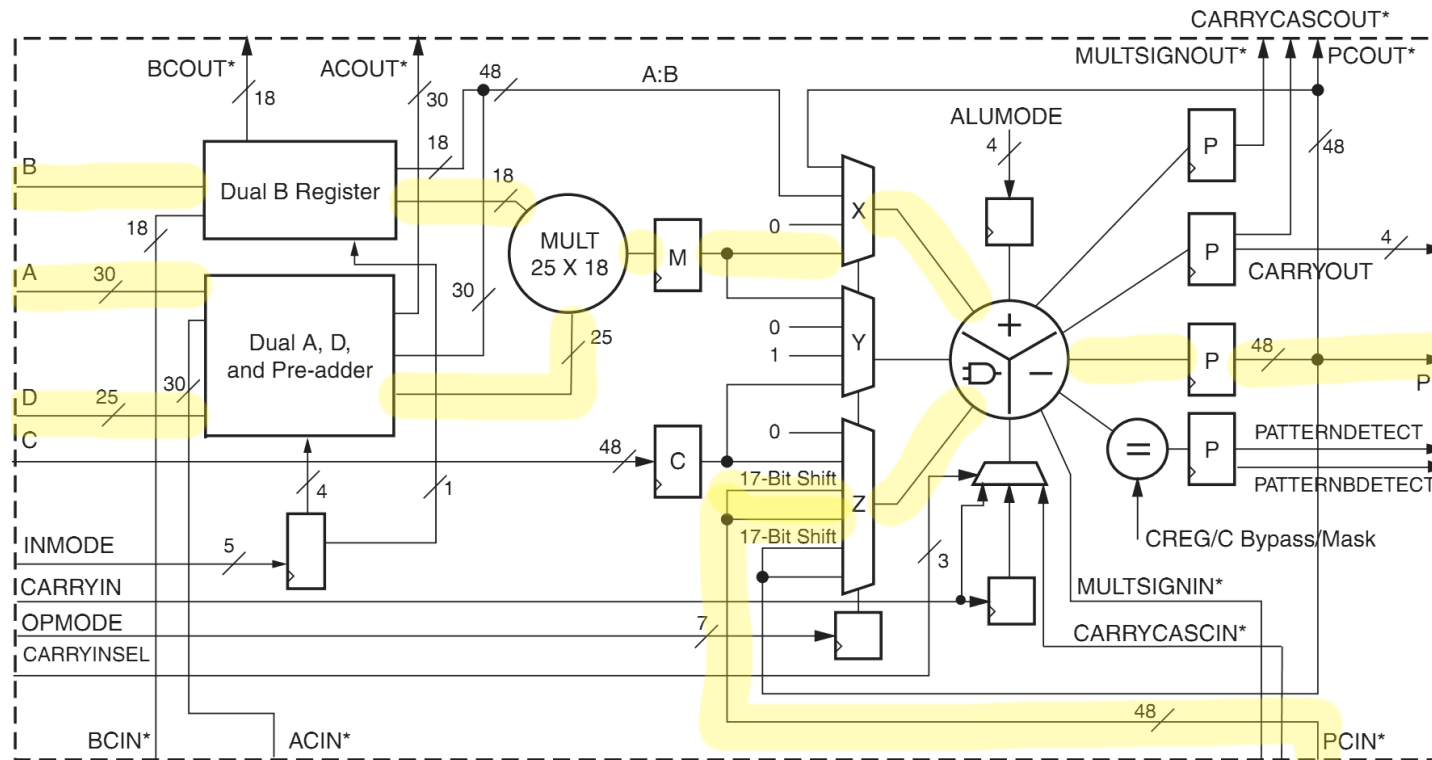


\*These signals are dedicated routing paths internal to the DSP48E1 column. They are not accessible via fabric routing resources.

UG369\_c1\_01\_052109

# Working With DSPs

- Most DSPs have the ability to pre-add, then multiply, then post-add: Here is  $P = (A + D) * B + PCIN$



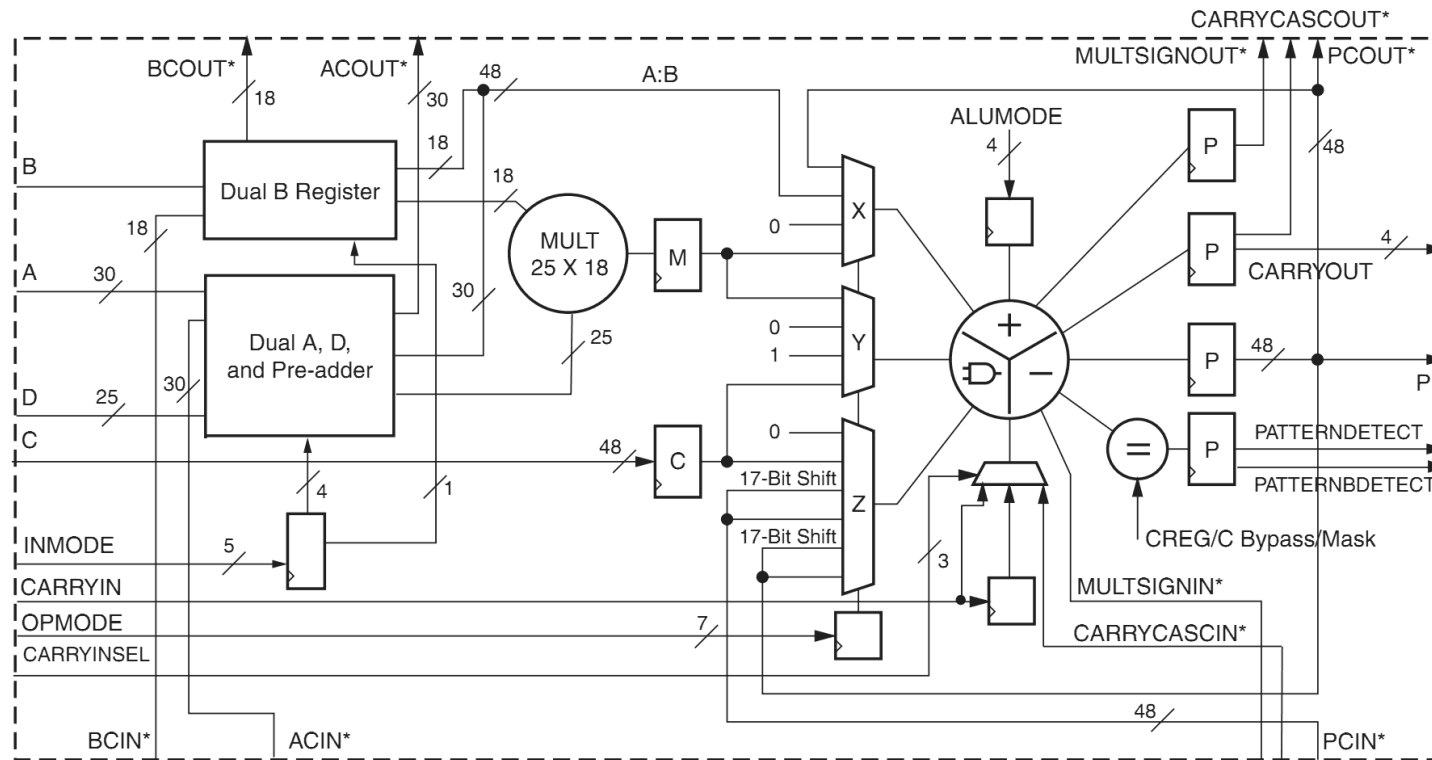
\*These signals are dedicated routing paths internal to the DSP48E1 column. They are not accessible via fabric routing resources.

UG369\_c1\_01\_052109



# Working With DSPs

- Note: Ports on the top/bottom have dedicated routing.
- Only ACOUT, BCOUT, etc. can connect to ACIN, BCIN, etc.




\*These signals are dedicated routing paths internal to the DSP48E1 column. They are not accessible via fabric routing resources.

UG369\_c1\_01\_052109

# Example: H.264 Sub-Pixel Interpolator

- The pixel between  $p_3$  and  $p_4$  is a filter of 6 input pixels:

$p_1, p_2, p_3, p_4, p_5, p_6$



- Filter taps are 1, -5, 20, 20, -5, 1, which gives:

$$(p_1 \cdot 1 + p_2 \cdot -5 + p_3 \cdot 20 + p_4 \cdot 20 + p_5 \cdot -5 + p_6 \cdot 1) / 32$$

- Notice the symmetry

- Re-write as:

$$((p_1 + p_6) \cdot 1 + (p_2 + p_5) \cdot -5 + (p_3 + p_4) \cdot 20) / 32$$

- Reduces to 2x DSPs : preadd, multiply, post-add

# Numbers Bigger Than 1 DSP?

- Xilinx DSP48E1 multiplier inputs are 18x25bits
- Question:  
How do you do 25-bit x 35 bit?  
 $A[24:0] \times B[34:0]$

# Numbers Bigger Than 1 DSP?

- Xilinx DSP48E1 multiplier inputs are 18x25bits
- Question:  
How do you do 25-bit x 35 bit?  
 $A[24:0] \times B[34:0]$
- Answer: Break B into 2 pieces ( $B_2$  = bottom 16)

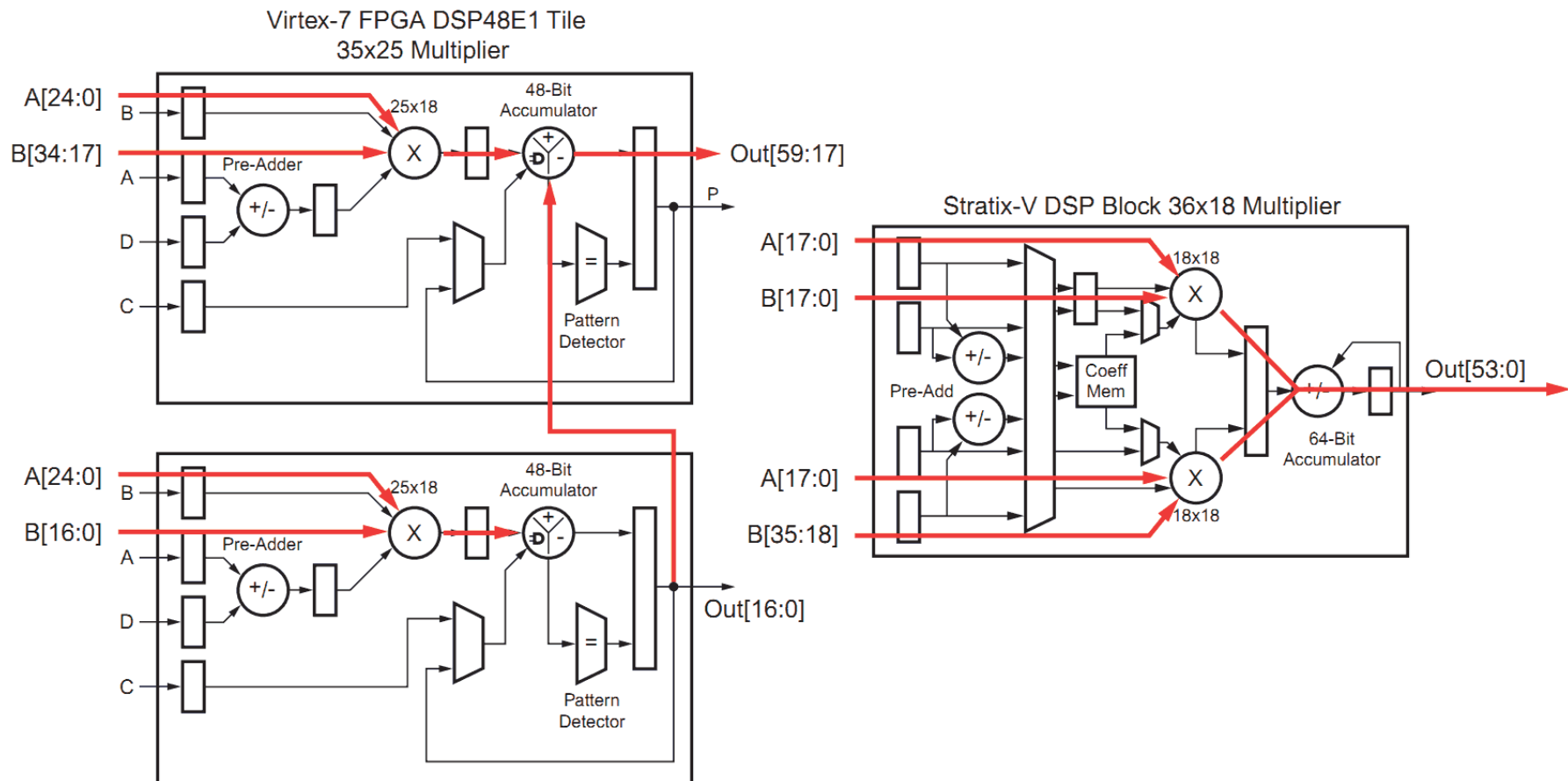
# Numbers Bigger Than 1 DSP?

- Xilinx DSP48E1 multiplier inputs are 18x25bits
- Question:  
How do you do 25-bit x 35 bit?  
 $A[24:0] \times B[34:0]$
- Answer: Break B into 2 pieces ( $B_2$  = bottom 16)
- $A \times B \Rightarrow (A \times B_1) + (A \times B_2)$

# Numbers Bigger Than 1 DSP?

- Xilinx DSP48E1 multiplier inputs are 18x25bits
- Question:  
How do you do 25-bit x 35 bit?  
 $A[24:0] \times B[34:0]$
- Answer: Break B into 2 pieces ( $B_2$  = bottom 16)
- $A \times B \Rightarrow (A \times B_1) + (A \times B_2)$
- $A \times B \Rightarrow (A \times (B_1/2^{16})) \times 2^{16} + (A \times B_2)$

# Numbers Bigger Than 1 DSP?



WP406\_04\_011713

Image credit: Xilinx WP406

# Signed Numbers in DSP

- The DSP48E1 inputs are signed numbers
- If your vector has less bits than the DSP input, you must "sign extend" it to the DSP's width

1	0	1	0	1	1	0	1
---	---	---	---	---	---	---	---

1	1	1	1	1	1	0	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---

- Tip: To save power, pack your bits towards the MSB and zero out the LSBs if you can



# Rounding

- Several methods, depending on your needs
- Also ways to restore precision

Method	"Truncate"	"Round"
Advantage	Fast	Better DC balance
Disadvantage	Loses DC balance	Requires an adder
Restoring precision	Closer to original	Not so close
Example: "10010110" (150)	"100101" (37 = off by -0.5)	"100110" (38 = off by 0.5)
Restore by "Echo-ing" LSBs	"10010101" (149 = off by 1)	"10011010" (154 = off by 4)

- There are dozens of ways to round..!

# Dividing

- By 2, 4, 8 etc. is easy, just shift right
- Otherwise, 2 main algorithms:
  - Restoring
  - Non-restoring

$$\begin{array}{l} \text{dividend} \\ \hline \text{divisor} \end{array} \frac{x}{d} = q + \text{rem}$$

quotient

# Restoring

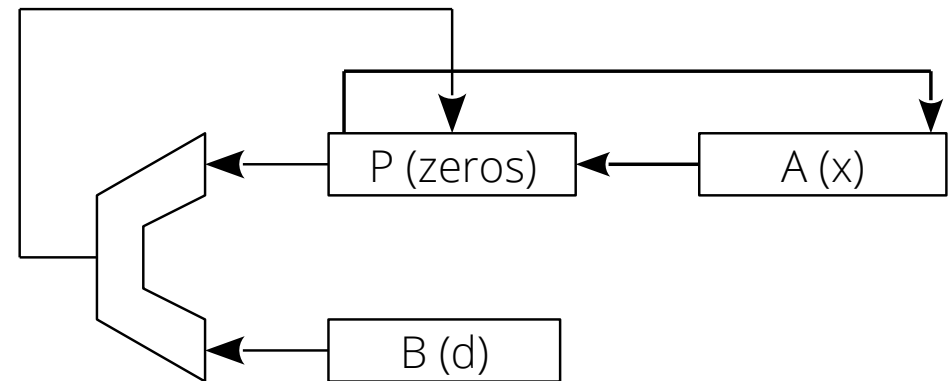
- Starting off:
- Put 'x' in register A  
'd' in register B  
Fill register P with '0's
- Do the next slide 'n' times  
('n' is the quotient length)

dividend

$$\frac{x}{d} = q + \text{rem}$$

divisor

quotient



# Restoring

1. Shift the register pair (P,A) one bit left
2. Subtract the contents of B from P, put the result back in P
3. If the result is -ve, set the low-order bit of A to 0 otherwise to 1
4. If the result is -ve, restore the old value of P by adding the contents of B back in P

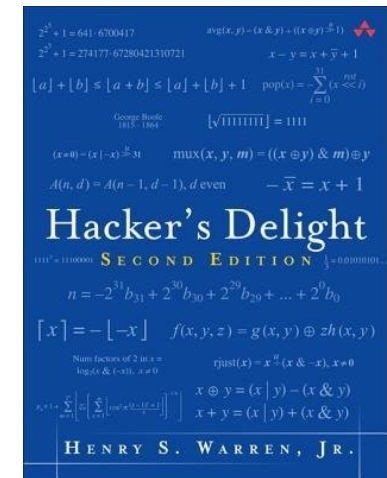
Text credit: Sherif Galal, Dung Pham

# Non-Restoring

1. If  $P$  is negative
  - Shift the register pair  $(P, A)$  one bit left
  - Add the contents of register  $B$  to  $P$
2. If  $P$  is positive
  - Shift the register pair  $(P, A)$  one bit left
  - Subtract the contents of register  $B$  from  $P$
3. If  $P$  is negative, set the low-order bit of  $A$  to 0, otherwise set it to 1
4. After  $n$  cycles
  - The quotient is in  $A$
  - If  $P$  is positive, it is the remainder, otherwise it has to be restored (add  $B$  to it) to get the remainder

# Divide By a Constant

- Best described in "Hacker's Delight"
- Example divide by 7 in Git repo, 'src' dir
- General method is to multiply by the reciprocal of the divisor (d) which looks like  $2^{32}/d$ , then extract the left-most 32 bits.
- Can operate in 1 clock cycle at >250MHz with 14-bit accuracy



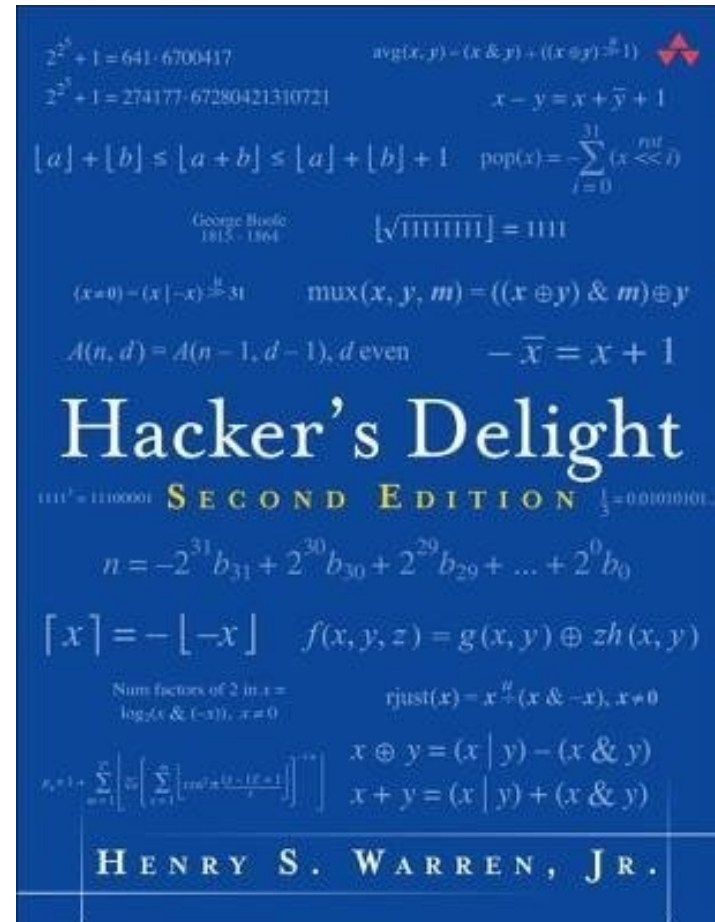
# Other Maths Functions

- Squaring is easy:  $A \times A$
- Square root is harder: see UG073, p62
- Can also be solved with a lookup table, a pre-index step and a post-index step
- Similarly for  $\log(n)$
- See example in 'src'

# References

- Hacker's Delight is filled with maths tricks for CPUs
- Some are very hardware-friendly

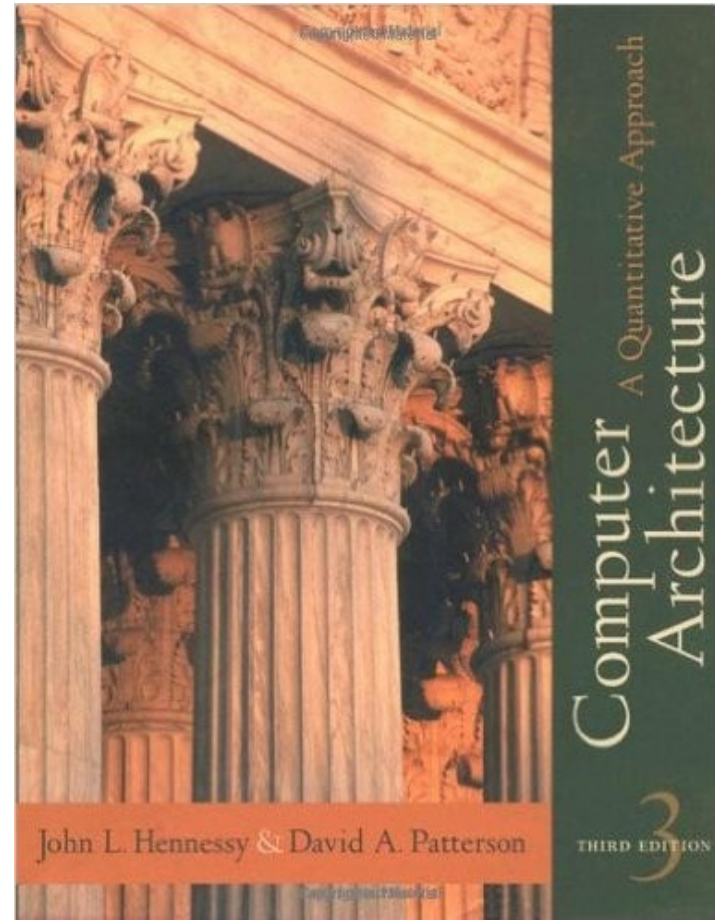
<http://www.hackersdelight.org/>





# References

- Computer Architecture:  
A Quantitative Approach  
Hennessy ,Patterson  
(Appendix J)
- Free Appendix J PDF at:  
[http://booksite.elsevier.com/  
9780123838728/references.  
php](http://booksite.elsevier.com/9780123838728/references.php)



# References

- Xilinx UG073 : Very good; maths, filters, more
- Xilinx UG195 Section 4 : maths with DSP48E
- Xilinx WP406 : Xilinx DSP vs. Altera DSP
- Altera's Advanced Synthesis Cookbook  
[www.altera.com/literature/manual/cookbook.zip](http://www.altera.com/literature/manual/cookbook.zip)
- Also, the old-school 8-bit demo-scene (!)
- Quake 3 source code (!)

# Questions?