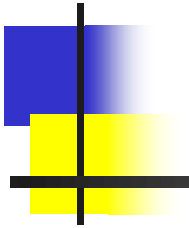


4 : Arithmetic for Computers

Rajesh Panicker, NUS

CG3207

Acknowledgement :

- Text by Patterson and Hennessey and companion slides by Mary Jane Irwin
- Text and companion slides by Harris and Harris
- Some slides from A/Prof. Bharadwaj Veeravalli
- Some figures from Wikipedia

Note:

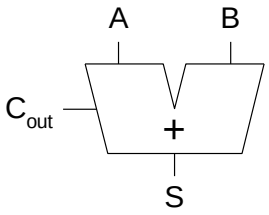
- Not all slides will be covered in the lecture. The rest are left as a self-learning exercise.

Contents

- Adder, Carry lookahead adder, Subtractor
- ALU with ADD/SUB/AND/OR
- Shifter
- Multiplier, Divider
- Floating point arithmetic

1-Bit Adders

Half Adder

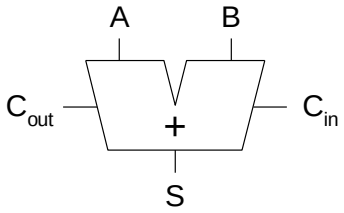


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



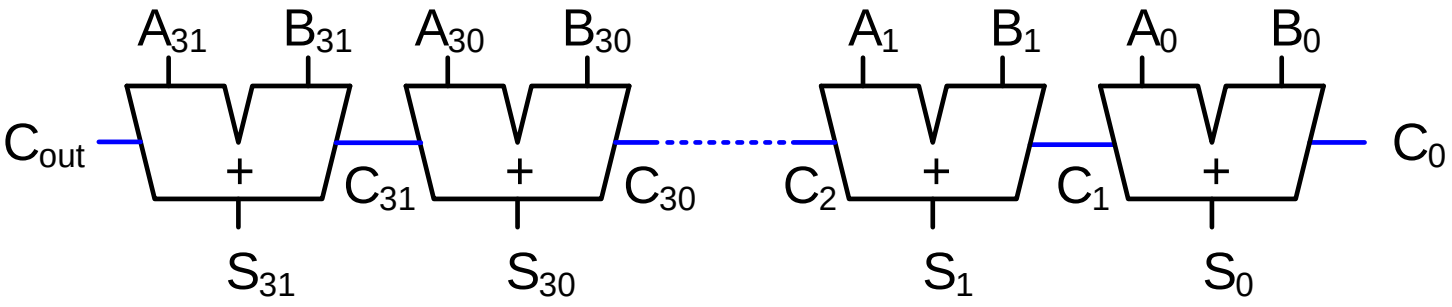
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Advantage : simple hardware
- Disadvantage: slow (could be a part of the critical path)



Carry-Lookahead Adder

- Possible solution : express and implement C_{i+1} directly in terms of (A_0,...,A_i, B_0,...,B_i,C_0) instead of waiting for it to ripple in
  - The number of terms would be huge for higher C_{i+1}s, implying huge number of gates, with each gate having a high fan-in
  - Fan-in / fan-out is the number of inputs / outputs driving / driven by a gate. Large fan-in/out usually means the gate will have a larger delay
- Carry-lookahead - understanding when we would generate a carry and when we would propagate

Carry-Lookahead Adder

- Idea : compute C_{i+1} from a pre-processed form of inputs. Observe the following
  - $C_{i+1}=1$ when $A_i = B_i = 1$ regardless of C_i. Stage i *generates* a carry. $g_i = A_i.B_i$
  - $C_{i+1}=1$ when $C_i = 1$ and either $A_i = 1$ or $B_i = 1$. Stage i *propagates* a carry. $p_i = A_i + B_i$
  - $C_1 = p_0C_0+g_0$
  - $C_2 = p_1C_1+g_1 = p_1p_0C_0 + p_1g_0 + g_1$

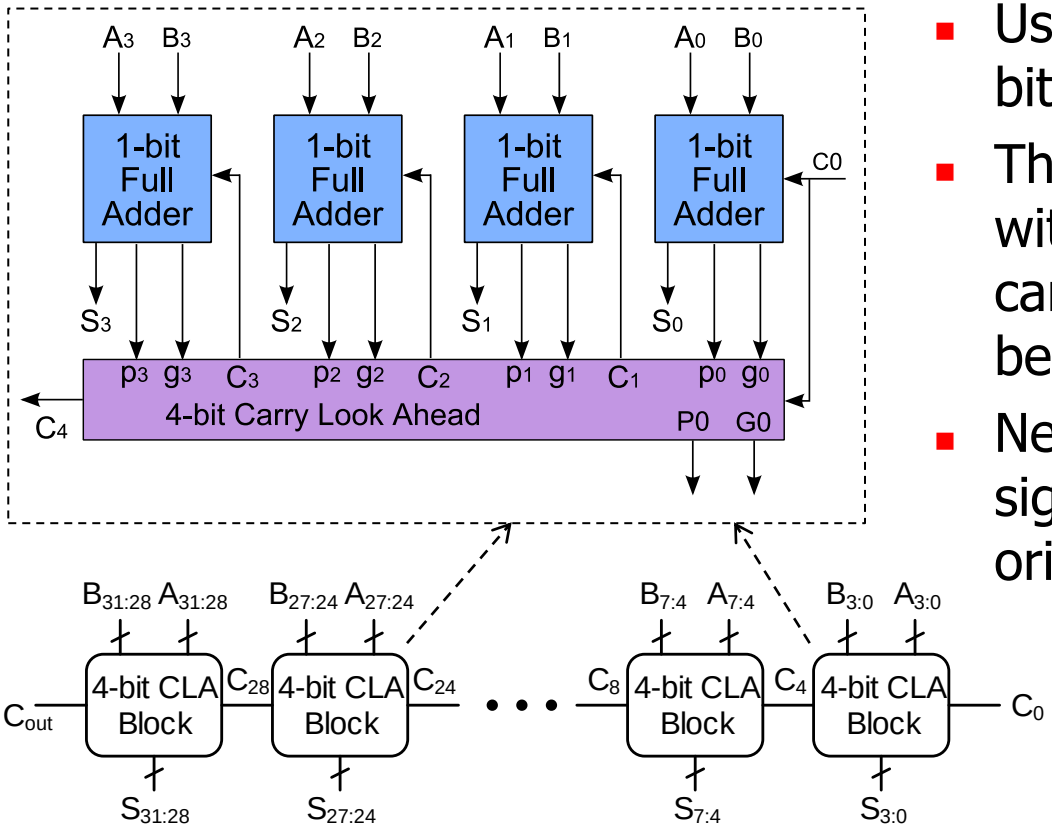
    We can continue this … but

    How far can we go?

    Doesn't this defeat our purpose?

Carry-Lookahead Adder

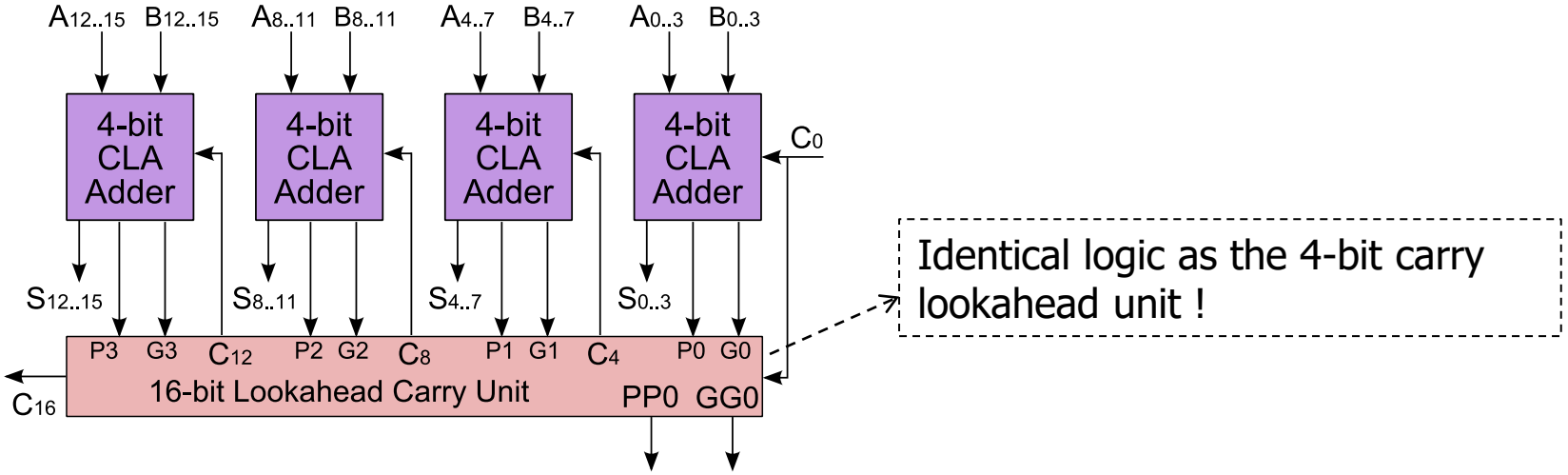
- Doing this for 32 stages is cumbersome. Do this for, say, 4 stages and call this 4 stage unit as a fundamental block
  - The first block takes A_0,B_0, …, A_3,B_3,C_0 and gives (S_0,S_1,S_2,S_3,C_4)



- Use 8 such units to handle 32 bits
- The ripple effect is avoided within a block, but the ripple carry effect will be present between blocks
- Nevertheless, the delays are significantly lower than the original ripple carry adder
  - A better solution?

Carry-Lookahead Adder

- Block propagate (P_i) and generate (G_i) signals for 4-bit blocks
  (i refers to the block number here)
  - $P_0 = p_3 p_2 p_1 p_0$
  - $G_0 = g_3 + p_3 (g_2 + p_2 (g_1 + p_1 g_0))$
- 16-bit adder using 4-bit CLAs

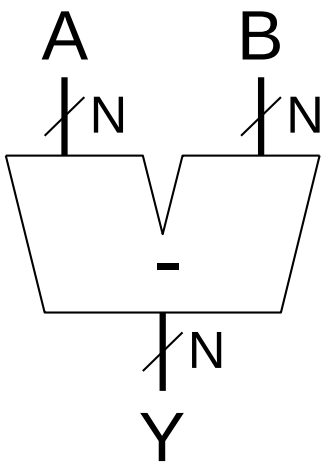


- How about 32 bit?
  - Cascade 2 x 16-bit units
  - Or … ?

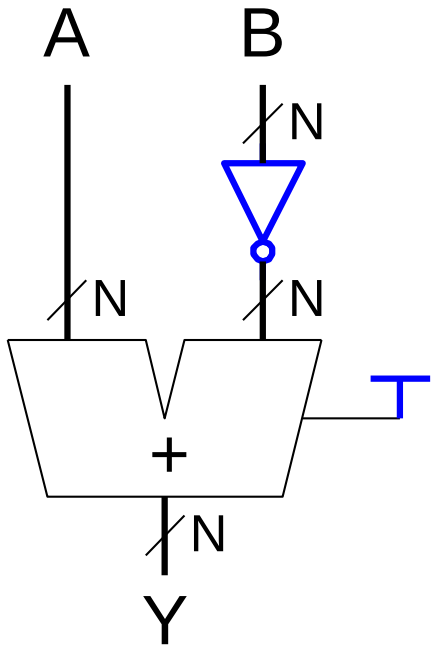
Subtractor

- Basic idea : A-B = A+B'+1

Symbol

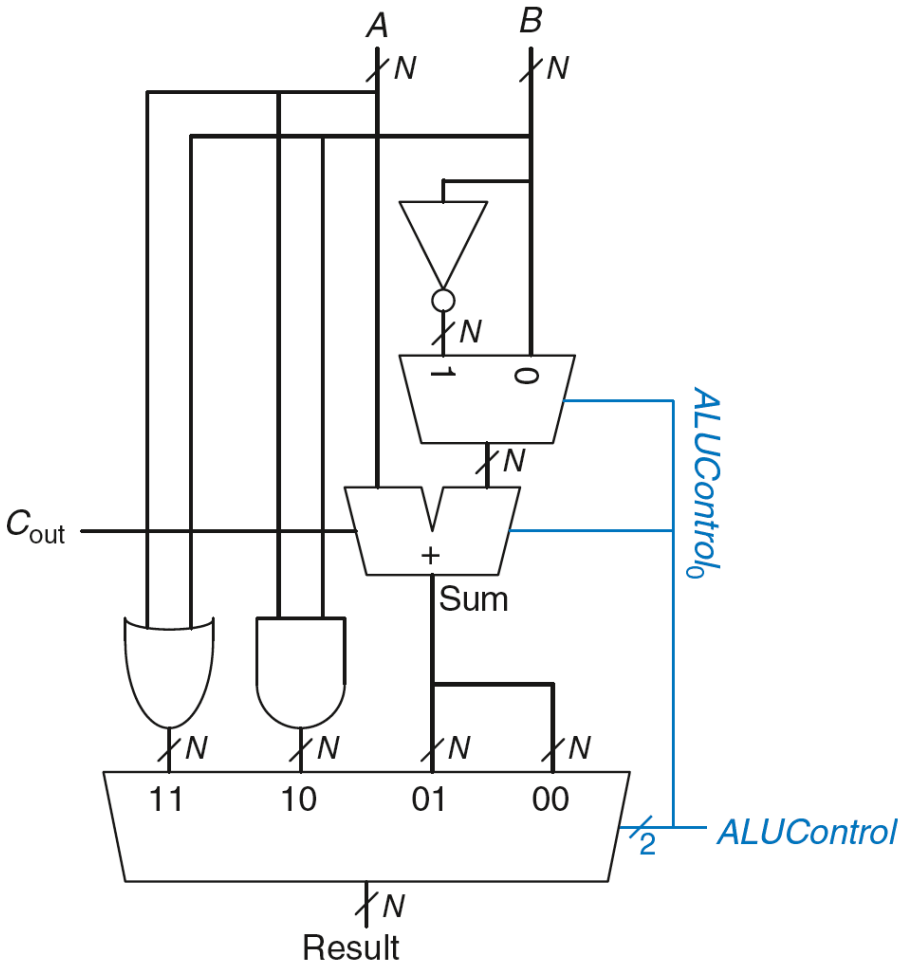


Implementation



ALU

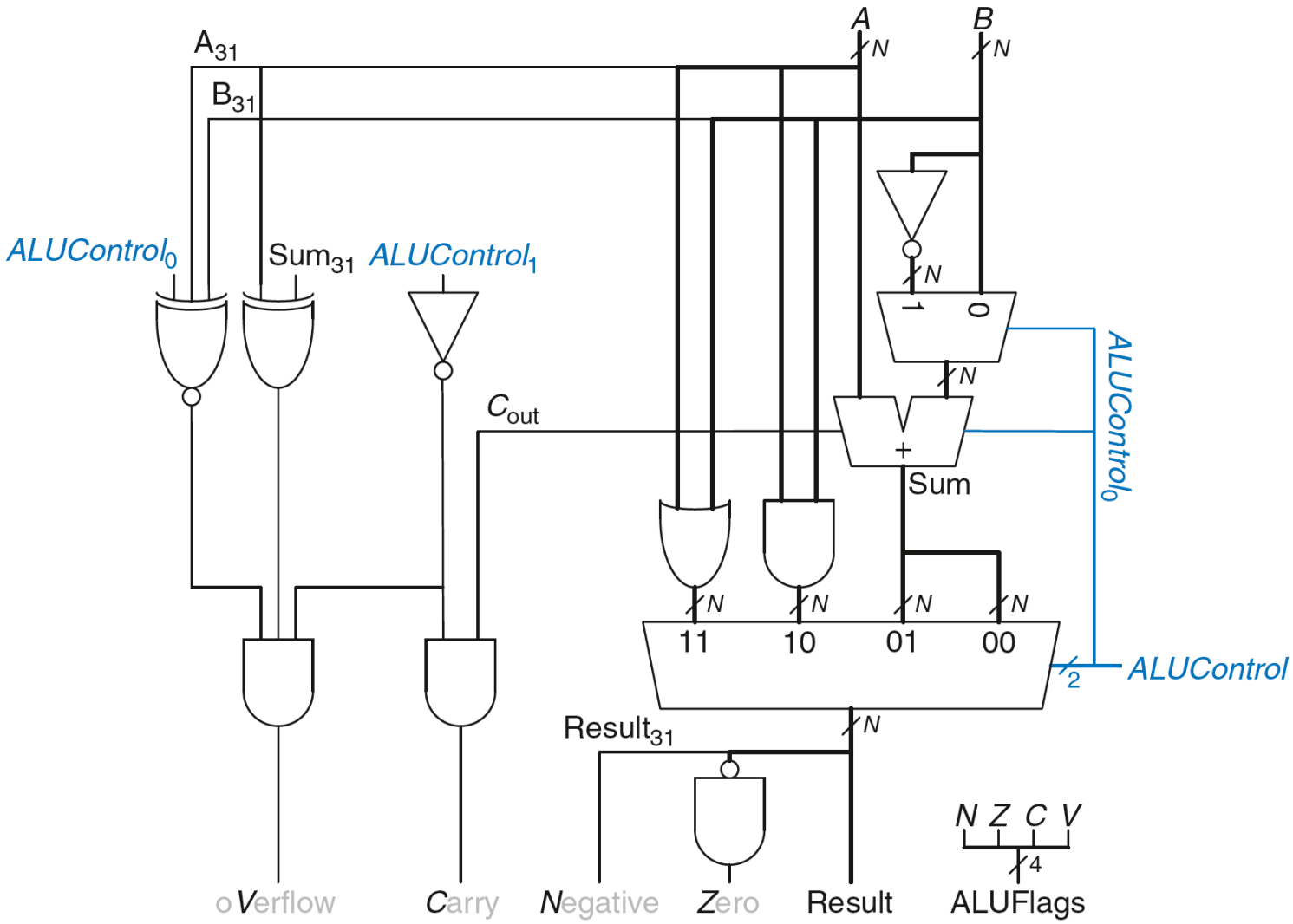
$ALUControl_{1:0}$	Function
00	ADD
01	SUB
10	AND
11	OR



N, Z, C, V Flags

- $N = 1$ if *Result* is negative, i.e., $N = Result_{31}$
- $Z = 1$ if all of the bits of *Result* are 0
- $C = 1$ if
  - C_{out} of adder is 1 **AND**
  - ALU is performing ADD/SUB ($ALUControl_1 = 0$)
- $V = 1$ if
  - ALU is performing ADD/SUB ($ALUControl_1 = 0$) **AND**
  - A and Sum have opposite signs **AND**
  - If ADD ($ALUControl_0 = 0$) : A and B have same sign
  - If SUB ($ALUControl_0 = 1$) : A and B have different signs

ALU with Flags

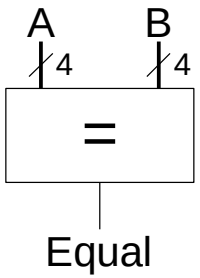


To Do: Make sense of how each condition code is related to NZCV flags

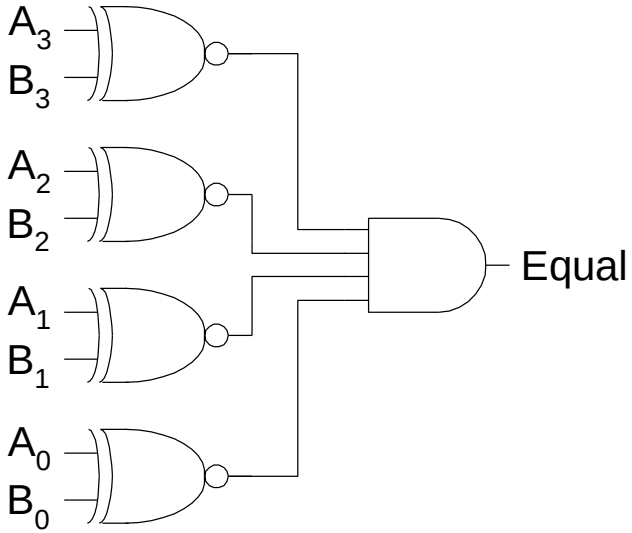
Comparator: Equality

- Comparison in ARM is done using subtraction (SUBS/CMP)
- ALU does not contain dedicated comparison hardware
- However, equality check might still be required for other purposes
  - Pipeline hazard detection
  - Branch prediction

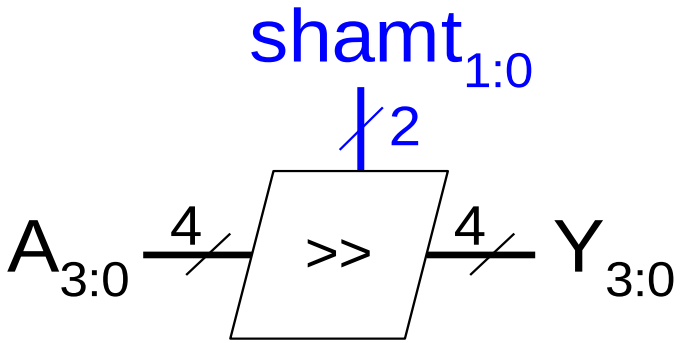
Symbol



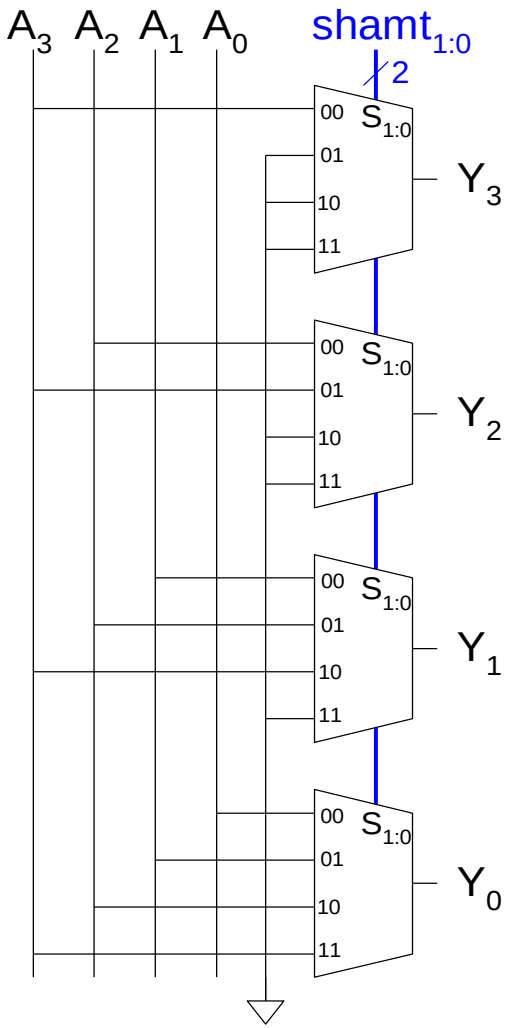
Implementation



Shifter

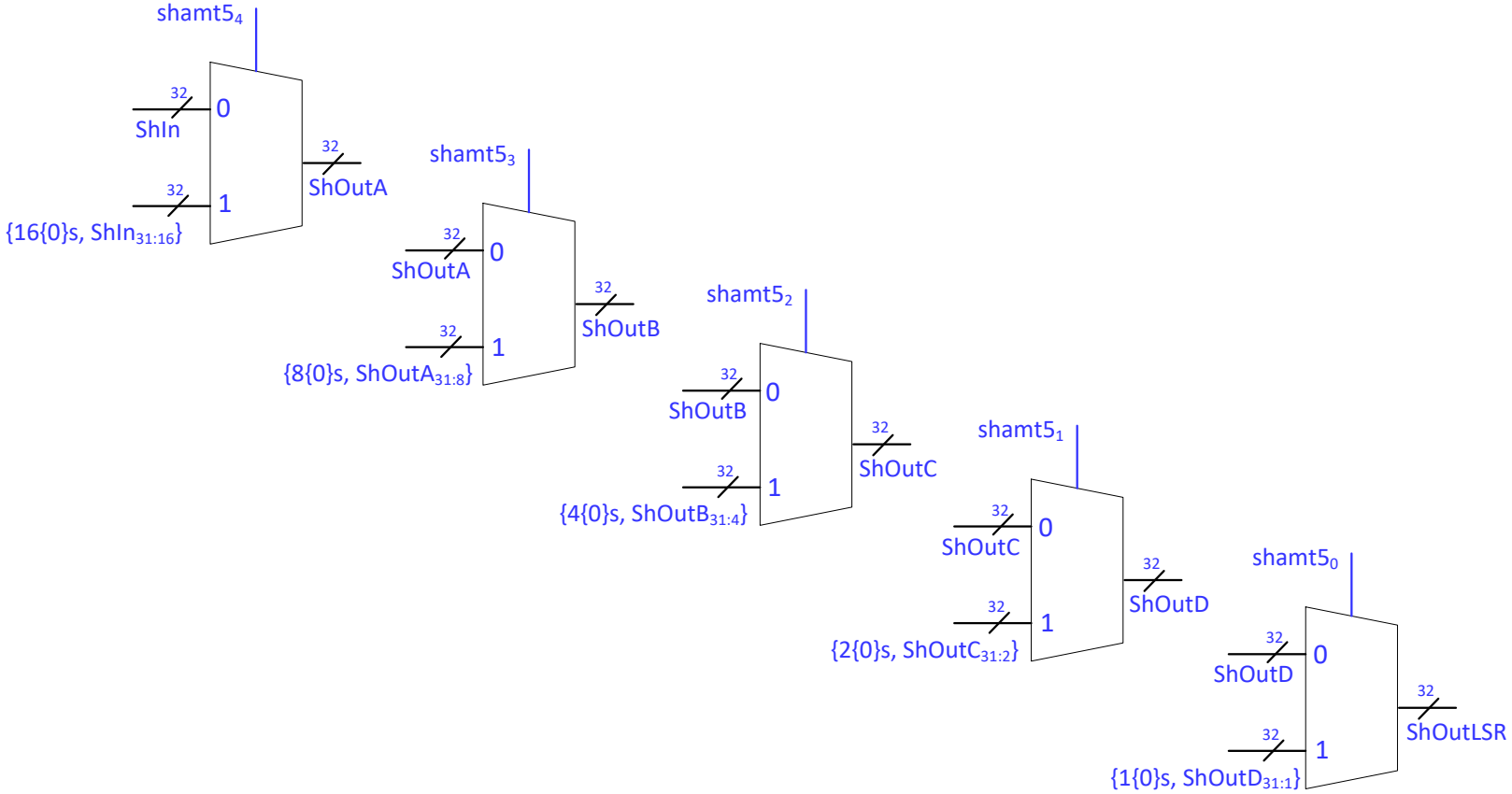


- High hardware usage for 32 bit shifting
  - 32 x 32-1 mux
- Such a shifter which can combinationally shift an input by an arbitrary amount is called a barrel shifter



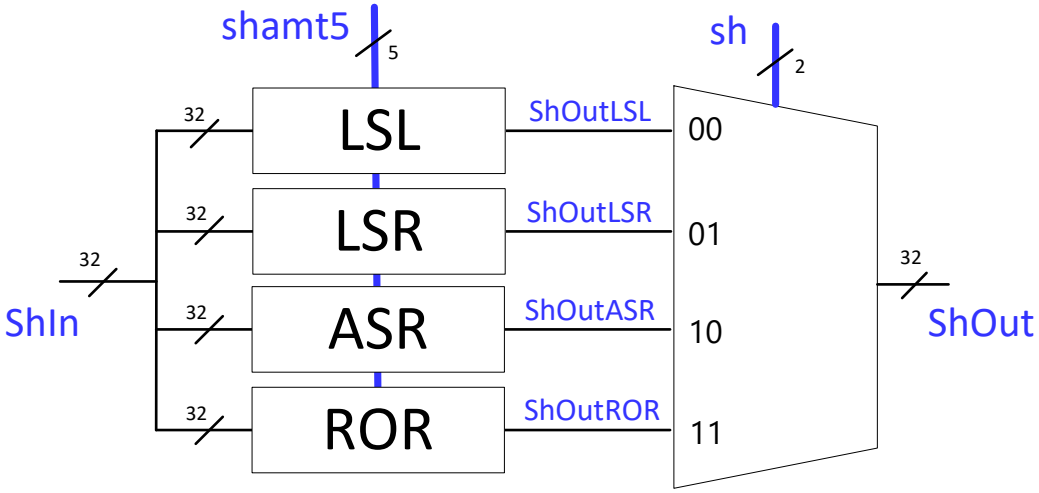
Shifter

- Hardware-efficient shifter
  - 5 x 32 x 2-1 mux



Shifter

- LSL, ASR and ROR can also be implemented following a similar logic
- The 4 different shifts are combined using a multiplexer, controlled by *sh*
- More efficient shifter designs available!



Multipliers

- Partial products formed by multiplying a single digit of the multiplier with multiplicand
- Shifted partial products summed to form result

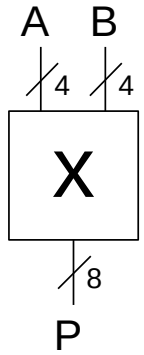
Decimal		**Binary**
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
9660		0101
		+ 0000
	result	0100011

230 x 42 = 9660

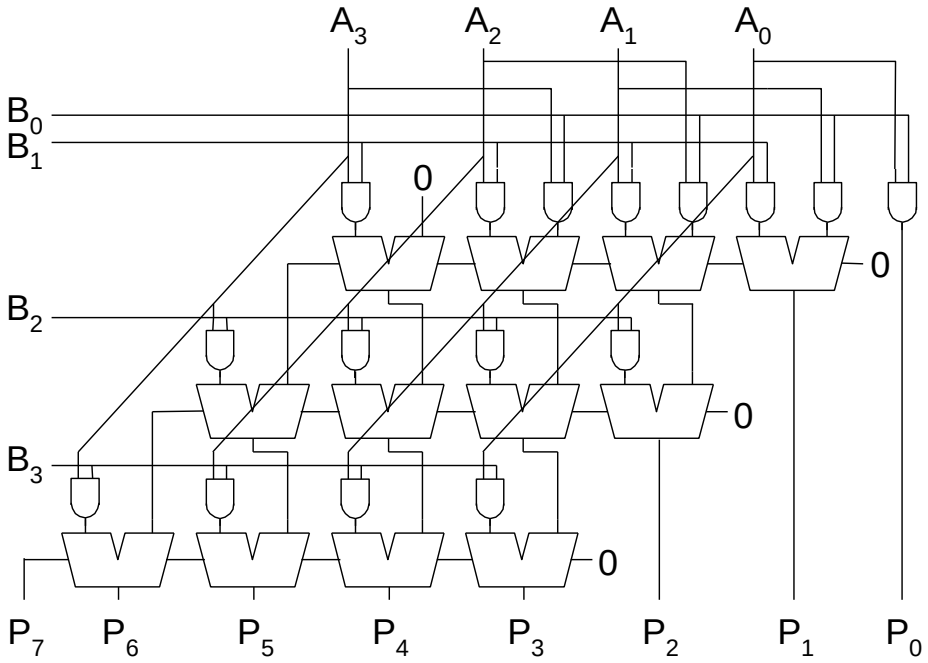
5 x 7 = 35

Multiplier (4 x 4)

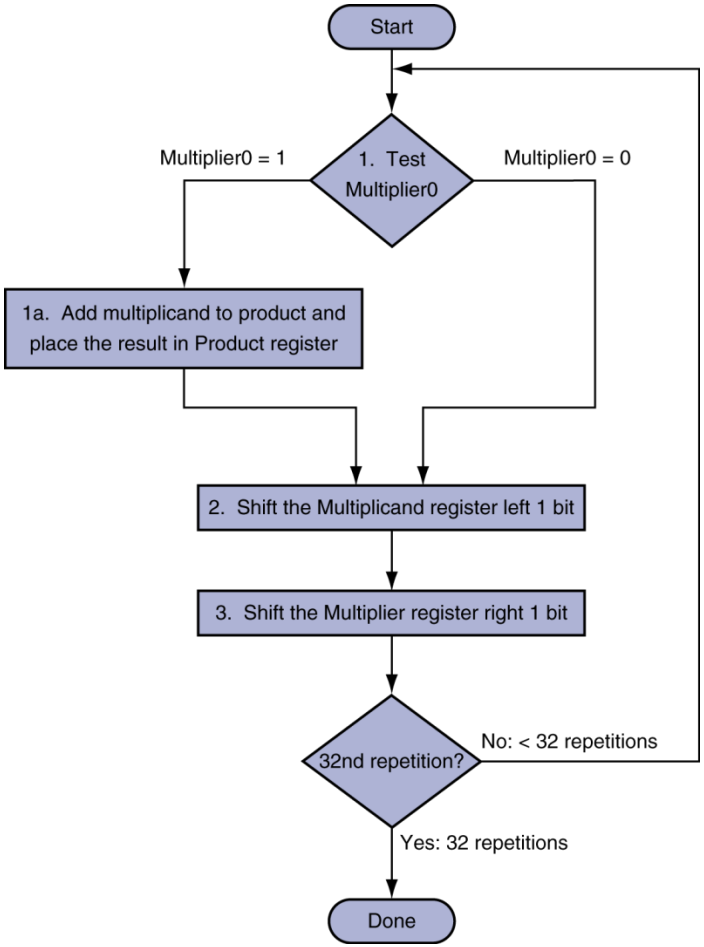
- Array multiplier
  - Pro : combinational (single cycle)
  - Cons: Very high hardware usage, delay (likely in the critical path)

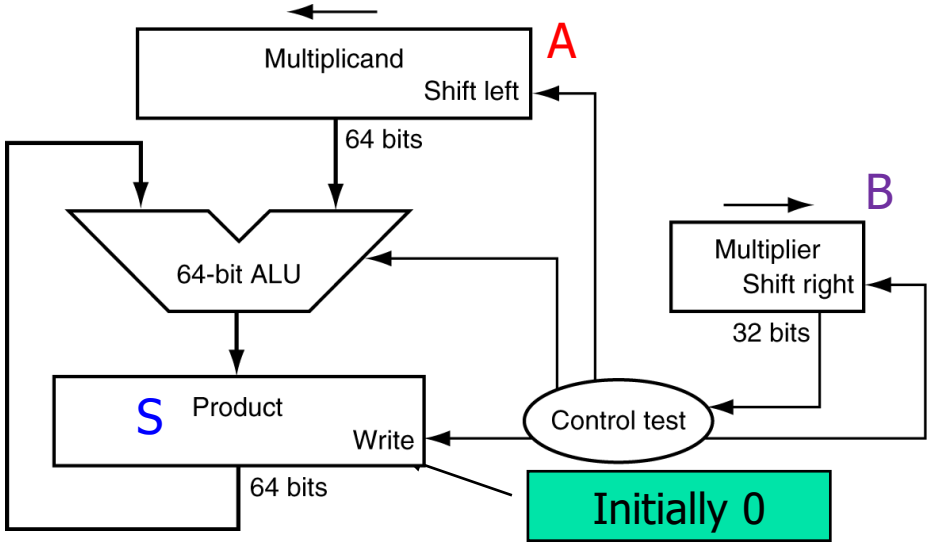


$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



Sequential Multiplier





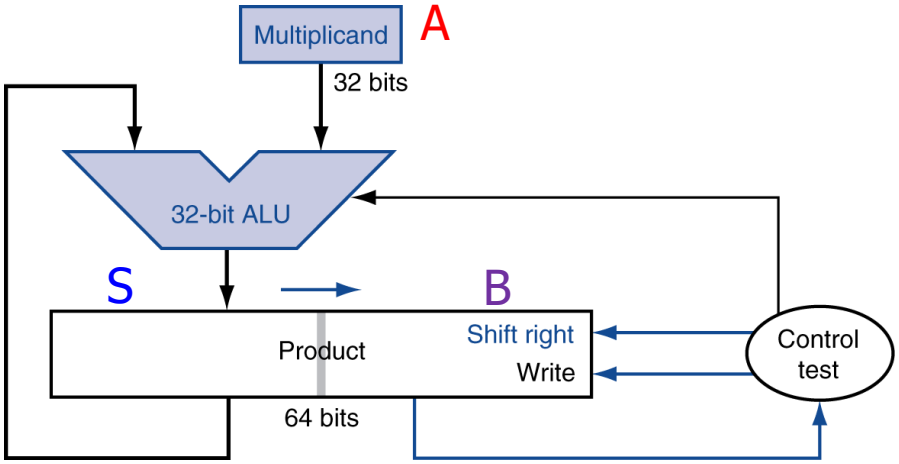
Let S=000000, A=011, B=101 (initial values)

#1: B_0(orig B_0)=1
add: S = S + A = 000000+000011 = 000011
A=A<<1; Thus A = 000110
B=B>>1; Thus B = 010

#2: B_0(orig B_1)=0; no addition required;
A=A<<1; Thus A = 001100
B=B>>1; Thus B = 001

#3: B_0(orig B_2)=1
add: S = S + A = 000011+001100 = 001111

Improved Sequential Multiplier



Idea is to always align A to the "high side" (3 most significant bits of S) and then perform addition; Shift S to the right and do not shift A

Least significant bits of S populated with B, which keeps getting shifted to the right with S (avoids the need for an extra register for B)

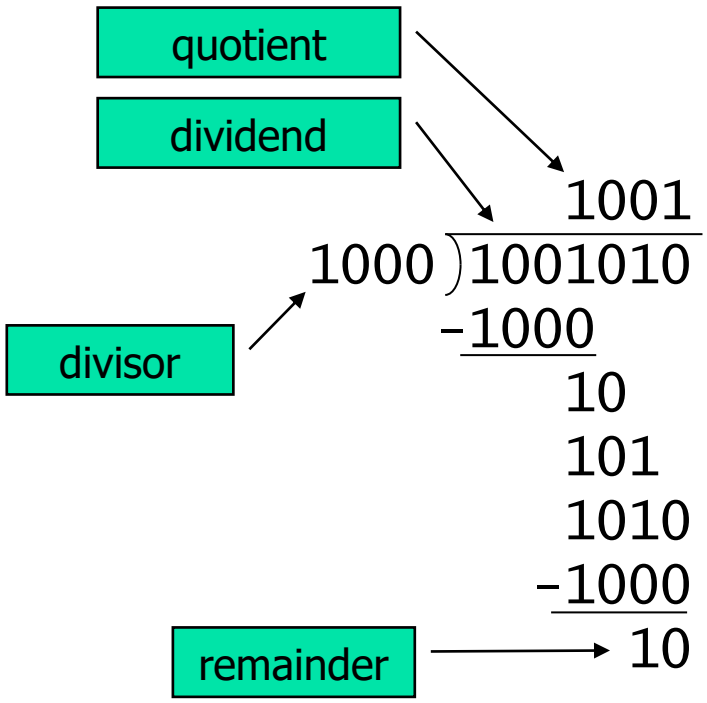
Let S=000101 (000B), A=011 (initial values)

#1: S_0(orig B_0)=1; add: S = S + A000 = 000101 + 011000 = 011101
S=S>>1; Thus S=001110

#2: S_0(orig B_1)=0; no addition required;
S=S>>1; Thus S=000111

#3: S_0(orig B_2)=1; add: S = S + A000 = 000111+ 011000 = 011111;
S=S>>1; Thus S=001111, which is the final result

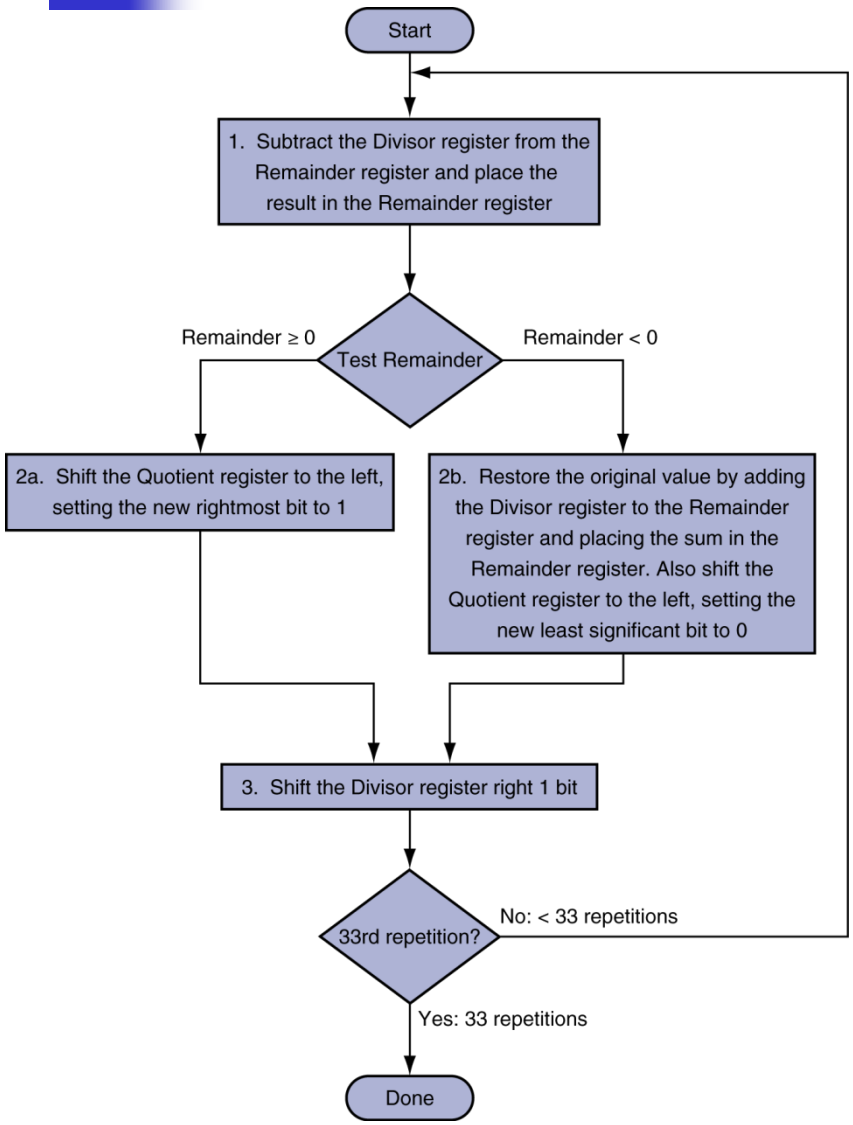
Division

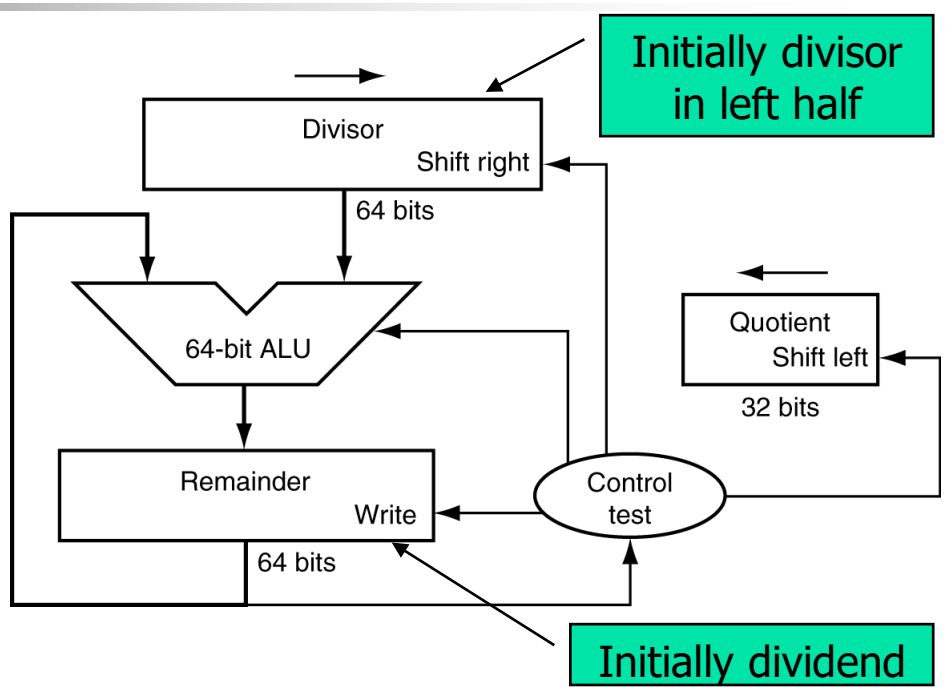


n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back

Division Hardware





- More efficient hardware (similar to that for multiplication) possible
- Better schemes such as non-restoring division exists
  - Left as self-study
- Multiplier and Divider can share most of the hardware

Example

- Steps involved when 0000 0111 is divided by 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Signed Multiplication / Division

```
   11x
   10
 ----
   00
  110
 ----
  110
```

Unsigned interpretation
3 x 2 = 6

Signed interpretation
-1 x -2 = +6

```
  1111x
  1110
 -----
  0000
  1110
  1100
  1000
 -----
  0010
```

Corrected signed multiplication

Need to do sign-extension!

- When multiplying two n-bit numbers, the least significant n-bits are the same irrespective of signed / unsigned multiplication
- For division, signed/unsigned always matter
- There are more efficient signed multiplication techniques – read up!
- One simple trick – convert negatives to positives before multiplication / division, and negate the result if the original operands are of different signs

Numbers with Fractions

- Two common notations
  - Fixed-point: binary point fixed
  - Floating-point: binary point floats to the right of the most significant 1
- Fixed point
  - Binary point is implied
  - The number of integer and fraction bits must be agreed upon beforehand. Notations not so standard
  - Example : 6.75 using 4 integer bits and 4 fraction bits

    01101100

    0110.1100

    $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Signed Fixed Point

- Representation using Sign&Mag or 2's compl.
- Example: Represent -7.510 using 4 integer and 4 fraction bits
  - Sign/magnitude
    - 11111000
  - 2's complement

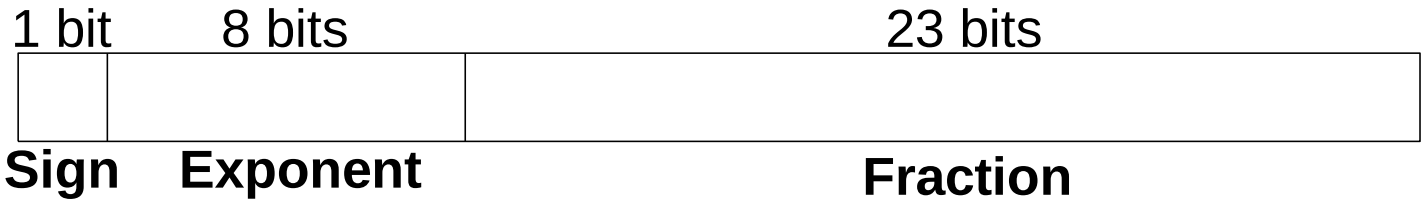
    1. +7.5: 01111000
    2. Invert bits: 10000111
    3. Add 1 to LSB: +1

    10001000
- After multiplication / division, the position of point needs to be adjusted by shifting to the left / right by the number of fraction bits (why?)
- Generally computationally simpler than floating point and hence popular

Floating-Point Numbers

- Binary point 'floats' unlike fixed point
- Scientific notation usually used
- The IEEE 754 floating-point standard
  - a number is written as

$$\pm 1.\text{Fraction} \times 2^{(\text{Exponent}-127)}$$



- The '1' to the left of the point is implicit/hidden
- 127 is the 'bias' so that the Exponent field is always positive (known as excess-127 format)
- Significand is also known as Mantissa and some other terms; may or may not include the hidden bit depending on context

Floating Point Example

Write -58.25_{10} in floating point (IEEE 754)

1. Convert decimal to binary:

   $58.25_{10} = 111010.01_2$

2. Write in binary scientific notation:

   1.1101001×2^5

3. Fill in fields:

   Sign bit: 1 (negative)
   8 exponent bits: (127 + 5) = 132 = 10000100_2
   23 fraction bits: 110 1001 0000 0000 0000 0000

in hexadecimal: 0xC2690000

IEEE 754 : Special Cases

Number	Sign	Exponent	significand
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

- Double-Precision (64-bit)
  - 1 sign bit, 11 exponent bits, 52 fraction bits. Bias = 1023
- Half-Precision (16-bit)
  - 1 sign bit, 5 exponent bits, 10 fraction bits. Bias = 15
  - Popular in machine learning / neural networks
  - Supported by modern GPUs (e.g., Nvidia Pascal and newer) and CPU ISAs/ISA extensions such as ARMv8.1-M, x86 AVX-512 etc.

Floating Point : Rounding

- Overflow: number too large to be represented
- Underflow: number too small to be represented
- Rounding modes: Down, Up, Toward zero, To nearest
- Example: round 1.100101 (1.578125) to only 3 fraction bits
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

Floating Point Addition : Steps

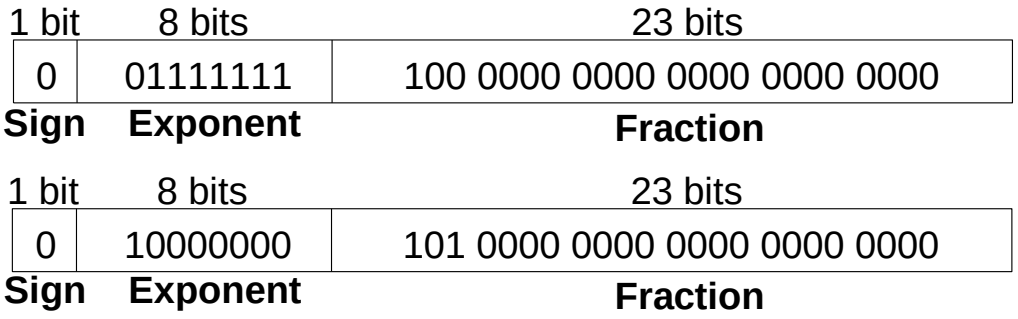
1. Extract exponent and fraction bits
2. Prepend leading 1 to form the significand
3. Compare exponents
4. Shift the smaller significand to the right
5. Add significands
6. Normalize significand and adjust exponent if necessary
7. Round result
8. Assemble exponent and fraction back into floating-point format

Floating Point Addition : Example

- Add the following floating-point numbers

  0x3FC00000, 0x40500000

1. Extract exponent and fraction bits



For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

2. Prepend leading 1 to form significand

N1: 1.1

N2: 1.101

Floating Point Addition : Example

3. Compare exponents

   127 – 128 = -1, so shift N1 right by 1 bit

4. Shift smaller significand if necessary

   shift N1's significand: 1.1 >> 1 = 0.11 ($\times 2^1$)

5. Add significands

$$\begin{array}{rr} & 0.11 \times 2^1 \\ + & 1.101 \times 2^1 \\ \hline & 10.011 \times 2^1 \end{array}$$

Floating Point Addition : Example

6. Normalize significand and adjust exponent if necessary

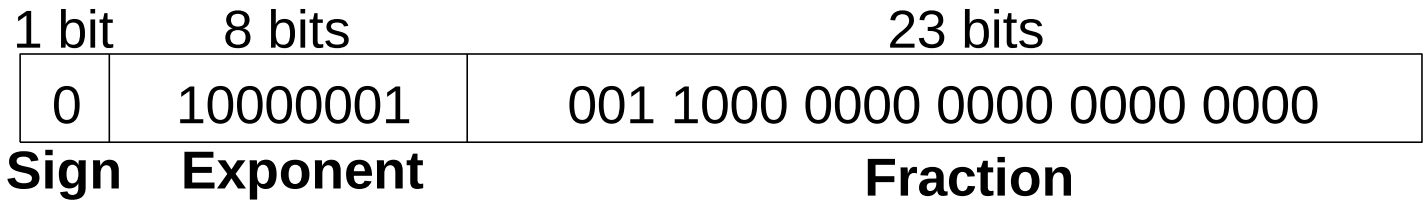
   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. Round result

   No need (fits in 23 bits)

8. Assemble exponent and fraction back into floating-point format

   $S = 0$, $E = 2 + 127 = 129 = 10000001_2$, $F = 001100..$



in hexadecimal: 0x40980000