
Computer Science 141

Computing Hardware

Fall 2009

Harvard University

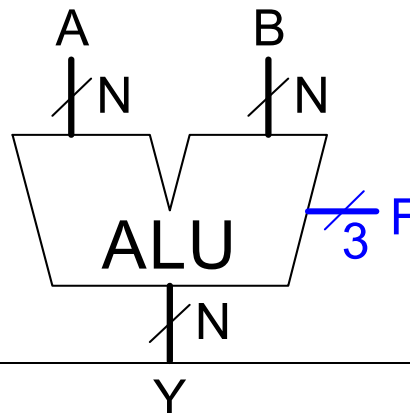
Instructor: Prof. David Brooks

dbrooks@eecs.harvard.edu

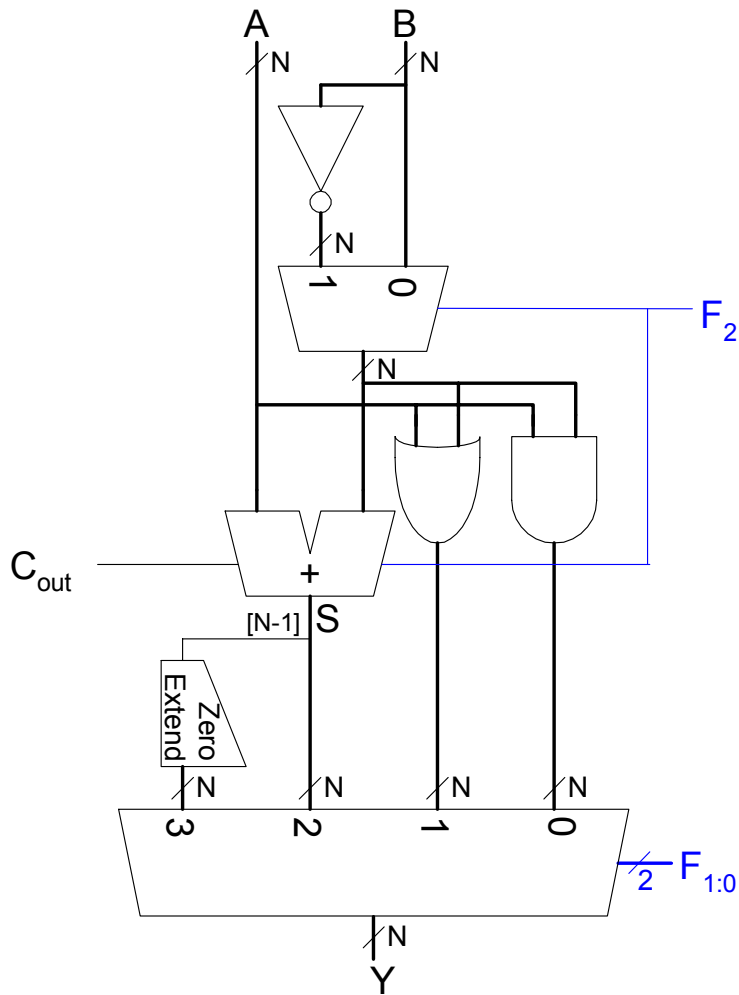
Arithmetic Logic Unit (ALU)

- combinational network that computes general logic and arithmetic operations
- Can be implemented as bit slice (design 1 bit and duplicate)
- MIPS ALU performs
 - $A \text{ AND } B$, $A \text{ OR } B$, $A + B$, $A - B$
 - Set on $A < B$
if $(A < B)$ output 1; else output 0

$F_{2:0}$	Function
000	$A \ \& \ B$
001	$A \ \ B$
010	$A + B$
011	not used
100	$A \ \& \ \sim B$
101	$A \ \ \sim B$
110	$A - B$
111	SLT

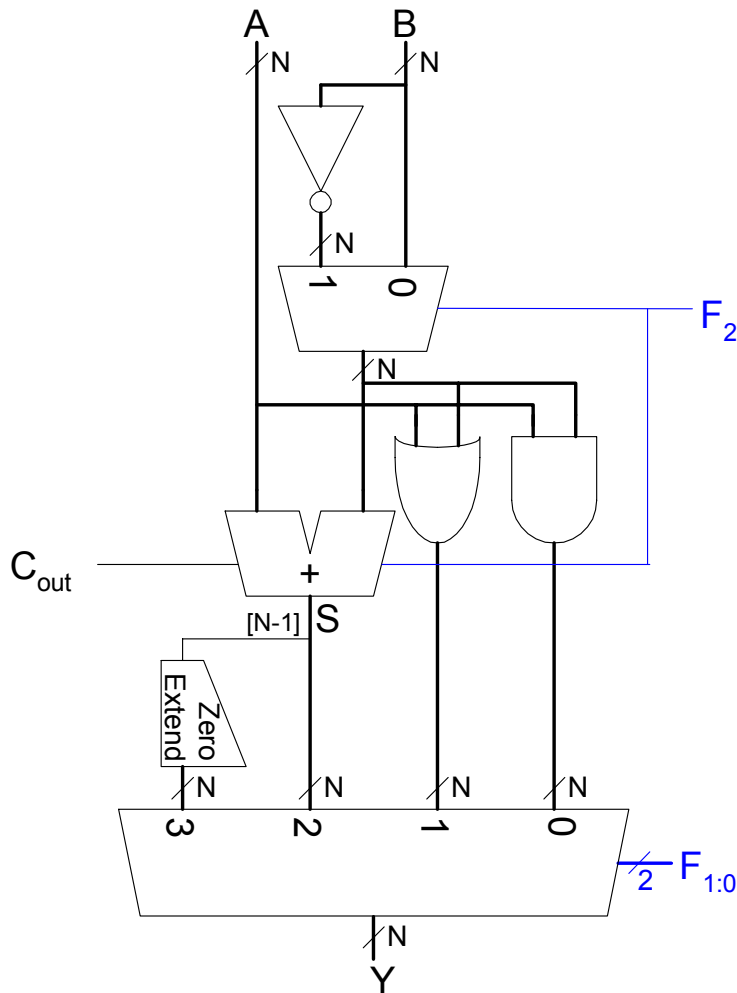


Set Less Than (SLT) Example



- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.

Set Less Than (SLT) Example

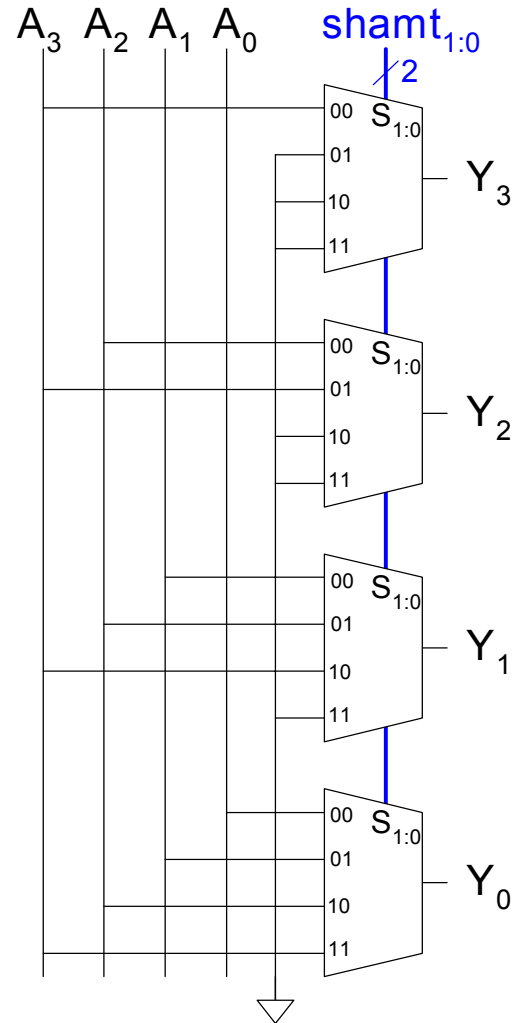
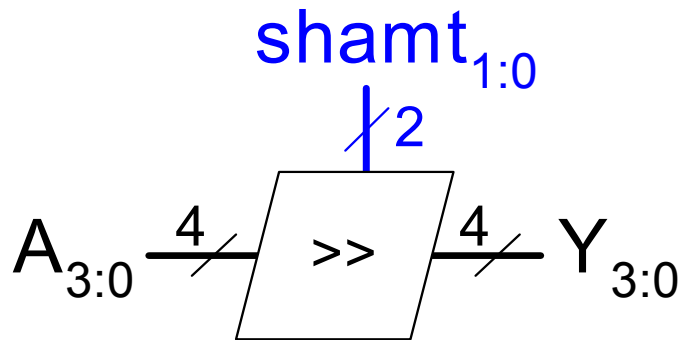


- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.
 - A is less than B , so we expect Y to be the 32-bit representation of 1 (0x00000001).
 - For SLT, $F_{2:0} = 111$.
 - $F_2 = 1$ configures the adder unit as a subtractor. So $25 - 32 = -7$.
 - The two's complement representation of -7 has a 1 in the most significant bit, so $S_{31} = 1$.
 - With $F_{1:0} = 11$, the final multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
 - Ex: 11001 >> 2 =
 - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
 - Ex: 11001 >>> 2 =
 - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
 - Ex: 11001 ROR 2 =
 - Ex: 11001 ROL 2 =

Shifter Design



Shifters as Multipliers and Dividers

- A left shift by N bits multiplies a number by 2^N
 - Ex: $00001 \lll 2 = 00100$ ($1 \times 2^2 = 4$)
 - Ex: $11101 \lll 2 = 10100$ ($-3 \times 2^2 = -12$)
- The arithmetic right shift by N divides a number by 2^N
 - Ex: $01000 \ggg 2 = 00010$ ($8 \div 2^2 = 2$)
 - Ex: $10000 \ggg 2 = 11100$ ($-16 \div 2^2 = -4$)

Floating-Point Numbers

- The binary point floats to the right of the most significant 1.
- Similar to decimal scientific notation.
- For example, write 273_{10} in scientific notation:

$$273 = 2.73 \times 10^2$$

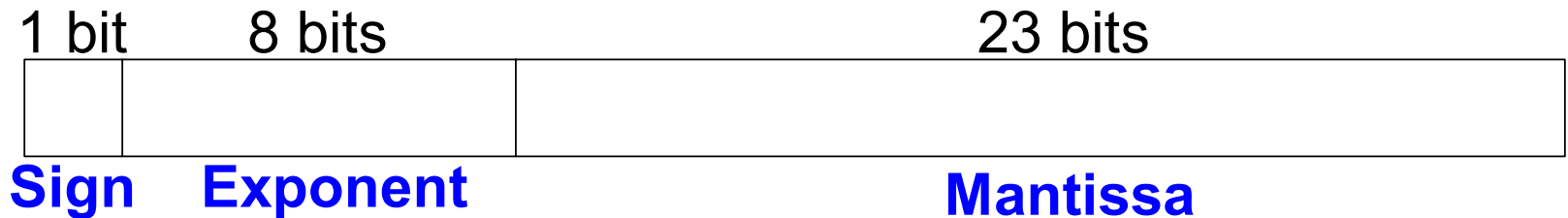
- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

Where,

- **M** = mantissa
- **B** = base
- **E** = exponent
- In the example, $M = 2.73$, $B = 10$, and $E = 2$

Floating-Point Numbers



- **Example:** represent the value 228_{10} using a 32-bit floating point representation

We show three versions - final version is called the **IEEE 754 floating-point standard**

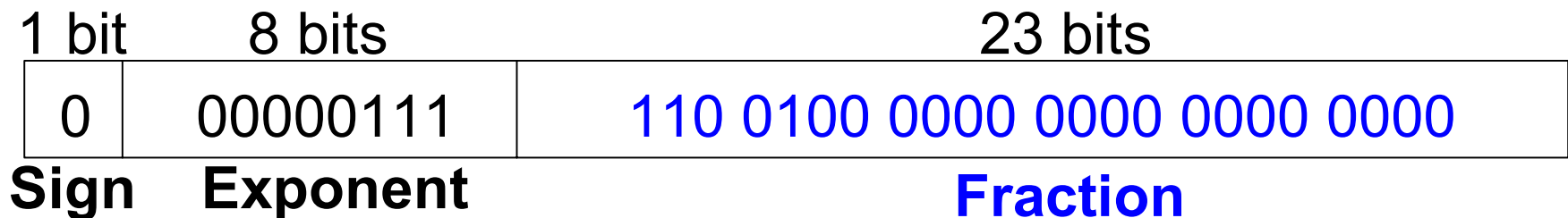
Floating-Point Representation 1

- Convert the decimal number to binary:
 - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Fill in each field of the 32-bit number:
 - The sign bit is positive (0)
 - The 8 exponent bits represent the value 7
 - The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	Exponent	Mantissa

Floating-Point Representation 2

- First bit of the mantissa is always 1:
 - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Thus, storing the most significant 1, also called the *implicit leading 1*, is redundant information.
- Instead, store just the fraction bits in the 23-bit field. The leading 1 is implied.



Floating-Point Representation 3

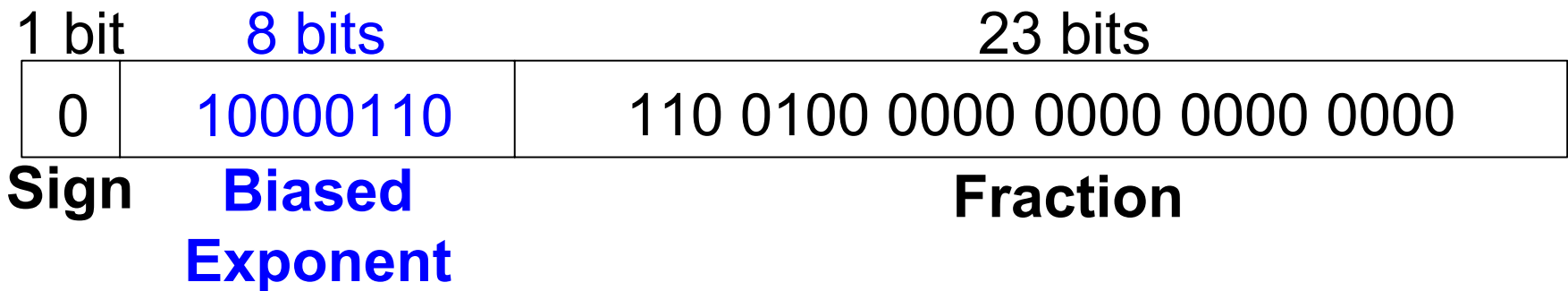
- *Biased exponent*: bias = 127 (01111111_2)

- Biased exponent = bias + exponent

- Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$

- The **IEEE 754 32-bit floating-point representation** of 228_{10}



Multiplication

multiplicand	1101	(13)
multiplier	x <u>1011</u>	(11)
	1101	
	1101	
	0000	
	<u>1101</u>	
product	10001111	(143)

- algorithm (for unsigned integers):
 - check lsb of multiplier
 - if 1, add multiplicand to partial product
 - shift partial product and multiplier right 1 bit
 - repeat n times
- overflow much worse:
 - n-bit number X m-bit number can produce (n+m)-bit number

Multiplication

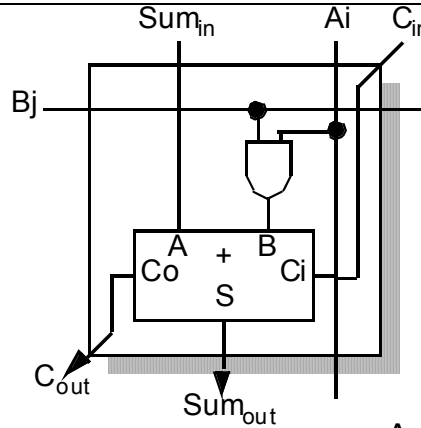
- partial product accumulation
 - combinational circuit for multiply

				A3	A2	A1	A0
				B3	B2	B1	B0
				<hr/>			
				A3•B0	A2•B0	A1•B0	A0•B0
			A3•B1	A2•B1	A1•B1	A0•B1	
		A3•B2	A2•B2	A1•B2	A0•B2		
	A3•B3	A2•B3	A1•B3	A0•B3			
	<hr/>						
S7	S6	S5	S4	S3	S2	S1	S0

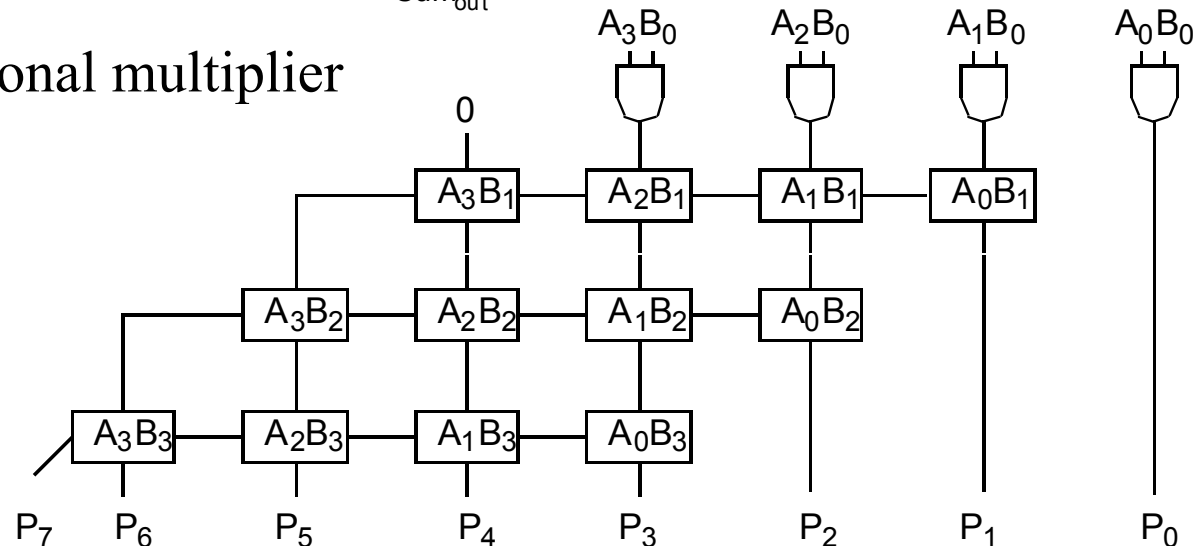
- requires:
 - AND gates for individual bit partial products
 - full adder (FA) circuits for column sums

Partial product accumulation implementation

- basic building block



- 4x4 combinational multiplier



uses traditional ripple-carry adders (slow!)

Faster Partial Product Accumulation

- Fundamental problem: Trying to add m numbers, n bits wide
 - Requires $m - 1$ additions, total gate delay of $O(m \lg n)$
 - Can form trees, reduce to $O(\lg m \lg n)$
- How can we do better?
 - Carry Save Adders: $x + y + z = c + s$
 - And, compute this in constant time (not dependent on carry chains)

Carry Save Addition (Decimal)

carry:		1	1	2	1	
X:		1	2	3	4	5
Y:		3	8	1	7	2
Z:	+	2	0	5	8	7
<hr/>						
Sum:		7	1	1	0	4

Each s_i computed independently

X:		1	2	3	4	5
Y:		3	8	1	7	2
Z:	+	2	0	5	8	7
<hr/>						
s:		6	0	9	9	4

Each c_i computed independently

X:		1	2	3	4	5
Y:		3	8	1	7	2
Z:	+	2	0	5	8	7
<hr/>						
c:		1	0	1	1	

s:		6	0	9	9	4
c:	+	1	0	1	1	0
<hr/>						
sum:		7	1	1	0	4

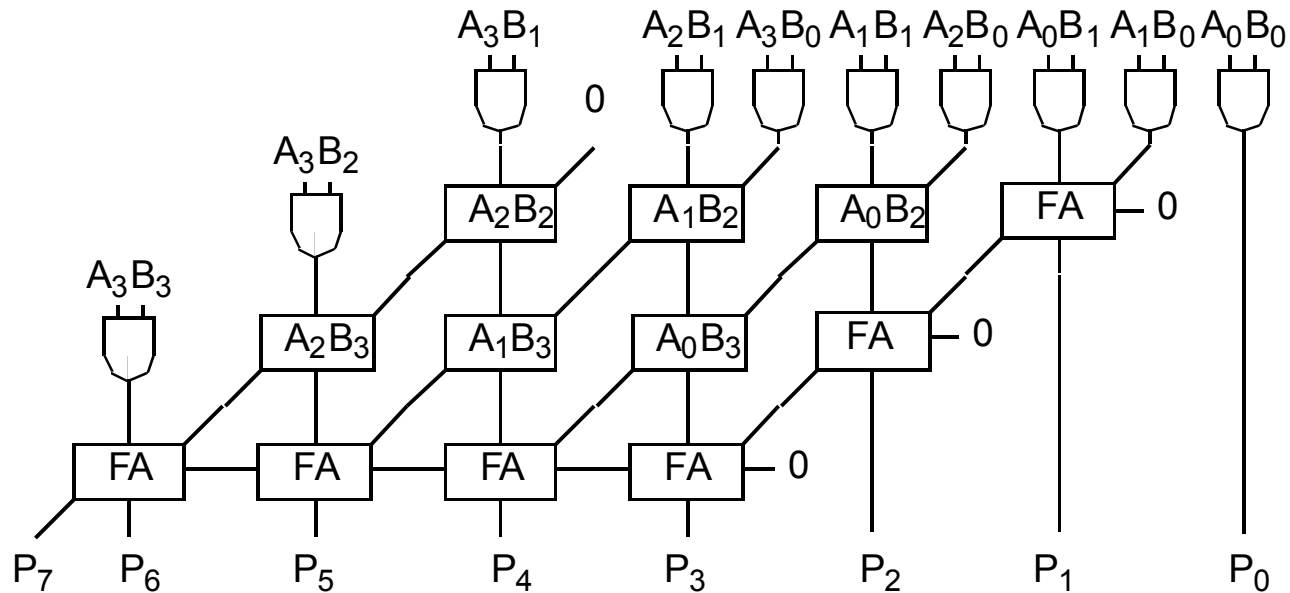
Example from Gabe Loh,
GaTech

Carry Save Addition (Binary)

$$\begin{array}{r} X: \quad \quad 1 \ 0 \ 0 \ 1 \ 1 \\ Y: \quad \quad 1 \ 1 \ 0 \ 0 \ 1 \\ Z: \ + \quad \quad 0 \ 1 \ 0 \ 1 \ 1 \\ \hline s: \\ c: \ + \\ \hline \text{sum:} \end{array}$$

Building fast multipliers with CSAs

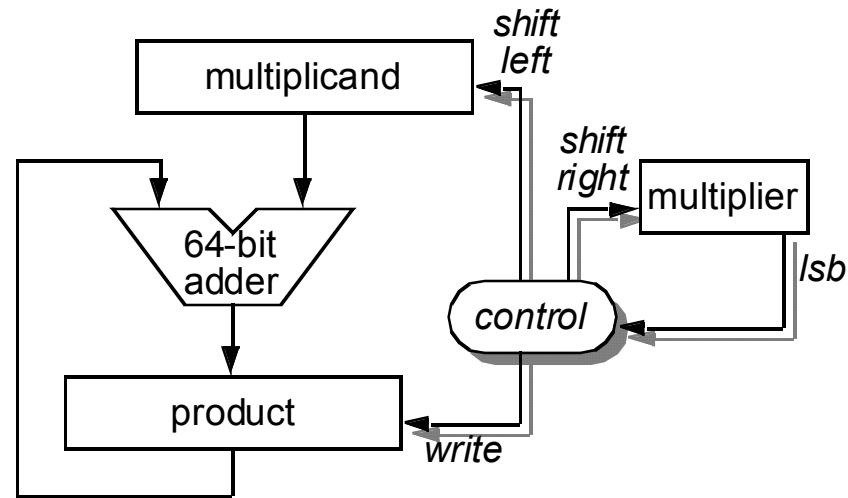
- true combinational multipliers use carry-save adders: regular layout, much faster



- idea behind using carry-save adder
 - FA adds 3 inputs and produces 2 outputs
 - look at outputs as 2 independent sums: S' and C'
 - last step adds S' and C' using a normal adder
 - benefit: carry propagation delayed till end of sum
 - benefit increases as the number of numbers to add increases

iterative multiplication implementation

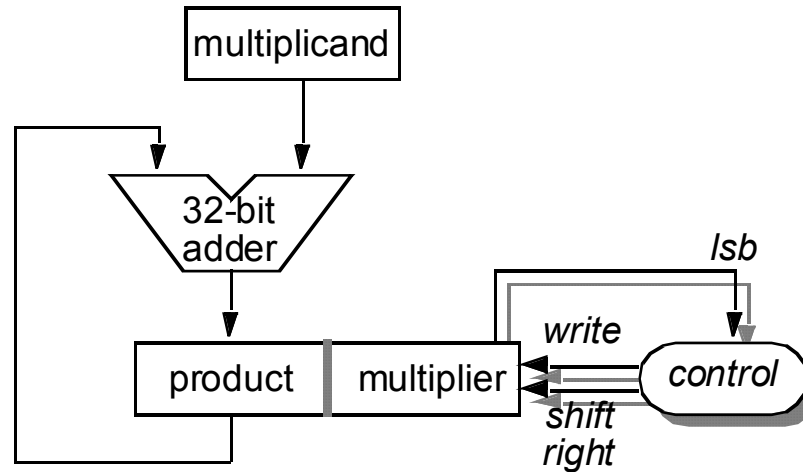
- iterative multiplication implementation (for 32-bit numbers)



- relies on registers to keep state \Rightarrow sequential circuit!
 - slower but less hardware than combinational multiplier
 - requires clocks and control logic
-

Multiplication and Division

- hardware-optimized iterative multiplier implementation



- division
 - can implement with very similar hardware
 - tricky parts:
 1. division by 0 is special case operation
 2. different check—did subtraction leave positive remainder?
-