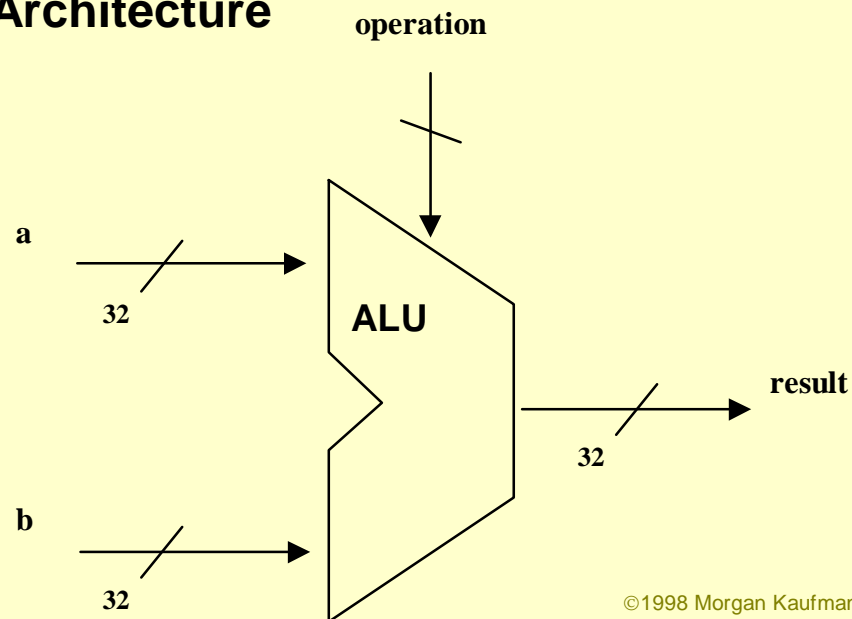


Arithmetic

- **Where we've been:**
 - Performance (seconds, cycles, instructions)
 - Abstractions:
 - Instruction Set Architecture**
 - Assembly Language and Machine Language**
- **What's up ahead:**
 - **Implementing the Architecture**



Numbers

- **Bits are just bits (no inherent meaning)**
 - conventions define relationship between bits and numbers
- **Binary numbers (base 2)**
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
decimal: $0 \dots 2^n - 1$
- **Of course it gets more complicated:**
 - numbers are finite (overflow)
 - fractions and real numbers
 - negative numbers
 - e.g., no MIPS `subi` instruction; `addi` can add a negative number)
- **How do we represent negative numbers?**
i.e., which bit patterns will represent which numbers?

Possible Representations

- | Sign Magnitude: | One's Complement | Two's Complement |
|------------------------|-------------------------|-------------------------|
| 000 = +0 | 000 = +0 | 000 = +0 |
| 001 = +1 | 001 = +1 | 001 = +1 |
| 010 = +2 | 010 = +2 | 010 = +2 |
| 011 = +3 | 011 = +3 | 011 = +3 |
| 100 = -0 | 100 = -3 | 100 = -4 |
| 101 = -1 | 101 = -2 | 101 = -3 |
| 110 = -2 | 110 = -1 | 110 = -2 |
| 111 = -3 | 111 = -0 | 111 = -1 |
- Issues: balance, number of zeros, ease of operations**
- Which one is best? Why?**

MIPS

- **32 bit signed numbers:**

0000	0000	0000	0000	0000	0000	0000	0000	$= 0_{\text{ten}}$	
0000	0000	0000	0000	0000	0000	0000	0001	$= + 1_{\text{ten}}$	
0000	0000	0000	0000	0000	0000	0000	0010	$= + 2_{\text{ten}}$	
...									
0111	1111	1111	1111	1111	1111	1111	1110	$= + 2,147,483,646_{\text{ten}}$	/ <i>maxint</i>
0111	1111	1111	1111	1111	1111	1111	1111	$= + 2,147,483,647_{\text{ten}}$	
1000	0000	0000	0000	0000	0000	0000	0000	$= - 2,147,483,648_{\text{ten}}$	\ <i>minint</i>
1000	0000	0000	0000	0000	0000	0000	0001	$= - 2,147,483,647_{\text{ten}}$	
1000	0000	0000	0000	0000	0000	0000	0010	$= - 2,147,483,646_{\text{ten}}$	
...									
1111	1111	1111	1111	1111	1111	1111	1101	$= - 3_{\text{ten}}$	
1111	1111	1111	1111	1111	1111	1111	1110	$= - 2_{\text{ten}}$	
1111	1111	1111	1111	1111	1111	1111	1111	$= - 1_{\text{ten}}$	

Two's Complement Operations

- **Negating a two's complement number: invert all bits and add 1**
 - remember: “negate” and “invert” are quite different!
- **Converting n bit numbers into numbers with more than n bits:**
 - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
 - copy the most significant bit (the sign bit) into the other bits
 - 0010 -> 0000 0010
 - 1010 -> 1111 1010
 - "sign extension" (lbu vs. lb)

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

$$\begin{array}{r} 0111 \\ + 0110 \\ \hline \end{array}$$

$$\begin{array}{r} 0111 \\ - 0110 \\ \hline \end{array}$$

$$\begin{array}{r} 0110 \\ - 0101 \\ \hline \end{array}$$

- Two's complement operations easy
 - subtraction using addition of negative numbers

$$\begin{array}{r} 0111 \\ + 1010 \\ \hline \end{array}$$

- **Overflow** (result too large for finite computer word):
 - e.g., adding two n-bit numbers does not yield an n-bit number

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

*note that overflow term is somewhat misleading,
it does not mean a carry “overflowed”*

Detecting Overflow

- **No overflow when adding a positive and a negative number**
- **No overflow when signs are the same for subtraction**
- **Overflow occurs when the value affects the sign:**
 - **overflow when adding two positives yields a negative**
 - **or, adding two negatives gives a positive**
 - **or, subtract a negative from a positive and get a negative**
 - **or, subtract a positive from a negative and get a positive**
- **Consider the operations $A + B$, and $A - B$**
 - **Can overflow occur if B is 0 ?**
 - **Can overflow occur if A is 0 ?**

Effects of Overflow

- An exception (interrupt) occurs
 - Control jumps to predefined address for exception
 - Interrupted address is saved for possible resumption
- Details based on software system / language
 - example: flight control vs. homework assignment
- Don't always want to detect overflow
 - new MIPS instructions: `addu`, `addiu`, `subu`

note: addiu still sign-extends!

note: sltu, sltiu for unsigned comparisons

Review: Boolean Algebra & Gates

- **Problem: Consider a logic function with three inputs: A, B, and C.**

Output D is true if at least one input is true

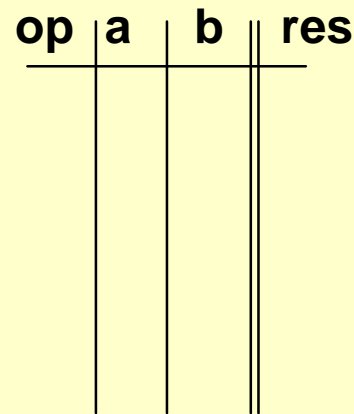
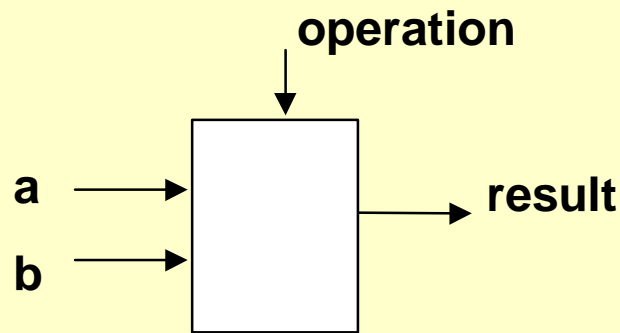
Output E is true if exactly two inputs are true

Output F is true only if all three inputs are true

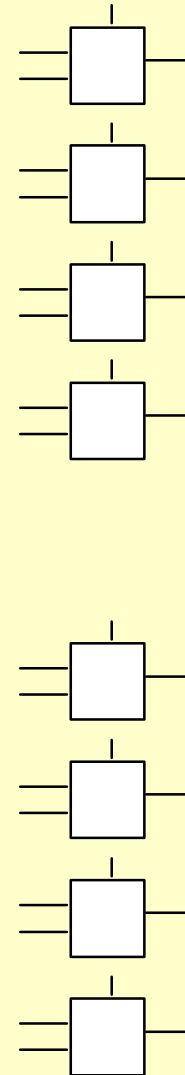
- **Show the truth table for these three functions.**
- **Show the Boolean equations for these three functions.**
- **Show an implementation consisting of inverters, AND, and OR gates.**

An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
 - we'll just build a 1 bit ALU, and use 32 of them

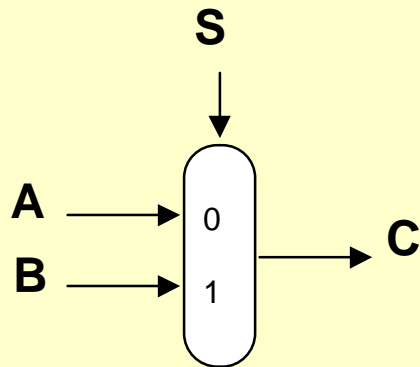


- Possible Implementation (sum-of-products):



Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

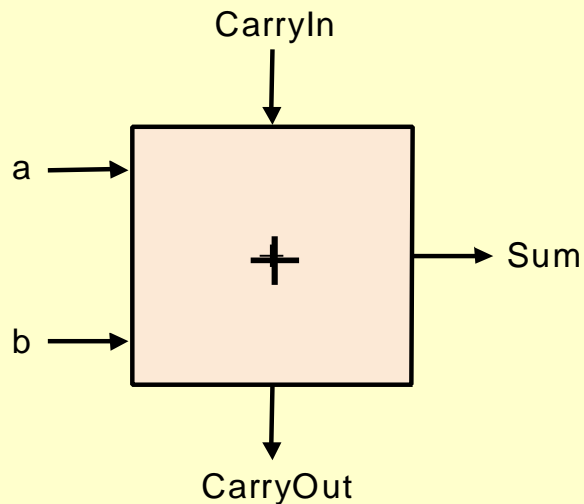


*note: we call this a 2-input mux
even though it has 3 inputs!*

- Lets build our ALU using a MUX:

Different Implementations

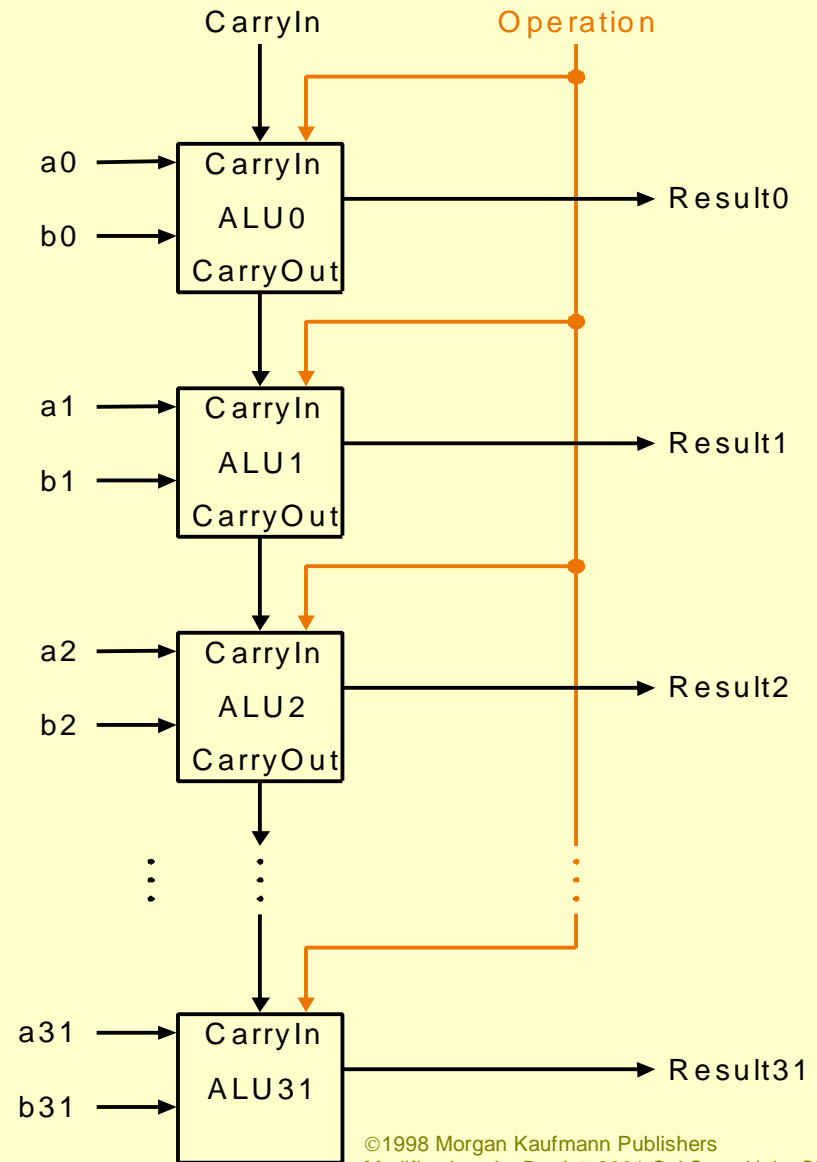
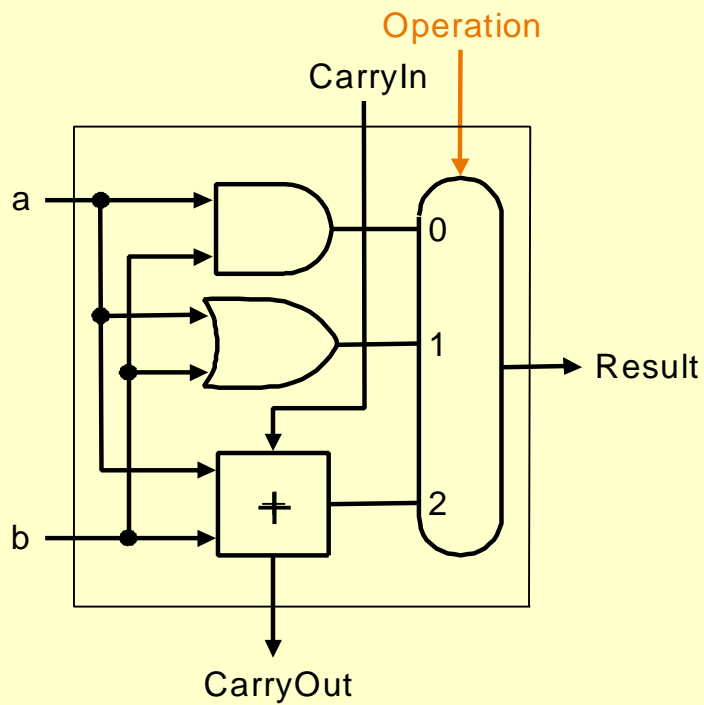
- Not easy to decide the “best” way to build something
 - Don't want too many inputs to a single gate
 - Don't want to have to go through too many gates
 - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



$$c_{out} = a b + a c_{in} + b c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

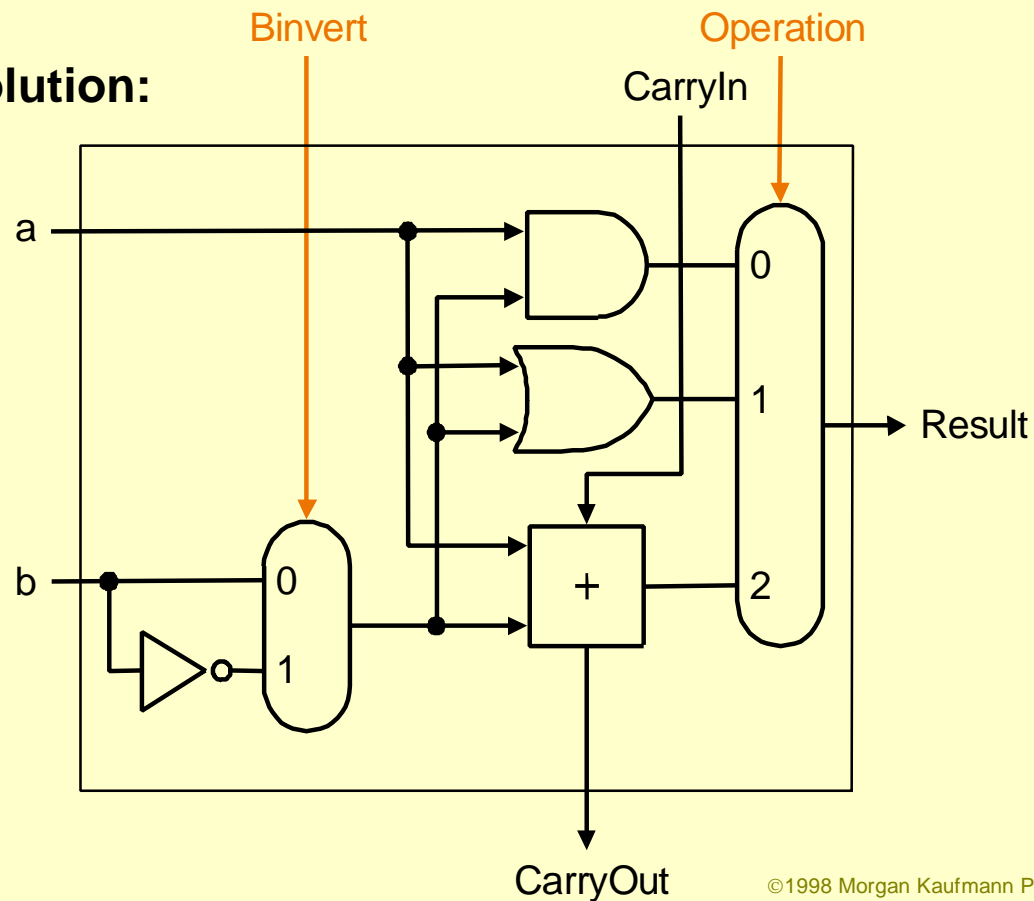
Building a 32 bit ALU



What about subtraction (a - b) ?

- Two's complement approach: just negate b and add.
- How do we negate?

- A very clever solution:

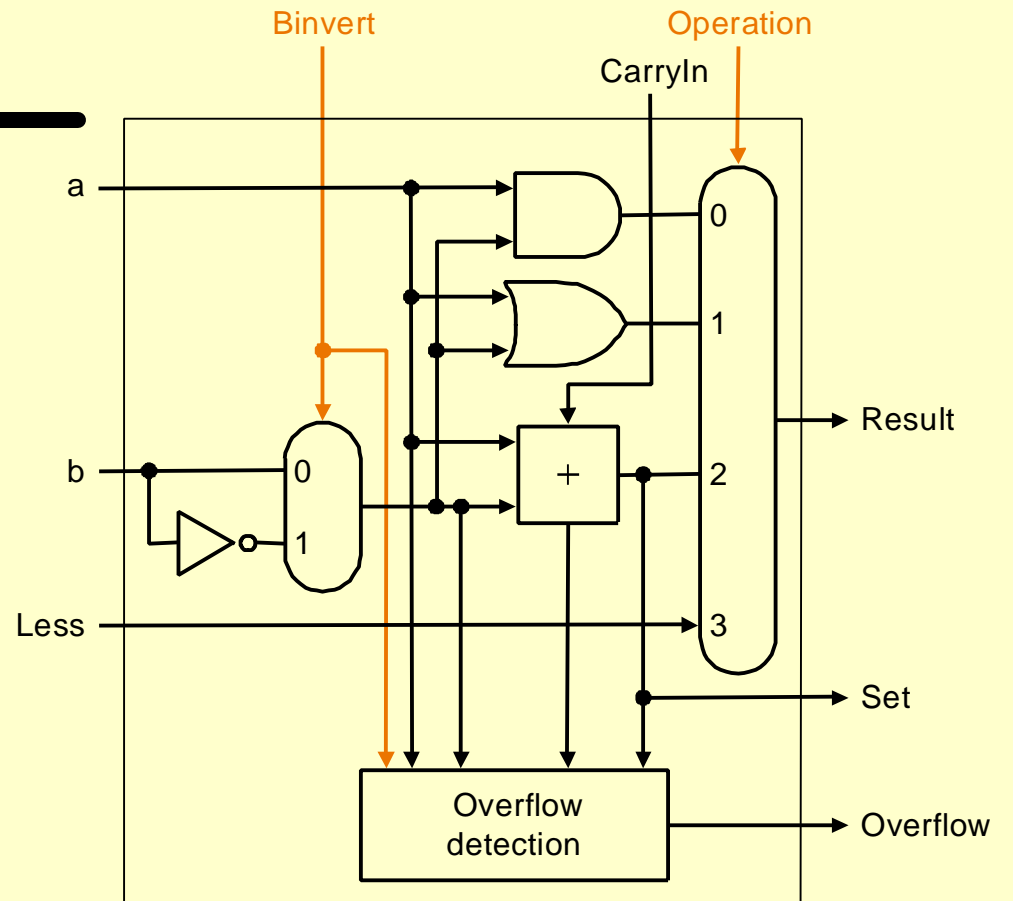
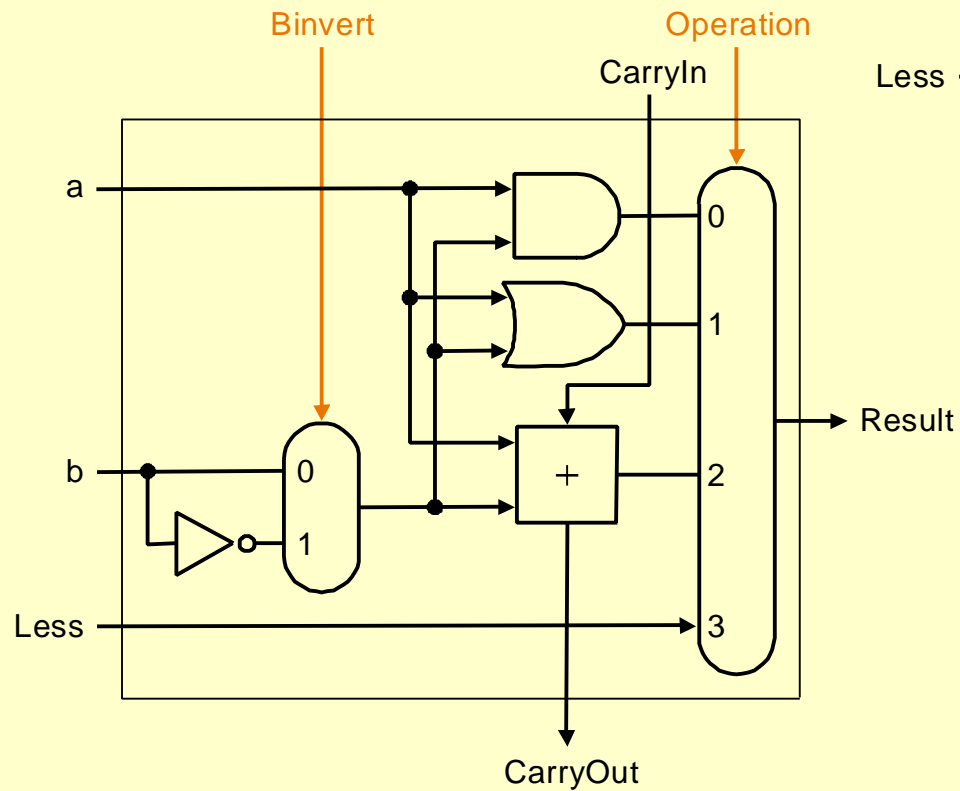


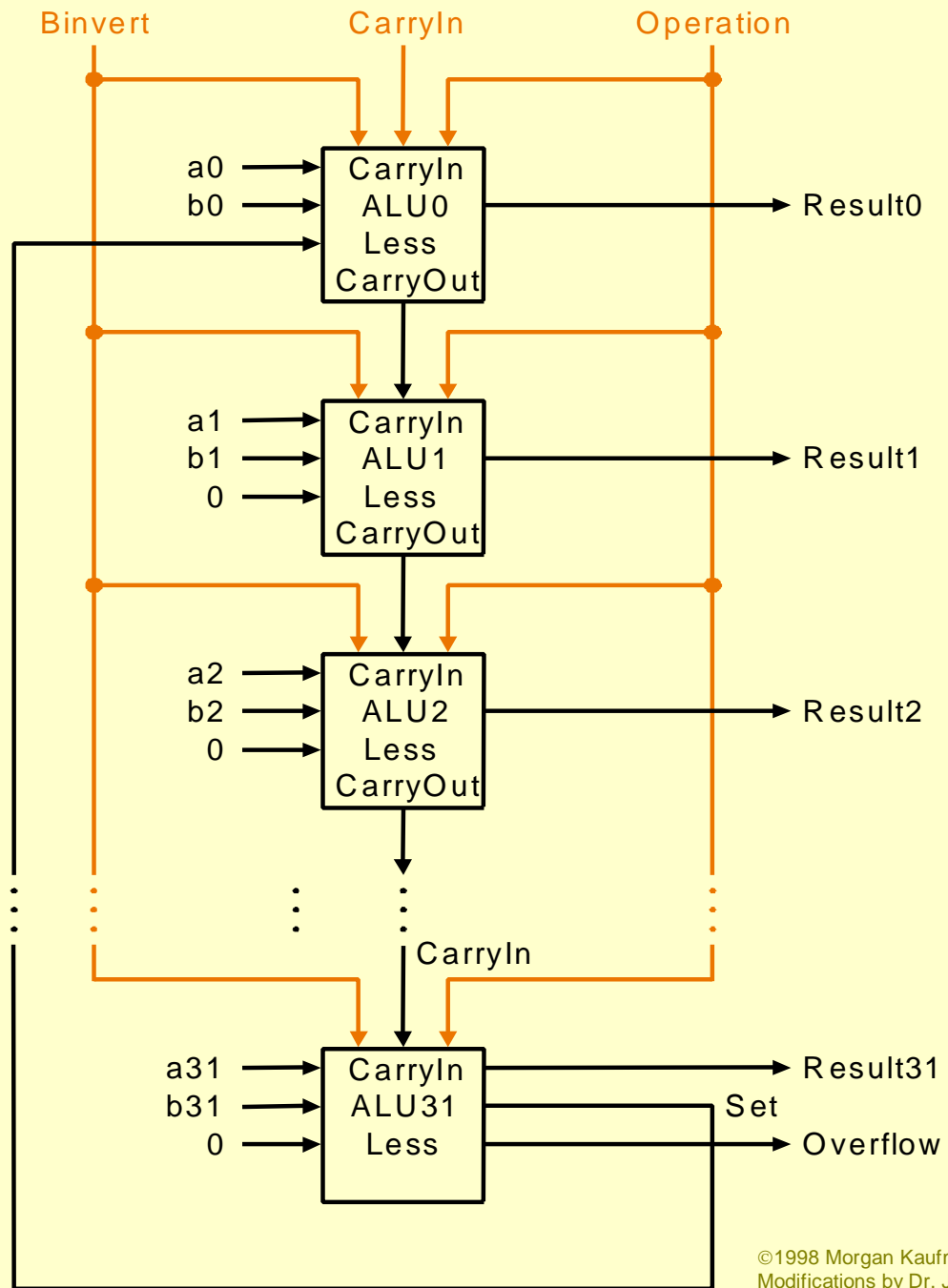
Tailoring the ALU to the MIPS

- **Need to support the set-on-less-than instruction (slt)**
 - remember: **slt is an arithmetic instruction**
 - produces a 1 if $rs < rt$ and 0 otherwise
 - use subtraction: $(a-b) < 0$ implies $a < b$
- **Need to support test for equality (beq \$t5, \$t6, \$t7)**
 - use subtraction: $(a-b) = 0$ implies $a = b$

Supporting slt

- Can we figure out the idea?





Test for equality

- Notice control lines:

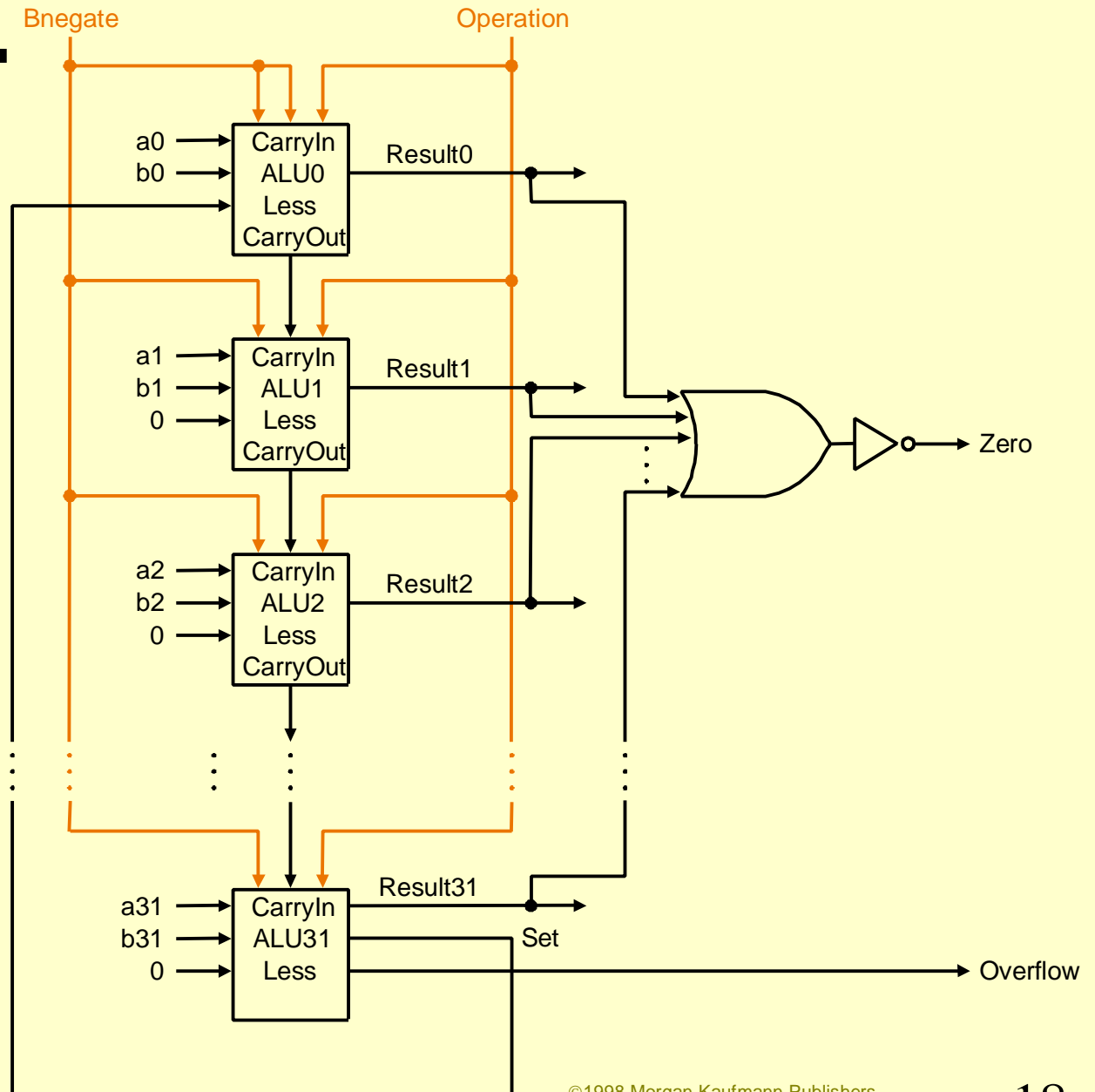
000 = and

001 = or

010 = add

110 = subtract

111 = slt



Conclusion

- **We can build an ALU to support the MIPS instruction set**
 - **key idea: use multiplexor to select the output we want**
 - **we can efficiently perform subtraction using two's complement**
 - **we can replicate a 1-bit ALU to produce a 32-bit ALU**
- **Important points about hardware**
 - **all of the gates are always working**
 - **the speed of a gate is affected by the number of inputs to the gate**
 - **the speed of a circuit is affected by the number of gates in series (on the “critical path” or the “deepest level of logic”)**
- **Our primary focus: comprehension, however,**
 - **Clever changes to organization can improve performance (similar to using better algorithms in software)**
 - **we'll look at two examples for addition and multiplication**

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
 - two extremes: ripple carry and sum-of-products

Can you see the *ripple*? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$

$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$

$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$

$$c_2 =$$

$$c_3 =$$

$$c_4 =$$

Not feasible! Why?

Carry-lookahead adder

- An approach in-between our two extremes

- Motivation:

- If we didn't know the value of carry-in, what could we do?
- When would we always generate a carry? $g_i = a_i b_i$
- When would we propagate the carry? $p_i = a_i + b_i$

- Did we get rid of the ripple?

$$c_1 = g_0 + p_0 c_0$$

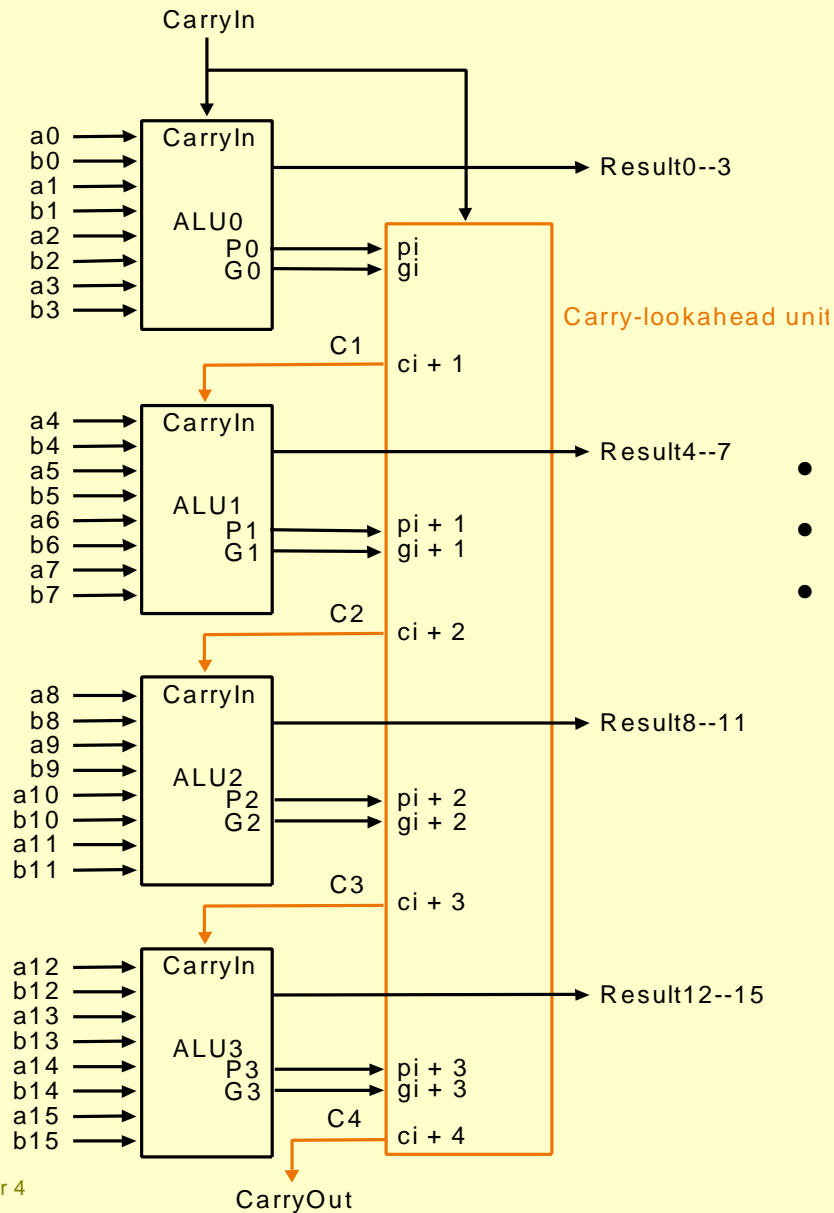
$$c_2 = g_1 + p_1 c_1 \quad c_2 =$$

$$c_3 = g_2 + p_2 c_2 \quad c_3 =$$

$$c_4 = g_3 + p_3 c_3 \quad c_4 =$$

Feasible! Why?

Use principle to build bigger adders



- Can't build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

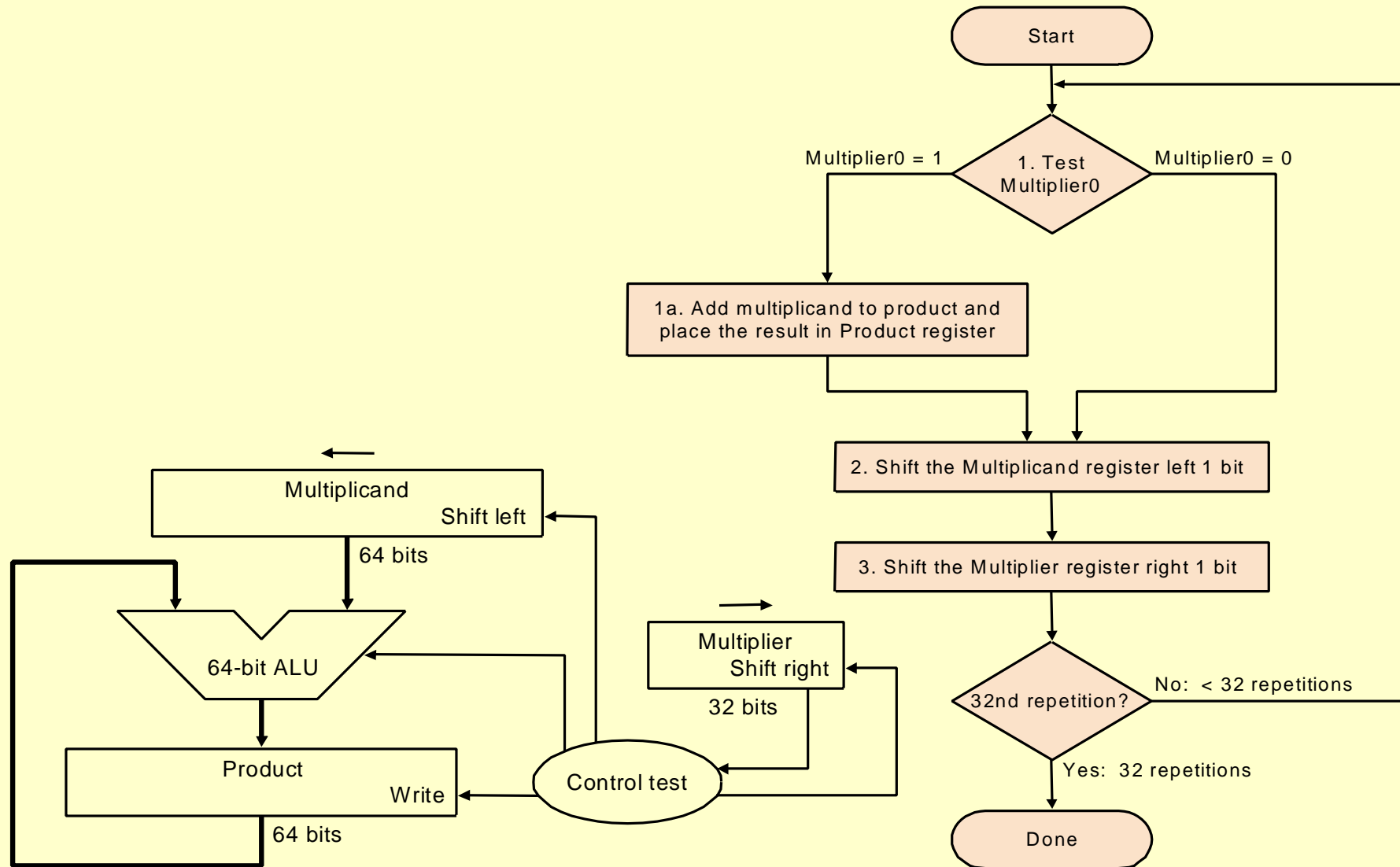
Multiplication

- **More complicated than addition**
 - accomplished via shifting and addition
- **More time and more area**
- **Let's look at 3 versions based on gradeschool algorithm**

$$\begin{array}{r} 0010 \text{ (multiplicand)} \\ \underline{\underline{x}} \underline{\underline{1011}} \text{ (multiplier)} \end{array}$$

- **Negative numbers: convert and multiply**
 - there are better techniques, we won't look at them

Multiplication: Implementation



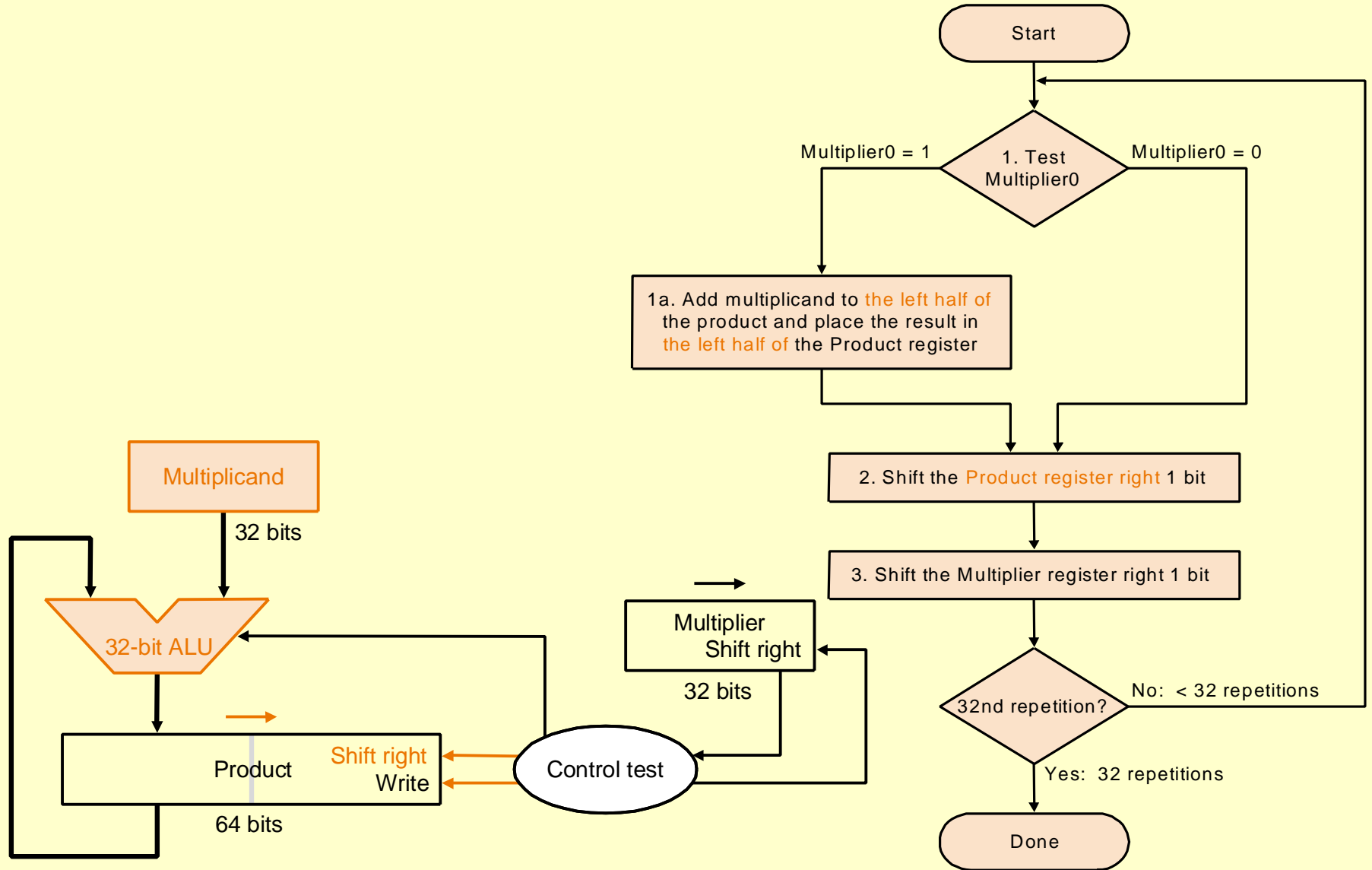
Example: First Multiply Algorithm

(See Example, p. 253)

- Using 4-bit numbers, multiply $2_{10} \cdot 3_{10}$, or $0010_2 \cdot 0011_2$.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: (1) Prod=Prod+Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: (1) Prod=Prod+Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: (0) no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: (0) no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Second Version



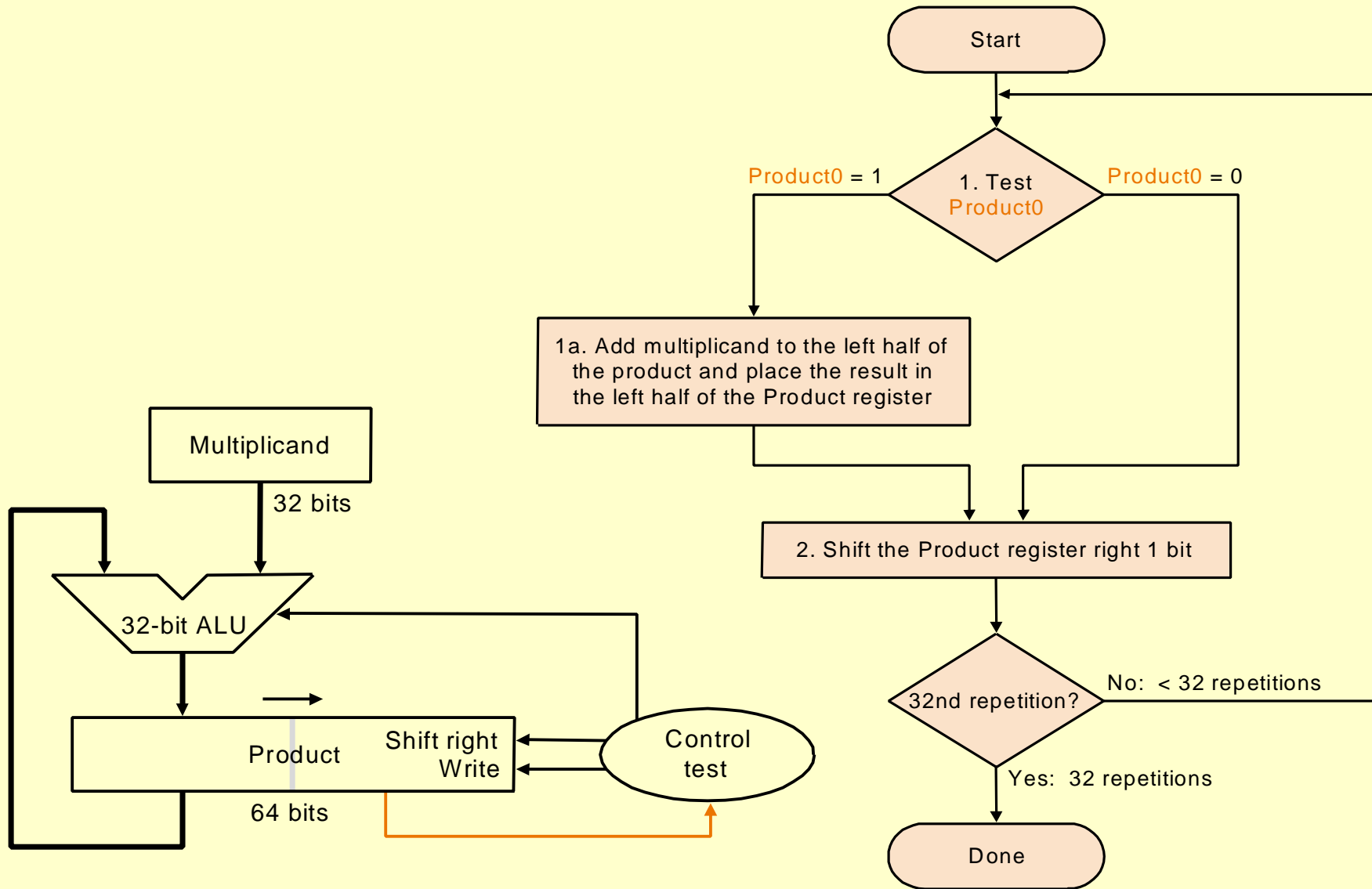
Example: Second Multiply Algorithm

(See Example, p. 256)

- Using 4-bit numbers, multiply $2_{10} \cdot 3_{10}$, or $0010_2 \cdot 0011_2$.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0010	0000 0000
1	1a: (1) Prod=Prod+Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	0001	0010	0001 0000
2	1a: (1) Prod=Prod+Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	0000	0010	0001 1000
3	1: (0) no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	0000	0010	0000 1100
4	1: (0) no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

Final Version



Example: Third Multiply Algorithm

(See Example, p. 257-258)

- Using 4-bit numbers, multiply $2_{10} \cdot 3_{10}$, or $0010_2 \cdot 0011_2$.

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: (1) Prod=Prod+Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: (1) Prod=Prod+Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: (0) no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: (0) no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

Signed Multiplication: Booth's Algorithm

- **Motivation:**
 - **Possible procedure for signed numbers:**
 - convert multiplier and multiplicand to positive, remembering signs
 - run algorithm for 31 iterations, leaving signs out
 - negate product if original signs disagree
 - **Alternately:**
 - Extend sign of product during the shifting steps
- **Booth's Algorithm**
 - uses both add and subtract based on bit patterns

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of run of 1s	0000 1111 1 000
1	1	Middle of run of 1s	0000 11 11 1000
0	1	End of run of 1s	0000 1 111 1000
0	0	Middle of run of 0s	0000 1111 1000

Signed Multiplication: Booth's Algorithm

- **Note: Test portion looks at two (2) bits of multiplier**
- **Algorithm:**
 - **Depending on the current and previous bits, do one of the following:**
 - 00: Middle of string of 0s, so no arithmetic operation
 - 01: End of string of 1s, so *add* the multiplicand to the left half of the product
 - 10: Beginning of a string of 1s, so *subtract* the multiplicand from the left half of the product
 - 11: Middle of string of 1s, so no arithmetic operation
 - **As in the previous algorithm, shift the Product register right 1 bit.**

Signed Multiplication: Booth's Algorithm

- Using 4-bit numbers, multiply $2_{10} \cdot -3_{10} = -6_{10}$, or $0010_2 \cdot 1101_2 = 1111\ 1010_2$.

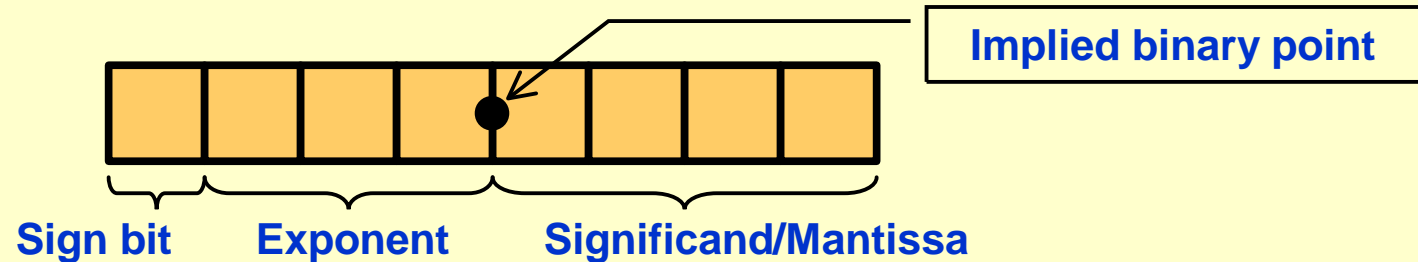
Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: (10) Prod=Prod-Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: (01) Prod=Prod+Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: (10) Prod=Prod-Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: (11) no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

Floating Point (a brief look)

- We need a way to represent
 - numbers with fractions, e.g., 3.1416
 - very small numbers, e.g., .000000001
 - very large numbers, e.g., 3.15576×10^9
- Representation:
 - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
 - more bits for significand gives more accuracy
 - more bits for exponent increases range
- IEEE 754 floating point standard:
 - single precision: 8 bit exponent, 23 bit significand
 - double precision: 11 bit exponent, 52 bit significand

Simplified 8-bit example

- Consider an 8-bit word used to represent floating point numbers as follows:



- The exponent uses *biased* notation. Using n bits for biased notation means that the bias is $2^{n-1}-1$. So, in our example, the 3-bit biased exponent is biased by $2^{3-1}-1 = 2^2-1 = 4-1 = 3$.
- Biased notation versus 2's complement

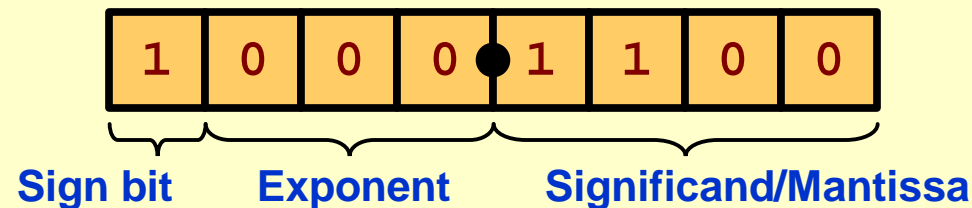
	<i>Binary</i>	000	001	010	011	100	101	110	111
<i>Decimal</i>	2's C	0	1	2	3	-4	-3	-2	-1
	Biased	-3	-2	-1	0	1	2	3	4

Simplified 8-bit example

- **Example:** How would you represent $-7/16$ using the 8-bit scheme?

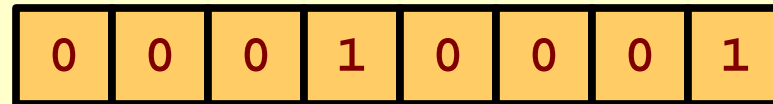
$$\frac{-7}{16} = \frac{-7}{2^4} \quad \Rightarrow \quad \begin{array}{l} 0.4375 \\ 0.875 \\ 1.75 \\ 1.5 \\ 1.0 \end{array} \quad \Rightarrow \quad \begin{array}{l} \underbrace{0}_{2^0} \cdot \underbrace{0}_{2^{-1}} \underbrace{1}_{2^{-2}} \underbrace{1}_{2^{-3}} \underbrace{1}_{2^{-4}} \underbrace{0}_{2^{-5}} \\ \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \end{array}$$

- Since $-0.001110 \cdot 2^0 = -1.110 \cdot 2^{-3}$
- Hence, we have ...



Simplified 8-bit example

- **Question:** What is the following 8-bit floating point number in decimal?



- **Question:** What is the *smallest* (most negative) decimal number representable by the above scheme?
- **Question:** What is the *largest* (most positive) decimal number representable by the above scheme?
- **Question:** What is the *smallest* (positive) fraction representable by the above scheme?
- **Question:** If we double the word length from 8 bits to 16 bits, what factor(s) affect the precision/accuracy of our representation? What affect the range of representable values?

IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit
- Exponent is “biased” to make sorting easier
 - all 0s is smallest exponent all 1s is largest
 - bias of 127 for single precision and 1023 for double precision
 - summary: $(-1)^{\text{sign}} \cdot (1 + \text{significand}) \cdot 2^{\text{exponent} - \text{bias}}$
- Example:
 - decimal: $-.75 = -3/4 = -3/2^2$
 - binary: $-.11 = -1.1 \times 2^{-1}$
 - floating point: exponent = 126 = 01111110
 - IEEE single precision: 1 01111110 100000000000000000000000

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
 - IEEE 754 keeps two extra bits, guard and round
 - four rounding modes
 - positive divided by zero yields “infinity”
 - zero divide by zero yields “not a number”
 - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
 - see text for description of 80x86 and Pentium bug!

Chapter Four Summary

- **Computer arithmetic is constrained by limited precision**
- **Bit patterns have no inherent meaning but standards do exist**
 - **two's complement**
 - **IEEE 754 floating point**
- **Computer instructions determine “meaning” of the bit patterns**
- **Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).**

- **We are ready to move on (and implement the processor)**

you may want to look back (Section 4.12 is great reading!)