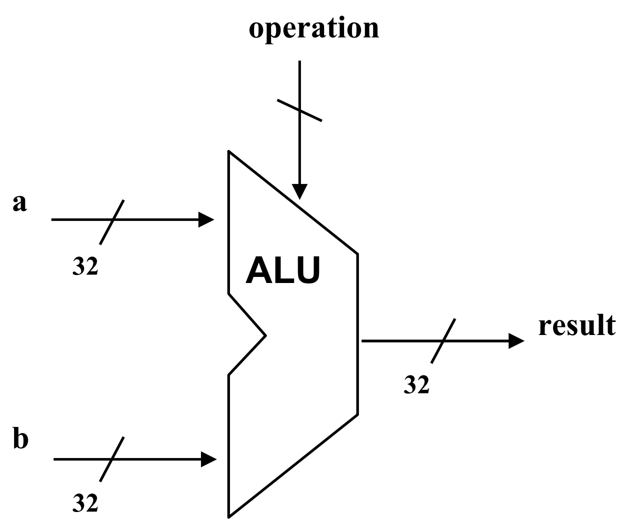
Chapter 4: Arithmetic

- **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: $0...2^n-1$
- **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:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$ — *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

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)**

```
  0111          0111          0110
+ 0110        - 0110        - 0101
```

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

```
  0111
+ 1010
```

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

```
  0111
+ 0001
_ 1000
```

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*

Detecting overflow in software

```
addu   $t0,   $t1,     $t2            $t0=sum
xor    $t3,   $t1,     $t2            Check if signs differ
slt    $t3,   $t3,     $zero          $t3=1 is signs differ
bne    $t3,   $zero,   No_overflow    $t1, $t2 signs differ, can’t be overflow
xor    $t3,   $t0,     $t1            Sign of sum match too?
slt    $t3,   $t3,     $zero          $t3=1 if sum sign different
bne    $t3,   $zero,   Overflow       Yup, overflow…
```

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

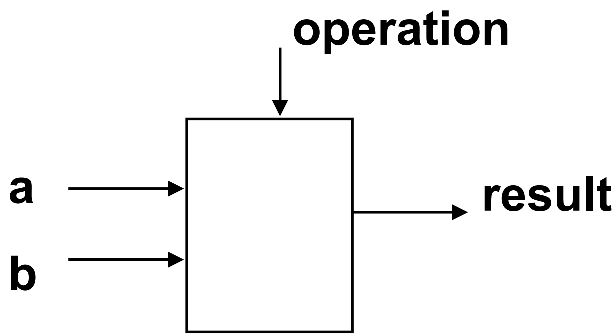
- **Show the truth table for these three functions.**

- **Show the Boolean equations for these three functions.**

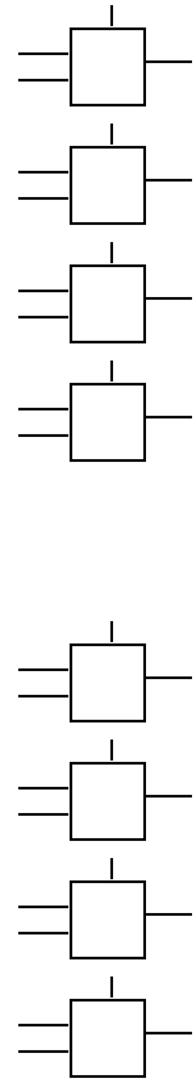
- **Show an implementation consisting of inverters, AND, and OR gates.**

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



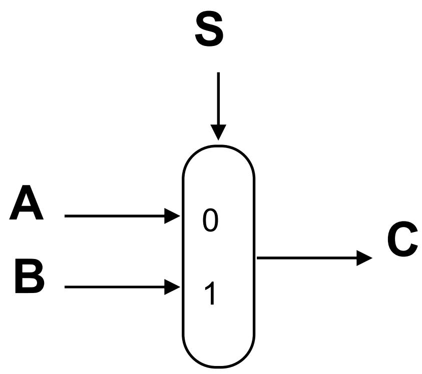
op	a	b	res



- Possible Implementation (sum-of-products):

Review: The Multiplexor

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

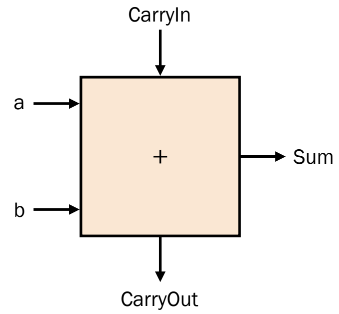


note: we call this a 2-input mux
even though it has 3 inputs!

- **Lets build our ALU using a MUX:**

Different Implementations

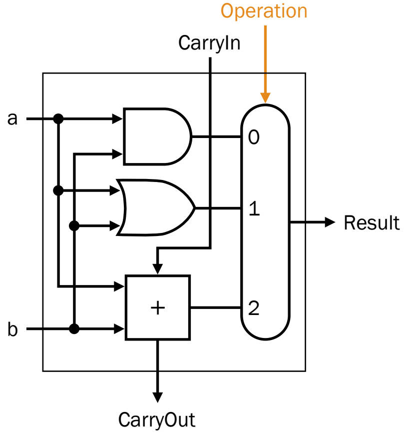
- **Not easy to decide the "best" way to build something**
  - **Don't want too many inputs to a single gate**
  - **Don't want to have to go through too many gates**
  - **for our purposes, ease of comprehension is important**
- **Let's look at a 1-bit ALU for addition:**

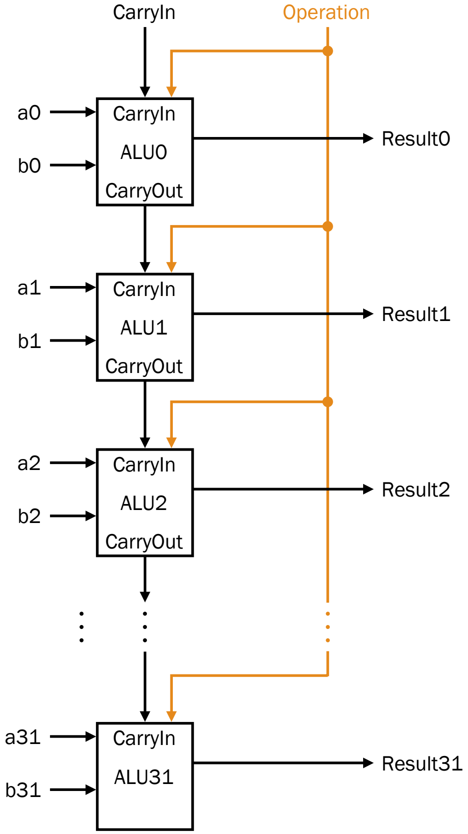


$$c_{out} = a\,b + a\,c_{in} + b\,c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- **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

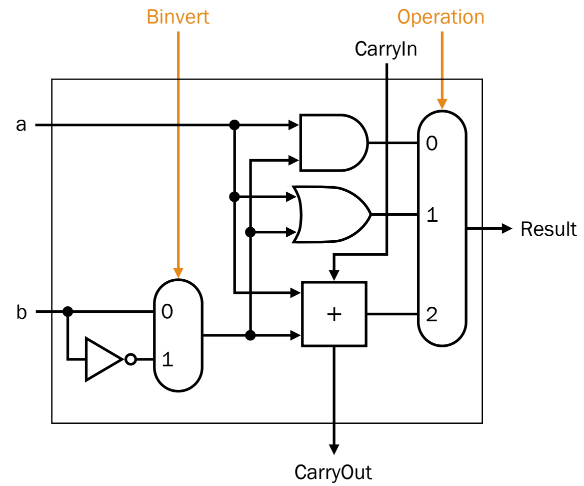




What about subtraction (a – b) ?

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

- A very clever solution:

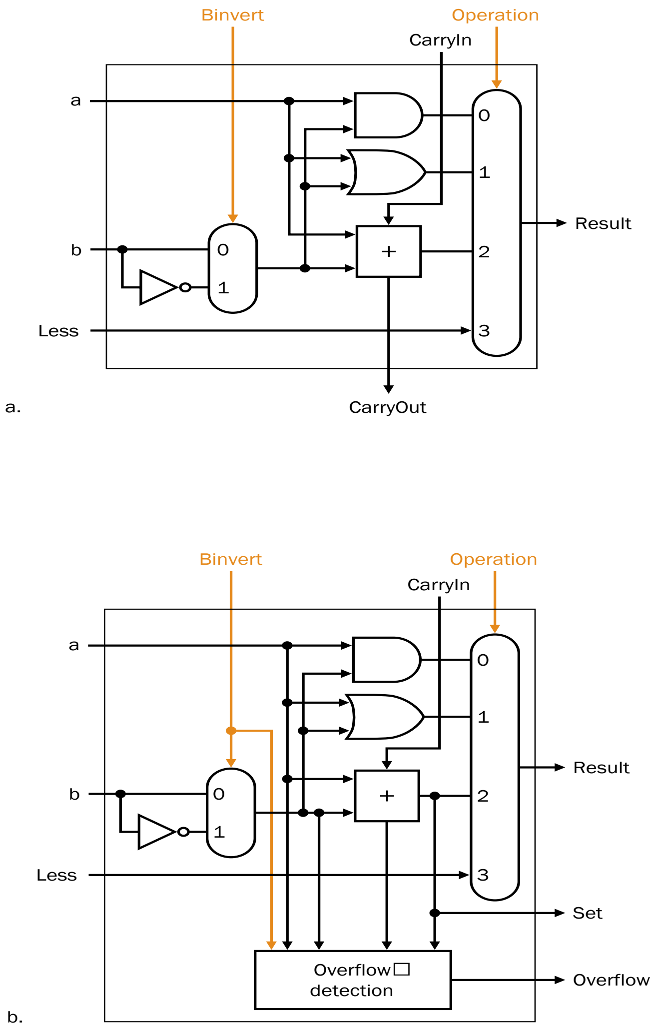


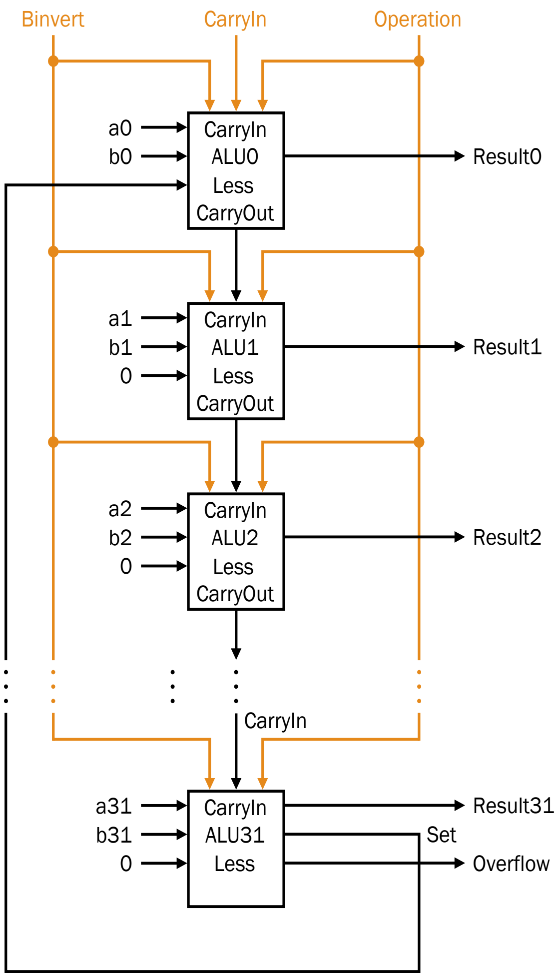
Tailoring the ALU to the MIPS

- **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?**



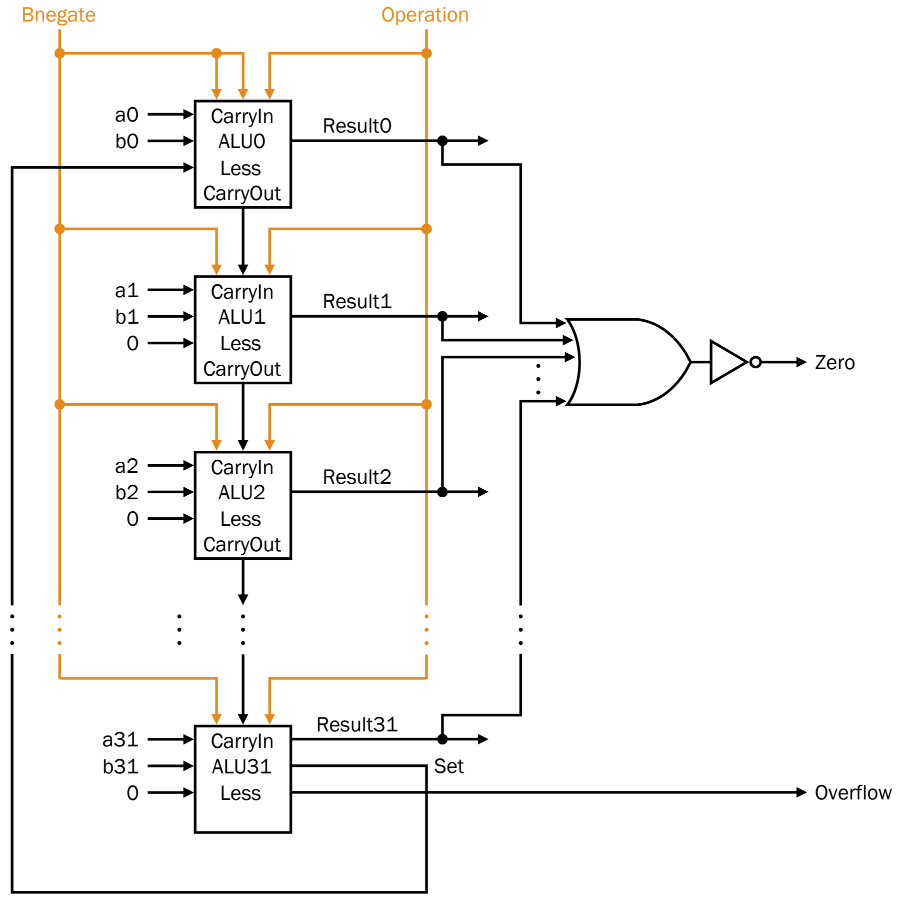


Test for equality

- **Notice control lines:**

  000 = and
  001 = or
  010 = add
  110 = subtract
  111 = slt

•*Note: zero is a 1 when the result is zero!*



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?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\qquad c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\qquad c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\qquad c_4 =$

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?

$c_1 = g_0 + p_0 c_0$

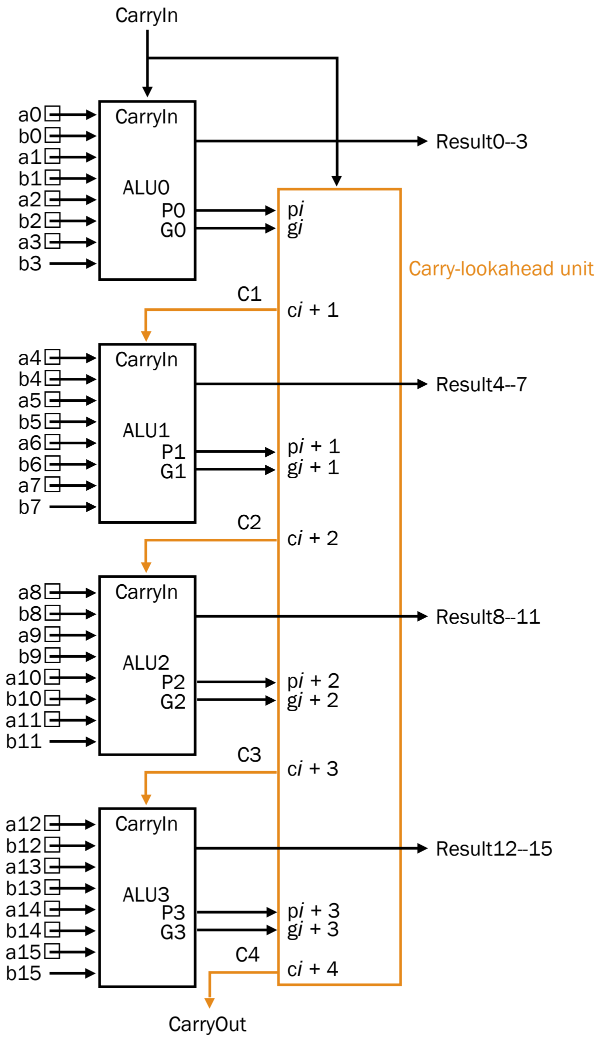
$c_2 = g_1 + p_1 c_1$ $\quad c_2 =$

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

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

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!**

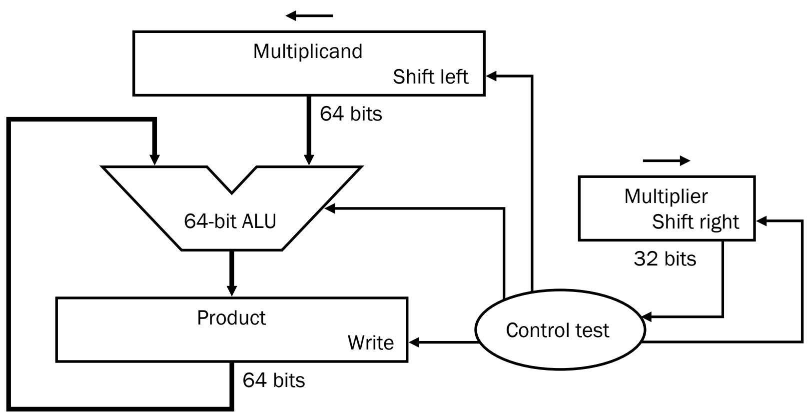
Multiplication

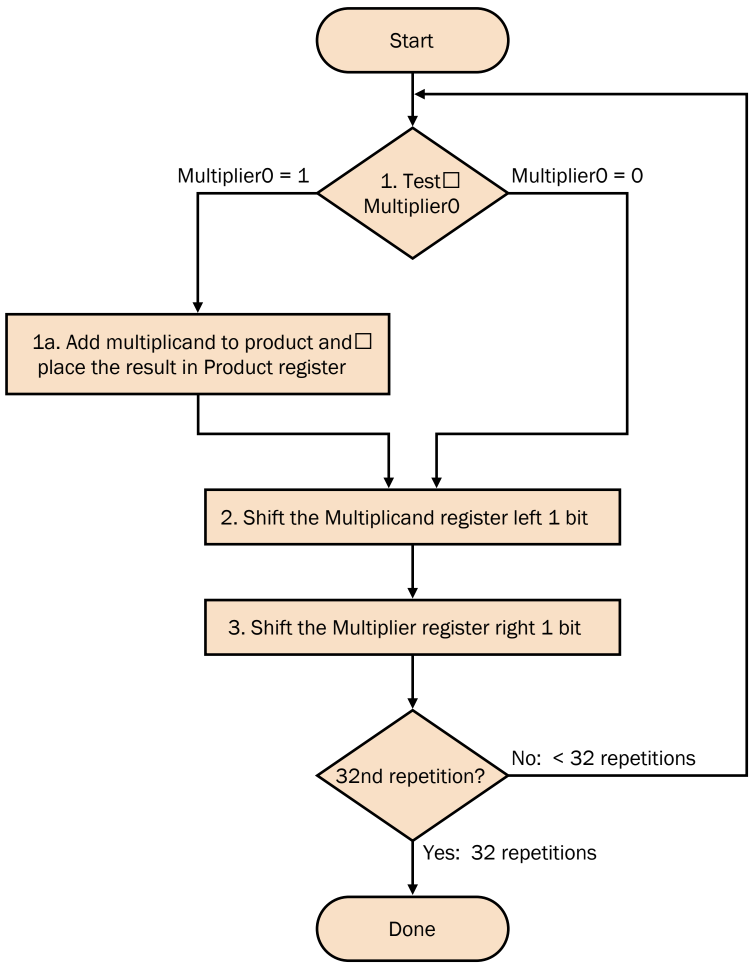
- **More complicated than addition**
  - **accomplished via shifting and addition**
- **More time and more area**
- **Let's look at 3 versions based on gradeschool algorithm**

```
      0010   (multiplicand)
    x 1011   (multiplier)
```

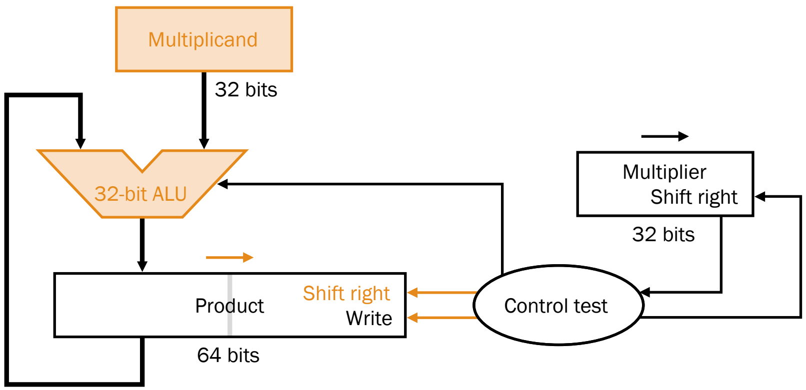
- **Negative numbers: convert and multiply**
  - **there are better techniques, we won't look at them**

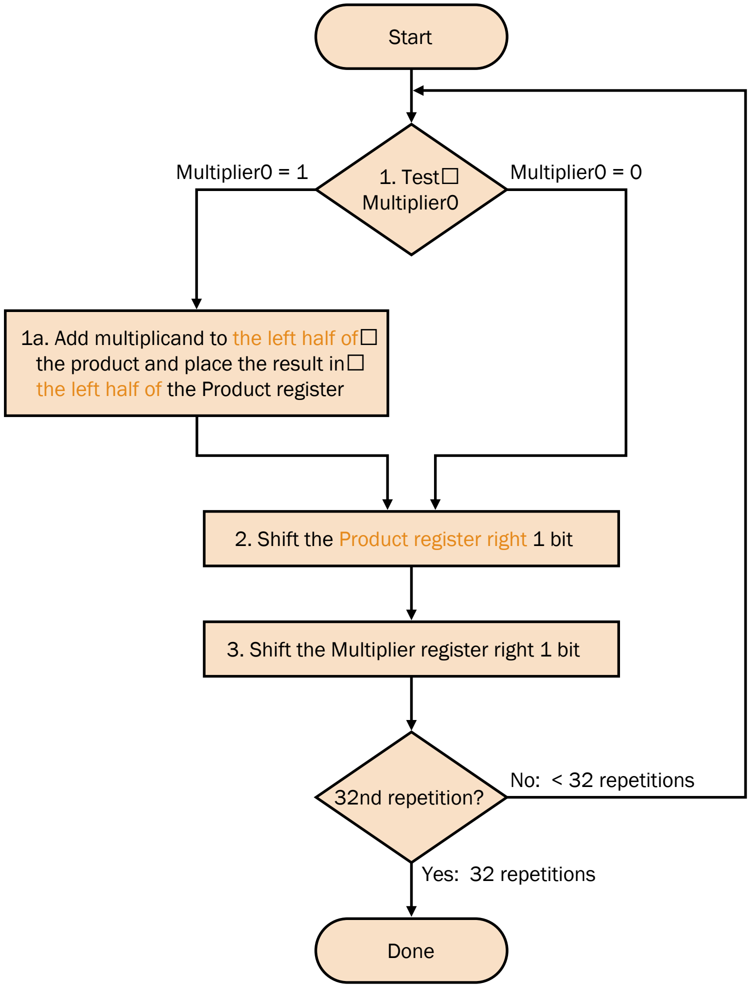
Multiplication: Implementation



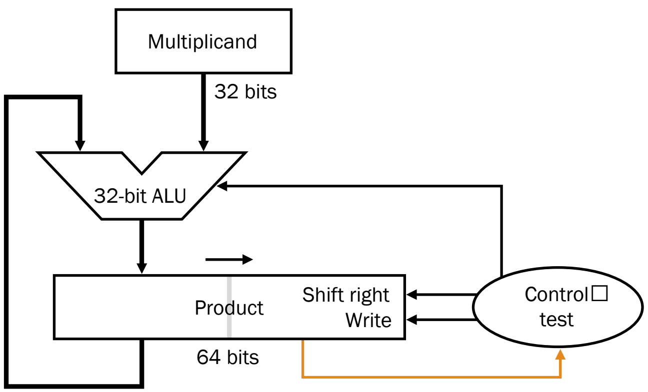


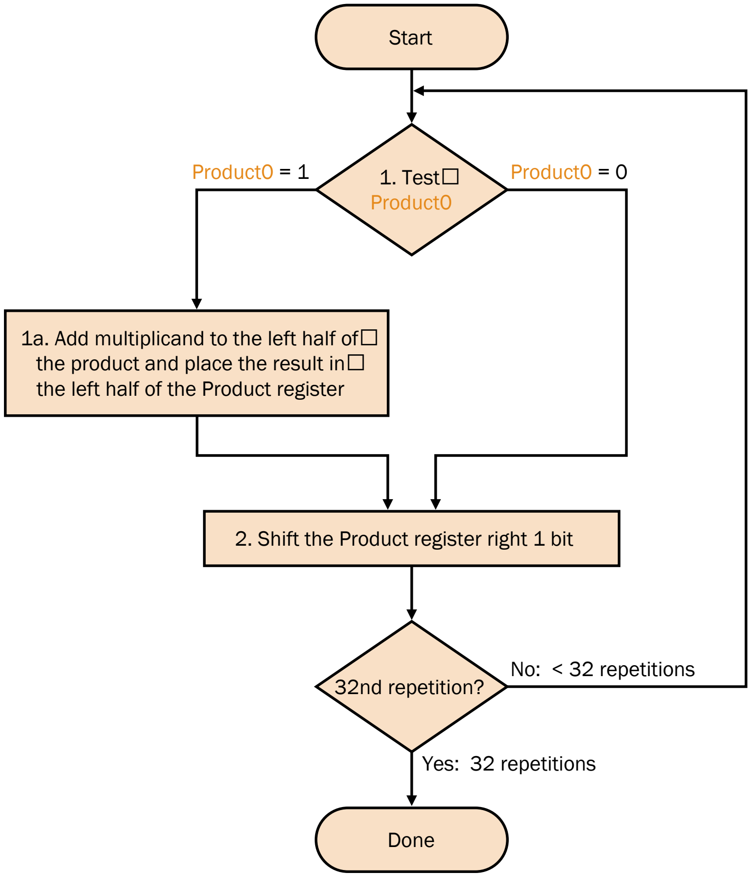
Second Version





Final Version





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: $(-1)^{sign} \times significand \times 2^{exponent}$
  - 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

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:** $(-1)^{sign} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$
- **Example:**
  - **decimal:** $-.75 = -3/4 = -3/2^2$
  - **binary:** $-.11 = -1.1 \times 2^{-1}$
  - **floating point: exponent = 126 = 01111110**
  - **IEEE single precision:** `10111111010000000000000000000000`

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!)