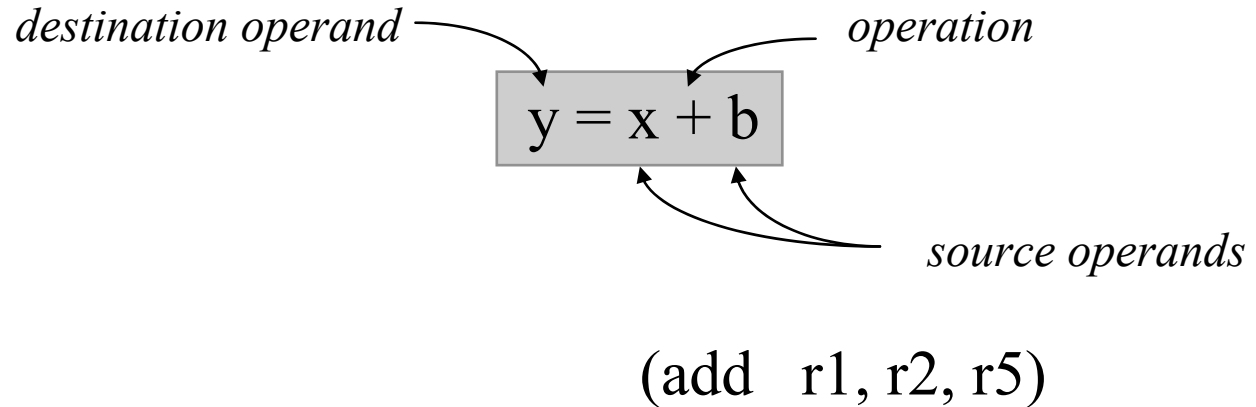


Midterm Review

Or, I'm a survivor, a can reconstruct a computer
after the end of the world

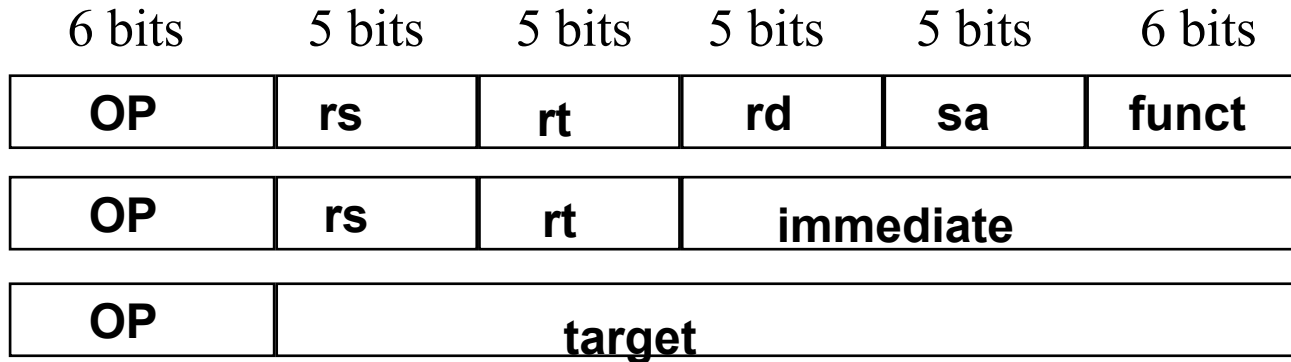
Key ISA decisions

- operations
 - how many?
 - which ones
- operands
 - how many?
 - location
 - types
 - how to specify?
- instruction format
 - size
 - how many formats?



*how does the computer know what
0001 0100 1101 1111
means?*

MIPS Instruction Formats



- the opcode tells the machine which format
- so `add r1, r2, r3` has
 - `opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0`
 - `000000 00010 00011 00001 00000 100000`

Accessing the Operands

- operands are generally in one of two places:
 - registers (32 int, 32 fp)
 - memory (2^{32} locations)
- registers are
 - easy to specify
 - close to the processor (fast access)
- the idea that we want to access registers whenever possible led to *load-store architectures*.
 - normal arithmetic instructions only access registers
 - only access memory with explicit loads and stores

Load-store architectures

can do:

add r1=r2+r3

and

load r3, M(address)

⇒ forces heavy dependence on registers, which is exactly what you want in today's CPUs

can't do

add r1 = r2 + M(address)

- more instructions
+ fast implementation (e.g., easy pipelining)

Addressing Modes

how do we specify the operand we want?

- **Register direct** **R3**
- **Immediate (literal)** **#25**
- **Direct (absolute)** **M[10000]**

- **Register indirect** **M[R3]**
- **Base+Displacement** **M[R3 + 10000]**
 if register is the program counter, this is *PC-relative*
- **Base+Index** **M[R3 + R4]**
- **Scaled Index** **M[R3 + R4*d + 10000]**
- **Autoincrement** **M[R3++]**
- **Autodecrement** **M[R3 - -]**

- **Memory Indirect** **M[M[R3]]**

Alternative Architectures

- Design alternative:
 - provide more powerful operations
 - goal is to reduce number of instructions executed
 - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
 - virtually all new instruction sets since 1982 have been RISC
 - VAX: minimize code size, make assembly language easy
instructions from 1 to 54 bytes long!
- We’ll look at PowerPC and 80x86

Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Four principles of IS architecture
 - simplicity favors regularity
 - smaller is faster
 - good design demands compromise
 - make the common case fast

- Know equations for components of performance
- Amdahl's law of diminishing returns
- Relative performance
- MIPS, MFLOPS, arguments for and against usefulness

How to measure Execution Time?

```
% time program
... program results ...
90.7u 12.9s 2:39 65%
%
```

- Wall-clock time?
- user CPU time?
- user + kernel CPU time?
- Answer:

Our definition of Performance

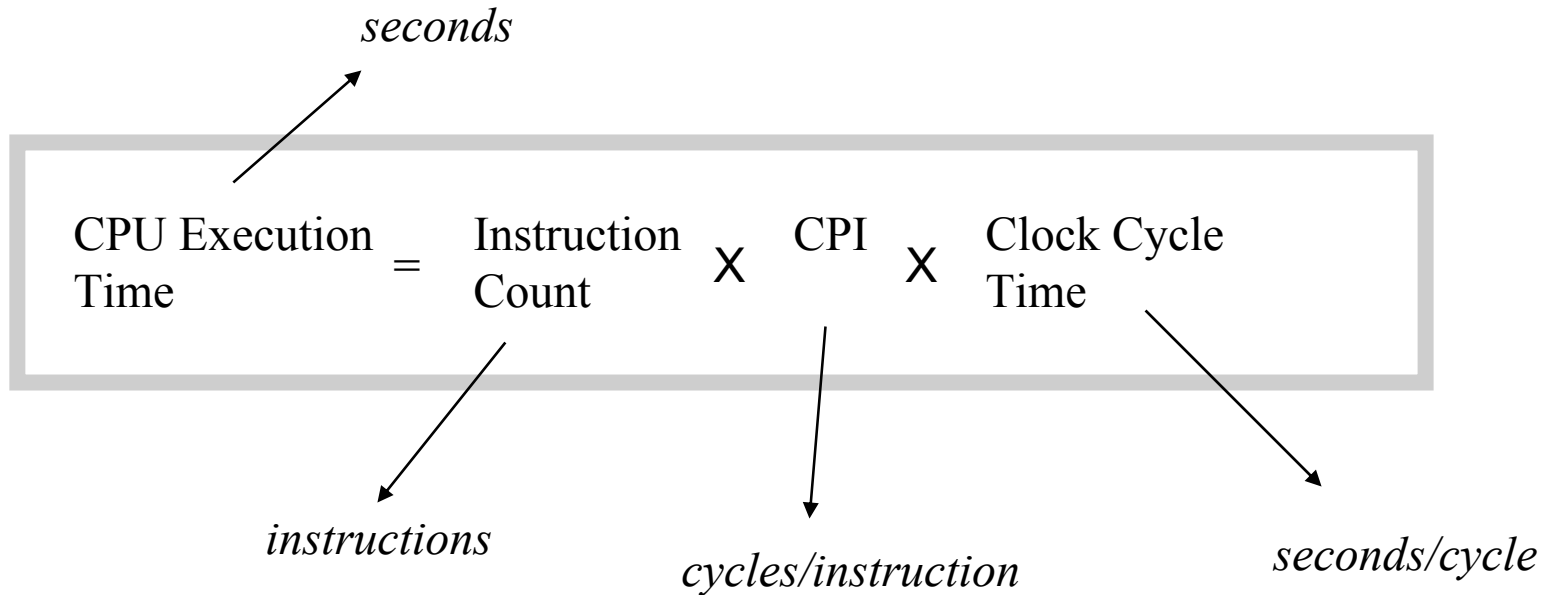
$$\text{Performance}_X = \frac{1}{\text{Execution Time}_X}, \text{ for program } X$$

- only has meaning in the context of a program or workload
- Not very intuitive as an absolute measure

Relative Performance

- can be confusing
 - A runs in 12 seconds
 - B runs in 20 seconds
 - $A/B = .6$, so A is 40% faster, or 1.4X faster, or B is 40% slower
 - $B/A = 1.67$, so A is 67% faster, or 1.67X faster, or B is 67% slower
- needs a precise definition

All Together Now



Amdahl's Law

- The impact of a performance improvement is limited by the percent of execution time affected by the improvement

$$\text{Execution time after improvement} = \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} + \text{Execution Time Unaffected}$$

- Make the common case fast!!

Key Points

- Be careful how you specify performance
- Execution time = instructions * CPI * cycle time
- Use real applications
- Use standards, if possible
- Make the common case fast

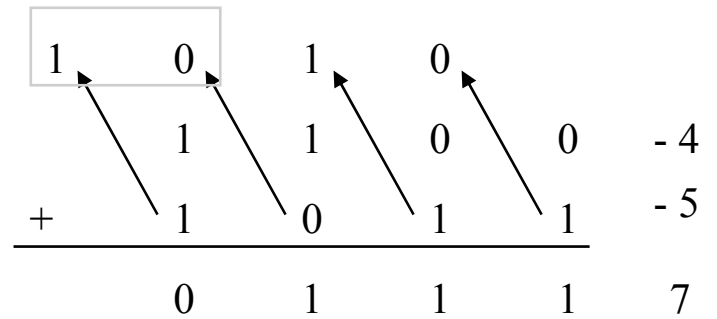
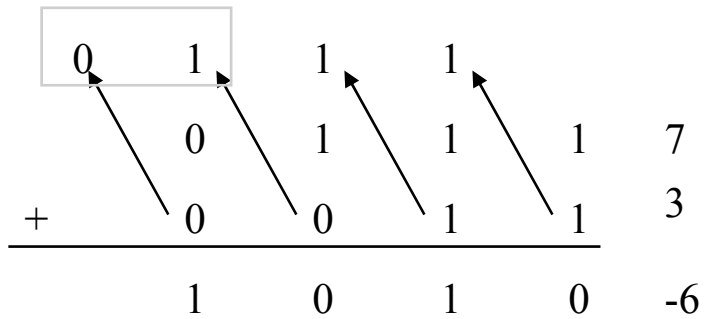
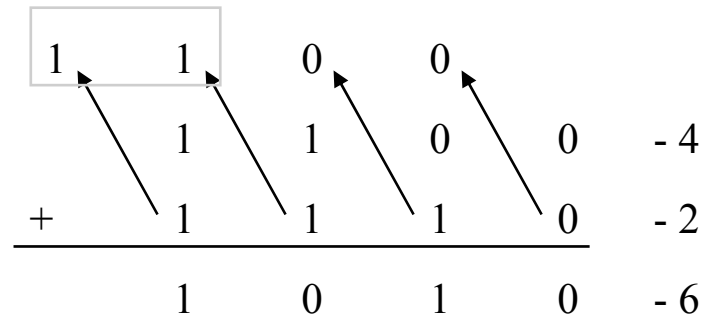
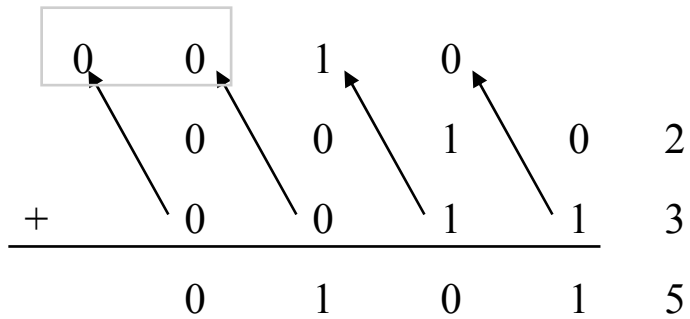
- Know two's complement
- Hardware and algorithms for addition, subtract, multiplication, division
- Know logic for carry in and carry out and sum

Two's Complement Representation

- 2's complement representation of negative numbers
 - Take the bitwise inverse and add 1
- Biggest 4-bit Binary Number: 7 Smallest 4-bit Binary Number: -8

<u>Decimal</u>	<u>Two's Complement Binary</u>
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

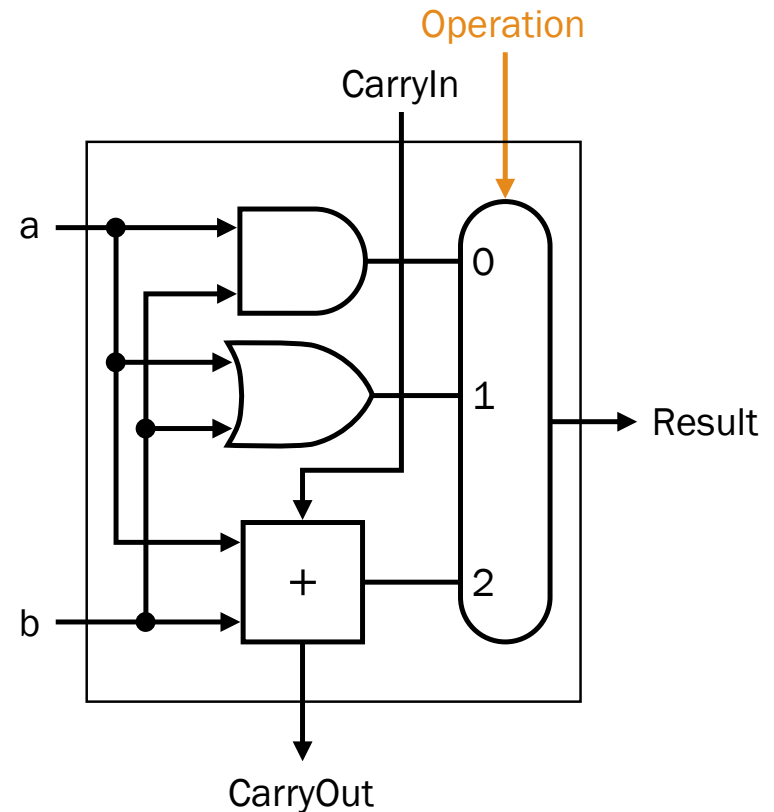
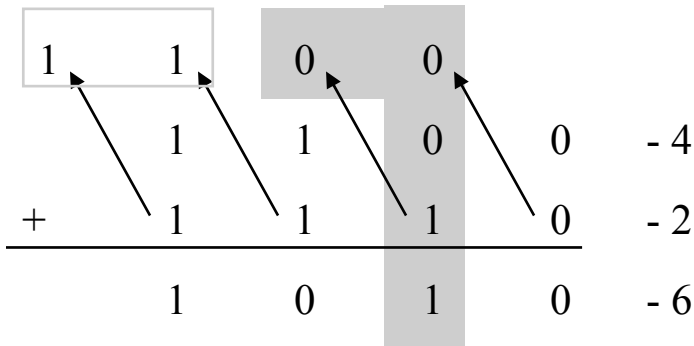
Overflow Detection



So how do we detect overflow?

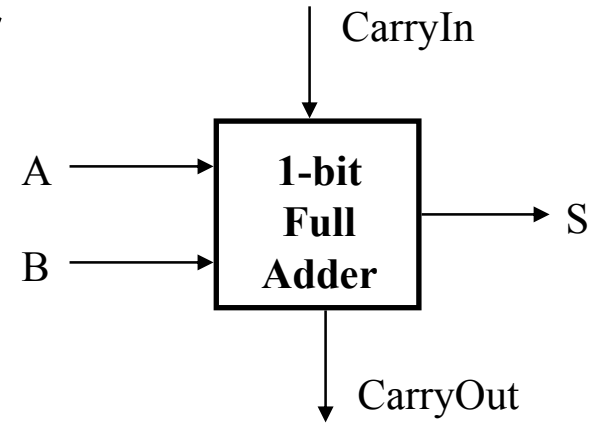
A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD



A One-bit Full Adder

- This is also called a (3, 2) adder
- *Half Adder*: No CarryIn
- Truth Table:



Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

Logic Equation for CarryOut

Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

$$\text{CarryOut} = (!A \& B \& \text{CarryIn}) \mid (A \& !B \& \text{CarryIn}) \mid (A \& B \& !\text{CarryIn}) \\ \mid (A \& B \& \text{CarryIn})$$

$$\text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B$$

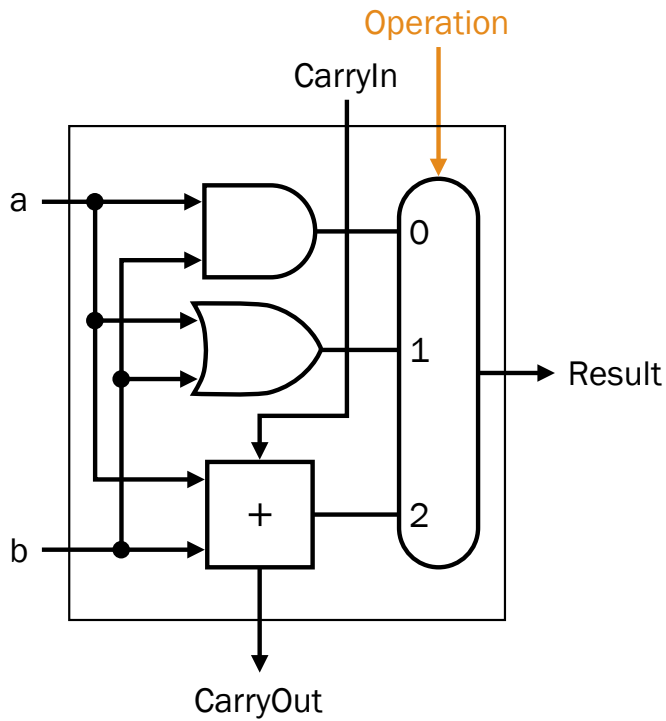
Logic Equation for Sum

Inputs			Outputs		Comments
A	B	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 1 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

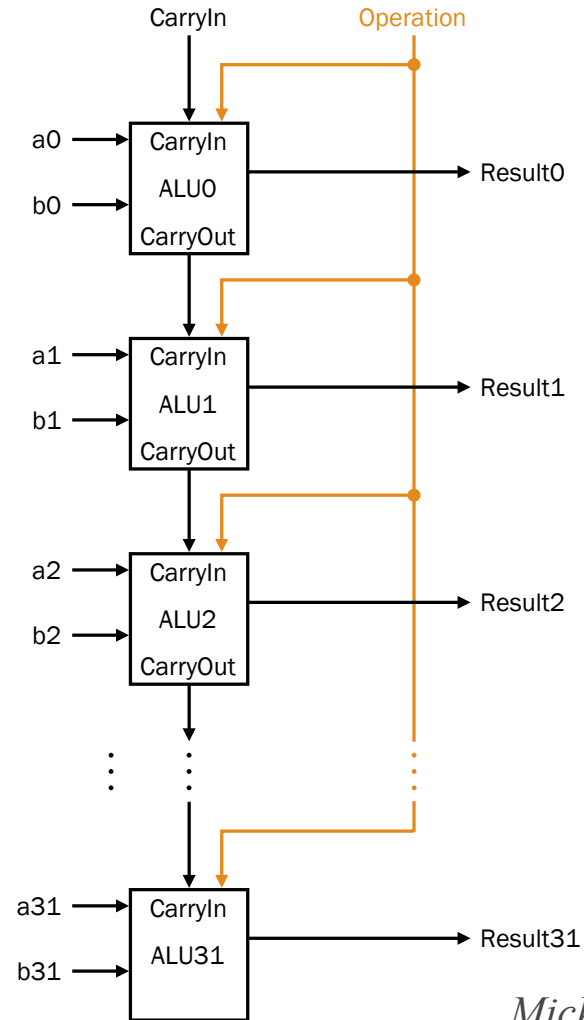
$$\text{Sum} = (!A \& !B \& \text{CarryIn}) \mid (!A \& B \& !\text{CarryIn}) \mid (A \& !B \& !\text{CarryIn}) \mid (A \& B \& \text{CarryIn})$$

A 32-bit ALU

1-bit ALU

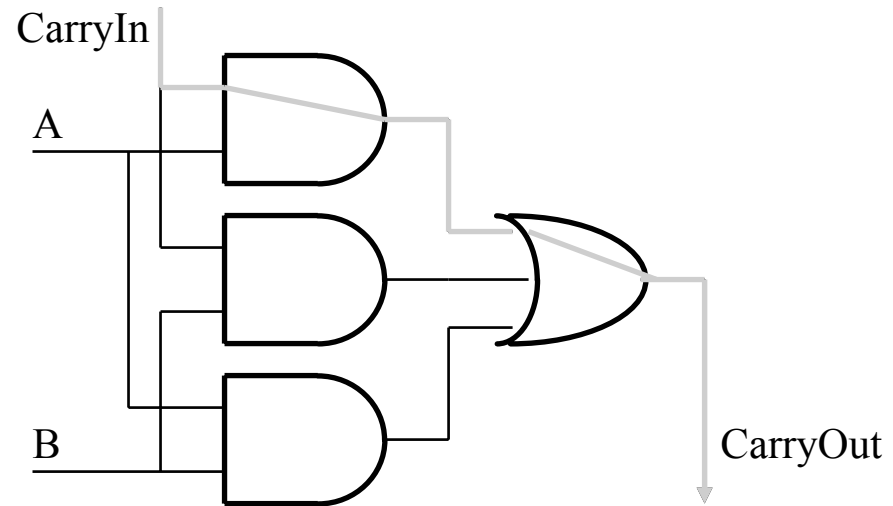
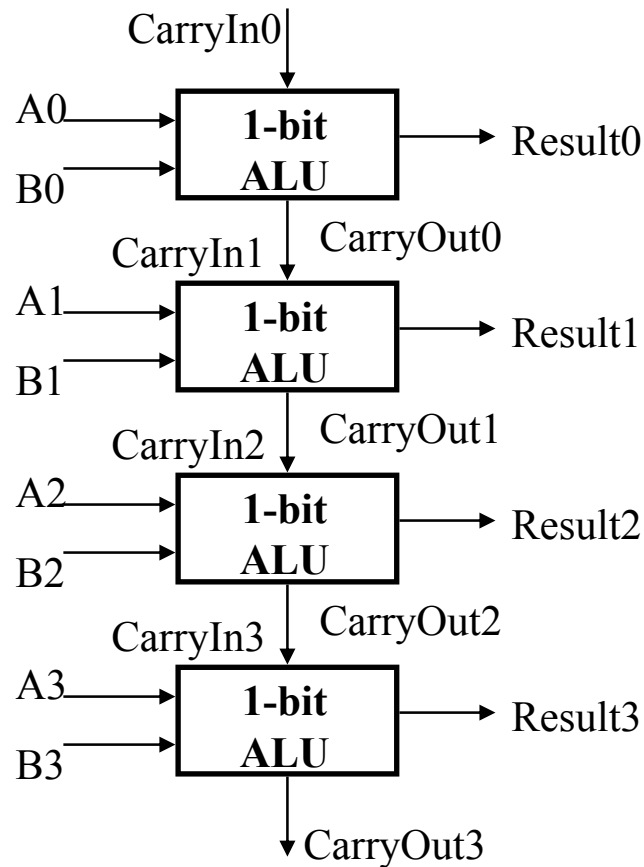


32-bit ALU



The Disadvantage of Ripple Carry

- The adder we just built is called a “Ripple Carry Adder”
 - The carry bit may have to propagate from LSB to MSB
 - Worst case delay for an N-bit RC adder: $2N$ -gate delay

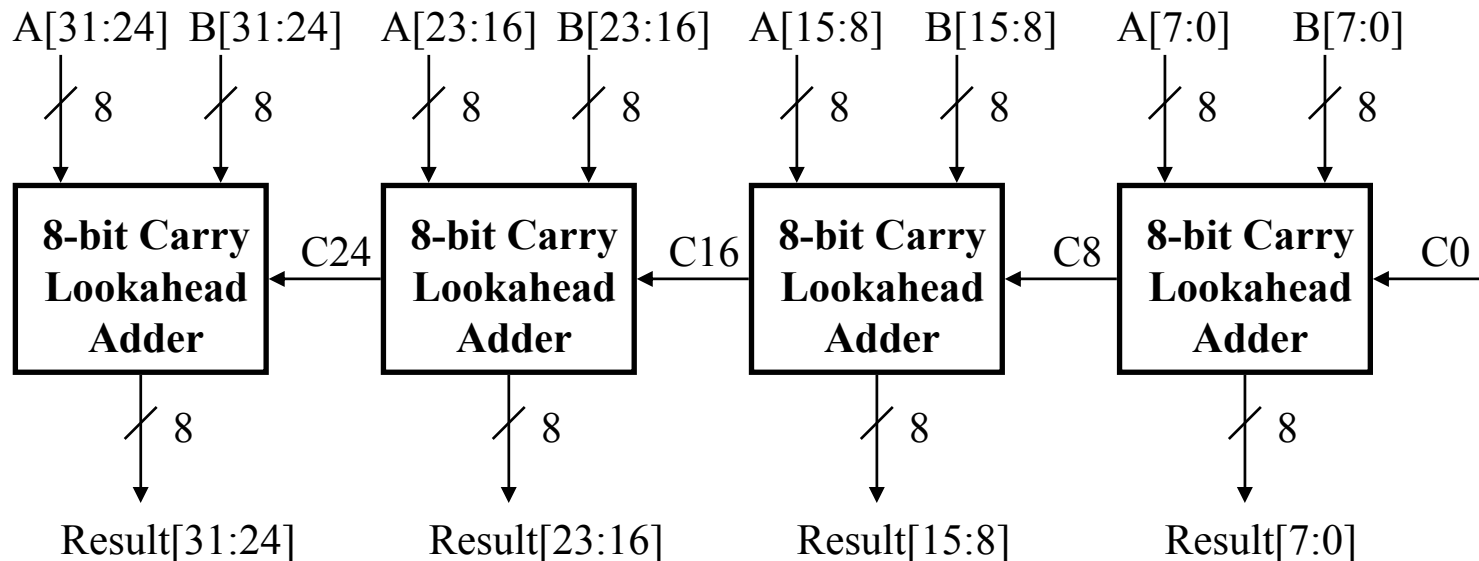


Carry Lookahead (Continued)

- Using the two new terms we just defined:
 - Generate Carry at Bit i $g_i = A_i \ \& \ B_i$
 - Propagate Carry via Bit i $p_i = A_i \ \text{or} \ B_i$
- We can rewrite:
 - $C_{in1} = g_0 \ | \ (p_0 \ \& \ C_{in0})$
 - $C_{in2} = g_1 \ | \ (p_1 \ \& \ g_0) \ | \ (p_1 \ \& \ p_0 \ \& \ C_{in0})$
 - $C_{in3} = g_2 \ | \ (p_2 \ \& \ g_1) \ | \ (p_2 \ \& \ p_1 \ \& \ g_0) \ | \ (p_2 \ \& \ p_1 \ \& \ p_0 \ \& \ C_{in0})$
- Carry going into bit 3 is 1 if
 - We generate a carry at bit 2 (g_2)
 - Or we generate a carry at bit 1 (g_1) and bit 2 allows it to propagate ($p_2 \ \& \ g_1$)
 - Or we generate a carry at bit 0 (g_0) and bit 1 as well as bit 2 allows it to propagate ($p_2 \ \& \ p_1 \ \& \ g_0$)
 - Or we have a carry input at bit 0 (C_{in0}) and bit 0, 1, and 2 all allow it to propagate ($p_2 \ \& \ p_1 \ \& \ p_0 \ \& \ C_{in0}$)

A Partial Carry Lookahead Adder

- It is very expensive to build a “full” carry lookahead adder
 - Just imagine the length of the equation for C_{in31}
- Common practices:
 - Connect several N-bit Lookahead Adders to form a big adder
 - Example: connect four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder



MULTIPLY

- Paper and pencil example:

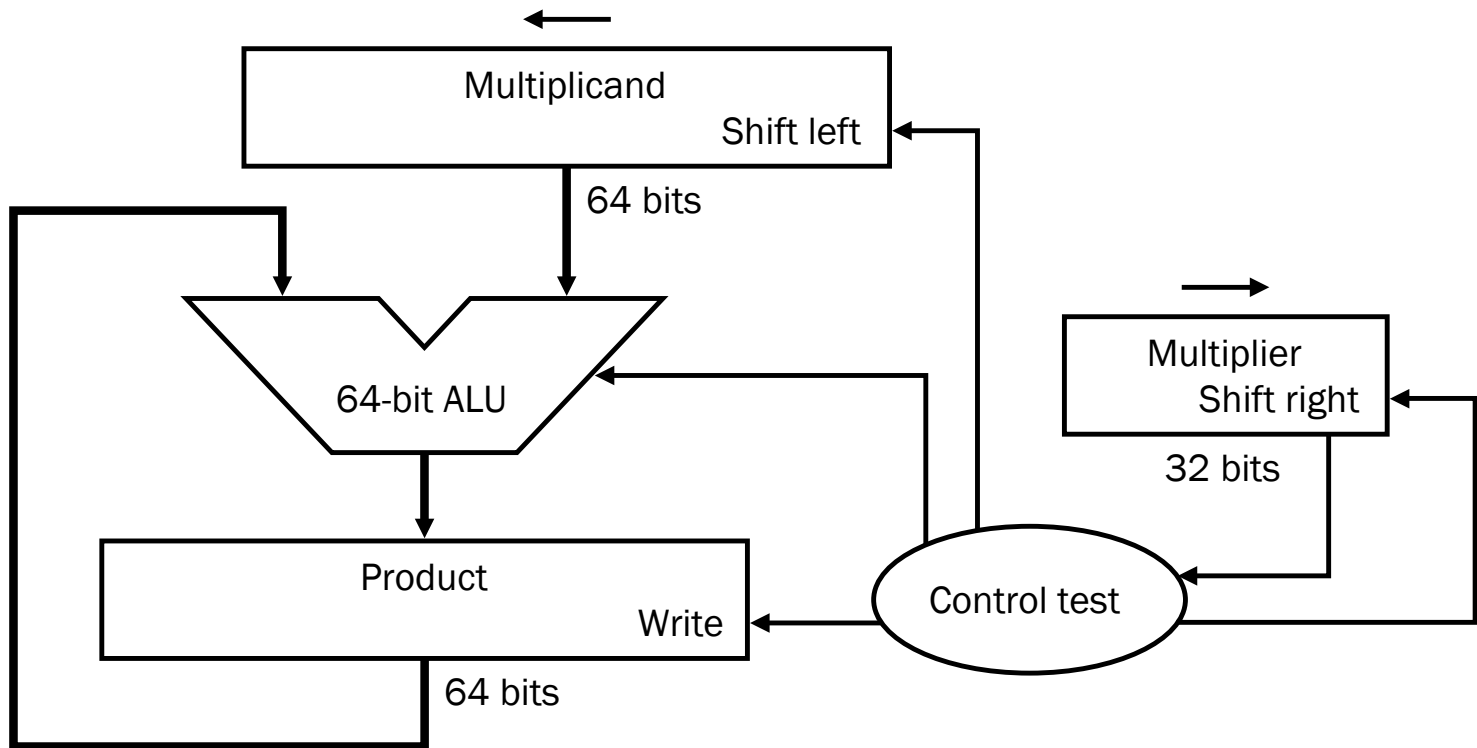
Multiplicand	1000
Multiplier	x <u>1001</u>
	1000
	0000
	0000
	<u>1000</u>
Product	1001000

- m bits x n bits = m+n bit product
- Binary makes it easy:
 - 0 => place 0 (0 x multiplicand)
 - 1 => place multiplicand (1 x multiplicand)
- we'll look at a couple of versions of multiplication hardware

MULTIPLY HARDWARE

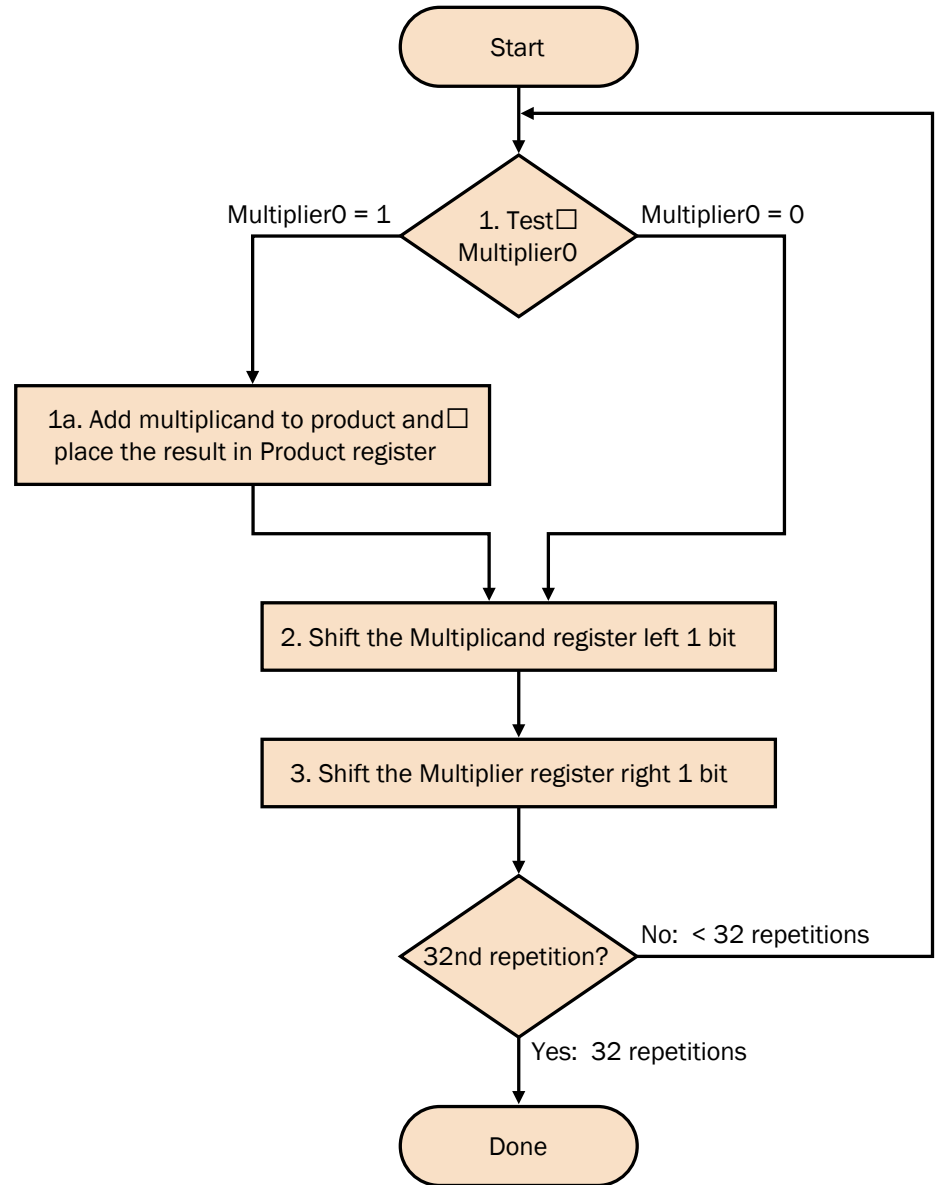
Version 1

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiply Algorithm Version 1

Multiplier	Multiplicand	Product
0101	0000 0110	0000 0000



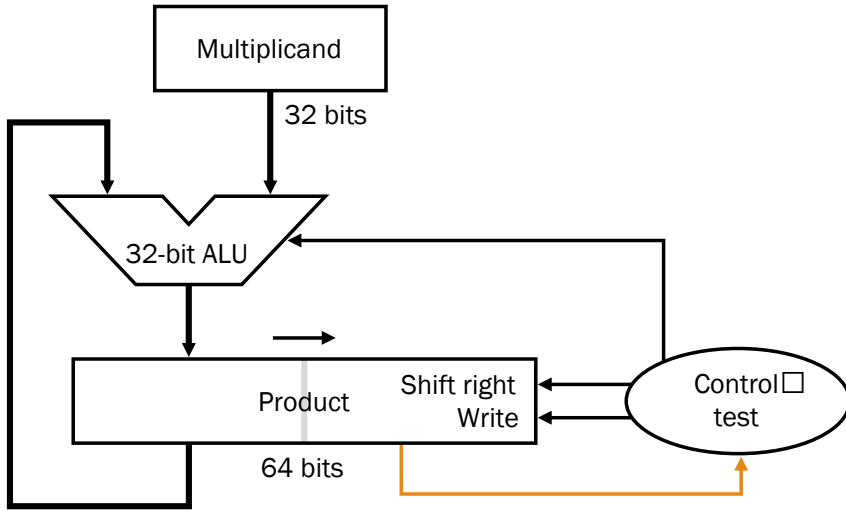
Observations on Multiply Version 1

- 1 clock per cycle \Rightarrow 100 clocks per multiply
 - Ratio of multiply to add 100:1
- 1/2 bits in multiplicand always 0
 - \Rightarrow 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
 - \Rightarrow least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Wasted space (zeroes) in product register exactly matches meaningful bits of multiplier at all times. Combine?

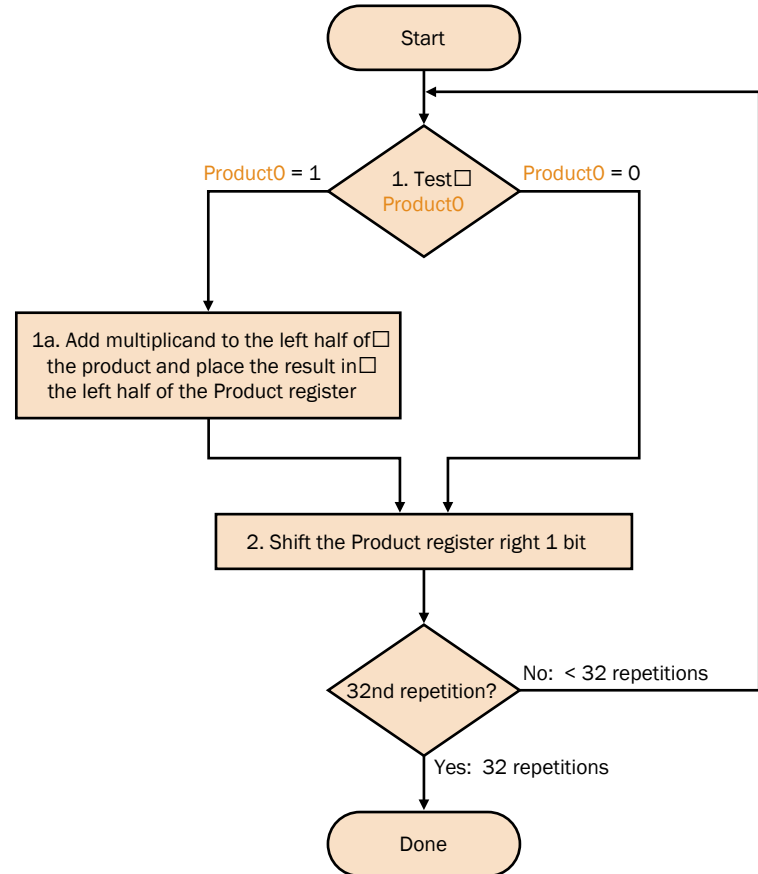
MULTIPLY HARDWARE

Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



Multiplicand Product
 0110 0000 0101



Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- 32-bit adder
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- What about signed multiplication?
 - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)

Key Points

- Instruction Set drives the ALU design
- ALU performance, CPU clock speed driven by adder delay
- Multiply is expensive

Divide: Paper & Pencil

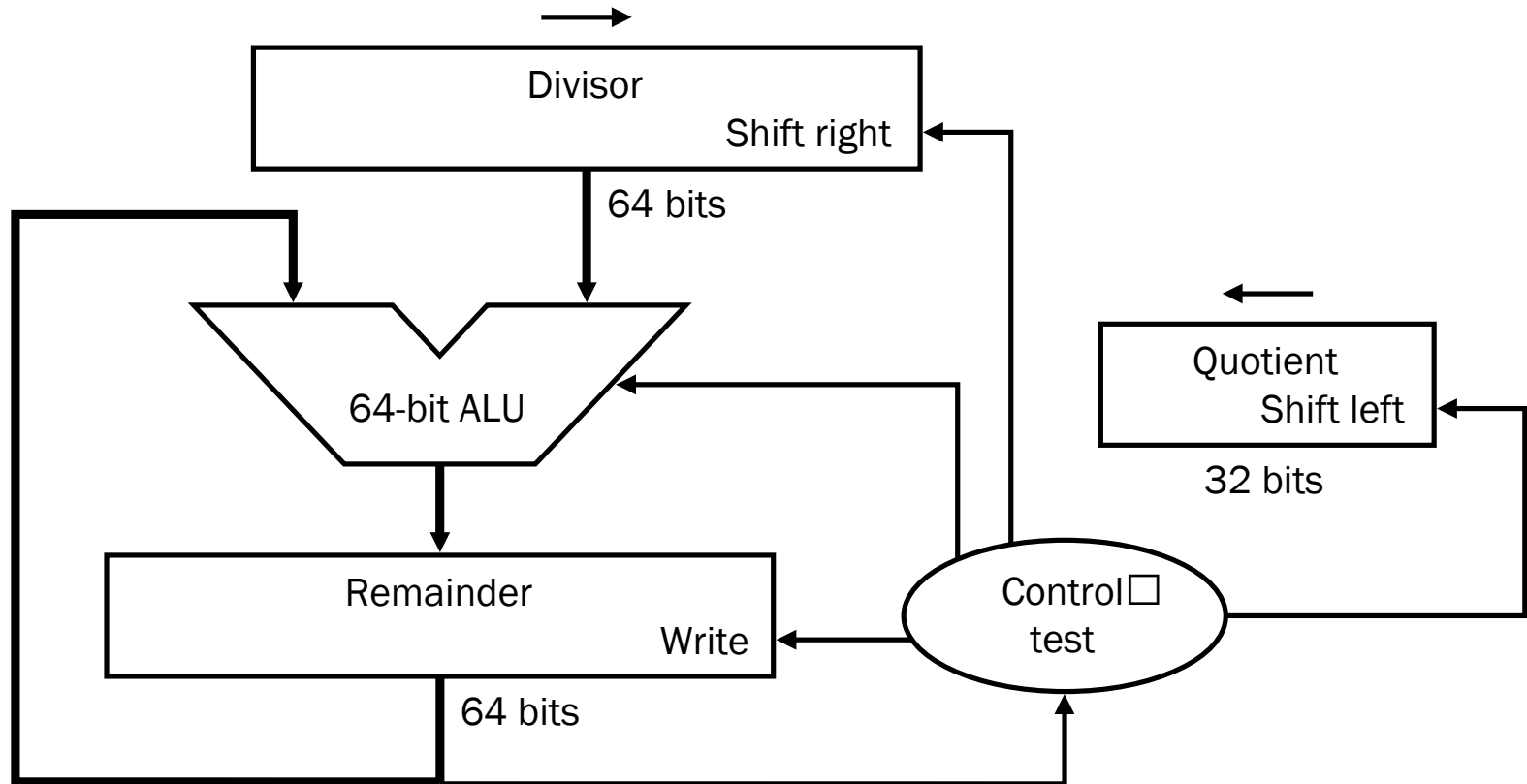
	1001	Quotient
Divisor 1000	$\overline{1001010}$	Dividend
	$\underline{-1000}$	
	10	
	101	
	1010	
	$\underline{-1000}$	
	10	Remainder

- See how big a number can be subtracted, creating quotient bit on each step
 - Binary \Rightarrow 1 * divisor or 0 * divisor
- Dividend = Quotient x Divisor + Remainder

DIVIDE HARDWARE

Version 1

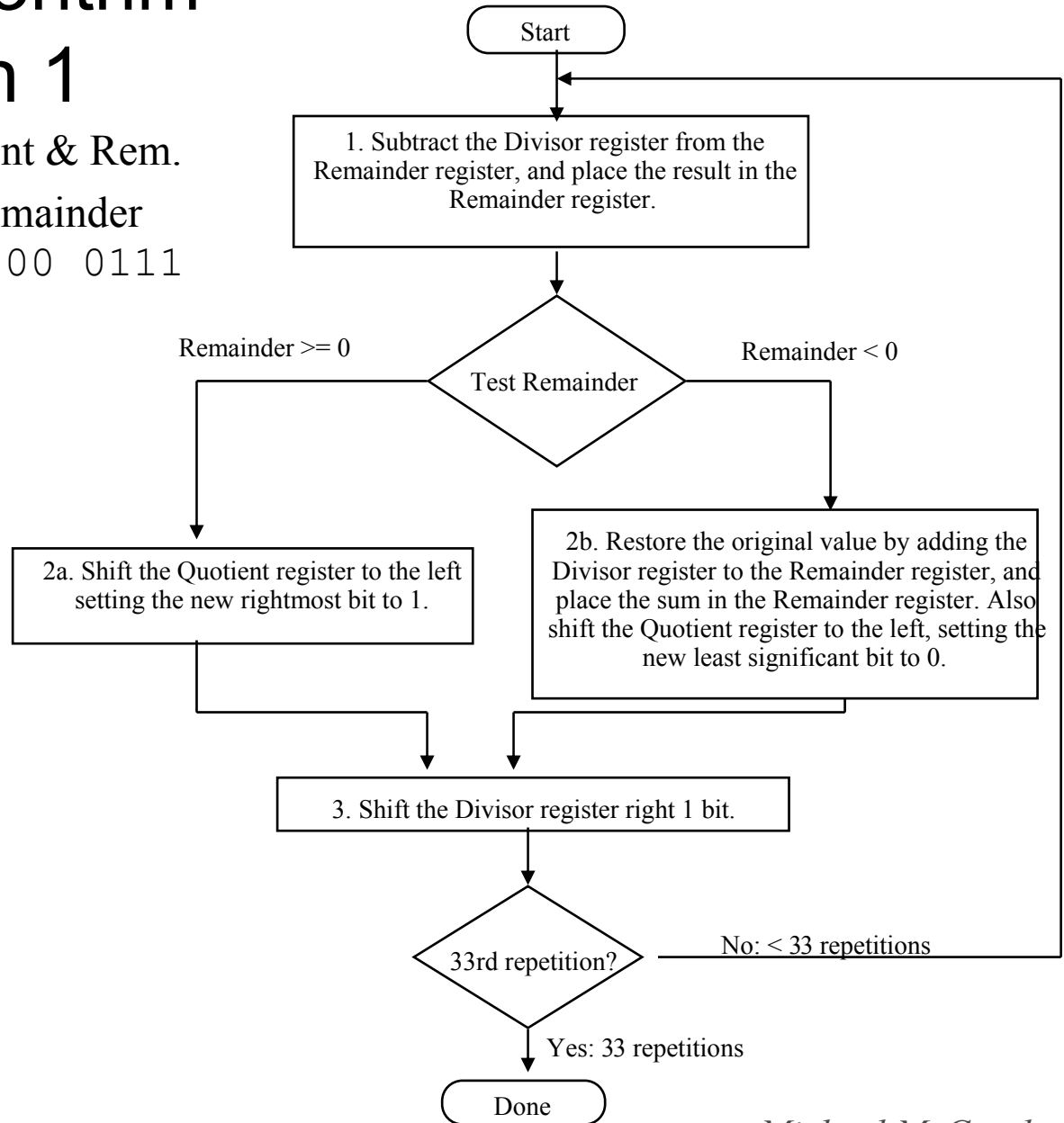
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

- Takes $n+1$ steps for n -bit Quotient & Rem.

Quotient	Divisor	Remainder
0000	0011 0000	0000 0111



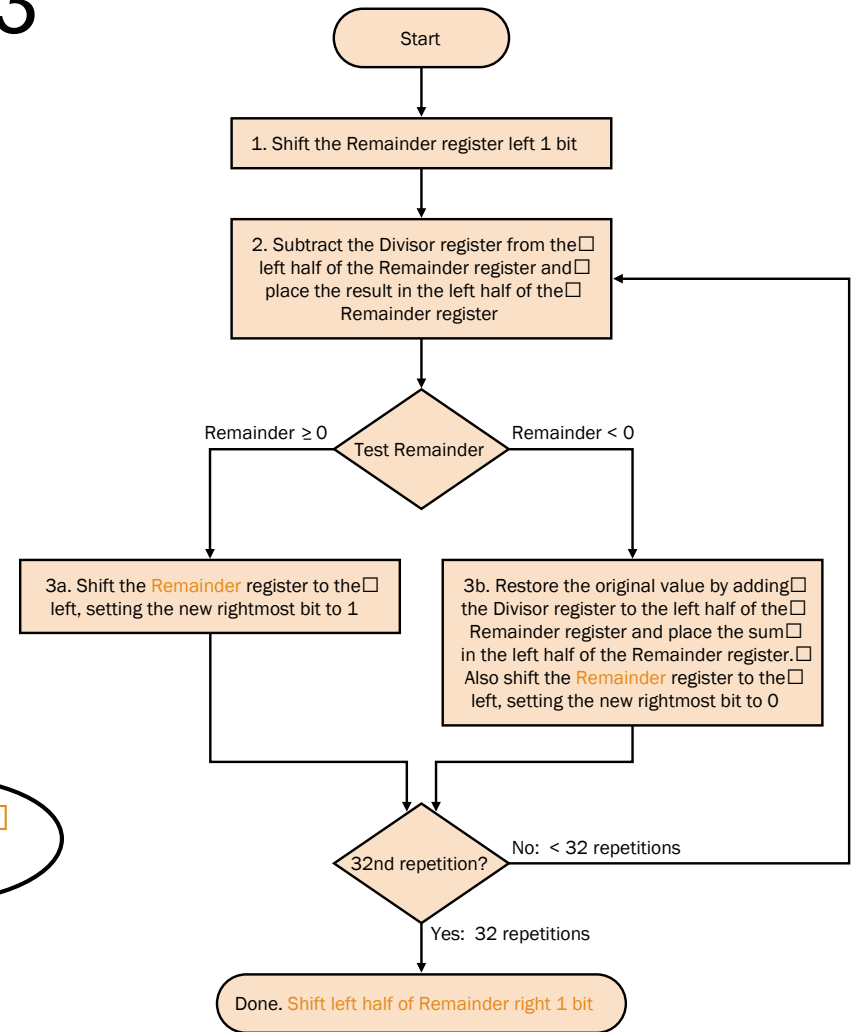
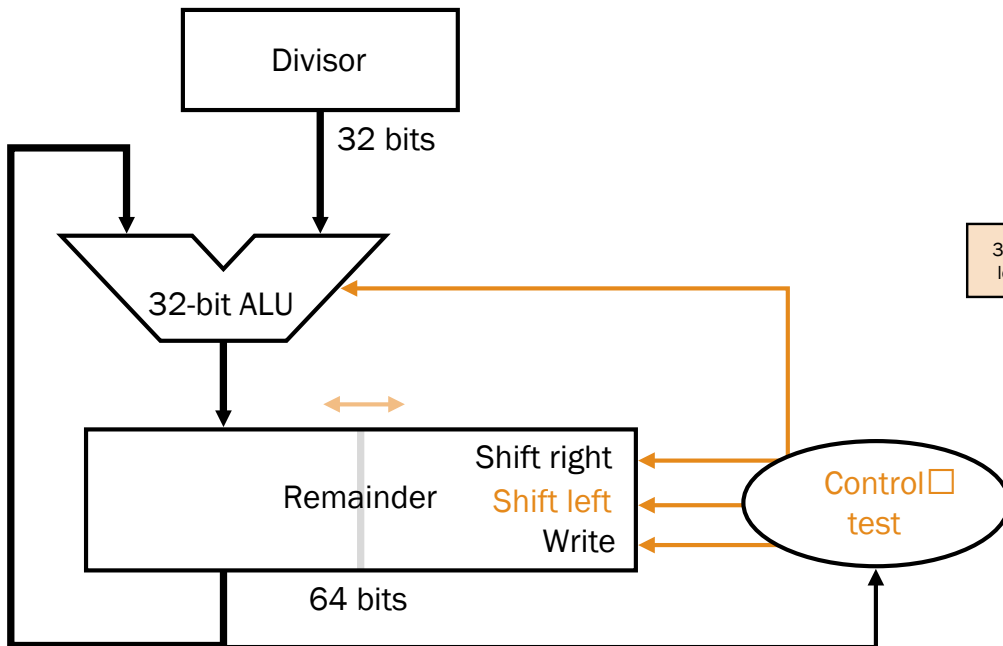
Improvements to Version 1

- Observation: the divisor register only ever has $\frac{1}{2}$ of its bits containing useful info. Let's make it only 32 bits and shift remainder left instead (same effect).
- If we are shifting the remainder register left, and shifting in zeros from the right (the symmetric thing to shifting the divisor register to the right and zeros from the left as in Version 1) and doing our subtract operations on only the upper $\frac{1}{2}$ of the “remainder” register then let's just use the lower $\frac{1}{2}$ for the quotient
- You have to shift once just to have anything meaningful to subtract (first step of Version 1 can't generate a 1 in the quotient)

DIVIDE HARDWARE

Version 3

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

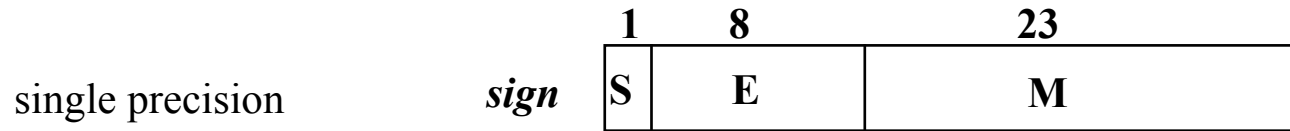


Observations on Divide Version 3

- The consequence of combining remainder and quotient registers, and of shifting first and then subtracting (as opposed to subtracting and then shifting as in Version 1) is that remainder is shifted left one time too many.
- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
 - Note: Dividend and Remainder must have same sign
 - Note: Quotient negated if Divisor sign & Dividend sign disagree

Floating-Point Numbers

Representation of floating point numbers in IEEE 754 standard:



exponent:

excess 127

binary integer

(actual exponent is $e = E - 127$)

mantissa:

sign + magnitude, normalized

binary significand w/ hidden

integer bit: 1.M

$$N = (-1)^S 2^{E-127} (1.M) \quad 0 < E < 255$$

$$0 = 0\ 00000000\ 0\dots 0 \quad -1.5 = 1\ 01111111\ 10\dots 0$$

$$325 = 101000101 \times 2^0 = 1.01000101 \times 2^8$$

$$= 0\ 10000111\ 010001010000000000000000$$

$$.02 = .0011001101100\dots \times 2^0 = 1.1001101100\dots \times 2^{-3}$$

$$= 0\ 01111100\ 1001101100\dots$$

- range of about 2×10^{-38} to 2×10^{38}
- always normalized (so always leading 1, thus never shown)
- special representation of 0 (E = 00000000) (why?)
- can do integer compare for greater-than, sign

Key Points

- Multiplication and division take much longer than addition, requiring multiple addition steps.
- Floating Point extends the range of numbers that can be represented, at the expense of precision (accuracy).
- FP operations are very similar to integer, but with pre- and post-processing.
- Rounding implementation is critical to accuracy over time.

Single Cycle Data Path and Control

- Be able to draw missing parts on datapath (connect the components)
- Be able to fill in control

