

Number Systems and Arithmetic

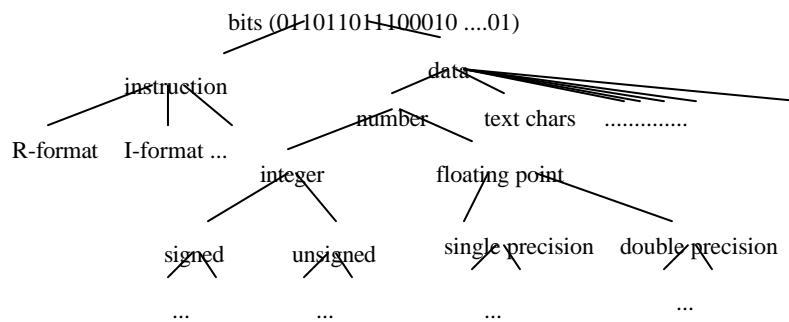
or

Computers go to elementary school

CSE 141

Allan Snively

What do all those bits mean now?



CSE 141

Allan Snively

Questions About Numbers

- How do you represent
 - negative numbers?
 - fractions?
 - really large numbers?
 - really small numbers?
- How do you
 - do arithmetic?
 - identify errors (e.g. overflow)?
- What is an ALU and what does it look like?
 - ALU=arithmetic logic unit

CSE 141

Allan Snively

Introduction to Binary Numbers

Consider a 4-bit binary number

Decimal	Binary	Decimal	Binary
0	0000	4	0100
1	0001	5	0101
2	0010	6	0110
3	0011	7	0111

Examples of binary arithmetic:

$$\begin{array}{r} 3 + 2 = 5 \\ \begin{array}{cccc} & & 1 & \\ & & \swarrow & \\ 0 & 0 & 1 & 1 \\ + 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \end{array} \end{array}$$

$$\begin{array}{r} 3 + 3 = 6 \\ \begin{array}{cccc} & & 1 & 1 \\ & & \swarrow & \swarrow \\ 0 & 0 & 1 & 1 \\ + 0 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 0 \end{array} \end{array}$$

CSE 141

Allan Snively

Negative Numbers?

- We would like a number system that provides
 - obvious representation of 0,1,2...
 - uses adder for addition
 - single value of 0
 - equal coverage of positive and negative numbers
 - easy detection of sign
 - easy negation

Some Alternatives

- Sign Magnitude -- MSB is sign bit, rest the same
 - 1 == 1001
 - 5 == 1101
- One's complement -- flip all bits to negate
 - 1 == 1110
 - 5 == 1010

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

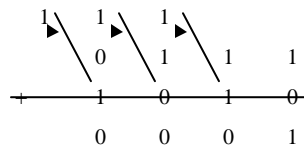
CSE 141

Allan Snively

Two's Complement Arithmetic

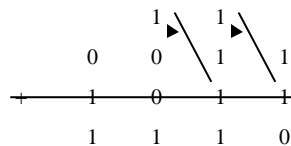
Decimal	2's Complement Binary	Decimal	2's Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

- Examples: $7 - 6 = 7 + (-6) = 1$



CSE 141

- $3 - 5 = 3 + (-5) = -2$



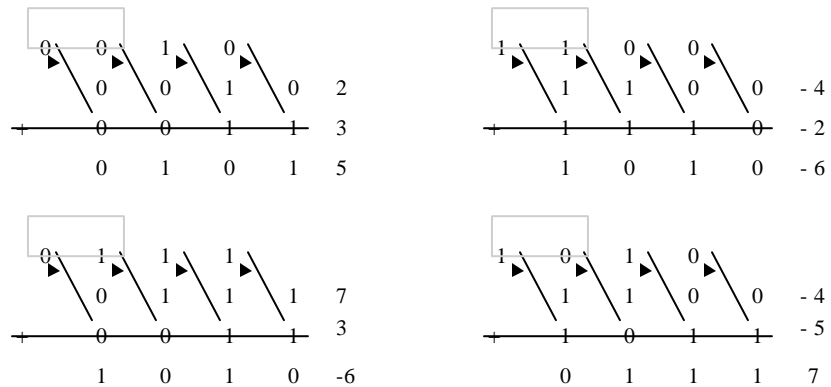
Allan Snively

Some Things We Want To Know About Our Number System

- negation
- sign extension
 - +3 => 0011, 00000011, 0000000000000011
 - -3 => 1101, 11111101, 1111111111111101
- overflow detection

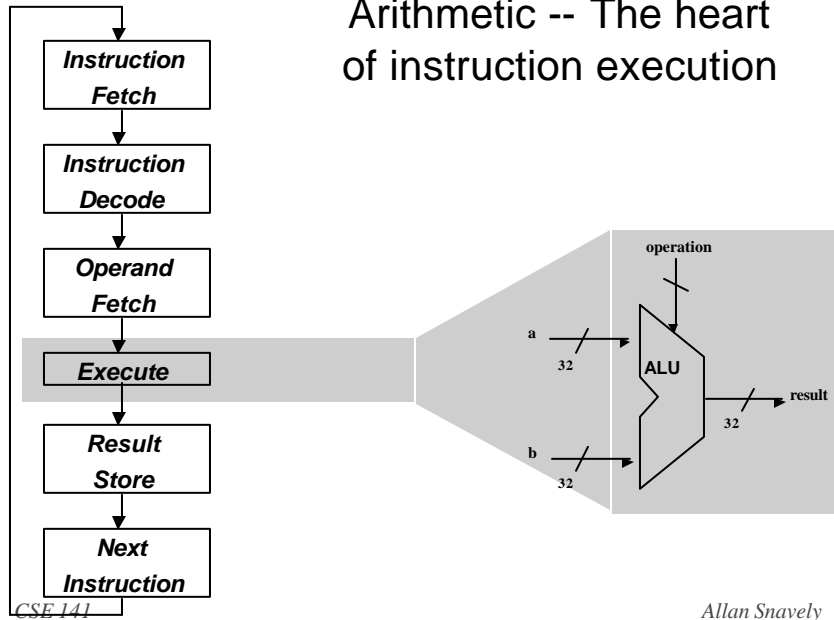
0101	5
+ 0110	6

Overflow Detection



So how do we detect overflow?

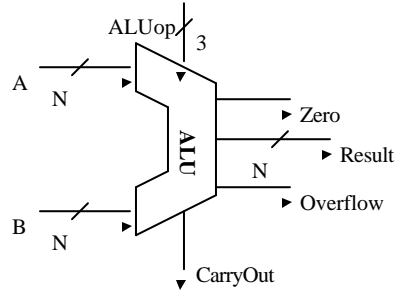
Arithmetic -- The heart of instruction execution



CSE 141

Allan Snively

Designing an Arithmetic Logic Unit



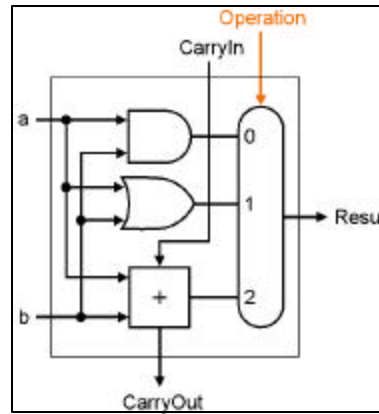
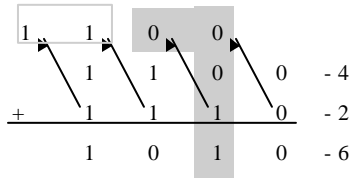
ALU Control Lines (ALUop)	Function
- 000	And
- 001	Or
- 010	Add
- 110	Subtract
- 111	Set-on-less-than

CSE 141

Allan Snively

A One Bit ALU

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

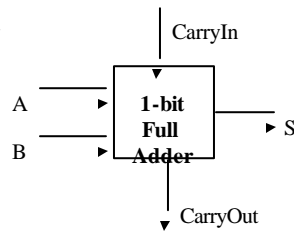


CSE 141

Allan Snively

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

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

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

CSE 141

Allan Snively

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

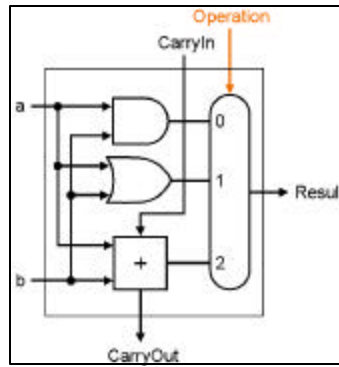
$$\text{Sum} = (!A \& !B \& \text{CarryIn}) \mid (!A \& B \& !\text{CarryIn}) \mid (A \& !B \& !\text{CarryIn}) \mid (A \& B \& \text{CarryIn})$$

CSE 141

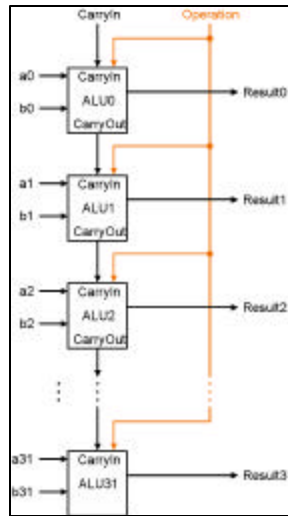
Allan Snively

A 32-bit ALU

1-bit ALU



32-bit ALU

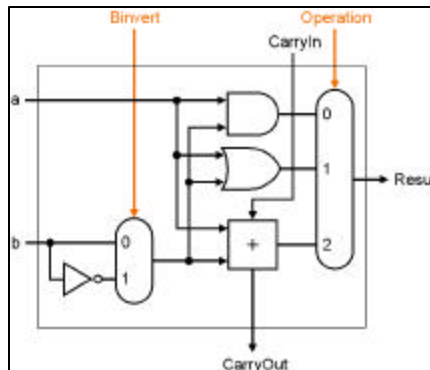


CSE 141

Snively

How About Subtraction?

- Keep in mind the following:
 - $(A - B)$ is the same as: $A + (-B)$
 - 2's Complement negate: Take the inverse of every bit and add 1
- Bit-wise inverse of B is $\neg B$:
 - $A - B = A + (-B) = A + (\neg B + 1) = A + \neg B + 1$

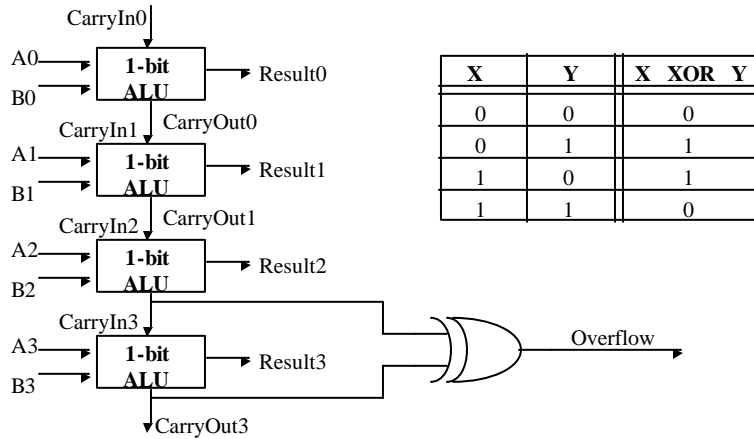


CSE 141

Allan Snively

Overflow Detection Logic

- Carry into MSB \neq Carry out of MSB
 - For a N-bit ALU: $\text{Overflow} = \text{CarryIn}[N - 1] \text{ XOR } \text{CarryOut}[N - 1]$

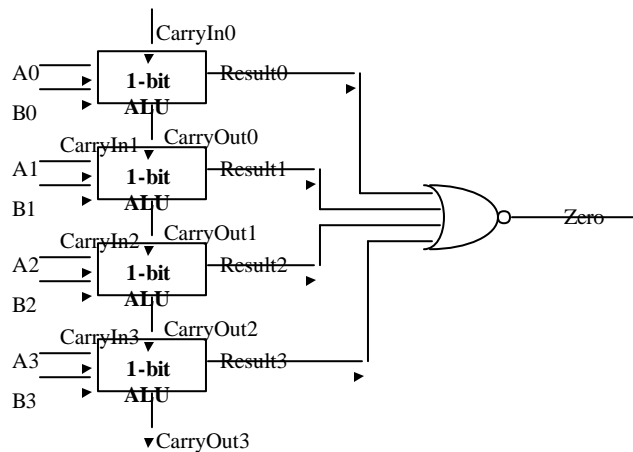


CSE 141

Allan Snively

Zero Detection Logic

- Zero Detection Logic is just one BIG NOR gate
 - Any non-zero input to the NOR gate will cause its output to be zero



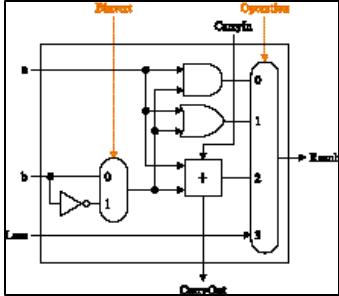
CSE 141

Allan Snively

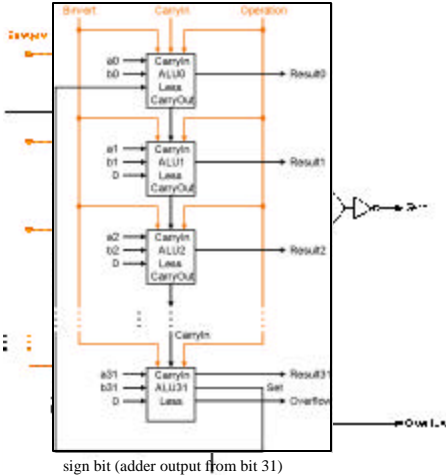
Set-on-less-than

- Do a subtract
- use sign bit
 - route to bit 0 of result
 - all other bits zero

Full ALU

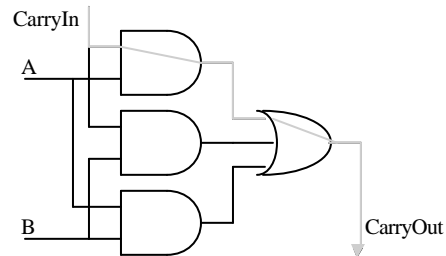
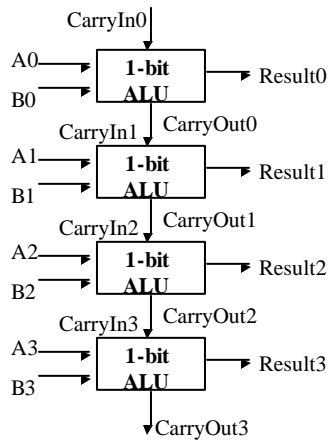


what signals accomplish:
 neg oper
 add?
 sub?
 and?
 or?
 beq?
 slt?



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



Problem: ripple carry adder is slow

- 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?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

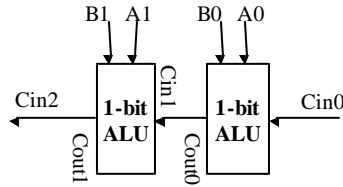
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \quad c_2 =$$

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

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

Not feasible! Why?

Carry Lookahead Adders



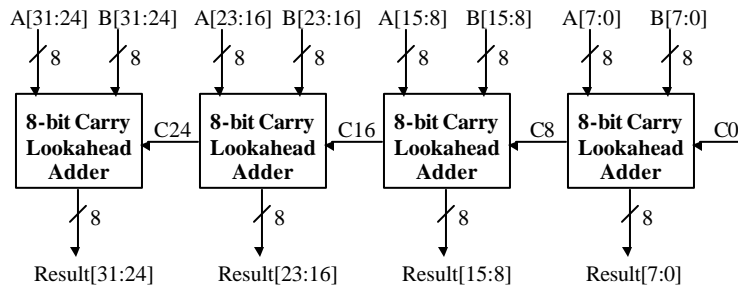
- We'll define two new terms, based on the relationship between C_{in} and C_{out}
 - **Generate** Carry at Bit i $g_i = A_i \& B_i$
 - **Propagate** Carry via Bit i $p_i = A_i \text{ or } B_i$

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:
 - $C_{in1} = g_0 \mid (p_0 \& C_{in0})$
 - $C_{in2} = g_1 \mid (p_1 \& g_0) \mid (p_1 \& p_0 \& C_{in0})$
 - $C_{in3} = g_2 \mid (p_2 \& g_1) \mid (p_2 \& p_1 \& g_0) \mid (p_2 \& p_1 \& p_0 \& C_{in0})$
- 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

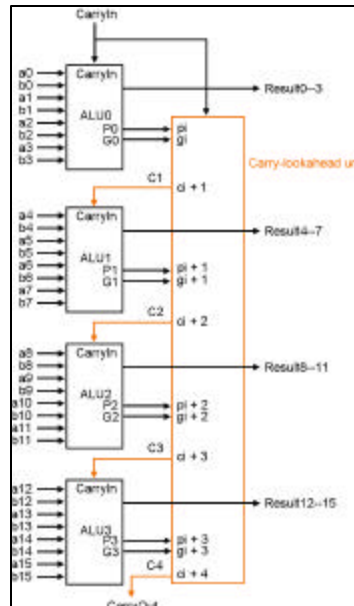
- It is very expensive to build a “full” carry lookahead adder
 - Just imagine the length of the equation for C_{in31}
- Common practices:
 - Connect several N-bit Lookahead Adders to form a big adder
 - Example: connect four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder



CSE 141

Worst-case delay??

Allan Snively



CSE 141

Allan Snively

Hierarchical Carry-Lookahead Adders

$$P_0 = p_0 \& p_1 \& p_2 \& p_3$$

Worst-case delay??

MULTIPLY

- Paper and pencil example:

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

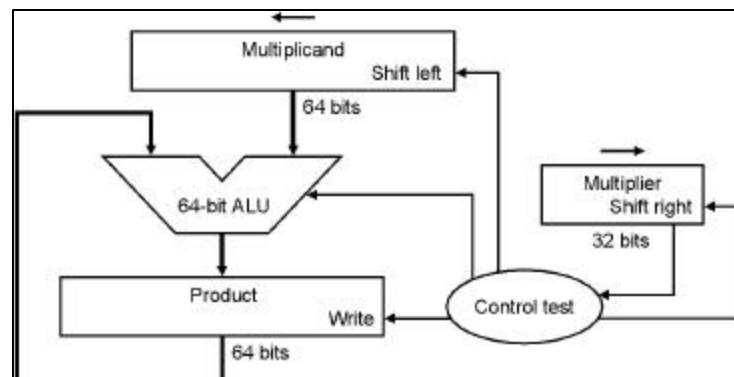
- m bits x n bits = m+n bit product
- Binary makes it easy:
 - 0 => place 0 (0 x multiplicand)
 - 1 => place multiplicand (1 x multiplicand)
- we'll look at a couple of versions of multiplication hardware

CSE 141

Allan Snively

MULTIPLY HARDWARE Version 1

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

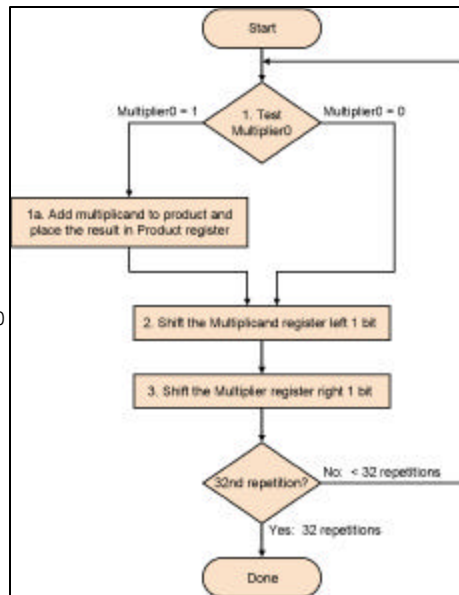


CSE 141

Allan Snively

Multiply Algorithm Version 1

Multiplier	Multiplicand	Product
0101	0000 0110	0000 0000



CSE 141

Allan Snively

Observations on Multiply Version 1

- 1 clock per cycle => 100 clocks per multiply
 - Ratio of multiply to add 100:1
- 1/2 bits in multiplicand always 0
=> 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
=> 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?

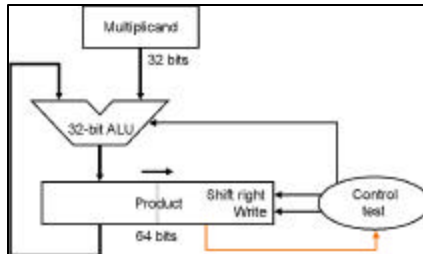
CSE 141

Allan Snively

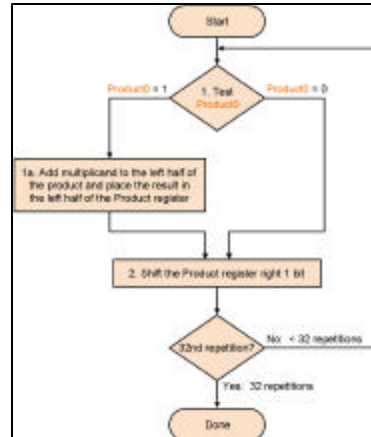
MULTIPLY HARDWARE

Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (Q-bit Multiplierreg)



Multiplicand Product
0110 0000 0101



CSE 141

Allan Snively

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)

CSE 141

Allan Snively

Key Points

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