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?
  - When would we propagate the carry?
- Did we get rid of the ripple?

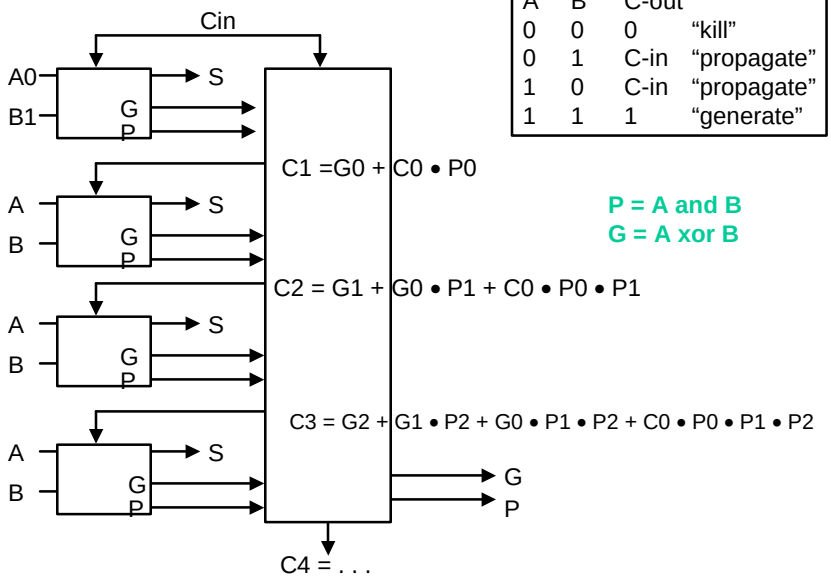
$g_i = a_i\ b_i$
$p_i = a_i + b_i$

$c_1 = g_0 + p_0c_0$
$c_2 = g_1 + p_1c_1$
$c_3 = g_2 + p_2c_2$
$c_4 = g_3 + p_3c_3$

$c_2 =$
$c_3 =$
$c_4 =$

Feasible! Why?

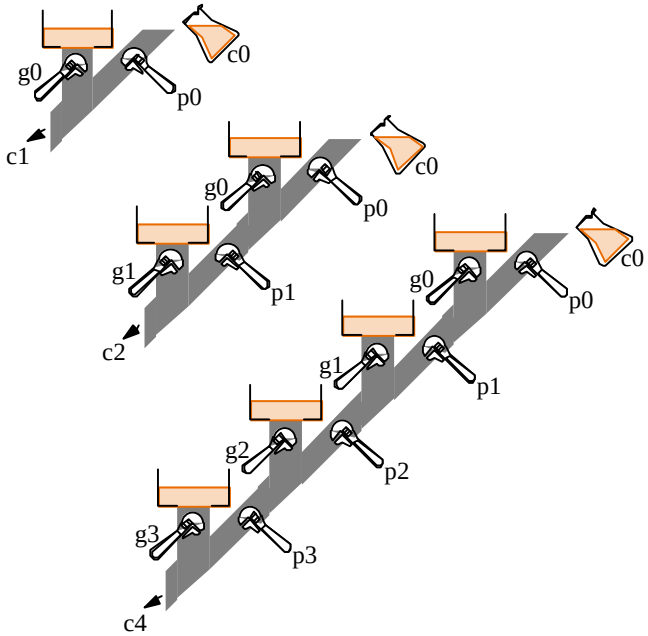
Carry Look Ahead



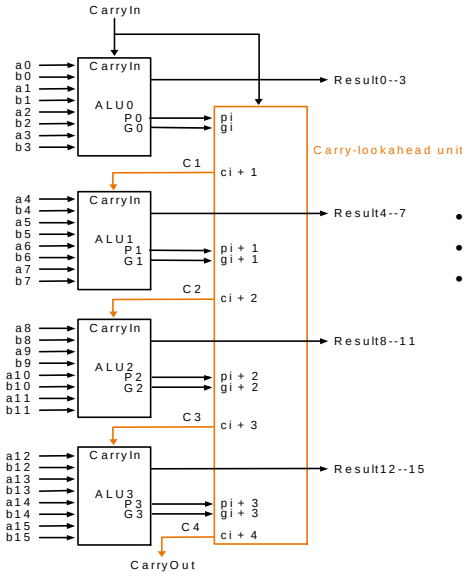
A	B	C-out	
0	0	0	“kill”
0	1	C-in	“propagate”
1	0	C-in	“propagate”
1	1	1	“generate”

P = A and B
G = A xor B

Plumbing as Carry Lookahead Analogy

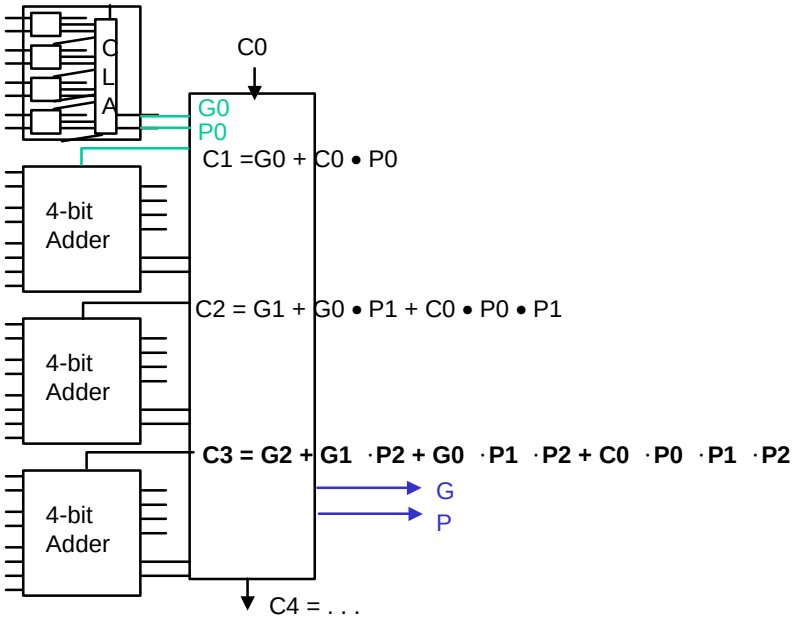


Use principle to build bigger adders

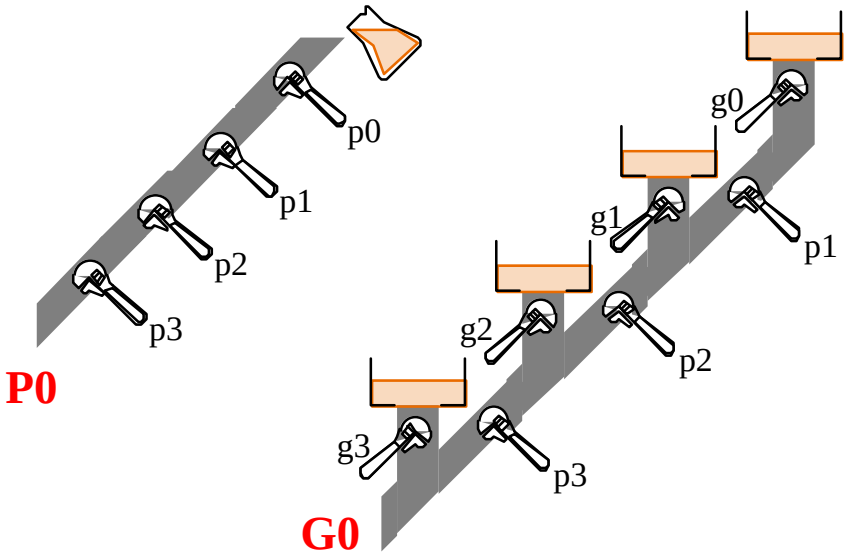


- Can't build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

Cascaded Carry Look-ahead (16-bit): Abstraction



2nd level Carry, Propagate as Plumbing



MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu $2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

MULTIPLY (unsigned)

Paper and pencil example (unsigned):

Multiplicand	**1000**
Multiplier	**1001**
	1000
	0000
	0000
	1000
Product	**01001000**

m bits x n bits = m+n bit product

Binary makes it easy:

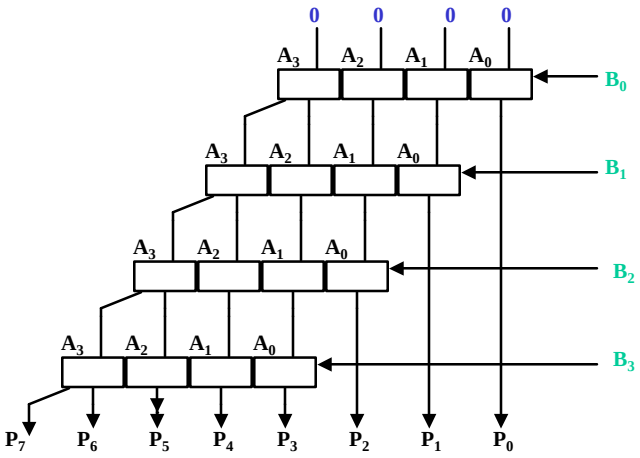
0 => place 0 (0 x multiplicand)

1 => place a copy (1 x multiplicand)

3 versions of multiply hardware & algorithm:

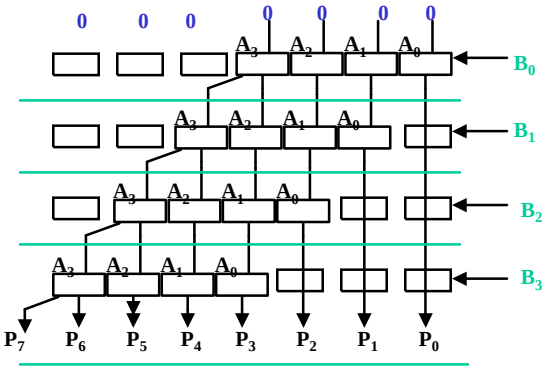
successive refinement

Unsigned Combinational Multiplier



- Stage i accumulates $A * 2^i$ if $B_i == 1$

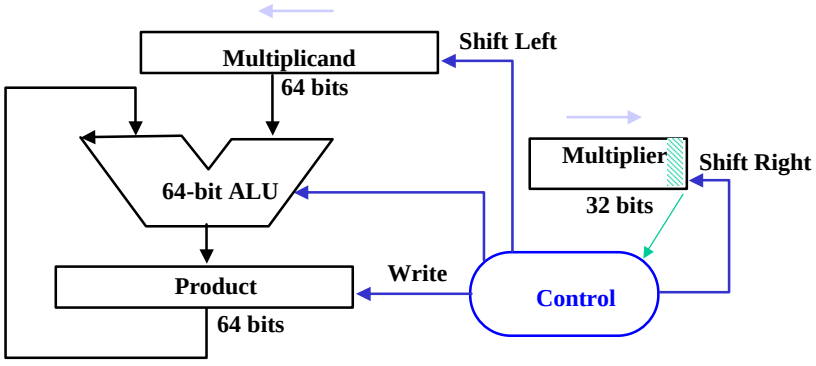
How does it work?



- at each stage shift A left (x 2)
- use next bit of B to determine whether to add in shifted multiplicand
- accumulate 2n bit partial product at each stage

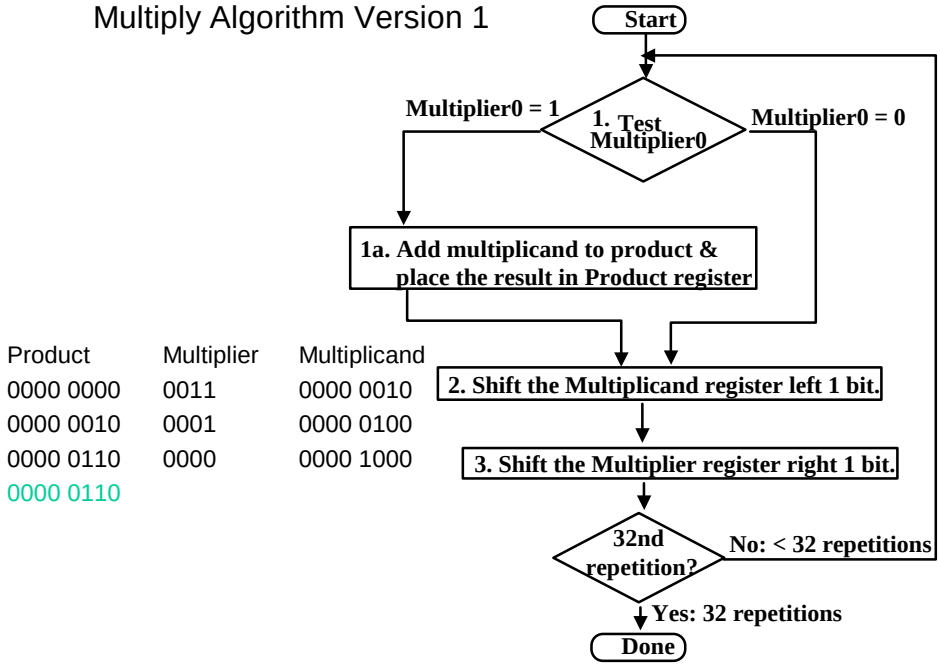
Unisigned shift-add multiplier (version 1)

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



Multiplier = datapath + control

Multiply Algorithm Version 1



Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

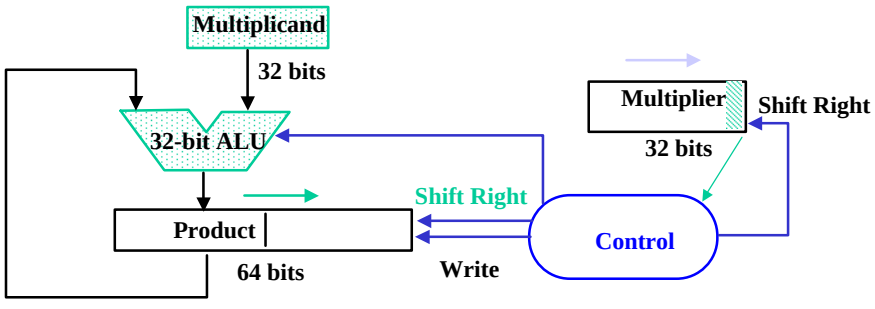
	M'ier: 0011	M'and: 0000 0010	P: 0000 0000
1a. 1=>P=P+Mcand	M'ier: 0011	Mcand: 0000 0010	P: 0000 0010
2. Shl Mcand	M'ier: 0011	Mcand: 0000 0100	P: 0000 0010
3. Shr M'ier	M'ier: 0001	Mcand: 0000 0100	P: 0000 0010
1a. 1=>P=P+Mcand	M'ier: 0001	Mcand: 0000 0100	P: 0000 0110
2. Shl Mcand	M'ier: 0001	Mcand: 0000 1000	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0000 1000	P: 0000 0110
1. 0=>nop	M'ier: 0000	Mcand: 0000 1000	P: 0000 0110
2. Shl Mcand	M'ier: 0000	Mcand: 0001 0000	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0001 0000	P: 0000 0110
1. 0=>nop	M'ier: 0000	Mcand: 0001 0000	P: 0000 0110
2. Shl Mcand	M'ier: 0000	Mcand: 0010 0000	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0010 0000	P: 0000 0110

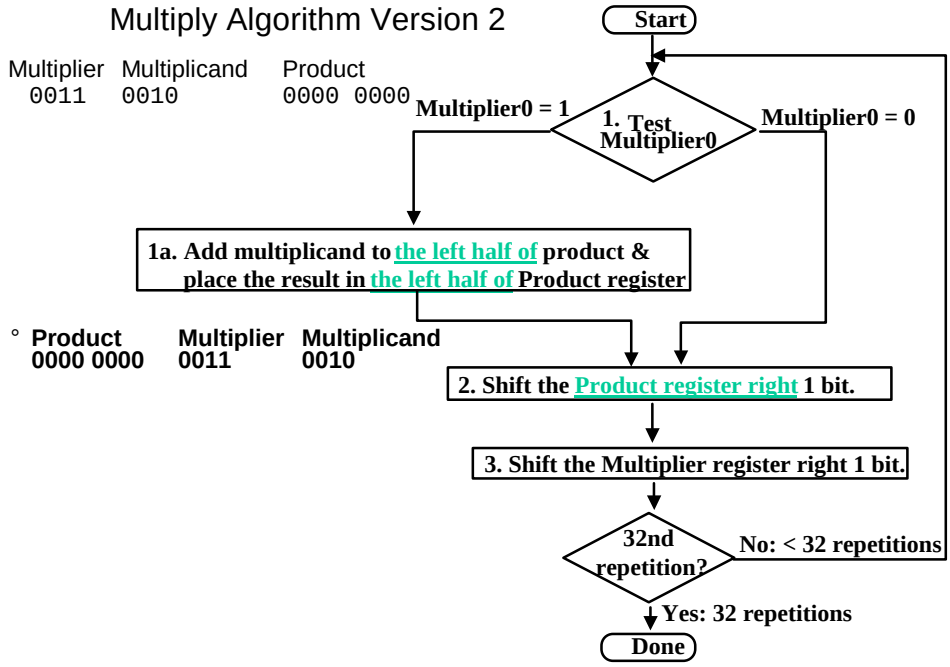
Observations on Multiply Version 1

- 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?

MULTIPLY HARDWARE Version 2

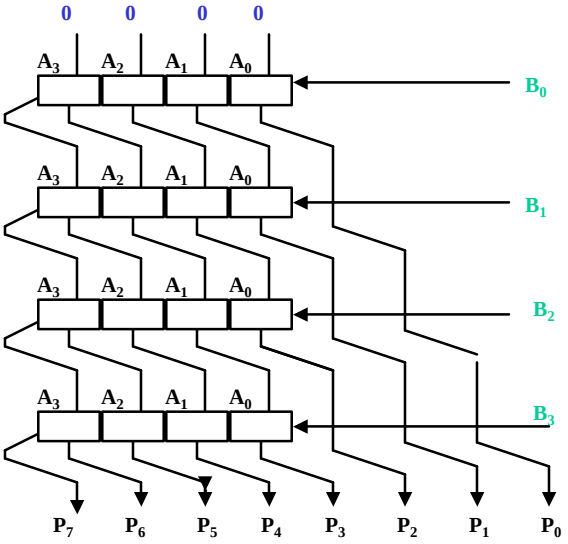
- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg





	M'ier: 0011	Mcand: 0010	P: 0000 0000
1a. 1=>P=P+Mcand	M'ier: 0011	Mcand: 0010	P: 0010 0000
2. Shr P	M'ier: 0011	Mcand: 0010	P: 0001 0000
3. Shr M'ier	M'ier: 0001	Mcand: 0010	P: 0001 0000
1a. 1=>P=P+Mcand	M'ier: 0001	Mcand: 0010	P: 0011 0000
2. Shr P	M'ier: 0001	Mcand: 0010	P: 0001 1000
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0001 1000
1. 0=>nop	M'ier: 0000	Mcand: 0010	P: 0001 1000
2. Shr P	M'ier: 0000	Mcand: 0010	P: 0000 1100
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0000 1100
1. 0=>nop	M'ier: 0000	Mcand: 0010	P: 0000 1100
2. Shr P	M'ier: 0000	Mcand: 0010	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0000 0110

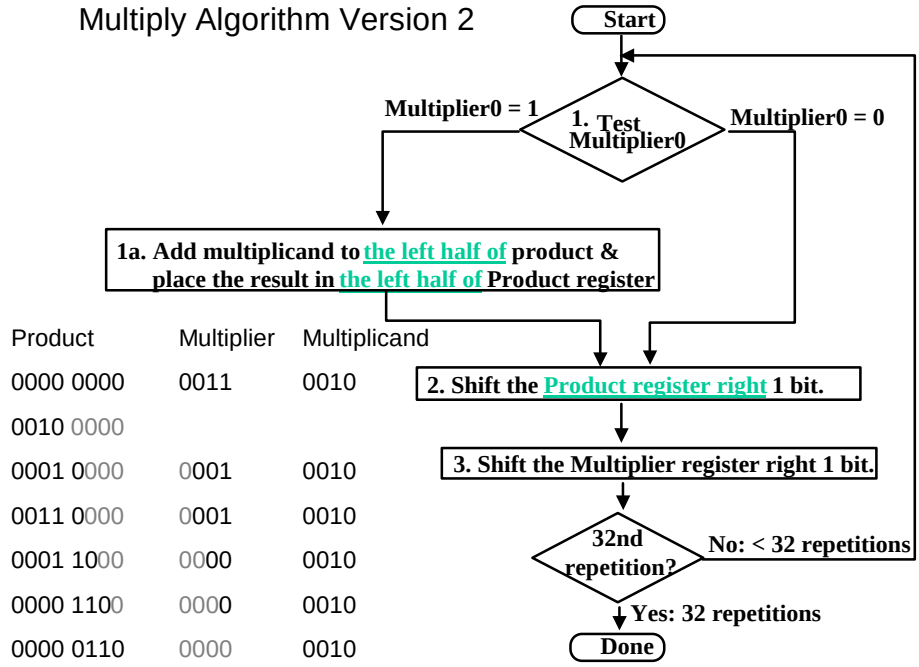
What's going on?



- Multiplicand stays still and product moves right

Break

- 5-minute Break/ Do it yourself Multiply
- Multiplier 0011 Multiplicand 0010 Product 0000 0000



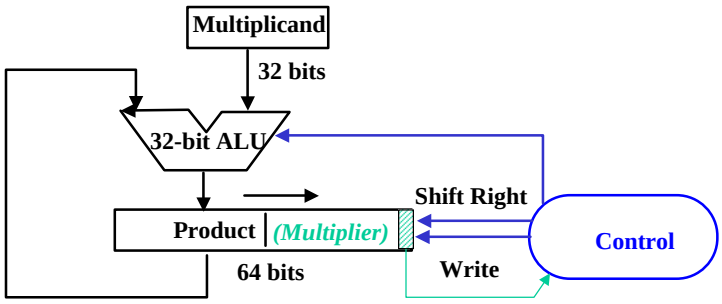
	M'ier: 0011	Mcand: 0010	P: 0000 0000
1a. 1=>P=P+Mcand	M'ier: 0011	Mcand: 0010	P: 0010 0000
2. Shr P	M'ier: 0011	Mcand: 0010	P: 0001 0000
3. Shr M'ier	M'ier: 0001	Mcand: 0010	P: 0001 0000
1a. 1=>P=P+Mcand	M'ier: 0001	Mcand: 0010	P: 0011 0000
2. Shr P	M'ier: 0001	Mcand: 0010	P: 0001 1000
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0001 1000
1. 0=>nop	M'ier: 0000	Mcand: 0010	P: 0001 1000
2. Shr P	M'ier: 0000	Mcand: 0010	P: 0000 1100
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0000 1100
1. 0=>nop	M'ier: 0000	Mcand: 0010	P: 0000 1100
2. Shr P	M'ier: 0000	Mcand: 0010	P: 0000 0110
3. Shr M'ier	M'ier: 0000	Mcand: 0010	P: 0000 0110

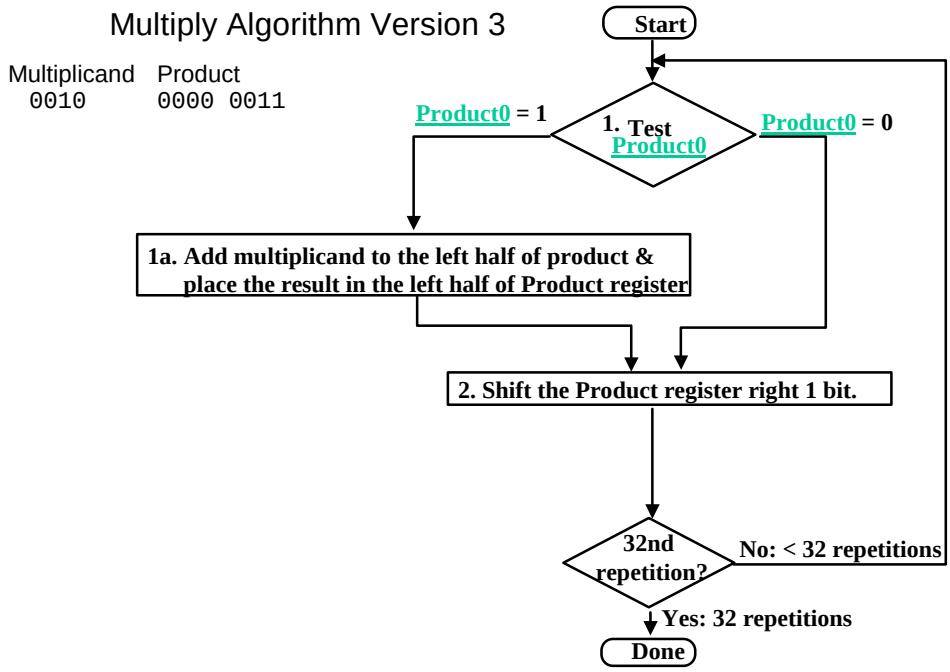
Observations on Multiply Version 2

- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

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





		Mcand: 0010 P: 0000 0011
1a. 1=>P=P+Mcand	Mcand: 0010	P: 0010 0011
2. Shr P		Mcand: 0010 P: 0001 0001
1a. 1=>P=P+Mcand		Mcand: 0010 P: 0011 0001
2. Shr P		Mcand: 0010 P: 0001 1000
1. 0=>nop		Mcand: 0010 P: 0001 1000
2. Shr P		Mcand: 0010 P: 0000 1100
1. 0=>nop		Mcand: 0010 P: 0000 1100
2. Shr P		Mcand: 0010 P: 0000 0110

Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- How can you make it faster?
- 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)
  - – apply definition of 2's complement
    - need to sign-extend partial products and subtract at the end
  - – Booth's Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
    - can handle multiple bits at a time

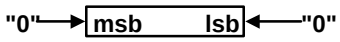
MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

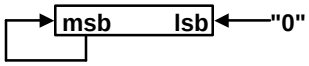
Shifters

Two kinds:

***logical*-- value shifted in is always "0"**



***arithmetic*-- on right shifts, sign extend**



Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Floating-Point

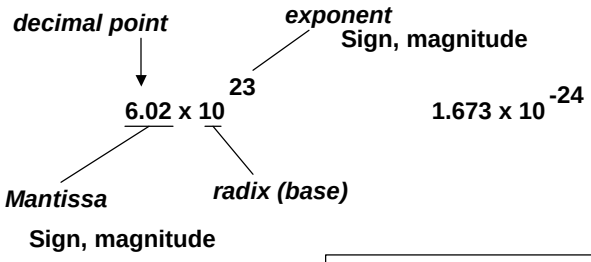
What can be represented in N bits?

- Unsigned 0 to 2^N
- 2s Complement -2^{N-1} to $2^{N-1} - 1$
- But, what about?
  - very large numbers?
    9,349,398,989,787,762,244,859,087,678
  - very small number?
    0.00000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π

Floating Point

- 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

Recall Scientific Notation



IEEE F.P. $\pm 1.M \times 2^{e - 127}$

- Issues:
  - Arithmetic (+, -, *, /)
  - Representation, Normal form
  - Range and Precision
  - Rounding
  - Exceptions (e.g., divide by zero, overflow, underflow)
  - Errors
  - Properties (negation, inversion, if A < B then A - B < 0)

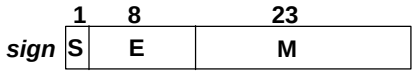
IEEE 754 floating-point standard

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

IEEE 754 Standard

Representation of floating point numbers in IEEE 754 standard:

single precision



exponent: **bias 127 binary integer**

mantissa: **sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M**

actual exponent is e = E - 127

0 < E < 255

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

0 = 0 00000000 0 . . . 0 **-1.5 = 1 01111111 10 . . . 0**

Magnitude of numbers that can be represented is in the range:

2^{-126} (1.0) to 2^{127} ($2 - 2^{23}$)

which is approximately:

1.8×10^{-38} to 3.40×10^{38}

Floating Point Complexities

- Operations are somewhat more complicated
- 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).
- Next class: floating pt arithmetic, rounding
- After that we are ready to move on (and implement the processor)