L06: ALU Design

Adapted from:

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

Overview

- Integer Multiplier
- Shifter
- Integer Divider
- Floating Point Representation & Operations (self study)

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

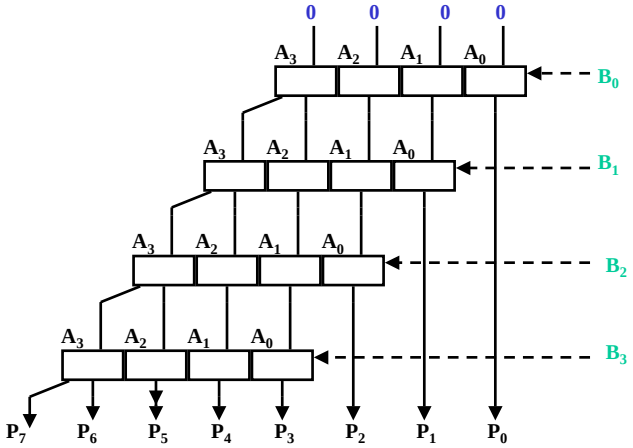
MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

```
Multiplicand    1000
Multiplier      1001
                1000
               0000
              0000
             1000
Product      01001000
```

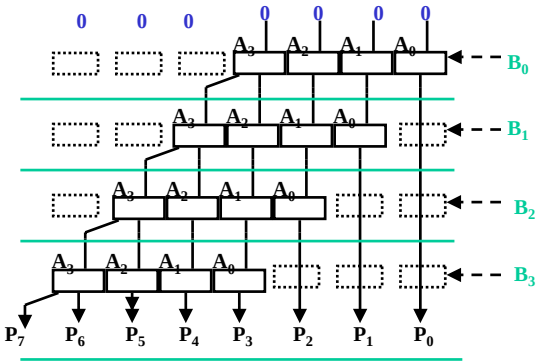
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
- 4 versions of multiply hardware & algorithm:
  - successive refinement

Unsigned Combinational Multiplier (ver 0)



- Stage i accumulates $A * 2^i$ if $B_i == 1$
- Q: How much hardware for 32 bit multiplier? Critical path?

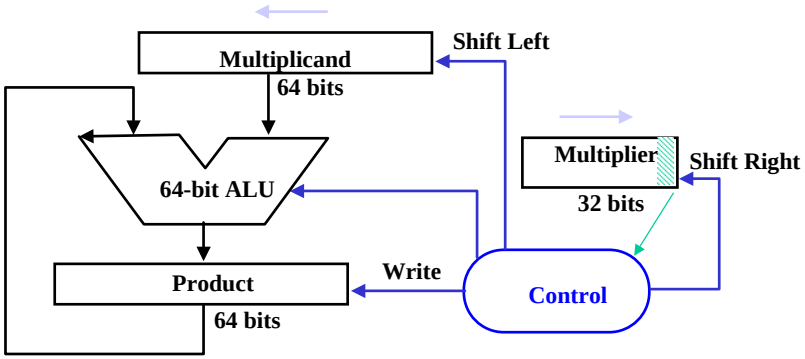
How does it work?



- 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

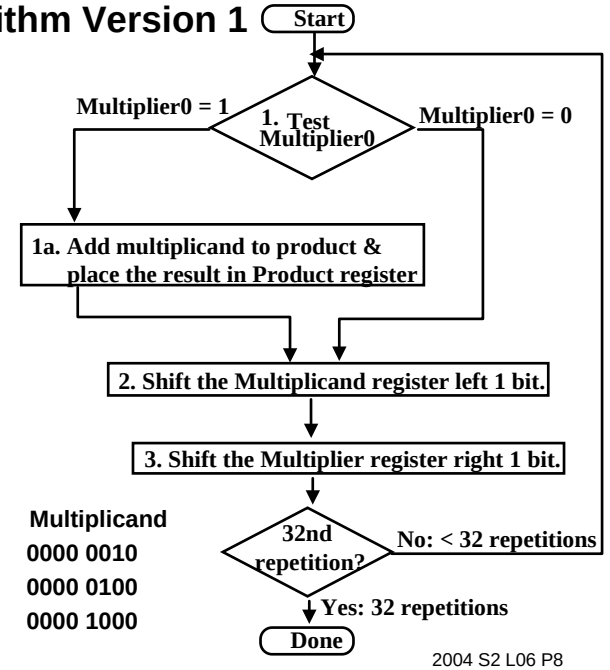
Unisigned shift-add multiplier (version 1)

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



Multiplier = datapath + control

Multiply Algorithm Version 1



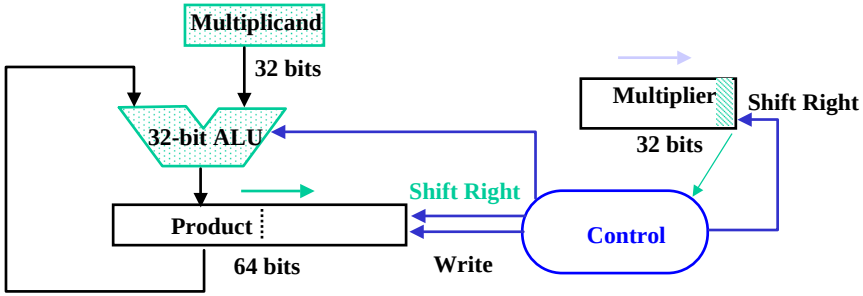
Iter	Product	Multiplier	Multiplicand
0.	0000 0000	0011	0000 0010
1.	0000 0010	0001	0000 0100
2.	0000 0110	0000	0000 1000
3.	0000 0110…		

Observations on Multiply Version 1

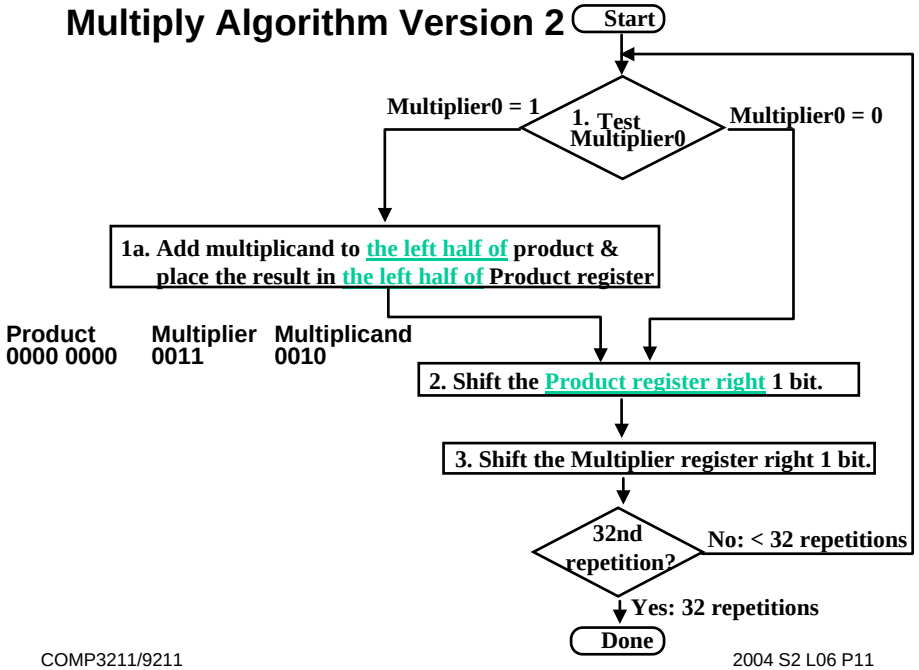
- 1 clock per cycle => 2n clocks for n-bit multiply
  - Add is 5:1 to 100:1 times more popular than mult
  - But slow op will affect performance significantly
- 1/2 bits in multiplicand always 0 => 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

MULTIPLY HARDWARE Version 2

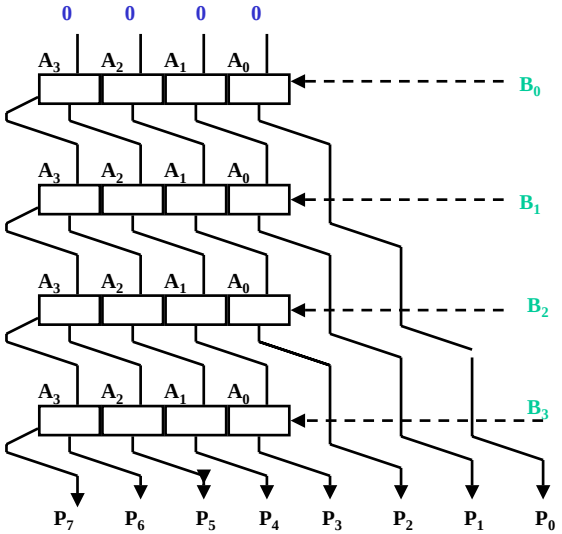
- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg



Multiply Algorithm Version 2



What's going on?

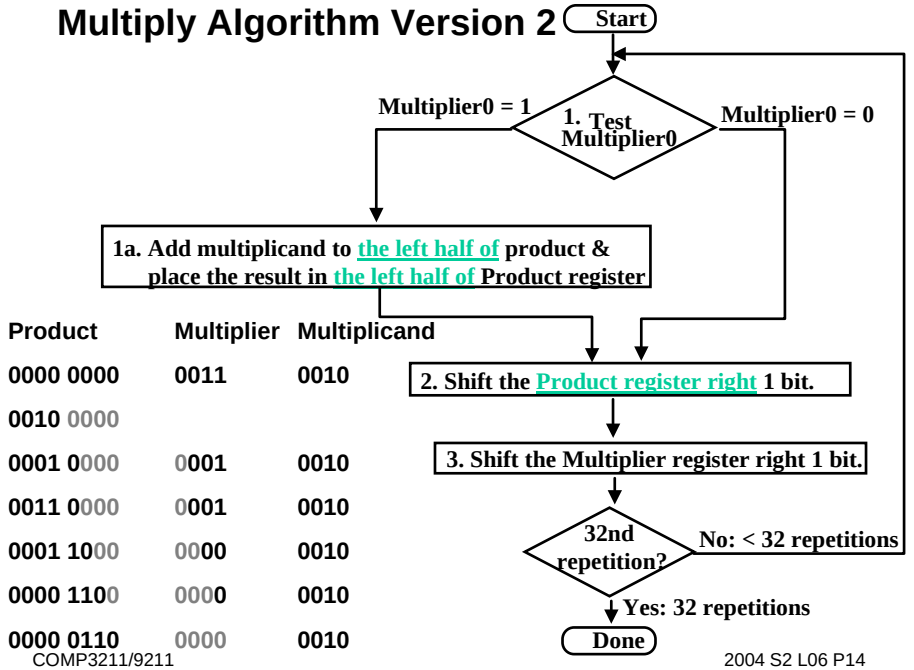


- Multiplicand stay's still and product moves right

Break

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

Multiply Algorithm Version 2

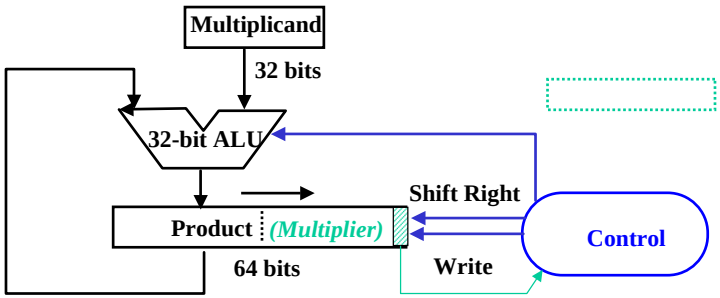


Observations on Multiply Version 2

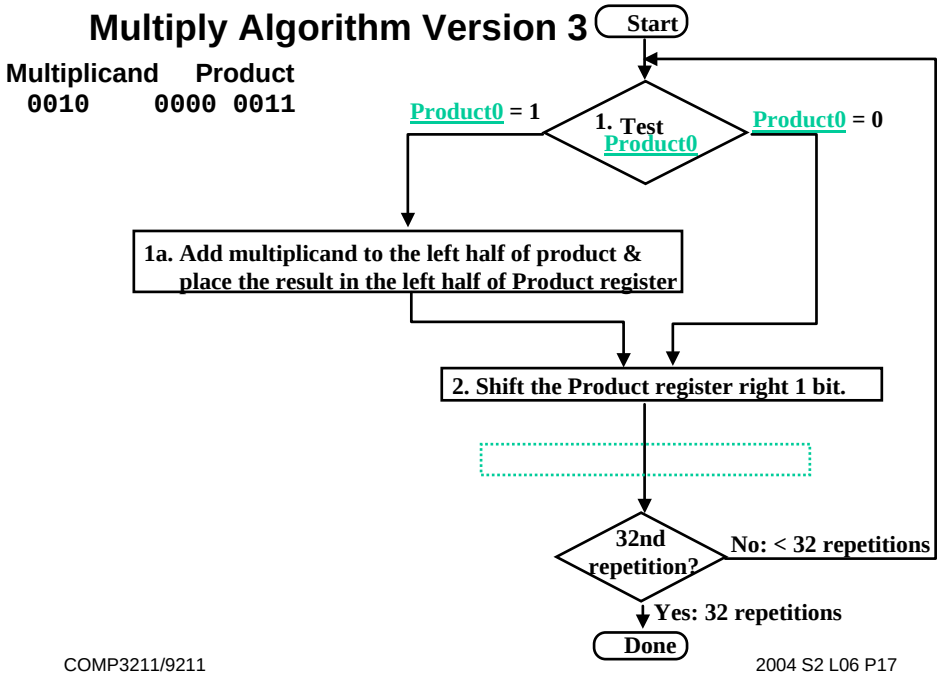
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

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



Multiply Algorithm Version 3



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

Motivation for Booth's Algorithm

Example 2 x 6 = 0010 x 0110:

```
              0010
        x     0110
        +     0000    shift (0 in multiplier)
        +    0010     add (1 in multiplier)
        +   0010      add (1 in multiplier)
        +  0000       shift (0 in multiplier)
           00001100
```

ALU with add or subtract gets same result in more than one way:

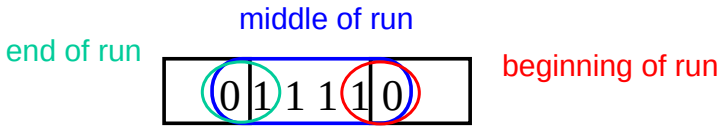
```
6     = – 2 + 8
0110  = – 00010 + 01000 = 11110 + 01000
```

For example

```
              0010
        x     0110
              0000  shift (0 in multiplier)
        –     0010  sub (first 1 in multpl.)
             0000   shift (mid string of 1s)
        +   0010    add (prior step had last 1)
           00001100
```


Booth's Algorithm

Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one



Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	000111**10**00	sub
1	1	Middle of run of 1s	00011**11**000	none
0	1	End of run of 1s	00**01**111000	add
0	0	Middle of run of 0s	0**00**1111000	none

Originally for Speed (when shift was faster than add)

```
   -1
+ 10000
  01111
```

Booths Example (2 x 7)

Current bit, Bit to right

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+ 0010	
		0001 1100 1	shift
4b.	0010	0000 1110 0	done

Booths Example (2 x -3)

Current bit, Bit to right

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+ 1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Some questions

How long does an *n*-bit multiply take with Booth's algorithm?

How long does it take to do *m n*-bit multiplications?

Can we do better?

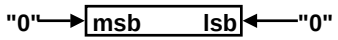
MIPS logical instructions

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

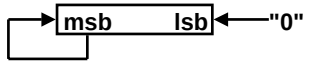
Shifters

Two kinds:

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

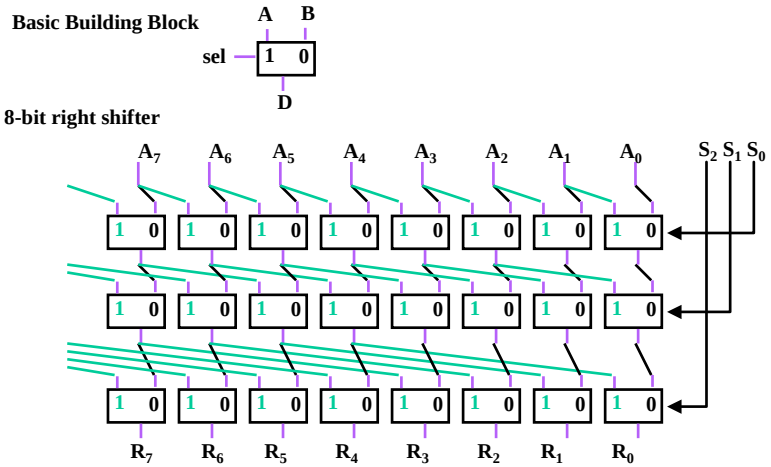


***arithmetic*-- on right shifts, sign extend**



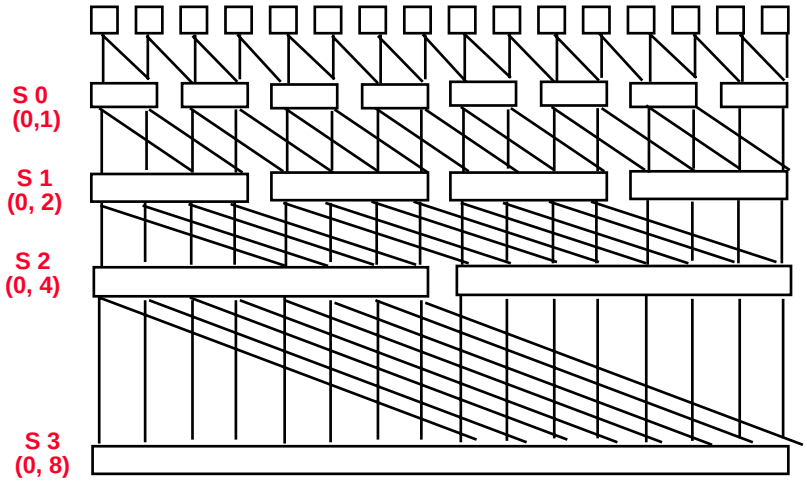
Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Combinational Shifter from MUXes



- **What comes in the MSBs?**
- **How many levels for 32-bit shifter?**
- **What if we use 4-1 Muxes ?**

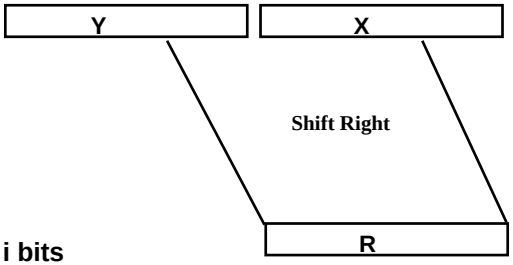
General Shift Right Scheme using 16 bit example



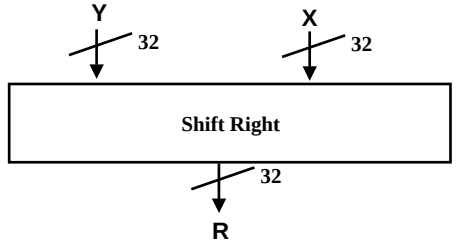
If added Right-to-left connections could support Rotate (not in MIPS but found in ISAs)

Funnel Shifter

Instead Extract 32 bits of 64.

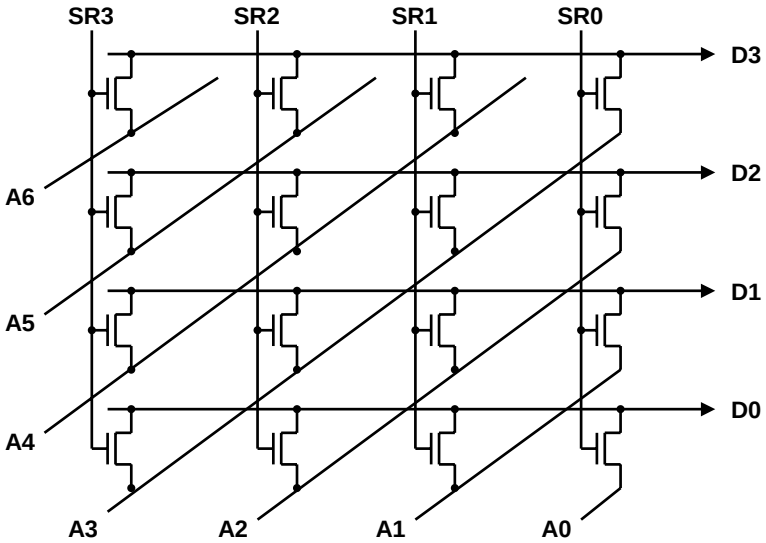


- **Shift A by i bits (sa= shift right amount)**
- **Logical: Y = 0, X=A, sa=i**
- **Arithmetic? Y = _, X=_, sa=_**
- **Rotate? Y = _, X=_, sa=_**
- **Left shifts? Y = _, X=_, sa=_**



Barrel Shifter

Technology-dependent solutions: transistor per switch



Divide: Paper & Pencil

```
                  1001  Quotient
Divisor 1000  | 1001010  Dividend
              -1000
                  10
                 101
                1010
               -1000
                  10  Remainder (or Modulo result)
```

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

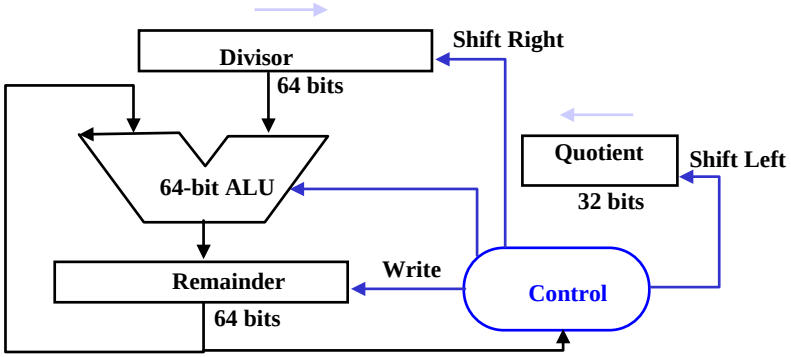
Dividend = Quotient x Divisor + Remainder

=> | Dividend | = | Quotient | + | Divisor |

3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

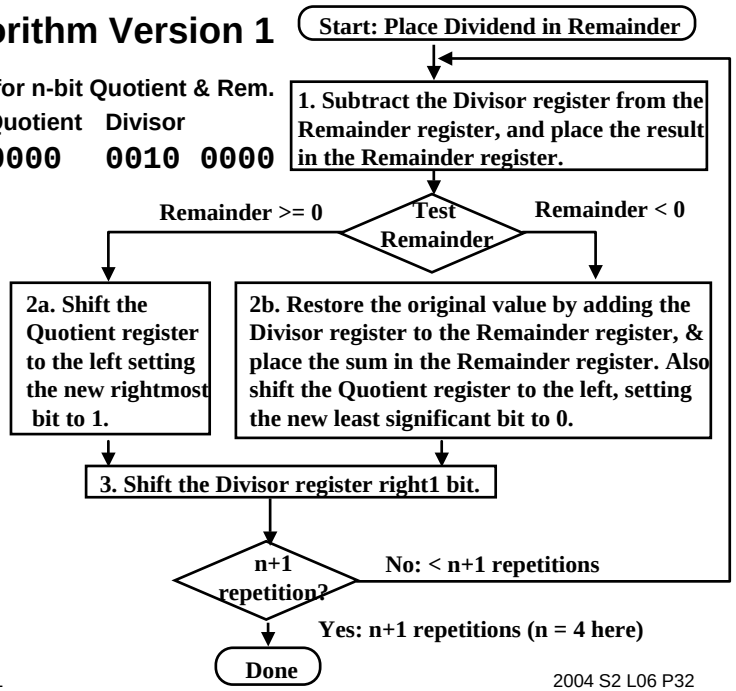
- **64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg**



Divide Algorithm Version 1

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

Remainder	Quotient	Divisor
0000	0111	0000 0010 0000

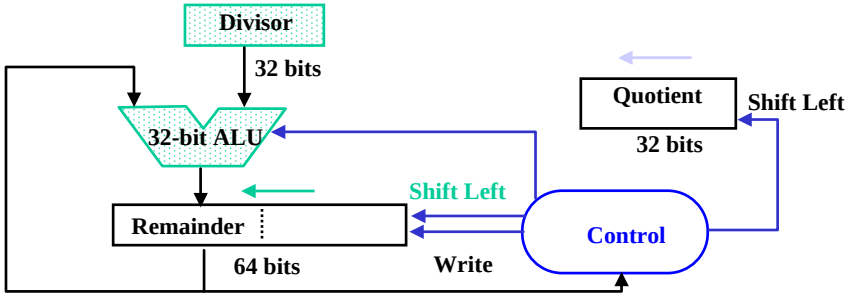


Observations on Divide Version 1

- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise too big)
  => switch order to shift first and then subtract, can save 1 iteration

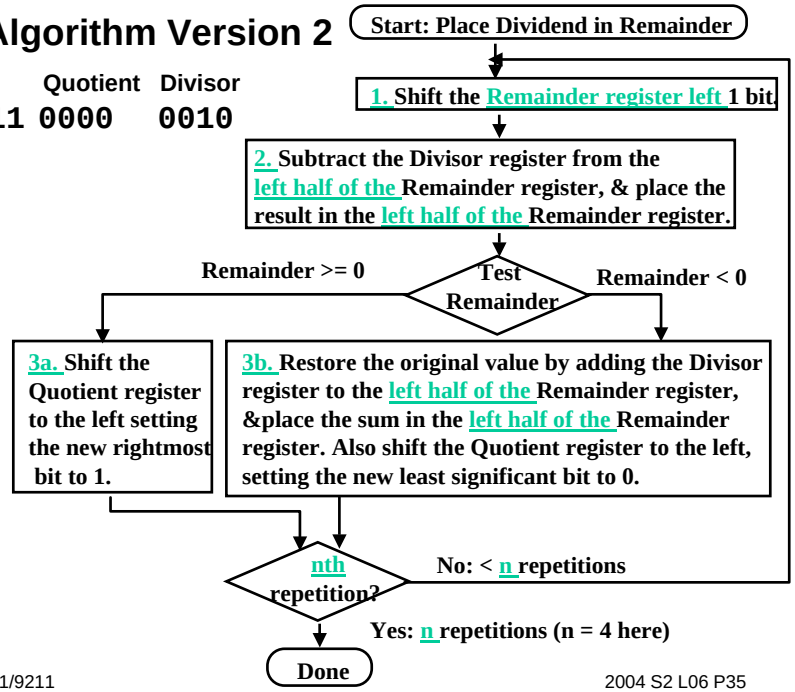
DIVIDE HARDWARE Version 2

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



Divide Algorithm Version 2

Remainder	Quotient	Divisor
0000	0111	0000 0010

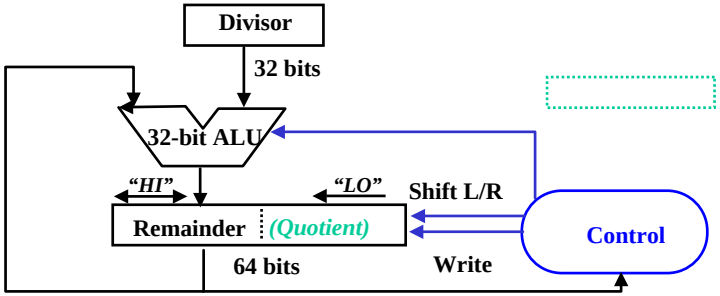


Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will be shifted left one time too many.
  - Thus a final correction step must shift back only the remainder in the left half of the register

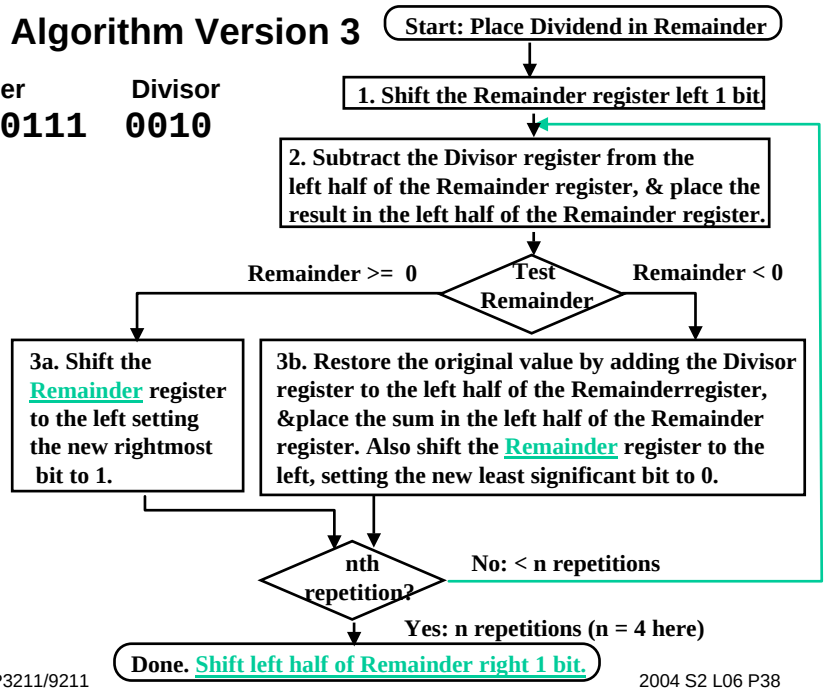
DIVIDE HARDWARE Version 3

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



Divide Algorithm Version 3

Remainder Divisor
0000 0111 0010



Observations on Divide Version 3

- 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 e.g., –7 ÷ 2 = –3, remainder = –1
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits ("called saturation")

Summary

- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth's algorithm to handle signed multiplies
  - There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39 in COD)
- Shifter: success refinement 1/bit at a time shift register to barrel shifter
- Divide can use same hardware as multiply: Hi & Lo registers in MIPS
- Floating point basically follows paper and pencil method of scientific notation using integer algorithms for multiply and divide of significands