EEL-4713 Computer Architecture
Multipliers and shifters

Outline

- Multiplication and shift registers
  - Chapter 3, section 3.4
- Next lecture
  - Division, floating-point
  - 3.5 – 3.6

Deriving requirements of ALU

- Start with instruction set architecture: must be able to do all operations in ISA
- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget
- MIPS ISA

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,	Lo = quotient, Hi = remainder
			Hi = $2 mod $3
divide unsigned	divu $2,$3	Lo = $2 ÷ $3,	Unsigned quotient &
			remainder
			Hi = $2 mod $3
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

MIPS ALU requirements

- Add, AddU, Sub, SubU, AddI, AddIU
  => 2’s complement adder with overflow detection & inverter
- SLTI, SLTIU (set less than)
  => 2’s complement adder with inverter, check sign bit of result
- BEQ, BNE (branch on equal or not equal)
  => 2’s complement adder with inverter, check if result = 0
- And, Or, AndI, OrI
  => Logical AND, logical OR
- ALU from last lecture supports these ops

Additional MIPS ALU requirements

- Xor, Nor, XorI
  => Logical XOR, logical NOR
- Sll, Srl, Sra
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits
- Mult, MultU, Div, DivU
  => Need 32-bit multiply and divide, signed and unsigned

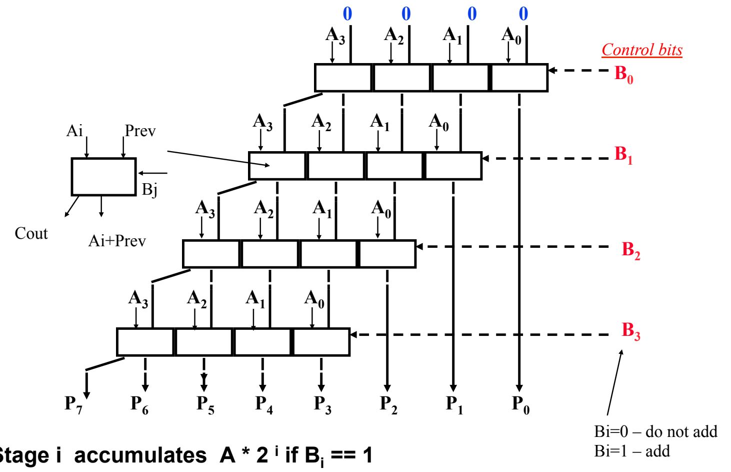
MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

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

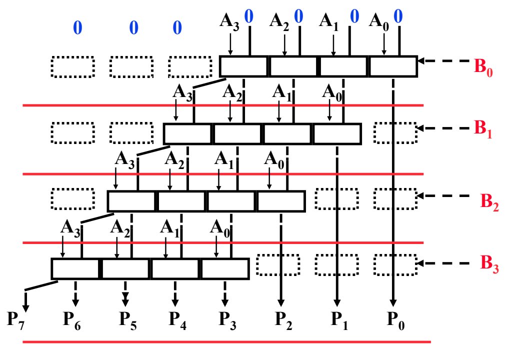
- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
- 4 versions of multiply hardware & algorithm:
  - successive refinement

Unsigned Combinational Multiplier: A * B



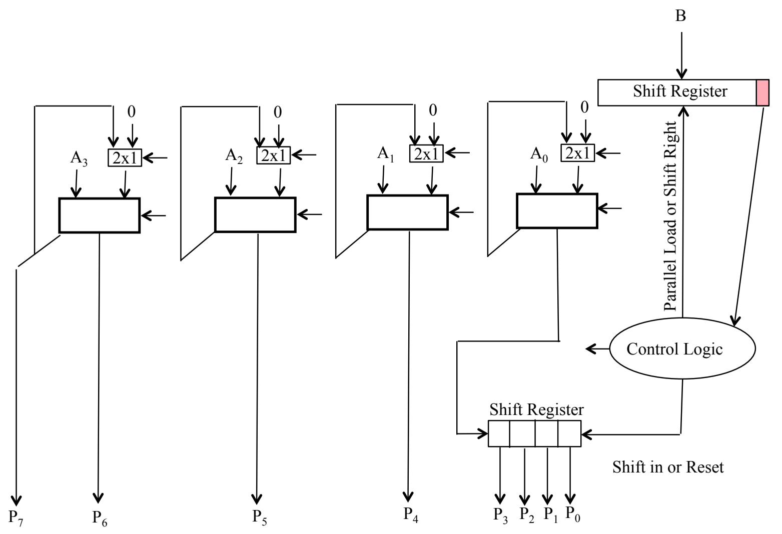
- Stage i accumulates A * 2^i if B$_i$ == 1
- Q: How much hardware for 32 bit multiplier? Critical path?

How does it work?



- at each stage, shift A left once more (effectively A*2)
- use next bit of B to determine whether to add in shifted multiplicand
- accumulate 2 n-bit partial products at each stage

Same principle, less hardware, multi-cycled



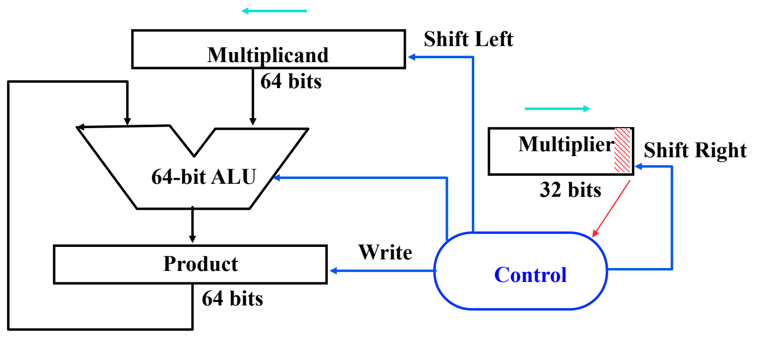
B
Shift Register
Parallel Load or Shift Right
0 0 0 0
A_3 2x1 A_2 2x1 A_1 2x1 A_0 2x1
Control Logic
Shift Register
Shift in or Reset
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0

Improvements

- Same operations in each step
  - Shift, add
  - Shift – hardwired; add: separate adders in each stage = LOTS OF HARDWARE!
- Can reduce size by performing one step per cycle, reusing the adder
  - Using a shift register rather than hard-wiring
  - Multi-cycle instead of one long cycle
- Advantage: uses less hardware, may be pipelined
- Disadvantage: it may be slower due to pipelining overhead and unbalanced pipeline stages

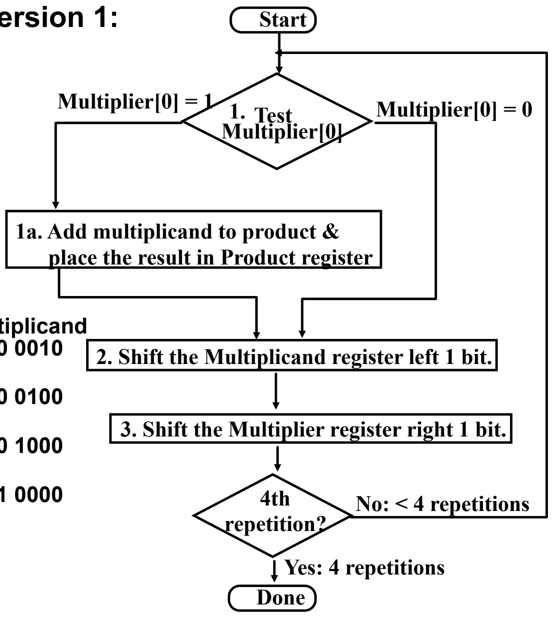
Unsigned shift-add multiplier (version 1)

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



Multiplier = datapath + control

Multiply Algorithm Version 1: 0011 * 0010


Start
1. Test Multiplier[0]
Multiplier[0] = 1
Multiplier[0] = 0
1a. Add multiplicand to product & place the result in Product register
2. Shift the Multiplicand register left 1 bit.
3. Shift the Multiplier register right 1 bit.
4th repetition?
No: < 4 repetitions
Yes: 4 repetitions
Done

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

Potential Optimizations

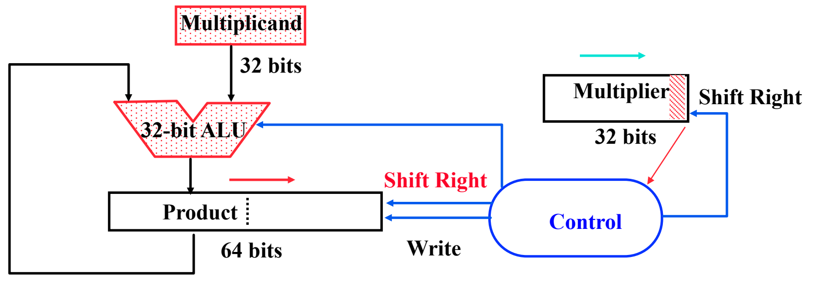
- In state 1:
  - Test multiplier for 0, exit
- Observation:
  - If multiplier[0] = 1, 2 cycles needed (add, then shifts)
  - If multiplier[0] = 0, 1 cycle needed (shifts only)
  - Before start, set multiplier and multiplicand intelligently
    - Multiplier as smallest value?
      - Find left-most 1 bit for each input, the right-most of those two bits is the smallest value, therefore exit sooner out of state 1
    - Multiplier as least number of ones?
      - X = 1000000 and Y = 0111111
      - If multiplier = X, 9 cycles
      - If multiplier = Y, 13 cycles

Observations on Multiply Version 1

- 1 clock per add and shifts (can shift in parallel) => 32*2 clocks per multiply
- Half of 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?

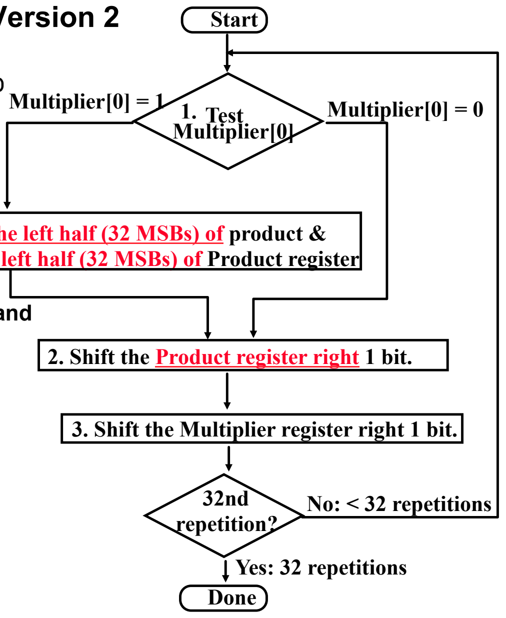
MULTIPLY HARDWARE Version 2

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



*Multiply Algorithm Version 2

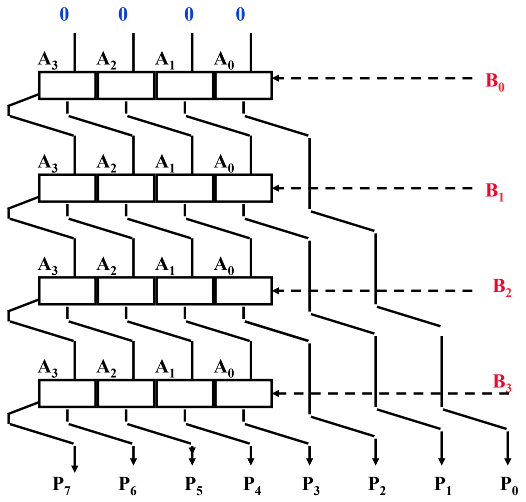
Multiplier	Multiplicand	Product
0011	0010	0000 0000



Start
Multiplier[0] = 1
1. Test Multiplier[0]
Multiplier[0] = 0
1a. Add multiplicand to the left half (32 MSBs) of product & place the result in the left half (32 MSBs) of Product register
2. Shift the Product register right 1 bit.
3. Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

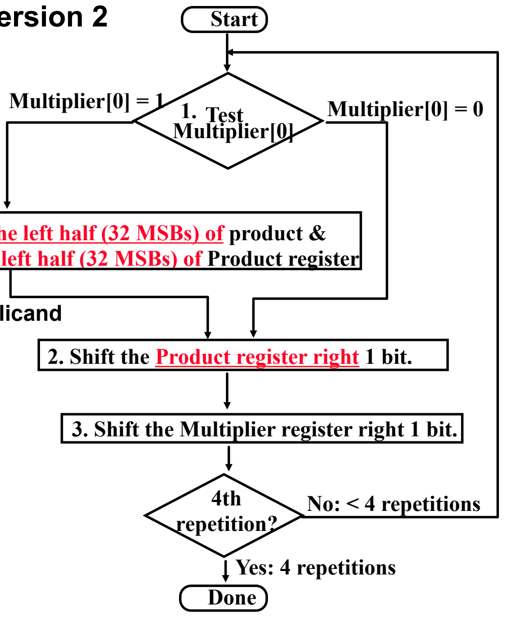
Product	Multiplier	Multiplicand
0000 0000	0011	0010

What's going on?



° Multiplicand stays still and product moves right

Multiply Algorithm Version 2

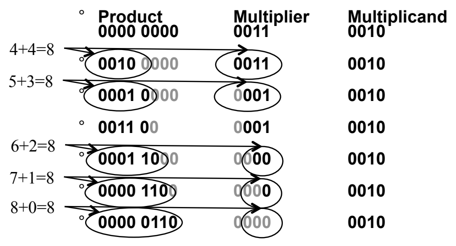


Start
Multiplier[0] = 1
1. Test Multiplier[0]
Multiplier[0] = 0
1a. Add multiplicand to the left half (32 MSBs) of product & place the result in the left half (32 MSBs) of Product register
2. Shift the Product register right 1 bit.
3. Shift the Multiplier register right 1 bit.
4th repetition?
No: < 4 repetitions
Yes: 4 repetitions
Done

Product	Multiplier	Multiplicand
0000 0000	0011	0010
0010 0000	0011	0010
0001 0000	0001	0010
0011 00	0001	0010
0001 1000	0000	0010
0000 1100	0000	0010
0000 0110	0000	0010

Observations on Multiply Version 2

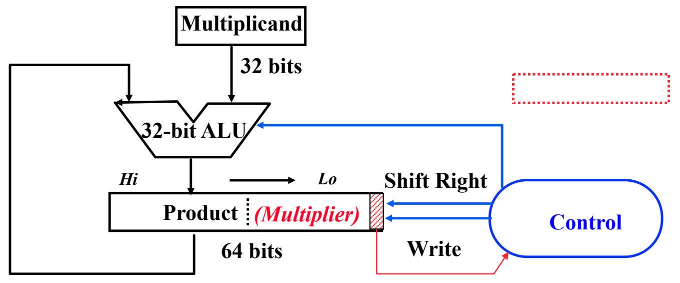
° Wasted space in the product register exactly matches size of multiplier
=> combine Multiplier register and Product register



	Product	Multiplier	Multiplicand
	0000 0000	0011	0010
4+4=8	0010 0000	0011	0010
5+3=8	0001 0000	0001	0010
	0011 00	0001	0010
6+2=8	0001 1000	0000	0010
7+1=8	0000 1100	0000	0010
8+0=8	0000 0110	0000	0010

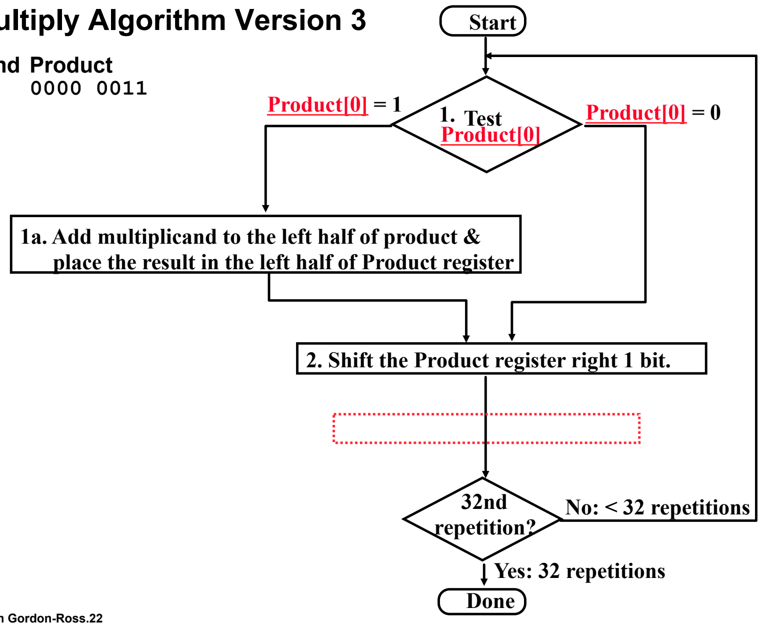
MULTIPLY HARDWARE Version 3

° 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg, Multiplier stored in LSBs of Product)



Multiply Algorithm Version 3

Multiplicand	Product
0010	0000 0011



Start
1. Test Product[0]
Product[0] = 1
Product[0] = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
2. Shift the Product register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Observations on Multiply Version 3

- 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

Representing sequences of 1s as a difference

```
0   0   0   0   1   1   1   1
                2³  2²  2¹  2⁰
                8+  4+  2+  1  =  15  =  2⁴-2⁰  =  16-0
```

$$0\ 0\ 0\ 0 \ldots 0\ \underbrace{\overset{n}{1}\ \overset{n-1}{1}\ \overset{n-2}{1}\ 1\ \overset{\ldots}{1}\ 1\ 1\ \overset{n-8}{1}}_{2^n+2^{n-1}+\ldots+2^{n-8}}\ 0\ 0\ 0 \qquad = 2^{n+1}-2^{n-8}$$

Representing sequences of 1s as a difference

- Why?
  - x = 2^n + 2^(n-1) + … + 2^k
  - 2x = 2^(n+1) + 2^n + … + 2^(k+1) (essentially shift left 1 bit)
  - 2x – x = [2^(n+1) + 2^n + … + 2^(k+1)] – [2^n + 2^(n-1) + … + 2^k]
  - 2^n through 2^(k+1) cancel out
  - x = 2^(n+1) – 2^k

- How is this useful?
  - A sequence of (n-k) 1s in a multiplier
    - (n-k) adds, or
    - 1 add and 1 sub
  - Look at each sequence of 1s and add together

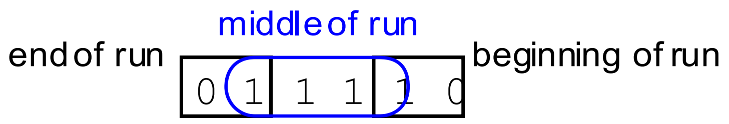
Motivation for Booth's Algorithm

- 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
```

- 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.

Booth's Algorithm Insight



Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

Speed motivation if shift faster than add

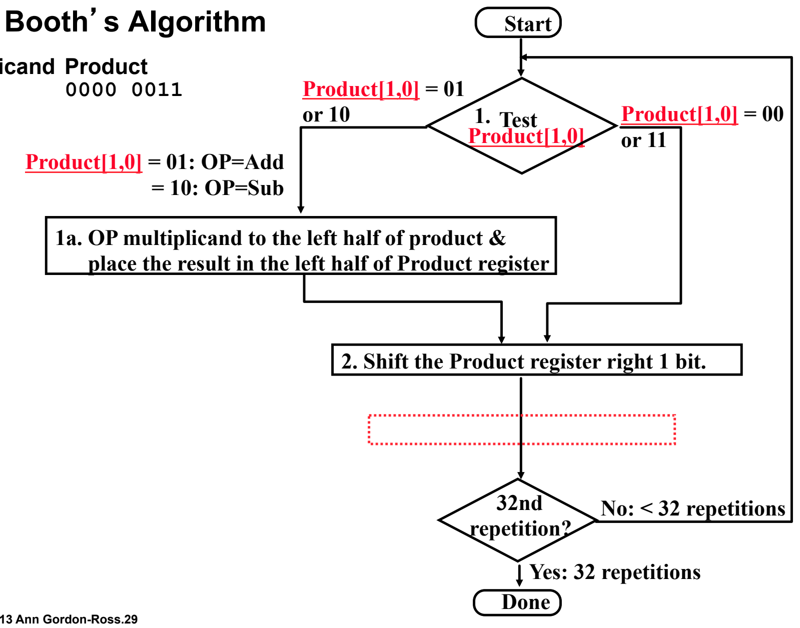
Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:
   - 00: a. Middle of a string of 0s, so no arithmetic operations.
   - 01: b. End of a string of 1s, so add the multiplicand to the left half of the product.
   - 10: c. Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
   - 11: d. Middle of a string of 1s, so no arithmetic operation.
2. As in the previous algorithm, shift the Product register right (arith) 1 bit.

Booth's Algorithm

Multiplicand Product
0010 0000 0011



Start
Product[1,0] = 01
or 10
1. Test
Product[1,0]
Product[1,0] = 00
or 11
Product[1,0] = 01: OP=Add
= 10: OP=Sub
1a. OP multiplicand to the left half of product &
place the result in the left half of Product register
2. Shift the Product register right 1 bit.
32nd
repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Booths Example: 2 (0010) * 7 (0111) = 14 (0000 1110)

1 extra bit for comparison

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	→ 10 -> sub
1a. P = P - m	1110 *add (-m)+1*	+ 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
		14	

Booths Example: 2 (0010) * -3 (1101) = -6 (1111 1010)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110 *add (-m)+1*	+ 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
		Neg. 6	

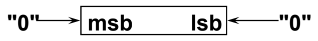
MIPS logical instructions

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 (sign extend)

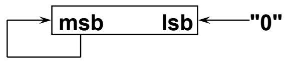
Shifters

Three different kinds:

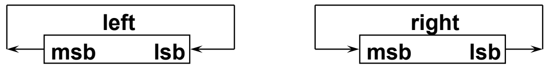
logical-- value shifted in is always "0"



arithmetic-- on right shifts, sign extend

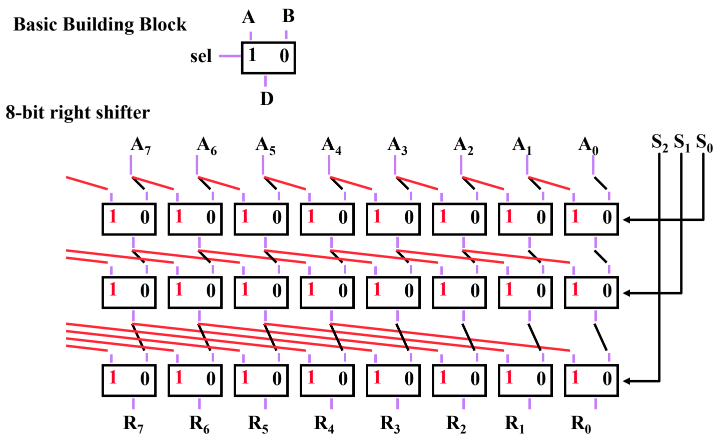


rotating-- shifted out bits are wrapped around (not in MIPS)



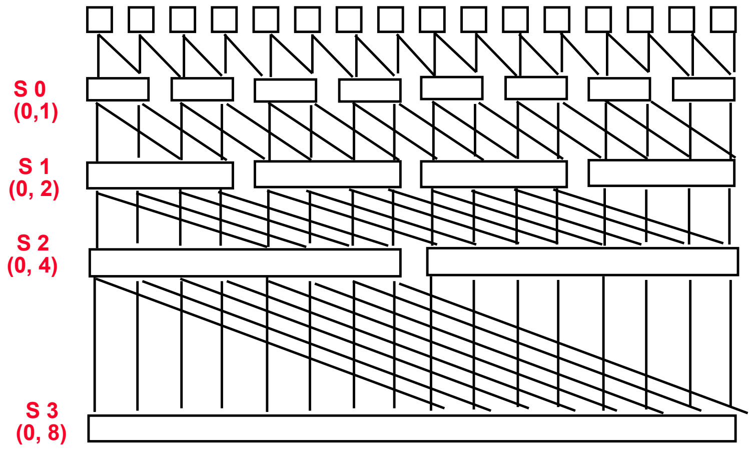
Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Combinational Shifter from MUXes



° How many levels for 32-bit shifter?

General Shift Right Scheme using 16 bit example



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

Summary

° Instruction Set drives the ALU design

° 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

° What's Missing from MIPS is Divide & Floating Point Arithmetic