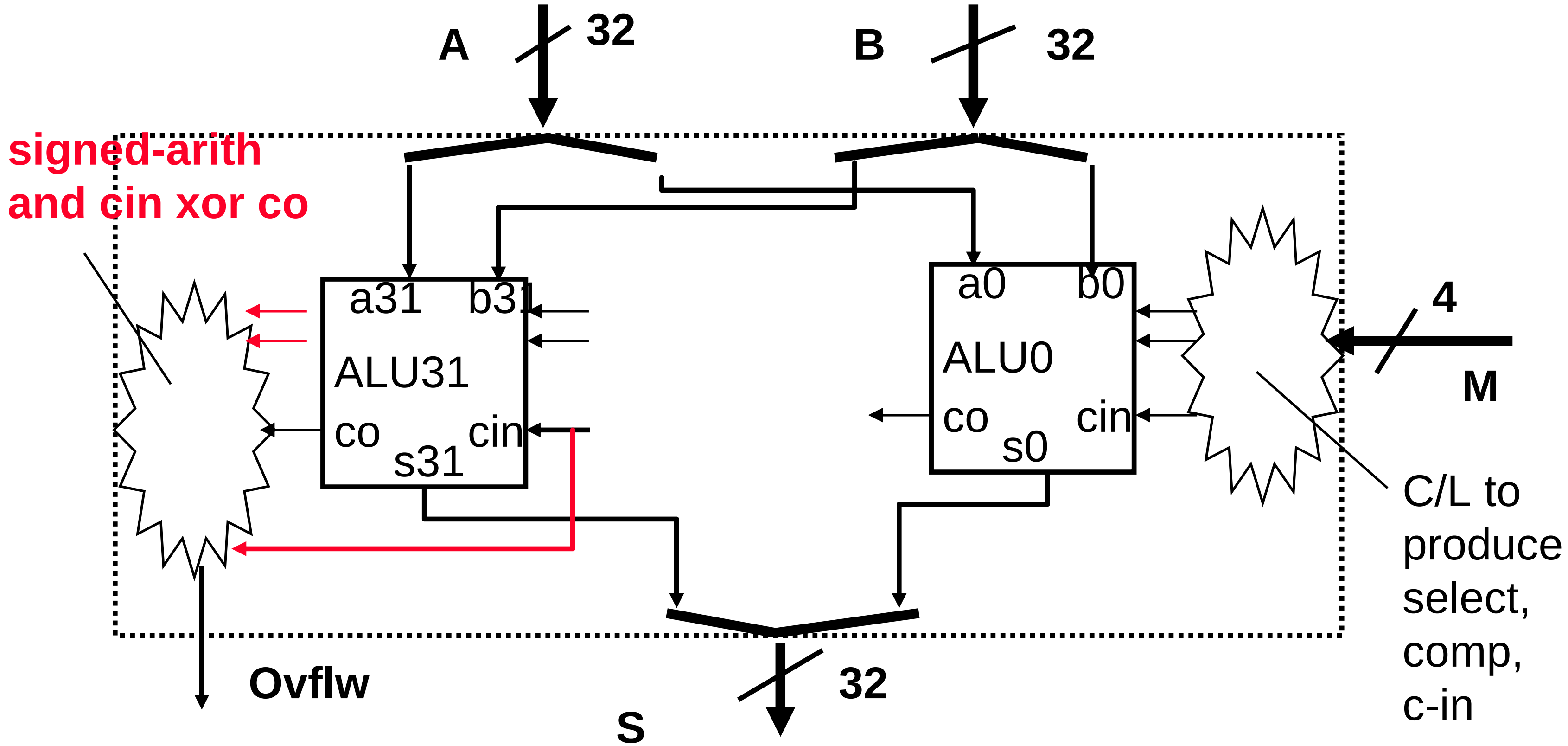
Computer Architecture
EECS 361
Lecture 6: ALU Design

Review: ALU Design

- Bit-slice plus extra on the two ends
- Overflow means number too large for the representation
- Carry-look ahead and other adder tricks



Review: Elements of the Design Process

- Divide and Conquer (e.g., ALU)
  - Formulate a solution in terms of simpler components.
  - Design each of the components (subproblems)
- Generate and Test (e.g., ALU)
  - Given a collection of building blocks, look for ways of putting them together that meets requirement
- Successive Refinement (e.g., multiplier, divider)
  - Solve "most" of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.
- Formulate High-Level Alternatives (e.g., shifter)
  - Articulate many strategies to "keep in mind" while pursuing any one approach.
- Work on the Things you Know How to Do
  - The unknown will become “obvious” as you make progress.

Outline of Today's Lecture

- Deriving the ALU from the Instruction Set
- Multiply

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 logical instructions

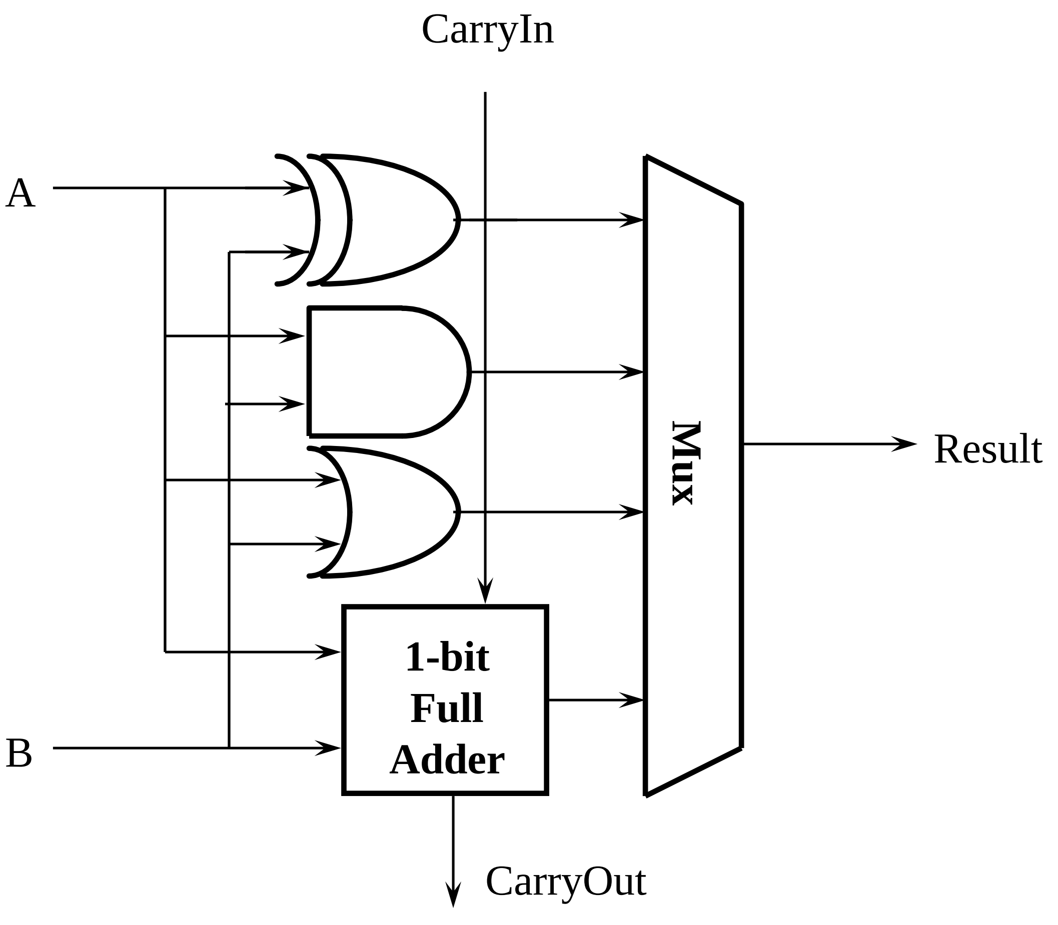
Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$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

Additional MIPS ALU requirements

- Xor, Nor, XorI
  => Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111....1111
- 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

Add XOR to ALU

- Expand Multiplexor

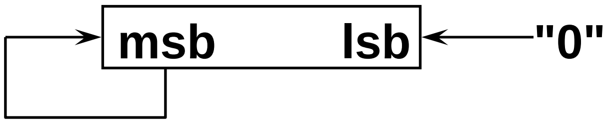


Shifters

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!

Administrative Matters

MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

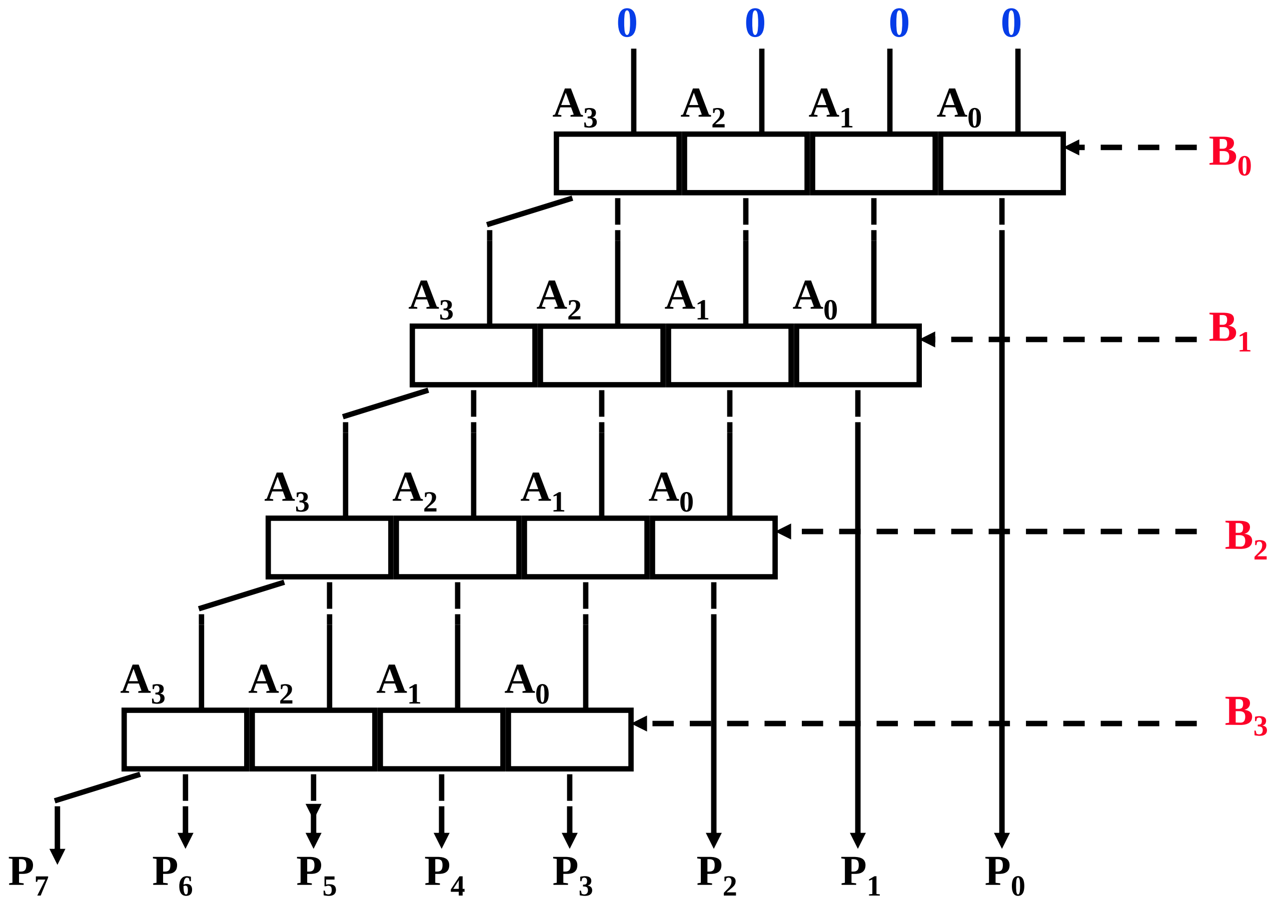
```
Multiplicand      1000
  Multiplier      1001
                         1000
                        0000
                       0000
                     1000
Product             01001000
```

- 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)

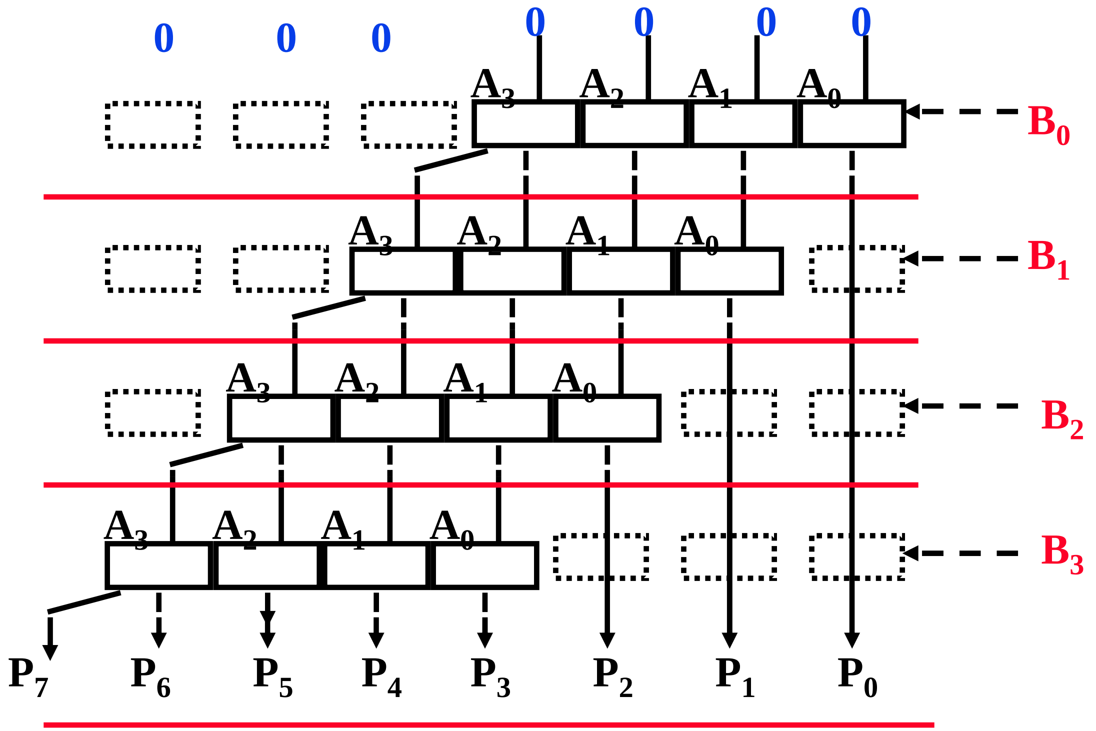
- 4 versions of multiply hardware & algorithm:
  - •successive refinement

Unsigned Combinational Multiplier



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

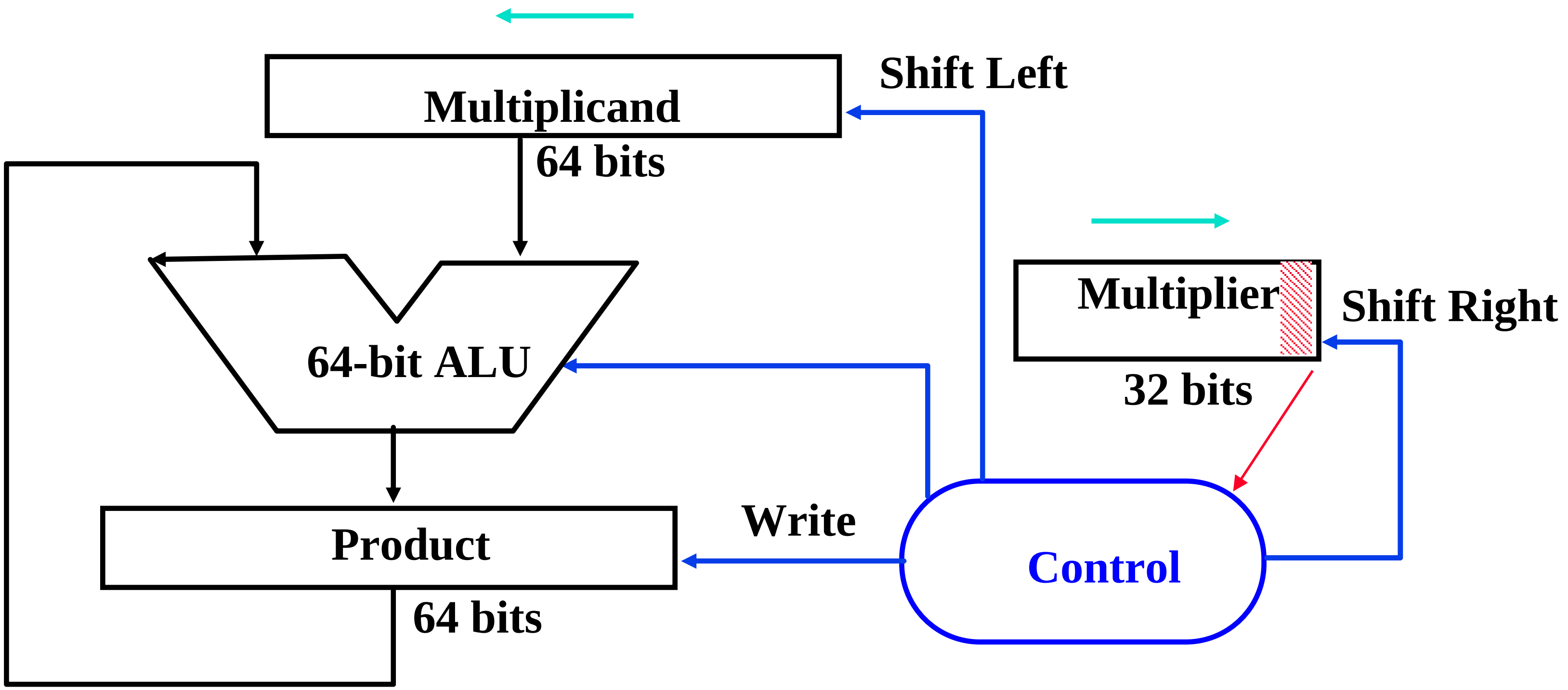
How does it work?



- 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

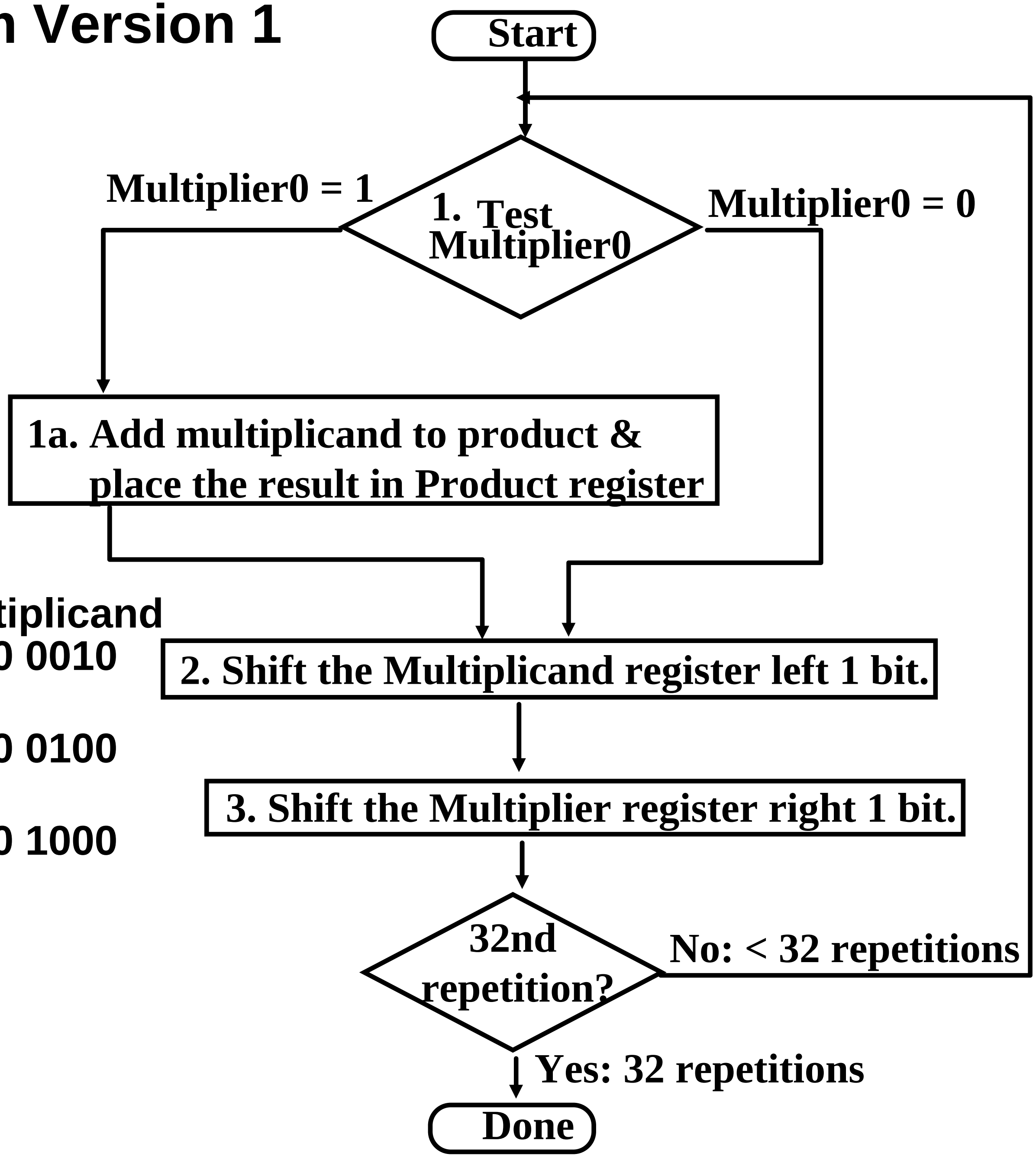
Unisigned shift-add multiplier (version 1)

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



Multiplier = datapath + control

Multiply Algorithm Version 1



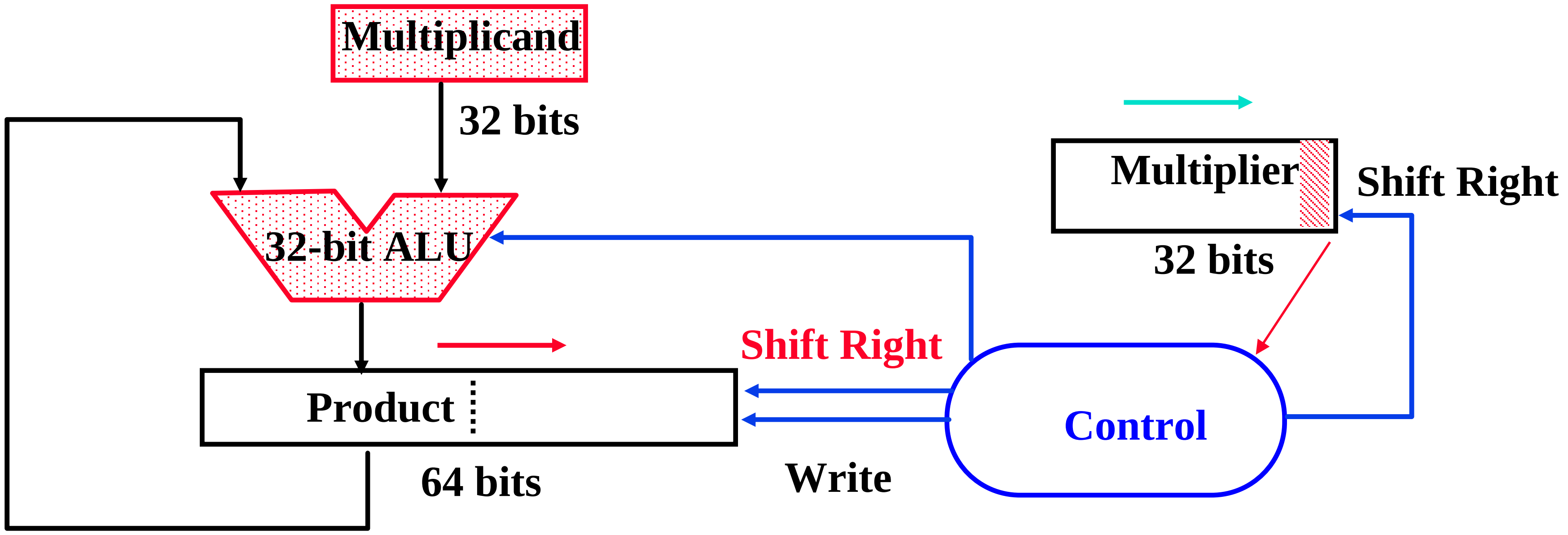
Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

Observations on Multiply Version 1

- 1 clock per cycle => - 100 clocks per multiply
  - Ratio of multiply to add 5:1 to 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?

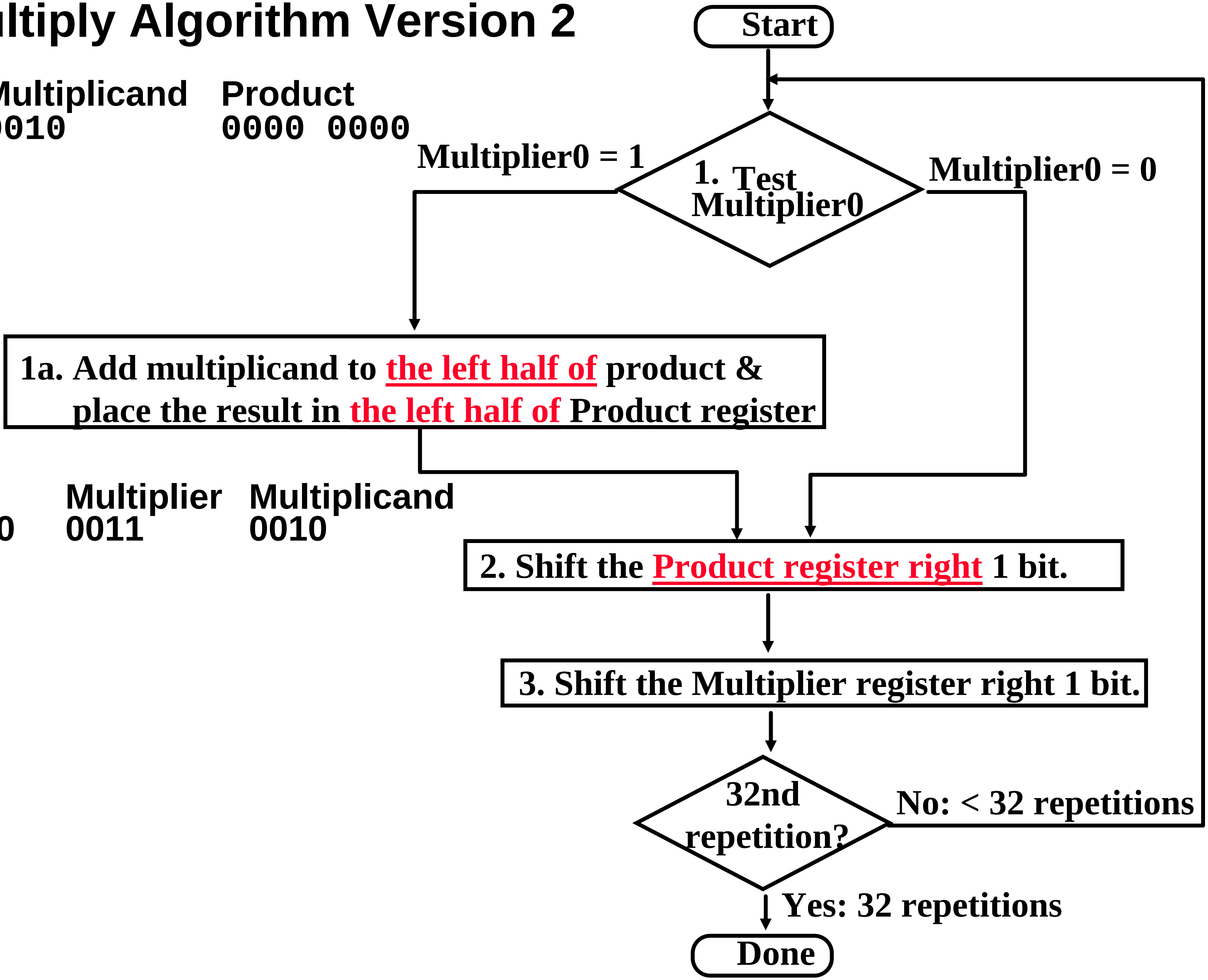
MULTIPLY HARDWARE Version 2

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



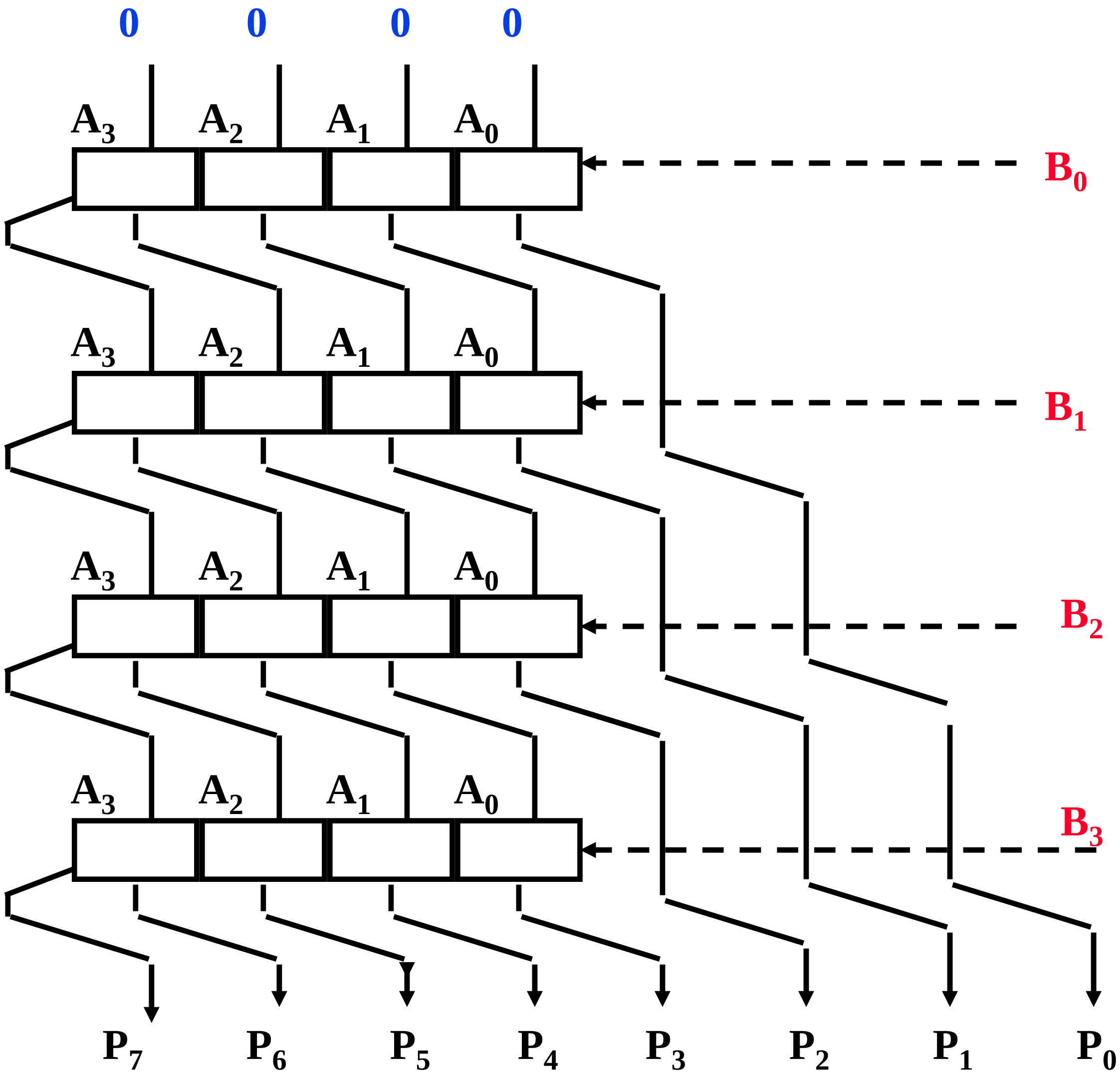
Multiply Algorithm Version 2

Multiplier	Multiplicand	Product
0011	0010	0000 0000



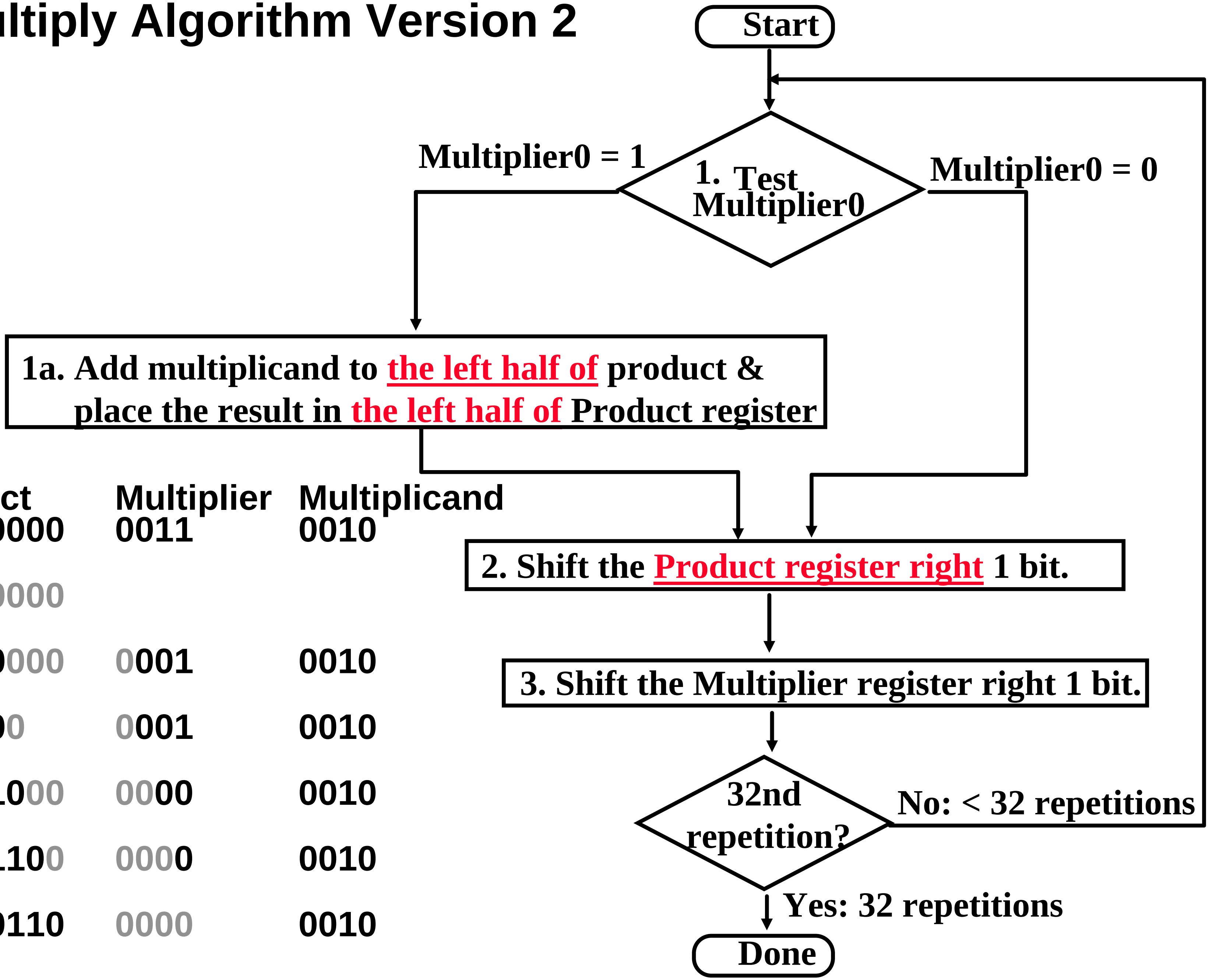
Product	Multiplier	Multiplicand
0000 0000	0011	0010

What's going on?



- **Multiplicand stay's still and product moves right**

Multiply Algorithm Version 2



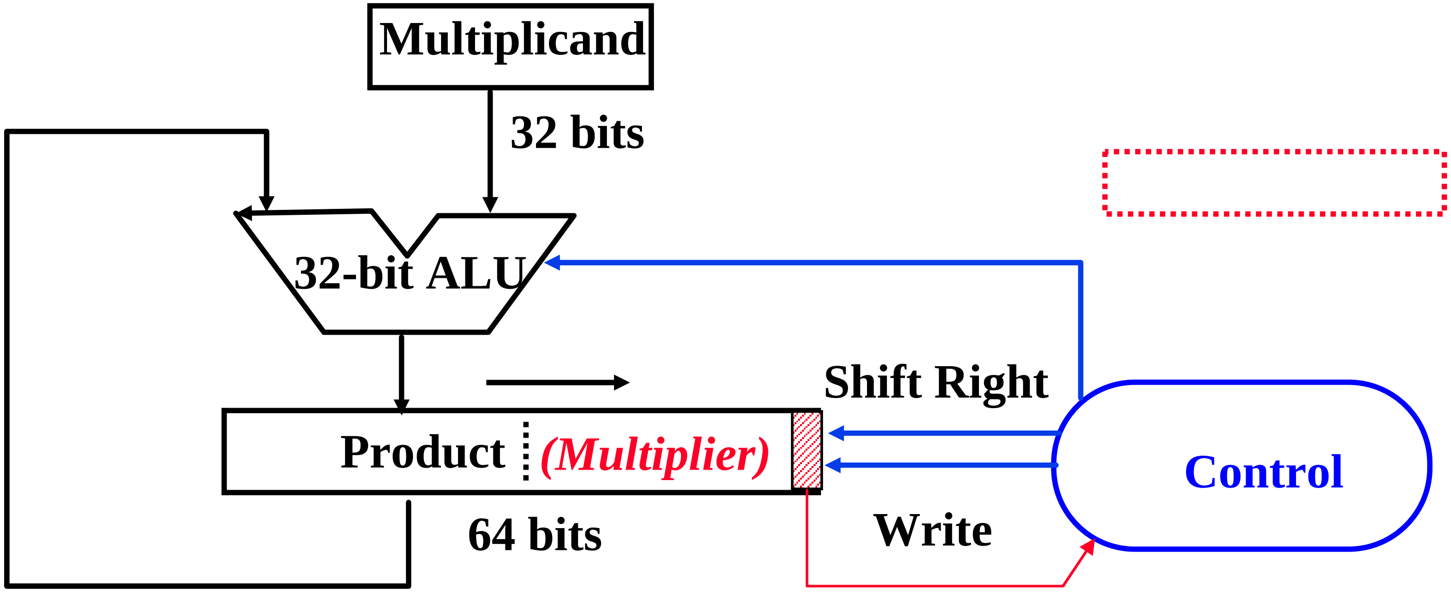
Product	Multiplier	Multiplicand
0000 0000	0011	0010
0010 0000		
0001 0000	0001	0010
0011 00	0001	0010
0001 1000	0000	0010
0000 1100	0000	0010
0000 0110	0000	0010

Observations on Multiply Version 2

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

MULTIPLY HARDWARE Version 3

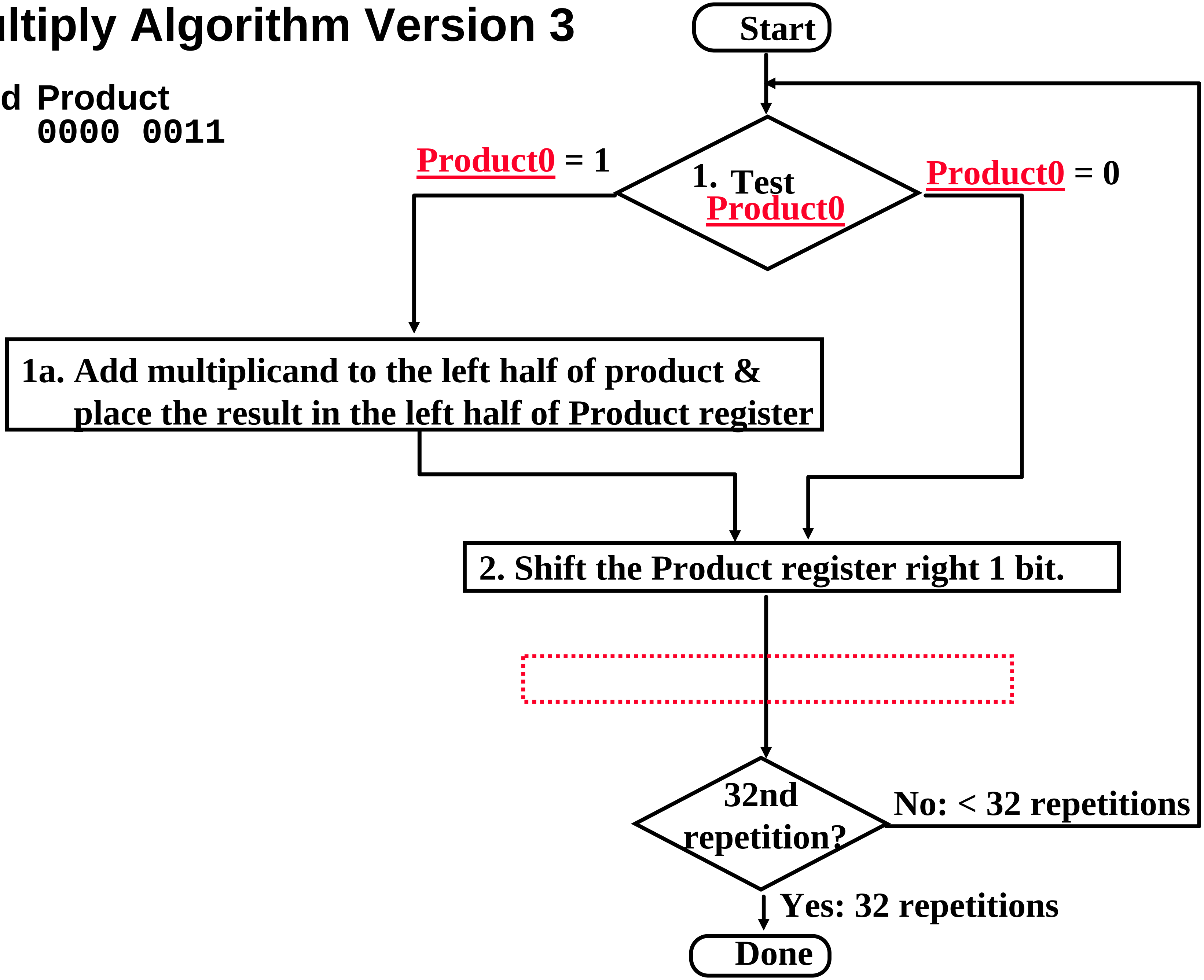
- **32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)**



Multiply Algorithm Version 3

Multiplicand **Product**

`0010` `0000 0011`



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)
+             0100        add (1 in multiplier)
+            0000         shift (0 in multiplier)
--------------------
           00001100
```

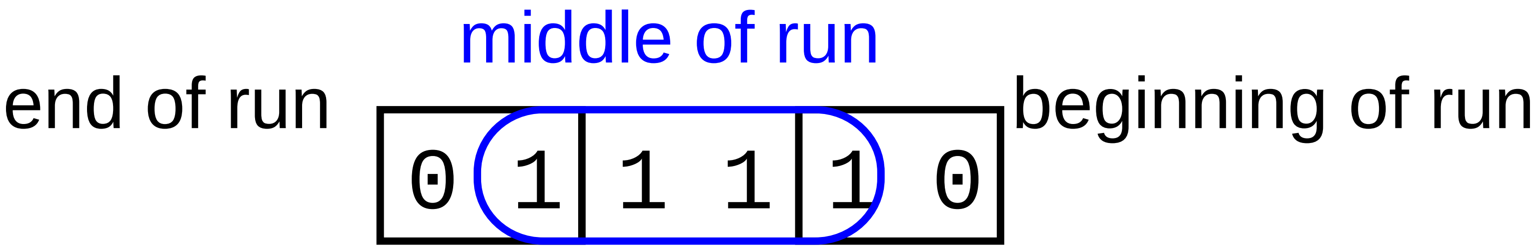
- **ALU with add or subtract gets same result in more than one way:**

```
6            = – 2 + 8          , or
0110         = – 0010+ 1000
```

- **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. For example**

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

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

Originally for Speed since shift faster than add for his machine

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

Booths Example (2 x 7)

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)

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

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.

MIPS logical instructions

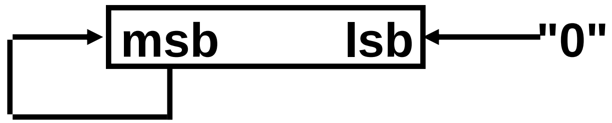
Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$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"

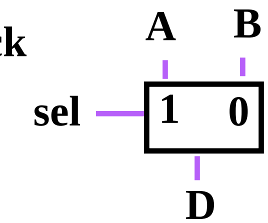
arithmetic-- on right shifts, sign extend



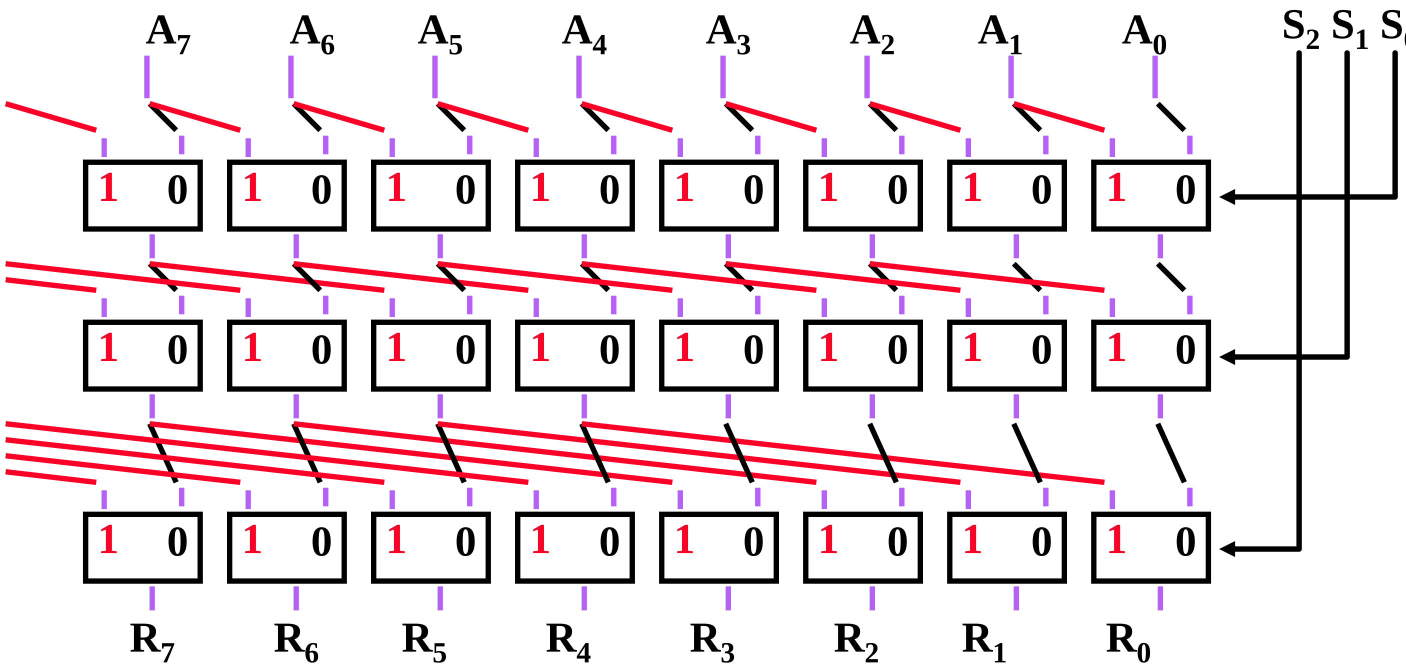
<u>Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!</u>

Combinational Shifter from MUXes

Basic Building Block

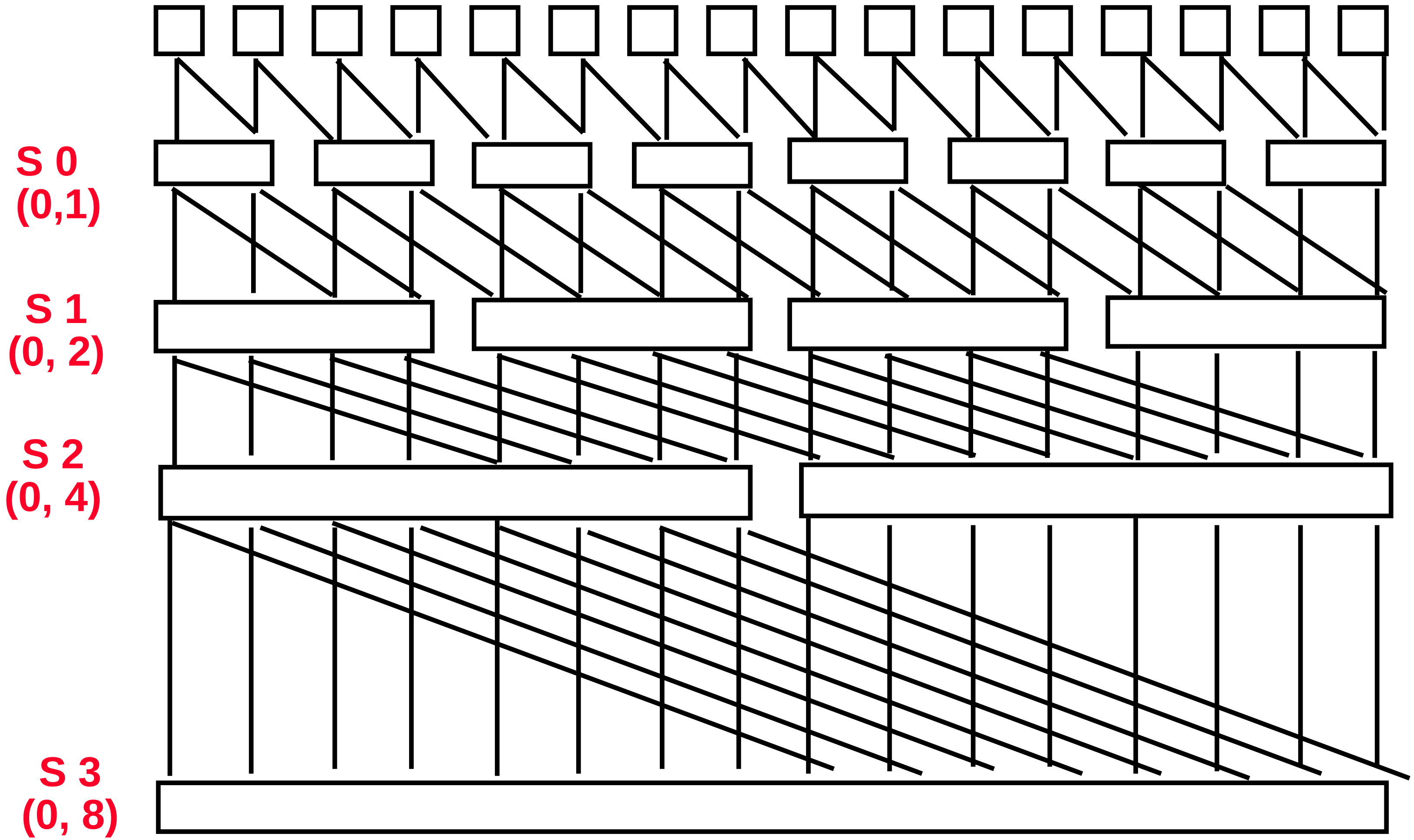


8-bit right shifter



- **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)

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
- There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39)
- What's Missing from MIPS is Divide & Floating Point Arithmetic