CS152 – Computer Architecture and Engineering

Lecture 2 – Verilog & Multiply Review

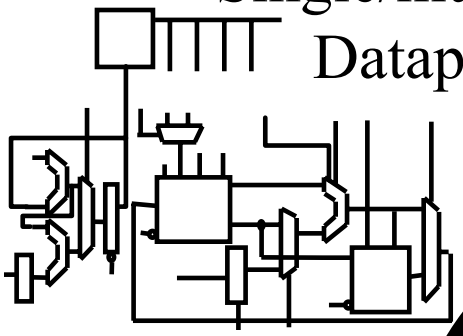
2003-08-28

Dave Patterson
(www.cs.berkeley.edu/~patterson)

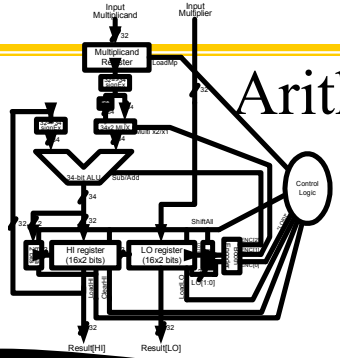
www-inst.eecs.berkeley.edu/~cs152/

Review: CS 152 roadmap

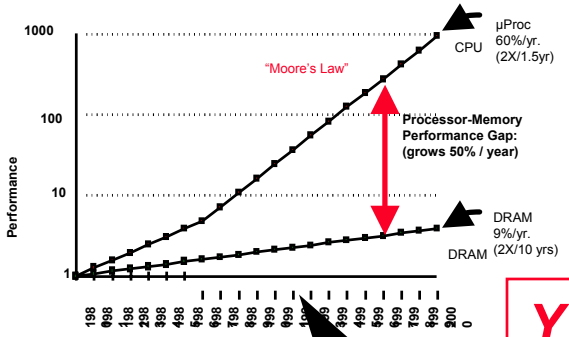
Single/multicycle Datapaths

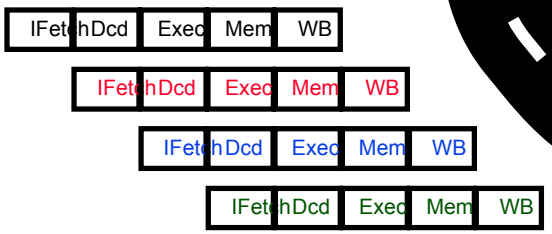


Arithmetic

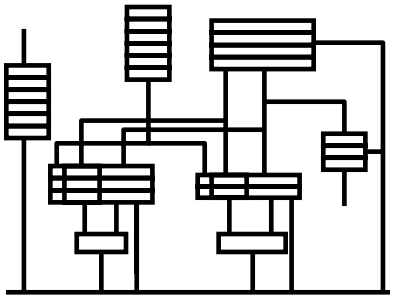


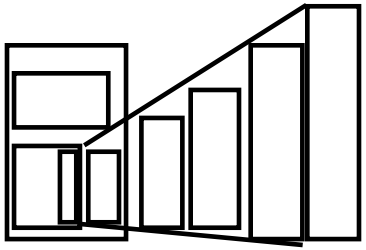
CS152 Fall '03



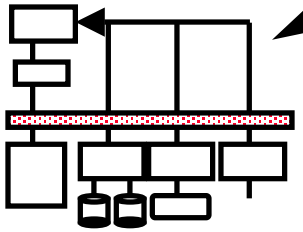


Pipelining





Memory Systems



I/O

YOUR CPU

Review

- Continued rapid improvement in Computing
  - 2X every 1.5 years in processor speed; every 2.0 years in memory size; every 1.0 year in disk capacity; Moore's Law enables processor, memory (2X transistors/chip/ ~1.5 yrs)
- 5 classic components of all computers

Control Datapath Memory Input Output

Processor

Verilog

- 1 of 2 popular Hardware Description Languages (other is VHDL)
- A language for describing and testing logic circuits.
  - text based way to talk about designs
  - easier to simulate before silicon
  - translate into silicon directly
- No sequential execution, normally hardware just “runs” continuously.
- Verilog: A strange version of C, with some changes to account for time

Overview

- Behavioral vs. Structural Verilog
- Time in Verilog and wave forms
- Testing and Test Benches in Verilog
- Nice online tutorial:

www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html

Verilog

- Verilog description composed of modules:

  `module` *Name* (*port list*) ;

  Declarations and Statements;

  `endmodule;`

- Modules can have instantiations of other modules, or use primitives supplied by language

- Note that Verilog varies from C syntax, borrowing from Ada programming language at times (`endmodule`)

Verilog

- Verilog has 2 basic modes

1. Structural composition: describes that structure of the hardware components, including how ports of modules are connected together
   - module contents are built-in gates (`and`, `or`, `xor`, `not`, `nand`, `nor`, `xnor`, `buf`) or other modules previously declared
2. Behavioral: describes what should be done in a module
   - module contents are C-like assignment statements, loops

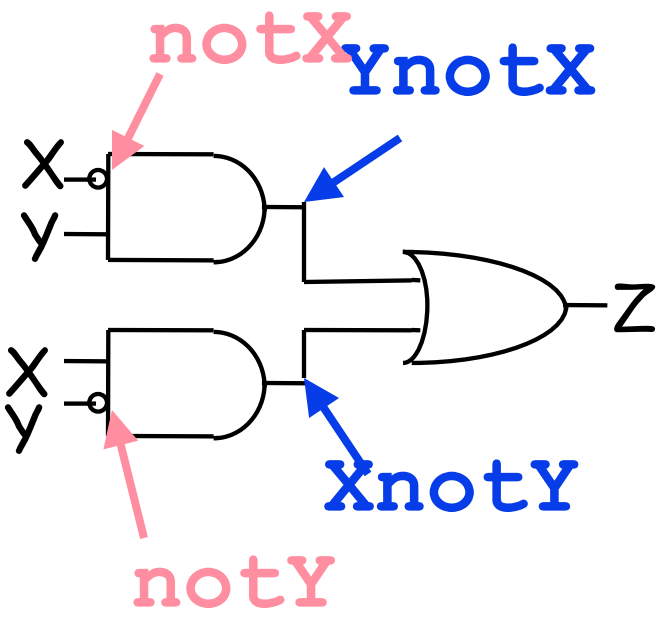
Example: Structural XOR (xor built-in, but..)

```
module xor(Z, X, Y);
  input X, Y;
  output Z;
  wire notX, notY,
    XnotY, YnotX;
  not
    (notX, X),
    (notY, Y);
  or
    (Z, YnotX, XnotY);
  and
    (YnotX, notX, Y),
    (XnotY, X, notY);
endmodule
```

Says which "ports" input, output

Default is 1 bit wide data

"nets" to connect components



Note: order of gates doesn't matter, since structure determines relationship

Example: Behavioral XOR in Verilog

```
module xorB(Z, X, Y);
  input X, Y;
  output Z;
  reg Z;
  always @ (X or Y)
      Z = X ^ Y; // ^ is C operator for xor
endmodule;
```

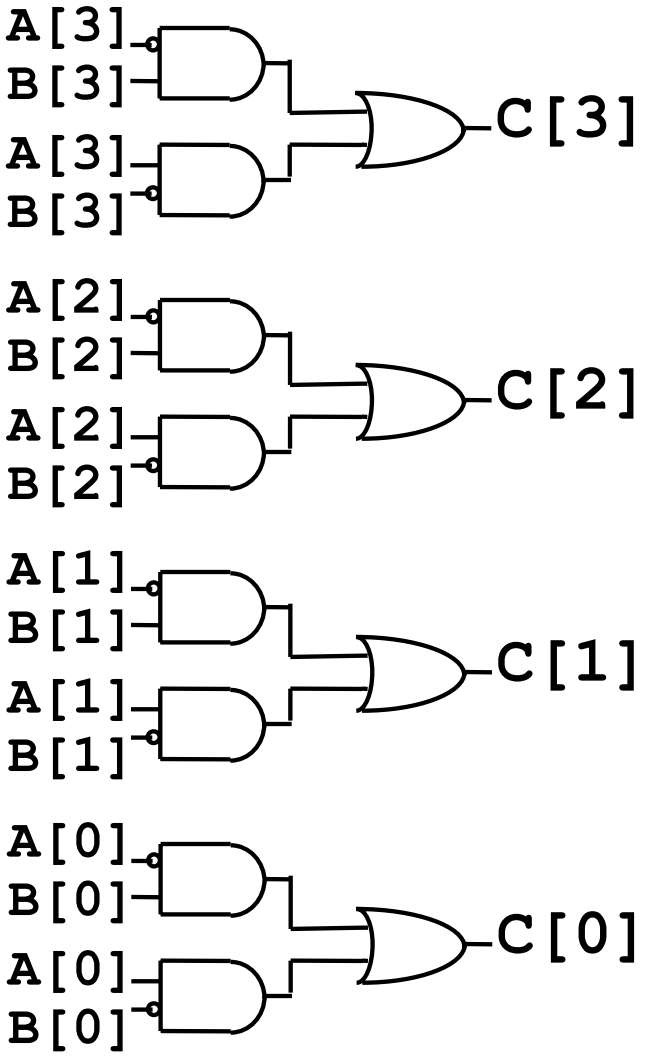
- Unusual parts of above Verilog
  - "`always @ (X or Y)`" => whenever X or Y changes, do the following statement
  - "`reg`" is only type of behavioral data that can be changed in assignment, so must redeclare `Z` as `reg`
  - Default is single bit data types: X, Y, Z

Verilog: replication, hierarchy

- Often in hardware need many copies of an item, connected together in a regular way
  - Need way to name each copy
  - Need way to specify how many copies
- Specify a module with 4 XORs using existing module example

Example: Replicated XOR in Verilog

```
module 4xor(C, A, B);
  input [3:0] A, B;
  output[3:0] C;
  xor foo4xor[3:0]
    (.X(A), .Y(B), .Z(C) );
endmodule;
```



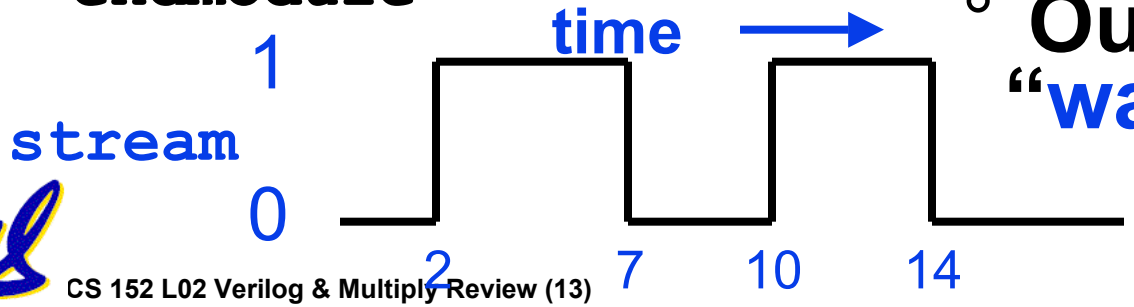
- Note 1: can associate ports explicitly by name,
  - `(.X (A), .Y(B), .Z(C))`
- or implicitly by order (as in C)
  - `(C, A, B)`
- Note 2: must give a name to new instance of xors (`foo4xor`)

Verilog big idea: Time in code

- Difference from normal prog. lang. is that time is part of the language
  - part of what trying to describe is when things occur, or how long things will take
- In both structural and behavioral Verilog, determine time with `#n` : event will take place in n time units
  - structural: `not #2 (notX, X)` says notX does not change until time advances 2 ns
  - behavioral: `#2 Z = A ^ B;` says Z does not change until time advances 2 ns
  - Default unit is nanoseconds; can change

Example:

```
module test(stream);
   output stream;
   reg stream;
   initial
      begin
         stream = 0;
         #2 stream = 1;
         #5 stream = 0;
         #3 stream = 1;
         #4 stream = 0;
      end
endmodule
```



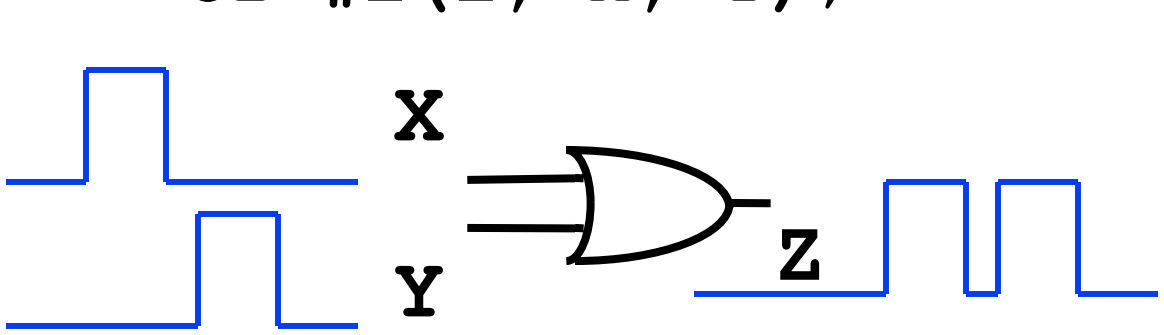
- "Initial" means do this code once
- Note: Verilog uses `begin ... end` vs. { ... } as in C
- `#2 stream = 1` means wait 2 ns before changing stream to 1
- Output called a "waveform"

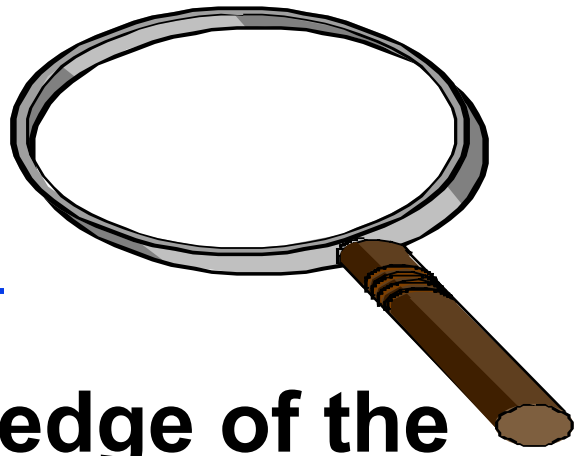
Time and updates

- When do assignments take place with respect to $time?
- At next update of clock

Time, variable update, and monitor

```
or #2(Z, X, Y);
```





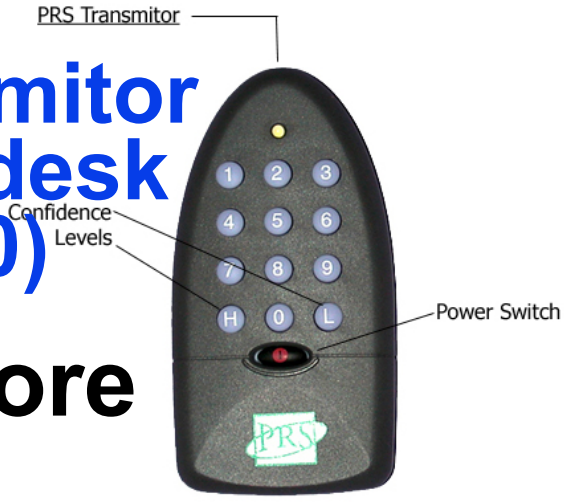
- The instant before the rising edge of the clock, all outputs and wires have thier OLD values. This includes inputs to flip flops. Therefore, if you change the inputs to a flip flop at a particular rising edge, that change will not be reflected at the output until the NEXT rising edge. This is because when the rising edge occurs, the flip flop still sees the old value. So when simulated time changes in Verilog, then ports, registers updated

Testing in Verilog

- Code above just defined a new module
- Need separate code to test the module (just like C/Java)
- Since hardware is hard to build, major emphasis on testing in HDL
- Testing modules called "test benches" in Verilog;
  - like a bench in a lab dedicated to testing
- Can use time to say how things change

Administrivia

- Prerequisite Quiz Friday
  1. 11 AM - 1 PM 320 Soda (John)
  2. 2 PM - 4 PM 4 Evans (Kurt)
  3. 3 PM - 5 PM 81 Evans (Jack)
- Lab 1 due next Wednesday (Mon. holiday)
- Tuesday: buy $37 PRS Transmitor from behind ASUC textbook desk (Chem 1A, CS 61ABC, 160)
  - Can sell back to bookstore



Example Verilog Code

```
//Test bench for 2-input multiplexor.
// Tests all input combinations.
module testmux2;
 reg [2:0] c;
 wire f;
 reg expected;
 mux2 myMux (.select(c[2]), .in0(c[0]),
 .in1(c[1]), .out(f));
 initial
  begin
   c = 3'b000; expected=1'b0; ...
```

- Verilog constants syntax N'Bxxx where
  N is size of constant in bits
  B is base: b for binary, h for hex, o for octal
  xxx are the digits of the constant

Example Verilog Code

```
… begin
    c = 3'b000; expected=1'b0;
    repeat(7)
     begin
       #10 c = c + 3'b001;
       if (c[2]) expected=c[1];
           else expected=c[0];
     end
    #10 $finish;
  end
```

- Verilog `if` statement, `for` and `while` loops like C
  - `repeat (n)` loops for n times (restricted `for`)
  - `forever` is an infinite loop
- Can select a bit of variable (`c[0]`)
  - `$finish` ends simulation

Rising and Falling Edges and Verilog

- Challenge of hardware is when do things change relative to clock?
  - Rising clock edge? ("positive edge triggered")
  - Falling clock edge? ("negative edge triggered")
  - When reach a logical level? ("level sensitive")
- Verilog must support any "clocking methodology"
- Includes events "posedge", "negedge" to say when clock edge occur, and "wait" statements for level

Verilog Odds and Ends

- Memory is register with second dimension
```
reg [31:0] memArray [0:255];
```
- Can assign to group on Left Hand Side
```
{Cout, sum} = A + B + Cin;
```
- Can connect logical value 0 or 1 to port via `supply0` or `supply1`
- If you need some temporary variables (e.g., for loops), can declare them as `integer`
- Since variables declared as number of bits, to place a string need to allocate 8 * number of characters
```
reg [1 : 6*8] Msg;
Msg = "abcdef";
```

Blocking v. Nonblocking assignment

- Blocking assignment statement (= operator) like in traditional programming languages
  - The whole statement is done before control passes on to the next statement.
- Non-blocking (<= operator) evaluates all right-hand sides for current time unit and assigns left-hand sides at the end of the time unit
  - Use old values of variables at beginning of current time unit and to assign registers new values at end of current time unit. This reflects how register transfers occur in some hardware systems.

Blocking v. Nonblocking Example

```
 // testing blocking and non-blocking assignment
module blocking;
reg [0:7] A, B;
initial begin: init1
        A = 3;
        #1 A = A + 1;    // blocking procedural assignment
           B = A + 1;
           $display("Blocking:     A= %b B= %b", A, B );

        A = 3;
        #1 A <= A + 1;  // non-blocking procedural assignment
           B <= A + 1;

        #1 $display("Non-blocking: A= %b B= %b", A, B );
end

endmodule
```

° **Above module produces the following output:**

```
Blocking:     A= 00000100 B= 00000101
Non-blocking: A= 00000100 B= 00000100
```

Peer Instruction

- How many mistakes in this module?

```
module test(X);
    output X;
    initial
        begin
            X = 0;
            X = 1;
        end;
end;
```

A. None

B. 1

C. 2

D. 3

E. 4

F. >=5

Peer Instruction

- How many mistakes in this module?

```
module test2(X);
    output X; reg X; // so can change
    initial
        begin
            X = 0;
            #2 X = 1; // no time delay
        end_   // no ;
endmodule_ // Ada style end, and no ;
```

A. None
B. 1
C. 2
D. 3
E. 4
F. >=5

Peer Instruction

- Suppose writing a MIPS interpreter in Verilog. Which sequence below is best organization for the interpreter?

I. A `repeat` loop that fetches instructions

II. A `while` loop that fetches instructions

III. Increments PC by 4

IV. Decodes instructions using `case` statement

V. Decodes instr. using chained `if` statements

VI. Executes each instruction

A. I, IV, VI

B. II, III, IV, VI

C. II, V, VI, III

D. IV, VI, III, II

E. V, VI, III, I

F. VI, III, II

Peer Instruction

- Suppose writing a MIPS interpreter in Verilog. Which sequence below is best organization for the interpreter?

I. A `repeat` loop that fetches instructions

II. A `while` loop that fetches instructions

III. Increments PC by 4

IV. Decodes instructions using `case` statement

V. Decodes instr. using chained `if` statements

VI. Executes each instruction

A. I, IV, VI

B. II, III, IV, VI

C. II, V, VI, III

D. IV, VI, III, II

E. V, VI, III, I

F. VI, III, II

Verilog conclusion

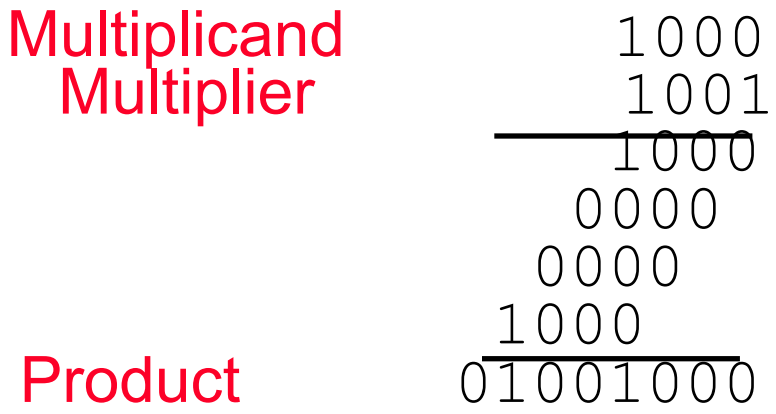
- Verilog allows both structural and behavioral descriptions, helpful in testing
- Syntax a mixture of C (operators, for, while, if, print) and Ada (begin… end, case…endcase, module …endmodule)
- Some special features only in Hardware Description Languages
  - # time delay, initial vs. always, forever loops
- Verilog can describe everything from single gate to full computer system; you get to design a simple processor

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):



° m bits x n bits = m+n bit product

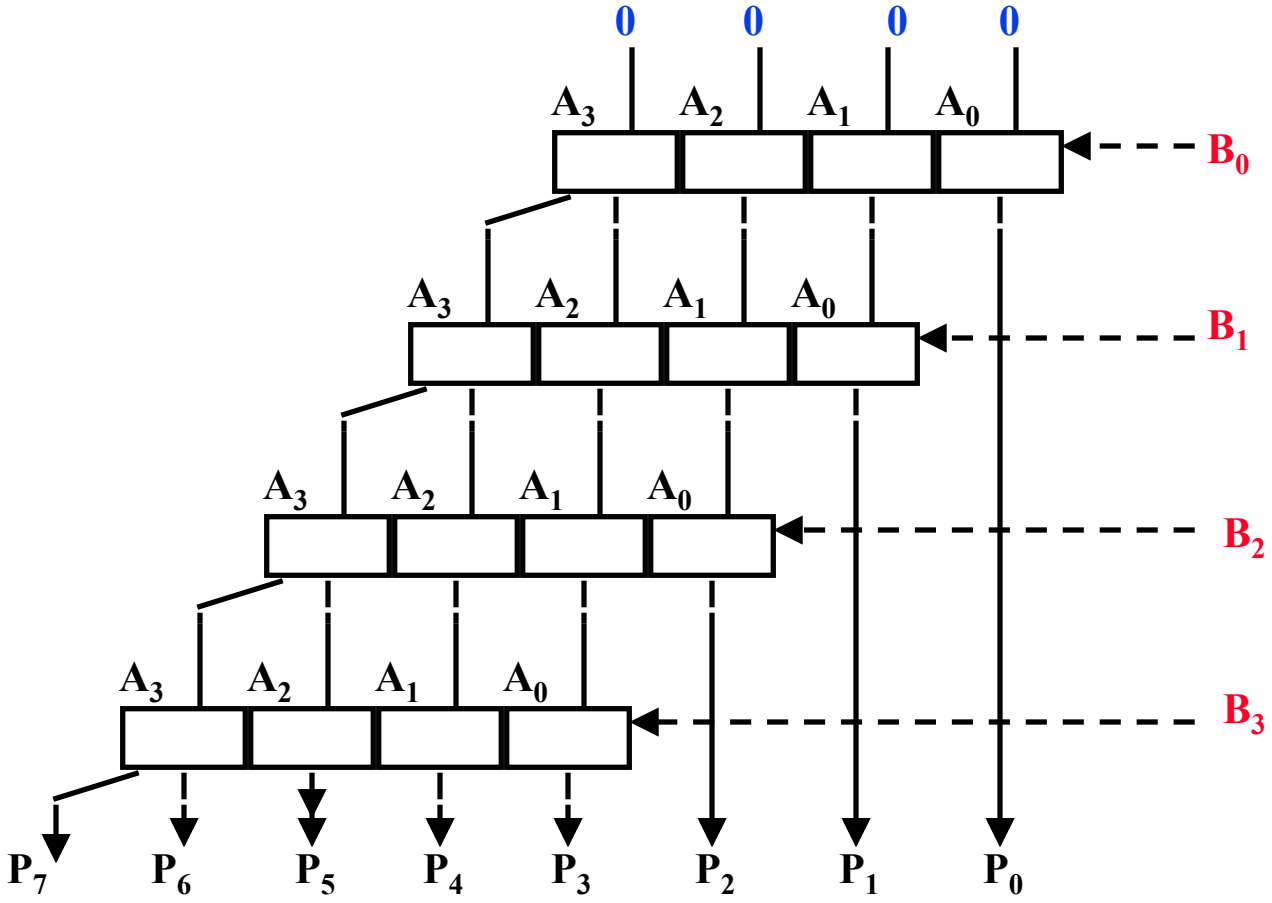
° Binary makes it easy:

- 0 => place 0 (0 x multiplicand)
- 1 => place a copy (1 x multiplicand)

° 4 versions of multiply hardware & algorithm:

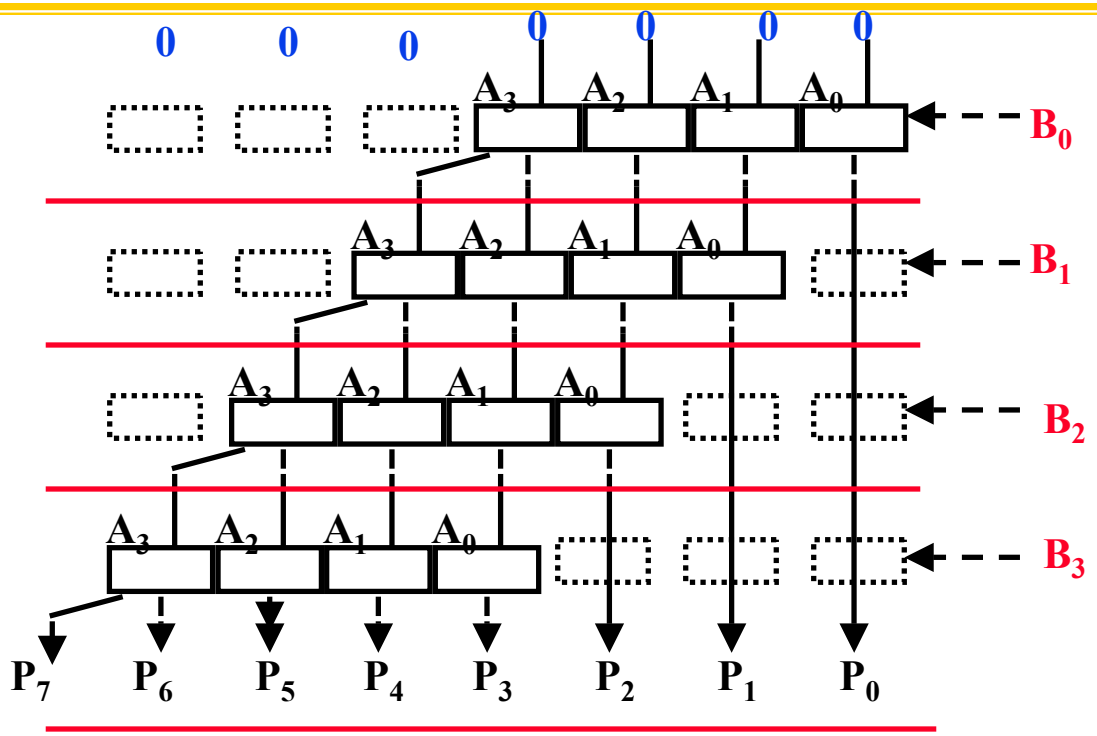
- successive refinement

Unsigned Combinational Multiplier



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

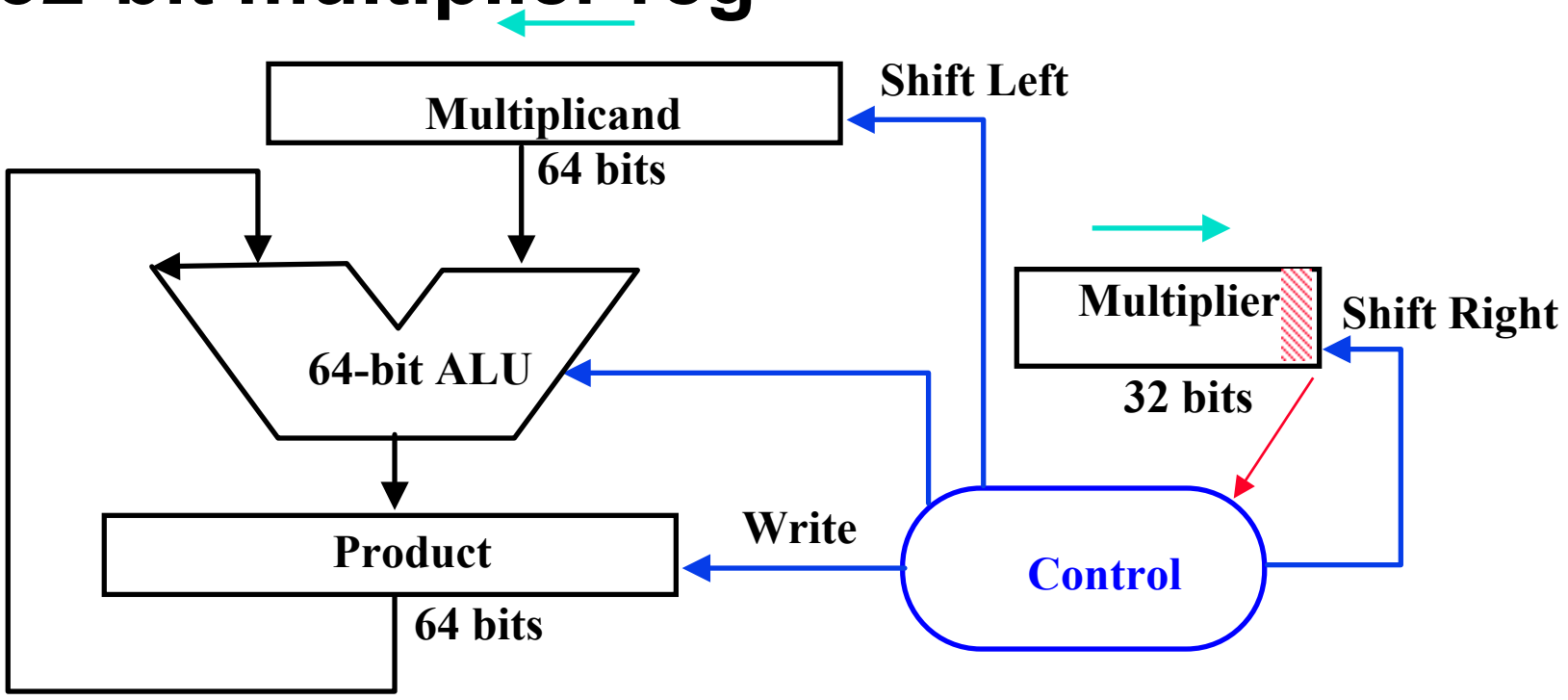
How does it work?



- 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

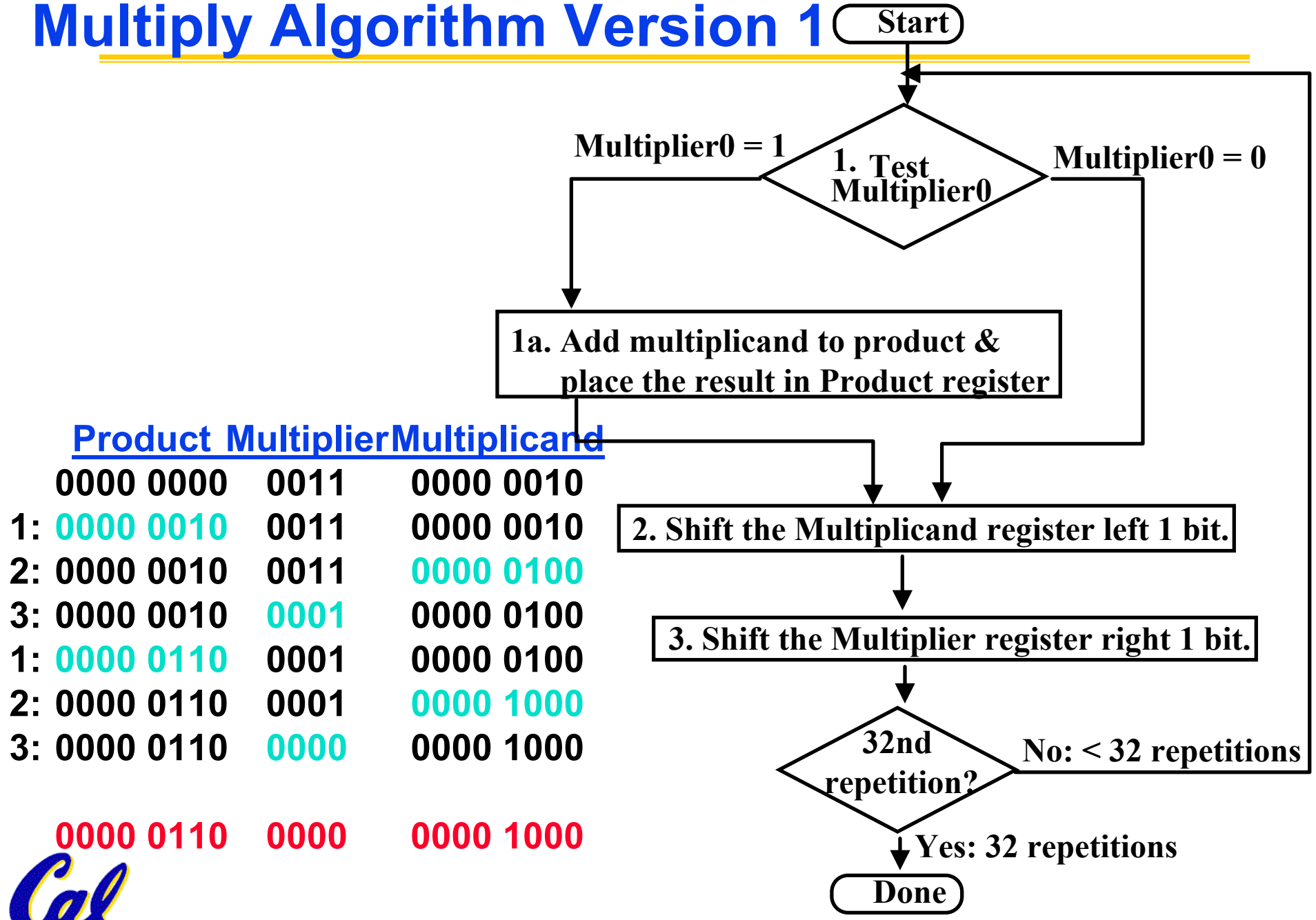
Unisigned shift-add multiplier (version 1)

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



Multiplier = datapath + control

Multiply Algorithm Version 1



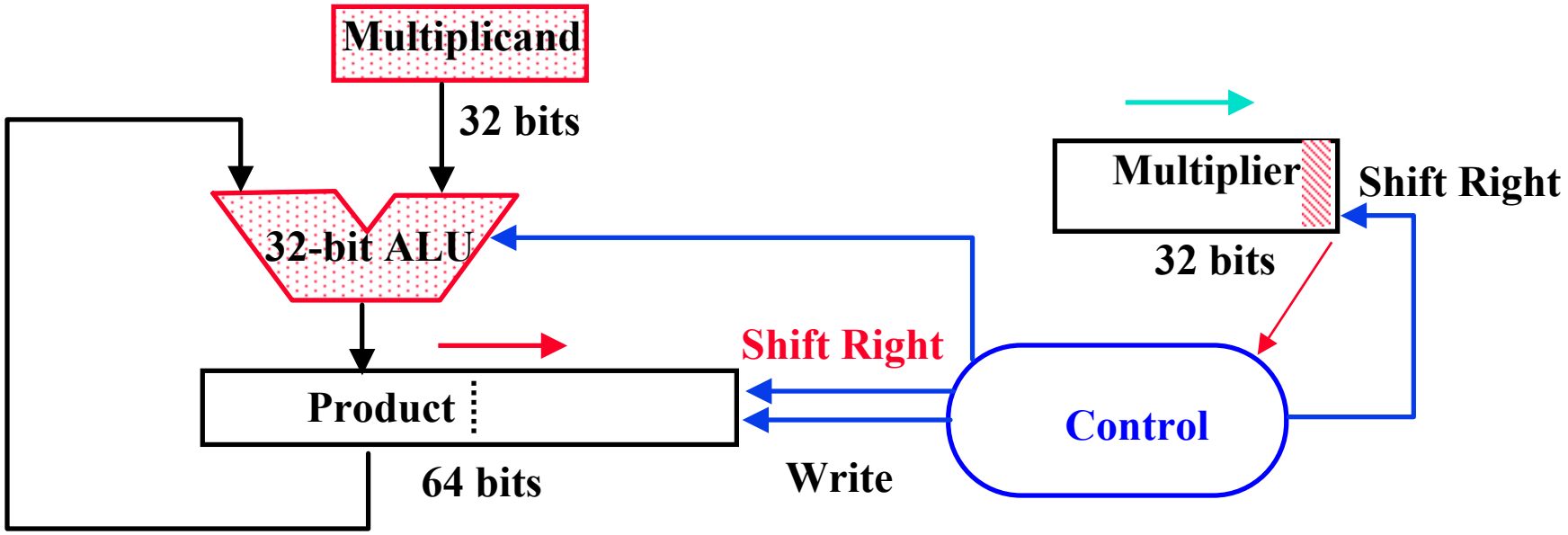
	Product	Multiplier	Multiplicand
	0000 0000	0011	0000 0010
1:	0000 0010	0011	0000 0010
2:	0000 0010	0011	0000 0100
3:	0000 0010	0001	0000 0100
1:	0000 0110	0001	0000 0100
2:	0000 0110	0001	0000 1000
3:	0000 0110	0000	0000 1000
	0000 0110	0000	0000 1000

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 right of multiplicand as shifted => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

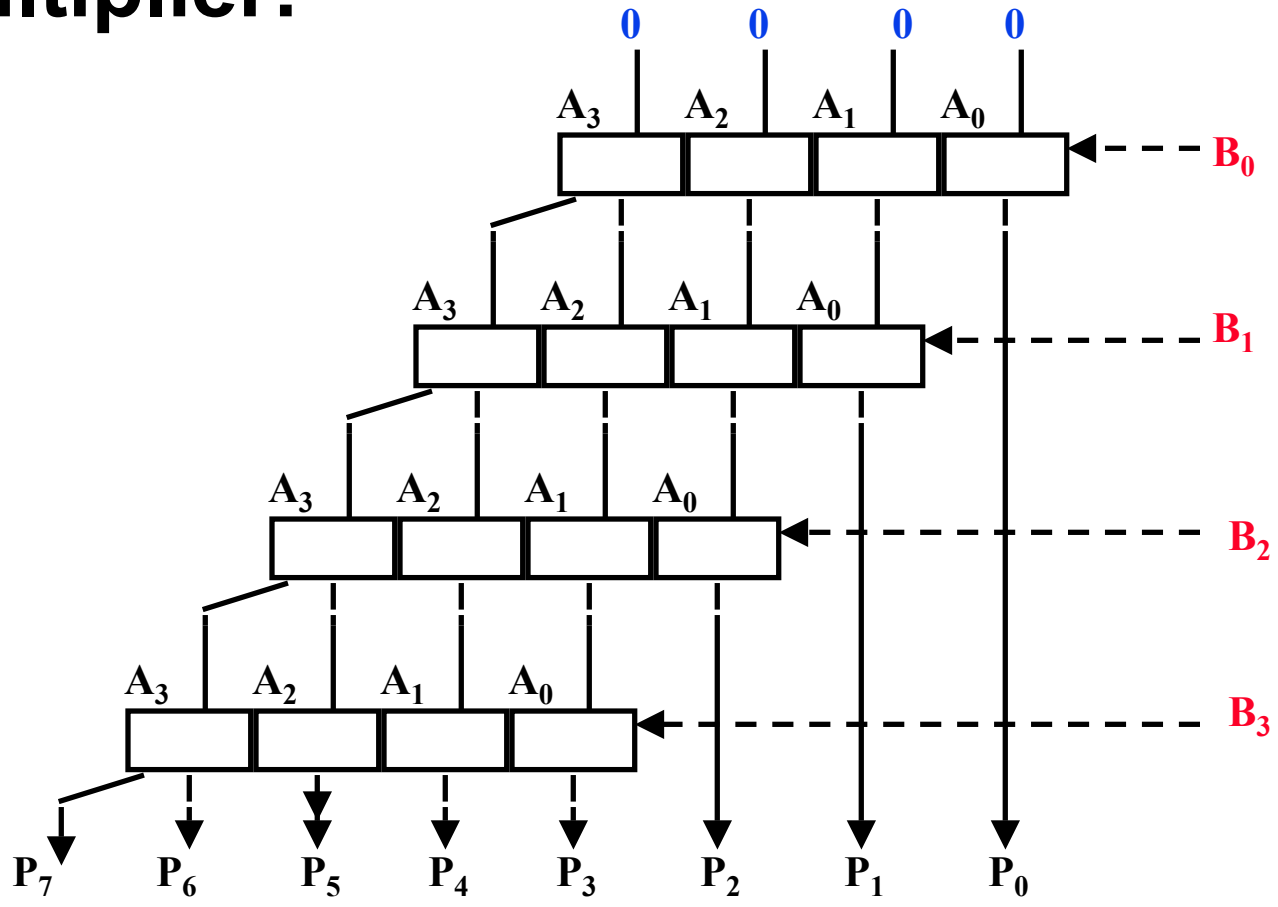
MULTIPLY HARDWARE Version 2

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

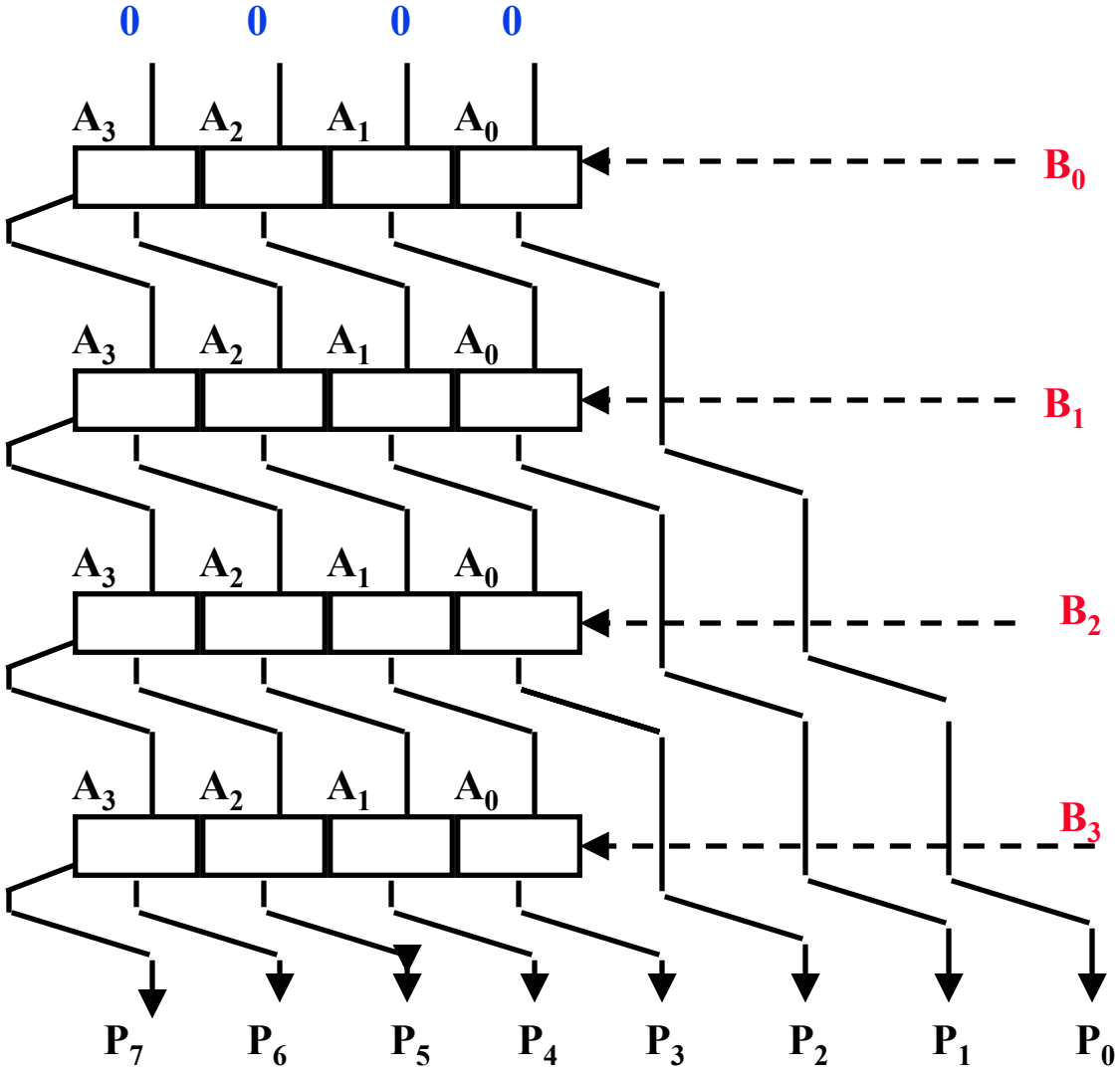


How to think of this?

Remember original combinational multiplier:

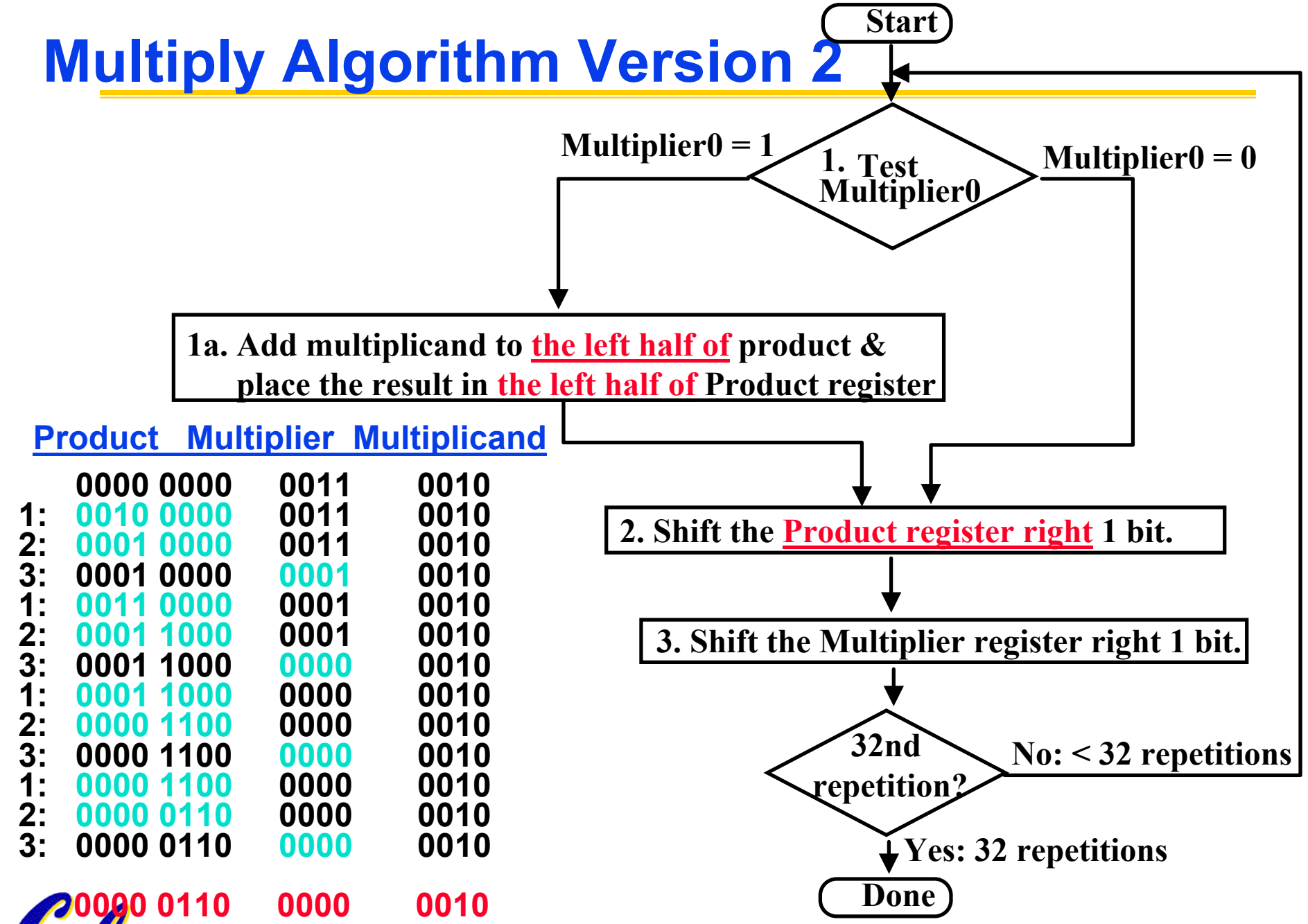


Simply warp to let product move right...



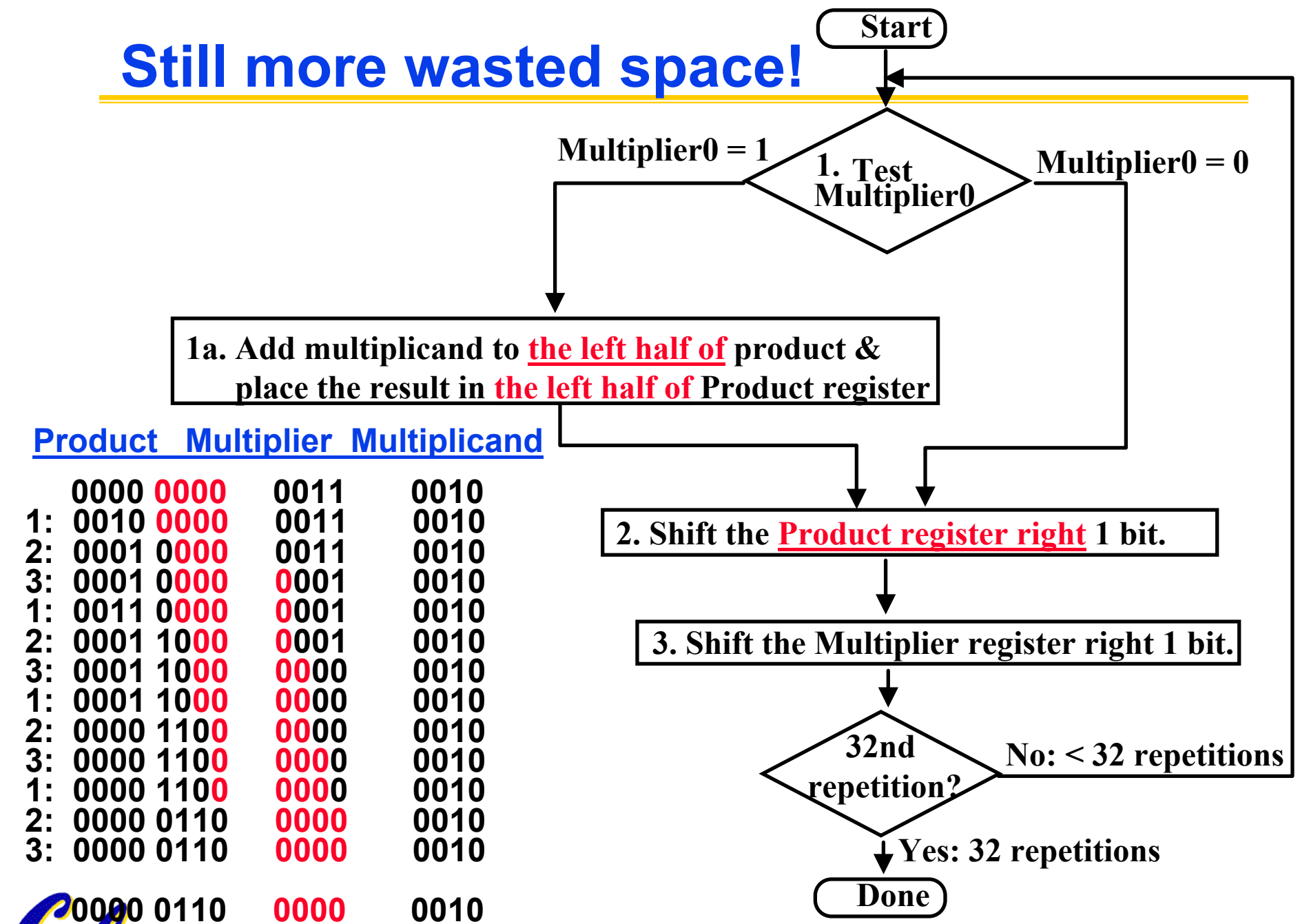
° Multiplicand stay's still and product moves right

Multiply Algorithm Version 2



	Product	Multiplier	Multiplicand
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

Still more wasted space!



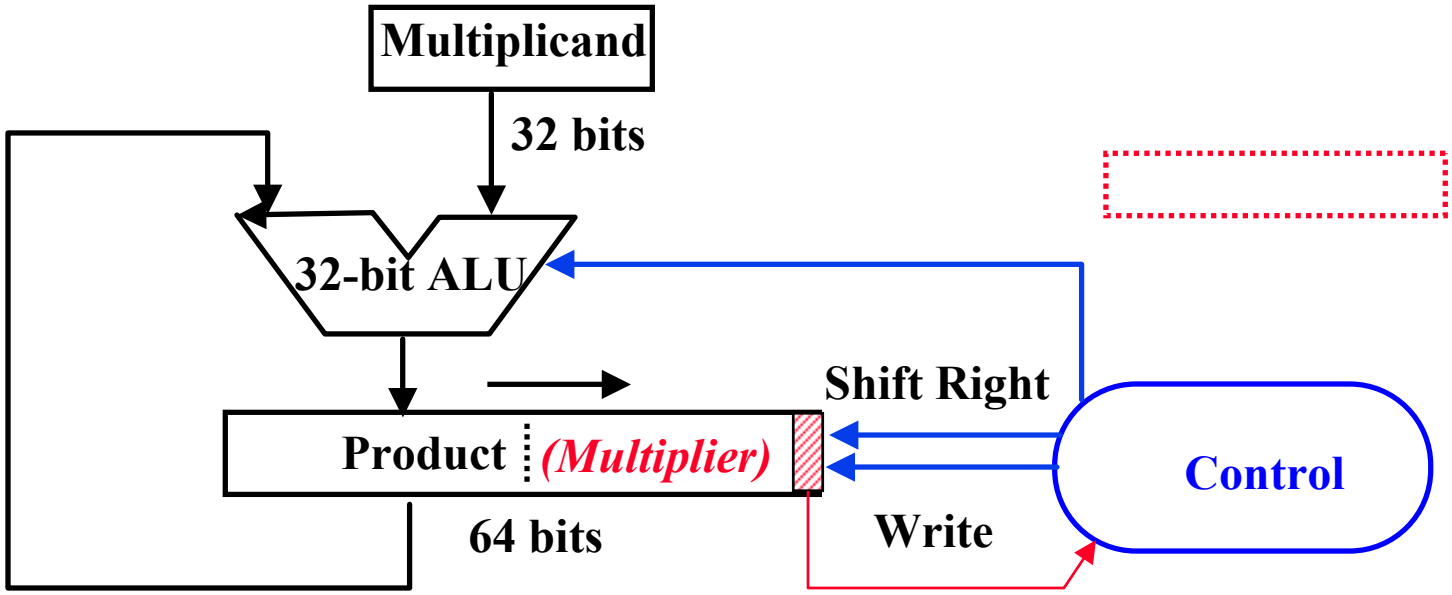
	Product	Multiplier	Multiplicand
	0000	0000	0011	0010
1:	0010	0000	0011	0010
2:	0001	0000	0011	0010
3:	0001	0000	0001	0010
1:	0011	0000	0001	0010
2:	0001	1000	0001	0010
3:	0001	1000	0000	0010
1:	0001	1000	0000	0010
2:	0000	1100	0000	0010
3:	0000	1100	0000	0010
1:	0000	1100	0000	0010
2:	0000	0110	0000	0010
3:	0000	0110	0000	0010
	0000	0110	0000	0010

Observations on Multiply Version 2

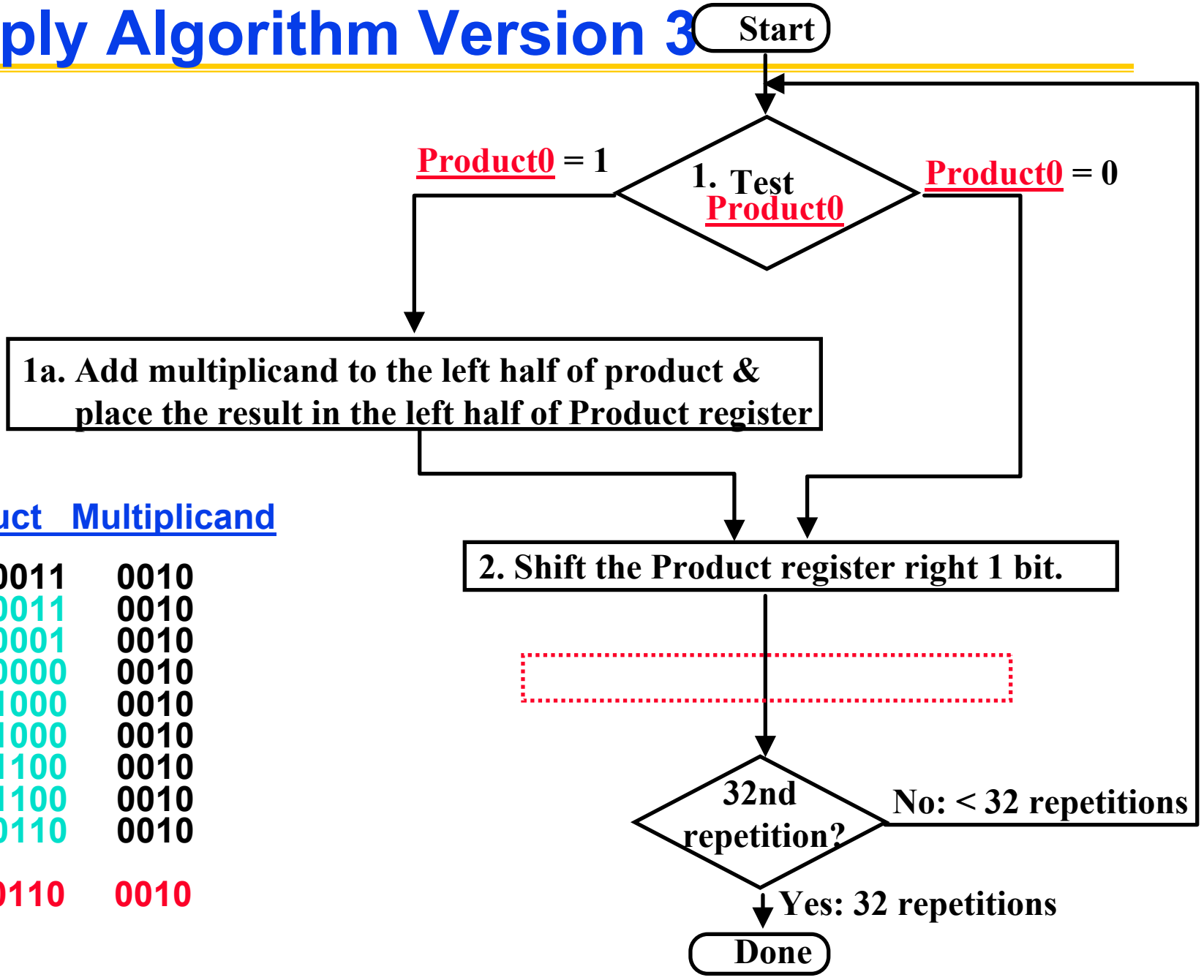
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

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



Multiply Algorithm Version 3



	Product	Multiplicand
	0000 0011	0010
1:	0010 0011	0010
2:	0001 0001	0010
1:	0011 0000	0010
2:	0001 1000	0010
1:	0001 1000	0010
2:	0000 1100	0010
1:	0000 1100	0010
2:	0000 0110	0010
	0000 0110	0010

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

In Conclusion...

- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39 in COD)