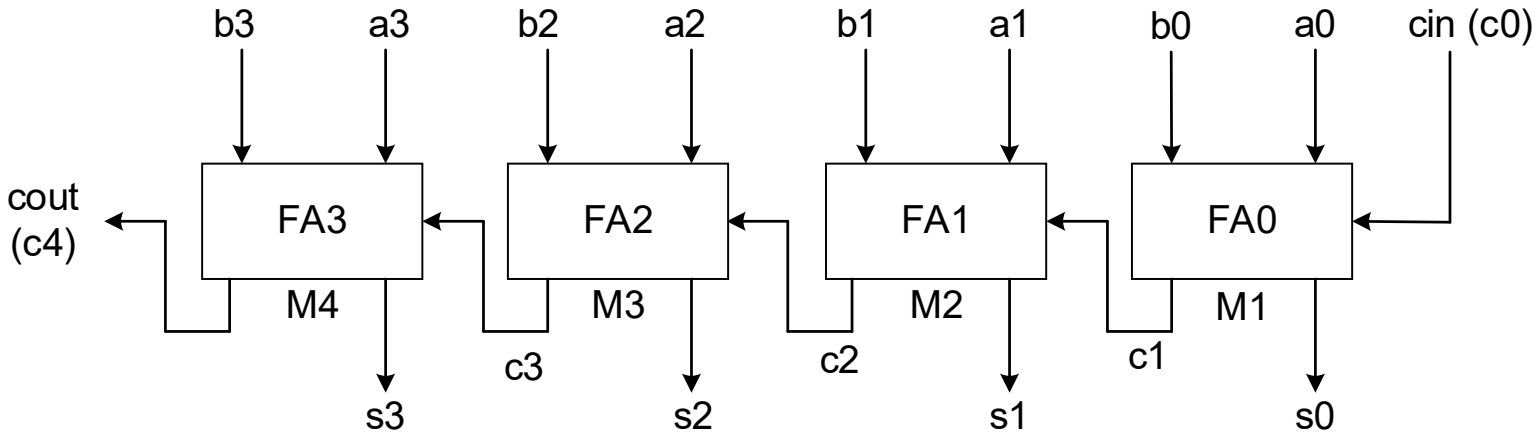
Lecture 7 – Adders and Multipliers

Ripple Carry Adder



A	B	cin	cout	
0	0	0	0	Kill
0	0	1	0	
0	1	0	0	Propagate
1	0	0	0	
0	1	1	1	
1	0	1	1	
1	1	0	1	Generate
1	1	1	1	

- Key observations, the value of the carry into any stage of a multi-cell adder depends only on
  - The data bit of previous stage
  - The carry into the 1st stage
  - When both inputs 0, no carry
  - When one is 0, the other is 1, *propagate* carry input
  - When both are 1, then *generate* a carry

Carry-lookahead adder

- Generate

  Gi = ai * bi

- Propagate

  Pi = ai xor bi; or Pi = ai + bi

- Pi and Gi are mutually exclusive
  - Because Pi is asserted when (ai,bi)={(1,0), (0,1)};

    Gi is asserted when (ai,bi)=(1,1)

- Si = ai xor bi xor ci = Pi xor ci

- c(i+1) = (ai+bi)*ci + a*b = (ai xor bi) * ci + a*b = Pi* ci + Gi

- Write carry out as function of preceding G, P, and cout

  c1 = G0 + P0*c0

  c2 = G1 + P1*c1

  c3 = G2 + P2*c2

  c4 = G3 + P3*c3

Reducing the complexity

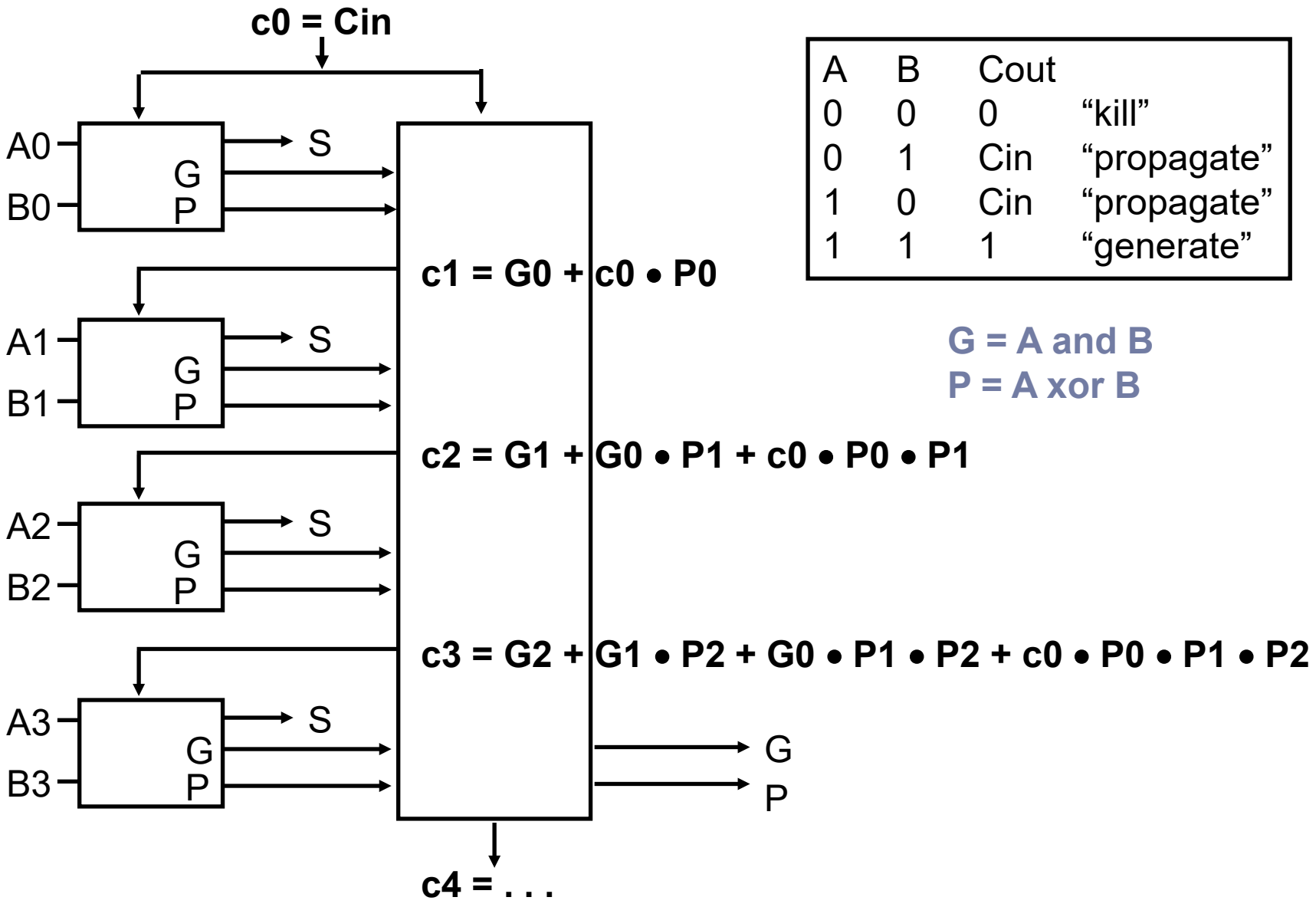
- c1 = G0 + (P0 * c0)
- c2 = G1 + (P1 * [G0 + P0 * c0])
  = G1 + (P1 * G0) + (P1 * P0 * c0)
- c3 = G2 + (P2 * G1) + (P2 * P1 * G0) +
  (P2 * P1 * P0 * c0)

That is ci can only use c0 and P(i-1) ... P0 and G(i-1) ... G0

Increase speed at what cost ?

Can you illustrate how to build a 32-bit adder with carry look ahead?

Carry Look Ahead Adder

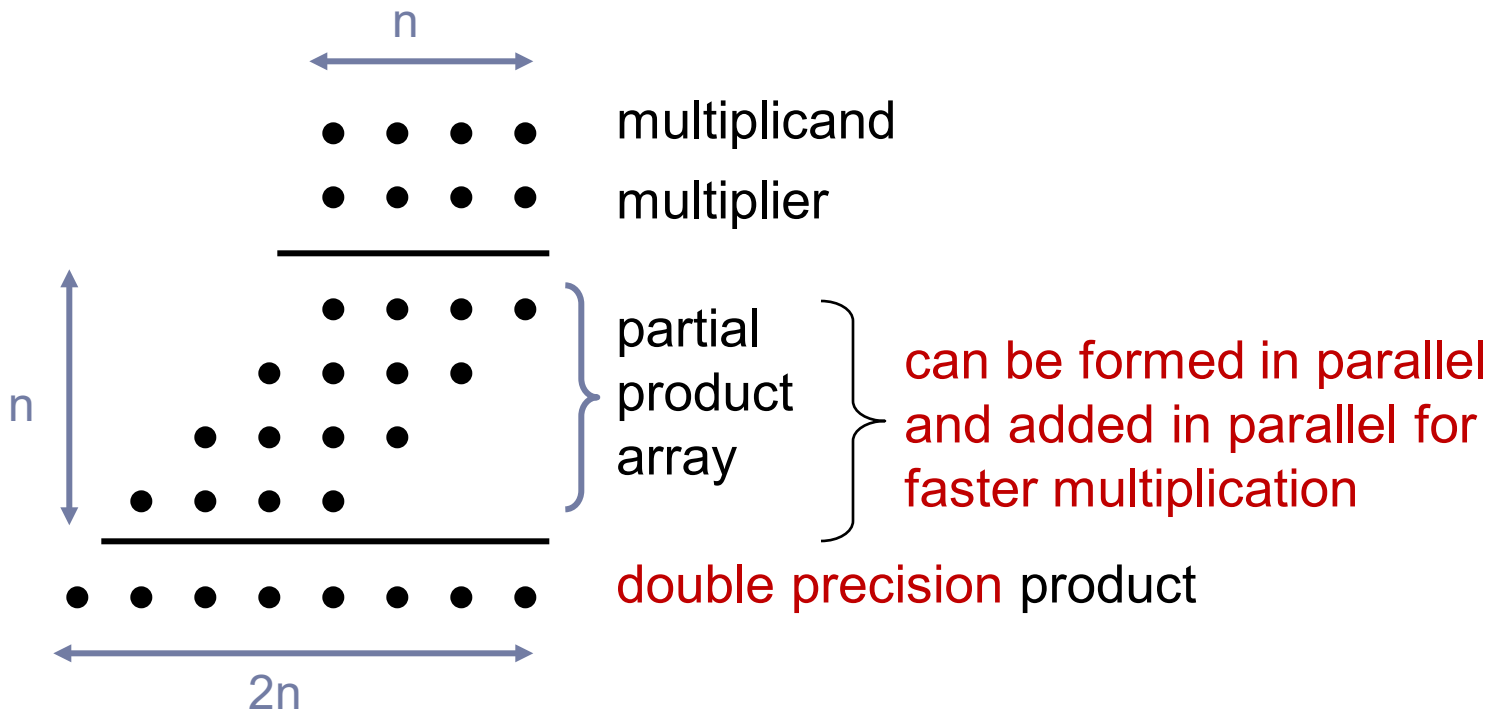


A	B	Cout	
0	0	0	“kill”
0	1	Cin	“propagate”
1	0	Cin	“propagate”
1	1	1	“generate”

G = A and B
P = A xor B

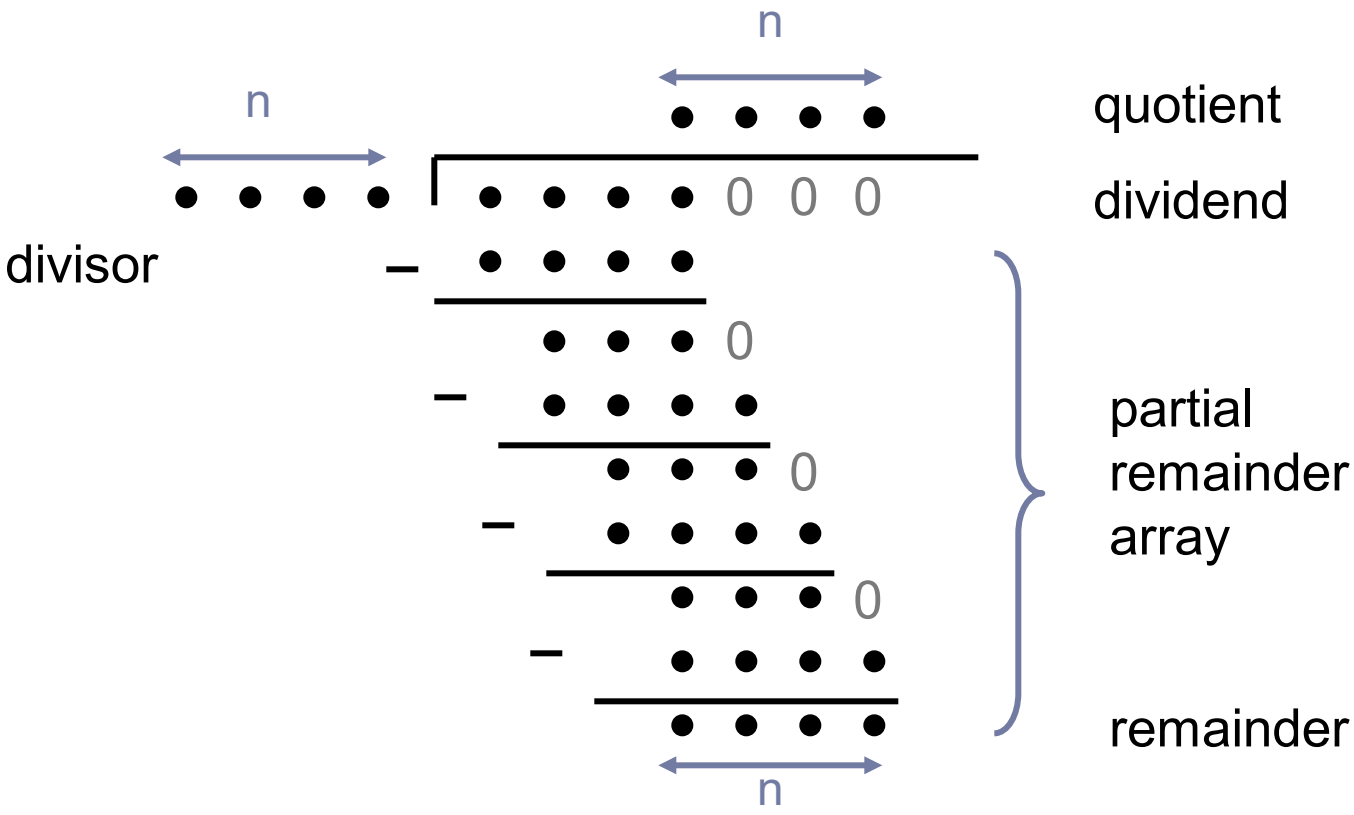
Multiply Overview

- Binary multiplication is just a *bunch* of left shifts and adds



Division Overview

- Division is just a *bunch* of quotient digit guesses and right shifts and subtracts

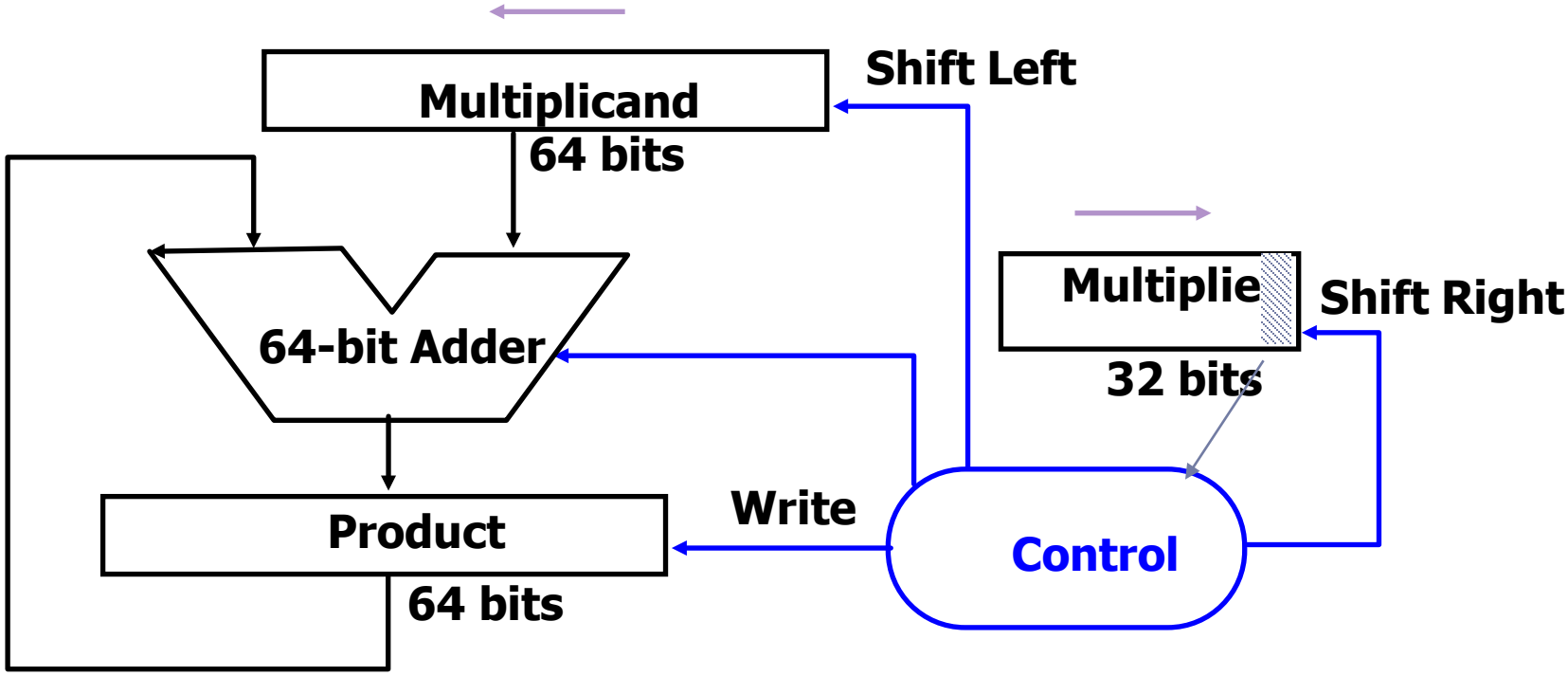


Multiplication: design and implementation

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
  - m bits x n bits = m+n bit product
- Let's look at 3 (unsigned) versions of multiplication designs in the next few slides

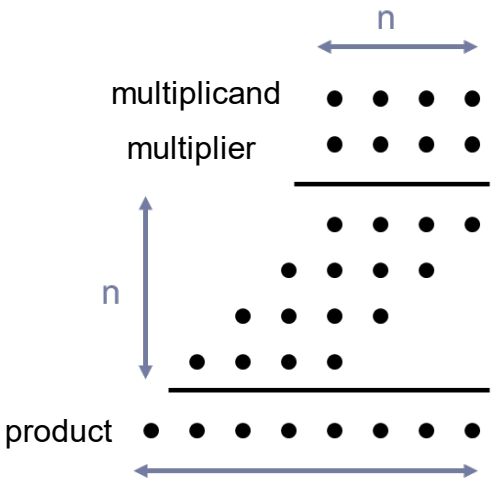
Unisigned shift-add multiplier (version 1)

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

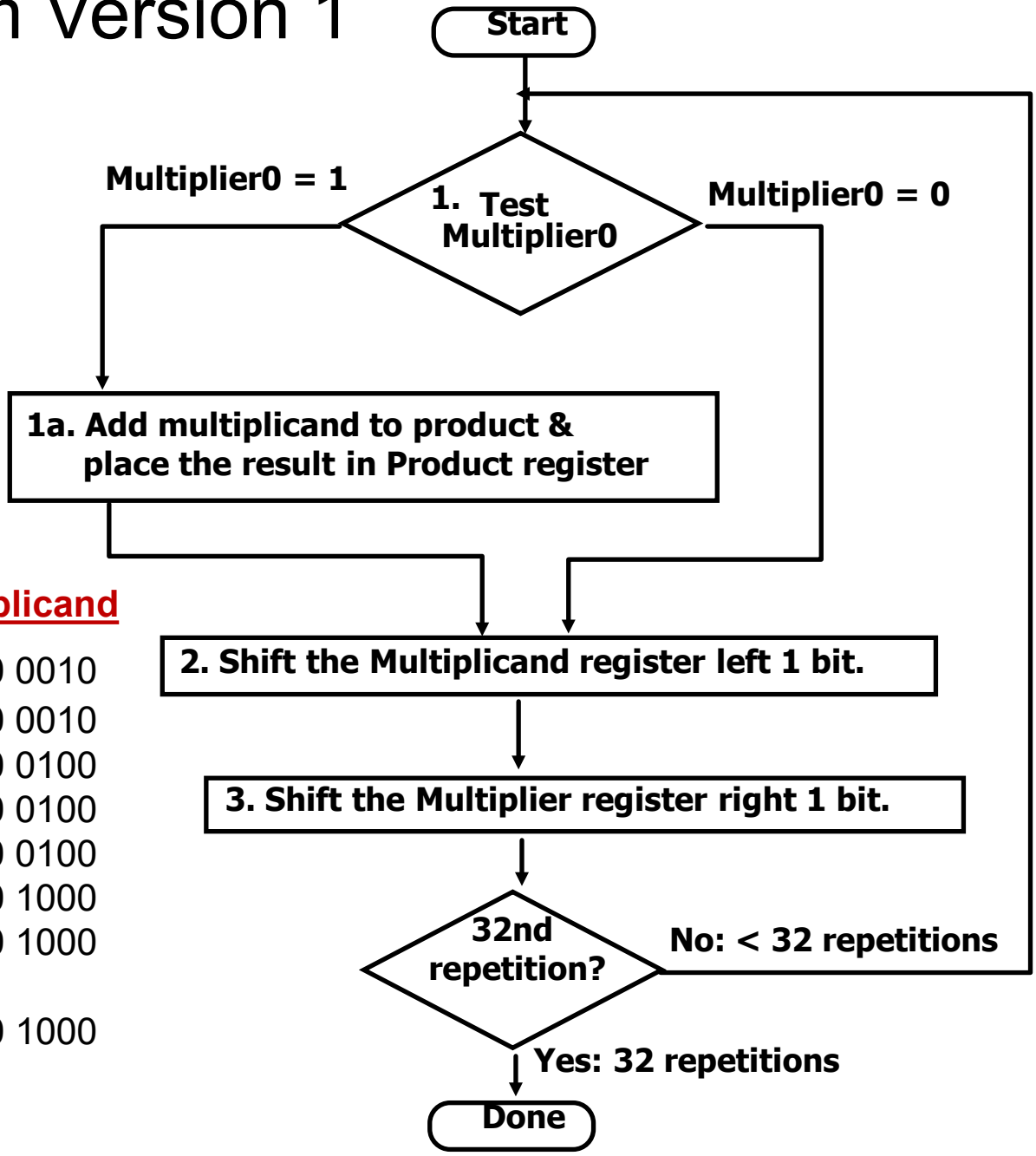


Multiplier = datapath + control

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

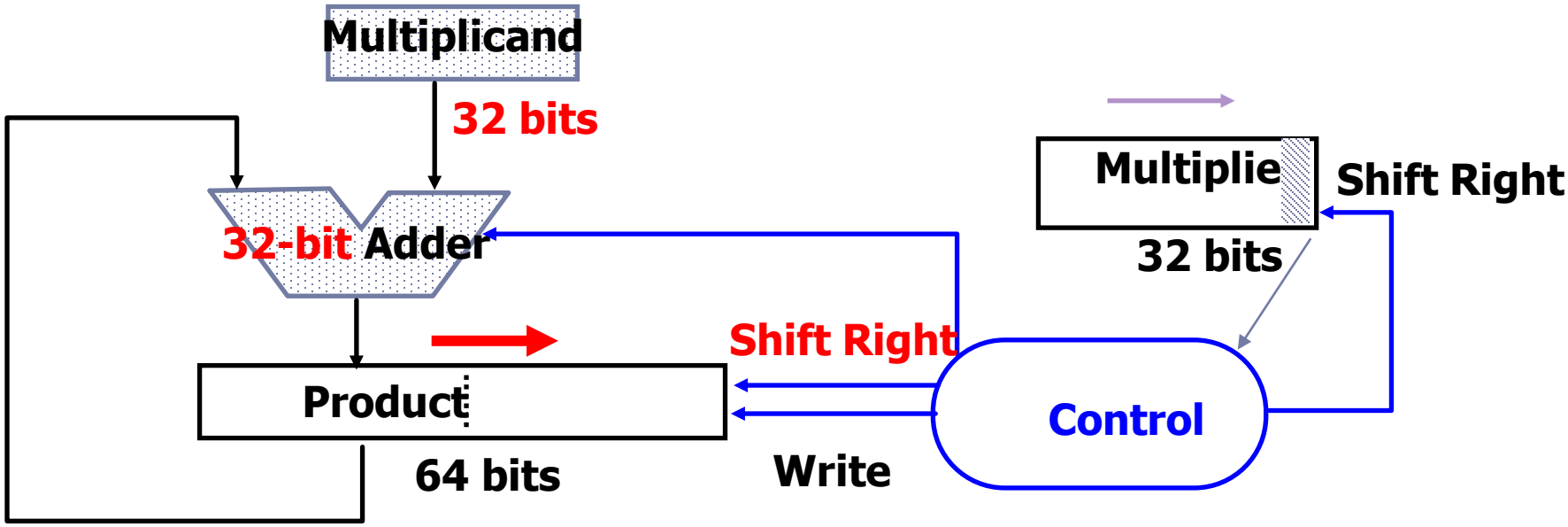
- 1 clock per cycle => ≈ 100 clocks per multiply because of 32 repetitions, 3 steps in one repetition
  - Ratio of add/sub to multiply is from 5:1 to 100:1
  - Slow
- 0's inserted in the rightmost bit of multiplicand as shifting left
  => least significant bits of product never changed once formed
- 1/2 bits in multiplicand always 0
  - MSB are 0s at the beginning
  - 0 is inserted in LSB as multiplicand shifting left

  => 64-bit multiplicand register is wasted

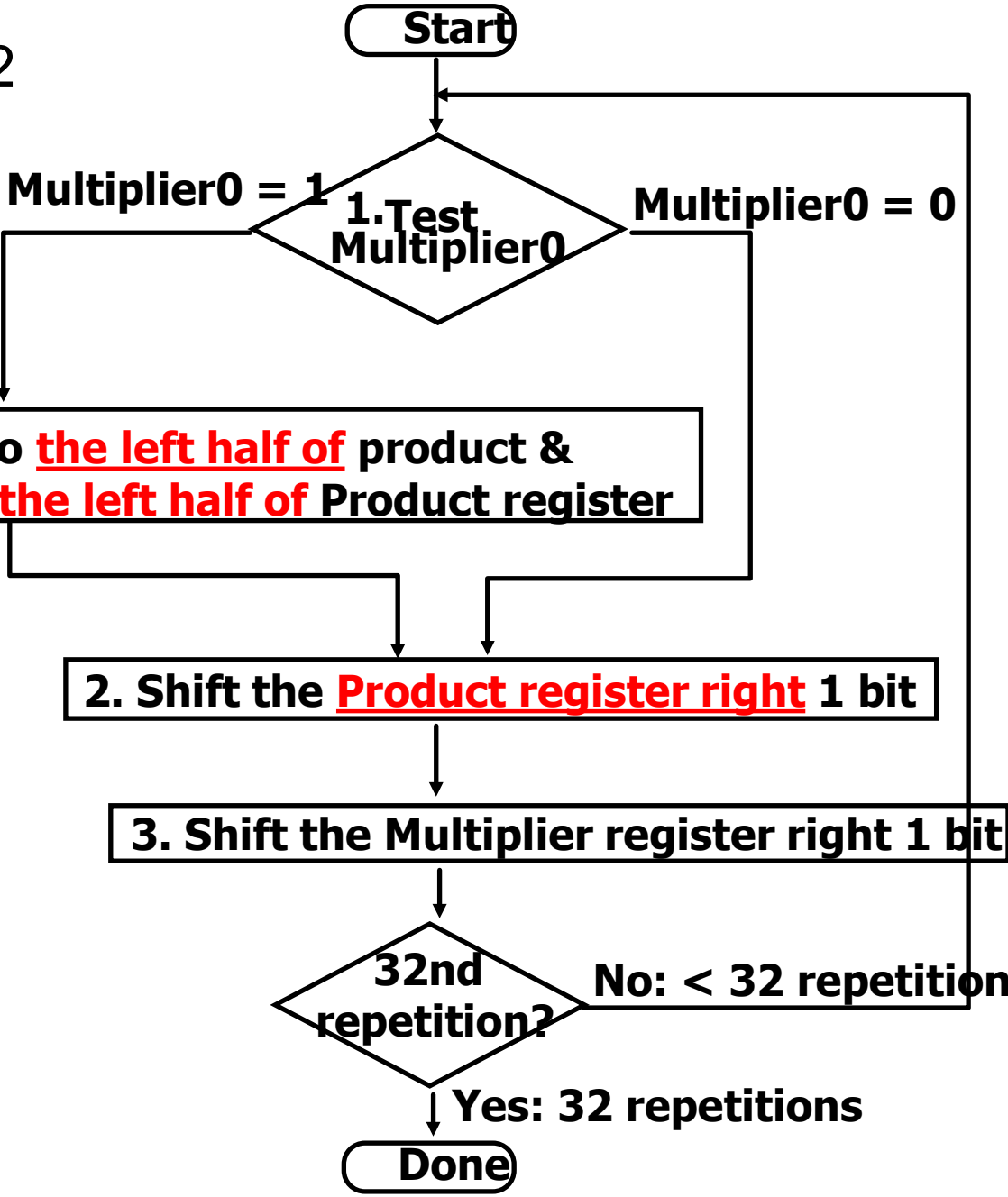
  => 64-bit adder is wasted
- Instead of shifting multiplicand to left, let's shift *product* to right

MULTIPLY HARDWARE Version 2

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



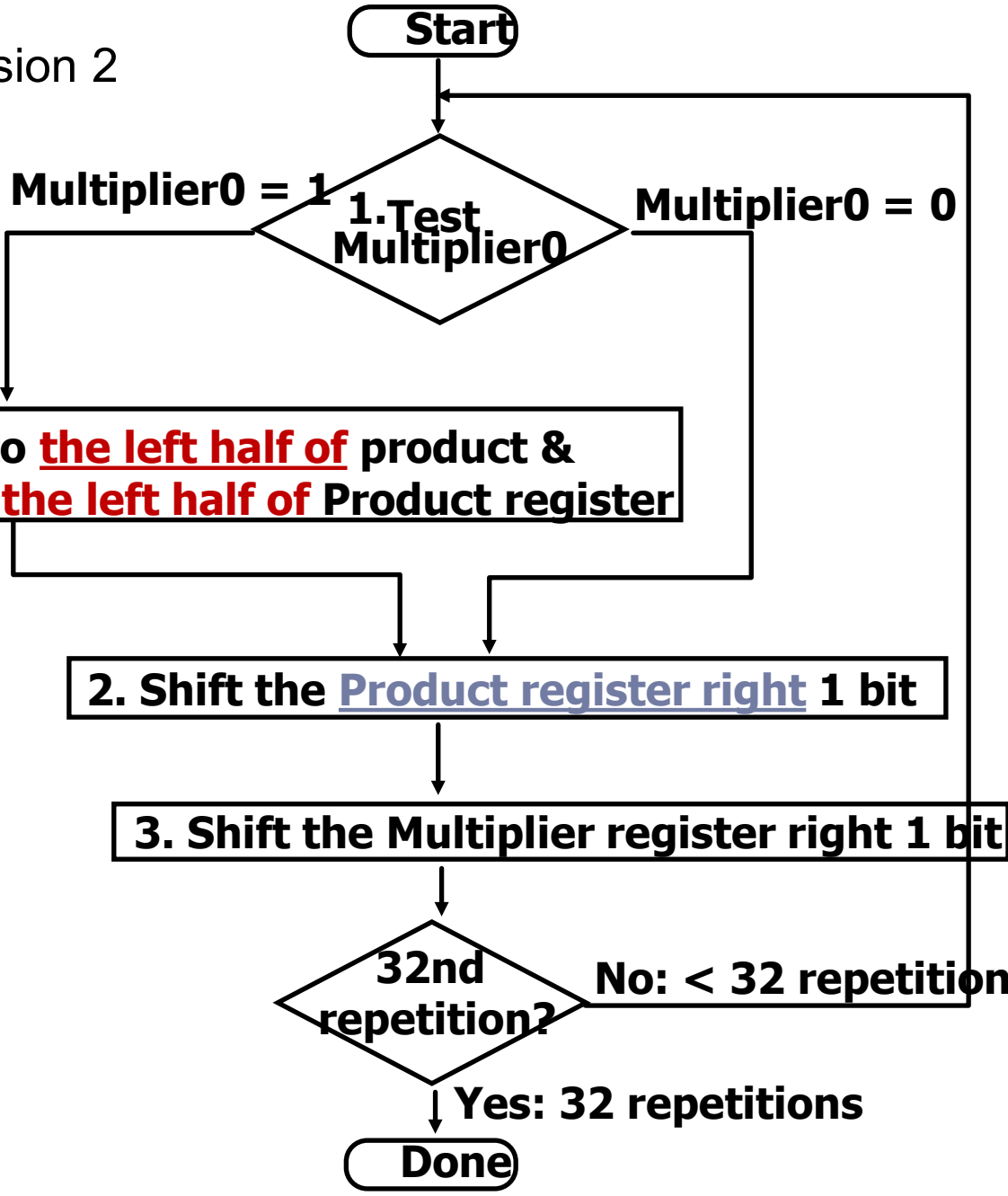
Multiply Algorithm Version 2



	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 in 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

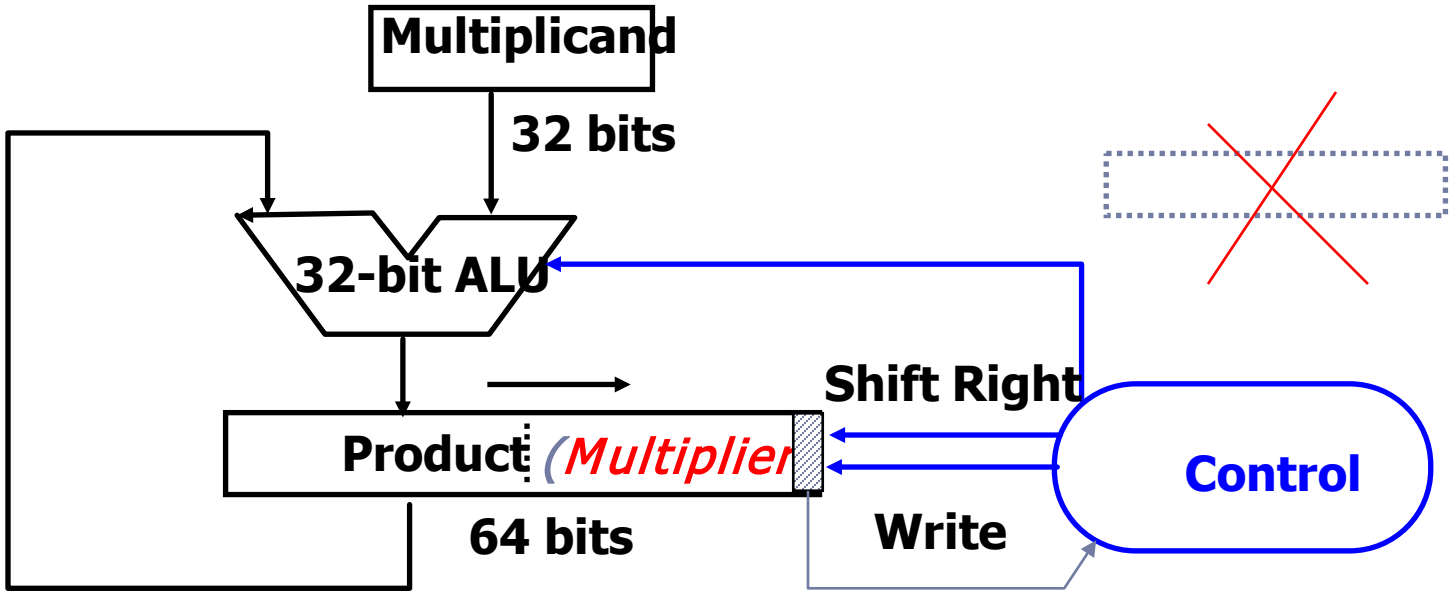


Observations on Multiply Version 2

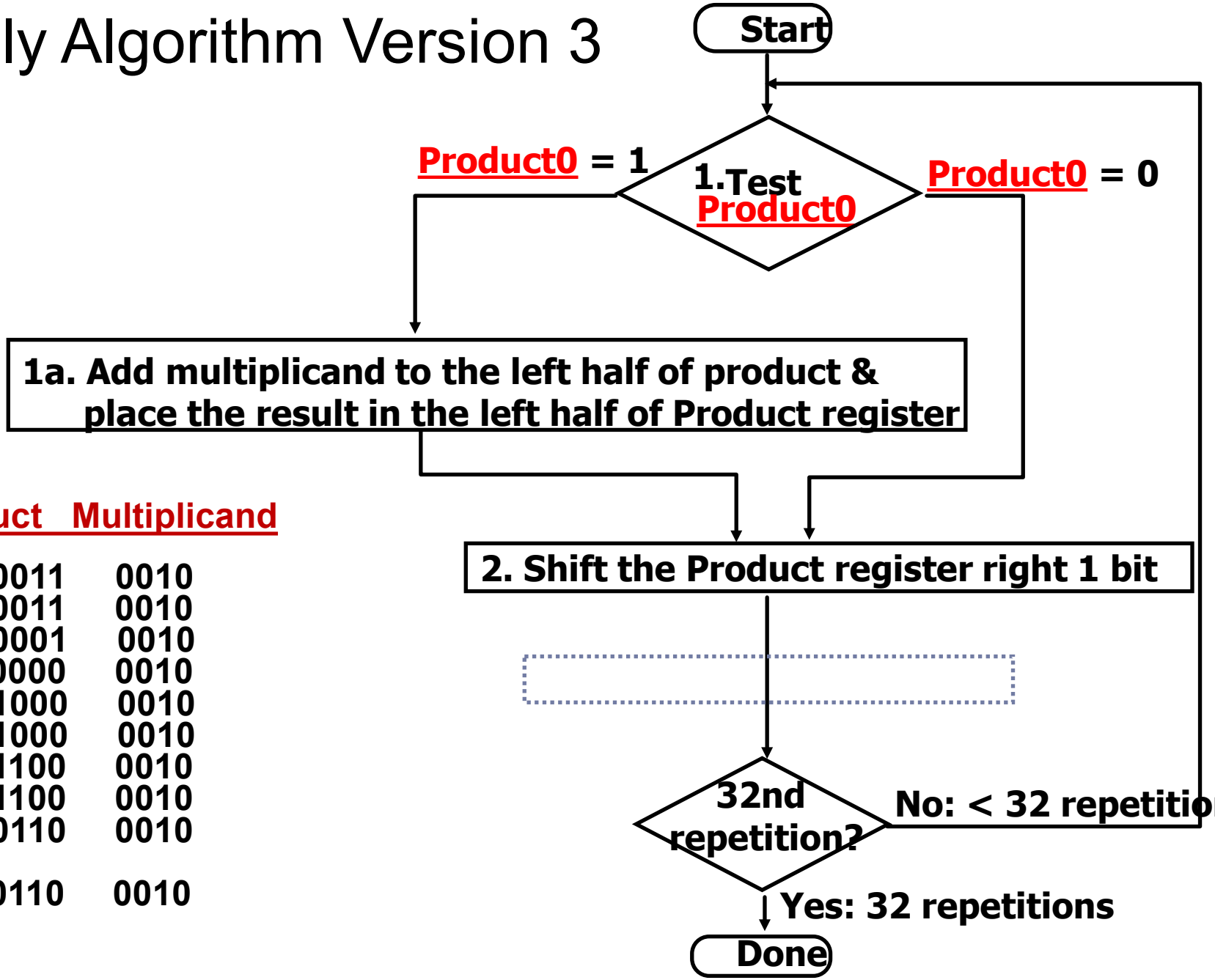
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

MULTIPLY HARDWARE Version 3

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

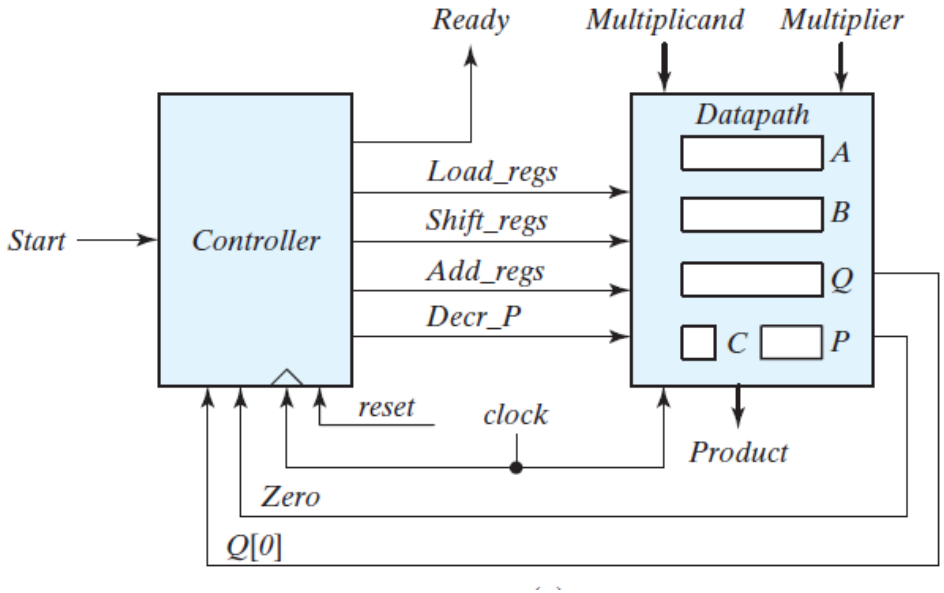


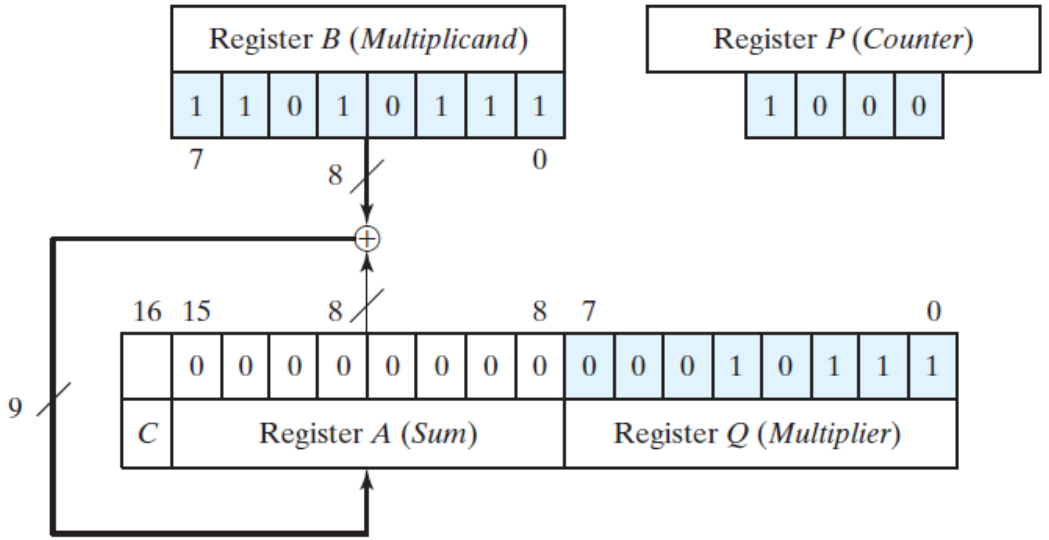
Multiply Algorithm Version 3



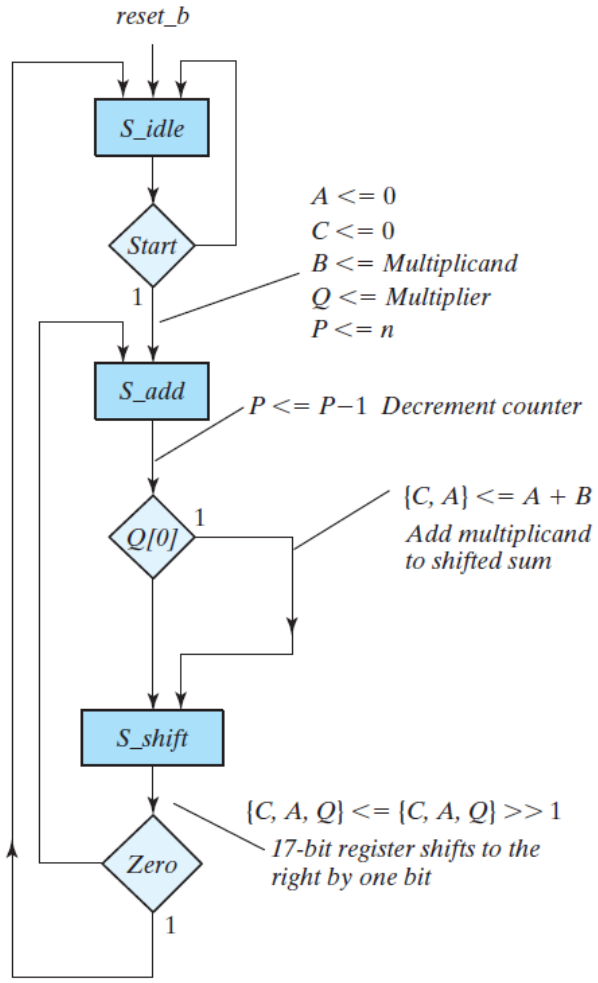
	Product	Multiplicand
	0000 0011	0010
1:	0010 0011	0010
2:	0001 0001	0010
1:	0011 0000	0010
2:	0001 1000	0010
1:	0001 1000	0010
2:	0000 1100	0010
1:	0000 1100	0010
2:	0000 0110	0010
	0000 0110	0010

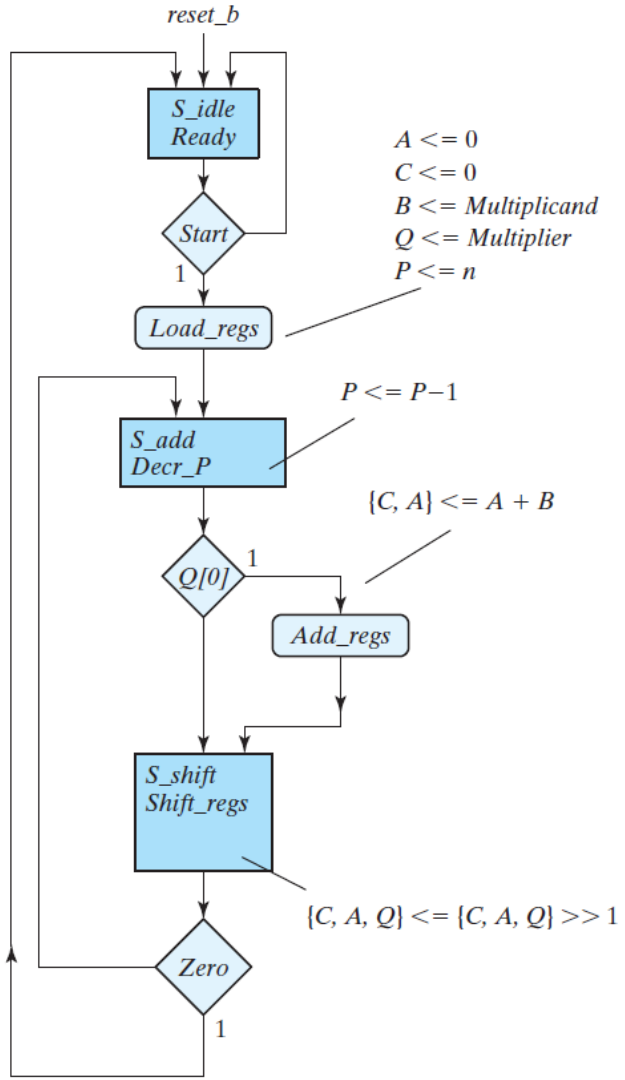
Multiplier- Datapath and Control



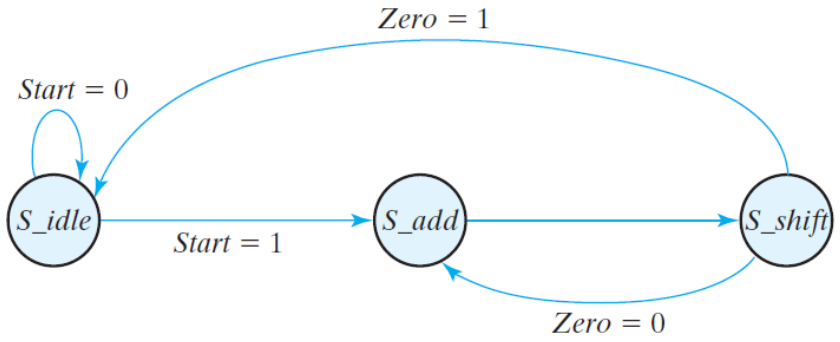


ASMD Chart for Binary Multiplier





Control Specification



(a)

State Transition		Register Operations
From	To	
S_idle		Initial state
S_idle	*S_add*	$A <= 0, C <= 0, P <= dp\_width$
S_add	*S_shift*	$P <= P - 1$ if $(Q[0])$ then $(A <= A + B, C <= C_{out})$
S_shift		shift right $\{CAQ\}, C <= 0$

(b)

Verilog Code

```
module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start, clock, reset_b);
// Default configuration: five-bit datapath
parameter dp_width = 5; // Set to width of datapath
output [2*dp_width -1: 0] Product;
output Ready;
input [dp_width -1: 0] Multiplicand, Multiplier;
input Start, clock, reset_b;

parameter BC_size = 3; // Size of bit counter
parameter S_idle = 3'b001, // one-hot code
          S_add = 3'b010,
          S_shift = 3'b100;
reg [2: 0] state, next_state;
reg [dp_width -1: 0] A, B, Q; // Sized for datapath
reg C;
reg [BC_size -1: 0] P;
reg Load_regs, Decr_P, Add_regs, Shift_regs;
// Miscellaneous combinational logic
assign Product = {A, Q};
wire Zero = (P == 0); // counter is zero // Zero = ~|P; // alternative
wire Ready = (state == S_idle); // controller status
```

Verilog Code – Cont.

```
// control unit
always @ ( posedge clock, negedge reset_b)
if (~reset_b) state <= S_idle; else state <= next_state;

always @ (state, Start, Q[0], Zero) begin
next_state = S_idle;
Load_regs = 0;
Decr_P = 0;
Add_regs = 0;
Shift_regs = 0;
case (state)
        S_idle: begin if (Start) next_state = S_add;
        Load_regs = 1; end

        S_add: begin next_state = S_shift; Decr_P
        = 1; if ( Q[0]) Add_regs = 1; end

        S_shift: begin Shift_regs = 1;
        if (Zero) next_state = S_idle;
        else next_state = S_add; end

        default : next_state = S_idle;
endcase
end
```

```
// datapath unit
always @ ( posedge clock) begin
if (Load_regs) begin
                P <= dp_width;
                A <= 0;
                C <= 0;
                B <= Multiplicand;
                Q <= Multiplier;
end
if (Add_regs) {C, A} <= A + B;
if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
if (Decr_P) P <= P -1;
end
endmodule
```