C335
Computer Structures

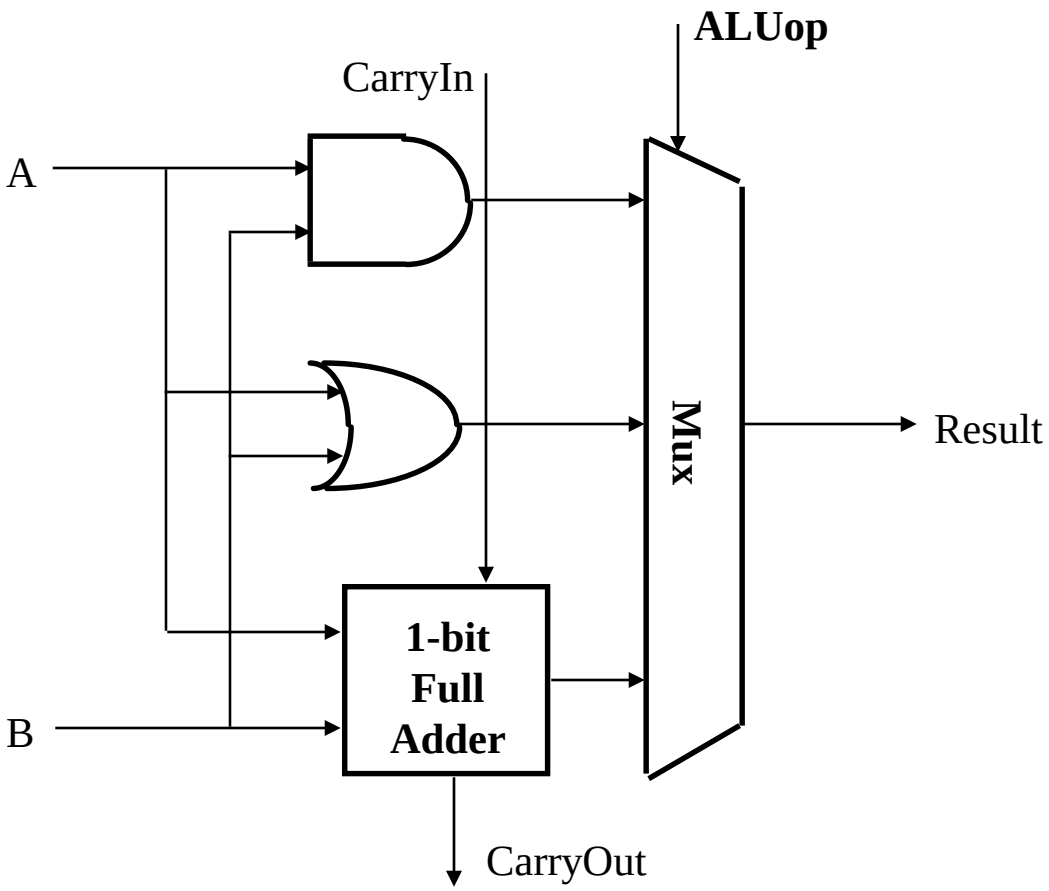
ALU Design (II)

Dr. Liqiang Zhang

Department of Computer and Information Sciences

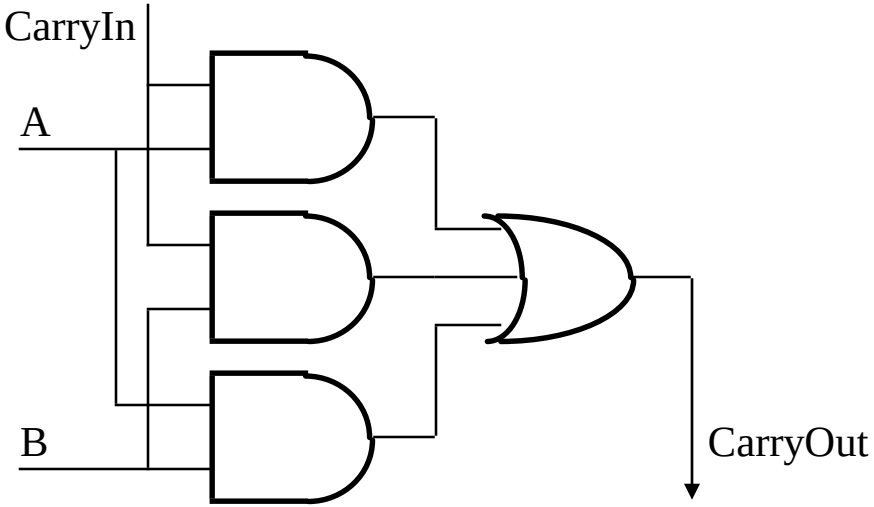
Review: A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD

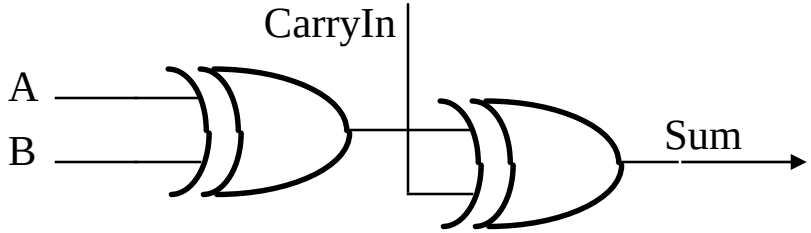


Review: Logic Diagrams for CarryOut and Sum

- CarryOut = B • CarryIn + A • CarryIn + A • B

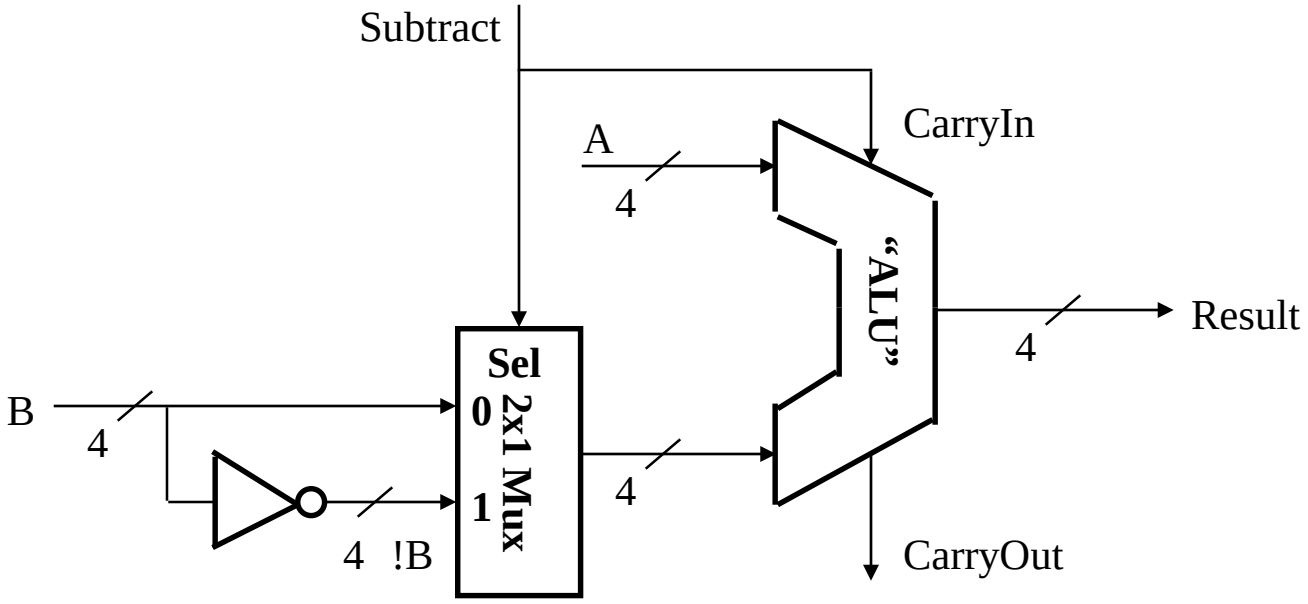


- Sum = A XOR B XOR CarryIn



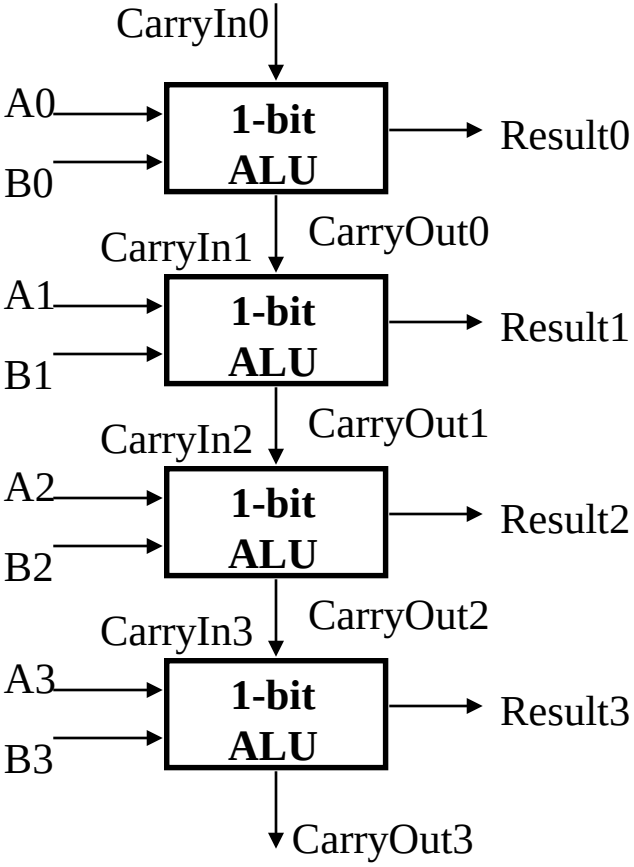
Review: How About Subtraction?

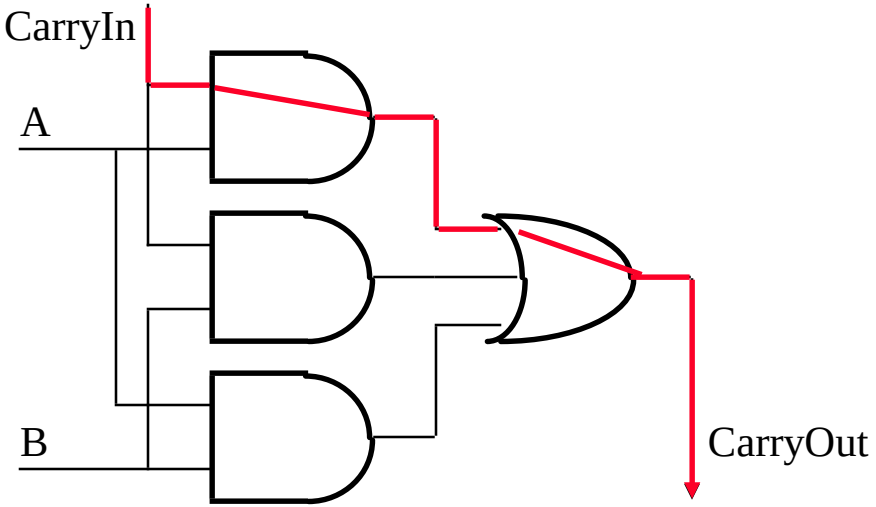
- Keep in mind the followings:
  - (A - B) is the that as: A + (-B)
  - 2's Complement: Take the inverse of every bit and add 1
- Bit-wise inverse of B is B':
  - A + B' + 1 = A + (B' + 1) = A + (-B) = A - B



Review: the Disadvantage of Ripple Carry

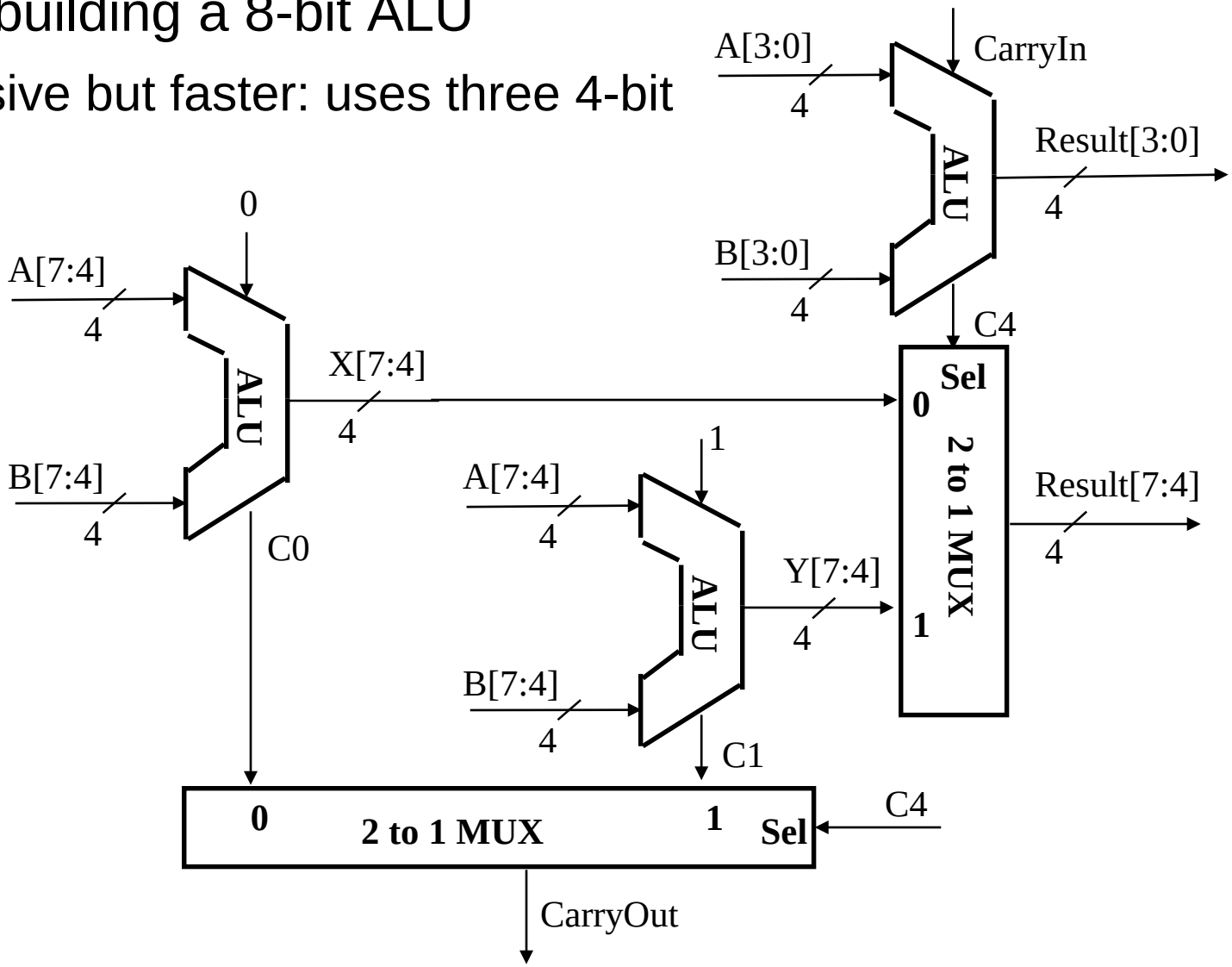
- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for a N-bit adder: 2N-gate delay



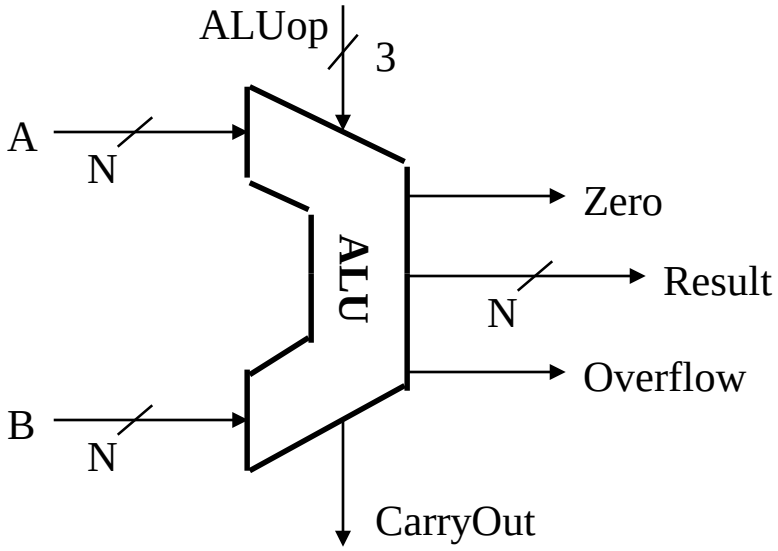


Review: Carry Select Header

- Consider building a 8-bit ALU
  - Expensive but faster: uses three 4-bit ALUs



Review: Functional Specification of the ALU



- ALU Control Lines (ALUop) — Function
  - 000 — And
  - 001 — Or
  - 010 — Add
  - 110 — Subtract
  - 111 — Set-on-less-than

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 the frequency of occurrence and hardware budget
- MIPS ISA

MIPS ALU requirements

- Add, Sub, AddI
  => 2's complement adder with overflow detection & inverter
- SLT, SLTI (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

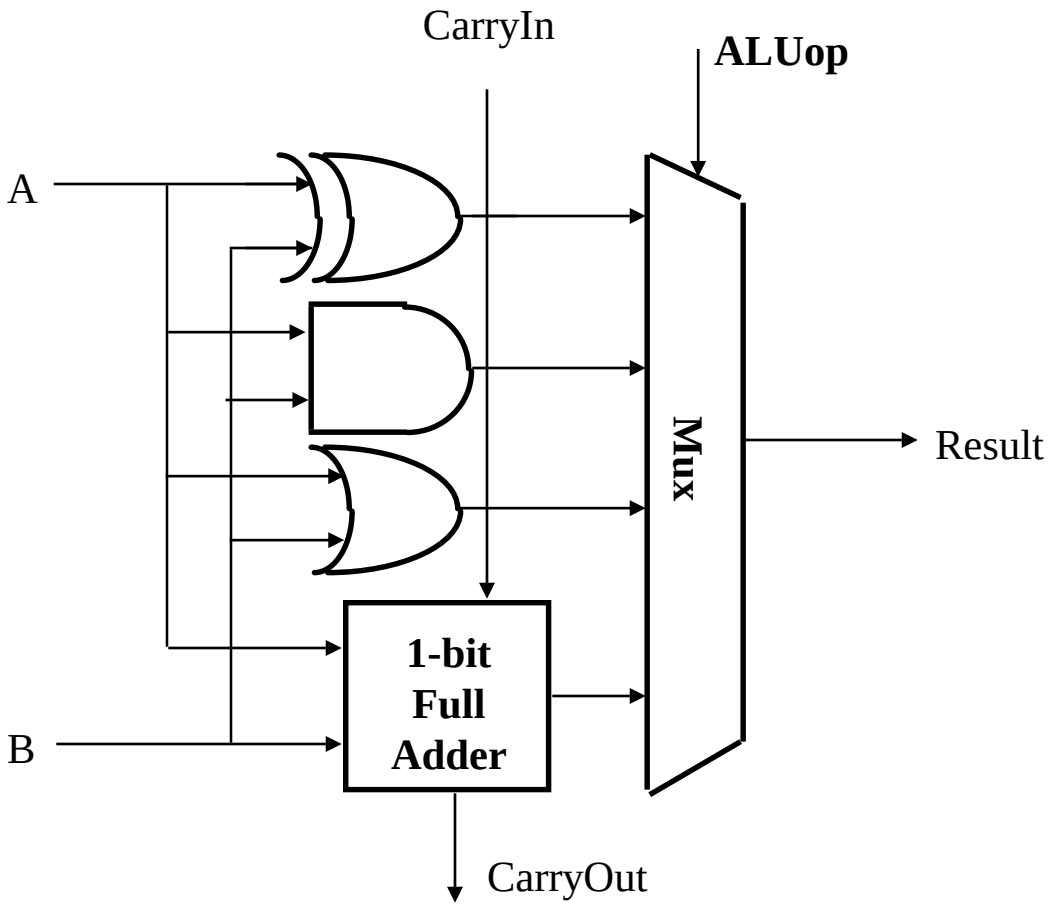
- Xor, Nor
  => Logical XOR, logical NOR or use 2 steps: (A OR B) XOR 1111....1111

- Sll, Srl, Sra
  => Need left shift, right shift, right shift arithmetic by 0 to 31 bits

- Mult, Div
  => Need 32-bit multiply and divide

Add XOR to ALU

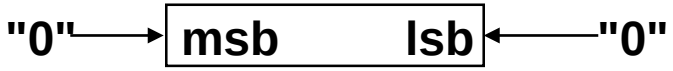
- Expand Multiplexer



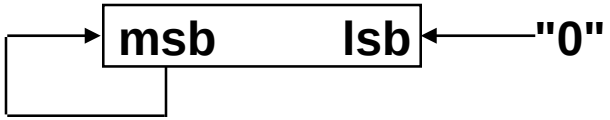
Shifters

Three different kinds:

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



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



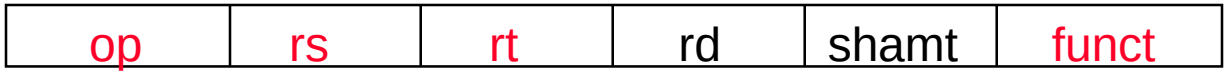
***rotating*-- shifted out bits are wrapped around (not in MIPS)**

Note: these are single bit shifts. A given instruction might request 0 to 31 bits to be shifted!

MIPS Multiply Instruction

- Multiply produces a 64-bit product

```
mult   $s0, $s1        # hi||lo = $s0 * $s1
multu   $s0, $s1       # hi||lo = $s0 * $s1
```



  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

- Multiplies are done by fast, dedicated hardware and are much more complex (and slower) than adders

MULTIPLY

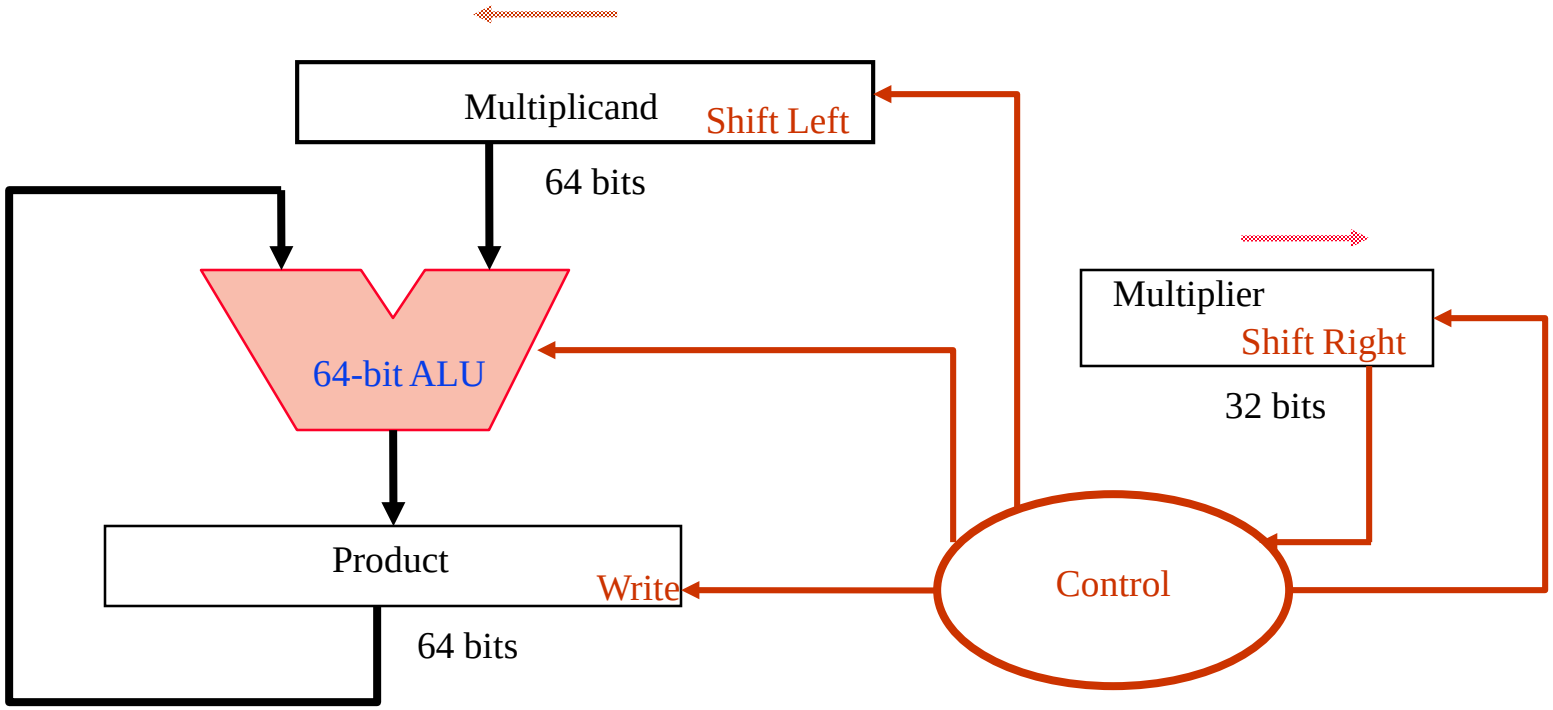
- Paper and pencil example:

```
Multiplicand        1000
 Multiplier       x  1001
                    1000
                   0000
                  0000
                 1000
 Product         1001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy of multiplicand (1 x multiplicand)
- 3 versions of multiply hardware & algorithm: successive refinement.
- Suppose, for now, multiply only positive numbers!

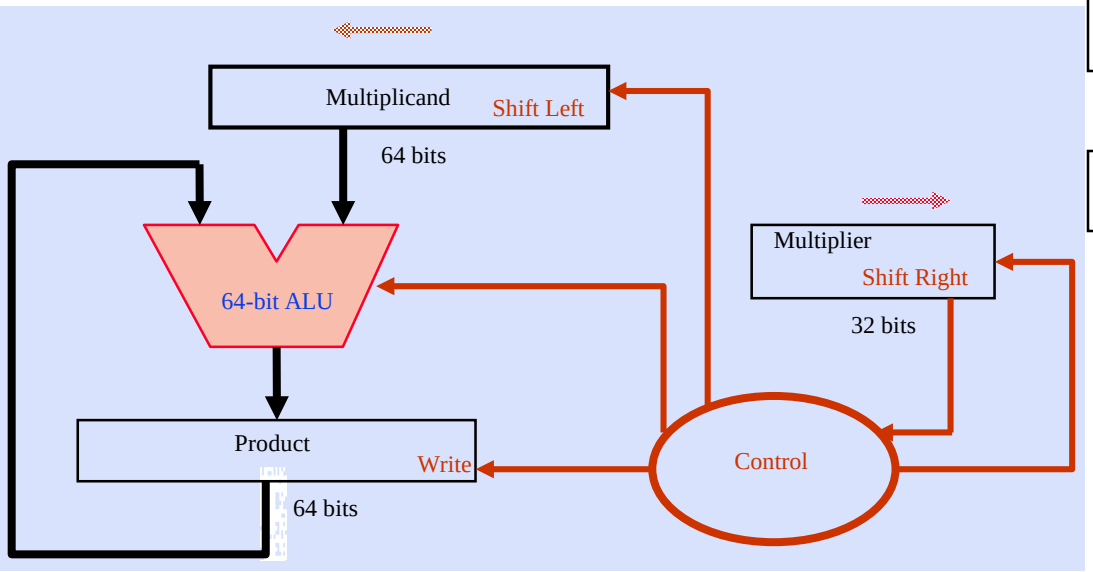
MULTIPLY HARDWARE Version 1

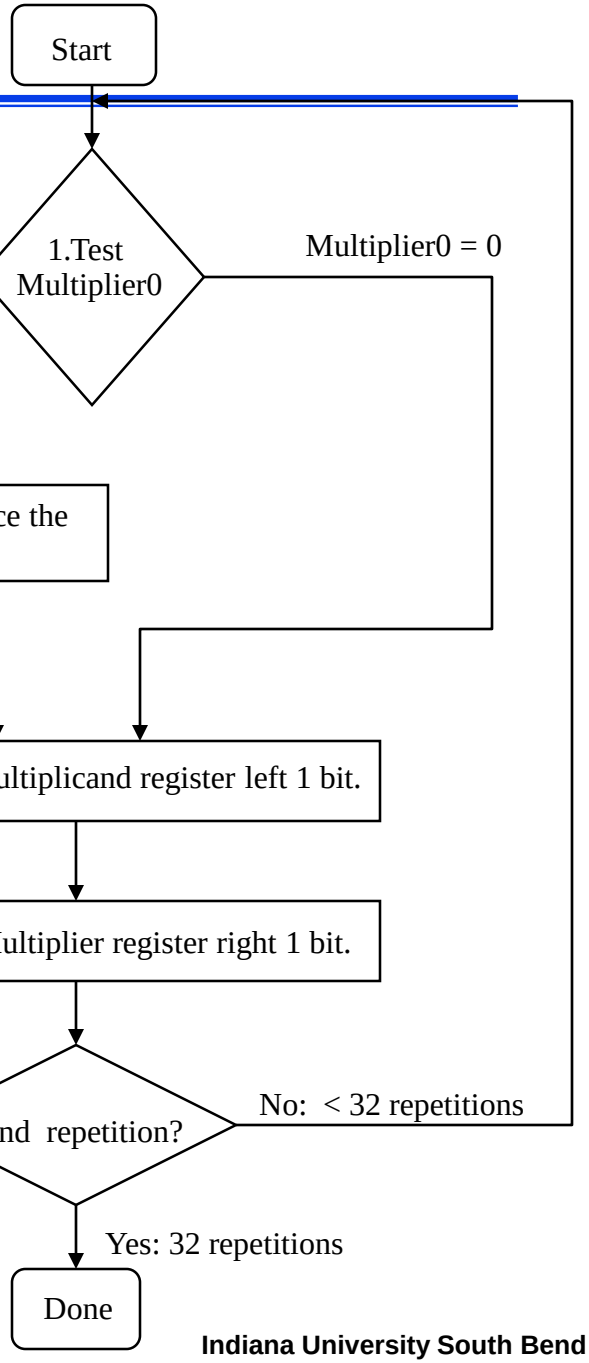
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiply Algorithm Version 1

Multiplier	Multiplicand	Product
0011	0000 0010	0000 0000



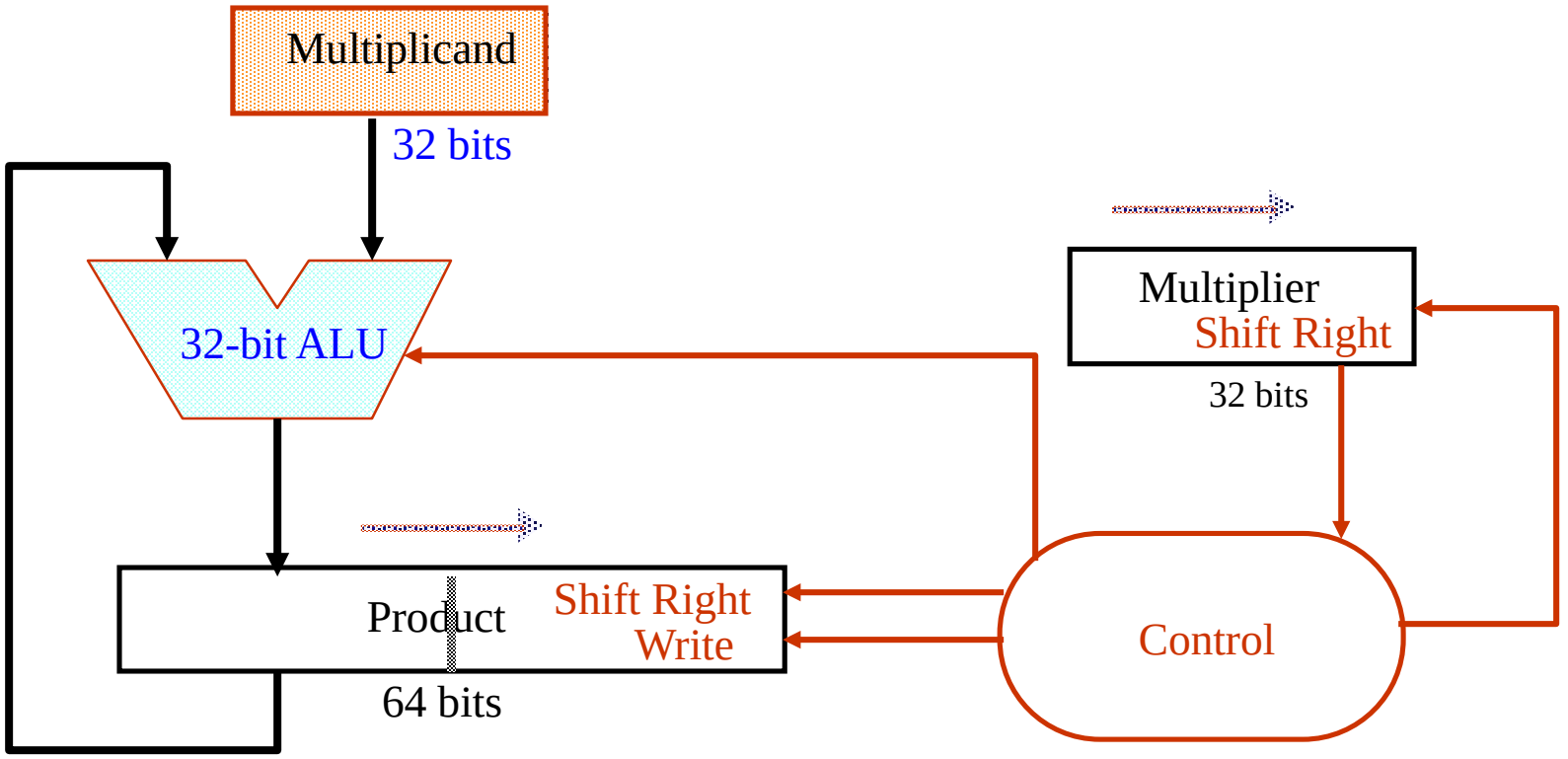


Observations on Multiply Version 1

- 1 step per clock cycle => ≈100 clocks per multiply
  - Ratio of multiply to add/sub operations: 1:5 to 1:100
- 1/2 bits in multiplicand always 0
  => the full 64-bit adder is wasteful and slow, since half of the adder bits are adding 0 to the intermediate sum
- 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?
- 32-bit ALU and multiplicand

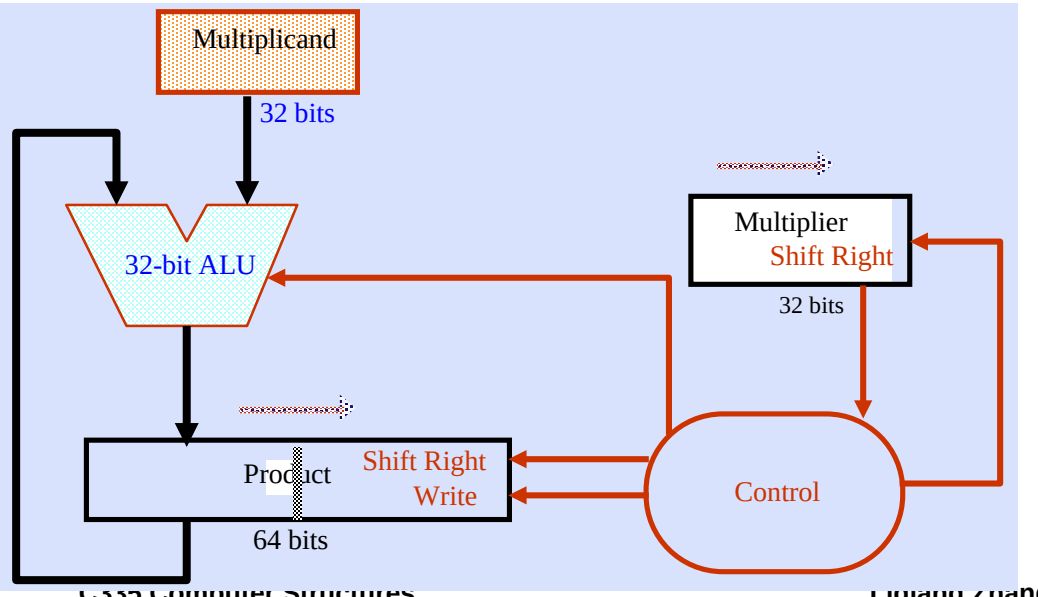
MULTIPLY HARDWARE Version 2

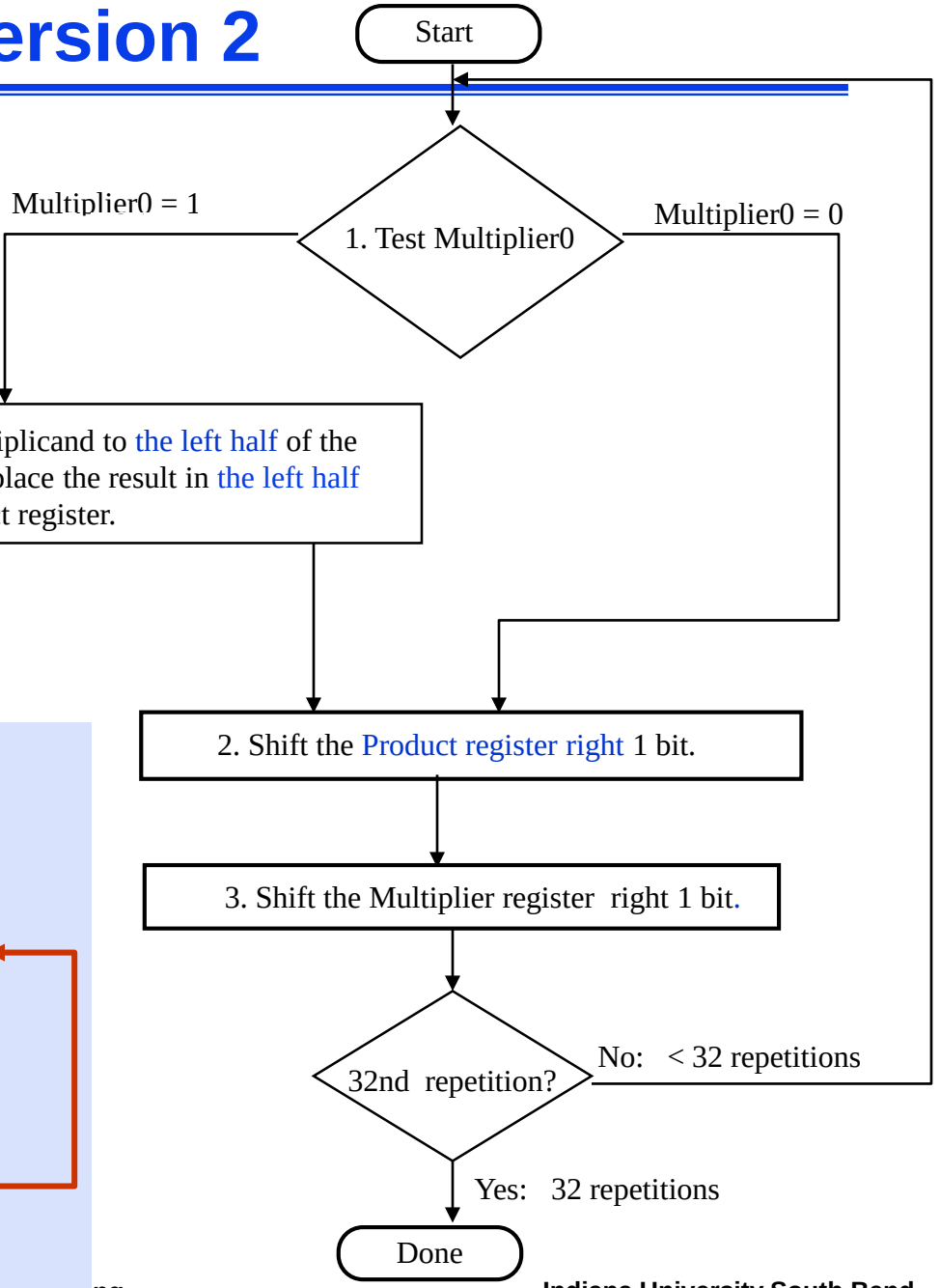
- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg



Multiply Algorithm Version 2

Multiplier	Multiplicand	Product
0011	0010	0000 0000



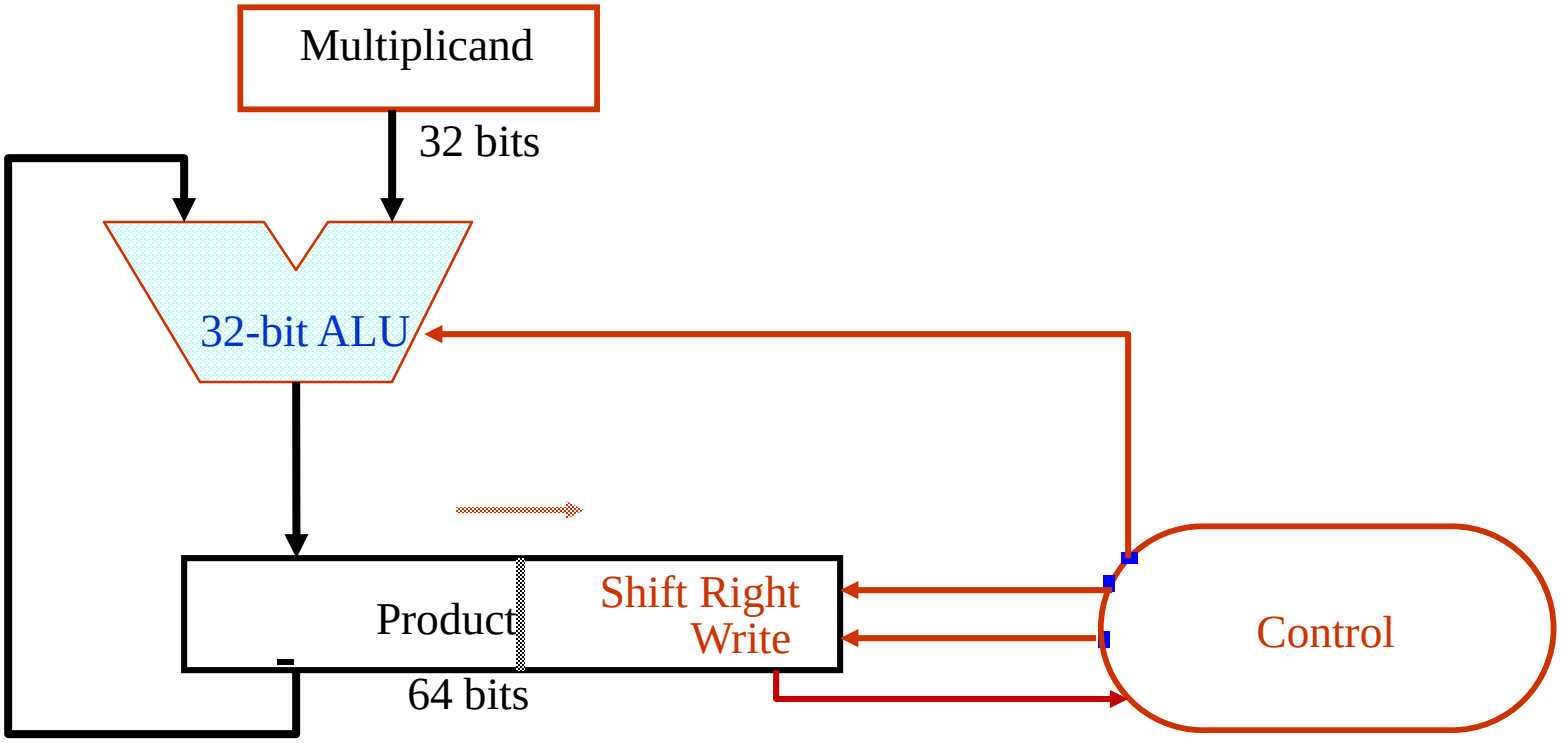


Observations on Multiply Version 2

- Product register wastes space that exactly matches size of multiplier
- As the wasted space in product disappears, so do the bits of the multiplier.
- Thus, combine Multiplier register and Product register

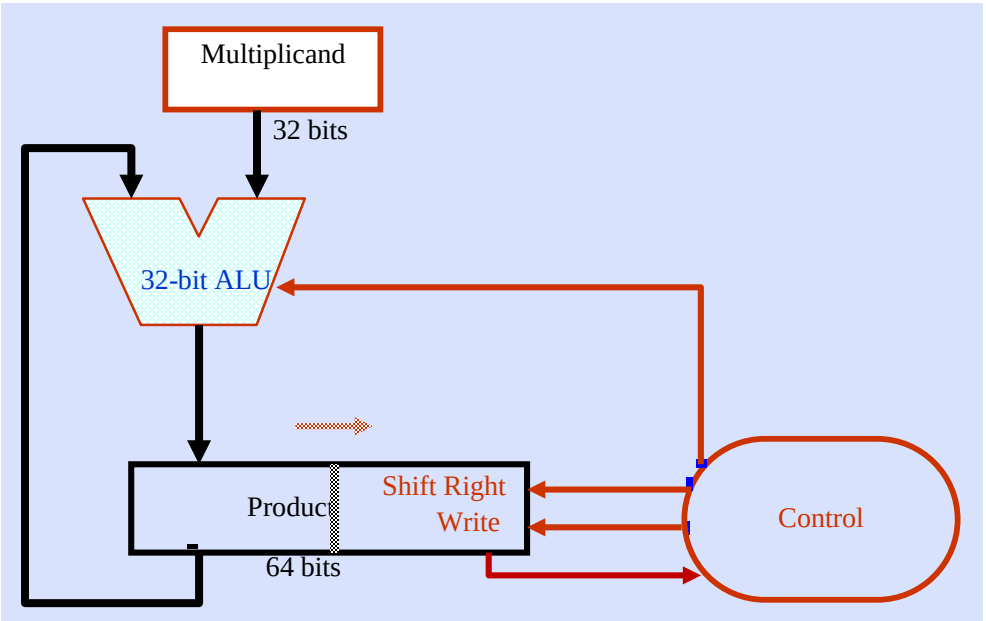
MULTIPLY HARDWARE Version 3

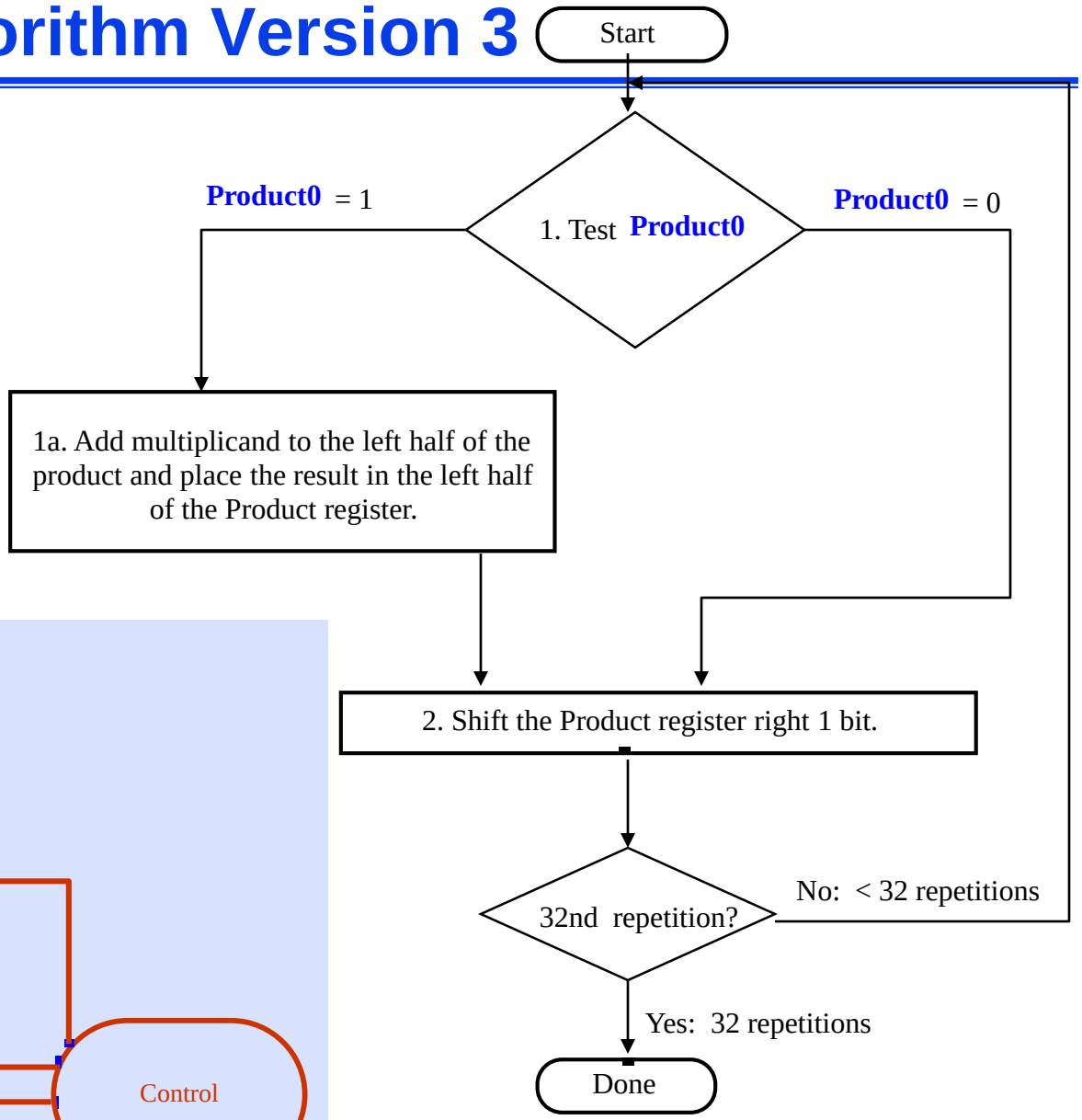
- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



Multiply Algorithm Version 3

Multiplicand Product
0010 0000 0011





Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined, reducing instruction steps considerably!
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction Multu
- 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)
  - Booth's Algorithm is more elegant way to multiply signed numbers using same hardware as before.

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
```

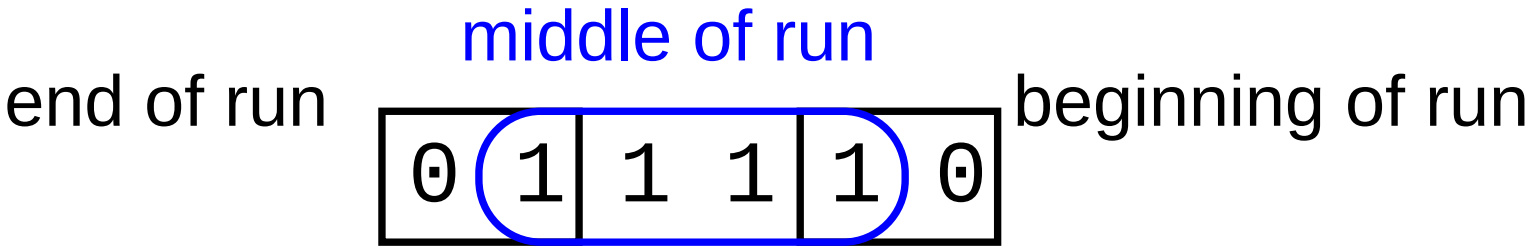
- ALU with add or subtract can get the same result in more than one way:

```
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 this machine

For some bit patterns the algorithm is faster, it handles signed numbers too. In what case is Booth's algorithm not so desirable?

Booth's Algorithm

1. Depending on the current bit and the bit to the right, which was the current bit in the previous step, do one of the following:

   00: a. Middle of a string of 0s, so no arithmetic operation.
   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.

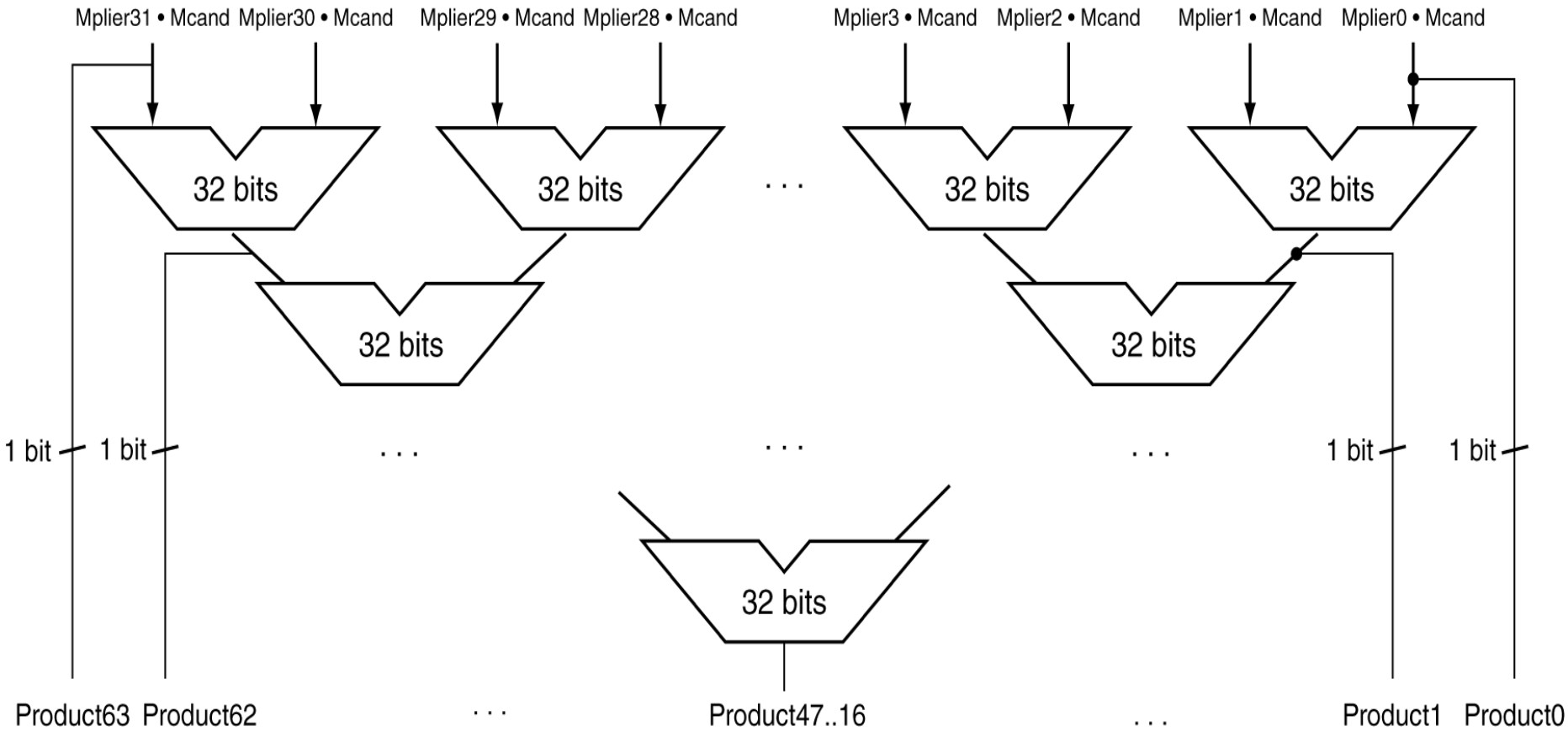
2. As in the previous algorithm, shift the Product register right (arith.) 1 bit.

If you are Interested

- Using booth's algorithm to do the following calculations:
  - Multiplicand Multiplier
  - -2 3
  - 2 -3
  - -2 -3
  - 2 -8
  - -8 3
- Does it work well for all these cases?

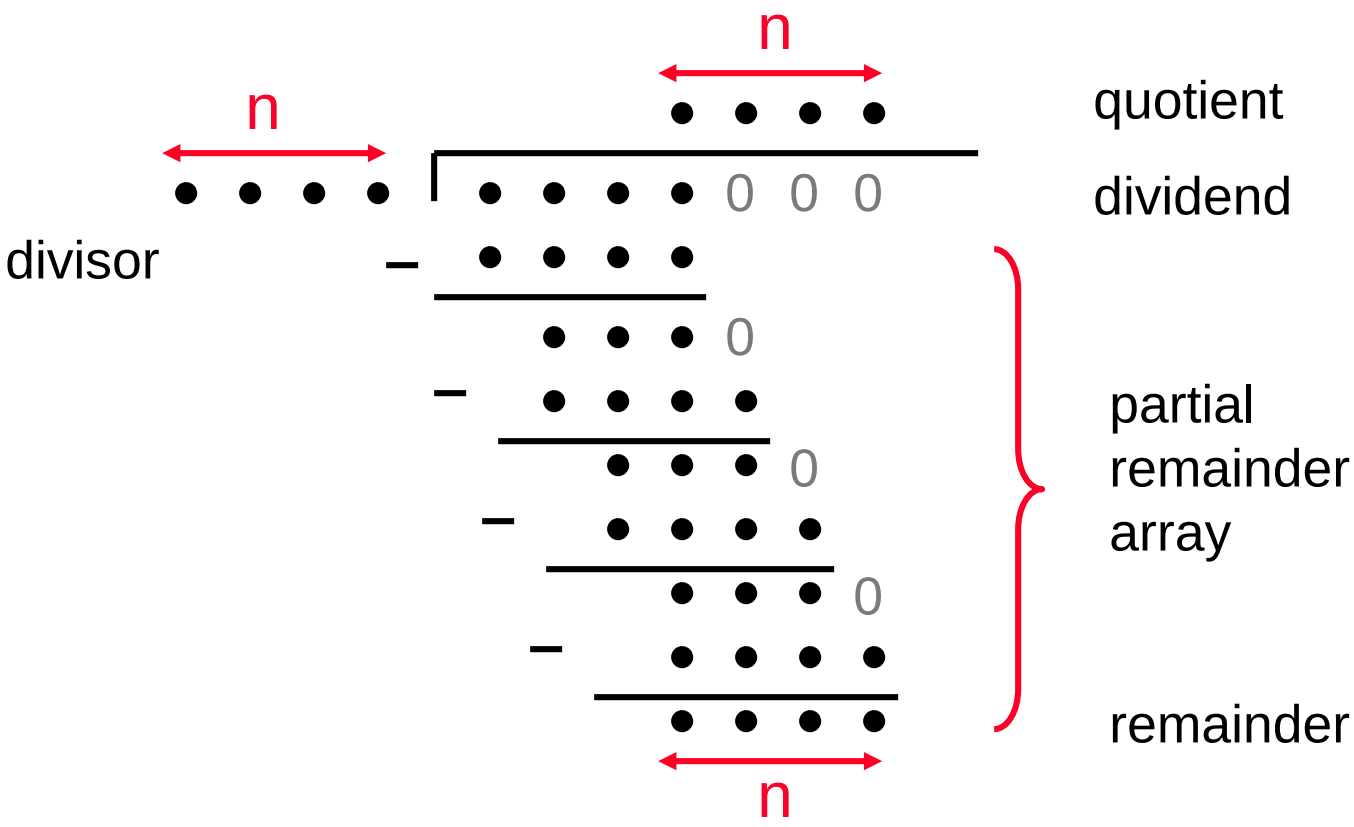
Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



Division

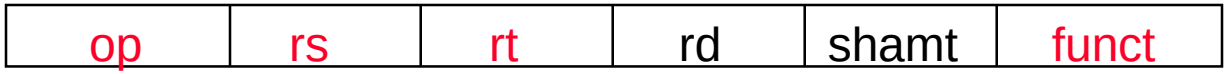
- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts



MIPS Divide Instruction

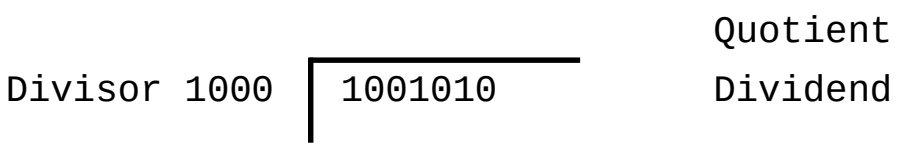
- Divide generates the remainder in `hi` and the quotient in `lo`

```
div  $s0, $s1        # lo = $s0 / $s1
                     # hi = $s0 mod $s1
```



  - Instructions `mflo rd` and `mfhi rd` are provided to move the quotient and remainder to (user accessible) registers in the register file

Divide: Paper & Pencil



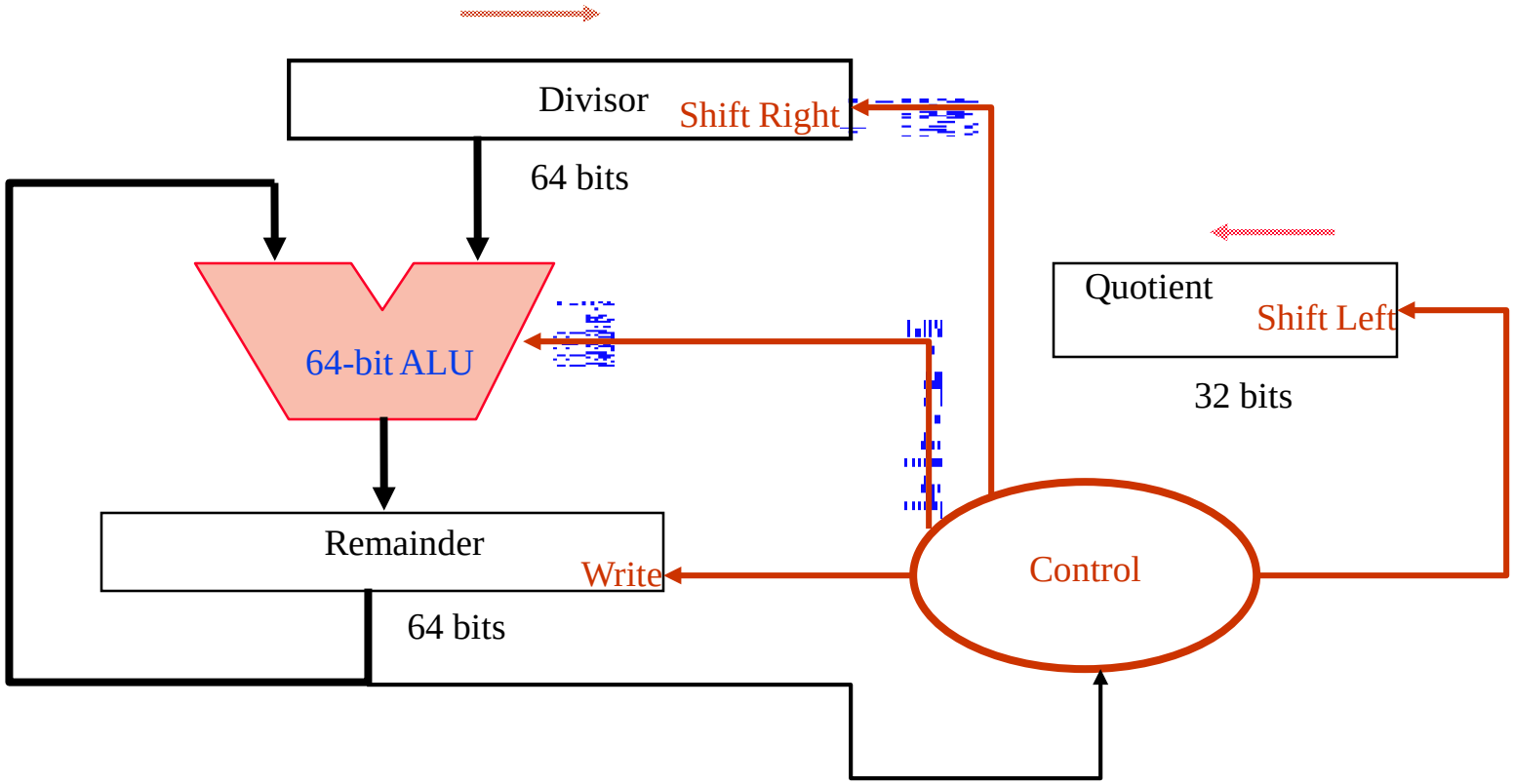
Divide: Paper & Pencil

```
                    1001          Quotient
Divisor 1000 ┌ 1001010             Dividend
            -1000
                 10
                 101
                 1010
                -1000
                   10             Remainder
```

- 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
- 3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

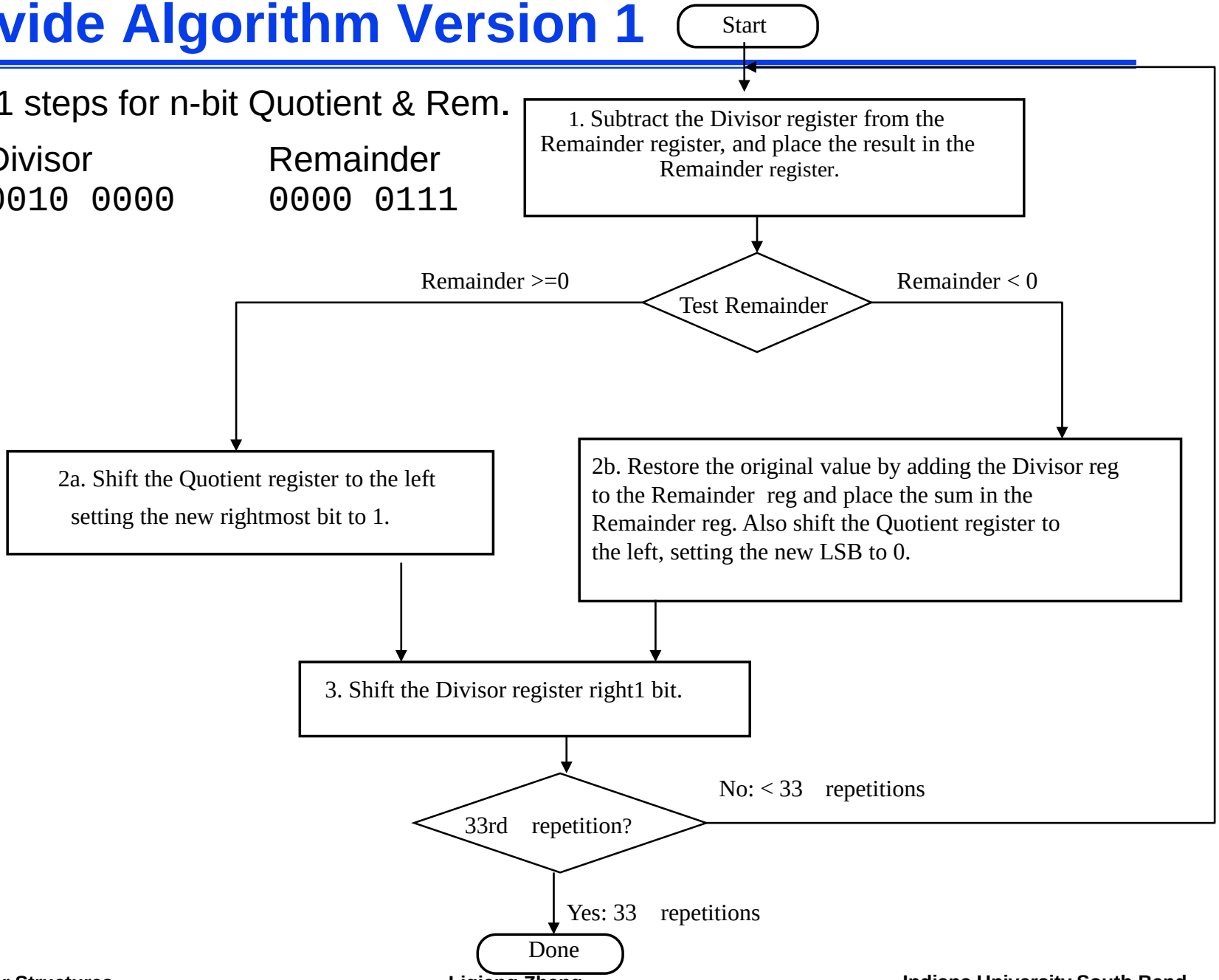
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

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

Quotient	Divisor	Remainder
0000	0010 0000	0000 0111

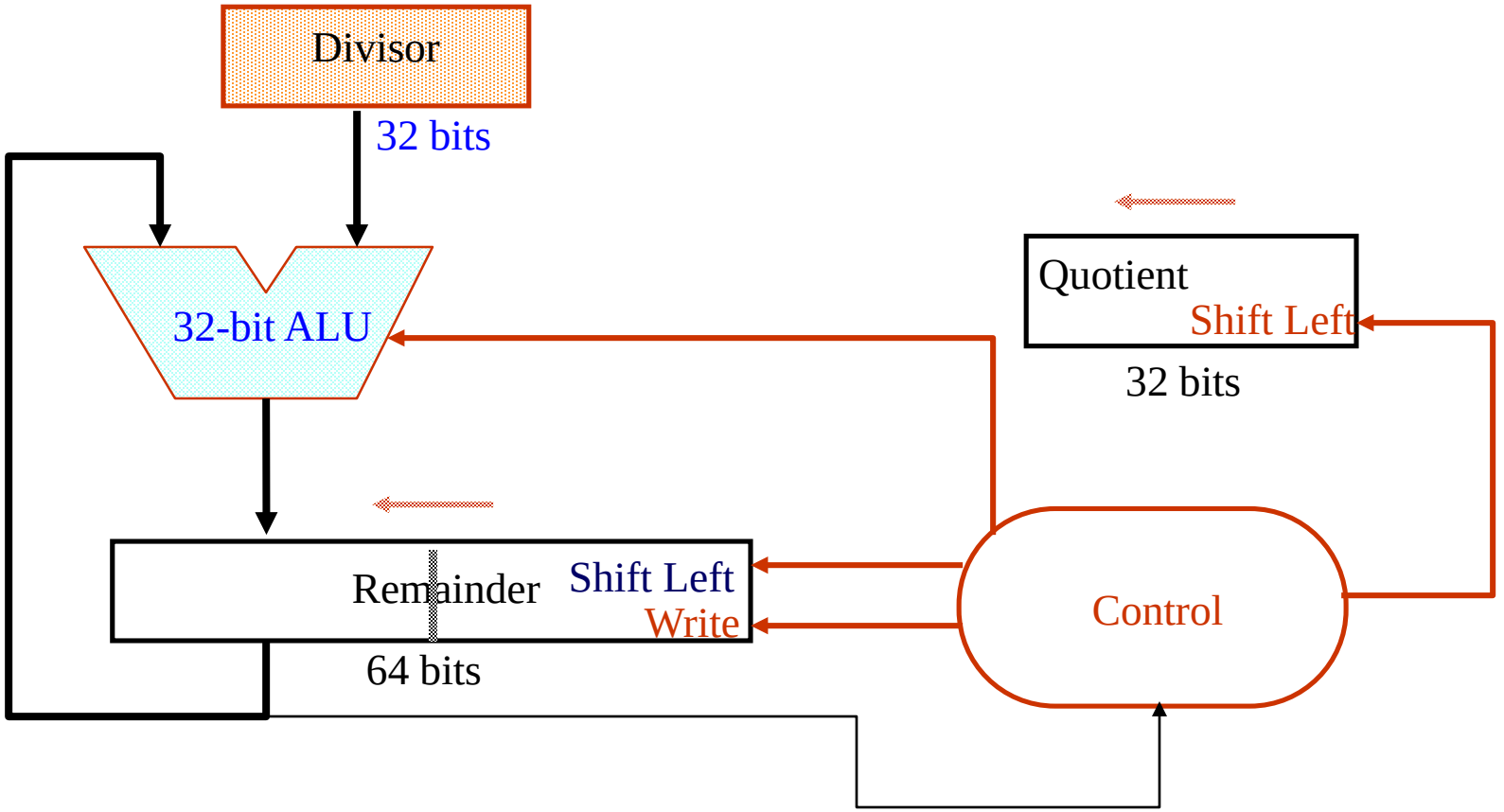


Observations on Divide Version 1

- 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

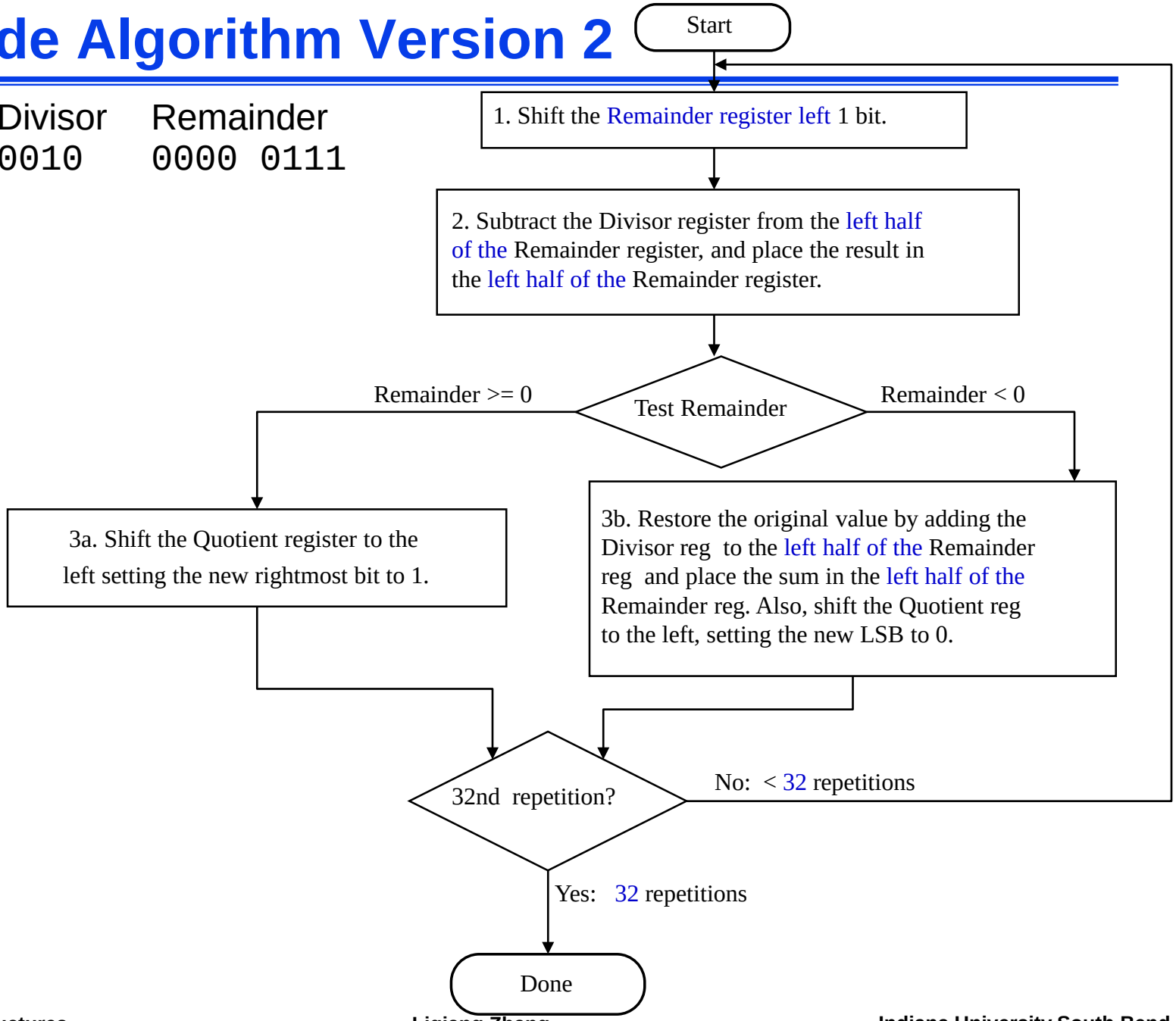
DIVIDE HARDWARE Version 2

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



Divide Algorithm Version 2

Quotient	Divisor	Remainder
0000	0010	0000 0111

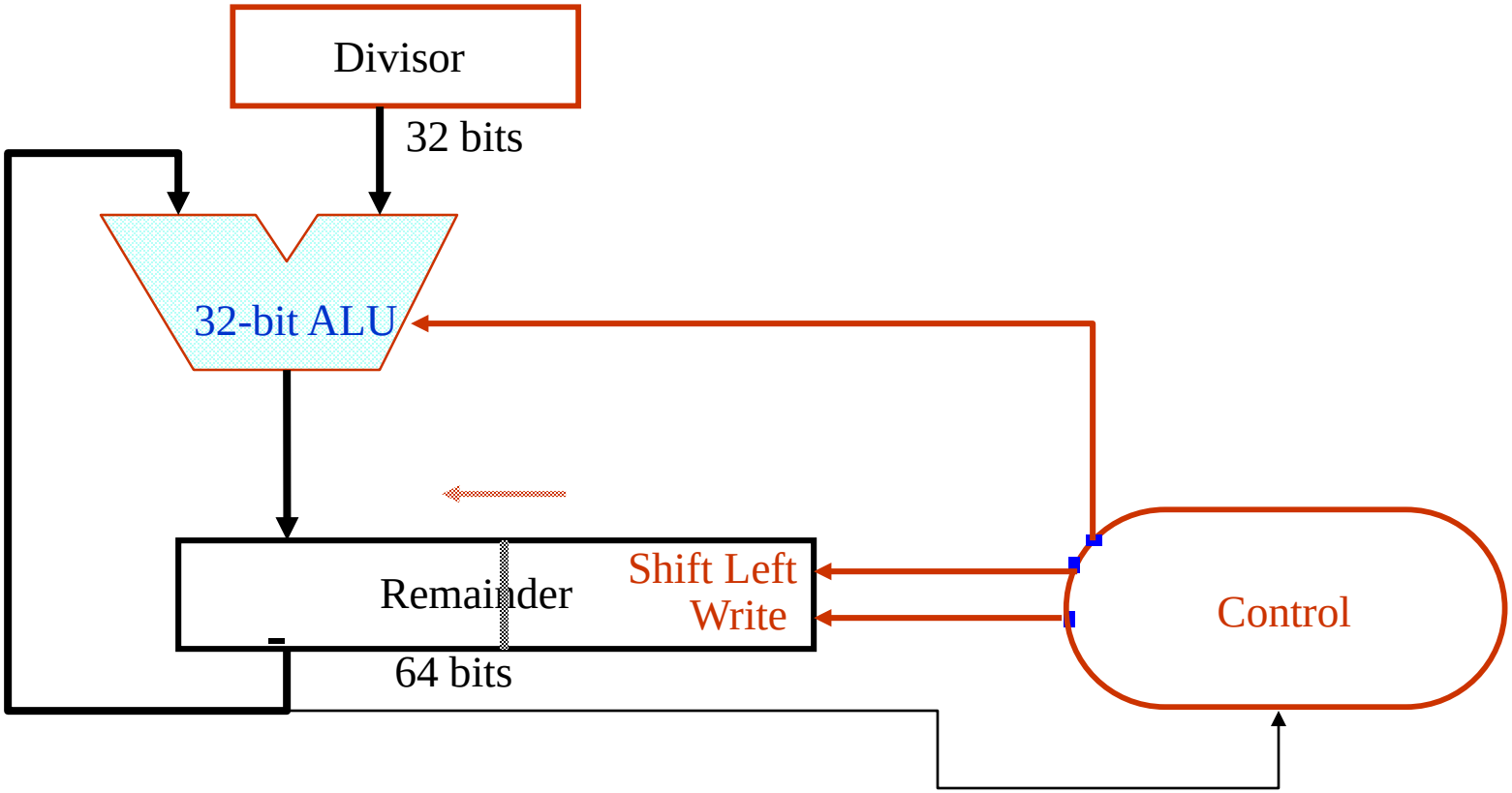


Observations on Divide Version 2

- 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 the final correction step must shift back only the remainder in the left half of the register.

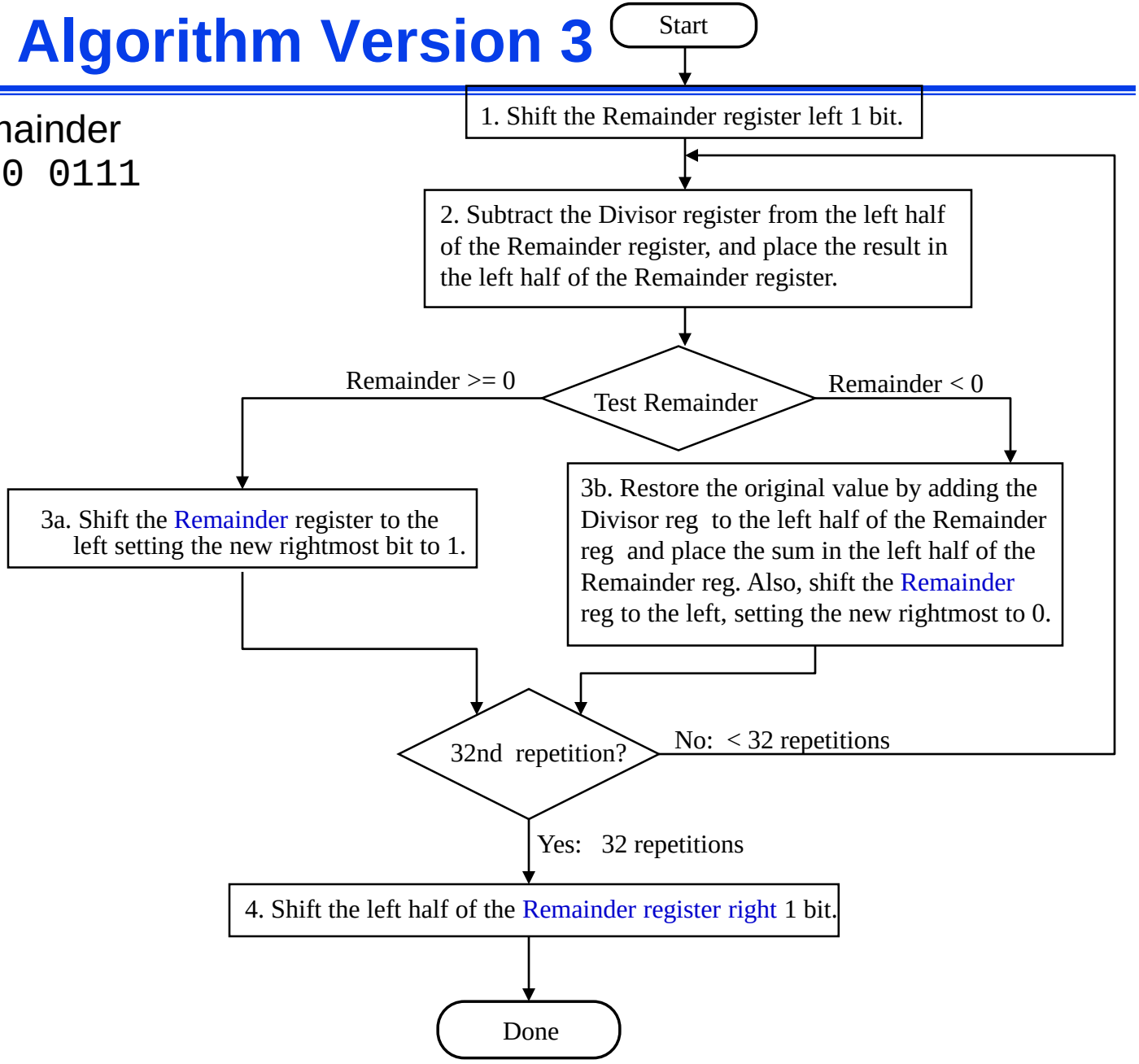
DIVIDE HARDWARE Version 3

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



Divide Algorithm Version 3

Divisor Remainder
0010 0000 0111



Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-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

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
- Divide can use same hardware as multiply: Hi & Lo registers in MIPS.