Lecture 6:

Arithmetic 2/3

Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

Multiply (unsigned)

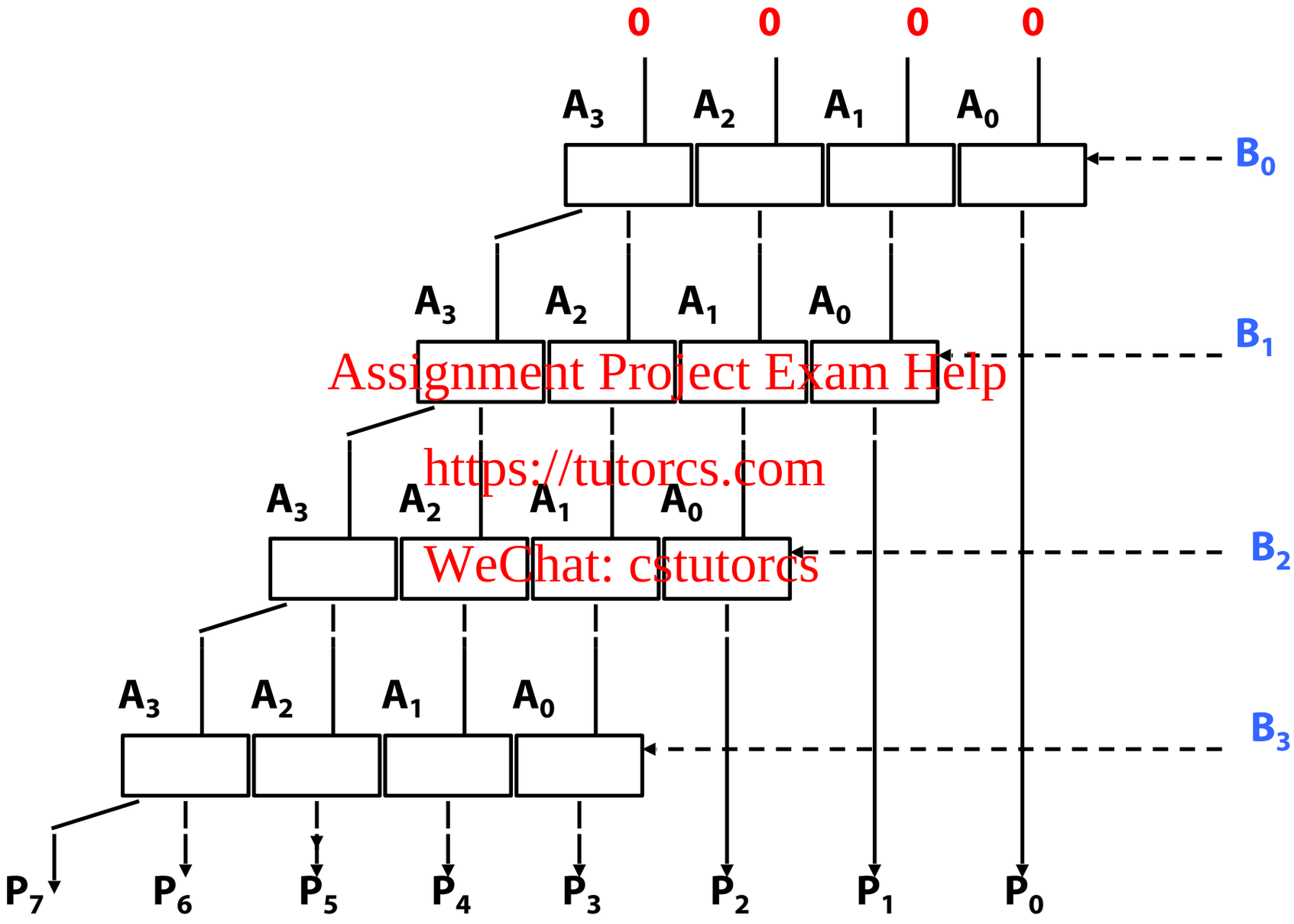
- **Paper and pencil example (unsigned):**

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

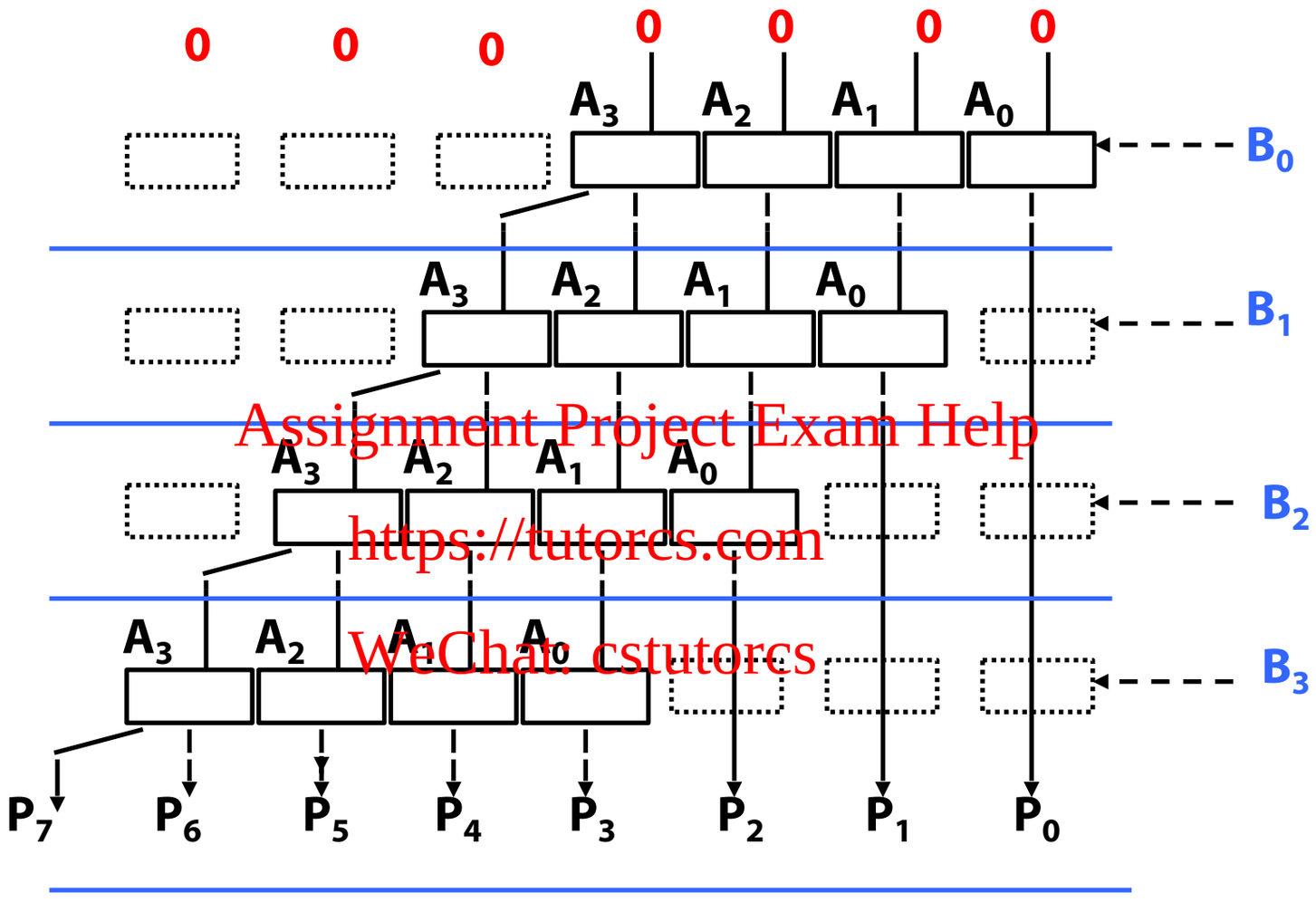

- **m bits x n bits = m+n bit product**

- **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

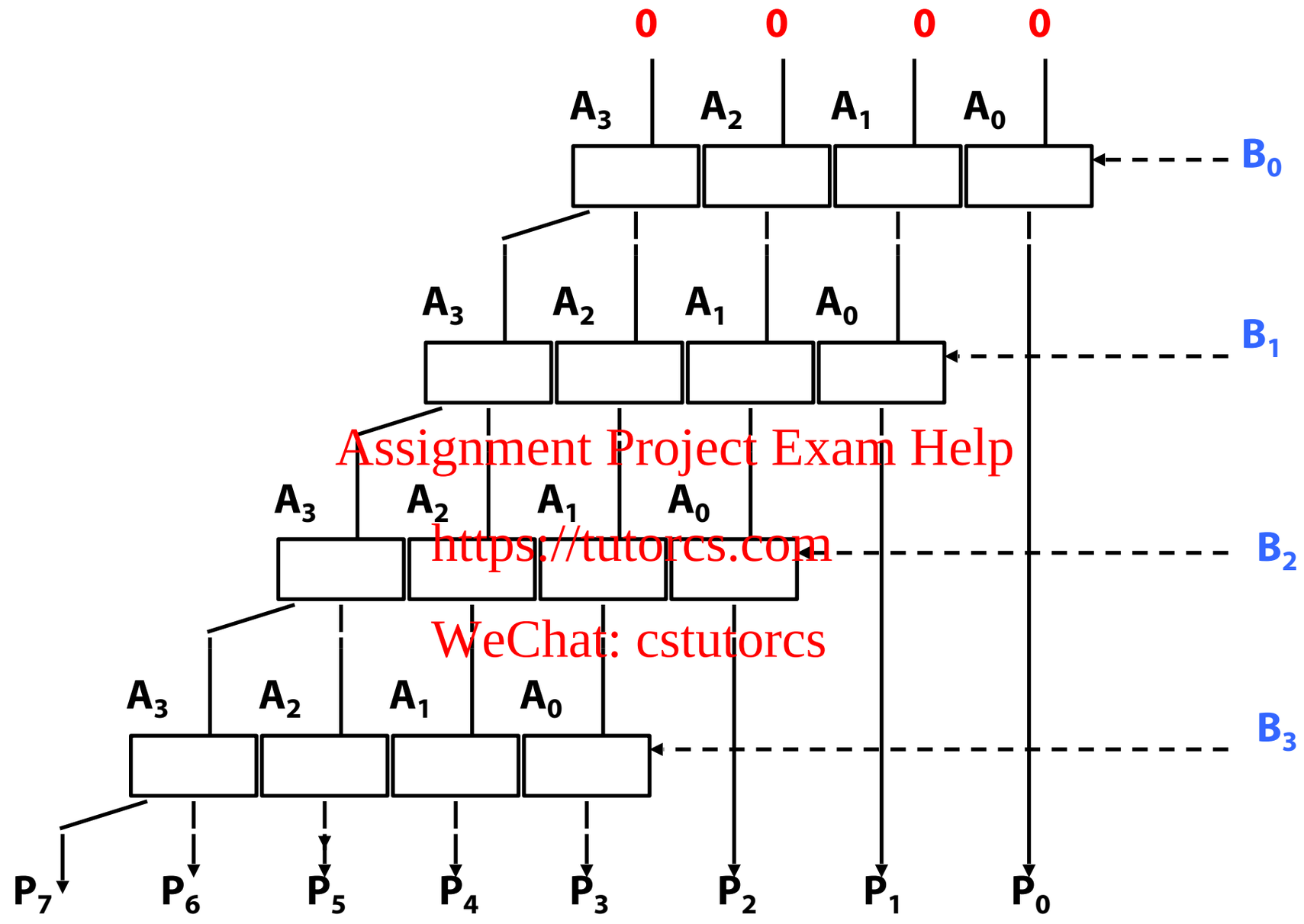


How does it work?



- At each stage shift A left (multiply it by 2)
- Use next bit of B to determine whether to add in shifted multiplicand
- Accumulate 2n bit partial product at each stage

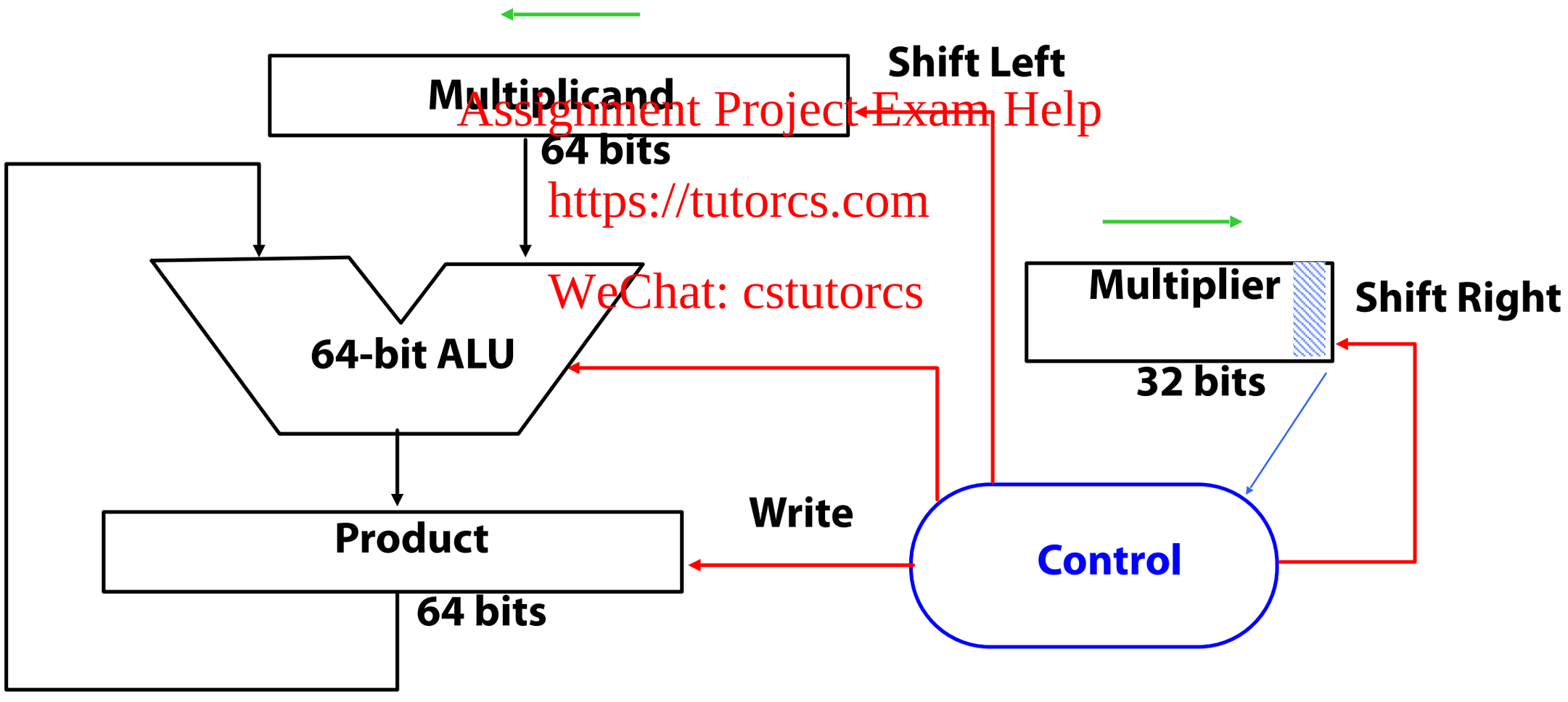
Unsigned Combinational Multiplier



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

Unsigned shift-add multiplier (version 1)

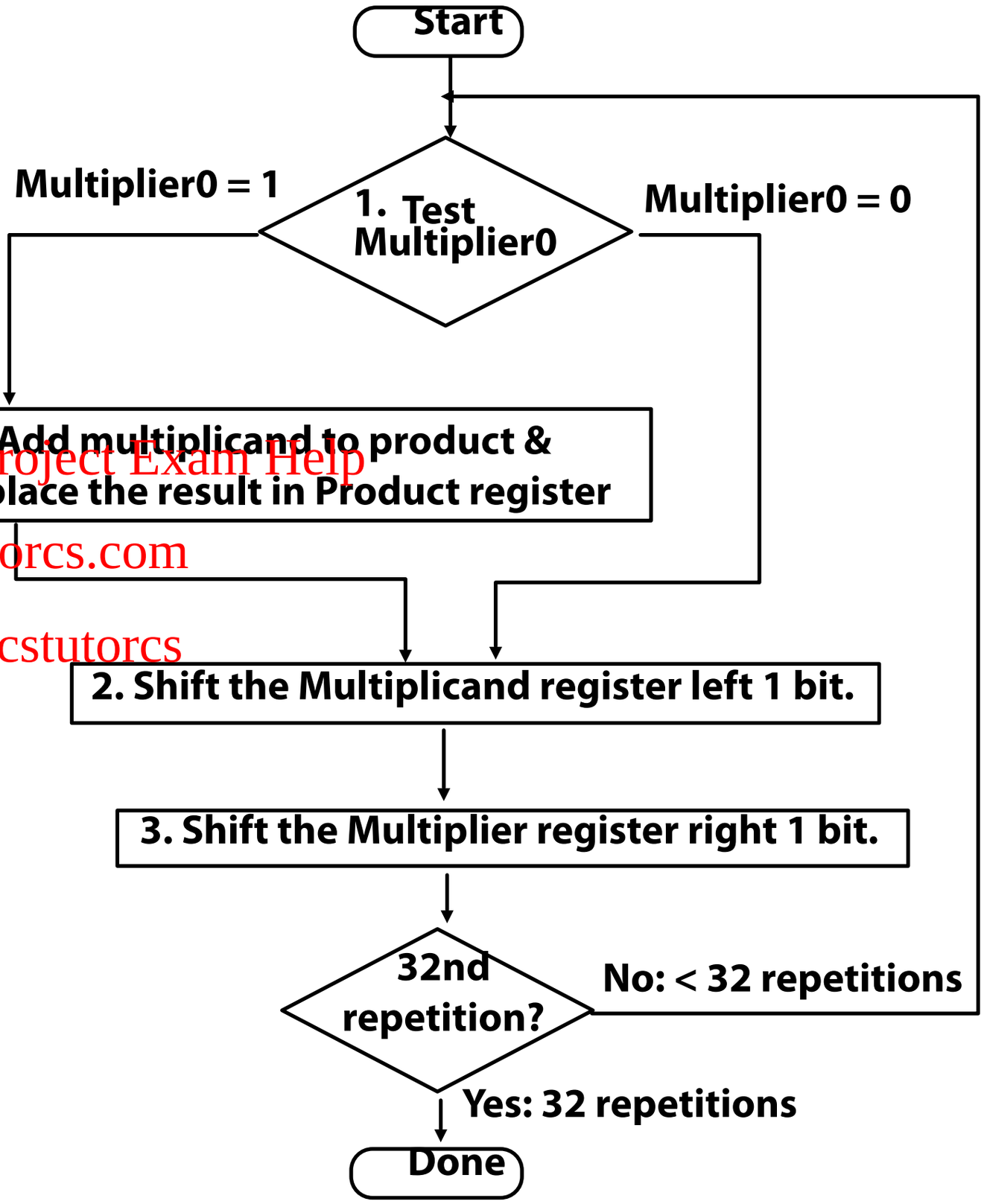
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

Multiply Algorithm V1

	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

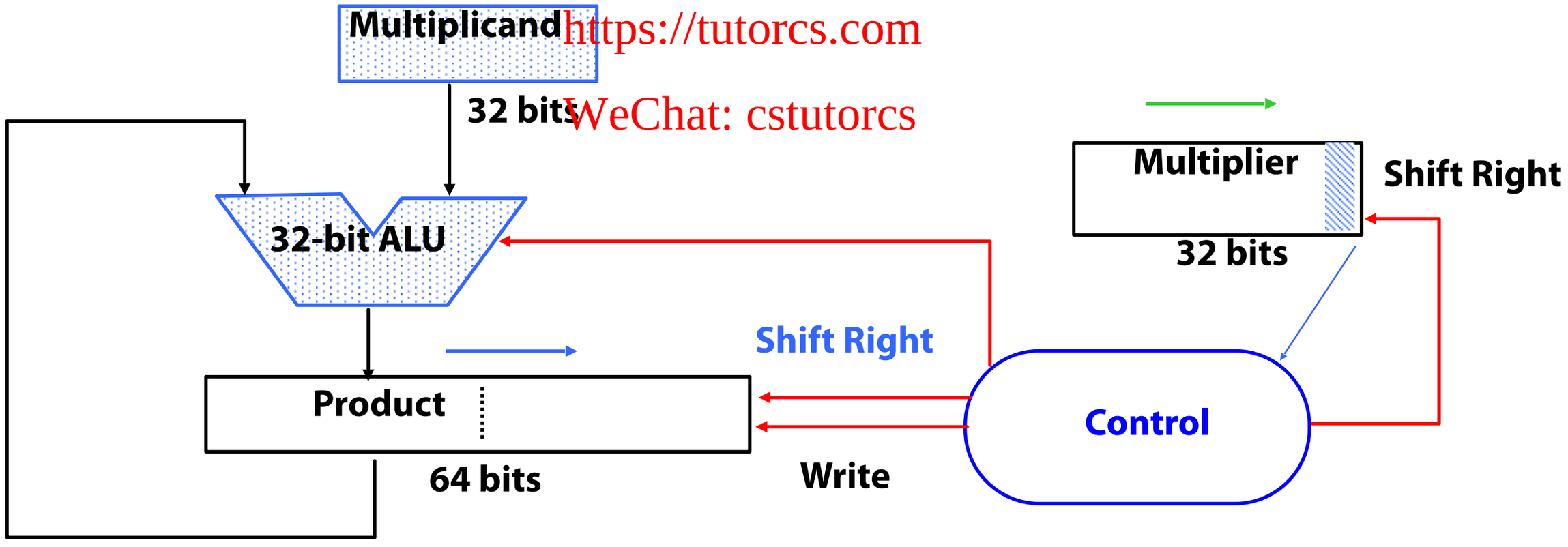


Observations on Multiply Version 1

- 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?*

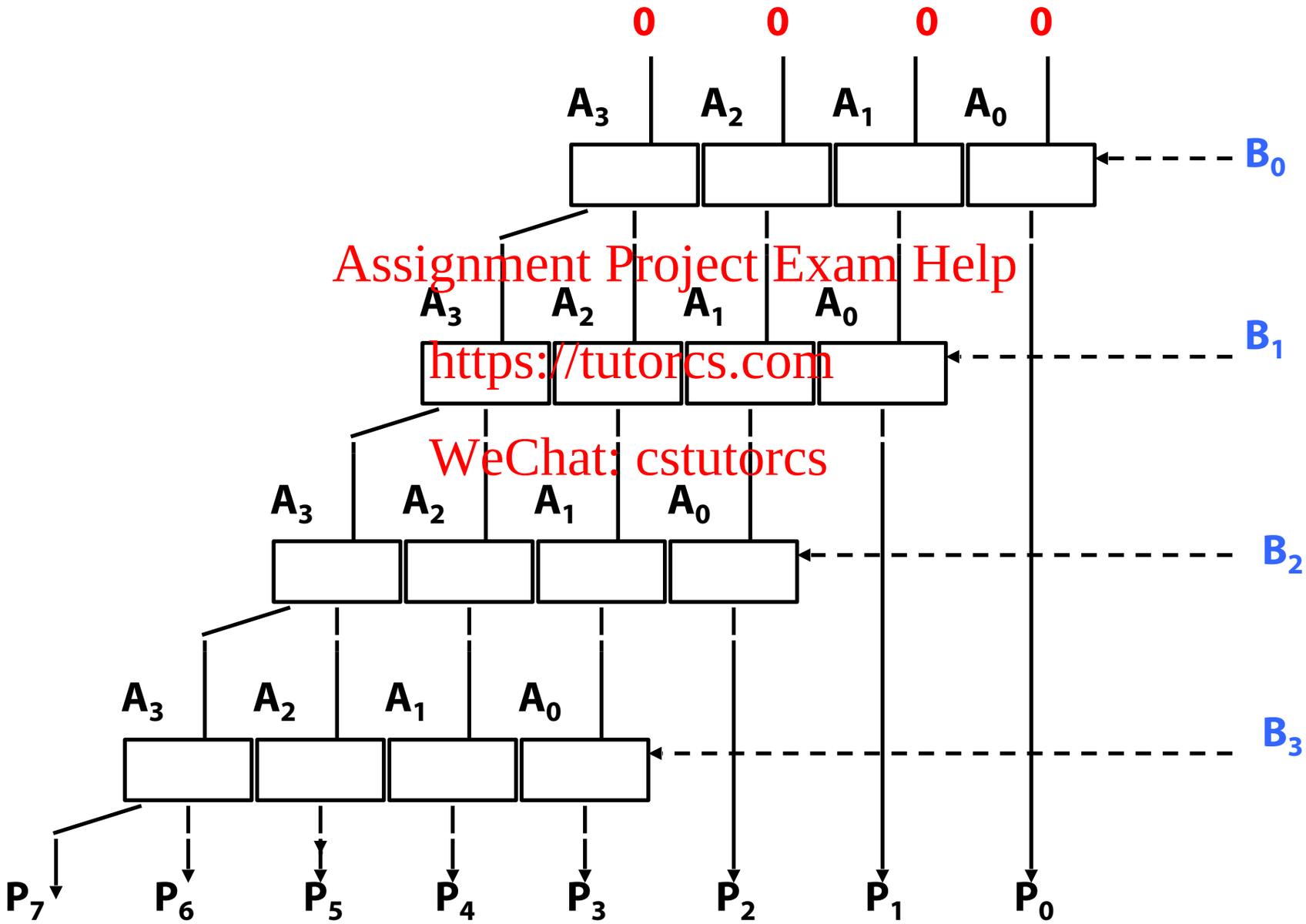
Multiply Hardware Version 2

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

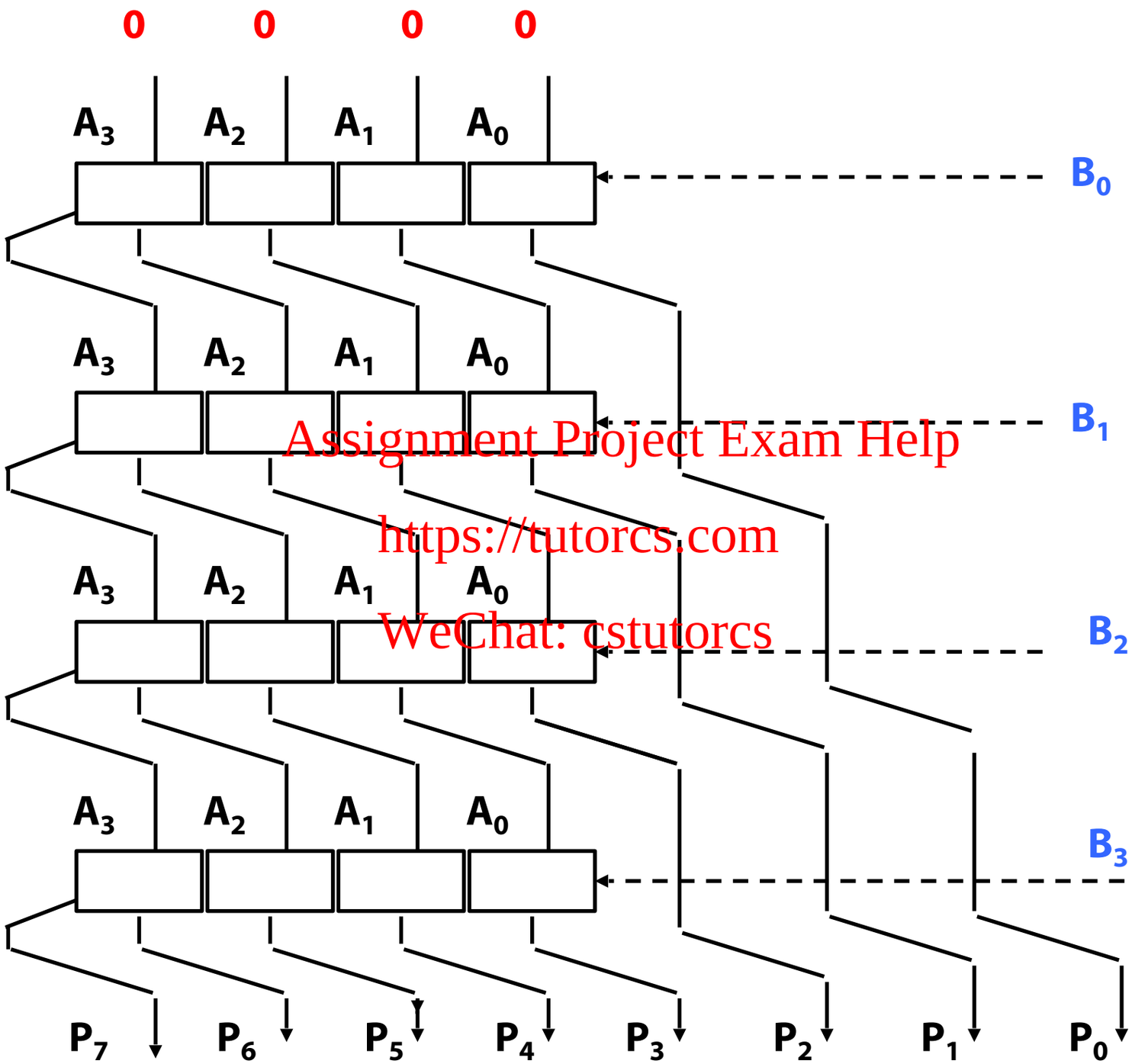


How to think of this?

- Remember original combinational multiplier:



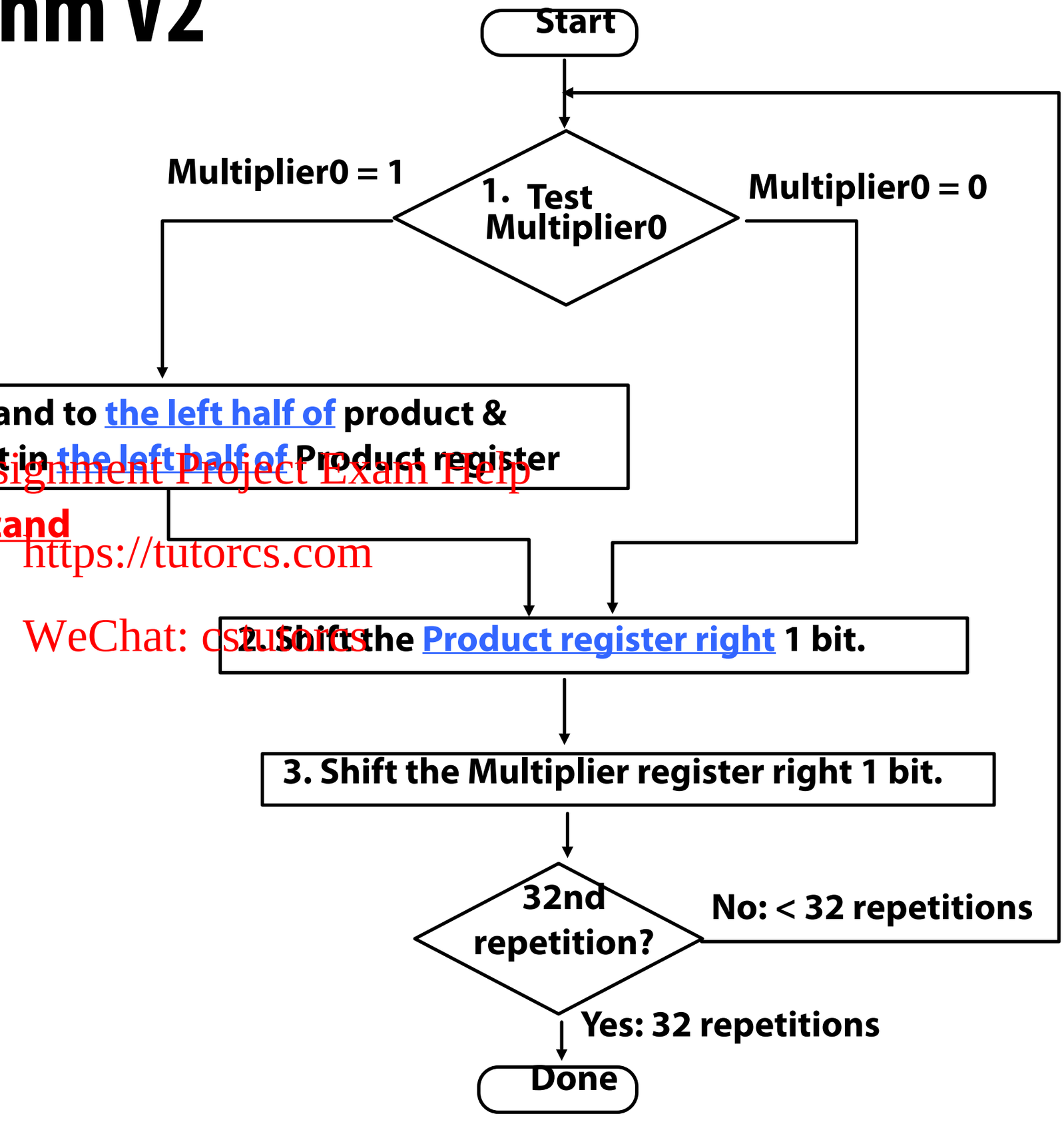
Simply warp to let product move right…



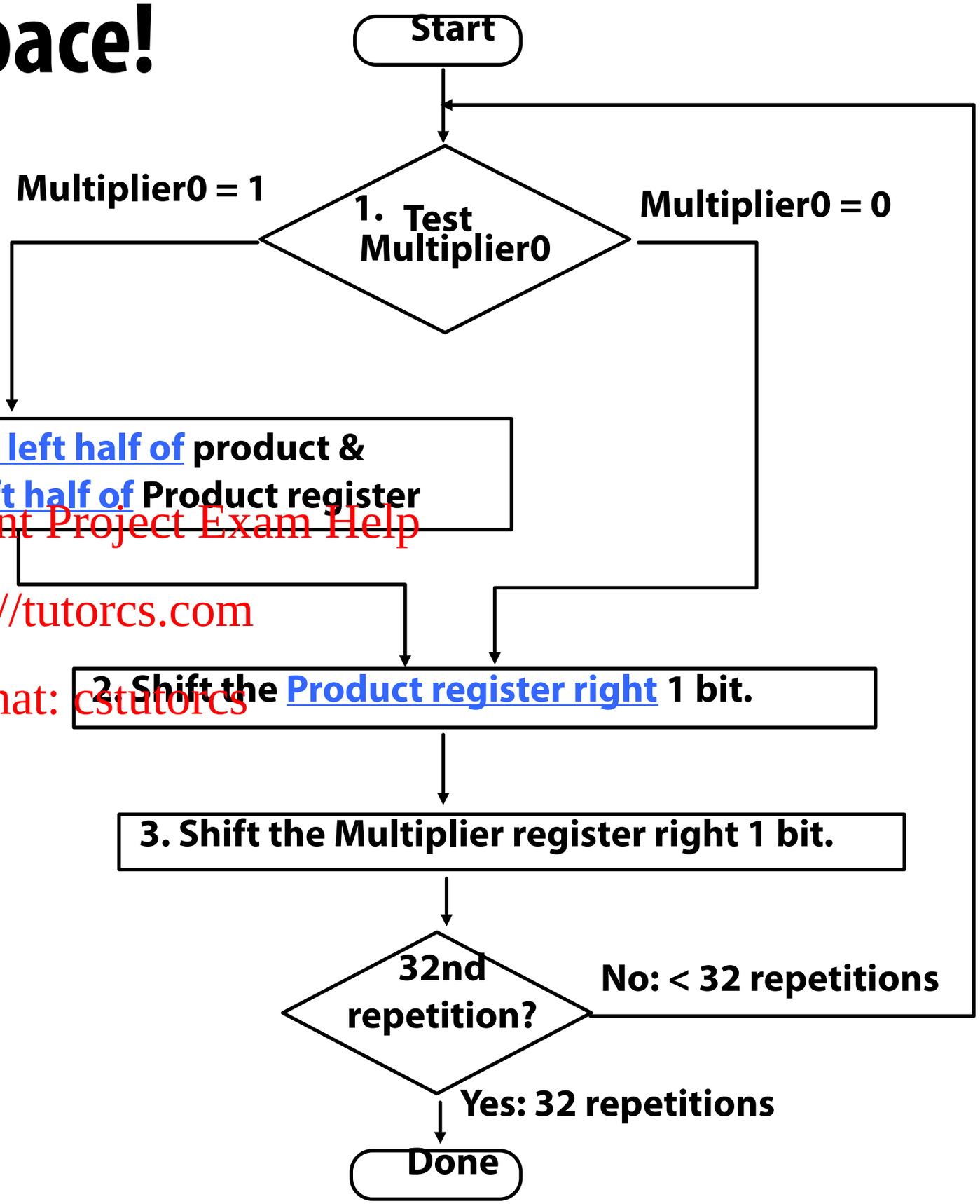
- Multiplicand stays still and product moves right

Multiply Algorithm V2

	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

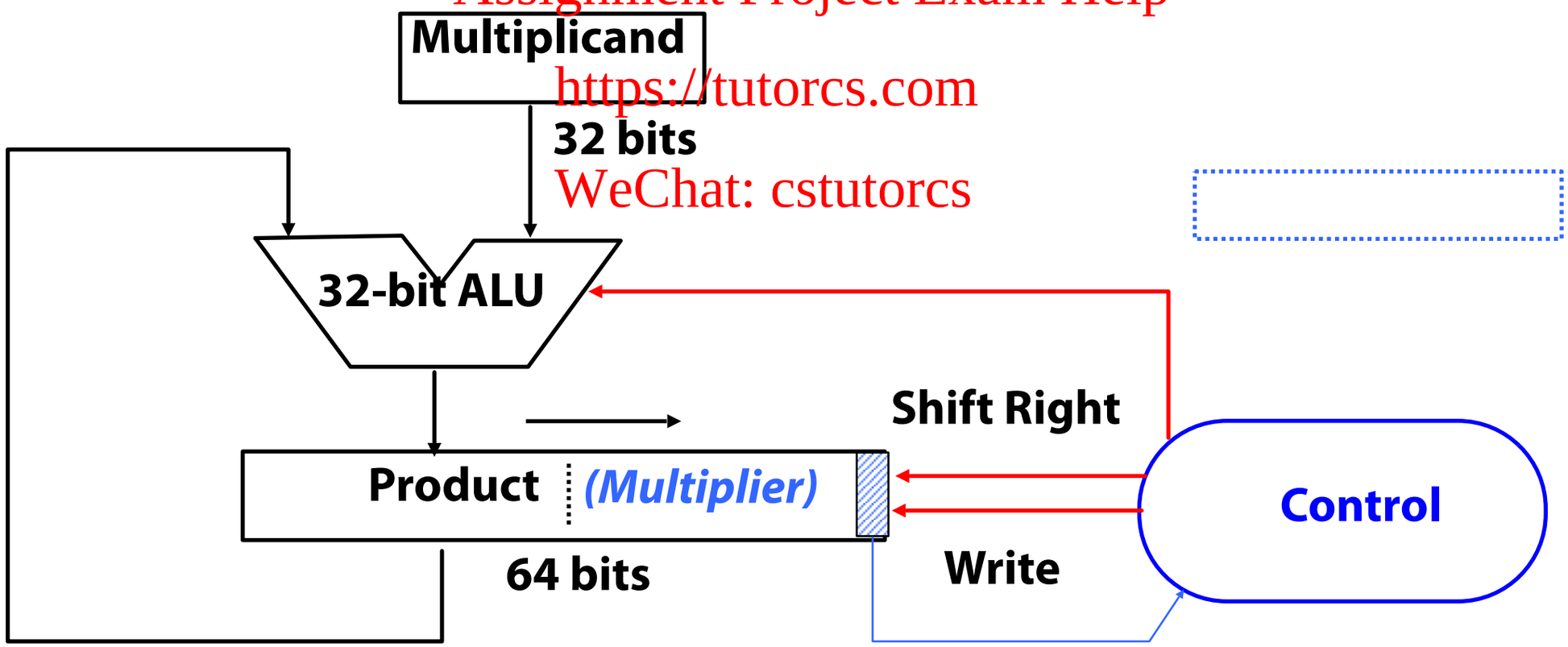
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

Observations on Multiply Version 2

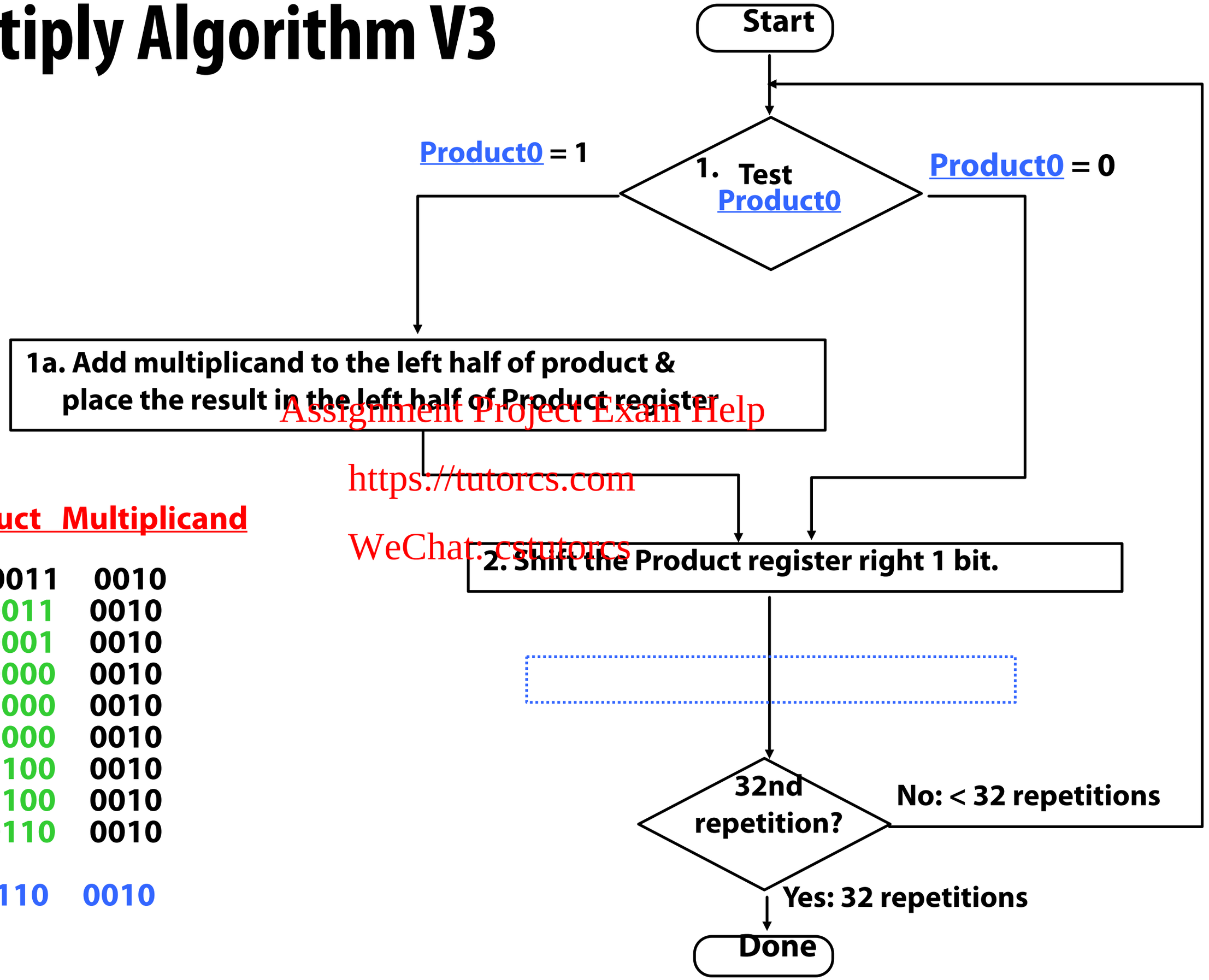
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

Multiply Hardware Version 3

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



Multiply Algorithm V3



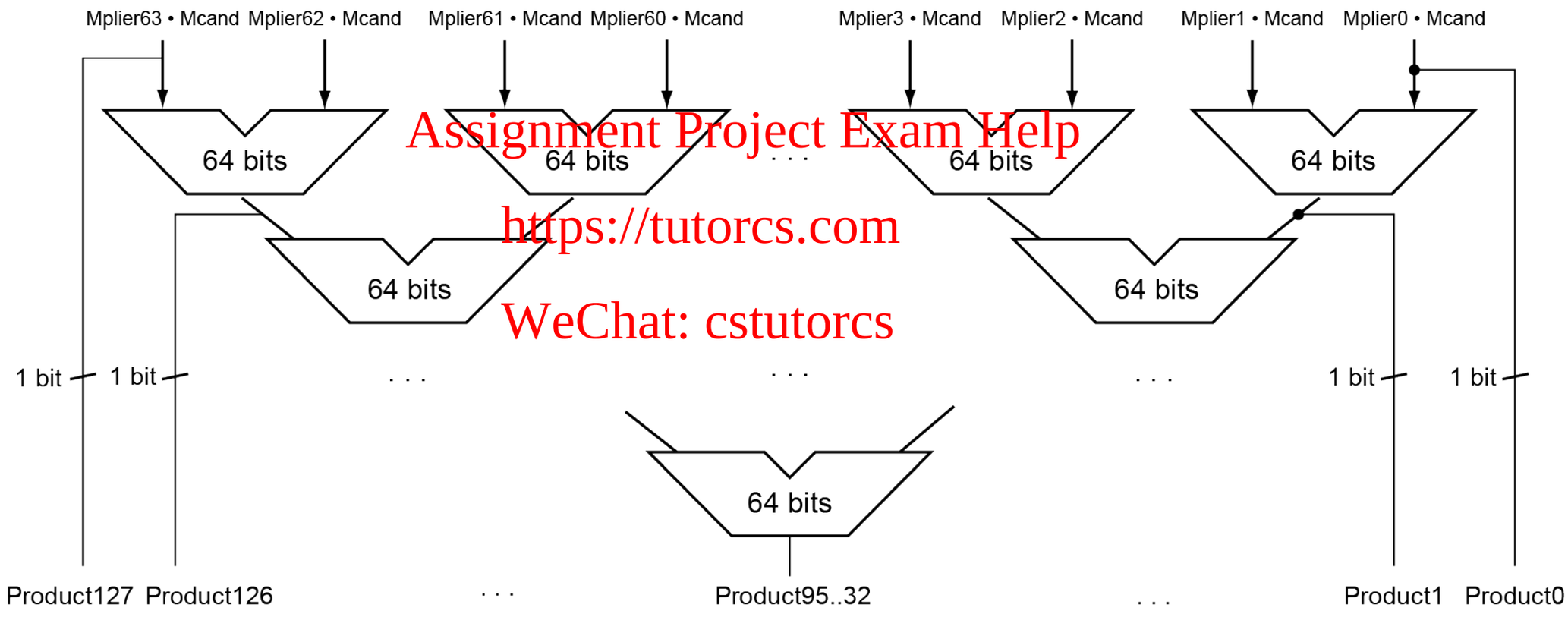
	Product	Multiplicand
	0000	0011 0010
1:	0010	0011 0010
2:	0001	0001 0010
1:	0011	0000 0010
2:	0001	1000 0010
1:	0001	1000 0010
2:	0000	1100 0010
1:	0000	1100 0010
2:	0000	0110 0010
	0000	0110 0010

Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- 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

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplications performed in parallel

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)
  ------------------------------------
     0001100
```

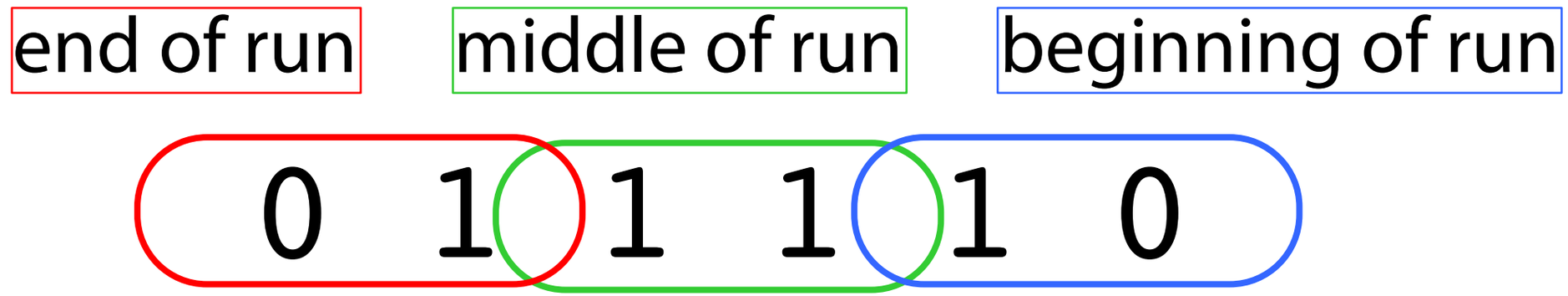
Motivation for Booth's Algorithm

- **ALU with add or subtract can get same result in more than one way:**
- `6 = 4 + 2 = -2 + 8`

  `0110 = -00010 + 01000 = 11110 + 01000`
- **For example:**


```
        0010   (2 [multiplier])
x       0110   (6 [multiplicand])
------------------------------------
        0000    shift (0 in multiplier)
-       0010    sub (first 1 in multpl.)
      0000      shift (mid string of 1s)
+     0010      add (prior step had last 1)
------------------------------------
    00001100
```

Booth's Algorithm



Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	none

- Originally for speed (when shift was 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
- Handles two's complement!

Booth's Example (2 x 7)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	0010 + 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

Blue + red = multiplier (red is 2 bits to compare); Green = arithmetic ops; Black = product

Booth's 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 + 0010	1111 0110 1	01 -> add
2a.		0001 0110 1	shift P
2b.	0010 + 1110	0000 1011 0	10 -> sub
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a.		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Blue + red = multiplier (red is 2 bits to compare); Green = arithmetic ops; Black = product

Radix-4 Modified Booth's Algorithm

Current Bits	Bit to the Right	Explanation	Example	Recode
0 0	0	Middle of zeros	00 00 00 **00** 00	0
0 1	0	Single one	00 00 00 **01** 00	1
1 0	0	Begins run of 1s	00 01 11 **10** 00	-2
1 1	0	Begins run of 1s	00 01 11 **11** 00	-1
0 0	1	Ends run of 1s	00 **00** 11 11 00	1
0 1	1	Ends run of 1s	00 **01** 11 11 00	2
1 0	1	Isolated 0	00 11 **10** 11 00	-1
1 1	1	Middle of run	00 11 **11** 11 00	0

Same insight as one-bit Booth's, simply adjust for alignment of 2 bits.

Allows multiplication 2 bits at a time.

RISC-V Multiplication Support

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 64-bit overflow

RISC-V Support for multiply

- **"If both the high and low bits of the same product are required, then the recommended code sequence is:** `MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2` **(source register specifiers must be in same order and** `rdh` **cannot be the same as** `rs1` **or** `rs2`**). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies."**

Multiplication Summary

- Iterative algorithm
- Design techniques:
  - Analyze hardware—what's not in use?
  - Spend more hardware to get higher performance
  - Booth's Algorithm—more general (2's complement)
  - Booth's Algorithm—recoding is powerful technique to think about problem in a different way
  - Booth's Algorithm—more bits at once gives higher performance

RISC-V Support for divide

- **4 instructions:**
  - `{div, divu, rem, remu} rd, rs1, rs2`
    - `div`: **rs1 / rs2, treat as signed**
    - `divu`: **rs1 / rs2, treat as unsigned**
    - `rem`: **rs1 mod rs2, treat as signed**
    - `remu`: **rs1 mod rs2, treat as unsigned**
- **"If both the quotient and remainder are required from the same division, the recommended code sequence is:** `DIV[U] rdq, rs1, rs2; REM[U] rdr, rs1, rs2` **(**`rdq` **cannot be the same as** `rs1` **or** `rs2`**). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides."**
- **Overflow and division-by-zero don't produce errors**
  - **Just return defined results**
  - **Faster for the common case of no error**

MIPS Support for multiply/divide

- Rather than target the general-purpose registers:
  - mul placed its output into two special `hi` and `lo` registers
  - div placed its divide output into `lo` and its rem output into `hi`
  - MIPS provided `mflo` and `mfhi` instructions (destination: general-purpose register)

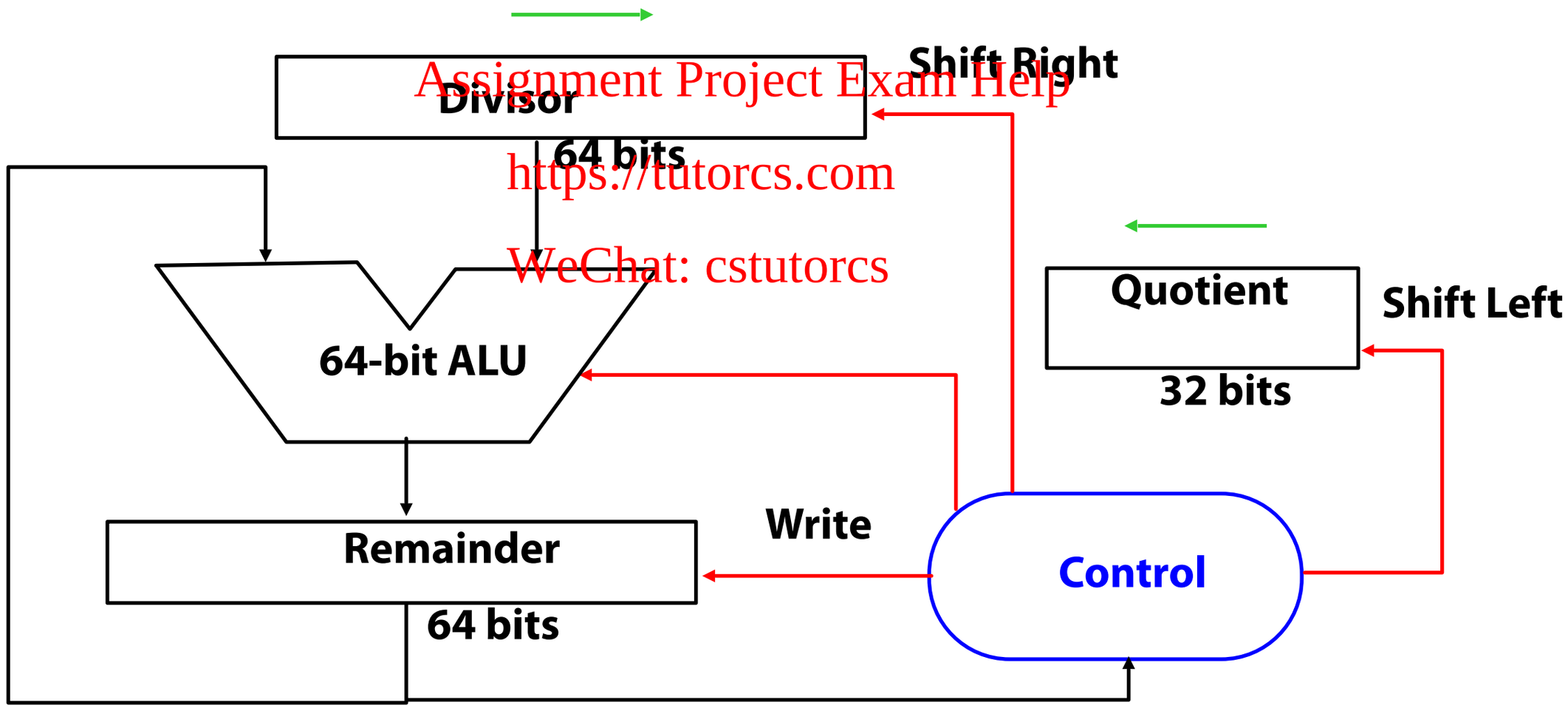
Divide: Paper & Pencil

```
                    1001   Quotient
                 ┌────────
Divisor    1000  │ 1001010      Dividend
                 -1000
                 ──────
                     10
                     101
                     1010
                    -1000
                    ─────
                       10      Remainder
                               (or Modulo result)
```


- 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

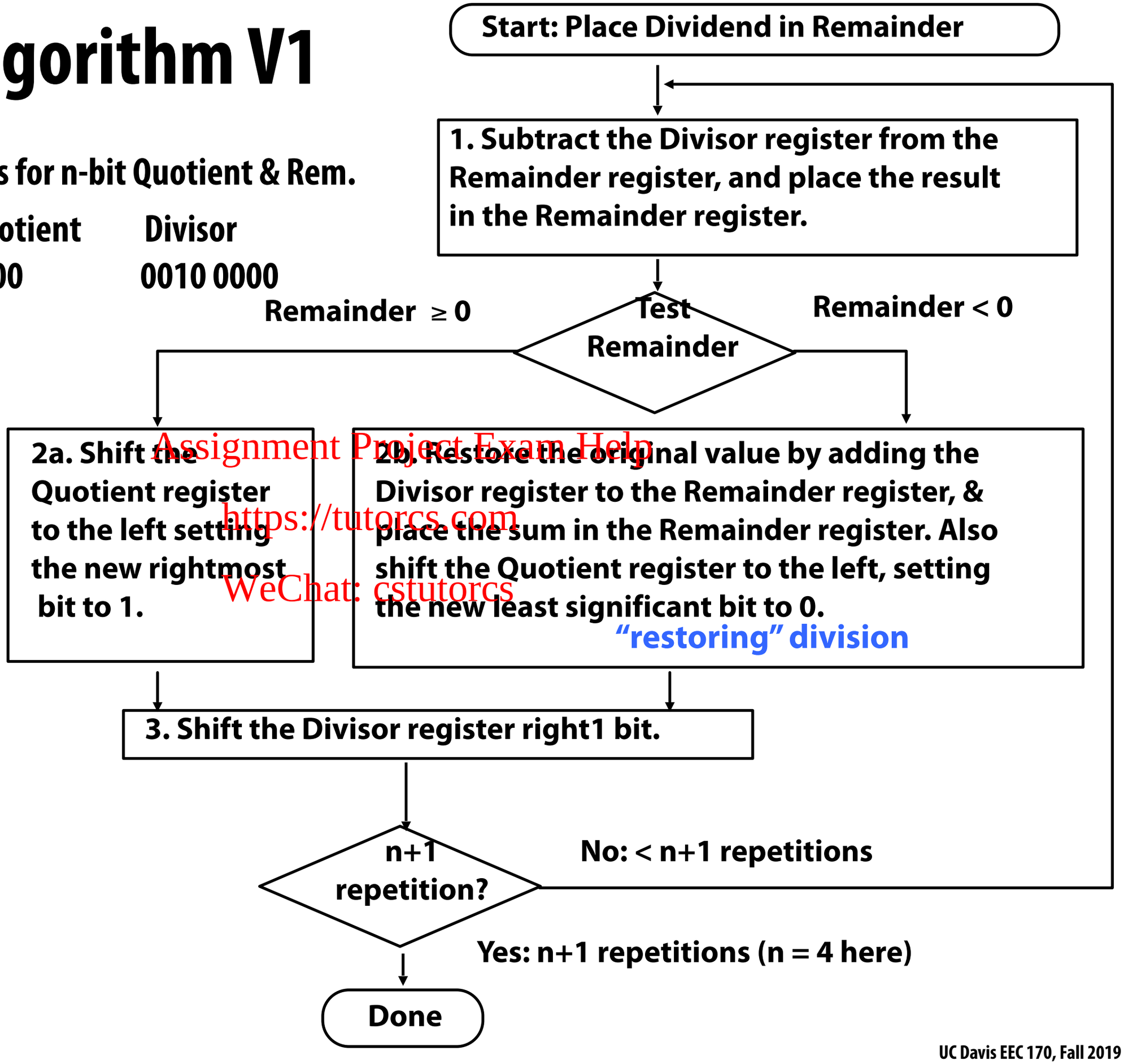
Divide Hardware Version 1

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



Divide Algorithm V1

- Takes n+1 steps for n-bit Quotient & Rem.
- Remainder Quotient Divisor
  0000 0111 0000 0010 0000



Divide Algorithm I example (7 / 2)

	Remainder	Quotient	Divisor
	0000 0111	00000	0010 0000
1:	1110 0111	00000	0010 0000
2:	0000 0111	00000	0010 0000
3:	0000 0111	00000	0001 0000
1:	1111 0111	00000	0001 0000
2:	0000 0111	00000	0001 0000
3:	0000 0111	00000	0000 1000
1:	1111 1111	00000	0000 1000
2:	0000 0111	00000	0000 1000
3:	0000 0111	00000	0000 0100
1:	0000 0011	00000	0000 0100
2:	0000 0011	00001	0000 0100
3:	0000 0011	00001	0000 0010
1:	0000 0001	00001	0000 0010
2:	0000 0001	00011	0000 0010
3:	0000 0001	00011	0000 0001

Answer:
Quotient = 3
Remainder = 1

Divide: Paper & Pencil

```
                    01010     Quotient
Divisor 0001 |00001010        Dividend
             |00001
              -0001
                 0000
                  0001
                 -0001
                     0
                     00       Remainder
(or Modulo result)
```


- No way to get a 1 in leading digit!
  - (this is an overflow, i.e quotient would have n+1 bits)
  - ⇒ switch order to shift first and then subtract, can save 1 iteration

Observations on Divide Version 1

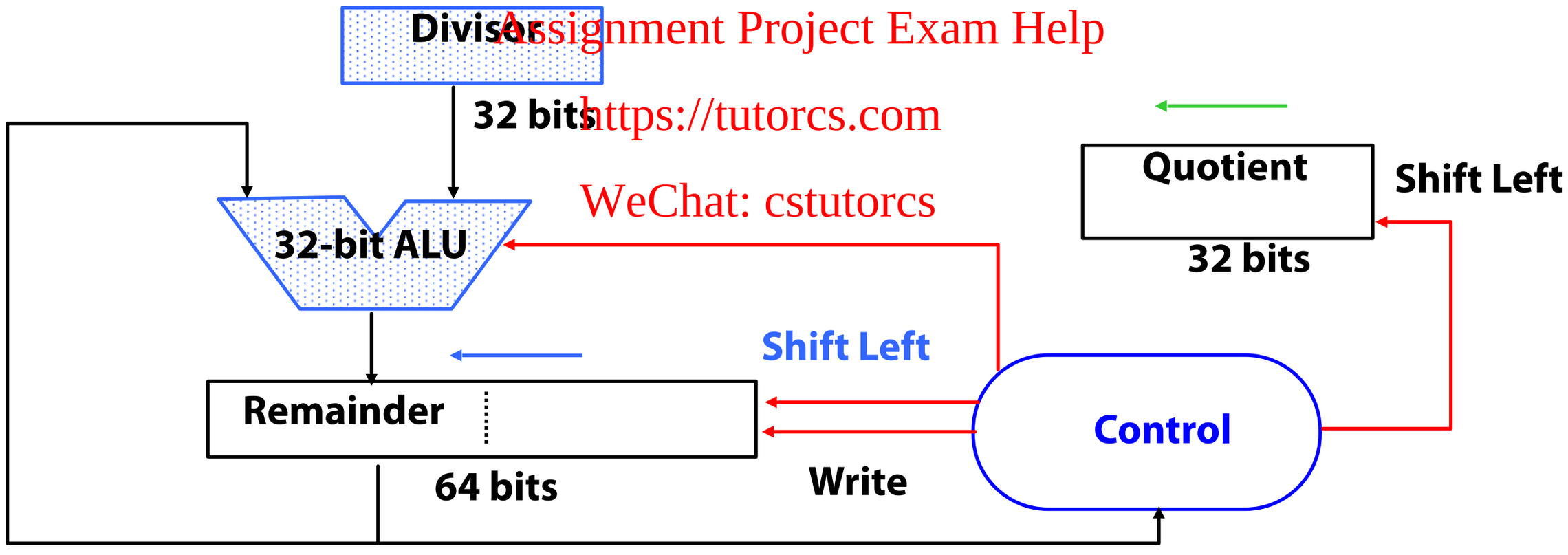
- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?

Divide Algorithm I example: wasted space

Remainder	Quotient	Divisor
0000 0111	00000	0010 0000
1:1110 0111	00000	0010 0000
2:0000 0111	00000	0010 0000
3:0000 0111	00000	0001 0000
1:1111 0111	00000	0001 0000
2:0000 0111	00000	0001 0000
3:0000 0111	00000	0000 1000
1:1111 1111	00000	0000 1000
2:0000 0111	00000	0000 1000
3:0000 0111	00000	0000 0100
1:0000 0011	00000	0000 0100
2:0000 0011	00001	0000 0100
3:0000 0011	00001	0000 0010
1:0000 0001	00001	0000 0010
2:0000 0001	00011	0000 0010
3:0000 0001	00011	0000 0010

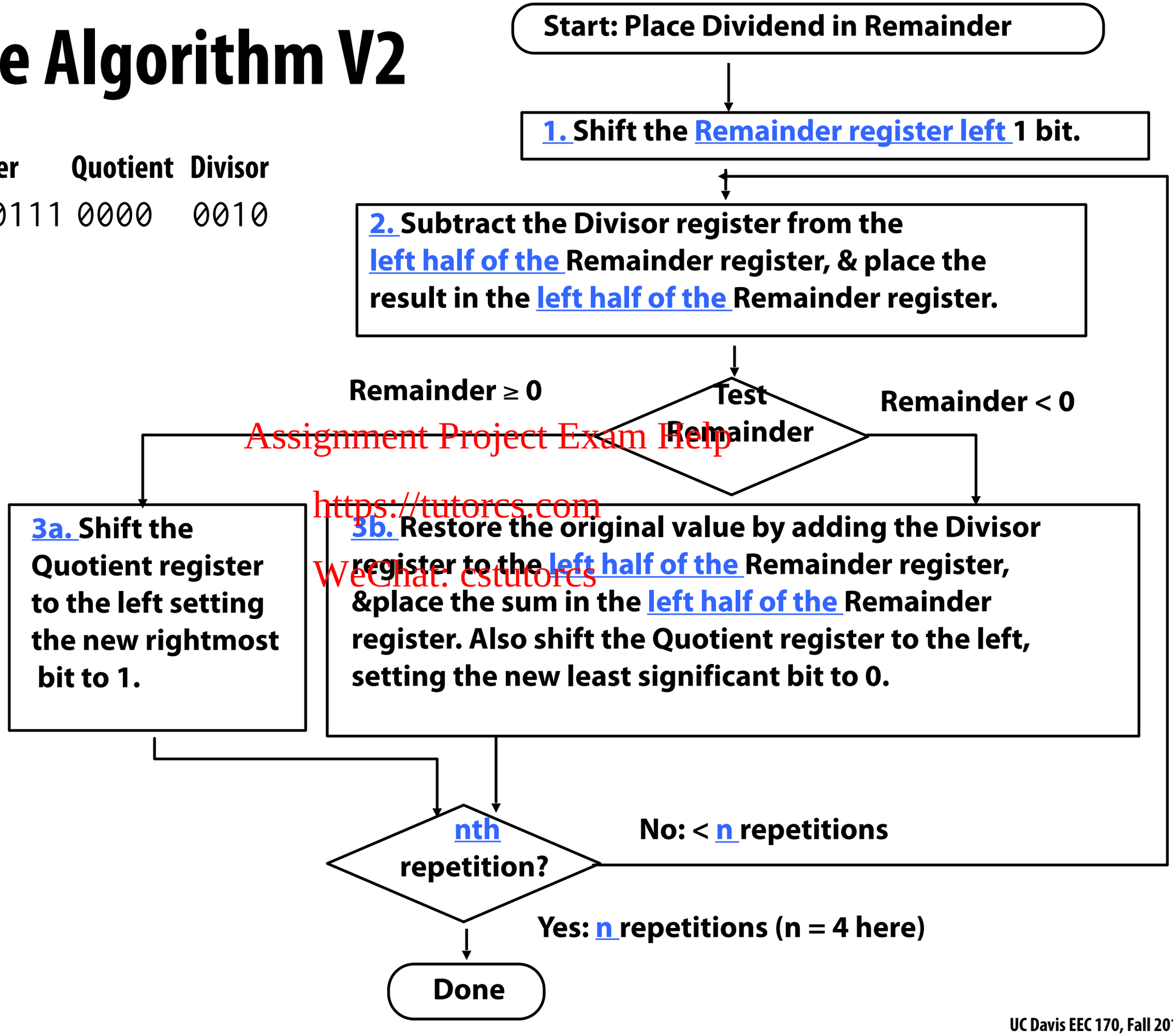
Divide Hardware Version 2

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



Divide Algorithm V2

Remainder	Quotient	Divisor
0000 0111	0000	0010

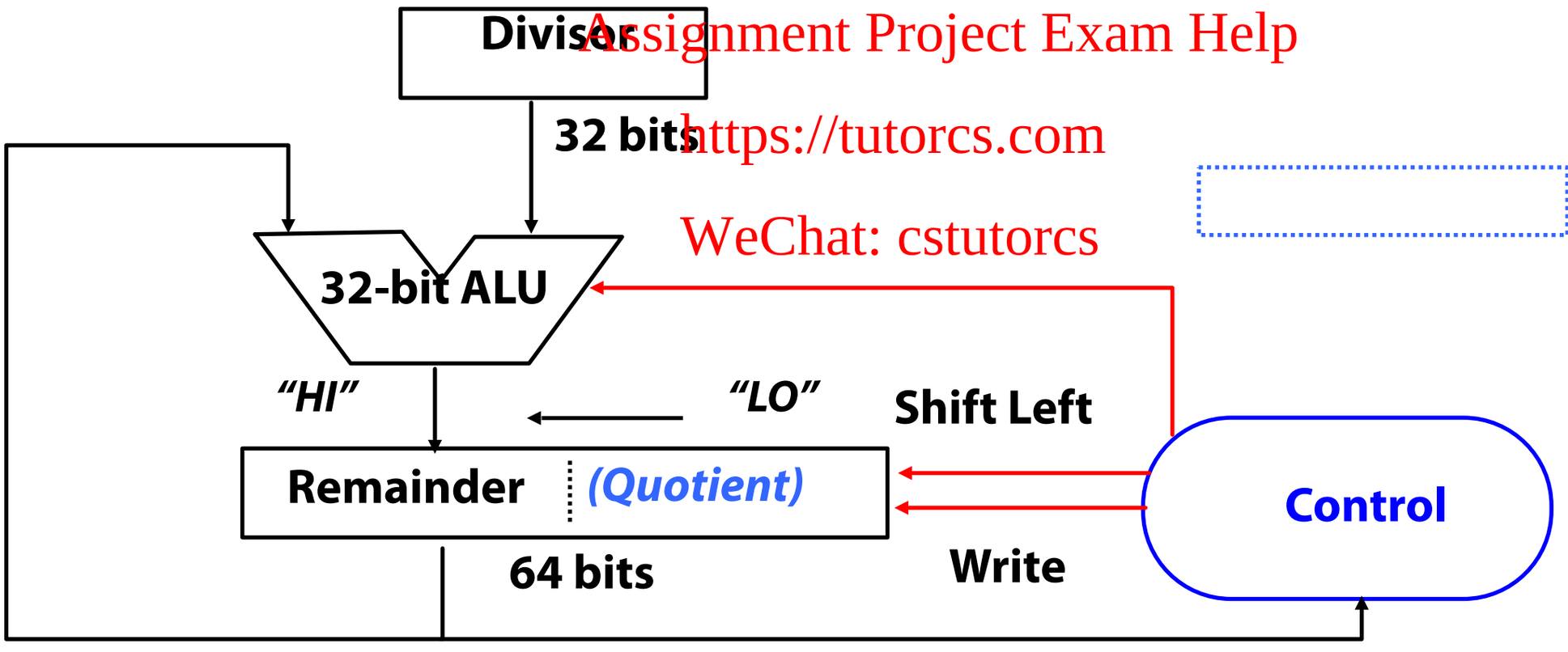


Observations on Divide Version 2

- 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 shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

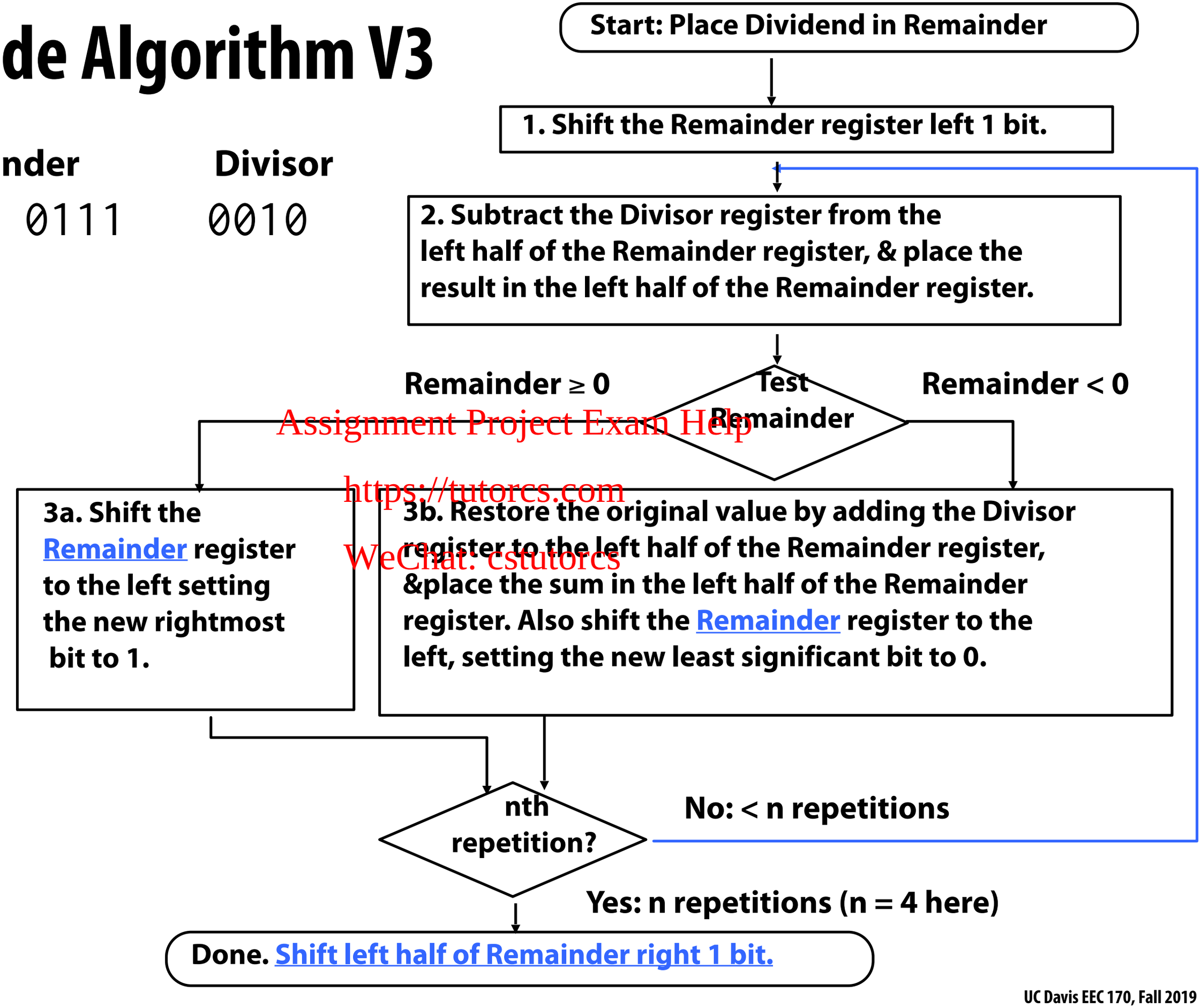
Divide Hardware Version 3

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

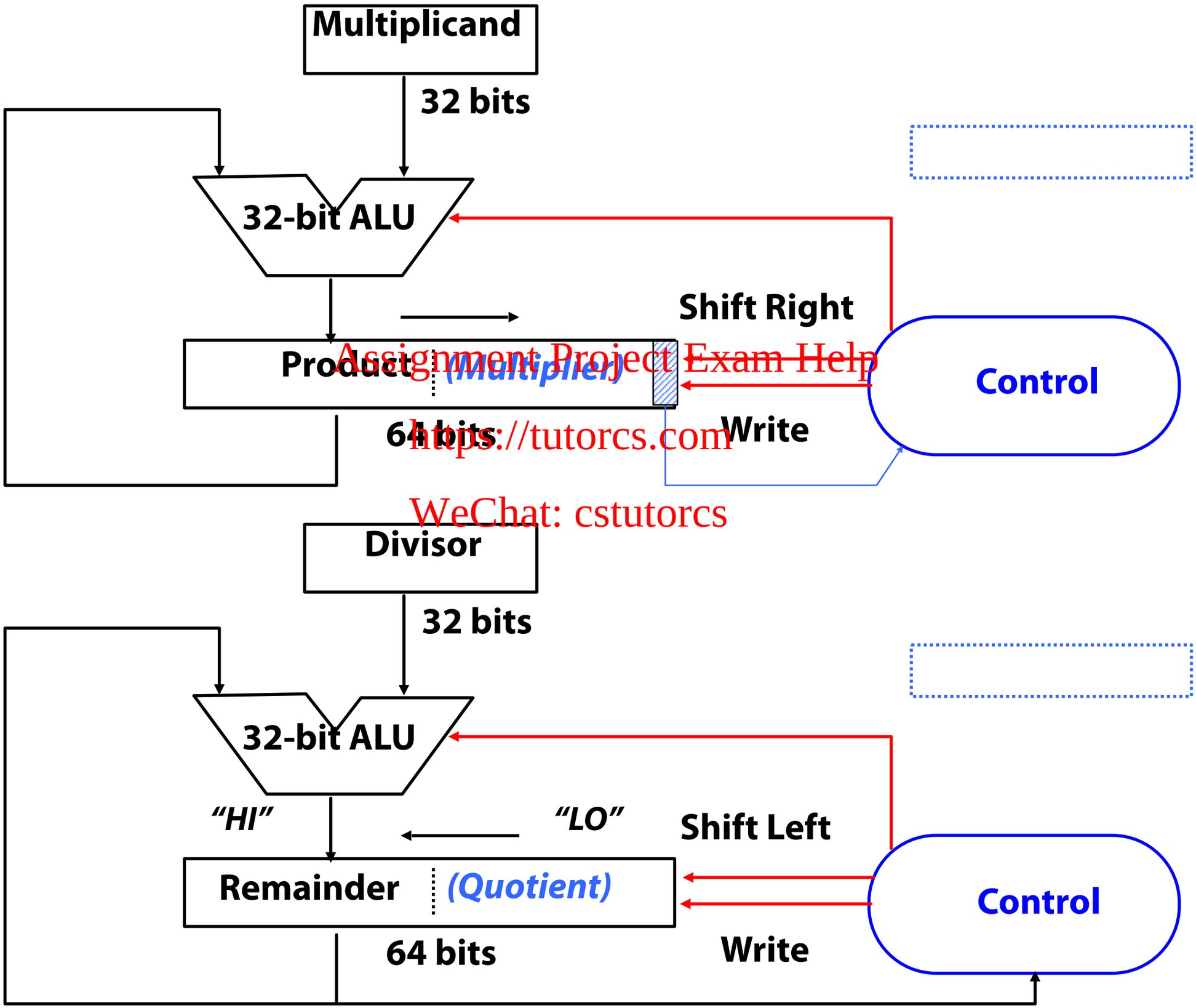


Divide Algorithm V3

Remainder	Divisor
0000 0111	0010



Final Multiply / Divide Hardware

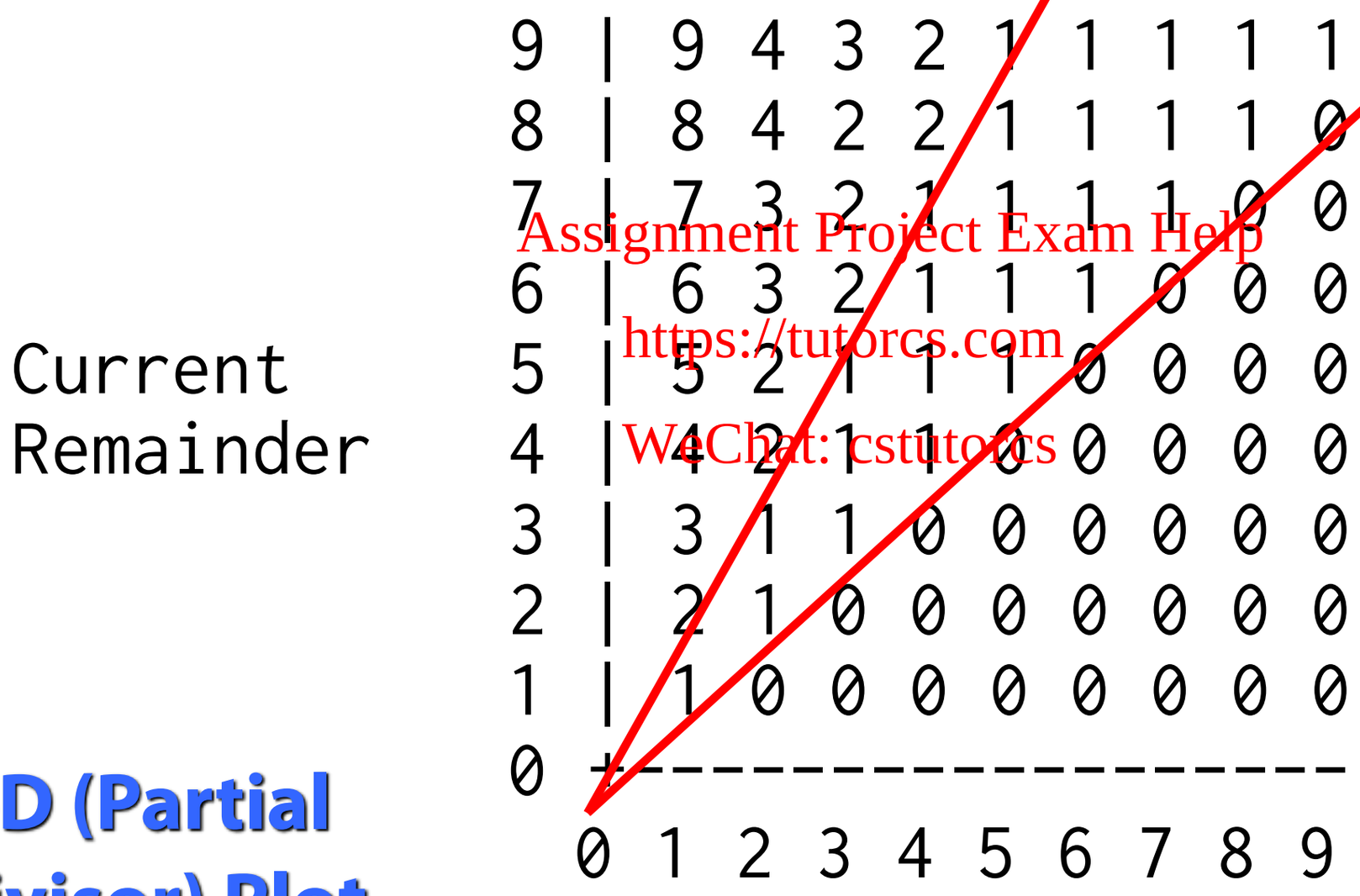


Observations on Divide Version 3

- 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
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree
    e.g., –7 ÷ 2 = –3, remainder = –1
  - What about? –7 ÷ 2 = –4, remainder = +1
  - See http://mathforum.org/library/drmath/view/52343.html
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits (called "saturation")

SRT Division

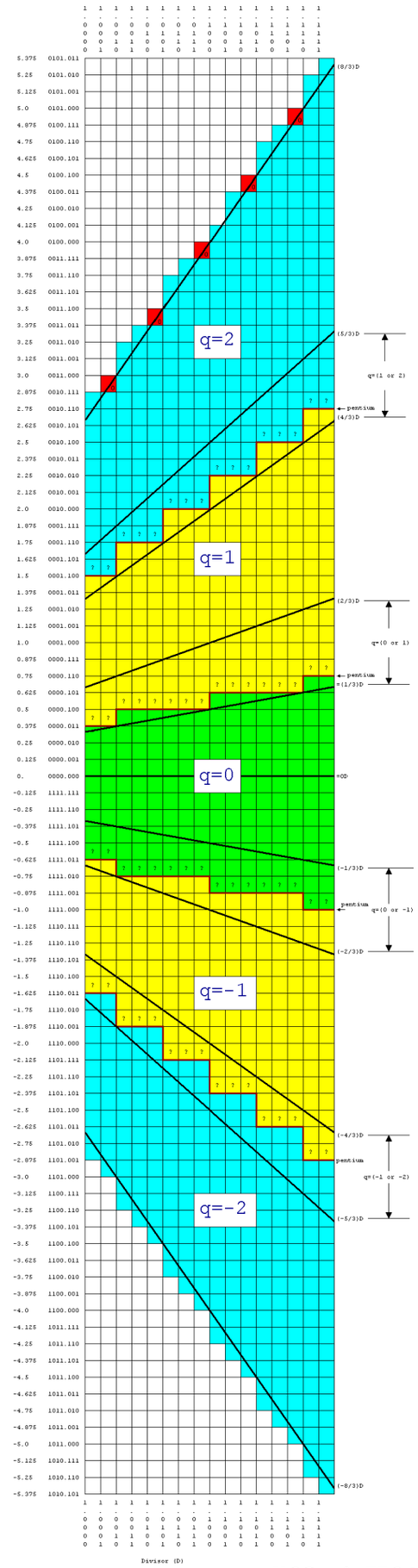
D. Sweeney of IBM, J.E. Robertson of the University of Illinois, and T.D. Tocher of Imperial College, London



P-D (Partial Divisor) Plot

SRT Division

- Intel Pentium divide implementation: SRT division with 2 bits/iteration (radix 4)
- Allows negative entries
- 1066 entries in lookup table



[http://members.cox.net/srice1/pentbug/introduction.html]

Faster Division

- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
  - Still require multiple steps

Division Lessons

- In practice, slower than multiplication
  - Also less frequent
  - But, in the simple case, can use same hardware!
- Generates quotient and remainder together
- Floating-point division faster than integer division (why?)
- Similar hardware lessons as multiplier:
  - Look for unused hardware
  - Can process multiple bits at once at cost of extra hardware

RISC-V logical instructions

Instruction	Meaning	Pseudocode
`XORI rd,rs1,imm`	Exclusive Or Immediate	rd ← ux(rs1) ⊕ ux(imm)
`ORI rd,rs1,imm`	Or Immediate	rd ← ux(rs1) ∨ ux(imm)
`ANDI rd,rs1,imm`	And Immediate	rd ← ux(rs1) ∧ ux(imm)
`SLLI rd,rs1,imm`	Shift Left Logical Immediate	rd ← ux(rs1) « ux(imm)
`SRLI rd,rs1,imm`	Shift Right Logical Immediate	rd ← ux(rs1) » ux(imm)
`SRAI rd,rs1,imm`	Shift Right Arithmetic Immediate	rd ← sx(rs1) » ux(imm)
`SLL rd,rs1,rs2`	Shift Left Logical	rd ← ux(rs1) « rs2
`XOR rd,rs1,rs2`	Exclusive Or	rd ← ux(rs1) ⊕ ux(rs2)
`SRL rd,rs1,rs2`	Shift Right Logical	rd ← ux(rs1) » rs2
`SRA rd,rs1,rs2`	Shift Right Arithmetic	rd ← sx(rs1) » rs2
`OR rd,rs1,rs2`	Or	rd ← ux(rs1) ∨ ux(rs2)
`AND rd,rs1,rs2`	And	rd ← ux(rs1) ∧ ux(rs2)

Shift Operations

- **Bit manipulation:**
  - `  S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM`
    `& 0 11111111 00000000000000000000000`
    `-------------------------------------`
    `  0 EEEEEEEE 00000000000000000000000`
  - **Right shift 23 bits to get**
    `00000000000000000000000 EEEEEEEE`
  - **Do arithmetic manipulation**
    `00000000000000000000000 ENEWENEW`
  - **Left shift 23 bits to get**
    `0 ENEWENEW 00000000000000000000000`

Shift Operations

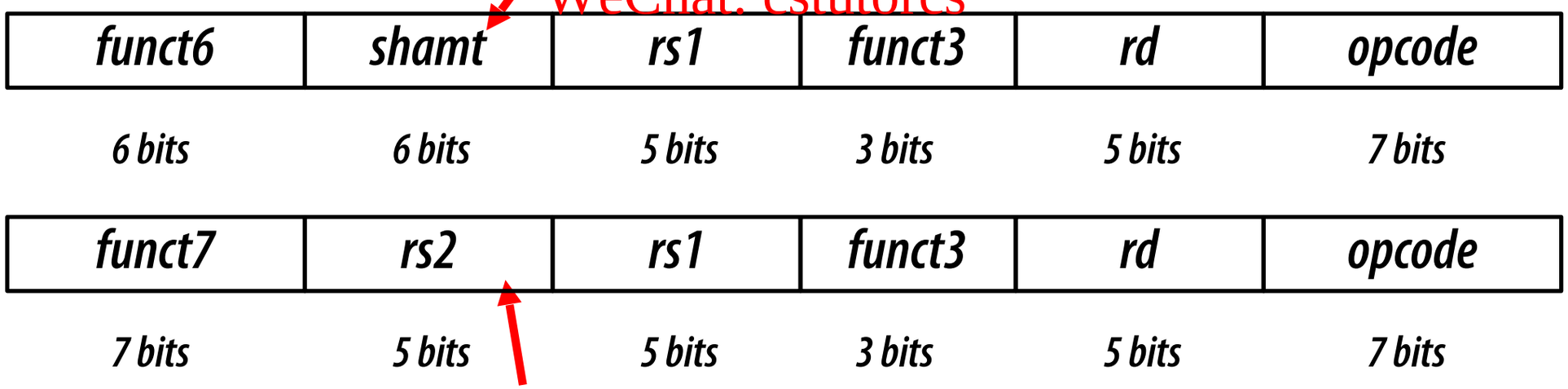
- Arithmetic operation:
  - Example: 00011 << 2 [3 left shift 2]
    - 00011 << 2 = 01100 = 12 = 2 * 4
  - Each bit shifted left == multiply by two
  - Example: 01010 >> 1 [10 right shift 1]
    - 01010 >> 1 = 00101 = 5 = 10/2
- Each bit shifted right == divide by two
- Why?
- Compilers do this—"strength reduction"

Shift Operations

- With left shift, what do we shift in?
  - 00011 << 2 = 01100 (arithmetic)
  - 0000XXXX << 4 = XXXX0000 (logical)
  - We shifted in zeroes
- How about right shift?
  - XXXX0000 >> 4 = 0000XXXX (logical)
    - Shifted in zero
  - 00110 (= 6) >> 1 = 00011 (3) (arithmetic)
    - Shifted in zero
  - 11110 (= -2) >> 1 = 11111 (-1) (arithmetic)
    - Shifted in one

Shift Operations

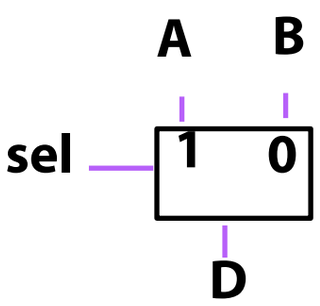
- How about right shift?
  - XXXX0000 >> 4 = 0000XXXX: Logical shift
    - Shifted in zero
  - 00110 (= 6) >> 1 = 00011 (3)
    11110 (= -2) >> 1 = 11111 (-1): Arithmetic shift
    - Shifted in sign bit
- RISC-V supports both logical and arithmetic:
  - slli, srai, srli: Shift amount taken from within instruction ("imm")



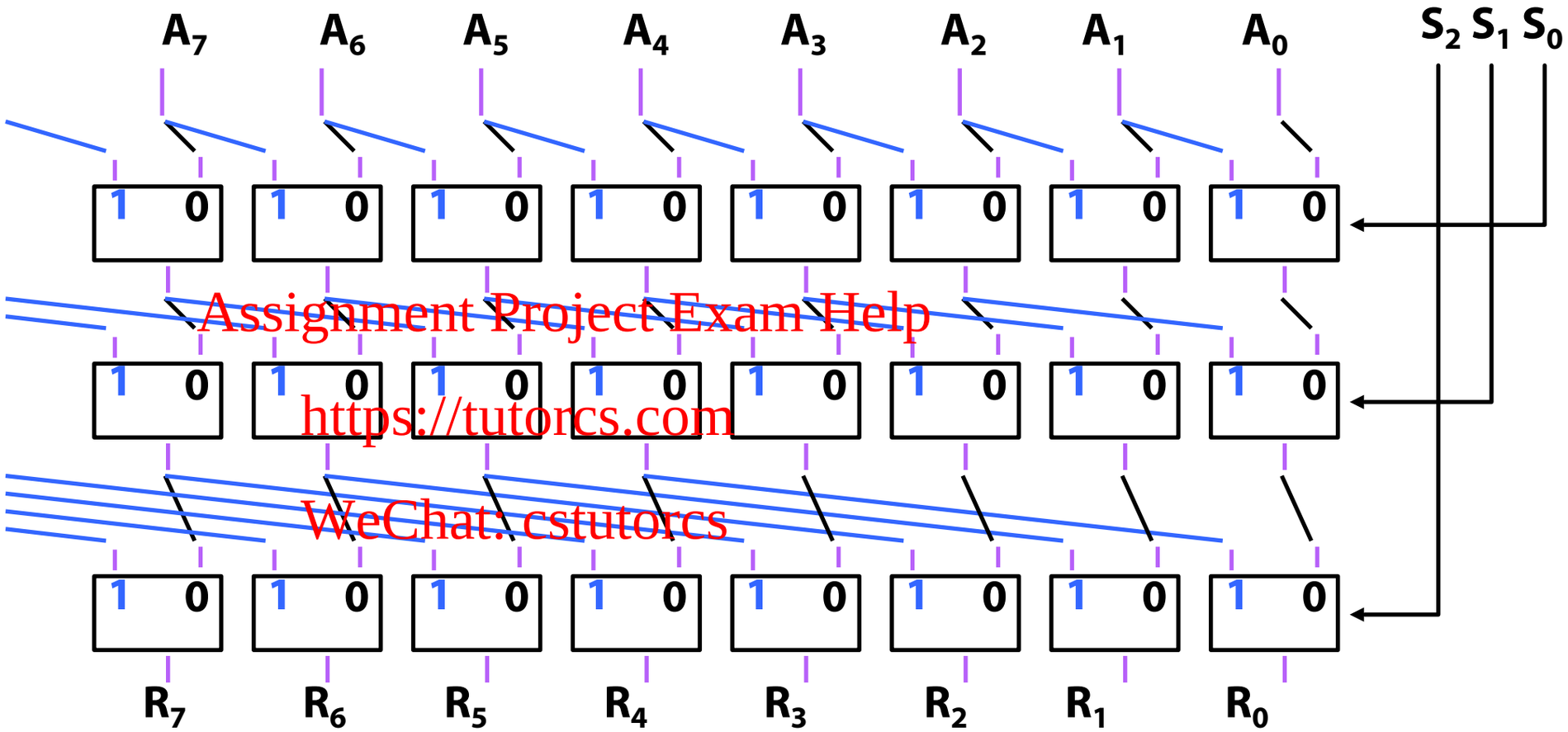
  - sll, sra, srl: shift amount taken from register ("variable")
  - How far can we shift with slli/srai/slli? With sll/sra/srl?

Combinational Shifter from MUXes

Basic Building Block

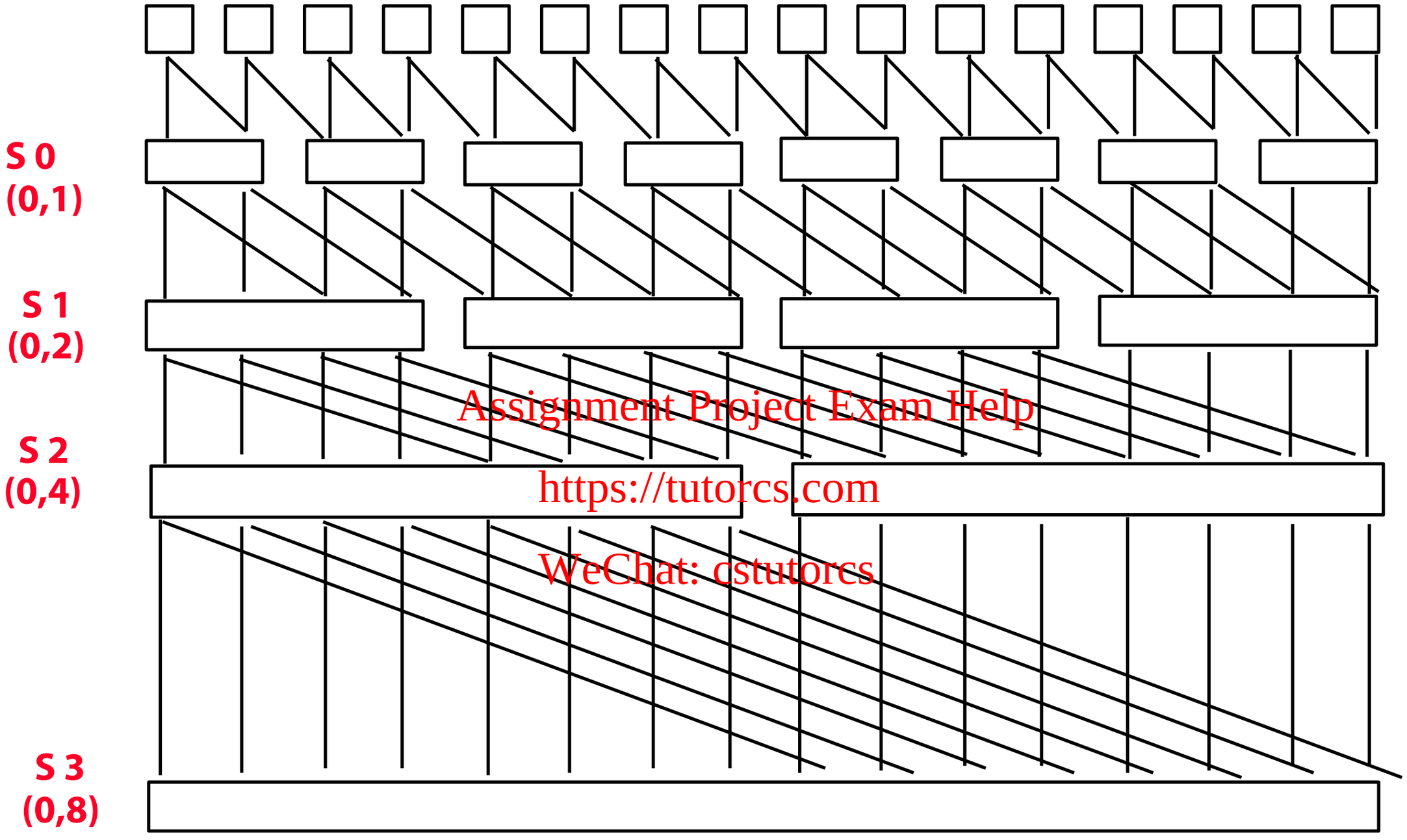


8-bit right shifter



- What comes in the MSBs?
- How many levels for 64-bit shifter?
- What if we use 4-1 Muxes ?

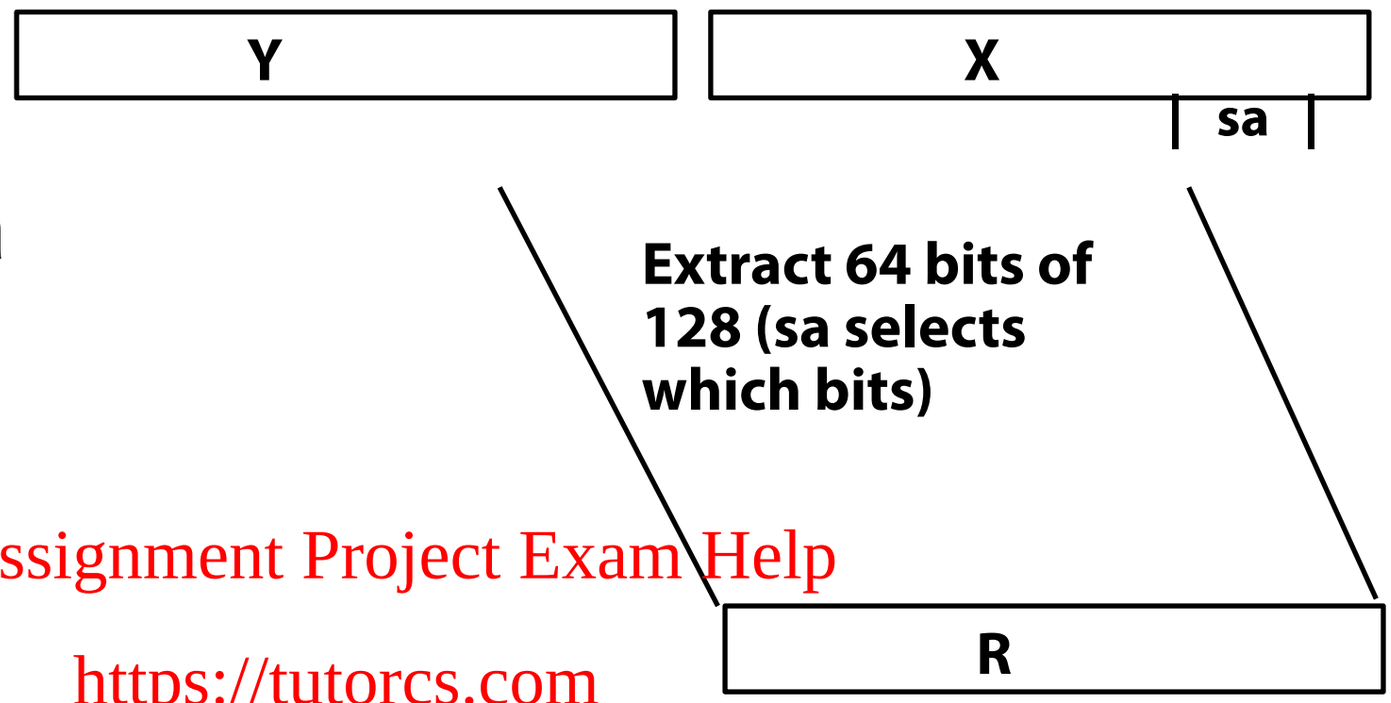
General Shift Right Scheme using 16 bit example



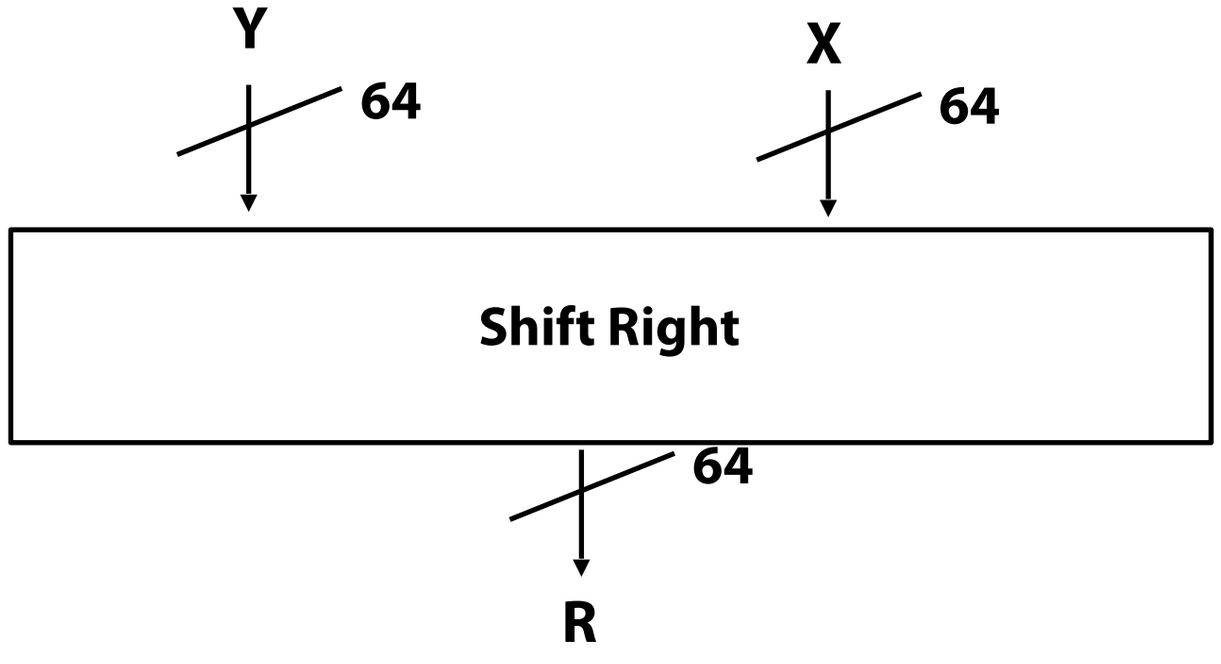
- **If we added right-to-left connections, we could support** ROTATE **(not in RISC-V but found in other ISAs)**

Funnel Shifter

- Shift A by i bits
- Problem: Set Y, X, sa
- Logical:

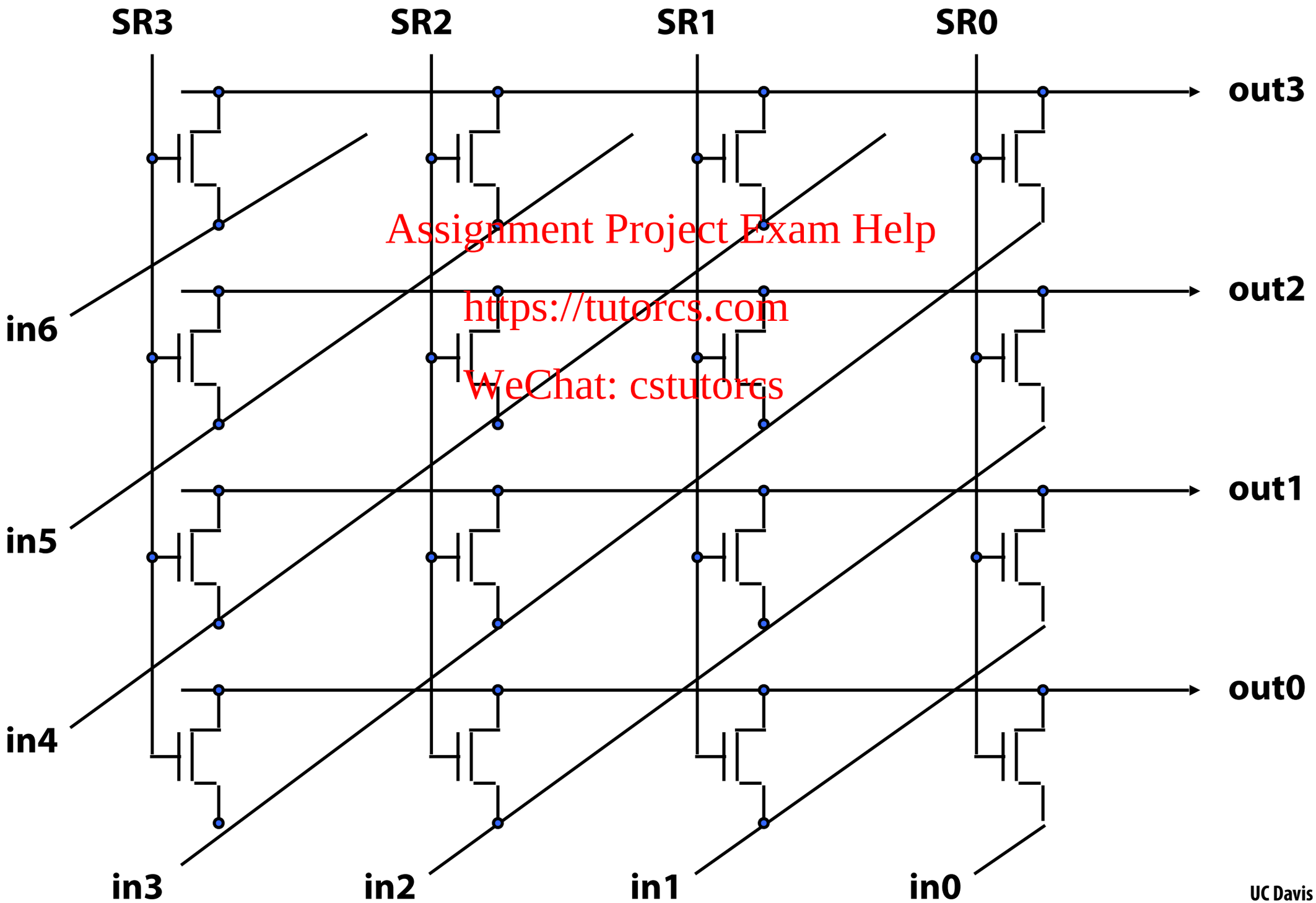


- Arithmetic:
- Rotate:
- Left shifts:



Barrel Shifter

- Technology-dependent solutions: transistor per switch



Shifter Summary

- Shifts common in logical ops, also in arithmetic
- MIPS has:
  - 2 flavors of shift: logical and arithmetic
  - 2 directions of shift: right and left
  - 2 sources for shift amount: immediate, variable
- Lots of cool shift algorithms, but . . .
  - Barrel shifter prevalent in today's hardware