Lecture 6:

Arithmetic 2/3

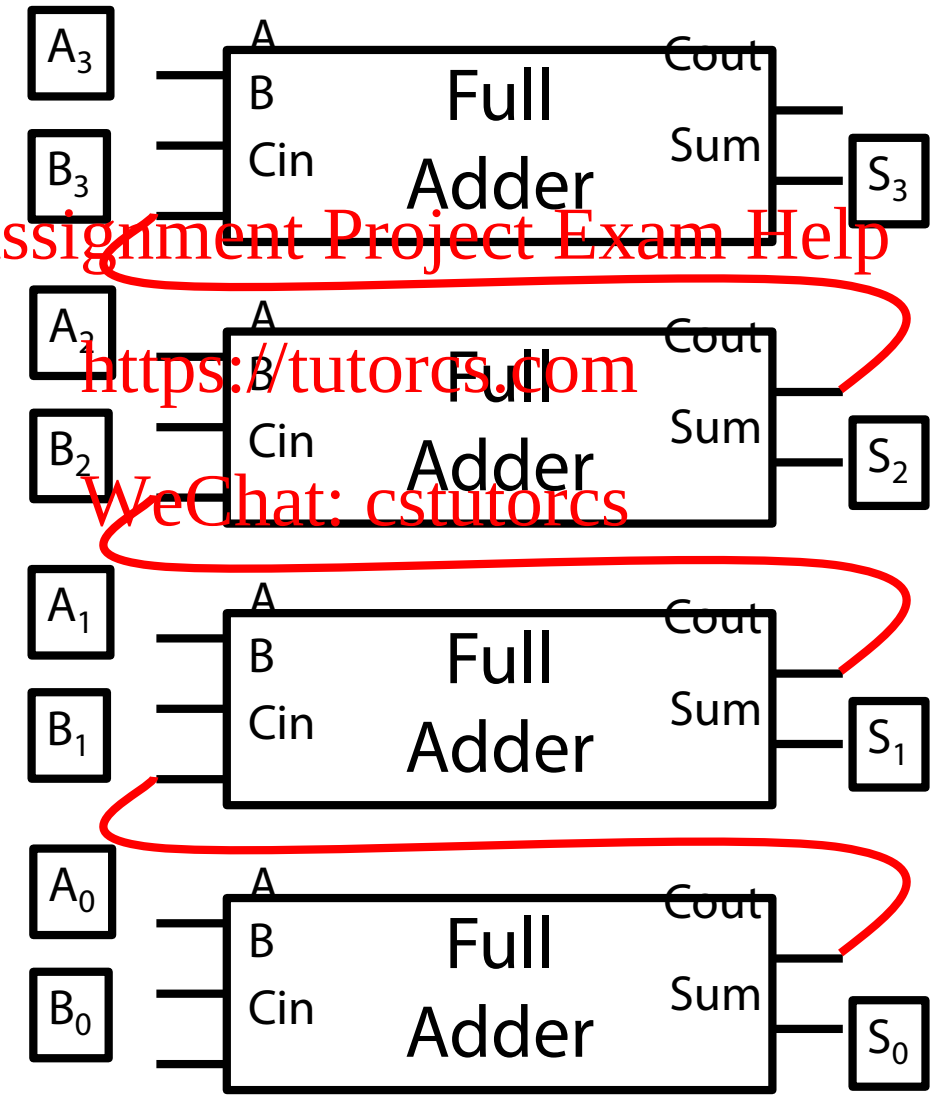
John Owens

Introduction to Computer Architecture

UC Davis EEC 170, Winter 2021

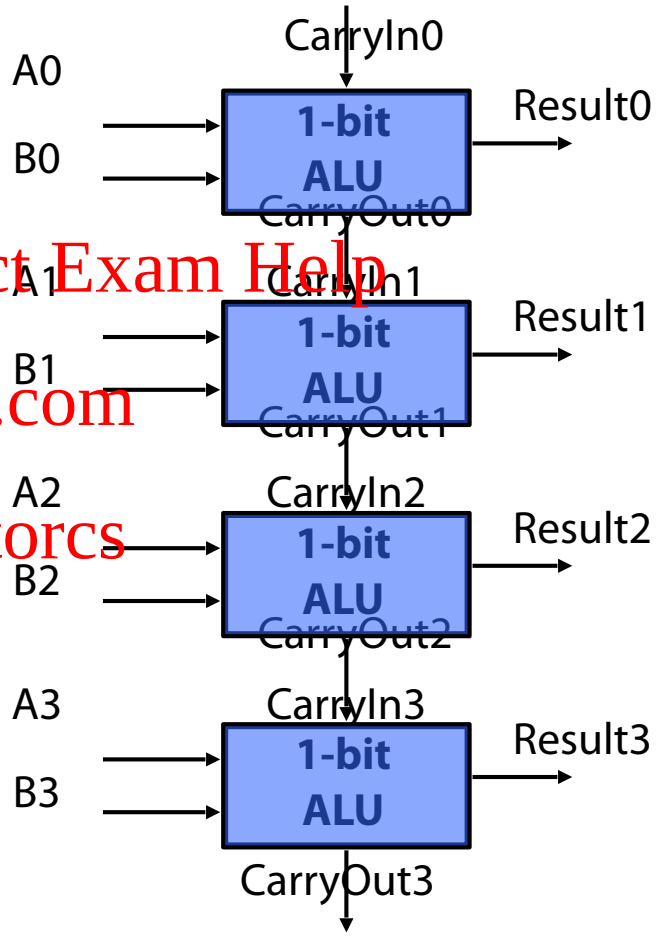
Cascading Adders

- Cascade Full Adders to make multibit adder:
- A+B=S



But What About Performance?

- Critical path of one bitslice is CP
- Critical path of n-bit rippled-carry adder is n*CP
  - Thought experiment:
  - @ 7 nm: FO4 is ~2.5 ps
  - 2 gate delays * 64b = 320 ps
  - 4 GHz clock period is 250 ps
- Design Trick:
  - Throw hardware at it

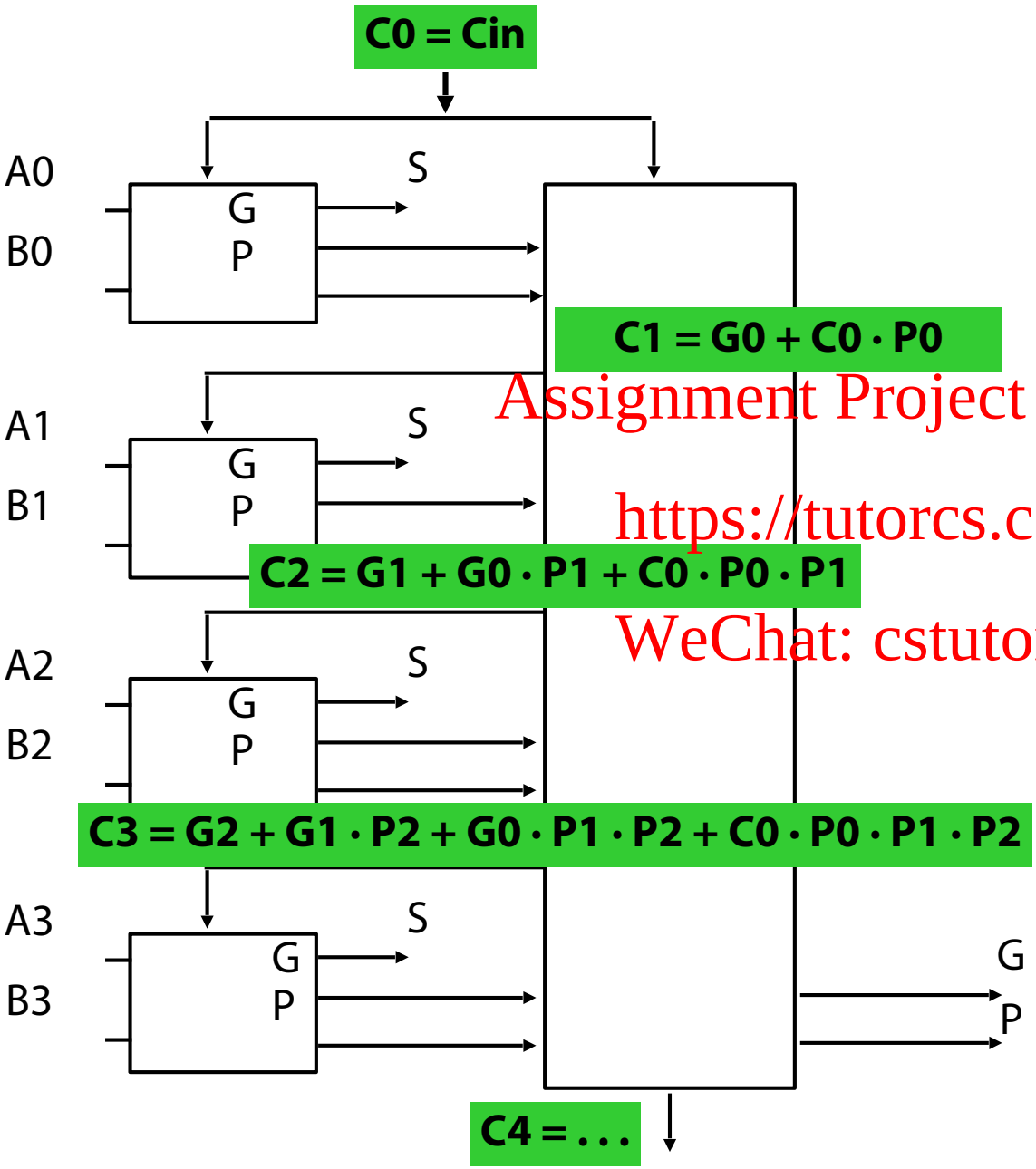


Truth Table for Adder Bit Slice

- 3 inputs (A, B, Cin); 2 outputs (Sum, Cout)

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0=Cin
0	1	1	0	1=Cin
1	0	0	1	0=Cin
1	0	1	0	1=Cin
1	1	0	0	1
1	1	1	1	1

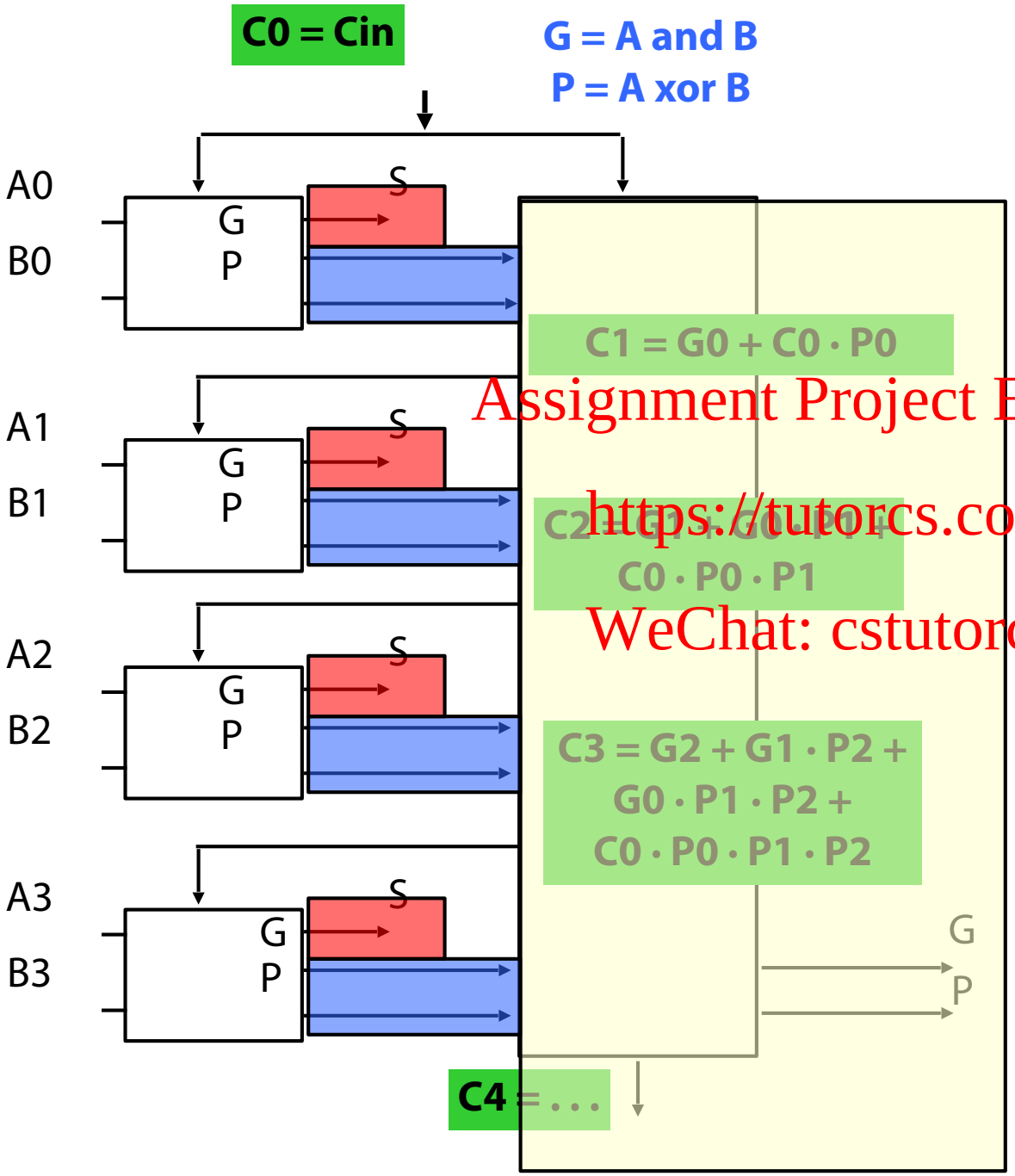
Carry Look Ahead (Design trick: peek)

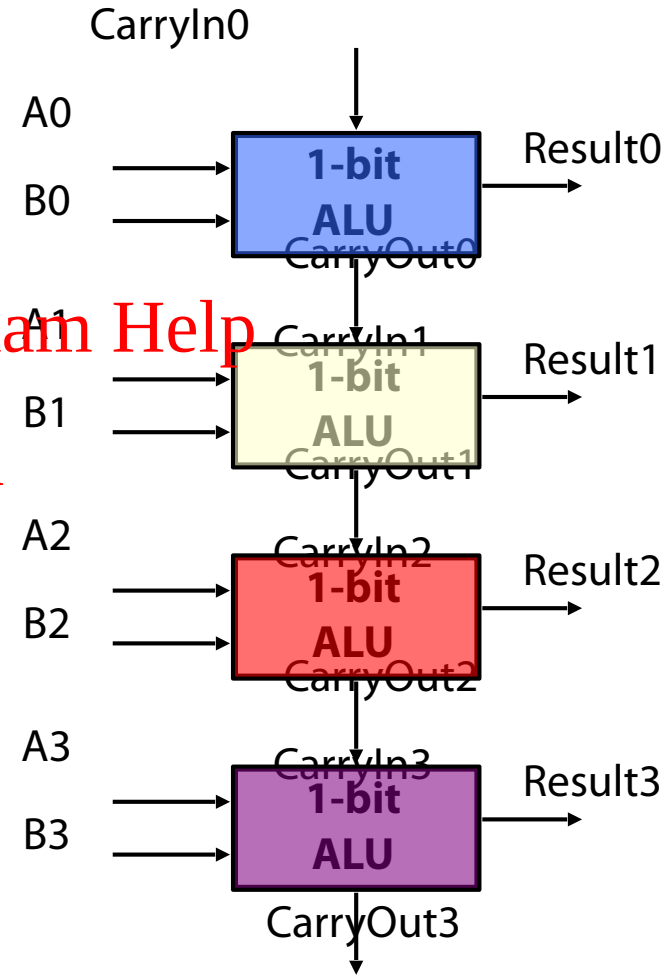


A	B	Cout	
0	0	0	"kill"
0	1	Cin	"propagate"
1	0	Cin	"propagate"
1	1	1	"generate"

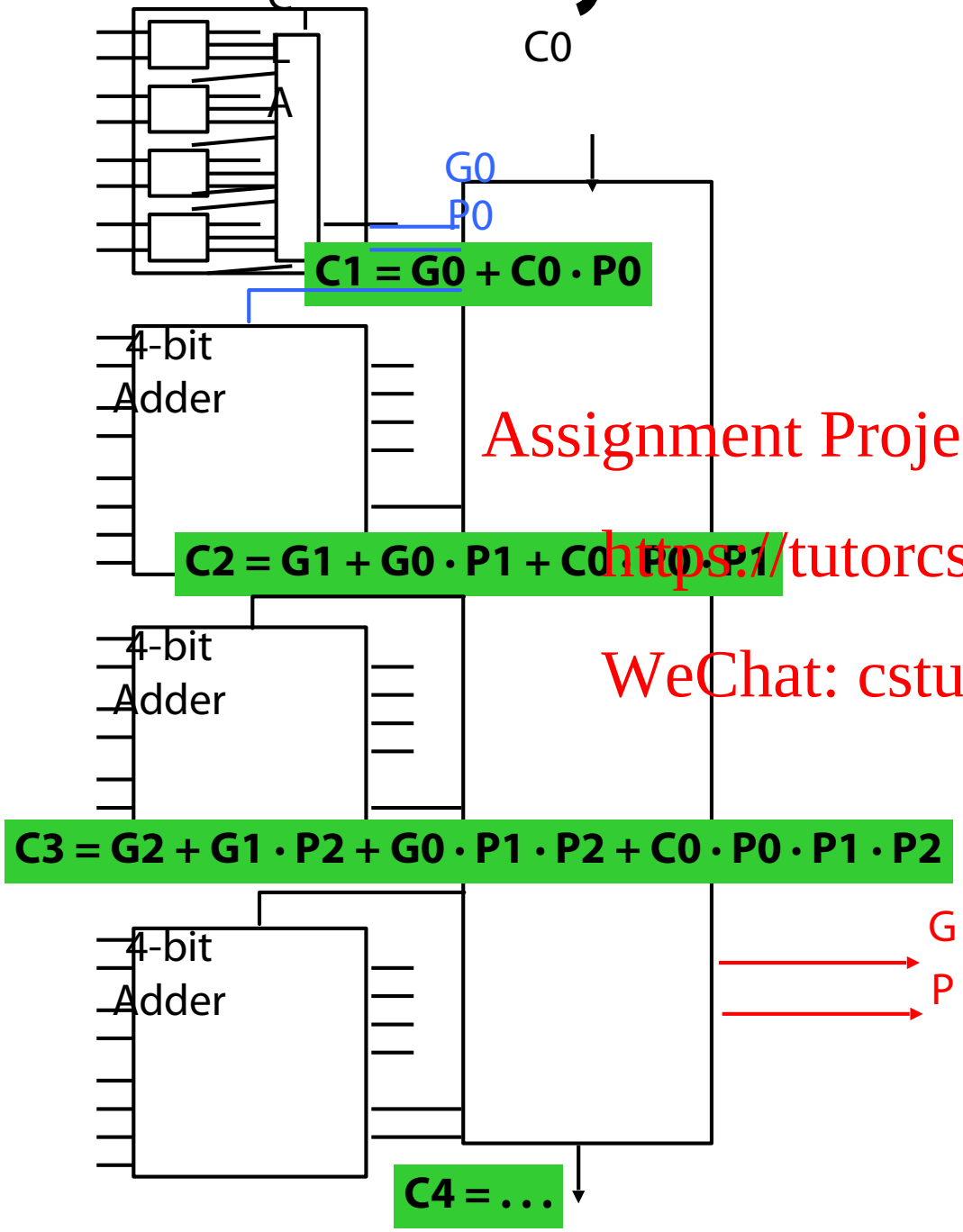
G = A and B
P = A xor B
WHY are these interesting?

CLA vs. Ripple





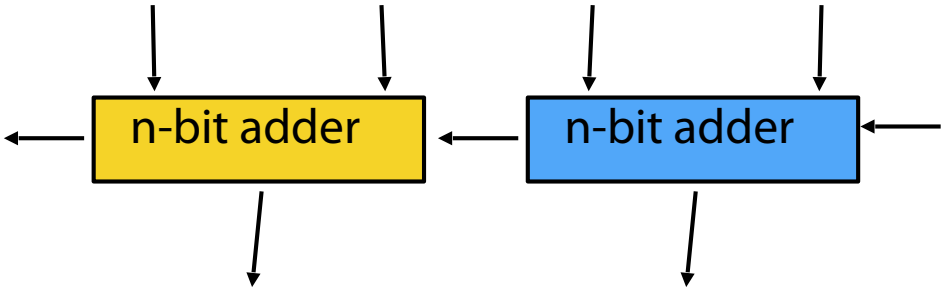
Cascaded Carry Look-ahead (16-bit)



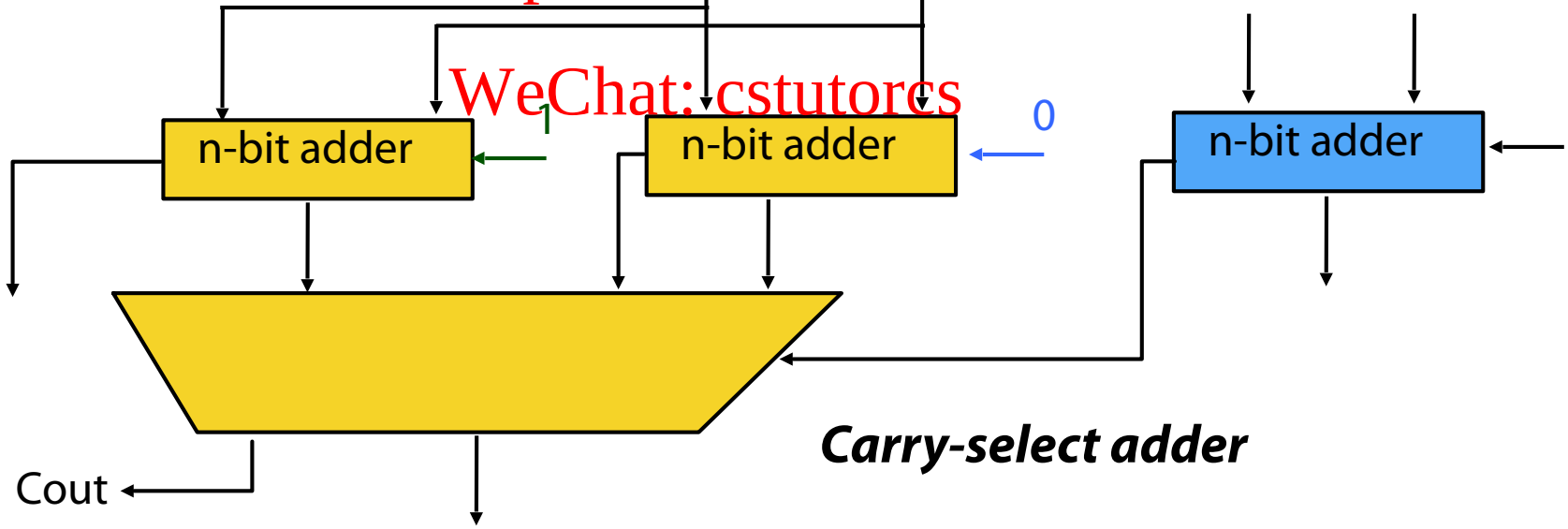
- Abstraction!
- Good exercise to work out what G and P are for a 4-bit adder

Design Trick: Guess (or "Precompute")

CP(2n) = 2*CP(n)

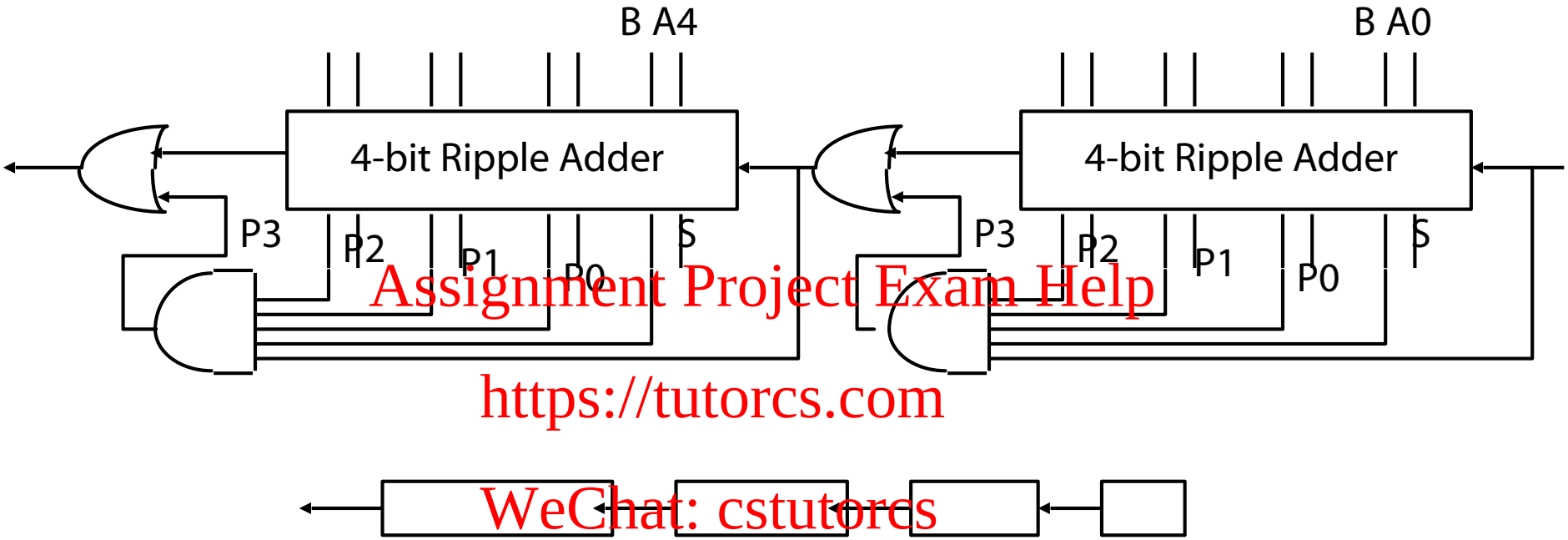


CP(2n) = CP(n) + CP(mux)



Carry-select adder

Carry Skip Adder: reduce worst case delay



- Just speed up the slowest case for each block

- Exercise: optimal design uses variable block sizes (why?)

Adder Lessons

- Reuse hardware if possible
  - +/- reuse is compelling argument for 2's complement
- For higher performance:
  - Look for critical path, optimize for it
    Reorganize equations
    [propagate/generate / carry lookahead]
  - Precompute [carry save]
  - Reduce worst-case delay [carry skip]

Multiply (unsigned)

- Paper and pencil example (unsigned):

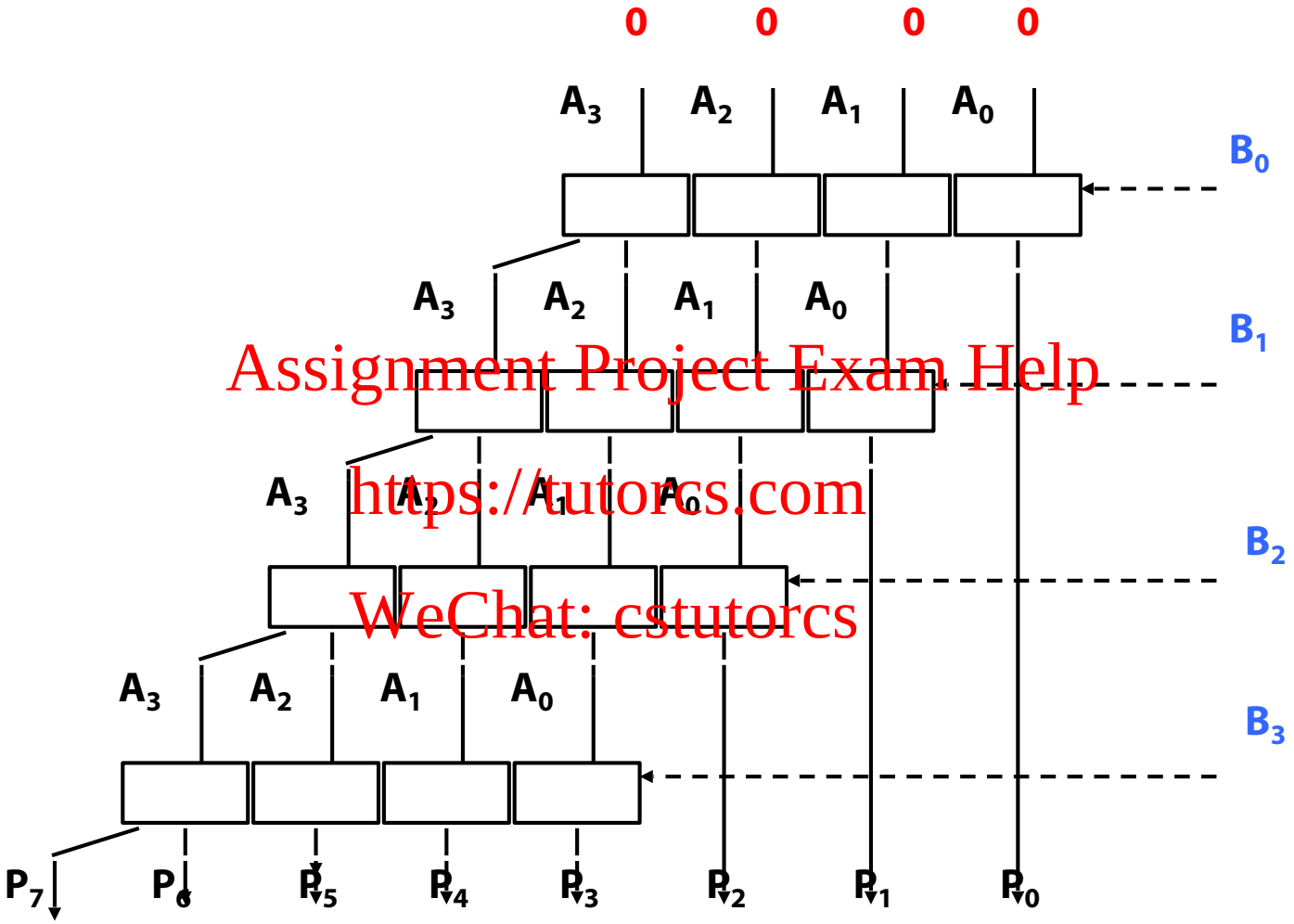
```
Multiplicand   1000
Multiplier   x 1001
               1000
              0000
             0000
            1000
Product    01001000
```


- 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 (see book):
  - successive refinement

m bits x n bits = $m+n$ bit product

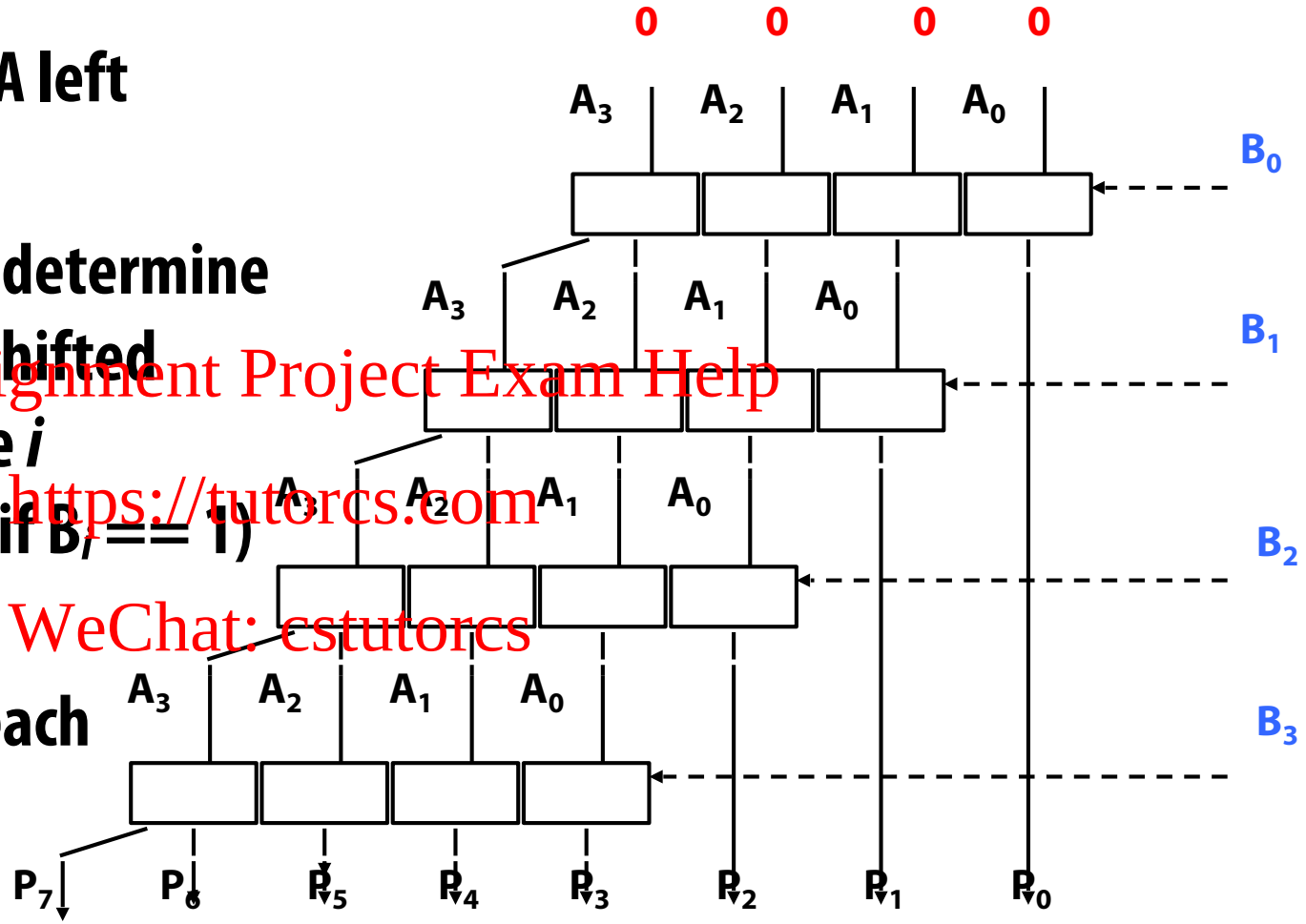
How do we store a 128b result from a 64b x 64b multiply?

Unsigned Combinational Multiplier



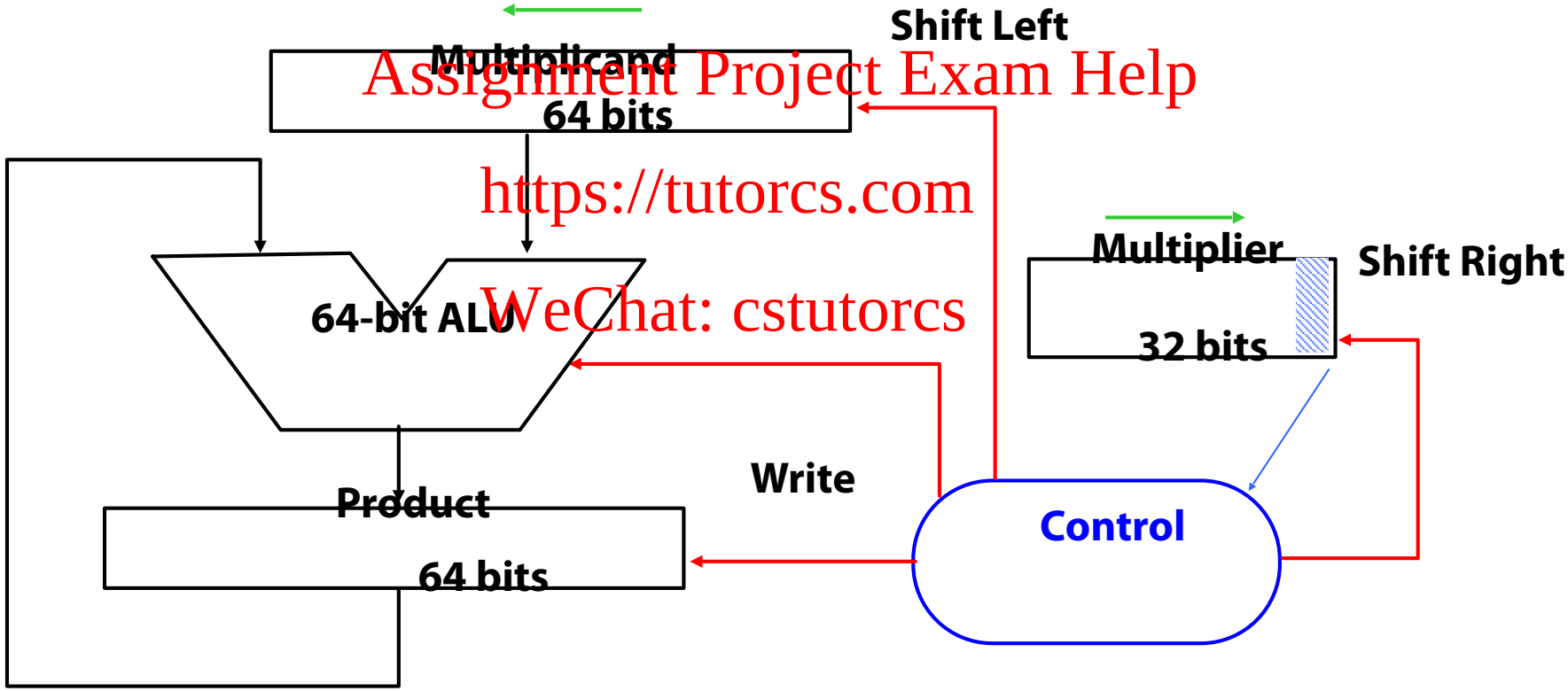
Unsigned Combinational Multiplier

- At each stage shift A left (multiply it by 2)
- Use next bit of B to determine whether to add in shifted multiplicand (Stage *i* accumulates A * 2*i* if $B_i == 1$)
- Accumulate 2n bit partial product at each stage
- Q: How much hardware for 32 bit multiplier? Critical path?



Unsigned shift-add multiplier (version 1)

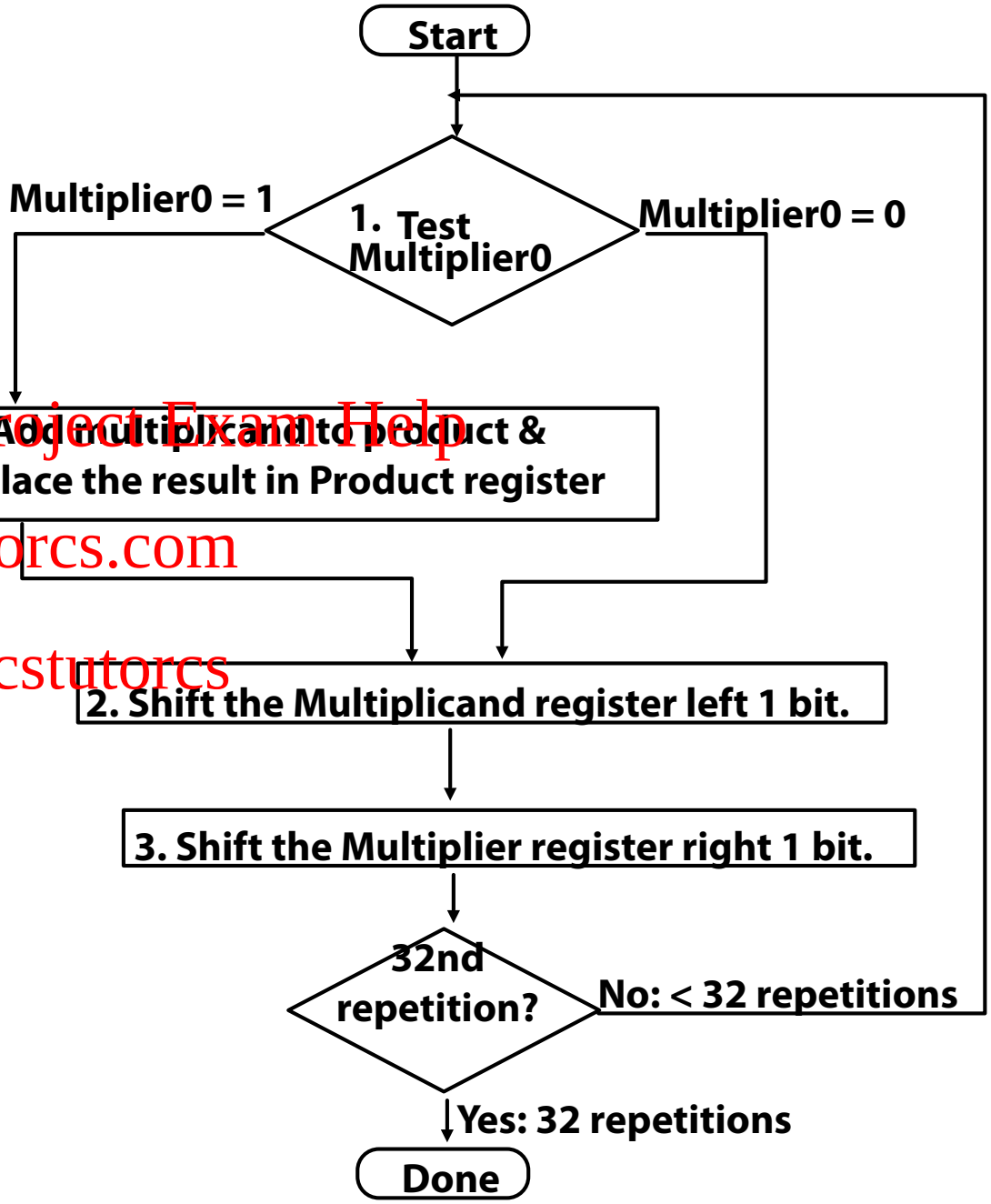
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

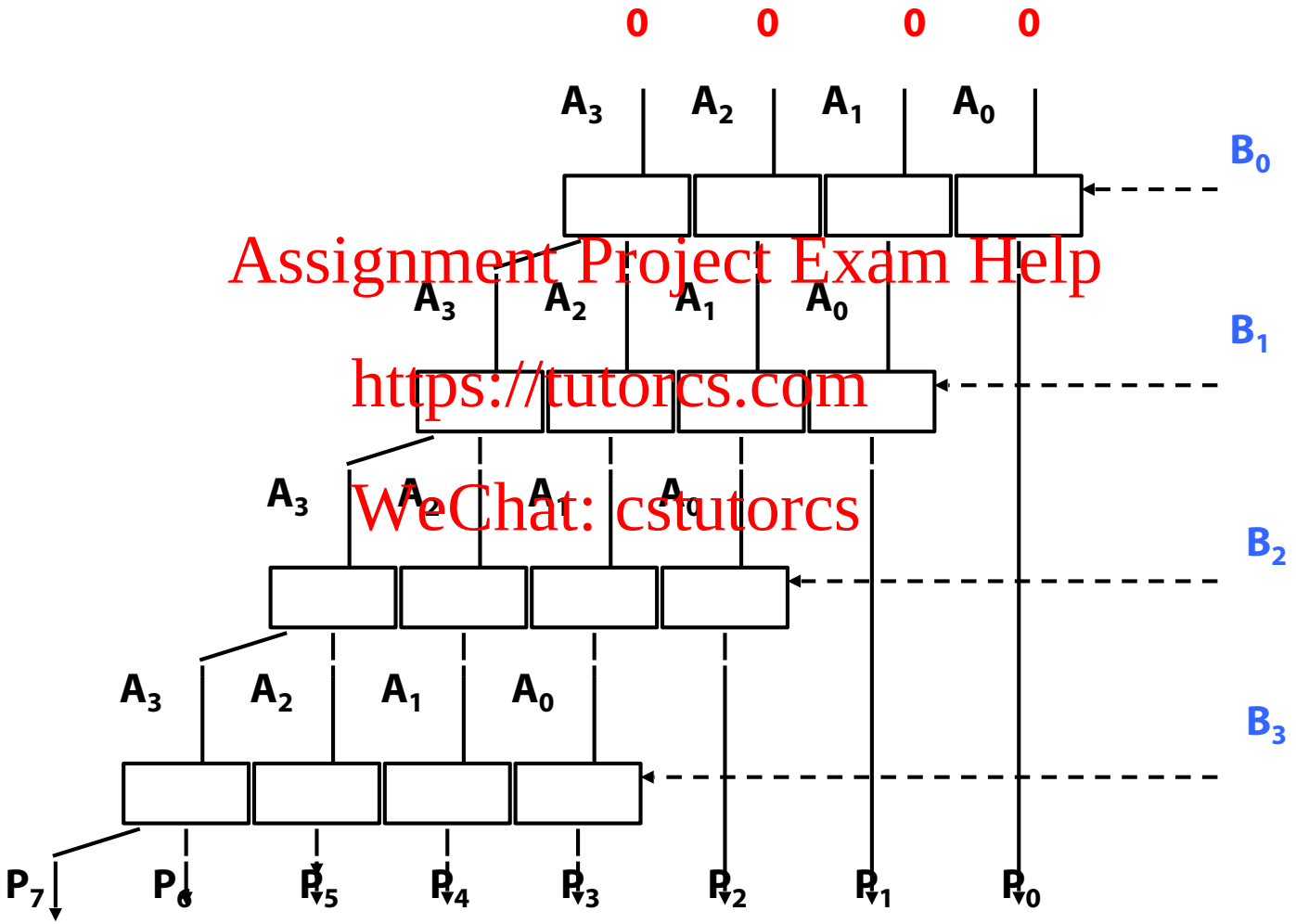
Multiply Algorithm V1

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

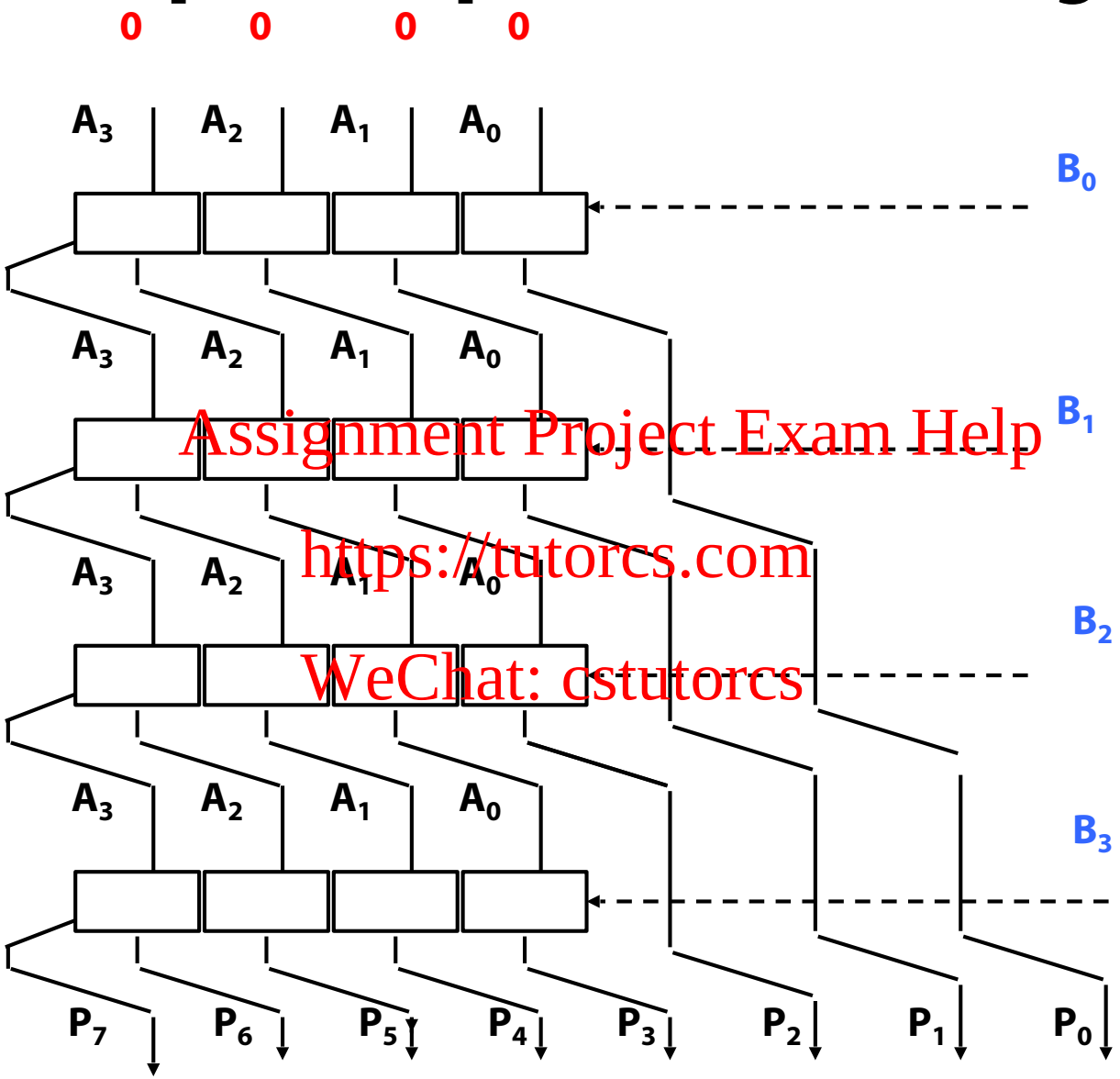


How can we make this more efficient?

- Remember original combinational multiplier:



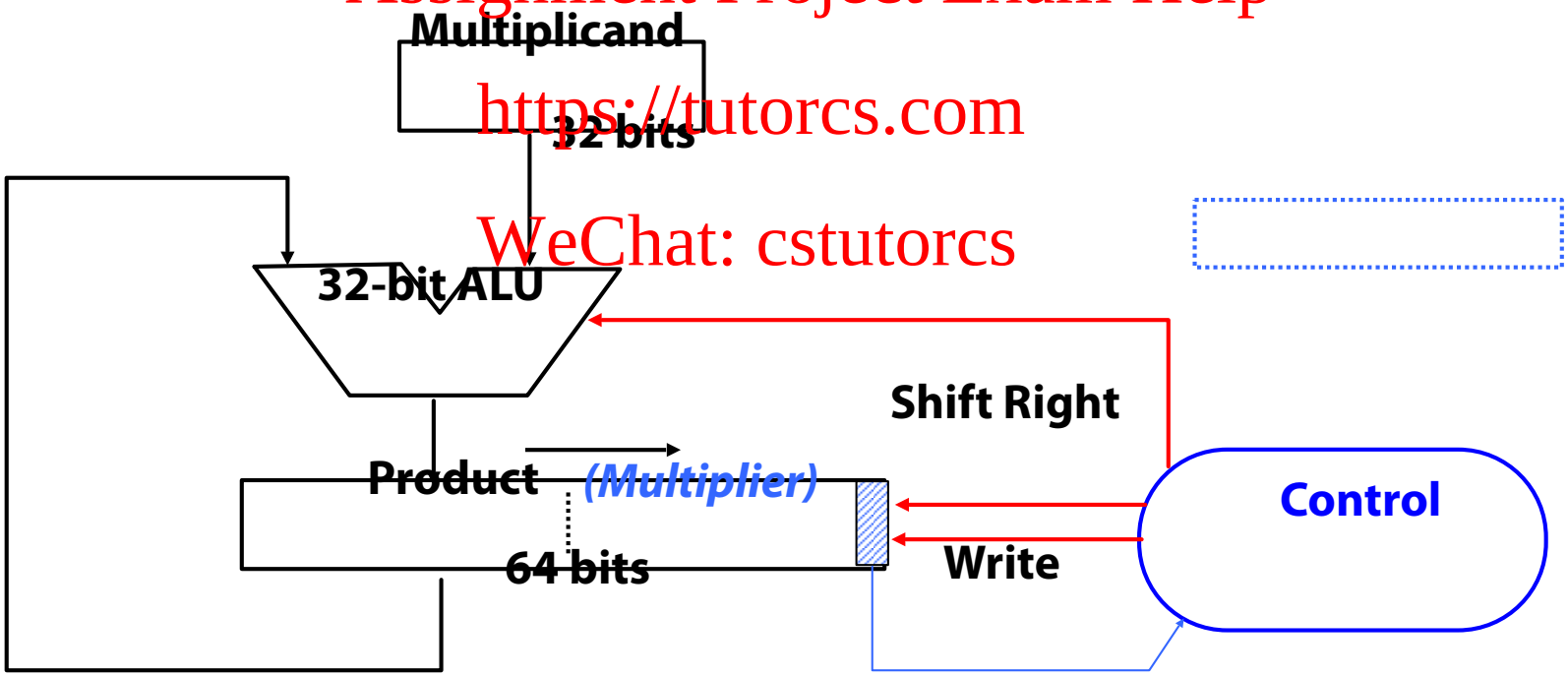
Simply warp to let product move right…



- Multiplicand stays still and product moves right

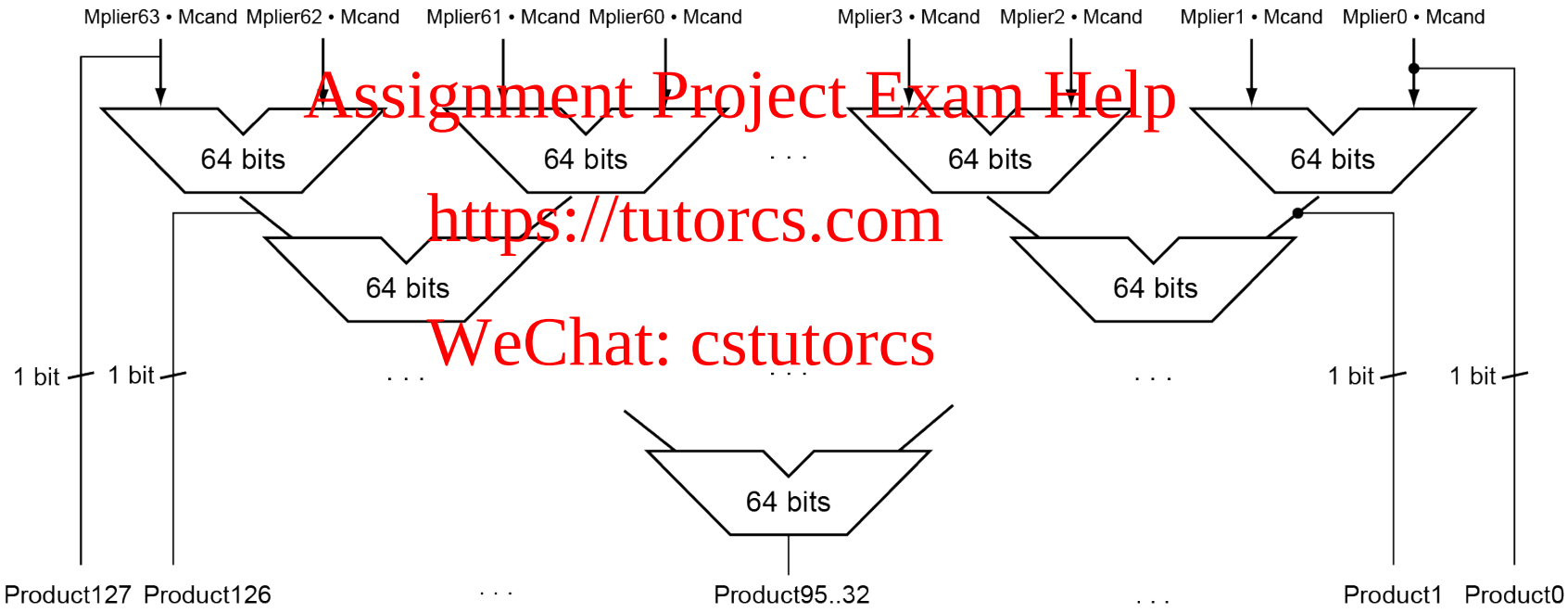
Multiply Hardware Version 3

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, (0-bit Multiplier reg)
  - Your book has details on this



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:

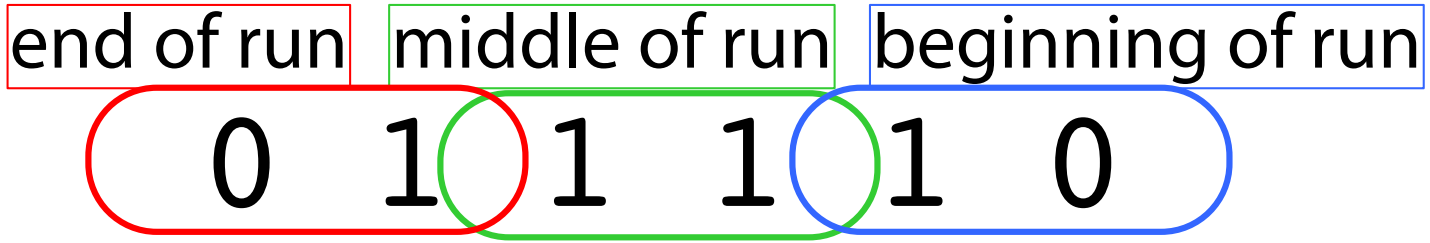
```
        0010
  x     0110
 ------------------------------------
  +     0000  shift (0 in multiplier)
  +    0010   add (1 in multiplier)
  +   0010    add (1 in multiplier)
  +  0000     shift (0 in multiplier)
 ------------
    0001100
```

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)

Big picture: Treat 7 as 8 - 1

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	1111 0110 1	01 -> add + 0010
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)

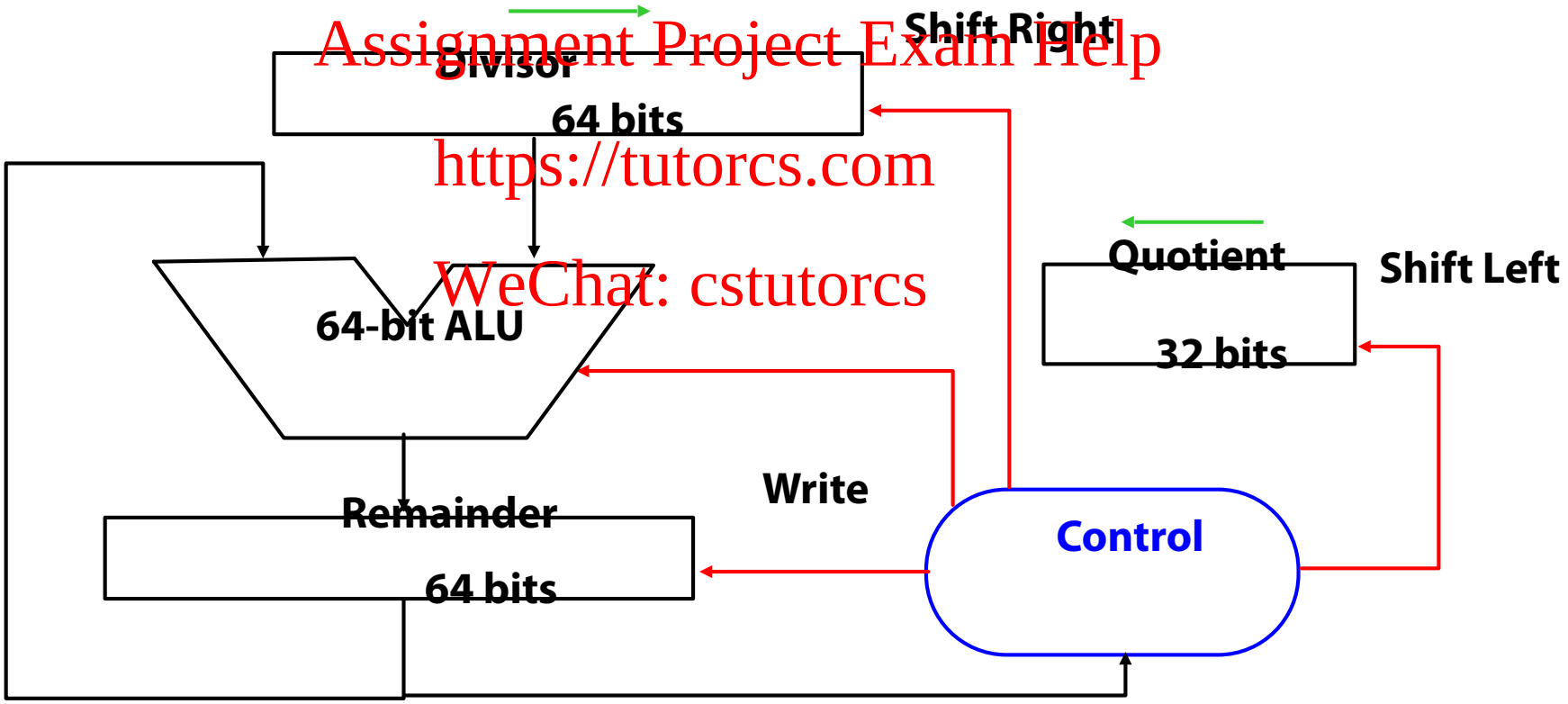
Divide: Paper & Pencil

```
                    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

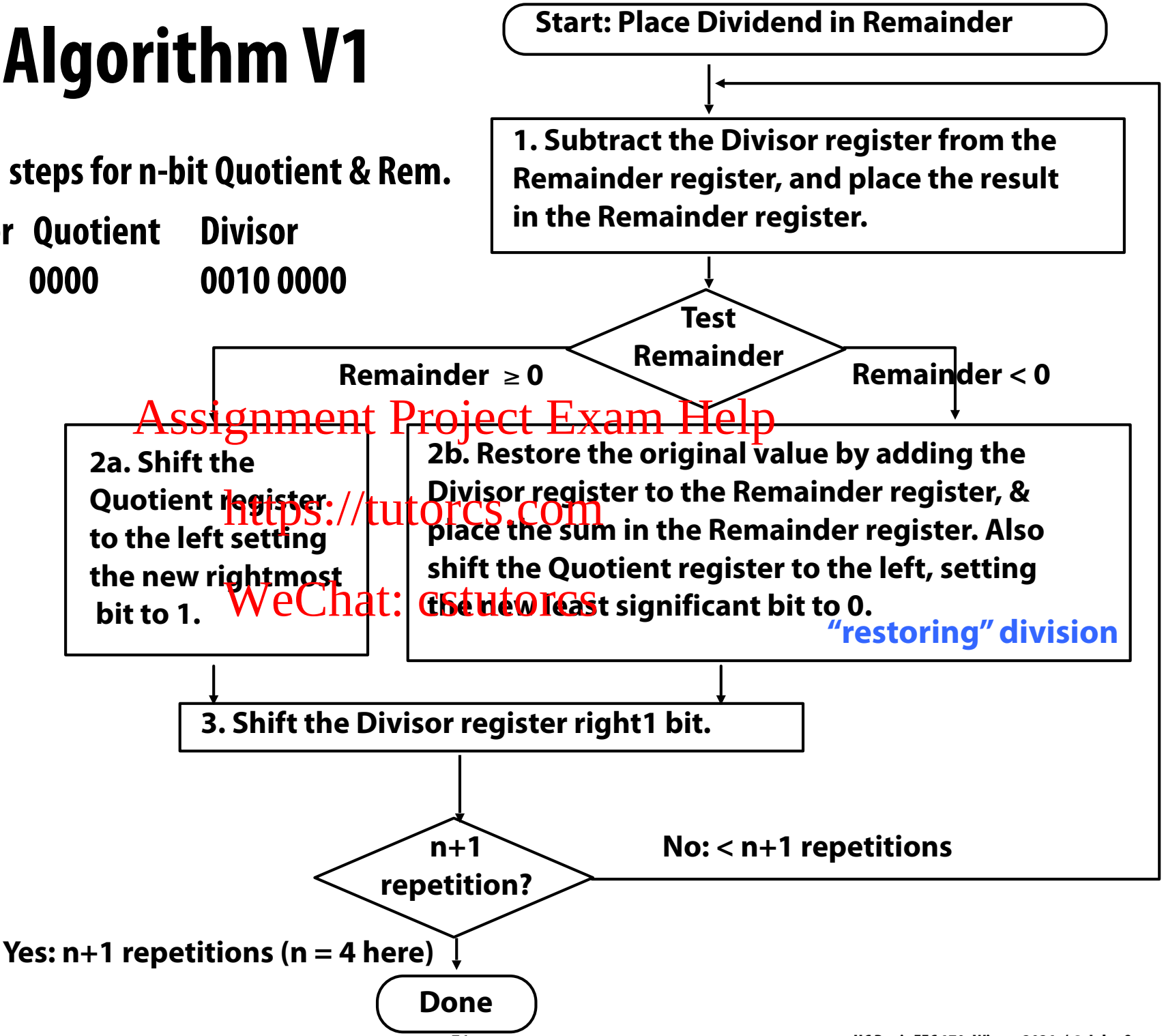
Divide Hardware Version 1

- 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



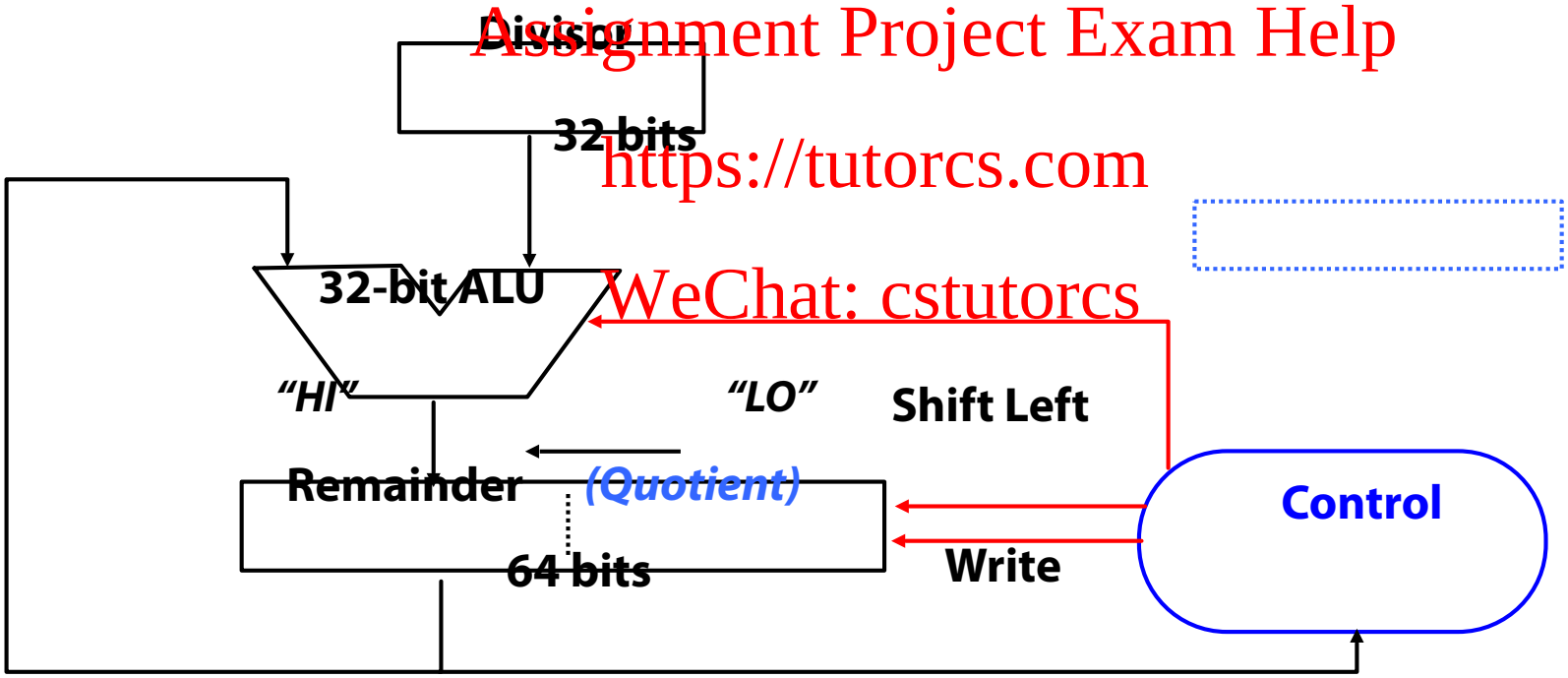
Divide Algorithm I example (7 / 2)

```
   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
```

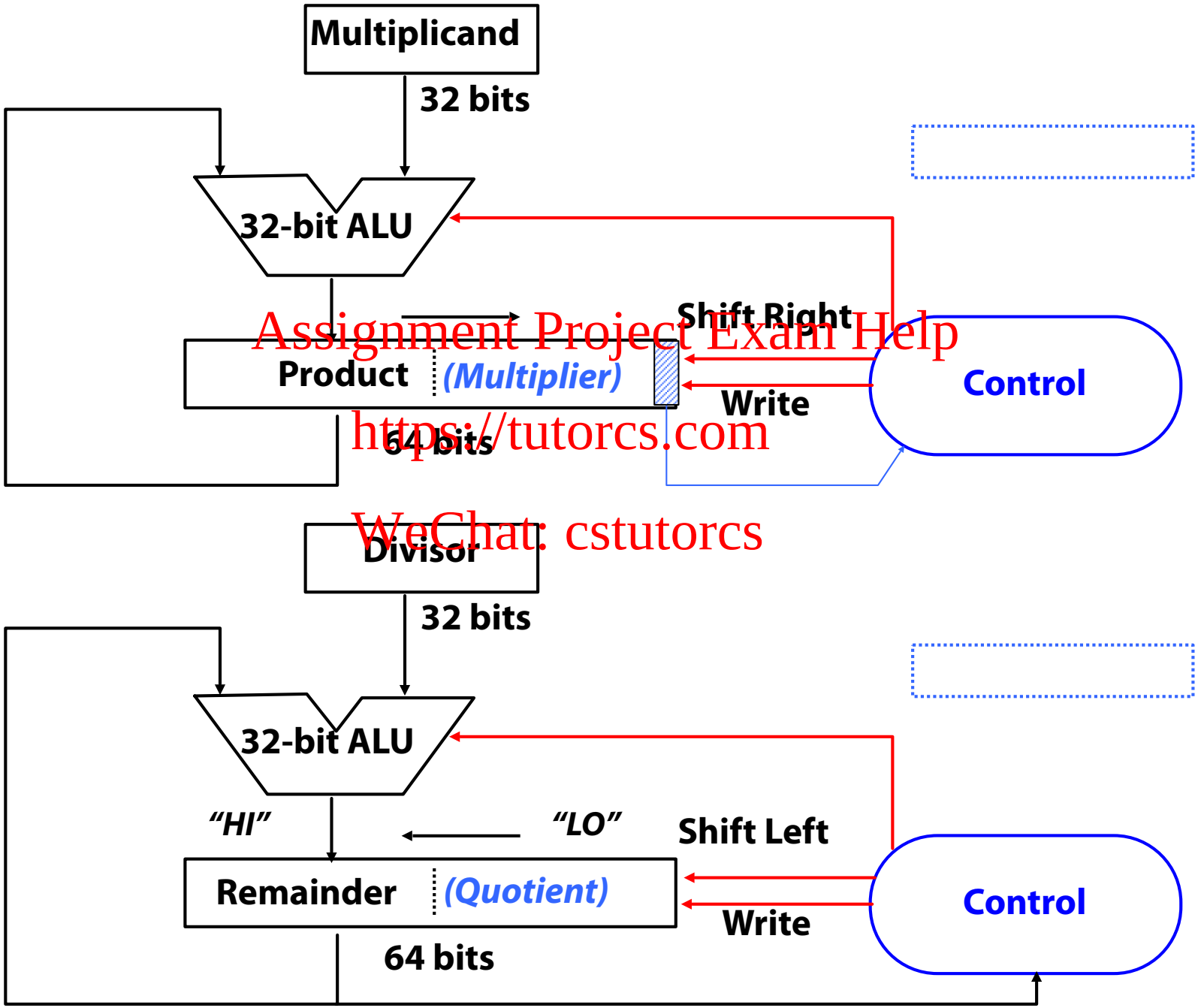
Answer:
Quotient = 3
Remainder = 1

Divide Hardware Version 3

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

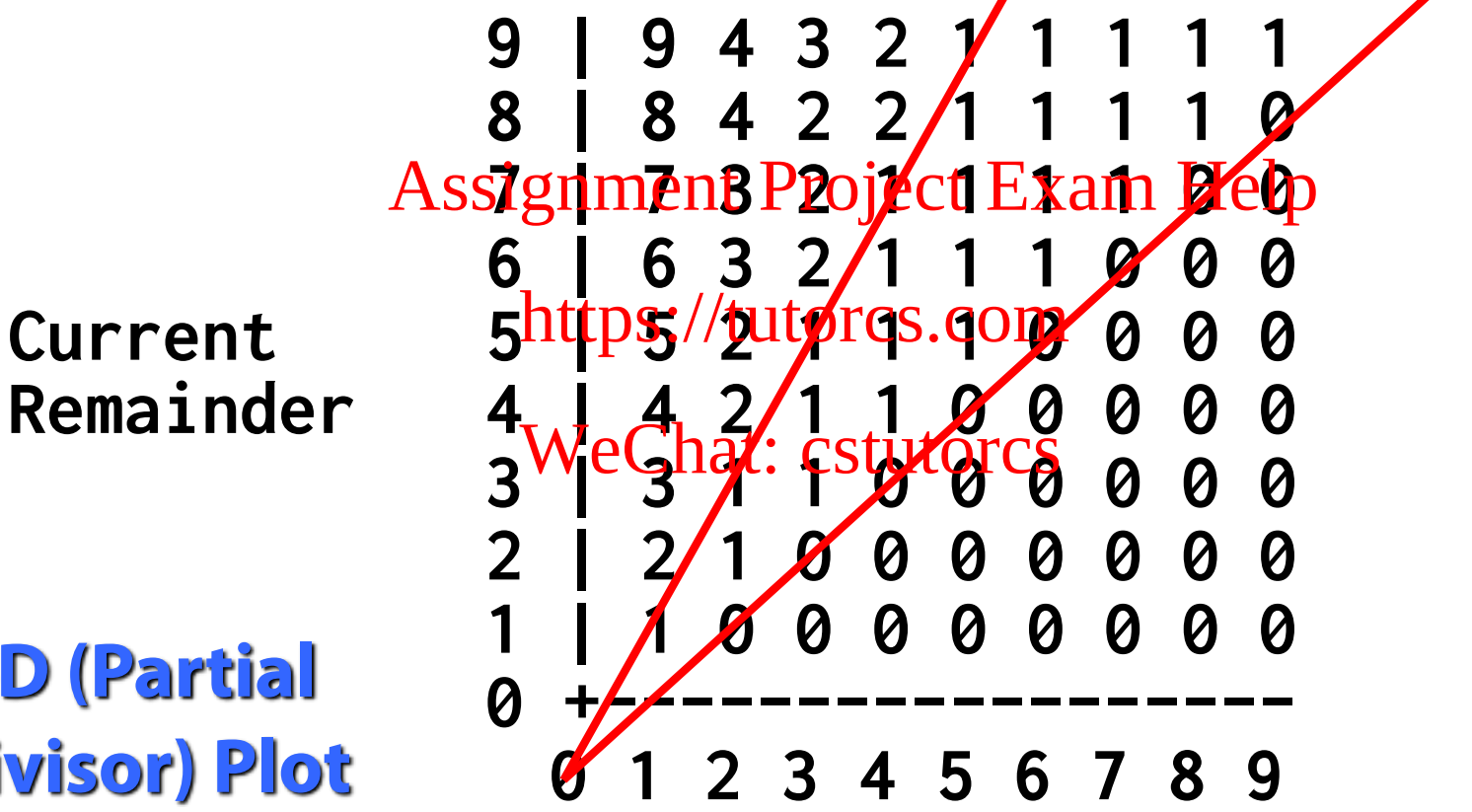


Final Multiply / Divide Hardware



SRT Division

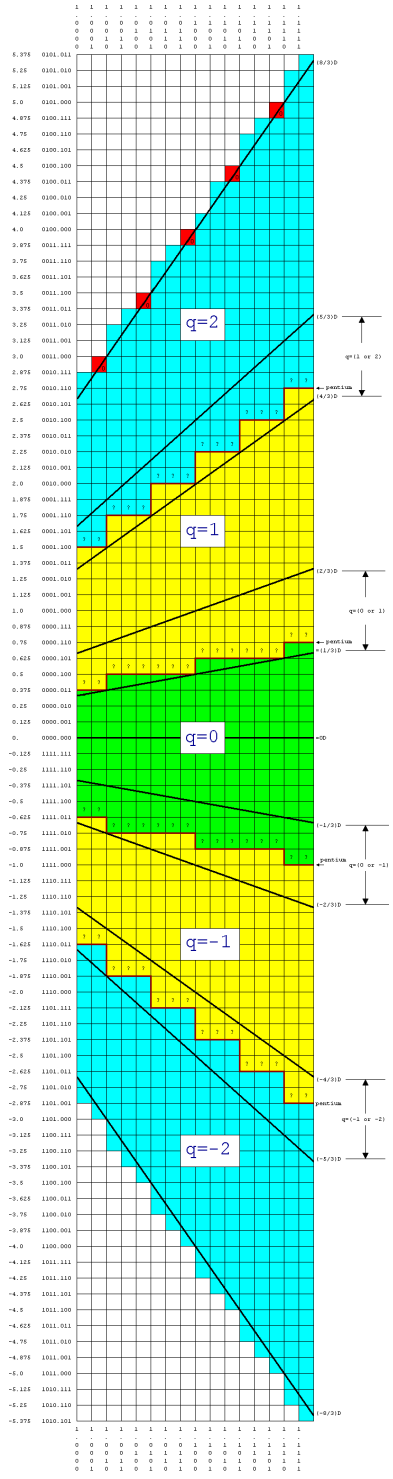
D. Sweeney of IBM, J.E. Robertson of the University of Illinois, and T.D. Tocher of Imperial College, London



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 division) 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)

Manipulating Bits with Shift, Or, And

- Mask out exponent with *and*

```
  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
```

- Zero out old exponent

```
  S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
& 1 00000000 11111111111111111111111
-----------------------------------
  S 00000000 MMMMMMMMMMMMMMMMMMMMMMM
```

- *Or* in new exponent

```
  S 00000000 MMMMMMMMMMMMMMMMMMMMMMM
| 0 ENEWENEW 00000000000000000000000
-----------------------------------
  S ENEWENEW MMMMMMMMMMMMMMMMMMMMMMM
```

Shift Operations

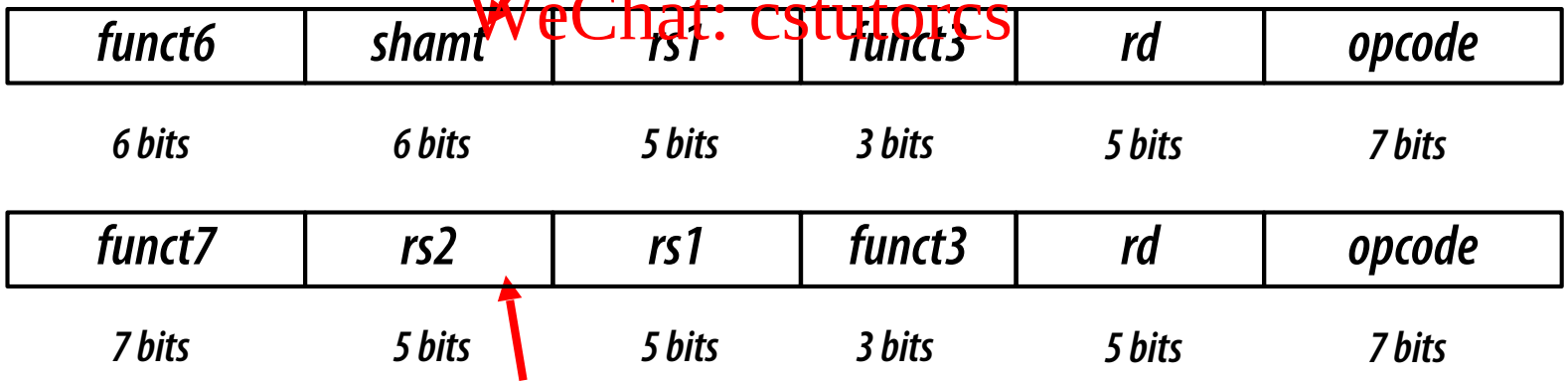
- Arithmetic operation:
  - Example: 00011 << 2 [3 left shift 2]
    - 00011 << 2 = 01100 = 12 = 2 * 4
  - Each bit shifted left == multiply by two
  - Example: 01010 >> 1 [10 right shift 1]
    - 01010 >> 1 = 00101 = 5 = 10/2
  - Each bit shifted right == divide by two
  - Why?
  - Compilers do this—"strength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

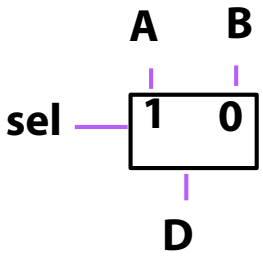
- How about right shift?
  - XXXX0000 >> 4 = 0000XXXX: Logical shift
    - Shifted in zero
  - 00110 (= 6) >> 1 = 00011 (3)
    11110 (= -2) >> 1 = 11111 (-1): Arithmetic shift
    - Shifted in sign bit
- RISC-V supports both logical and arithmetic:
  - slli, srai, srli: Shift amount taken from within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



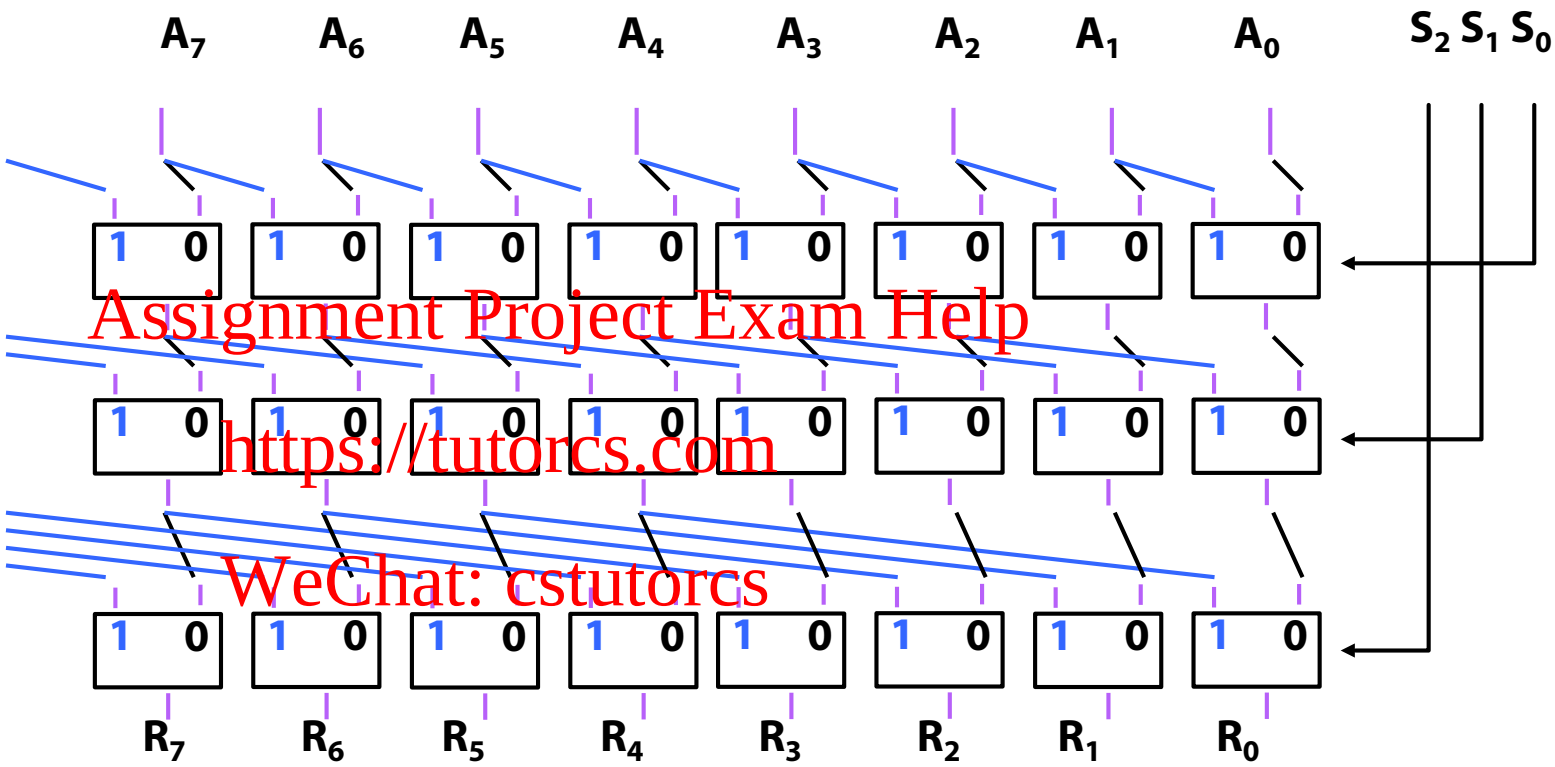
  - sll, sra, srl: shift amount taken from register ("variable")
  - How far can we shift with slli/srai/slli? With sll/sra/srl?

Combinational Shifter from MUXes

Basic Building Block

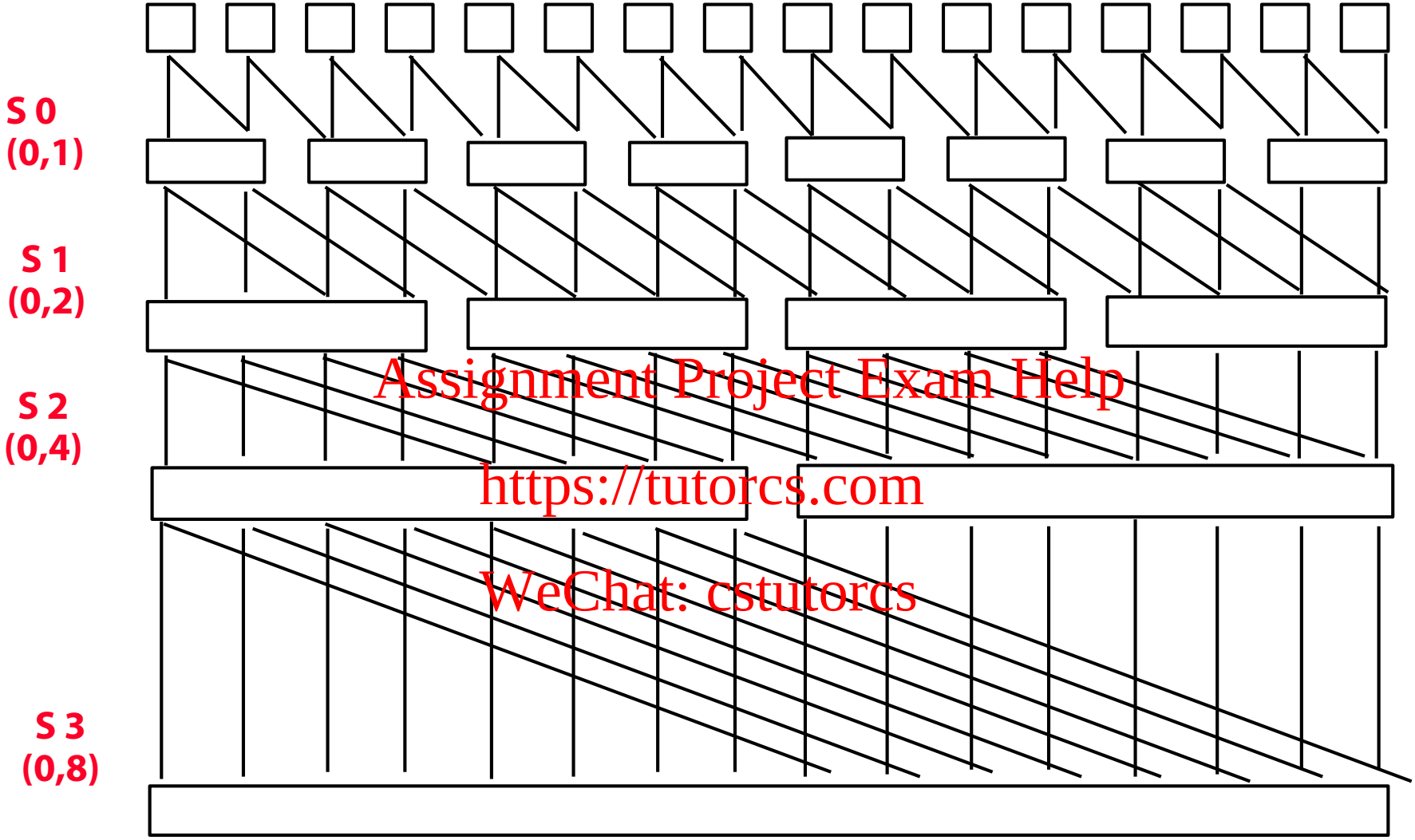


8-bit right shifter



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

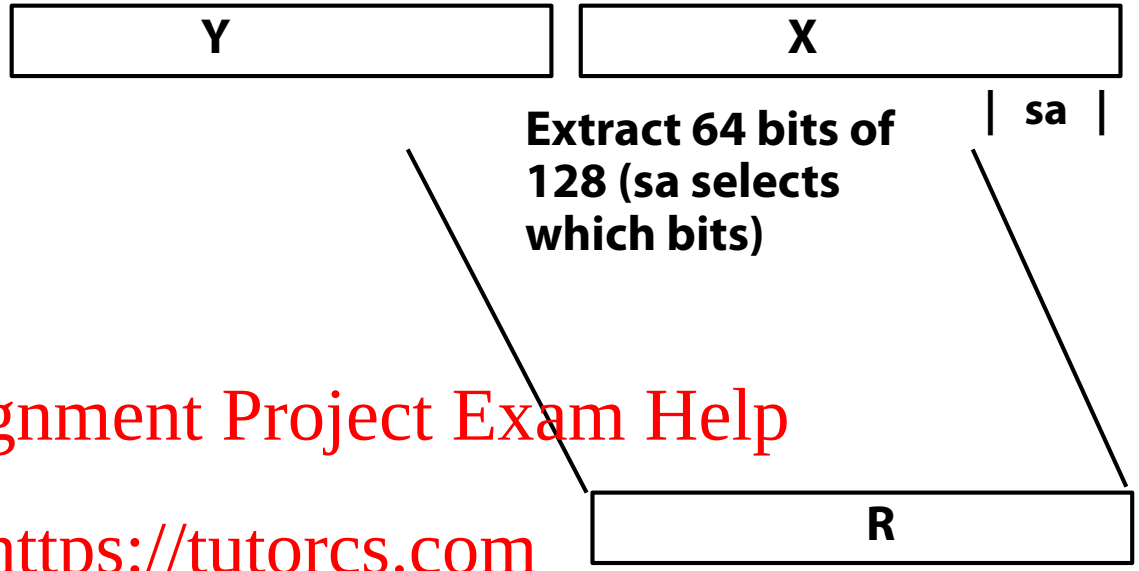
General Shift Right Scheme using 16 bit example

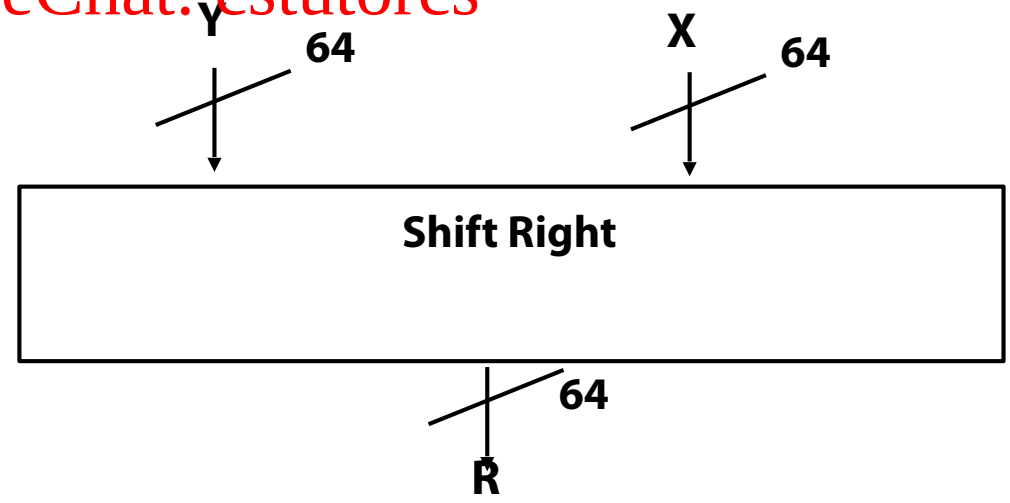


- If we added right-to-left connections, we could support ROTATE (not in RISC-V but found in other ISAs)

Funnel Shifter

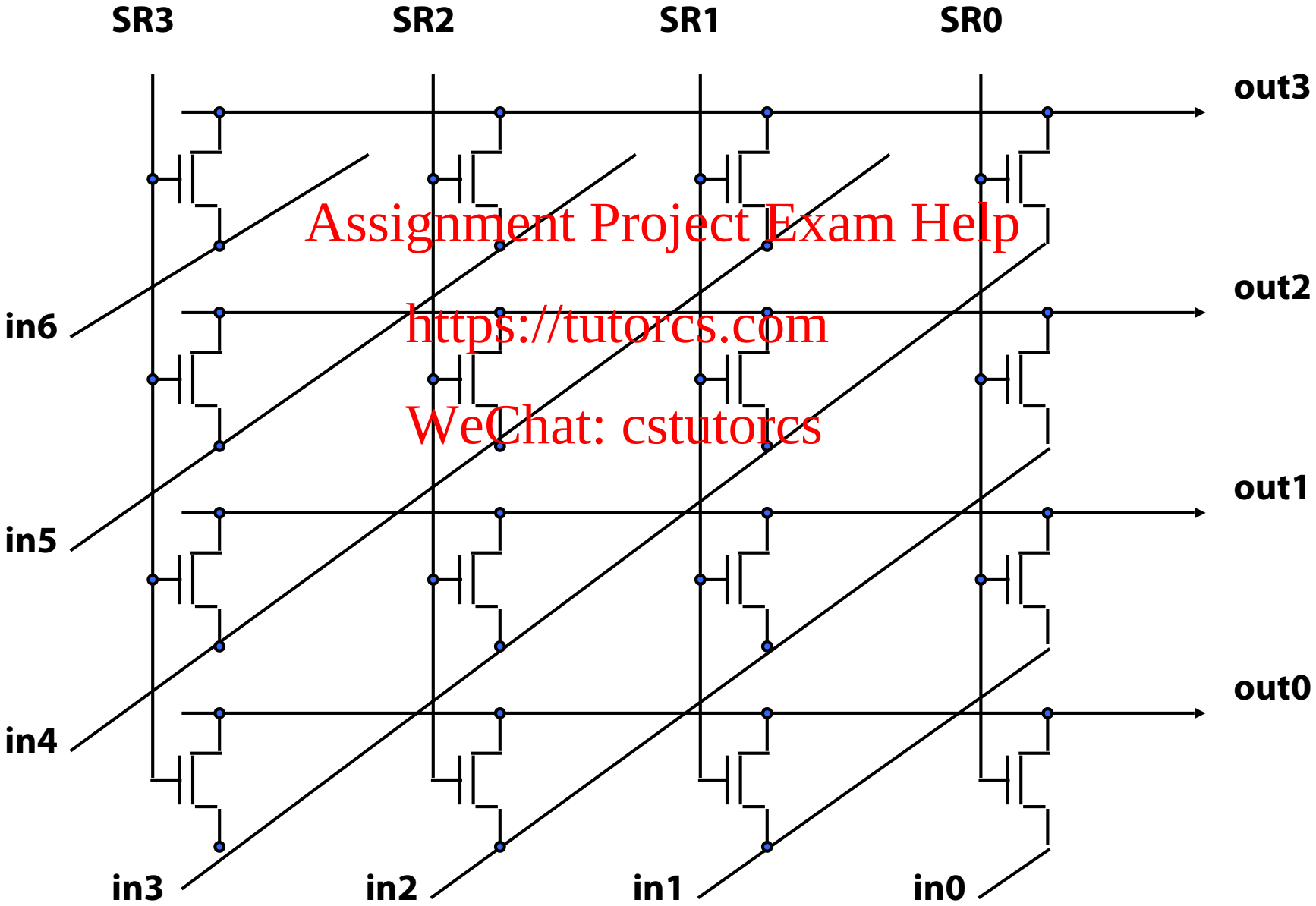
- Shift A by i bits
- Problem: Set Y, X, sa
- Logical:
- Arithmetic:
- Rotate:
- Left shifts:





Barrel Shifter

- Technology-dependent solutions: transistor per switch



Shifter Summary

- Shifts common in logical ops, also in arithmetic
- RISC-V 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

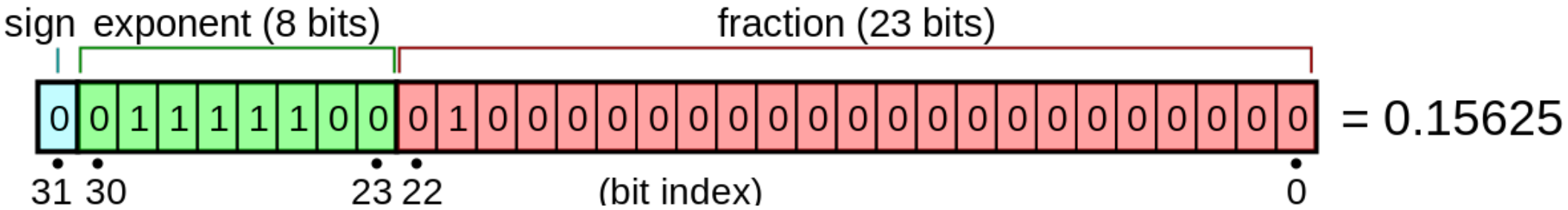
Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} *normalized*
  - $+0.002 \times 10^{-4}$ *not normalized*
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

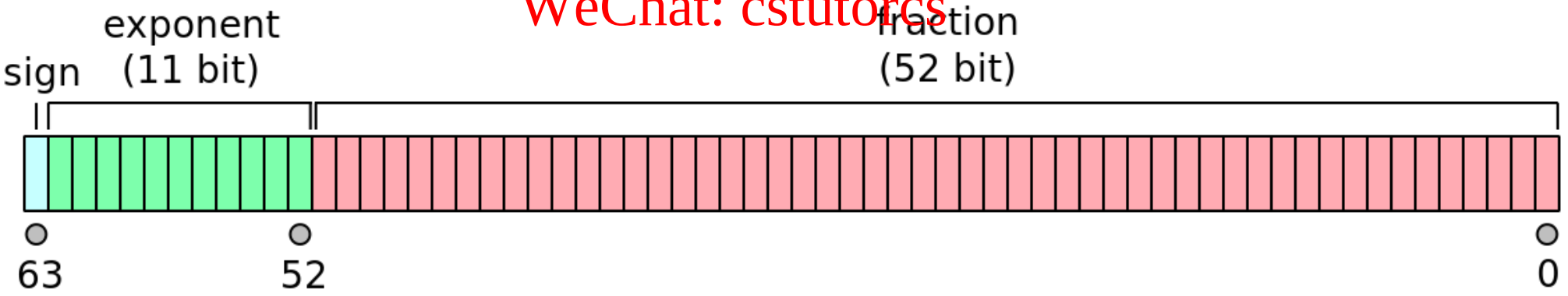
Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

Floating-point Formats



- **Single-precision (32 bits)**



- **Double precision (64 bits)**

IEEE Floating-Point Format

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Fraction:
single: 23 bits
double: 52 bits

Exponent:
single: 8 bits
double: 11 bits

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \le |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent $-0.75 = -(0.5 + 0.25) = -(1.0 + 0.5) / 2$
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = −1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 10111111010000…00
- Double: 10111111111010000…00

Floating-Point Example

- What number is represented by the single-precision float

  1 10000001 01000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$

- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Half-precision Floating Point