Lecture 5:

Arithmetic 1/3

John Owens

Introduction to Computer Architecture

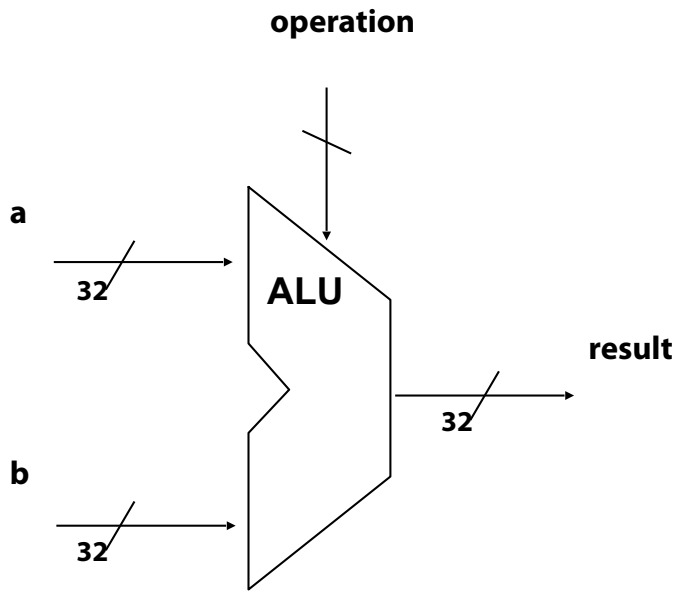
UC Davis EEC 170, Winter 2021

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    - Instruction Set Architecture
    - Assembly Language and Machine Language
- What's up ahead:
  - Number Representation
  - Implementing an ALU
  - Implementing the Architecture



32 bit signed numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2,147,483,646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2,147,483,647_{ten}$

maxint

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2,147,483,648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2,147,483,646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

minint

What happens if you compute –minint?

Two's Complement Operations

- Negating a two's complement number:
  invert all bits and add 1
  - remember: "negate" and "invert" are quite different!
- Converting *n* bit numbers into numbers with more than *n* bits:
  - RISC V *n* bit immediate gets converted to 64 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - "sign extension" (lbu vs. lb)

```
-   mem [x1] = 0xff
-   lb x2, 0(x1) -> x2 = ffff ffff
-   lbu x2, 0(x1) -> x2 = 0000 00ff
```

Addition & Subtraction

- Two's complement operations easy
  - subtraction using addition of negative numbers

```
  0111  (7)          0111  (7)
+ 1010  (-6)       -  0110  (6)
```

Addition & Subtraction

- Overflow (result too large for finite computer word):
  - adding two n-bit numbers does not yield an n-bit number

```
   1111
+  0001
  10000
```

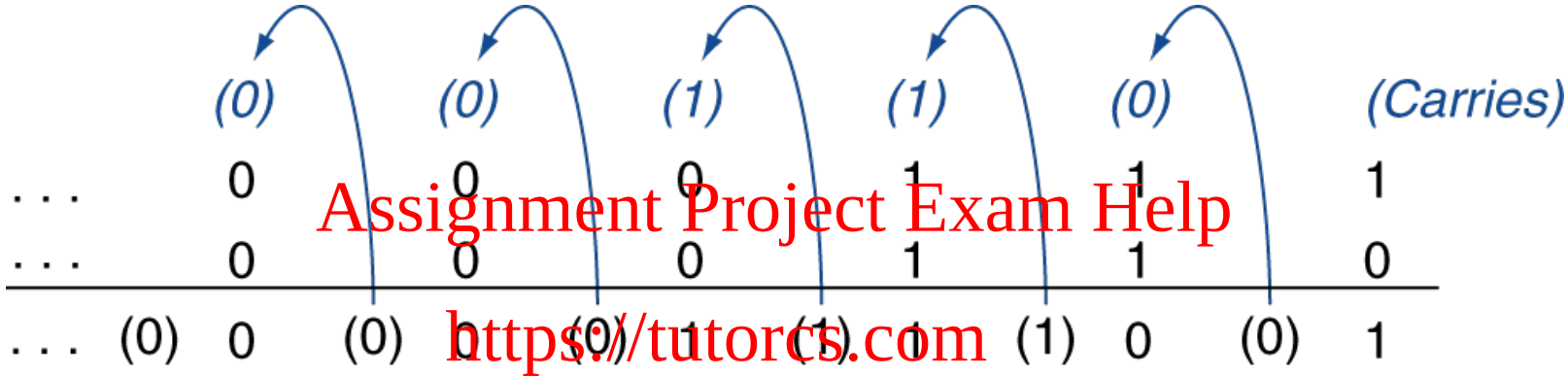
  - How about -1 + -1?

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?
- HW problem on figuring out exact criteria

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

  +7: 0000 0000 … 0000 0111
  –6: 1111 1111 … 1111 1010
  +1: 0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
  - C/Java do not detect overflow
  - Fortran (evidently) can detect overflow
- Don't always want to detect overflow
  - MIPS instructions (but *not* RISC-V): `addu`, `addiu`, `subu`
    - RISC-V advocates branches on overflow instead
  - `addiu` sign-extends
  - `sltu`, `sltiu` for unsigned comparisons

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    (this probably doesn't mean anything to you yet, but wait until the end of lecture)
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Review: Boolean Algebra & Gates

- Problem: Consider a logic function with three inputs: A, B, and C.

- Output D is true if at least one input is true
- Output E is true if exactly two inputs are true
- Output F is true only if all three inputs are true

Review: Boolean Algebra & Gates

- Show the Boolean equations for these three functions.

Review: Boolean Algebra & Gates

- Show an implementation consisting of inverters, AND, and OR gates.

You should know …

- Boolean algebra
- Logic gates (and, or, not, xor, nor, multiplexors [muxes], decoders, etc.)
- Converting between equations, truth tables, and gate representations
- Critical path
- Clocking, and registers / memory
- Finite state machines

- … COD5e Appendix A summarizes this material

Break

Administrivia

- We have a TA! I already put him to work.
  Trivikram Reddy
- By popular demand, I will have an office hour Fri 18 October, "when the train arrives" (9:30)–11 at the CoHo

Bit Slices

- Concentrate on one bit of the adder:
- 

```
  0111
+ 0110
```


- Could we build the same hardware for every bit?
  - This is a good idea. Why?
  - Each bit's hardware is called a "bit slice"

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
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

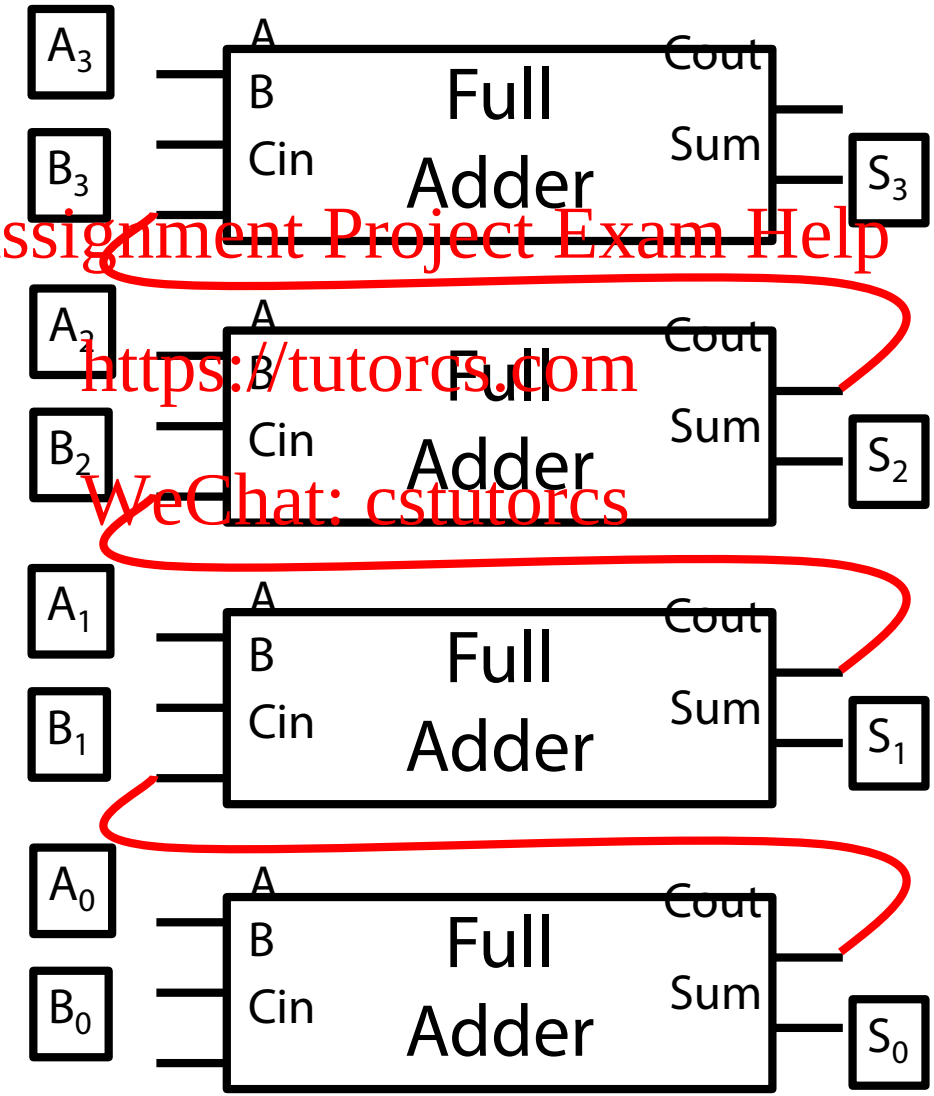
Adder Equations

- Sum = (A ⊕ B) ⊕ Cin
- Carry = AB + ACin + BCin

- Abstract as "Full Adder":

Cascading Adders

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



Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    (this probably doesn't mean anything to you yet, but wait until the end of lecture)
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

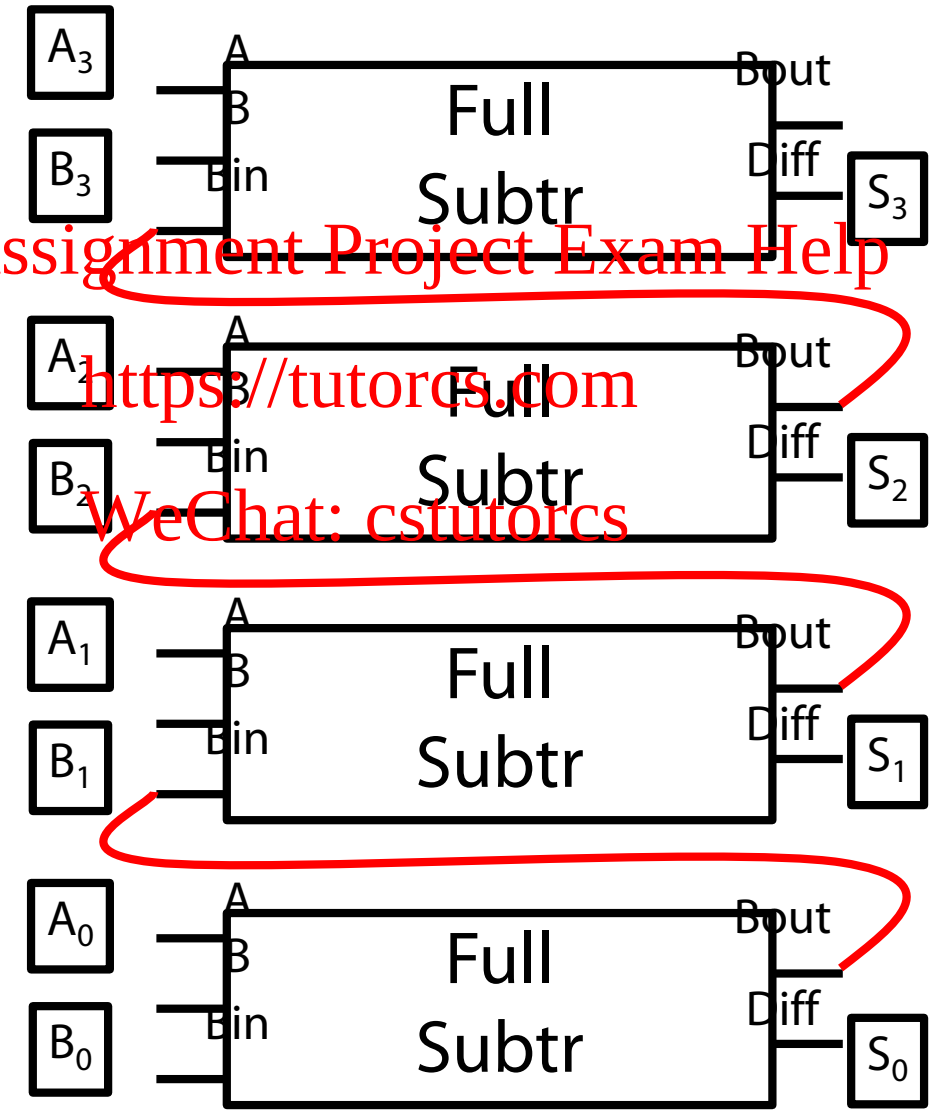
Truth Table for Subtractor Bit Slice

- 3 inputs (A, B, Bin); 2 outputs (Diff, Bout)

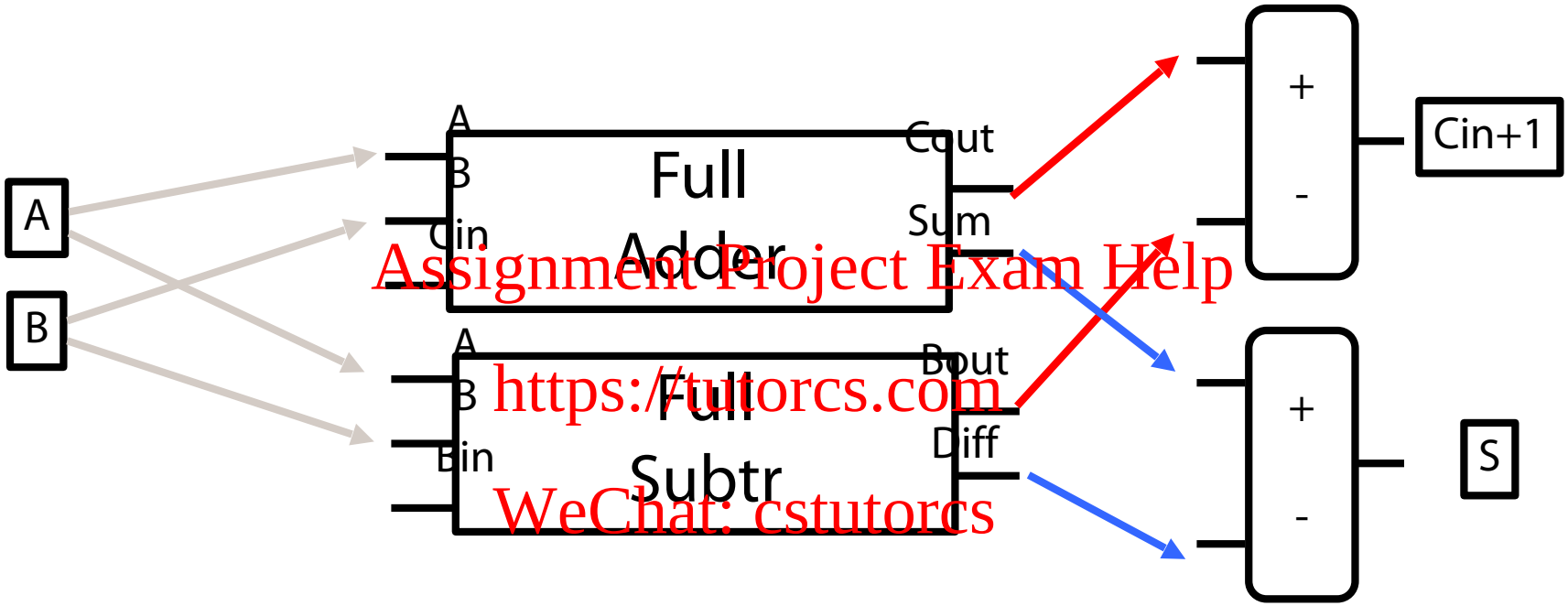
A	B	Bin	Diff	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Cascading Subtractors

- Cascade Full Subtrs to make multibit subtr:
- A-B=S



How can we combine + and -?

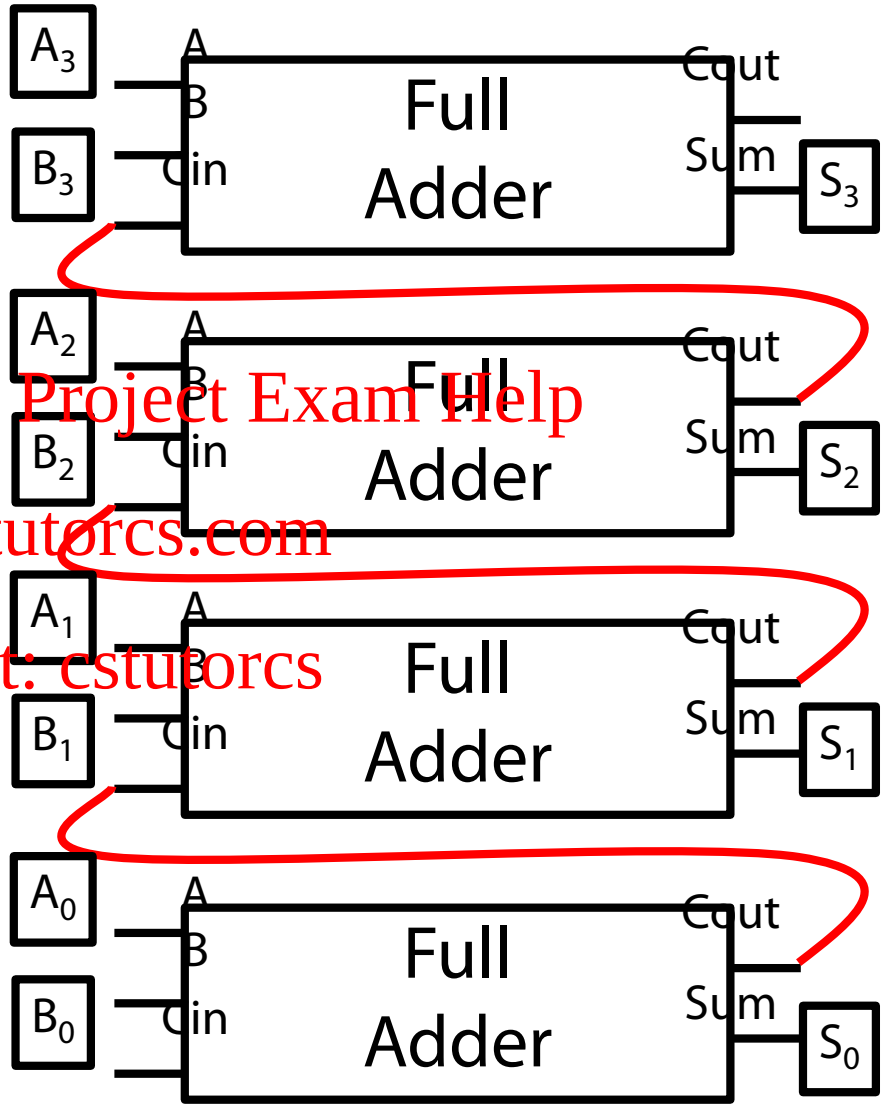


- This is common—it's what we'll do (for example) for logic functions (and, or, etc.)

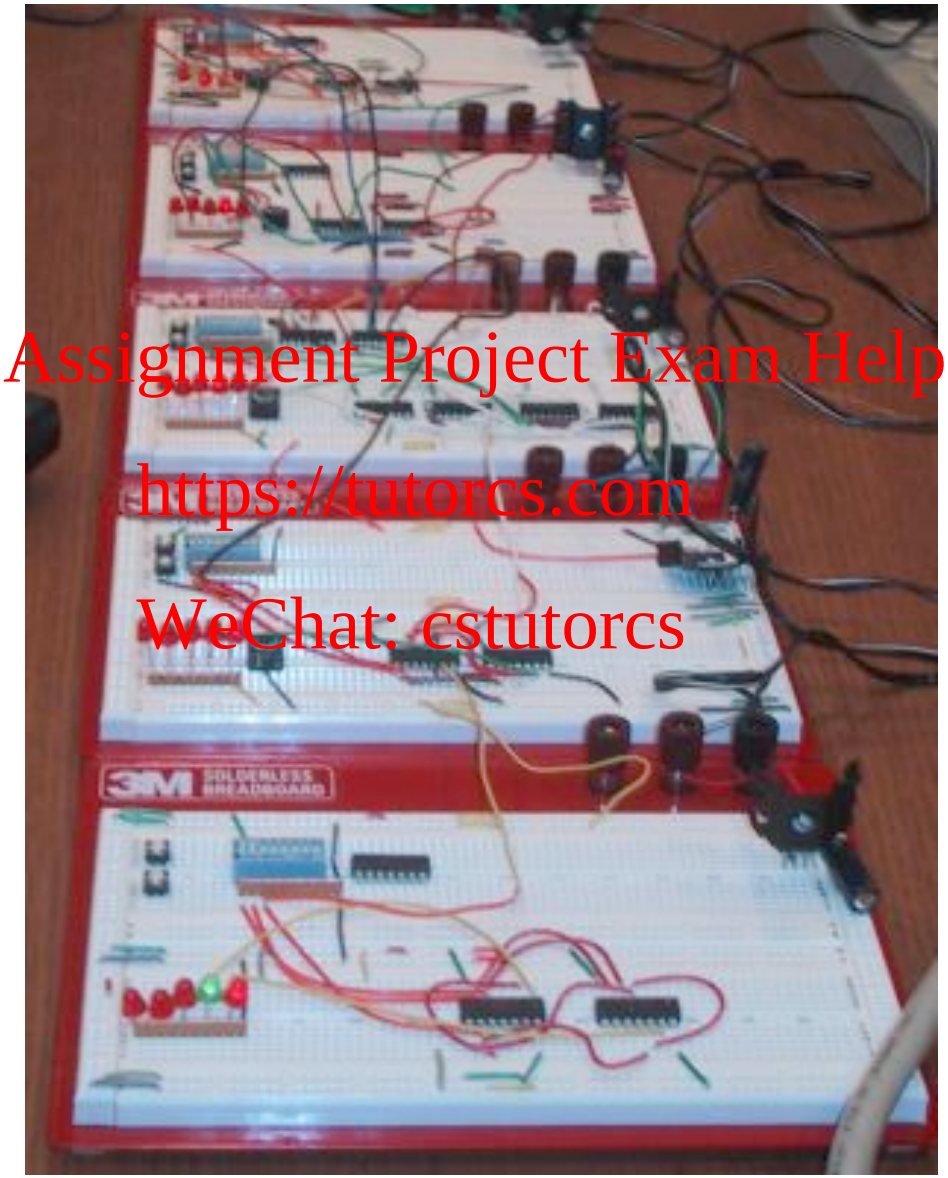
How can we combine + and -?

$S = A - B$

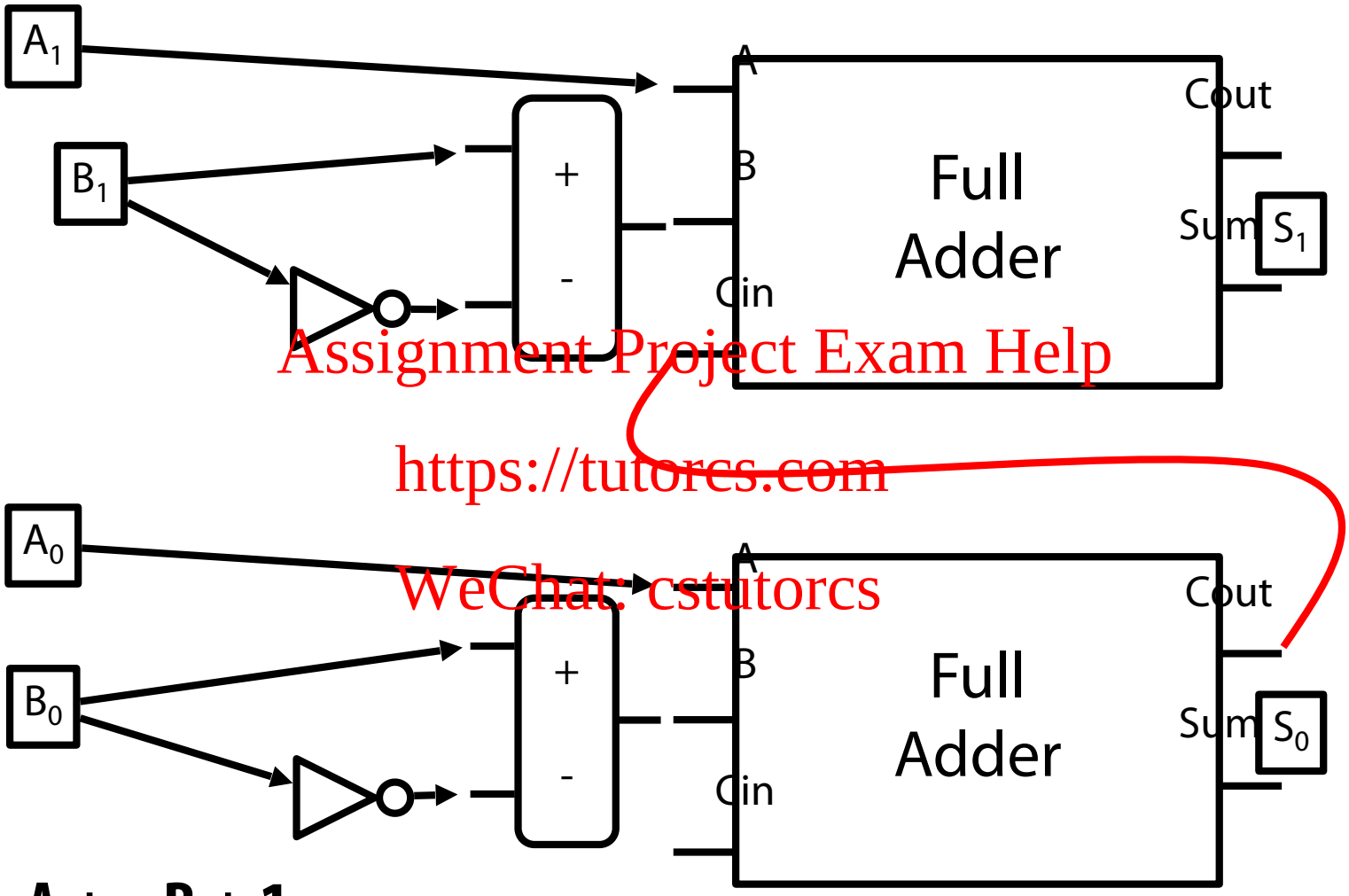
$S = A + (\sim B) + 1$



Lest you think this is only theoretical . . .



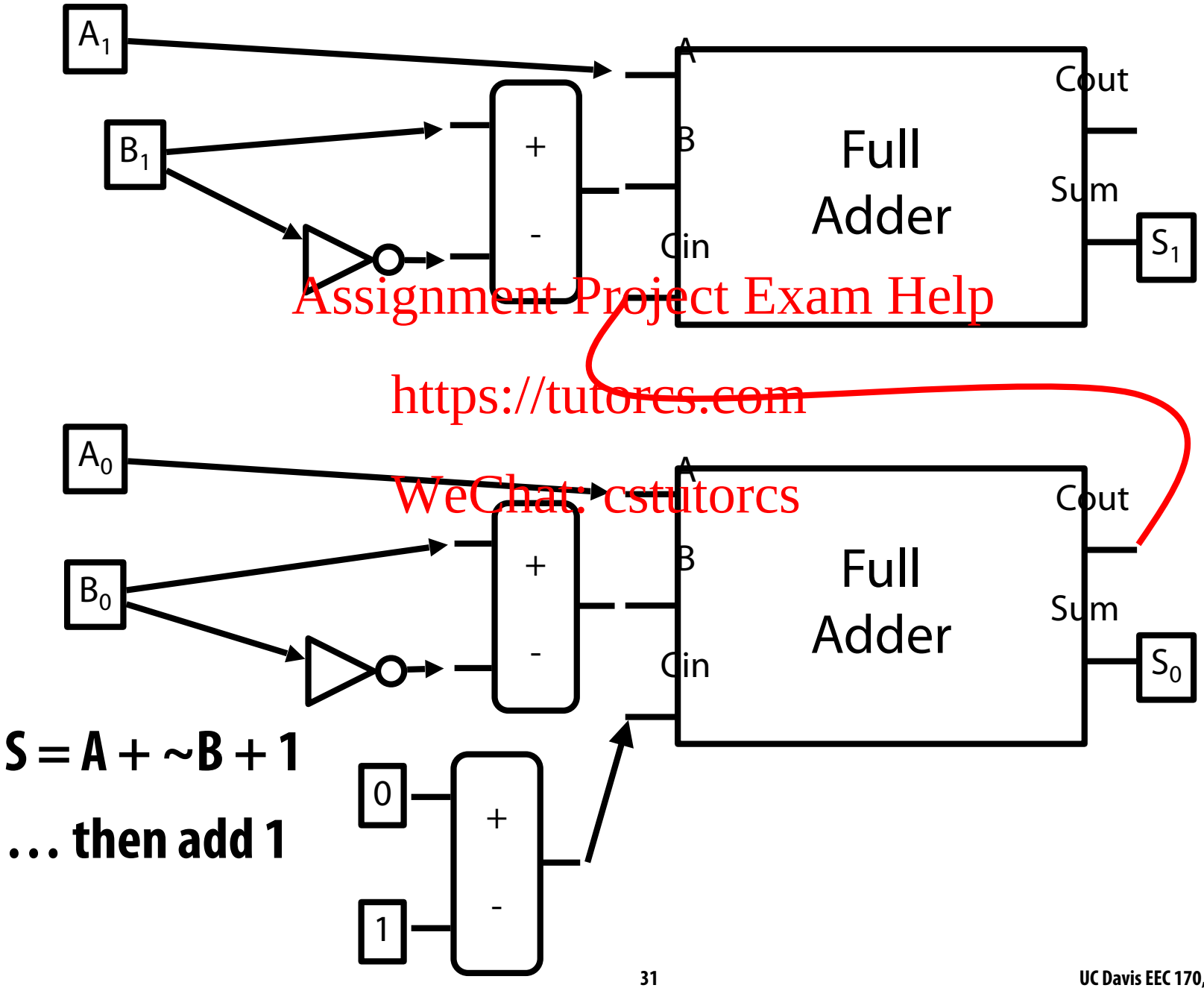
How can we combine + and -?



S = A + ~B + 1

First, negate B . . .

How can we combine + and -?



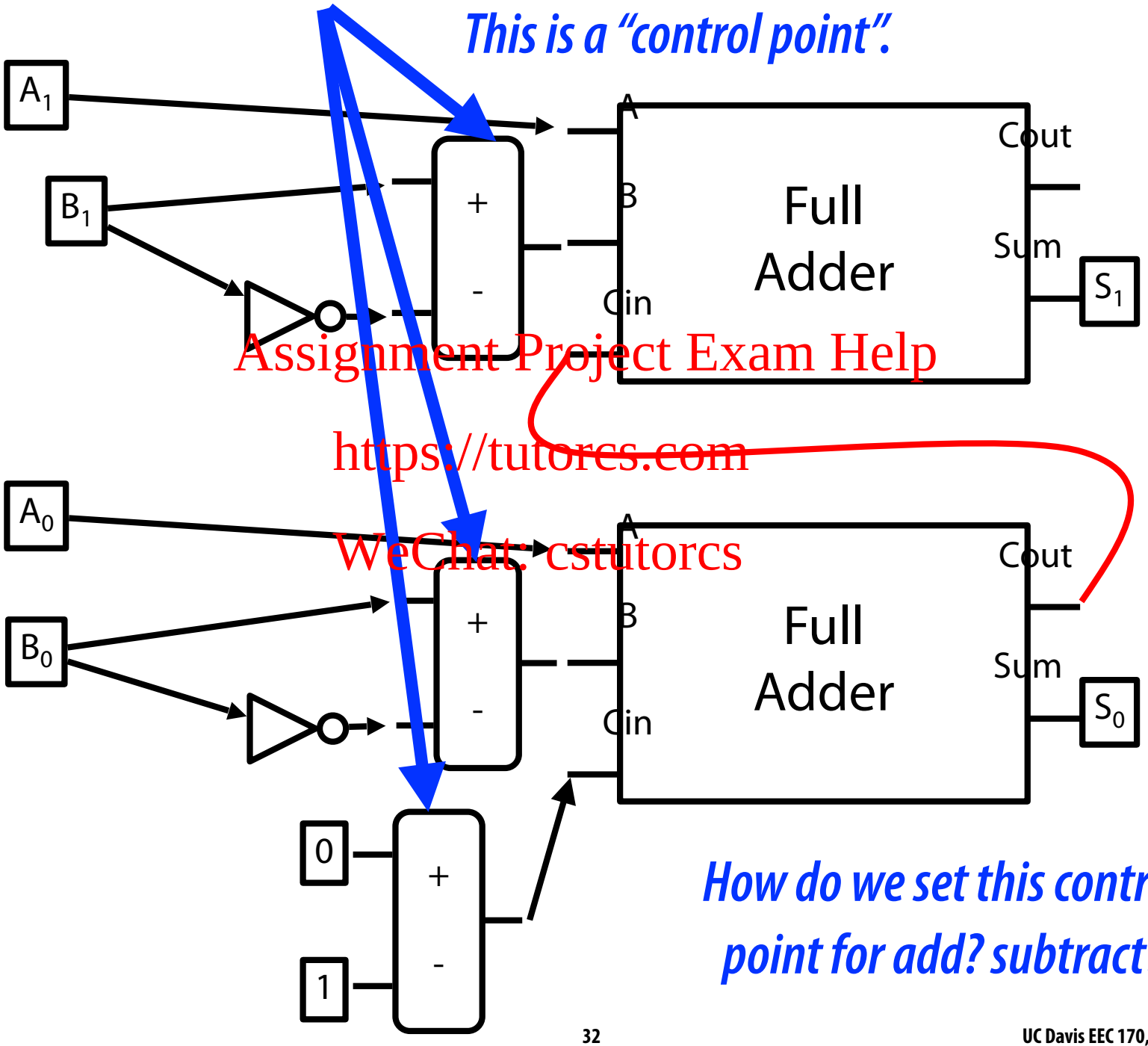
S = A + ~B + 1

. . . then add 1

Control for +/-

One bit controls three muxes.

This is a "control point".



How do we set this control point for add? subtract?

RISC-V instruction encodings

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU

RISC-V instruction encodings (funct7)

- ADD: 0000000
- SUB: 0100000

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU

RISC-V instruction encodings (funct3)

- ADDI: 000
- SLTI: 010
- SLTIU: 011
- SLT: 010
- SLTU: 011

000	rd	0010011	ADDI
010	rd	0010011	SLTI
011	rd	0010011	SLTIU
100	rd	0010011	XORI
110	rd	0010011	ORI
111	rd	0010011	ANDI
001	rd	0010011	SLLI
101	rd	0010011	SRLI
101	rd	0010011	SRAI
000	rd	0110011	ADD
000	rd	0110011	SUB
001	rd	0110011	SLL
010	rd	0110011	SLT
011	rd	0110011	SLTU

Bit Slices

- Concentrate on one bit of the adder:
-
```
    0111
+   0110
```


- Needs:
  - 2 inputs (A and B)
  - Carry from previous slice (Cin)
  - Output (Sum)
  - Carry to next slice (Cout)

MIPS Opcode Map

Opcode

31…29 \ 28…26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

SPECIAL function

5…3 \ 2…0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

REGIMM rt

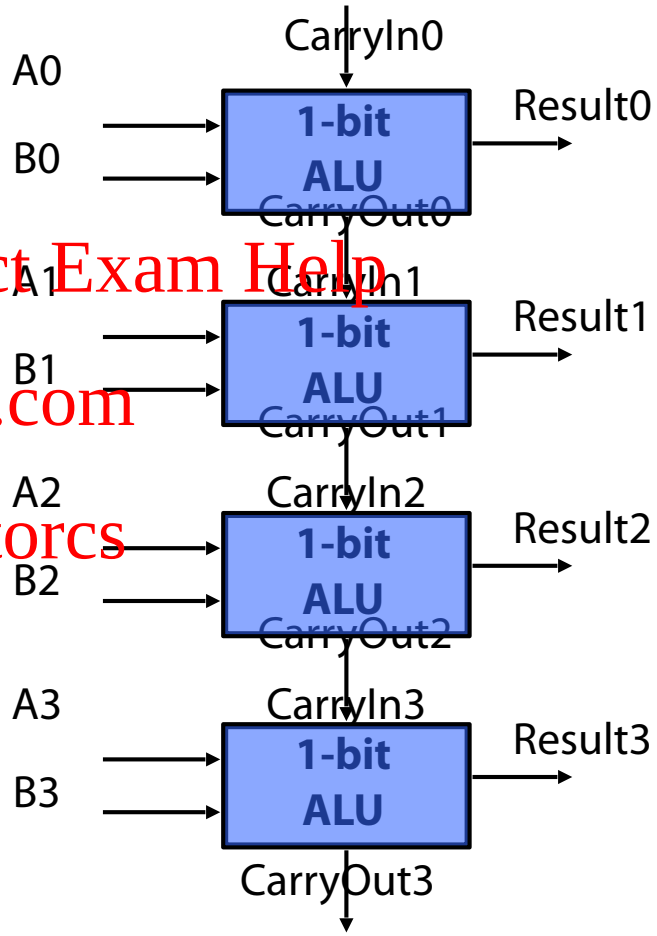
20…19 \ 18…16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

essor User's Manual / Joe Heinrich]

End of lecture / Quiz 1

But What About Performance?

- Critical path of one bitslice is CP
- Critical path of n-bit rippled-carry adder is n*CP
- Design Trick:
  - Throw hardware at it



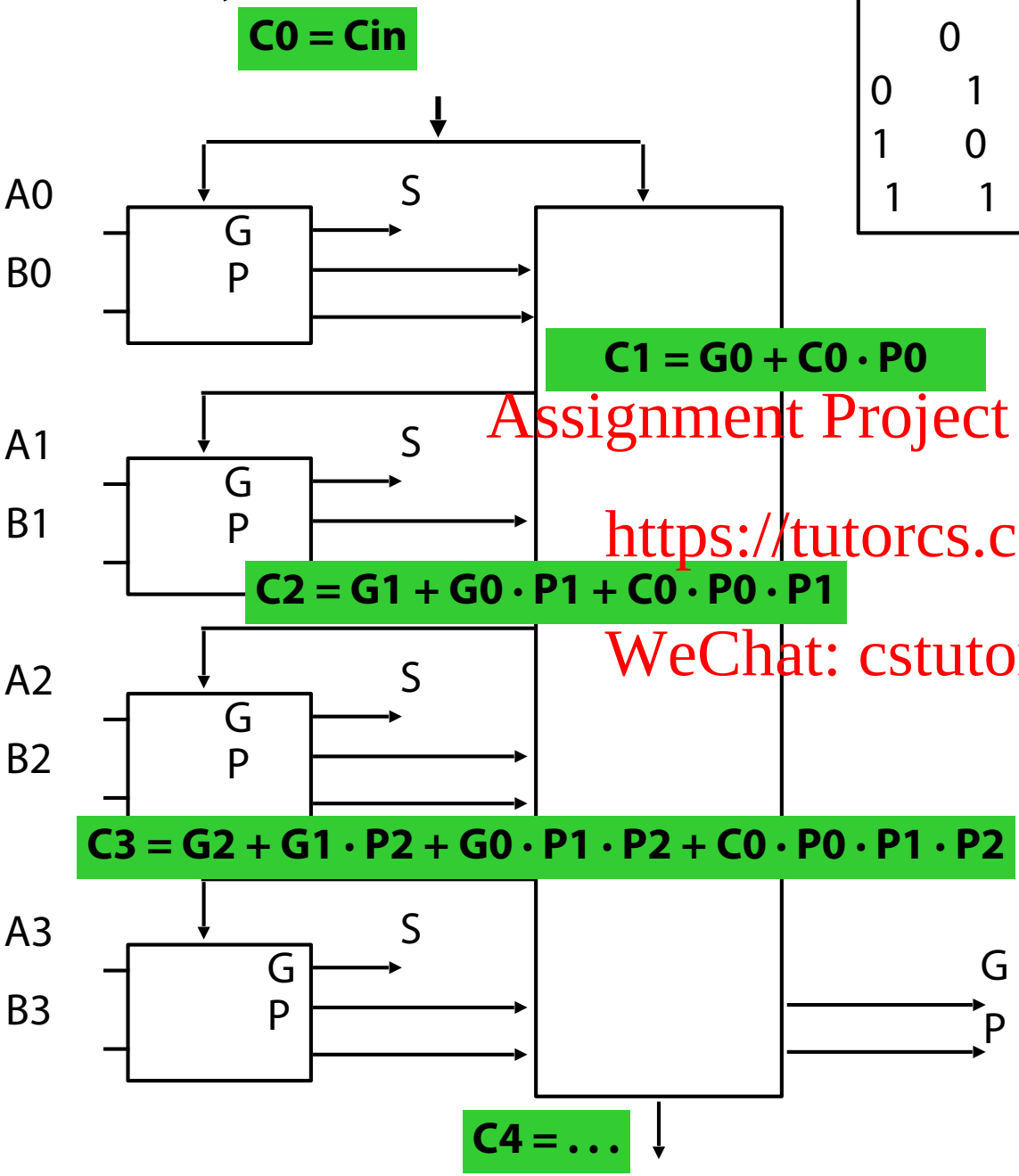
Truth Table for Adder Bit Slice

- 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

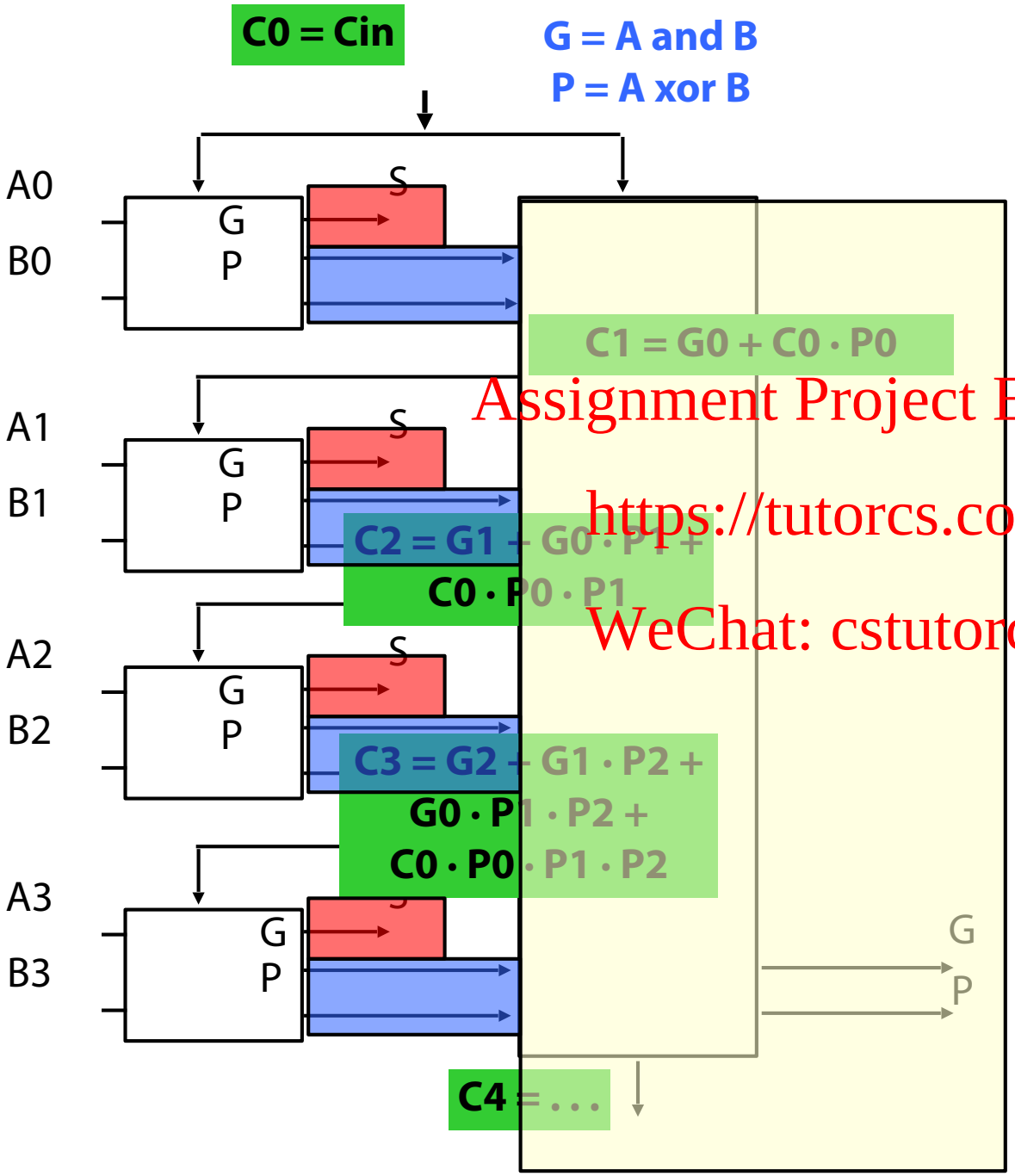
Carry Look Ahead (Design trick: peek)

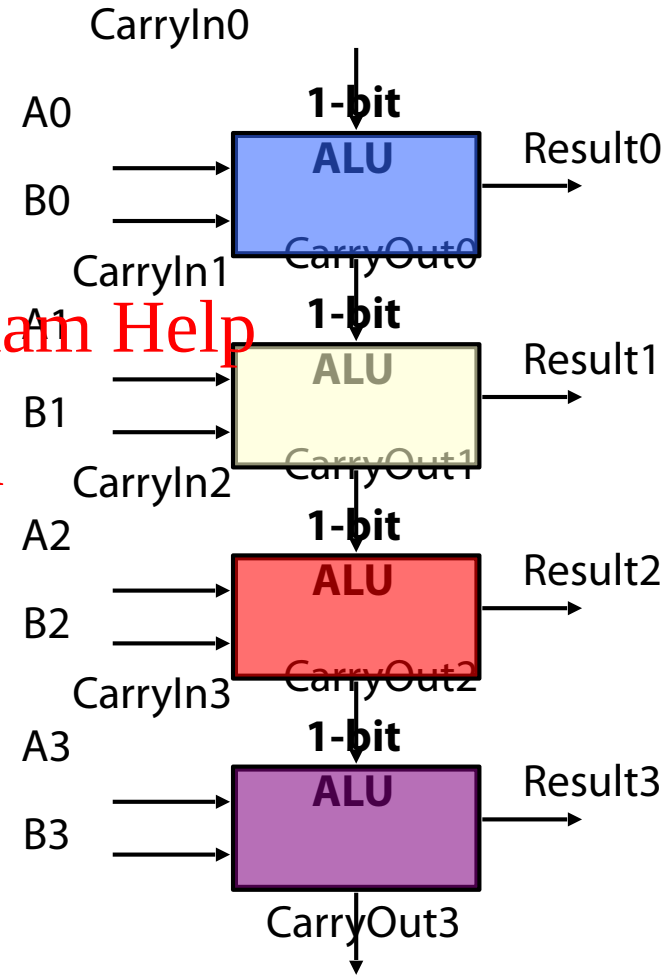
A	B	Cout
0	0	0 "kill"
0	1	Cin "propagate"
1	0	Cin "propagate"
1	1	1 "generate"



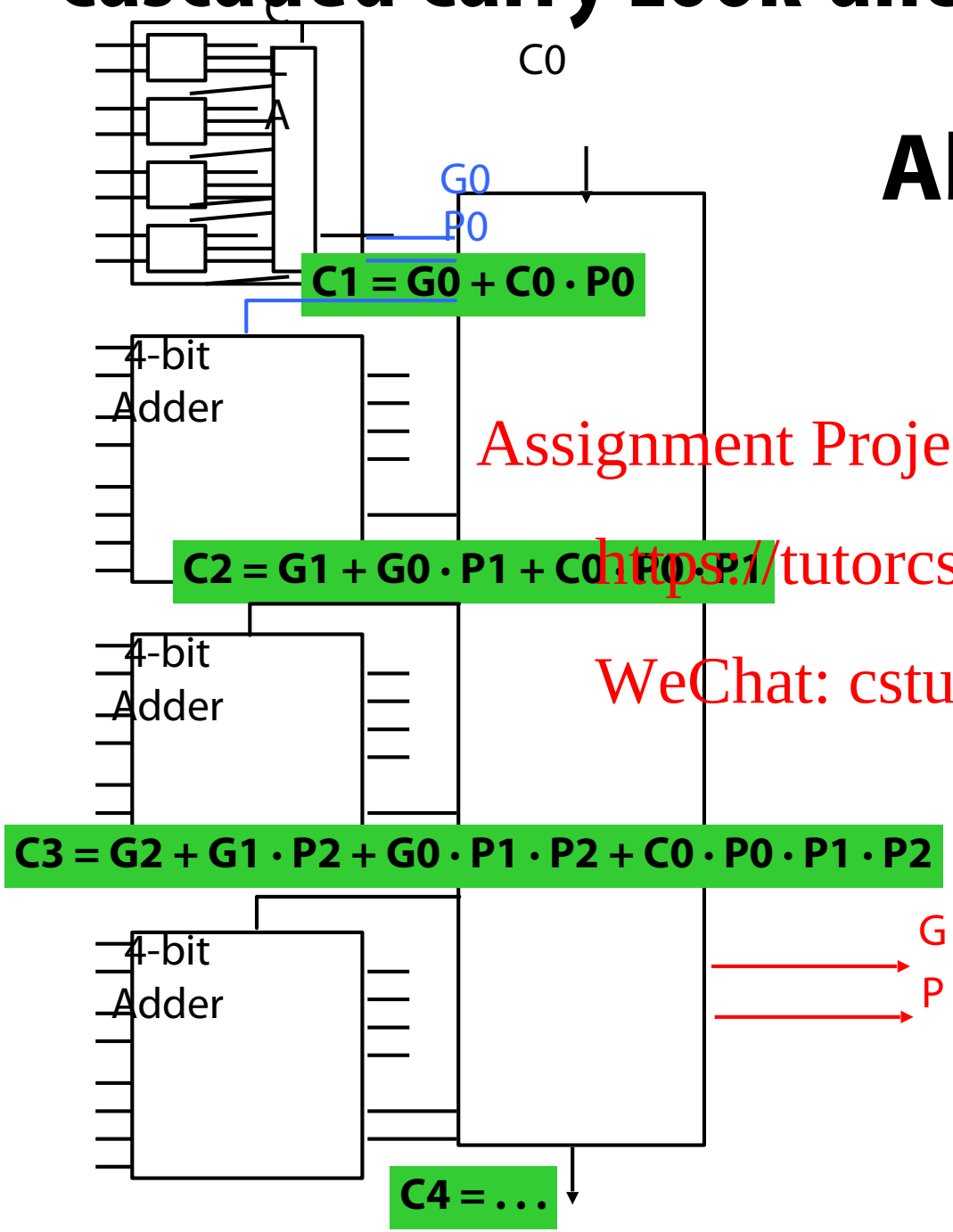
G = A and B
P = A xor B
WHY are these interesting?

CLA vs. Ripple





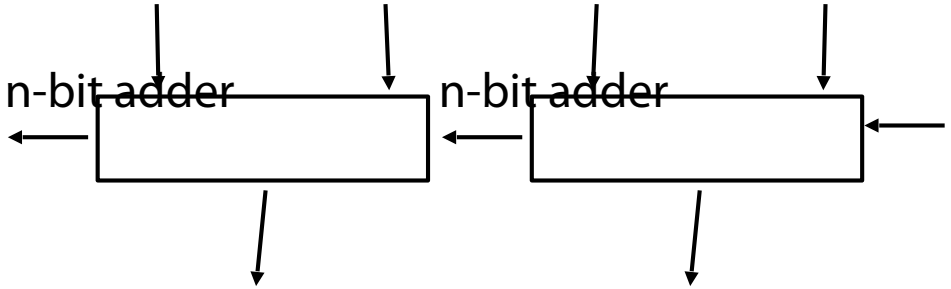
Cascaded Carry Look-ahead (16-bit)



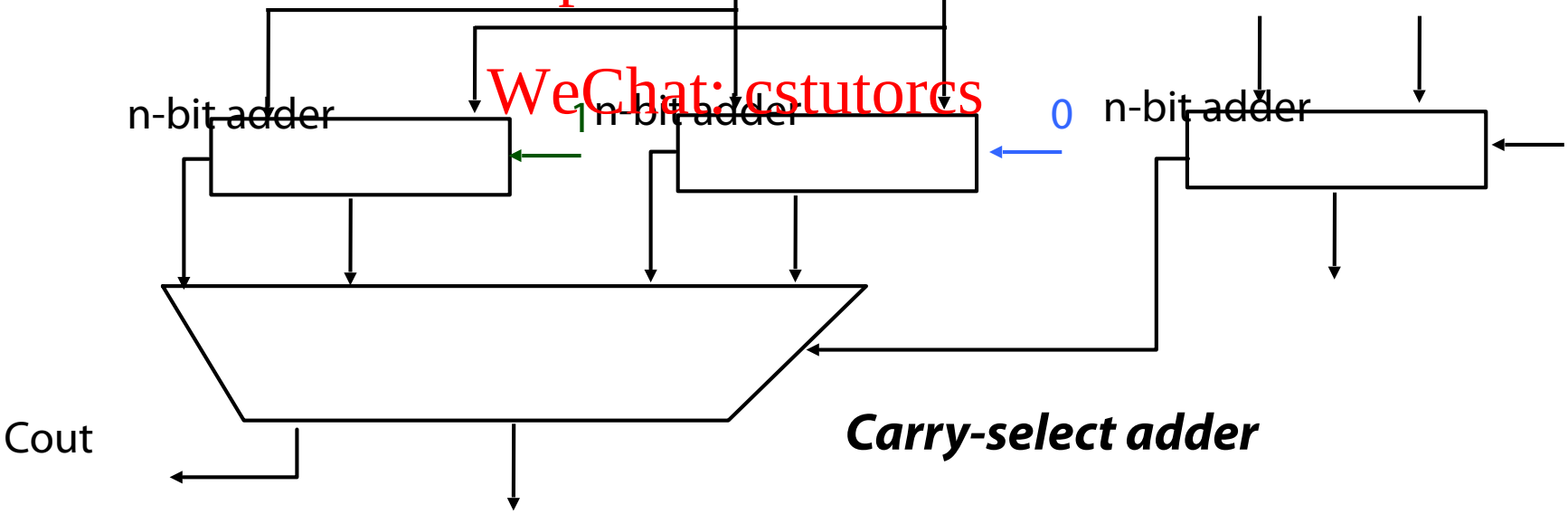
Abstraction!

Design Trick: Guess (or "Precompute")

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



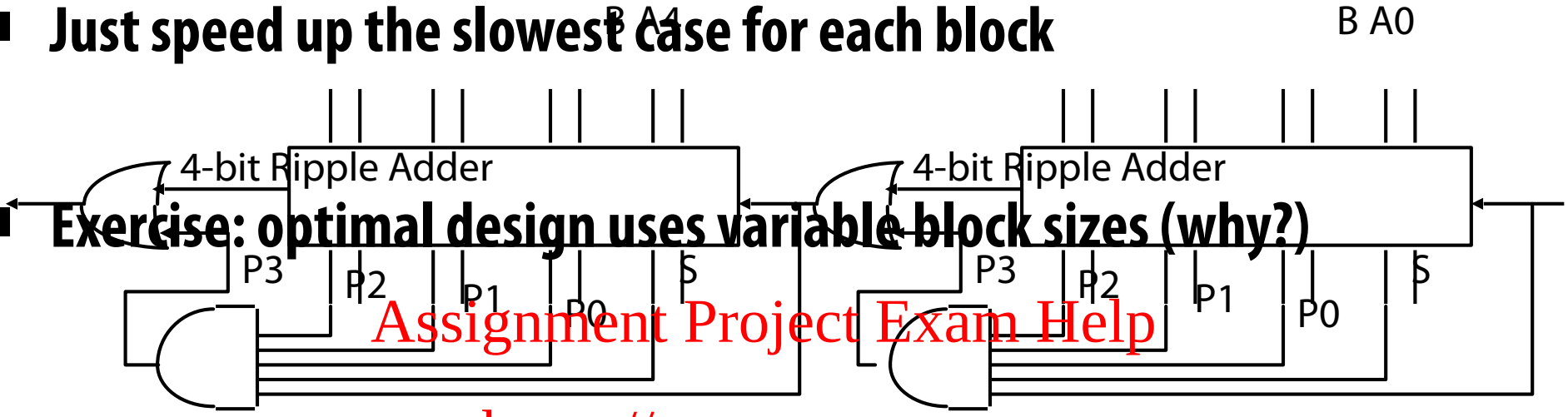
P(2n) = CP(n) + CP(mux)



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]

Finished way early (1:20 in, 30 minutes

End of lecture

Lecture 6:

Arithmetic 2/3

John Owens

Introduction to Computer Architecture

UC Davis EEC 170, Winter 2021

Multiply (unsigned)

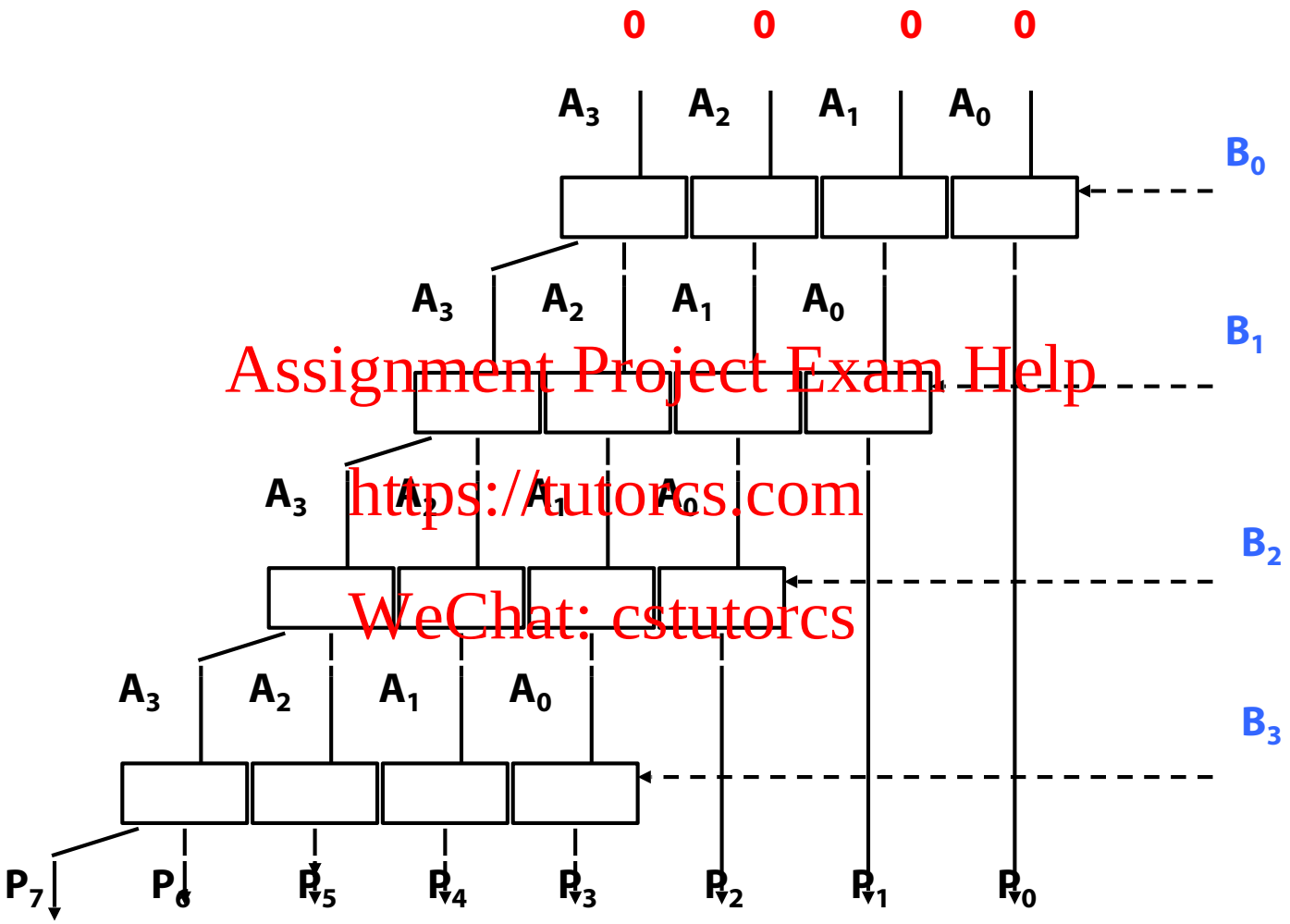
- Paper and pencil example (unsigned):

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

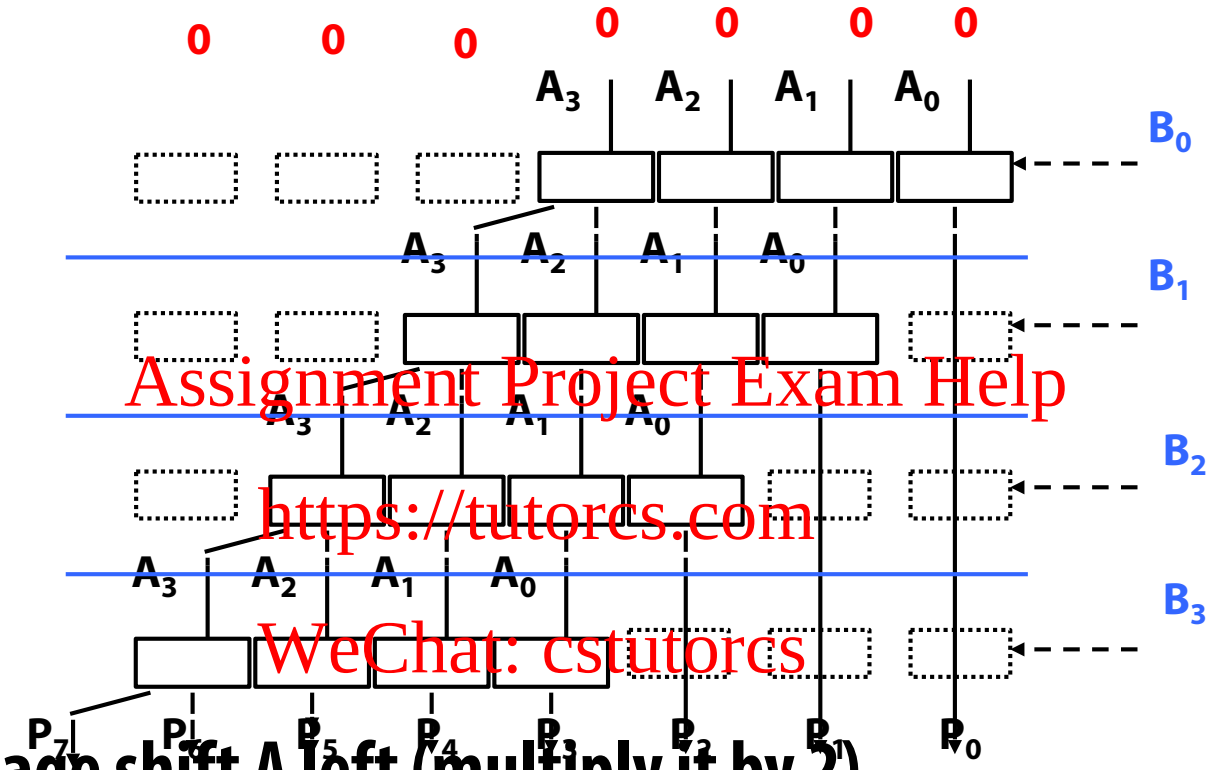

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
- 4 versions of multiply hardware & algorithm:
  - successive refinement

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

Unsigned Combinational Multiplier

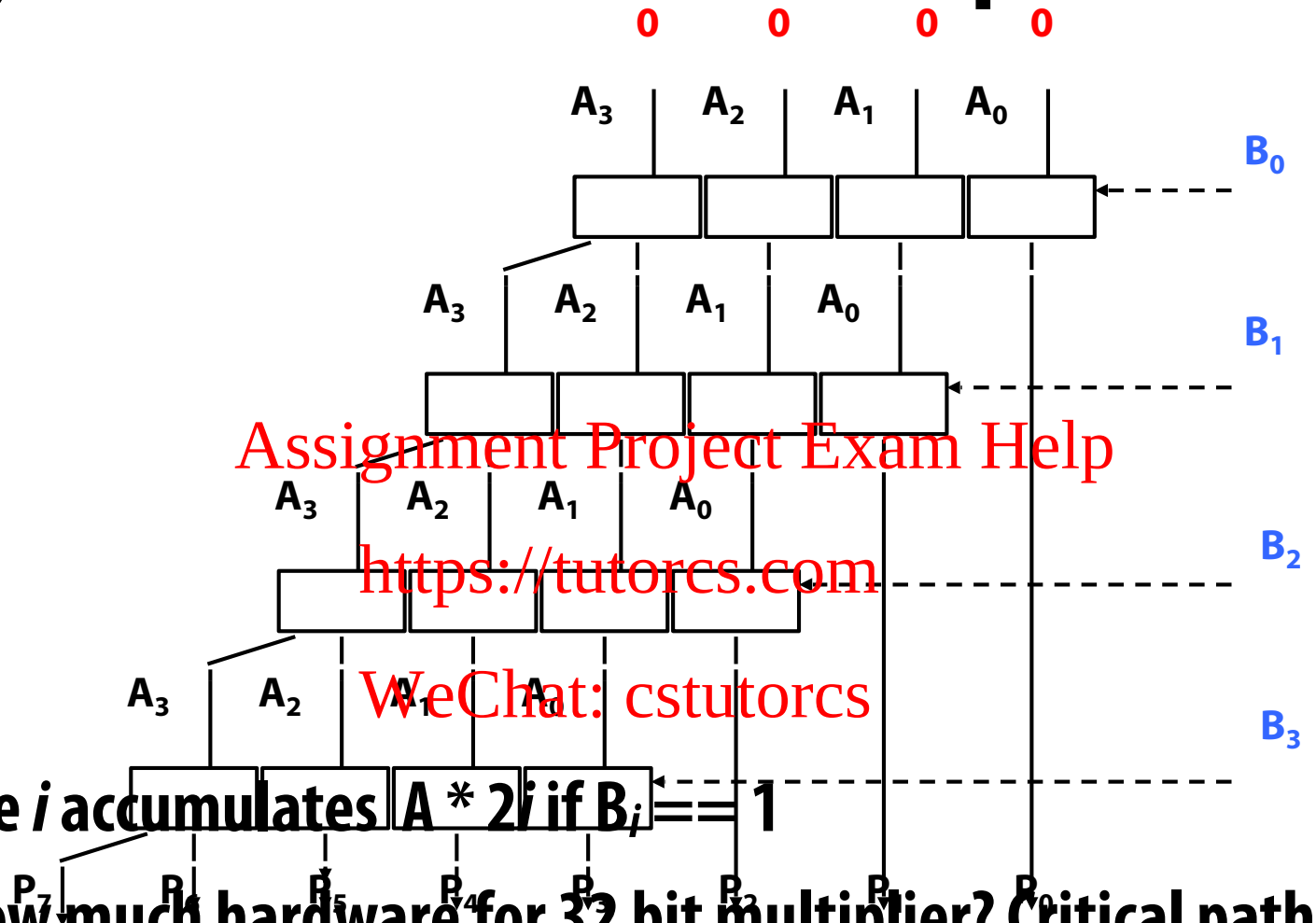


How does it work?



- 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

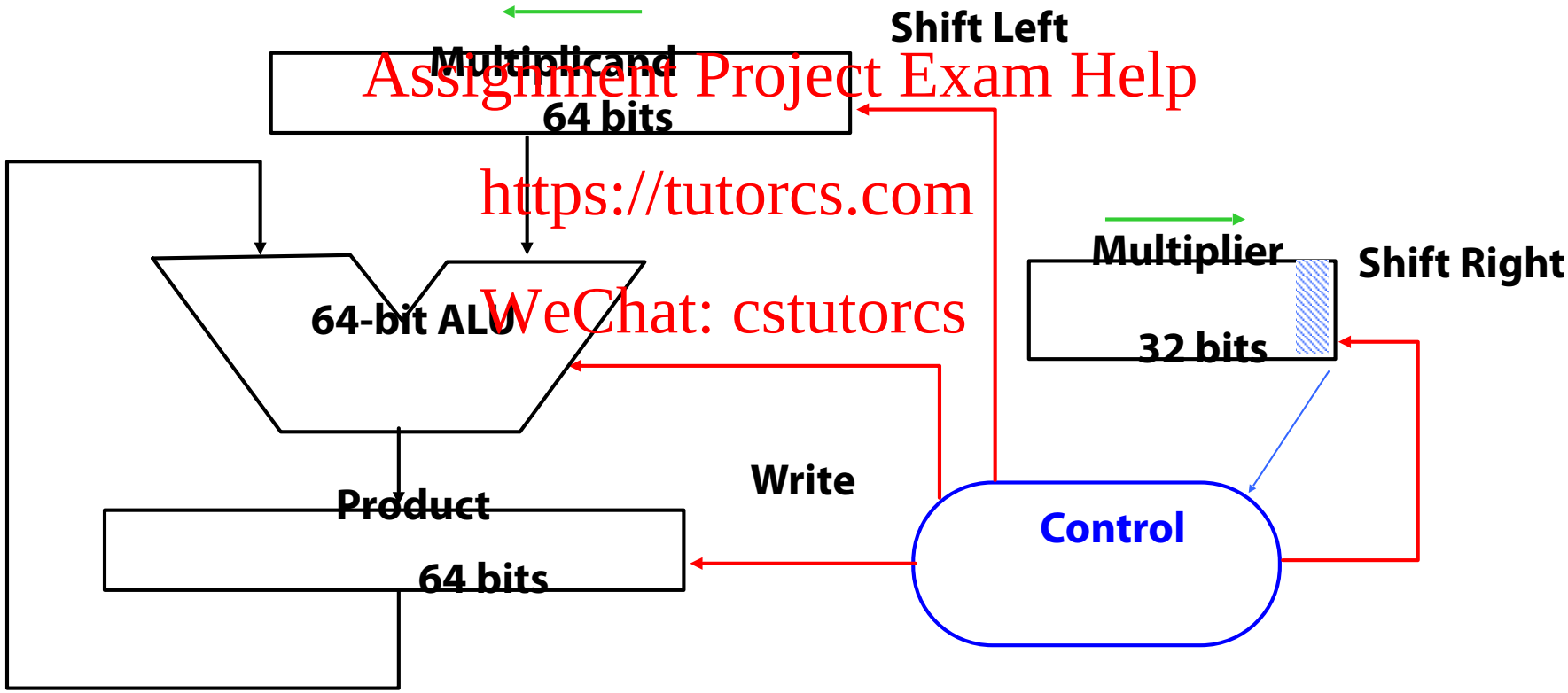
Unsigned Combinational Multiplier



- 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)

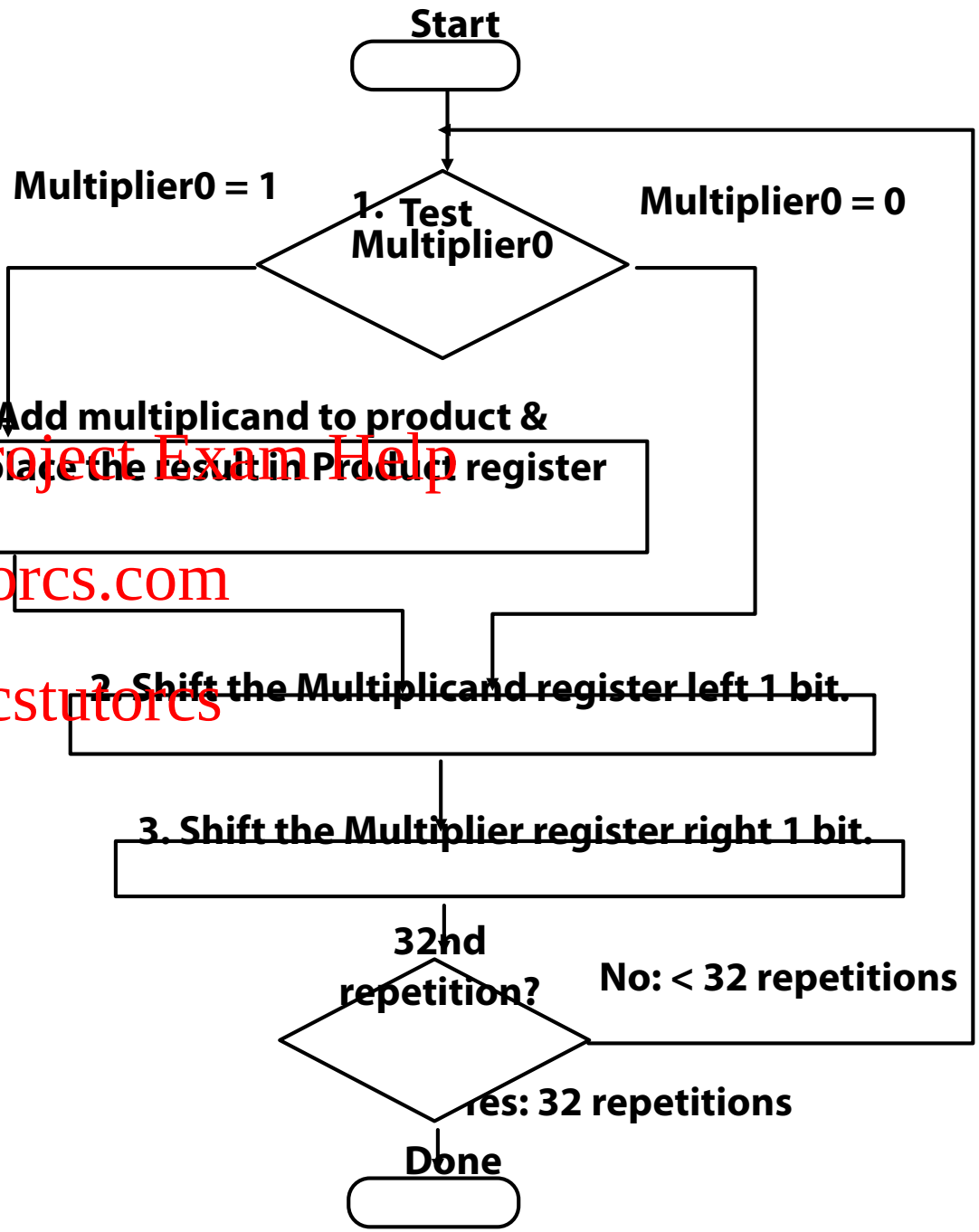
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



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

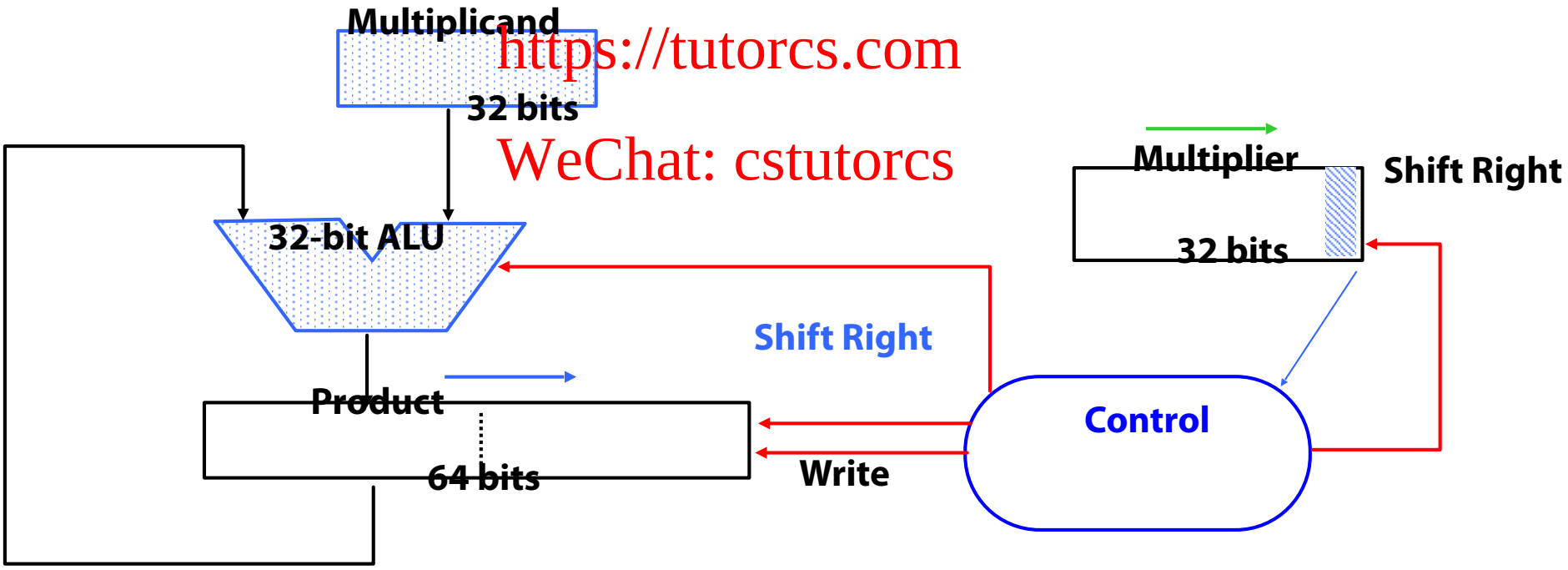


Observations on Multiply Version 1

- 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?*

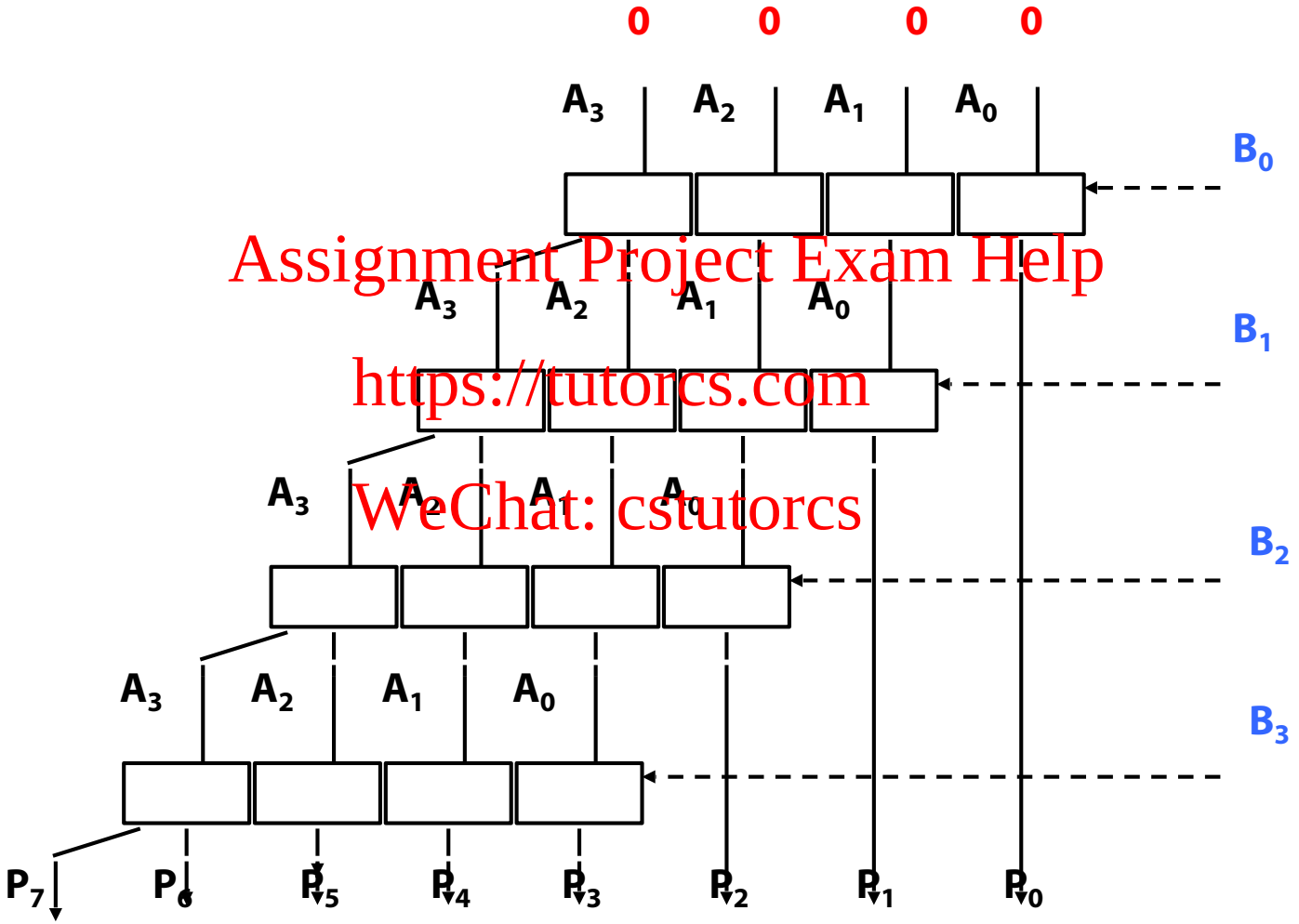
Multiply Hardware Version 2

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

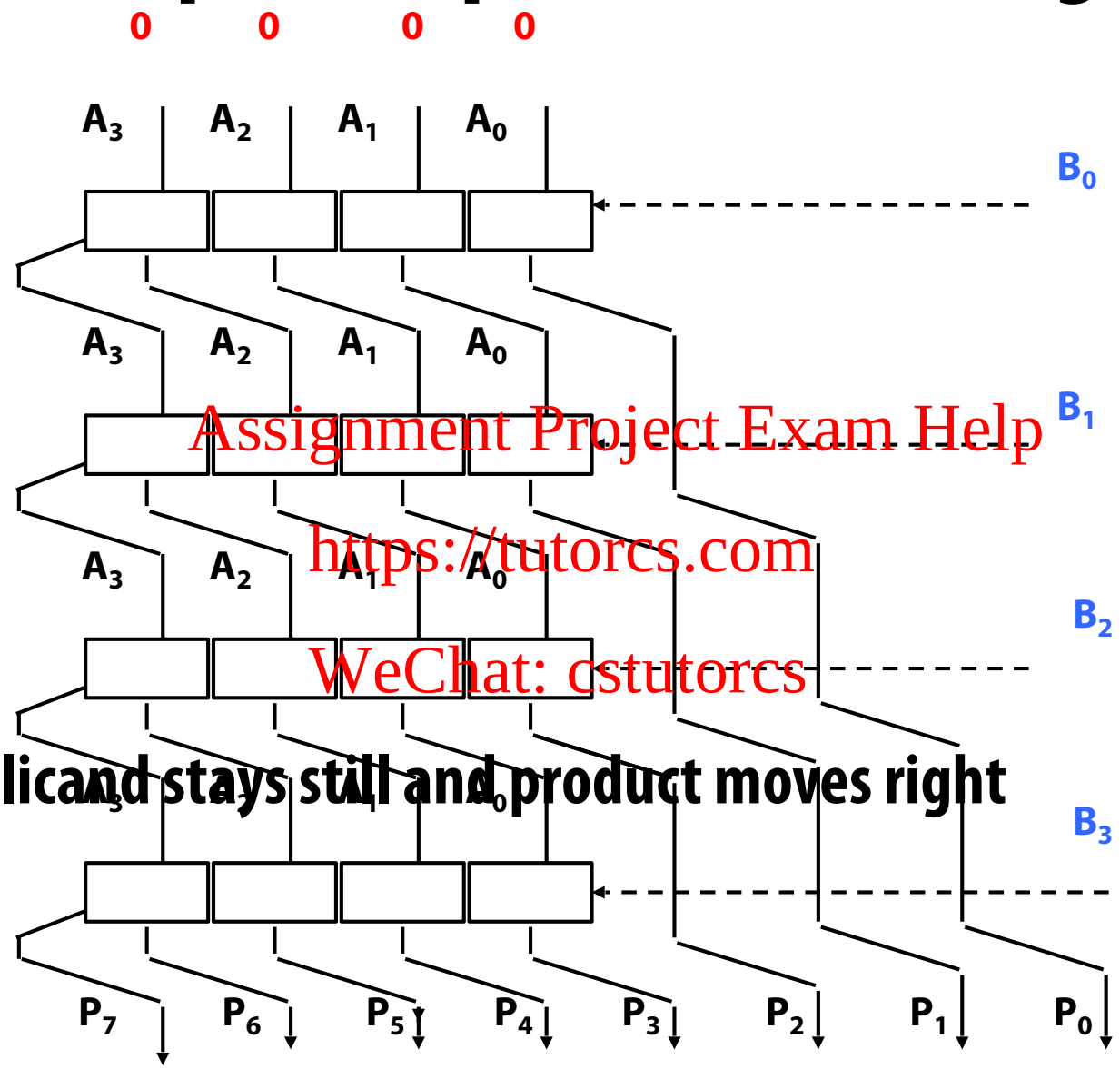


How to think of this?

- Remember original combinational multiplier:

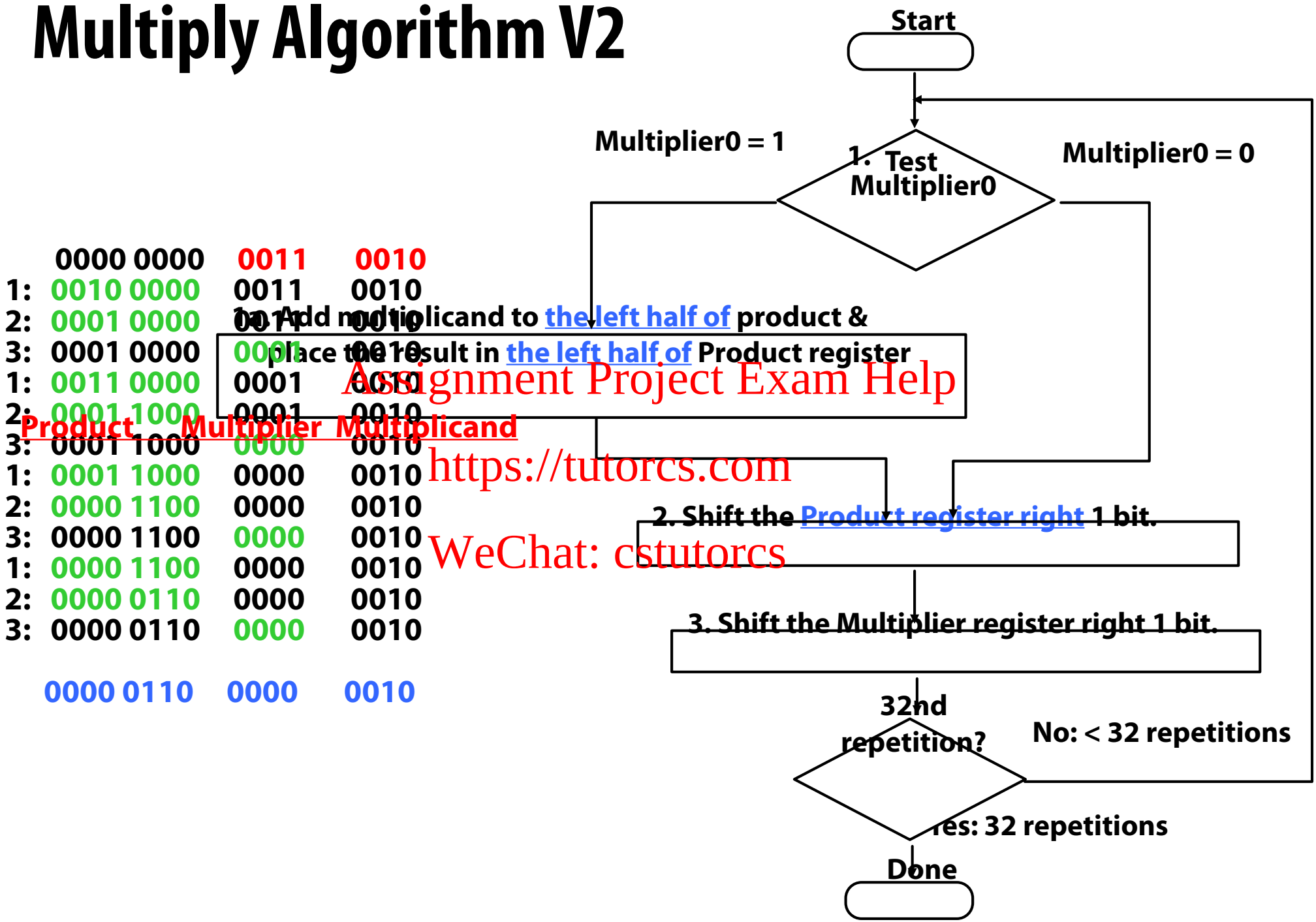


Simply warp to let product move right…



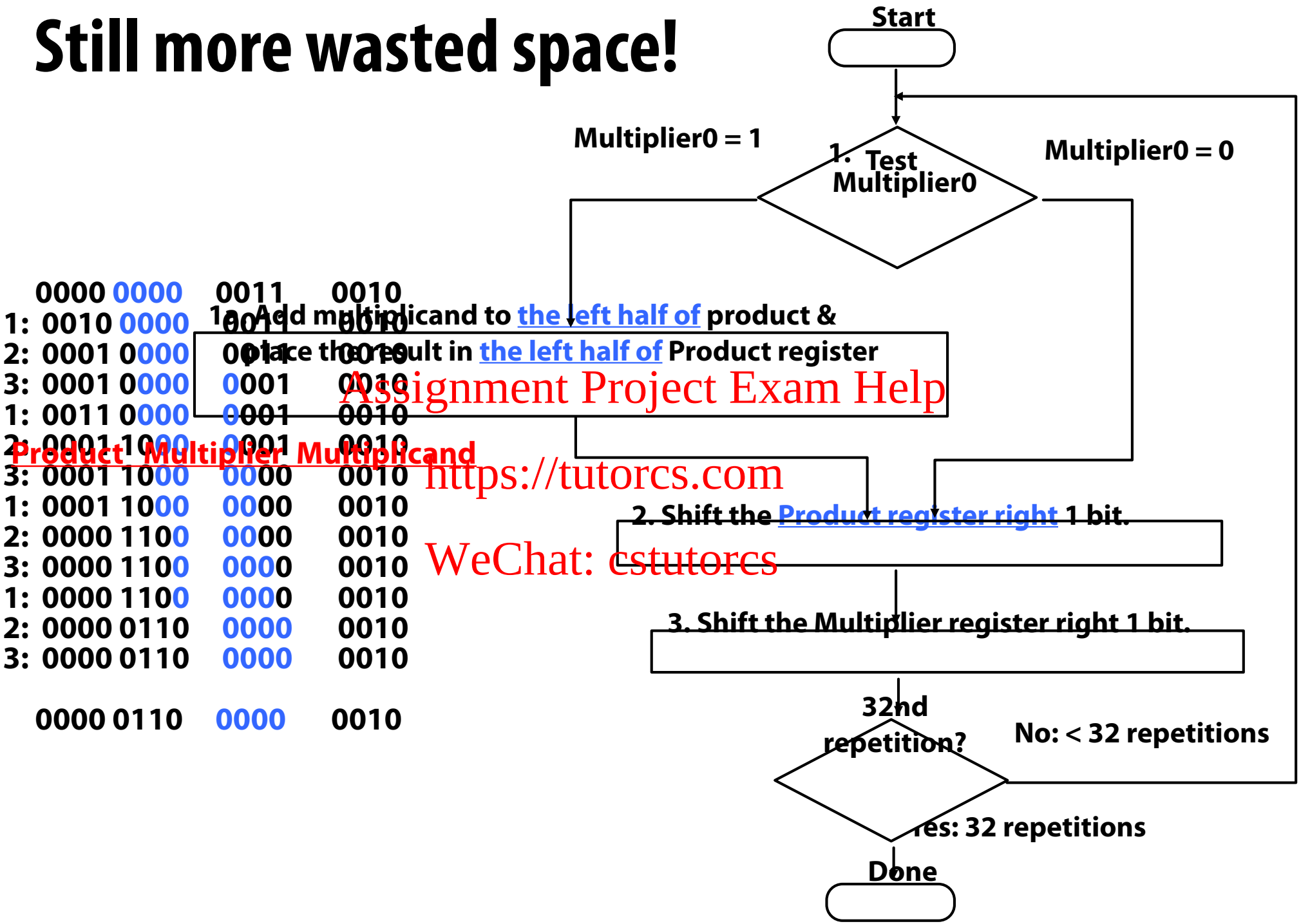
- Multiplicand stays still and product moves right

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!

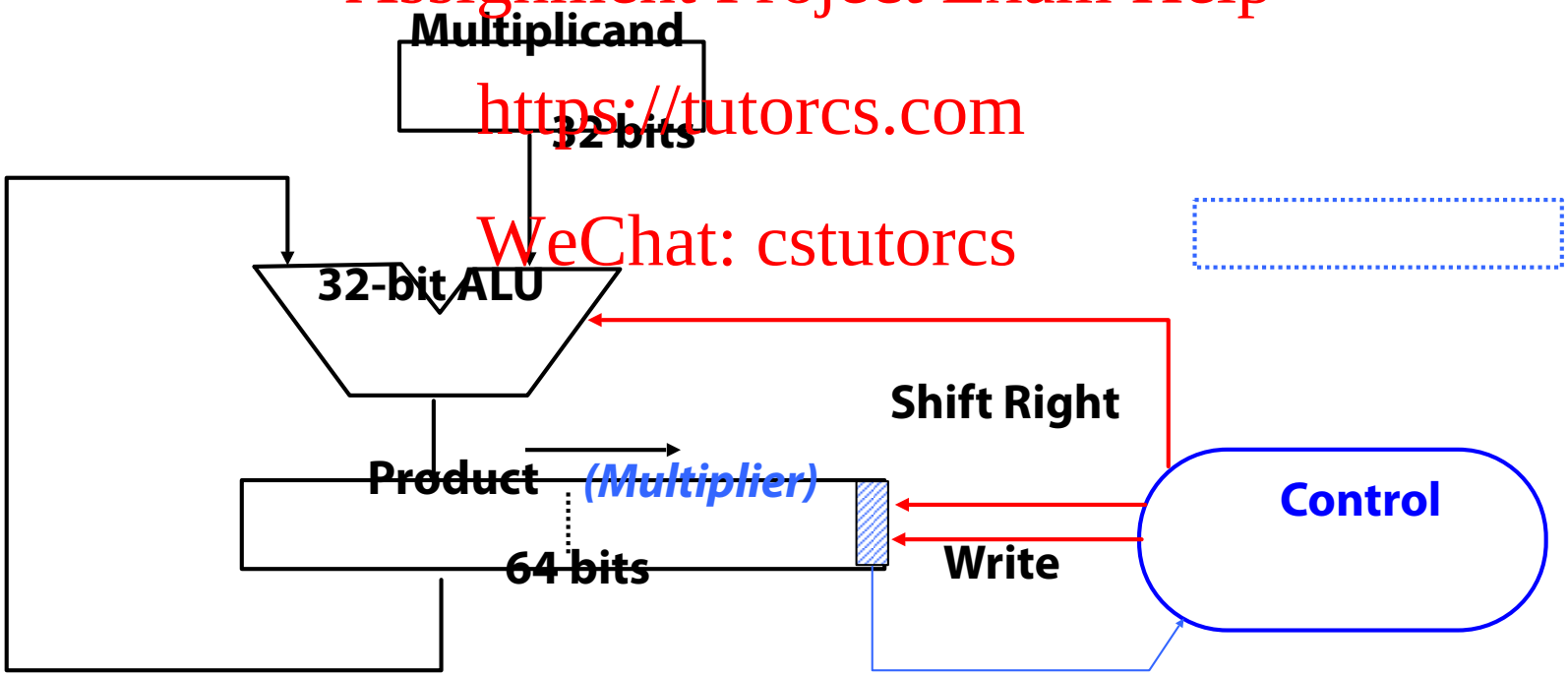


Observations on Multiply Version 2

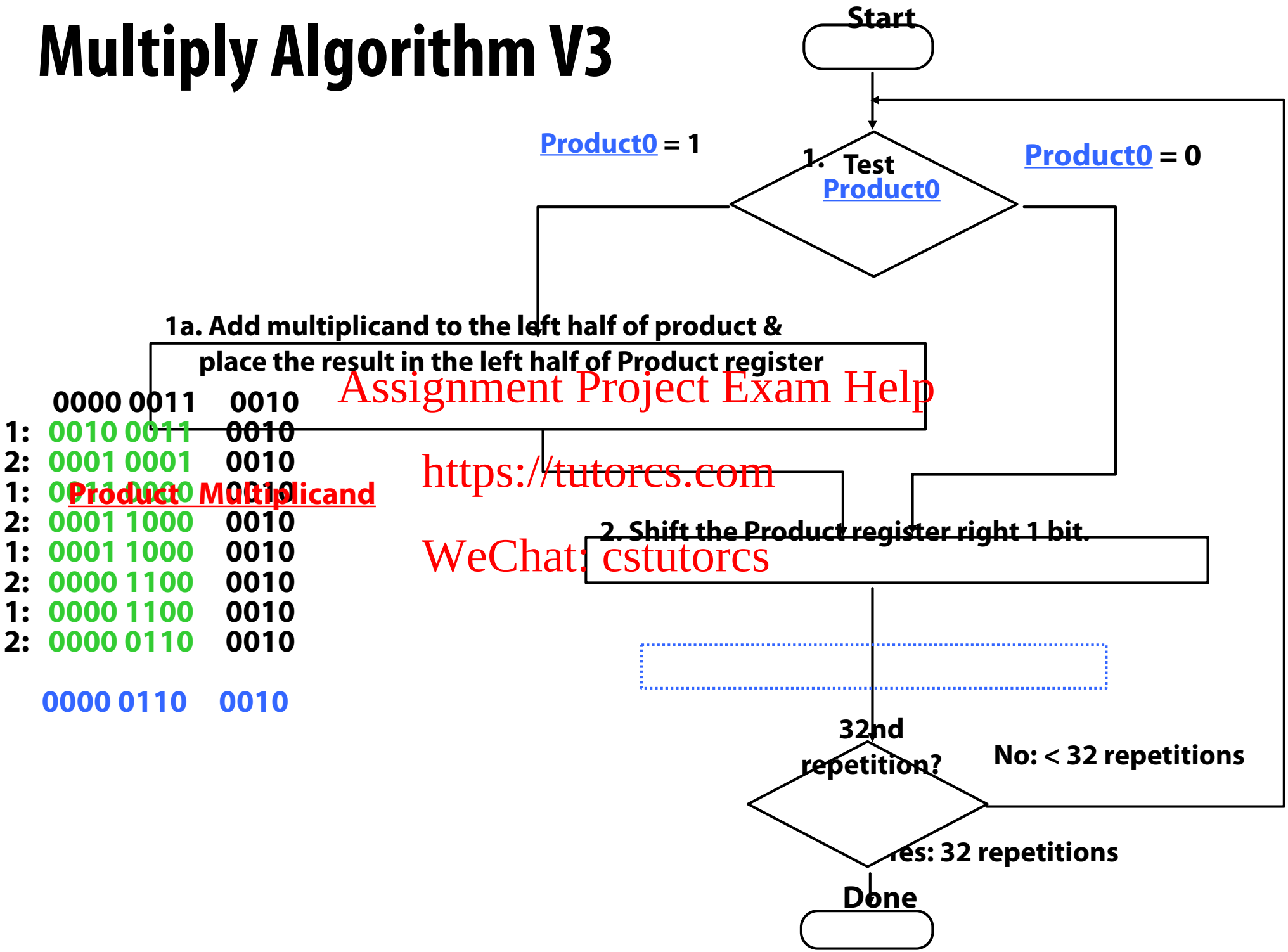
- Product register wastes space that exactly matches size of multiplier
  => combine Multiplier register and Product register

Multiply Hardware Version 3

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



Multiply Algorithm V3



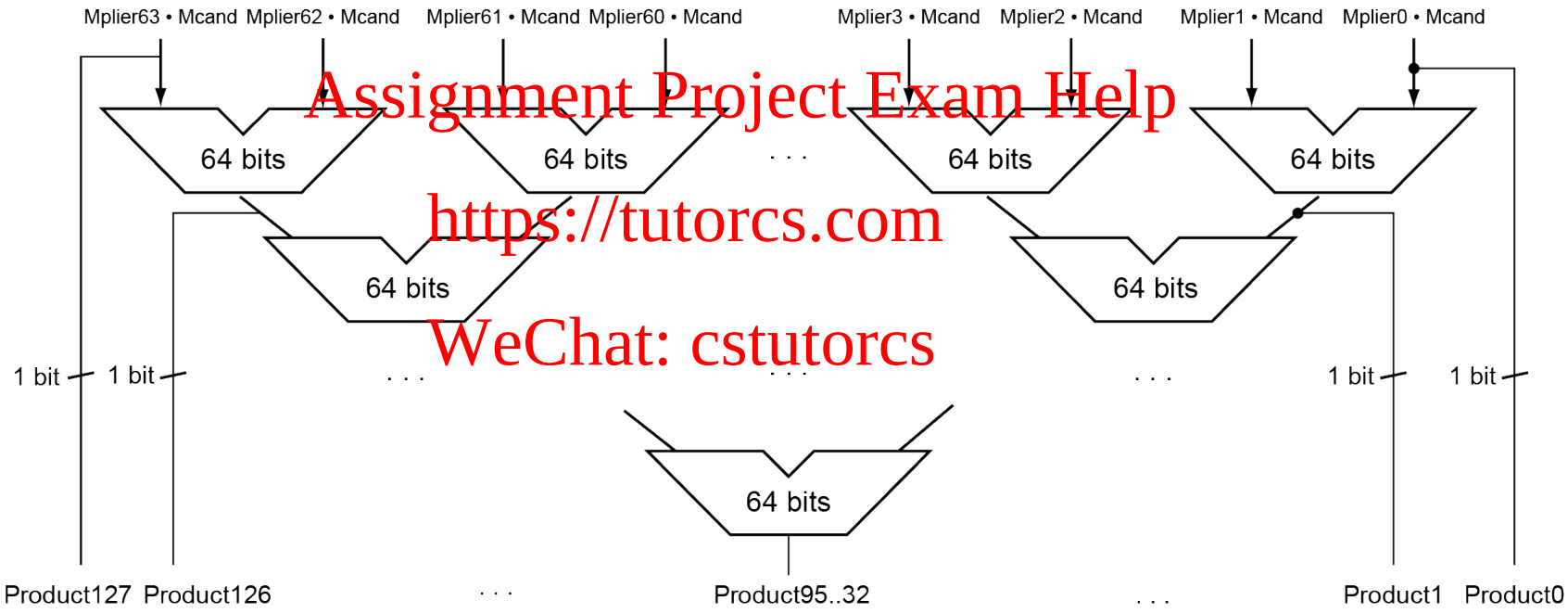
	Product	Multiplicand
	0000 0011	0010
1:	0010 0011	0010
2:	0001 0001	0010
1:	0011 0001	0010
2:	0001 1000	0010
1:	0001 1000	0010
2:	0000 1100	0010
1:	0000 1100	0010
2:	0000 0110	0010
	0000 0110	0010

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

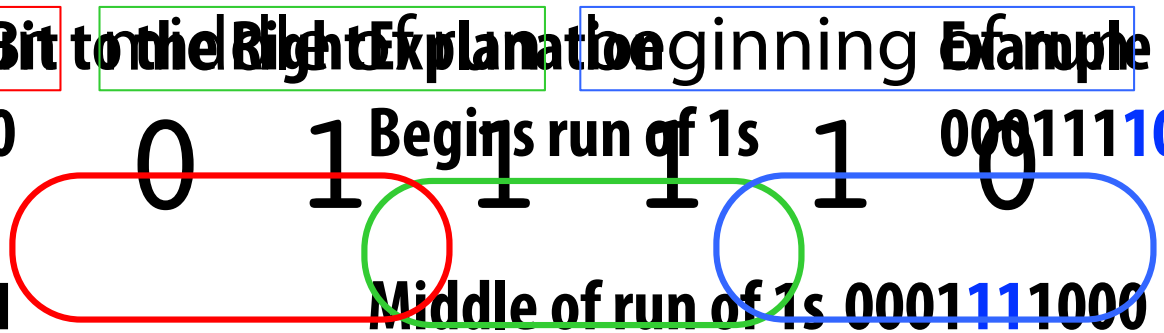
- ALU with add or subtract can get same result in more than one way:
- 6 = 4 + 2 = -2 + 8
  0110 = -00010 + 01000 = 11110 + 01000
- For example:


```
      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

Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Beginning of 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	1111 0110 1	01 -> add
2a.		0001 0110 1	shift P
2b.	0010	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

Break

Administrivia

- TA Trivikram coming at 11:30 to present the project
  - You're going to write 4 RISC-V procedures.
- HW 3 released last Friday, due on Friday
  - Solutions will be outside my office, Kemper 3175
- Midterm is week from today
  - Open book, open note
  - Bring a calculator

Is NVIDIA Doubling Down On RISC-V?

- **Betteridge's law of headlines is an adage that states: "Any headline that ends in a question mark can be answered by the word no".**

Is NVIDIA Doubling Down On RISC-V?

Synced Follow

Oct 10 · 4 min read

Is NVIDIA Doubling Down On RISC-V?

- "Six RISC-V positions have been advertised by NVIDIA, based in Shanghai and pertaining to architecture, design, and verification."
- "Due its light weight and extensibility, RISC-V is gaining mainstream adoption across many new sectors, including datacenter accelerators, mobile & wireless, automotive, and IoT."
- "In a 2017 RISC-V workshop in Shanghai, NVIDIA explained that shortcomings such as low performance and lack of caches and thread protection meant Falcon's architecture could not meet growing complexity demands."
- "NVIDIA listed the technical criteria for its next-gen architecture: more than twice the performance of Falcon, less than twice the area cost of Falcon, support for caches, tightly coupled memories, 64-bit addresses, and suitability for modern operating systems. They concluded only RISC-V meets all criteria. The new RISC-V micro-controllers will outperform Falcon micro-controllers by three times, Tom's Hardware has reported."

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)

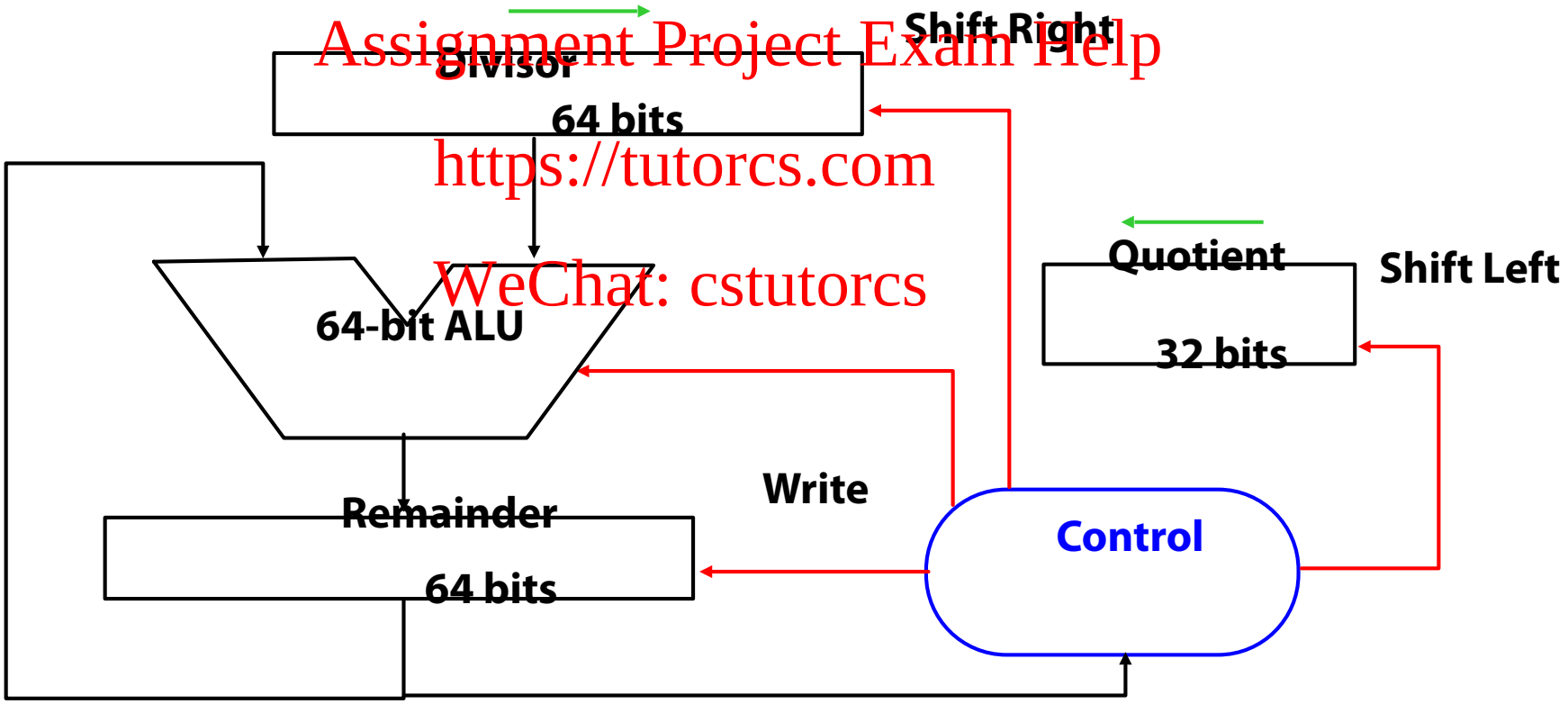
Divide: Paper & Pencil

```
                    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

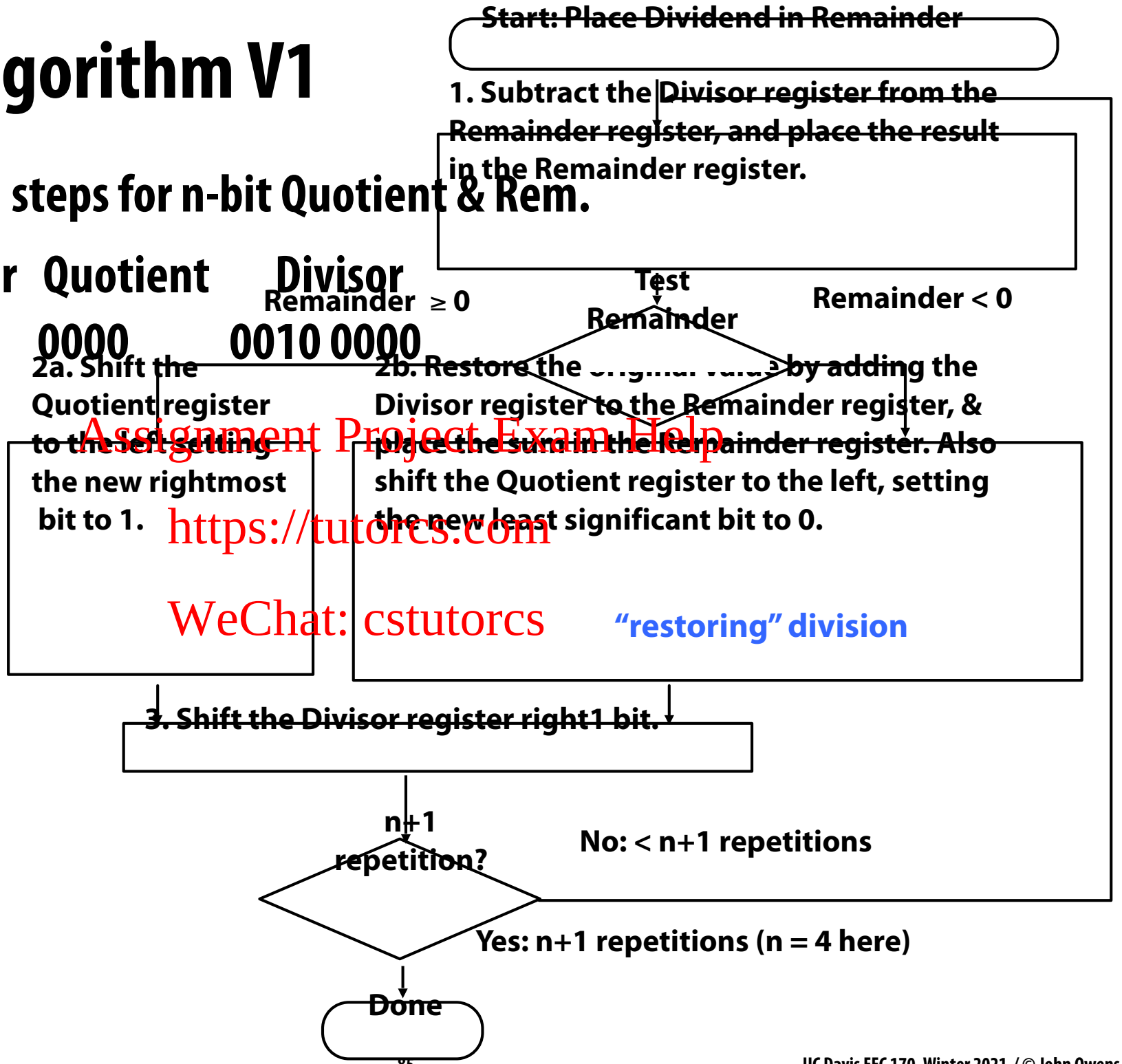
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



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

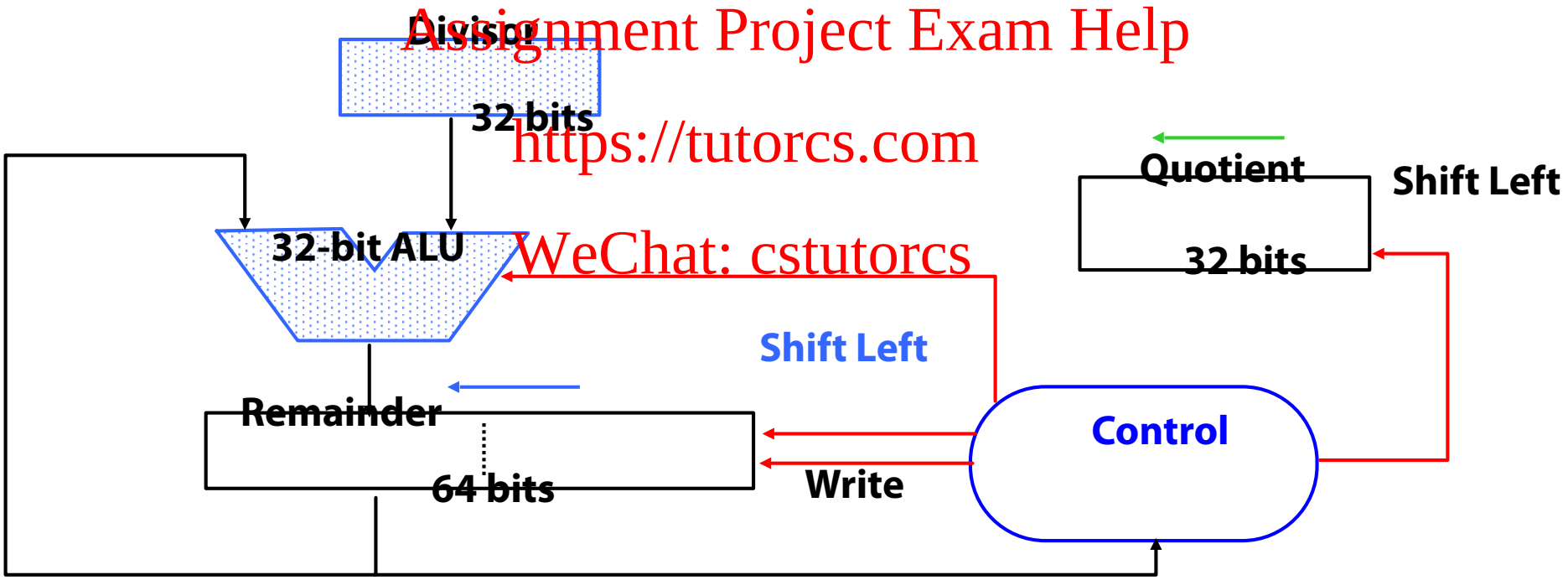
- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?

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

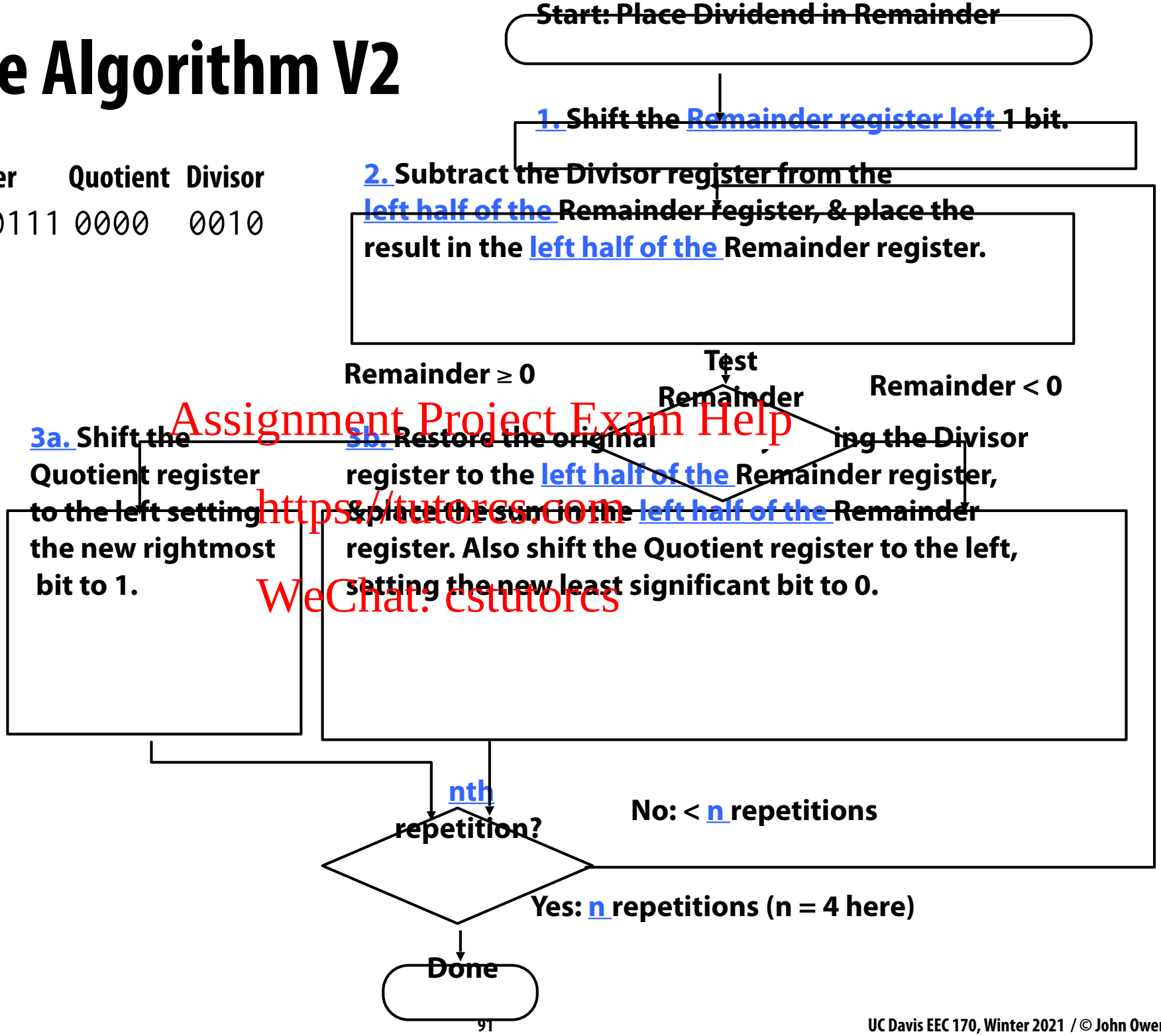
Divide Hardware Version 2

- 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

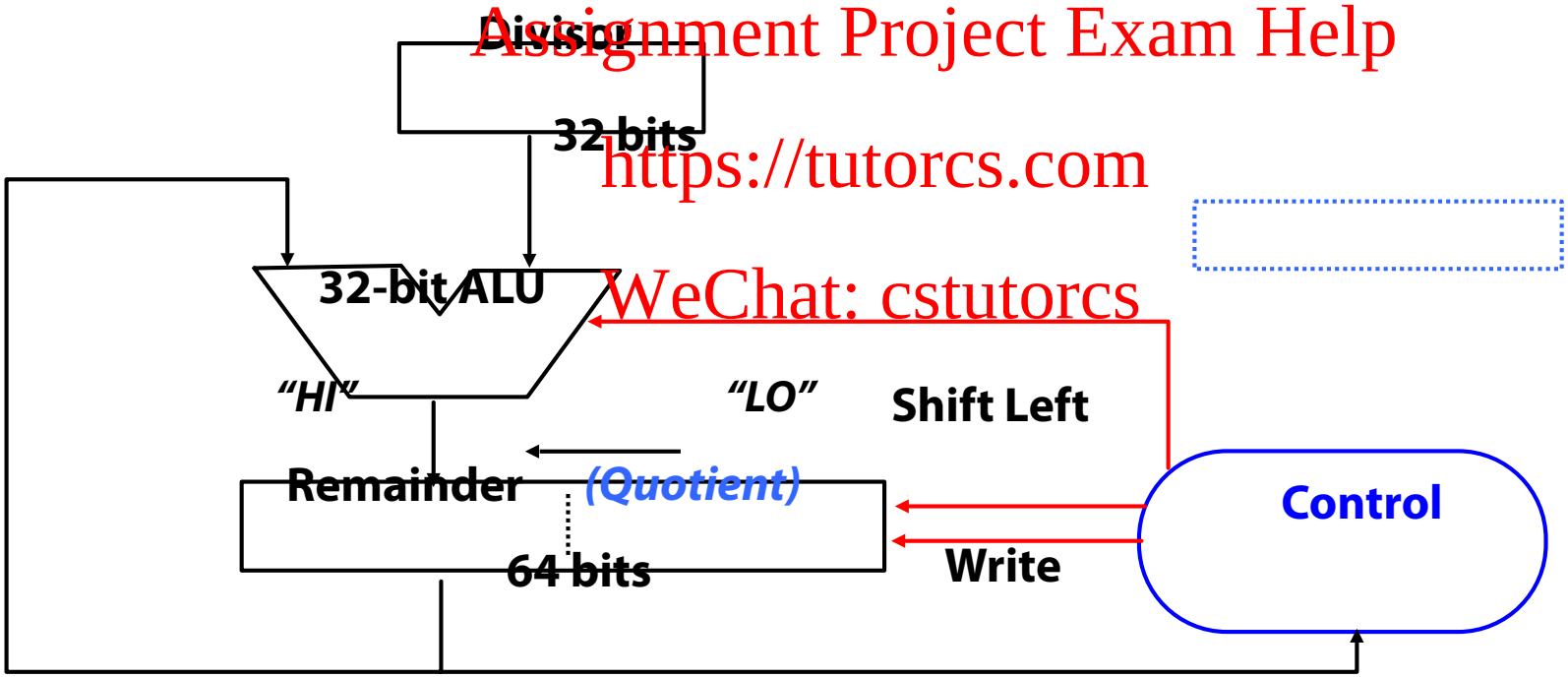


Observations on Divide Version 2

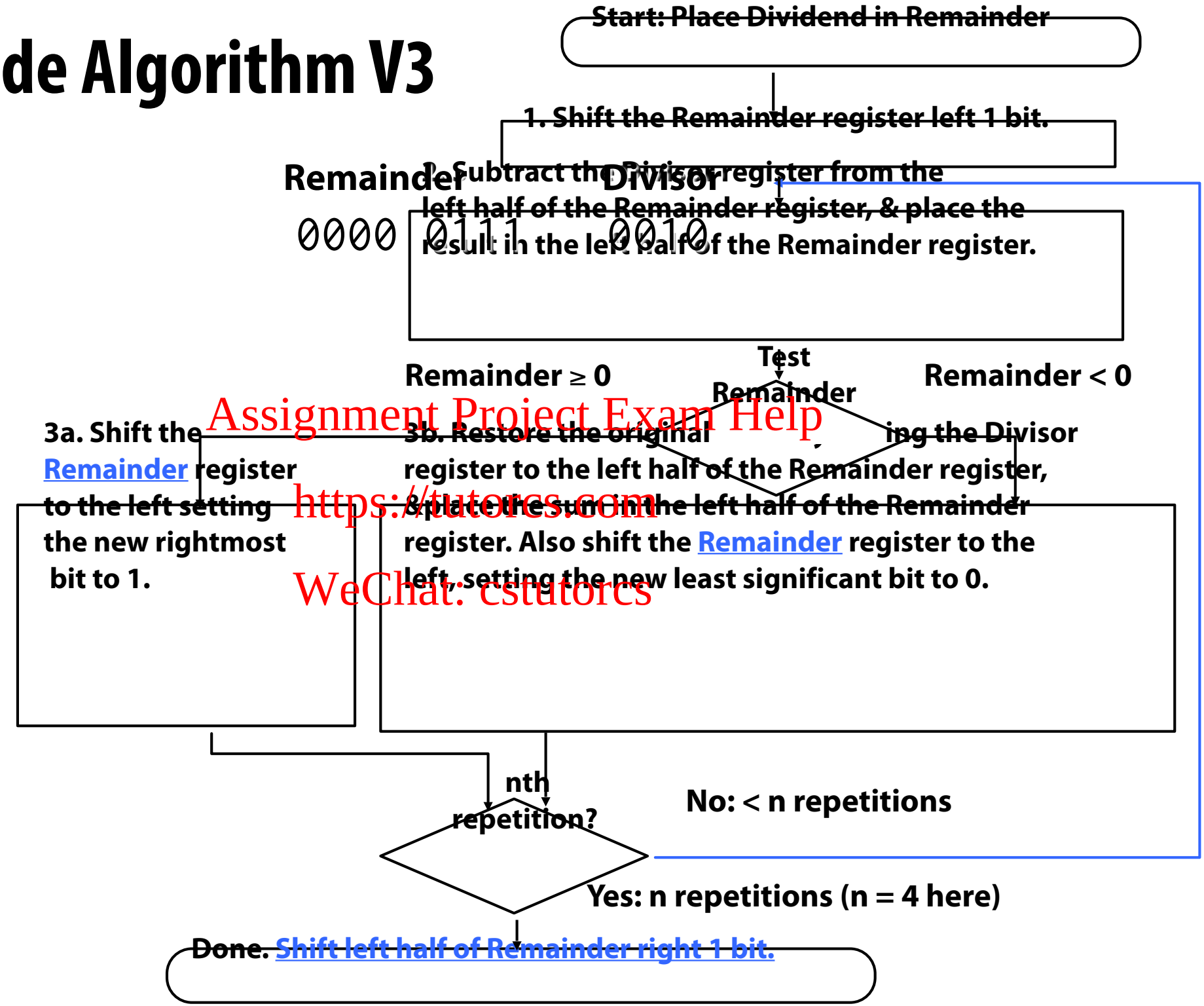
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The 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

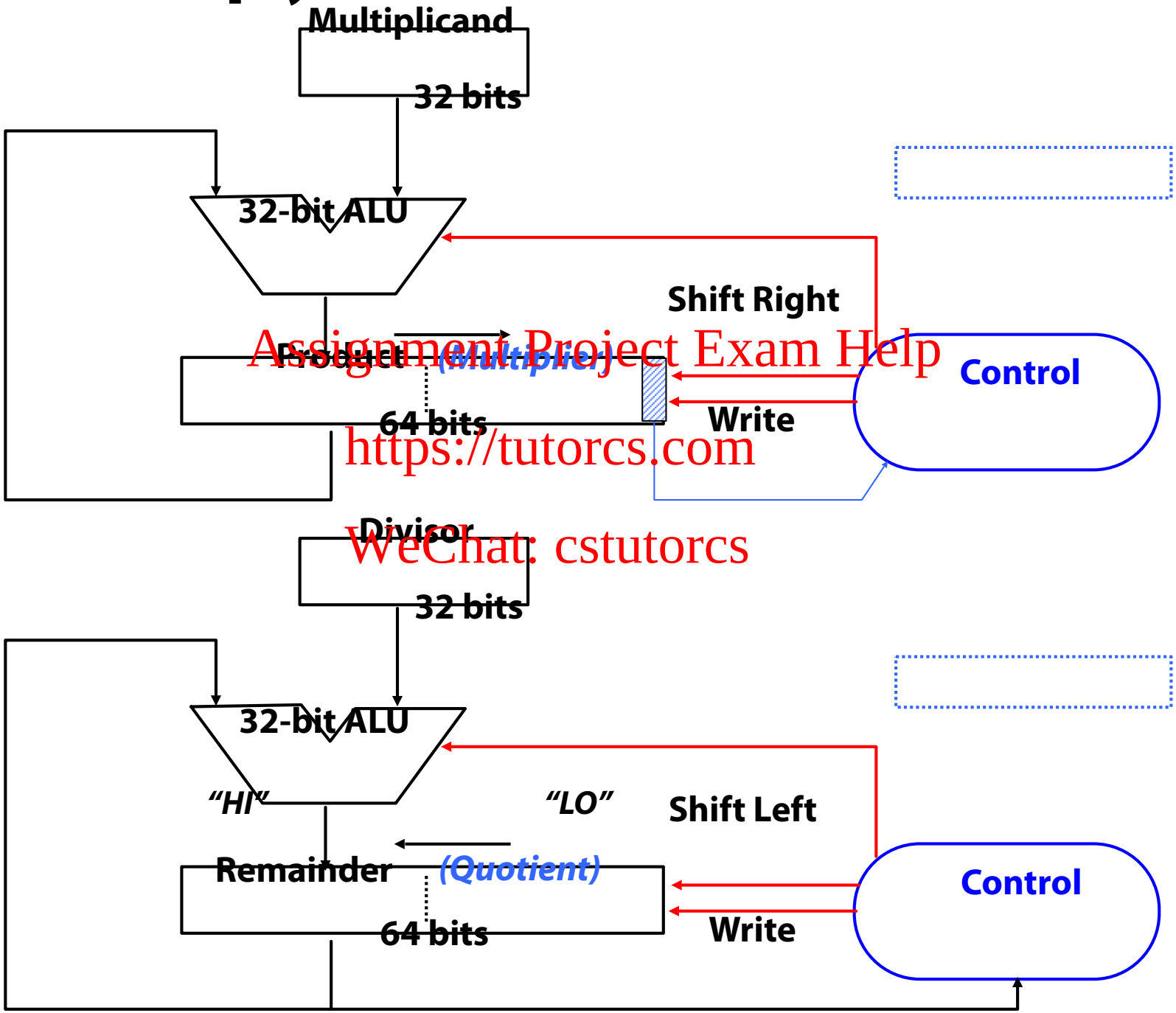
- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



Divide Algorithm V3



Final Multiply / Divide Hardware

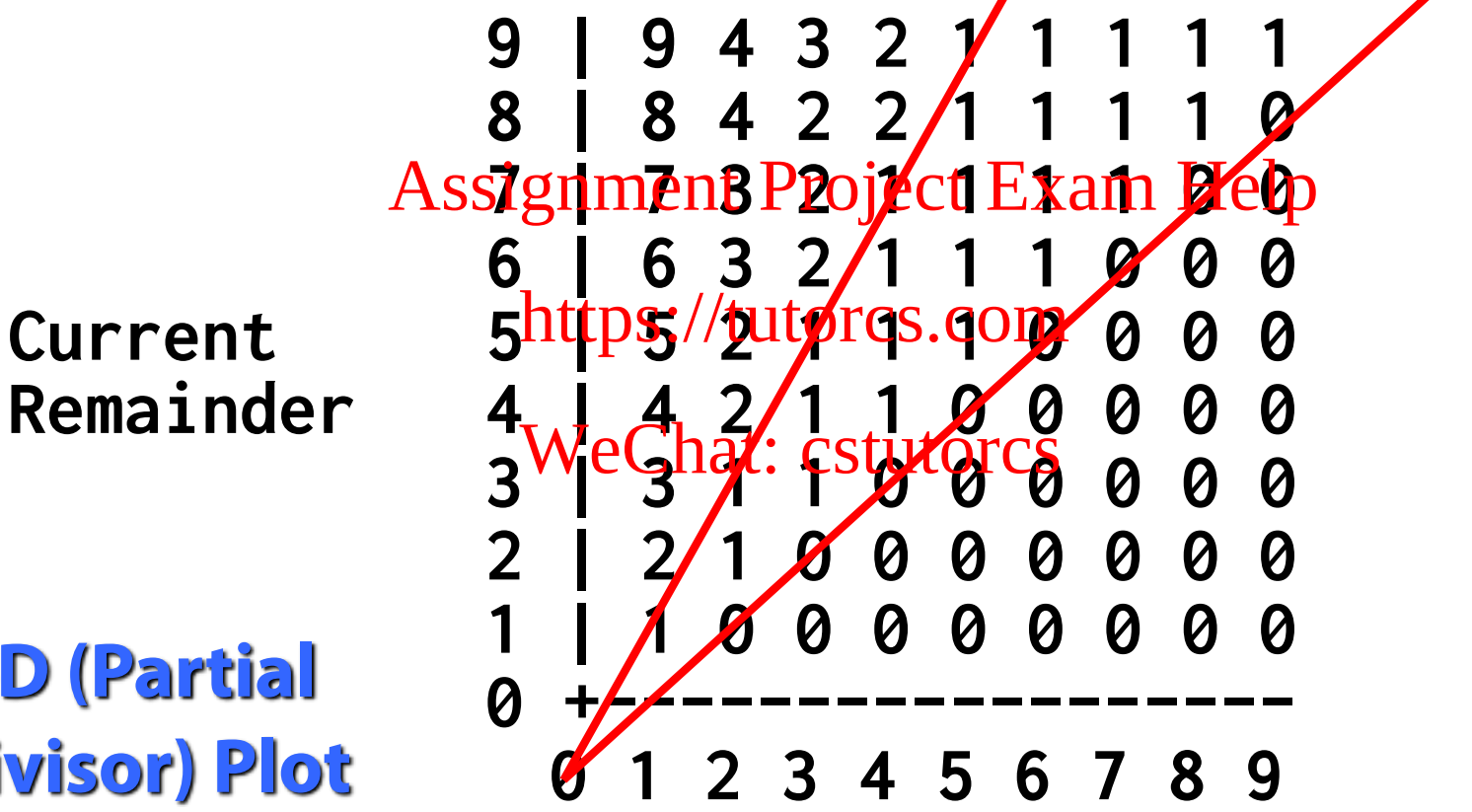


Observations on Divide Version 3

- 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

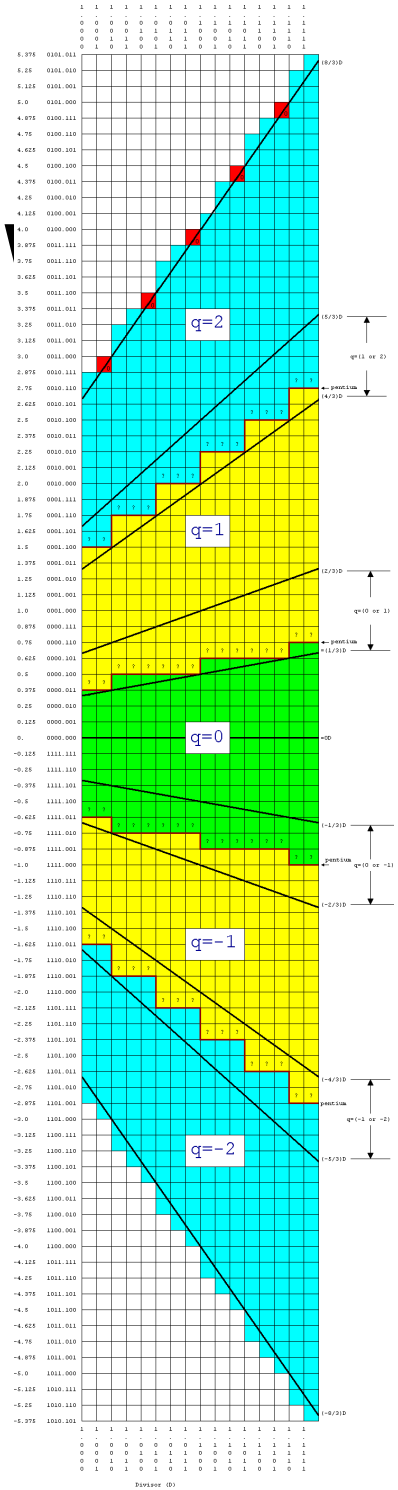
D. Sweeney of IBM, J.E. Robertson of the University of Illinois, and T.D. Tocher of Imperial College, London



P-D (Partial Divisor) Plot

SRT Division

- Intel Pentium divide implementation: SRT division w
  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

End of lecture (1:30 in, to allow project

Lecture 7:

Arithmetic 3/3

John Owens

Introduction to Computer Architecture

UC Davis EEC 170, Winter 2021

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

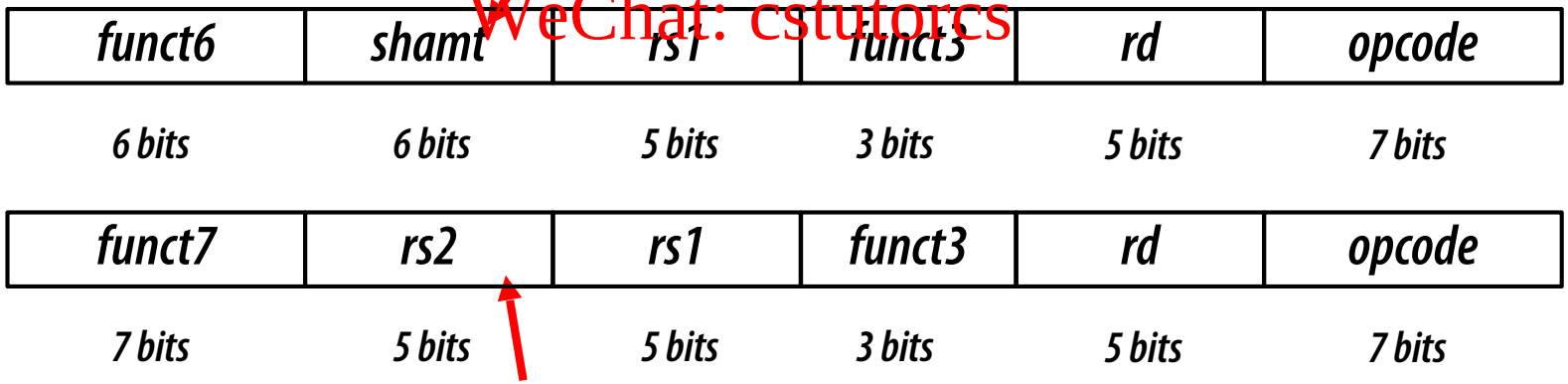
- Arithmetic operation:
  - Example: 00011 << 2 [3 left shift 2]
    - 00011 << 2 = 01100 = 12 = 2 * 4
  - Each bit shifted left == multiply by two
  - Example: 01010 >> 1 [10 right shift 1]
    - 01010 >> 1 = 00101 = 5 = 10/2
  - Each bit shifted right == divide by two
  - Why?
  - Compilers do this—"strength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

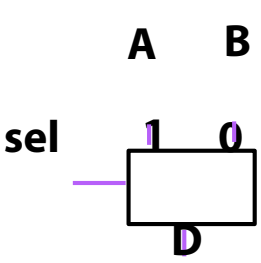
- How about right shift?
  - XXXX0000 >> 4 = 0000XXXX: Logical shift
    - Shifted in zero
  - 00110 (= 6) >> 1 = 00011 (3)
    11110 (= -2) >> 1 = 11111 (-1): Arithmetic shift
    - Shifted in sign bit
- RISC-V supports both logical and arithmetic:
  - slli, srai, srli: Shift amount taken from within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



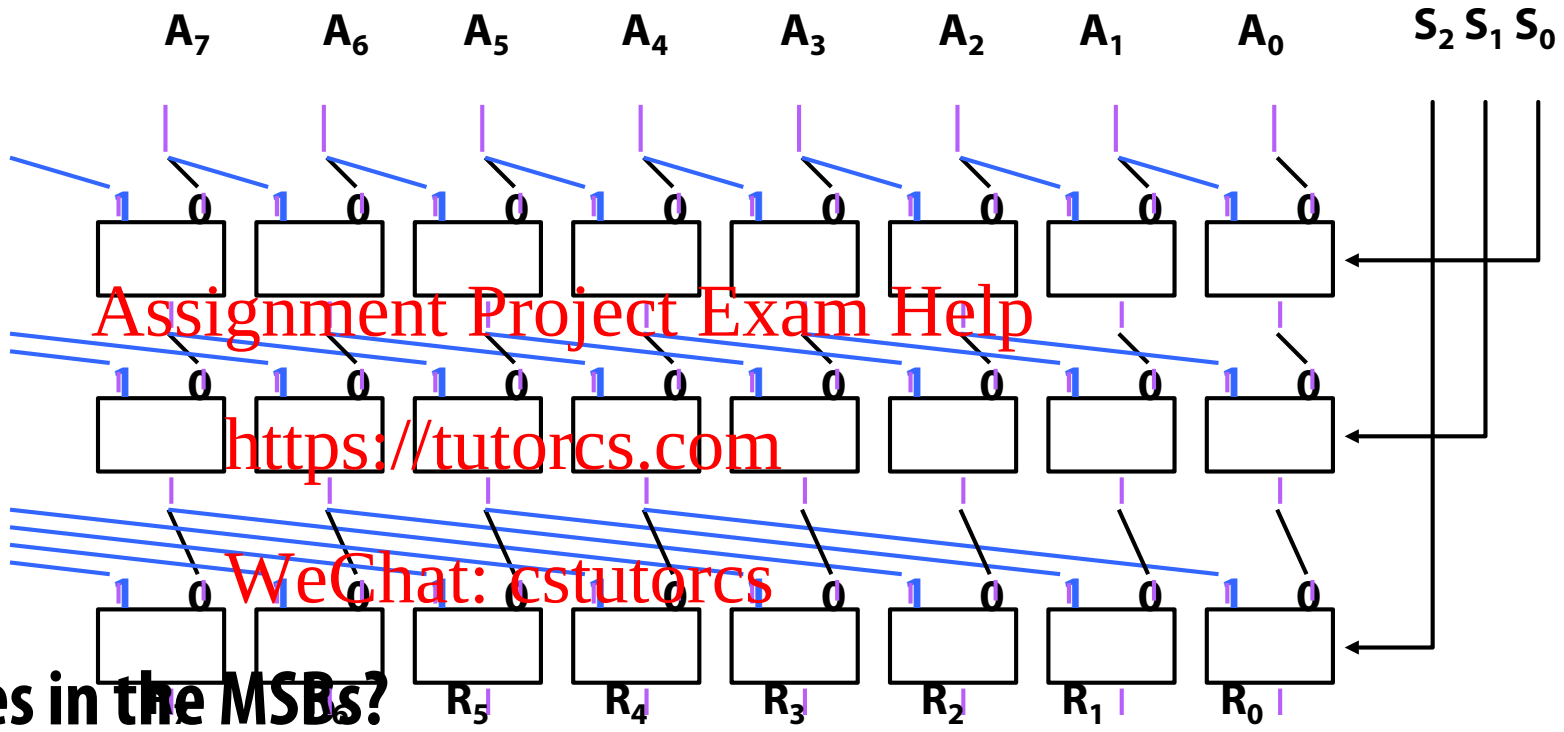
  - sll, sra, srl: shift amount taken from register ("variable")
  - How far can we shift with slli/srai/slli? With sll/sra/srl?

Combinational Shifter from MUXes

Basic Building Block



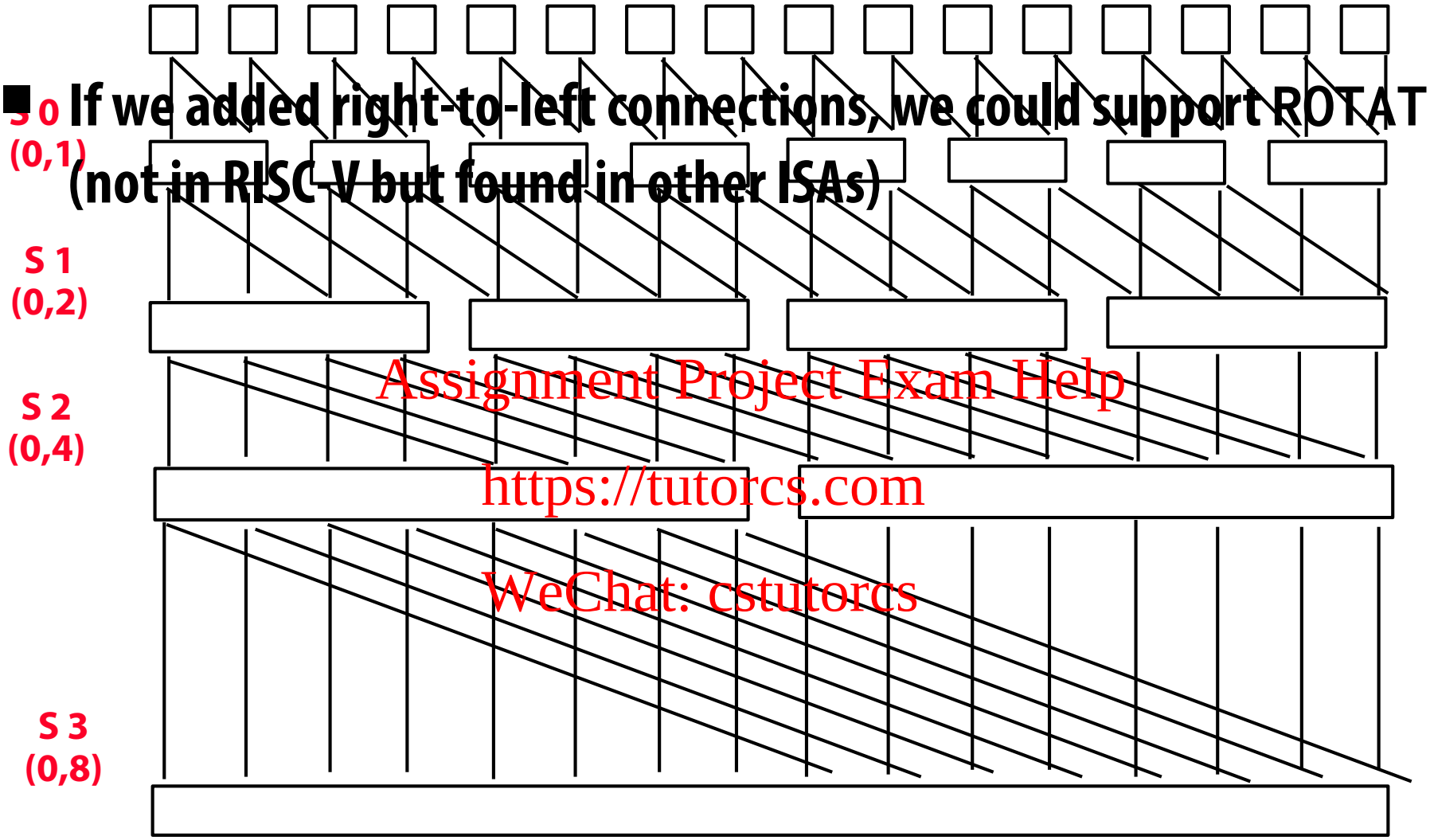
8-bit right shifter



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

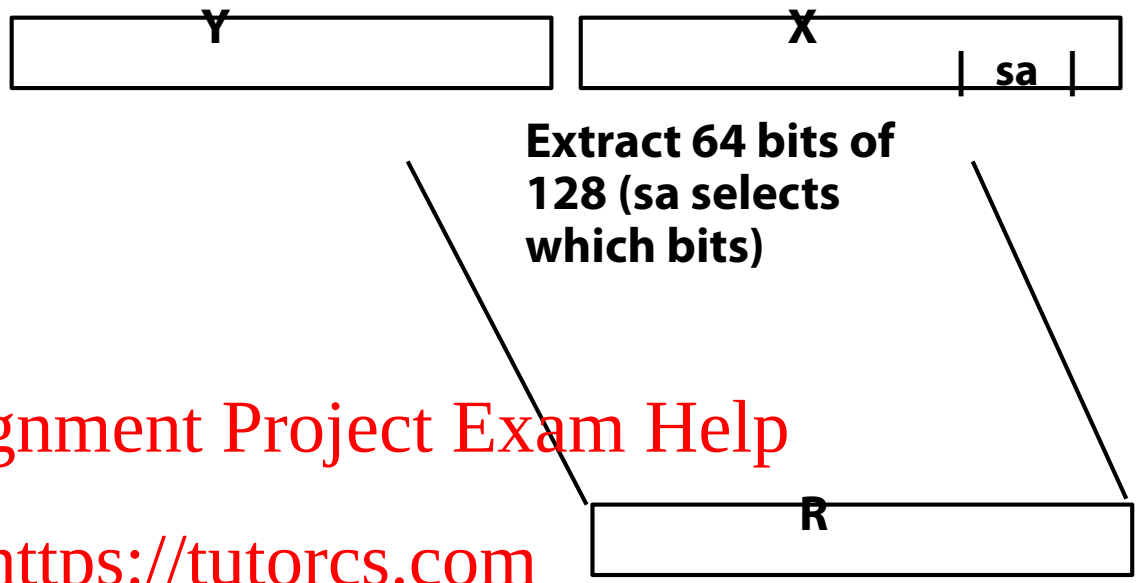
General Shift Right Scheme using 16 bit example

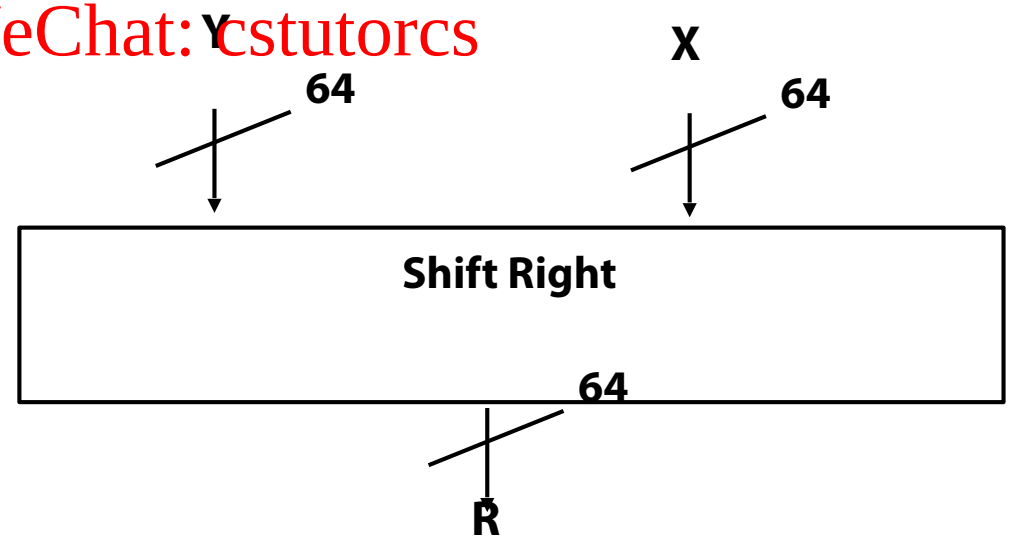
- If we added right-to-left connections, we could support ROTATE (not in RISC-V but found in other ISAs)



Funnel Shifter

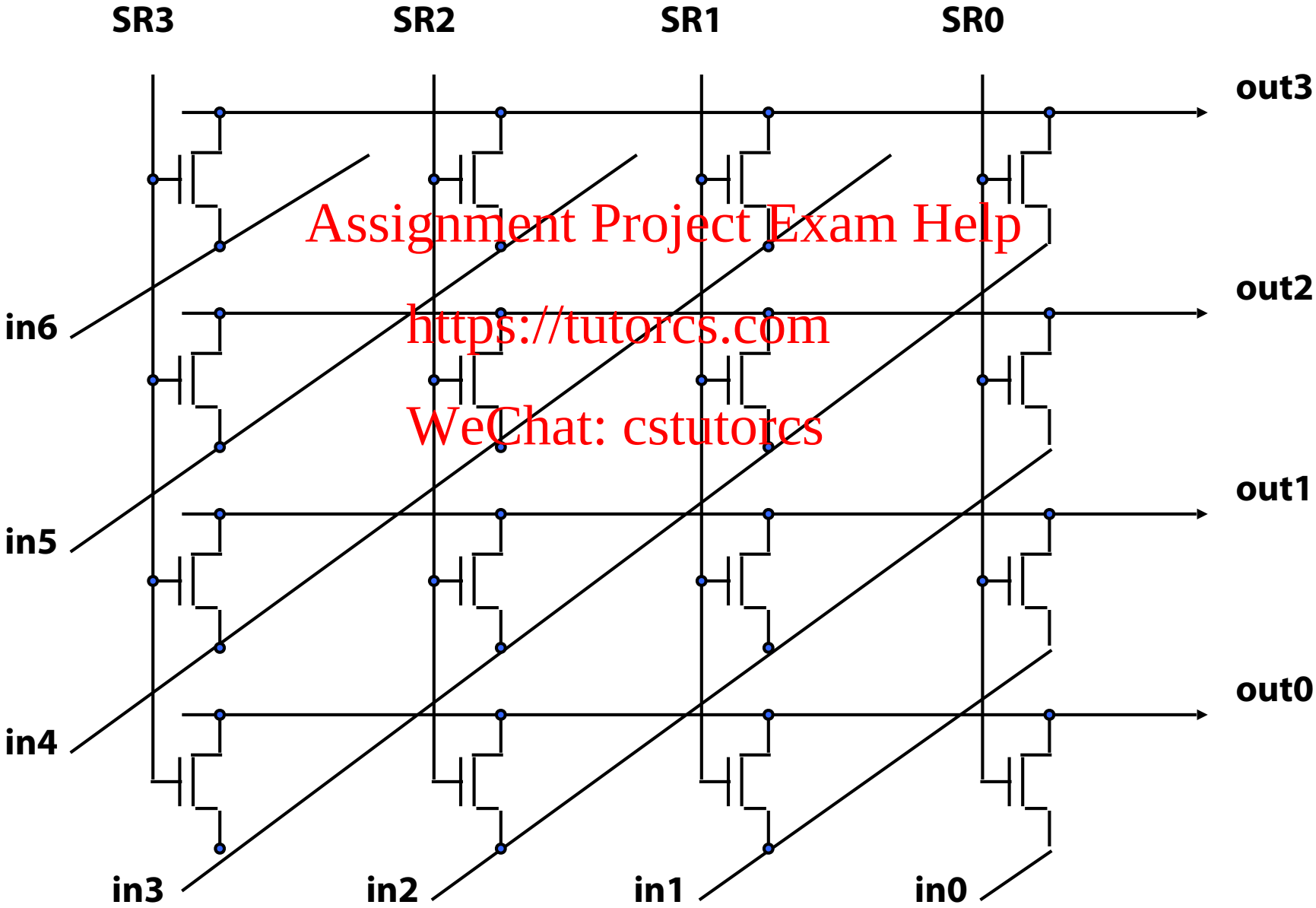
- Shift A by i bits
- Problem: Set Y, X, sa
- Logical:
- Arithmetic:
- Rotate:
- Left shifts:





Barrel Shifter

- Technology-dependent solutions: transistor per switch



Shifter Summary

- Shifts common in logical ops, also in arithmetic
- RISC-V (oops) 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

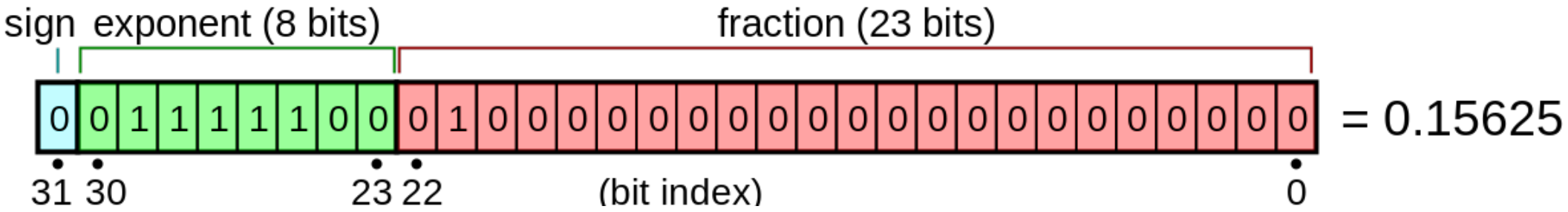
Floating Point

- 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

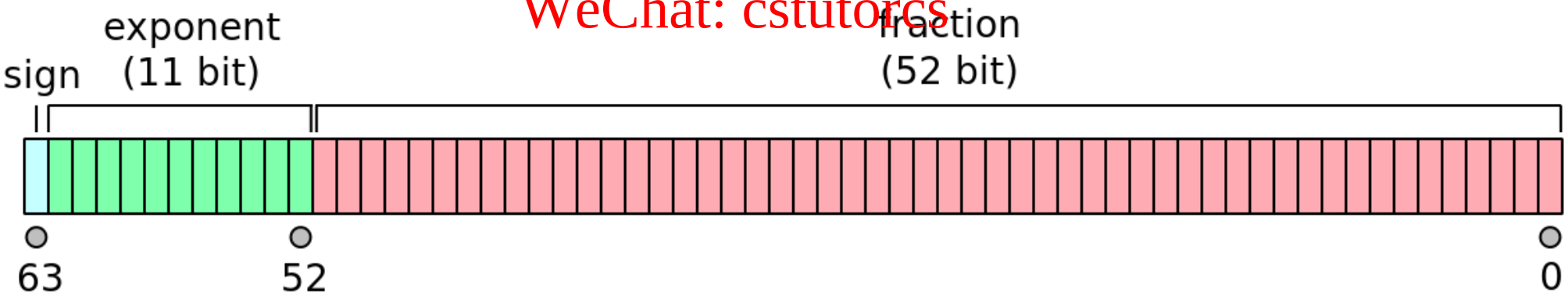
Floating Point Standard

- 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
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - 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

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^2
- 4. Round and renormalize if necessary
  - 1.002×10^2

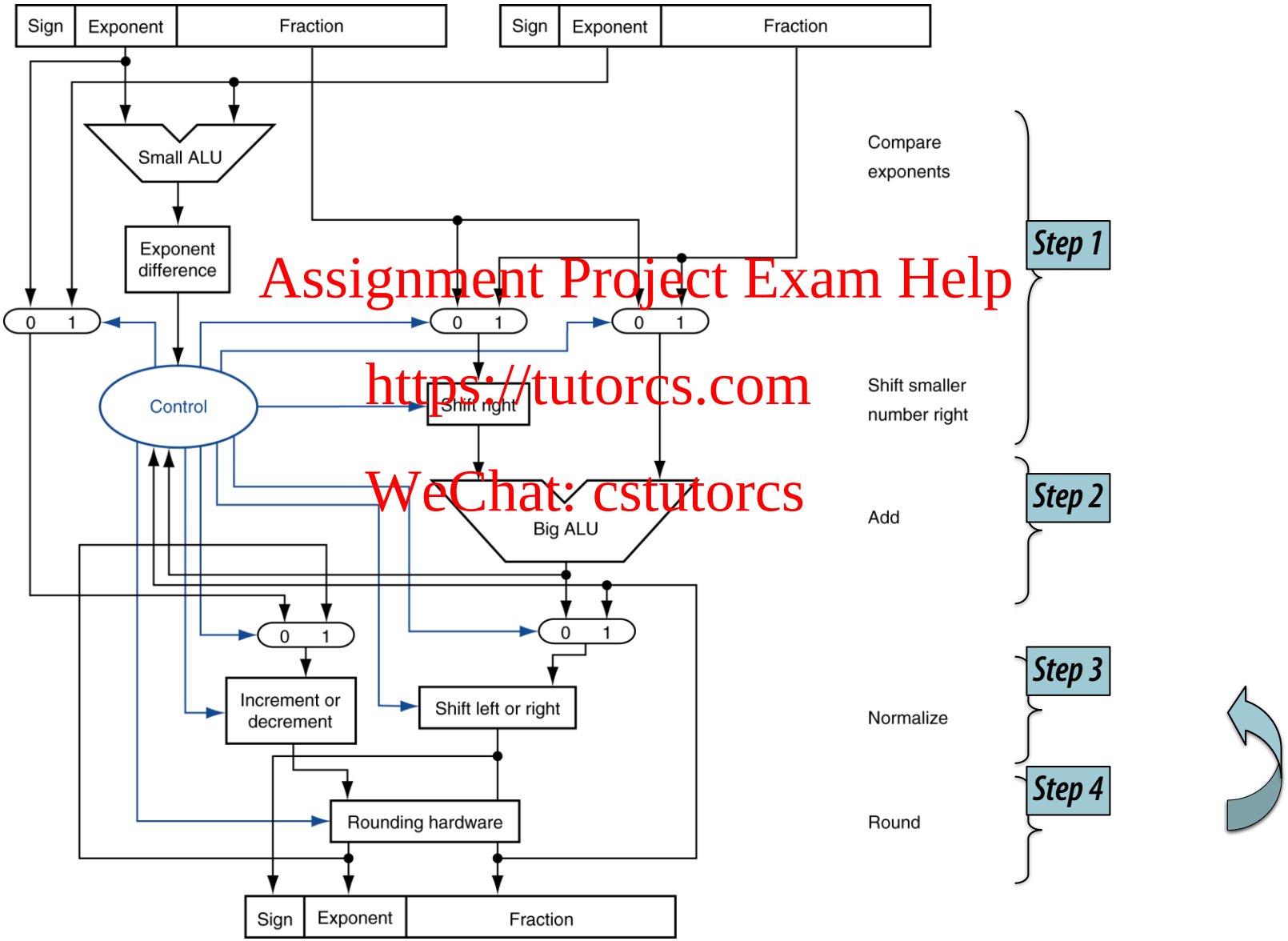
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + -5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in RISC-V

- Separate FP registers: f0, . . ., f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `flw, fld`
  - `fsw, fsd`

FP Instructions in RISC-V

- **Single-precision arithmetic**
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - **e.g.,** `fadds.s f2, f4, f6`
- **Double-precision arithmetic**
  - `fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`
    - **e.g.,** `fadd.d f2, f4, f6`
- **Single- and double-precision comparison**
  - `feq.s, flt.s, fle.s`
  - `feq.d, flt.d, fle.d`
  - **Result is 0 or 1 in integer destination register**
    - **Use beq, bne to branch on comparison result**
- **Branch on FP condition code true or false**
  - `B.cond`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in f10, result in f10, literals in global memory space

- Compiled RISC-V code:


```
f2c:
    flw    f0,const5(x3)  // f0 = 5.0f
    flw    f1,const9(x3)  // f1 = 9.0f
    fdiv.s f0, f0, f1  // f0 = 5.0f / 9.0f
    flw    f1,const32(x3) // f1 = 32.0f
    fsub.s f10,f10,f1  // f10 = fahr - 32.0
    fmul.s f10,f0,f10  // f10 = (5.0f/9.0f) * (fahr-32.0f)
    jalr   x0,0(x1)    // return
```

FP Example: Array Multiplication

- $C = C + A \times B$
  - All 32×32 matrices, 64-bit double-precision elements
- C code:
- ```
  void mm (double c[][],
           double a[][], double b[][]) {
    size_t i, j, k;
    for (i = 0; i < 32; i = i + 1)
      for (j = 0; j < 32; j = j + 1)
        for (k = 0; k < 32; k = k + 1)
          c[i][j] = c[i][j]
                  + a[i][k] * b[k][j];
  }
  ```
  - Addresses of c, a, b in x10, x11, x12, and i, j, k in x5, x6, x7
  ```

FP Example: Array Multiplication

- **RISC-V code:**

```
mm:...
            li    x28,32        // x28 = 32 (row size/loop end)
           li    x5,0          // i = 0; initialize 1st for loop
     L1:   li    x6,0          // j = 0; initialize 2nd for loop
     L2:   li    x7,0          // k = 0; initialize 3rd for loop
          slli  x30,x5,5      // x30 = i * 2**5 (size of row of c)
              add   x30,x30,x6    // x30 = i * size(row) + j
             slli  x30,x30,3     // x30 = byte offset of [i][j]
            add   x30,x10,x30   // x30 = byte address of c[i][j]
                    fld   f0,0(x30)     // f0 = c[i][j]
    L3:   slli  x29,x7,5      // x29 = k * 2**5 (size of row of b)
              add   x29,x29,x6    // x29 = k * size(row) + j
             slli  x29,x29,3     // x29 = byte offset of [k][j]
            add   x29,x12,x29   // x29 = byte address of b[k][j]
                    fld   f1,0(x29)     // f1 = b[k][j]
```

FP Example: Array Multiplication

```
                                …
slli   x29,x5,5     // x29 = i * 2**5 (size of row of a)
    add    x29,x29,x7   // x29 = i * size(row) + k
   slli   x29,x29,3    // x29 = byte offset of [i][k]
  add  x29,x11,x29  // x29 = byte address of a[i][k]
        fld    f2,0(x29)    // f2 = a[i][k]
    fmul.d f1, f2, f1   // f1 = a[i][k] * b[k][j]
fadd.d f0, f0, f1   // f0 = c[i][j] + a[i][k] * b[k][j]
        addi   x7,x7,1      // k = k + 1
     bltu   x7,x28,L3    // if (k < 32) go to L3
        fsd    f0,0(x30)    // c[i][j] = f0
         addi   x6,x6,1       // j = j + 1
     bltu   x6,x28,L2    // if (j < 32) go to L2
         addi   x5,x5,1       // i = i + 1
     bltu   x5,x28,L1    // if (i < 32) go to L1
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), ...
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

- **Optional variations**
  - **I: integer operand**
  - **P: pop operand from stack**
  - **R: reverse operand order**
  - **But not all combinations allowed**

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Matrix Multiply

- **Unoptimized code:**

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {
6.     double cij = C[i+j*n]; /* cij = C[i][j] */
7.     for (int k = 0; k < n; k++)
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.     C[i+j*n] = cij; /* C[i][j] = cij */
10.   }
11. }
```

Matrix Multiply

- x86 assembly code:

```
1. vmovsd (%r10),%xmm0  # Load 1 element of C into %xmm0
2. mov %rsi,%rcx        # register %rcx = %rsi
3. xor %eax,%eax        # register %eax = 0
4. vmovsd (%rcx),%xmm1  # Load 1 element of B into %xmm1
5. add %r9,%rcx         # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
7. add $0x1,%rax        # register %rax = %rax + 1
8. cmp %eax,%edi        # Compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>  # jump if %eax > %edi
11. add $0x1,%r11d      # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

Matrix Multiply

- **Optimized C code:**

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
6.    __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.    for( int k = 0; k < n; k++ )
8.     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                        _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                      _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```

Matrix Multiply

- **Optimized x86 assembly code:**

```
1. vmovapd (%r11),%ymm0       # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx              # register %rcx = %rbx
3. xor %eax,%eax              # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax              # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx               # register %rcx = %rcx + %r9
8. cmp %r10,%rax              # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0   # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>       # jump if not %r10 != %rax
11. add $0x1,%esi             # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)      # Store %ymm0 into 4 C elements
```

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers:
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - $11111011_2 >> 2 = 11111110_2 = -2$
      - Rounds toward $-\infty$
  - c.f. $11111011_2 >>> 2 = 00111110_2 = +62$

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹

- The Intel Pentium FDIV bug

  - The market expects accuracy

  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow

Break

Administrivia

- My office hour is in the Coffee House, not the Silo. The Silo is closed for construction.
- Maybe oops: https://en.wikipedia.org/wiki/Kahan_summation_algorithm
- Tell me about errors in slides / in homework
- Anyone try RARS?
- Midterm philosophy
- No typing on the midterm
- Bring a calculator

Problem: Design a "fast" ALU for the RISC-V ISA

- Requirements?
  - Must support the Arithmetic / Logic operations
  - Tradeoffs of cost and speed based on frequency of occurrence, hardware budget

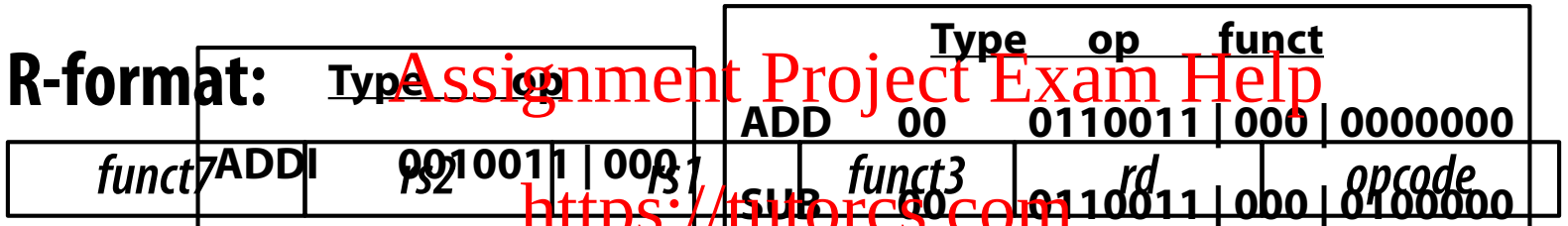
RISC-V ALU requirements

- Add, Sub, Addl, Addl
  - => 2's complement adder/sub
- And, Or, Andl, Orl, Xor, Xori
  - => Logical AND, logical OR, XOR
- SLTI, SLTIU (set less than)
  - => 2's complement adder with inverter, check sign bit of result
- See ALU from COD5E, appendix A.5

MIPS arithmetic instruction format

- **I-format:**

- **R-format:**



Type	op
ADDI	0010011 | 000
SLTI	0010011 | 010
SLTIU	0010011 | 011
ANDI	0010011 | 111
ORI	0010011 | 110
XORI	0010011 | 100

Type	op	funct
ADD	00	0110011 | 000 | 0000000
SUB	00	0110011 | 000 | 0100000
AND	00	0110011 | 111 | 0000000
OR	00	0110011 | 110 | 0000000
XOR	00	0110011 | 100 | 0000000
SLT	00	0110011 | 010 | 0000000
SLTU	00	0110011 | 011 | 0000000

Design Trick: divide & conquer

- Trick: Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU

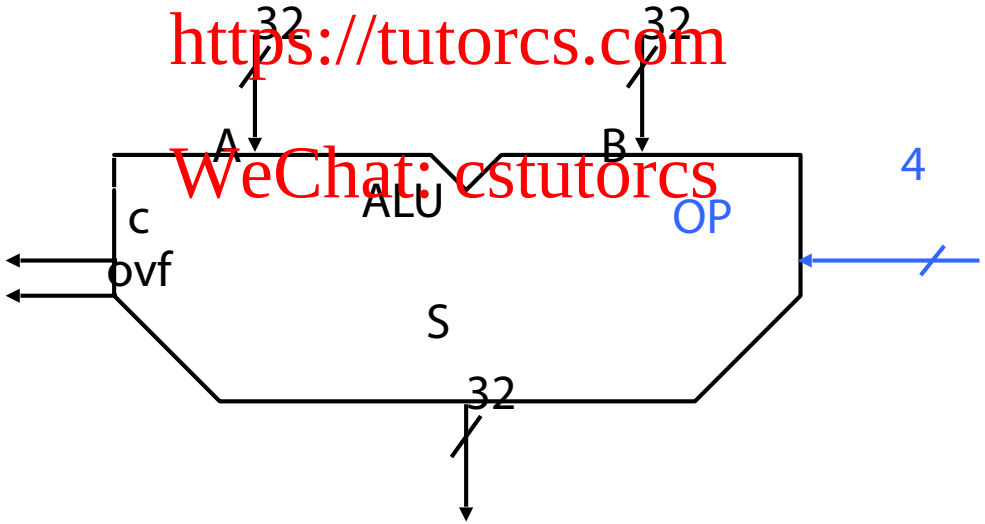
	Type	op
ADDI	0010011	000
ANDI	0010011	111
ORI	0010011	110
XORI	0010011	100
SLTI	0010011	010
SLTIU	0010011	011

7 operations (could be 3 bits, but really 4)

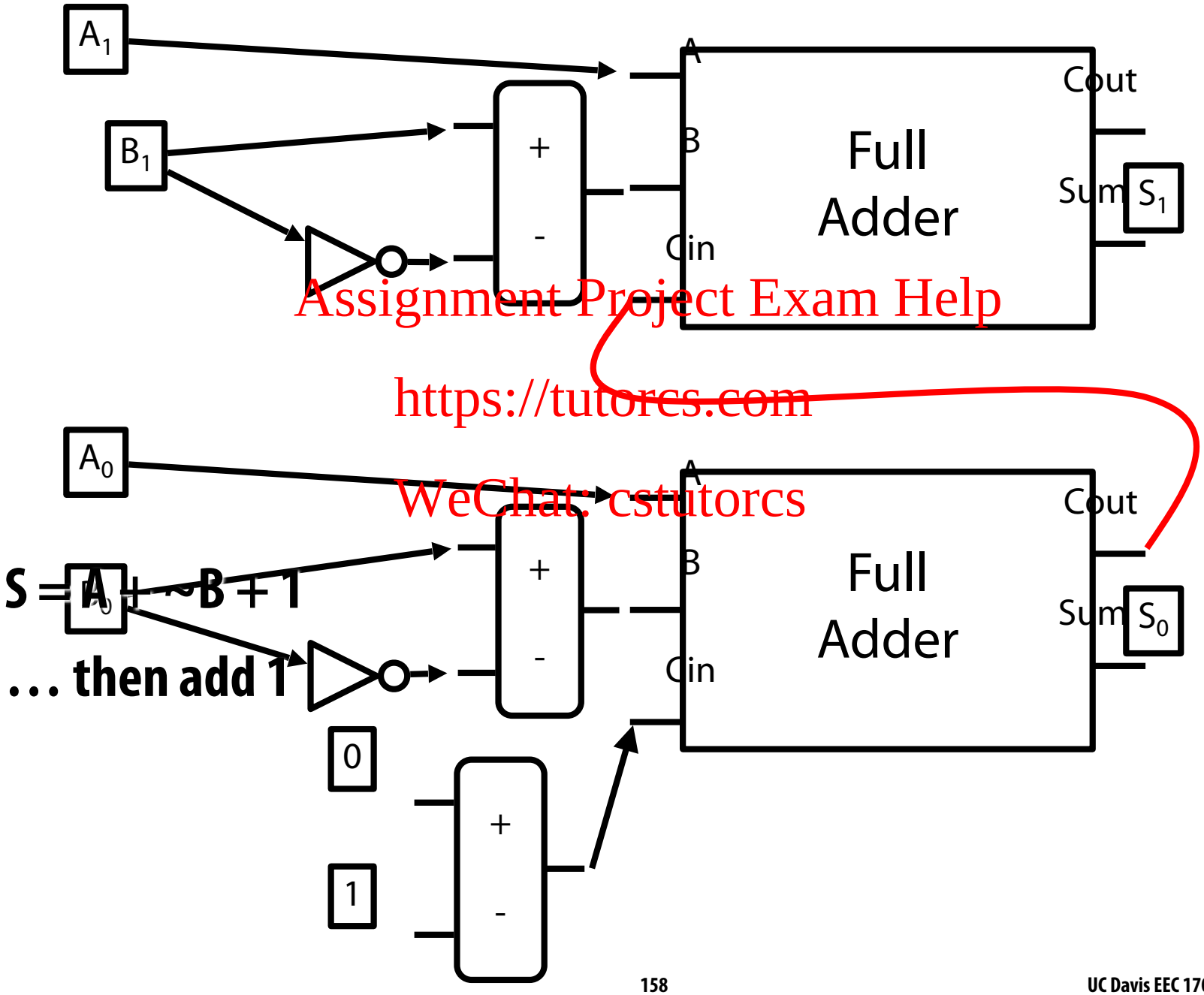
	Type	op	funct
ADD	00	0110011	000 | 0000000
SUB	01	0110011	000 | 0100000
AND	00	0110011	111 | 0000000
OR	00	0110011	110 | 0000000
XOR	00	0110011	100 | 0000000
SLT	00	0110011	010 | 0000000
SLTU	00	0110011	011 | 0000000

Let's Build a ALU

- **Functional Specification:**
  - **inputs:** 2 x 32-bit operands A, B, 4-bit OPeration
  - **outputs:** 32-bit result S, 1-bit carry, 1 bit overflow
  - **operations:** add, sub, and, or, xor, slt, sltu



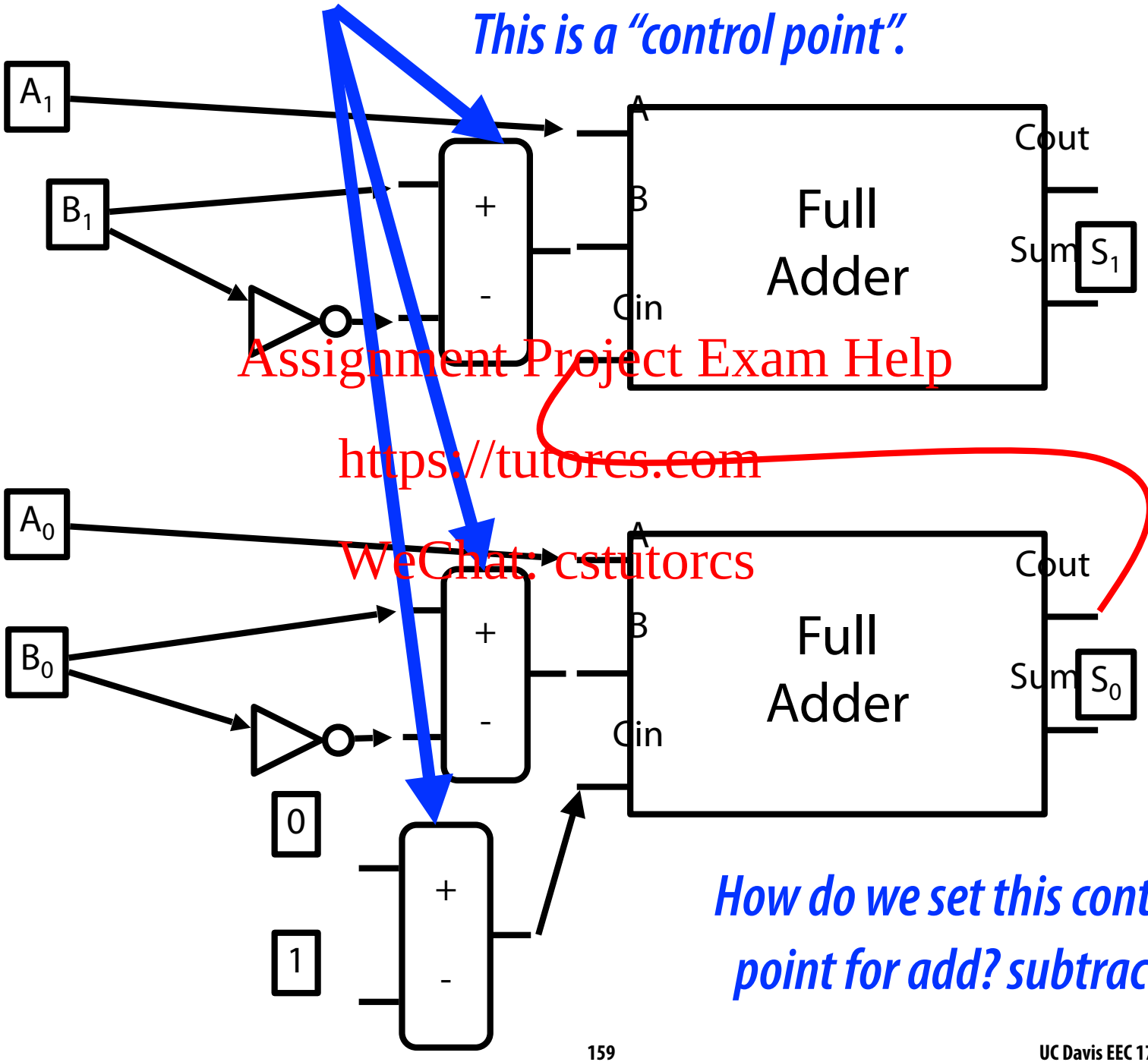
We already know how to do add/sub



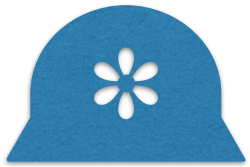
Control for +/-

One bit controls three muxes.

This is a "control point".

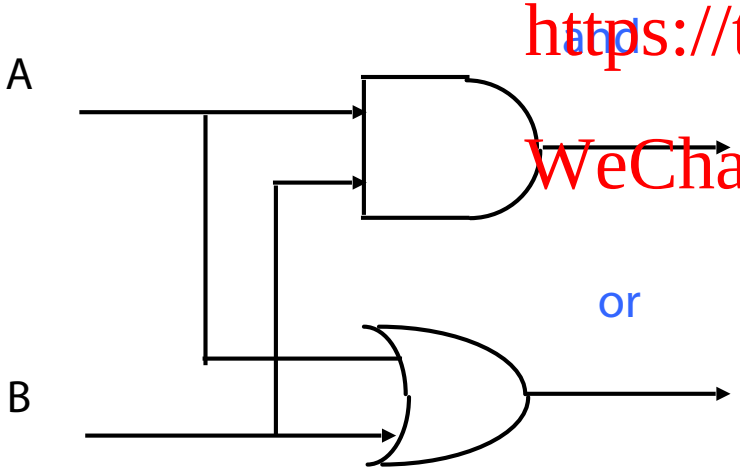


How do we set this control point for add? subtract?



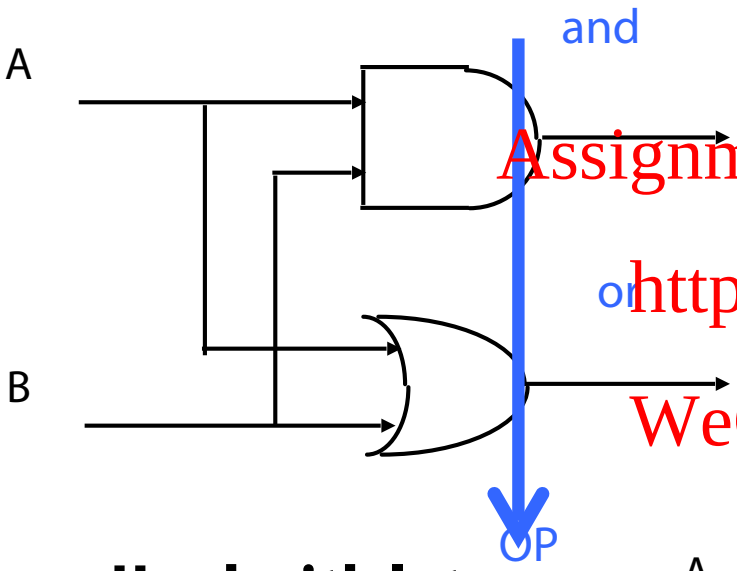
AND and OR

- Consider ALU that supports two functions, AND and OR
- How do we do this?



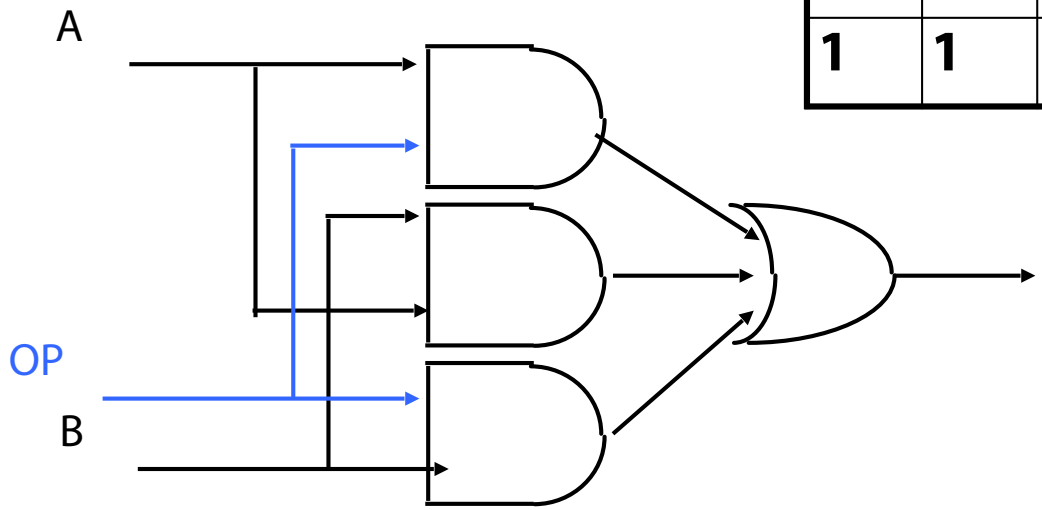
AND and OR

- Combinational logic:
  - Control bit OP is 0 for AND, 1 for OR



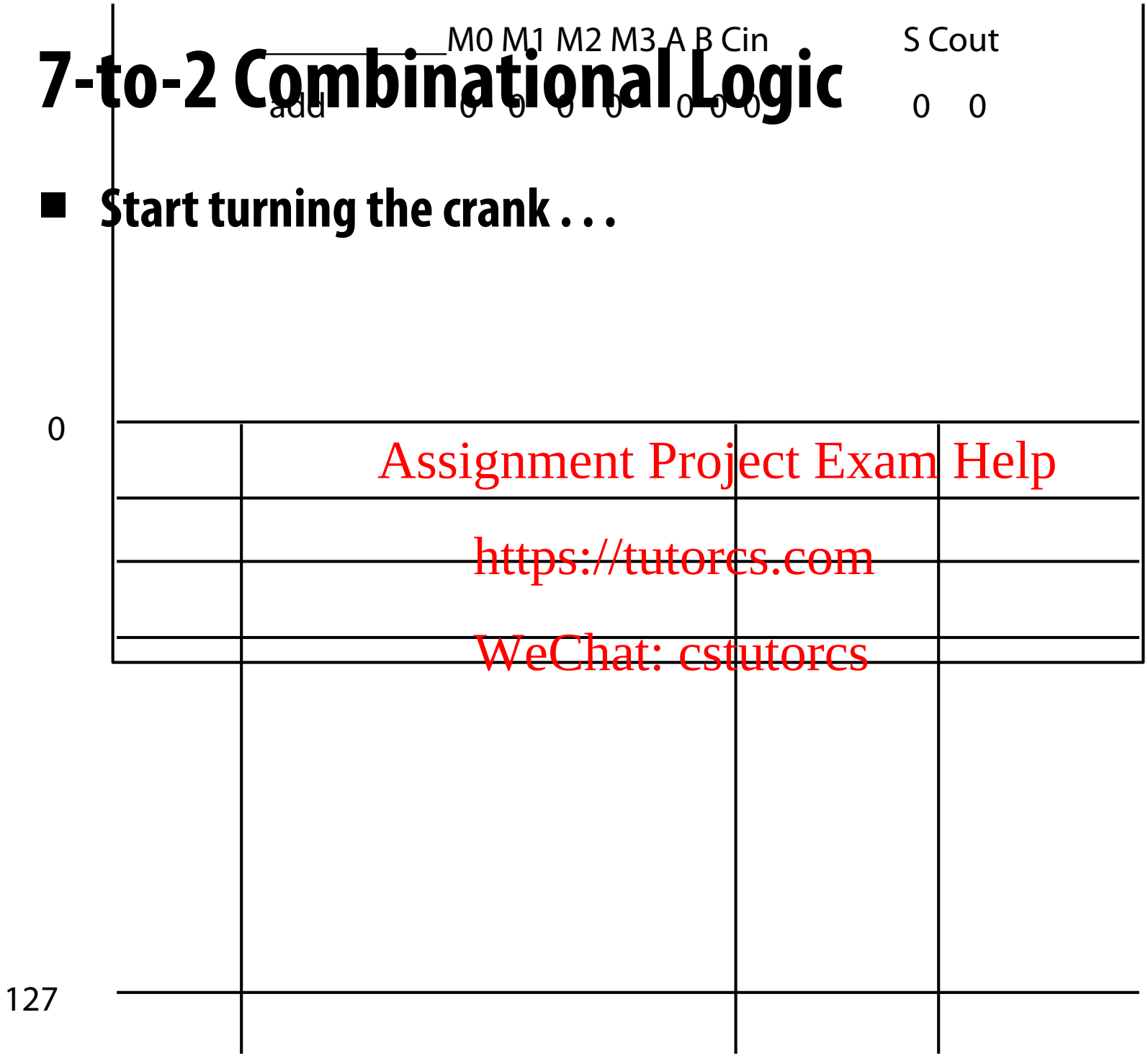
A	B	OP	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- Hard with lots of functions! But let's do it anyway.



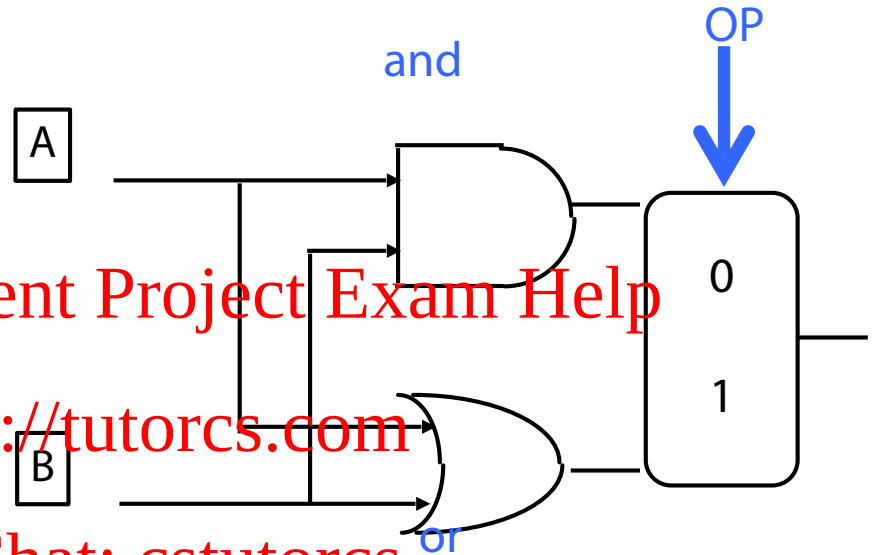
7-to-2 Combinational Logic

- Start turning the crank . . .



AND and OR

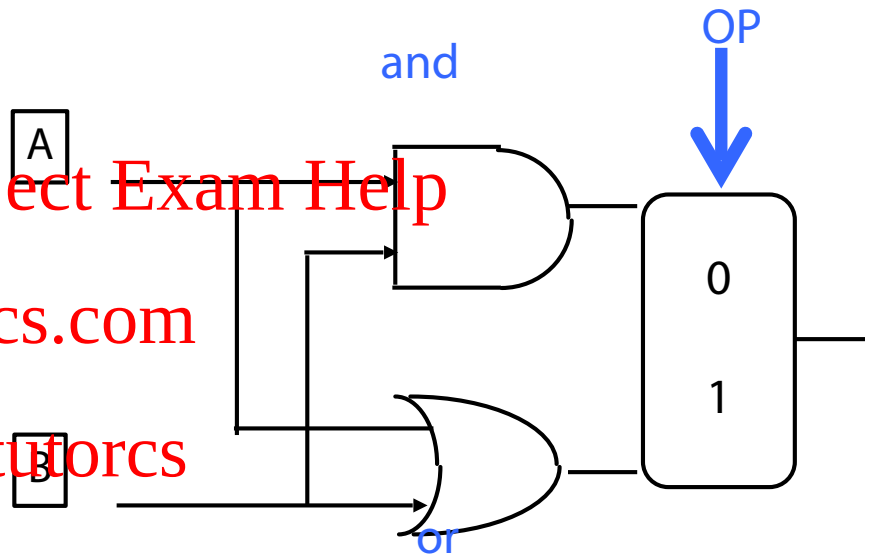
- Instead, generate several functions and use control bits to select using a mux



- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - For our purposes, ease of comprehension is important

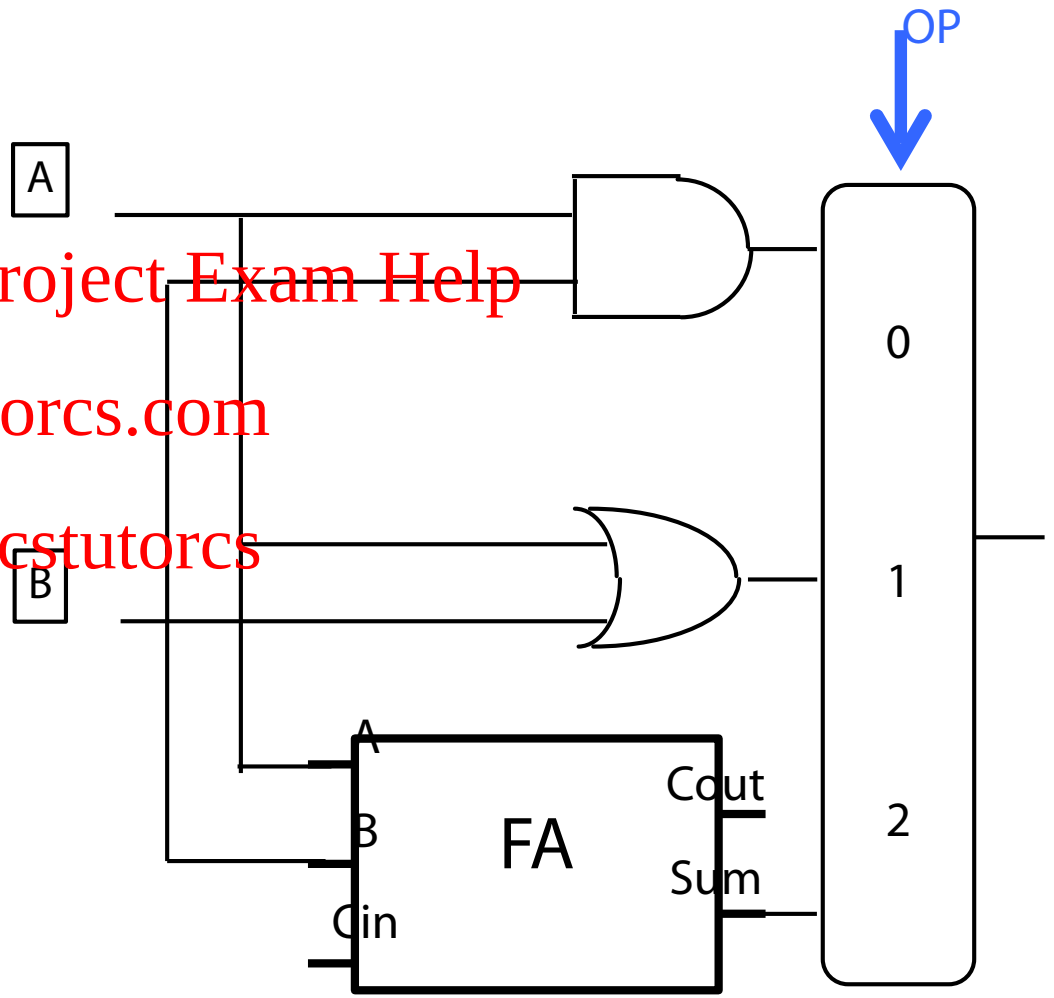
Supporting More Functions

- With the mux approach, it's easy to add other functions
  - Like add
  - To add more, just enlarge mux
  - Control signals in mux, not datapath



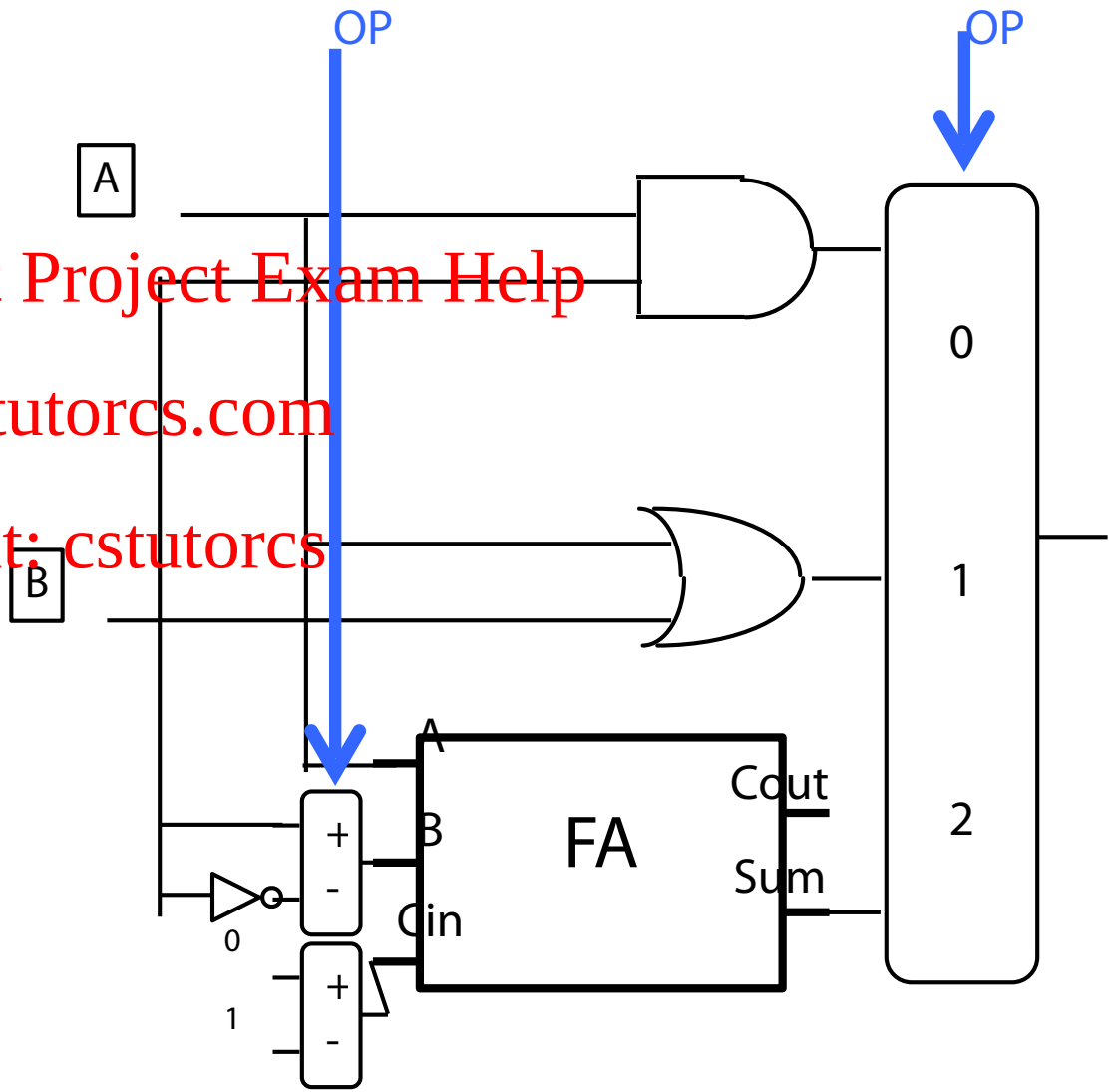
Supporting More Functions

- With the mux approach, it's easy to add other functions
  - Like add
  - To add more, just enlarge mux
  - Control signals in mux, not datapath



Supporting More Functions

- With the mux approach, it's easy to add other functions
  - Like add
  - To add more, just enlarge mux (or put more muxes elsewhere)
  - Control signals in muxes, not datapath



Tailoring the ALU to RISC-V

- Need to support the set-on-less-than instruction (slt)
  - slt produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
  - So now we've got a - b as our result. How does this translate to slt operation? What do we have to test and where does it go?
  - We test
  - To produce the proper result, it goes in

Tailoring the ALU to RISC-V

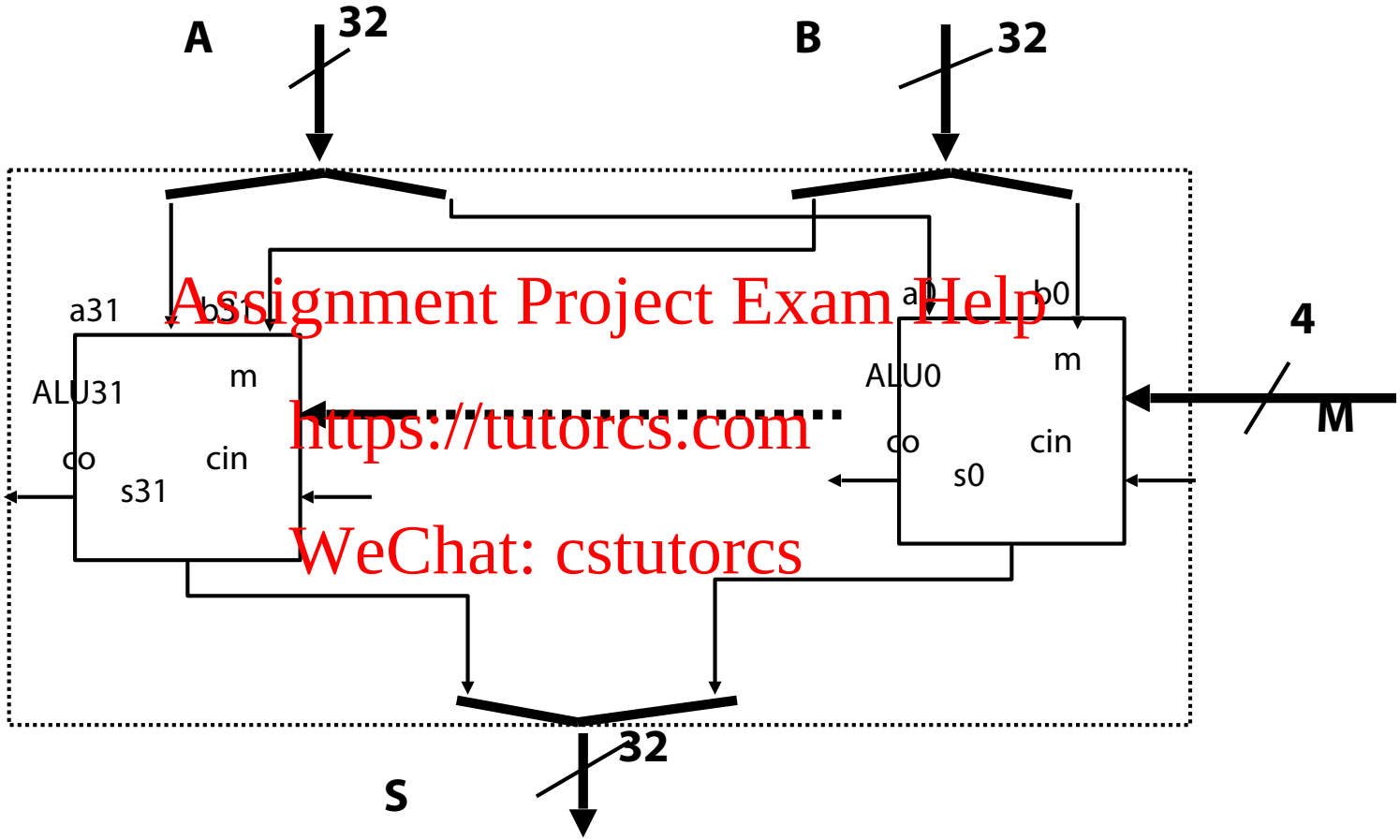
- Need to support the set-on-less-than instruction (slt)
  - slt produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
  - So now we've got a bus our result. How does this translate to slt operation? What do we have to test and where does it go?
  - We test
    the highest (sign) bit (S[31])
  - To produce the proper result, it goes in
    the lowest bit (S[0])

Why do you think the MIPS-V designers

Tailoring the ALU to the MIPS

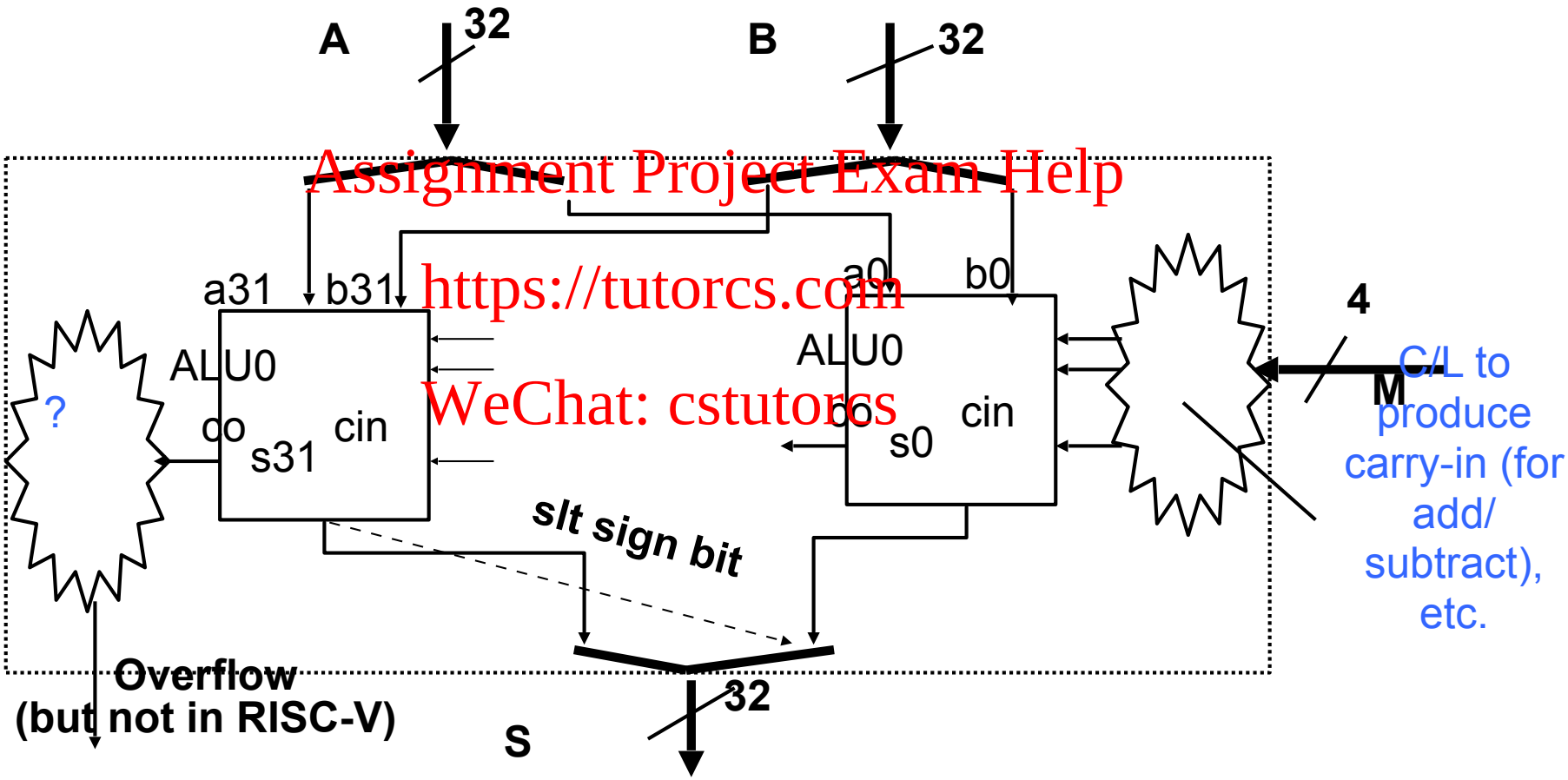
- Need to support test for equality (beq $t5, $t6, LABEL)
  - use subtraction: (a-b) = 0 implies a = b
  - How do we test if the product is zero?

Original Diagram: bit-slice ALU



Revised Diagram

- LSB and MSB need to do a little extra

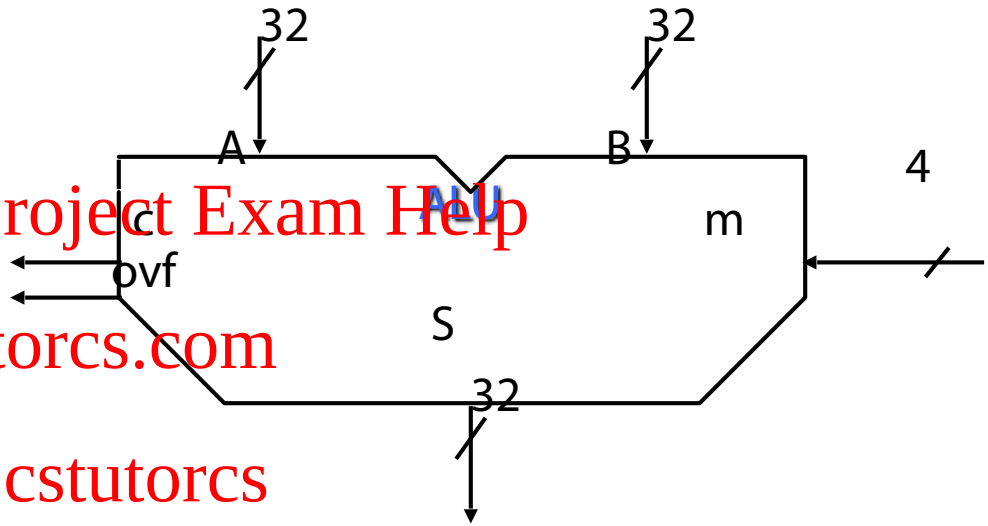


Behavioral Representation: Verilog

```
module ALU(A, B, m, S, c, ovf);
input [0:31] A, B;
input [0:3] m;
output [0:31] S;
output c, ovf;

reg [0:31] S;
reg c, ovf;

always @(A, B, m) begin
 case (m)
    0: S = A + B;
    1: S = A - B;
    2: S = ...
 . . .
end
endmodule
```



Conclusion

- We can build an ALU to support the RISC-V instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series

    (on the "critical path" or the "deepest level of logic")

MIPS Opcode Map

31...29 \ 28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

Opcode

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

SPECIAL function

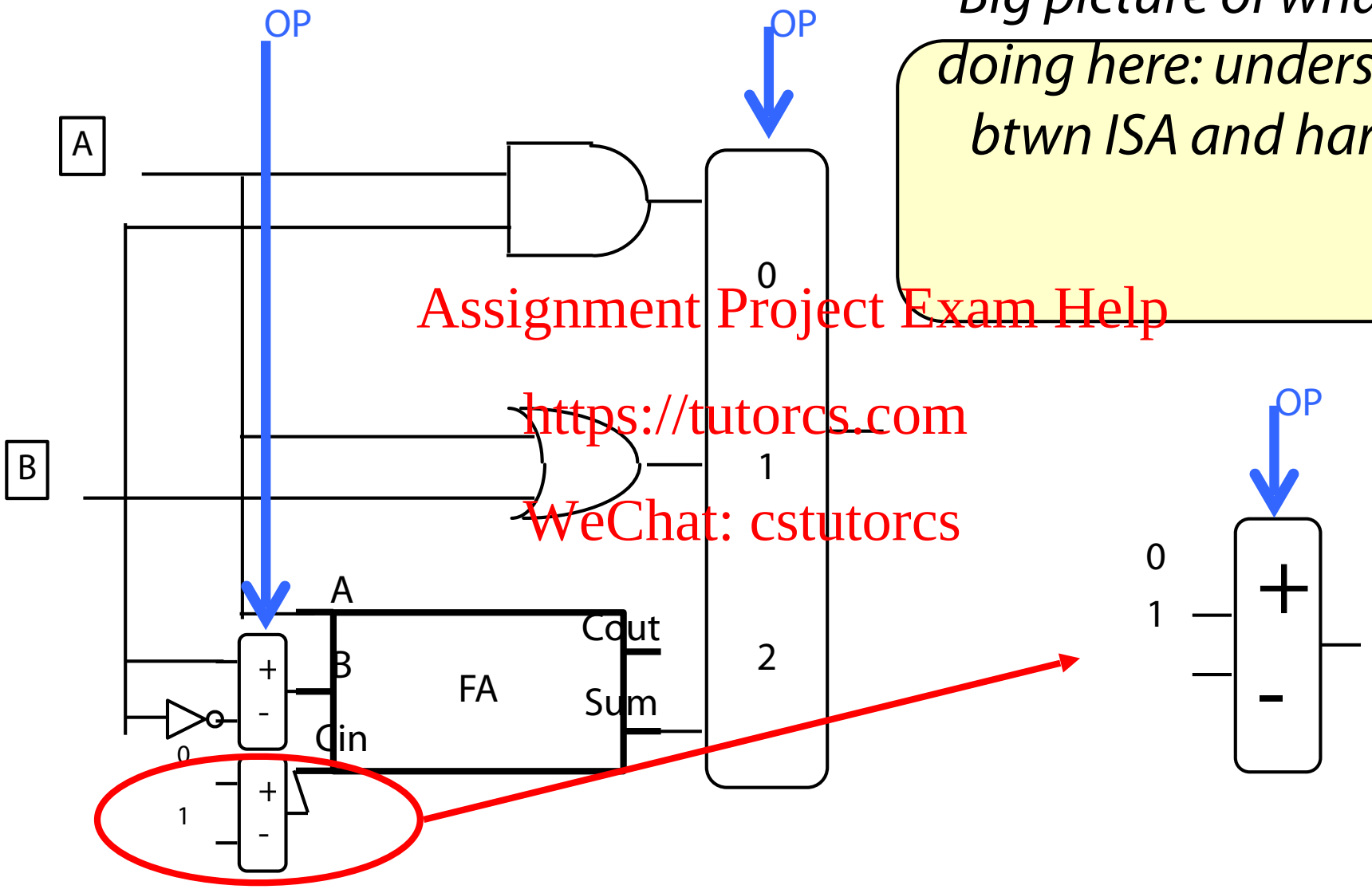
20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

REGIMM rt

[from MIPS R4000 Microprocessor User's Manual / Joe Heinrich]

Encodings for ADD, SUB

Big picture of what we're doing here: understand tie btwn ISA and hardware



MIPS Opcode Map

Opcode

31...29 \ 28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

SPECIAL function

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

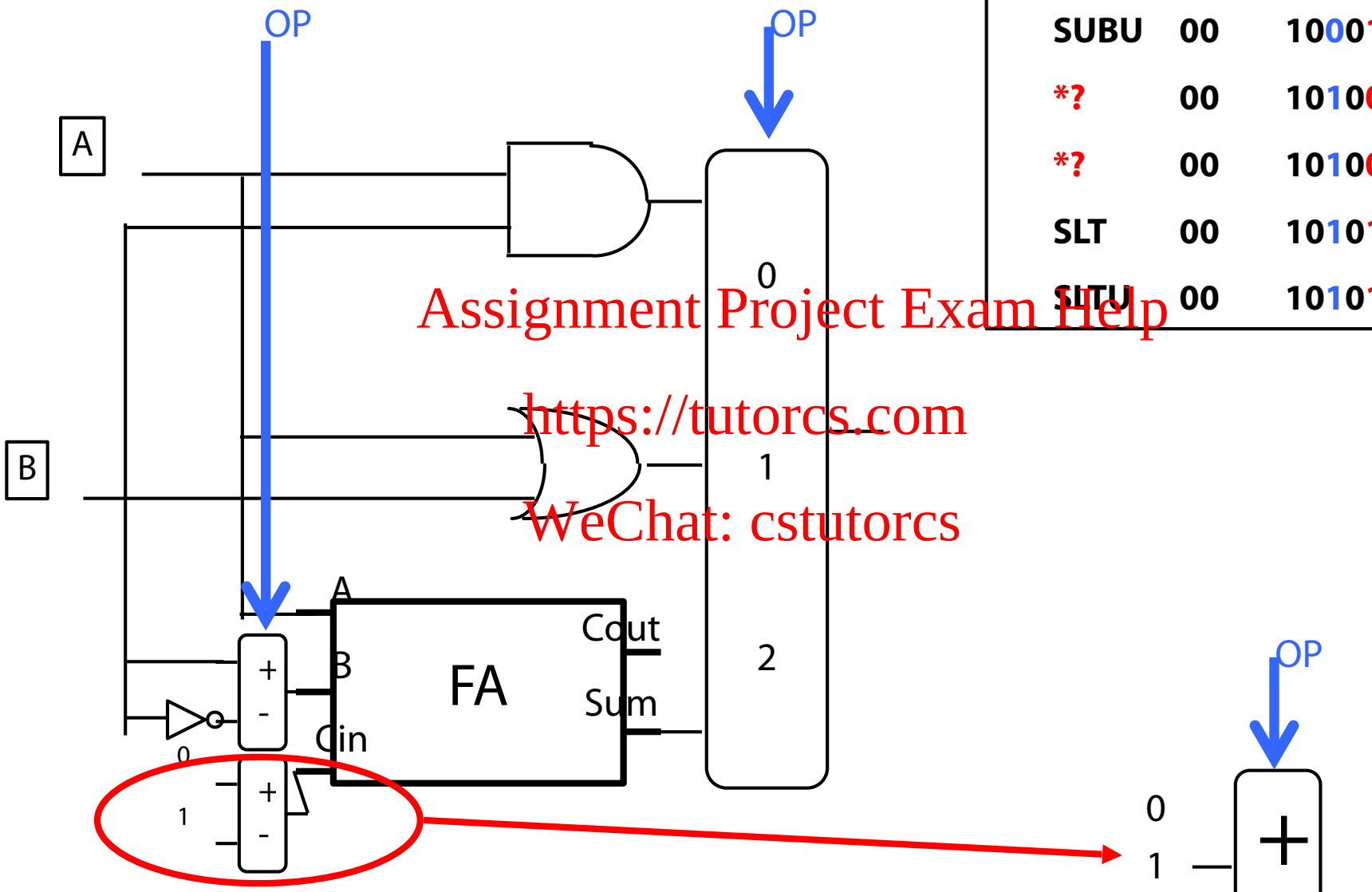
REGIMM rt

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

[from MIPS R4000 Microprocessor User's Manual / Joe Heinrich]

MIPS (really) Encodings for ADD, SUB, SLT

ADD	00	100000 (040)
ADDU	00	100001 (041)
SUB	00	100010 (042)
SUBU	00	100011 (043)
*?	00	101000 (050)
*?	00	101001 (051)
SLT	00	101010 (052)
SLTU	00	101011 (053)



ed bit do? Green? Blue?

MIPS Opcode Map

31...29 \ 28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

Opcode

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

SPECIAL function

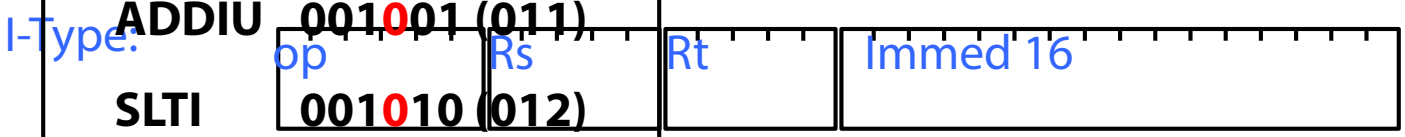
20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

REGIMM rt

[from MIPS R4000 Microprocessor User's Manual / Joe Heinrich]

MIPS arithmetic instruction format

Type	op
ADDI	001000 (010)
ADDIU	001001 (011)
SLTI	001010 (012)
SLTIU	001011 (013)
ANDI	001100 (014)
ORI	001101 (015)
XORI	001110 (016)
LUI	001111 (017)



What does the red bit do?

"Perhaps it is surprising that addiu and sltiu also sign-extend their immediates, but they do. The u stands for unsigned, but in reality addiu is often used simply as an add instruction that cannot overflow, and hence we often want to add negative numbers. *It's much harder to come up with an example for why sltiu sign extends its immediate field.*" COD2E p. 230

MIPS Opcode Map

31...29 \ 28...26 (Opcode)	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

5...3 \ 2...0 (SPECIAL function)	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

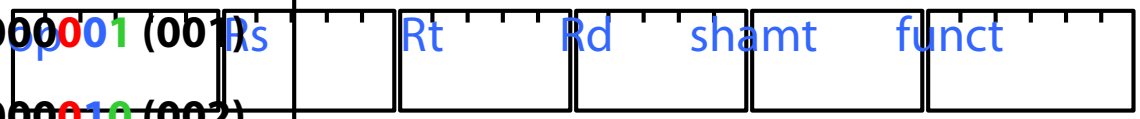
20...19 \ 18...16 (REGIMM rt)	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

[from MIPS R4000 Microprocessor User's Manual / Joe Heinrich]

MIPS arithmetic instruction format

Type	op	funct
SLL	00	000000 (000)
*	00	000001 (001)
SRL	00	000010 (002)
SRA	00	000011 (003)
SLLV	00	000100 (004)
*	00	000101 (005)
SRLV	00	000110 (006)
SRAV	00	000111 (007)

R-type:



What does the red bit do?

Green?

Blue?

* instructions?

MIPS Opcode Map

Opcode

31…29 \ 28…26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

SPECIAL function

5…3 \ 2…0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

REGIMM rt

20…19 \ 18…16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

[from MIPS R4000 Microprocessor User's Manual / Joe Heinrich]

MIPS arithmetic instruction format