Lecture 5:

Arithmetic 1/3

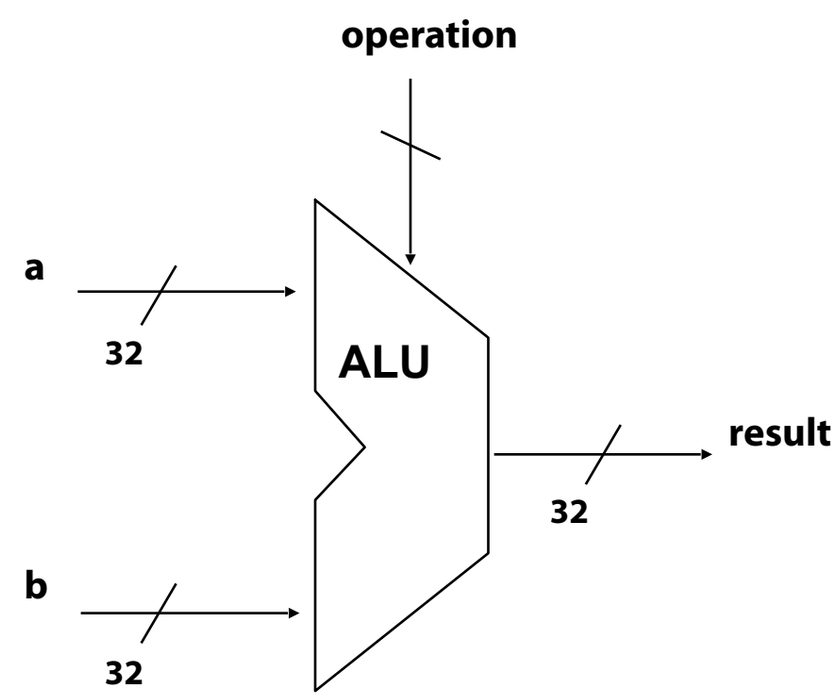
Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

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***

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

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
```

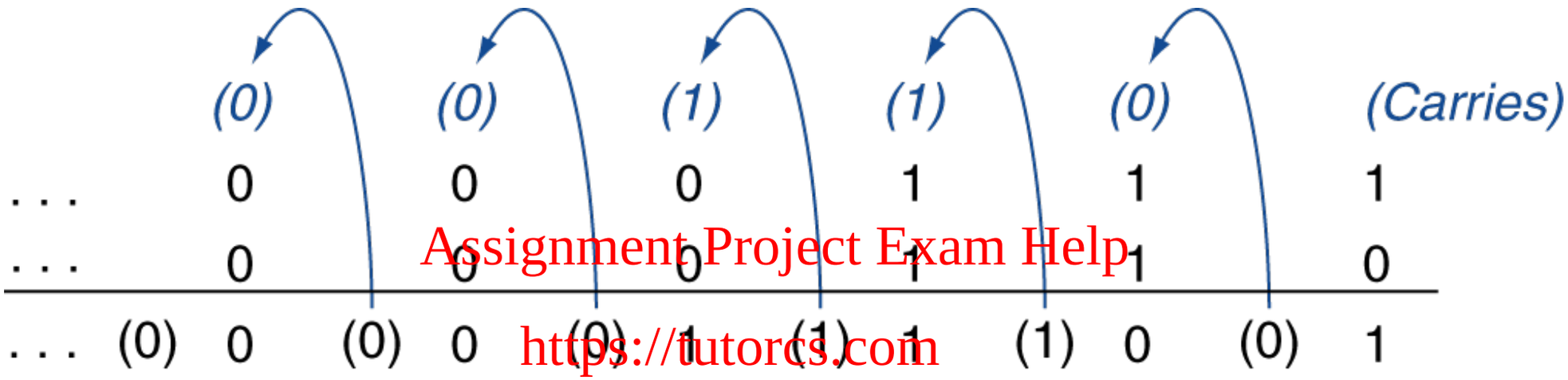

  - **How about -1 + -1?**

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 truth table for these three functions.**

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

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"

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)

Truth Table for Adder Bit Slice

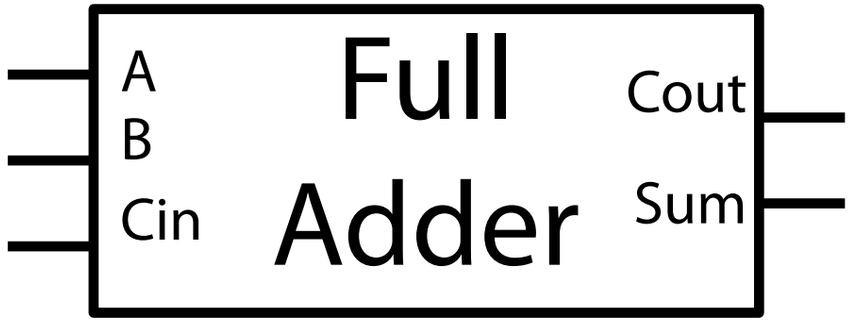
- 3 inputs (A, B, Cin); 2 outputs (Sum, Cout)

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

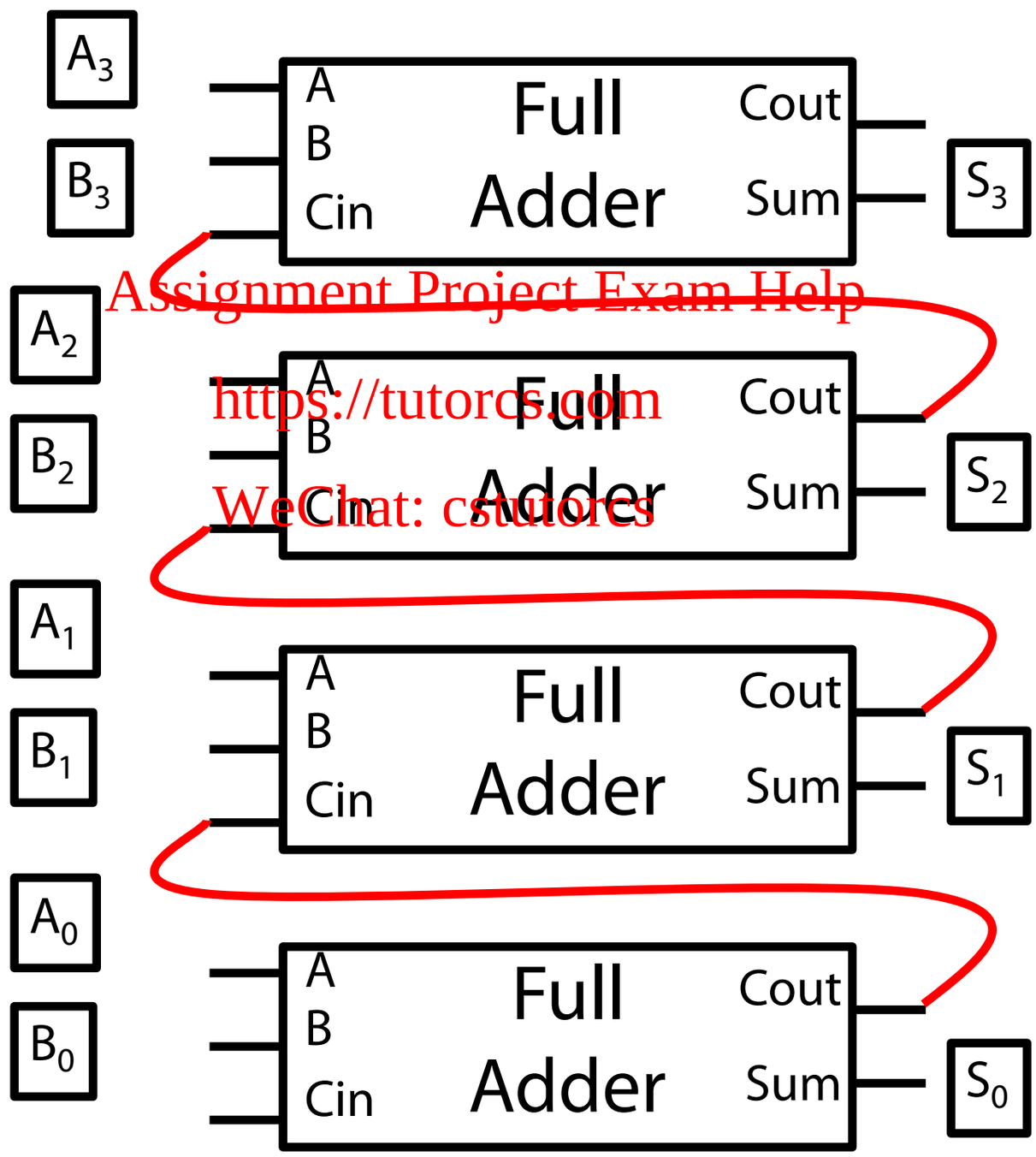
- $\text{Sum} = (A \oplus B) \oplus \text{Cin}$
- $\text{Carry} = AB + A\text{Cin} + B\text{Cin}$

- Abstract as "Full Adder":

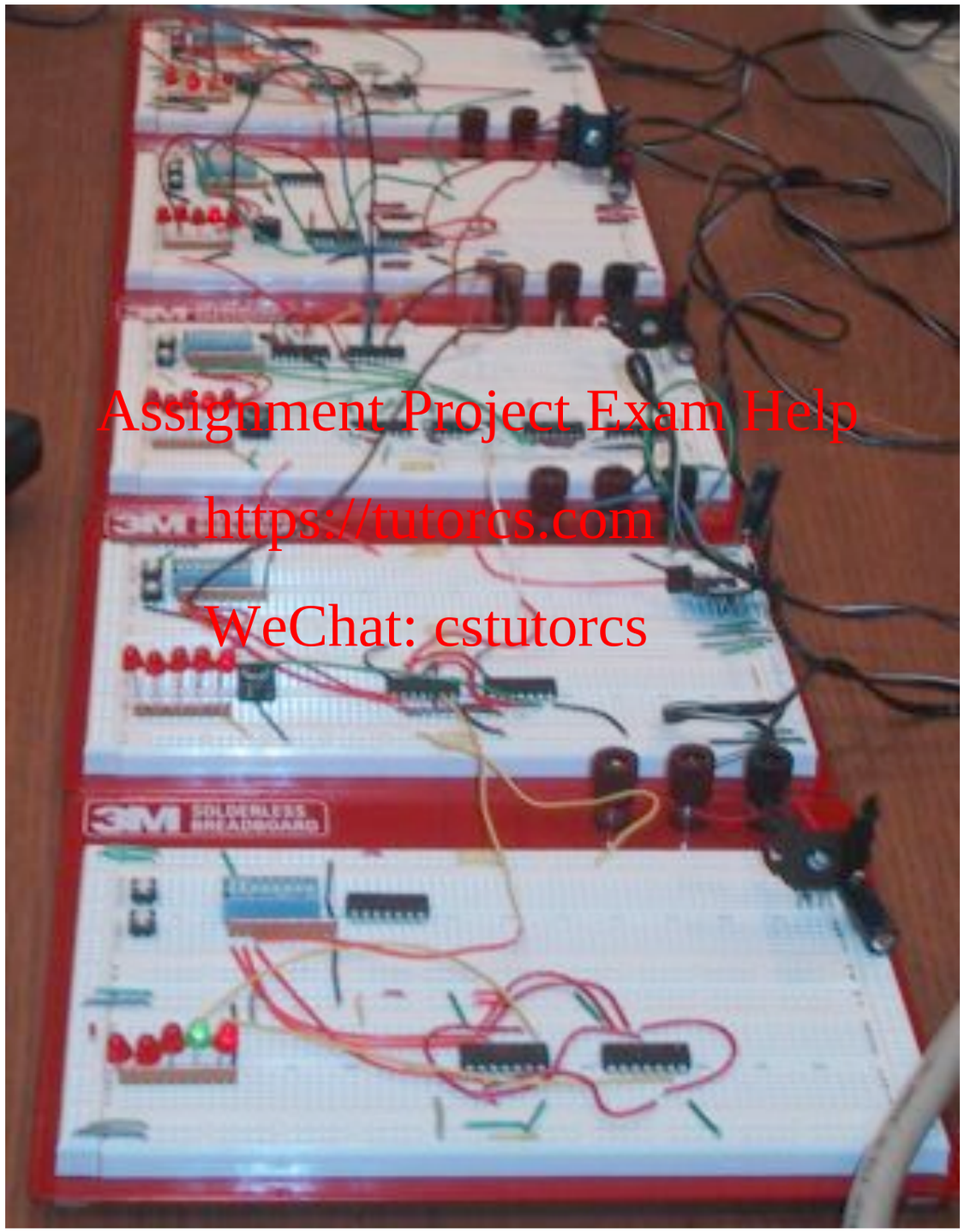


Cascading Adders

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



Lest you think this is only theoretical …



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

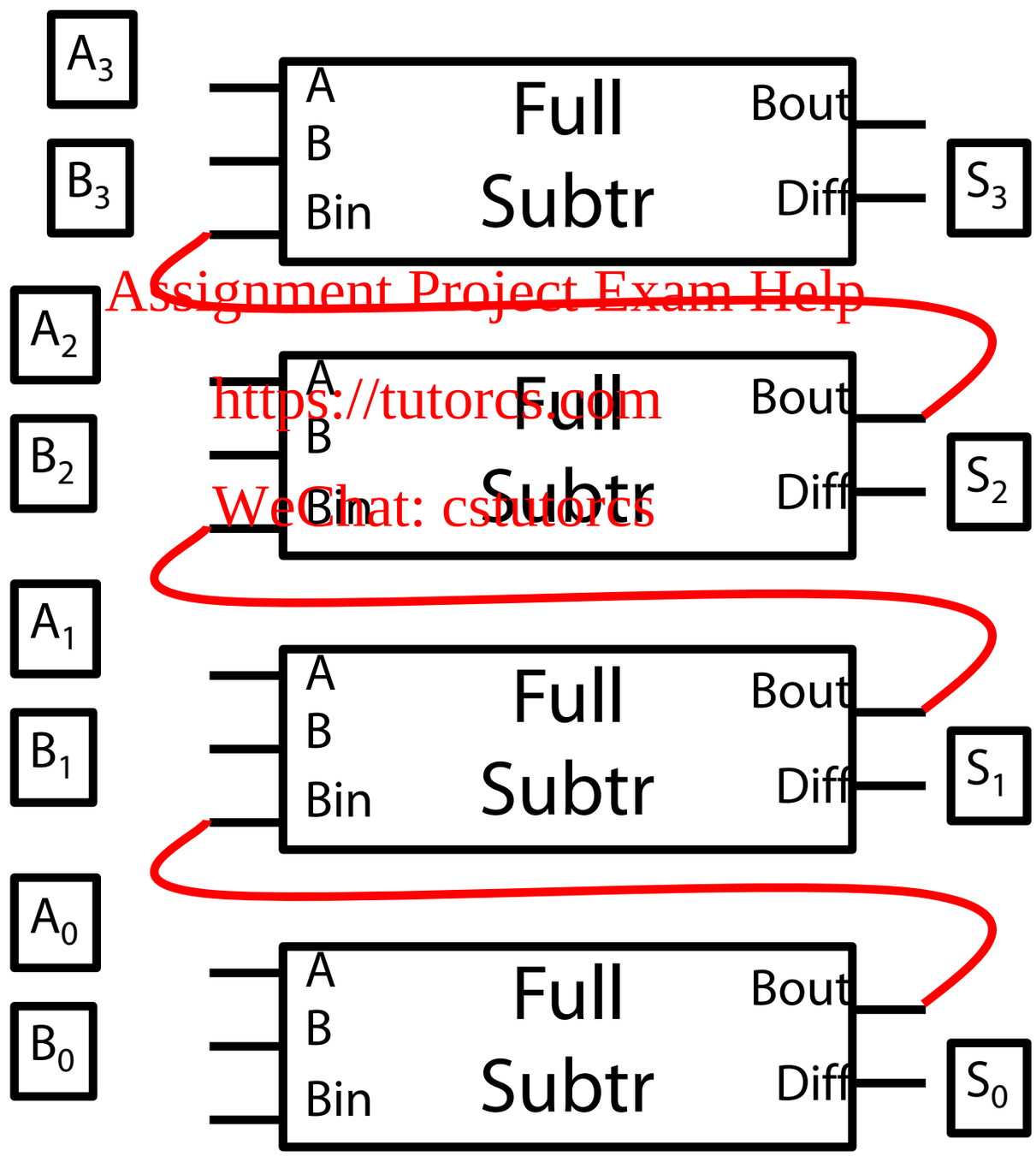
Truth Table for Subtractor Bit Slice

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

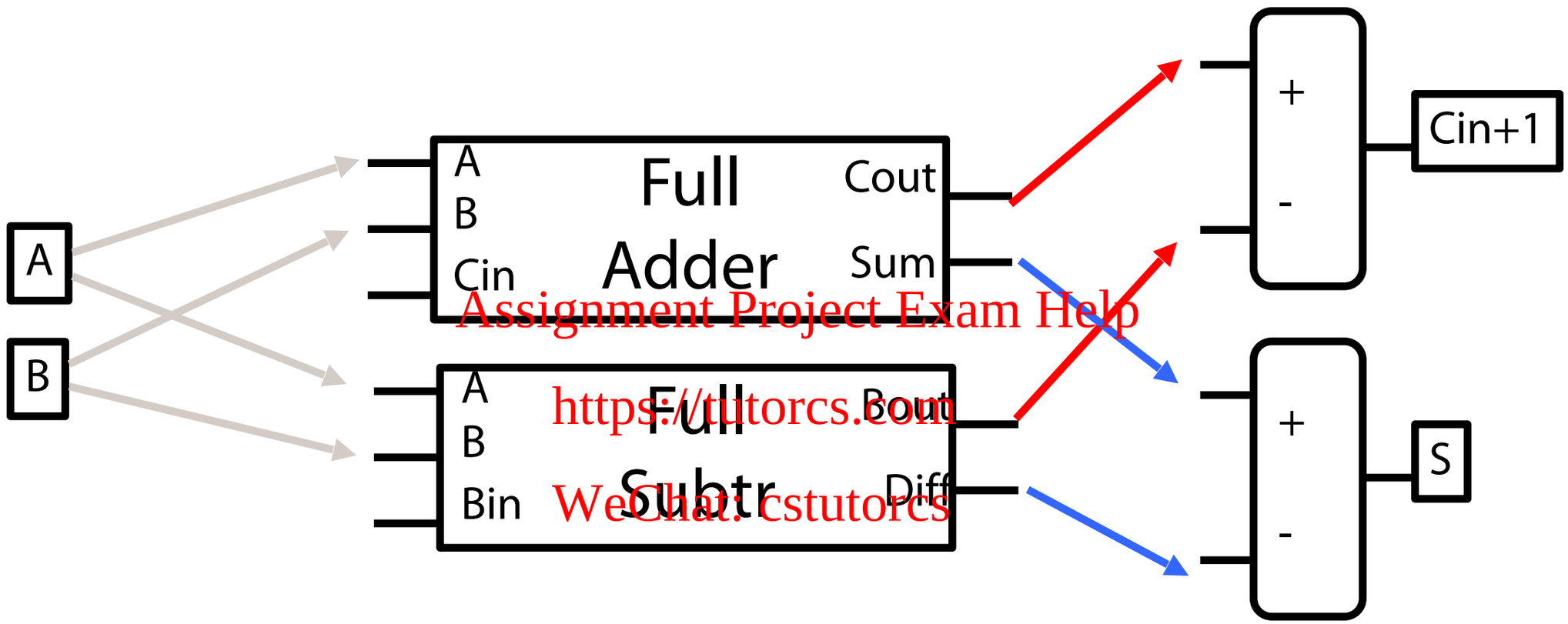
A	B	Bin	Diff	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Cascading Subtractors

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



How can we combine + and -?

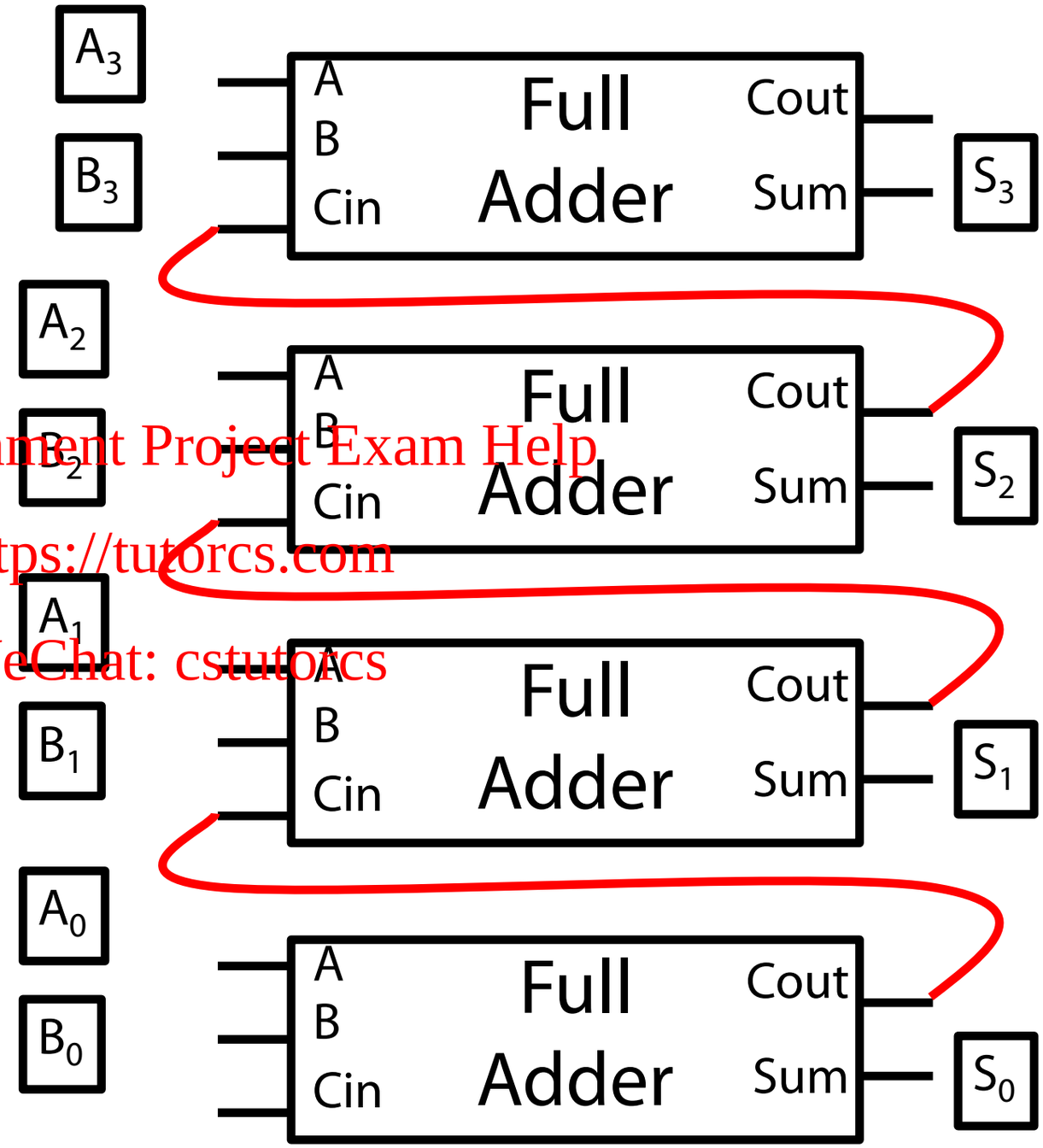


- **This is common—it's what we'll do (for example) for logic functions (and, or, etc.)**

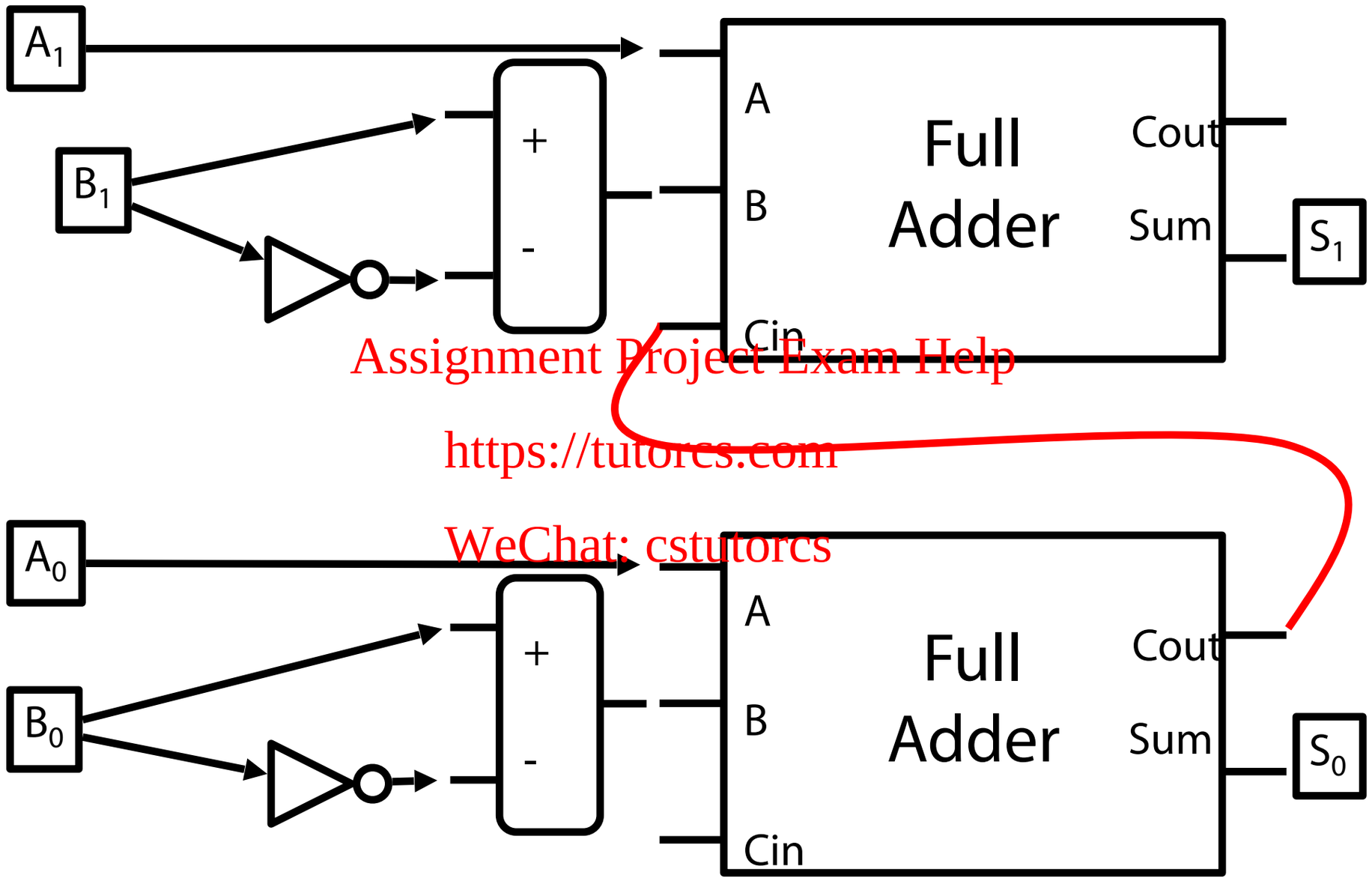
How can we combine + and -?

$S = A - B$

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



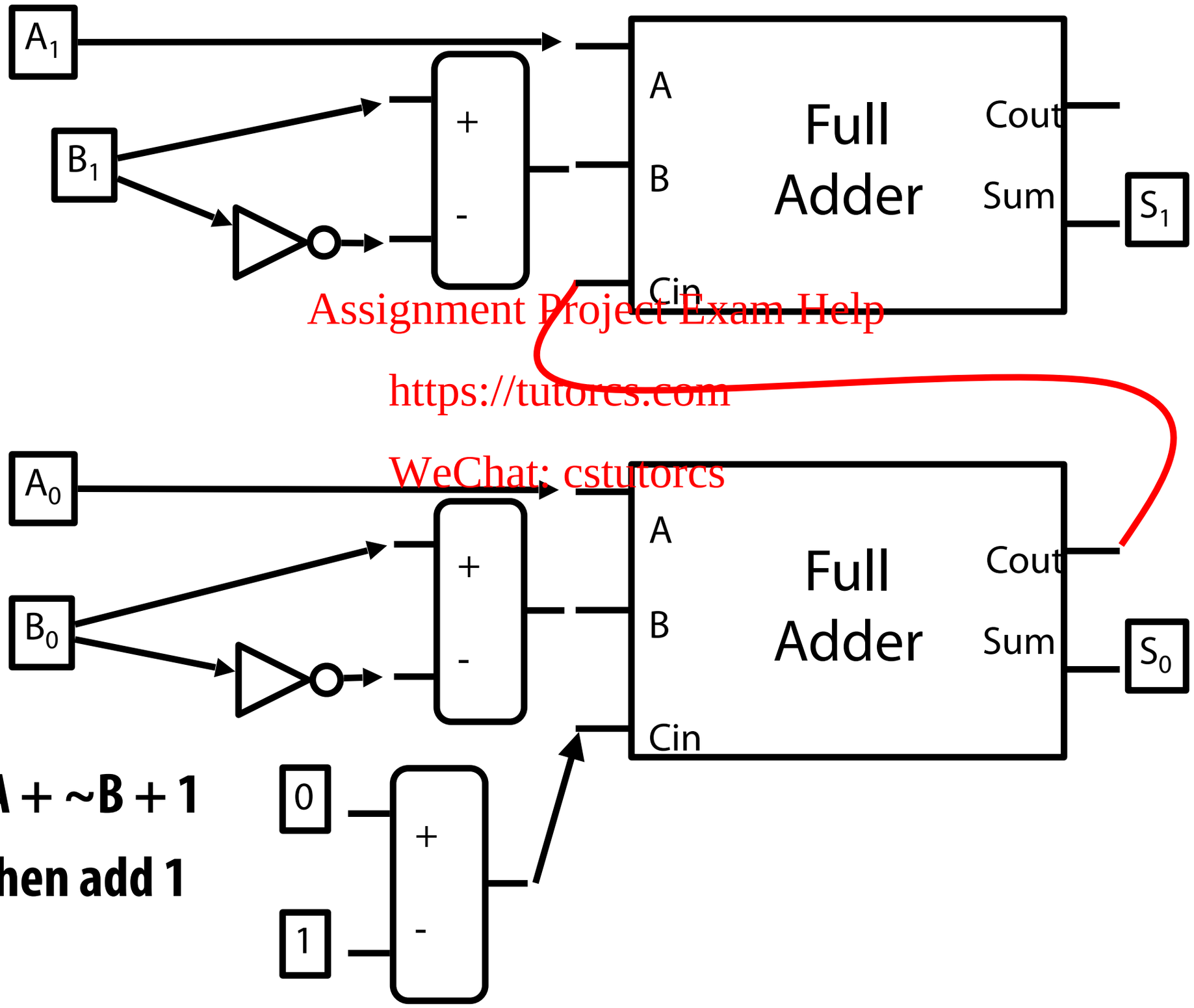
How can we combine + and -?



S = A + ~B + 1

First, negate B …

How can we combine + and -?



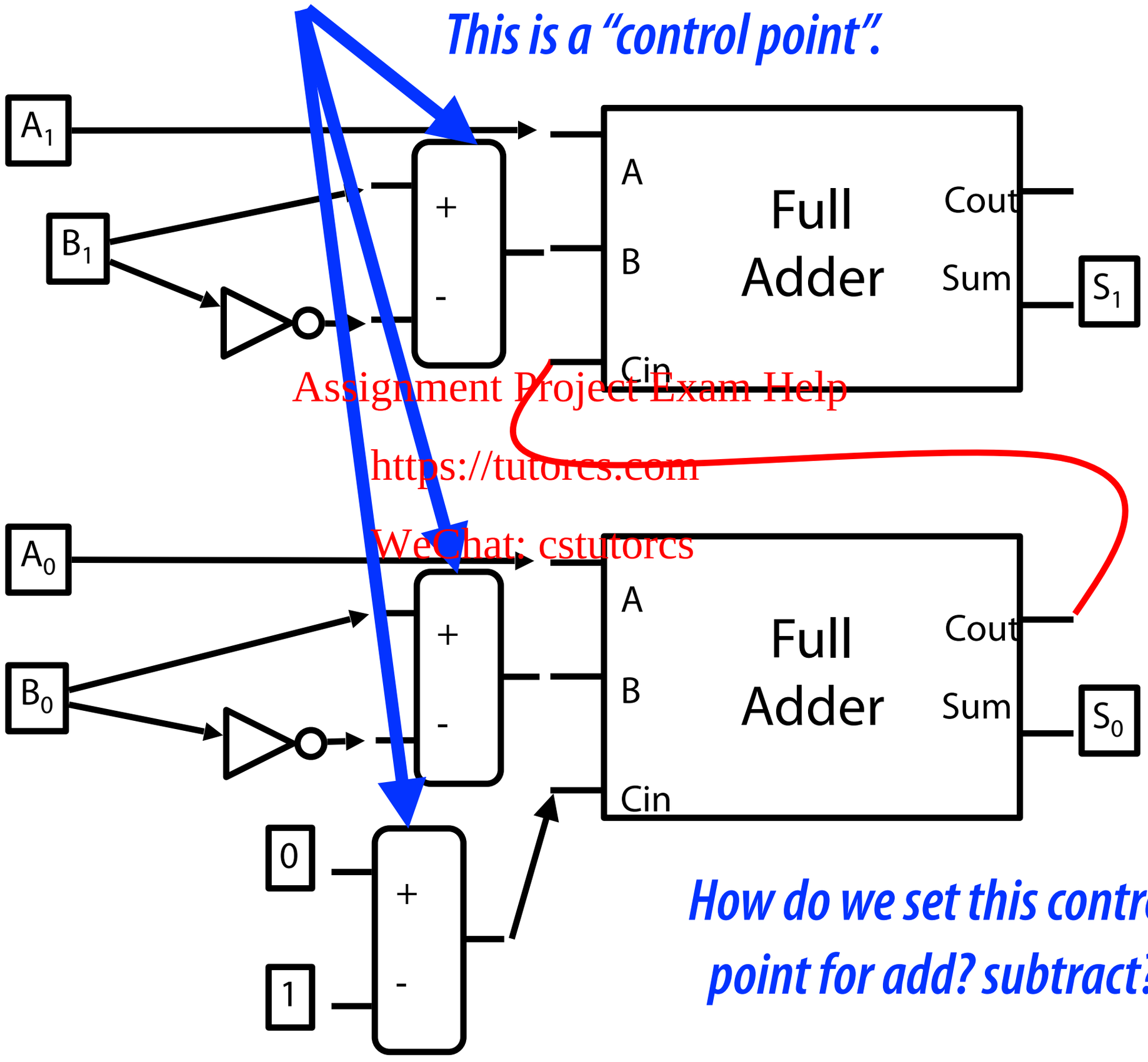
S = A + ~B + 1

… 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
0000000	rs2	rs1	100	rd	0110011	XOR

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

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

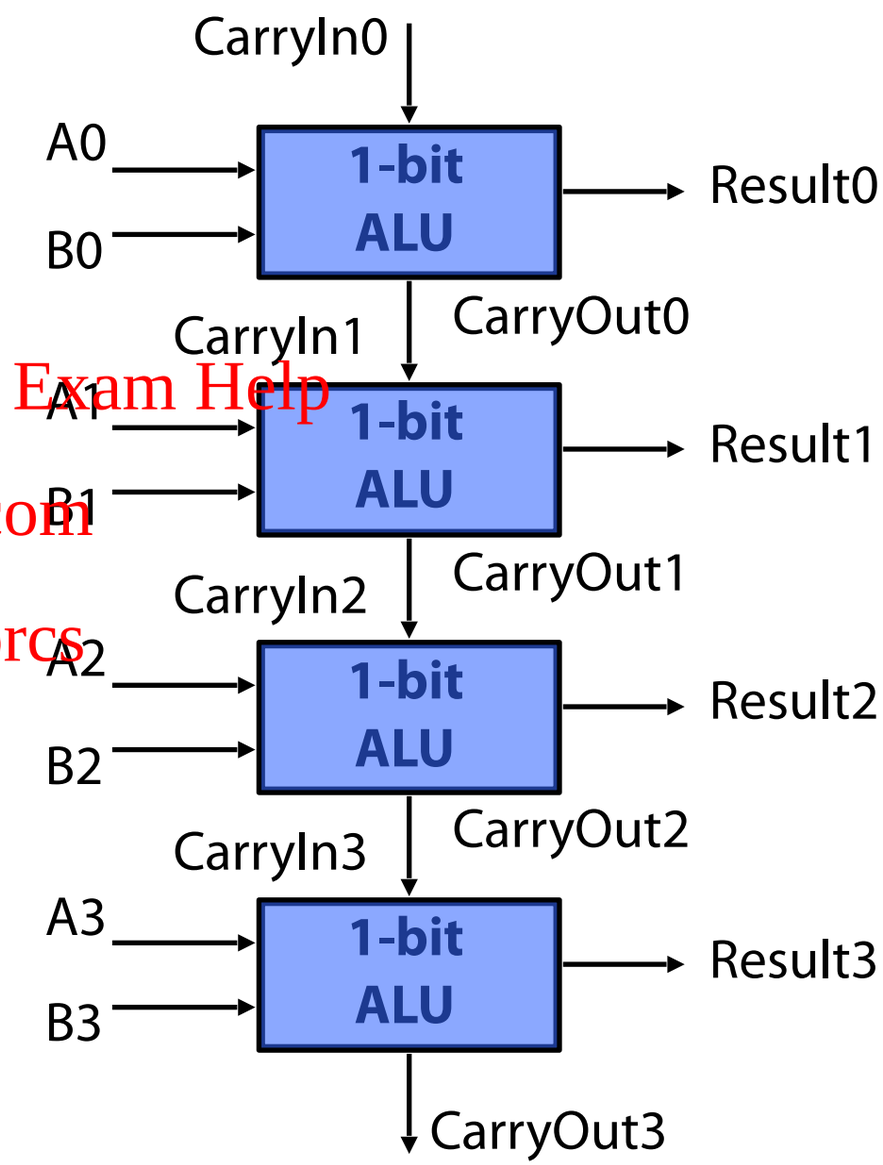
REGIMM rt

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]

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

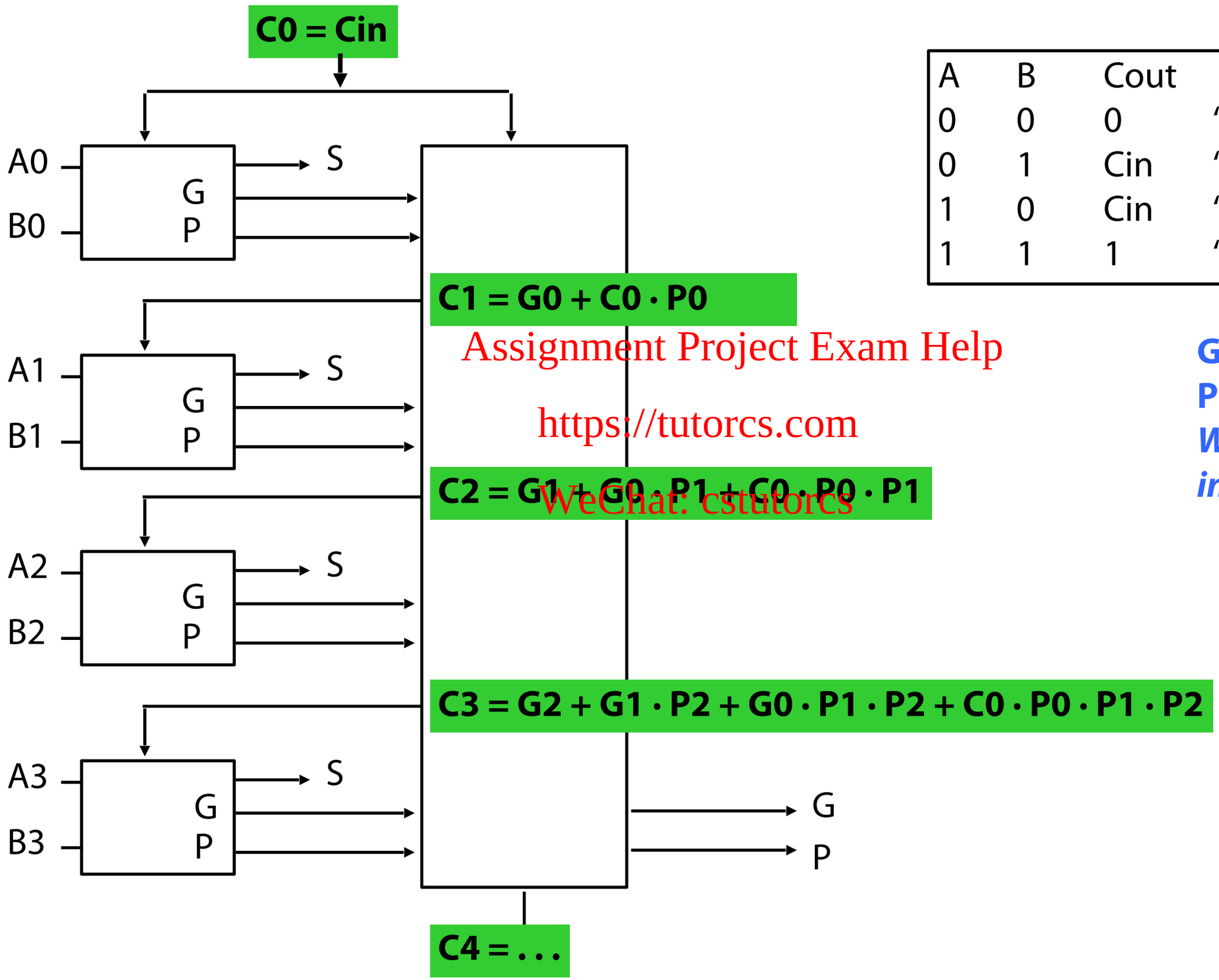


Truth Table for Adder Bit Slice

- 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

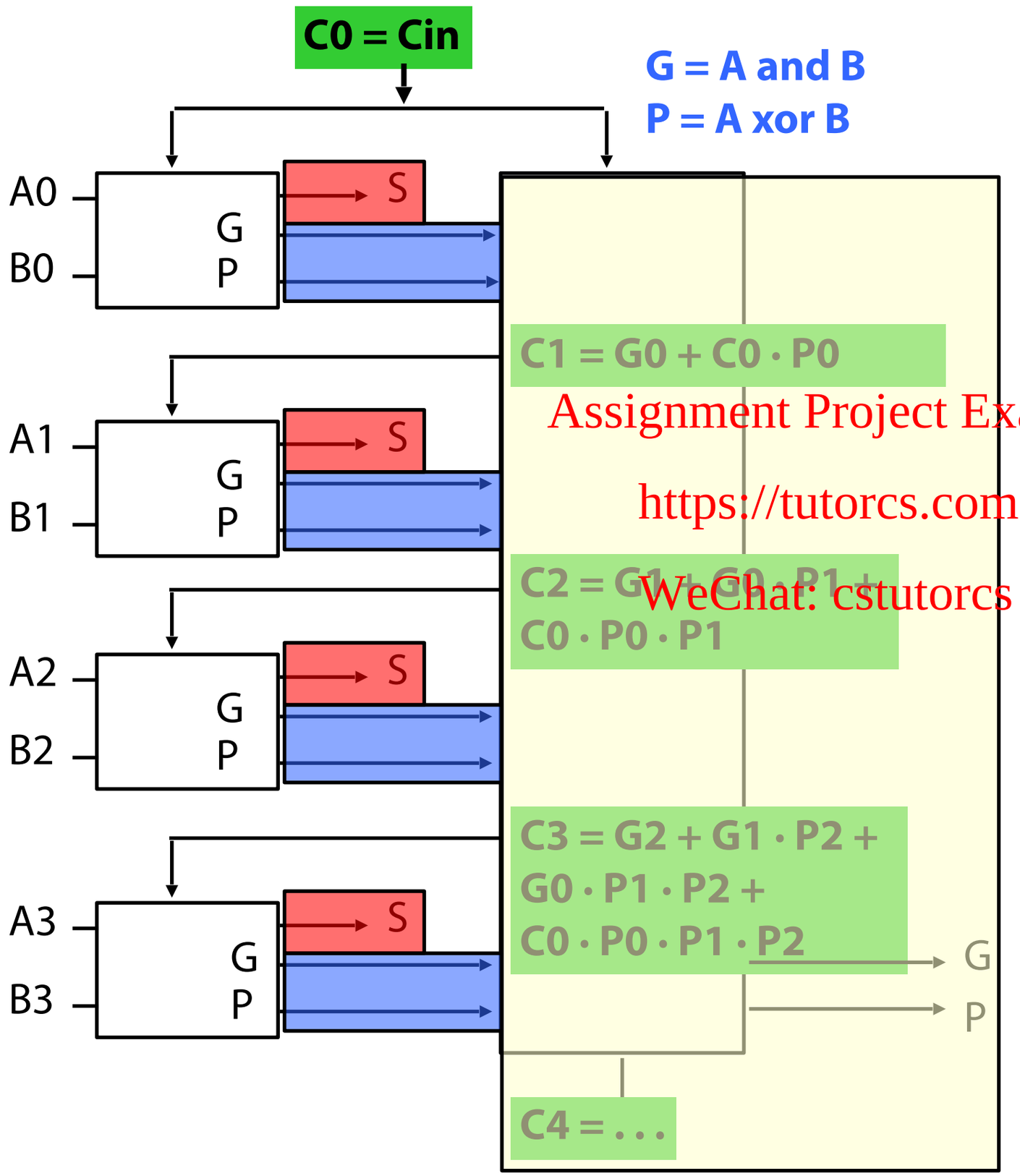
Carry Look Ahead (Design trick: peek)

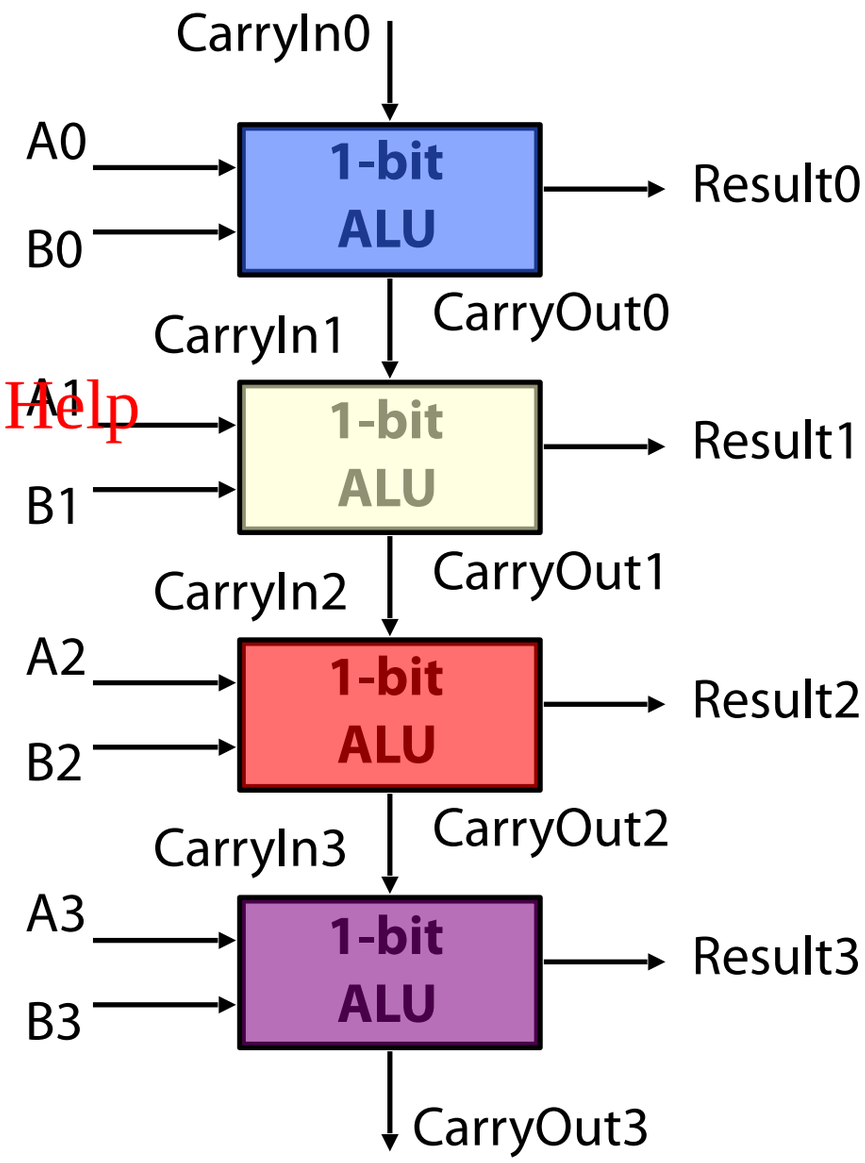


A	B	Cout	
0	0	0	"kill"
0	1	Cin	"propagate"
1	0	Cin	"propagate"
1	1	1	"generate"

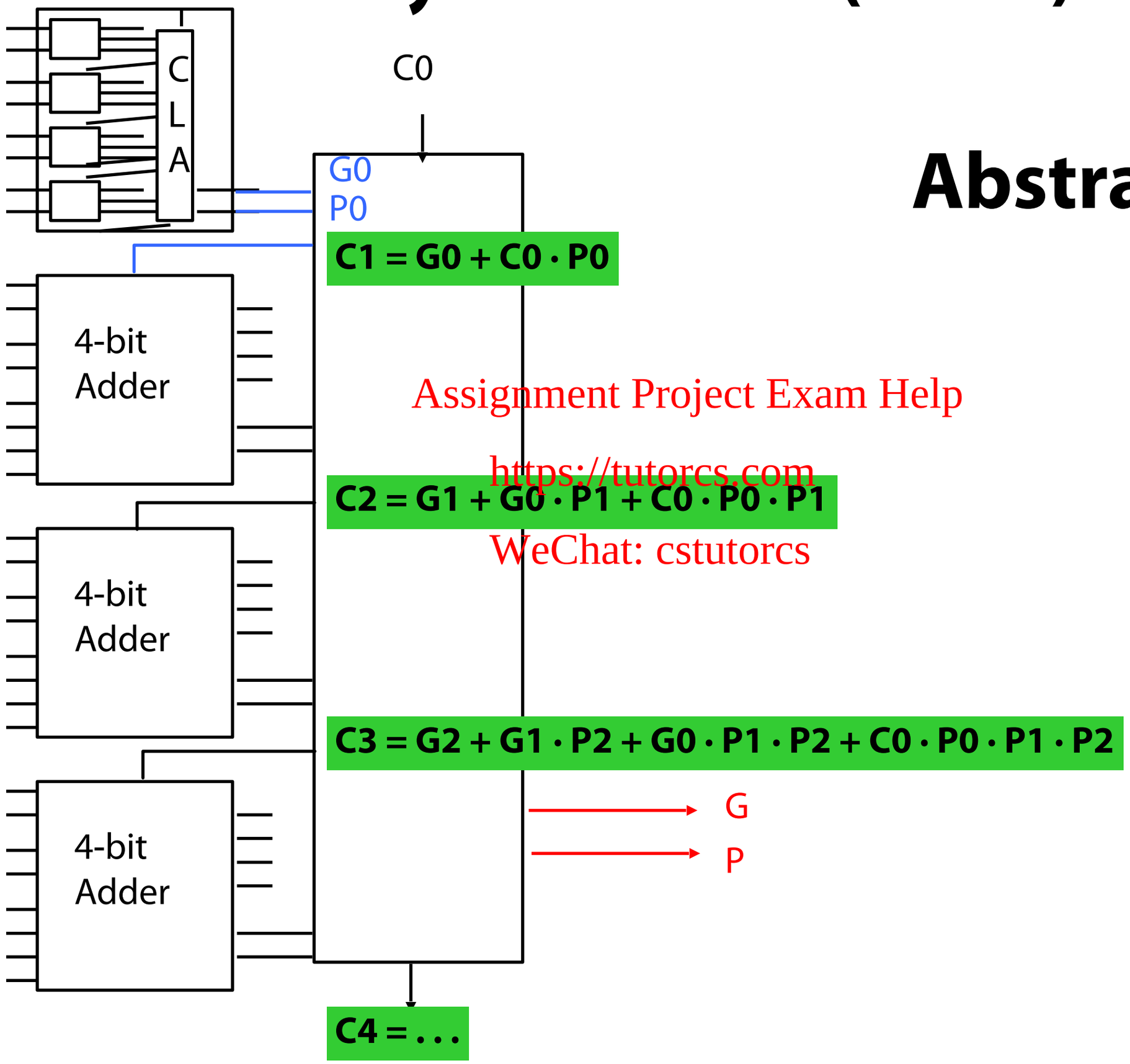
G = A and B
P = A xor B
WHY are these interesting?

CLA vs. Ripple





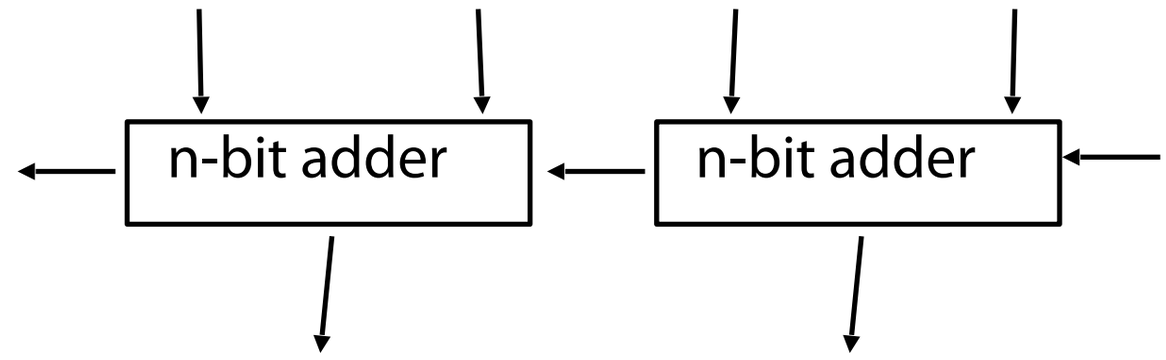
Cascaded Carry Look-ahead (16-bit)



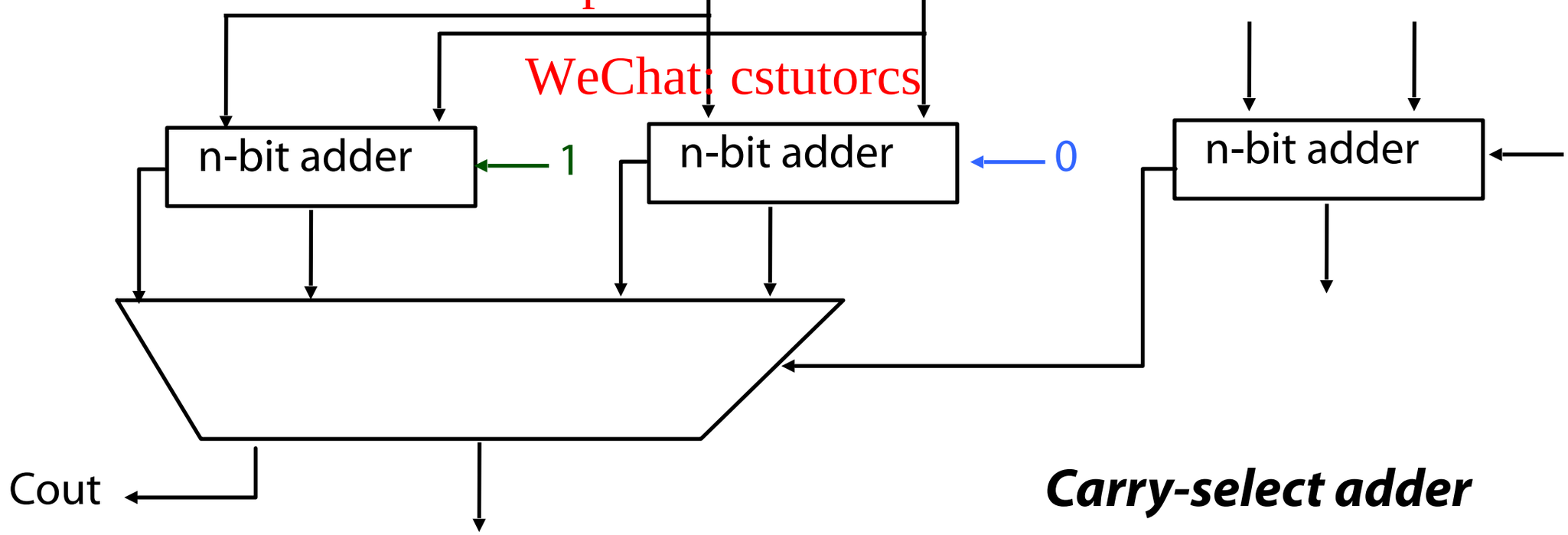
Abstraction!

Design Trick: Guess (or "Precompute")

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

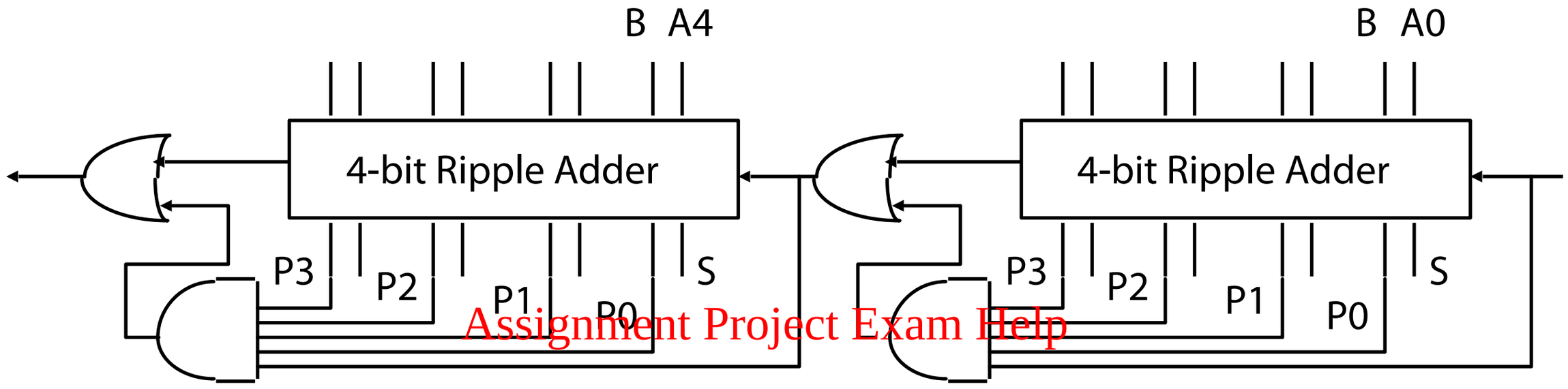


CP(2n) = CP(n) + CP(mux)



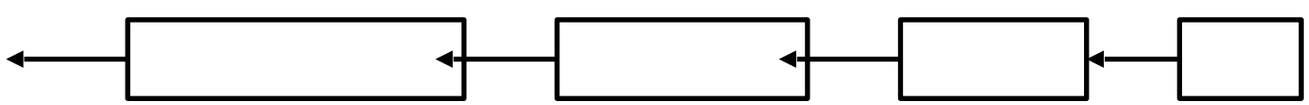
Carry-select adder

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]