Lecture 7:

Arithmetic 3/3

Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

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

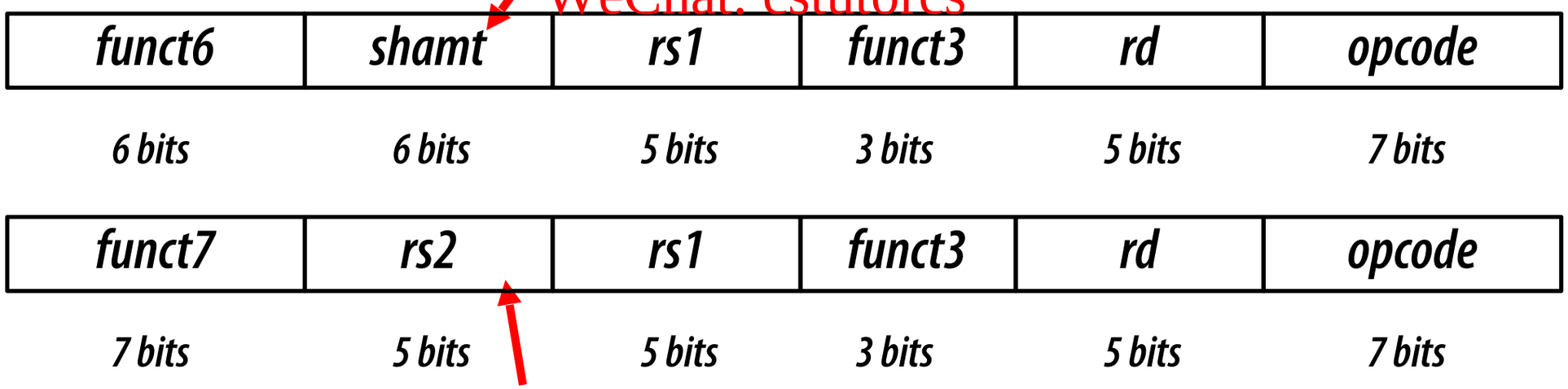
- Arithmetic operation:
  - Example: 00011 << 2 [3 left shift 2]
    - 00011 << 2 = 01100 = 12 = 2 * 4
  - Each bit shifted left == multiply by two
  - Example: 01010 >> 1 [10 right shift 1]
    - 01010 >> 1 = 00101 = 5 = 10/2
- Each bit shifted right == divide by two
- Why?
- Compilers do this—"strength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

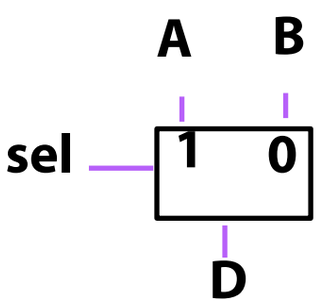
- How about right shift?
  - XXXX0000 >> 4 = 0000XXXX: Logical shift
    - Shifted in zero
  - 00110 (= 6) >> 1 = 00011 (3)
    11110 (= -2) >> 1 = 11111 (-1): Arithmetic shift
    - Shifted in sign bit
- RISC-V supports both logical and arithmetic:
  - slli, srai, srli: Shift amount taken from within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



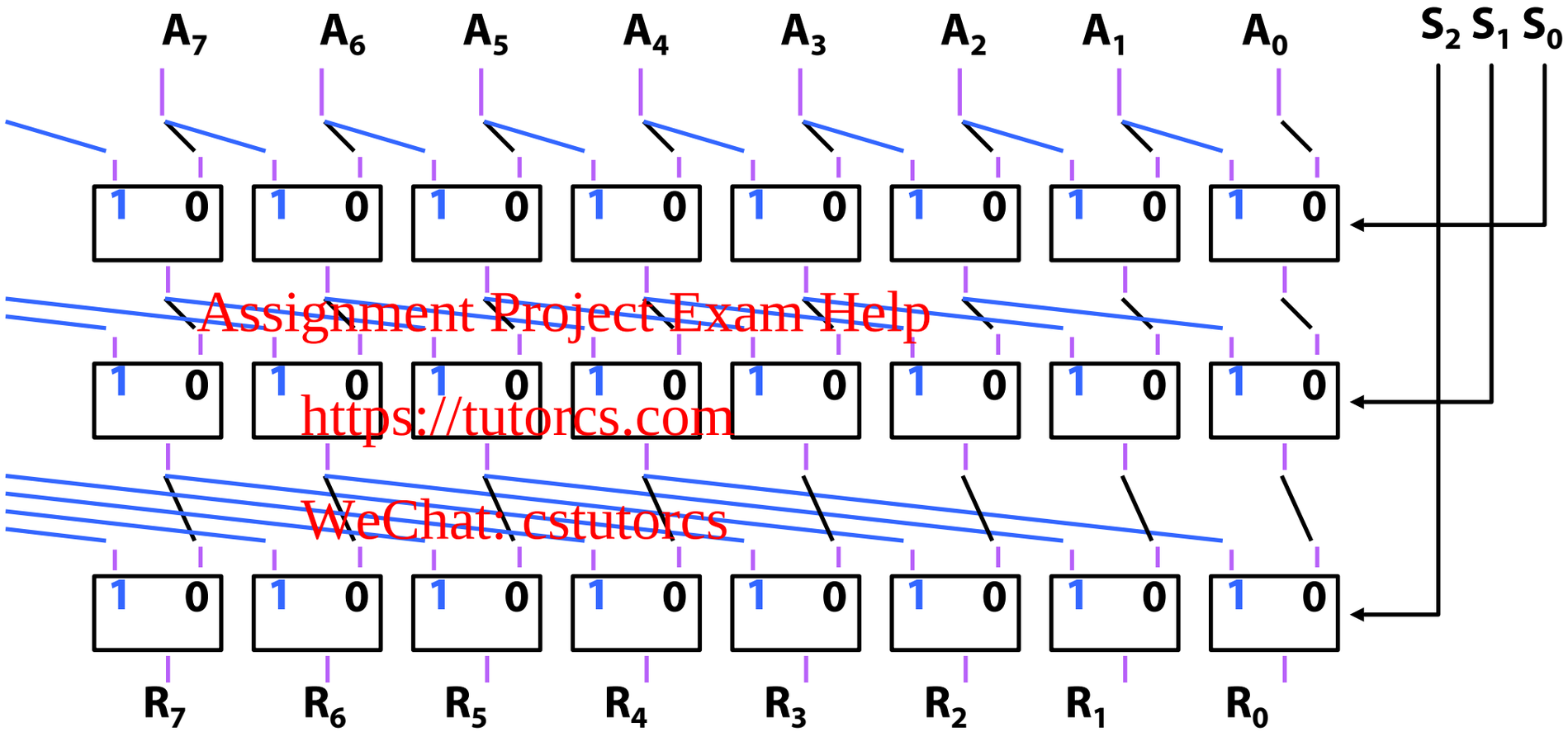
  - sll, sra, srl: shift amount taken from register ("variable")
  - How far can we shift with slli/srai/slli? With sll/sra/srl?

Combinational Shifter from MUXes

Basic Building Block

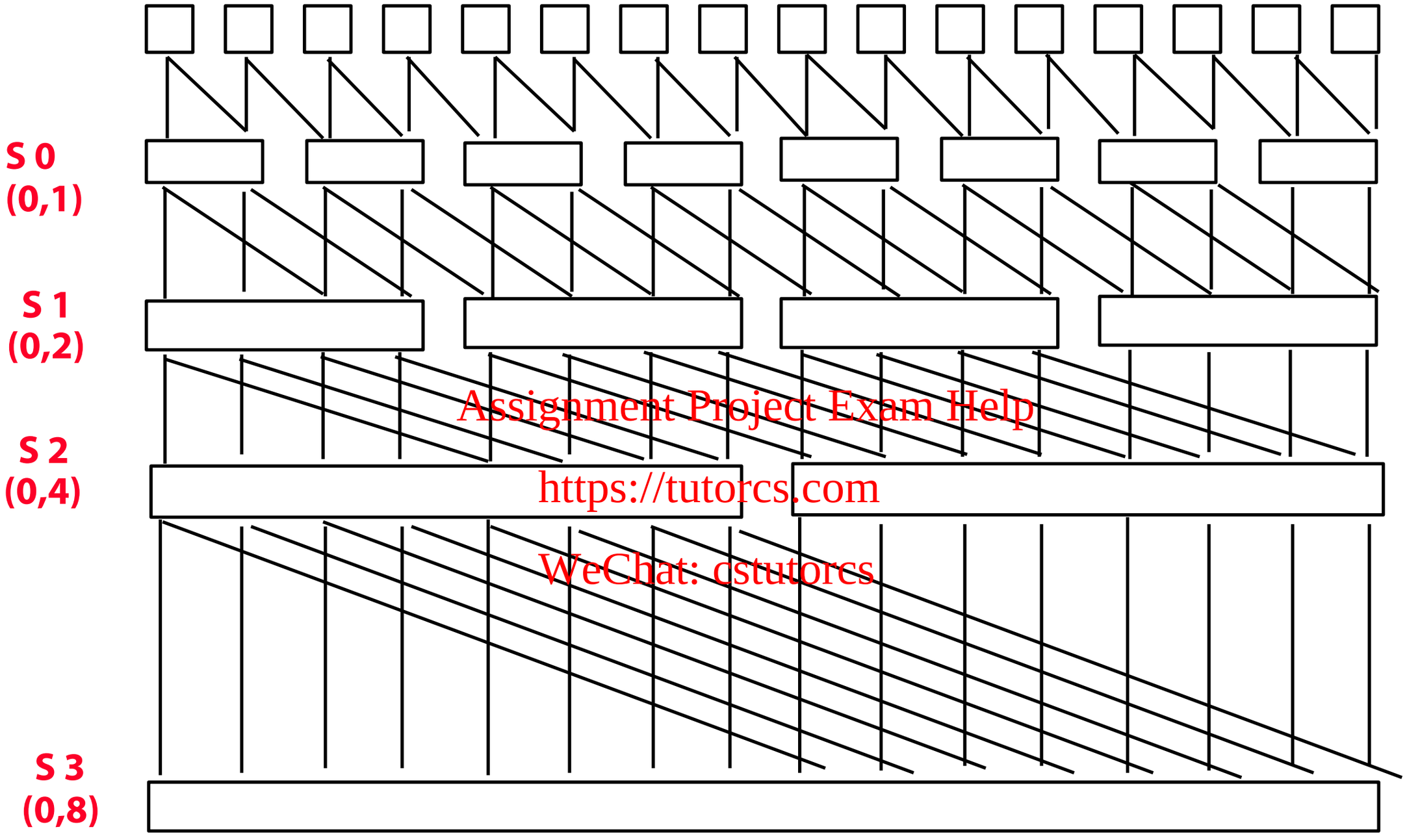


8-bit right shifter



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

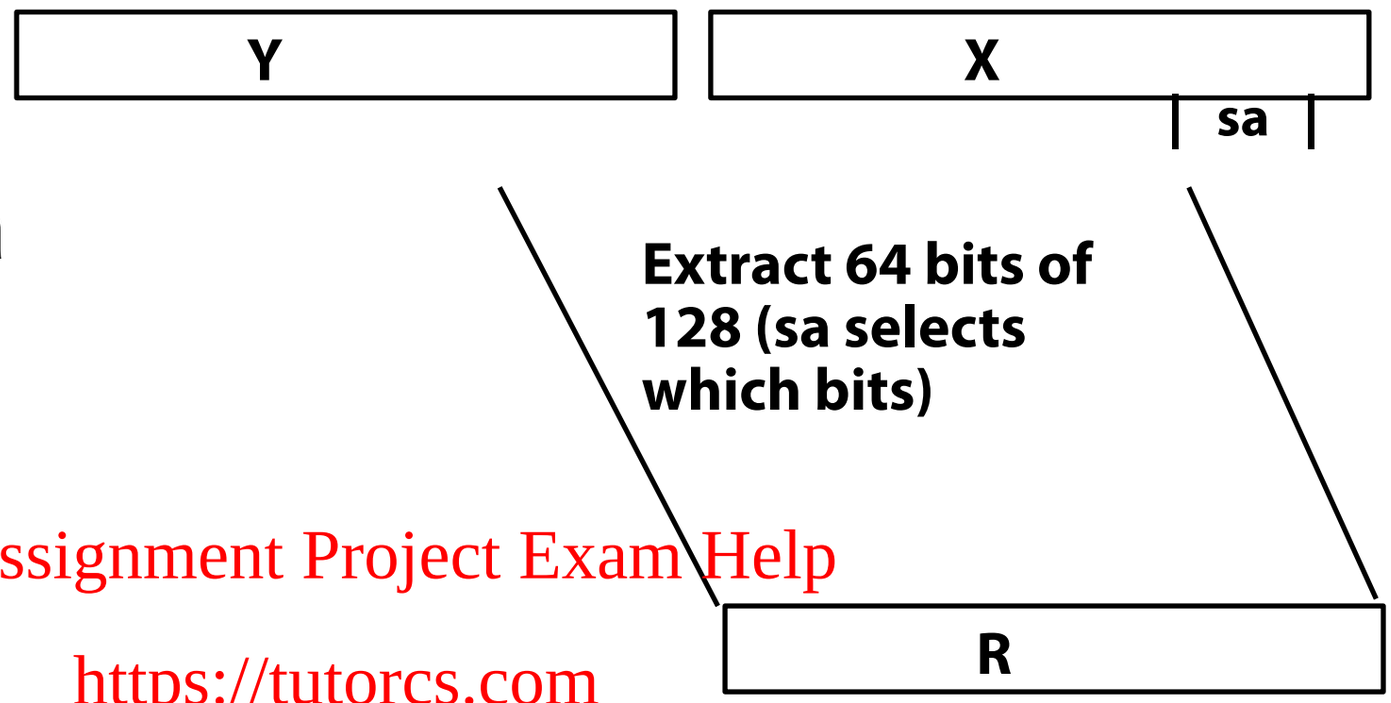
General Shift Right Scheme using 16 bit example



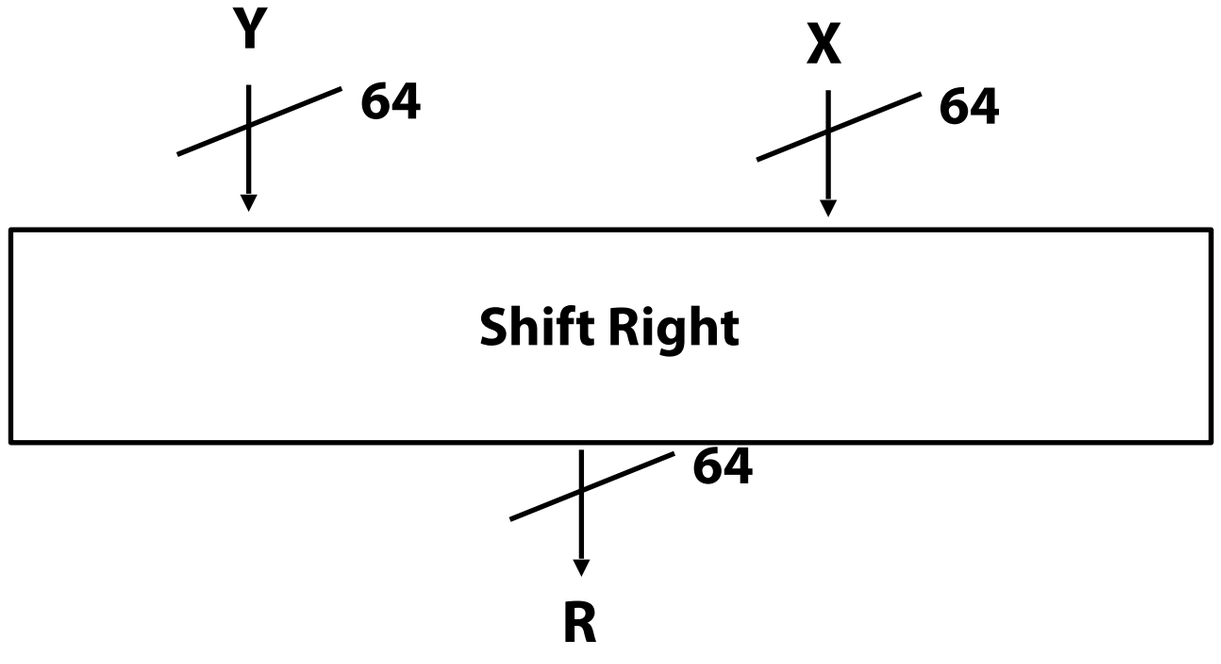
- **If we added right-to-left connections, we could support** ROTATE **(not in RISC-V but found in other ISAs)**

Funnel Shifter

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

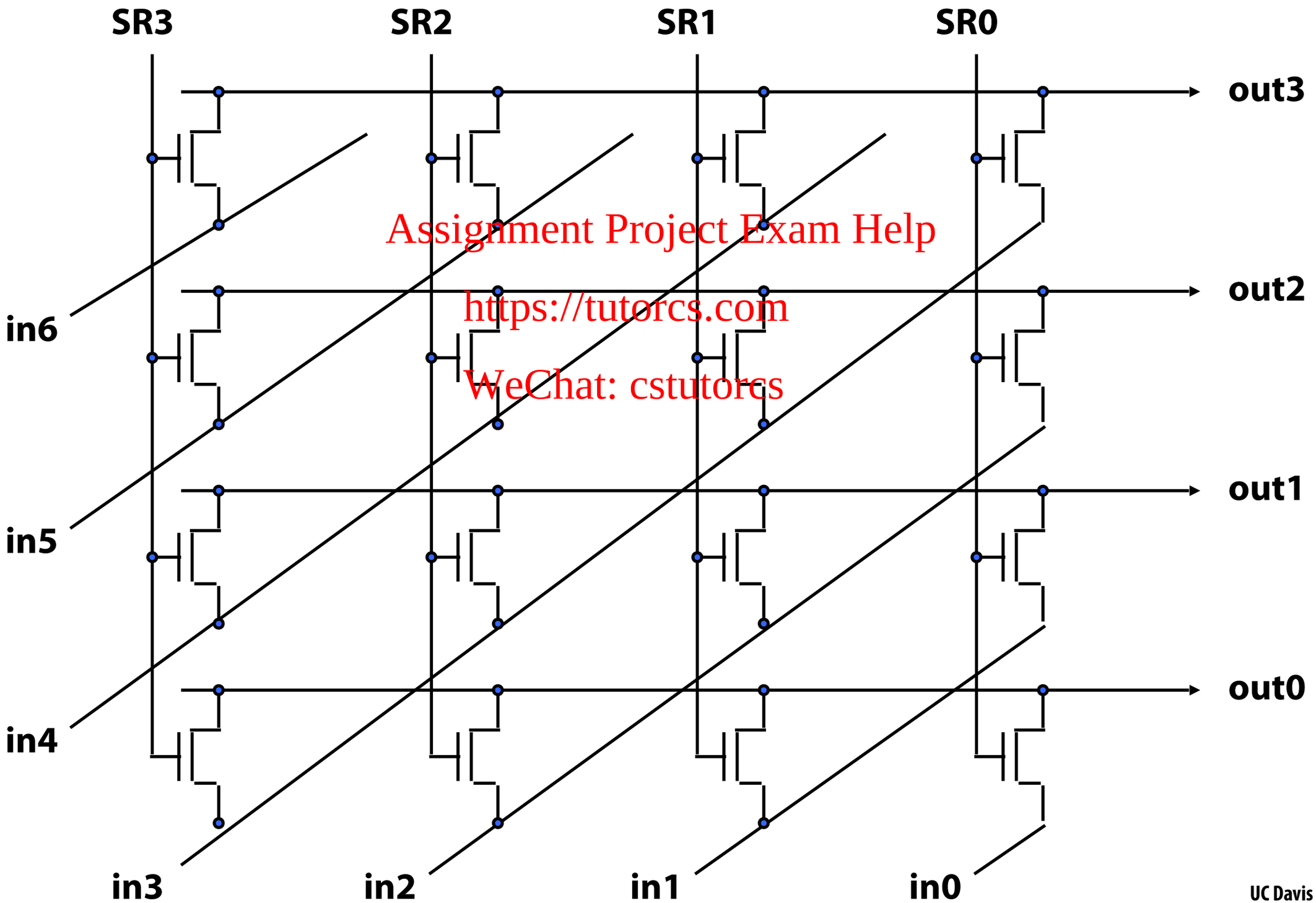


- Arithmetic:
- Rotate:
- Left shifts:



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

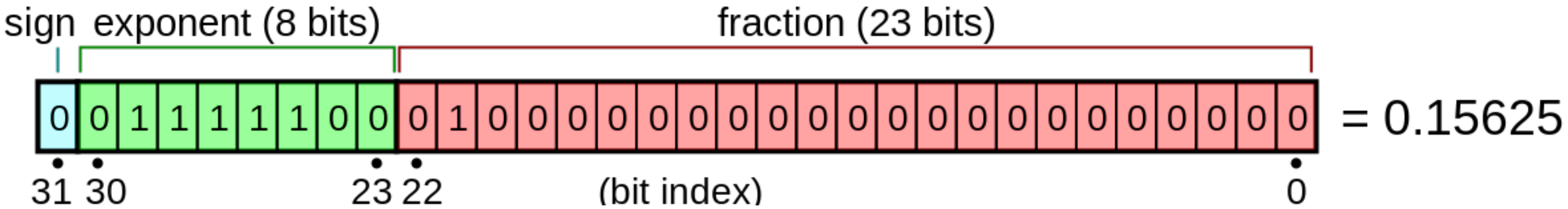
Floating Point

- 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*
  - $+987.02 \times 10^{9}$ ← *not normalized*
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

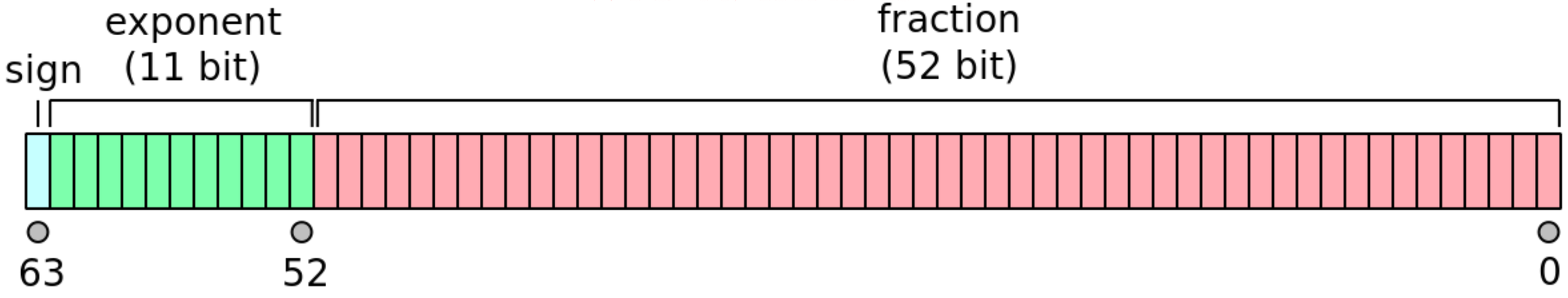
Floating Point Standard

- 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 ≤ |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

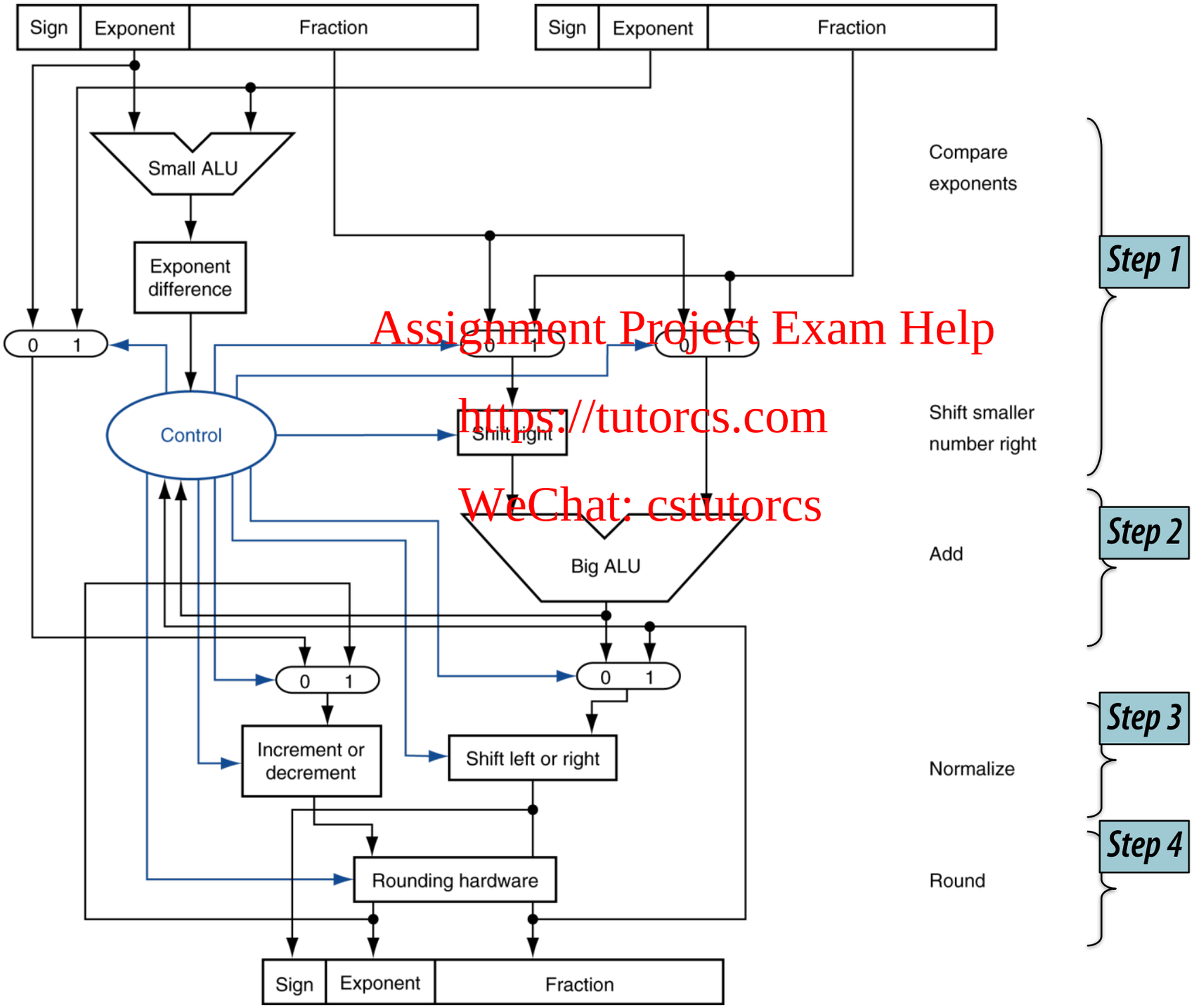
Floating-Point Addition

- 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

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	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

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

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 –∞
    - 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

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

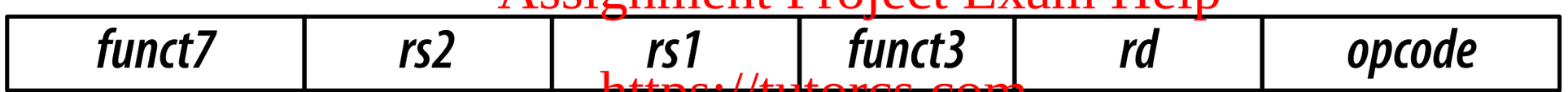
RISC-V ALU requirements

- 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

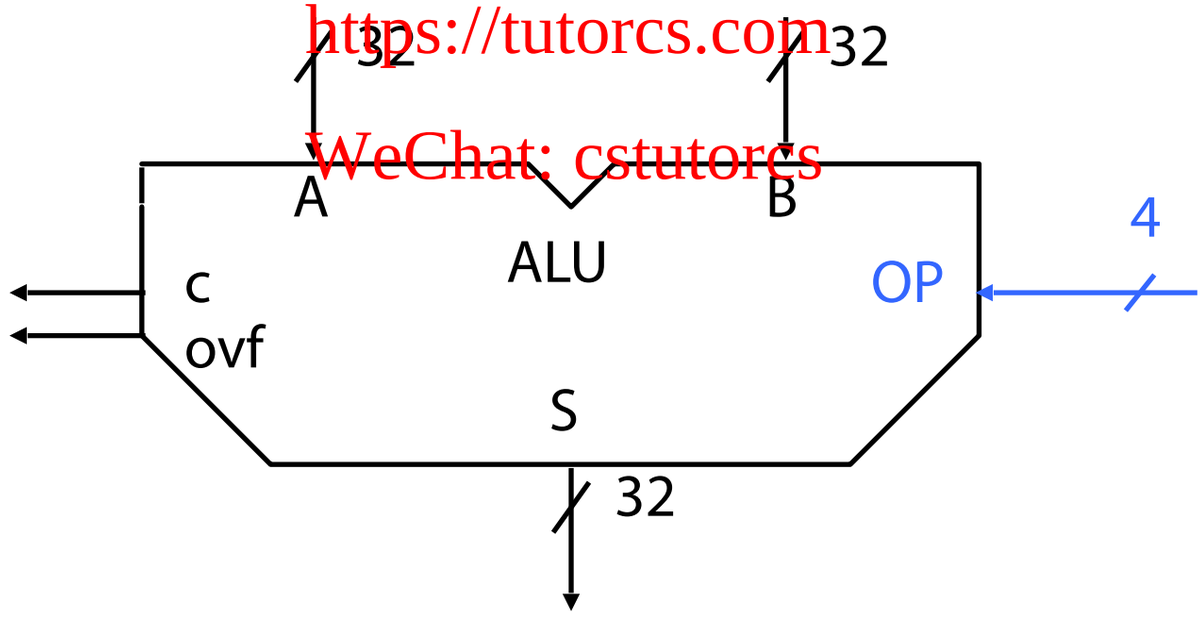
- Trick: Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - 7 operations (could be 3 bits, but really 4)

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

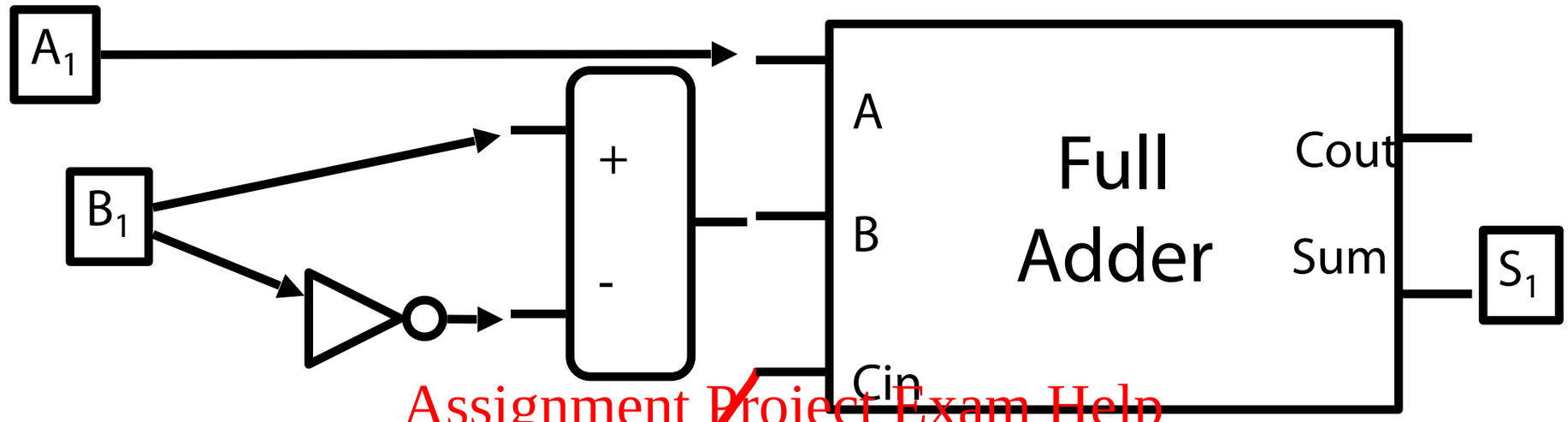
Type	op	funct
ADD	00	0110011 | 000 | 0000000
SUB	00	0110011 | 000 | 0100000
AND	00	0110011 | 111 | 0000000
OR	00	0110011 | 110 | 0000000
XOR	00	0110011 | 100 | 0000000
SLT	00	0110011 | 010 | 0000000
SLTU	00	0110011 | 011 | 0000000

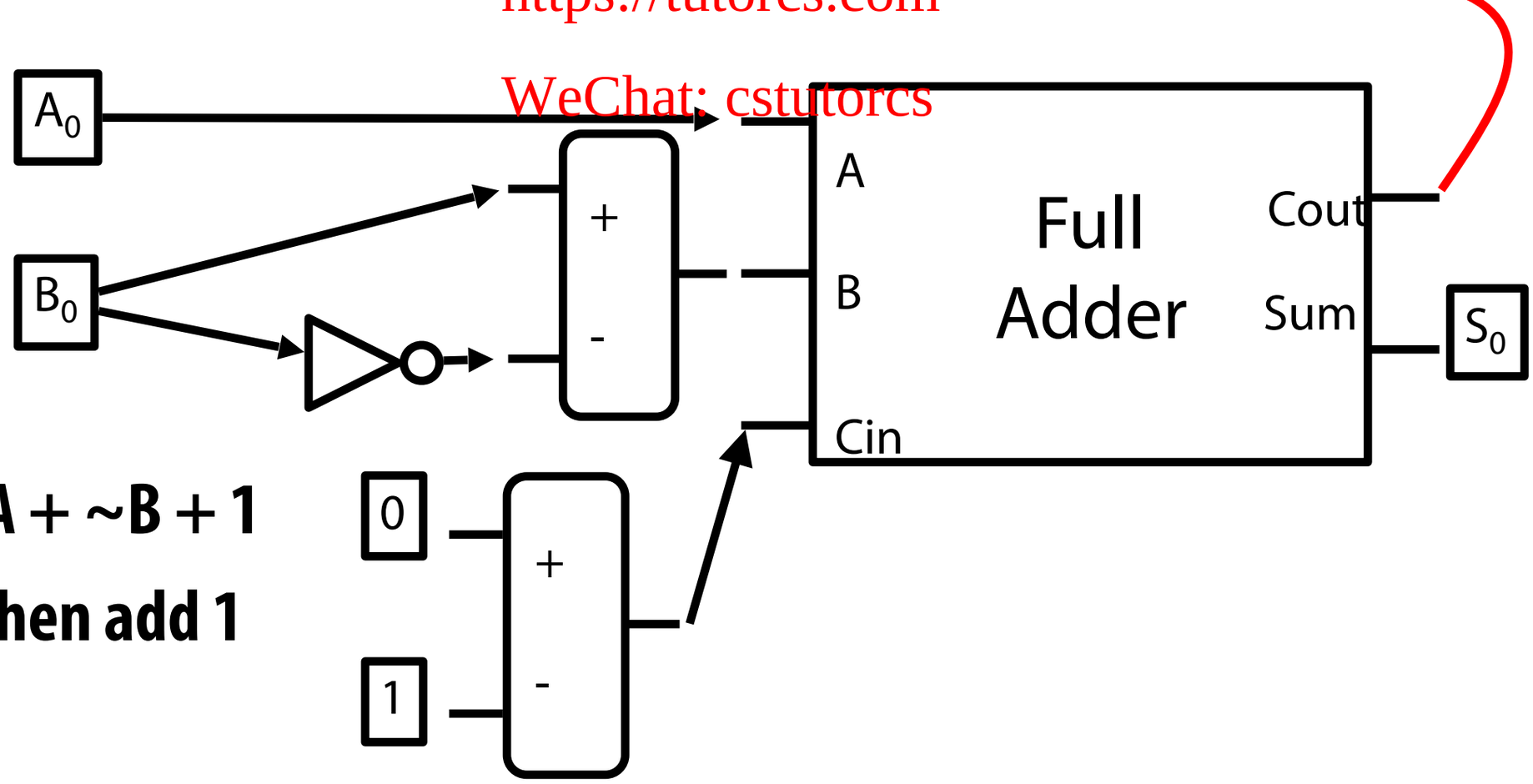
Let's Build a ALU

- **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





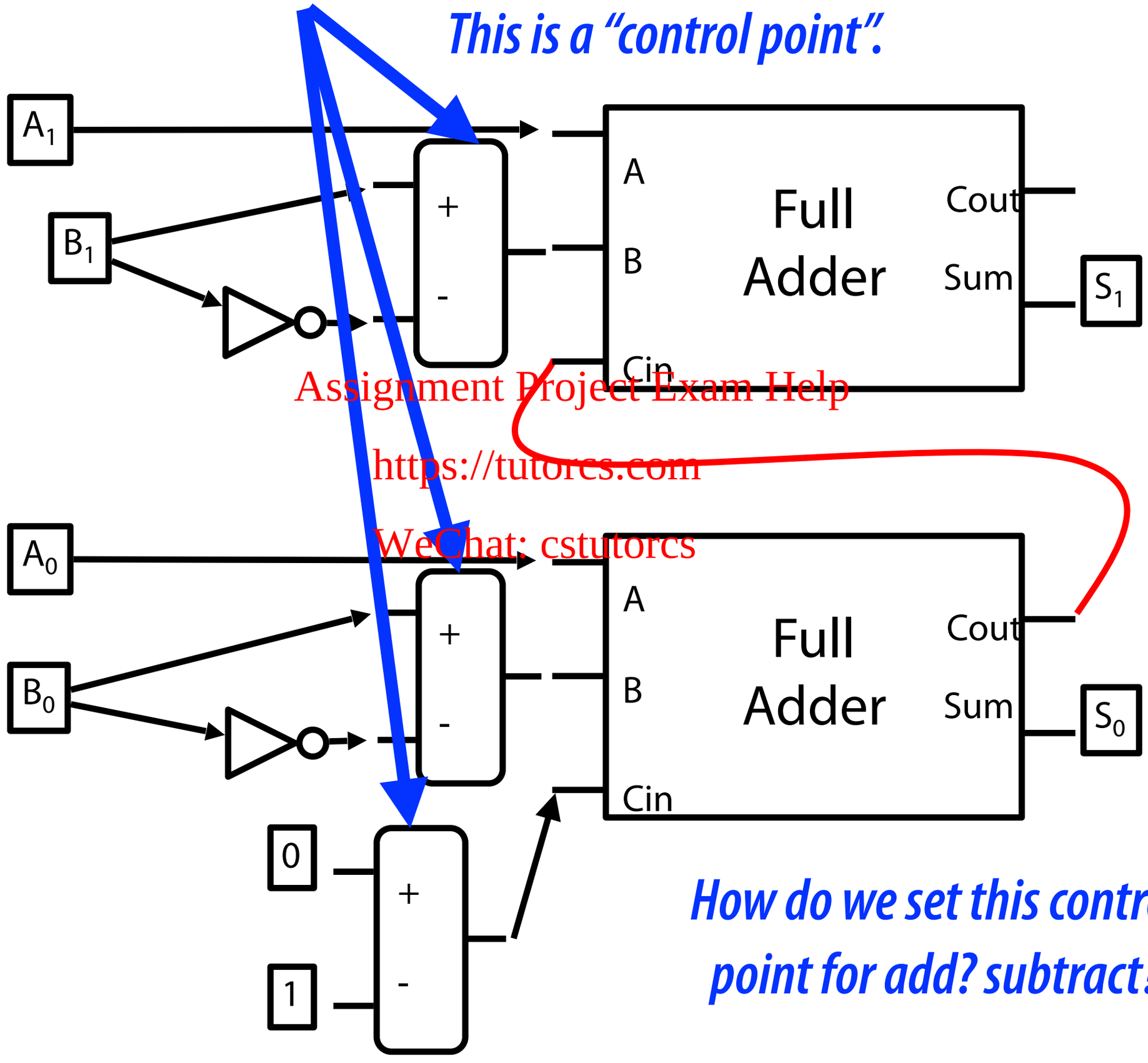
S = A + ~B + 1

… then add 1

Control for +/-

One bit controls three muxes.

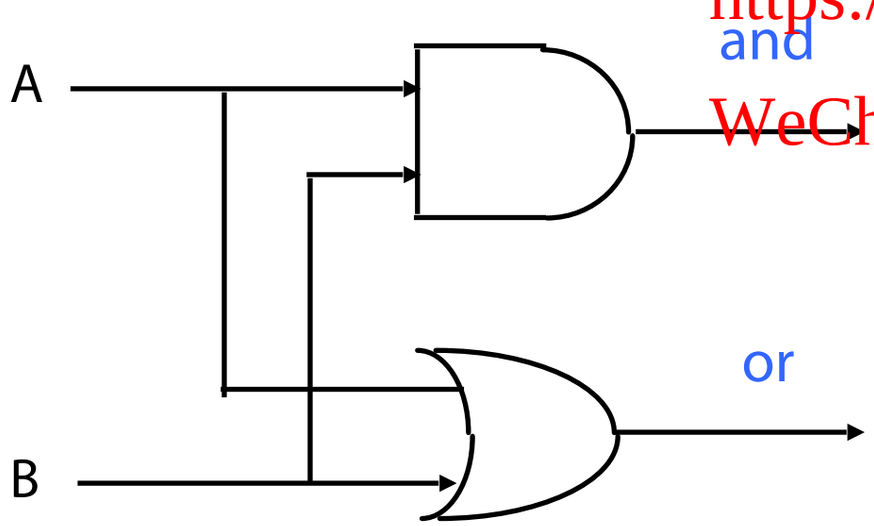
This is a "control point".



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

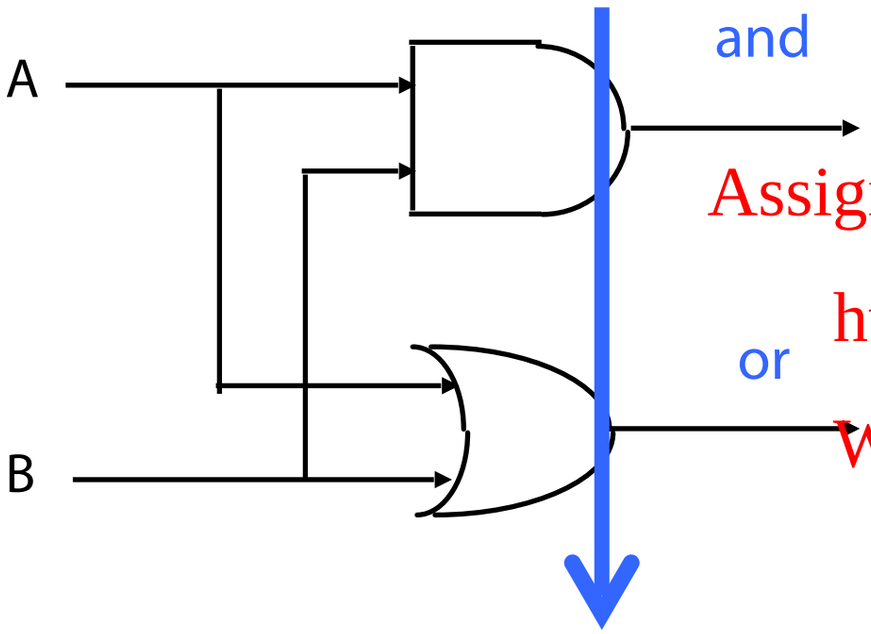
AND and OR

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



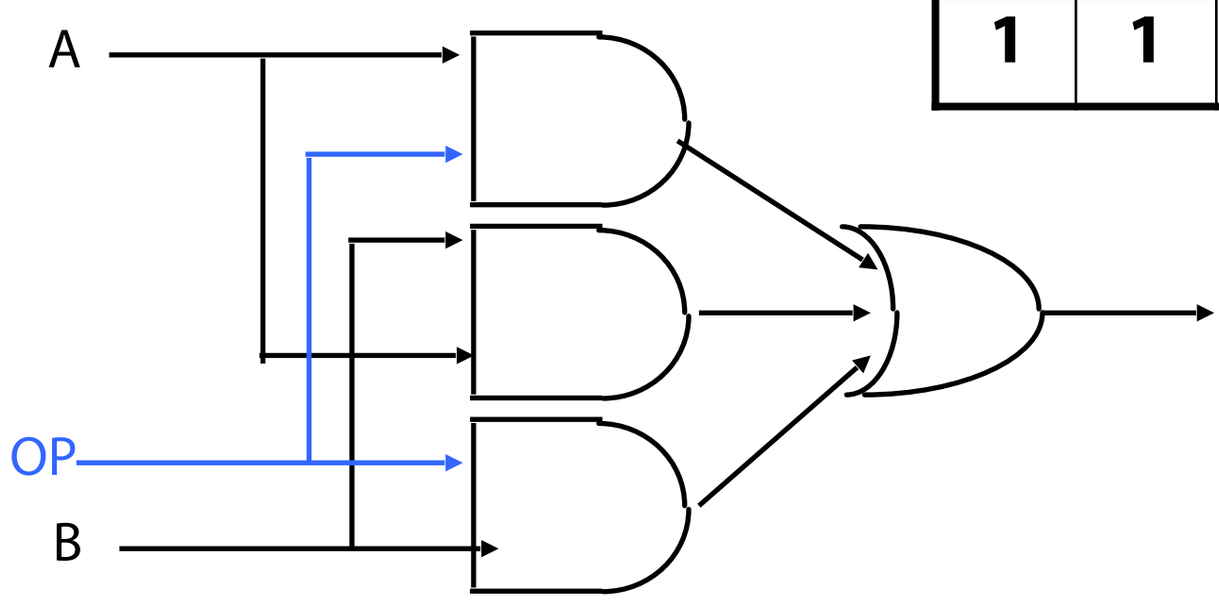
AND and OR

- 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.



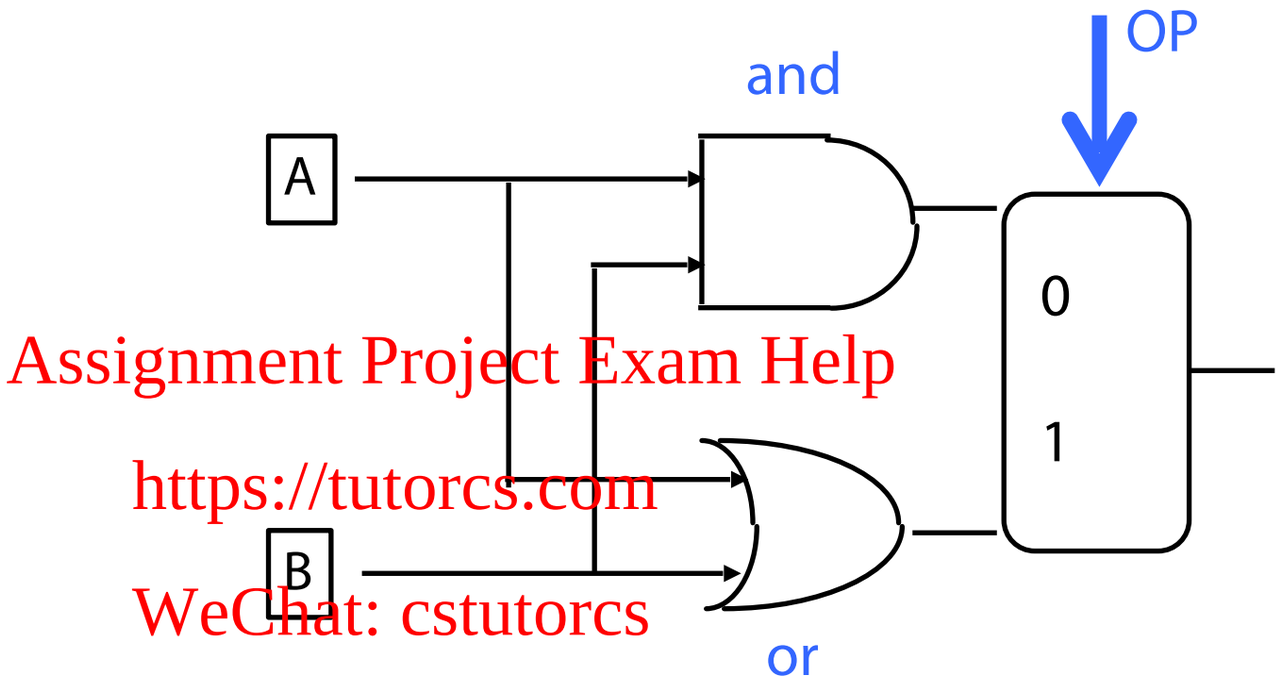
7-to-2 Combinational Logic

- **Start turning the crank . . .**

	Function	Inputs							Outputs		K-Map
		M0	M1	M2	M3	A	B	Cin	S	Cout	
0	add	0	0	0	0	0	0	0	0	0	
127											

AND and OR

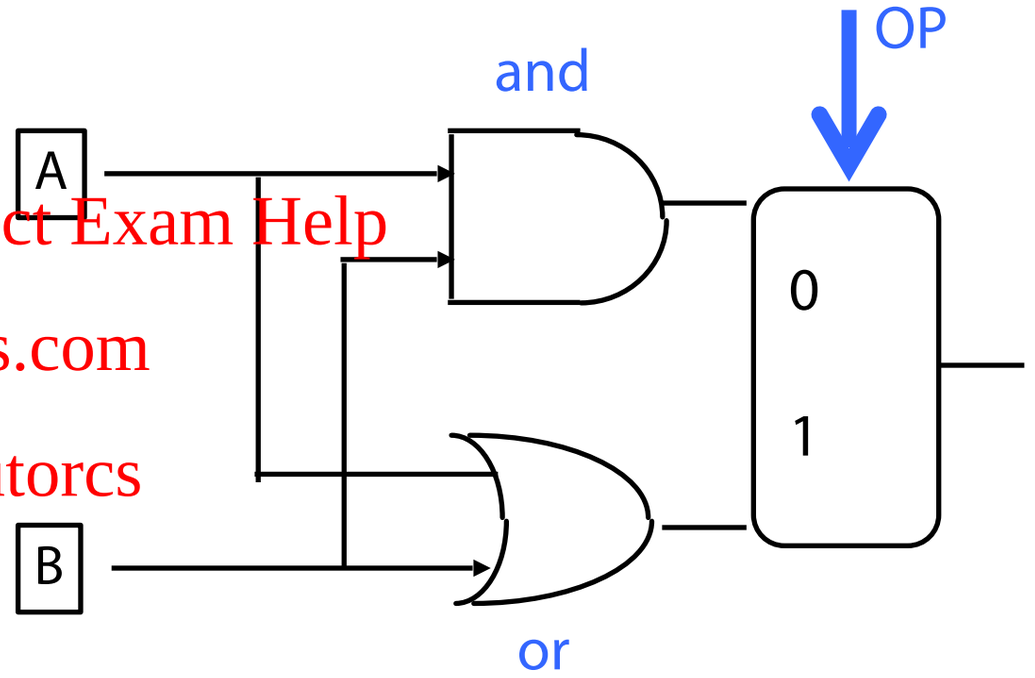
- Instead, generate several functions and use control bits to select using a mux



- 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

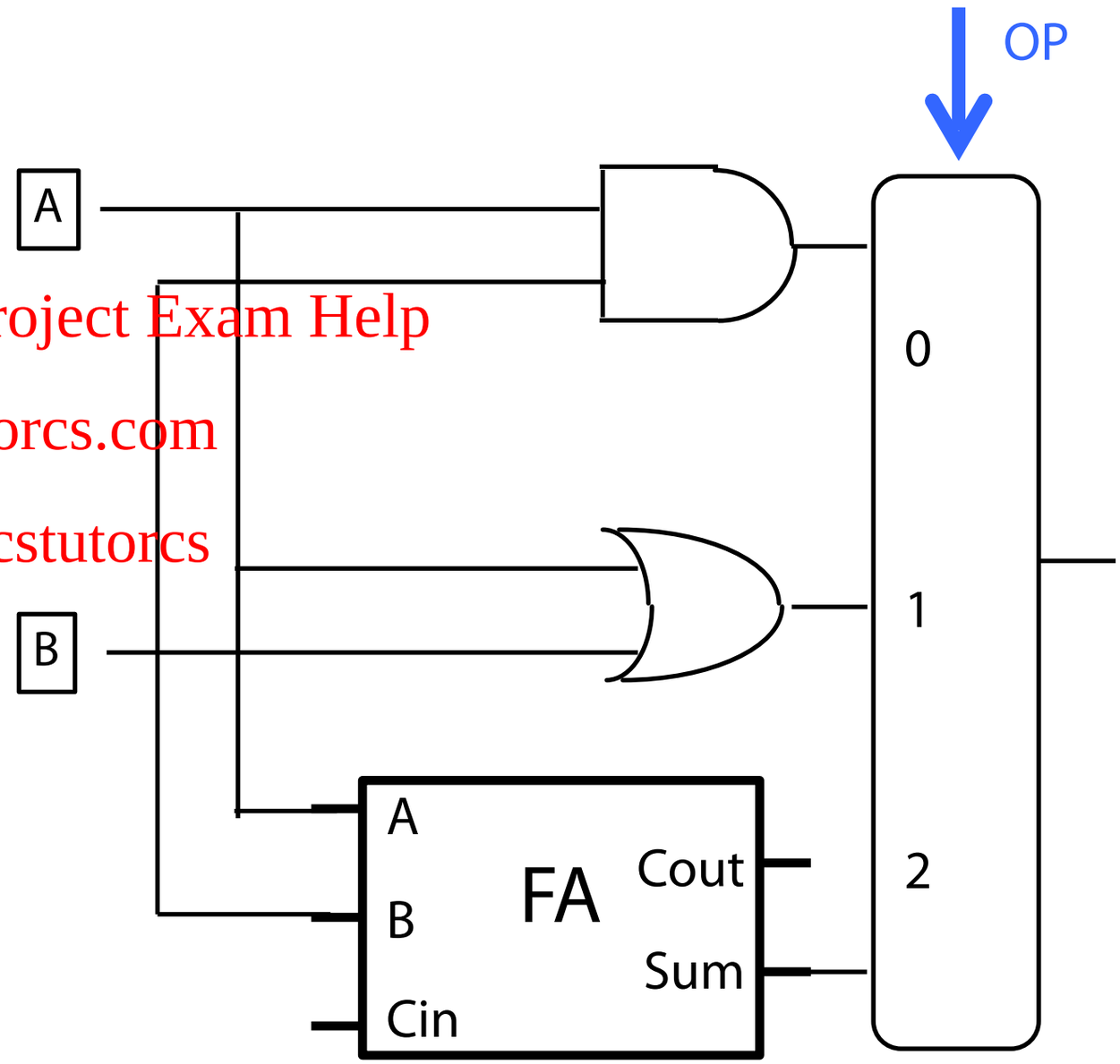
Supporting More Functions

- 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



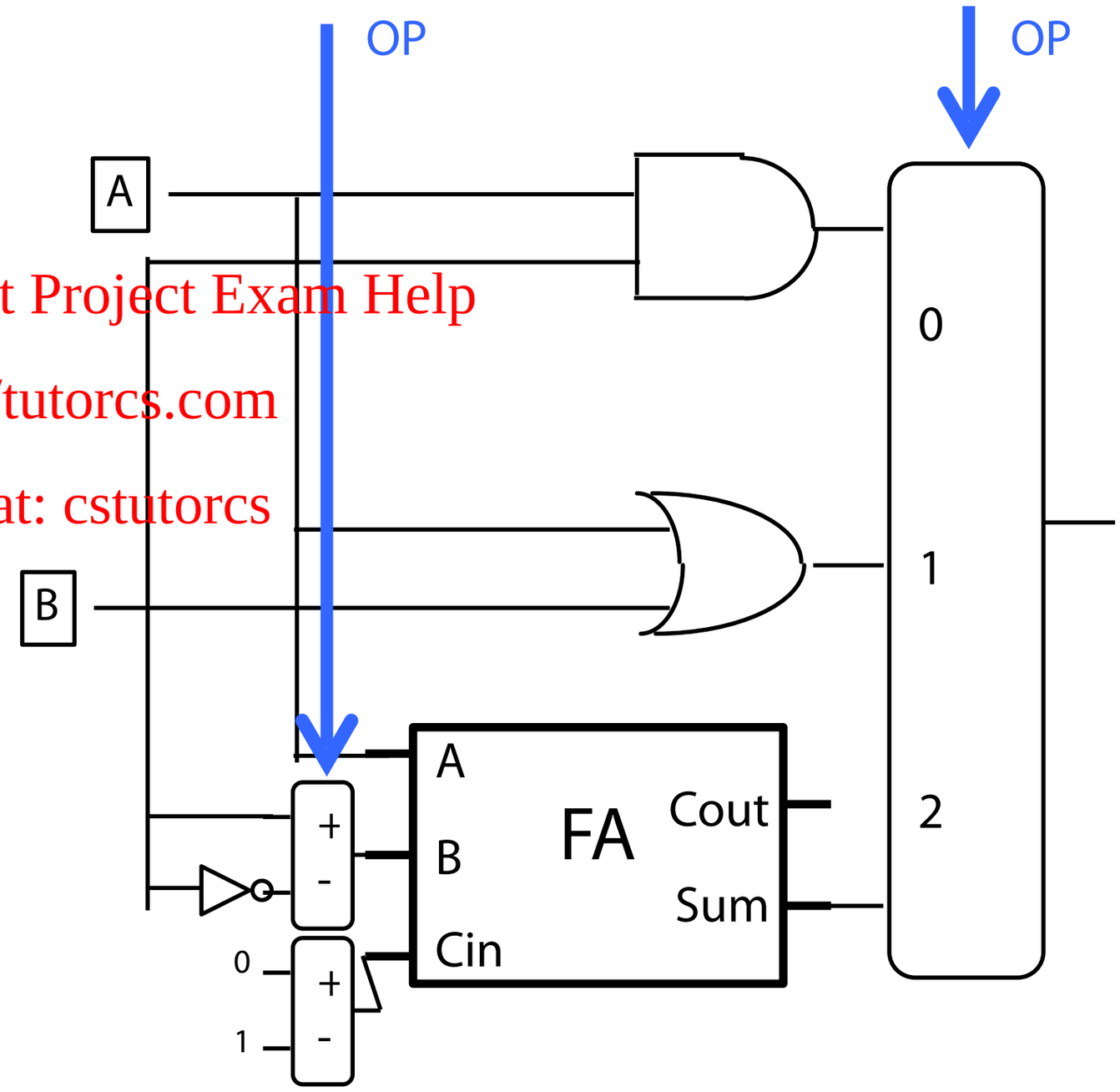
Supporting More Functions

- 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



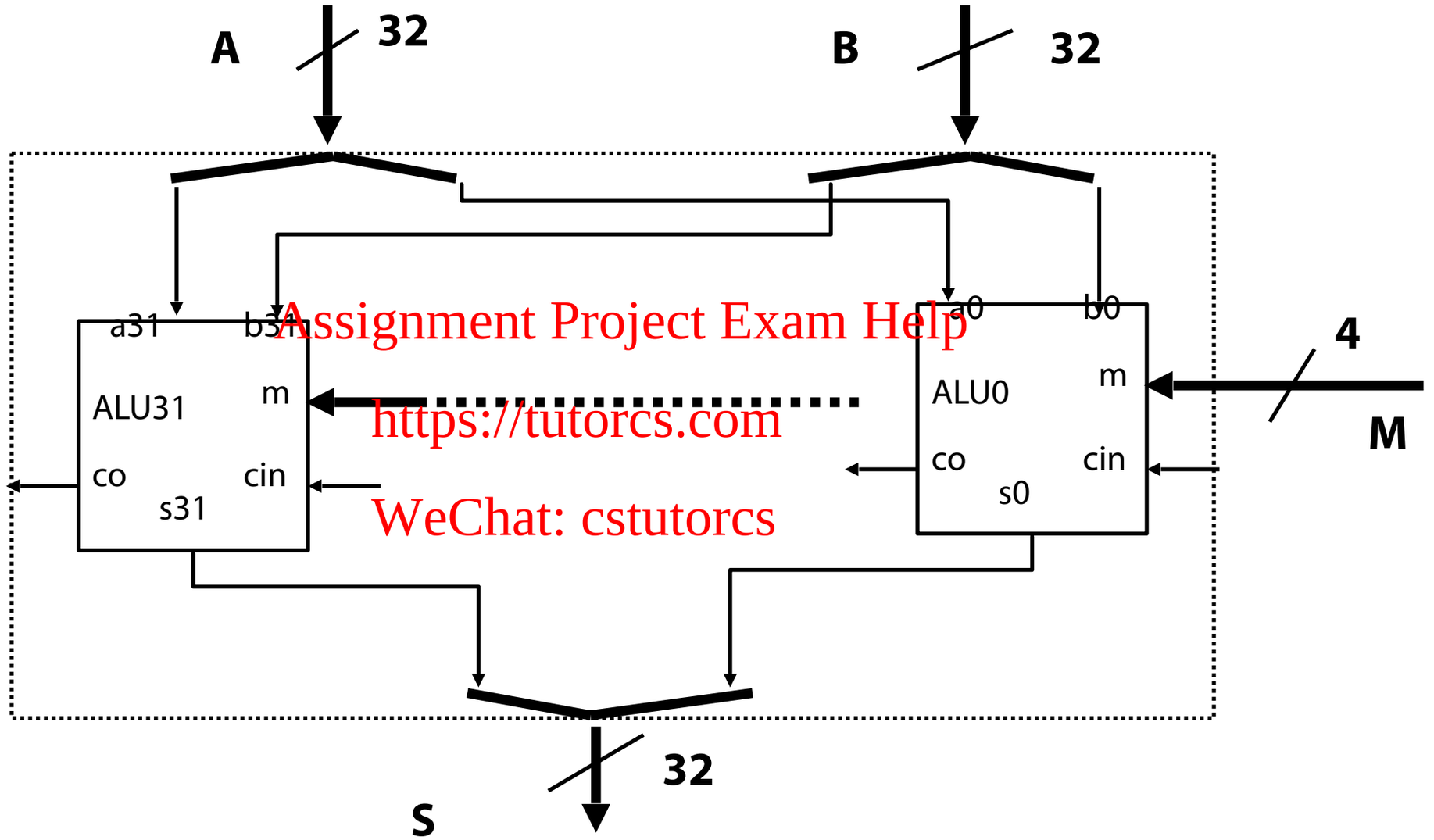
Tailoring the ALU to RISC-V

- 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 the MIPS

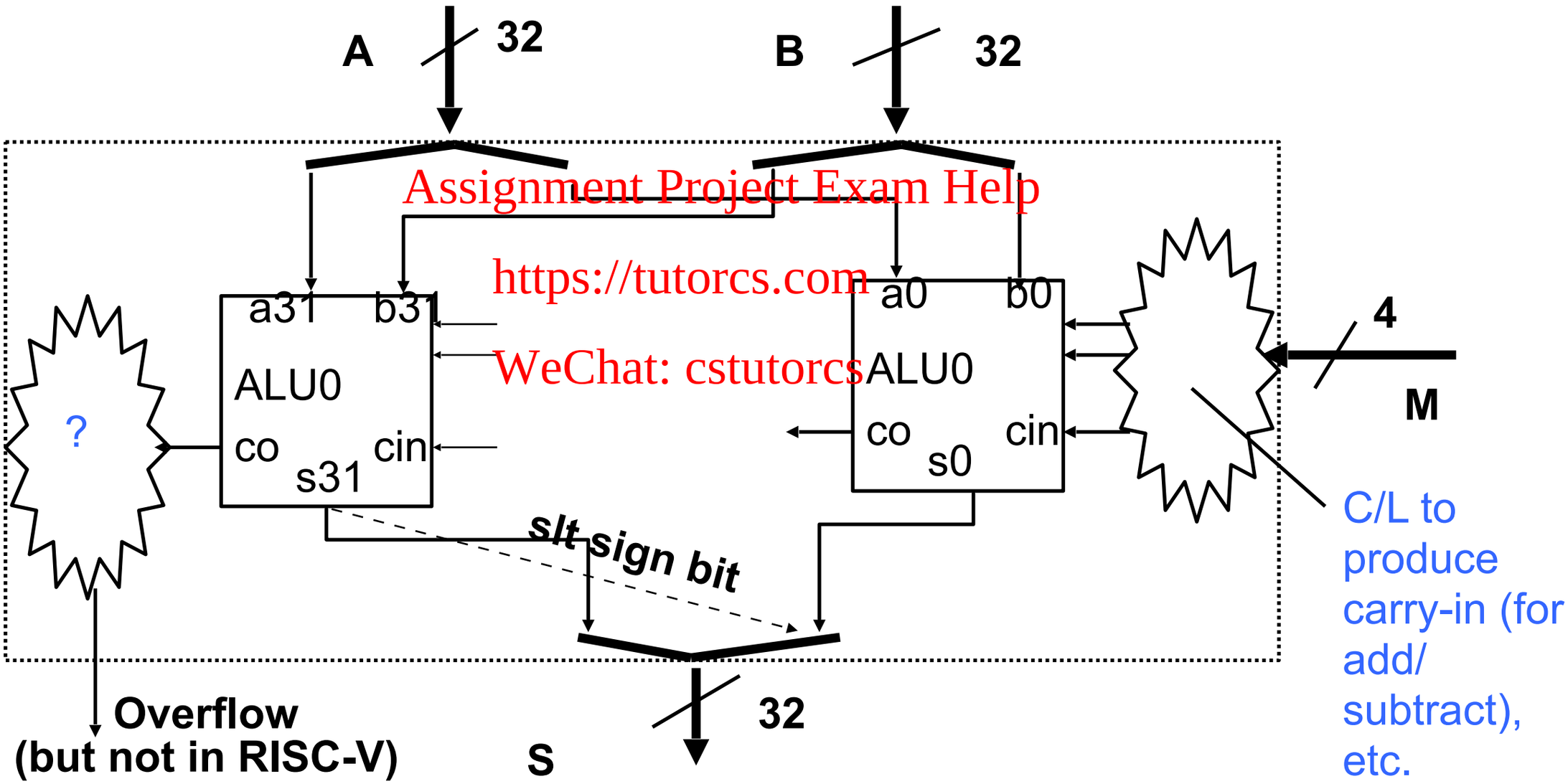
- Need to support test for equality (beq $t5, $t6, LABEL)
  - use subtraction: (a-b) = 0 implies a = b
  - How do we test if the product is zero?

Original Diagram: bit-slice ALU



Revised Diagram

- LSB and MSB need to do a little extra

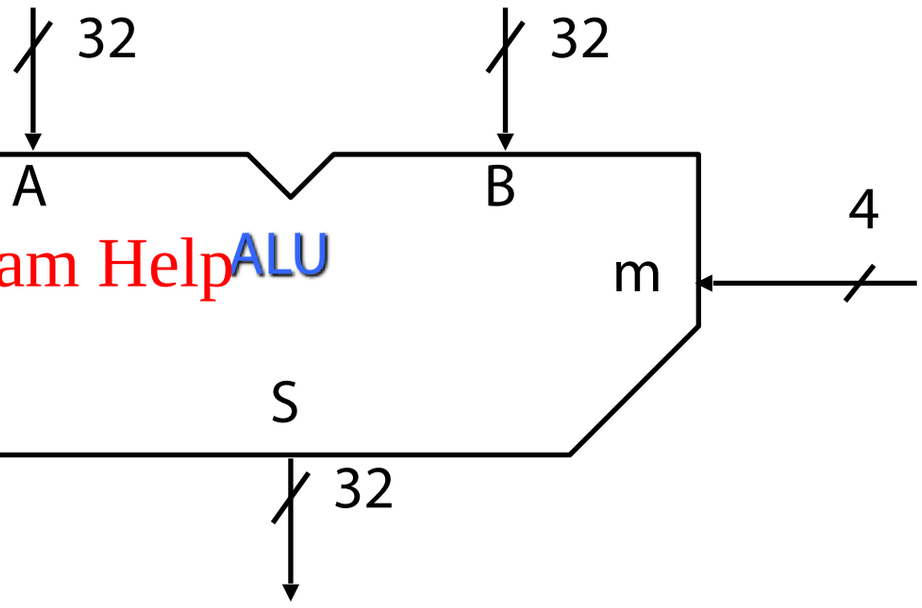


Behavioral Representation: Verilog

```
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

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

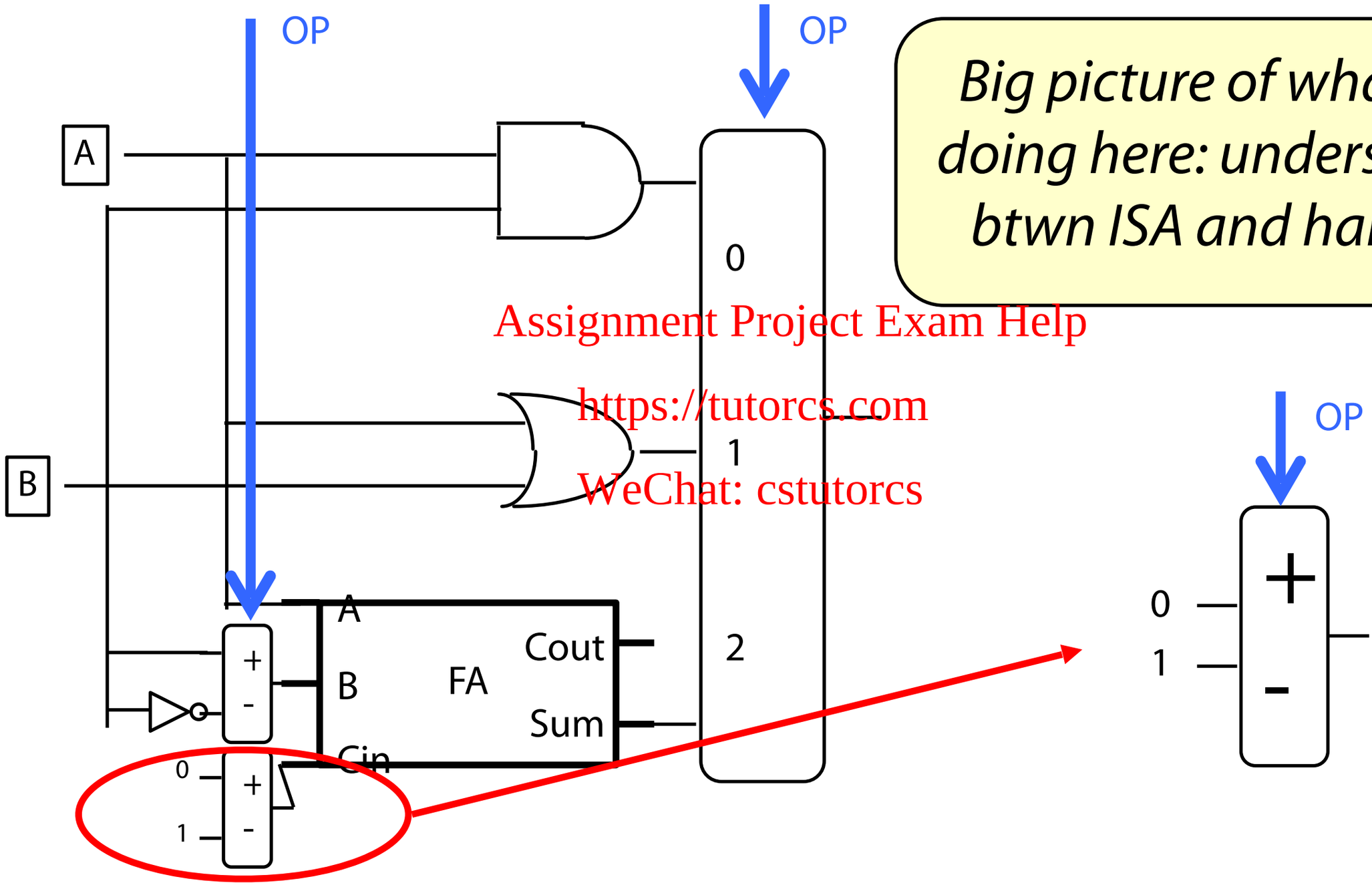
5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*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

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

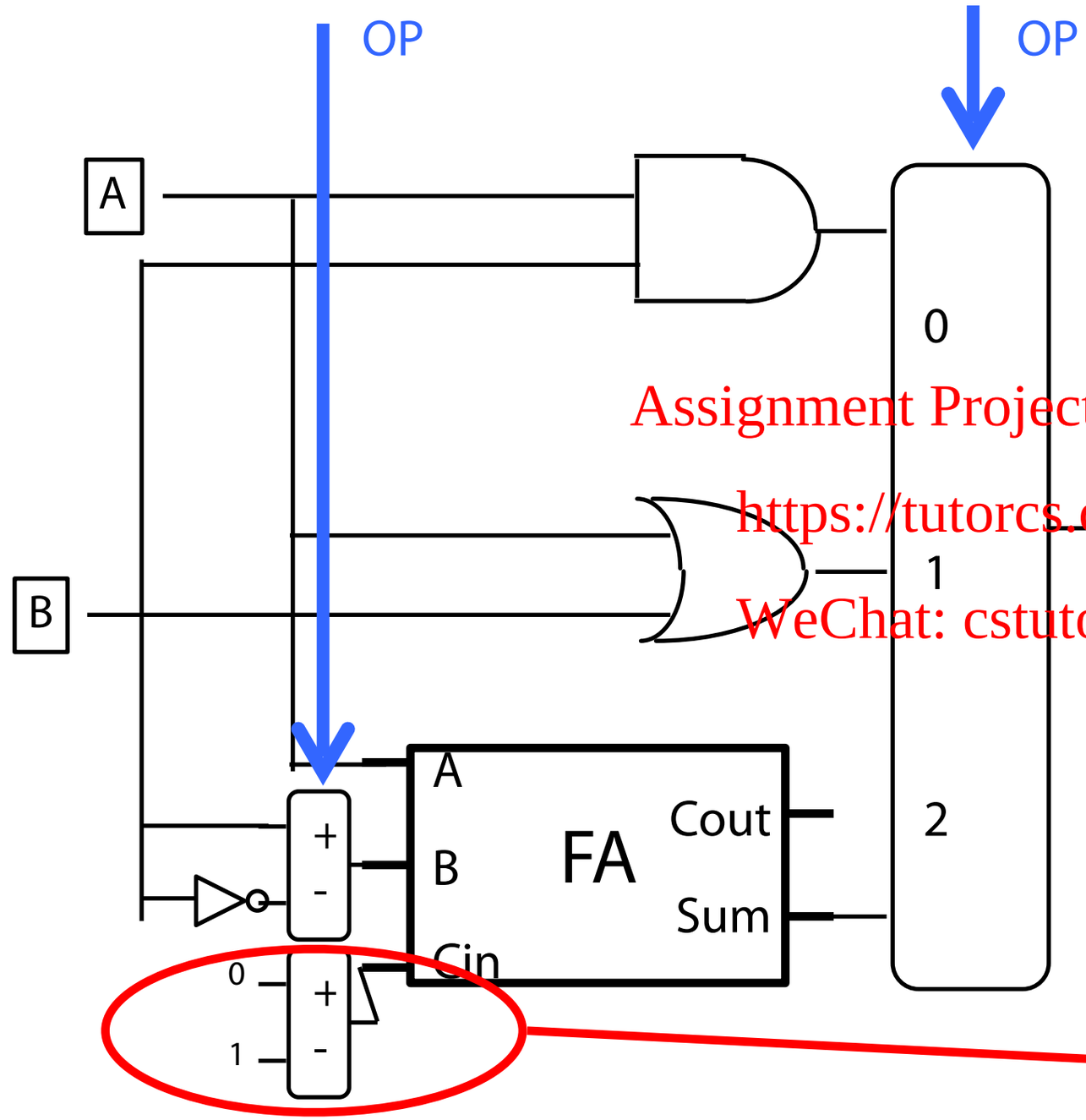
5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*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

REGIMM rt

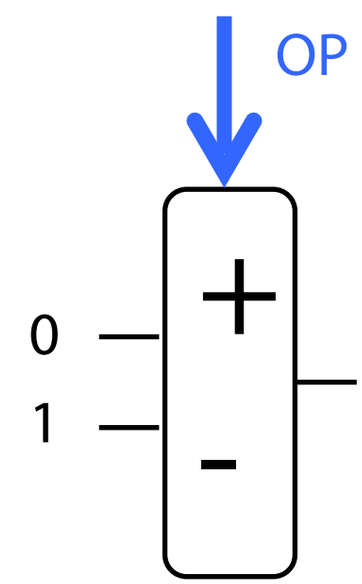
20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

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

MIPS (really) Encodings for ADD, SUB, SLT



Type	op	funct
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)



What does the red bit do? Green? Blue?

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

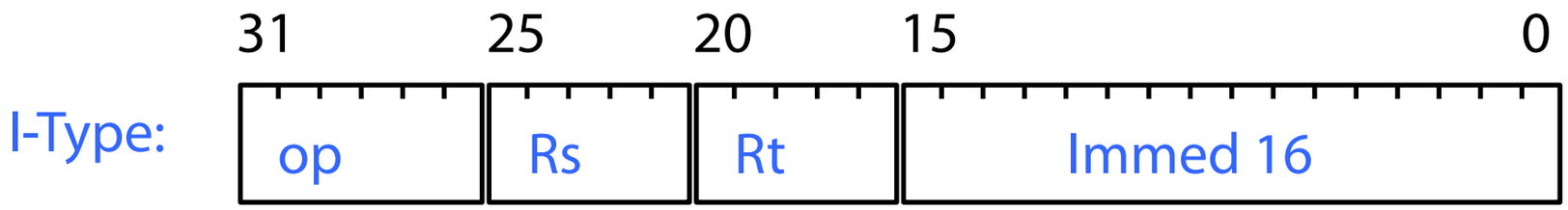
5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*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



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 excuse 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

SPECIAL function

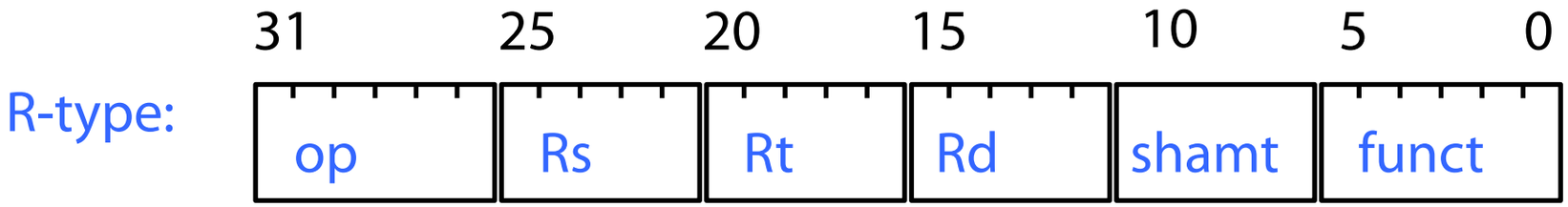
5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*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



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)

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

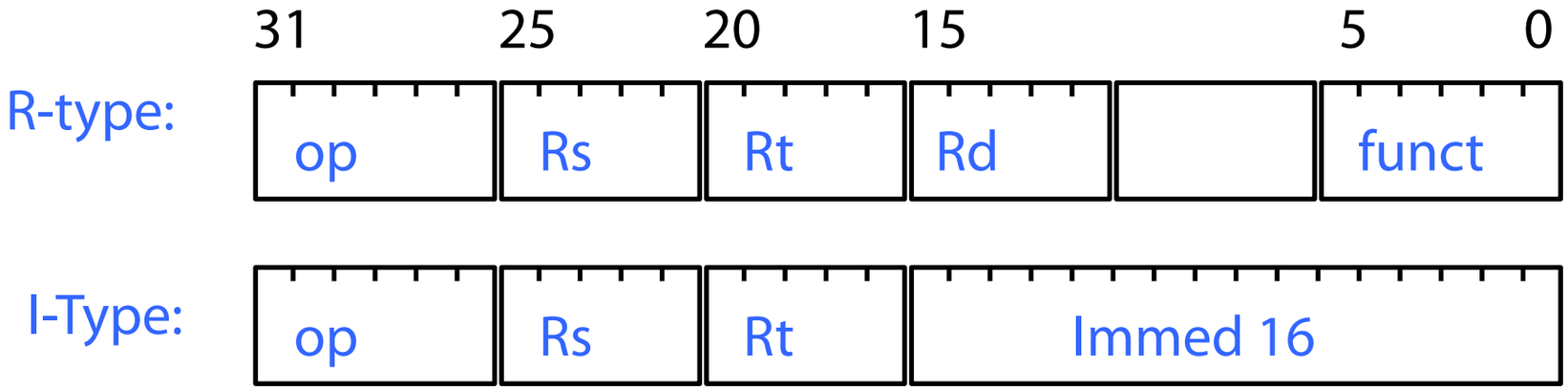
5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*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



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)

Type	op	funct
ADD	000000	100000 (040)
ADDU	000000	100001 (041)
SUB	000000	100010 (042)
SUBU	000000	100011 (043)
AND	000000	100100 (044)
OR	000000	100101 (045)
XOR	000000	100110 (046)
NOR	000000	100111 (047)