EECS4030: Computer Architecture

Arithmetic for Computers (I)

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University, Taiwan

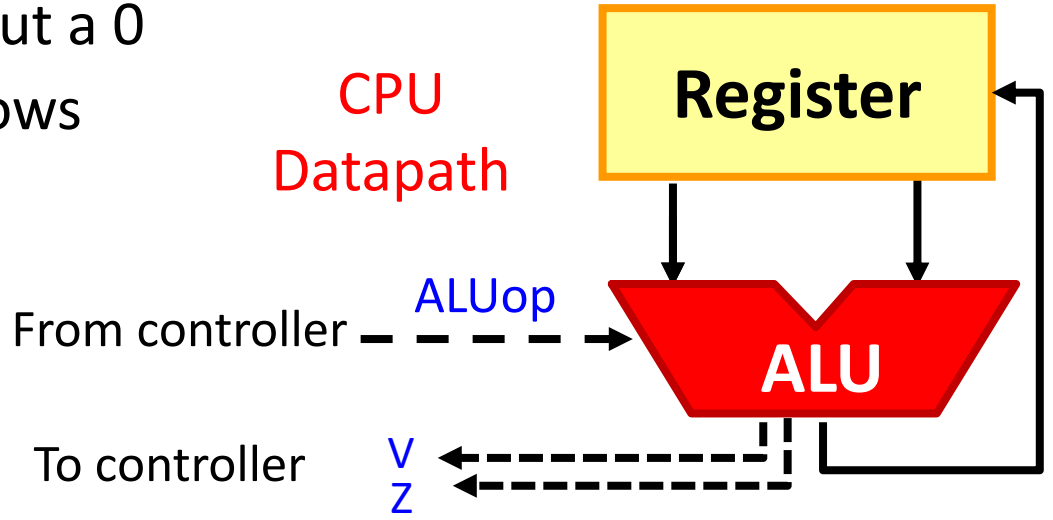
(Adapted from textbook slides https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides)

Outline

- Addition and subtraction (Sec. 3.2)
  - Constructing a basic ALU (Sec. A.5)
- Multiplication (Sec. 3.3)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)
- Parallelism and computer arithmetic: subword parallelism (Sec. 3.6)
- Streaming SIMD extensions and advanced vector extensions in x86 (Sec. 3.7)
- Subword parallelism and matrix multiply (Sec. 3.8)

Constructing a Basic ALU

- Support the following arithmetic and logic operations
  - add, sub: two's complement addition/subtraction with overflow detection
  - and, or, nor: bitwise logical AND, logical OR, logical NOR
  - slt (set-less-than): compare two input numbers (by subtraction) and output a 1 if first is less than second, otherwise output a 0
  - V: result overflows
  - Z: result is zero



Basic ALU and RISC-V Instructions

- ALU operations for executing RISC-V instructions
  - slt (set-less-than): **`slt`**, **`sltu`**, **`slti`**, **`sltiu`**, …

    `slt x9,x21,x22` x9 ← 1, if x21 < x22
    x9 ← 0, else
  - add, sub: **`add`**, **`addi`**, **`sub`**, **`beq`**, **`bne`**, **`blt`**, **`bge`**, **`ld`**, **`sd`**, **`slt`**, **`sltu`**, **`slti`**, **`sltiu`**, …
  - and, or: **`and`**, **`andi`**, **`or`**, **`ori`**, …
  - nor: not used in RISC-V
  - Z: **`beq`**, …

    `beq x21,x22,L1` jump to L1 if x21 = x22

    check if x21 – x22 = 0 is true ➔ Z ← 1

Functional Specification

ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	sub
0111	set-less-than
1100	nor

Fig. A.5.13

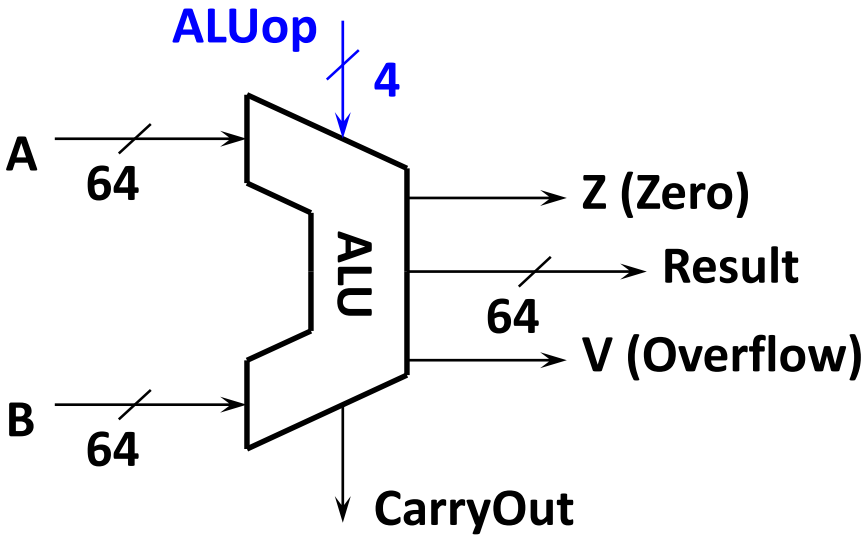
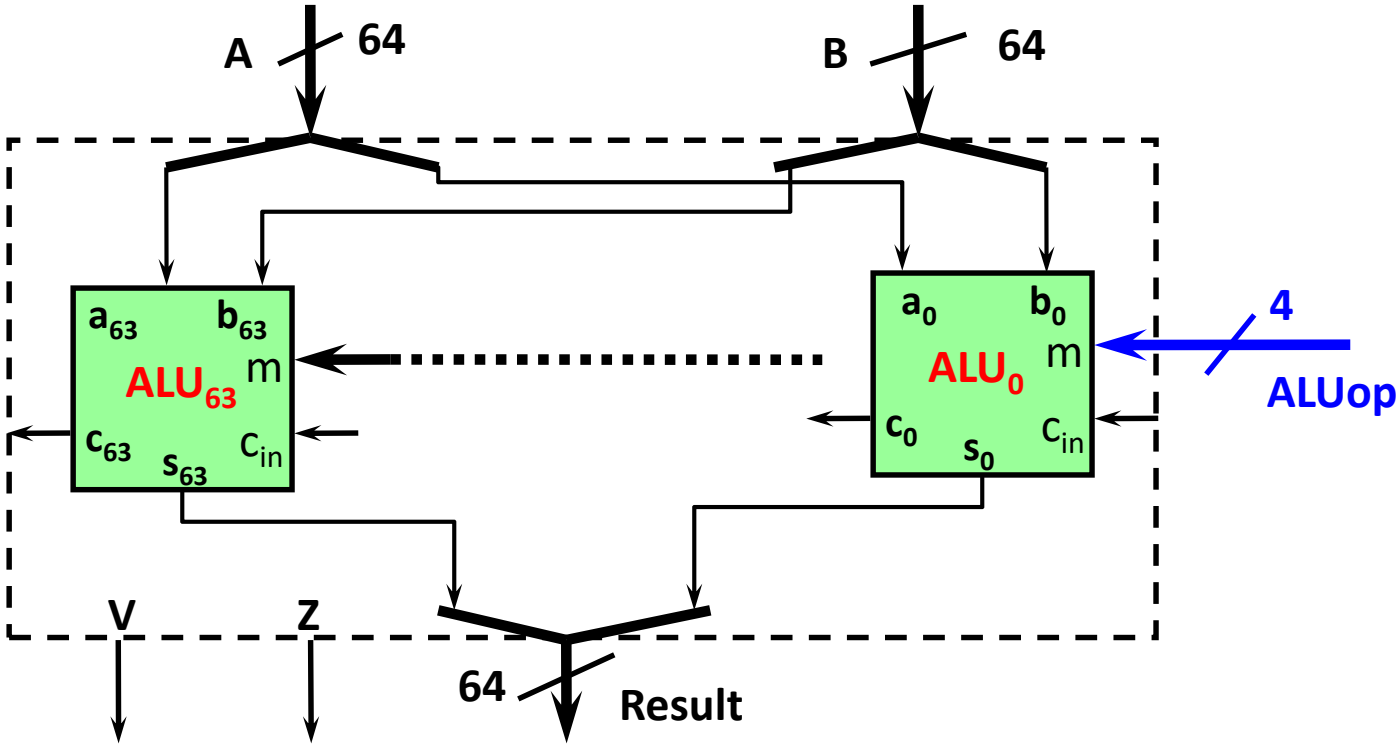


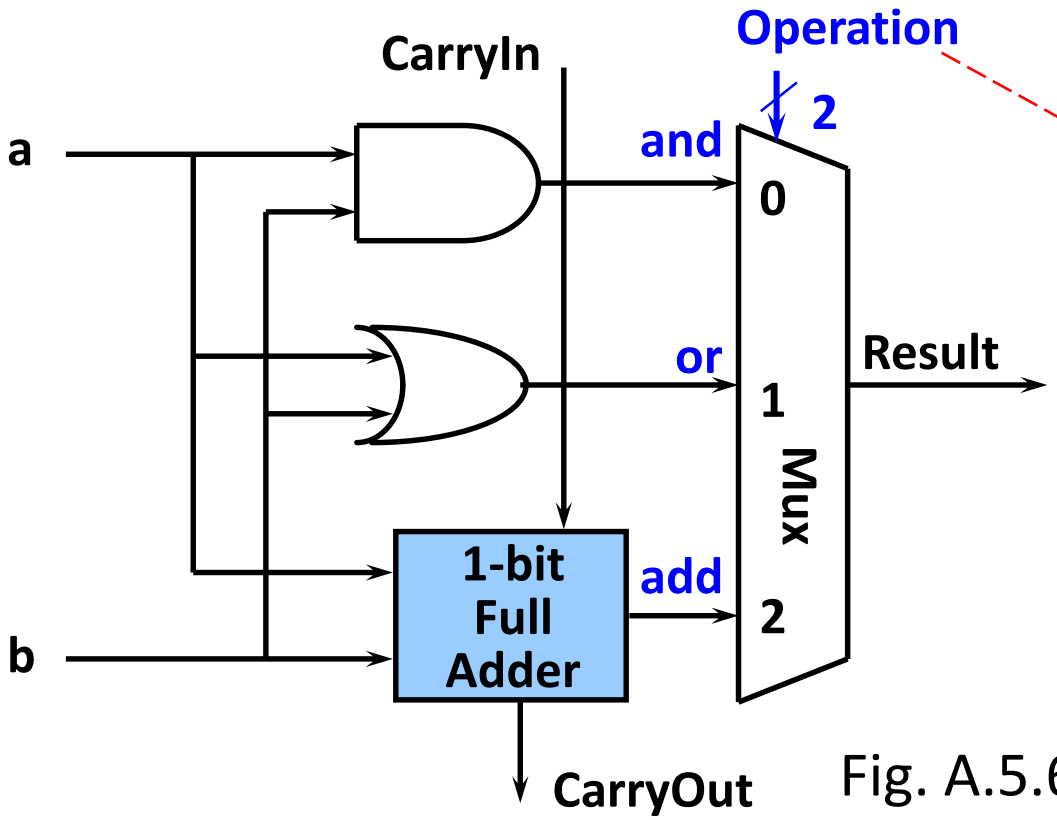
Fig. A.5.14

Design Strategy: Bit-slice ALU

- Design trick: divide and conquer
  - Break the problem into simpler problems, solve individual problems and glue together the solutions



Zoom-in to 1-bit ALU



ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	sub
0111	set-on-less
1100	nor

Fig. A.5.6

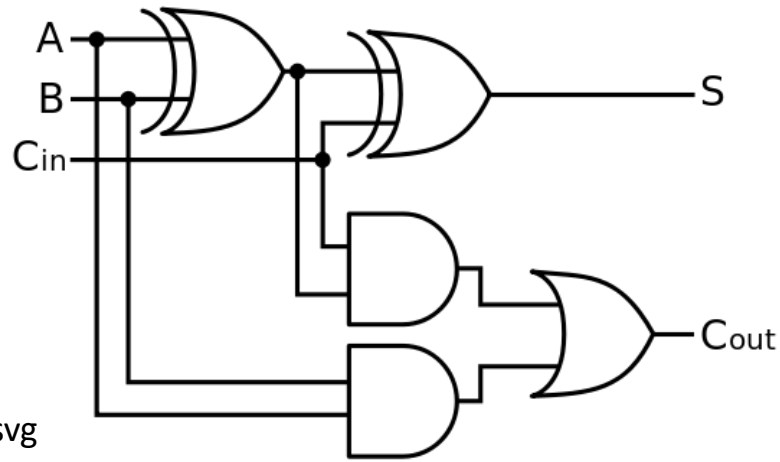
All logic gates operate to produce results and only the desired one is chosen

Only combinational logic

Recall 1-Bit Full Adder

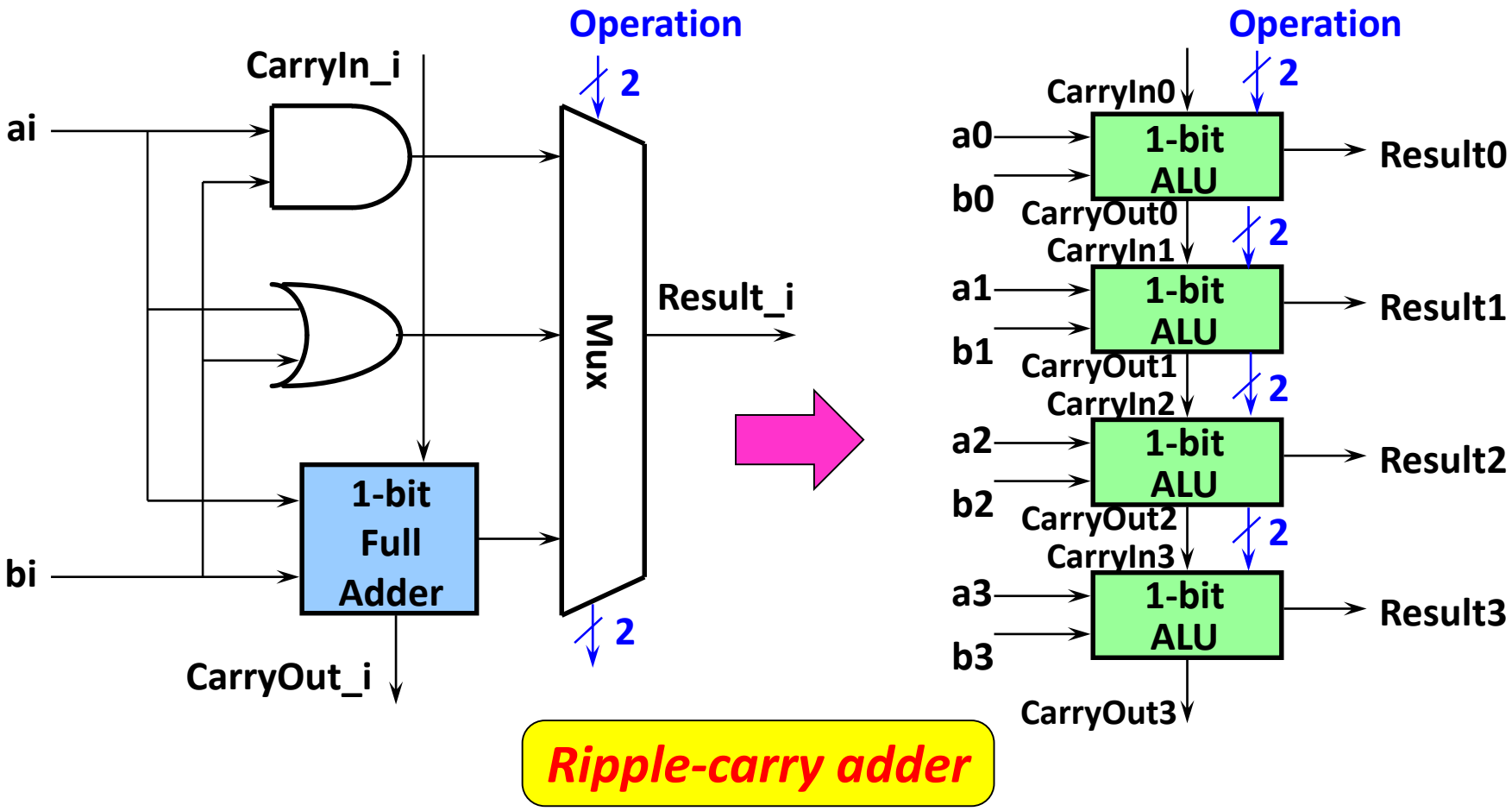
Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fig. A.5.3



https://commons.wikimedia.org/wiki/File:Full-adder.svg

From 1-bit to 4-bit ALU



How about Subtraction?

- 2's complement operation: take inverse of every bit and add 1 (at c_{in} of first stage)
- So, subtraction becomes addition of 2's complement
  - A – B = A + (-B) = A + (B' + 1) = A + B' + 1
  - B': bit-wise inverse of B

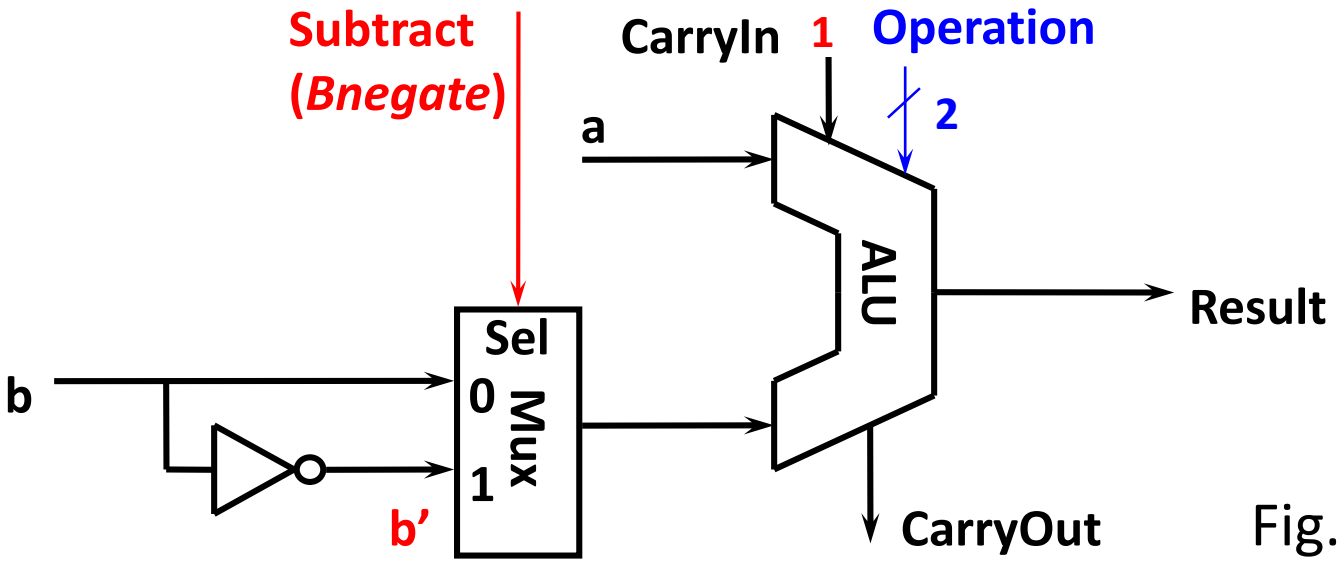
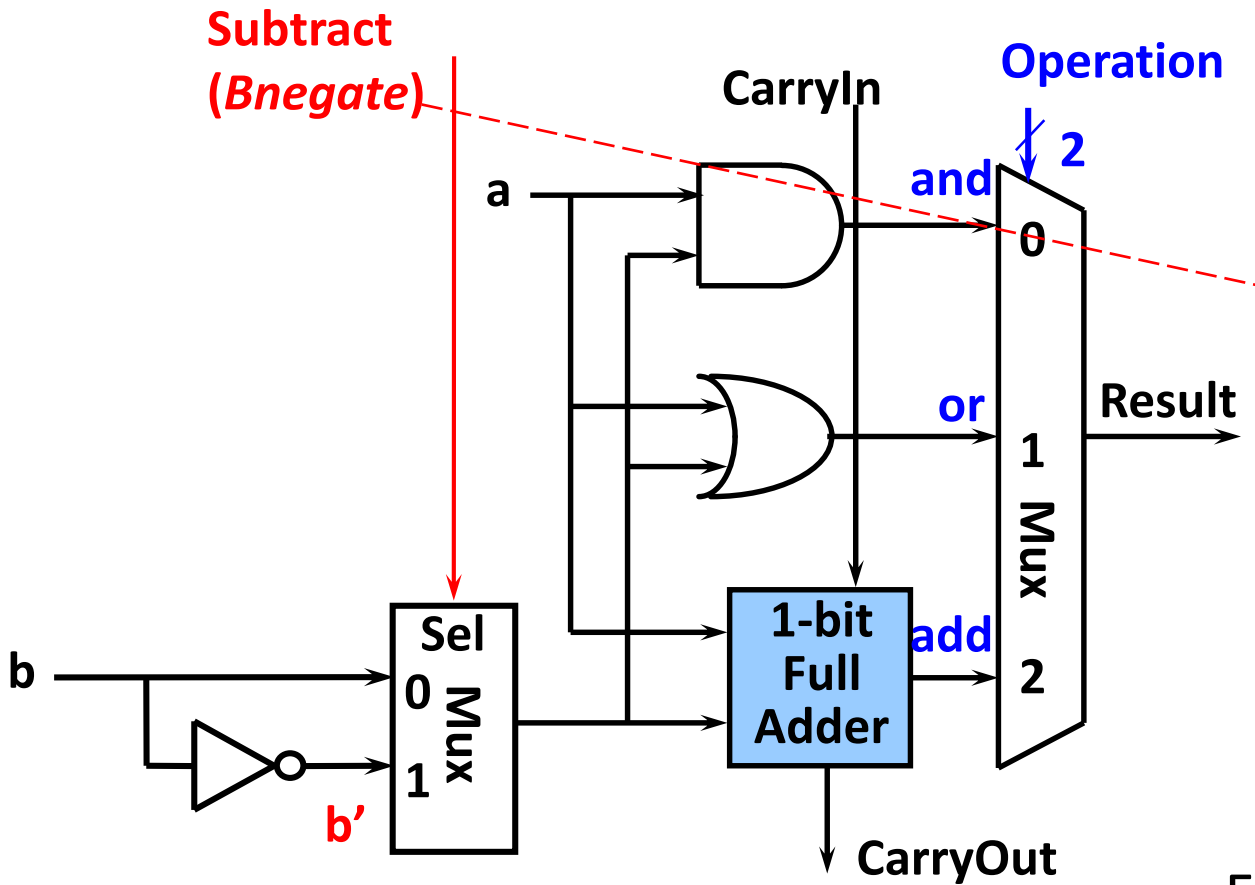


Fig. A.5.8

Zoom-in to 1-bit ALU

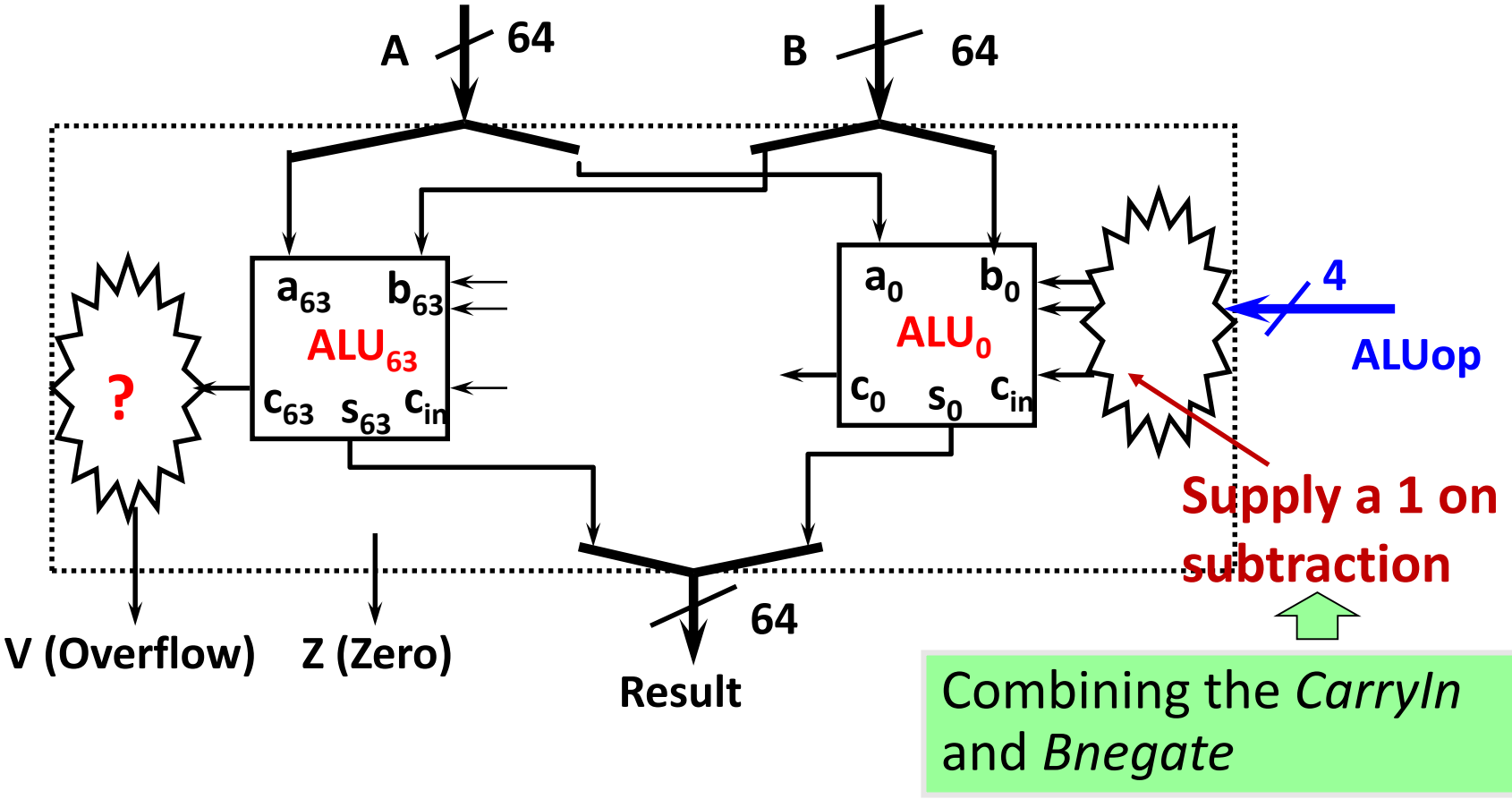


ALUop	Function
0000	and
0001	or
0010	add
0110	sub
0111	set-on-less
1100	nor

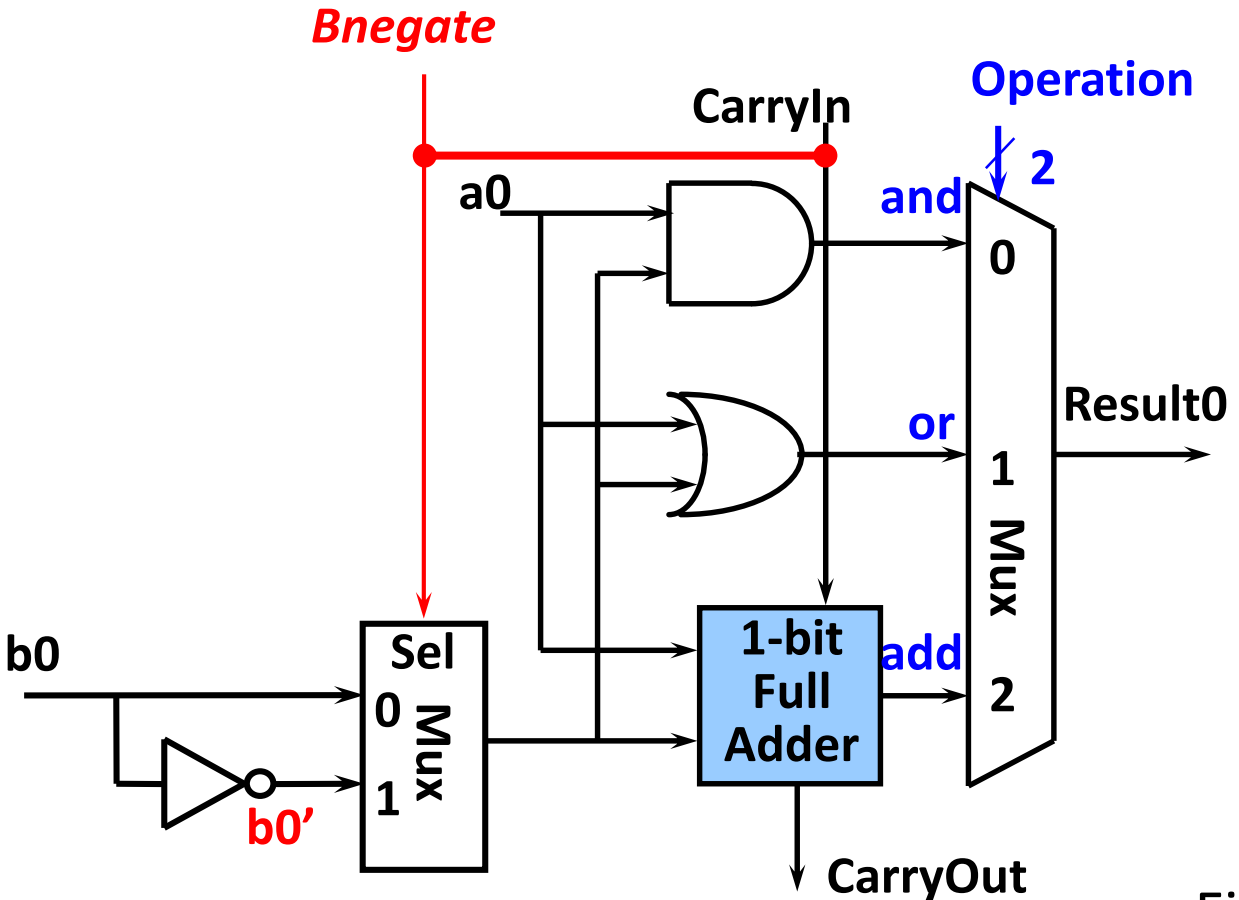
Fig. A.5.8

Revised Diagram

- LSB and MSB need to do a little extra



Zoom-in to Rightmost Bit of ALU



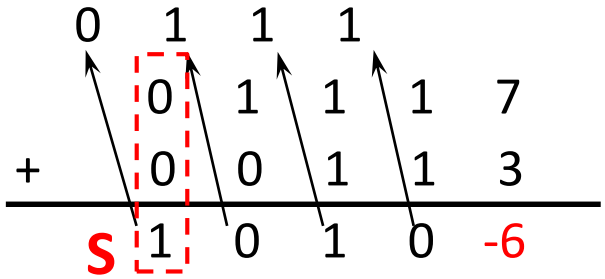
ALUop	Function
0000	and
0001	or
0010	add
0110	sub
0111	set-on-less
1100	nor

Fig. A.5.8

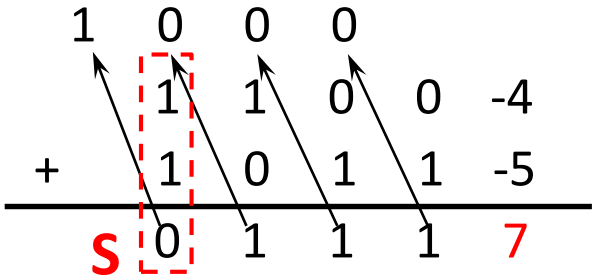
Overflow of Signed Integer Addition

Decimal	Binary	Decimal	2's complement
0	0000	0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

Ex: 7 + 3 = 10 but ...



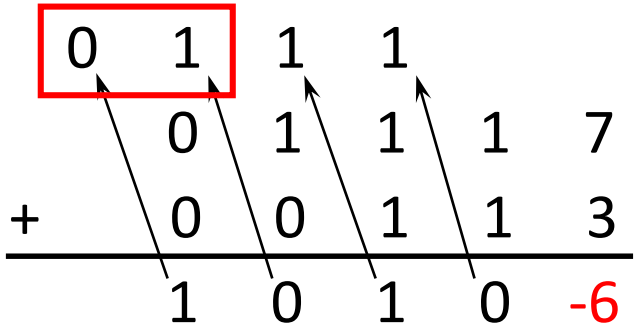
- 4 - 5 = - 9 but ...

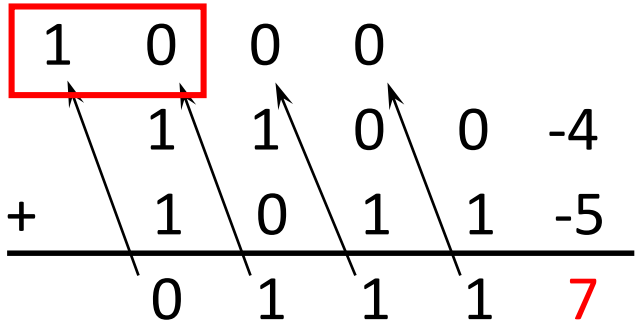


Overflow Detection

- Overflow: result too big/small to represent
  - -8 ≤ 4-bit binary number ≤ 7
  - Adding operands with different signs → no overflow
  - Overflow occurs when adding:
    - 2 positive numbers and the sum is negative
    - 2 negative numbers and the sum is positive

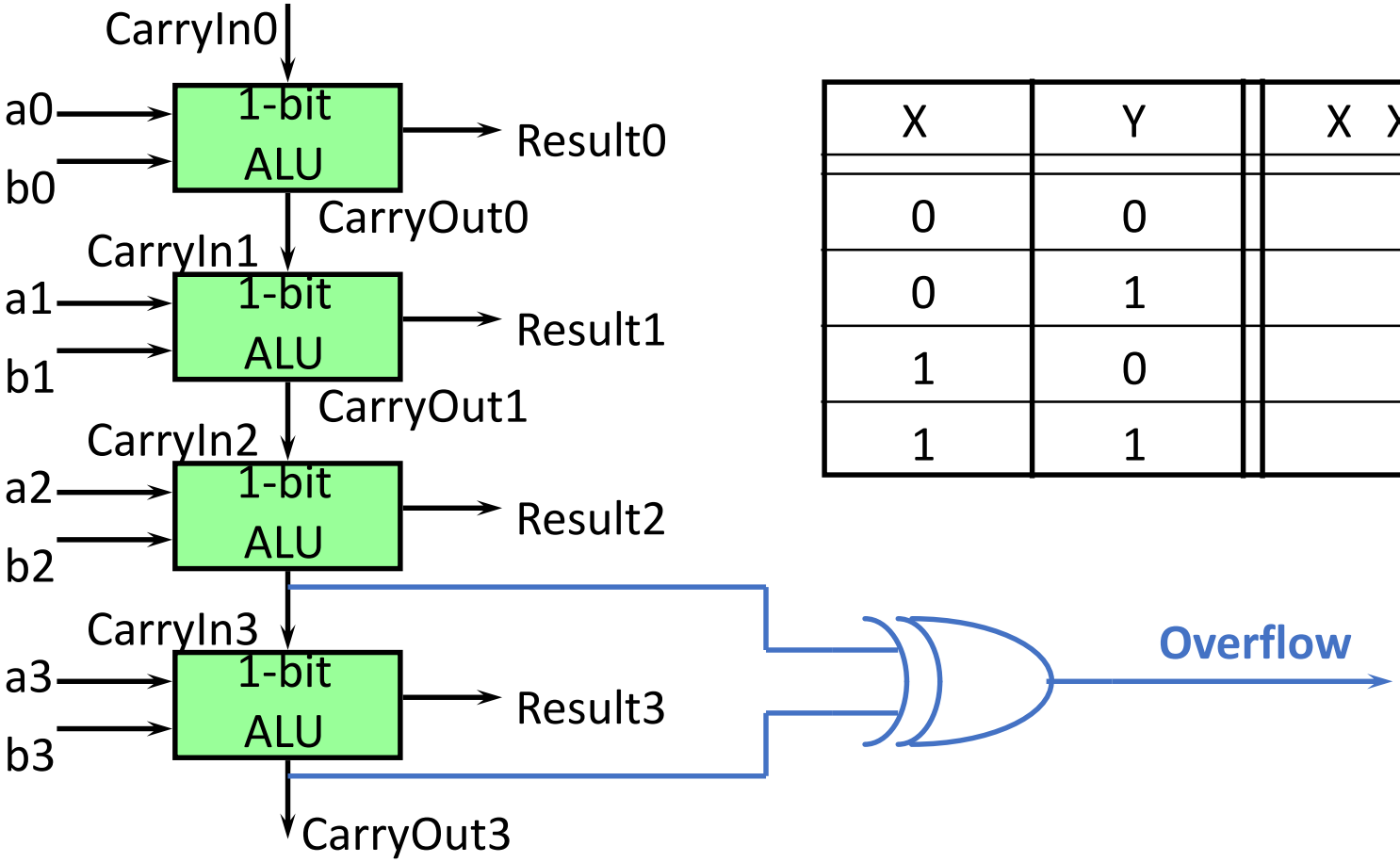
    → sign bit is set with the value of the result
  - Overflow: if Carry into MSB ≠ Carry out of MSB





Overflow Detection Logic

- Overflow = CarryIn[N-1] XOR CarryOut[N-1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

NOR Operation

- A nor B = (not A) and (not B)

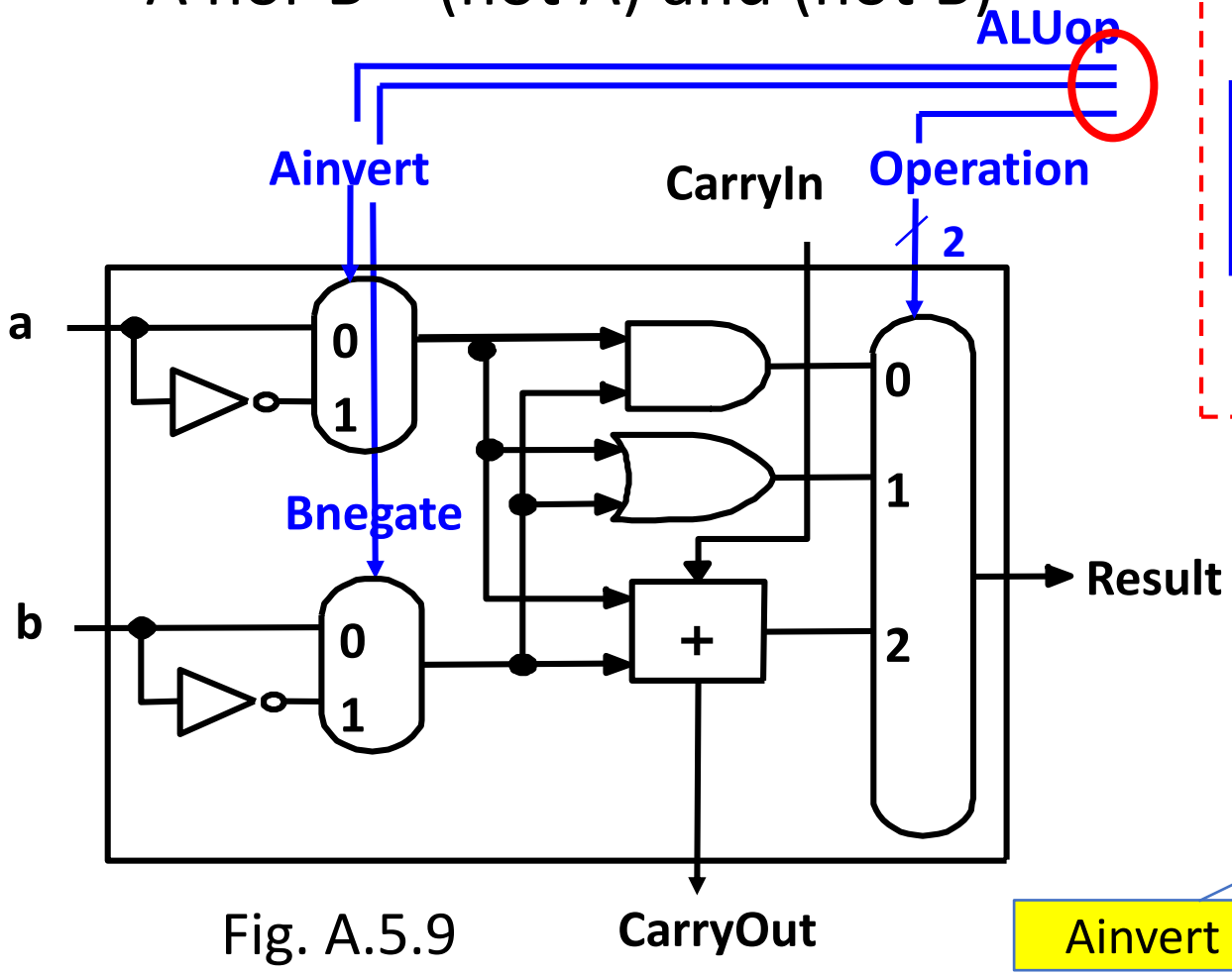
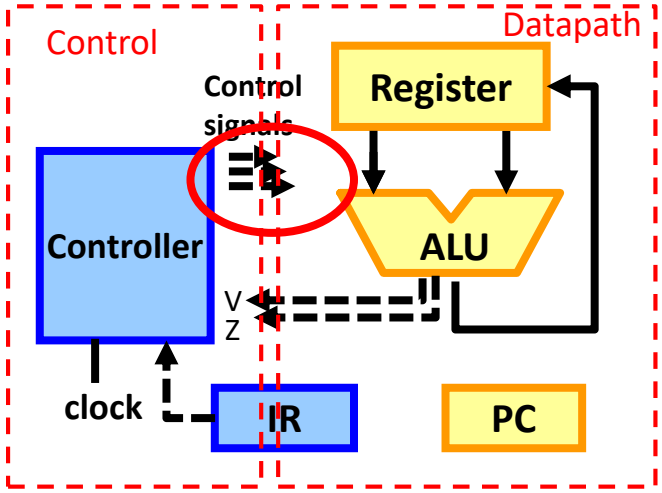


Fig. A.5.9

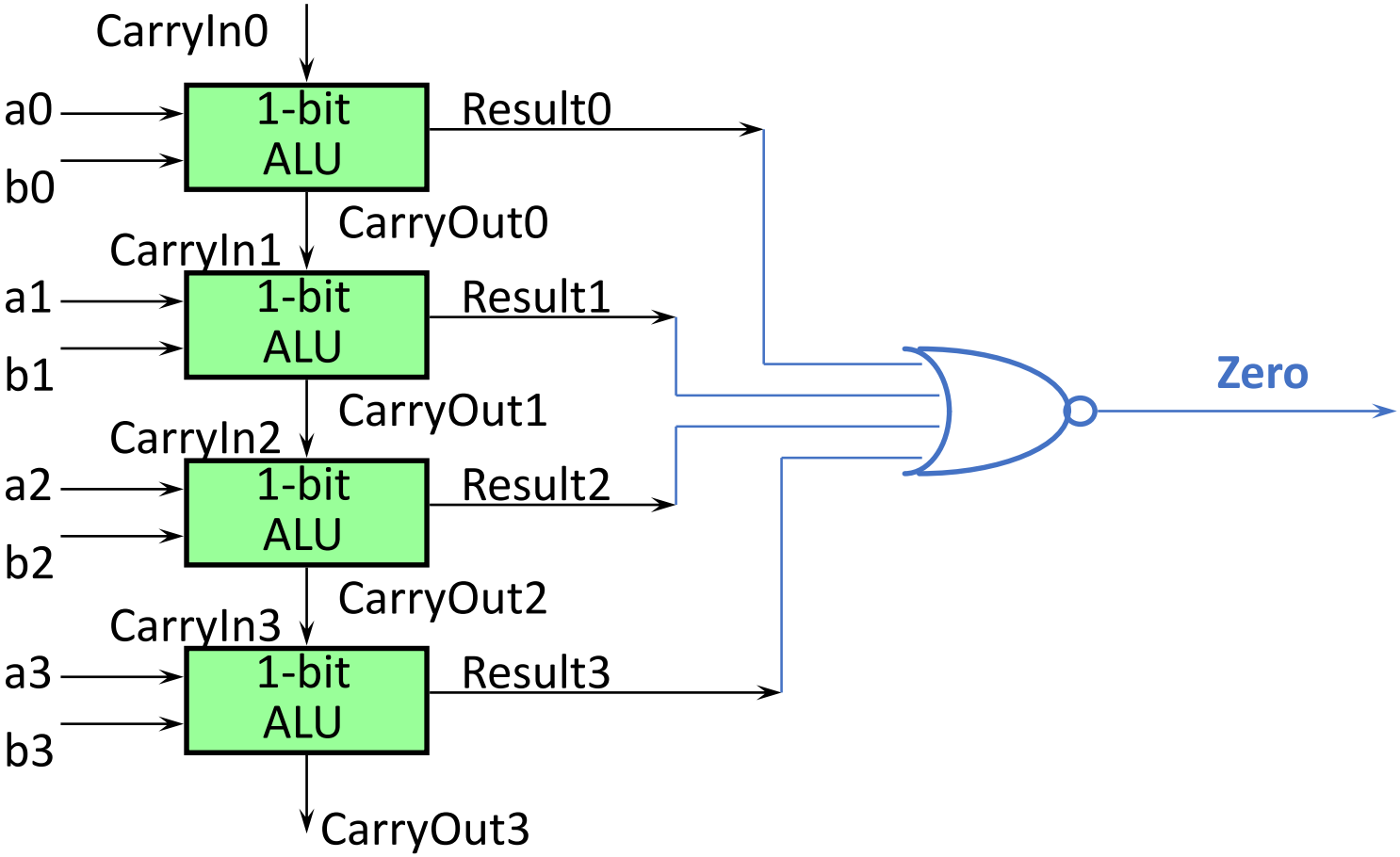


ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

Ainvert | Bnegate | operation

Zero Detection Logic

- To support conditional jump: by a one BIG NOR gate



Set-Less-Than

- If A < B, then Result = 00..01, else Result = 00..00
- If (A – B) < 0, then …
- If A – B & sign of Result is 1, then …
- If A – B & $Result_{63}$ = 1, then …
- If A – B & $Result_{63}$ = 1, then $Result_0$ =1, else $Result_0$ =0, and $Result_{63}$ ~ $Result_1$ = 0
- A – B & $Result_0$ = $Result_{63}$ & $Result_{63}$ ~ $Result_1$ = 0
  - No need to do if-then-else test

Set-Less-Than (I)

- 1-bit in ALU (for bits 0~62)

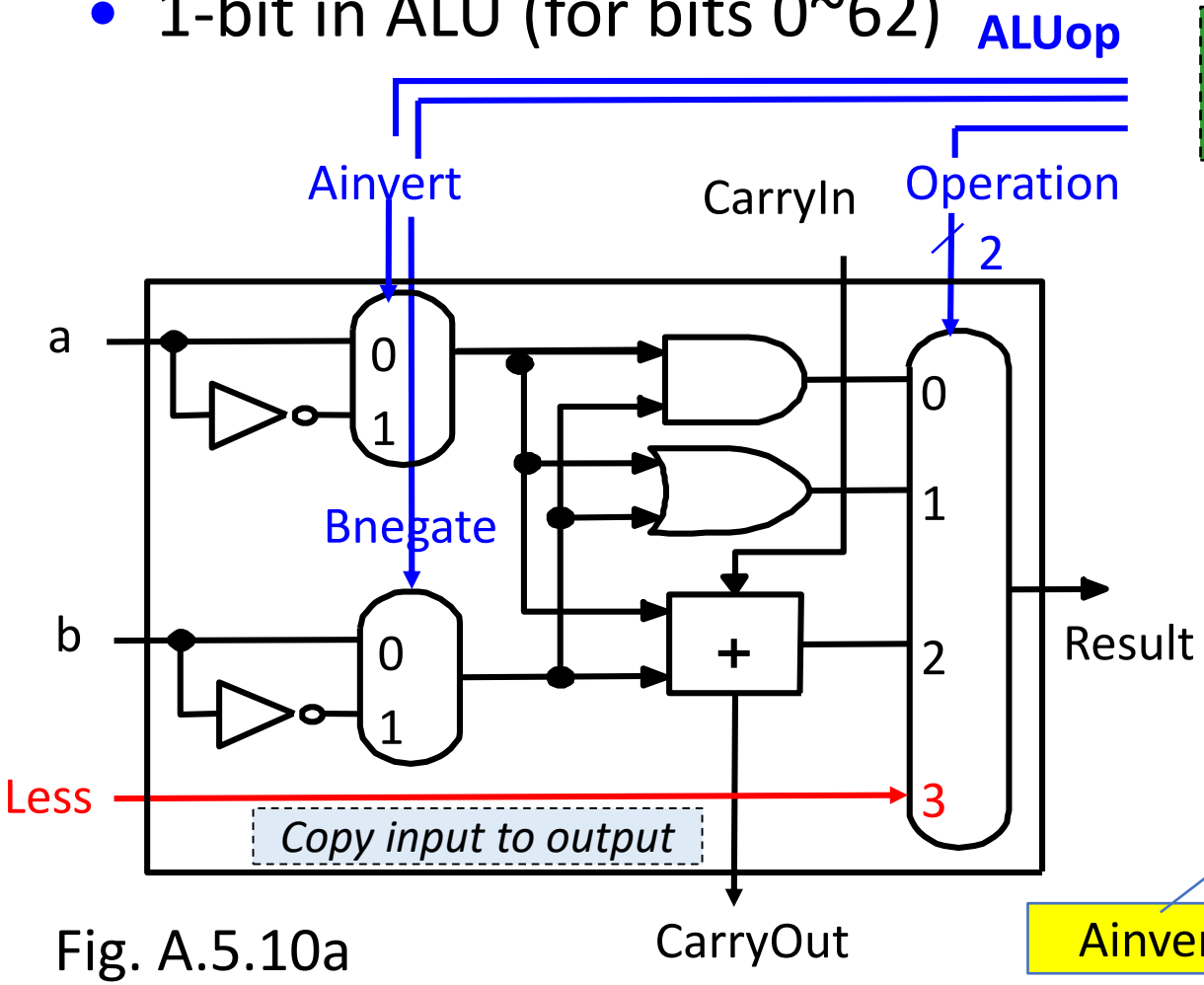


Fig. A.5.10a

Still perform a subtraction

But choose input as output

ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

operation

Bnegate

Ainvert

Set-Less-than (II)

- Sign bit in ALU (bit 63)

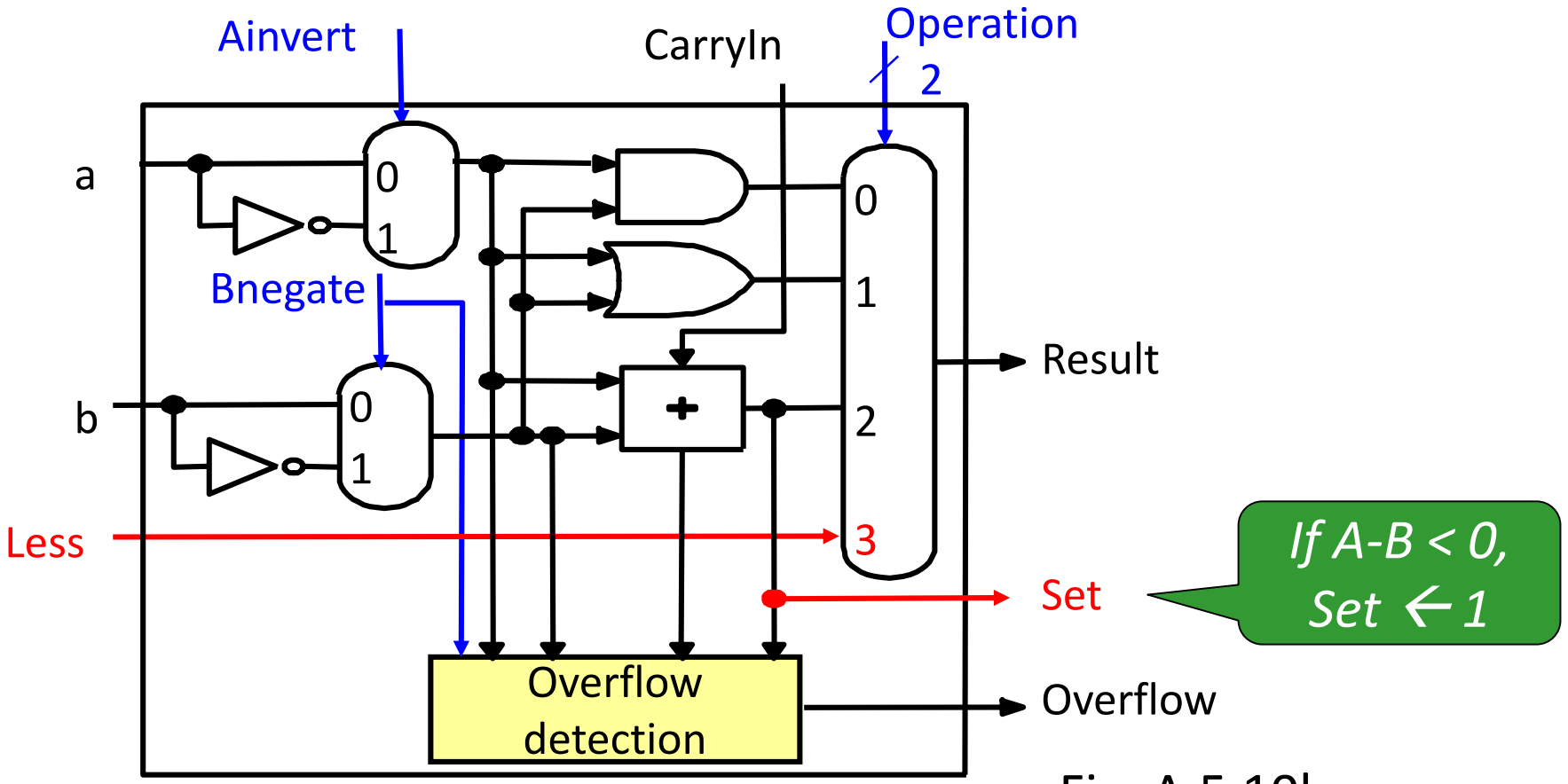
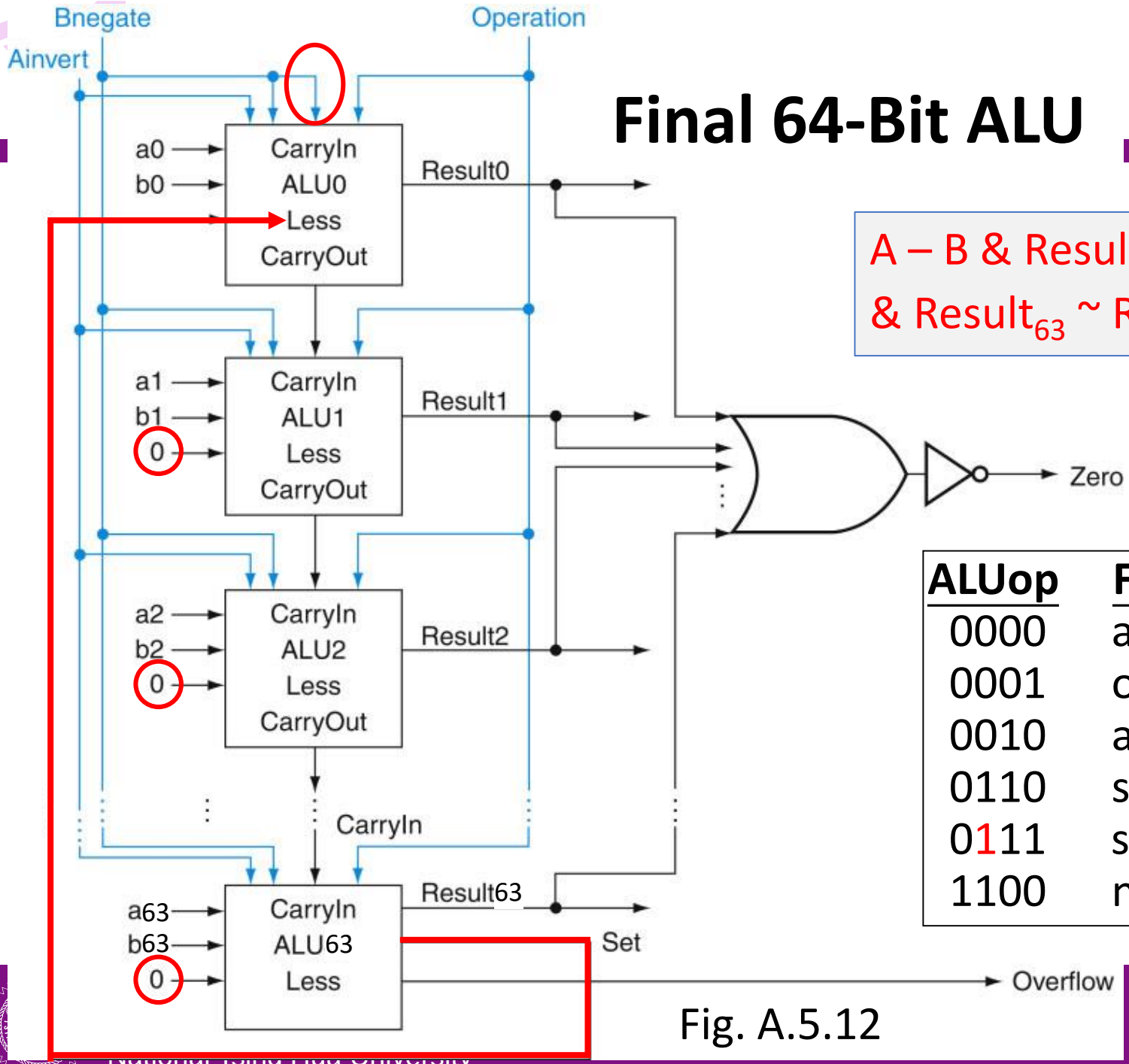


Fig. A.5.10b

Final 64-Bit ALU

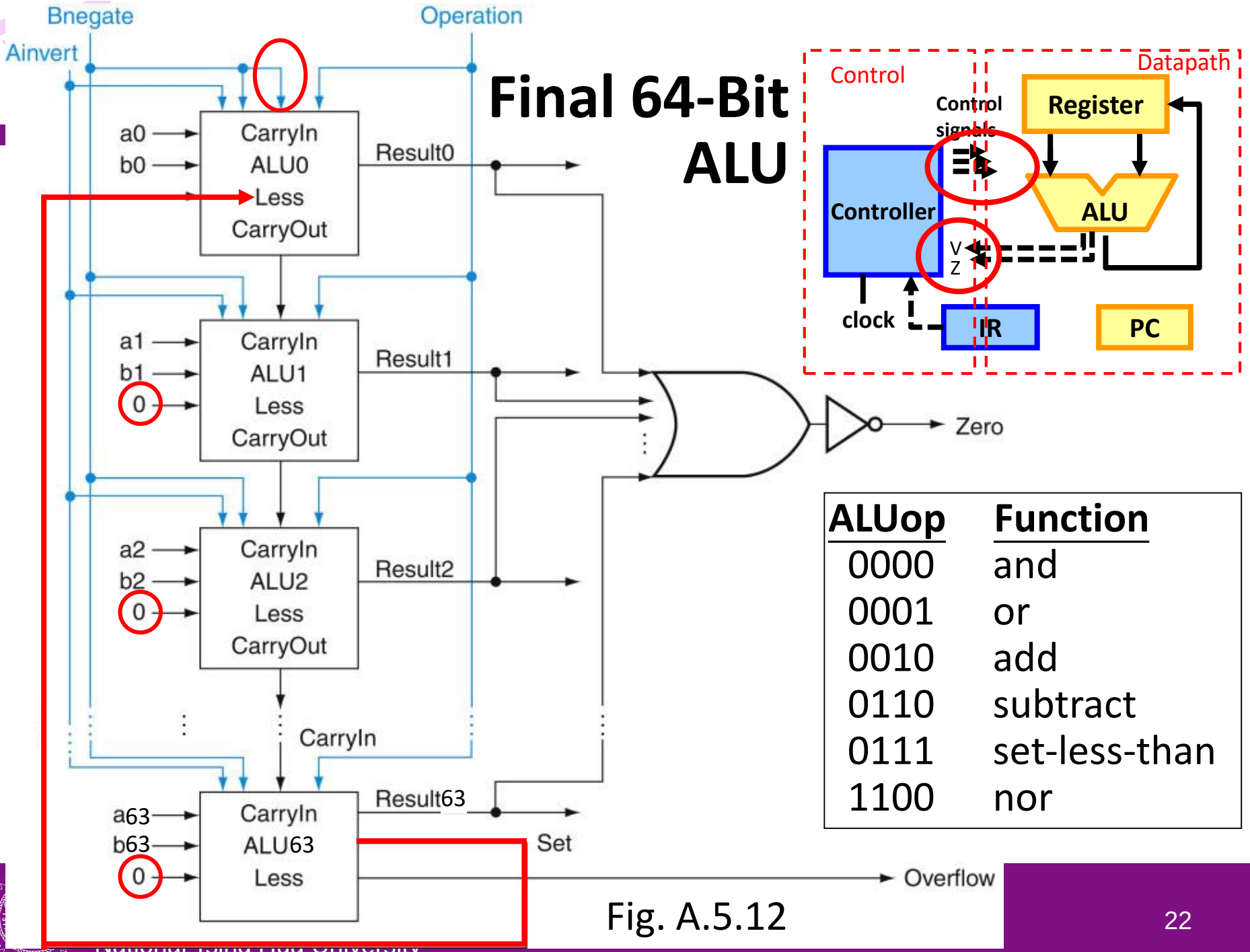


$A - B$ & $Result_0 = Result_{63}$
& $Result_{63} \sim Result_1 = 0$

ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

Fig. A.5.12

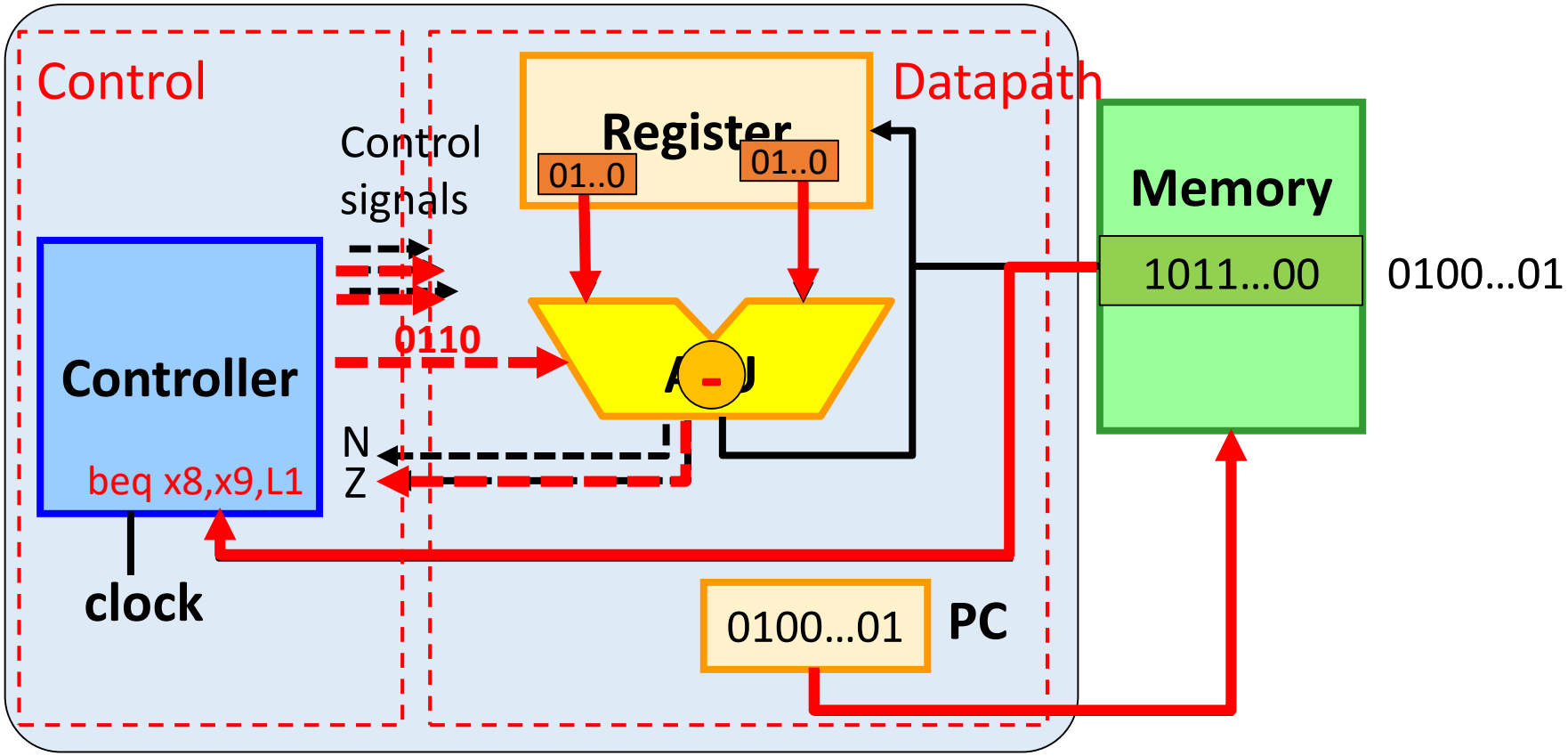
Final 64-Bit ALU



ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

Fig. A.5.12

Putting It Altogether



Verilog Behavioral Definition of 64-Bit ALU

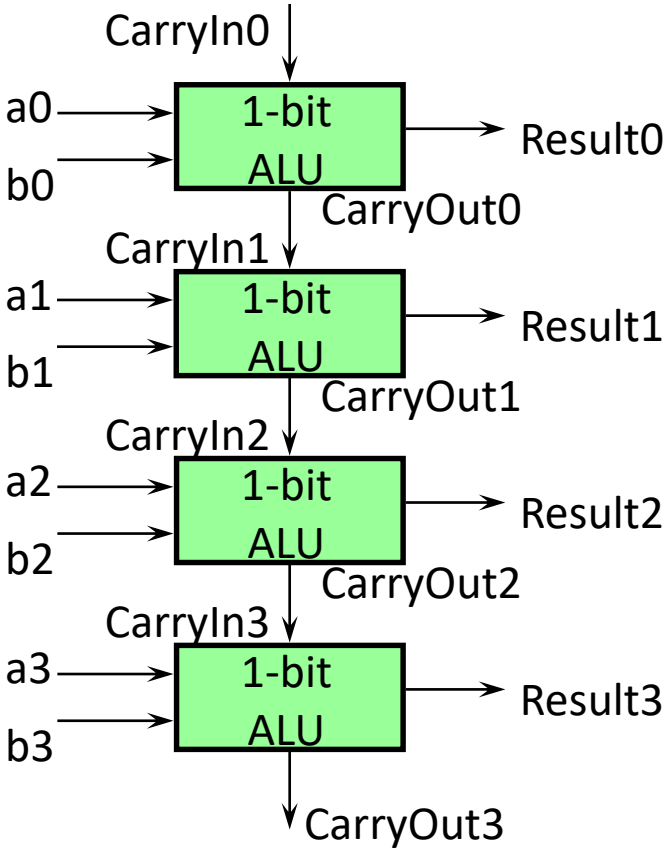
```
module RISCVALU(ALUctl, A, B, ALUOut, Zero);
  input [3:0] ALUctl;
  input [63:0] A,B;
  output reg [63:0] ALUOut;
  output Zero;
  assign Zero = (ALUOut==0); //Zero true if ALUOut 0
  always @(ALUctl, A, B) begin //reevaluate if change
    case (ALUctl)
      0: ALUOut <= A & B;
      1: ALUOut <= A | B;
      2: ALUOut <= A + B;
      6: ALUOut <= A - B;
      7: ALUOut (= A < B ? 1 : 0;
      12: ALUOut <= ~(A | B); // nor
      default: ALUOut <= 0;
    endcase
  end
endmodule
```

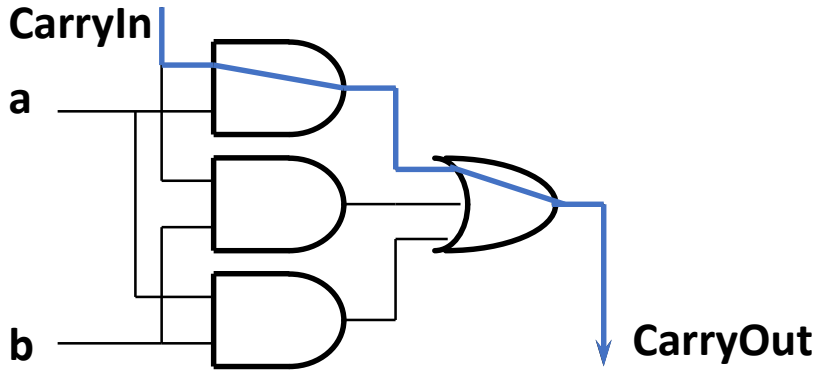
ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

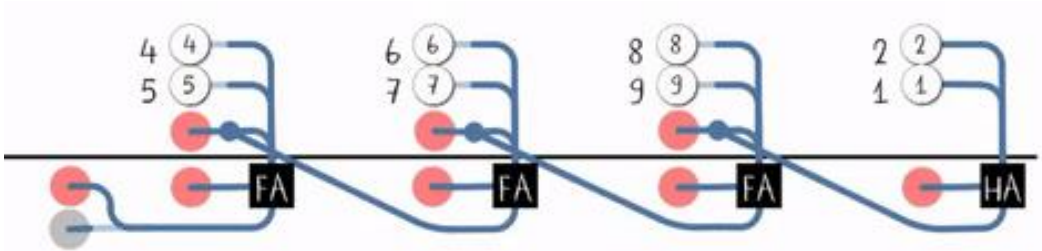
Fig. A.5.15

Problems with Ripple Carry Adder

- Carry bit may have to propagate from LSB to MSB
  → worst case delay: N-stage delay







By Marble machine - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48016289

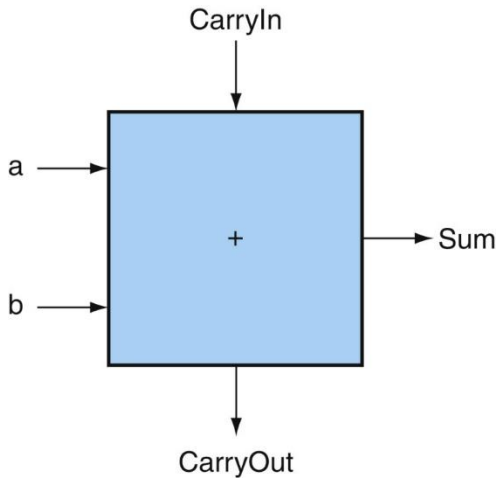
Improving Ripple Carry Adder

- Strategy: determine carry in to high-order bits sooner
- How?

  Examine how carry is generated and propagated
  - *Carry generation*: under what condition a carry is generated at a bit stage?
    - If a=1 and b=1, a carry is generated no matter what the value of carry input is

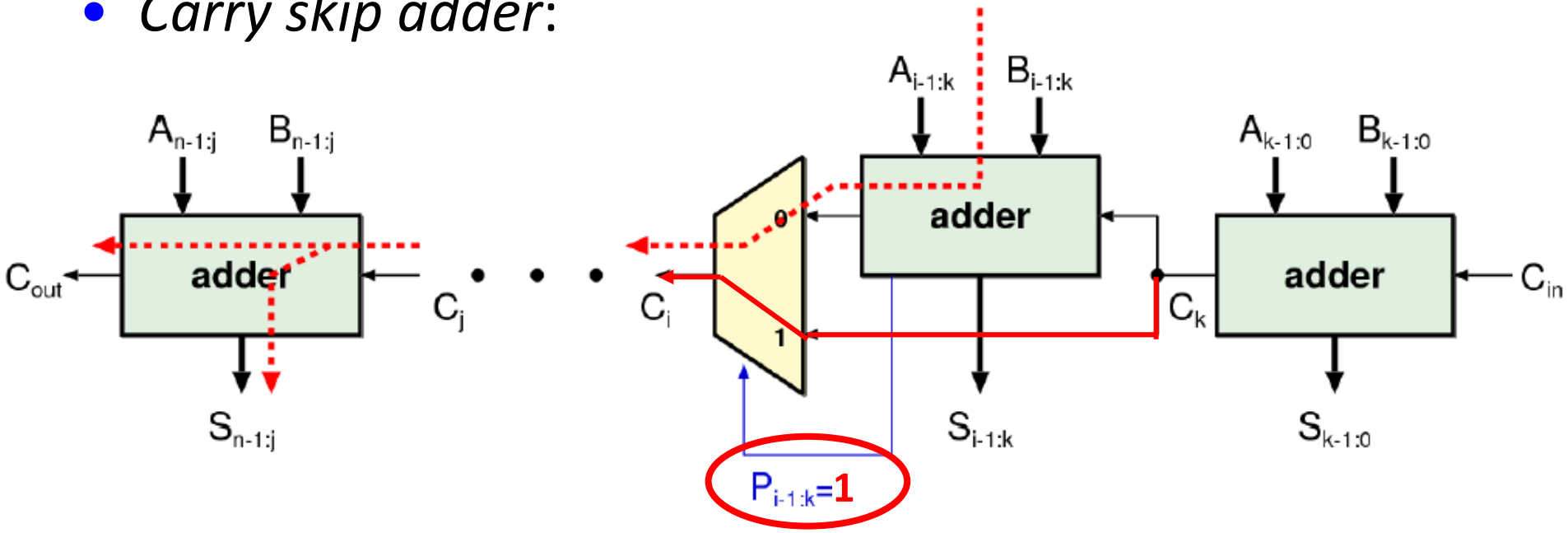
      Generate = g = a & b
  - *Carry propagation*: under what condition a carry can be propagated at a bit stage?
    - If only a=1 or only b=1, the value of carry input is propagated to the carry output

      Propagate = p = a xor b



Improving Ripple Carry Adder

- *Carry skip adder*:



  - $P_{i-1:k} = P_{i-1}$ & P_{i-2} & … & $P_k = 1 \rightarrow C_i = C_k$
  - $P_{i-1:k} = 0 \rightarrow$ this stage will not propagate carry → the result of the carry is generated within this stage

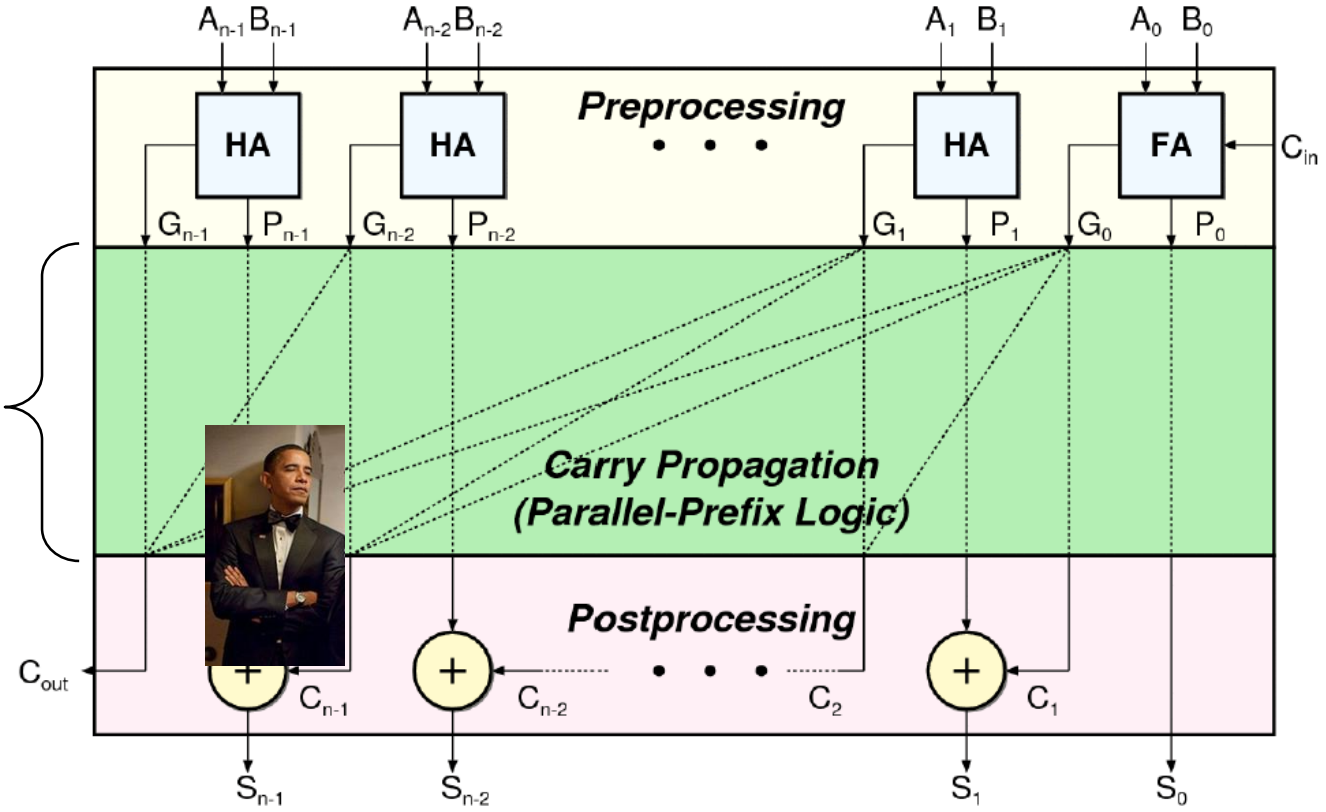
http://www.syssec.ethz.ch/education/Digitaltechnik_14

Improving Ripple Carry Adder

- *Carry lookahead adder*:
  - Look ahead and anticipate the carry to come (Sec. B.6)

$$C_{63}=G_{63}+P_{63}G_{62}+P_{63}P_{62}G_{61}+\ldots+P_{63}\ldots P_1G_0$$

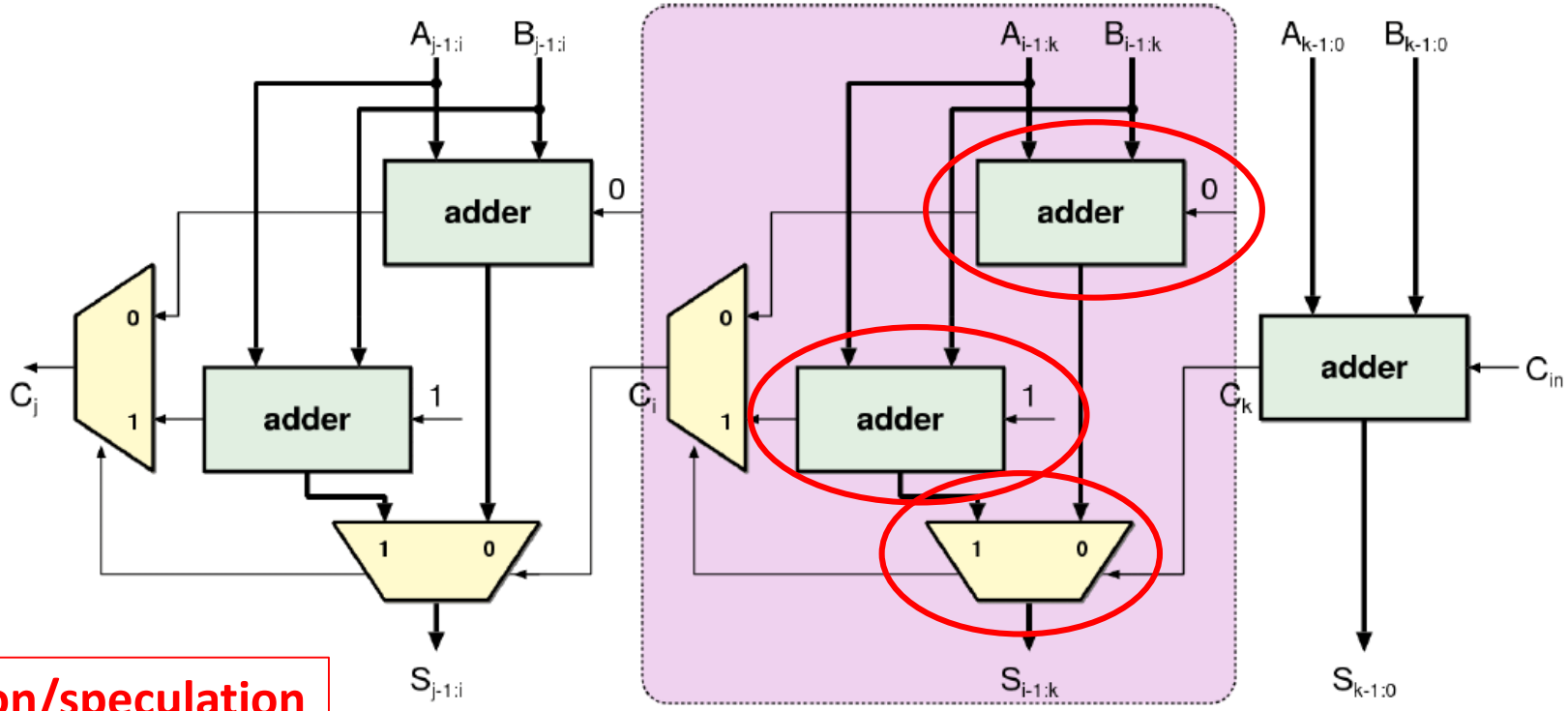
Fixed delay



http://www.syssec.ethz.ch/education/Digitaltechnik_14

Improving Ripple Carry Adder

- *Carry select adder*:
  - Design principle: parallelism (do both paths first and choose the desired result after carry in is known)



Prediction/speculation

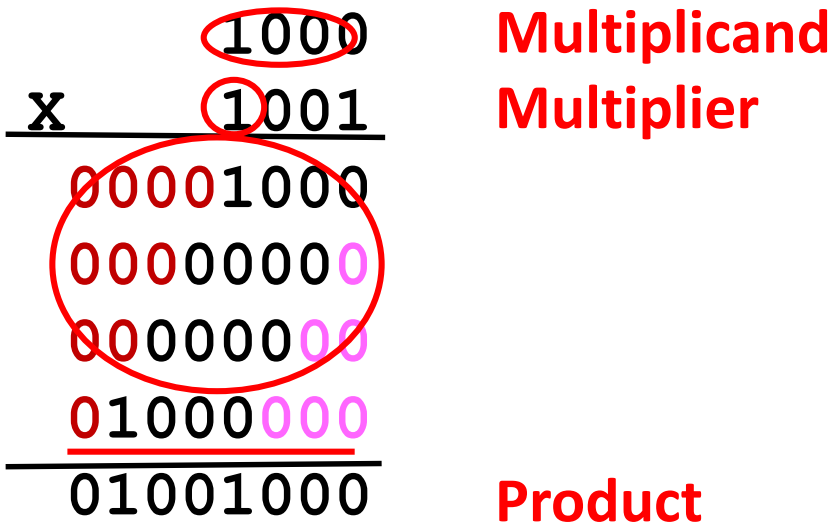
http://www.syssec.ethz.ch/education/Digitaltechnik_14

Outline

- Addition and subtraction (Sec. 3.2)
- Multiplication (Sec. 3.3)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)
- Parallelism and computer arithmetic: subword parallelism (Sec. 3.6)
- Streaming SIMD extensions and advanced vector extensions in x86 (Sec. 3.7)
- Subword parallelism and matrix multiply (Sec. 3.8)

Unsigned Multiply

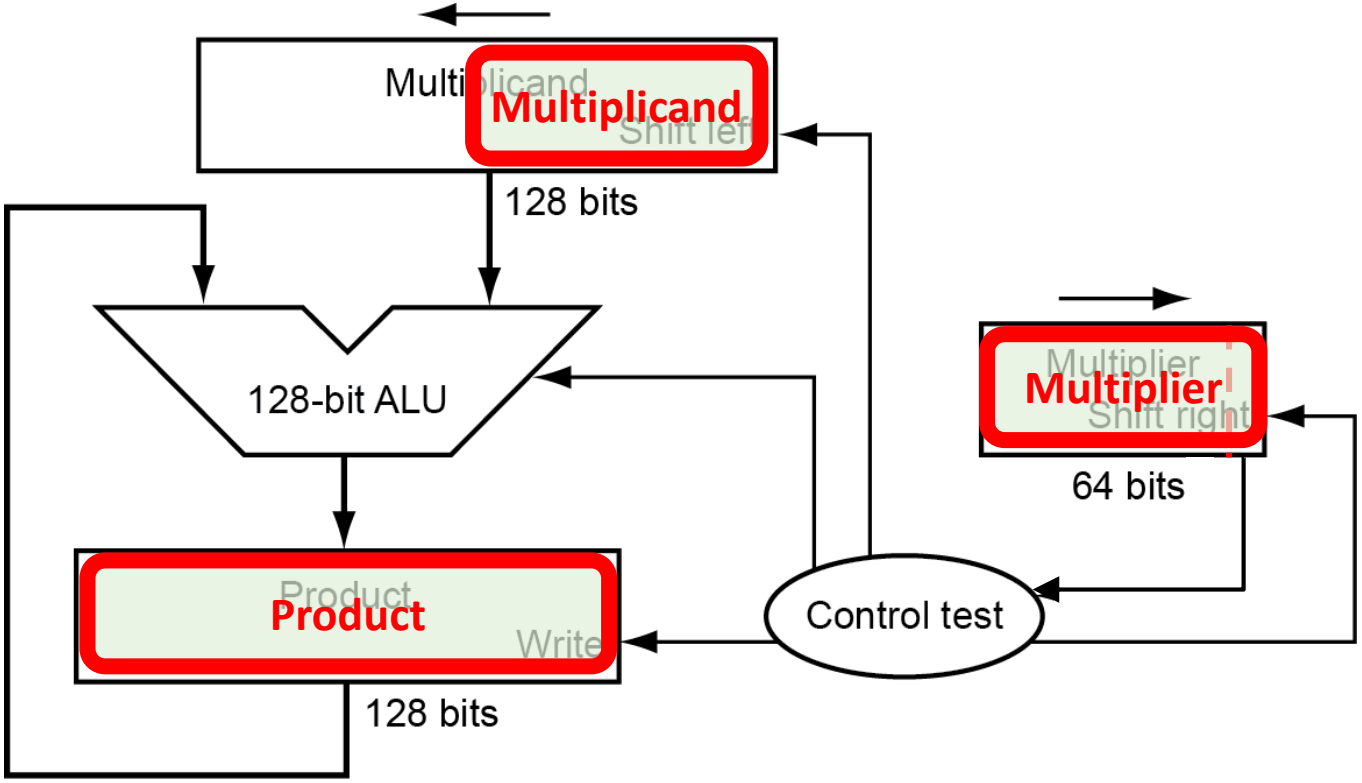
- Paper and pencil example (unsigned):



- Binary makes it easy:
  - 0 ➔ place 0 (0 x multiplicand)
  - 1 ➔ place a copy (1 x multiplicand)
- m bits x n bits = m+n bit product

Unsigned Multiplier for RISC-V (Ver. 1)

- 128-bit *multiplicand register* (with 64-bit multiplicand at right half), 128-bit ALU, 128-bit *product register*, 64-bit *multiplier register*




```
        1000
X       1001
  00001000
  00000000
  00000000
  01000000
  01001000
```

Fig. 3.3

Multiply Algorithm (Ver. 1)

0010 x 0011

Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110	0000	0001 0000
0000 0110	0000	0010 0000

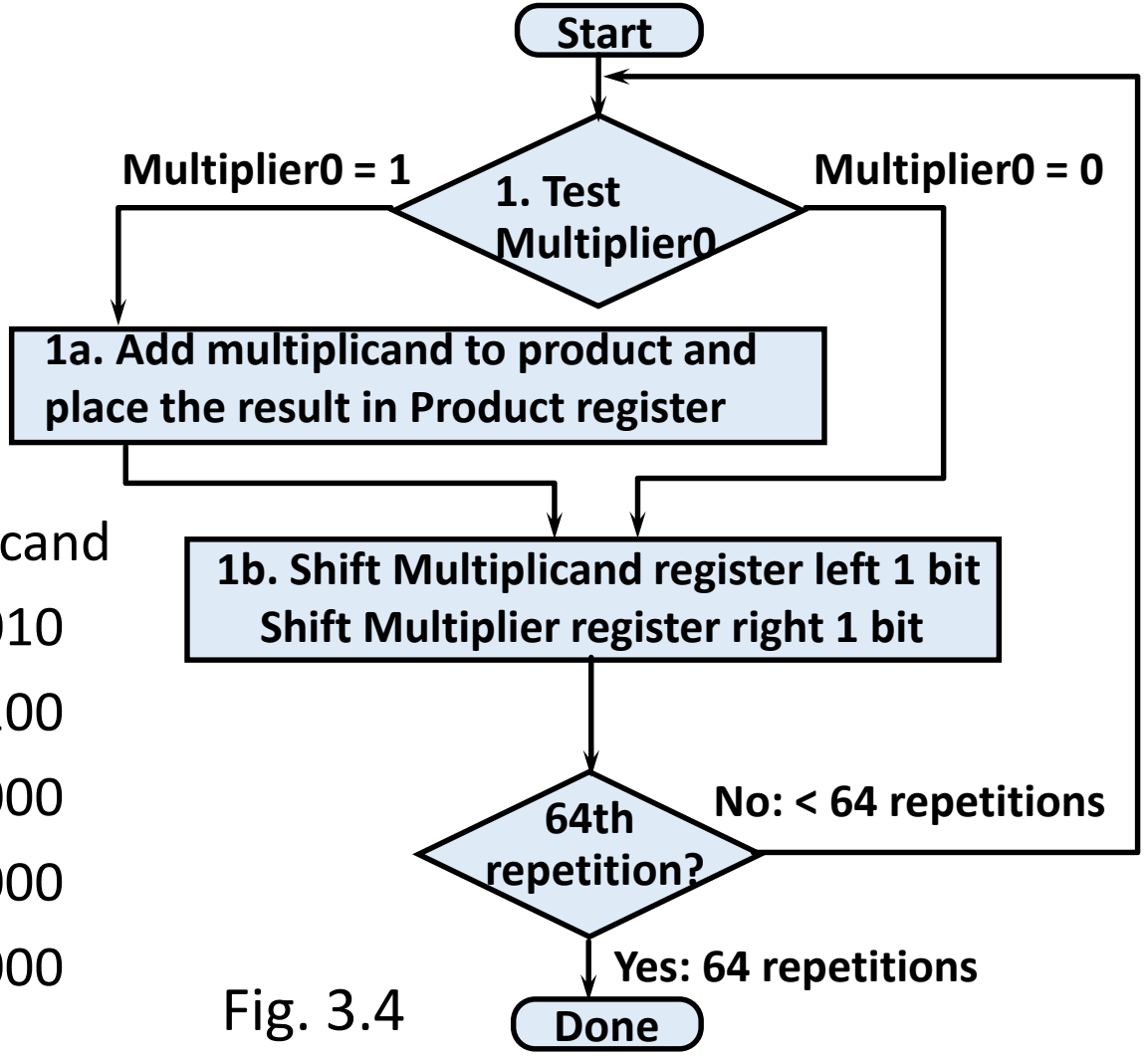


Fig. 3.4

Observations: Multiply Ver. 1

- 1 clock per iteration → ~64 clocks per multiply
  - Ratio of multiply to addition 5:1 to 100:1
- Half of the bits in multiplicand always 0
  → 128-bit adder is wasteful
- 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?
- Product register wastes space → combine Multiplier and Product register

Unsigned Multiplier for RISC-V (Ver. 2)

- 64-bit Multiplicand register, 64-bit ALU, 128-bit Product register (with 64-bit multiplier at right half), 0-bit Multiplier register

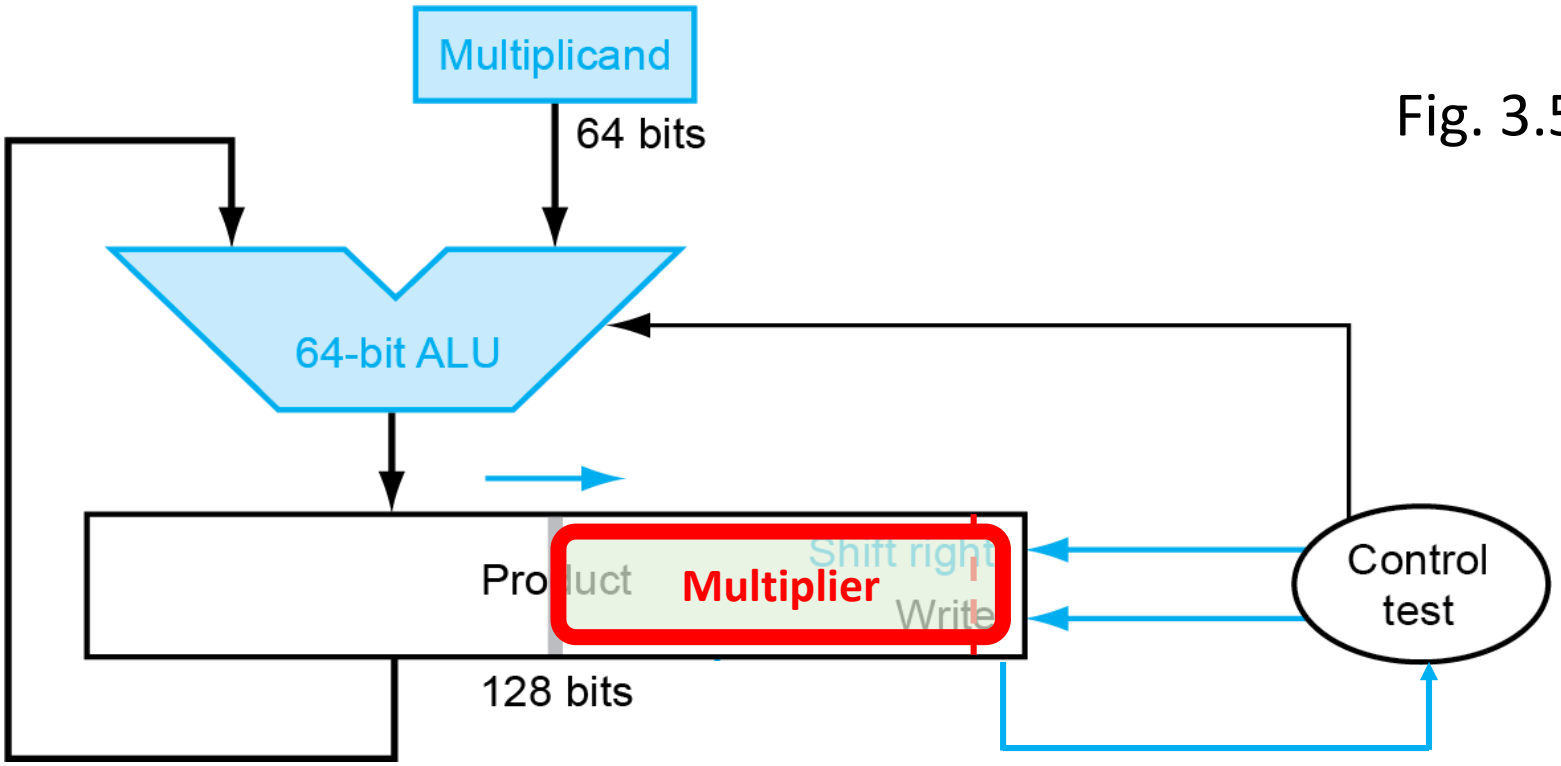
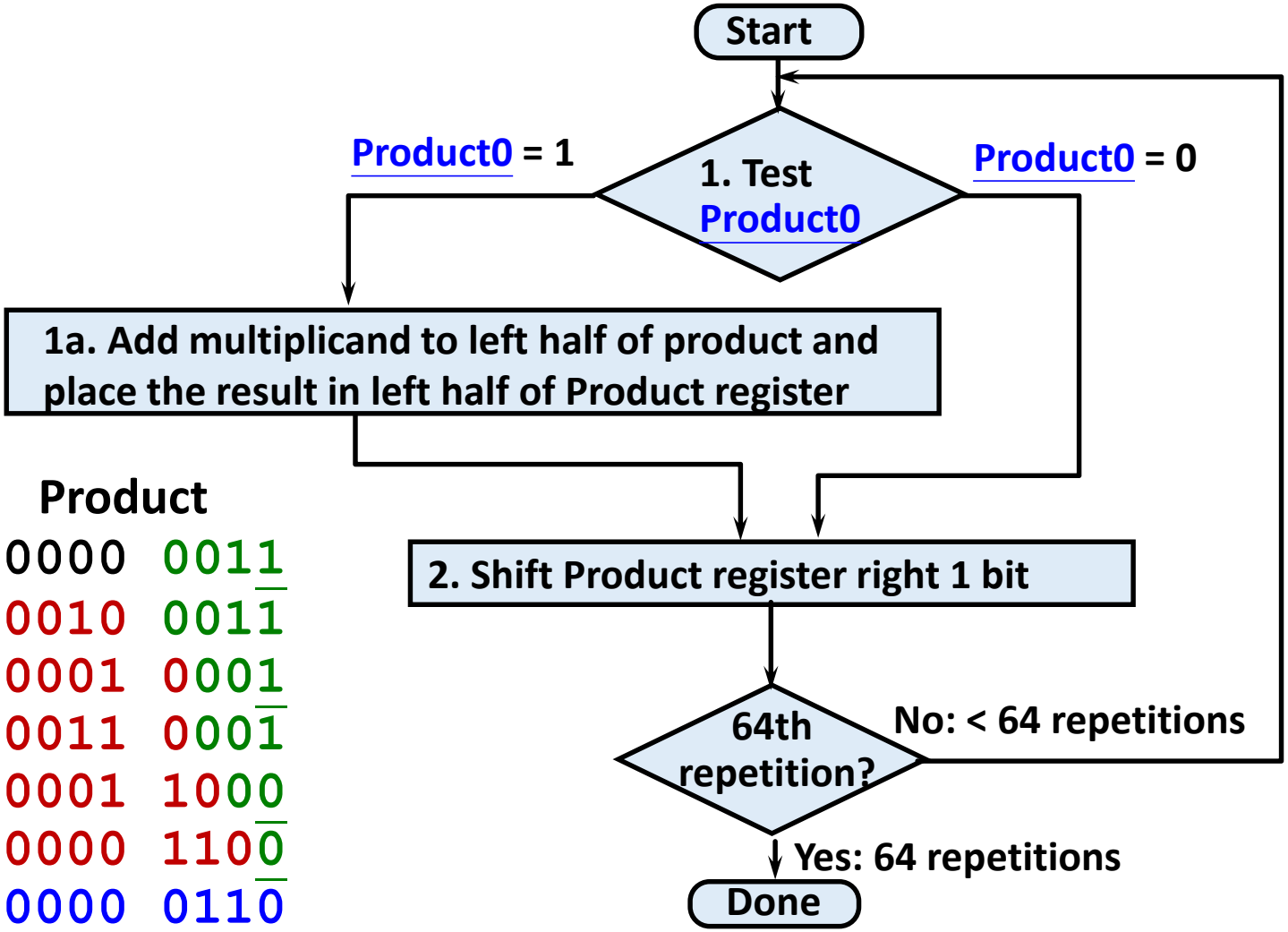


Fig. 3.5

Multiply Algorithm (Ver. 2)



Multiplicand	Product
0010	0000 0011
	0010 0011
0010	0001 0001
	0011 0001
0010	0001 1000
0010	0000 1100
0010	0000 0110

Observations: Multiply Ver. 2

- 2 steps in ver. 2, because Product register needs to be written twice (step 1a and step 2)
  - Steps 1a and 2 can be combined to complete in 1 cycle through hardware wiring

- What about signed multiplication?
  - The easiest solution is to make both positive and remember whether to complement product when done (leave out sign bit, run for 63 steps)
  - Apply definition of 2's complement
    - sign-extend partial products and subtract at end
  - Booth's algorithm: use same HW and save cycles

Faster Multiplier

1000 x 1111

- Use multiple adders: cost/performance tradeoff

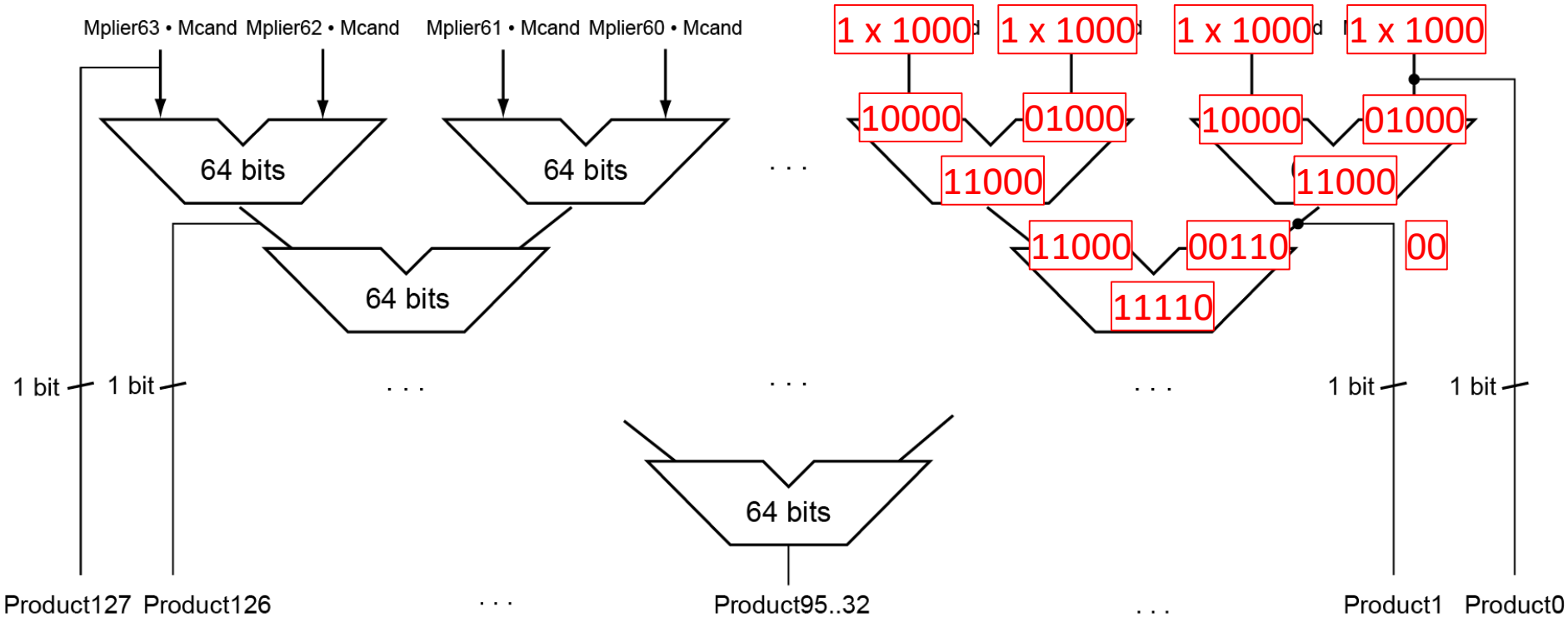


Fig. 3.7

- Can be pipelined: several multiplications in parallel

RISC-V Integer Multiplica

To get 128-bit product:

```
mulh[[s]u] rdh, rs1, rs2
mul        rdl, rs1,rs2
```

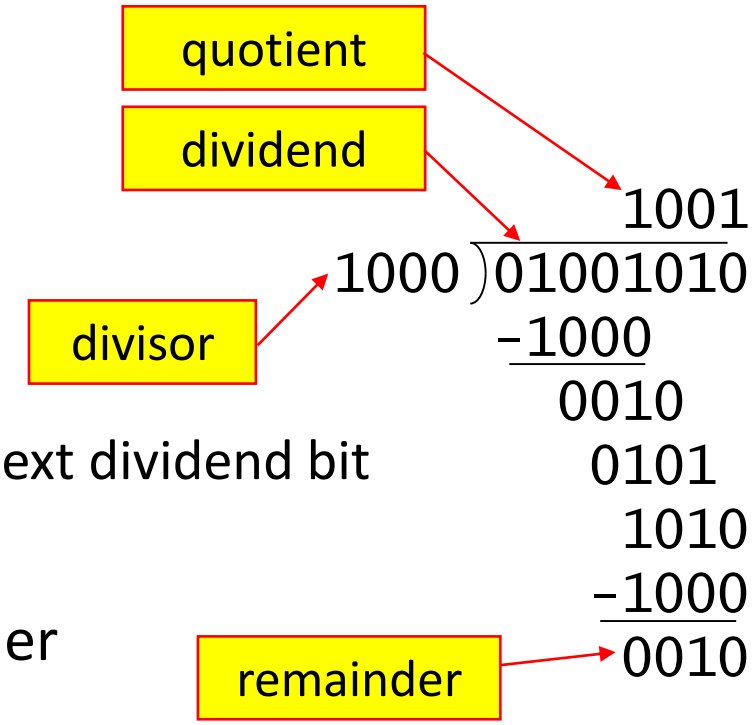
- **`mul`**: multiply
  - Gives lower 64 bits of the product in destination register
- **`mulh`**: multiply high
  - Gives upper 64 bits of the product in destination register, assuming signed × signed
- **`mulhu`**: multiply high unsigned
  - Gives upper 64 bits of the product in destination register, assuming unsigned × unsigned
- **`mulhsu`**: multiply high signed/unsigned
  - Gives upper 64 bits of the product in destination register, assuming signed × unsigned
- Use **`mulh`** result to check for 64-bit overflow

Outline

- Addition and subtraction (Sec. 3.2)
- Multiplication (Sec. 3.3)
- Division (Sec. 3.4)
- Floating point (Sec. 3.5)
- Parallelism and computer arithmetic: subword parallelism (Sec. 3.6)
- Streaming SIMD extensions and advanced vector extensions in x86 (Sec. 3.7)
- Subword parallelism and matrix multiply (Sec. 3.8)

Division

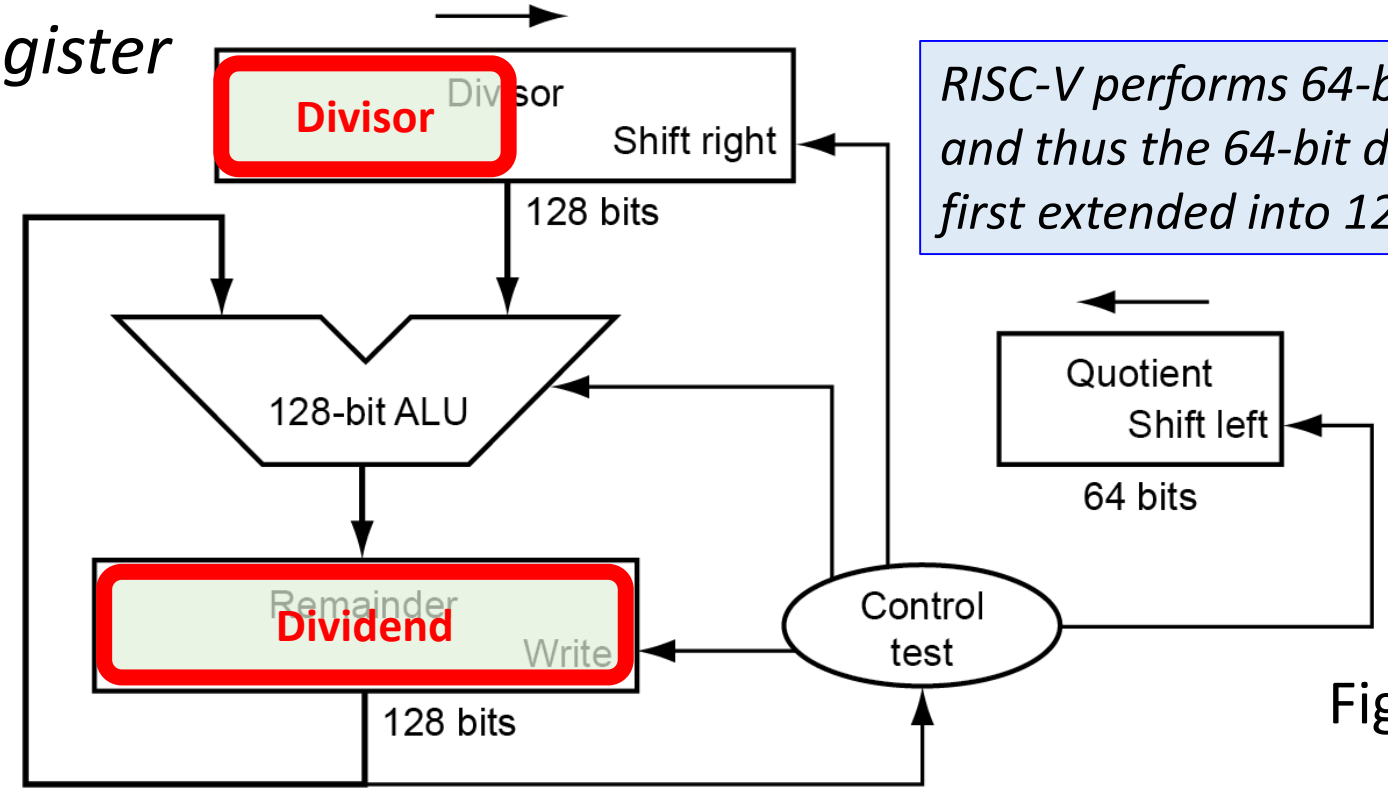
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division approach
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required



n-bit operands yield *n*-bit quotient and remainder

Divide Hardware for RISC-V (Version 1)

- 128-bit *Divisor register* (initialized with 64-bit divisor in left half), 128-bit ALU, 128-bit *Remainder register* (initialized with 128-bit dividend), 64-bit *Quotient register*



RISC-V performs 64-bit division, and thus the 64-bit dividend is first extended into 128 bits

Fig. 3.8

Divide Algorithm (Version 1)

Quot.	Divisor	Remainder
0000	00100000	00000111
		11100111
0000		00000111
0000	00010000	00000111
		11110111
0000		00000111
0000	00001000	00000111
		11111111
0000		00000111
0000	00000100	00000111
		00000011
0001		00000011
0001	00000010	00000011
		00000001
0011		00000001
0011	00000001	00000001

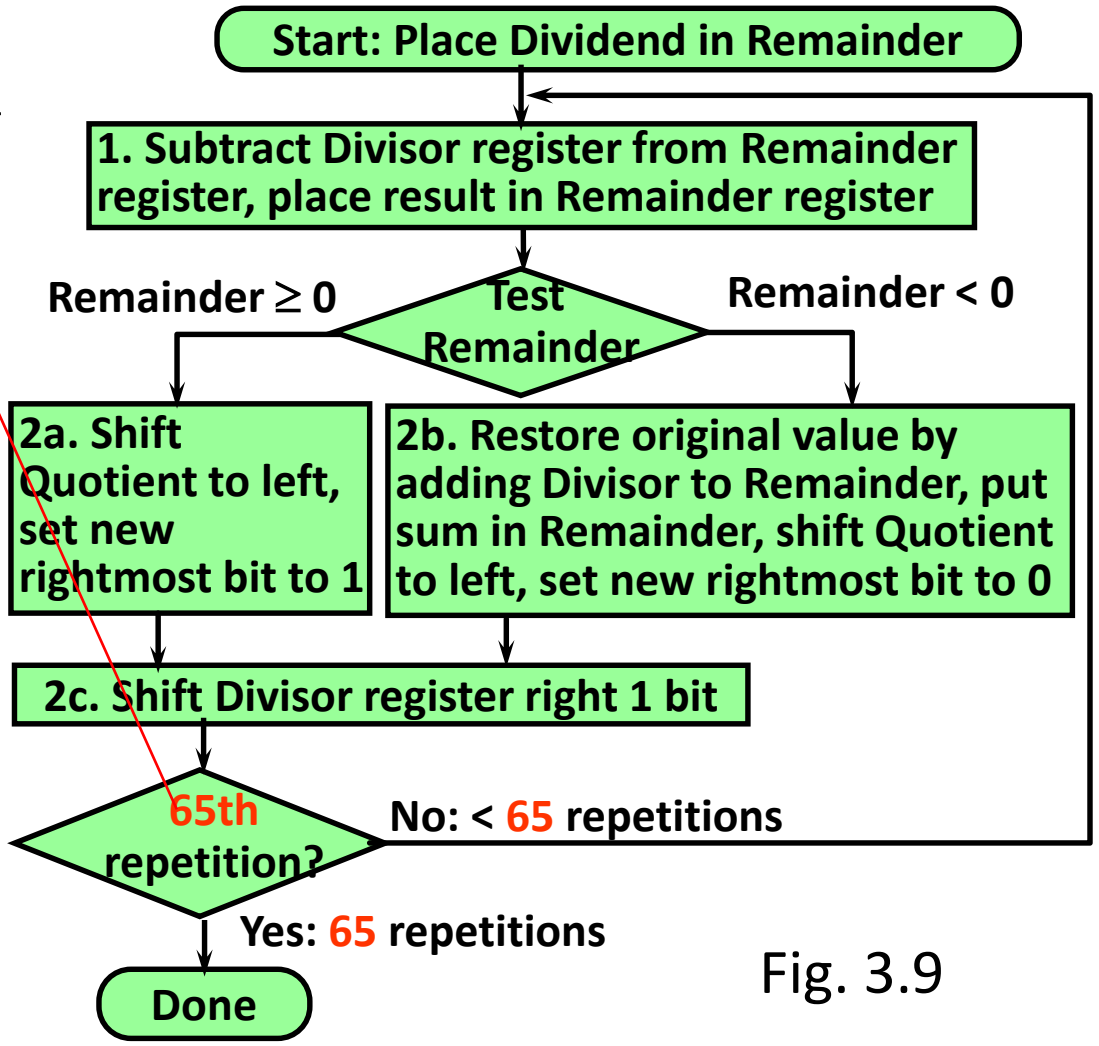


Fig. 3.9

Observations: Divide Version 1

- Half of the bits in divisor register always 0
  - → 1/2 of 128-bit adder is wasted
  - → 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- For 64-bit division, 1st step cannot produce a 1 in quotient bit (otherwise quotient is too big for the register)
  - → switch order to shift first and then subtract
  - → save 1 iteration
- Eliminate Quotient register by combining with Remainder register as shifted left

```
      11001
1000 )11001010
     -1000
      1001
     -1000
       0010
        0101
         1010
        -1000
           10
```

Remainder

0000	0111

Divide Hardware (Version 2)

- 64-bit Divisor register, 64-bit ALU, 128-bit Remainder register (initialized with 128-bit dividend), 0-bit Quotient register

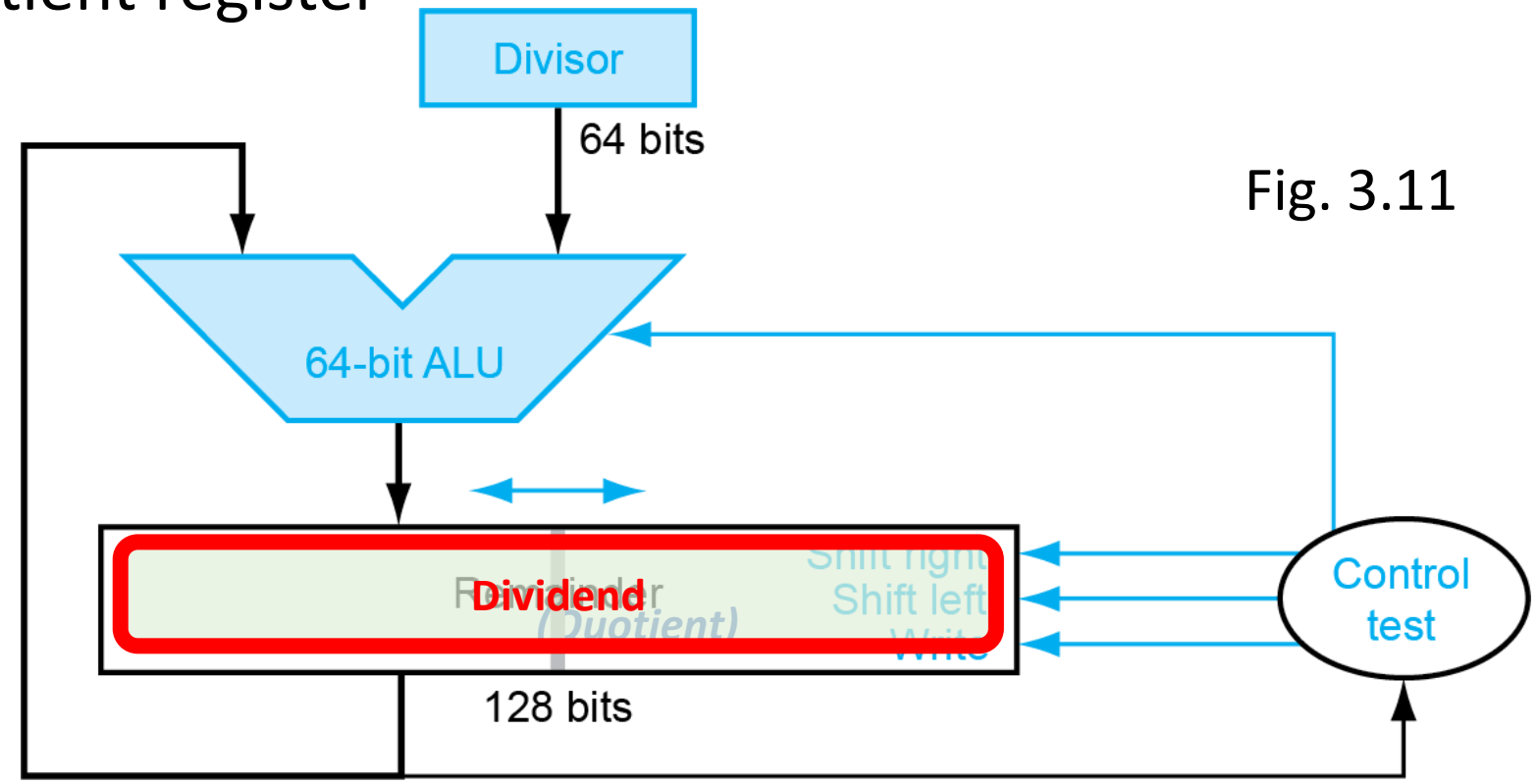
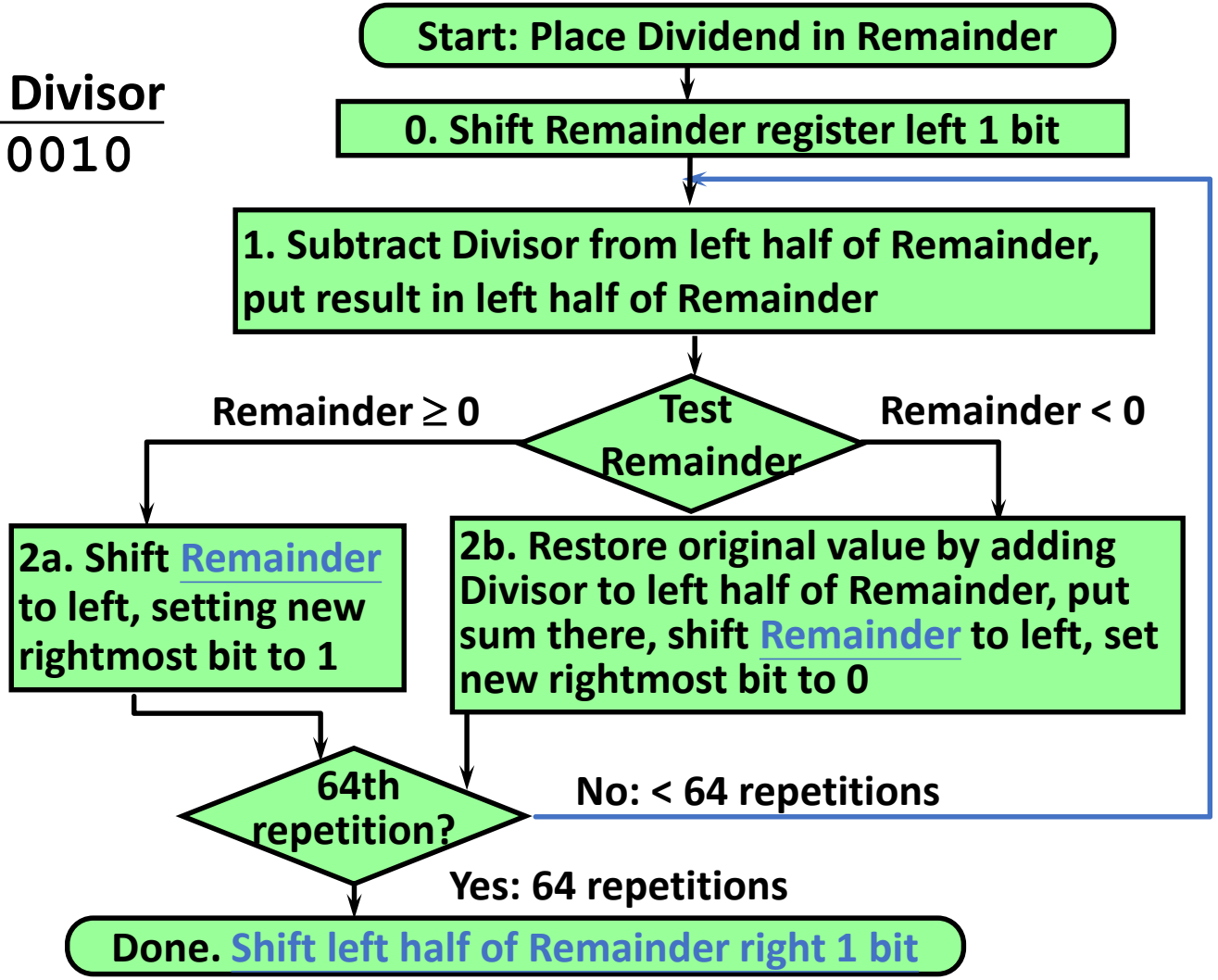


Fig. 3.11

Divide Algorithm (Version 2)

Step	Remainder	Divisor
	0000 0111	0010
0	0000 1110	
1.1	1110 1110	
1.2b	0001 1100	
2.1	1111 1100	
2.2b	0011 1000	
3.1	0001 1000	
3.2a	0011 0001	
4.1	0001 0001	
4.2a	0010 0011	
	0001 0011	



Signed Divide

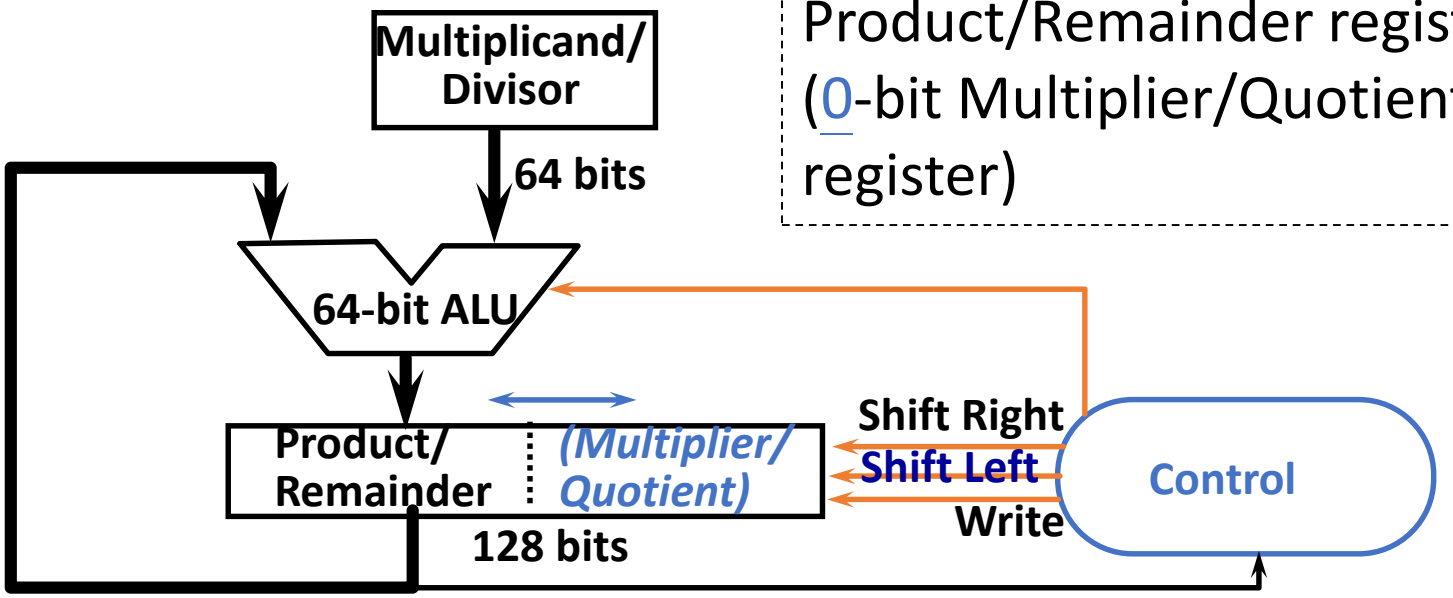
- Remember signs, make positive, complement quotient and remainder if necessary
- Alternative:
  Let Dividend and Remainder have same sign, negate Quotient if Divisor sign and Dividend sign disagree
  - e.g., -7 ÷ 2 = -3, remainder = -1
    -7 ÷ -2 = 3, remainder = -1
  - Satisfy Dividend = Quotient x Divisor + Remainder

- Possible for quotient to be too large:
  - If divide 64-bit integer by 1, quotient is 64 bits

Observation: Multiply and Divide

- Same hardware as multiply: just need ALU to add or subtract, and 128-bit register to shift left or shift right

64-bit Multiplicand/Divisor register, 64-bit ALU, 128-bit Product/Remainder register (0-bit Multiplier/Quotient register)



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

RISC-V Integer Division in M Extension

- **`div/divu`**: signed/unsigned division
  - Gives 64-bit signed/unsigned quotient in destination register
- **`rem/remu`**: signed/unsigned remainder
  - Gives 64-bit signed/unsigned remainder of the division
- If both quotient and remainder are required:

  `div[u]` rdq, rs1, rs2

  `rem[u]` rdr, rs1, rs2
  - Microarchitectures can fuse them into a single division
- Overflow and division-by-zero don't produce errors
  - Just return defined results
  - Faster for the common case of no error