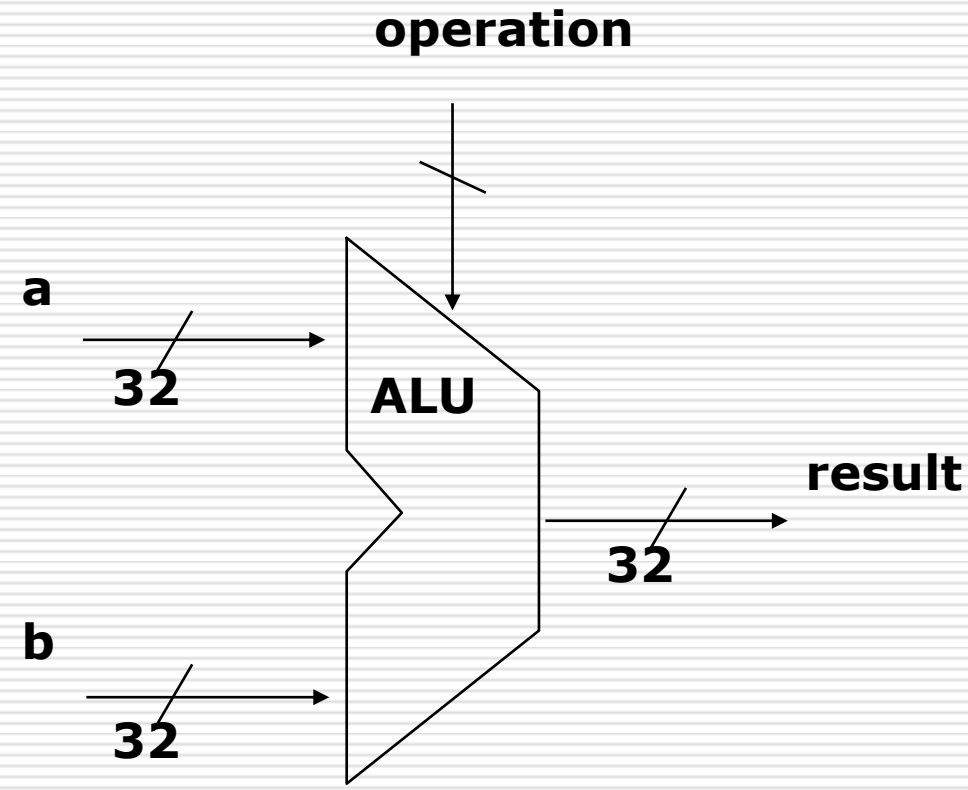
Computer Organization and Structure

Bing-Yu Chen
National Taiwan University

Arithmetic for Computers

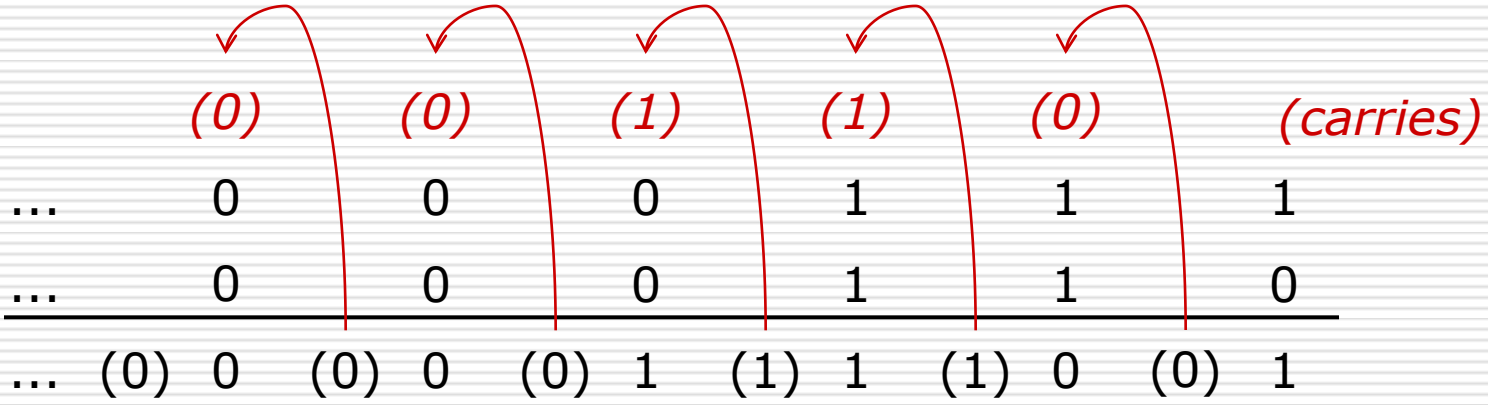
- Addition and Subtraction
- Gate Logic and K-Map Method
- Constructing a Basic ALU
  - Arithmetic Logic Unit
- Multiplication and Division
- Floating Point

Arithmetic



Integer Addition & Subtraction

- Example: 7 + 6 *(just like in grade school)*



- 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

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

- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

Detecting Overflow

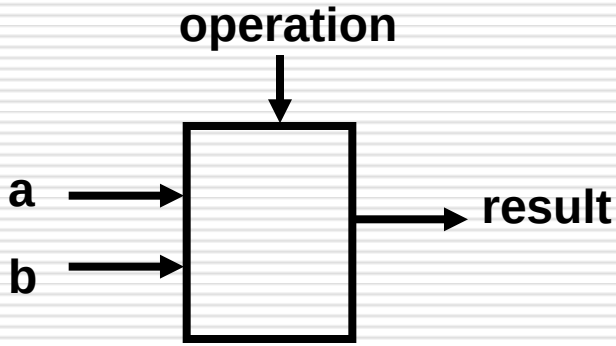
Operation	Operand A	Operand B	Result indicating overflow
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

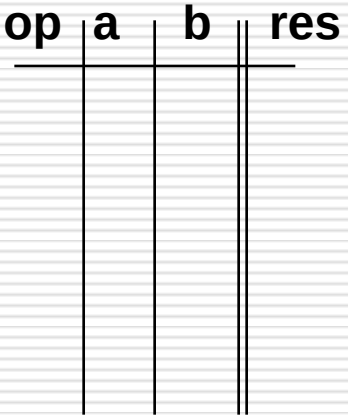
Dealing with Overflow

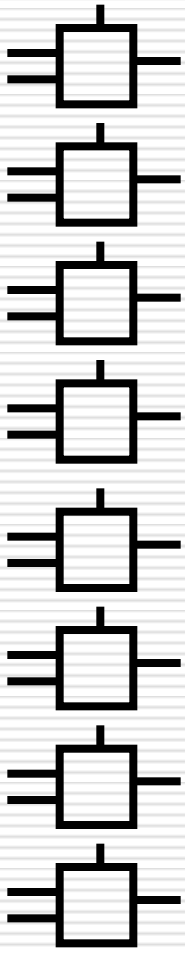
- An exception (interrupt) occurs
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
- Some languages (e.g., C) ignore overflow
  - new MIPS instructions: `addu`, `addiu`, `sub`
    - *note*: `addiu` *still sign-extends*!
    - *note*: `sltu`, `sltiu` *for unsigned comparisons*

An ALU (Arithmetic Logic Unit)

- build an ALU to support `andi` & `ori` instructions
  - just build a 1 bit ALU, and use 32 of them

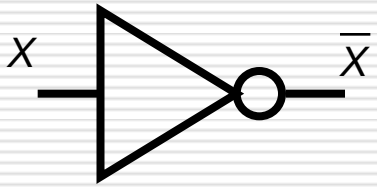
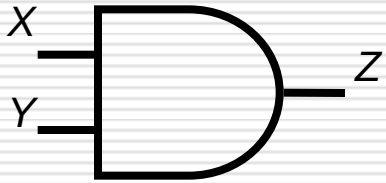
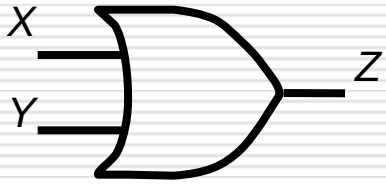






- Possible Implementation (sum-of-products):

Review: The NOT, AND, OR Gates

	Description	*Gates*	*Truth Table*
NOT	If $X = 0$ then $X' = 1$ If $X = 1$ then $X' = 0$		X | $\overline{X}$ 0 | 1 1 | 0
AND	$Z = 1$ if X and Y are both 1		X Y | Z 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	$Z = 1$ if X or Y (or both) are 1		X Y | Z 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1

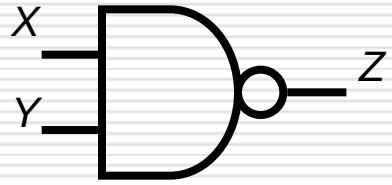
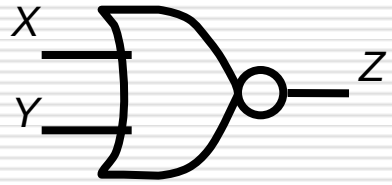
Review: NAND, NOR, XOR, XNOR

- 16 functions of two variables:

X	Y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		0	$X \bullet Y$		X		Y		$X+Y$			$\overline{Y}$		$\overline{X}$			1

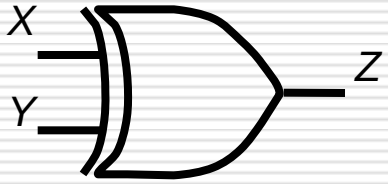
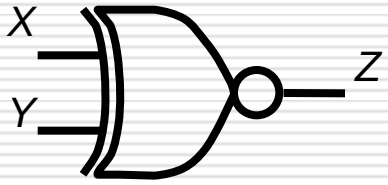
- X, X', Y, Y', X•Y, X+Y, 0, 1 only half of the possible functions

Review: NAND, NOR

	Description	*Gates*	*Truth Table*
NAND	*Z* = 1 if *X* is 0 or *Y* is 0		*X* *Y* *Z* 0 0 1 0 1 1 1 0 1 1 1 0
NOR	*Z* = 1 if both *X* and *Y* are 0		*X* *Y* *Z* 0 0 1 0 1 0 1 0 0 1 1 0

Review: XOR, XNOR

- XOR: X or Y but not both ("inequality", "difference")
  - $X \oplus Y = X\overline{Y} + \overline{X}Y$
- XNOR: X and Y are the same ("equality", "coincidence")
  - $\overline{X \oplus Y} = \overline{X}\,\overline{Y} + XY$

	Description	*Gates*	*Truth Table*
XOR	*Z* = 1 if *X* has a different value than *Y*		*X* *Y* *Z* 0 0 0 0 1 1 1 0 1 1 1 0
XNOR	*Z* = 1 if *X* has the same value as *Y*		*X* *Y* *Z* 0 0 1 0 1 0 1 0 0 1 1 1

Review: Truth Tables

- Tabulate all possible input combinations and their associated output values

Example: half adder
adds two binary digits
to form Sum and Carry

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

NOTE: 1 plus 1 is 0 with a
carry of 1 in binary

Example: full adder
adds two binary digits and
Carry in to form Sum and
Carry Out

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

Deriving Boolean Equations from Truth Tables *for Half Adder*

- OR'd together *product* terms for each truth table row where the function is 1
- if input variable is 0, it appears in complemented form;
- if 1, it appears uncomplemented

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$\text{Sum} = \overline{A}\,B + A\,\overline{B}$

$\text{Carry} = A\,B$

Example: Full Adder

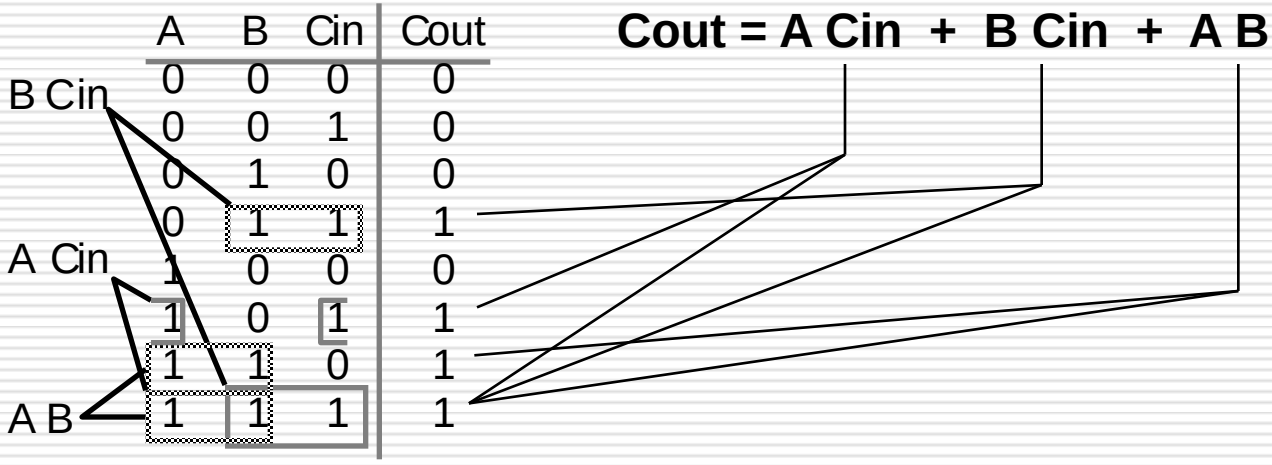
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$Sum = \overline{A}\,\overline{B}\,Cin + \overline{A}\,B\,\overline{Cin} + A\,\overline{B}\,\overline{Cin} + A\,B\,Cin$$

$$Cout = \overline{A}\,B\,Cin + A\,\overline{B}\,Cin + A\,B\,\overline{Cin} + A\,B\,Cin$$

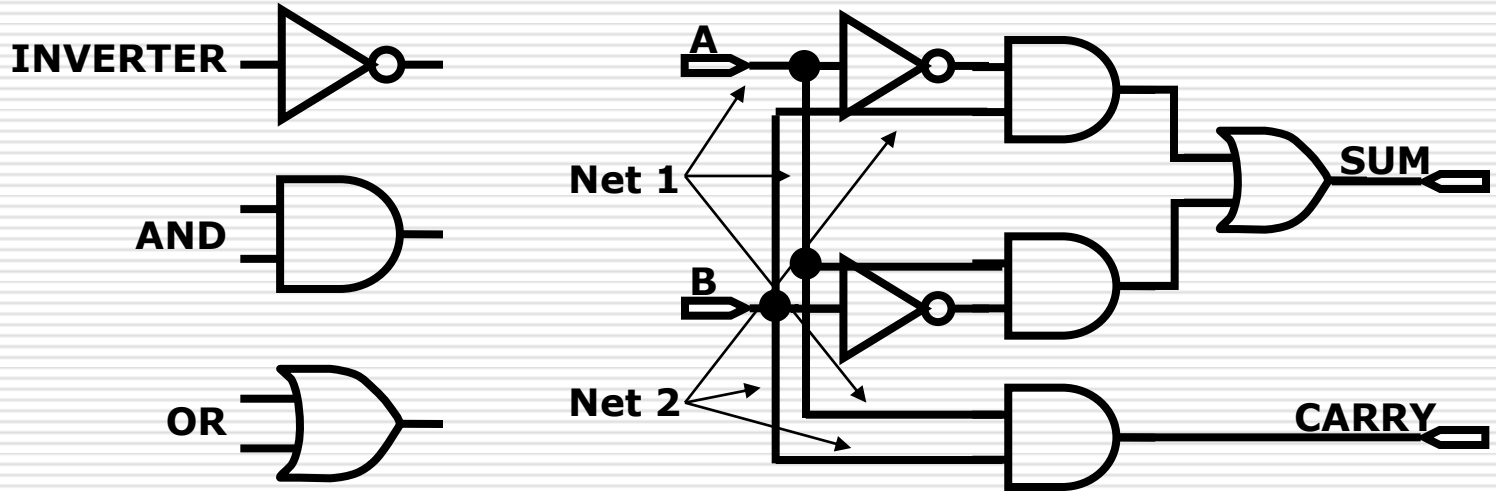
Reducing the Complexity of Boolean Equations

- each product term in the above equation covers exactly two rows in the truth table; several rows are "covered" by more than one term



Review: Gates & Net

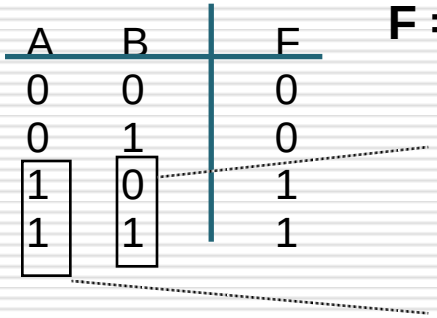
- most widely used primitive building block in digital system design
- Standard Logic Gate Representation



- *Net*: electrically connected collection of wires

Two-Level Simplification

- *Key Tool*: The Uniting Theorem —
  - A (B' + B) = A



F = A B' + A B = A (B' + B) = A

B's values change within the on-set rows

B is eliminated, A remains

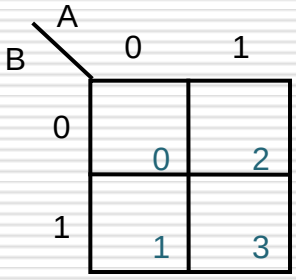
A's values don't change within the on-set rows

- *Essence of Simplification*:
  - find two element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated !*

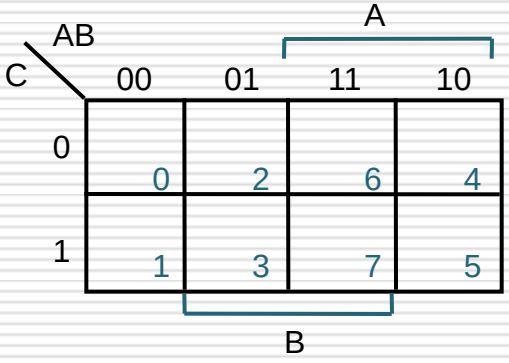
Karnaugh Map Method

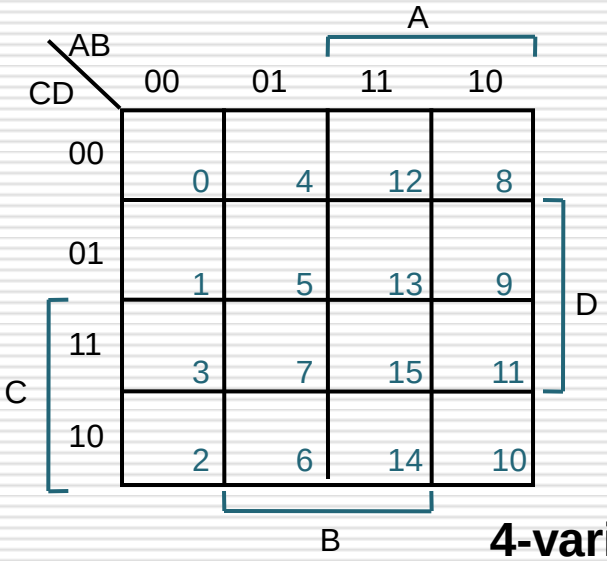
- K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions
- Beyond that, computer-based methods are needed

2-variable K-map



3-variable K-map



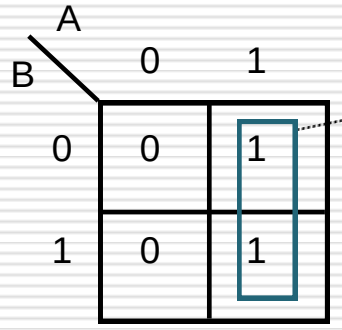


4-variable K-map

Karnaugh Map Method

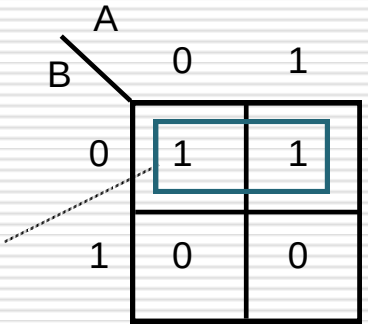
- *Numbering Scheme*: 00, 01, 11, 10
  - Gray Code — only a single bit changes from code word to next code word

K-Map Method Examples



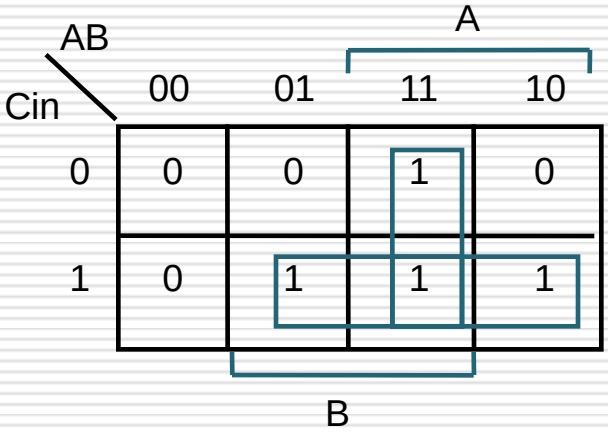
A asserted, unchanged
B varies

B complemented, unchanged
A varies

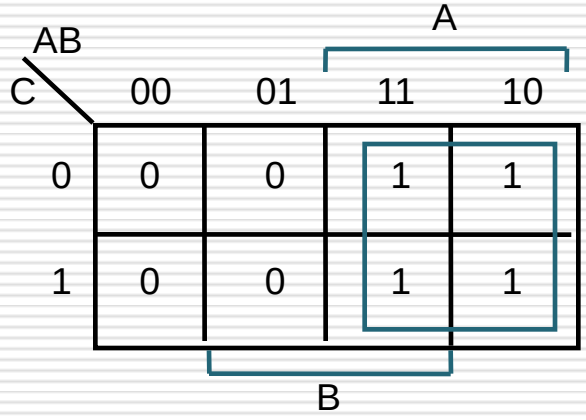


F = A

G = B'

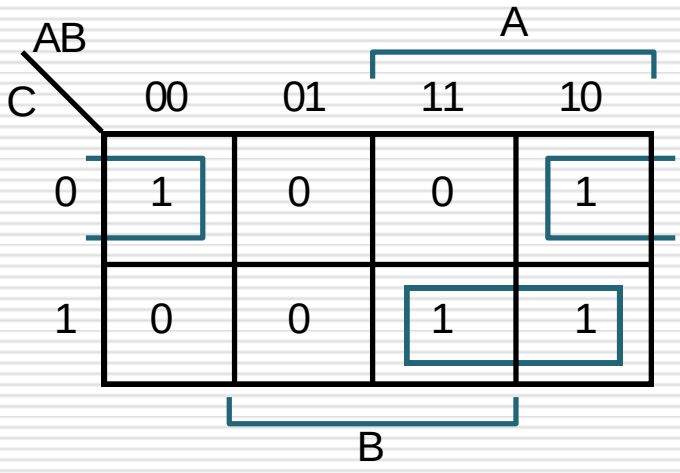


Cout = A B + B Cin + A Cin



F(A,B,C) = A

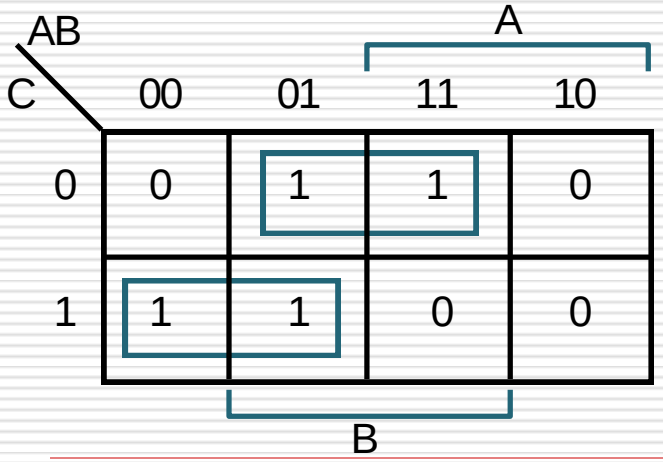
K-Map Method Examples, 3 Variables



F(A,B,C) = Σm(0,4,5,7)

F = B' C' + A C

In the K-map, adjacency wraps from left to right and from top to bottom



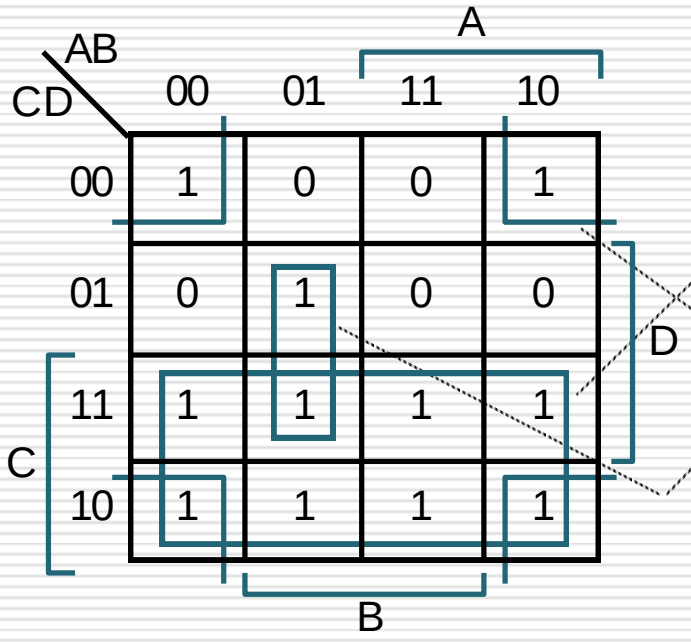
F' simply replace 1's with 0's and vice versa

F'(A,B,C) = Σm(1,2,3,6)

F' = B C' + A' C

compare with the method of using DeMorgan's Theorem and Boolean Algebra to reduce the complement!

K-map Method Examples: 4 variables



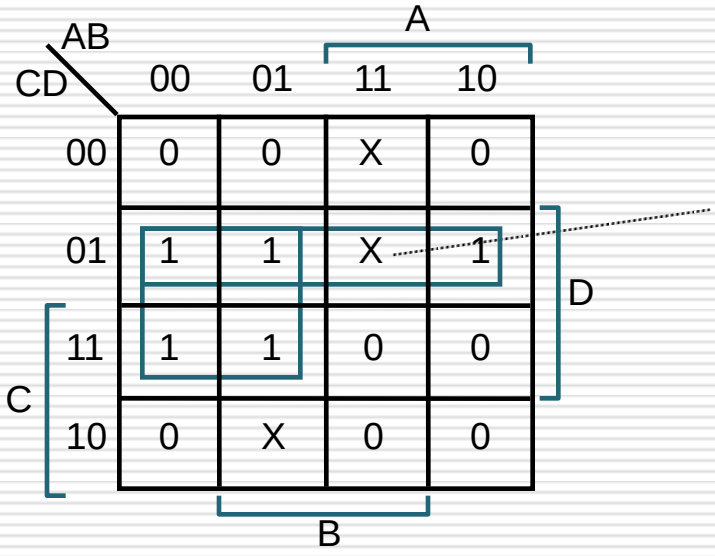
F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)

F = C + A' B D + B' D'

find the smallest number of the largest possible subcubes that cover the ON-set

K-map Example: Don't Cares

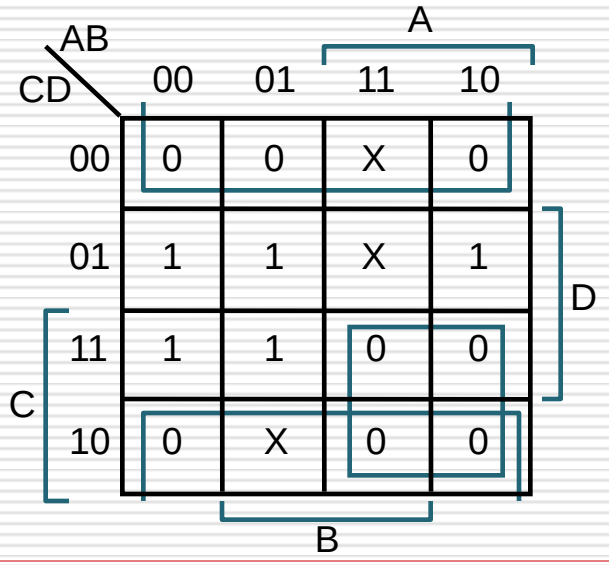
Don't Cares can be treated as 1's or 0's if it is advantageous to do so



F = A'D + B' C' D w/o don't cares

F = C' D + A' D w/ don't cares

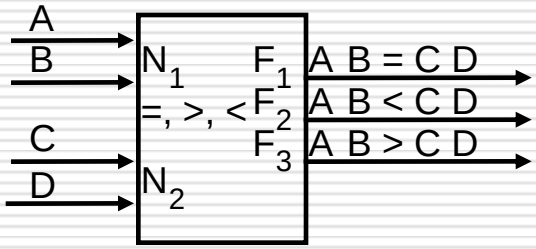
by treating this DC as a "1", a 2-cube can be formed rather than one 0-cube



In PoS form: F = D (A' + C')

same answer as above, but fewer literals

Design Example: Two Bit Comparator

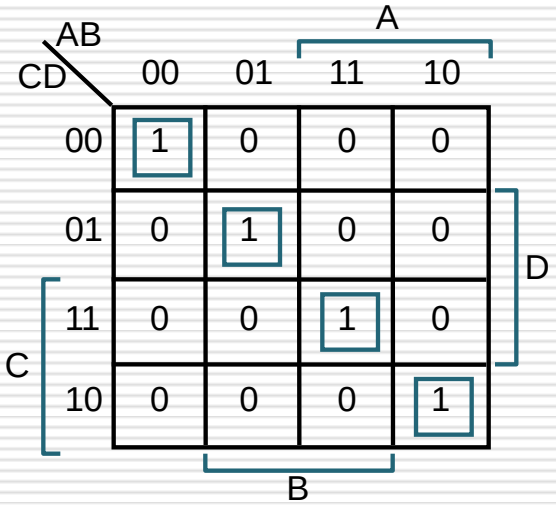


A	B	C	D	F_1	F_2	F_3
0	0	0	0	1	0	0
		0	1	0	1	0
		1	0	0	1	0
		1	1	0	1	0
0	1	0	0	0	0	1
		0	1	1	0	0
		1	0	0	1	0
		1	1	0	1	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	1	0	0
		1	1	0	1	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	1	0	0

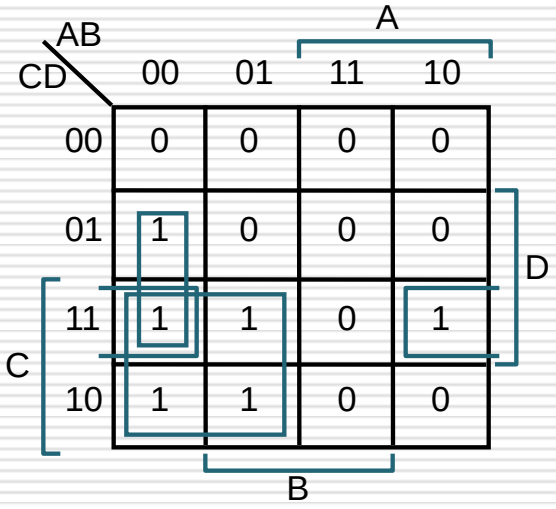
Block Diagram and Truth Table

A 4-Variable K-map for each of the 3 output functions

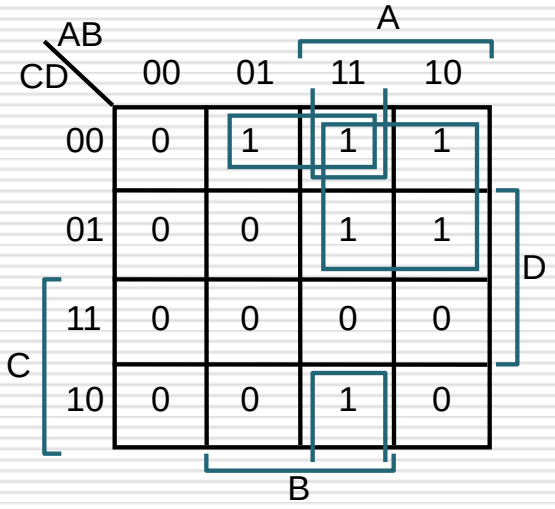
Design Example: Two Bit Comparator



K-map for F_1



K-map for F_2



K-map for F_3

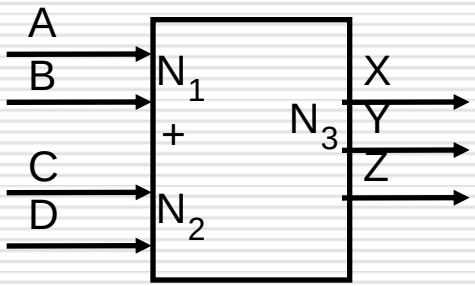
F1 = A' B' C' D' + A' B C' D + A B C D + A B' C D'

F2 = A' B' D + A' C + B' C D

F3 = B C' D' + A C' + A B D'

(A xnor C)(B xnor D)

Design Example: Two Bit Adder

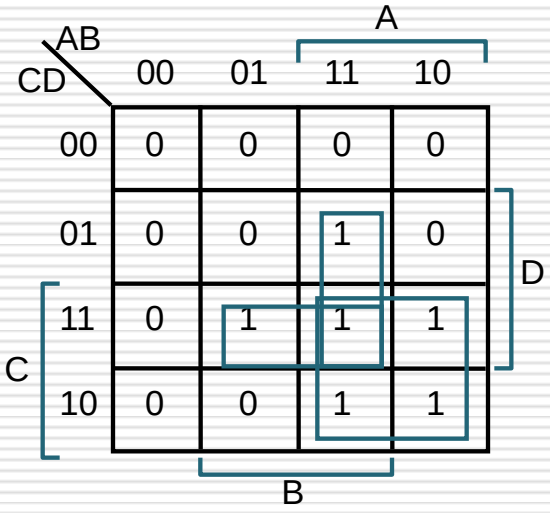


A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
		0	1	0	0	1
		1	0	0	1	0
		1	1	0	1	1
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	0	1	1
		1	1	1	0	0
1	0	0	0	0	1	0
		0	1	0	1	1
		1	0	1	0	0
		1	1	1	0	1
1	1	0	0	0	1	1
		0	1	1	0	0
		1	0	1	0	1
		1	1	1	1	0

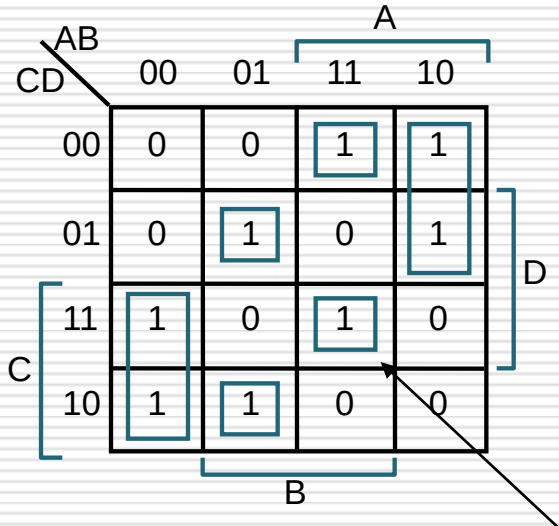
Block Diagram and Truth Table

A 4-variable K-map for each of the 3 output functions

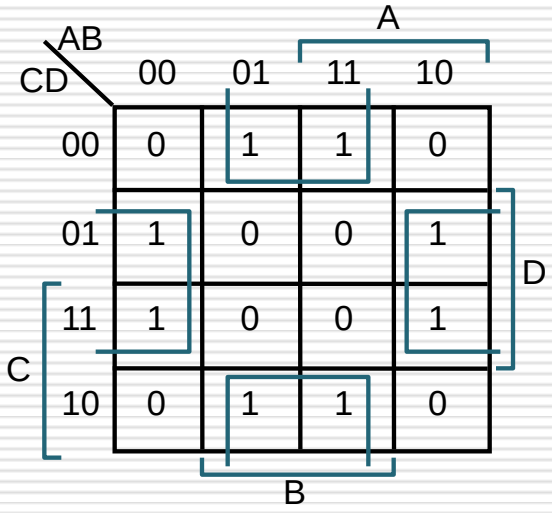
Design Example: Two Bit Adder



K-map for X



K-map for Y



K-map for Z

1's on diagonal suggest XOR!
Y K-Map not minimal as drawn

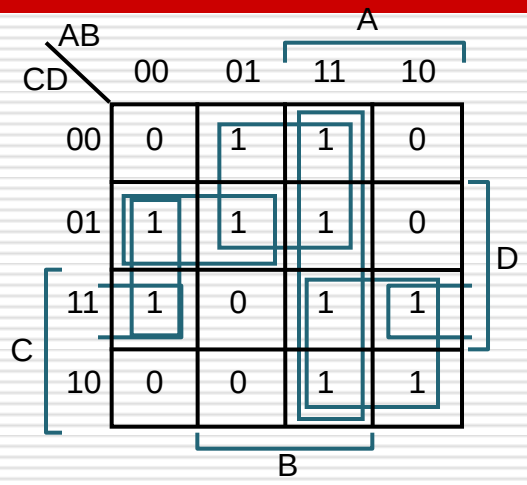
X = A C + B C D + A B D
Z = B D' + B' D = B xor D
Y = A' B' C + A B' C' + A' B C' D + A' B C D' + A B C' D' + A B C D
= B' (A xor C) + A' B (C xor D) + A B (C xnor D)
= B' (A xor C) + B (A xor B xor C)

gate count reduced if XOR available

Two Level Simplification

- Definition of Terms
  - *implicant*:
    - single element of the ON-set or any group of elements that can be combined together in a K-map
  - *prime implicant*:
    - implicant that cannot be combined with another implicant to eliminate a term
  - *essential prime implicant*:
    - if an element of the ON-set is covered by a single prime implicant, it is an essential prime
- Objective:
  - grow implicants into prime implicants
  - cover the ON-set with as few prime implicants as possible
  - essential primes participate in ALL possible covers

Examples to Illustrate Terms

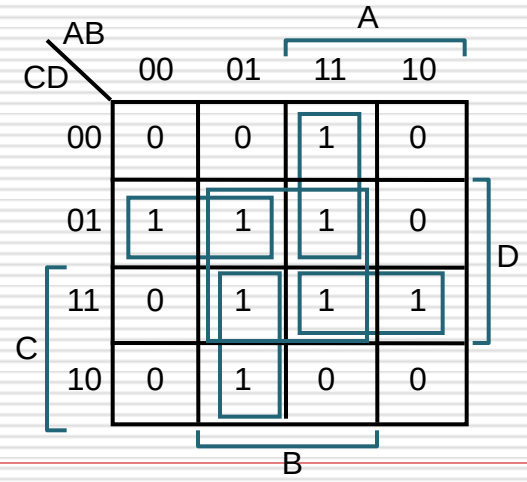


6 Prime Implicants:

A' B' D, B C', A C, A' C' D, A B, B' C D

essential

Minimum cover = B C' + A C + A' B' D



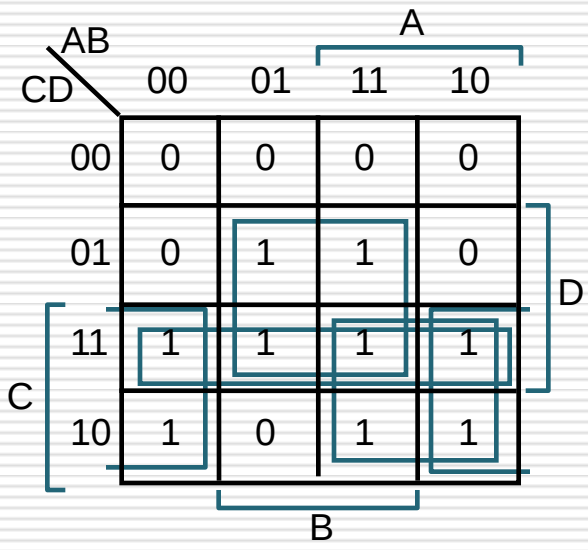
5 Prime Implicants:

B D, A B C', A C D, A' B C, A' C' D

essential

Essential implicants form minimum cover

Examples to Illustrate Terms



Prime Implicants:

B D, C D, A C, B' C

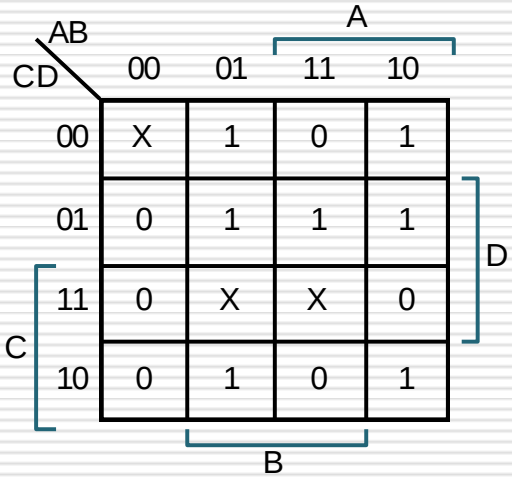
essential

Essential primes form the minimum cover

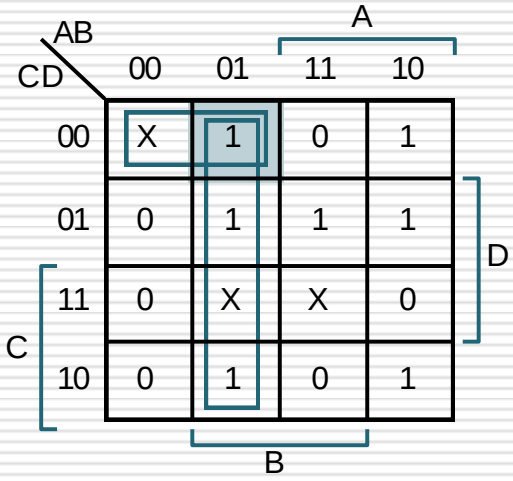
Algorithm: Minimum Sum of Products Expression from a K-Map

1. Choose an element of ON-set not already covered by an implicant
2. Find "maximal" groupings of 1's and X's adjacent to that element. Remember to consider top/bottom row, left/right column, and corner adjacencies. This forms prime implicants (always a power of 2 number of elements).
- Repeat Steps 1 and 2 to find all prime implicants
3. Revisit the 1's elements in the K-map. If covered by single prime implicant, it is essential, and participates in final cover. The 1's it covers do not need to be revisited
4. If there remain 1's not covered by essential prime implicants, then select the smallest number of prime implicants that cover the remaining 1's

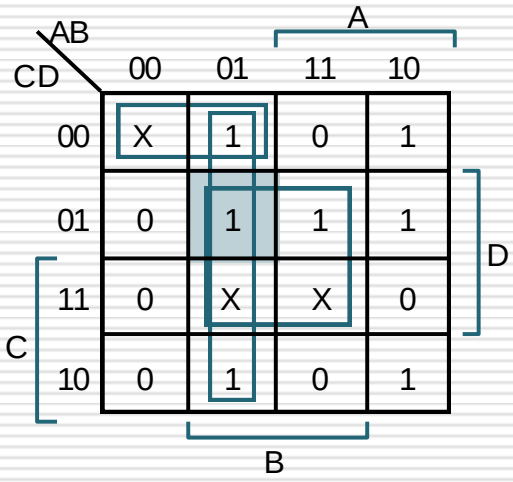
Example



Initial K-map

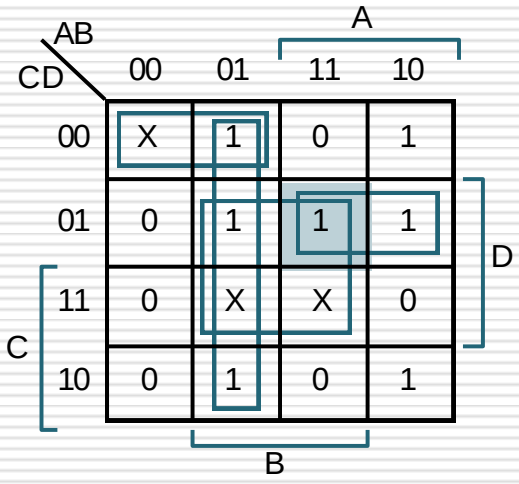


Primes around A' B C' D'

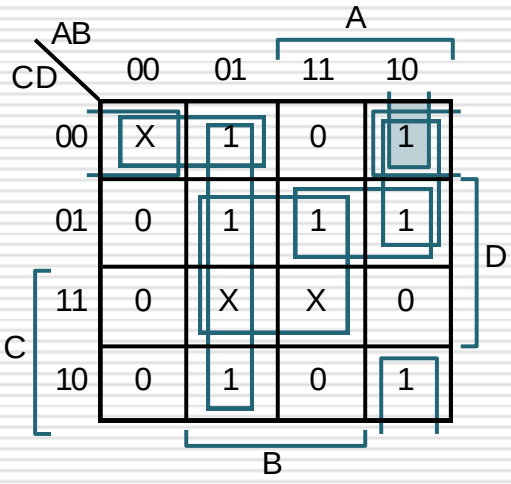


Primes around A B C' D

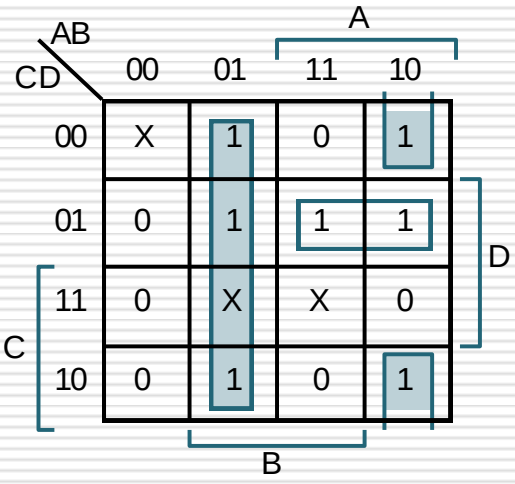
Example



Primes around A B C' D



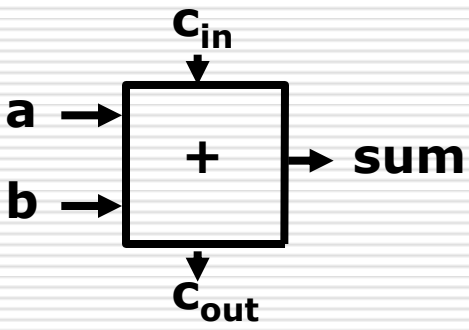
Primes around A B' C' D'



Essential Primes with Min Cover

Different Implementations

- 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
- Let's look at a 1-bit ALU for addition:

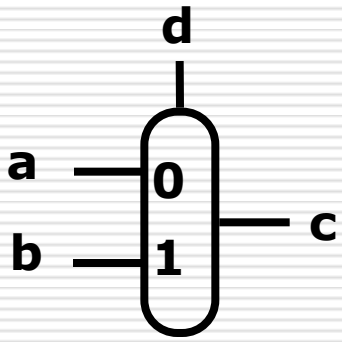


$$c_{out} = ab + ac_{in} + bc_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

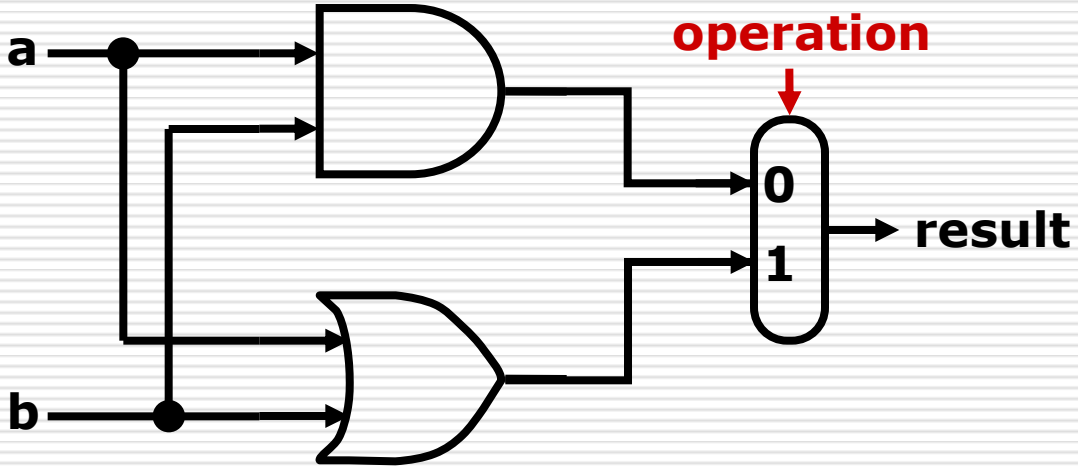


d	c
0	a
1	b

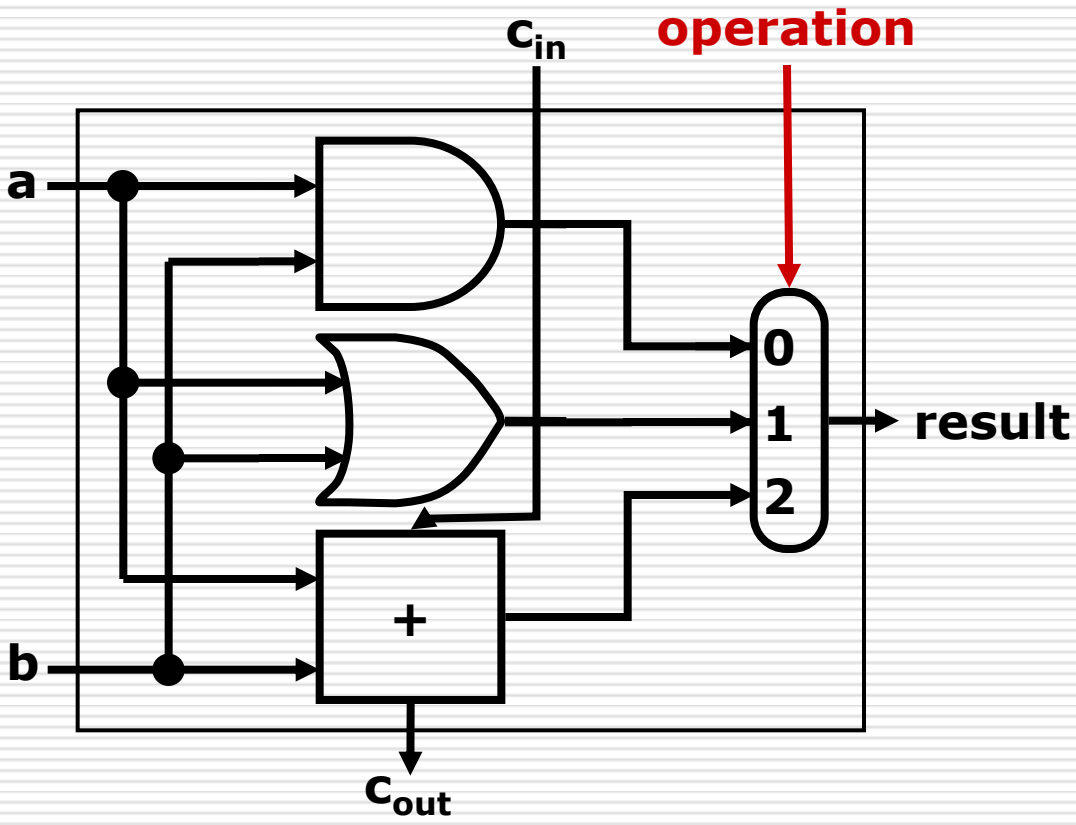
note: we call this a 2-input mux even though it has 3 inputs!

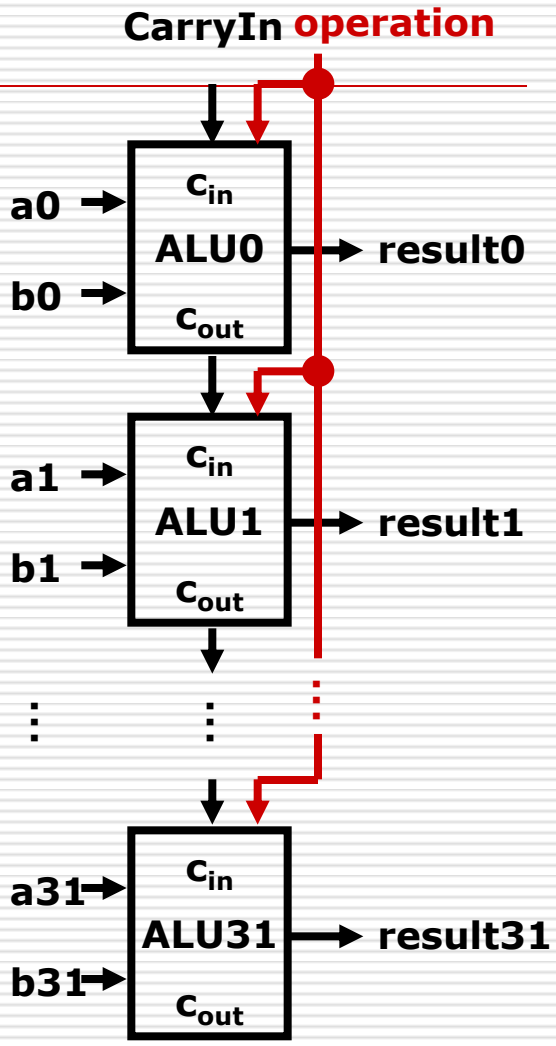
- Lets build our ALU using a MUX

1-bit ALU for AND & OR

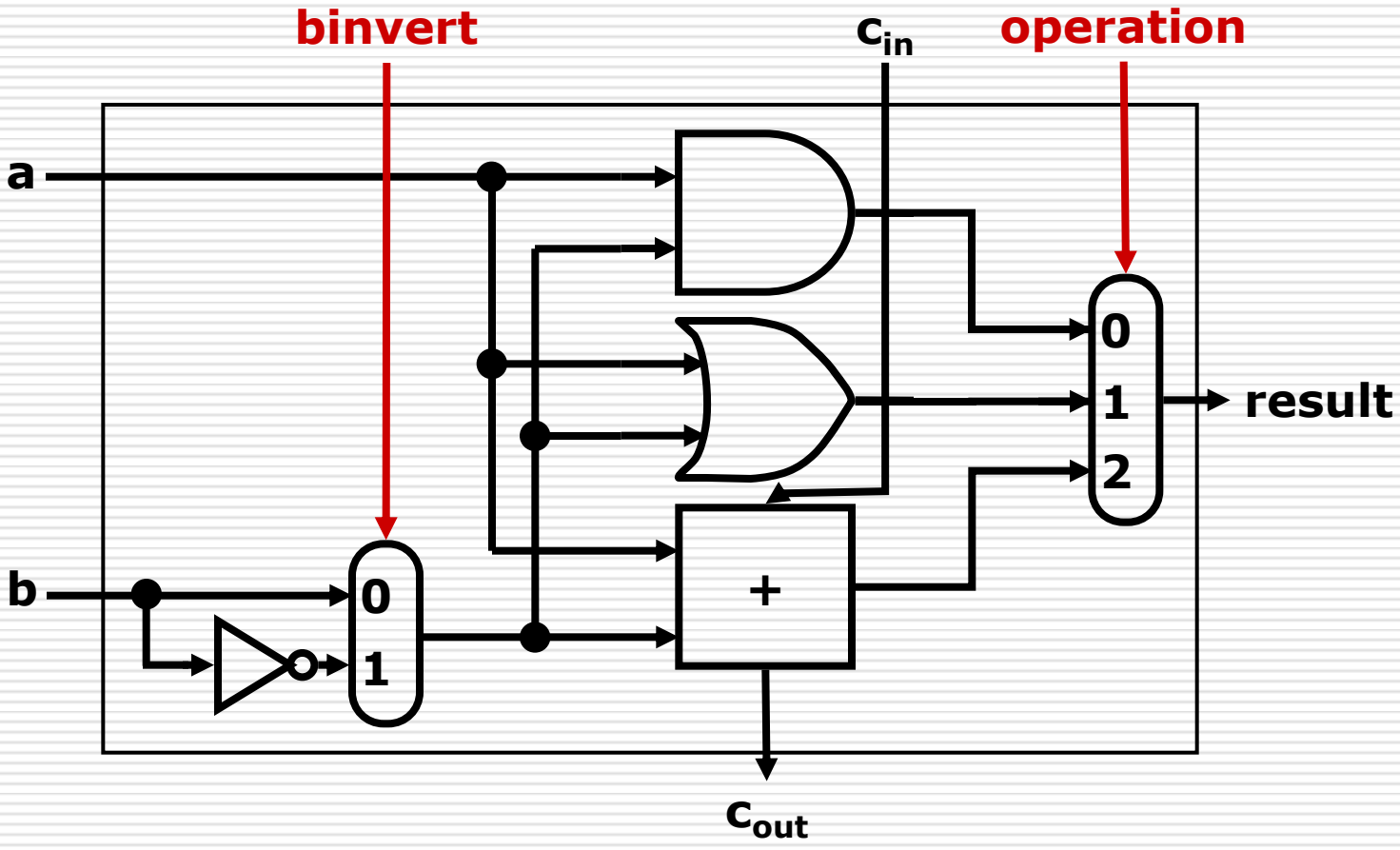


Building a 32 bit ALU





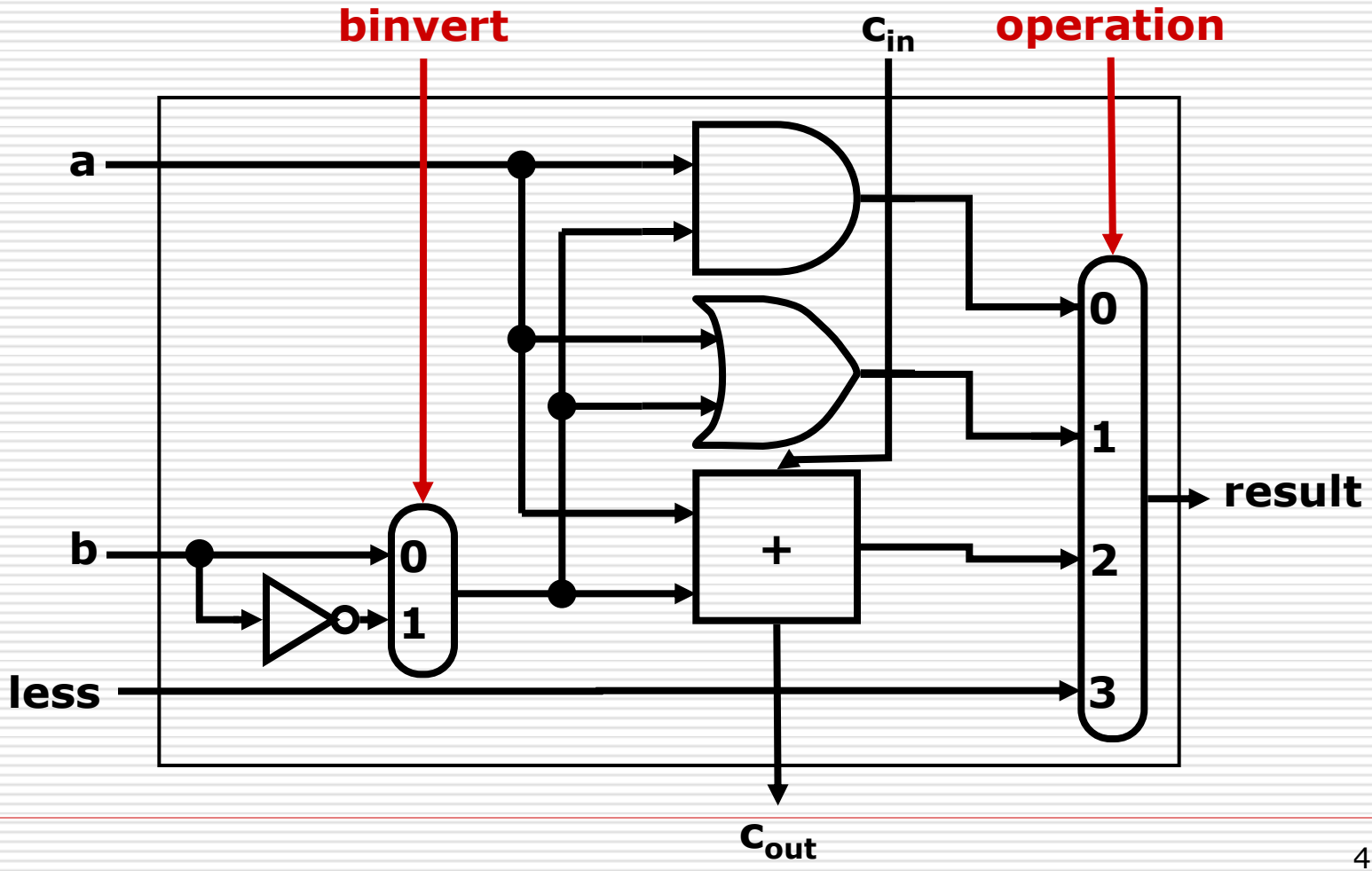
What about Subtraction ?



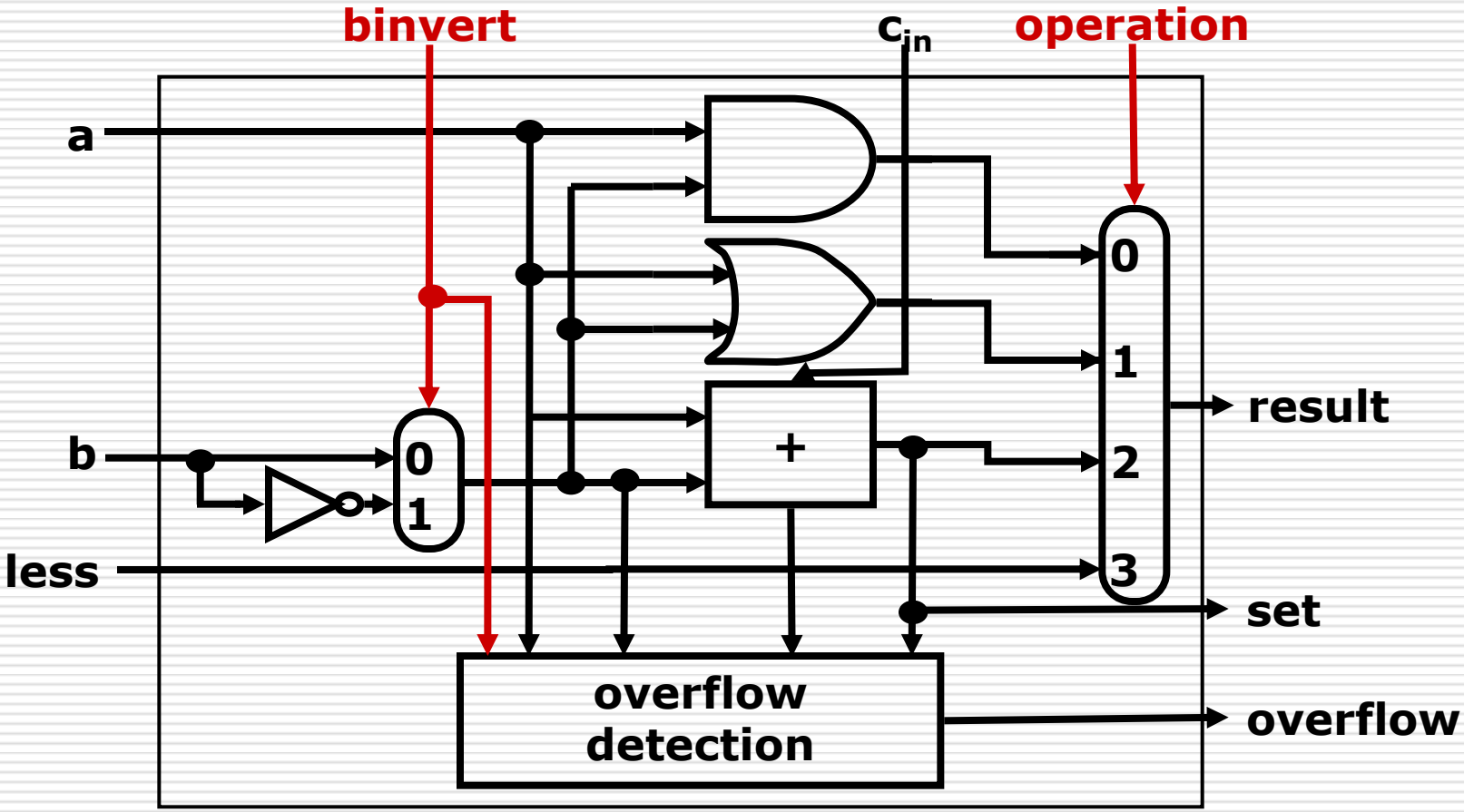
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (`slt`)
  - remember: `slt` is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (`beq $t5, $t6, $t7`)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt – without overflow



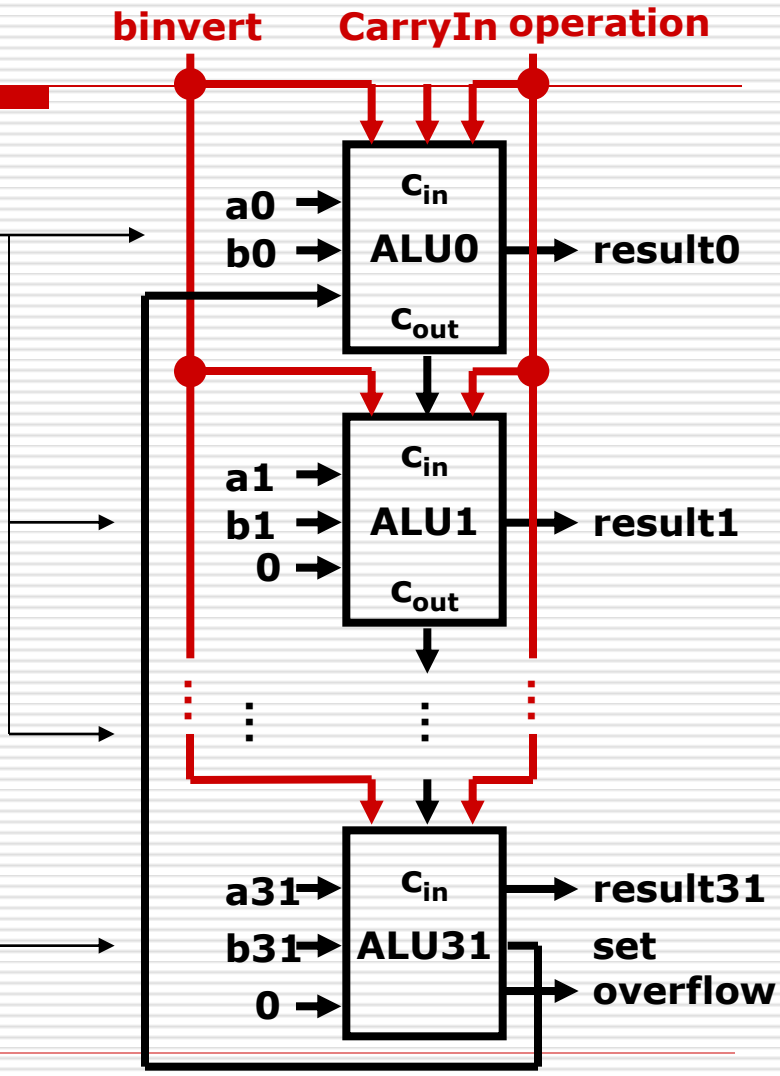
Supporting slt – with overflow & set



A 32 bit ALU

1 bit ALU without overflow & set

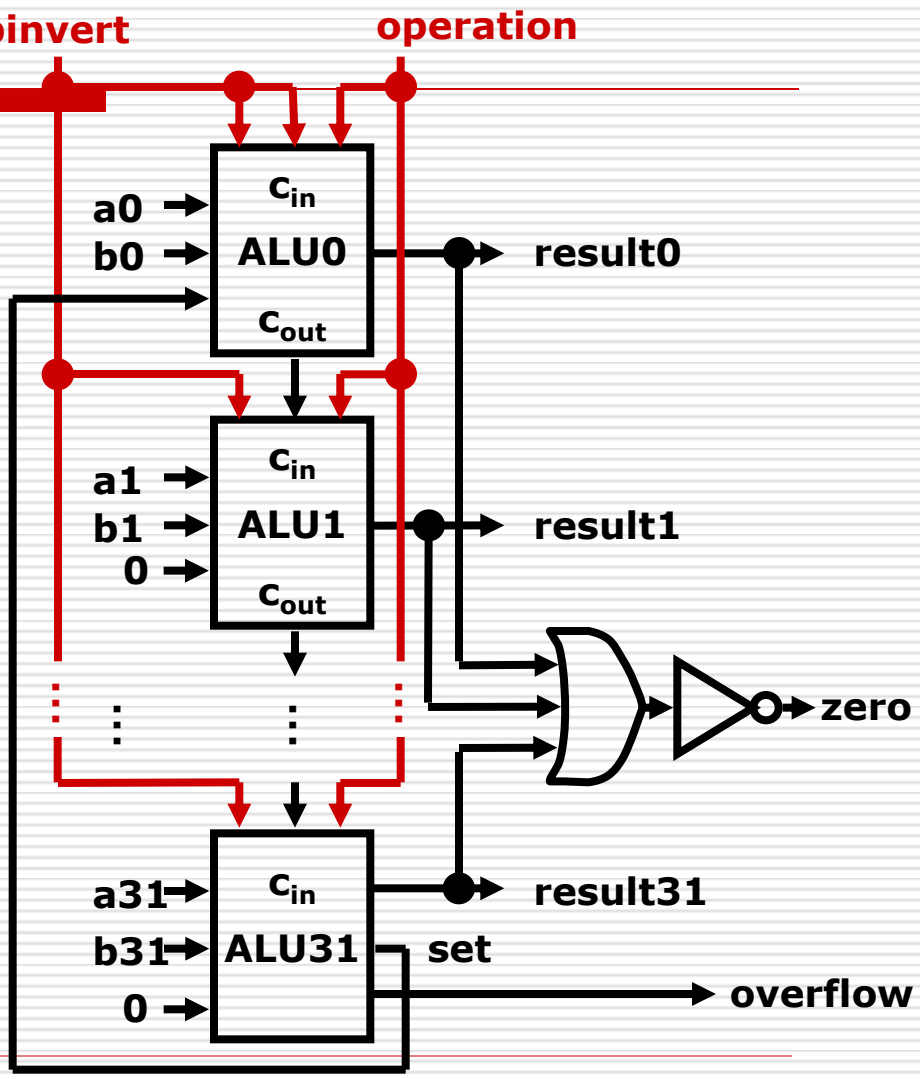
1 bit ALU with overflow & set



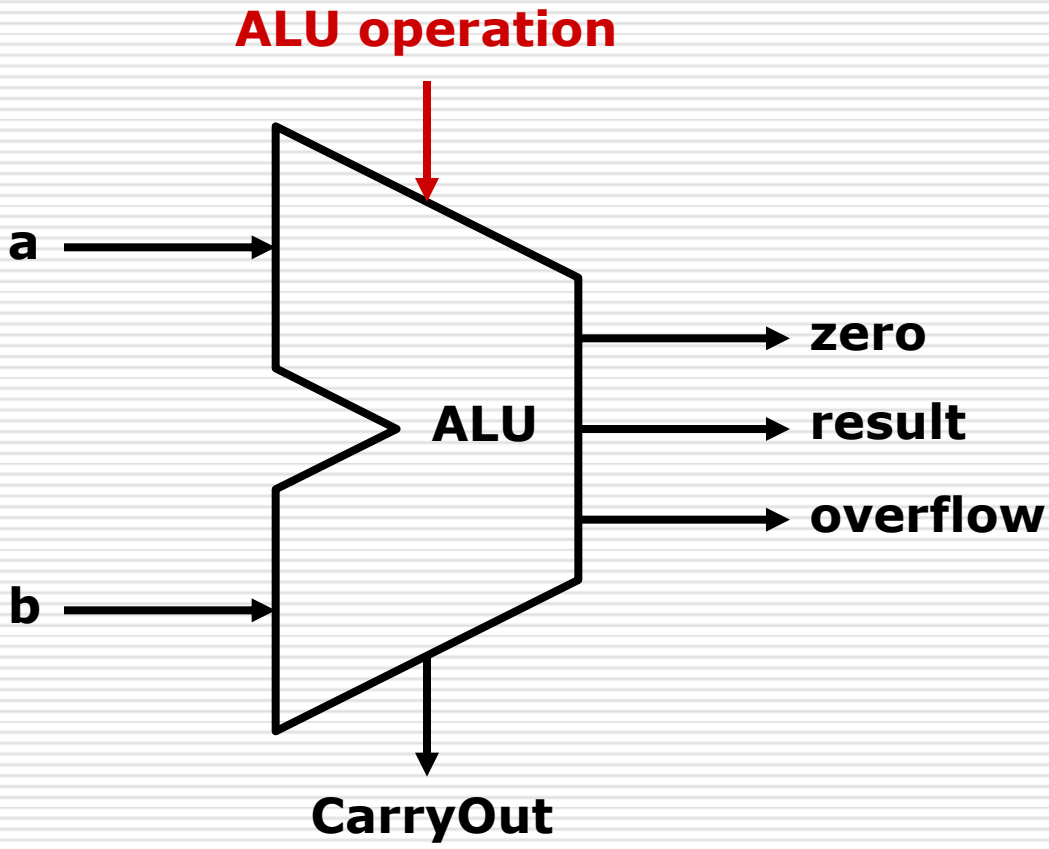
Test for Equality

- control lines:
  - 000 = and
  - 001 = or
  - 010 = add
  - 110 = subtract
  - 111 = slt
  - 110 = beq

(use zero as the output)



The ALU Symbol



Multiplication

- more complicated than addition
  - accomplished via shifting and addition
- more time and more area
- Let's look at 3 versions based on grade school algorithm

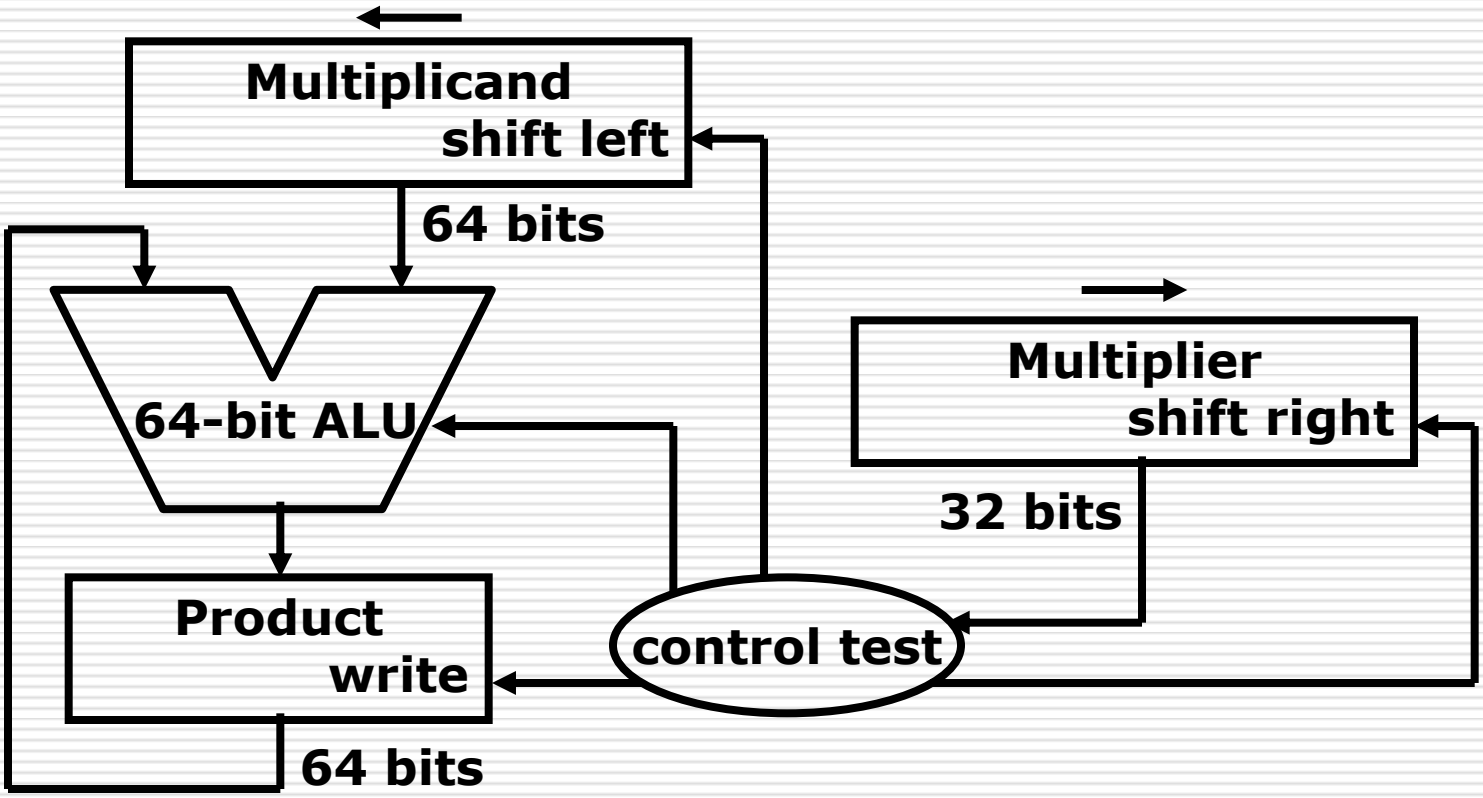
```
      0010    (multiplicand)
x     0011    (multiplier)
```

- negative numbers: convert and multiply

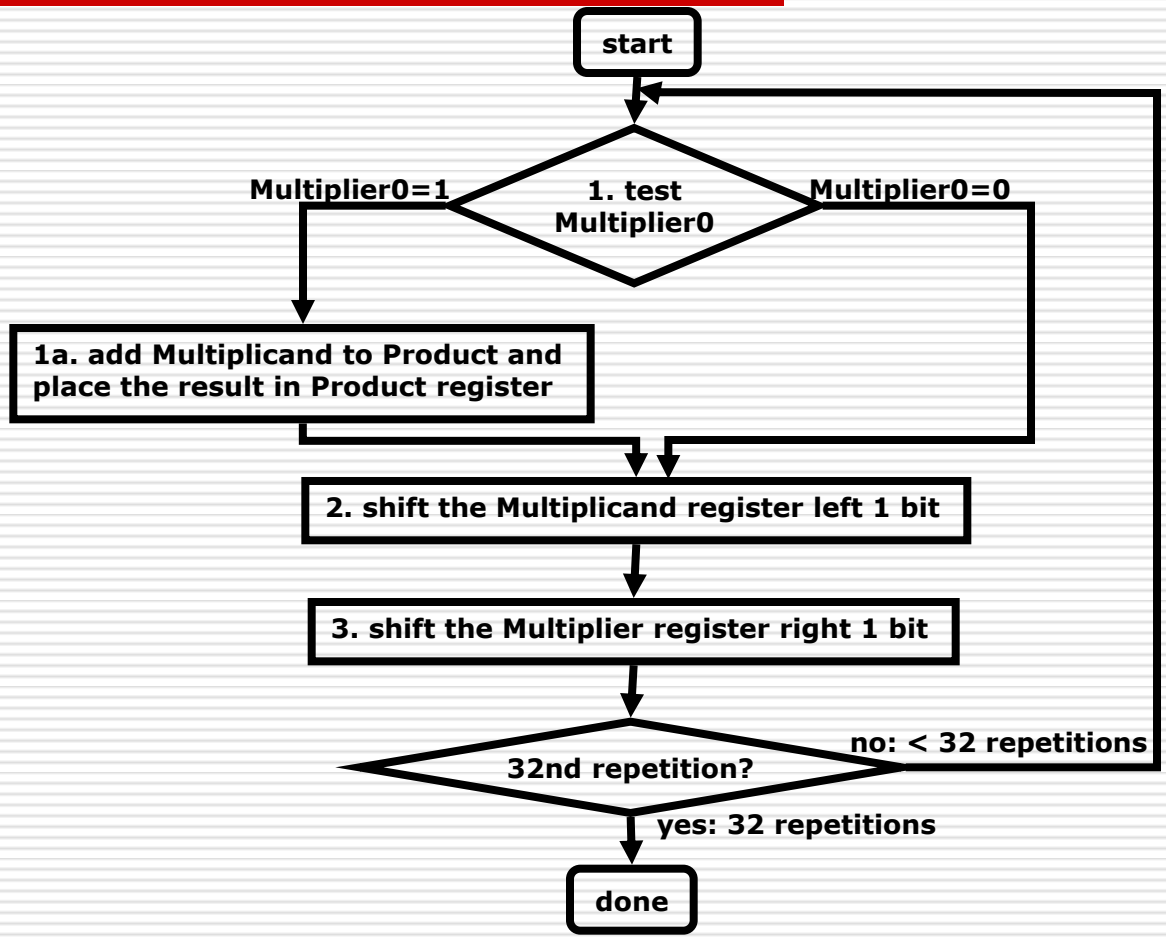
Multiplication

```
      0010    (multiplicand)
x     0011    (multiplier)
      0010
     0010
    0000
   0000
  ---------
  0000110
```

Multiplication Hardware: First Version



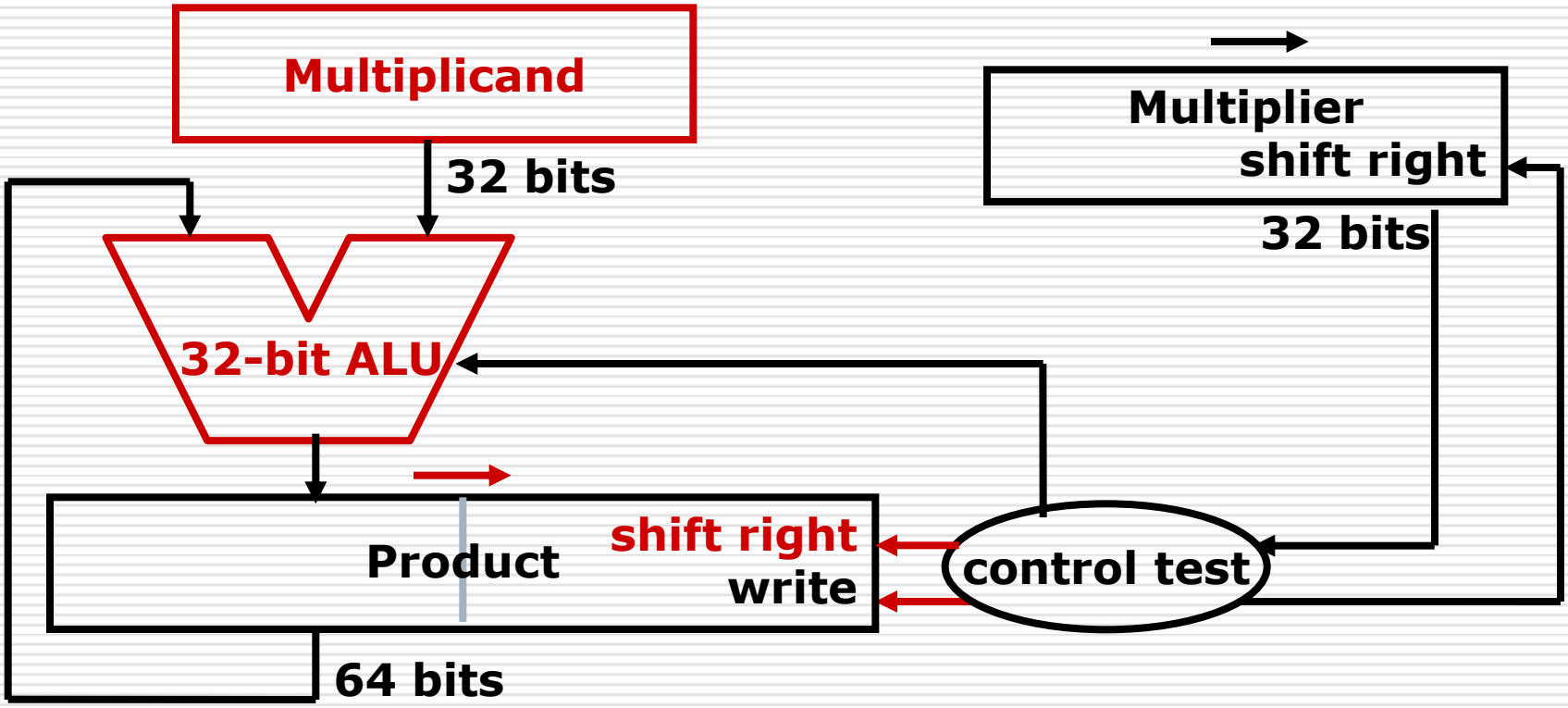
Multiplication Algorithm: First Version



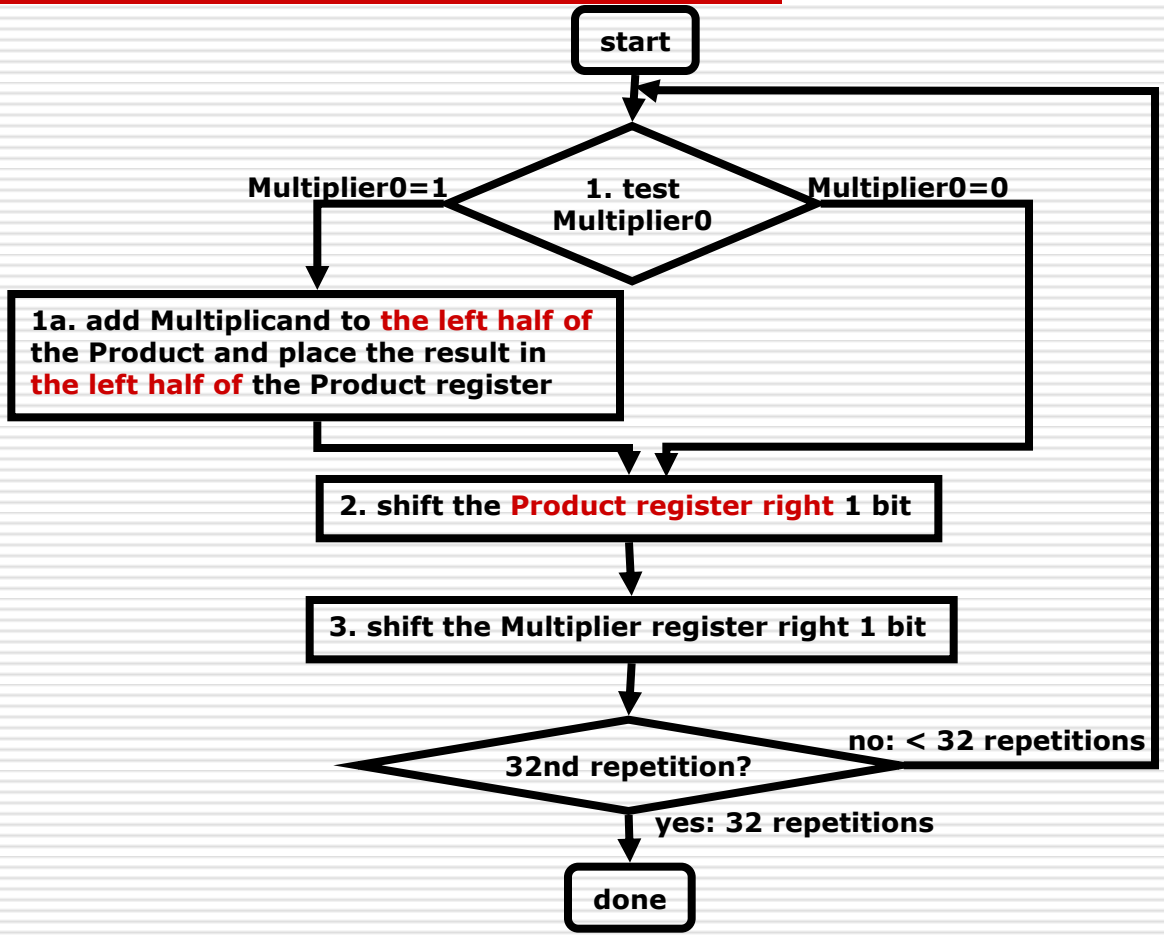
Multiplication Example: First Version

iteration	step	Multiplier	Multiplicand	Product
0	initial value	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: shift left Multiplicand	0011	0000 0100	0000 0010
	3: shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: shift left Multiplicand	0001	0000 1000	0000 0110
	3: shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: shift left Multiplicand	0000	0001 0000	0000 0110
	3: shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: shift left Multiplicand	0000	0010 0000	0000 0110
	3: shift right Multiplier	0000	0010 0000	0000 0110

Multiplication Hardware: Second Version



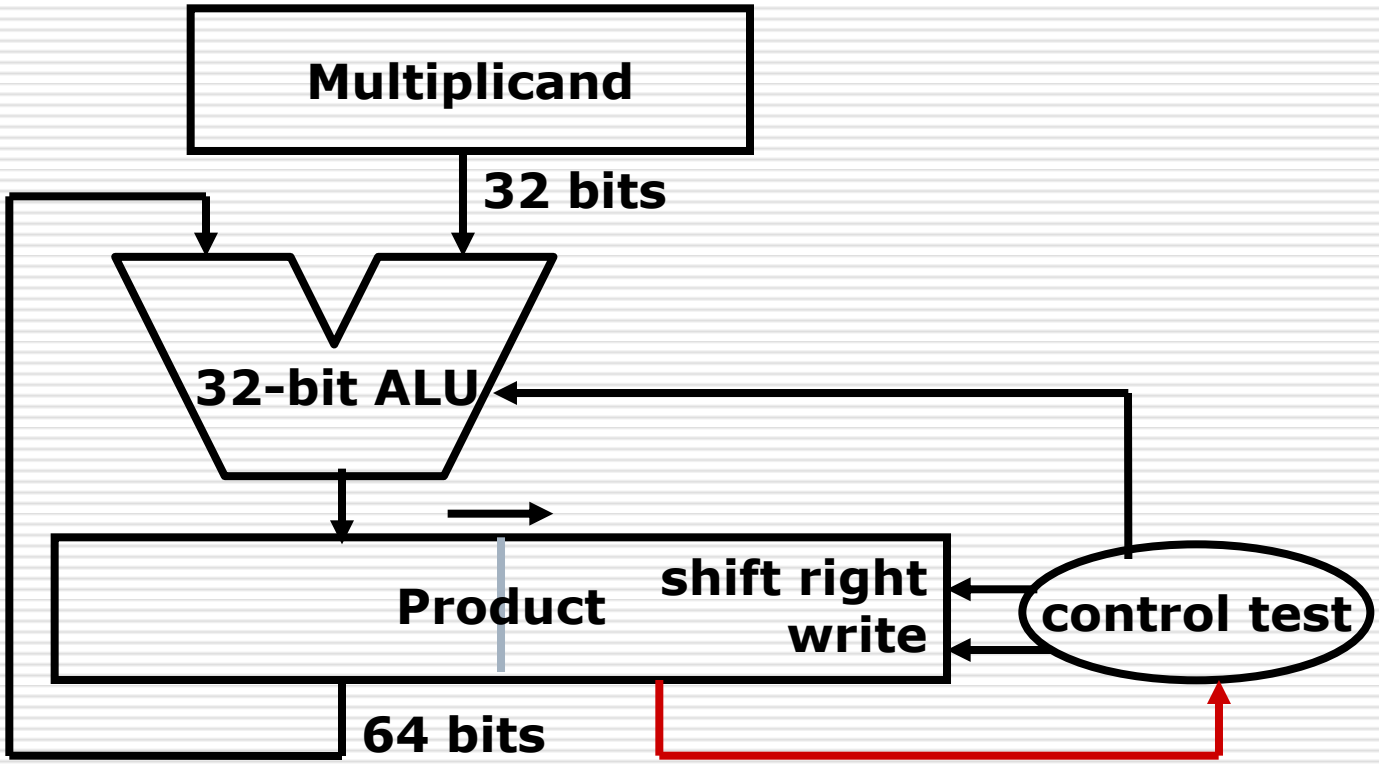
Multiplication Algorithm: Second Version



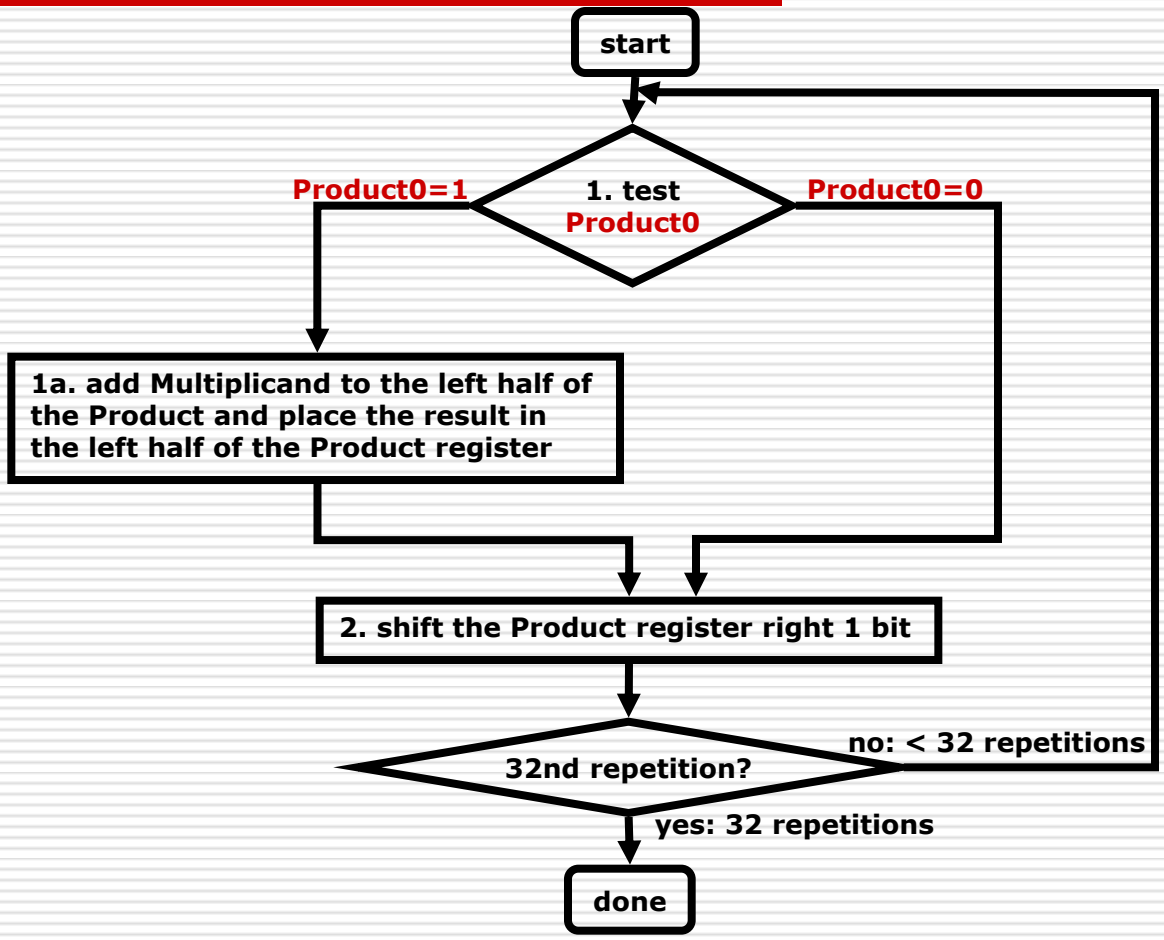
Multiplication Example: Second Version

iteration	step	Multiplier	Multiplicand	Product
0	initial value	0011	0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0010	0010 0000
	2: shift right Product	0011	0010	0001 0000
	3: shift right Multiplier	0001	0010	0001 0000
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0010	0011 0000
	2: shift right Product	0001	0010	0001 1000
	3: shift right Multiplier	0000	0010	0001 1000
3	1: 0 ⇒ no operation	0000	0010	0001 1000
	2: shift right Product	0000	0010	0000 1100
	3: shift right Multiplier	0000	0010	0000 1100
4	1: 0 ⇒ no operation	0000	0010	0000 1100
	2: shift right Product	0000	0010	0000 0110
	3: shift right Multiplier	0000	0010	0000 0110

Multiplication Hardware: Third Version



Multiplication Algorithm: Third Version



Multiplication Example: Third Version

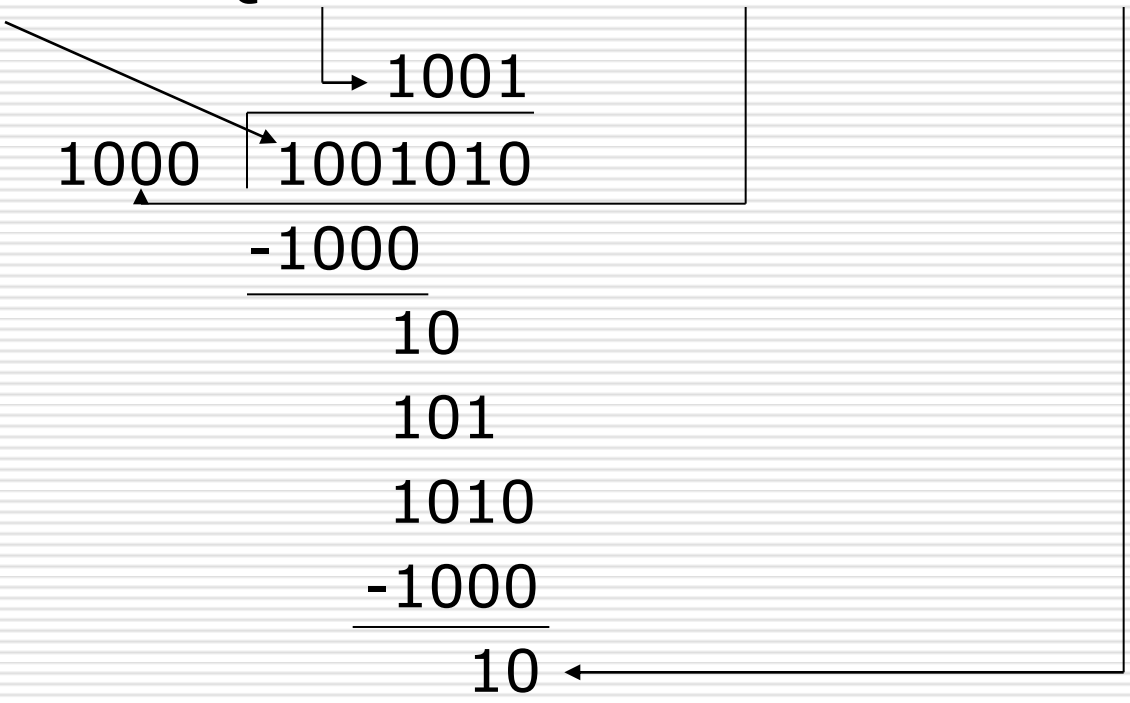
iteration	step	Multiplicand	Product
0	initial value	0010	0000 0011
1	1a: 1 ⇒ Prod = Prod + Mcand	0010	0010 0011
	2: shift right Product	0010	0001 0001
2	1a: 1 ⇒ Prod = Prod + Mcand	0010	0011 0001
	2: shift right Product	0010	0001 1000
3	1: 0 ⇒ no operation	0010	0001 1000
	2: shift right Product	0010	0000 1100
4	1: 0 ⇒ no operation	0010	0000 1100
	2: shift right Product	0010	0000 0110

MIPS Multiplication

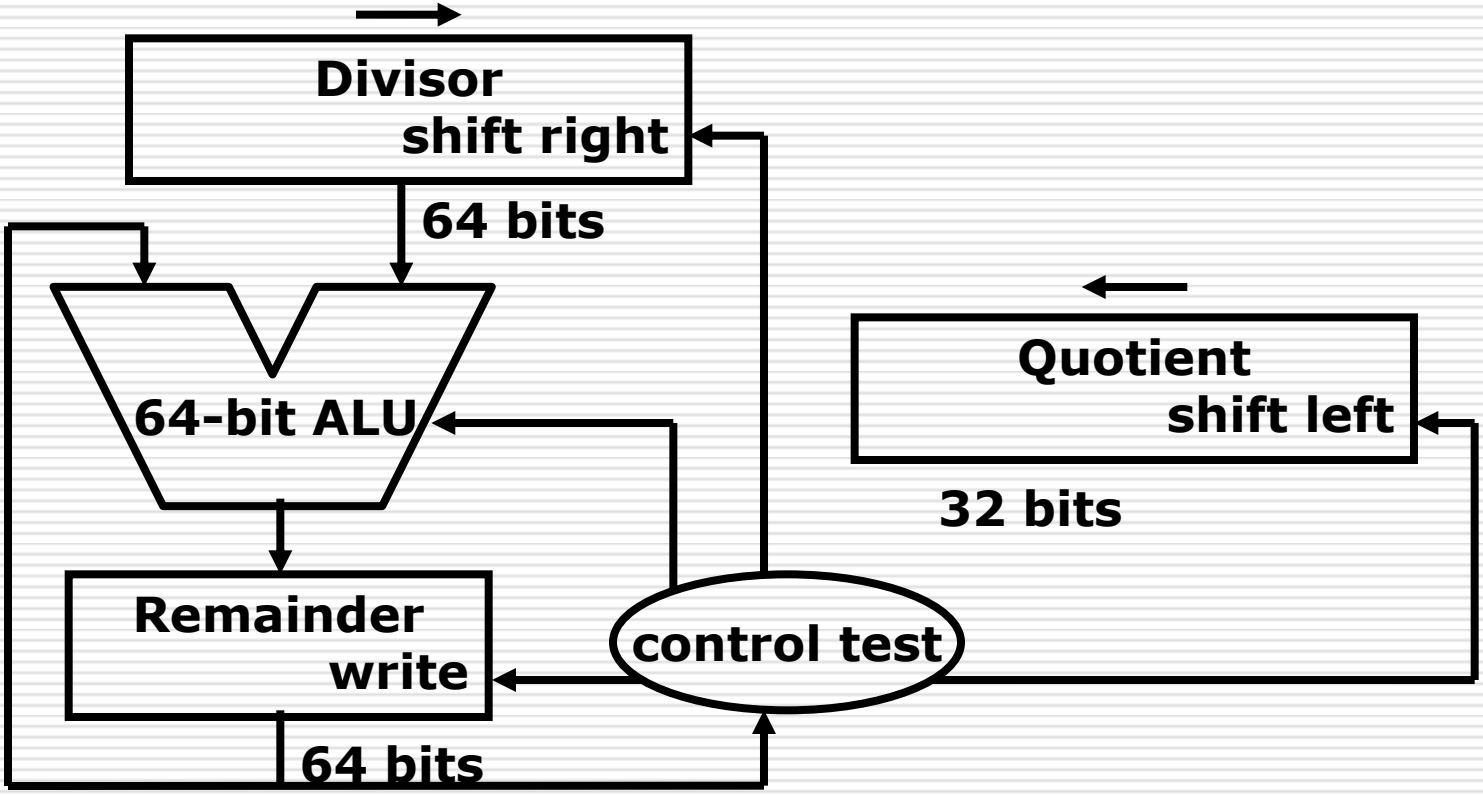
- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt` / `multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd` / `mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

Division

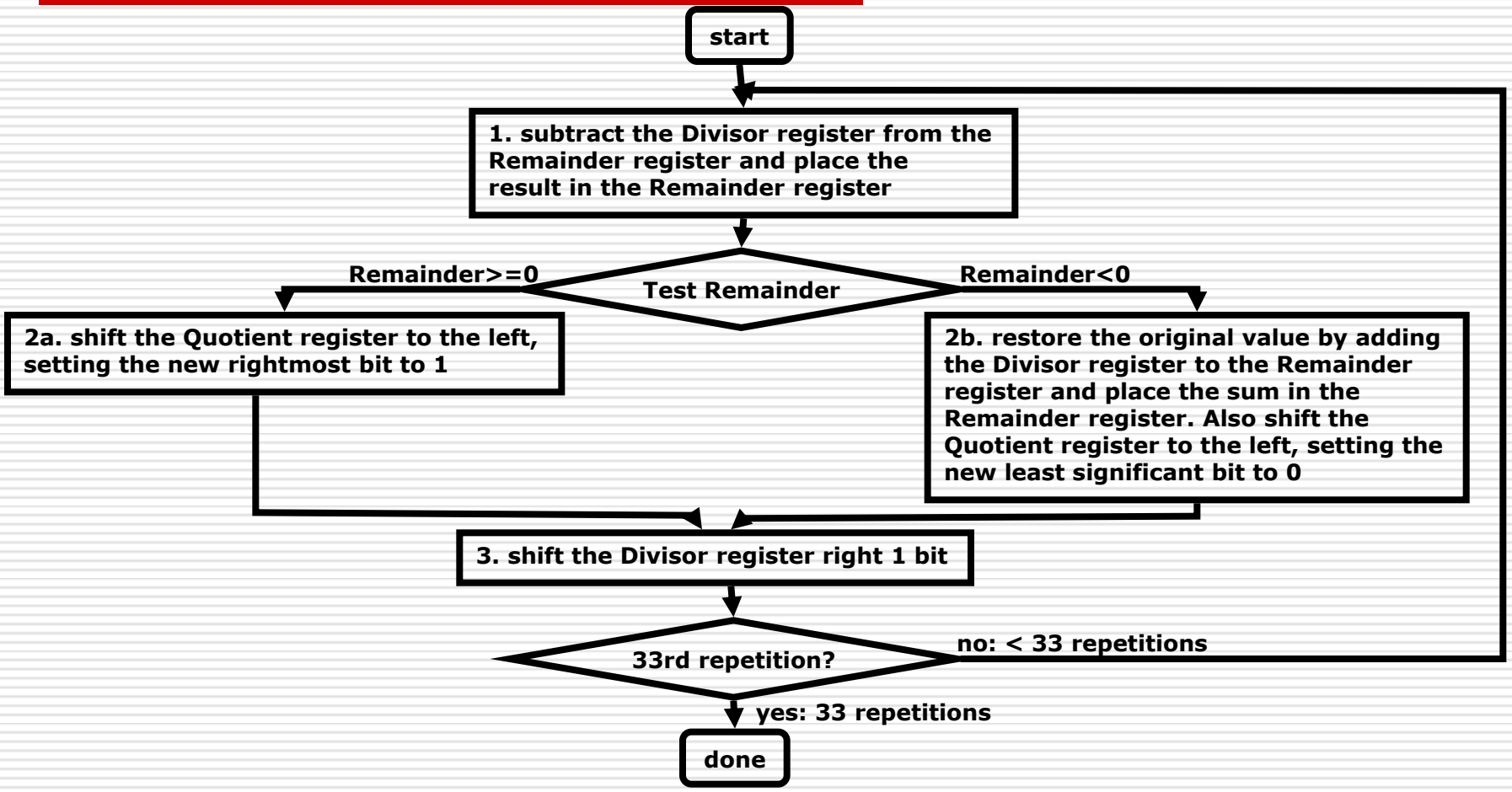
- Dividend = Quotient x Divisor + Remainder



Division Hardware: First Version



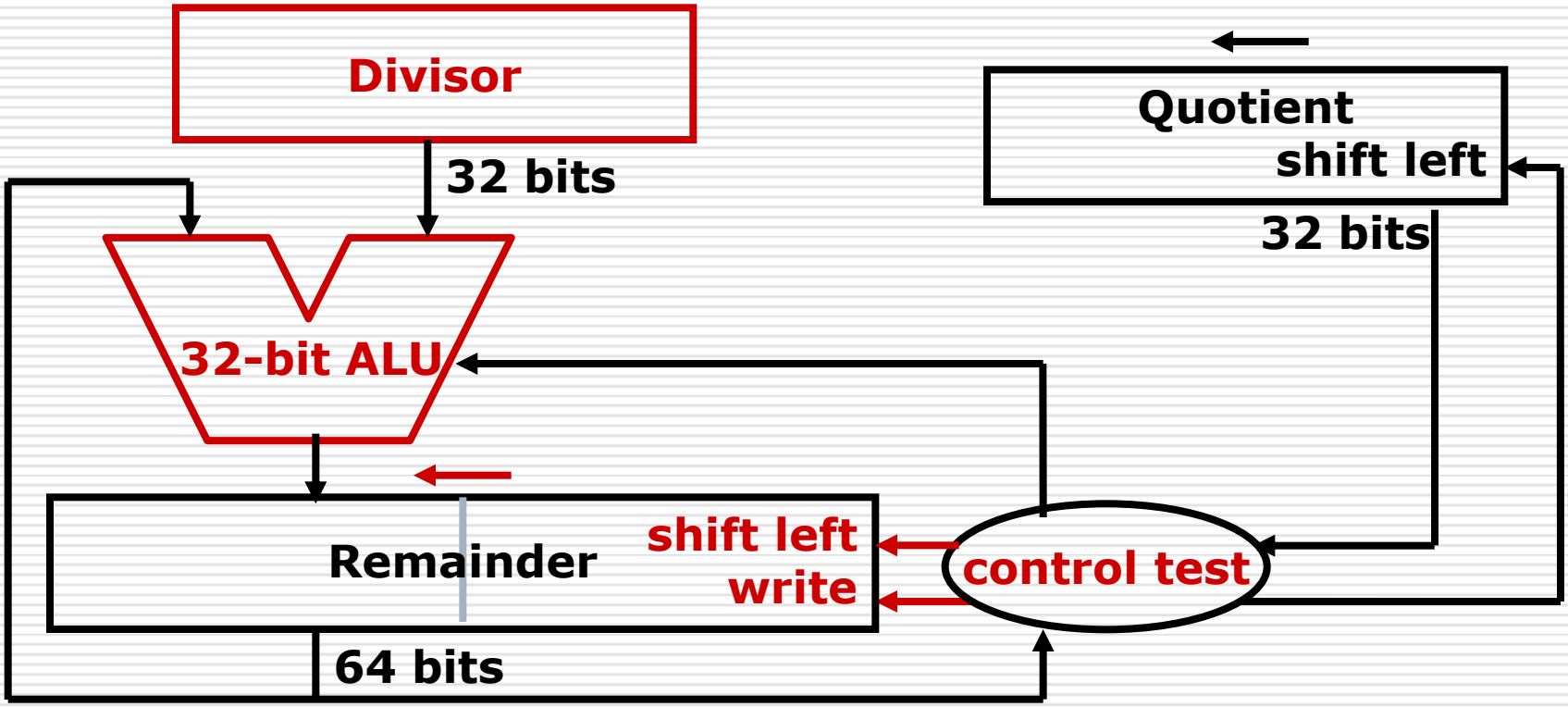
Division Algorithm: First Version



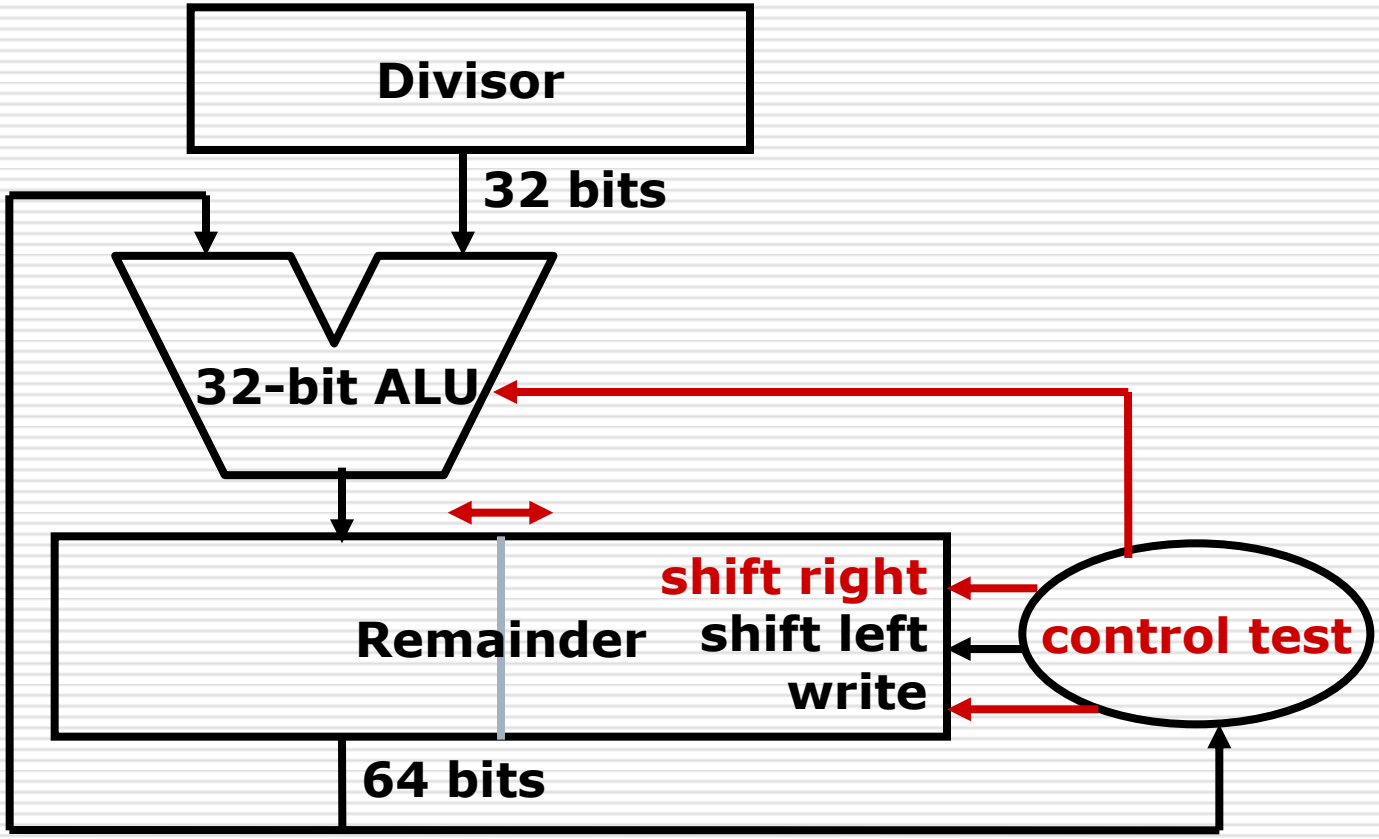
Division Example: First Version

iteration	step	Quotient	Divisor	Remainder
0	initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0=0	0000	0010 0000	0000 0111
	3: shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0=0	0000	0001 0000	0000 0111
	3: shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0=0	0000	0000 1000	0000 0111
	3: shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0=1	0001	0000 0100	0000 0011
	3: shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0=1	0011	0000 0010	0000 0001
	3: shift Div right	0011	0000 0001	0000 0001

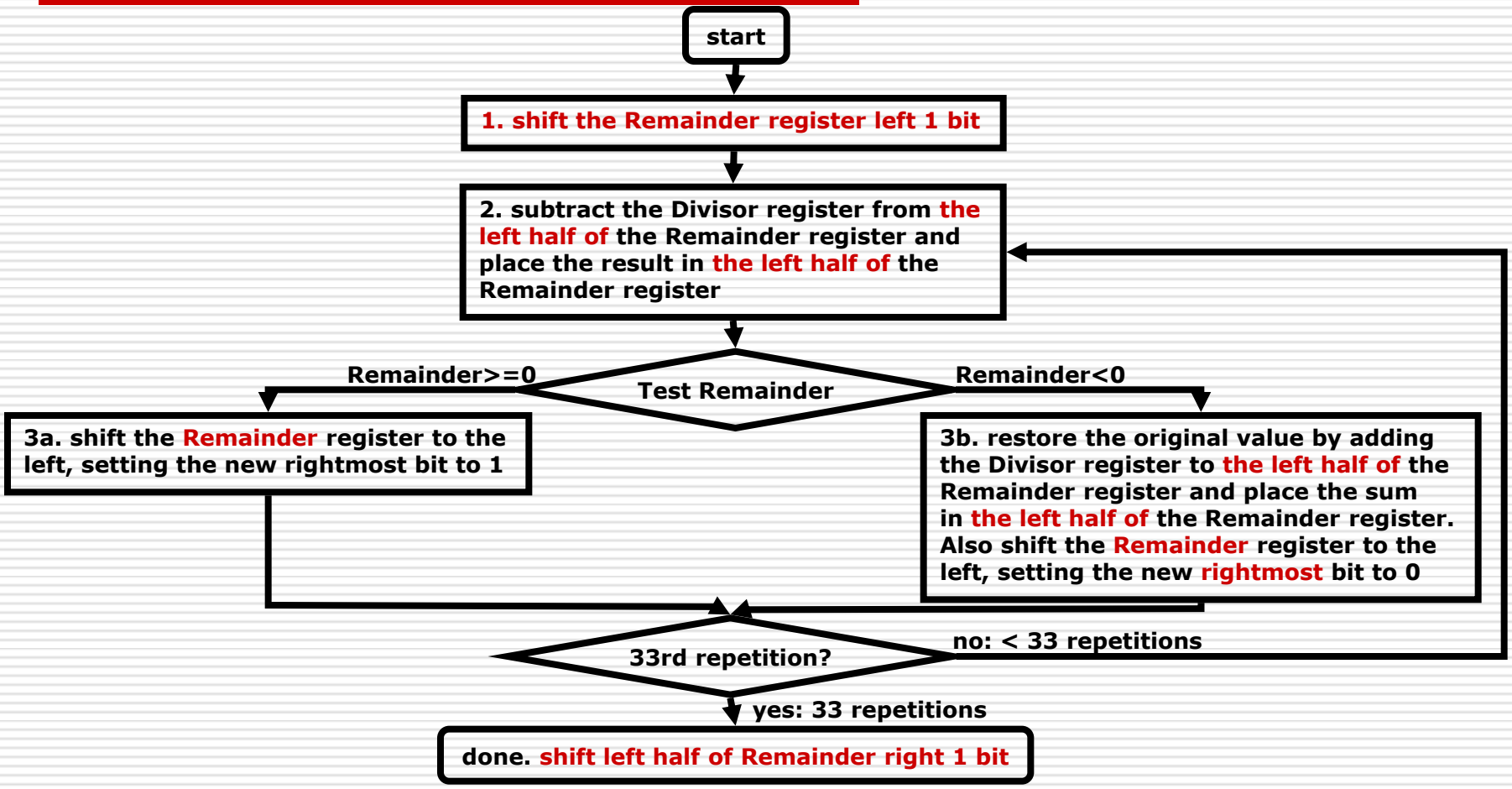
Division Hardware: Second Version



Division Hardware: Third Version



Division Algorithm: Third Version



Division Example: Third Version

iteration	step	Divisor	Remainder
0	initial value	0010	0000 0111
	shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 ⇒ +Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 ⇒ +Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⇒ +Div, sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⇒ +Div, sll R, R0 = 1	0010	0010 0011
	shift left half of Rem right 1	0010	0001 0011

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

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. $11111011_2 >>> 2 = 00111110_2 = +62$

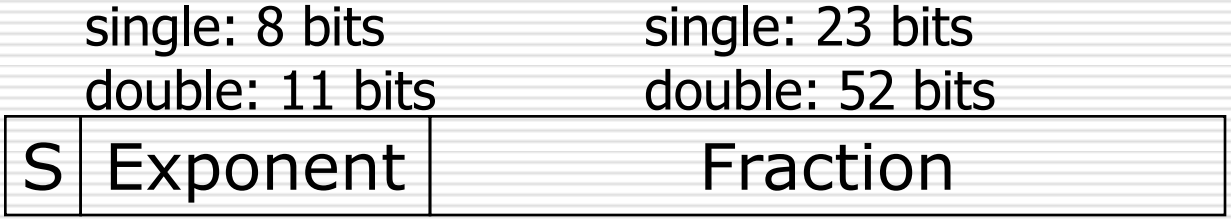
Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.14159265
  - very small numbers, e.g., 0.000000001
  - very large numbers, e.g., 3.15576×10^9
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- 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)

IEEE Floating-Point Format



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

- 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
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - Exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 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: 10111111101000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000010 1000…00

  - S = 1
  - Fraction = 01000…00$_2$
  - Exponent = 10000001$_2$ = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

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

  $= -5.0$

Floating-Point Addition

- $9.999 \times 10^{1} + 1.610 \times 10^{-1} = ?$

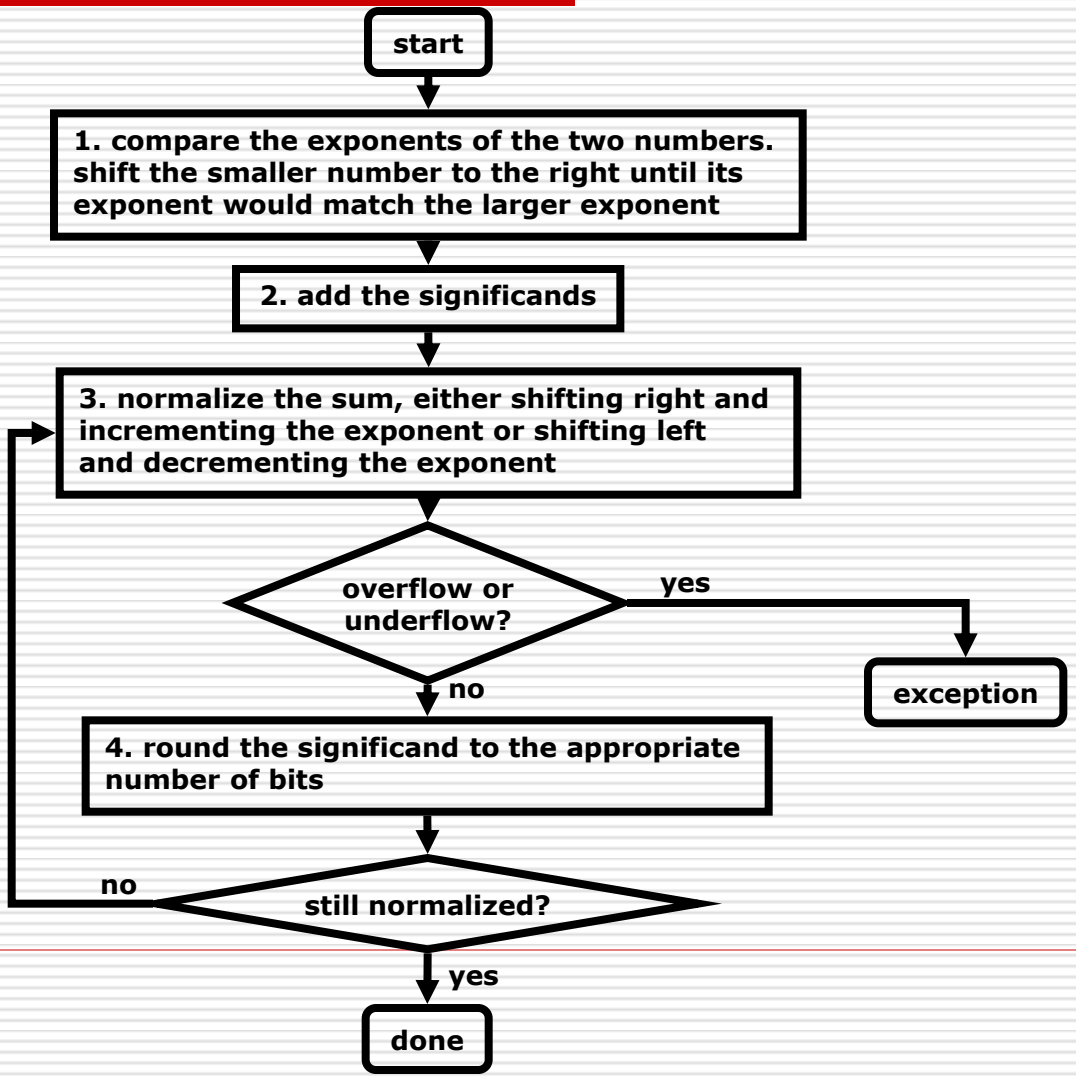
1. $1.610 \times 10^{-1} = \mathbf{0.0161 \times 10^{1}}$
   - *align decimal points*
   - *shift number with smaller exponent*
2. $9.999 + 0.016 = \mathbf{10.015}$
   - *add significands*
3. $10.015 \times 10^{1} = \mathbf{1.0015 \times 10^{2}}$
   - *normalize result & check for over/underflow*
4. $1.0015 \times 10^{2} = \mathbf{1.002 \times 10^{2}}$
   - *round and renormalize if necessary*

Floating-Point Addition

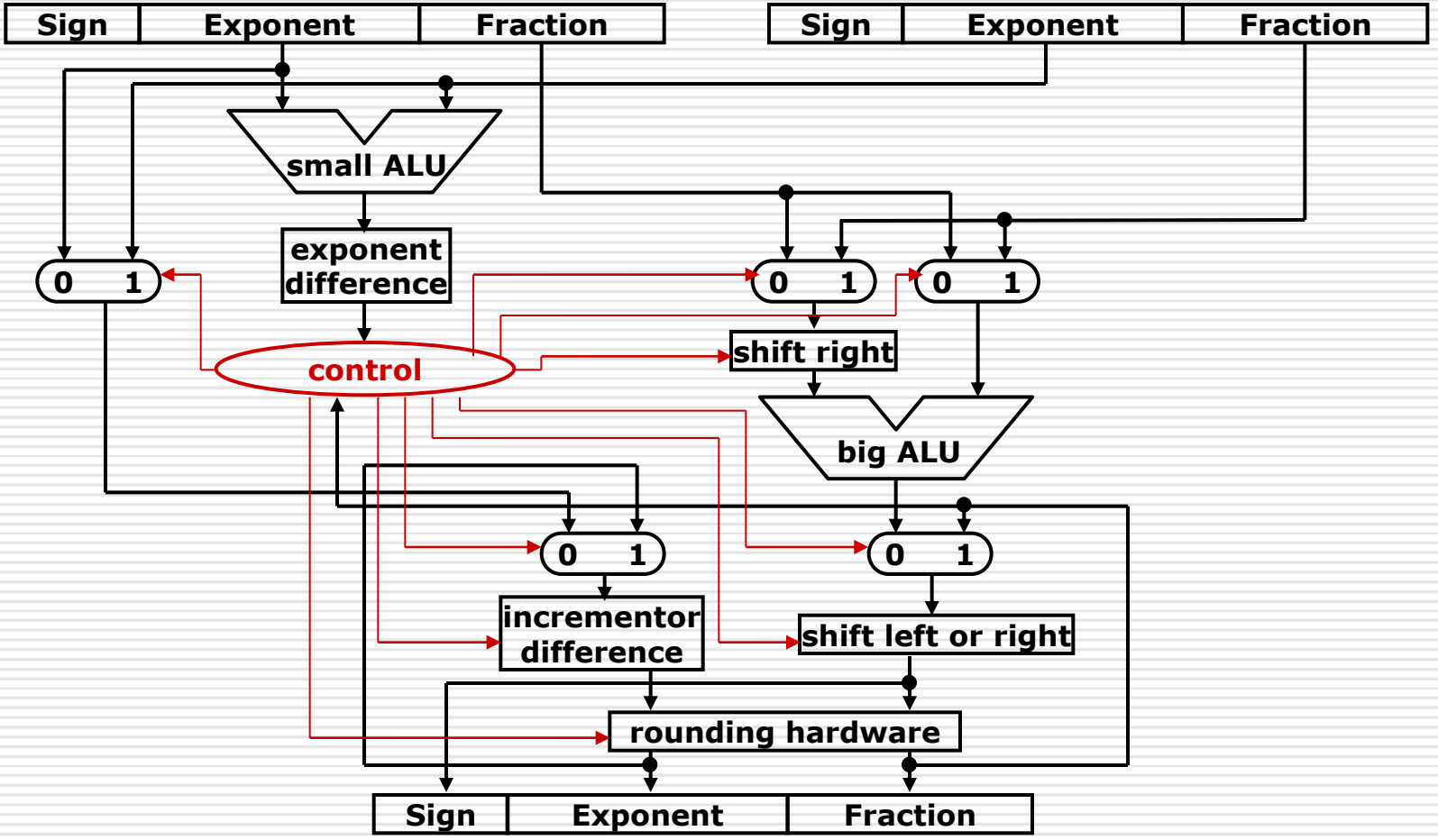
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

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

Floating-Point Addition



Floating-Point Addition



Floating-Point Multiplication

- $1.110 \times 10^{10} \times 9.200 \times 10^{-5} = ?$

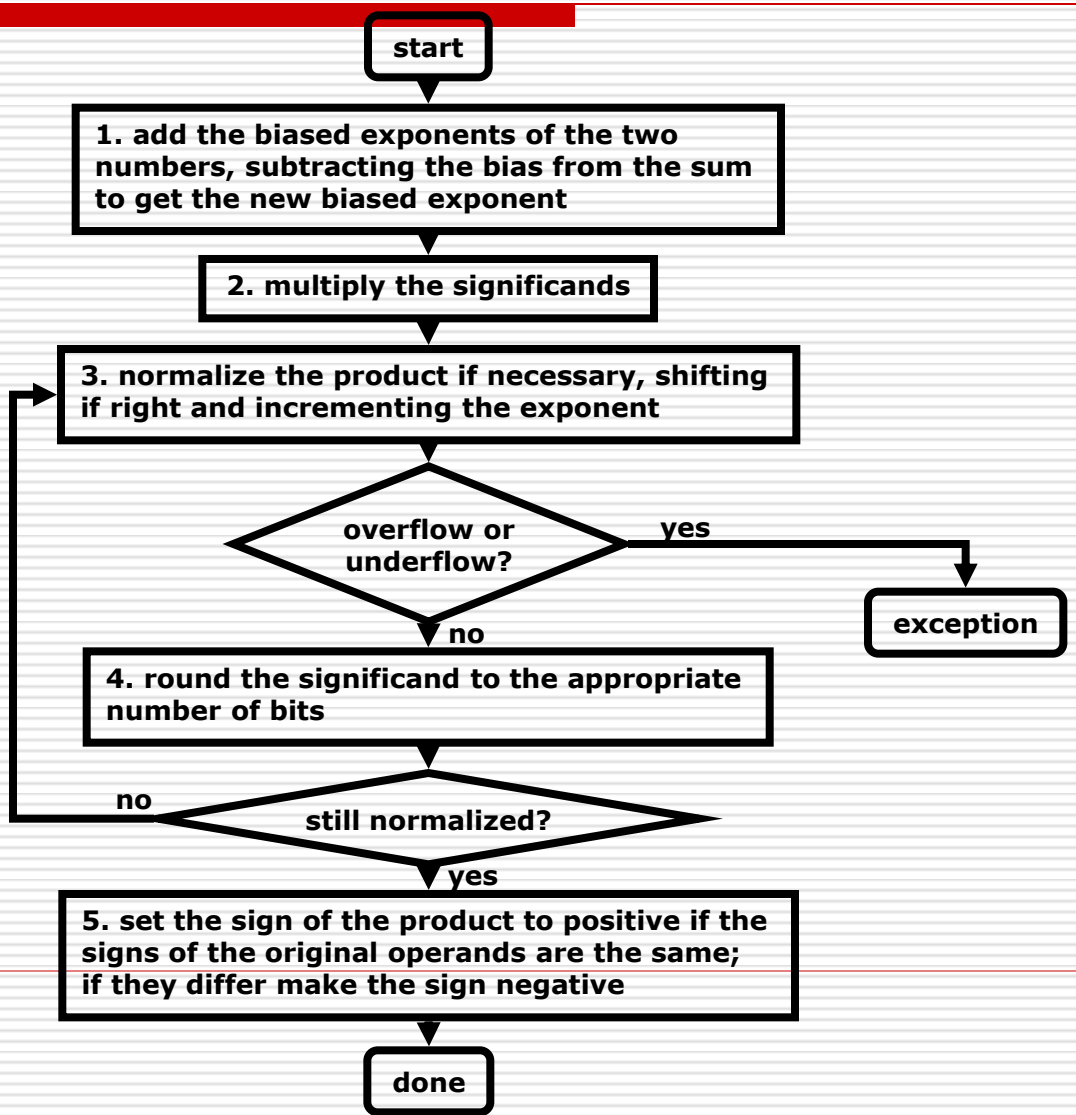
1. addition of exponents
   $10 + -5 =$ **5**
2. multiplication of significands
   $1.110 \times 9.200 =$ **10.212000**
3. $10.212 \times 10^{5} =$ **1.0212×10^{6}**
4. $1.0212 \times 10^{6} =$ **1.021×10^{6}**
5. determination of sign
   +

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
   - $-1 + -2 = -3$
2. Multiply significands
   - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
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
   - $-1.110_2 \times 2^{-3} = -0.21875$

Floating-Point Multiplication



Rounding with Guard Digits

- $2.56 \times 10^0 + 2.34 \times 10^2 = ?$

- with Guard Digits (2 extra bits)
  - ⇨ $0.0256 \times 10^2 + 2.3400 \times 10^2$
  - $= 2.3656 \times 10^2 =$ **2.37×10^2**
- without Guard Digits
  - ⇨ $0.02 \times 10^2 + 2.34 \times 10^2$
  - **$= 2.36 \times 10^2$**

FP Instructions in MIPS

- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
- 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
  - `lwc1, ldc1, swc1, sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s, sub.s, mul.s, div.s`
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d, sub.d, mul.d, div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s, c.`*xx*`.d` (*xx* is `eq, lt, le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t, bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

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

  - `fahr` in `$f12`, result in `$f0`, literals in global memory space

- MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```