COMP2611 COMPUTER ORGANIZATION
ARITHMETIC FOR COMPUTERS

Major Goals

- Revisit addition & subtraction for 2's complement numbers
- Explain the construction of a 32-bit arithmetic logic unit (ALU)
- Show algorithms and implementations of multiplication and division
- Demonstrate floating-point arithmetic operations (optional)

2’S COMPLEMENT ARITHMETIC

Addition and Subtraction

- **Addition**
  - Bits are added bit by bit **from right to left**, with **carries** passed to the next bit position to the left
- **Subtraction**
  - **Subtraction uses addition**
  - The subtrahend is negated and then added by 1, before being added to the minuend
- **Overflow**
  - The result is too large to fit (e.g. 32 bits in MIPS)

Example

- **Addition (7 + 6 = 13):**

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = \quad 7_{10} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 & = \quad 6_{10} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_2 & = \ 13_{10} \end{array}$$

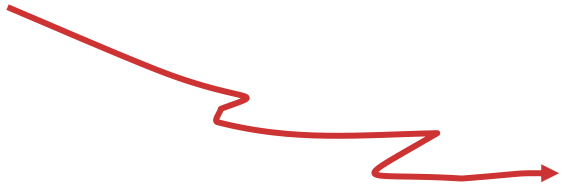
- **Subtraction (7 - 6 = 1):**

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = \quad 7_{10} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 & = \ -6_{10} \\ \hline = & \boxed{1}\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 & = \quad 1_{10} \end{array}$$

Example (cont.)

- **Addition (1073741824 + 1073741824 = 2147483648):**

$$
\begin{array}{rl}
 & 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 1073741824_{10} \\
+ & 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 1073741824_{10} \\
\hline
= & 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 \neq 2147483648_{10}
\end{array}
$$



In 2's complement the MSb is a sign bit

then, it means -2147483648_{10}

Overflow Detection

Addition (X + Y)

- **No overflow occurs when:**
  - X and Y are of different signs
- **Overflow occurs when:**
  - X and Y are of the same sign
  - But, X + Y is represented in a different sign

Subtraction (X - Y)

- **No overflow occurs when:**
  - X and Y are of the same sign
- **Overflow occurs when:**
  - X and Y are of different signs
  - But, X - Y is represented in a different sign from X

- **Overflow detection**

Operation	Sign Bit of X	Sign Bit of Y	Sign Bit of Result
X + Y	0	0	1
X + Y	1	1	0
X – Y	0	1	1
X – Y	1	0	0

Unsigned and Signed Integers

- **MIPS has a separate format for unsigned and signed integers**
- **Signed integers**
  - **negative** or **non-negative** integers, e.g. `int` in C/C++
  - saved as 32-bit words in 2's complement with the MSB reserved for sign
  - Smallest signed integer: 0x800000000 = $-2{,}147{,}483{,}648_{10}$
  - Largest signed integer: 0x7FFFFFFFF = $2{,}147{,}483{,}647_{10}$
- **Unsigned integers**
  - **non-negative** integers, e.g. `unsigned int` in C/C++
  - saved as 32-bit words
  - Smallest unsigned integer is 0x00000000 = 0_{10}
  - Largest unsigned integer is 0xFFFFFFFF = $4{,}294{,}967{,}295_{10}$

Signed vs. Unsigned Comparison

- `$s0` 1111 1111 1111 1111 1111 1111 1111 1111_2
- `$s1` 0000 0000 0000 0000 0000 0000 0000 0001_2

- What are the values in registers `$t0` and `$t1` in the examples below?
  - `slt $t0, $s0, $s1` # signed comparison

    `$s0` = -1_{10}, `$s1` = 1_{10}, `$t0` = 1
  - `sltu $t1, $s0, $s1` # unsigned comparison

    `$s0` = 4294967295_{10}, `$s1` = 1_{10}, `$t1` = 0

“Unsigned” MIPS Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add ~~unsigned~~	addu $s1, $s2, $s3	$s1 = $s2 + $s3	binary addition with overflow ignored
	subtract ~~unsigned~~	subu $s1, $s2, $s3	$s1 = $s2 - $s3	binary subtraction with overflow ignored
	add immediate ~~unsigned~~	addiu $s1, $s2, immd	$s1 = $s2 + sign-extend(immd)	binary addition, with the 16-bit immediate value sign-extended to 32 bits, overflow ignored
Comparison	set less than unsigned	sltu $s1, $s2, $s3	$s1 = 1 if $s2 < $s3, 0 otherwise	$s2 and $s3 hold unsigned integers
	set less than immediate unsigned	sltiu $s1, $s2, immd	$s1 = 1 if $s2 <sign-extend(immd), 0 otherwise	the 16-bit immediate value sign-extended to 32-bits; comparison on two unsigned integers
Data transfer	load half byte unsigned	lhu $s1, immd($s2)	Memory address: $s2 + sign-extended(immd) Load 2 bytes and then zero-extend to 32-bit, store it to $s1	
	load byte unsigned	lbu $s1, immd($s2)	Memory address: $s2 + sign-extended(immd) Load a single byte and then zero-extend to 32-bit, store it to $s1	

‘Unsigned’?

- The MIPS32 Instruction Set states that the word ‘unsigned’ as part of add and subtract instructions, is a misnomer.
- The difference between signed and unsigned versions of instructions is not a sign extension of the operands, but controls whether a trap is executed on overflow (i.e. `add`) or an overflow is ignored (i.e. `addu`).
- An immediate operand CONST to these instructions is always sign-extended.

Effects of Overflow

MIPS detects overflow with an exception (also called an interrupt)

- **Exceptions occur when unscheduled events disrupt program execution**
- **Some instructions are designed to cause exceptions on overflow**
  - e.g. **`add`**, **`addi`** and **`sub`** cause exceptions on overflow
  - But, **`addu`**, **`addiu`** and **`subu`** do not cause exceptions on overflow;

    programmers are responsible for using them correctly

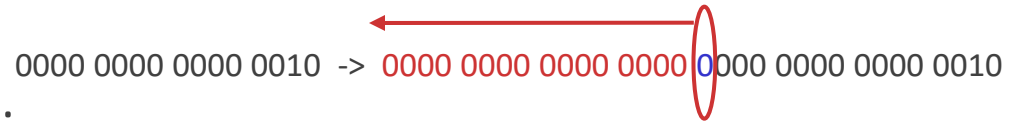
When an overflow exception occurs

- **Control jumps to a predefined address (code) to handle the exception**
- **The interrupted address is saved to EPC for possible resumption**
  - **EPC** = **exception program counter**; a special register
  - MIPS software return to the offending instruction via jump register

Revisit: Zero Extension and Sign Extension

- Operands in instruction may have mis-matched sizes
- All immediate values in the following I-format instructions are encoded with 16 bits
  - `addi $t0, $s0, -5`
  - `ori $t0, $s0, 0xFB32`
  - `lb $t0, 0($s0)`
- Need conversion from n-bit binary numbers into m-bit numbers (m > n)

- **Sign extension:** Fill the leftmost bits (n-th ~ (m-1)-th) with the sign bit
  - 2 (16 bits -> 32 bits):



  - -2 (16 bits -> 32 bits):

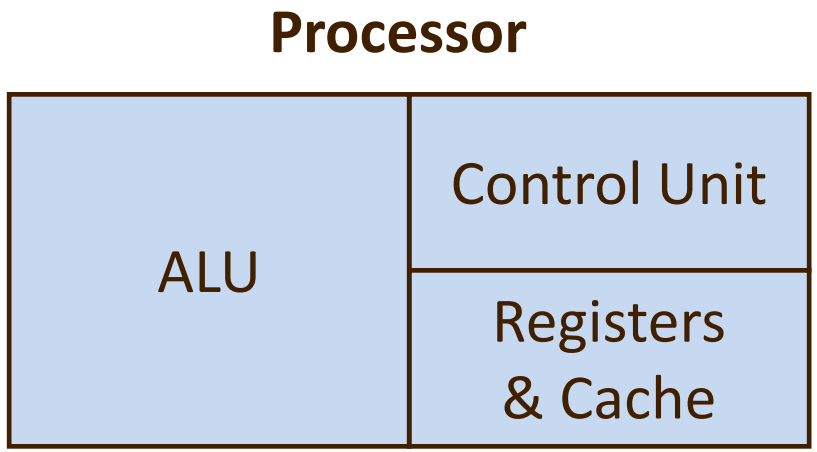
  - `addi, addiu, slti, lh, lb`

- **Zero extension:** Pad the leftmost bits (n-th ~ (m-1)-th) with 0
  - `andi, ori, lhu, lbu`

ARITHMETIC LOGIC UNIT

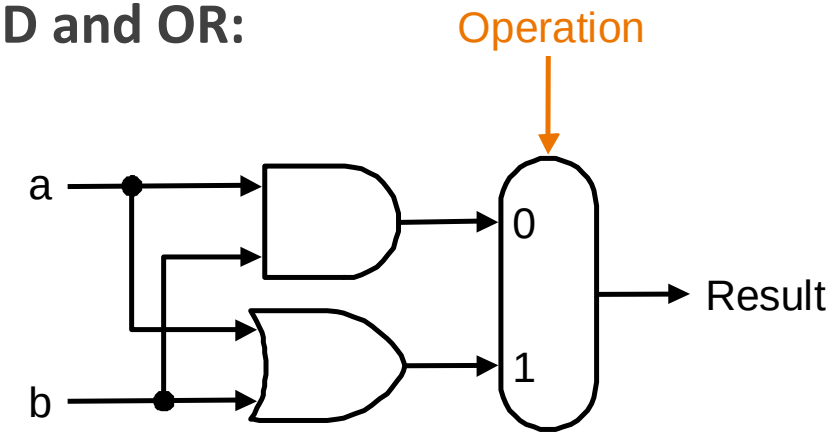
Constructing an Arithmetic Logic Unit

- The **arithmetic logic unit** (**ALU**) of a computer is the hardware component that performs:
  - **Arithmetic operations** (like addition and subtraction)
  - **Logical operations** (like AND and OR)



Constructing an Arithmetic Logic Unit (cont'd)

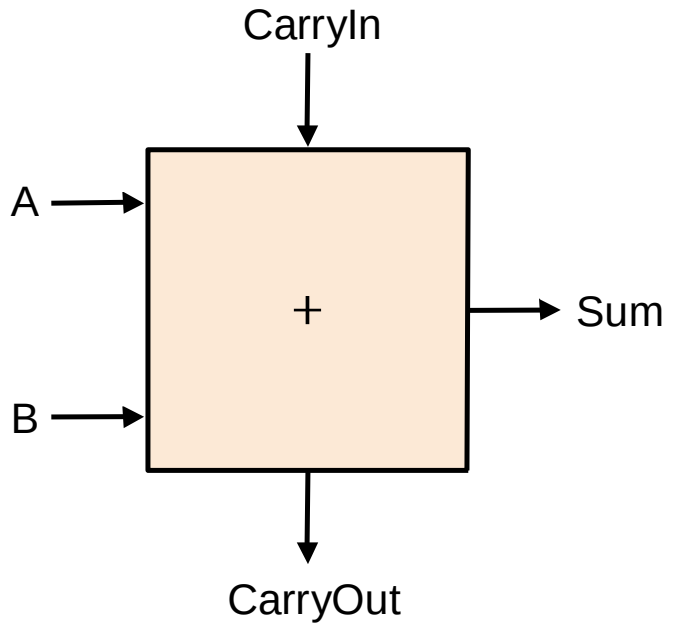
- Since a word in MIPS is 32-bit wide, we need a 32-bit ALU
- Ideally, we can build a 32-bit ALU by connecting 32 1-bit ALUs together (each of them takes care of the operation on one bit position)
- **1-bit** logical unit for AND and OR:

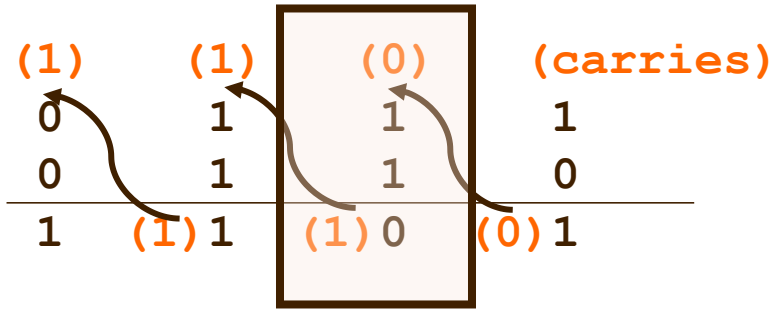


  - A multiplexor selects the appropriate result depending on the operation specified

1-Bit Full Adder

- **An adder must have**
  - Two inputs (bits) for the operands
  - A single-bit output for the sum
- **Also, must have a second output to pass on the carry, called carry-out**
  - Carry-out becomes the carry-in to the neighbouring adder
- **1-bit full adder is also called a (3, 2) adder (3 inputs and 2 outputs)**





Truth Table and Logic Equations for 1-Bit Adder

- **Truth table**

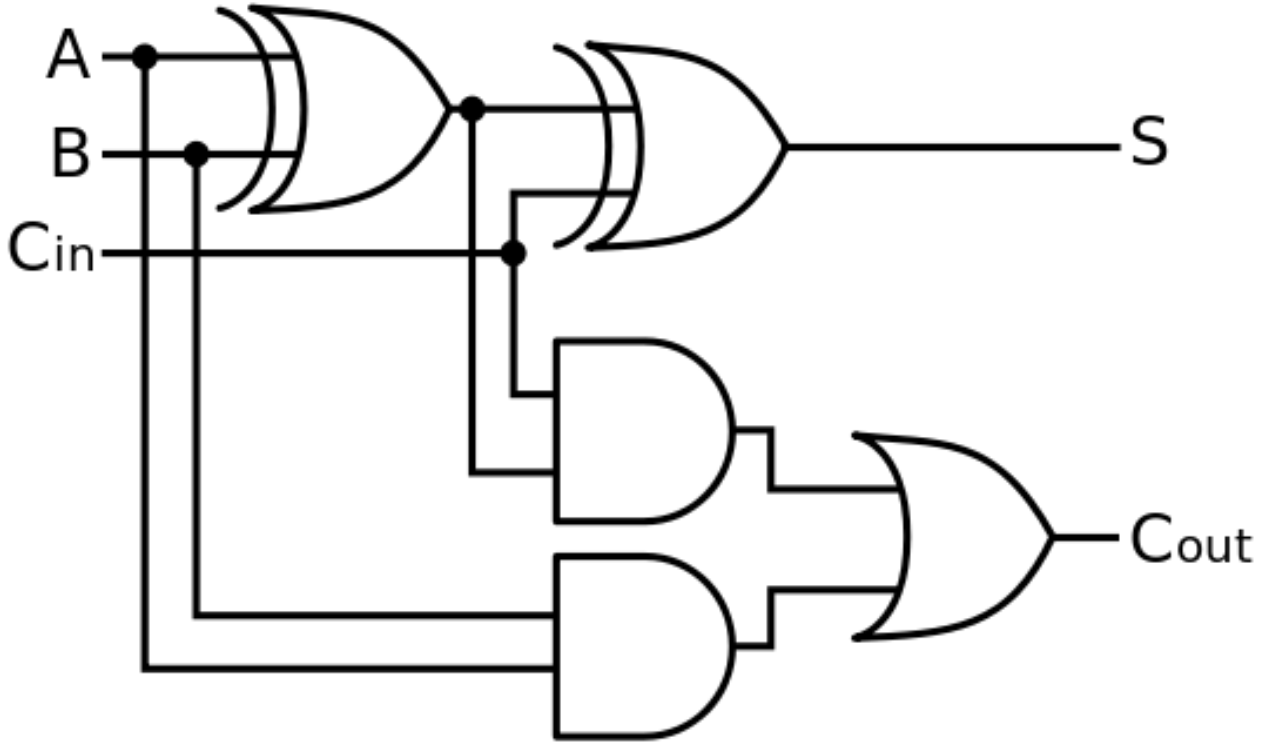
Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	SumOut	
0	0	0	0	0	$0 + 0 + 0 = 00_2$
0	0	1	0	1	$0 + 0 + 1 = 01_2$
0	1	0	0	1	$0 + 1 + 0 = 01_2$
0	1	1	1	0	$0 + 1 + 1 = 10_2$
1	0	0	0	1	$1 + 0 + 0 = 01_2$
1	0	1	1	0	$1 + 0 + 1 = 10_2$
1	1	0	1	0	$1 + 1 + 0 = 10_2$
1	1	1	1	1	$1 + 1 + 1 = 11_2$

- **Logic equations**

$$\begin{aligned} CarryOut &= (\bar{a} \cdot b \cdot CarryIn) + (a \cdot \bar{b} \cdot CarryIn) + (a \cdot b \cdot \overline{CarryIn}) + (a \cdot b \cdot CarryIn) \\ &= (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b) \end{aligned}$$

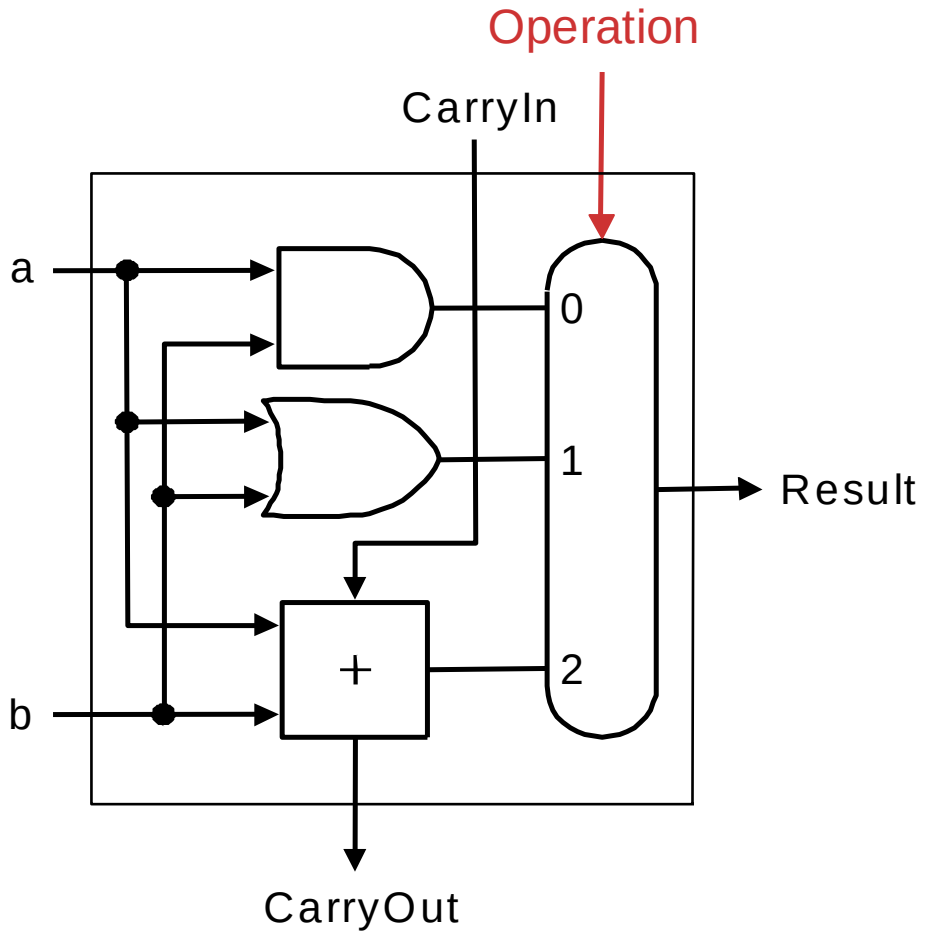
$$\begin{aligned} SumOut = &(a \cdot \bar{b} \cdot \overline{CarryIn}) + (\bar{a} \cdot b \cdot \overline{CarryIn}) + (\bar{a} \cdot \bar{b} \cdot CarryIn) \\ &+ (a \cdot b \cdot CarryIn) \end{aligned}$$

Hardware Implementation of 1-Bit Adder



1-Bit ALU (AND, OR, and Addition)

- **3 in 1 building block**
  - Use the Operation bits to decide what result to push out
  - Operation = 0, do AND
  - Operation = 1, do OR
  - Operation = 2, do addition



Subtraction

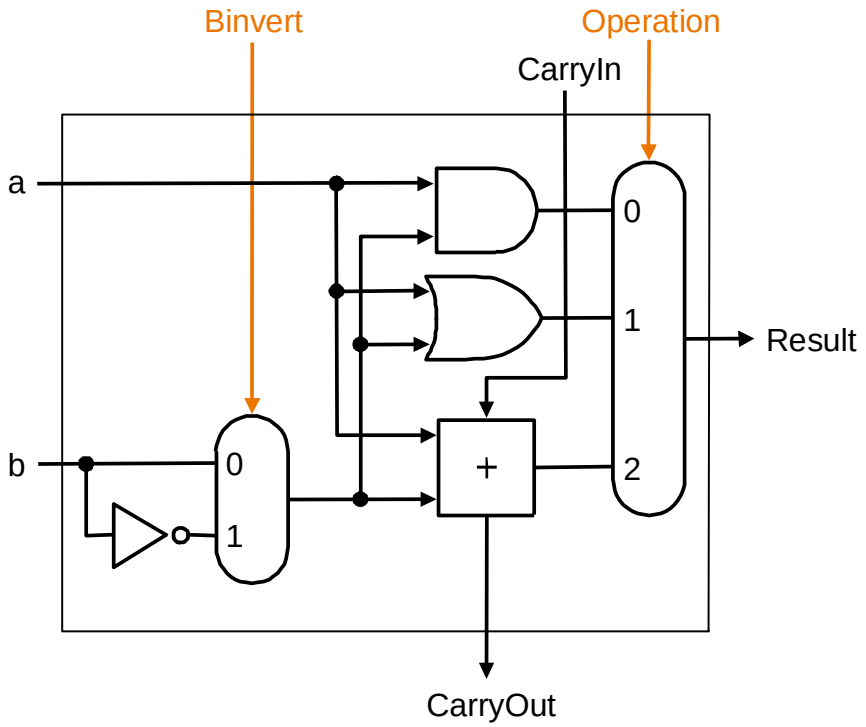
- **Subtraction is the same as adding the negated operand**
- **By doing so, an adder can be used for both addition and subtraction**

- **A 2:1 multiplexor is used to choose between**
  - an operand (for **addition**) and
  - its negative version (for **subtraction**)

- **Shortcut for negating a 2's complement number**
  - Invert each bit (to get the 1's complement representation)
  - Add 1: Obtained by setting the ALU0's carry bit to 1

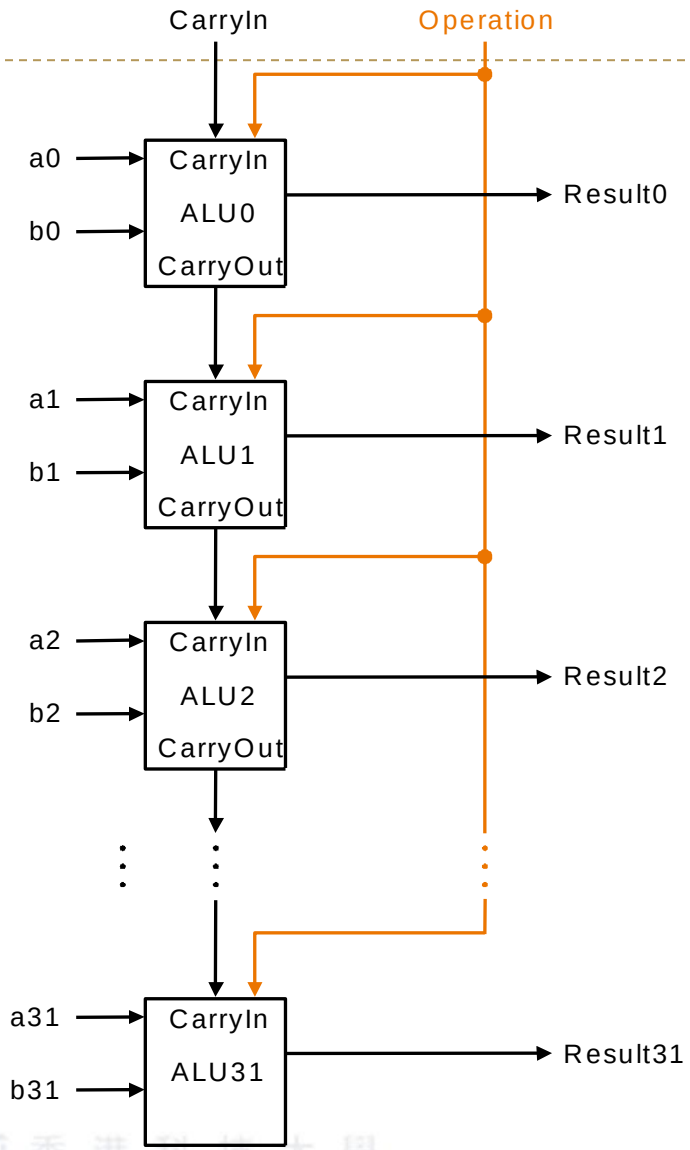
1-Bit ALU (AND, OR, Addition, and Subtraction)

- **To execute a – b we can execute a + (-b)**
  - Binvert: the selector input of a multiplexor to choose between addition and subtraction
- **To form a 32-bit ALU, connect 32 of these 1-bit ALUs**
  - To negate b we must invert it and add 1 (2's complement), so we must Set CarryIn input of the least significant bit (ALU0) to 1 for subtraction



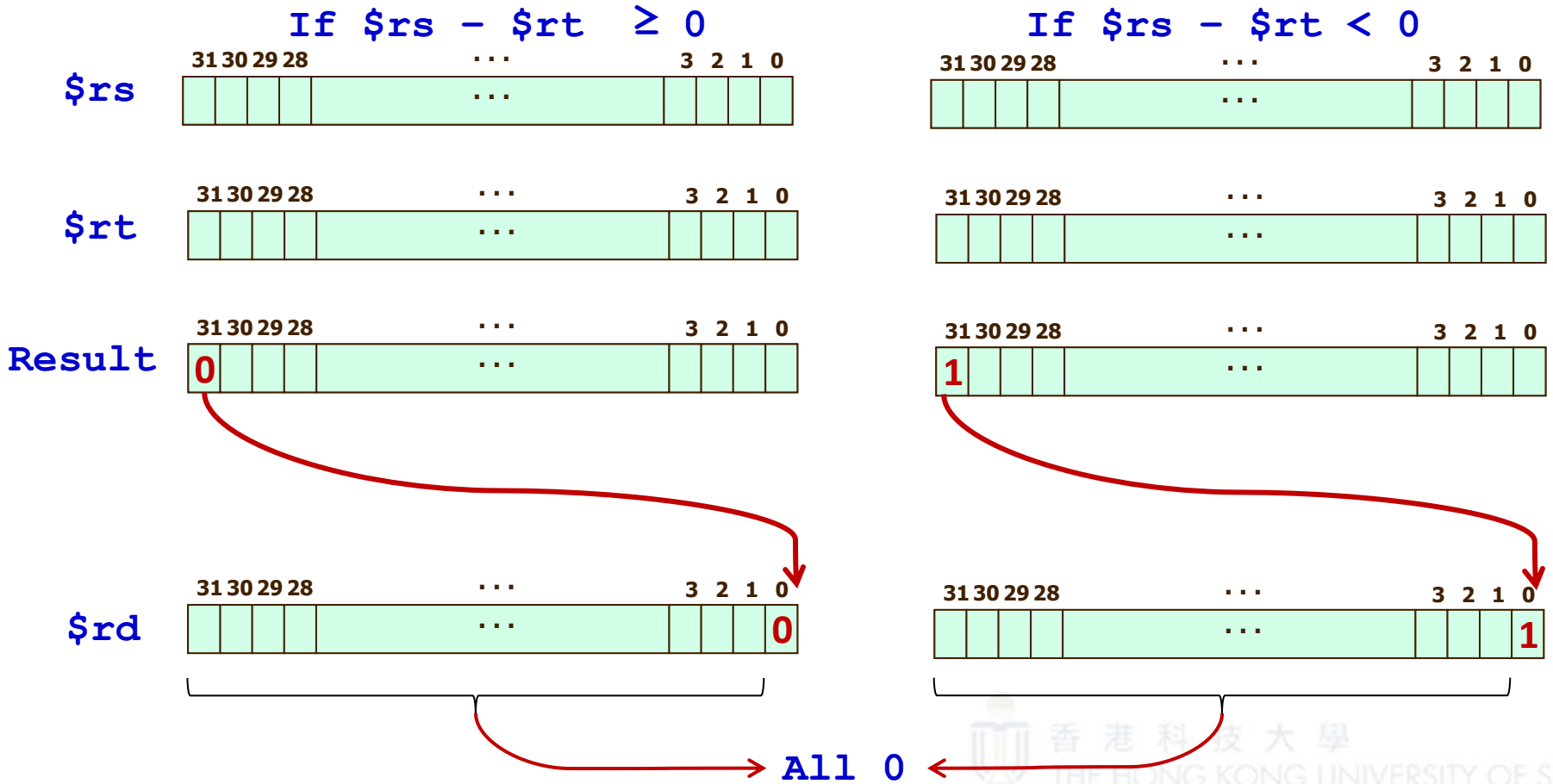
32-Bit ALU

- **Ripple carry organization of a 32-bit ALU constructed from 32 1-bit ALUs:**
  - A single carry out of the least significant bit (Result0) could ripple all the way through the adders, causing a carry out of the most significant bit (Result31)
  - There exist more efficient implementations (based on the **carry lookahead** idea to be explained later)

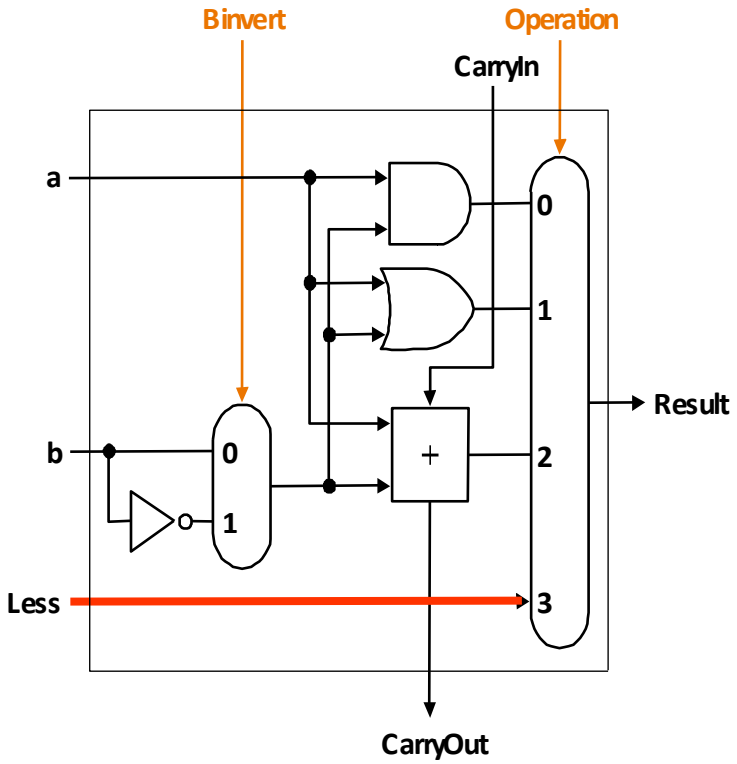


Tailoring the ALU for `slt`

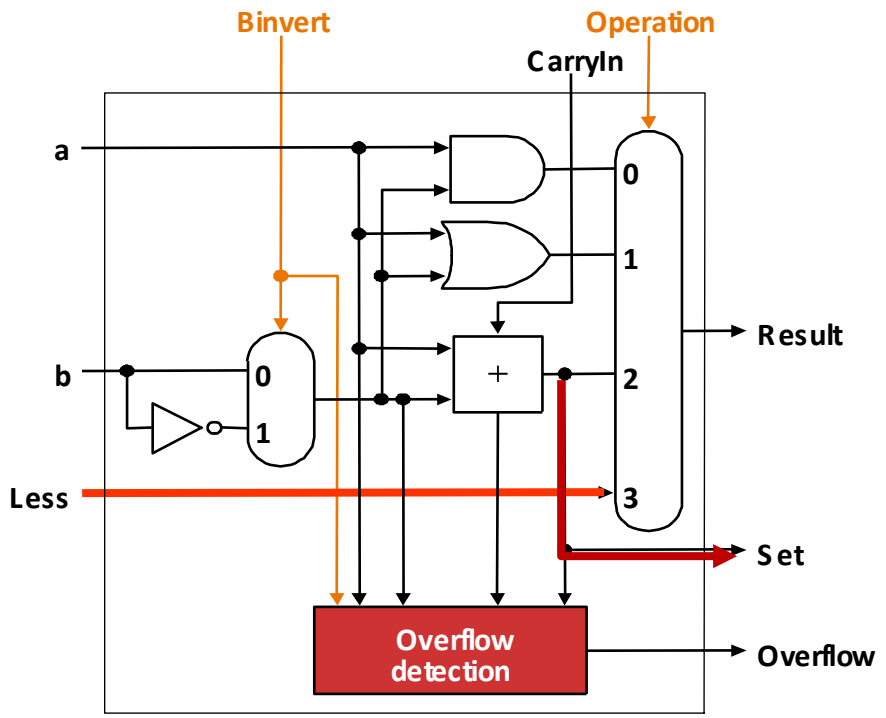
- `slt $rd, $rs, $rt`, the comparison is equivalent to testing if `($rs - $rt) < 0`
- `$rd` is either 0 or 1 (i.e. bit 1 to 31 are always 0, bit 0 is 0 or 1)



Tailoring the ALU for `slt` (cont'd)



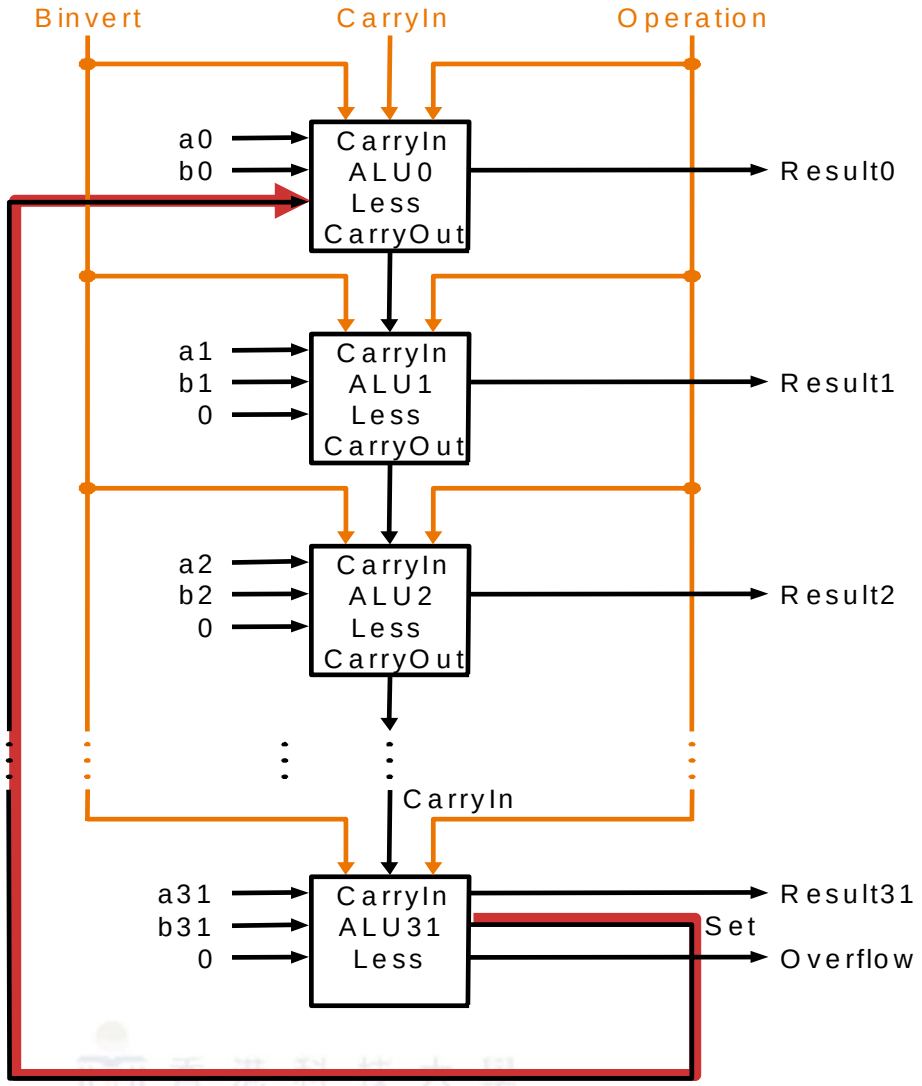
1-bit ALU for bits 0 to 30



1-bit ALU for the MSb (bit 31)

32-Bit ALU with `add`, `sub`, `and`, `or`, `slt`

- The "set" signal is the MSb of the result of the subtraction, A – B
- It is passed to LSB
- Result0 will equal to this "set" signal when operation = 3 (which means `slt` instruction is being executed)



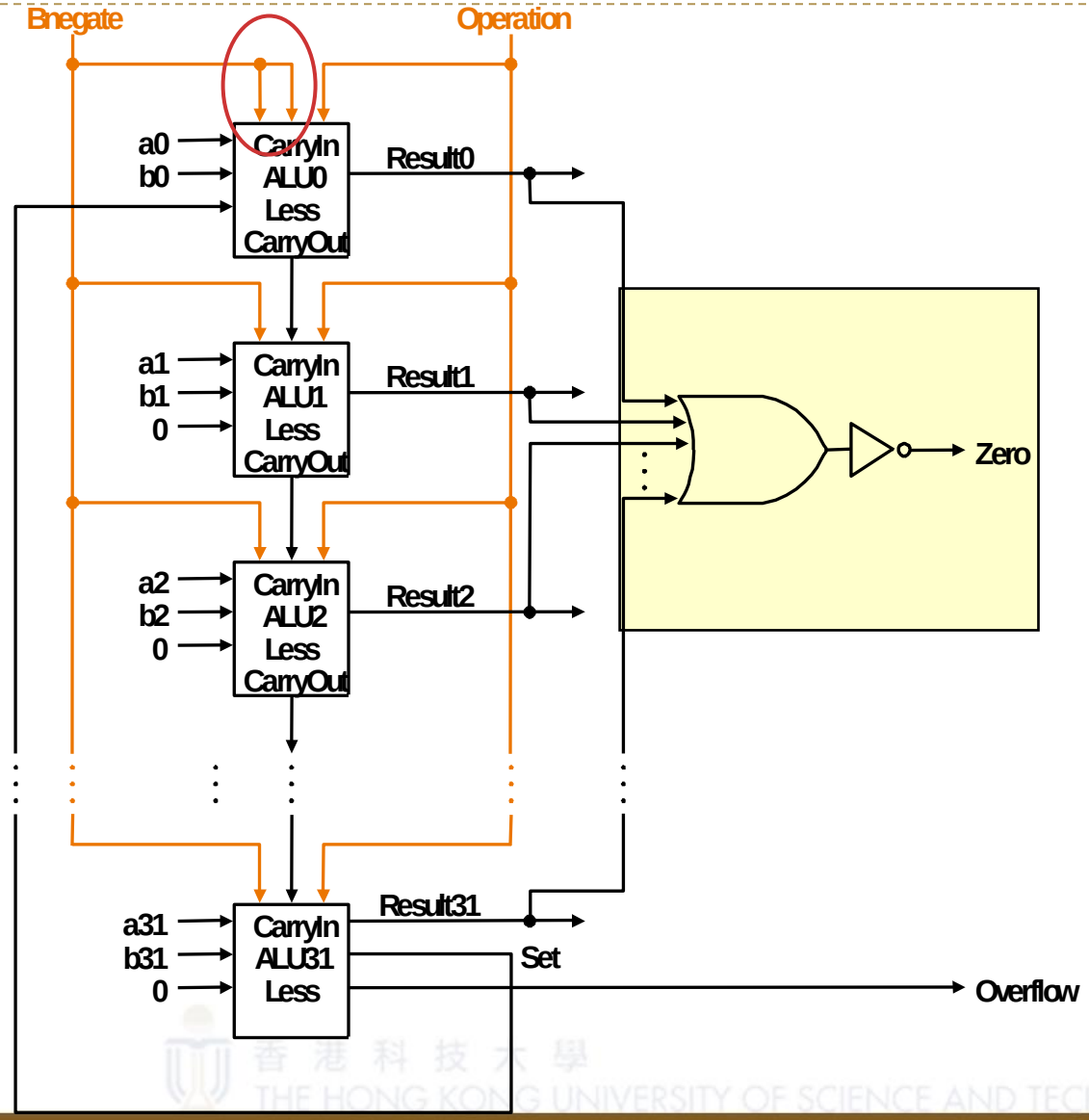
Tailoring the ALU for `beq`

- `beq $rs, $rt, L,` the equal comparison is equivalent to testing if `$rs – $rt == 0`
- If `$rs – $rt` is equal to 0, all bits of the output are 0
- Otherwise, at least one of them is non 0

32-Bit ALU with `add`, `sub`, `and`, `or`, `slt` ,`beq`

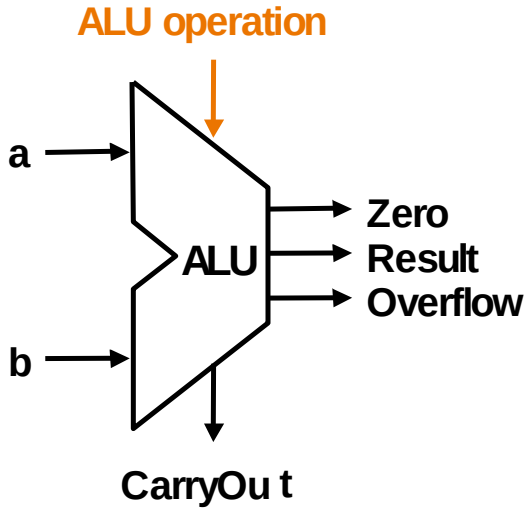
- Finally, this adds a zero detector

- For addition and AND/OR operations both `Bnegate` and `CarryIn` are 0 and for subtract, they are both 1 so we combine them into a single line



Universal Representation

- Knowing what is exactly inside a 32-bits ALU, from now on we will use the universal symbol for a complete ALU as follows:



ALU Control lines	Operation
000	AND
001	OR
010	ADD
110	SUB
111	SLT

Carry Lookahead (optional)

- **Using the ripple carry adder, the carry has to propagate from the LSb to the MSb in a sequential manner, passing through all the 32 1-bit adders one at a time. SLOW for time-critical hardware!**

- **Key idea behind fast carry schemes without the ripple effect:**

$$\text{CarryIn2} = (b1 \cdot \text{CarryIn1}) + (a1 \cdot \text{CarryIn1}) + (a1 \cdot b1)$$
$$\text{CarryIn1} = (b0 \cdot \text{CarryIn0}) + (a0 \cdot \text{CarryIn0}) + (a0 \cdot b0)$$

  - Substituting the latter into the former, we have:

$$\begin{aligned}\text{CarryIn2} = {} & (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot \text{CarryIn0}) + (a1 \cdot b0 \cdot \text{CarryIn0}) \\ & + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot \text{CarryIn0}) + (b1 \cdot b0 \cdot \text{CarryIn0}) \\ & + (a1 \cdot b1)\end{aligned}$$

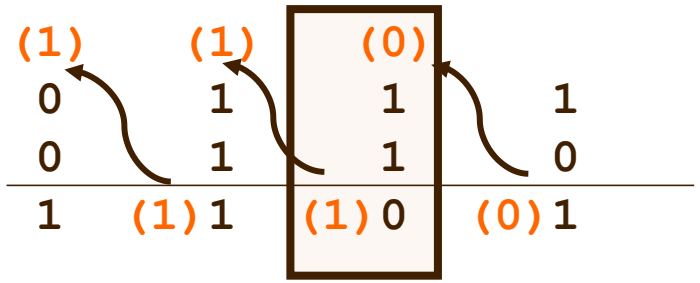
  - All other CarryIn bits can also be expressed using CarryIn0, a, b

Carry Lookahead (optional)

- A Bit position **generates** a Carry iff both inputs are 1: $G_i = a_i \cdot b_i$
- A Bit position **propagates** a Carry if exactly one input is 1: $P_i = a_i + b_i$
- CarryIn at bit `i+1` (or CarryOut at bit `i`)can be expressed as: $C_{i+1} = G_i + P_i \cdot C_i$

- **After substitution we have**

$$C_1 = G_0 + P_0 C_0$$
$$C_2 = G_1 + P_1 C_1 = G_1 + G_0 P_1 + C_0 P_0 P_1$$
$$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$$
$$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$$



- **WE can build a circuit to predict all Carries at the same time and do the additions in parallel**
- **Possible because electronic chips becoming cheaper and denser**

MULTIPLICATION

Multiplication Version 1

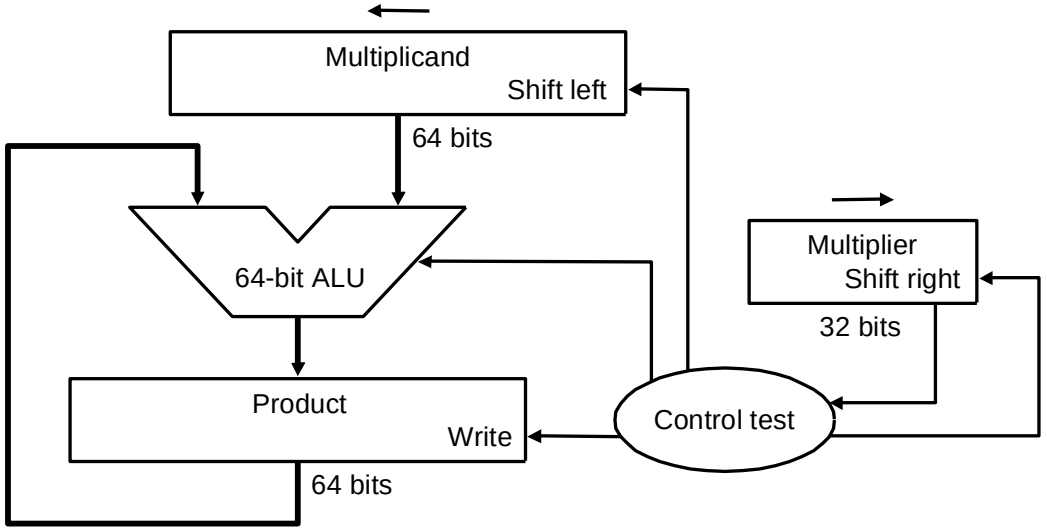
- Multiplication is much more complicated than addition and subtraction

- Paper-and-pencil example (1000_{10} x 1001_{10}):

```
Multiplicand (4-bit)          1000
Multiplier (4-bit)            1001
                              ----
                              1000
                             0000
                            0000
                           1000
                           ----
Product (8-bit)           1001000
```

- Observation
  - Suppose we limit ourselves to using only digits 0 and 1
  - If we ignore the sign bits (i.e., unsigned numbers), multiplying an N-bit multiplicand with an M-bit multiplier gives a product that is at most N+M bits long

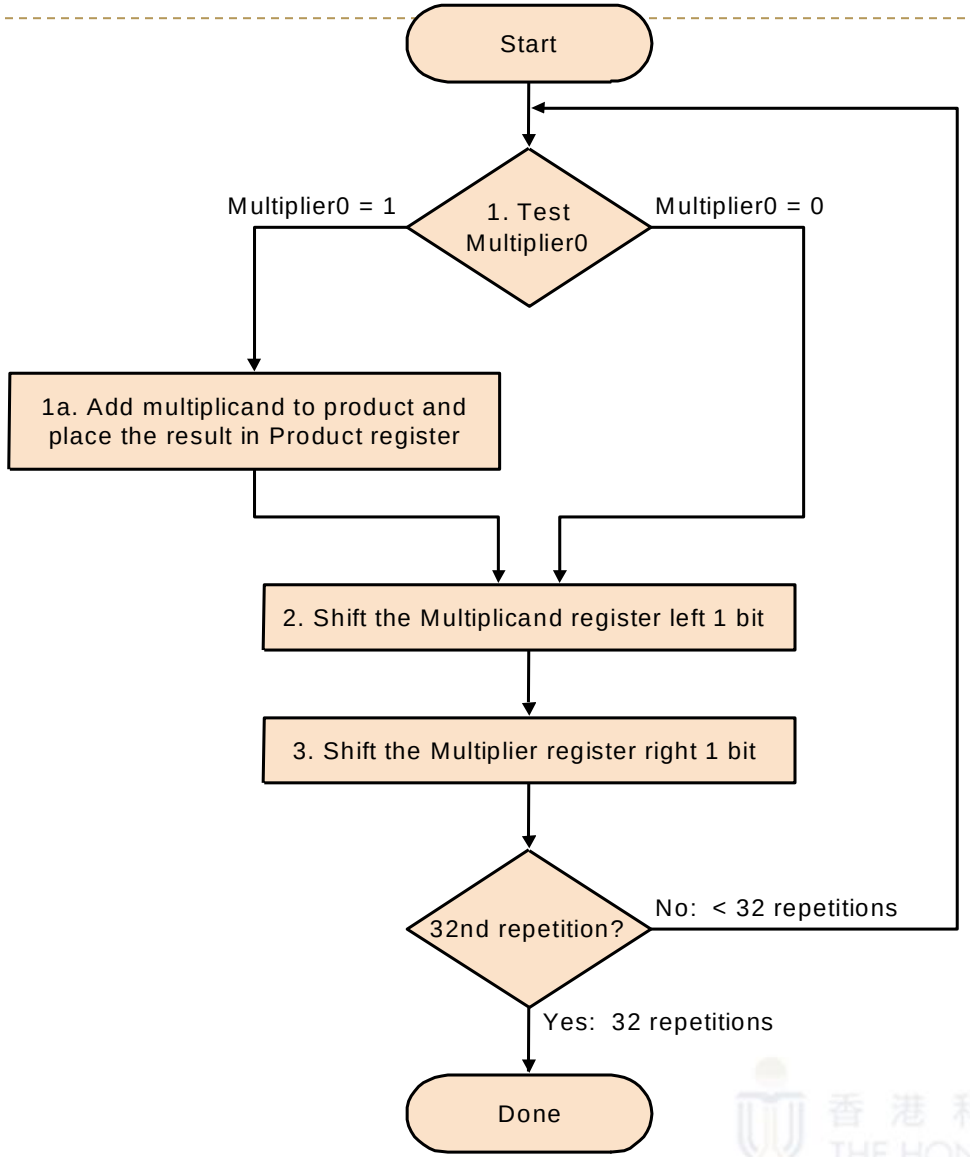
Sequential Hardware Version 1



- Follows the flow of paper-and-pencil example
- 64-bit ALU
- 64-bit multiplicand register, 64-bit product register, 32-bit multiplier register

- Operations:
  - The 32-bit multiplicand starts in the right half of the multiplicand register, and is shifted left 1 bit at each step
  - The multiplier register is shifted right 1 bit at each step
  - The product register is initialized to 0
  - Control decides when to shift the multiplicand and multiplier registers and when to write new values into the product register

Algorithm Version 1



Example Version 1

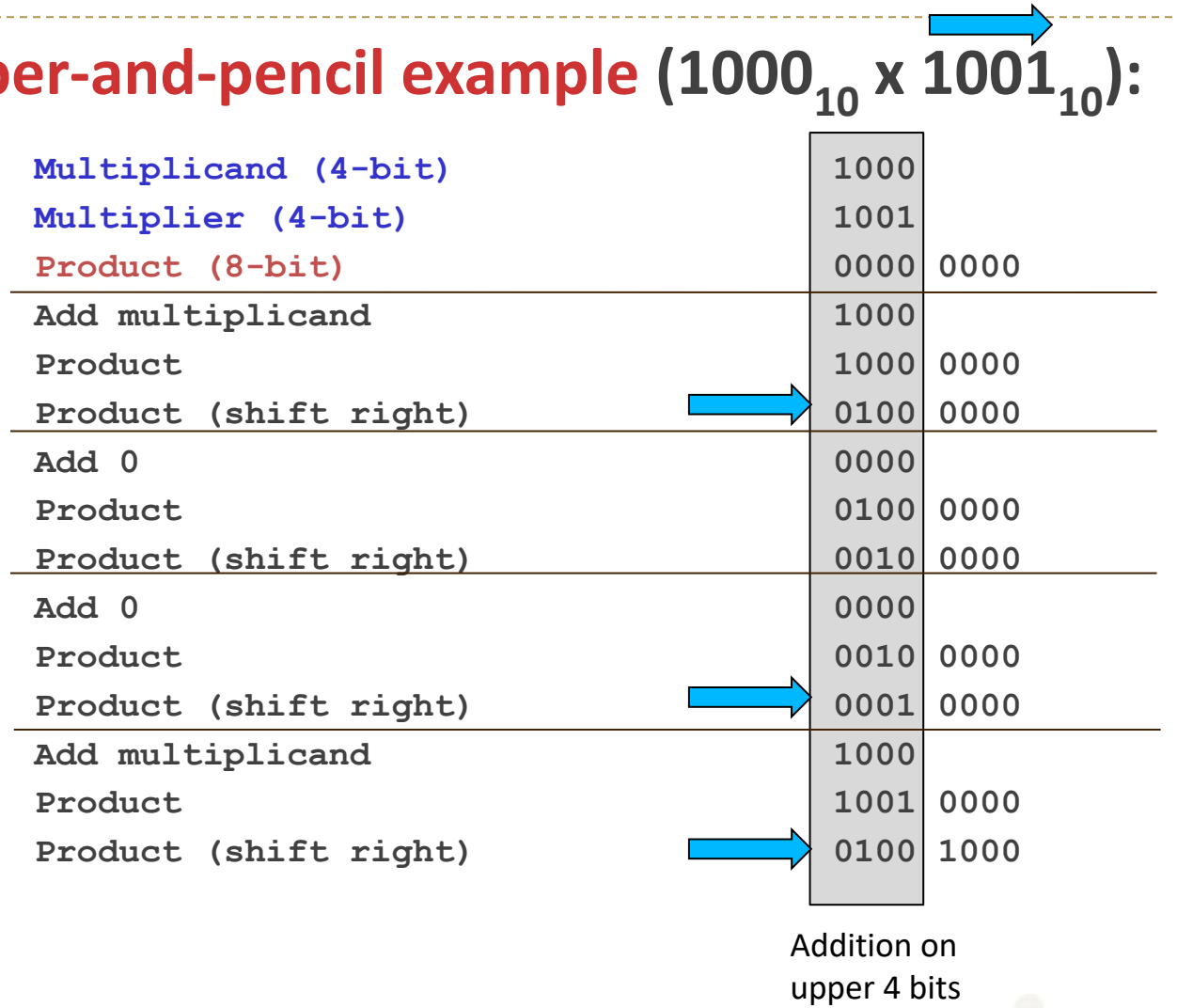
	Step	Multiplier	Multiplicand	Product
0	Initial values	**0011**	0000 **0010**	**0000 0000**
1	1a: 1→ Prod = Prod + Mcand	001**1**	0000 0010	**0000 0010**
	2: Shift left Multiplicand	0011	**0000 0100**	0000 0010
	3: Shift right Multiplier	**0001**	0000 0100	0000 0010
2	1a: 1→ Prod = Prod + Mcand	000**1**	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	000**0**	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	000**0**	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	**0010 0000**	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Observations Version 1

- **Only half of the multiplicand register contains useful bit values**
  - Reduce multiplicand register size: 64-bit → 32-bit
- **A full 64-bit ALU is wasteful and slow**
  - Because half of the adder bits add 0 to the intermediate sum
  - Reduce ALU size: 64-bit → 32-bit
- **The multiplicand is shifted left with 0s inserted in the new positions**
  - The multiplicand cannot affect the least significant bits of the product after they settle down

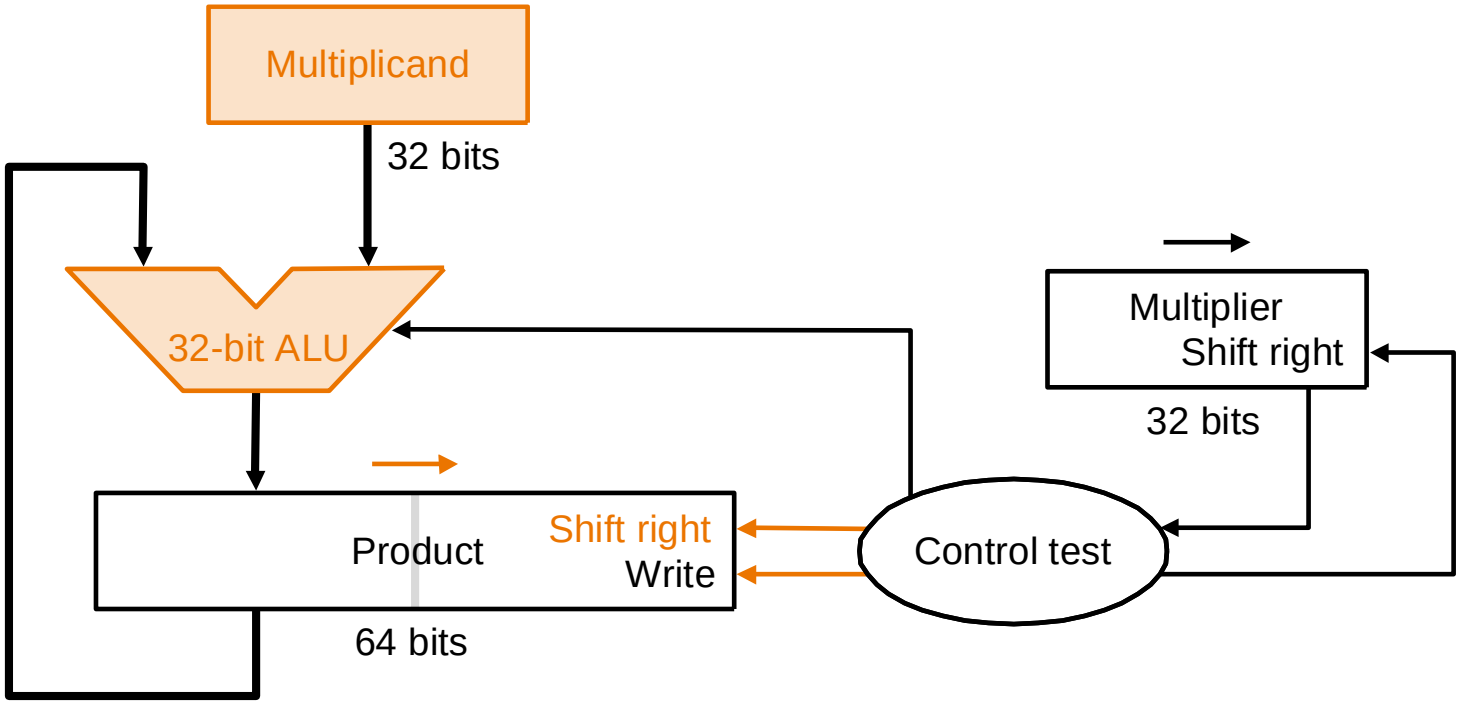
Multiplication Version 2

- **Paper-and-pencil example** (1000_{10} x 1001_{10}):



Sequential Hardware Version 2

- This version only needs a 32-bit multiplicand register and a 32-bit ALU
- This version shifts “product” instead of “multiplicand”

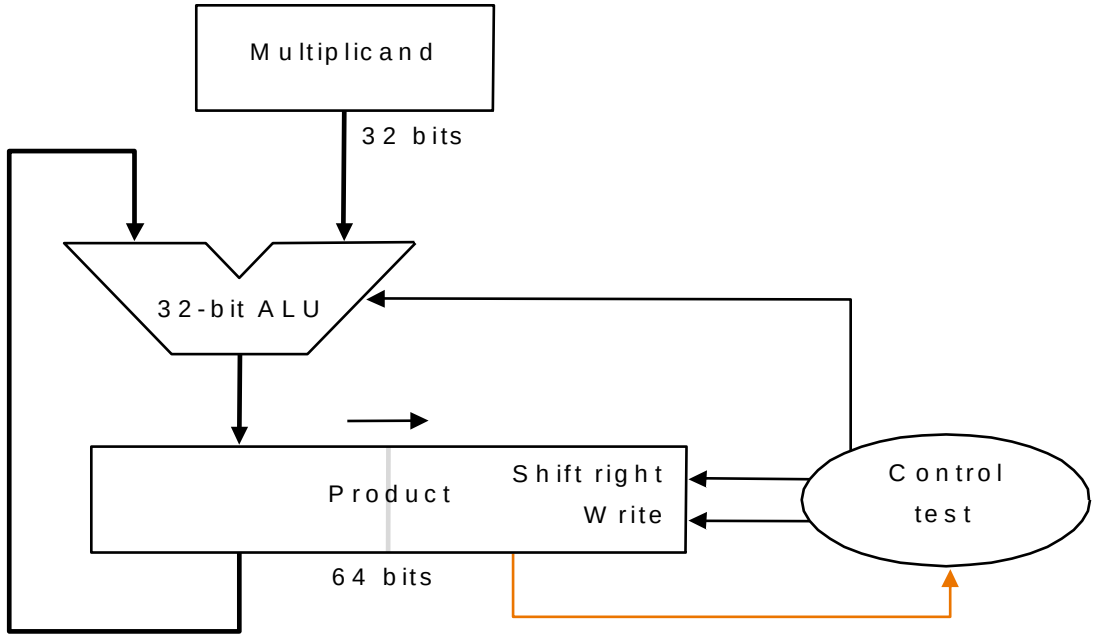


(changes made to previous version are highlighted in orange color)

Observations Version 2

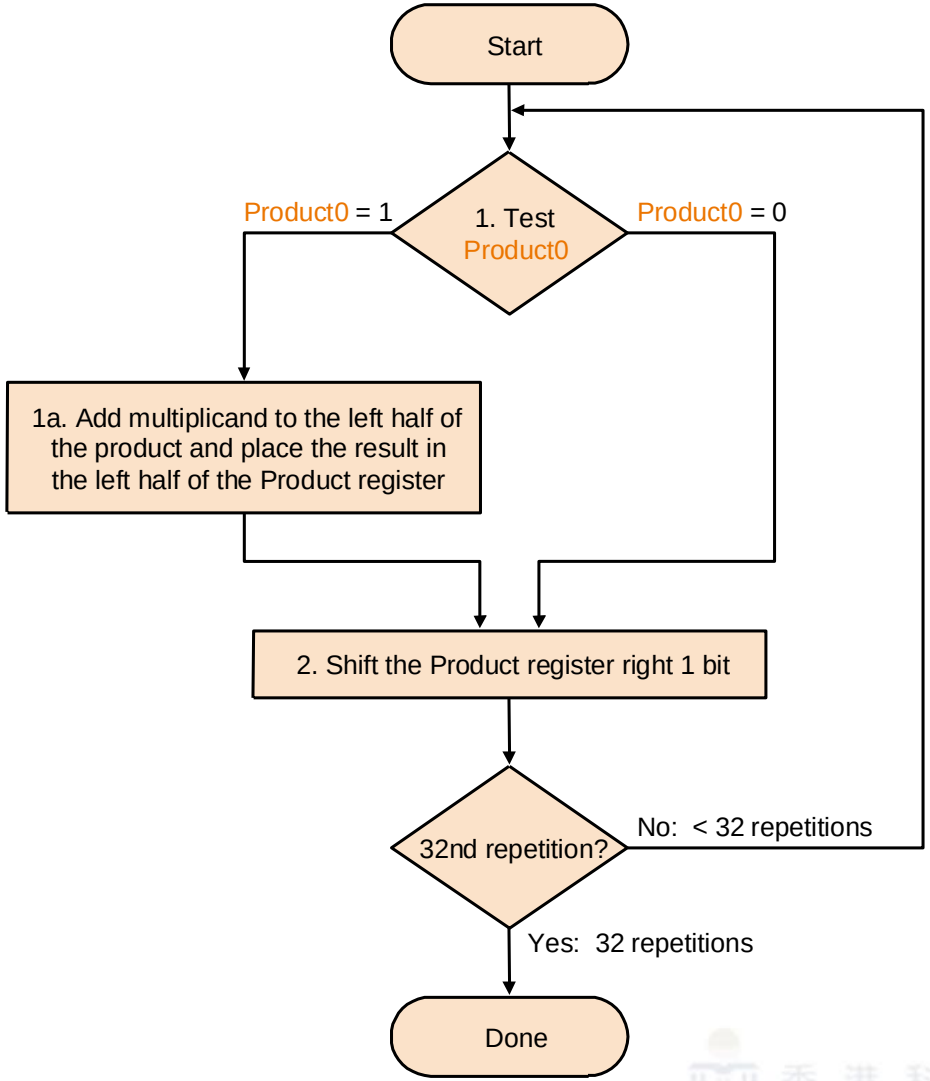
- The number of used bits in the product register increases by 1 bit at each step, from the initial value of 32 to the final value of 64
- The number of used bits in the multiplier register decreases by 1 bit at each step, from the initial value of 32 to the final value of 0
- Hence, the unused bits of the multiplier register can be used for storing part of the product
  - More specifically, the right half of the product register can be combined with the multiplier register to save hardware

Sequential Hardware Optimized Version



- **32-bit ALU**
- **Multiplicand register: 32 bits, Product register: 64 bits (right half also used for storing multiplier)**
- **Operations:**
  - The right half of the product register is initialized to the multiplier, and its left half is initialized to 0
  - The two right-shifts at each step for version 2 are combined into only a single right-shift because the product and multiplier registers have been combined

Algorithm Optimized Version



Example Optimized Version

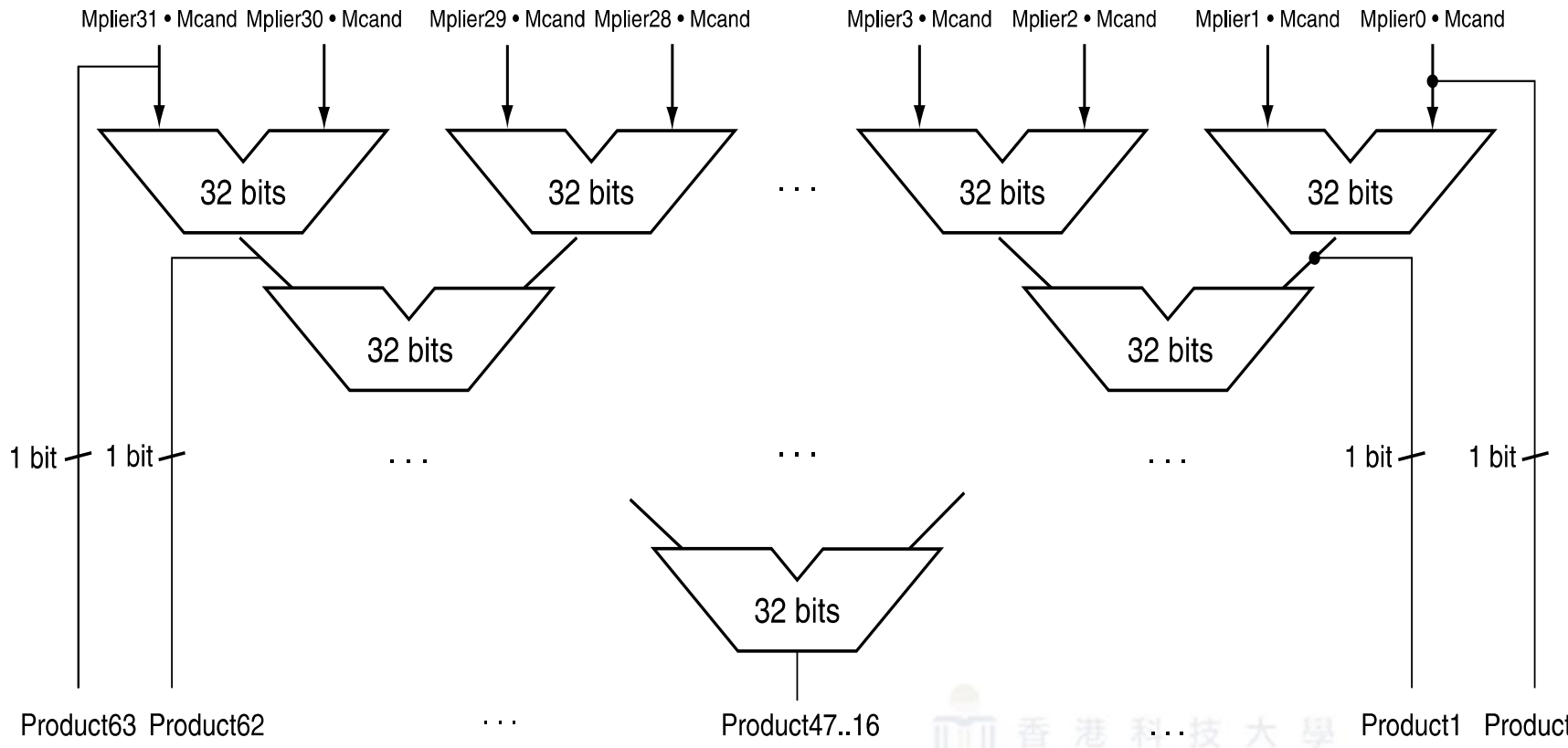
- Multiplication of two 4-bit unsigned numbers (0110 and 0011)

Multiplier

Iteration	Multiplicand (M)	Product (P)	Remark
0	0110	0000 0011	Initial state
1		0110 0011 0011 0001	Left(P) = Left(P) + M P = P >> 1
2		1001 0001 0100 1000	Left(P) = Left(P) + M P = P >> 1
3		0100 1000 0010 0100	No operation P = P >> 1
4		0010 0100 0001 0010	No operation P = P >> 1

Hardware Speedup

- Moore's law implies more and more cheaper hardware resources available
- Unroll the for loop and use 31 adders instead of single adder 32 times
- This organization minimizes delay to do 1 Multiply in 5-add time



Signed Multiplication

- If the multiplicand or multiplier is negative, we first negate it to get a positive number
- Use any one of the above methods to compute the product of two positive numbers
- The product should be negated if the original signs of the operands disagree
- Booth's algorithm: a more efficient and elegant algorithm for the multiplication of signed numbers (to be covered in tutorial)

Multiplication in MIPS

- The multiplication of two 32-bit numbers yields a 64-bit number.
- A pair of registers, **`Hi`** and **`Lo`** are used to store the product.
- The **high 32 bits** are placed in register **`Hi`**
- The **low 32 bits** are placed in register **`Lo`**

- **`mult` (multiply) and `multu` (multiply unsigned)**
  - `mult  $rs, $rt     # Hi, Lo = $rs x $rt, treat $rs, $rt as signed`
  - `multu $rs, $rt     # Hi, Lo = $rs x $rt, treat $rs, $rt as unsigned`
  - Both are R-format
  - Both ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits

- **Fetch the integer 32-bit product**
  - **`mflo`** (move from lo) `mflo $s1` `# $s1 = Lo`
  - **`mfhi`** (move from hi) `mfhi $s1` `# $s1 = Hi`
  - If the product is know to fit in 32-bit, than you can just retrieve **`Lo`**
  - **`mfhi`** can transfer **`Hi`** to a general-purpose register to test for overflow

Multiplication Overflow

- **Unsigned overflow**
  - Product too large to fit in a 32-bit unsigned word
  - There is no overflow if **`Hi`** is 0 for **`multu`**

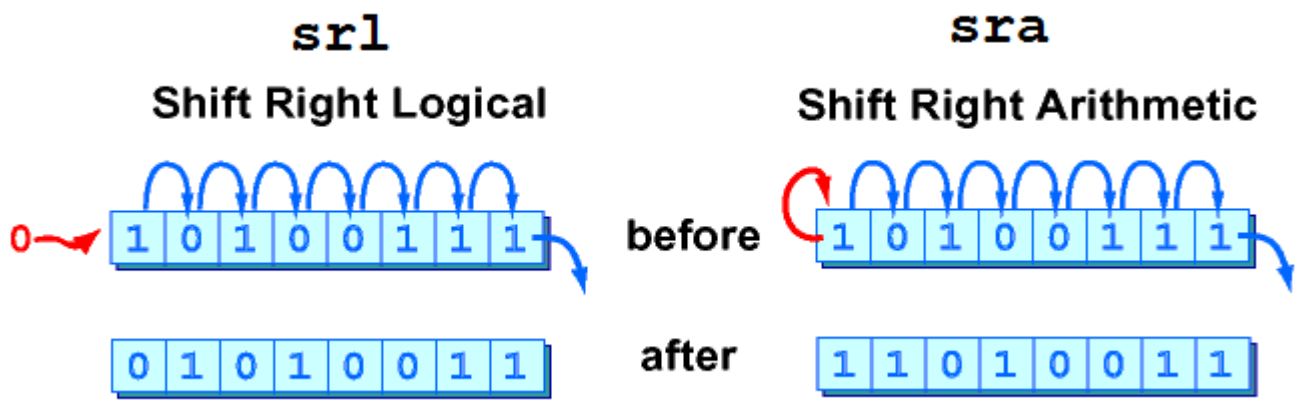
```
    multu $s0, $s1
    mfhi $at
    mflo $s2
    beq $at, $0, no_overflow
    j overflow
no_overflow:
```

- **Signed overflow**
  - Product too large to fit in a 32-bit signed word
  - There is no overflow if every bit in **`Hi`** is the same as sign bit of **`Lo`**

```
    mult $s0, $s1
    mfhi $at
    mflo $s2
    sra $t0, $s2, 31
    beq $at, $t0, no_overflow
    j overflow
no_overflow:
```

The `sra` and `srl` instructions

- The `sra` (shift right arithmetic) instruction shifts a value to the right and fills the left over bits at the left with the proper **sign value**.
- The `srl` (shift right logical) instruction shifts a value to the right and fills the bits at the left with **0**.



DIVISION

Division

- Division is the reciprocal operation of multiplication

- **Paper-and-pencil example** (1001010_{ten} / 1000_{ten}):

```
                      1001      Quotient
Divisor  1000    / 1001010      Dividend
                 -1000000
                   0001010
                   0001010
                   0001010
                    -1000
                        10      Remainder
```

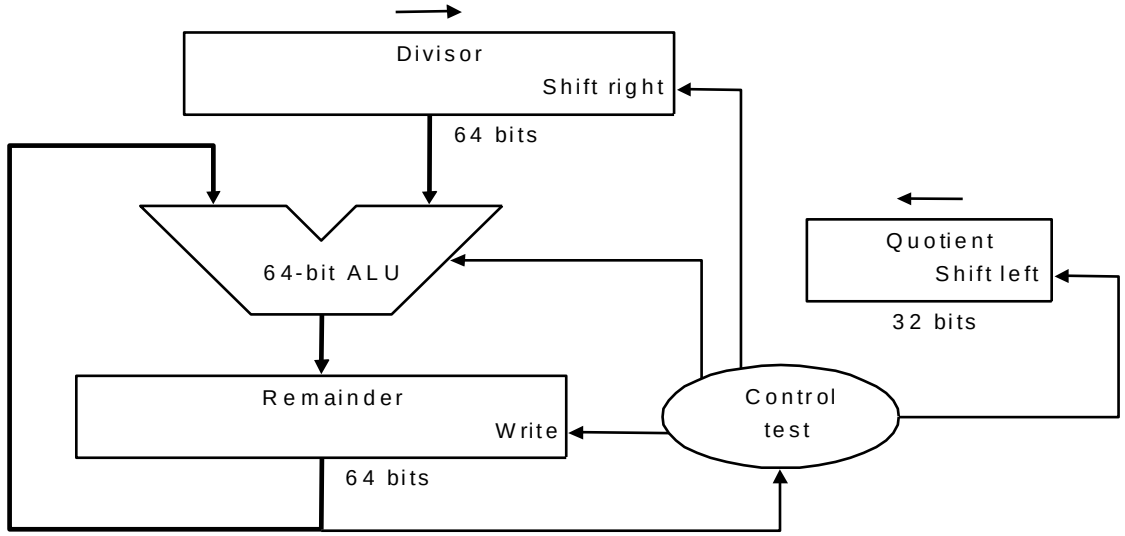
Dividend = Quotient x Divisor + Remainder

Division in Binary

- **Paper-and-pencil 4-bit example (0111_2 / 0010_2):**

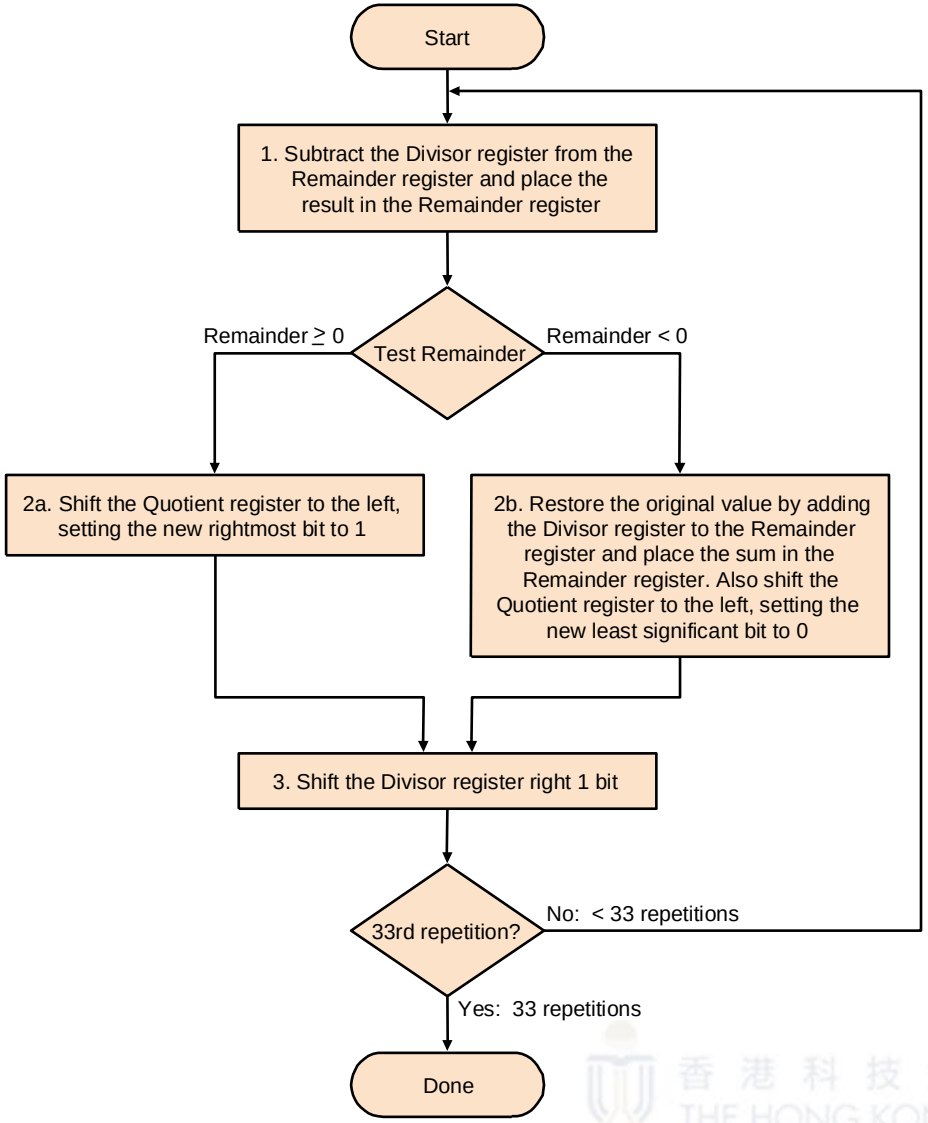
```
                      00011      Quotient
Divisor  0010    /00000111      Dividend
                -00100000
                 -0010000
                  -001000
                   -00100
                    00011
                   -00010
                    00001      Remainder
```

Sequential Hardware - Version 1



- **64-bit ALU**
- **Divisor register: 64 bits, Quotient register: 32 bits, Remainder register: 64 bits**
- **Operations:**
  - 32-bit divisor starts in the left half of divisor register; is shifted right 1 bit at each step
  - Quotient register is initialized to 0; shifted left 1 bit at each step
  - Remainder register is initialized with the dividend
  - Control decides
    - when to shift the divisor and quotient registers
    - when to write new values into the remainder register

Algorithm - Version 1



Observations Version 1

Similar to the first version of the multiplication hardware

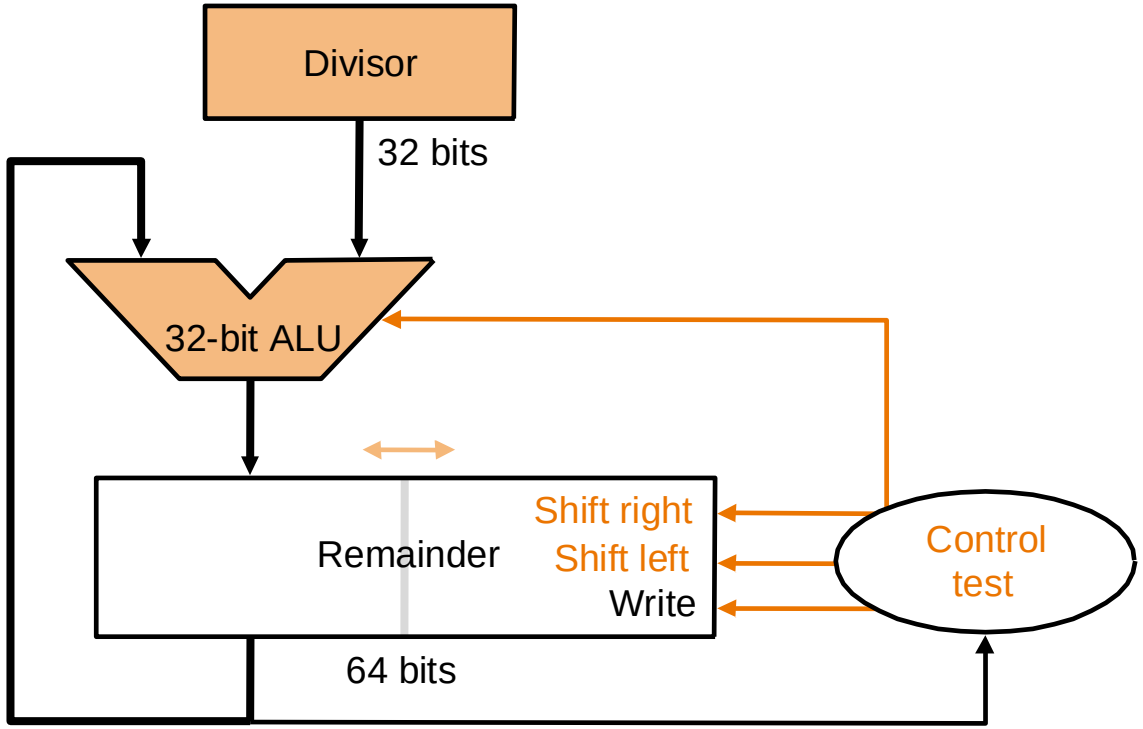
- **At most half of the divisor register has useful information**
  - Both the divisor register and ALU could potentially be cut in half
- **Shift divisor register to right =>Shift remainder register to left**
  - Produce the same alignment
  - But, simplify hardware necessary for the ALU and divisor register
- **Combine the remainder and quotient registers**

Example Optimized Division

Paper-and-pencil example (0111_2 / 0010_2):

```
                    0 0011      Quotient
Divisor  0010     | 00000111    Dividend
                  -0010
                   00001110
                  -0010
                   00011100
                  -0010
                   00111000
                  -0010
                   00011000
                   00110000
                  -0010
                   0001       Remainder
```

Hardware Optimized Version



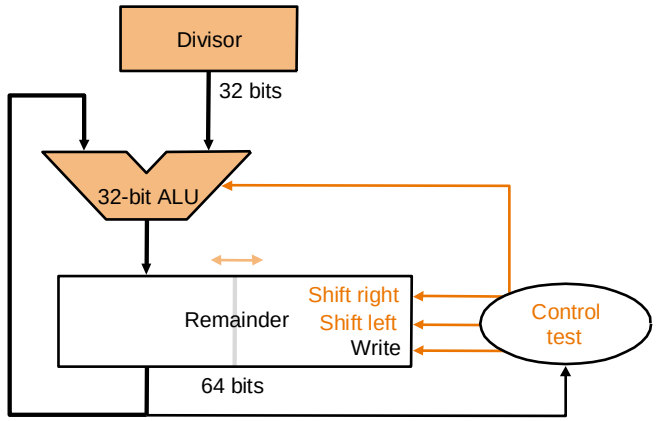
(changes made to previous version are highlighted in orange color)

Hardware Optimized Version (cont.)

- **32-bit ALU**

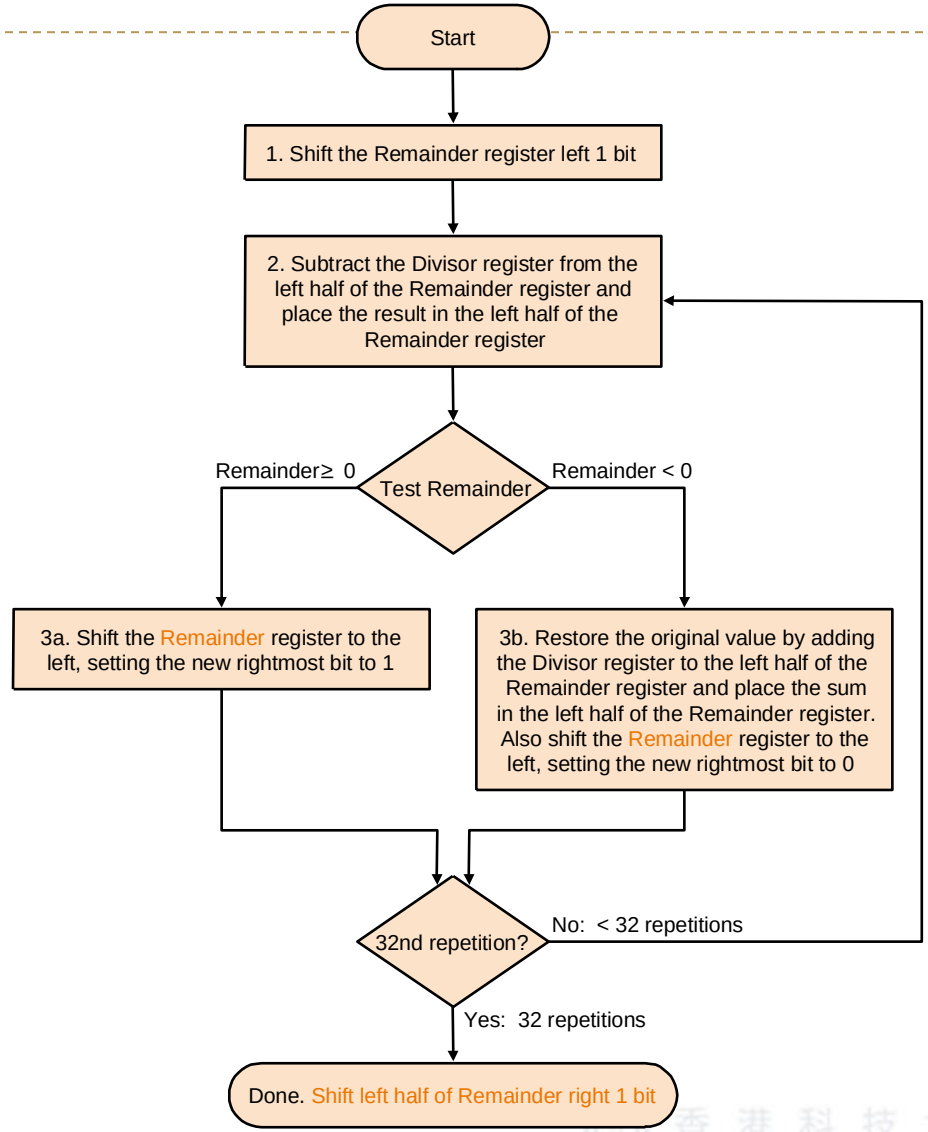
- **Two registers:**
  - **Divisor register: 32 bits**
  - **Remainder register: 64 bits**

  (right half also used for storing quotient)



- **Operations:**
  - 32-bit divisor is always subtracted from the left half of remainder register
    - The result is written back to the left half of the remainder register
  - The right half of the remainder register is initialized with the dividend
    - Left shift remainder register by one before starting
  - The new order of the operations in the loop is that the remainder register will be **shifted left one time too many**
    - Thus, **final correction step:** must **right shift back only the remainder** in the left half of the remainder register

Algorithm Optimized Version



Example

- Division of a 4-bit unsigned number (0111) by another one (0011)

Iteration	Divisor (D)	Remainder (R)	Remark
0	0011	0000 0111 0000 1110	Initial state R = R << 1
1		1101 1110 0000 1110 0001 1100	Left(R) = Left(R) – D Undo R = R << 1, R_0 = 0
2		1110 1100 0001 1100 0011 1000	Left(R) = Left(R) – D Undo R = R << 1, R_0 = 0
3		0000 1000 0001 0001	Left(R) = Left(R) – D R = R << 1, R_0 = 1
4		1110 0001 0001 0001 0010 0010	Left(R) = Left(R) – D Undo R = R << 1, R_0 = 0
extra		0001 0010	Left(R) = Left(R) >> 1

Remainder **Quotient**

correction

Signed Division

- If the signs of the divisor and dividend are different, then the quotient should be negated
- If the remainder is nonzero, then its sign should be the same as that of the dividend
- Example:

Dividend	Divisor	Quotient	Remainder
+7	+2	+3	+1
−7	+2	−3	−1
+7	−2	−3	+1
−7	−2	+3	−1

Division in MIPS

- **`div`** ('divide')
- **`divu`** ('divide unsigned')

- Examples:
  - `div  $rs, $rt` `# Lo = $rs / $rt; Hi = $rs mod $rt`
  - `divu  $rs, $rt` `# Lo = $rs / $rt; Hi = $rs mod $rt`

FLOATING POINT ARITHMETIC (OPTIONAL)

IEEE 754 Standard for Floating Point Arithmetic (optional)

Single precision

Significand \ Exponent	0	1 - 254	255
0	0	$(-1)^S \times (1.F) \times (2)^{E-127}$	$(-1)^S \times (\infty)$
≠ 0	$(-1)^S \times (0.F) \times (2)^{-126}$		non-numbers e.g. 0/0 , $\sqrt{-1}$

Double precision

Significand \ Exponent	0	1 - 2046	2047
0	0	$(-1)^S \times (1.F) \times (2)^{E-1023}$	$(-1)^S \times (\infty)$
≠ 0	$(-1)^S \times (0.F) \times (2)^{-1022}$		non-numbers e.g. 0/0 , $\sqrt{-1}$

Floating-Point Addition (optional)

- **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

- $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

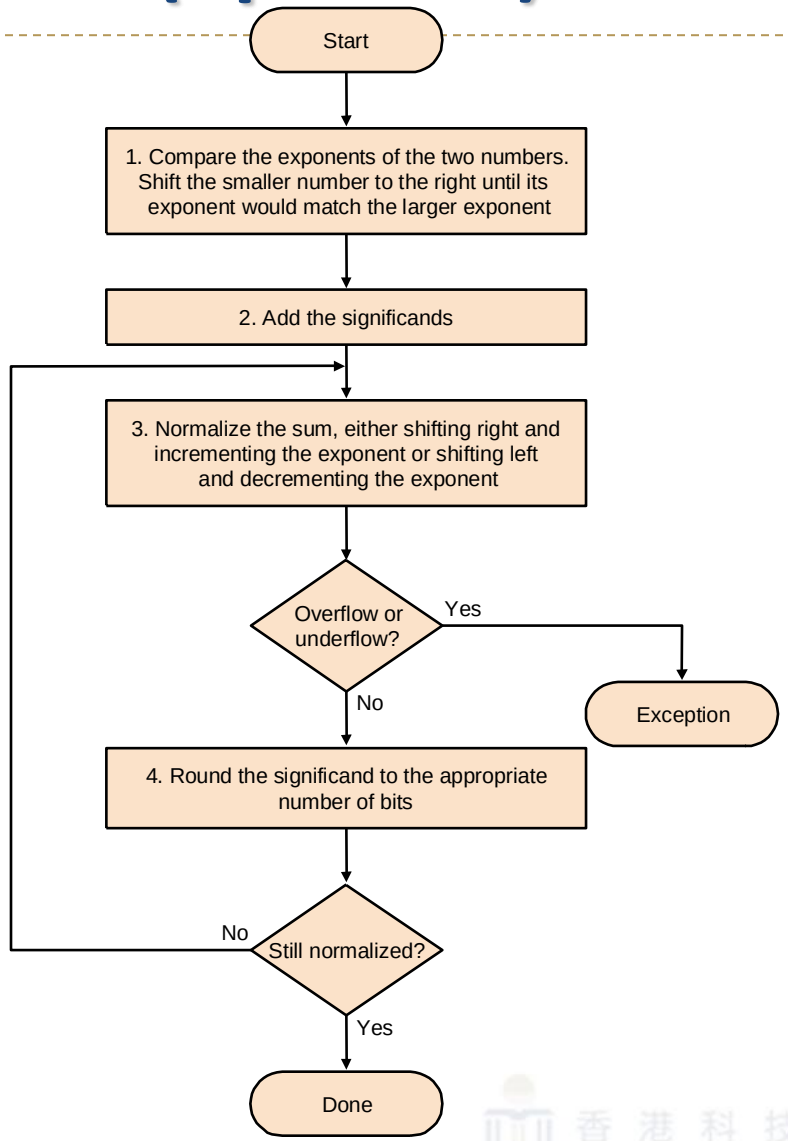
3. Normalize result & check for over/underflow

- 1.0015×10^{2}

4. Round and renormalize if necessary

- 1.002×10^{2}

Floating-Point Addition (optional)



Binary Example (optional)

- **4-bit 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

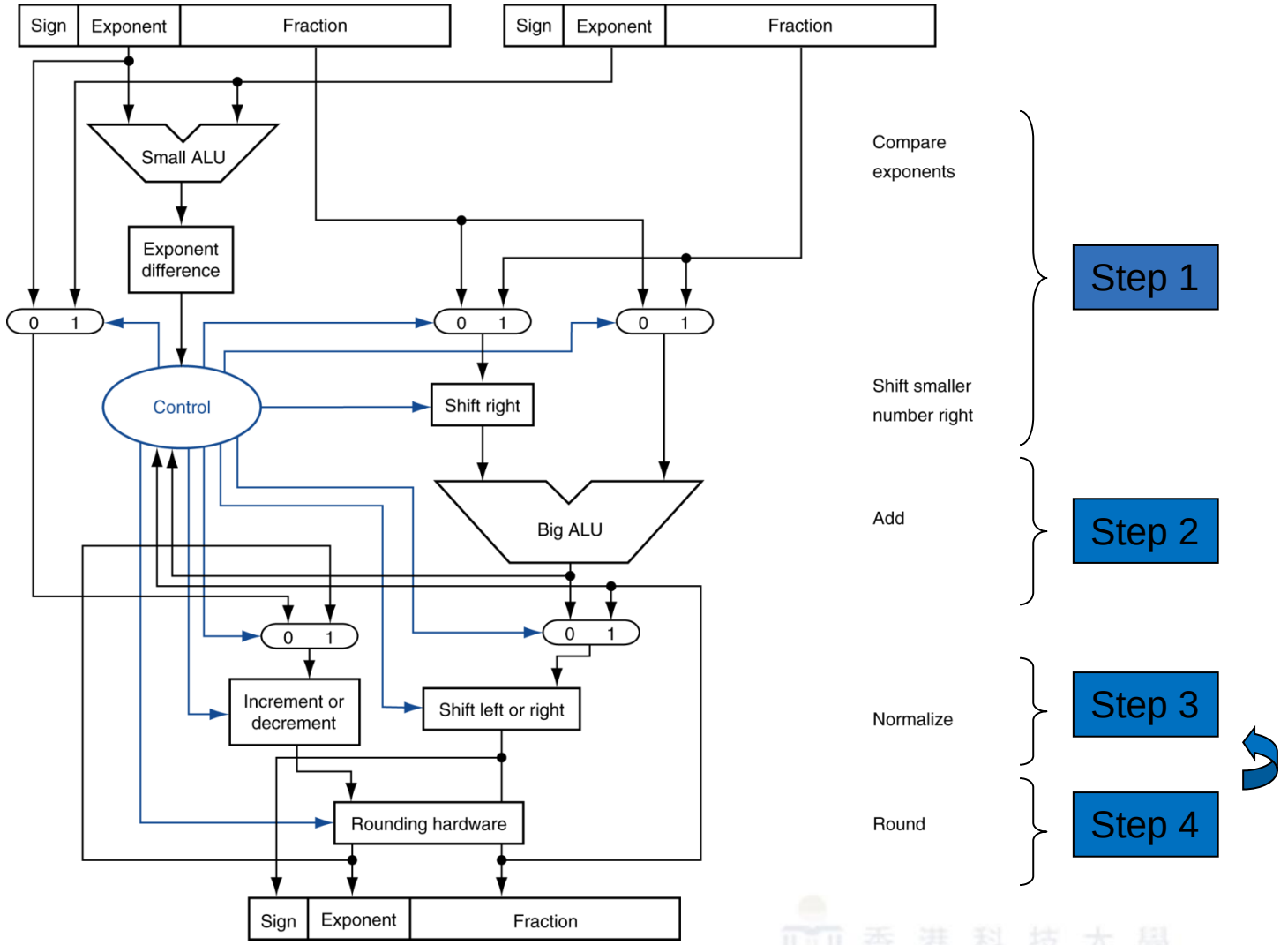
4. Round and renormalize if necessary

- $1.000_2 \times 2^{-4}$ (no change) = 0.0625

Floating-Point Addition Hardware (optional)

- **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

Floating-Point Addition Hardware cont. (optional)



Floating Point Multiplication (optional)

- **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}

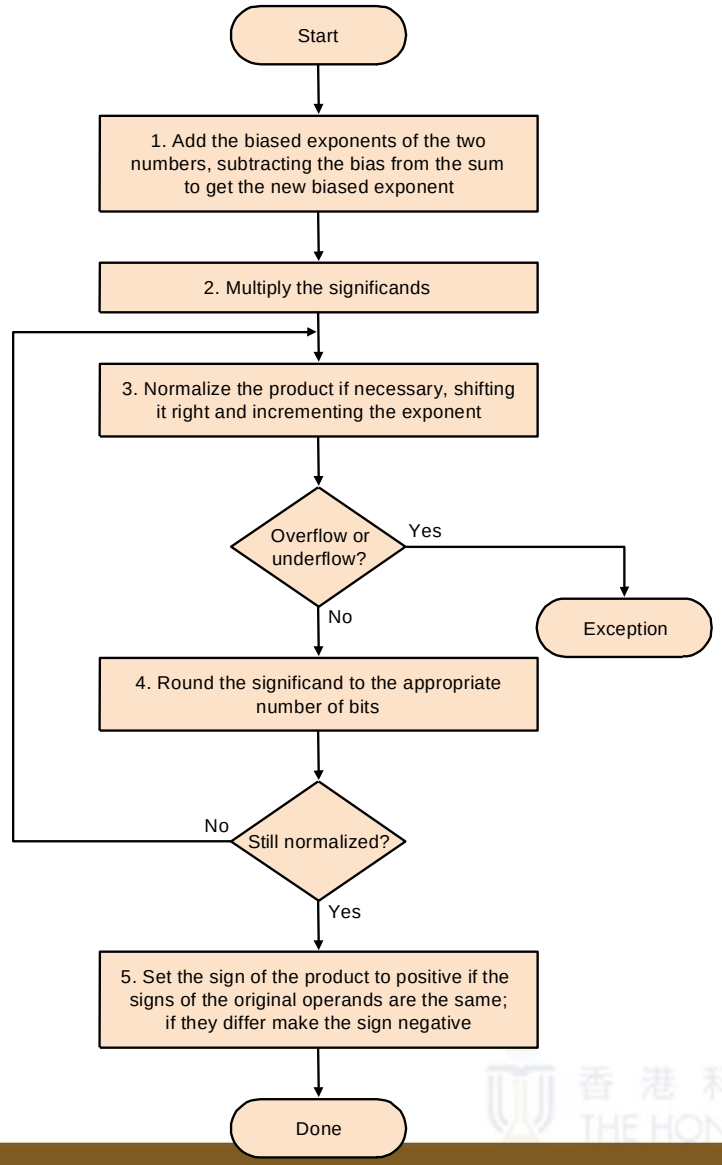
4. Round and renormalize if necessary

- 1.021×10^{6}

5. Determine sign of result from signs of operands

- $+1.021 \times 10^{6}$

Floating-Point Multiplication (optional)



Binary Example (optional)

- **4-bit 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

- $-1.110_2 \times 2^{-3} = -0.21875$

Floating-Point Arithmetic Hardware (optional)

- **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 $\leftrightarrow$ integer conversion
- **Operations usually takes several cycles**
  - Can be pipelined

MIPS Instructions for Floating-Point Operations (optional)

- MIPS supports IEEE 754 single-precision and double-precision formats
- **Addition:**
  - **`add.s`** ('addition, single'), **`add.d`** ('addition, double')
- **Subtraction:**
  - **`sub.s`** ('subtraction, single'), **`sub.d`** ('subtraction, double')
- **Multiplication:**
  - **`mul.s`** ('multiplication, single'), **`mul.d`** ('multiplication, double')
- **Division:**
  - **`div.s`** ('division, single'), **`div.d`** ('division, double')
- **Comparison:**
  - **`c.x.s`** ('comparison, single'), **`c.x.d`** ('comparison, double')
  - where **`x`** may be **`eq`**, **`neq`**, **`lt`**, **`le`**, **`gt`**, **`ge`**
- **Branch:**
  - **`bclt`** ('branch, true'), **`bclf`** ('branch, false')

Floating-Point Register (optional)

- **MIPS has a FP co-processor**
  - Referred to as co-processor 1
  - Has its own floating-point (FP) registers: **`$f0`**, **`$f1`**, **`$f2`**, …
  - These registers are used for either single or double precision

- **Separate loads and stores for FP registers: `lwc1` and `swc1`**

- **Example:**
  - load two single precision numbers from memory
  - then, add them and store the sum

```
lwc1    $f4, 4($sp)      # Load 32-bit f.p. number into F4
lwc1    $f6, 8($sp)      # Load 32-bit f.p. number into F6
add.s   $f2, $f4, $f6    # F2 = F4 + F6 single precision
swc1    $f2, 0($sp)      # Store 32-bit f.p. number from F2
```

FP Example: °F to °C (optional)

- 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

- Compiled 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
```

FP Example: Array Multiplication (optional)

- **X = X + Y × Z**
  - All 32 × 32 matrices, 64-bit double-precision elements
- **C code:**

```
void mm (double x[][],
         double y[][], double z[][]) {
  int 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)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of `x`, `y`, `z` in `$a0`, `$a1`, `$a2`, and `i`, `j`, `k` in `$s0`, `$s1`, `$s2`

MIPS Code (optional)

```
    li   $t1, 32        # $t1 = 32 (row size/loop end)
    li   $s0, 0         # i = 0; initialize 1st for loop
L1: li   $s1, 0         # j = 0; restart 2nd for loop
L2: li   $s2, 0         # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5    # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1  # $t2 = i * size(row) + j
    sll  $t2, $t2, 3    # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2  # $t2 = byte address of x[i][j]
    l.d  $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5    # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1  # $t0 = k * size(row) + j
    sll  $t0, $t0, 3    # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0  # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
    …
```

MIPS Code cont. (optional)

…

```
    sll  $t0, $s0, 5        # $t0 = i*32 (size of row of y)
    addu  $t0, $t0, $s2     # $t0 = i*size(row) + k
    sll   $t0, $t0, 3       # $t0 = byte offset of [i][k]
    addu  $t0, $a1, $t0     # $t0 = byte address of y[i][k]
    l.d   $f18, 0($t0)      # $f18 = 8 bytes of y[i][k]
    mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
    add.d $f4, $f4, $f16    # f4=x[i][j] + y[i][k]*z[k][j]
    addiu $s2, $s2, 1       # $k k + 1
    bne   $s2, $t1, L3      # if (k != 32) go to L3
    s.d   $f4, 0($t2)       # x[i][j] = $f4
    addiu $s1, $s1, 1       # $j = j + 1
    bne   $s1, $t1, L2      # if (j != 32) go to L2
    addiu $s0, $s0, 1       # $i = i + 1
    bne   $s0, $t1, L1      # if (i != 32) go to L1
```

Accurate Arithmetic (optional)

- **Floating-point numbers are normally approximations**
  - An infinite variety of real numbers exists between 0 and 1
  - No more than 2^{53} can be exactly represented in double precision

- **Do the best we can**
  - Get floating-point representation close to actual number
  - Keeps 2 extra bits on the right during intermediate additions
    - guard and round
  - Example:
    - $2.56_{10} \times 10^0 + 2.34_{10} \times 10^2$, assume 3 significant decimal digits

With guard and round digits

$$\begin{array}{r} 2.3400_{10} \\ +0.0256_{10} \\ \hline 2.3656_{10} \end{array}$$

Guard round

After rounding $2.37_{10} \times 10^2$

without

$$\begin{array}{r} 2.34_{10} \\ +0.02_{10} \\ \hline 2.36_{10} \end{array}$$

Accurate Arithmetic cont. (optional)

- **IEEE754 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**

Concluding Remarks

- A 32-bit ALU can be built by connecting 32 1-bit ALUs together
  - Subtraction makes use of addition
  - SLT makes use of subtraction
  - A multiplexor is used in an ALU to select appropriate result
  - Carry lookahead adders faster than ripple carry adders

- Multiplication: through a series of addition and shift operations
- Division: through a series of subtraction and shift operations
  - Make sure you understand how the hardware algorithms work

- Overflow (a type of exception)
  - A result of addition or subtraction
  - Detected by checking the signs of the operands and result