COMP2611: Computer Organization

Arithmetic for Computers

Major Goals

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

1. 2's Complement Arithmetic

Numbers

- Bits: the basis for binary number representation in digital computers

- Questions to answer next:
  - ✓ How to represent negative numbers
  - ✓ How to represent fractions and real numbers
  - ❍ How to handle numbers that go beyond the representable range
    - What is a representable range?

2's Complement Representation

- All computers use 2's complement representation for signed numbers
- Bit 31 is called the sign bit: (0 for non-negative, 1 for negative)
  - The positive half uses the same representation as before
  - The negative half uses conversion illustrated below:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 = 6_{10}$

i) Invert bits to get 1's complement

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1001_2 = -7_{10}$

ii) Add 1 to get 2's complement

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 = -6_{10}$

- Largest integer represented by a MIPS word:

  $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = (2^{31} - 1)_{10} = 2{,}147{,}483{,}647_{10}$
- Smallest integer represented by a MIPS word:

  $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2^{31}{}_{10} = -2{,}147{,}483{,}648_{10}$

- Immediate part for **lw**, **sw** & **addi** are represented in 2's complement

Signed and Unsigned Numbers

- **Signed numbers**
  - **negative** or **non-negative** integers, e.g. `int` in C/C++
- **Unsigned numbers**
  - **non-negative** integers, e.g. `unsigned int` in C/C++
- **Operations for unsigned numbers**
  - Comparison:

    `sltu (set on less than unsigned), sltiu`
  - Arithmetic:

    `addu, subu (add/subtract unsigned)`

    - Treat values of all registers as non-negative

    `addiu (add unsigned sign-extended immediate)`

    - The 16-bit immediate is sign-extended then addition as above

    `addi (add signed immediate`
  - Load:

    `lbu (load byte unsigned), lhu (load half unsigned)`

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`

Sign Extension

- e.g. `lb $t0, 0($s0)` # load a 8-bit signed number to 32-bit register
  - Bits 0~7 of `$t0` will contain the **byte** value stored at `0($s0)`
  - If the **byte** is a negative number, what happens to bits 8~24 of `$t0`?

Conversion of **n**-bit binary signed numbers into **m**-bit numbers (**m** > **n**)

- Done by filling the leftmost bits (**n**-th ~ (**m**-1)-th) with the sign bit
- For example:
  - 2 (16 bits -> 32 bits):

    0000 0000 0000 0010 -> 0000 0000 0000 0000 0000 0000 0000 0010

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

    1111 1111 1111 1110 -> 1111 1111 1111 1111 1111 1111 1111 1110
- If the immediate in the **`addi`** instruction = 1111 1111 1111 1110, the sign is extended as shown above before the ALU starts addition
- For unsigned addition operation **`addiu`**, sign is also extended as shown above before the ALU starts addition

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 appropriate operand is negated before being added to the other operand
- **Overflow**
  - The result is too large to fit into a word (32 bits)

Examples

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

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

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

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

Examples

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

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

In 2's complement the MSb is a sign bit

then, it means -2147483648_{10}

Detecting Overflow

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 condition**

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

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

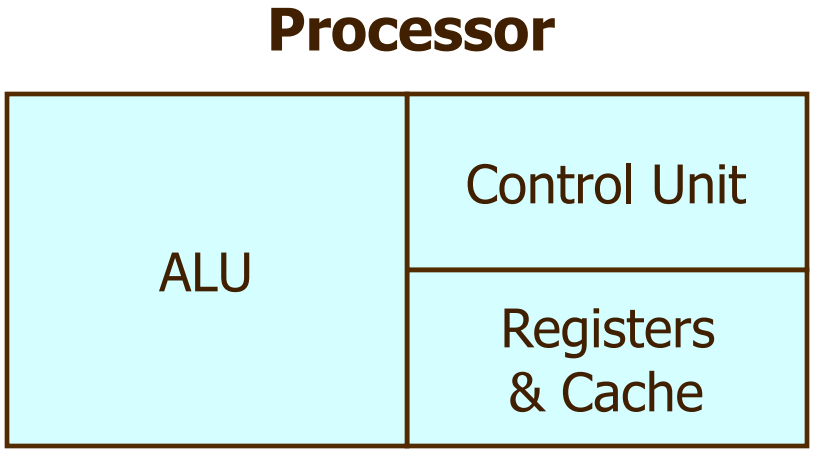
When an overflow exception occurs

- 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

2. Arithmetic Logic Unit

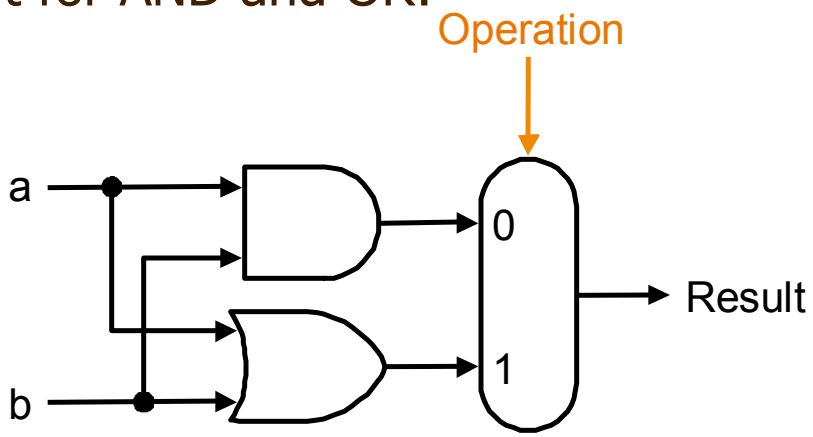
Constructing an Arithmetic Logic Unit

- 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)

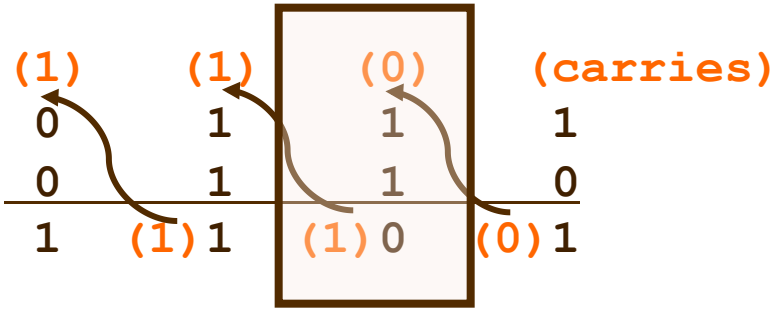
- Since a word in MIPS is 32 bits wide, we need a 32-bit ALU
- Ideally, we can build a 32-bit ALU by connecting 32 1-bit ALUs together (each of them takes care of the operation on one bit position)
- **1-bit** logical unit for AND and OR:

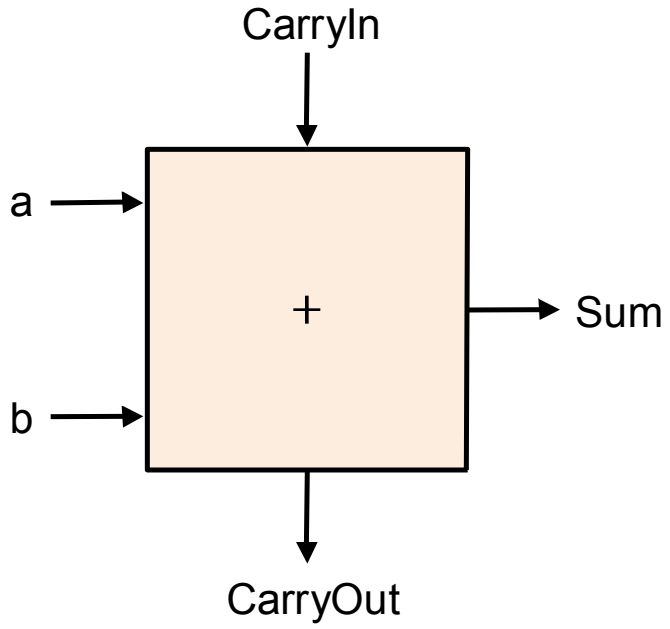


  - A multiplexor selects the appropriate result depending on the operation specified

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	$1 + 1 + 1 = 11_2$

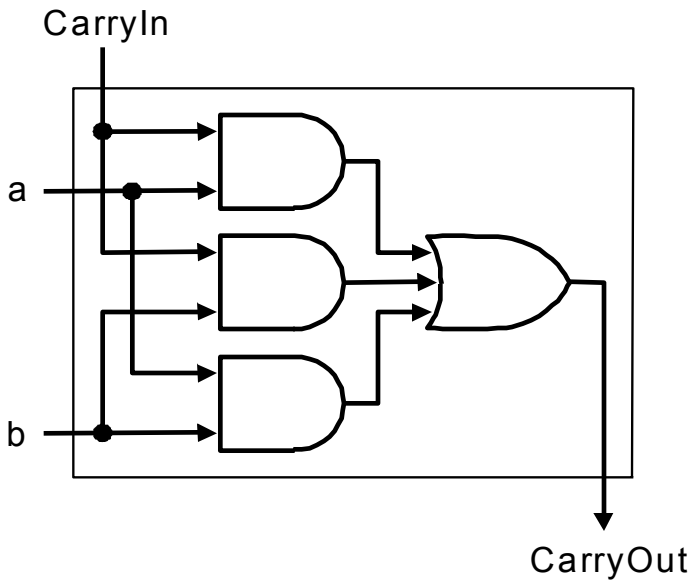
- Logic equations:

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

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

Hardware Implementation of 1-Bit Adder

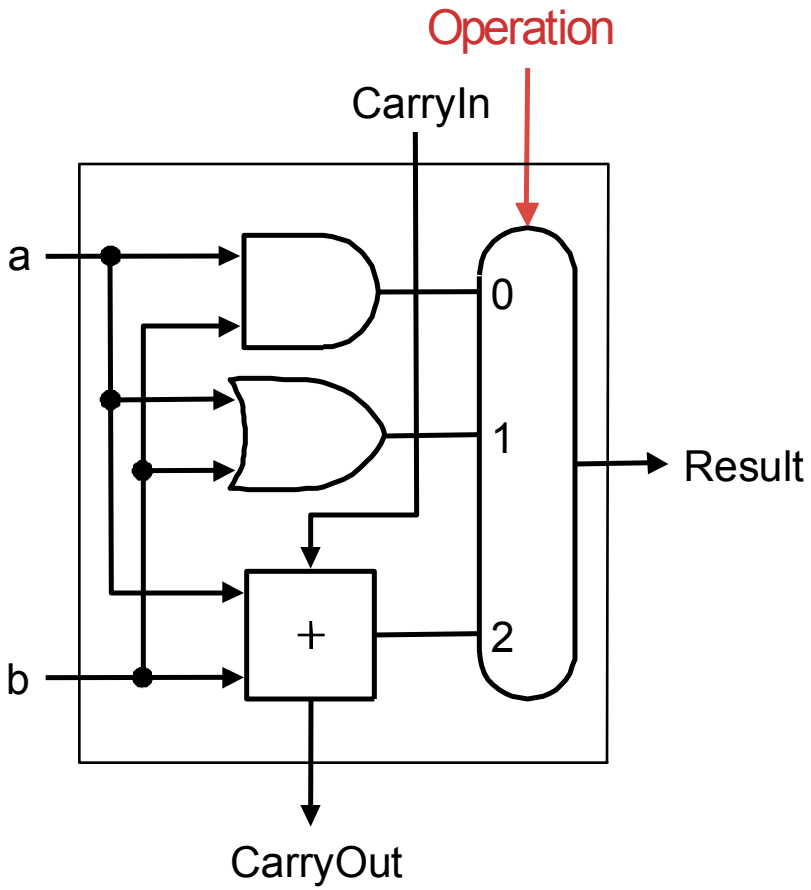
- $\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$
  $= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$



- SumOut bit: (It is left as an exercise)

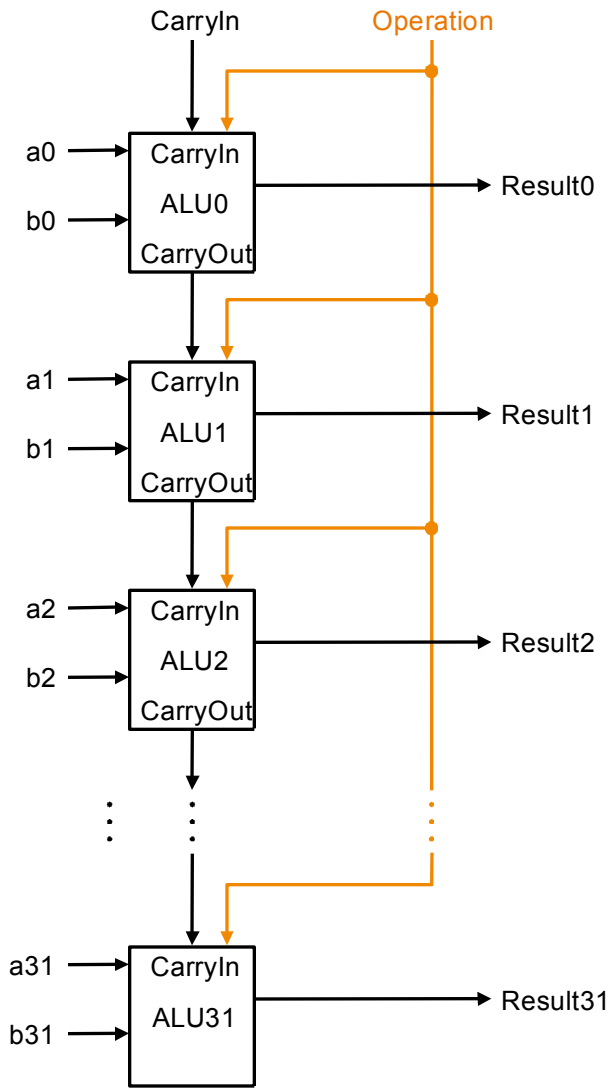
1-Bit ALU (AND, OR, and Addition)

- 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



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)



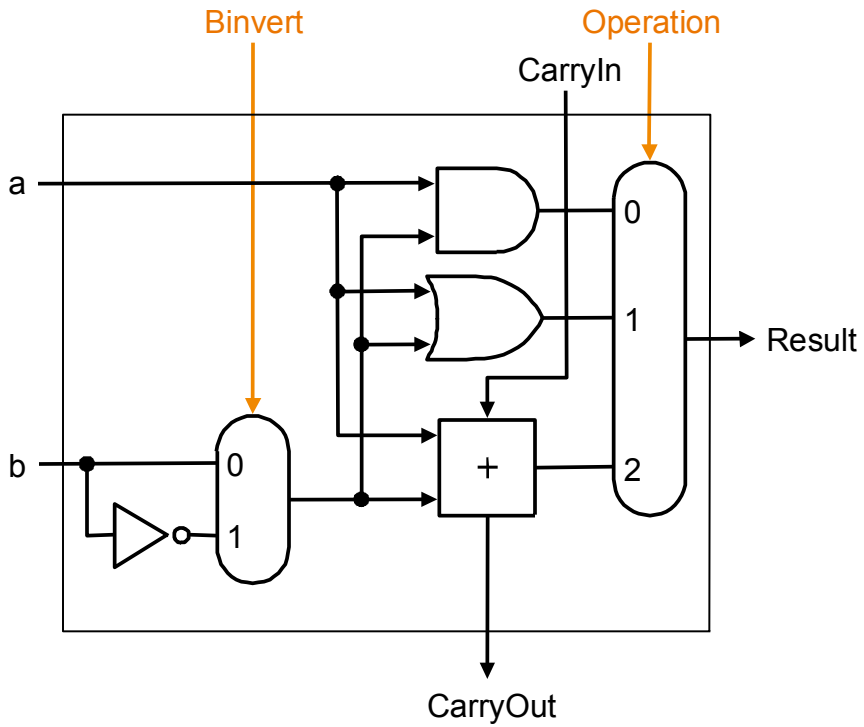
Subtraction

- **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

Tailoring the ALU for MIPS

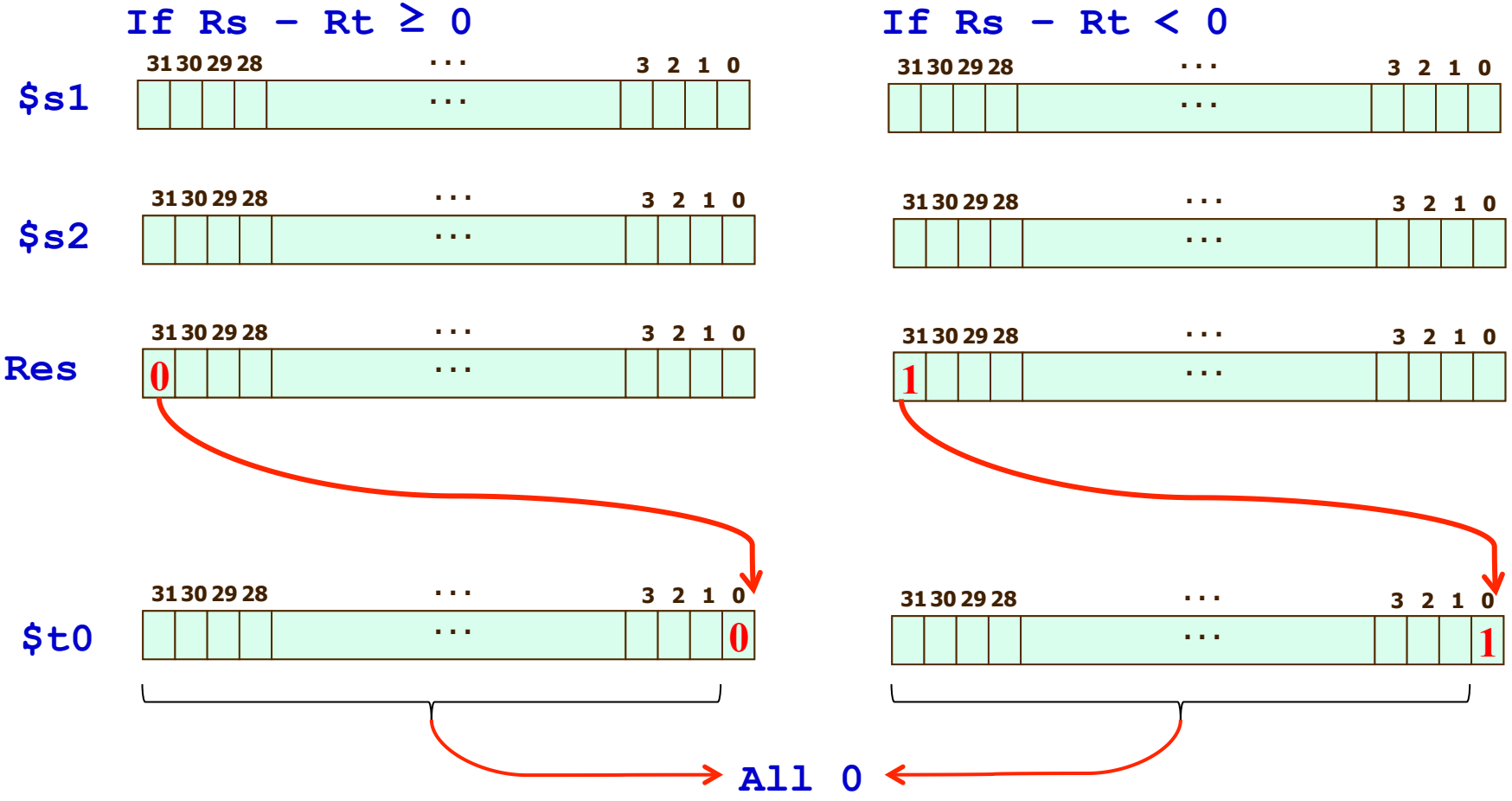
- The 32-bit ALU being designed so far can perform **`add`**, **`sub`**, **`and`**, **`or`** operations which constitute a large portion of MIPS' instruction set
- Two instructions not yet supported are: **`slt`** and **`beq`**

- When we need to compare **`Rs`** to **`Rt`**
  - By definition of **`slt`**, if **`Rs`** < **`Rt`**
    - LSb of the output is set to 1
    - Otherwise, it is reset to 0
- How to implement it?
  - The comparison is equivalent to testing if **`(Rs – Rt) < 0`**
  - If (**`Rs – Rt`**) is smaller than **`0`**
    - MSb of the subtraction (**`Rs – Rt`**) equals to 1 (means negative)
    - Otherwise, MSb of the subtraction equals to 0
  - Notice that the outcome of MSb is similar to the result of **`slt`**
  - ☑ Idea: **copy the MSb of the subtraction result to the LSb of `slt`'s output. All other bits of the output are 0**
  - ☑ **`slt`** can be done using two types of 1-bit ALUs

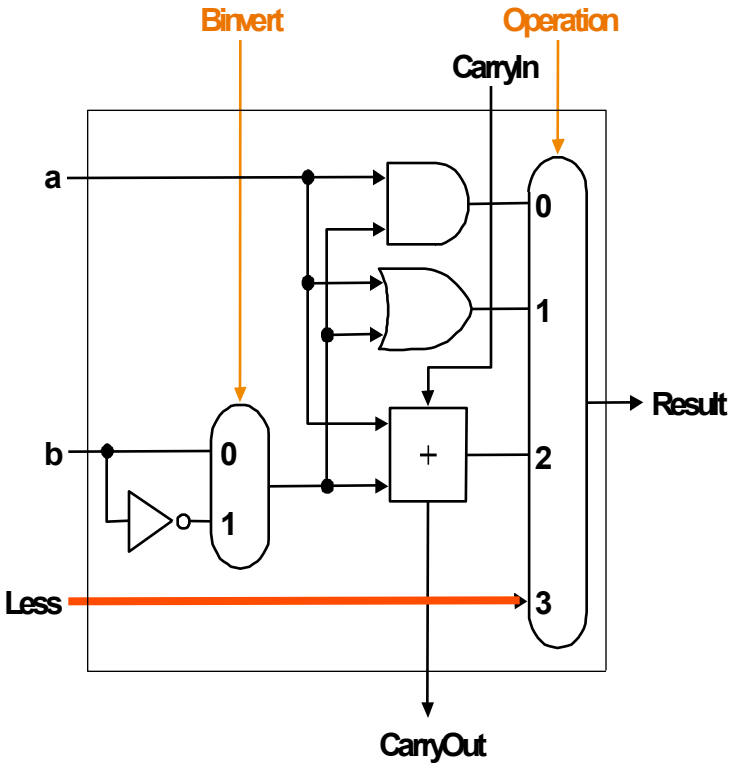
Tailoring the ALU for MIPS

- How to implement it?
  - The comparison is equivalent to testing if `(Rs – Rt) < 0`

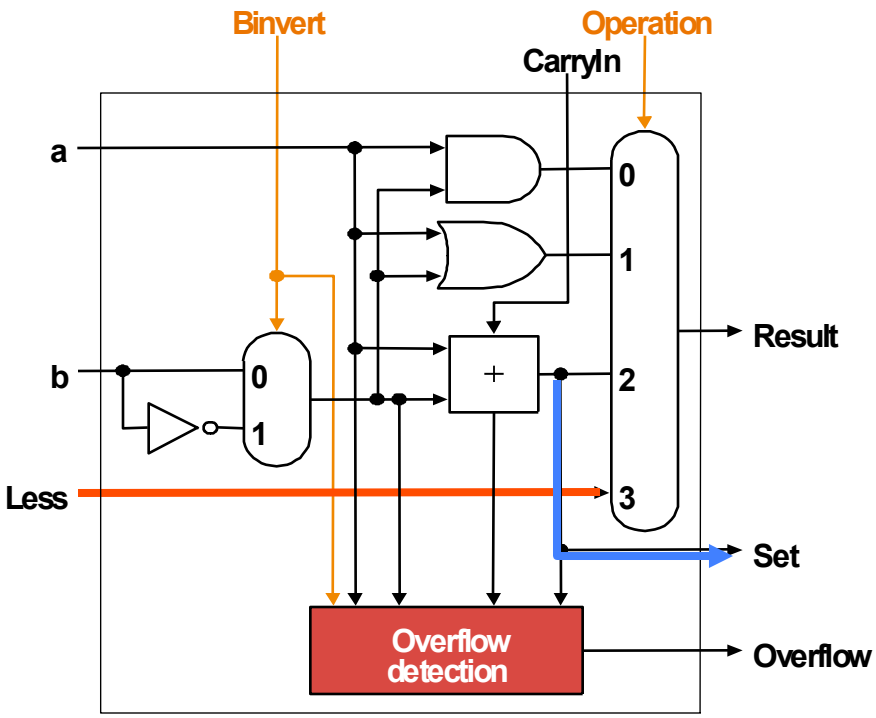
```
slt $t0, $s1, $s2
```



Tailoring the ALU for MIPS (cont'd)



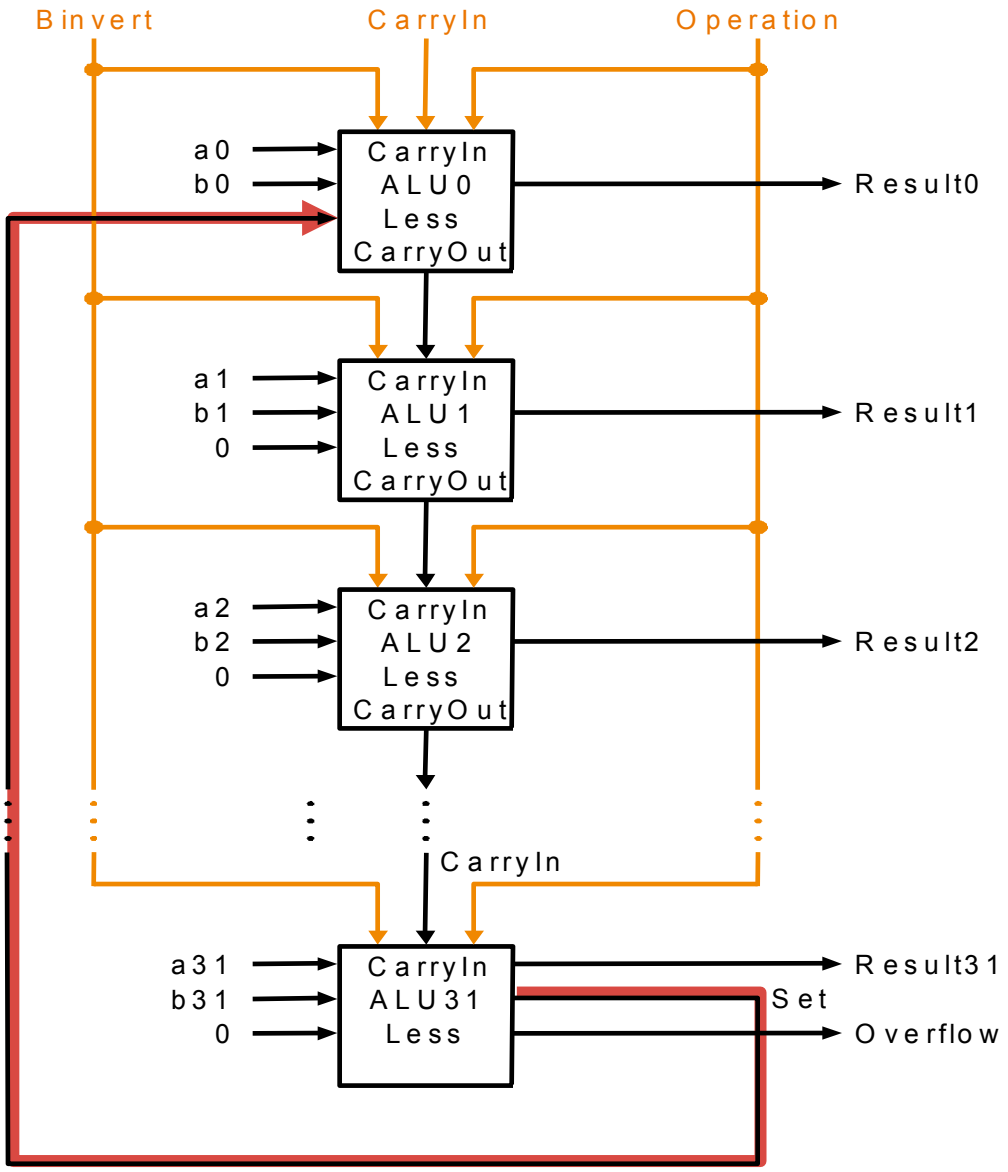
1-bit ALU for bits 0 to 30



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)

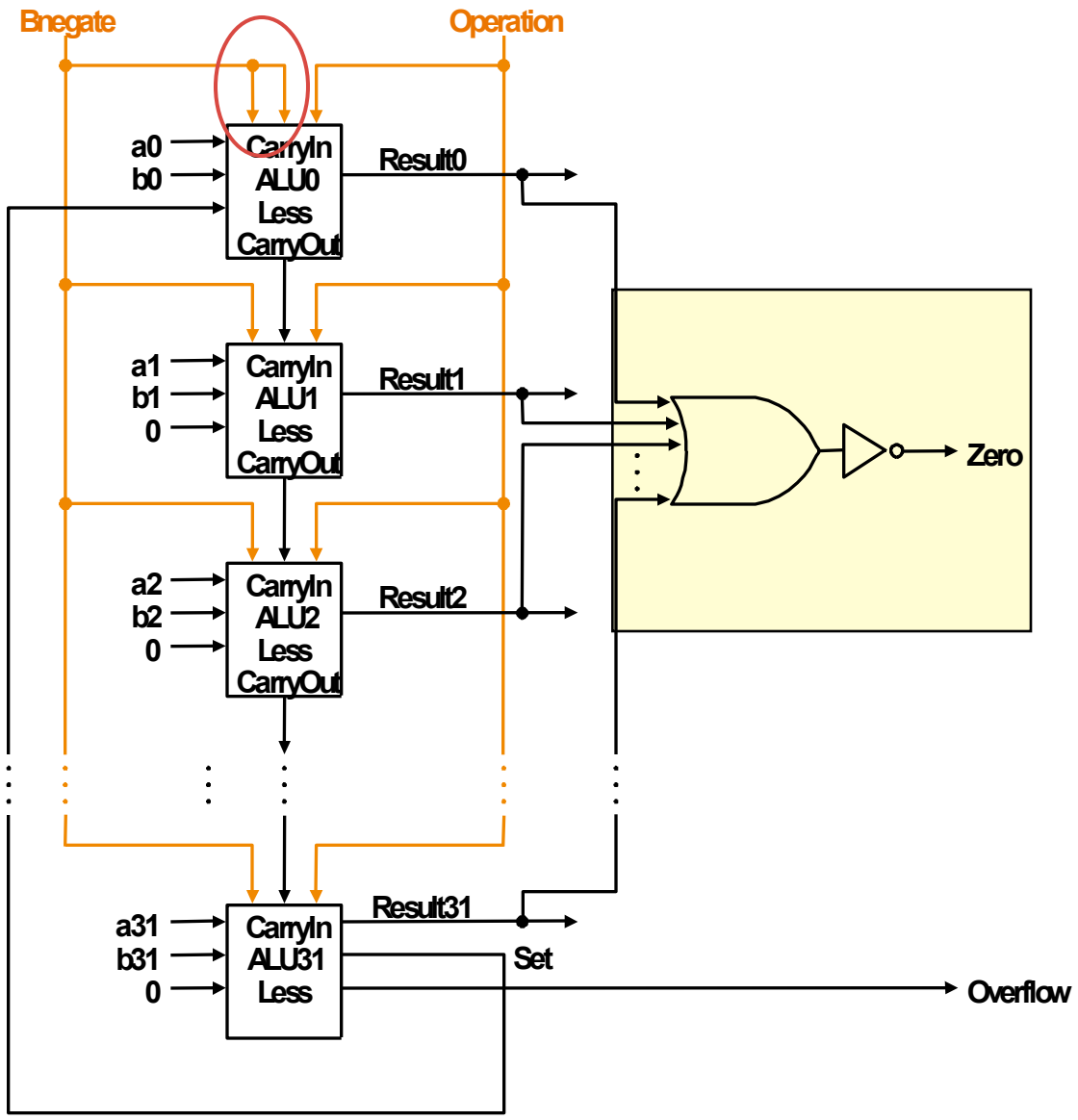


Tailoring the ALU for MIPS

- To support `beq`
- We need to compare `Rs` to `Rt`
  - The 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

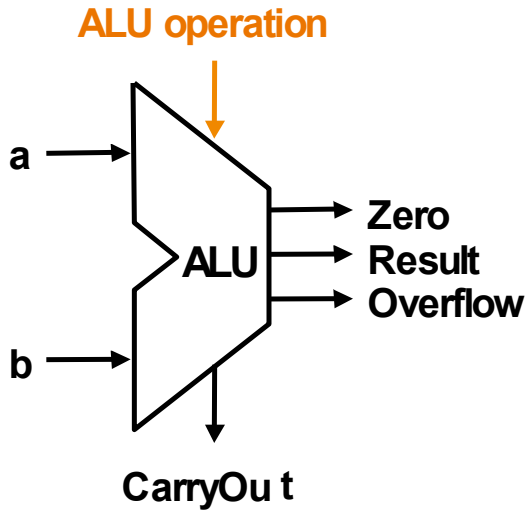
32-Bit ALU with (add, sub, AND, OR, slt)

- 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

- 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

Carry Lookahead

- A Bit position generates a Carry iff both inputs are 1: $G_i = a_i \bullet b_i$
- A Bit position propagates a Carry if exactly one input is 1: $P_i = a_i + b_i$
- Carryout at bit i can be expressed as:

$$C_i = G_i + P_i \bullet C_{i-1}$$

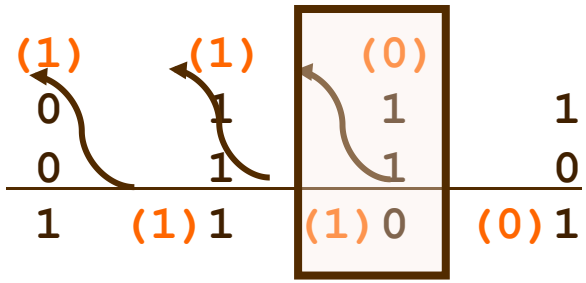
- After substitution we have

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + G_0 P_1 + C_0 P_0 P_1$$

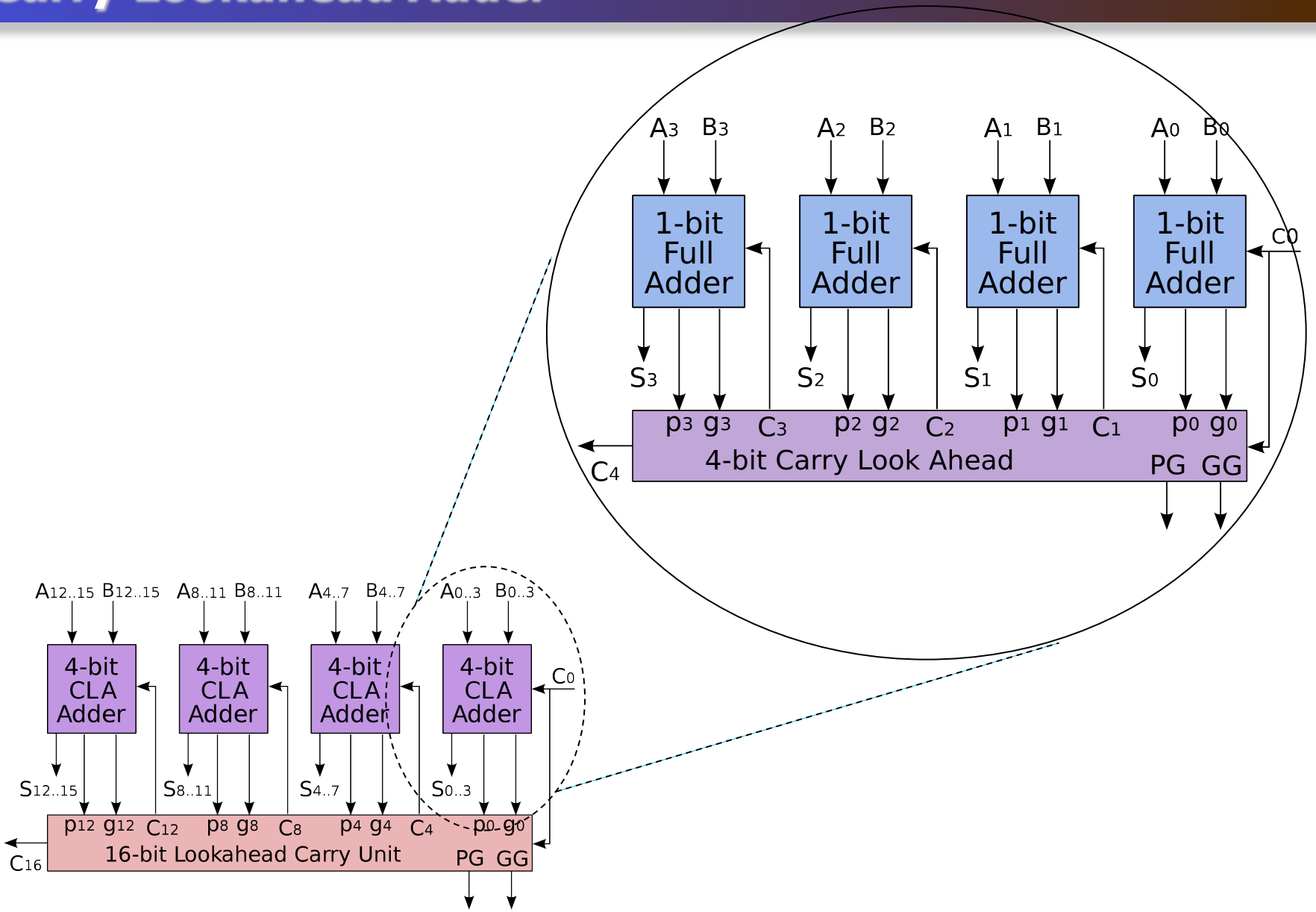
$$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$$

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



- 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

Carry Lookahead Adder



3. Multiplication

Multiplication

- Multiplication is much more complicated than addition and subtraction

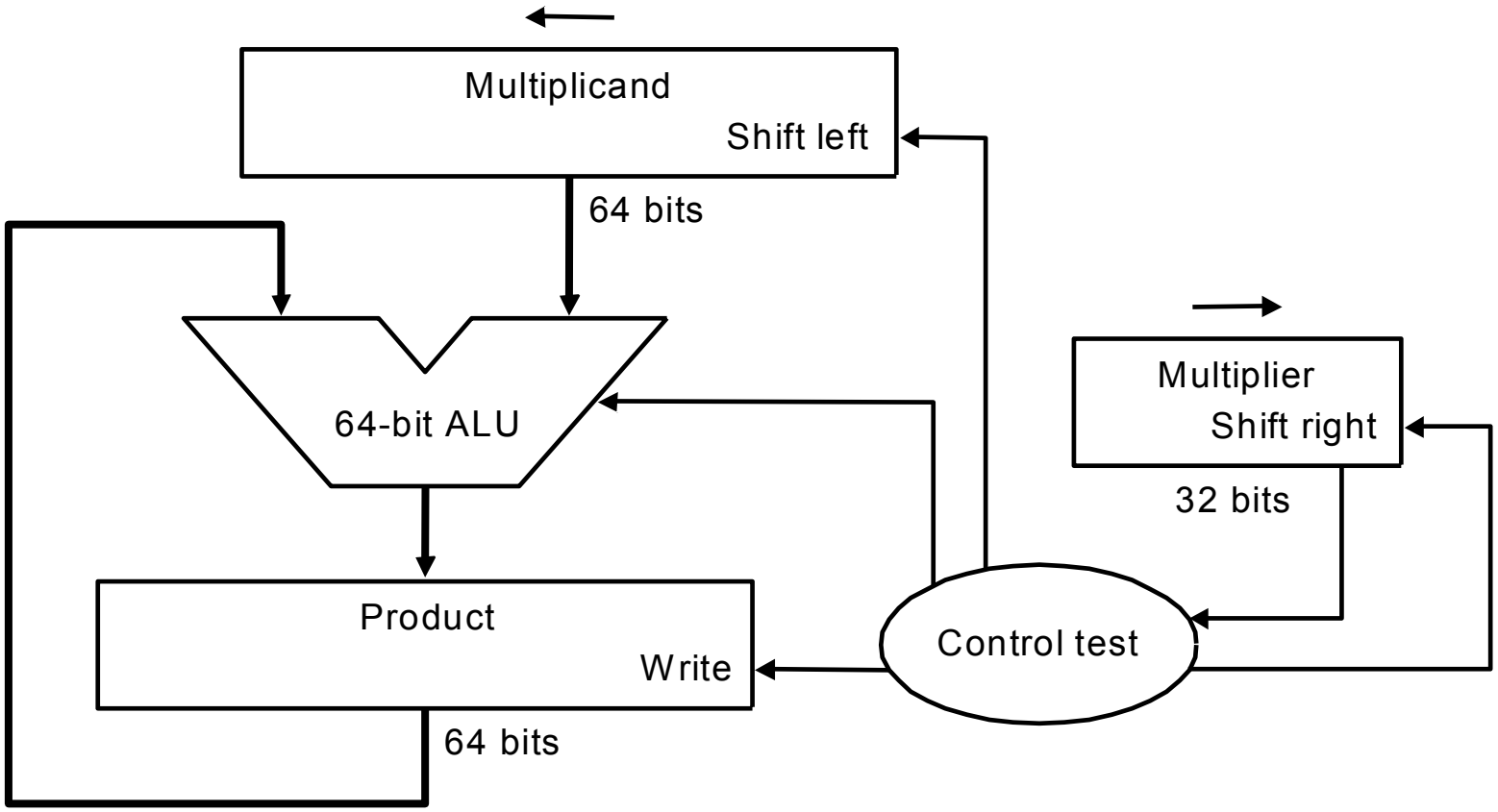
- **Paper-and-pencil example** (1000_{10} x 1001_{10}):

Multiplicand	1000
Multiplier	1001
	1000
	0000
	0000
	1000
Product	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 Multiplication Hardware - Version 1

- This version simply follows the flow of the paper-and-pencil example

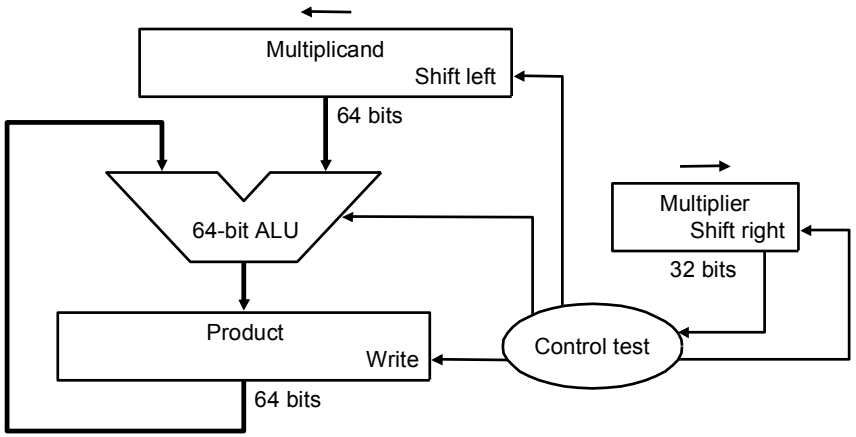


Example again

Multiplicand	00001000
Multiplier	1001
	00001000
	00000000
	00000000
	01000000
Product	1001000

Sequential Multiplication Hardware - Version 1

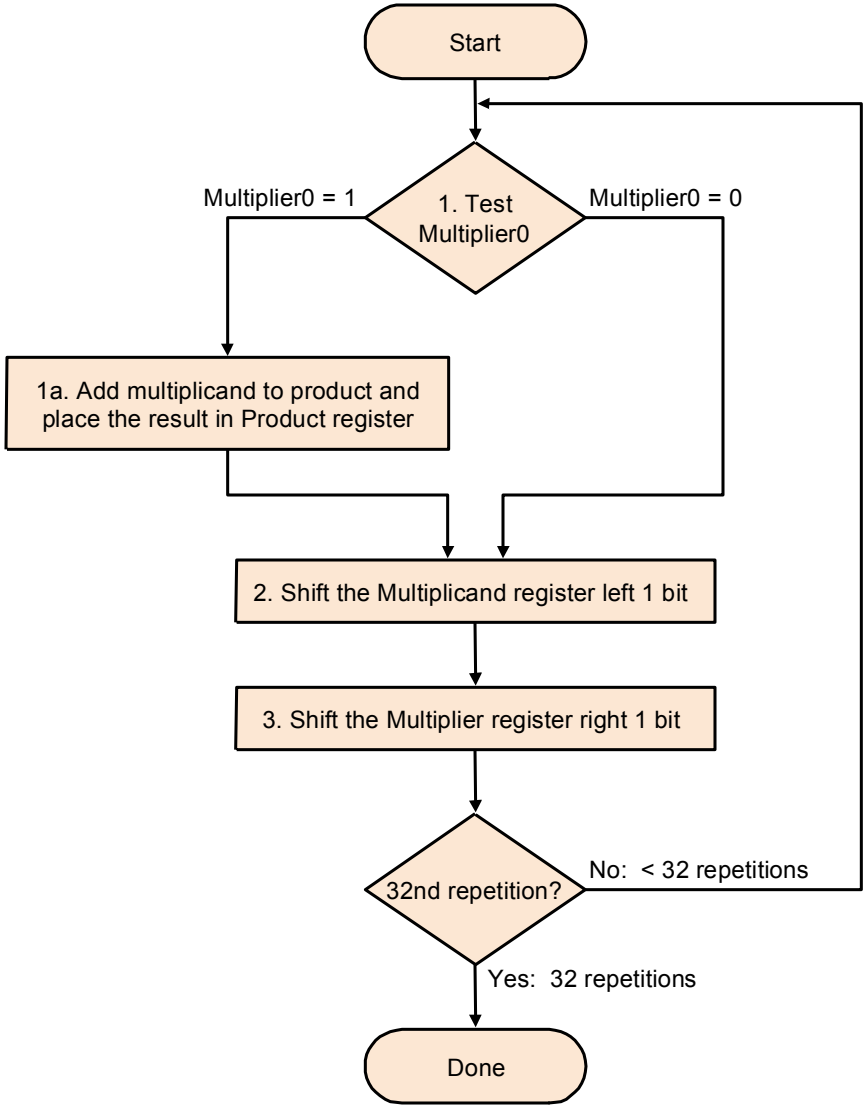
- 64-bit ALU



- **Three registers**:
  - Multiplicand register: 64 bits
  - Multiplier register: 32 bits
  - Product register: 64 bits

- **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

Multiplication Algorithm - Version 1



- Three basic steps needed for each bit
- If we need one clock cycle for each step then about 100 clock cycles are needed to multiply two 32-bit numbers
- Slow!

Example for Multiplication Version 1

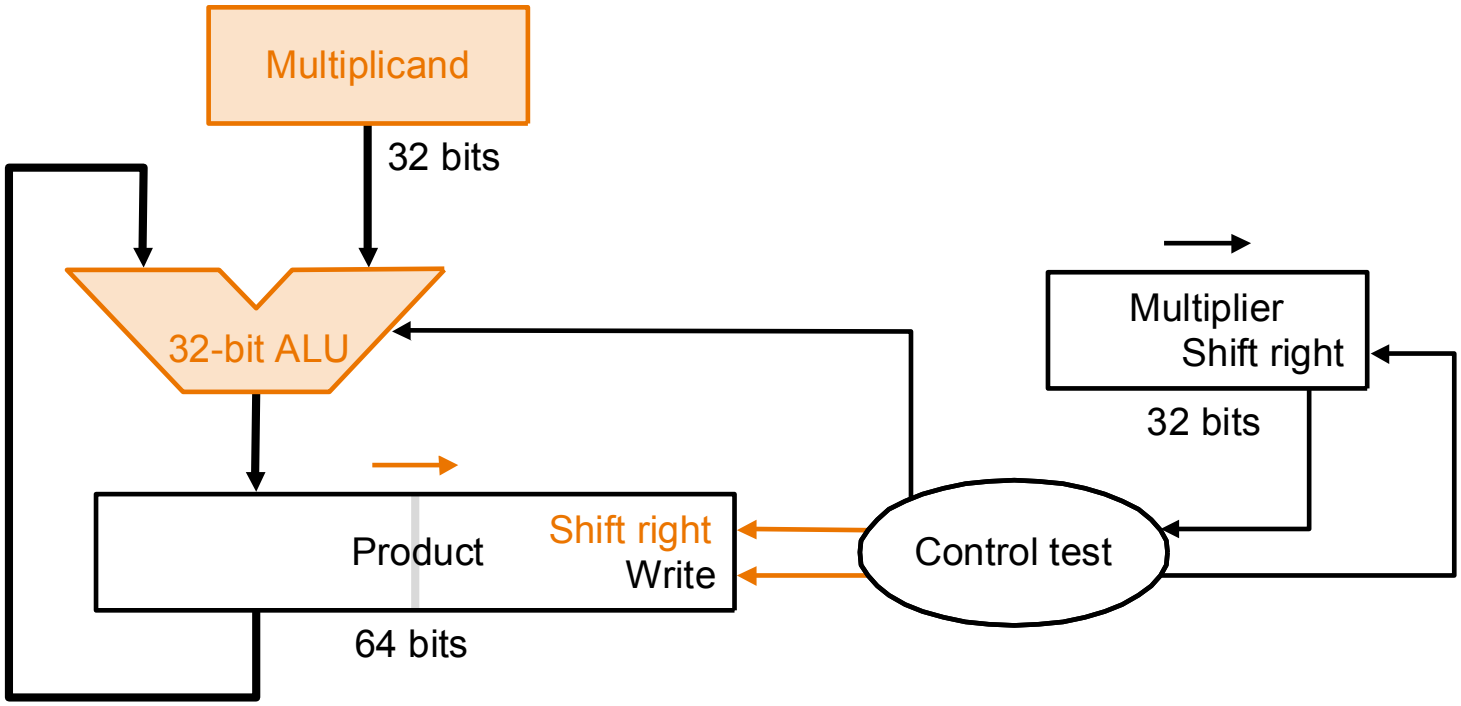
	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1→ Prod = Prod + Mcand	0011	0000 0010	**0000 0010**
	2: Shift left Multiplicand	0011	**0000 0100**	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1→ Prod = Prod + Mcand	0001	0000 0100	**0000 0110**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

Some Observations About Version 1

- Half of the bits of the 64-bit multiplicand register are always 0
  - i.e. only half could contain useful bit values
  - So, we can 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
  - So, we can reduce ALU size: 64-bit → 32-bit
- The multiplicand is shifted left with 0s inserted in the new positions
  - So, the multiplicand cannot affect the least significant bits of the product after they settle down

Sequential Multiplication 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)

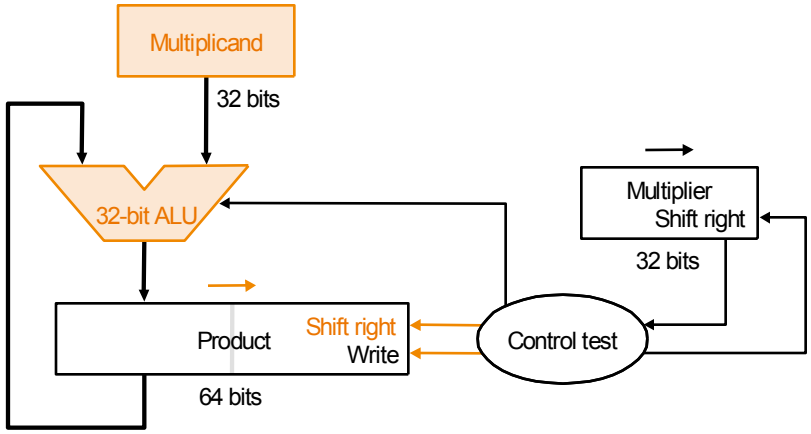
Example to Explain Version 2

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

Multiplicand	1000	
Multiplier	1001	
	(+1000)	
	1000	(shift right)
	1000	(shift right)
	1000	(shift right)
	(+1000)	
Product	1001000	

Sequential Multiplication Hardware - Version 2

- 32-bit ALU



- **Three registers**:
  - Multiplicand register: 32 bits
  - Multiplier register: 32 bits
  - Product register: 64 bits

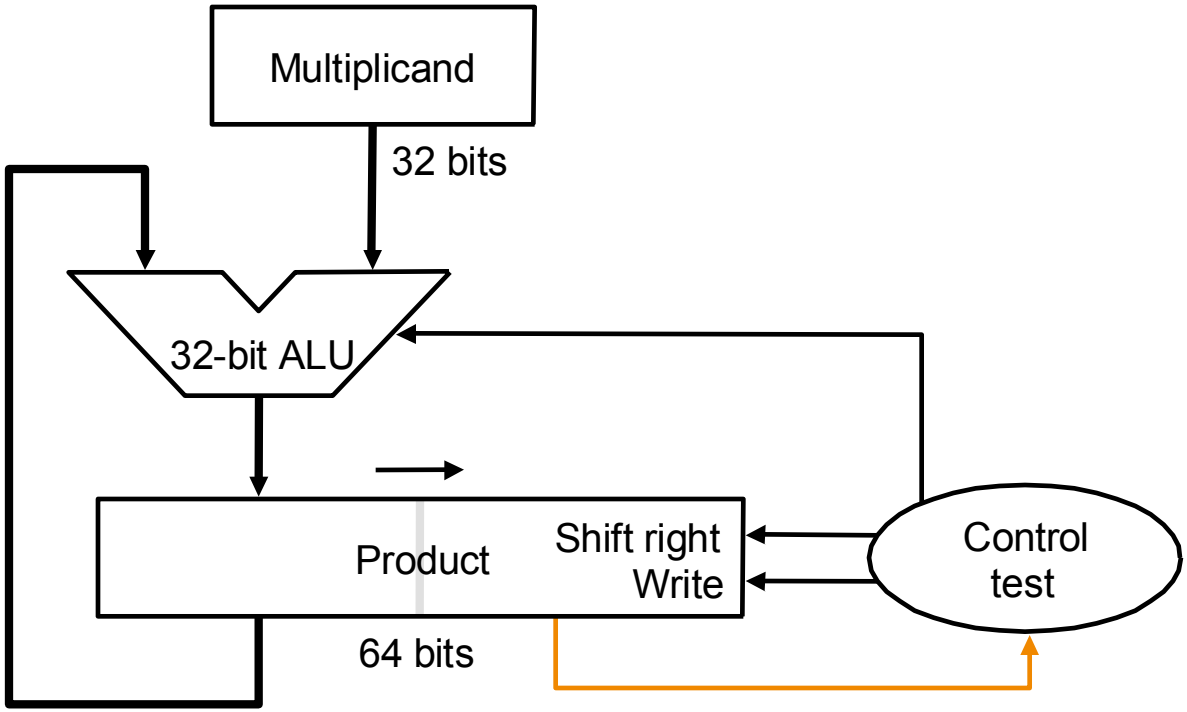
- **Operations**:
  - Instead of shifting the multiplicand register left, this version shifts the product register right 1 bit at each step
  - The 32-bit multiplicand is always added to the left half of the product register (hence only a 32-bit adder is needed)
  - The sum is written back to the left half of the product register

Some Observations About 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**

Multiplication Hardware - Refined Version

- Combine the right half of product register with the multiplier register

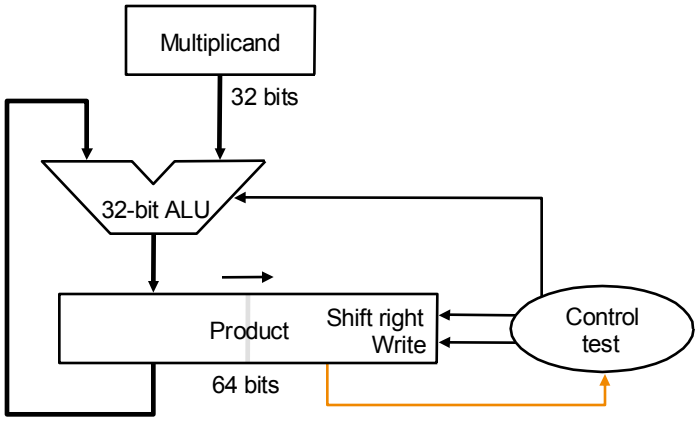


(changes made to previous version are highlighted in orange color)

Multiplication Hardware - Refined Version

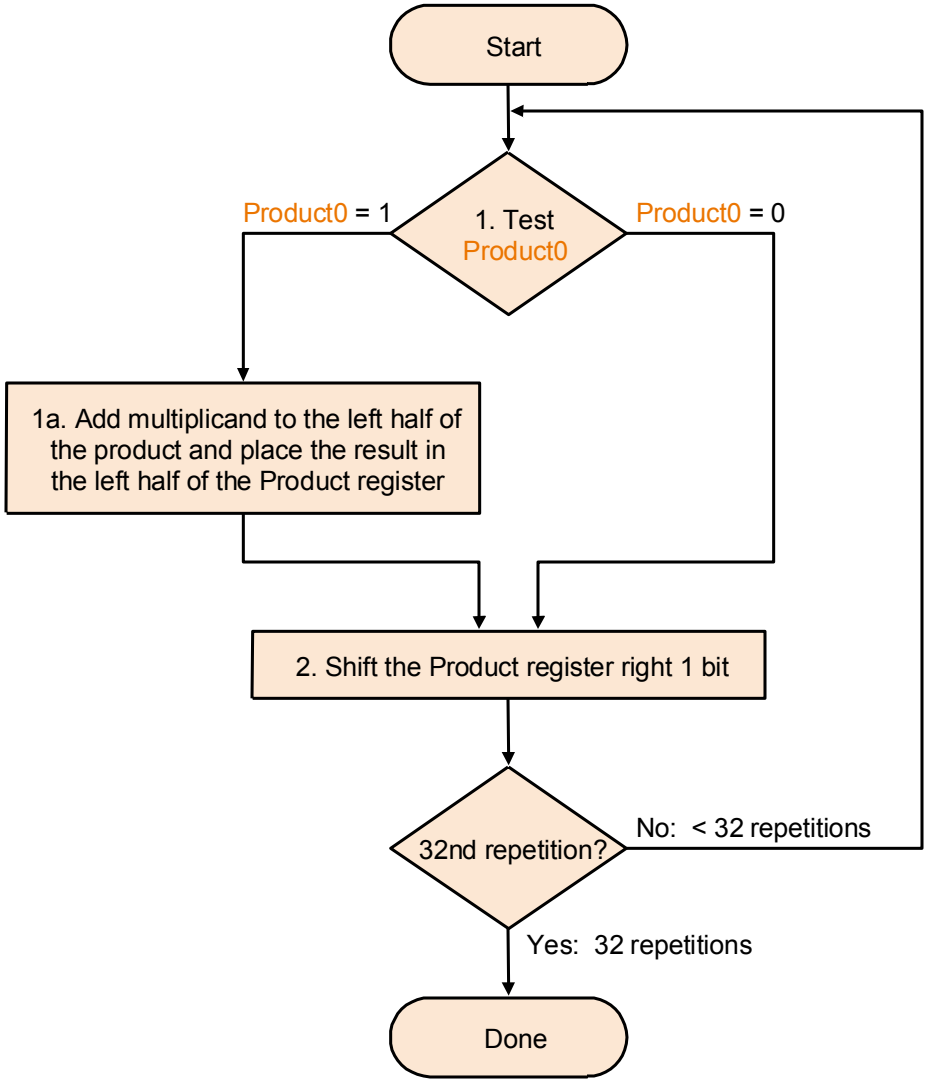
- **32-bit ALU**

- **Two registers**:
  - **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

Multiplication Algorithm - Refined Version



Example

- 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

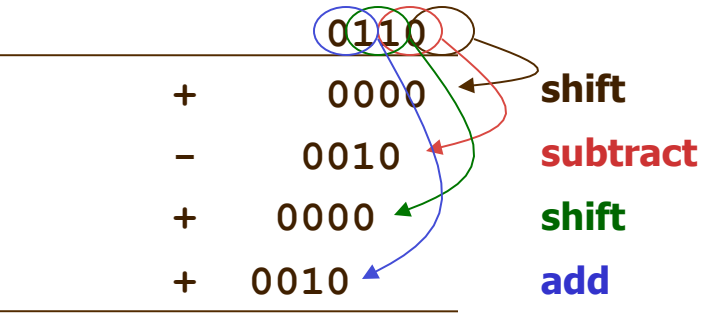
Signed Multiplication

- 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

Motivation behind Booth Algorithm

- Let's consider multiplying 0010_2 and 0110_2

	Convention		Booth	
Multiplicand	0010		0010	
Multiplier	x 0110		0110	
	+ 0000	+	0000	shift
	+ 0010	-	0010	subtract
	+ 0010	+	0000	shift
	+ 0000	+	0010	add
Product	= 0001100	=	0001100	

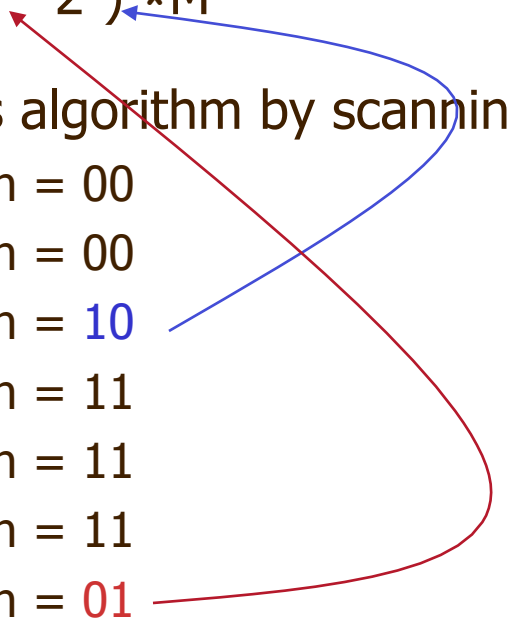


Idea of Booth Algorithm

- Looks at two bits of multiplier at a time from right to left
- Assume that shifts are much faster than adds
- Basic idea to speed up the calculation: avoid unnecessary additions

Example to Explain the Math

- Multiplier = 00111100
  - i.e. $i_1 = 2$, $i_2 = 5$
- M x 00111100 = $2^2 * M + 2^3 * M + 2^4 * M + 2^5 * M$
  $= 2^2 * (2^0 + 2^1 + 2^2 + 2^3) * M$
  $= 2^2 * (2^4 - 1) * M$
  $= (2^6 - 2^2) * M$
- Running the Booth's algorithm by scanning multiplier from right to left
  - Iteration 0, pattern = 00
  - Iteration 1, pattern = 00
  - Iteration 2, pattern = 10
  - Iteration 3, pattern = 11
  - Iteration 4, pattern = 11
  - Iteration 5, pattern = 11
  - Iteration 6, pattern = 01



Booth's Algorithm

To find out why, do the math:

- Consider a series of ones in the multiplier (from bit i_1 to bit i_2)
- M: multiplicand; multiplying M with this series of ones results in

$$\begin{aligned} \text{Prod} &= (2^{i1}) * M + (2^{i1+1}) * M + \ldots + (2^{i2}) * M \\ &= (2^{i2+1} - 2^{i1}) * M \end{aligned}$$

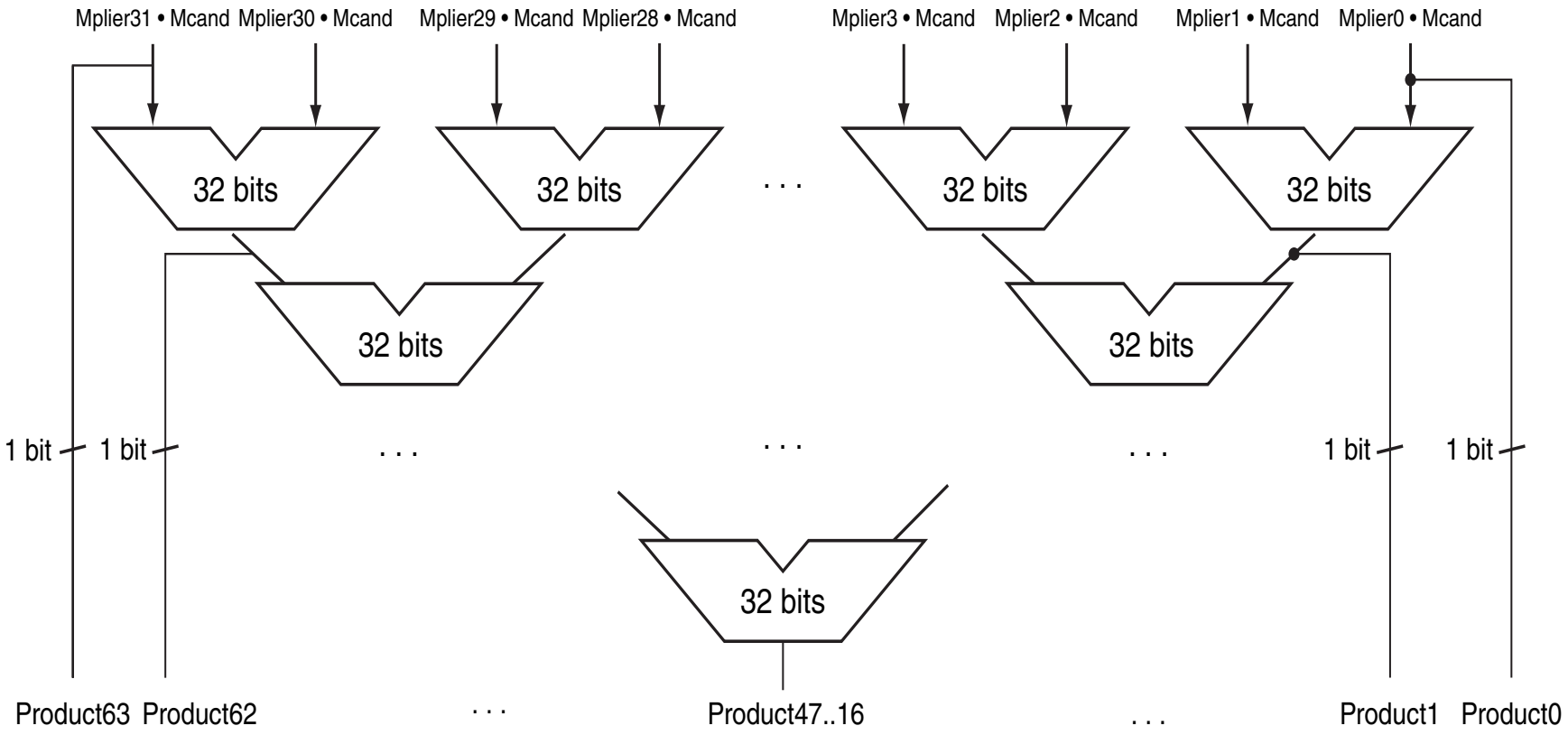
- Thus, $(i_2\text{-}i_1)$ adds in revised algorithm ⇒ one add and one subtract

Detailed algorithm:

- We look at 2 bits at a time (current bit and previous one):
  - 00: middle of a string of 0's; no arithmetic operation
  - 01: end of a string of 1's; add M to the left half of product
  - 10: start of a string of 1's; subtract M from the left half of product
  - 11: middle of a string of 1's- no arithmetic operations
- Previous bit is set to 0 for the first iteration to form a two-bit pattern

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



Multiply in MIPS

- Separate pair of 32-bit registers to contain 64-bit product, **`Hi`** and **`Lo`**

- **`mult`** (multiply) and **`multu`** (multiply unsigned)
  - **`mult  $s2, $s3`** **`# Hi, Lo = $s2 x $s3`**
  - **`multu $s2, $s3`** **`# Hi, Lo = $s2 x $s3`**
  - Both MIPS multiply instructions ignore overflow
  - No overflow if **`Hi`** is 0 for **`multu`** or the replicated sign of **`Lo`** for **`mult`**

- Fetch the integer 32-bit product
  - **`mflo`** (move from lo) **`mflo $s1`** **`# $s1 = Lo`**
  - **`mfhi`** (move from hi) **`mfhi $s1`** **`# $s1 = Hi`**
  - **`mfhi`** can transfer **`Hi`** to a general-purpose register to test for overflow

4. Division

Division

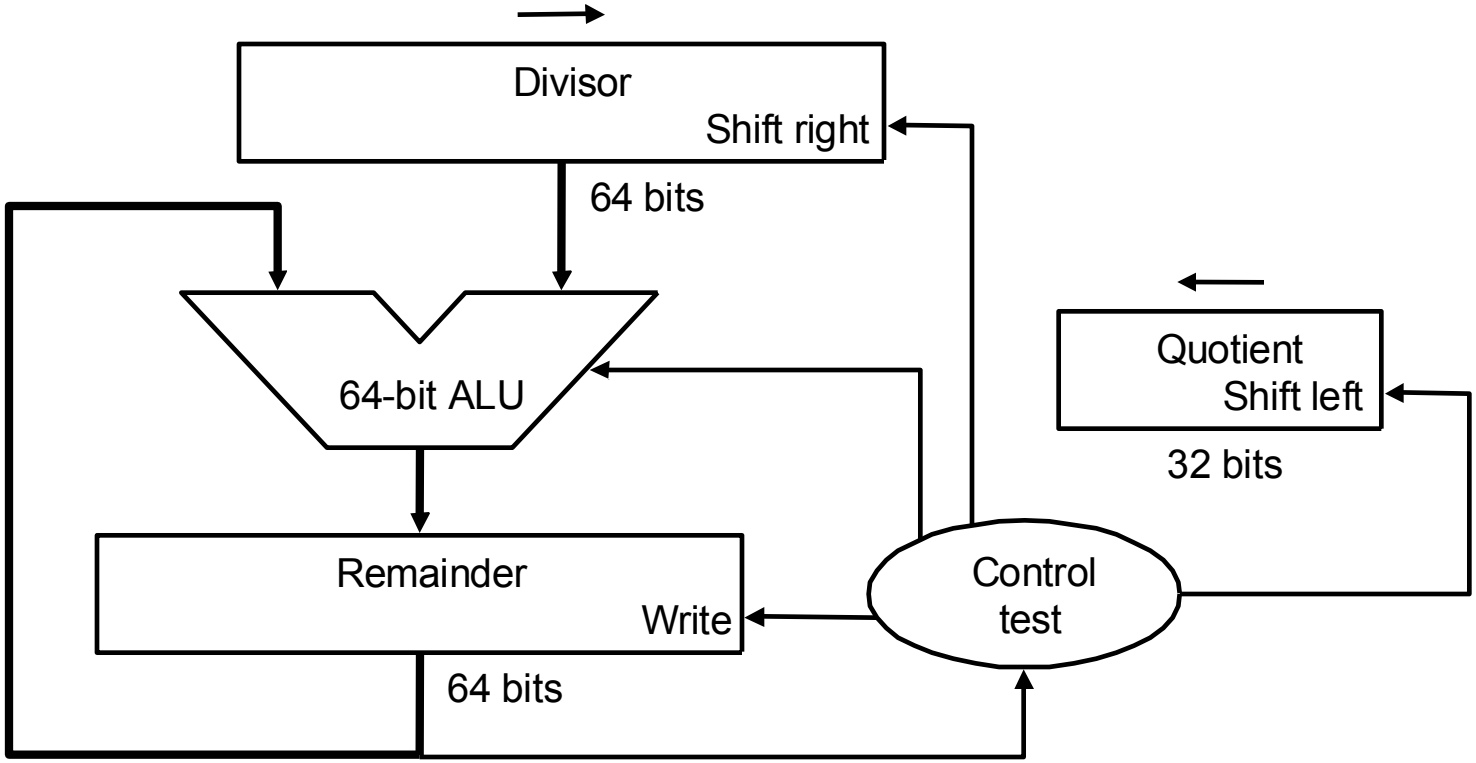
- Division is the reciprocal operation of multiplication
- **Paper-and-pencil example** (1001010_{ten} / 1000_{ten}):

```
                          1001      Quotient
Divisor  1000     |1001010          Dividend
                 -1000000
                    0010 10
                    0001 010
                     000 1010
                   -1000
                      10           Remainder
```

  - Dividend = Quotient x Divisor + Remainder

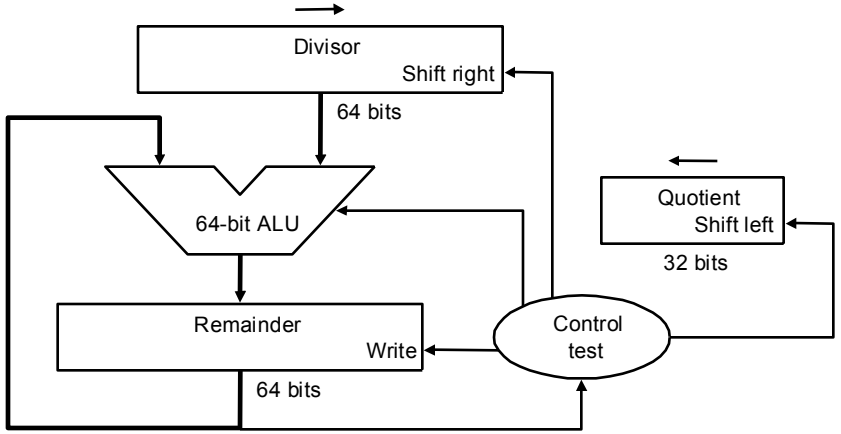
Division Hardware

- This version simply follows the flow of the paper-and-pencil example



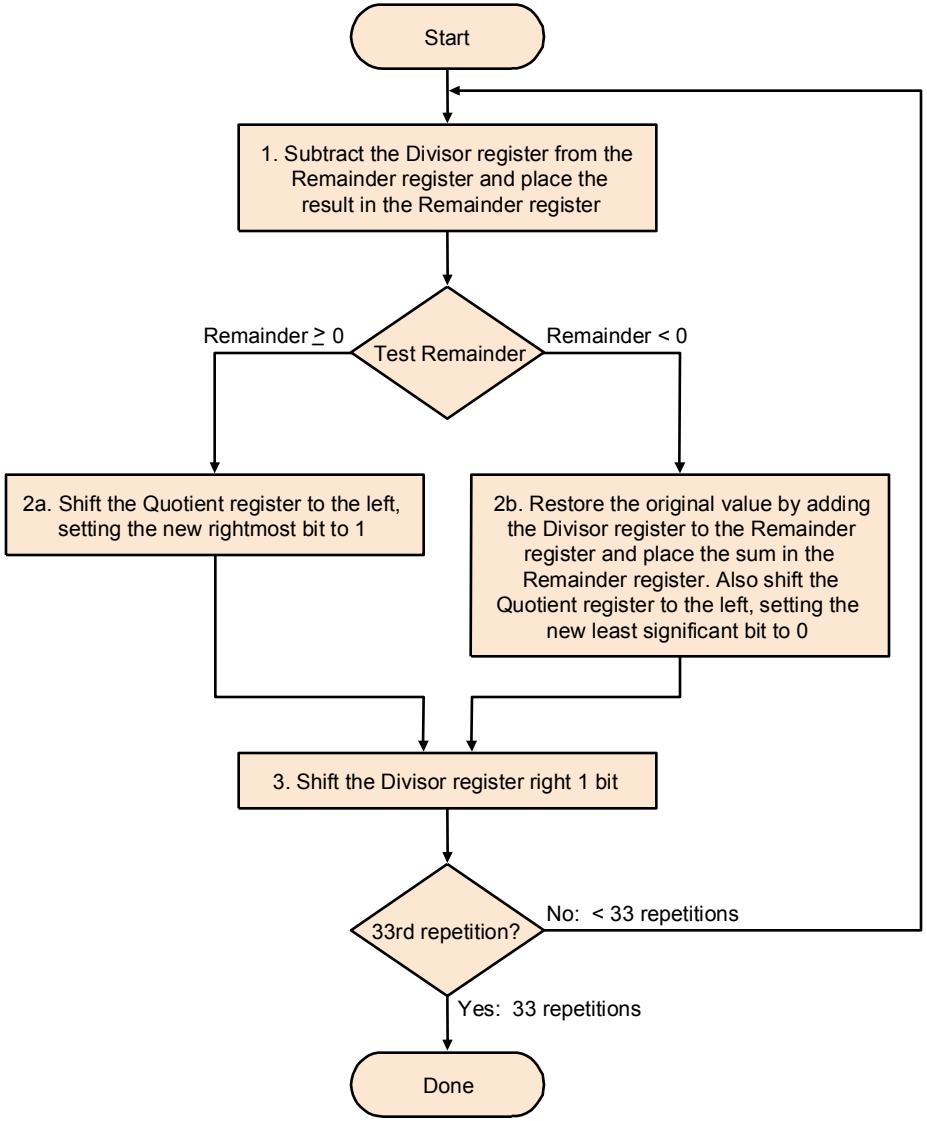
Division Hardware

- 64-bit ALU



- **Three registers**:
  - 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

Division Algorithm



Example

- **Paper-and-pencil example** (0000111_2 / 0010_2):

```
                        00011      Quotient
Divisor  0010     00000111        Dividend
                 -00100000
                 -00010000
                 -00001000
                 -00000100
                  00000011
                 -00000010
                  00000001        Remainder
```

Example

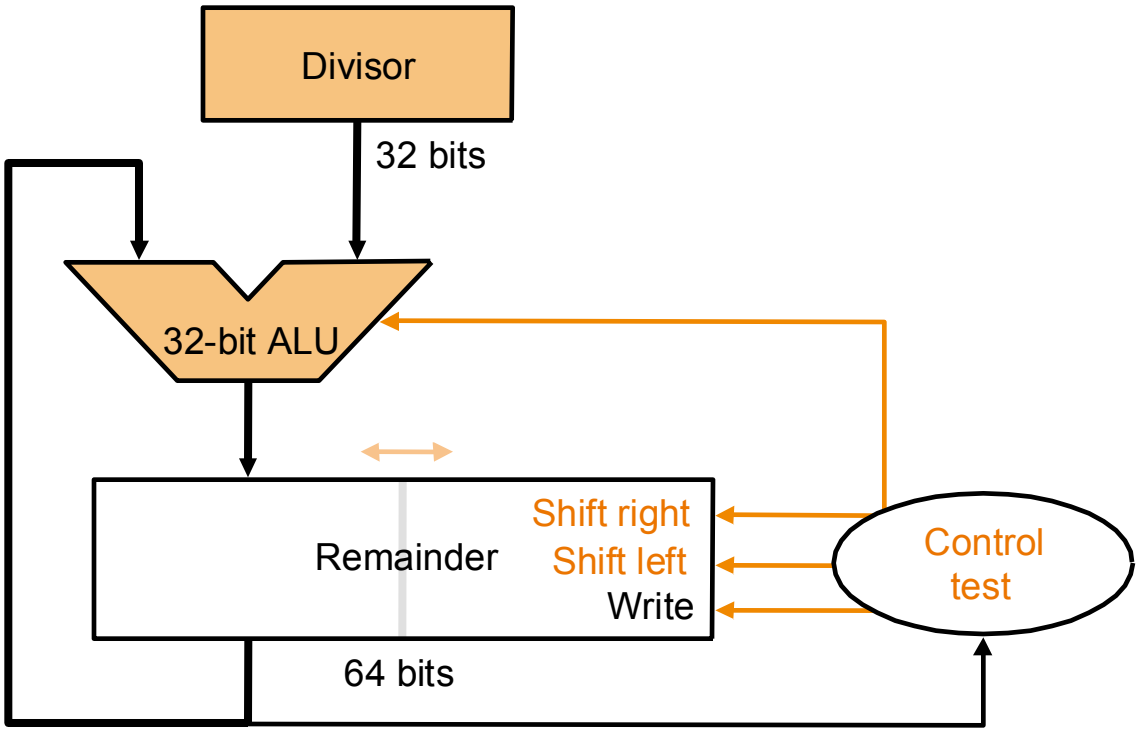
	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem <0 → +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem <0 → +Div, sll Q, Q0 = 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
	2b: 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
	2b: Rem>=0 → sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Some Observations

Like 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

Division Hardware – Improved Version



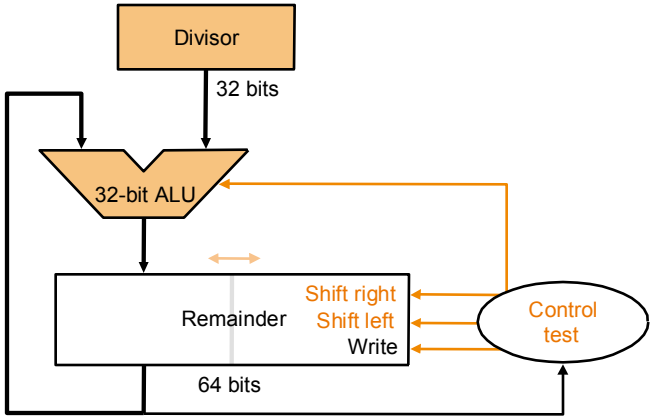
(changes made to previous version are highlighted in orange color)

Division Hardware – Improved Version

- **32-bit ALU**

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

    (right half also used for storing quotient)



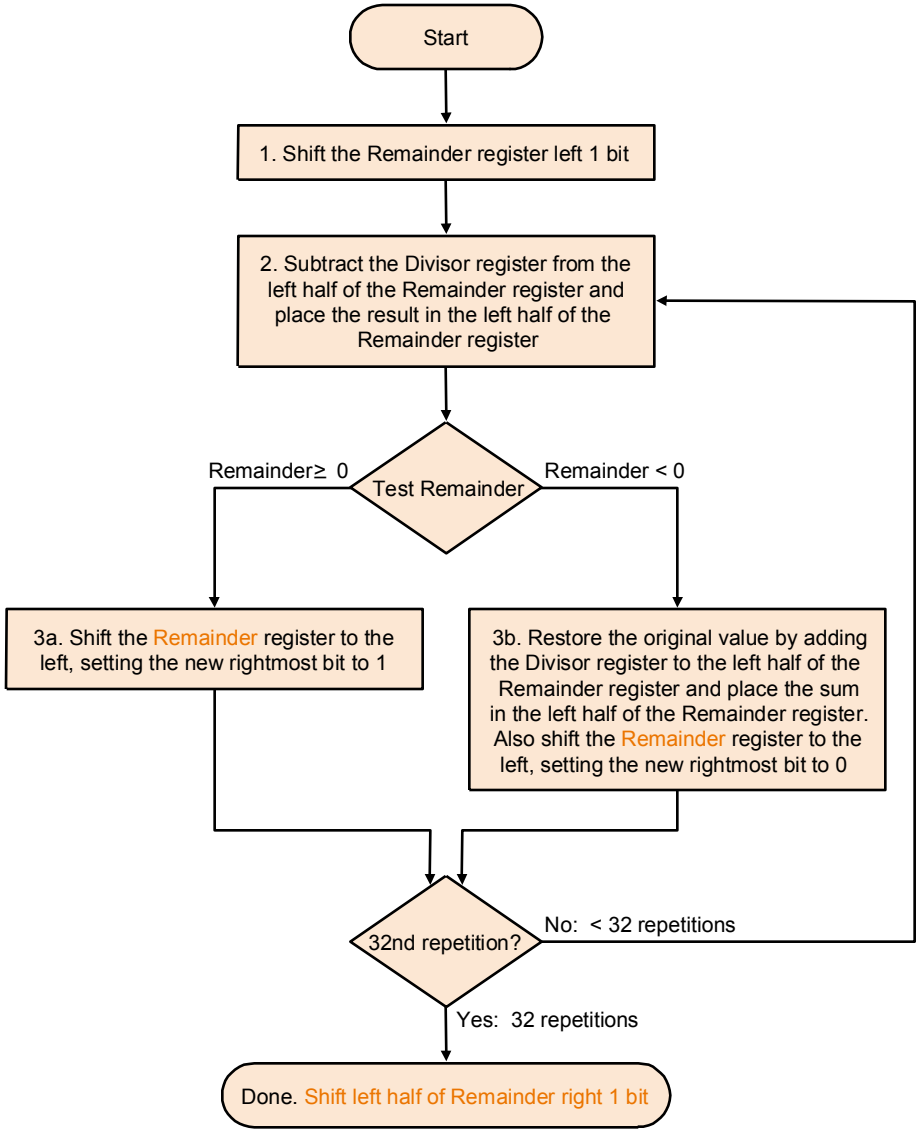
- **Operations**:
  - 32-bit divisor is always subtracted from the left half of remainder register
    - The result is written back to the left half of the remainder register
  - The right half of the remainder register is initialized with the dividend
    - Left shift remainder register by one before 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

Example

- **Paper-and-pencil example** (0000111_2 / 0010_2):

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

Division Algorithm – Improved 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

- Similar to signed multiplication, the signs of the divisor and dividend are checked to determine whether the results (quotient and remainder) should be negated.

- **Two rules to follow**:
  - 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

Divide in MIPS

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

- Examples:
  - `div  $s1, $s2  # Lo = $s1 / $s2; Hi = $s1 mod $s2`
  - `divu  $s1, $s2  # Lo = $s1 / $s2; Hi = $s1 mod $s2`

5. Floating Point Arithmetic

IEEE 754 Standard for floating point Arithmetic

Single precision:

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

Double precision:

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

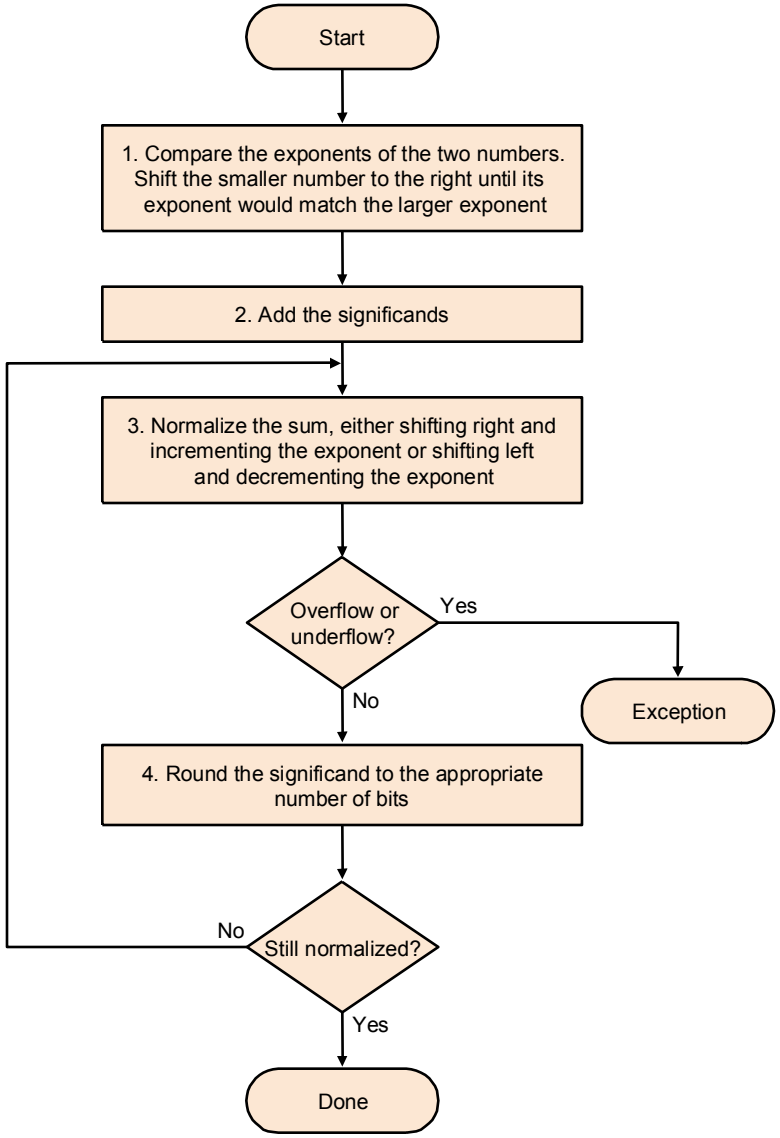
Floating-Point Addition

- Example: $9.999_{10} \times 10^{1} + 1.610_{10} \times 10^{-1}$
- Assumptions:
  - Significand size = 4 decimal digits
  - Exponent size = 2 decimal digits

Algorithm:

1. Align the decimal point of the number that has the smaller exponent
   - e.g. $1.610_{10} \times 10^{-1}$ becomes $0.016_{10} \times 10^{1}$
2. Add the significands of the two numbers together
   - e.g. $9.999_{10} \times 10^{1} + 0.016_{10} \times 10^{1} = 10.015_{10} \times 10^{1}$
3. Normalize the sum
   - e.g. $10.015_{10} \times 10^{1}$ becomes $1.0015_{10} \times 10^{2}$
4. Round the normalized sum
   - e.g. $1.0015_{10} \times 10^{2}$ becomes $1.002_{10} \times 10^{2}$

Floating-Point Addition



Binary Example

- Add 0.5_{10} and -0.4375_{10} in binary using the above algorithm
- Assume for simplicity that we only keep 4 bits of precision

- **Answer**:

- $0.5_{10} = 1.000_2 \times 2^{-1}$
- $-0.4375_{10} = -1.110_2 \times 2^{-2}$

1. $-1.110_2 \times 2^{-2} \Rightarrow -0.111_2 \times 2^{-1}$
2. $1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1}) = 0.001_2 \times 2^{-1}$
3. $0.001_2 \times 2^{-1} \Rightarrow 1.000_2 \times 2^{-4}$ (no overflow/underflow)
4. $1.000_2 \times 2^{-4}$ (fits in 4 bits, no need for rounding)

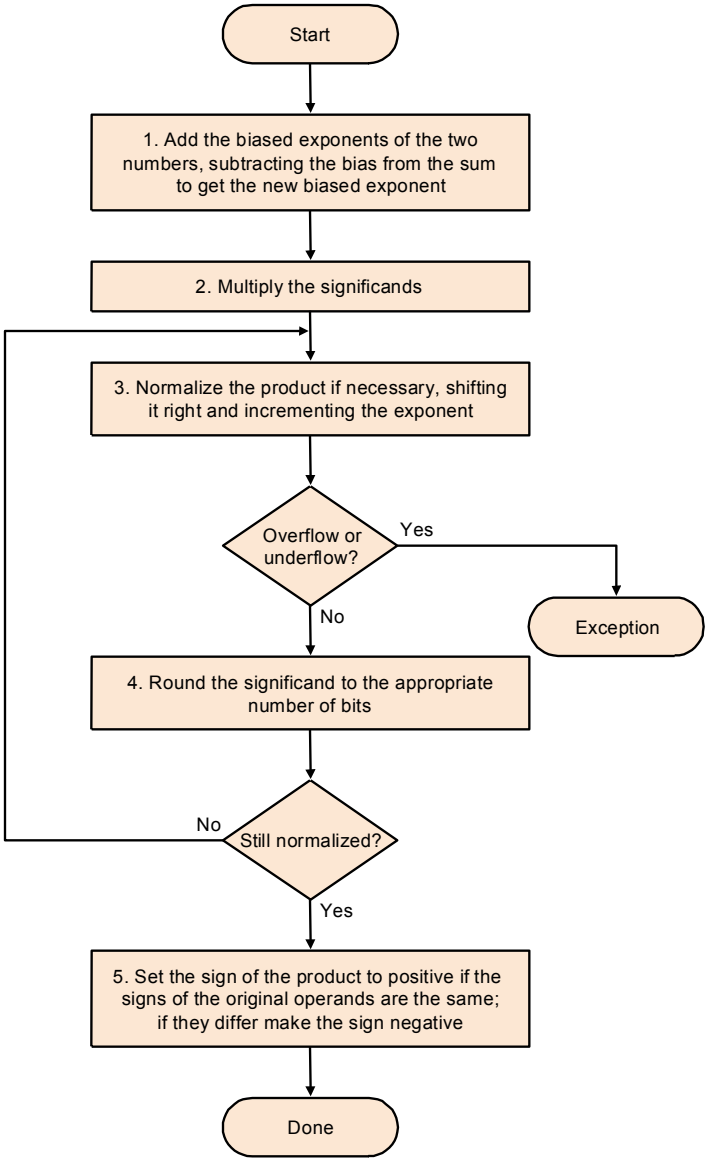
Floating-Point Multiplication

- Example: $(1.110_{10} \times 10^{10}) \times (9.200_{10} \times 10^{-5})$
- Assumptions:
  - Significand size = 4 decimal digits
  - Exponent size = 2 decimal digits

Algorithm:

1. Add the exponents together,
   - new exponent = 10 + (-5) = 5
2. Multiply the significands together
   - new significand = $1.110_{10} \times 9.200_{10} = 10.212_{10}$
3. Normalize the product,
   - $10.212_{10} \times 10^{5} \Rightarrow 1.0212_{10} \times 10^{6}$
4. Round the product
   - $1.0212_{10} \times 10^{6} \Rightarrow 1.021_{10} \times 10^{6}$
5. Find the sign of the product
   - $+1.021_{10} \times 10^{6}$

Floating-Point Multiplication



Binary Example

- Multiply 0.5_{10} and -0.4375_{10} in binary using the above algorithm
- Assume for simplicity that we only keep 4 bits of precision

- **Answer**:
  - $0.5_{10} = 1.000_2 \times 2^{-1}$
  - $-0.4375_{10} = -1.110_2 \times 2^{-2}$

- new exponent = $-1 + (-2) = -3$
- new significand = $1.000_2 \times 1.110_2 = 1.110_2$
- $1.110_2 \times 2^{-3}$ remains unchanged (no overflow/underflow)
- $1.110_2 \times 2^{-3}$ fits in 4 bits (no need for rounding)
- product = $-1.110_2 \times 2^{-3}$

MIPS Instructions for Floating-Point Operations

- 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

- 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
```

Accurate Arithmetic

- 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

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

Key Concepts to Remember

- **2's complement representation** for signed numbers

- 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** better 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

Key Concepts to Remember (2)

- **Floating-point numbers**
  - Representation follows closely the **scientific notation**
  - Almost all computers, including MIPS, follow **IEEE 754 standard**
- In MIPS,
  - **Single-precision** floating-point representation takes 32 bits
  - **Double-precision** floating-point representation takes 64 bits
  - Has a FP **co-processor** and separate **FP registers**
- **Overflow** (**underflow**) in floating-point representation occurs
  - When the exponent is too large (small) to be represented

Backup

Example

- Convert a temperature from Fahrenheit to Celsius

```
float f2c (float fahr)  {return ((5.0/9.0)*(fahr - 32))}
```

- Assume
  - Floating-point argument `fahr` is passed via `$f12`
  - Result go in `$f0`
  - Compiler places three floating-point constants in memory within easy reach of the global pointer `$gp`

MIPS Code:

```
lwc1      $f16, const5($gp)    # $f16 = 5.0 (5.0 in memory)
lwc1      $f18, const9($gp)    # $f18 = 9.0 (9.0 in memory)
div.s     $f16, $f16, $f18     # $f16 = 5.0/9.0
lwc1      $f18, const32($gp)   # $f18 = 32.0
sub.s     $f18, $f12, $f18     # $f18 = fahr - 32.0
mult.s    $f0, $f16, $f18      # $f0 = (5/9)*(fahr-32.0)
jr        $ra                  # return
```

Arithmetic Unit for Floating-Point Addition