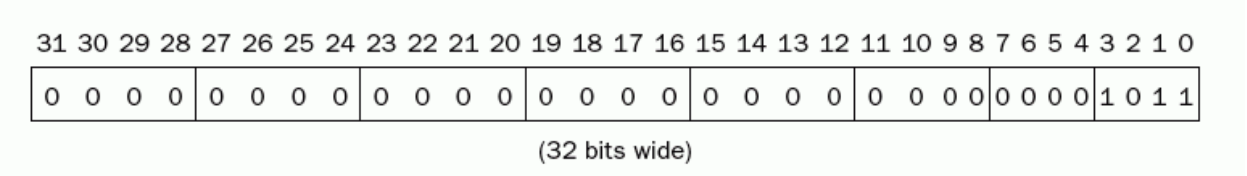
Ch4: Binary Arithmetic and ALU Design

Introduction

- Computer words are composed of bits; thus words can be represented as binary numbers.
  - How are negative numbers represented?
  - What is the largest number that can be represented in a computer word?
  - What happens if an operation creates a number bigger than can be represented?
  - What about fractions and real numbers?

Signed and Unsigned Numbers

- The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two}:



- The phrase least significant bit is used to refer to the right most bit (bit 0 above) and most significant bit to the left most bit (bit 31).
- The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns (from 0 to $2^{32} - 1$).

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$
$$\ldots \qquad\qquad \ldots$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$$

- Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative.

Possible Representations

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = - 0	100 = - 3	100 = - 4
101 = - 1	101 = - 2	101 = - 3
110 = - 2	110 = - 1	110 = - 2
111 = - 3	111 = - 0	111 = - 1

- Sign and magnitude representation has several shortcomings:
  1. It's not obvious where to put the sign bit. To the right? To the left?
  2. Adders for sign and magnitude may need an extra step to set the sign.
  3. A separate sign bit means that sign and magnitude has both a positive and negative zero, which can lead to problems for inattentive programmers.
- Two's complement representation make the hardware simple and remove the ambiguity (each number has unique representation)
- MIPS: 32-bit signed integers

$$
\begin{aligned}
&0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_{two} = 0_{ten}\\
&0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} = 1_{ten}\\
&0000\,0000\,0000\,0000\,0000\,0000\,0000\,0010_{two} = 2_{ten}\\
&\ldots \qquad\qquad\qquad\qquad \ldots\\
&0111\,1111\,1111\,1111\,1111\,1111\,1111\,1101_{two} = 2{,}147{,}483{,}645_{ten}\\
&0111\,1111\,1111\,1111\,1111\,1111\,1111\,1110_{two} = 2{,}147{,}483{,}646_{ten}\\
&0111\,1111\,1111\,1111\,1111\,1111\,1111\,1111_{two} = 2{,}147{,}483{,}647_{ten}\\
&1000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_{two} = -2{,}147{,}483{,}648_{ten}\\
&1000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} = -2{,}147{,}483{,}647_{ten}\\
&1000\,0000\,0000\,0000\,0000\,0000\,0000\,0010_{two} = -2{,}147{,}483{,}646_{ten}\\
&\ldots \qquad\qquad\qquad\qquad \ldots\\
&1111\,1111\,1111\,1111\,1111\,1111\,1111\,1101_{two} = -3_{ten}\\
&1111\,1111\,1111\,1111\,1111\,1111\,1111\,1110_{two} = -2_{ten}\\
&1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111_{two} = -1_{ten}
\end{aligned}
$$

  - The positive half of the numbers, from 0 to 2^{31}-1
  - The negative half of the numbers, from -1 to -2^{31}
  - The bit pattern ($10000\ldots00000_{two}$) represent the most negative number -2^{31}

Prepared By: Eng. Randa Al_Dallah

- The 2's complement is imbalance (the negative number -2^{31} has no corresponding positive number). (problem for programmer)
    - Sign and magnitude had problems for both the programmer and the hardware designer.
    - Consequently, every computer today uses two's complement binary representations for signed numbers
- We can represent positive and negative 32- bit numbers in terms of the bit value times a power of 2:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

$$x = -B_{n-1} \bullet 2^{n-1} + \sum_{i=0}^{n-2} B_i \bullet 2^i$$

- Example:

What is the decimal value of this 32-bit two's complement number?

$$1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1100_{two}$$

Substituting the number's bit values into the formula above:

$$\begin{aligned}&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})\\ &= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0\\ &= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}\\ &= -4_{ten}\end{aligned}$$

Load/Add signed and unsigned numbers

- The function of a signed load is to copy the sign repeatedly to fill the rest of the register—called ***sign extension***
    - Its purpose is to place a correct representation of the number within that register.
- Unsigned loads simply fill with 0s to the left of the data.
- When loading a 32-bit word into a 32-bit register, signed and unsigned loads are identical.
- MIPS offer two flavors of byte loads:
    - Load byte (lb) treats the byte as a signed number and thus sign-extends to fill the 24 left most bits of the register
    - Load byte unsigned (lbu) works with unsigned integers
    - MIPS also offer lh and lhu

- The immediate field in the load, store, branch, add, and set on less than instructions contains a 2's complement 16-bit number, representing -2^{15} to $2^{15} - 1$.

- To add the immediate field to a 32-bit register, the computer must convert that 16-bit number to its 32-bit equivalent.
    - Take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. (sign extension)

- Example: convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

  The 16-bit binary version of the number 2 is
  $0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

  The 32-bit binary is
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

  The 16-bit binary version of the number -2 is
  $1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

  The 32-bit binary is
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

- Example:
    - If the immediate in the addi instruction = 1111 1111 1111 1110, the sign is extended. i.e. –2 is used in the addition.
    - For unsigned addition operation addiu, sign is not extended. i.e. 65534 (why this value?) is used in the addition.

Signed versus Unsigned Comparison

- In signed numbers, a 1 in the most significant bit represents a negative number and is less than any positive number, which must have a 0 in the most significant bit.
- In unsigned number, a 1 in the most significant bit represents a number that is larger than any that begins with a 0.
- MIPS offers two versions of the set on less than comparison:
    - slt and slti work with signed integers
    - sltu and sltiu work with unsigned integers.

- Example:

Suppose register $s0 has the binary number

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register $s1 has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers $t0 and $t1 after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

Solution:

- The value in register $s0 represents -1 if it is integer and $4,294,967,295_{ten}$ if it is an unsigned integer.
- The value in register $s1 represents 1 in either case.
- The register $t0 has the value 1, since $-1_{ten} < 1_{ten}$
- The register $t1 has the value 0, since $4,294,967,295_{ten} > 1_{ten}$

Addition and Subtraction

- Example: adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

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

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \end{array}$$

Or via addition using the two's complement representation of -6:

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} & = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \end{array}$$

Overflow

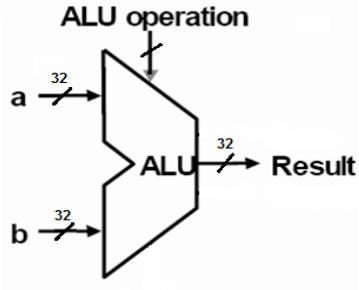
- The overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word.
- When can overflow occur in addition?
  - When adding operands with different signs, overflow cannot occur.
- When can overflow occur in subtraction? (opposite principle)
  - When the signs of the operands are the same, overflow cannot occur.
  - Ex: x-y = x+(-y). so, when we subtract operands of the same sign we end up by adding operands of different signs.
- How to detect when the overflow occur?
  - Overflow occurs when adding two positive numbers and the sum is negative, or vice versa.
  - Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means a borrow occurred from the sign bit

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

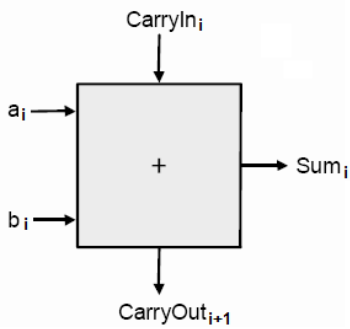
- What about unsigned integers?
  - Unsigned integers are commonly used for memory addresses where overflows are ignored.
- MIPS have two kinds of arithmetic instructions to recognize the two choices:
  - Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
  - Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do not cause exceptions on overflow.
- MIPS detects overflow with an exception, also called an interrupt on many computers.
  - The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception.

Constructing an Arithmetic Logic Unit

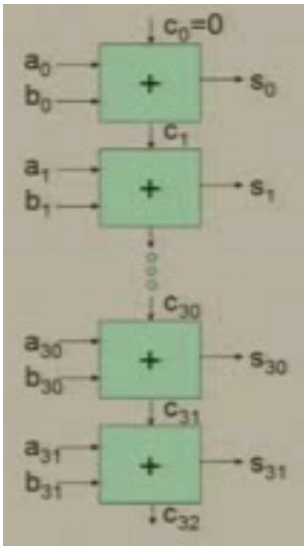
- Implementing the architecture. (How basic operations are done in the hardware).



Adder Circuit



1-bit Addition



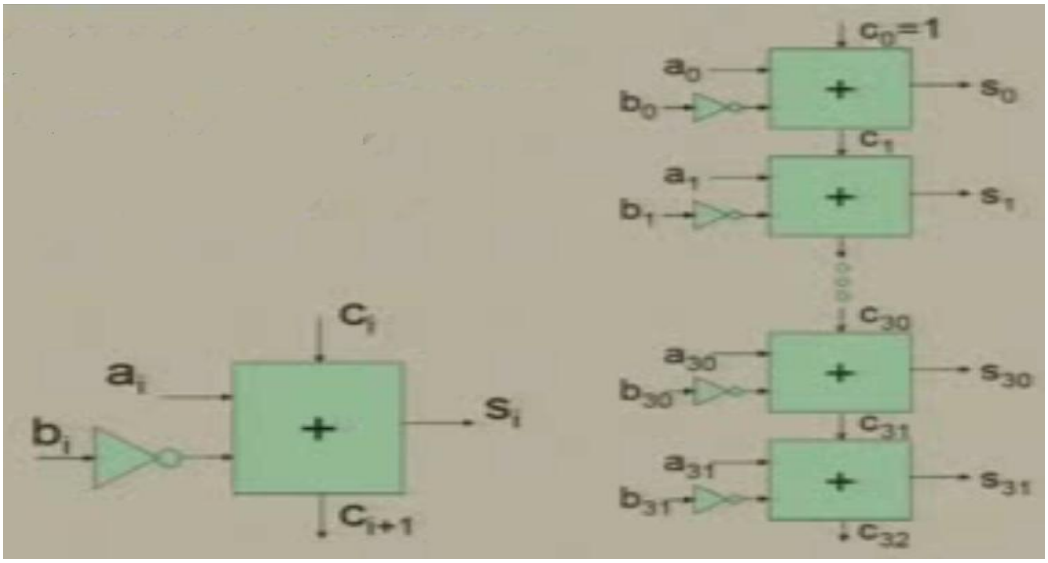
- Boolean expressions for adder:

$Sum_i = a_i \oplus b_i \oplus c_i$

$CarryOut_{i+1} = a_i b_i + a_i c_i + b_i c_i$

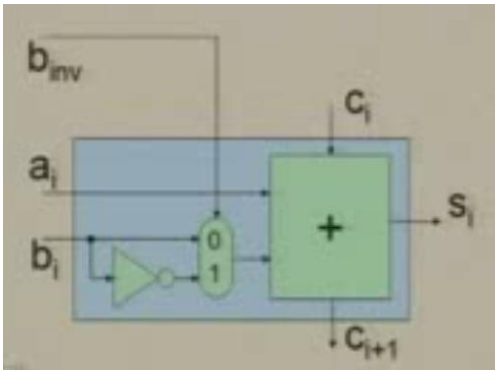
Subtraction Circuit

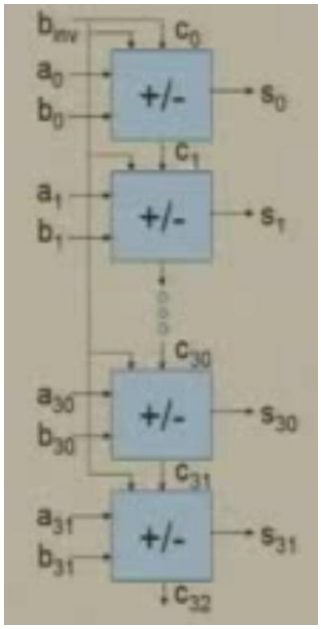
- x – y = x + y' +1



Combining Addition and Subtraction

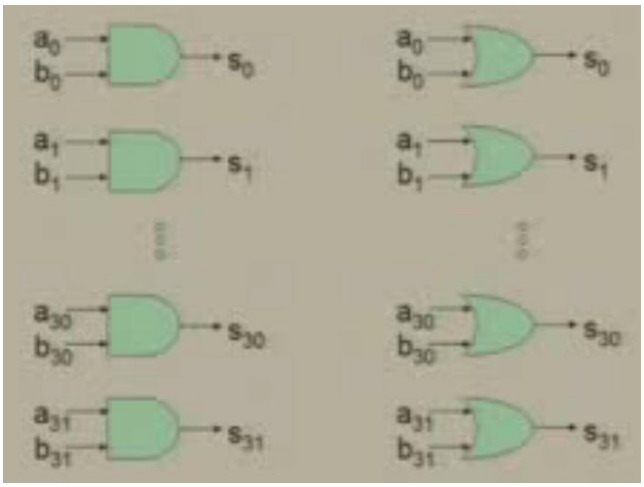
- Use a multiplexer circuit



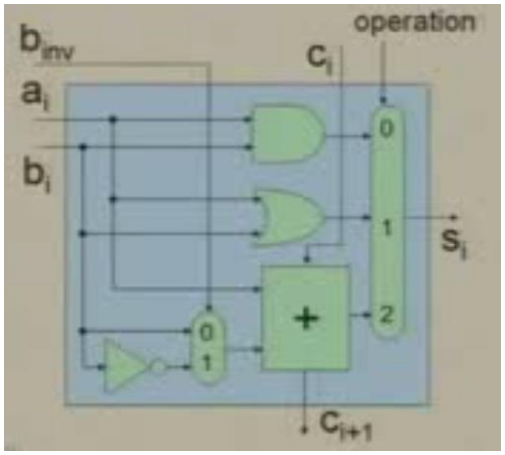


Logical Operations: AND, OR

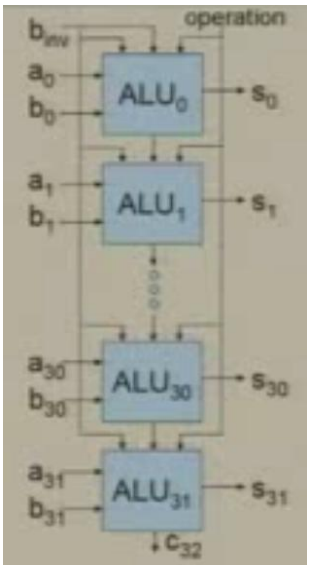
- MIPS logical instructions require bit by bit operation on 32 bit strings.
- Circuit for “and”, “or” instructions



Combining and, or, add, sub

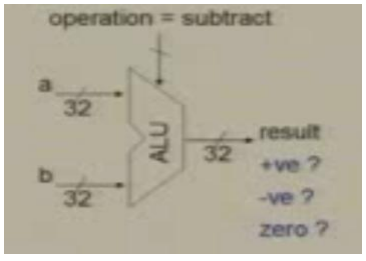


1-bit unit of ALU (ignore the detection of overflow)

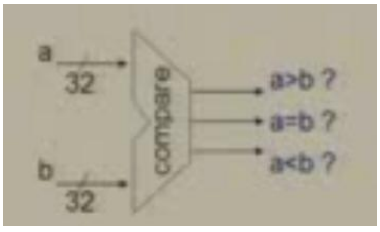


Comparing two integers

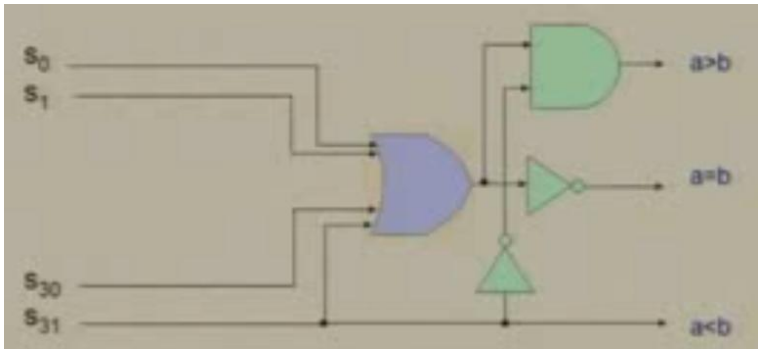
1. Subtract and check the result



2. Compare directly

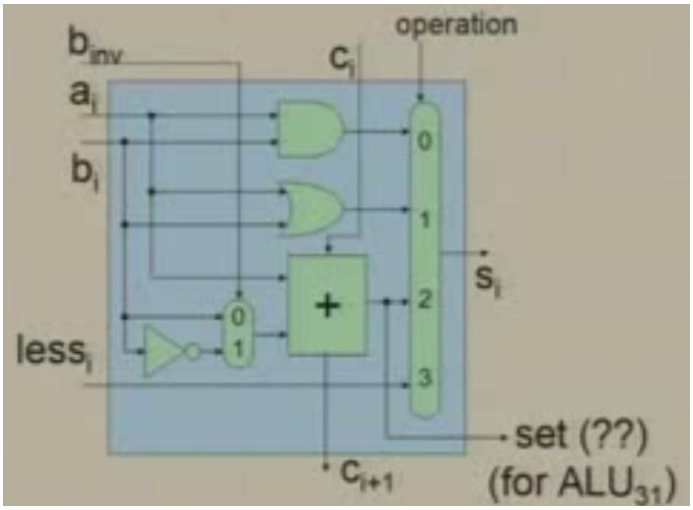


Subtract and check the result



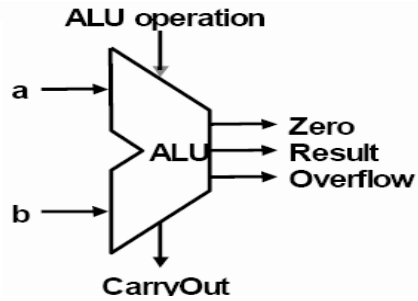
- S_0 .. S_{31} are the subtract output
- S_{31} is a sign bit

Extending ALU_i for "slt" instruction



ALU Universal Representation

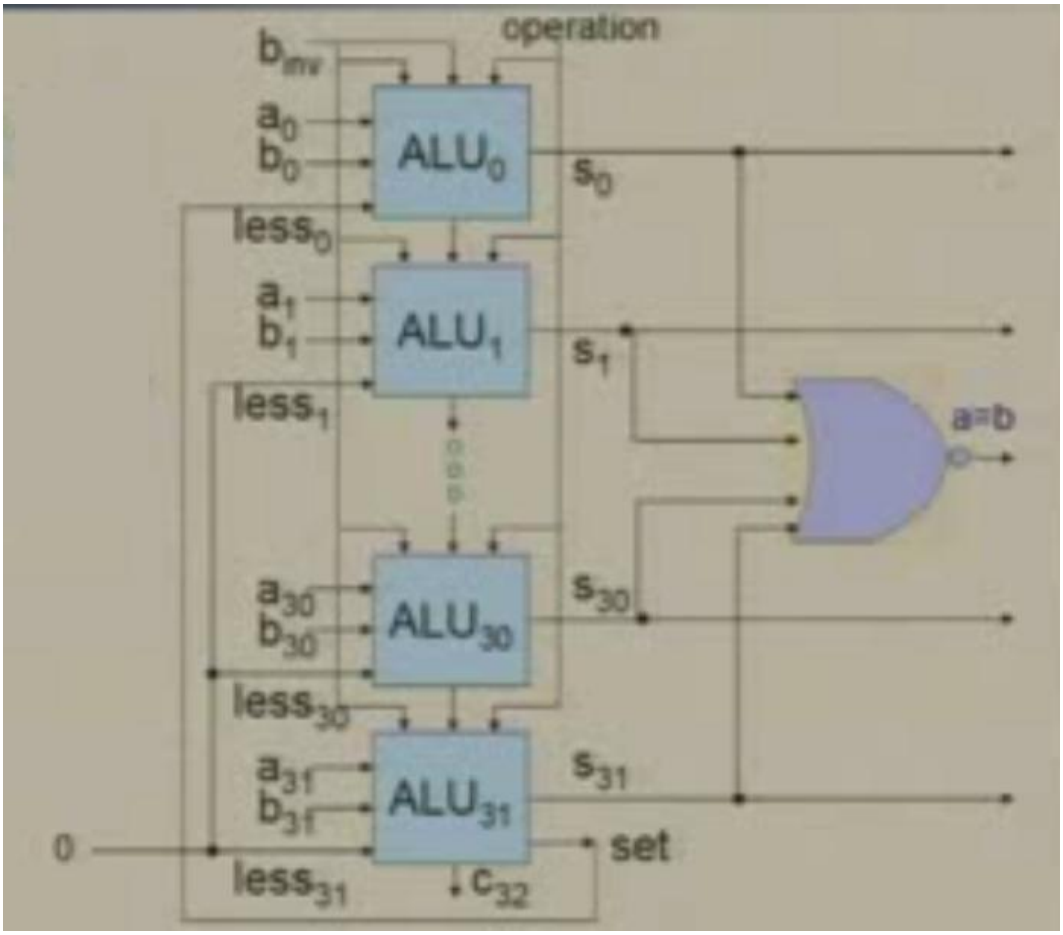
- universal symbol for a complete ALU as follows:



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

Prepared By: Eng. Randa Al_Dallah

ALU for and, or, add, sub, beq, slt



Multiplier Design

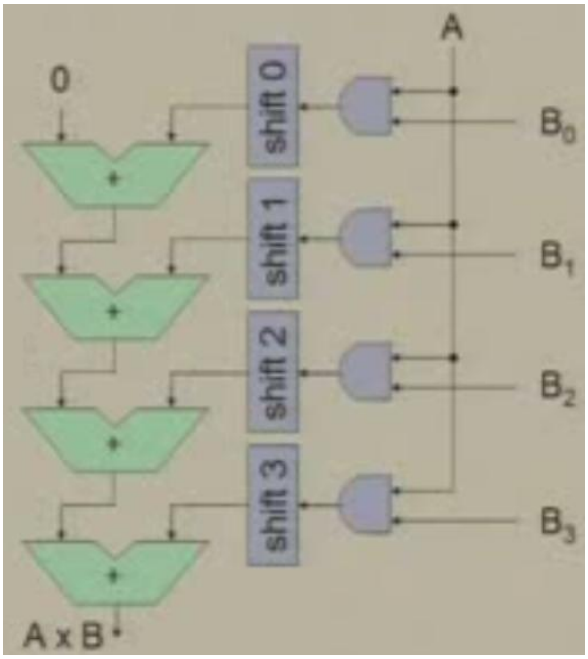
- Shift and add multiplication
- Multiplication: paper-pencil method

A			0	1	1		0	0	0	0	0	Multiplicand
B			1	0	1		0	0	0	1	1	Multiplier
			---	---	---		0	0	0	1	1	
			0	1	1							
		0	0	0	x							
	0	1	1	x	x							
	---	---	---	---	---							
AxB	0	1	1	1	1		0	1	1	1	1	Product

- The first operand is called the multiplicand and the second the multiplier. The final result is called the product.
- If we ignore the sign bits, the length of the multiplication of n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long.
- In binary multiplication there are only two choices, each step of the multiplication is simple:
  1. Place a copy of the multiplicand in the proper place if the multiplier digit is a 1, or
  2. Place 0 in the proper place if the digit is 0.

- The circuit for shift_add multiplication requires n adders

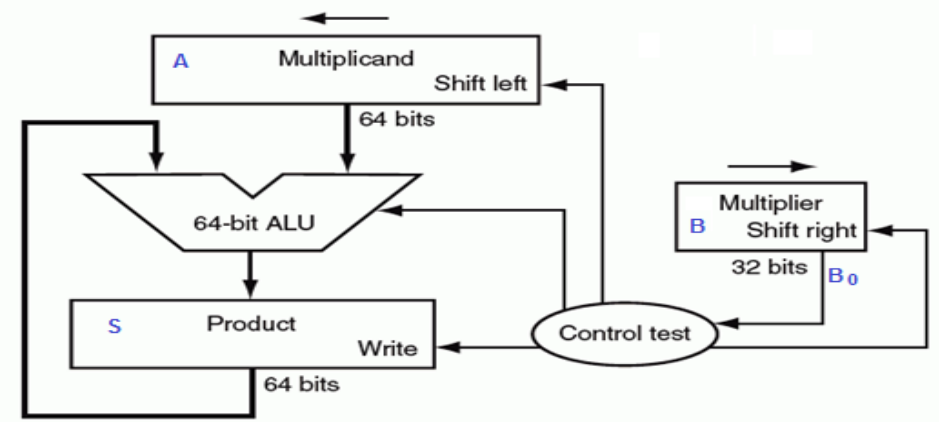
$$A \times B = \sum_{i=0}^{n-1} A \bullet B_i \times 2^i$$



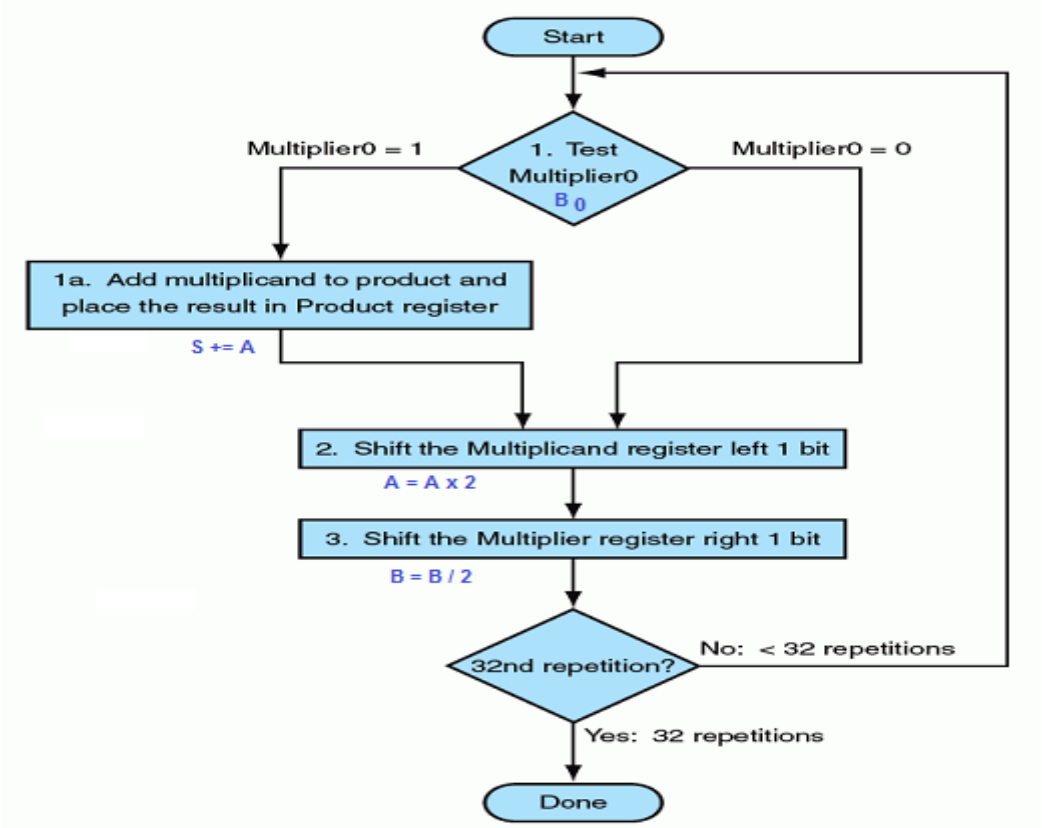
First Version of the multiplication hardware

- We can simplify the shift_add multiplier circuit by using only one adder as follows

Step 1: i=0; s=0
Do{
Step2:
$S+=A.B_i \times 2^i$
i++
}while(i<n)
	Step 1: i=0; s=0
	Do{
	Step2:
	If (B_i) S+=A
	A=2xA
	i++
	}while(i<n)
		Step 1: i=0; s=0
		Do{
		Step2:
		If (B_0) S+=A
		A=2xA
		B=B/2
		i++
		}while(i<n)



First Version of the multiplication hardware

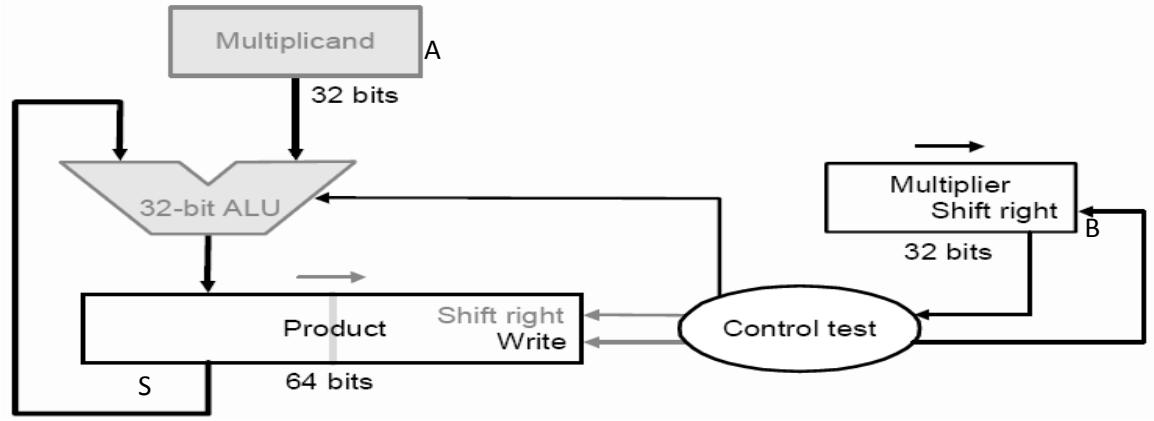


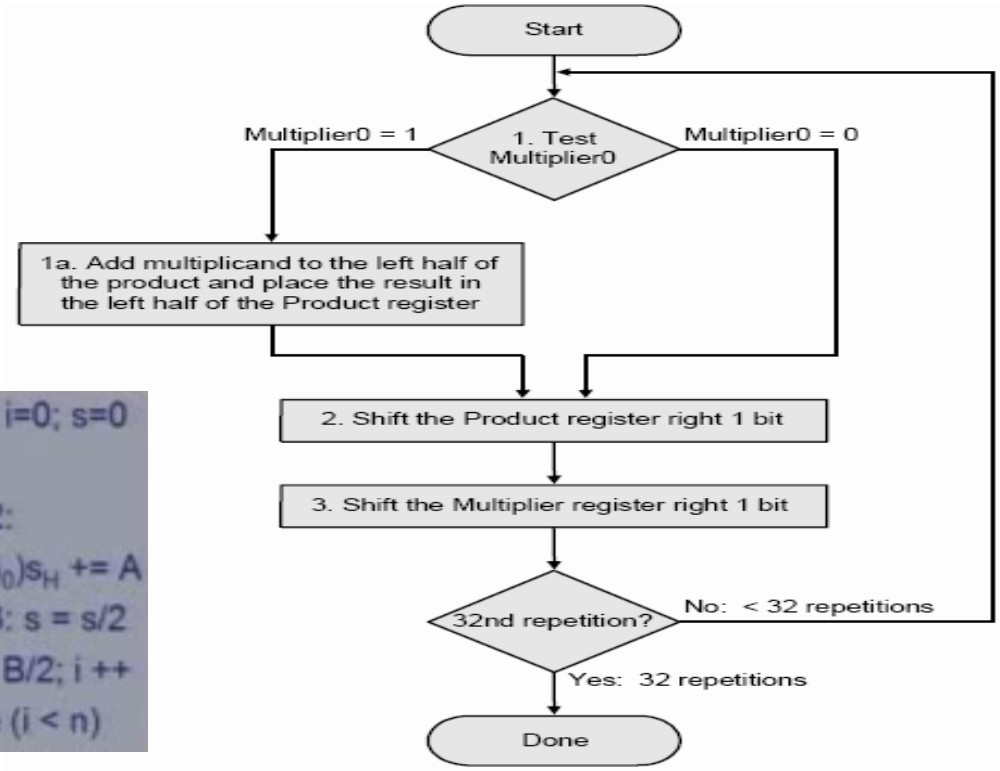
- 64-bit ALU
- Three registers:
  - Multiplicand register: 64 bits
  - Multiplier register: 32 bits
  - Product register: 64 bits
- Operations:
  - The 32-bit multiplicand starts in the right half of the multiplicand register, and is shifted left 1 bit at each step.
  - The multiplier register is shifted 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.

- Example: Using 4-bit numbers to save space, multiply 2_{ten} x 3_{ten}, or 0010_{two} x 0011_{two}

Iteration	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

Second Version of the multiplication hardware





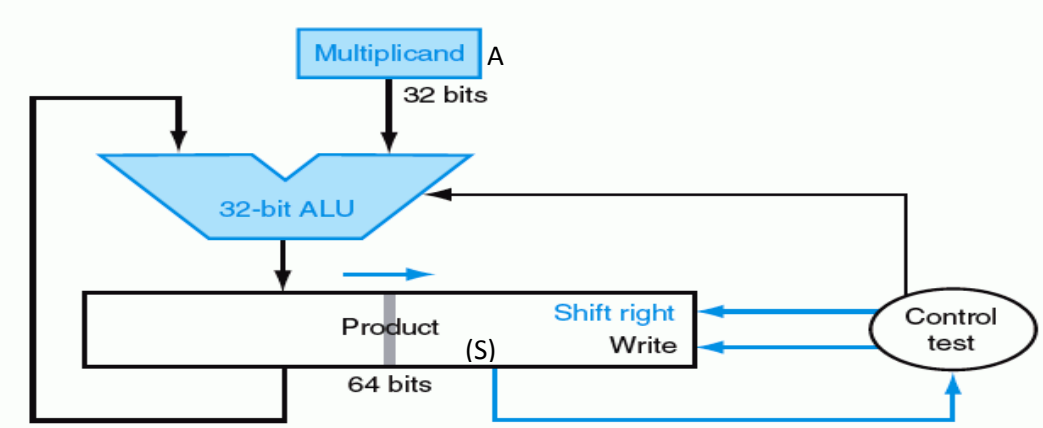

```
step1: i=0; s=0
do {
  step2:
    if (B0)sH += A
  step3: s = s/2
    B = B/2; i ++
} while (i < n)
```

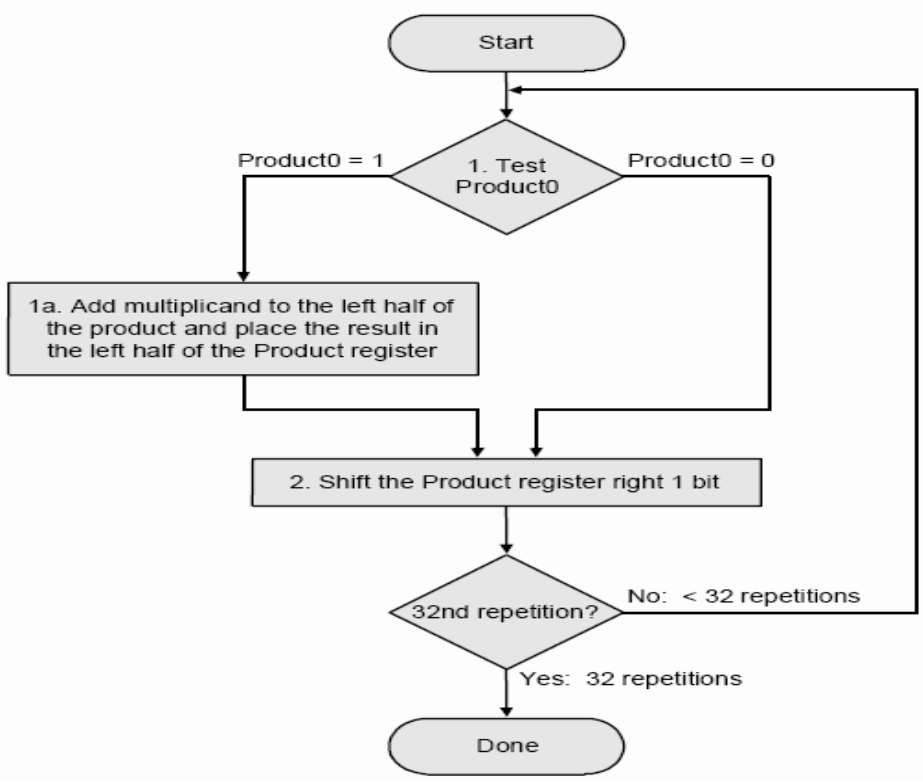
- 32-bit ALU
- Three registers:
  - Multiplicand register: 32 bits
  - Multiplier register: 32 bits
  - Product register: 64 bits
- Operations:
  - Instead of shifting the multiplicand register left, this version shifts the product register right 1 bit at each step.
  - The 32-bit multiplicand is always 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.

- This version only needs a 32-bit multiplicand register and a 32-bit ALU

Final Version of the multiplication hardware

- The loop in previous algorithms contains 3 statements which are repeated 32 times to obtain the product. If each statement took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers.
- The previous algorithm and hardware are easily refined to take 1 clock cycle per step.
    - By performing the operations in parallel






```
step1: i=0; s=0|B
do {
  step2:
    if (s0) sH += A
  step3:
    s = s/2; i ++
} while (i < n)
```

- 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.
- This version combines the right half of the product register with the multiplier register.

- **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	Left(P) = Left(P) + M
		0011 0001	P = P >> 1
2		1001 0001	Left(P) = Left(P) + M
		0100 1000	P = P >> 1
3		0100 1000	No operation
		0010 0100	P = P >> 1
4		0010 0100	No operation
		0001 0010	P = P >> 1

More on multiplication

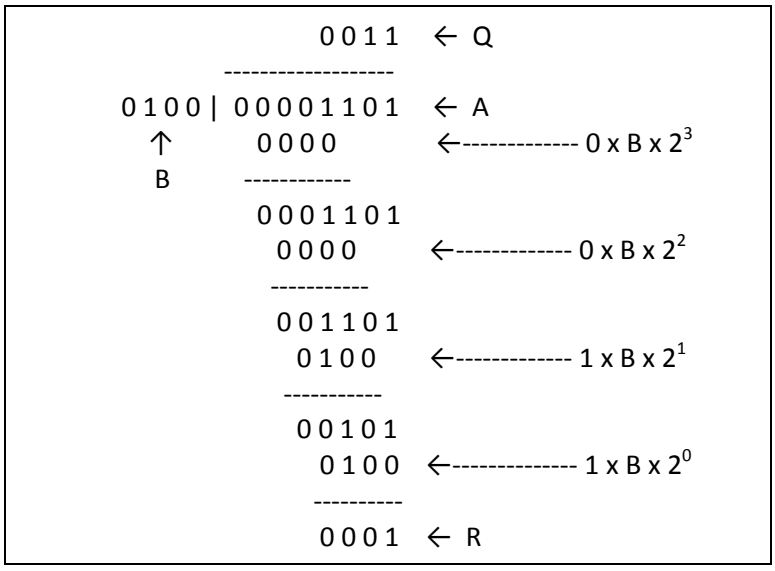
- If the multiplicand or multiplier is negative, we first negate it to get a positive number.
- Use any one of the above methods to compute the product of two positive numbers.
- The product should be negated if the original signs of the operands disagree.
- What is booth's algorithm? **HW**

Multiply in MIPS

- MIPS provide a separate pair of 32-bit registers to contain the 64-bit product, called Hi and Lo.
- To produce a properly signed or unsigned product, MIPS has two instructions: multiply (mult) and multiply unsigned (multu).
- To place the product into registers, we can use move from lo (mflo) and move from hi (mfhi) instructions.

Divider Design

- Division example:



First Version of the division Hardware

- A is a dividend, B is a divisor, Q is a quotient and R is a remainder.
- Algorithm:

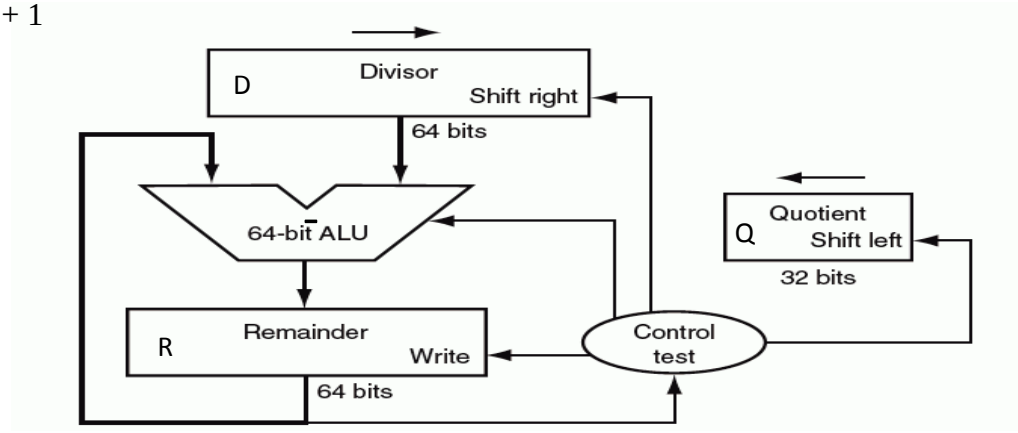
Step1: i = 0; R = A; Q = 0; D = B
Do {
Step2:
If (D x 2^{n-i-1} <= R) R = R – D x 2^{n-i-1} ; Q_{n-i-1} = 1
Else Q_{n-i-1} = 0
i++
} while (i < n)

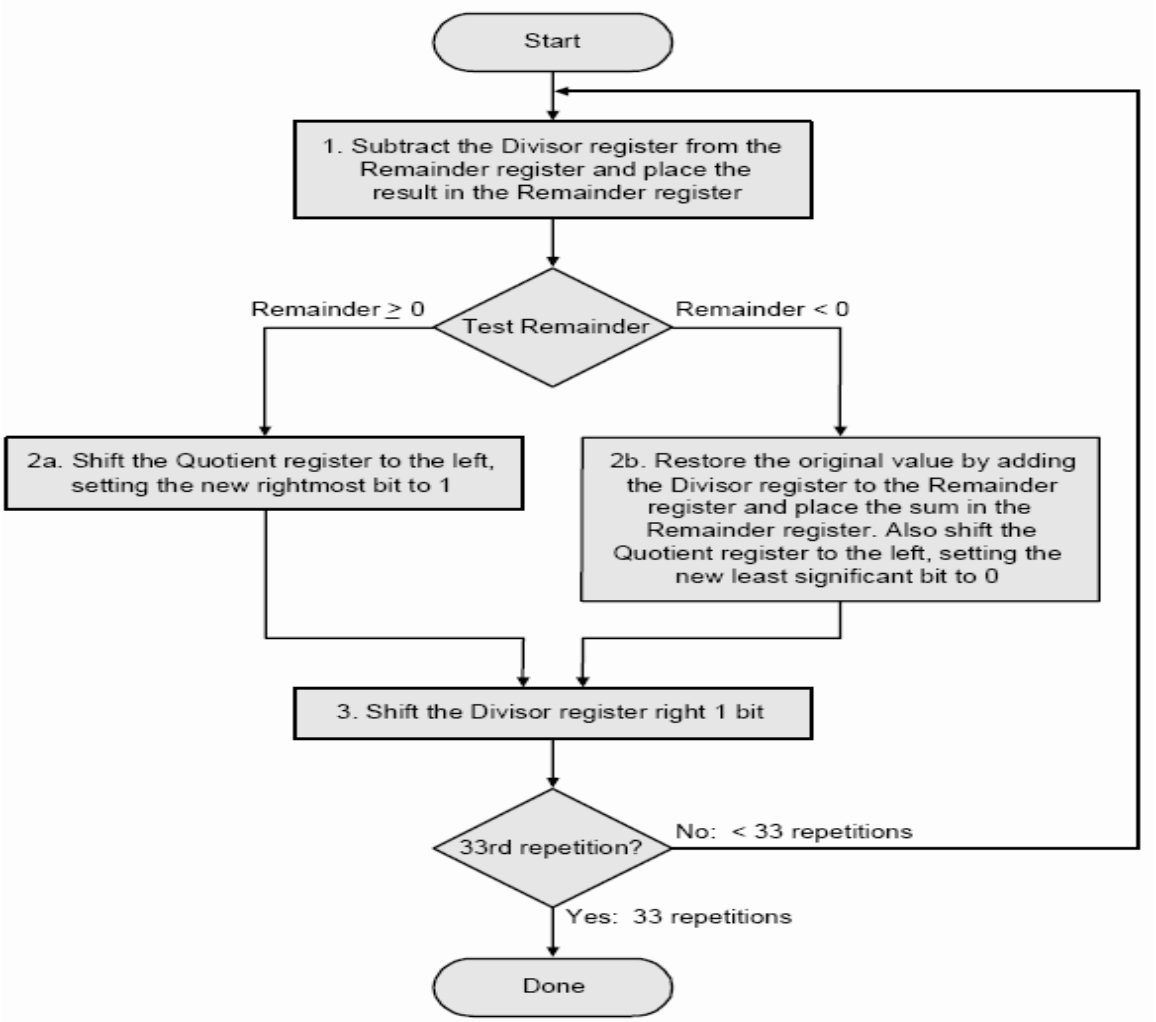
- Example:

1. *Initializing the variables*
   R=1101, Q=0, D=0100, i=0
2. *Iteration 1:*
   (0100000 <= 1101) ? → false
   Q = 0, i=1
   (i < 4) ? → true
3. *Iteration 2:*
   (010000<=1101) ? → false
   Q = 0, i = 2
   (i<4) ? → true
4. *Iteration 3:*
   (01000 <= 1101) ? → true
   R=1101 – 01000 = 0101, Q=001, i = 3
   (i<4) ? → true
5. *Iteration 4:*
   (0100 <= 0101) ? → true
   R= 0101 – 0100 = 0001, Q = 0011, i=4
   (i<4)? → false → stop

- Modify the algorithm to make it suitable for hardware (by introducing shift registers).

Step1: i = 0; R = A; Q = 0; D = B x 2^{n-1}
Do {
Step2:
R = R - D
If (R < 0) R = R + D; Q = 2 x Q
Else Q = 2 x Q + 1
D = D / 2; i++
} while (i < n)





- Dividend = Quotient x Divisor + Remainder
- 64-bit ALU
- Three registers:
  - Divisor register: 64 bits
  - Quotient register: 32 bits
  - Remainder register: 64 bits
- Operations:
  - The 32-bit divisor starts in the left half of the divisor register, and is shifted right 1 bit at each step.
  - The quotient register is initialized to 0, and is shifted left 1 bit at each step.
  - The remainder register is initialized with the dividend.
  - Control decides when to shift the divisor and quotient registers and when to write new values into the remainder register.
- **Example**: using a 4-bit version of the algorithm, let's try dividing 7_{ten} by 2_{ten}, or 00000111_{two} by 0010_{two}

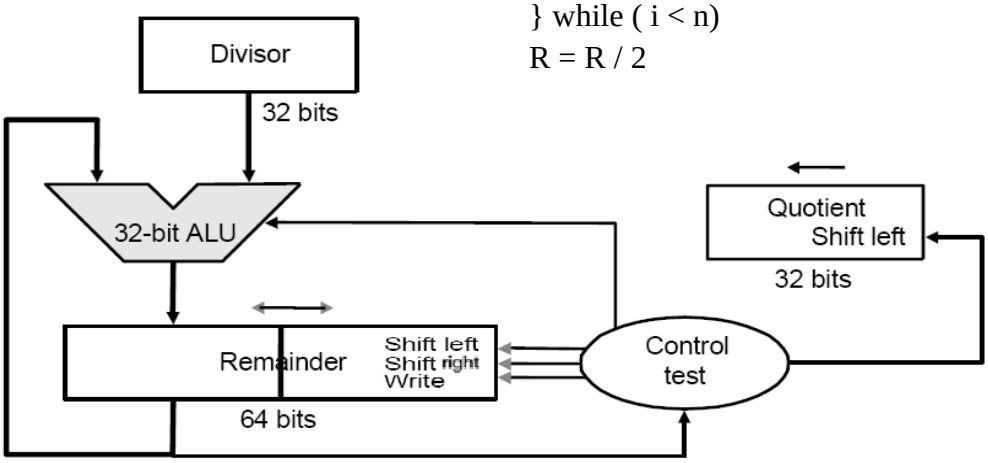
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div,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

Second Version of the division Hardware

- Like the first version of the multiplication hardware, at most half of the divisor register has useful information, and so both the divisor register and ALU could potentially be cut in half.
- Shifting the remainder register to the left instead of shifting the divisor register to the right produces the same alignment and accomplishes the goal of simplifying the hardware necessary for the ALU and the divisor register.

- Reducing subtractor size

Step1: i = 0; R = 2 x A; Q = 0; D = B
Do {
    Step2:
      $R_H = R_H - D$
      If ($R_H < 0$) $R_H = R_H + D$; Q = 2 x Q
      Else Q = 2 x Q +1
      R = 2 x R; i++
} while (i < n)
R = R / 2

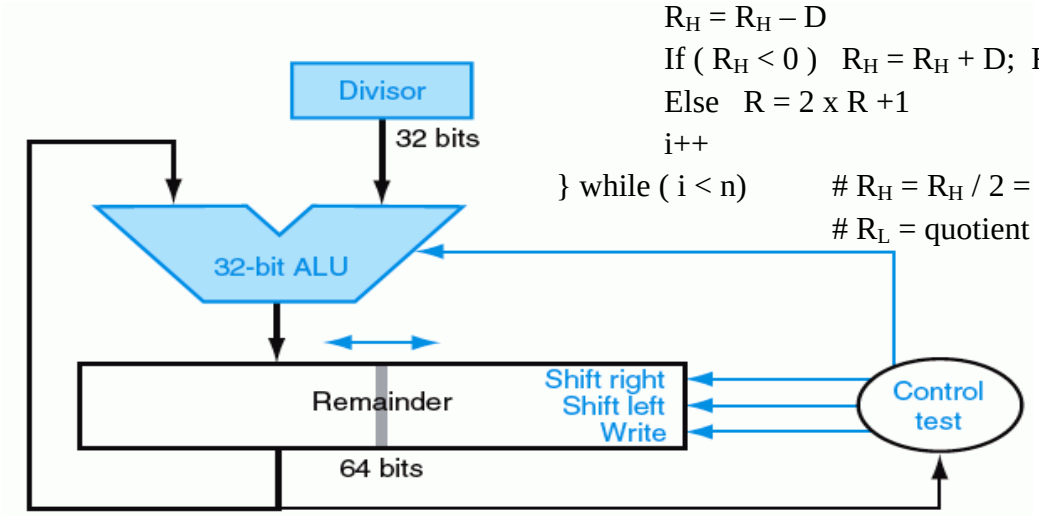


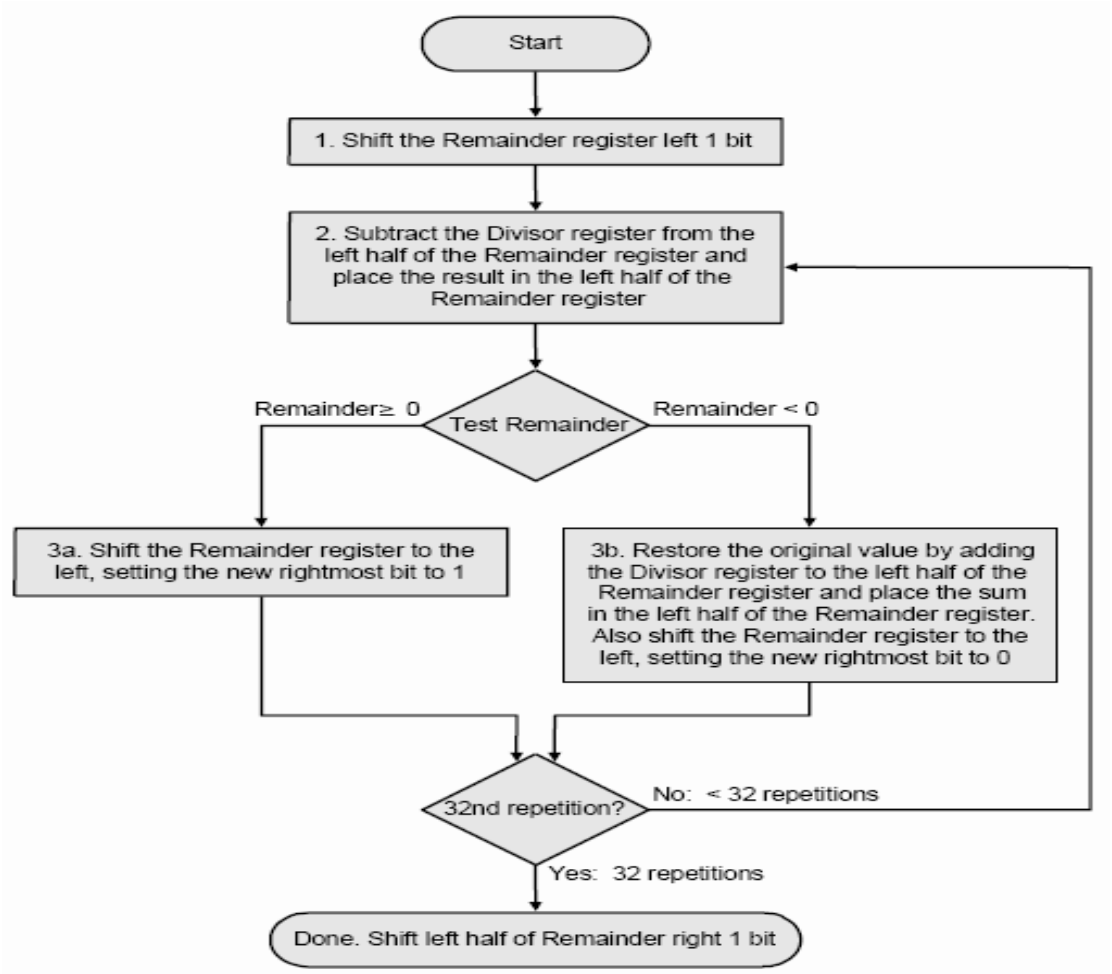
- This version only needs a 32-bit divisor register and a 32-bit ALU.
- 32-bit ALU
- Three registers:
    - Divisor register: 32 bits
    - Quotient register: 32 bits
    - Remainder register: 64 bits
- Operations:
    - The 32-bit divisor is always subtracted from the left half of the remainder register. The result is written back to the left half of the remainder register.
    - The first step of this algorithm cannot produce a 1 in the quotient bit; if it did, the quotient would be too large for the register. By switching the order of the operations to shift and then subtract, one iteration of the algorithm can be removed.

Final Version of the division Hardware

- Reducing registers

Step1: i = 0; R = 2 x A; D = B
Do {
Step2:
$R_H = R_H - D$
If ($R_H < 0$) $R_H = R_H + D$; R = 2 x R
Else R = 2 x R +1
i++
} while (i < n) # $R_H = R_H / 2$ = remainder.
R_L = quotient





- This version combines the right half of the remainder register with the divisor register.
- 32-bit ALU
- Two registers:
    - Divisor register: 32 bits
    - Remainder register: 64 bits
- (Right half also used for storing quotient)
- Operations:
    - The two left-shifts at each step for version 2 are combined into only a single left-shift because the remainder and quotient registers have been combined.
    - The consequence of combining the two registers and the new order of the operations in the loop (as in version 2) is that the remainder register will be shifted left one time too many. Thus the final correction step (right-shift) must shift back only the remainder in the left half of the remainder register.

- **Example**: Division of a 4-bit unsigned number (0111) by another one (0010)

Iteration	**Divisor**	**Restoring-Divide Algorithm**	
		Step	**Product**
0	0010	Initial values	0000 0111
		Shift remainder left by 1	0000 1110
1		2. Remainder = Remainder – Divisor	**1110** 1110
		3b. (Remainder < 0); + Div; Shift left; $R_0 = 0$	0001 110**0**
2		2. Remainder = Remainder – Divisor	**1111** 1100
		3b. (Remainder < 0); + Div; Shift left; $R_0 = 0$	0011 100**0**
3		2. Remainder = Remainder – Divisor	**0001** 1000
		3a. (Remainder > 0); Shift left; $R_0 = 1$	0011 000**1**
4		2. Remainder = Remainder – Divisor	**0001** 0001
		3a. (Remainder > 0); Shift left; $R_0 = 1$	0010 001**1**
Done		Shift left half of remainder right by 1	**0001** 0011

Signed Division

- We must set the sign for both quotient and the reminder
- Dividend = Quotient x Divisor + Reminder

- Example all combinations of ±7 ÷ ±2
  - +7 ÷ +2: Quotient +3, Reminder +1: 7= 3x2+(+1)
  - -7 ÷ +2: Quotient -3, Reminder = Dividend - Quotient x Divisor = -7 – (-3x2) =-1

- Rule: The Dividend and the Reminder must have the same signs
  - +7 ÷ -2: Quotient -3, Reminder +1
  - -7 ÷ -2: Quotient +3, Reminder -1

Division in MIPS

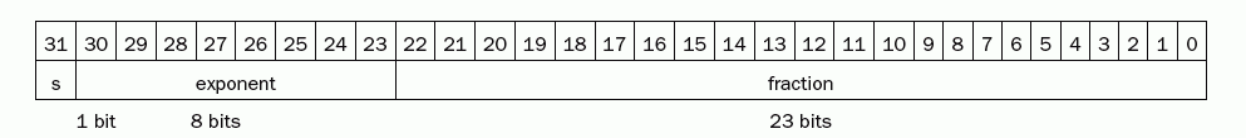
- MIPS has two instructions for both signed and unsigned instructions:
  - **div** ('divide')
  - **divu** ('divide unsigned')
  - Examples:
    - div $s1, $s2 # Lo = $s1 / $s2; Hi = $1 mod $s2
    - divu $s1, $s2 # Lo = $s1 / $s2; Hi = $1 mod $s2
- MIPS divide instructions ignore overflow
- MIPS software must check the divisor it is zero as well as overflow.

Floating Point

- Programming languages support numbers with fractions, which are called reals in mathematics.
- Examples: Numbers with fractions: 3.12159265_{ten}
  Very small numbers: 0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$
  Very large numbers: $3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$
- Notice that some numbers bigger than we could represent with a 32-bit signed integer. The alternative notation is called scientific notation.
- Scientific notation has a single digit to the left of the decimal point.
- A number in scientific notation that has no leading 0s is called a normalized number.
  - $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.
- We can also show binary numbers in scientific notation: $1.0_{two} \times 2^{-1}$
- The advantages of a scientific notation for reals in normalized form are:
  - It simplifies exchange of data that includes floating point numbers.
  - It simplifies the floating point arithmetic algorithms.
  - It increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

Floating Point Representation

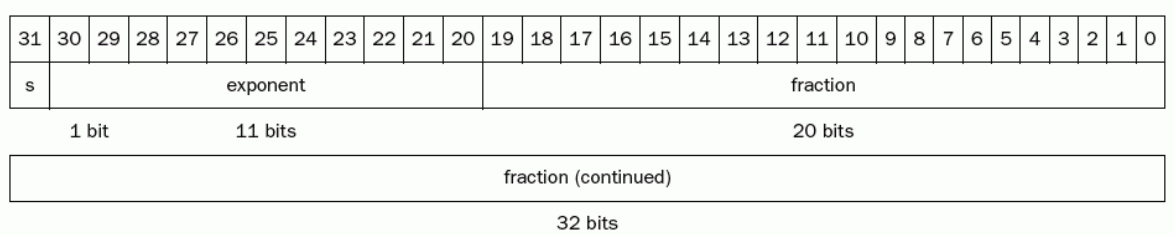
- A designer of a floating point representation must find a compromise between the size of the fraction and the size of the exponent.
  - Increasing the size of the fraction enhances the precision of the fraction.
  - Increasing the size of the exponent increases the range of numbers that can be represented.
  - ***Good design demands good compromise***
- The representation of a MIPS floating point number is shown below, where s is the sign of the floating point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), the fraction (usually called mantissa or significand) is the 23-bit number.



- This representation called sign and magnitude for **single precision** floating point
- Floating point numbers are generally of the form: $(-1)^S \times F \times 2^E$
- In this representation, the value range for F is $1 <= F <= 2\text{-}2^{-23}$ or $1 <= F < 2$
  - Largest positive/negative number = $\pm (2 - 2^{-23}) \times 2^{127} \approx \pm 2 \times 10^{38}$
  - Smallest positive/negative number = $\pm 1 \times 2^{-126} \approx \pm 2 \times 10^{-38}$

Double Precision Floating Point Numbers

- Overflow interrupts can occur in floating point arithmetic. Overflow means the exponent is too large to be represented in the exponent field.
- Underflow occurs when the negative exponent is too large to fit in the exponent field.
- Another format that has a lager exponent is used to reduce chances of underflow or overflow, this format called double precision.
- Double precision takes two MIPS words, where s is still the sign of the number, exponent is the value of the 11-bit exponent field, and fraction is the 52-bit number in the fraction.



- The value range for F is $1 <= F <= 2\text{-}2^{-52}$ or $1 <= F < 2$.
  - Largest positive/negative number = $\pm (2 - 2^{-52}) \times 2^{1023} \approx \pm 2 \times 10^{308}$
  - Smallest positive/negative number = $\pm 1 \times 2^{-1022} \approx \pm 2 \times 10^{-308}$

Normalized Floating Point Numbers

- These formats are part of the IEEE 754 floating point standard
- To pack even more bits into the significand, this standard makes the **leading 1 bit of normalizes binary numbers implicit**.
- Interpretation: $(-1)^S \times (1 + \text{Fraction}) \times 2^E$
- Effective number of bits used for representing the significand:
  - **24** (i.e., 23 + 1) – for single precision
  - **53** (i.e., 52 + 1) – for double precision
- The bits of the fraction represent a number between 0 and 1. If we number the bits of the fraction from left to right s1,s2,…. , then the value is

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + \ldots\ldots) \times 2^E$$

- Sorting through integer comparisons:
  - The sign is in the most significant bit, allowing a test of less than, greater than, or equal to 0 to be formed quickly.
  - The exponent is placed before the significand, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.
- Negative exponents pose a challenge to simplified sorting. A negative exponent will look like a big number. For example, $1.0_{two} \times 2^{-1}$ would be represented as:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

- (Remember that the leading 1 is implicit in the significand.) the value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Biased Exponent:

- A bias is used to represent exponent instead of using the 2's complement.
- A bias (127 for single precision and 1023 for double precision) is added to the exponent so that the most negative exponent is represented as $00\ldots00_{two}$ and the most positive as $11\ldots11_{two}$
- Interpretation: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})}$

- In single precision, the range of biased exponent is between 0 and 255, leaving the two extreme 0 and 255 for special meanings. The real range is 1 to 254, 1 represent the most negative exponent (-126) and 254 represent the most positive exponent (127).
- In double precision numbers, the range is -1022 <= E <= 1023 (all 0's and all 1's have special meanings)

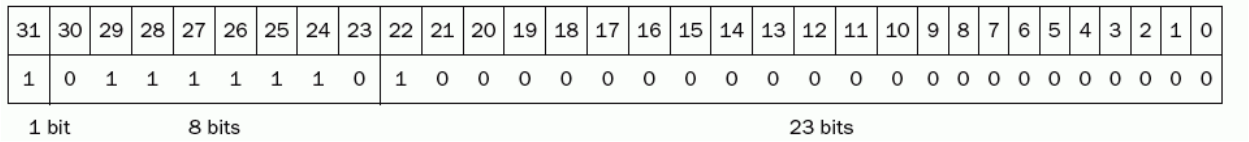
- Representing Zero:
  - All bits of F are zero; the hardware does not attach a leading 1 to it.
  - E has all bits zero.
  - Sign bit is also zero.
  - Exactly same as integer zero.

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1–254	anything	1–2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign.

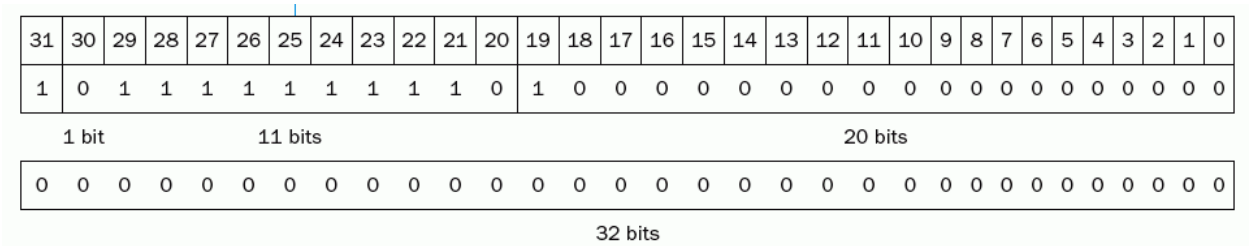
Example: show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

- In decimal the value is: -0.75 = - (½ + ¼)
- In binary: - 0.11
- In scientific notation, the value is: $-0.11_{two} \times 2^{0}$
- In normalized scientific notation, it is: $-1.1_{two} \times 2^{-1}$
- The general representation for a single precision number is
  $(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$
  The result is:
  $(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$
- The single precision binary representation of -0.75_{ten} is

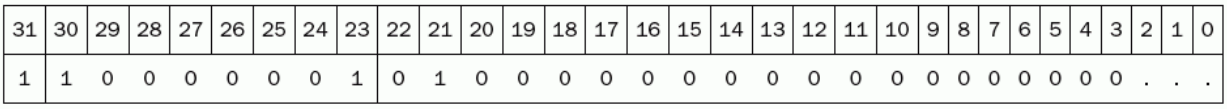


- The double precision representation is

$(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$



- Example: (Converting Binary to Decimal Floating Point)
  What decimal number is represented by this single precision float?



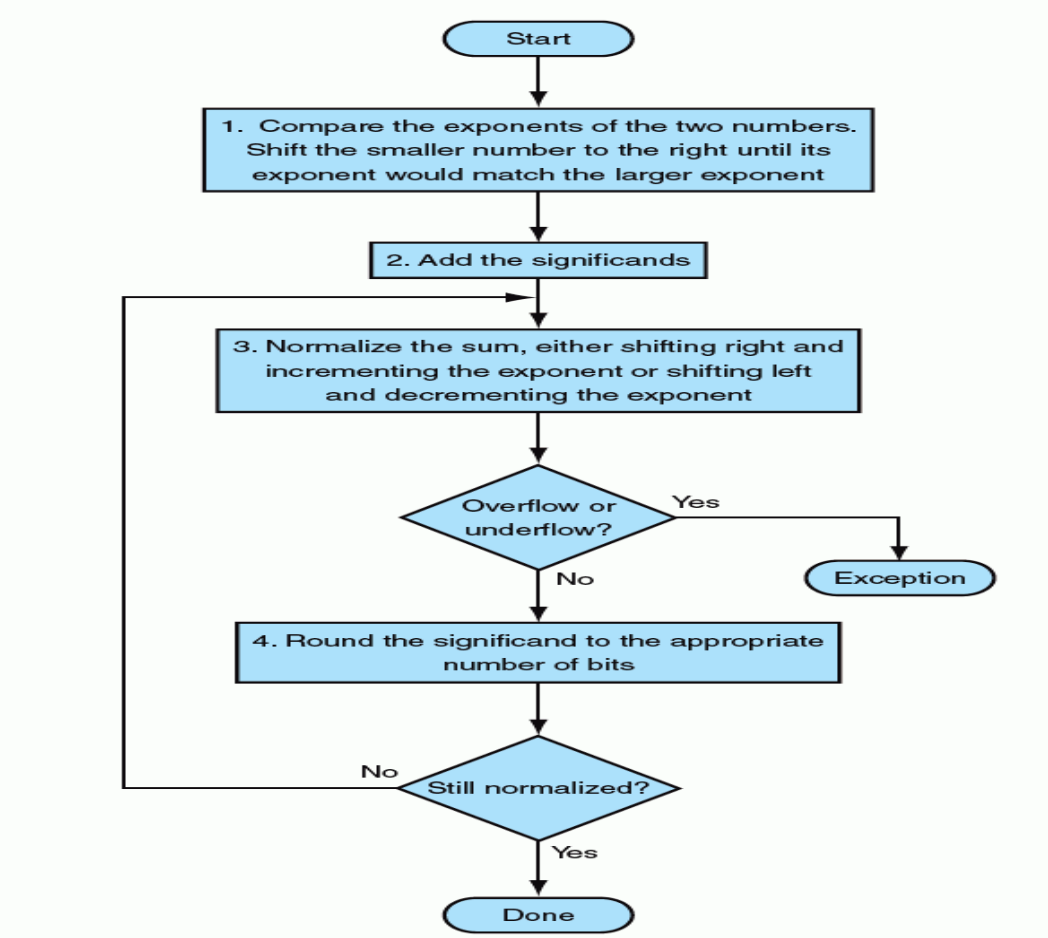
- The sign bit is 1, the exponent field contains 129, and the fraction field contains 1 x 2^{-2} = ¼, or 0.25

- Using the basic equation, $(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$
  $= (-1)^{1} \times (1 + 0.25) \times 2^{(129-127)}$
  $= -1 \times 1.25 \times 2^{2}$
  $= -1.25 \times 4 = -5.0$

Floating Point Addition

- An illustrative example: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$
  - Assumptions:
    - Significand size = 4 decimal digits
    - Exponent size = 2 decimal digits
  - ***Step 1:*** Align the decimal point of the number that has the smaller exponent.
    ($1.610_{ten} \times 10^{-1}$ becomes $0.016_{ten} \times 10^{1}$)
  - ***Step 2:*** Add the significands of the two numbers together.
    ($9.999_{ten} \times 10^{1} + 0.016_{ten} \times 10^{1} = 10.015_{ten} \times 10^{1}$)
  - ***Step 3:*** Normalize the sum.
    ($10.015_{ten} \times 10^{1}$ becomes $1.0015_{ten} \times 10^{2}$)
  - ***Step 4:*** Round the normalized sum
    ($1.0015_{ten} \times 10^{2}$ becomes $1.002_{ten} \times 10^{2}$)

Algorithm for Floating Point Addition

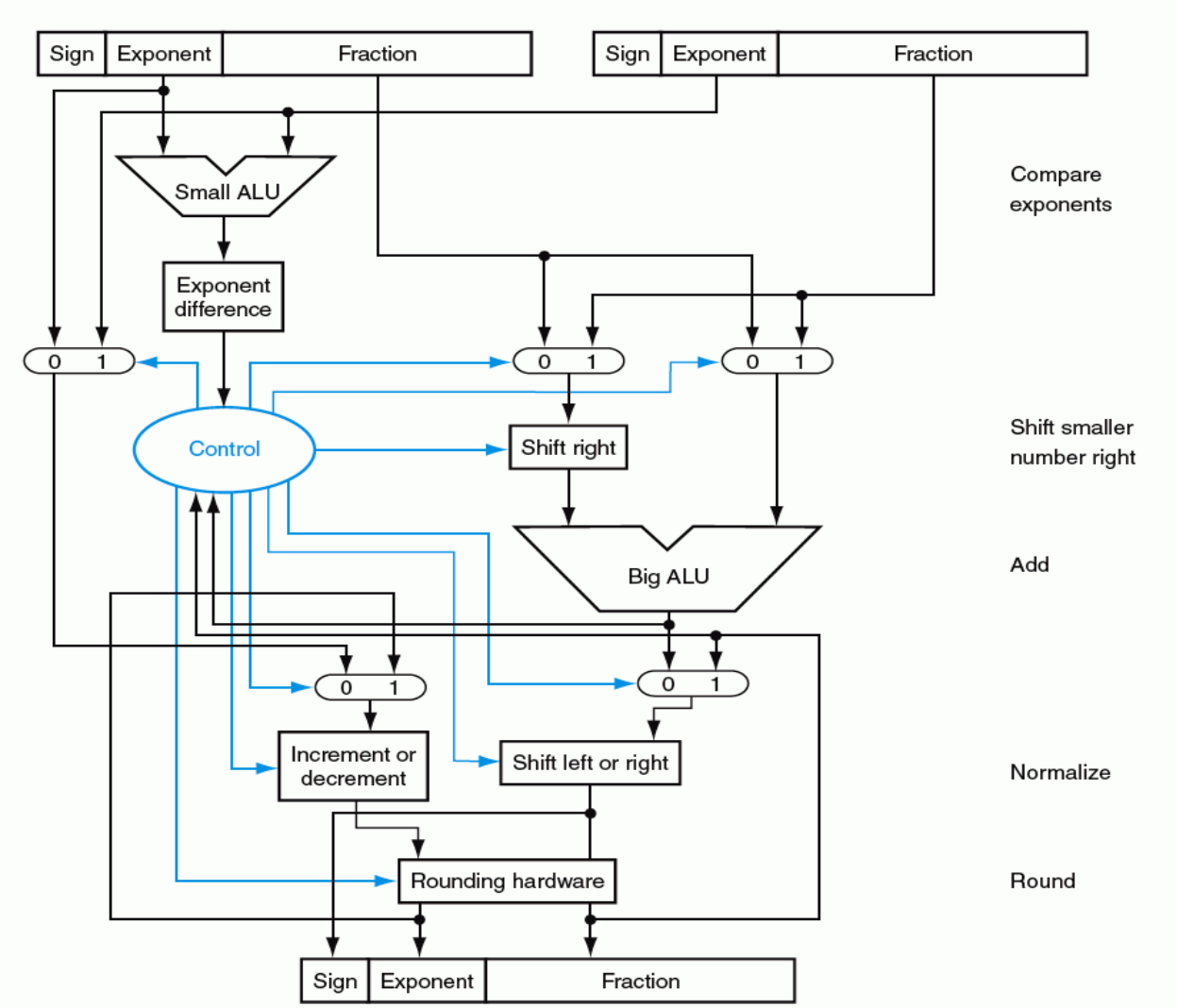


- Example: add the numbers 0.5_{ten} and -0.4375_{ten} in binary. Assuming that we keep 4 bits of precision:
  - $0.5_{ten} = 0.1_{two} = 1.000_{two} \times 2^{-1}$
  - $-0.4375_{ten} = -0.0111_{two} = -1.110_{two} \times 2^{-2}$
  - Now we follow the algorithm:
    - ***Step 1:*** Shift the smaller number ($-1.110_{two} \times 2^{-2}$) to the right until its exponent would match the larger exponent
      $-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$
    - ***Step 2:*** Add the significands:
      $1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$
    - ***Step 3:*** Normalize the sum, checking for overflow or underflow:
      $0.001_{two} \times 2^{-1} = 1.000_{two} \times 2^{-4}$
      Since 127 >= -4 >= -126, there is no overflow or underflow.
    - ***Step 4:*** Round the sum: $1.000_{two} \times 2^{-4}$
      the sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

      This sum is then $1.000_{two} \times 2^{-4} = 0.0625_{ten}$

Arithmetic Unit for Floating Point Addition

- Many computers dedicate hardware to run floating point operations as fast as possible



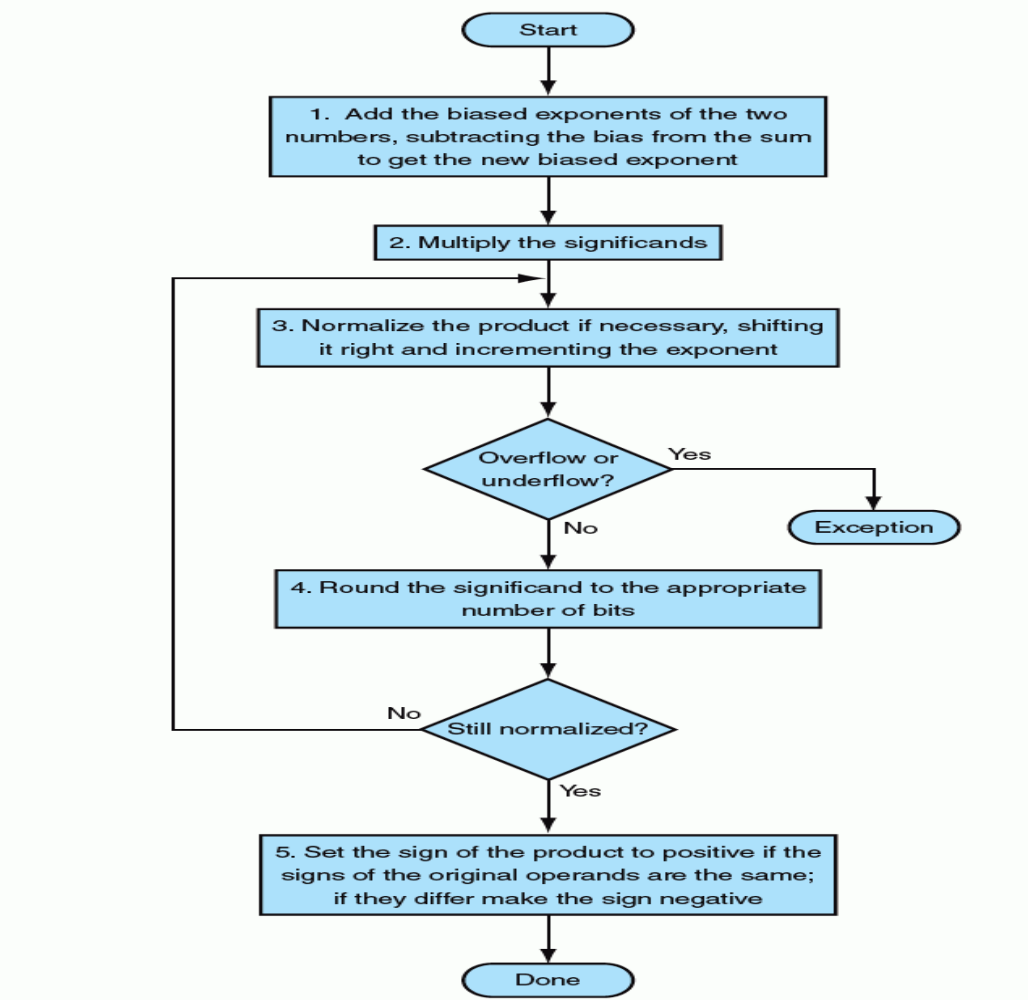
Floating Point Multiplication

- $[(-1)^{S1} \times F1 \times 2^{E1}] \times [(-1)^{S2} \times F2 \times 2^{E2}] = (-1)^{S1 \oplus S2} \times (F1 \times F2) \times 2^{E1+E2}$
  - $1 <= (F1 \times F2) < 4$ or $(1 \times 1) <= (F1 \times F2) < (2-2^{-23} \times 2-2^{-23})$
  - the result may need to be normalized

- An illustrative example: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$
  - Assumptions:
    - Significand size = 4 decimal digits
    - Exponent size = 2 decimal digits

- ***Step 1:*** Add the exponents together. (new exponent = 10 + (-5) = 5)
- ***Step 2:*** Multiply the significands together.
  (new significand = 1.110_{ten} x 9.200_{ten} = 10.212_{ten} if we can only keep 3 digits after the decimal point)
- ***Step 3:*** Normalize the product. (10.212_{ten} x 10^5 becomes 1.0212_{ten} x 10^6)
- ***Step 4:*** Round the product. (1.0212_{ten} x 10^6 becomes 1.021_{ten} x 10^6)
- ***Step 5:*** Find the sign of the product. ($+1.021_{ten}$ x 10^6)

Algorithm for Floating Point Multiplication



- Example: multiply the numbers 0.5_{ten} and -0.4375_{ten} in binary. Assuming that we keep 4 bits of precision:
  - $0.5_{ten} = 1.000_{two}$ x 2^{-1}
  - $-0.4375_{ten} = -1.110_{two}$ x 2^{-2}
  - Now we follow the algorithm:

- ***Step 1:*** Adding the exponents without bias: -1 + (-2) = -3 or, using the biased representation: (-1 + 127) + (-2 + 127) -127 = 124 or -3 + 127 = 124
- ***Step 2:*** Multiplying the significands:

$$\begin{array}{rl} & 1.000_{two} \\ \times & 1.110_{two} \\ \hline & 0000 \\ & 1000 \\ & 1000 \\ & 1000 \\ \hline & 1110000_{two} \end{array}$$

  The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$
- ***Step 3:*** Normalize the sum, checking for overflow or underflow:
  the product is already normalized and, since 127 >= -3 >= -126, there is no overflow or underflow. (using the biased representation, 254>= 124 >= 1, so the exponent fits)
- ***Step 4:*** Rounding the product makes no change: $1.110_{two} \times 2^{-3}$
- ***Step 5:*** Find the sign of the product:
  the sign is negative - $1.110_{two} \times 2^{-3}$
  Converting to decimal to check our results: $-1.110_{two} \times 2^{-3} = -0.21875_{ten}$

Floating Point Division

- $[(-1)^{S1} \times F1 \times 2^{E1}] \div [(-1)^{S2} \times F2 \times 2^{E2}] = (-1)^{S1 \oplus S2} \times (F1 \div F2) \times 2^{E1-E2}$
  Since $0.5 < (F1 \div F2) < 2$, the result may need to be normalized. (assume $F2 \neq 0$)

FP Instructions in MIPS

- MIPS supports the IEEE 754 single-precision and double-precision formats.
- 32 floating point registers $f0 .. $f31 (whose registers are used in pairs for double precision values)
- Single precision arithmetic
  - add.s, sub.s, mul.s, div.s
  - example: add.s $f2,$f4,$f6 #$f2 = $f4 + $f6
- double precision arithmetic (similar)
  - add.d, sub.d, mul.d, div.d
  - example: add.d $f2, $f4, $f6 #$f2||$f3 = $f4||$f5 + $f6||$f7