IMPERIAL COLLEGE LONDON

ISA AND COMPILERS: YEAR 2

ALU

Xin Wang

October 17, 2020

Abstract

Unlike other instruction architectures like the Mu0, the ALU is used in almost every instruction in the MIPS architecture from common arithmetic to address calculation.

MIPS architecture supports negative and positive arithmetic as well as floating numbers i.e. decimals so it is important to understand how the ALU enables those features. Also the following aspects need to be studied:

- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- How does hardware really multiply or divide numbers?

Contents

1 Signed and unsigned numbers **2**

2 Addition and Subtraction **3**

3 Multiplication **5**
- 3.1 Sequential version of the multiplication algorithm and Hardware 6
- 3.2 Signed multiplication 8
- 3.3 Multiply in MIPS 8

4 Division **9**
- 4.1 Division algorithm and Hardware 9
- 4.2 Signed division 11
- 4.3 Divide in MIPS 12

5 Constructing a basic ALU **13**
- 5.1 1-bit ALU 13
- 5.2 32-bit ALU 15
- 5.3 Tailoring the 32-Bit ALU to MIPS 16

6 MIPS in Verilog **18**

1 Signed and unsigned numbers

Numbers in computer hardware are represented as a series of high and low electronic signals i.e. binary numbers. A single digit of a binary number is thus the "atom" of computing, since all information is composed of binary digits or bits.

Binary digit: One of the two numbers in base 2 i.e. 0 or 1 that represents the components of information.

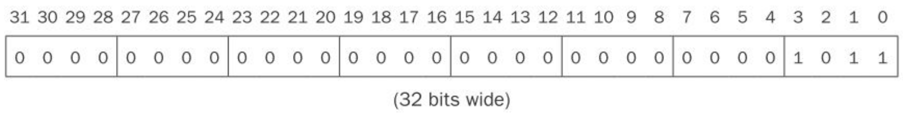


Figure 1: Standard word length in MIPS architecture

The MIPS word is 32 bits long, so can represent 2^{32} different 32-bit patterns i.e. 0 to $2^{32} - 1$

Least significant bit: The **rightmost** bit in a MIPS word.

Most significant bit: The **leftmost** bit in a MIPS word.

Hardware is designed to add, subtract, multiply, and divide these binary bit patterns. If the proper result of such operations cannot be represented by these hardware bits, **overflow** has occurred. It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so a system to distinguishes the positive from the negative is needed:

1. **Sign and magnitude**: Simplest system but adders for sign and magnitude will need an extra step to set the sign because it is not known in advance what the proper sign will be. Also, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems.

2. **Two's complement**: Leading 0s mean positive, and leading 1s mean negative. It has the advantage that all negative numbers have a 1 in the most significant bit i.e. **the sign bit**. Hardware needs to test only the sign bit to see if a number is positive or negative. Overflow can occur when the sign bit is incorrect e.g. a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Two's complements are used in modern designs and there are two convenient shortcuts when dealing with them:

1. A quick way to negate a two's complement binary number.

   Invert every 0 to 1 and every 1 to 0, then add 1 to the result. The reason is based on the observation that the sum of a number x and its inverted $\overline{x}$ must be -1. Since $x + \overline{x} = -1$, the following statements are true:

   - $x + \overline{x} + 1 = 0$
   - $\overline{x} + 1 = -x$

2. How to convert a binary number represented in n bits e.g. 16 bits to a number represented with **more than n bits** e.g. 32 bits.

   Sign extension: Take the sign bit from the smaller quantity and copy it to fill the new bits of the larger quantity. This works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware and sign extension simply restores some of them.

2 Addition and Subtraction

The simplest arithmetic operations. Digits are added bit by bit from right to left with carries passed to the next digit to the left. Subtraction also uses the concept addition: the **operand is negated before being added**.

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

Figure 2: Addition: $7 + 6$

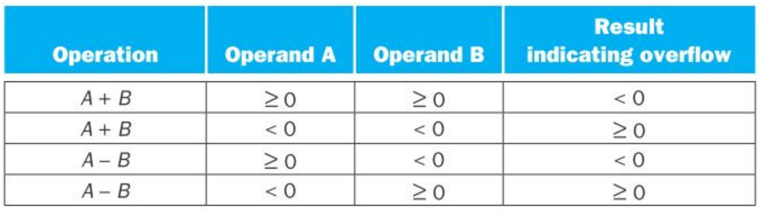
$$\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
\end{array}$$

Figure 3: Subtraction: $7 - 6$ where 6 is in two's complement

A concern in any arithmetic operation is **overflow**. Recall that overflow occurs when the result from an operation cannot be represented with the available hardware i.e. a 32 bit word.

Overflow can occur in **adding or subtracting two 32-bit numbers with the same signs** which yields a result that needs 33 bits to be fully expressed. Overflow cannot occur when **adding operands with different signs** because the sum is never larger than one of the operands.

Overflows are easily detected by looking at the **sign bit**. Overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This means that a carry out occurred into 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

Figure 4: Overflow conditions for addition and subtraction

When dealing with **unsigned integers**, overflows are ignored as they are commonly used for memory addresses and lack the sign bit. A computer architecture must therefore provide a way to determine if overflow is important or not.

The MIPS solution is to 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.

Interrupt: An unscheduled event that disrupts program execution, commonly used to detect overflow.

MIPS detects overflow with interrupts:

- The address of the instruction that overflowed is saved in a register
- The computer jumps to a predefined address to invoke the appropriate routine for that exception.
- The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

3 Multiplication

First, a recap of the multiplication of decimal numbers by hand to establish the steps of multiplication and the names of the operands.

The following working example 1000_{ten} and 1001_{ten}:

```
Multiplicand          1000ten
Multiplier     x      1001ten
                     --------
                      1000
                     0000
                    0000
                   1000
                   -------
Product            1001000ten
```

There are two observations in the multiplication process:

- The number of digits in the product is larger than the number in either the multiplicand or the multiplier.

  The length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is $n + m$-bits long. That is, $n + m$-bits are required to represent all possible products. Like add, multiplication must cope with overflow.

- Due to the binary design, there are only two possible actions at each step of multiplication:

  1. Place a copy of the multiplicand i.e. $1 \times \textit{multiplicand}$ in the proper place if **multiplier digit** is a 1.

  2. Place 0s i.e. $0 \times \textit{multiplicand}$ in the proper place if **multiplier digit** is a 0.

3.1 Sequential version of the multiplication algorithm and Hardware

Building, from the observations above, a simple design that supports only multiplying positive numbers:

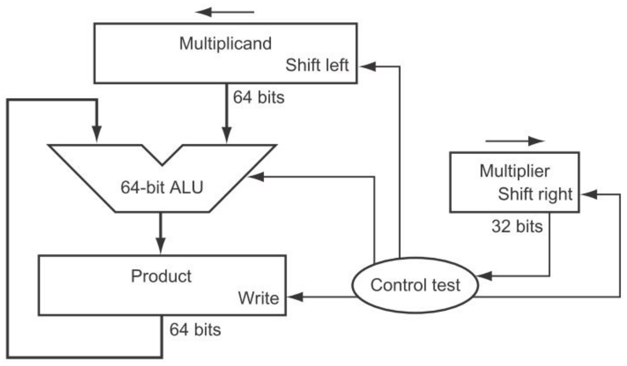


Figure 5: First version of the multiplication hardware

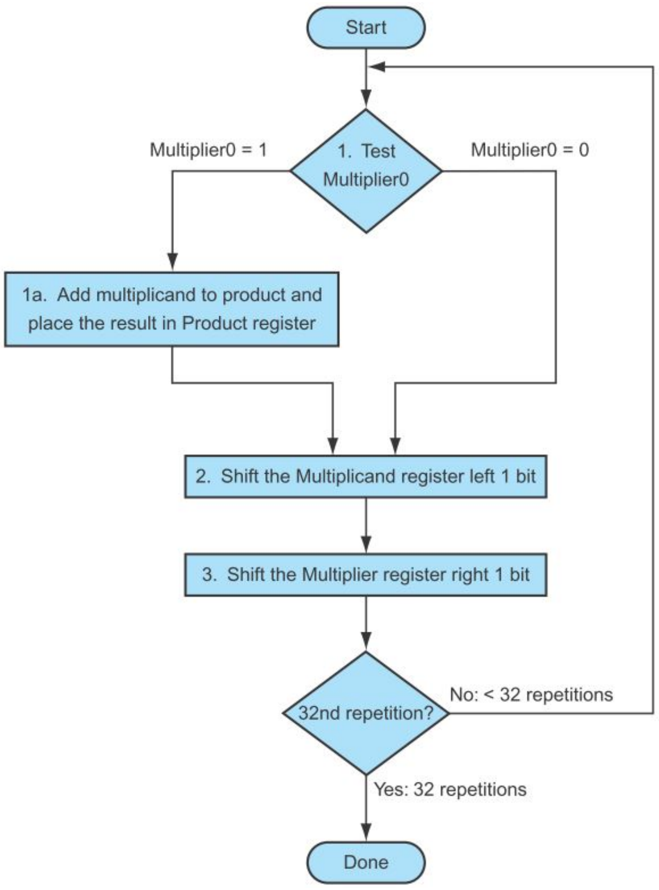

Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Figure 6: The first multiplication algorithm

Multiplication:

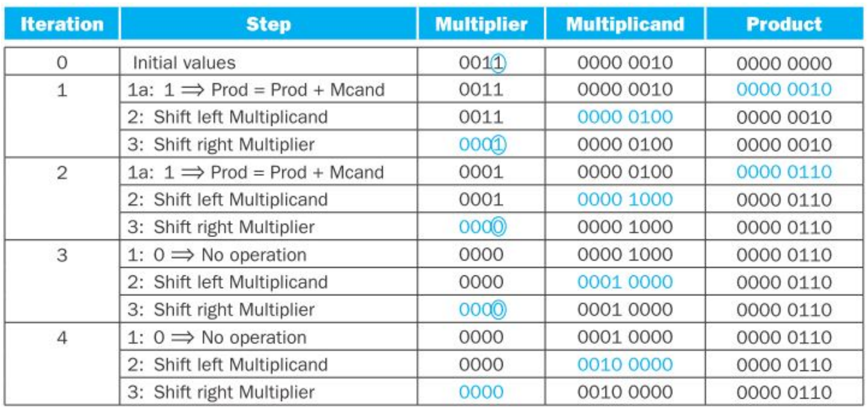
- The multiplier is in the 32-bit *multiplicand register* and that the 64-bit *product register* is initialized to 0.

- From the example above, multiplying means moving the multiplicand left one digit each step since it may be added to the intermediate products. So that means over 32 steps, a 32-bit multiplicand would move 32 bits to the left.

- Hence, a 64-bit *multiplicand register* is needed, initialized with the 32-bit multiplicand **in the right half and zero in the left half**.

- The *multiplicand register* is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit *product register.*

- Control decides when to shift the *Multiplicand register* and *Multiplier register* and when to write new values into the Product register.



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

Three steps of the flowchart:

- The least significant bit of the multiplier i.e. **Multiplier0** determines whether the multiplicand is added to the *product register.*

- The left shift in step 2 has the effect of moving the intermediate operands to the left.

- The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration.

This simple algorithm would take more than one clock cycle. However, it can be easily refined to take 1 clock cycle per step by performing the operations in parallel i.e. the *multiplier* and *multiplicand* are shifted while the *multiplicand* is added to the product if the multiplier bit is a 1.

The hardware has to ensure that it tests the right bit of the multiplier and gets the pre-shifted version of the multiplicand.

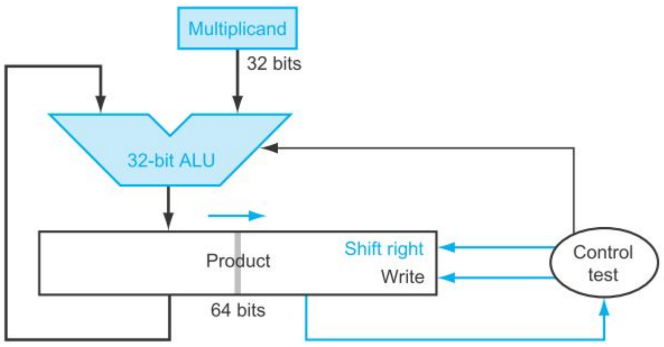


Figure 7: Optimised version of the multiplication hardware

3.2 Signed multiplication

Dealing with signs in multiplication is a simple addition on the existing system. The easiest way to understand how to deal with signed numbers is:

- Convert the *multiplier* and *multiplicand* to positive numbers.
- Remember the original signs

The algorithms would then be run for 31 iterations as normally design, leaving the signs out of the calculation. The shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

3.3 Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product called Hi and Lo. To produce proper signed or unsigned product, MIPS has two instructions:

- Multiply `mult`
- Multiply unsigned `multu`

To fetch the integer 32-bit product, the programmer uses instruction "move from lo" `mflo`. The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

4 Division

The reciprocal of multiply that is less frequent and more complicated especially considering the opportunity to perform a mathematically invalid operation: dividing by 0.

Like multiplication, the classic pen-and-paper is first revised to establish some observations.

The example is dividing 1001010 by 1000:

```
                        1001ten    Quotient
Divisor 1000ten |1001010ten    Dividend
               −1000
                   10
                   101
                   1010
                  −1000
                     10ten    Remainder
```

There are:

- **Two operands: Divisor and Dividend**
- **Two outputs: Quotient and Remainder**

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

4.1 Division algorithm and Hardware

Start with the 32-bit *Quotient register* set to 0. Each iteration of the algorithm needs to **move the divisor to the right one digit**, so the **text** divisor placed in the left-half of the 64-bit *Divisor register* and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

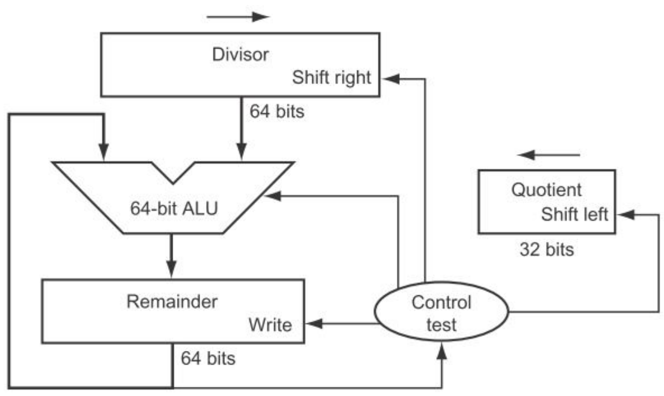


Figure 8: First version of the division hardware.

There are three steps of the division algorithm:

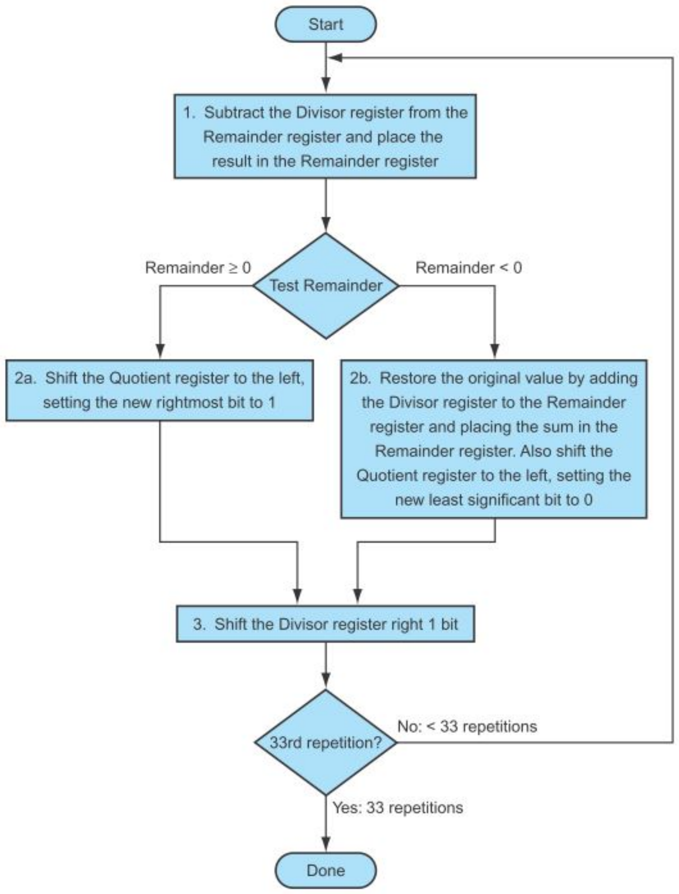
1. The computer needs to know in advance if the divisor is smaller than the dividend. It subtracts the divisor since this is how comparisons are performed in the set on less than instruction.

    - If result is positive, divisor is smaller or equal to the dividend. A 1 is outputted in the quotient.

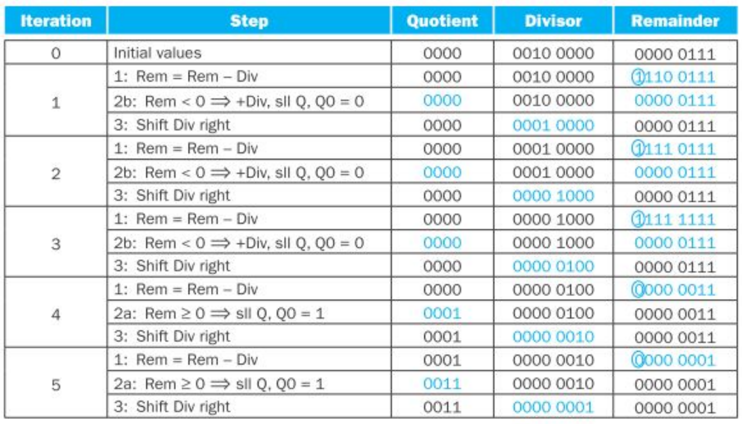
    - If result is negative, restore the original value by adding the divisor back to the remainder and output a 0 in the quotient.

2. The divisor is shifted right and then repeated again.

3. The remainder and quotient will be found in their respective registers after the iterations are complete.


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

This algorithm and hardware can be refined to be faster and cheaper. The optimisation comes from shifting the operands and the quotient **simultaneously with the subtraction**. This halves the width of the adder and registers by noticing where there are unused portions of registers and adders.



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
	1: Rem = Rem – Div	0000	0010 0000	1110 0111
1	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
	1: Rem = Rem – Div	0000	0001 0000	1111 0111
2	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
	1: Rem = Rem – Div	0000	0000 1000	1111 1111
3	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
	1: Rem = Rem – Div	0000	0000 0100	0000 0011
4	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
	1: Rem = Rem – Div	0001	0000 0010	0000 0001
5	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

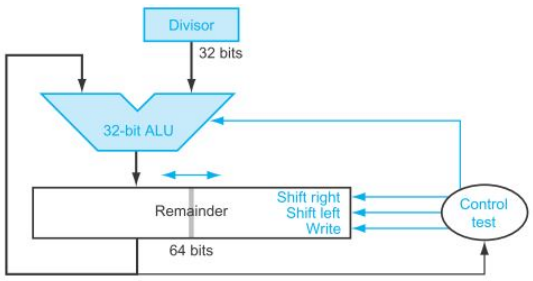


Figure 9: An improved version of the division hardware

4.2 Signed division

The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree. Remember that the following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

For example divide all the combinations of $(\pm)7$ by $(\pm)2$:

- $(+)7 \div (+)2$: Quotient $= (+)3$ and Remainder $= (+)1$
- $(-)7 \div (+)2$: Quotient $= (-)3$ and Remainder $= (-)1$
- $(+)7 \div (-)2$: Quotient $= (-)3$ and Remainder $= (+)1$
- $(-)7 \div (-)2$: Quotient $= (+)3$ and Remainder $= (-)1$

The rule observed is that the dividend and remainder **must have the same signs**. The correctly signed division algorithm:

- Negates the quotient if the signs of the operands are opposite
- Makes the sign of the nonzero remainder match the dividend

4.3 Divide in MIPS

The same sequential hardware can be used for both multiply and divide. The only requirement is:

- 64-bit register that can shift left or right
- 32-bit ALU that adds or subtracts

Hence, MIPS uses the 32-bit `Hi` and 32-bit `Lo` registers for both multiply and divide. `Hi` contains the remainder and `Lo` contains the quotient after the divide instruction completes.

To handle both signed integers and unsigned integers, MIPS has two instructions: divide `div` and divide unsigned `divu`.

The MIPS assembler allows divide instructions to specify three registers, generating the `mflo` or `mfhi` instructions to place the desired result into a general-purpose register.

5 Constructing a basic ALU

The *ALU* is the brain of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. Because the MIPS word is 32 bits wide, a 32-bit-wide *ALU* is needed.

5.1 1-bit ALU

The logical operations are easiest, because they map directly onto the hardware components.

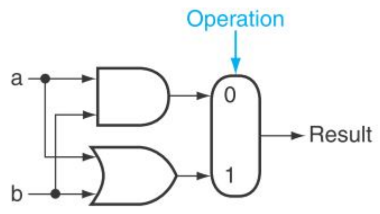


Figure 10: 1-bit logical unit for AND and OR

The multiplexor on the right then selects a AND b or a OR b, depending on whether the value of Operation is 0 or 1.

The next function to include is addition:

- An adder must have **two inputs** for the operands and a **single-bit output** for the sum.
- There must be a second output to pass on the carry `CarryOut`.
- Since the `CarryOut` from the neighbor adder must be included as an input, a third input `CarryIn` is defined.

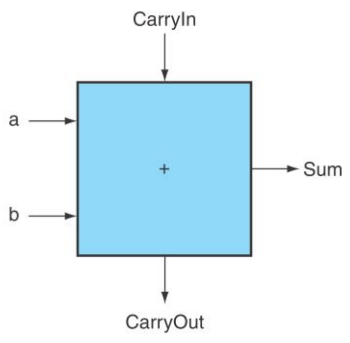


Figure 11: A 1-bit adder

The output functions `CarryOut` and `Sum` can be defined as logical equations and implemented with logic gates.

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Figure 12: Input and output specification for a 1-bit adder.

1-bit ALU derived by combining the adder with the earlier components. If designers want the ALU to perform a few more simple operations such as generating 0, the easiest way to add an operation is to expand the multiplexor controlled by the Operation line.

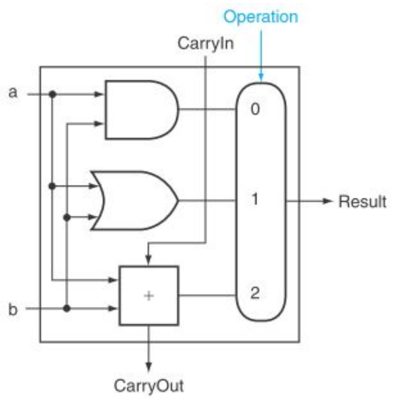


Figure 13: A 1-bit ALU that performs AND, OR, and addition

5.2 32-bit ALU

The full 32-bit ALU is created by connecting adjacent 1-bit ALUs. The adder created by directly linking the carries of 1-bit adders is a **ripple carry adder**.

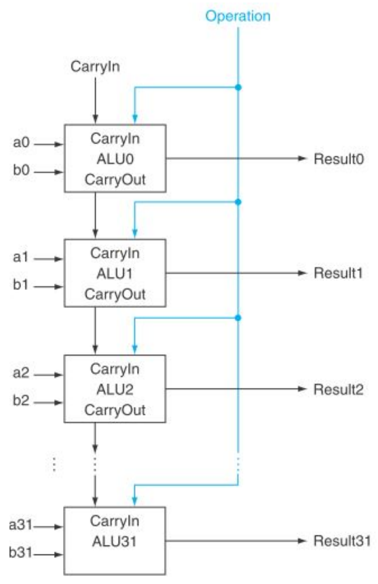


Figure 14: A 32-bit ALU constructed from 32 1-bit ALUs

Remember, subtraction is the same as adding the negative version of an operand and this is how adders would perform subtraction. The shortcut for negating a two's complement number is to invert each bit and then add 1.

To invert each bit, add a [2 : 1] multiplexor that chooses between b and $\overline{b}$

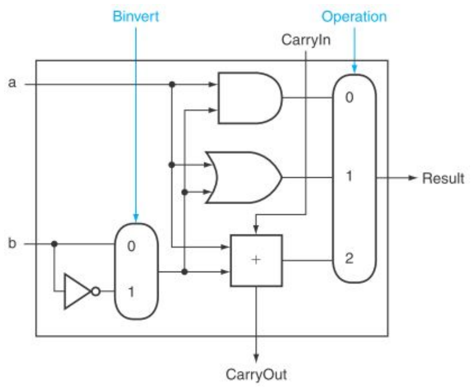


Figure 15: A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\overline{b}$

The simplicity of the hardware design of a two's complement adder helps explain why two's complement representation has become the universal standard for integer computer arithmetic.

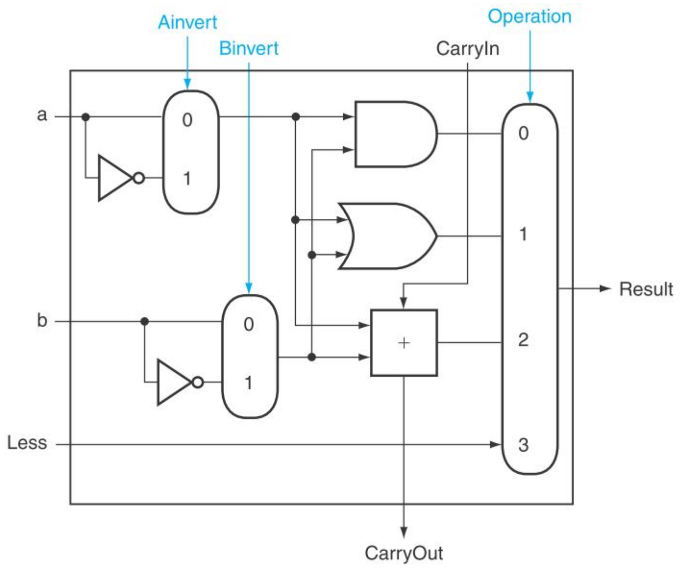
5.3 Tailoring the 32-Bit ALU to MIPS

The previous section implemented *add*, *sub*, AND and OR. There is one last instruction that is performed by the ALU in the MIPS architecture: "set on less than" `slt`.

The `slt` instruction sets the destination register's content to the value 1 if the first source register's contents are less than the second source register's contents i.e. $rs < rt$. Otherwise, it is set to the value 0.

slt \$destination, \$first source address, \$second source address

To add support for this instruction, the ALU needs to expand the three-input multi-plexor to add an input for the `slt` result.



From the description of `slt`, the upper 31 bits of the ALU are always 0. What remains to consider is how to compare and set the least significant bit for "set on less than" instructions.

Note that subtraction can be used, if the difference is negative then $a < b$:

$$\begin{aligned}(a-b) < 0 &\Rightarrow ((a-b)+b) < (0+b)\\ &\Rightarrow a < b\end{aligned}$$

A 1 if $a < b$ is negative and a 0 if it's positive. This result corresponds exactly to the sign bit values. Following this line of argument, just connect the sign bit from the adder output.

Some modifications to the existing ALU is required to support the `slt` instruction.

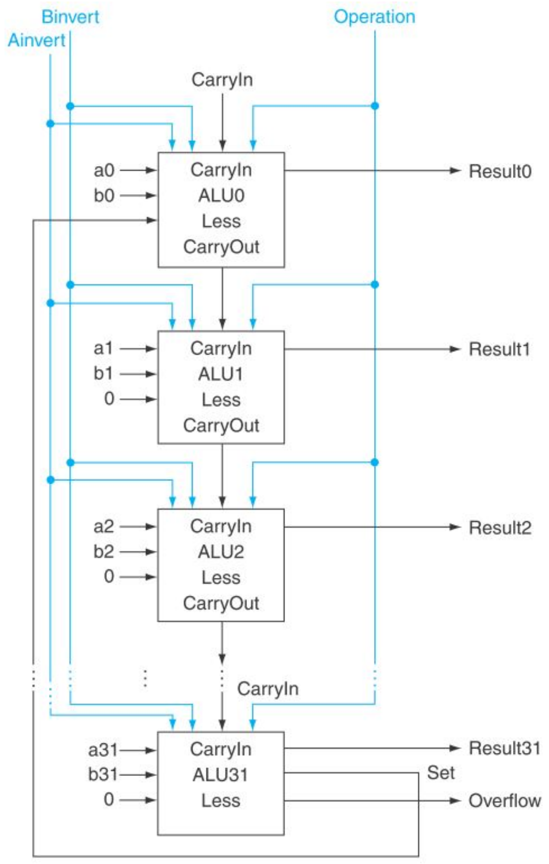


Figure 16: A 32-bit ALU constructed from the 31 copies of the 1-bit ALU as seen previously and one 1-bit ALU in the bottom.

The `Less` inputs are connected to 0 except for the least significant bit, which is connected to the `Set` output of the most significant bit:

- If the ALU performs $a - b$, the input 3 in the multiplexor shows `Result`= $0 \dots 001$
- If the ALU performs $a < b$, the input 3 in the multiplexor shows `Result`= $0 \dots 000$

6 MIPS in Verilog

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

Figure 17: Verilog behavioral definition of a MIPS ALU.

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2; // add
   34: ALUOp<=6; //subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7; // slt
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

Figure 18: MIPS ALU control: Combinational control logic.