ALU design of Simple microprocessor

Microprocessor Design:

Microprocessors typically have a single module that deals with arithmetic operations on integer

numbers. This being due to several of the various arithmetic and logical operations can be carried out

with the aid of similar (if not same) hardware. The component that deals with the arithmetic and logical

operations is known as the Arithmetic Logic Unit, or ALU.

The ALU is by far, one of the most vital components in a microprocessor, and is generally the part of

the processor that is implemented before the others. Once the ALU implementation is finalized, the remaining parts of the microprocessor is

designed to provide the ALU with operands and control codes.

Tasks of an ALU:

ALU components generally require to be able to carry out the fundamental logical operations (I.e. AND, OR) and the addition operation. Inverters are included on the inputs to enable the same ALU hardware to perform the subtraction operation (adding an inverted operand),

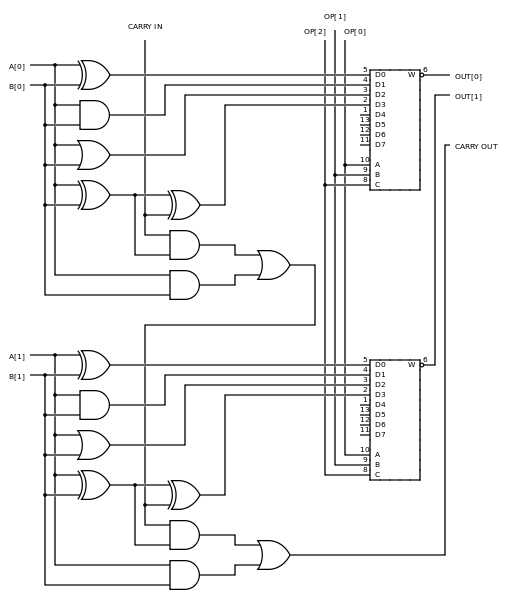
and the operations NAND and NOR.

A simple ALU design includes a culmination of "ALU Slices", which each can carry out the required operation on a single bit. There is a single ALU slice for every single bit in the operand.

ALU Slice:

Example: 2-bit ALU

This is an example of a simple 2-bit ALU. The boxes on the right hand side of the image are multiplexers and are used to select between several operations: OR, AND, XOR, and addition.

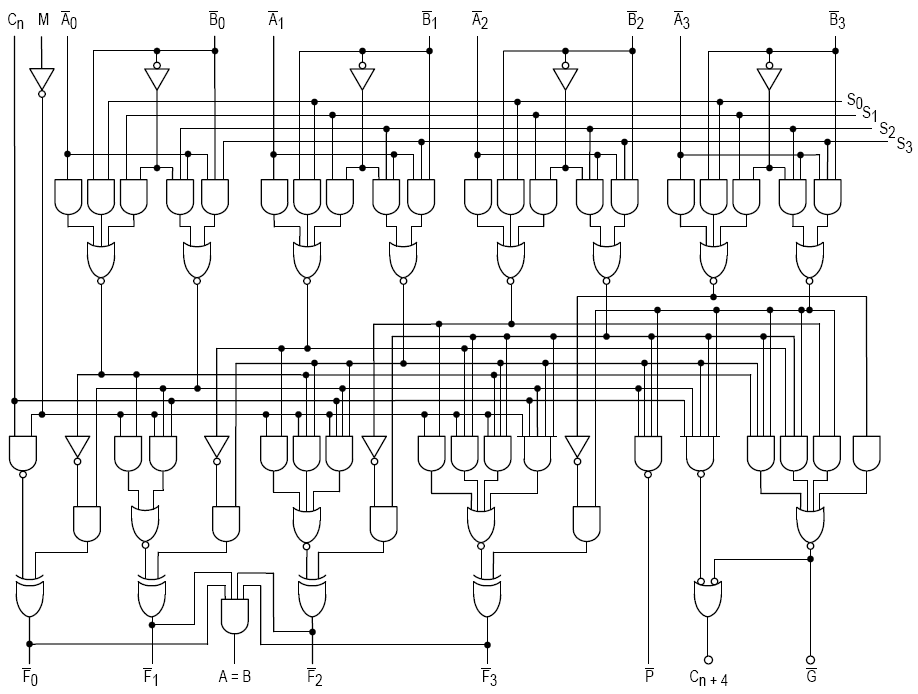


All the operations are carried out in parallel, and the select signal ("OP") is used to finalize which result to forward to the rest of the data path. The carry signal is created and exits through the ALU after

every operation. It deals with only addition. It is crucial that if we are not carrying out addition that we ignore the carry flag.

Example: 4-Bit ALU

Below is a circuit diagram of a 4 bit ALU.



Additional Operations

Logic and summation are few of the simplest, but also the most commonly used operations. For this reason, generally ALUs are made to solve these operations mostly, and other operations, such as multiplication and division, are performed in a different module.

The ALU units that we have shown here are only for integer data types, not floating-point data. Luckily , once integer ALU and multiplier units have been implemented, those units can also create floating-point units (FPU).

ALU Configurations

After the ALU is implement, we are required to define how it communicates with the other parts of the processor. We can select any single configuration from numerous different configurations, each with its own pros and cons. Every category of instruction set architecture (ISA) -- stack,

accumulator, register-memory , or register-register-load-store -- needs an alternative way to connect with the ALU.

In the images below, the orange signifies the memory structures inside the CPU (registers), and the purple signifies external memory (RAM).

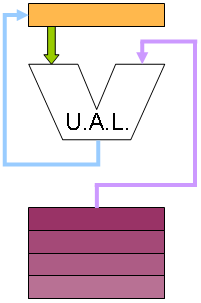
Accumulator

An accumulator machine has a special register, named the accumulator. The accumulator saves the result of each and every ALU operation, and is also one of the operands to every piece of instruction. This results in our ISA being less complex, as

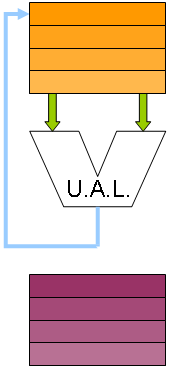
instructions require only to select one operand, rather than two and a destination. Accumulator architectures have basic ISAs and are usually very fast, but extra software require to be written to load the accumulator with accurate values.

Unfortunately, accumulator machines are cumbersome to pipeline.

One example of a type of computer system that is likely to use an accumulator is a common desk calculator.



**Register-to-Register**



One of the more commonly seen architectures is a Register-to-register architecture, also coined a 3 register operand machine. In this design, the programmer can control both the source operands, and a destination register. Inopportunely, the ISA requires to be expanded to include fields for both source operands and the destination operands. This needs longer instruction word lengths, and it also

Needs extra effort (compared to the accumulator) to write the results back to the register file

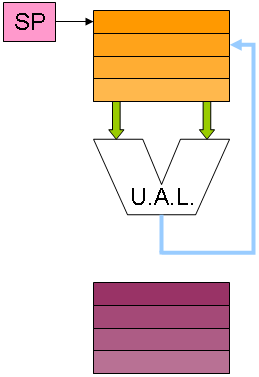
after execution has been carried out. This write-back step can result in problems with synchronization in pipelined processors.

Register Stack

A register stack is analogous to a combination of the Register-to-Register and the accumulator structures. In this design, the ALU reads the operands from the very top of the stack, and the result is pushed onto the top of the stack. Complex mathematical operations need decomposition into Reverse-Polish form, which can be cumbersome for programmers to use. On that note, several computer language compilers can generate reverse-polish notation with ease due to the use of binary trees to symbolize internal instructions. Moreover, hardware requires to be made to design the register stack, which will also include PUSH and POP operations, in addition to hardware to distinguish and handle errors in the stack (i.e. pushing on a full stack, or popping an empty stack).

The advantage results from a highly simplistic ISA. These machines are called "0-operand" or "zero address machines" as operands don't require to be specified, for all operations act on specific locations on the stack.

In the diagram, "SP" is the pointer to the top of the stack. This is one way to implement a stack structure, although it might be one of the simplest..



Register-and-Memory

One complex structure is a Register-and-Memory structure, see figure for reference. In this

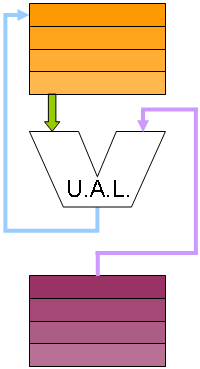
structure, one operand is passed from a register file, and the other one is passed from external memory . In this structure, the ISA is complex as each instruction word requires to be able to

store an entire memory address, which may be quite long. Practically, this scheme is not implemented

as it is, but is usually added onto a different scheme, such as a Register-to-Register scheme, for added flexibility.

Few CISC architectures have the option of specifying one of the operands to an instruction as

a memory address, even though they are typically specified as a register address.



Complicated Structures

There are numerous other structures available, few of which are novel, and others are a mixture of the types we discussed.

It is up to the computer architecture designer to resolve specifically how the microprocessor should be structured, and feed data into the ALU.

Example: IA-32

The Intel IA-32 ISA (x86 processors) use a register stack architecture for the floating point unit, but it uses an altered Register-to-Register structure for integer operations. Every integer operations can specify a register as the first operand, and a register or memory location as the second operand. The first operand acts as an accumulator, so that the final answer is stored in the first operand register. The con to this is that the instruction words are not consistent in length, which results in the instruction fetch and

decode modules of the processor requiring to be quite complicated.

A general IA-32 instruction is written as:

ADD AX, BX

Where AX and BX are the names of the registers. The resulting equation produces AX = AX + BX, so the result is stored back into AX.

Example: MIPS

MIPS uses a Register-to-Register structure. Every operation can specify two register operands, and a third destination register.

The disadvantage is that memory reads need to be made in individual separate operations, and the small format of the instruction words means that space is at a premium, and some tasks are difficult to perform.

An example of a MIPS instruction is:

ADD R1, R2, R3

Where R1, R2 and R3 are the names of registers. The resulting equation looks like: R1 = R2 + R3.

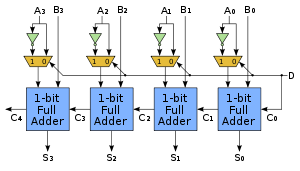
#####Editsz

## **Addition and Subtraction[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=1" \o "Edit section: Addition and Subtraction)]**

Addition and subtraction are similar algorithms. Taking a look at subtraction, we can see that:

{\displaystyle a-b=a+(-b)}

Using this simple relationship, we can see that addition and subtraction can be performed using the same hardware. Using this setup, however, care must be taken to invert the value of the second operand if we are performing subtraction. Note also that in twos-compliment arithmetic, the value of the second operand must not only be inverted, but 1 must be added to it. For this reason, when performing subtraction, the carry input into the LSB should be a 1 and not a zero.

[](https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder-subtracter.svg)

Our goal on this page, then, is to find suitable hardware for performing addition.

## **Bit Adders[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=2" \o "Edit section: Bit Adders)]**

### Half Adder**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=3" \o "Edit section: Half Adder)]**

A half adder is a circuit that performs binary addition on two bits. A half adder does not explicitly account for a carry input signal.

[](https://commons.wikimedia.org/wiki/File:Half-adder.svg)

In verilog, a half-adder can be implemented as follows:

module half\_adder(a, b, c, s)

input a, b;

output s, c;

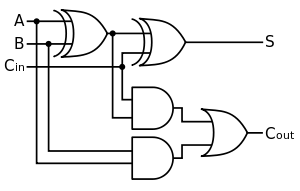
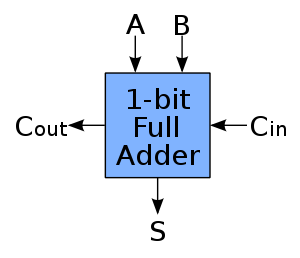
s = a ^ b;

c = a & b;

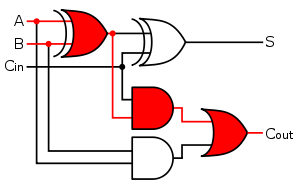
endmodule

### Full Adder**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=4" \o "Edit section: Full Adder)]**

Full adder circuits are similar to the half-adder, except that they do account for a carry input and a carry output. Full adders can be treated as a 3-bit adder with a 2-bit result, or they can be treated as a single stage (a 3:2 compressor) in a larger adder.

[](https://commons.wikimedia.org/wiki/File:Full-adder.svg)[](https://commons.wikimedia.org/wiki/File:1-bit_full-adder.svg)

As can be seen below, the number of gate delays in a full-adder circuit is 3:

[](https://commons.wikimedia.org/wiki/File:Full-adder_with_gate_delay.svg)

We can use verilog to implement a full adder module:

module full\_adder(a, b, cin, cout, s);

input a, b, cin;

output cout, s;

wire temp;

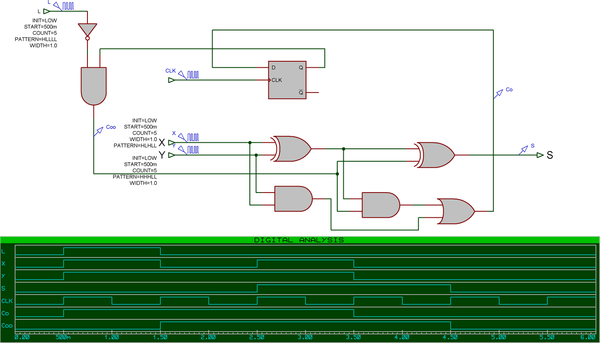
temp = a ^ b;

s = temp ^ cin;

cout = (cin & temp) | (a & b);

endmodule

## **Serial Adder[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=5" \o "Edit section: Serial Adder)]**

[](https://commons.wikimedia.org/wiki/File:Serialadder.png)

A serial adder is a kind of [ALU](https://en.wikibooks.org/wiki/Microprocessor_Design/ALU) that calculates each bit of the output, one at a time, re-using one full adder (total). This image shows a 2-bit serial adder, and the associated waveforms.

Serial adders have the benefit that they require the least amount of hardware of all adders, but they suffer by being the slowest.

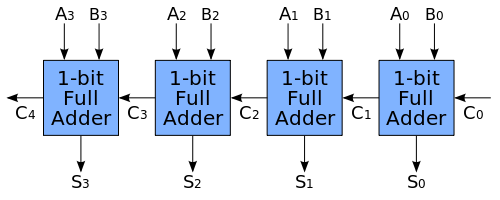
## **Parallel Adder[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=6" \o "Edit section: Parallel Adder)]**

A parallel adder is a kind of [ALU](https://en.wikibooks.org/wiki/Microprocessor_Design/ALU) that calculates every bit of the output more or less simultaneously, using one full adder for each output bit. The 1947 Whirlwind computer was the first computer to use a parallel adder.

In many CPUs, the CPU latches the final carry-out of the parallel adder in an external "carry flag" in a "status register".

In a few CPUs, the latched value of the carry flag is always wired to the first carry-in of the parallel adder; this gives "Add with carry" with 2s' complement addition. (In a very few CPUs, an end-around carry -- the final carry-out of the parallel adder is directly connected to the first carry-in of the same parallel adder -- gives 1's complement addition).

### Ripple Carry Adder**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=7" \o "Edit section: Ripple Carry Adder)]**

[](https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg)

Numbers of more than 1 bit long require more then just a single full adder to manipulate using arithmetic and bitwise logic instructions[*[citation needed](https://en.wikibooks.org/wiki/Wikibooks:OR" \o "Wikibooks:OR)*]. A simple way of operating on larger numbers is to cascade a number of full-adder blocks together into a **ripple-carry adder**, seen above. Ripple Carry adders are so called because the carry value "ripples" from one block to the next, down the entire chain of full adders. The output values of the higher-order bits are not correct, and the arithmetic is not complete, until the carry signal has completely propagated down the chain of full adders.

If each full adder requires 3 gate delays for computation, then an *n*-bit ripple carry adder will require *3n* gate delays. For 32 or 64 bit computers (or higher) this delay can be overwhelmingly large.

Ripple carry adders have the benefit that they require the least amount of hardware of all adders (except for serial adders), but they suffer by being the slowest (except for serial adders).

With the full-adder verilog module we defined above, we can define a 4-bit ripple-carry adder in Verilog. The adder can be expanded logically:

wire [4:0] c;

wire [3:0] s;

full\_adder fa1(a[0], b[0], c[0], c[1], s[0]);

full\_adder fa2(a[1], b[1], c[1], c[2], s[1]);

full\_adder fa3(a[2], b[2], c[2], c[3], s[2]);

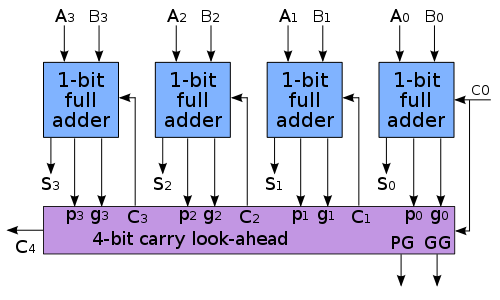
full\_adder fa4(a[3], b[3], c[3], c[4], s[3]);

At the end of this module, *s* contains the 4 bit sum, and c[3] contains the final carry out.

This "ripple carry" arrangement makes "add" and "subtract" take much longer than the other operations of an ALU (AND, NAND, shift-left, divide-by-two, etc). A few CPUs use a ripple carry ALU, and require the programmer to insert NOPs to give the "add" time to settle.[[1]](https://en.wikibooks.org/wiki/Microprocessor_Design/Add_and_Subtract_Blocks#cite_note-1) A few other CPUs use a ripple carry adder, and simply set the clock rate slow enough that there is plenty of time for the carry bits to ripple through the adder. A few CPUs use a ripple carry adder, and make the "add" instruction take more clocks than the "XOR" instruction, in order to give the carry bits more time to ripple through the adder on an "add", but without unnecessarily slowing down the CPU during a "XOR". However, it makes pipelining much simpler if every instruction takes the same number of clocks to execute.

### Carry Skip Adder**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=8" \o "Edit section: Carry Skip Adder)]**

### Carry Lookahead Adder**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=9" \o "Edit section: Carry Lookahead Adder)]**

[](https://commons.wikimedia.org/wiki/File:4-bit_carry_lookahead_adder.svg)

|  |  |
| --- | --- |
| Wikipedia-logo.png | [Wikipedia](https://en.wikipedia.org/wiki/) has related information at [***Carry look-ahead adder***](https://en.wikipedia.org/wiki/Carry_look-ahead_adder) |

Carry-lookahead adders use special "look ahead" blocks to compute the carry from a group of 4 full-adders, and passes this carry signal to the next group of 4 full adders. Lookahead units can also be cascaded, to minimize the number of gate delays to completely propagate the carry signal to the end of the chain. Carry lookahead adders are some of the fastest adder circuits available, but they suffer from requiring large amounts of hardware to implement. The number of transistors needed to implement a carry-lookahead adder is proportional to the number of inputs cubed.

The addition of two 1-digit inputs A and B is said to *generate* if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition 52 + 67, the addition of the tens digits 5 and 6 *generates* because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit clearly does not carry).

In the case of binary addition, {\displaystyle A+B} generates if and only if both A and B are 1. If we write {\displaystyle G(A,B)} to represent the binary predicate that is true if and only if {\displaystyle A+B} generates, we have:

{\displaystyle G(A,B)=A\cdot B}

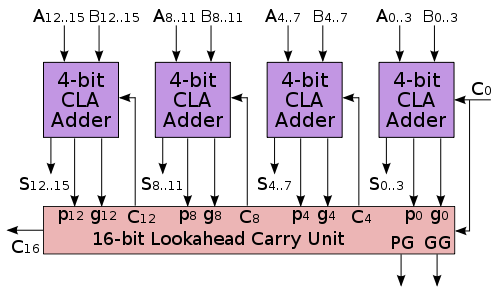
The addition of two 1-digit inputs A and B is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition 37 + 62, the addition of the tens digits 3 and 6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

In the case of binary addition, {\displaystyle A+B} propagates if and only if at least one of A or B is 1. If we write {\displaystyle P(A,B)} to represent the binary predicate that is true if and only if {\displaystyle A+B} propagates, we have:

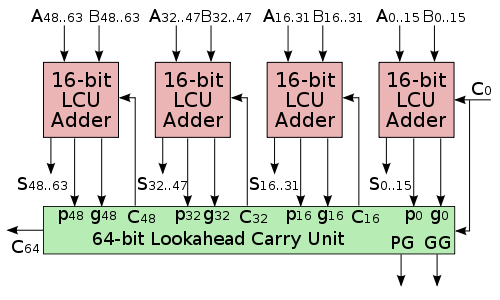
{\displaystyle P(A,B)=A+B}

### Cascading Adders**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=10" \o "Edit section: Cascading Adders)]**

The power of carry-lookahead adders is that the bit-length of the adder can be expanded without increasing the propagation delay too much. By cascading lookahead modules, and passing "propagate" and "generate" signals to the next level of the lookahead module. For instance, once we have 4 adders combined into a simple lookahead module, we can use that to create a 16-bit and a 64-bit adder through cascading:

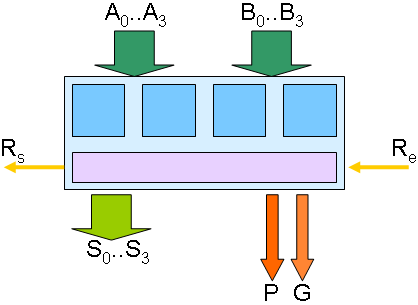
[](https://commons.wikimedia.org/wiki/File:16-bit_lookahead_carry_unit.svg)

The 16-Bit carry lookahead unit is exactly the same as the 4-bit carry lookahead adder.

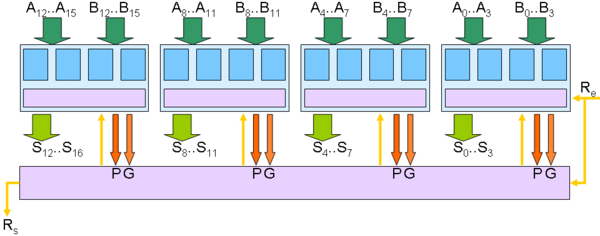
[](https://commons.wikimedia.org/wiki/File:64-bit_lookahead_carry_unit.svg)

the 64-bit carry lookahead unit is exactly the same as the 4-bit and 16-bit units. This means that once we have designed one carry lookahead module, we can cascade it to any large size.

### Generalized Cascading**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Add_and_Subtract_Blocks&action=edit&section=11" \o "Edit section: Generalized Cascading)]**

[](https://commons.wikimedia.org/wiki/File:Cla4bitsPG.png)

A generalized CLA block diagram. Each of the turquoise blocks represents a smaller CLA adder.

[](https://commons.wikimedia.org/wiki/File:Cla16bitsPG.png)

We can cascade the generalized CLA block above to form a larger CLA block. This larger block can then be cascaded into a larger CLA block using the same method.

#### Edits flag close

ALU Flags

For various reasons, it may be vital to export a number of status codes from the ALU, for detecting errors, and for making decisions.

## **Comparisons**

Comparisons among two values are generally carried out by finding their difference. We can determine the relationship between the two values by examining the difference:

* If the first is larger than the second, the result will be positive
* If the second is larger than the first, the result will be negative
* If the two are equal, the result will be zero.

#####EDITS flag open

## **Zero Flag**

Determining whether two values are equal requires the ALU to determine whether the result is zero. This can be accomplished by feeding each bit of the result into a NOR gate. The beauty of this is that a single multi-port NOR gate requires less hardware than an entire array of equivalent 2-port gates.

## **Overflow Flag**

It is good to know when the result of an addition or multiplication is larger than the maximum result size. Likewise, it is also good to know if the result of a subtraction or a division is smaller than possible, and thus creates underflow. Either two separate flags can be used for these conditions, or one flag can be interpreted in different ways, depending on the input operation.

## **Carry/Borrow flag[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/ALU_Flags&action=edit&section=4" \o "Edit section: Carry/Borrow flag)]**

This flag indicates when an operation results in a value larger than the accumulator can represent (carry/overflow) or smaller than the accumulator can represent (borrow/underflow). It can be used by software to implement arbitrary-width arithmetic, such as a "bignum" library.

## **Comparisons[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/ALU_Flags&action=edit&section=5" \o "Edit section: Comparisons)]**

Many ALUs need to compare data items, and determine if a particular value is greater than or less than another value. In these cases, the ALU will also export flags for these values.

A comparison in a processor can typically be performed by a subtraction operation. If the result is a positive number, the first item is greater than the second item. If the result is a negative number, the first item is less than the second. If the numbers being compared are unsigned, the value of the carry flag will serve the same purpose as the greater-than or less-than flag.

## **Latch ALU flags or not?[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/ALU_Flags&action=edit&section=6" \o "Edit section: Latch ALU flags or not?)]**

Some instruction sets refer to the ALU flags from some previous instruction:

CMP R1,R2 // compare

...

BEQ equal\_routine // branch if equal

Such instruction sets force the CPU designer to latch those ALU flags in some sort of "status register", and to be very careful to make sure it is possible to preserve those flags during an interrupt routine.

Other instruction sets never refer to previous ALU flags -- they always use the results from the ALU in the same instruction that they are calculated:

BEQ R1,R2,equal\_routine // compare and branch if equal

or

SKEQ R1,R2 // compare and skip next instruction if equal

JMP equal\_routine

Some CPU designers prefer such instruction sets that never refer to previous ALU flags. Such instruction sets make out-of-order execution much simpler. Many of Chuck Moore's CPU designs never refer to the ALU flags from any previous instruction.

## **Shift and Rotate[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Shift_and_Rotate_Blocks&action=edit&section=1" \o "Edit section: Shift and Rotate)]**

Shift and rotate blocks are essential elements in most processors. They are useful on their own, but they also are used in multiplication and division modules. In a binary computer, a left shift has the same effect as a multiplication by 2, and a right shift has the same effect as a division by 2. Since shift and rotate operations perform much more quickly than multiplication and division, they are useful as a tool in program optimization.

## **Logical Shift[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Shift_and_Rotate_Blocks&action=edit&section=2" \o "Edit section: Logical Shift)]**

|  |  |
| --- | --- |
| [Rotate left logically.svg](https://commons.wikimedia.org/wiki/File:Rotate_left_logically.svg) | [Rotate right logically.svg](https://commons.wikimedia.org/wiki/File:Rotate_right_logically.svg) |
| **A left logical shift** | **A right logical shift** |

In a logical shift, the data is shifted in the appropriate direction, and a zero is shifted into the new location.

## **Arithmetic shift[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Shift_and_Rotate_Blocks&action=edit&section=3" \o "Edit section: Arithmetic shift)]**

|  |
| --- |
| [Rotate right arithmetically.svg](https://commons.wikimedia.org/wiki/File:Rotate_right_arithmetically.svg) |
| **A right arithmetic shift** |

In an arithmetic shift, the data is shifted right so that the sign of the data item is preserved. This means that the MSB is the value that is shifted into the new position. An arithmetic left shift is the same as a logical left shift, and is therefore not shown here.

## **Rotations[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Shift_and_Rotate_Blocks&action=edit&section=4" \o "Edit section: Rotations)]**

|  |  |
| --- | --- |
| [Rotate left.svg](https://commons.wikimedia.org/wiki/File:Rotate_left.svg) | [Rotate right.svg](https://commons.wikimedia.org/wiki/File:Rotate_right.svg) |
| **A left rotation** | **A right rotation** |

A rotation is like a shift, except the bit shifted off the end of the register is then shifted into the new spot.

## **Fast Shift Implementations[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Shift_and_Rotate_Blocks&action=edit&section=5" \o "Edit section: Fast Shift Implementations)]**

The above images in each section help to indicate a method to shift a register more quickly, at the expense of requiring additional hardware. Instead of having one register that attempts to shift in place, we have two registers in parallel, with wires connecting the various blocks together. When a shift is indicated, gates open that allow the data to pass from one register to the next, the proper number of spaces forward or backward.

In practice, fast shift blocks are implemented as a "barrel shifter". The barrel shifter includes several "levels" of multiplexers, each connected to the previous one by straight wires (wires that transfer the data without a shift), and wires that cause a shift by successive powers of two. For instance, the first level of shift would be 4 spaces, the next level would be 2 spaces, and the last level would be 1 space. In this way, the value of each shift level corresponds to the binary representation of the number of spaces to shift. This implementation makes for very fast shifters that can shift an arbitrary number of spaces in a single clock cycle.

 processing time.

In hardware, multiplication and division are performed by a series of sequential additions and arithmetic shifts. for this reason, it is imperative that we have efficient adders and shifters at our disposal.

Multipliers and dividers are composed of shifters and adders. It is typically not possible, or not desirable to use the main adder and shifter units of the ALU, so a microprocessor will typically have multiple ALU units (a primary unit for addition and subtraction, and units embedded in the multiplication and division units). These are other good reasons why our ALU and shifters need to be small and fast.

## **Multiplication Algorithms[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=2" \o "Edit section: Multiplication Algorithms)]**

### Booth's Algorithm**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=3" \o "Edit section: Booth's Algorithm)]**

### Cascaded Multiplication**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=4" \o "Edit section: Cascaded Multiplication)]**

### Wallace tree**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=5" \o "Edit section: Wallace tree)]**

The Wallace tree, a specialized structure for performing multiplication, has been called one of the most important advances in computing.[[1]](https://en.wikibooks.org/wiki/Microprocessor_Design/Multiply_and_Divide_Blocks#cite_note-1)

A Wallace tree using many identical 3:2 compressors (aka full adders), such as the TI 74x275 chip, or the TI 74x183 chip, is one popular way to implement single-cycle multiplication. The datasheets for the TI 74x261 and 74x284 describe some practical details of implementing multiplication with a Wallace tree. The Dadda multiplier uses the same 3:2 compressors in a slightly more efficient arrangement.

## **Division Algorithm[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=6" \o "Edit section: Division Algorithm)]**

## **Multiply and Accumulate[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=7" \o "Edit section: Multiply and Accumulate)]**

**Multiply and accumulate** (MAC) operations perform a multiplication and an addition in a single instruction. For instance, the instruction:

MAC A, B, C

Would perform the operation:

A = A + (B × C)

This is valuable for math-intensive processors, such as graphics processors and DSPs.

An MAC tends to have a long critical path, so if your processor has an MAC operation it is probably possible to include other complicated arithmetic operations.

In a processor with an accumulator architecture, MAC operations will use the accumulator as the destination register, so the instruction:

MAC B, C

Will perform the operation:

ACC = ACC + (B × C)

### Fused Multiply-Add**[[edit](https://en.wikibooks.org/w/index.php?title=Microprocessor_Design/Multiply_and_Divide_Blocks&action=edit&section=8" \o "Edit section: Fused Multiply-Add)]**

A **fused multiply-add** operation is a floating-point operation that is similar to the MAC. However, in the fused operation, the floating-point values are not rounded between the multiply and the add, they are rounded afterwards. For more information about floating-point rounding, see [Floating Point](https://en.wikibooks.org/wiki/Floating_Point).

1. [Jump up↑](https://en.wikibooks.org/wiki/Microprocessor_Design/Multiply_and_Divide_Blocks#cite_ref-1) DTACK Grounded, The Journal of Simple 68000/16081 Systems [Issue # 29 - March 1984](http://www.easy68k.com/paulrsm/dg/dg29.htm) p. 6.

## Design and Implementation for 8-bit ALU

Arithmetic operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Behavior** | **Note** |
| **Addition (ADD)** | A + B | Arithmetic addition. Address generation. |
| **Subtract (SUB)** | A - B | Arithmetic subtraction |

Logial operations

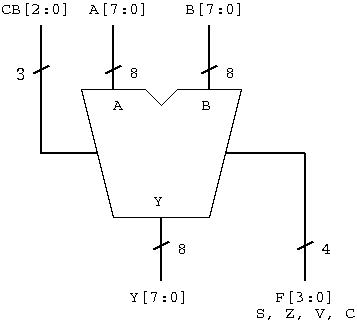
|  |  |  |
| --- | --- | --- |
| **Operation** | **Behavior** | **Note** |
| **Logical and (AND)** | A and B | Bitwise logical AND |
| **Logical or (OR)** | A or B | Bitwise Logical OR |
| **Logical not (NOT)** | not A | Bitwise not, 1's complement |

Shift and exchange operations

|  |  |  |
| --- | --- | --- |
| **Operation** | **Behavior** | **Note** |
| **Arithmetic shift right (ASR)** | http://www.arch.cs.kumamoto-u.ac.jp/~kuga/cad/verilog/alu8/nexys4ise_en/figure/ASR.png | Arithmetic right shift, Divid operand A by 2 |
| **Arithmetic shift left (ASL)** | http://www.arch.cs.kumamoto-u.ac.jp/~kuga/cad/verilog/alu8/nexys4ise_en/figure/ASL.png | Arithmetic left shift, Multiply operand A by 2 |
| **Nibble swap (SWP)** | Exchange upper nibble and lower nibble of operand A | for nibble operation |

## Define input and output variables

The following figure shows the logic symbol of ALU.  
Operand A, operand B and result Y are 8-bit buses.   
Control signal CB selects the operation of ALU.   
The signal has at least 3-bit bus so that the ALU has 8 operations.



There are 4 types of flag, sign S, zero Z, overflow V and carry C. These flags are corded as 4-bit bus F.

Describe this part of Verilog code.

`timescale 1ns/1ps

module alu ( inputs and outputs variables );

input ...

...

output ...

...

endmodule

## Define operation code

8 operations are encoded to 3-bit code.

|  |  |
| --- | --- |
| **Operations** | **Control code** |
| **Addition (ADD)** | **0 0 0** |
| **Subtract (SUB)** | **0 0 1** |
| **Logical OR (OR)** | **0 1 0** |
| **Logical AND (AND)** | **0 1 1** |
| **Logical NOT (NOT)** | **1 0 0** |
| **Arithmetic shift right (ASR)** | **1 0 1** |
| **Arithmetic shift left (ASL)** | **1 1 0** |
| **Nibble swap (SWP)** | **1 1 1** |

Define label of the operation code by "`define".  
By this define, the operation code is described with the defined label as distinct from bit pattern.  
These defines are described in "aluop.v" file.

`define IADD 3'b000

`define ISUB 3'b001

...

...

"alu.v" file includes "alu\_op.v" file to be able to refer.

`timescale 1ns/1ps

`include "alu\_op.v"

module alu ( ... );

...

endmodule

## Describe oprtations

Write each operation.  
The calculation result has 9-bit bus so that we can easily generate the carry flag.   
After that, carry flag and operation result are reassigned to each variable.

#### Edits flag close