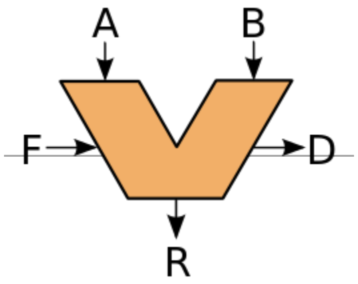
Microprocessor Design/ALU

Microprocessor Design

Microprocessors tend to have a single module that performs arithmetic operations on integer values. This is because many of the different arithmetic and logical operations can be performed using similar (if not identical) hardware. The component that performs the arithmetic and logical operations is known as the **Arithmetic Logic Unit**, or ALU. [1]



The ALU is one of the most important components in a microprocessor, and is typically the part of the processor that is designed first. Once the ALU is designed, the rest of the microprocessor is implemented to feed operands and control codes to the ALU.

Contents

Tasks of an ALU
ALU Slice
Example: 2-Bit ALU
Example: 4-Bit ALU
Additional Operations
ALU Configurations
- Accumulator
- Register-to-Register
- Register Stack
- Register-and-Memory
- Complicated Structures
- Example: IA-32
- Example: MIPS

References

Tasks of an ALU

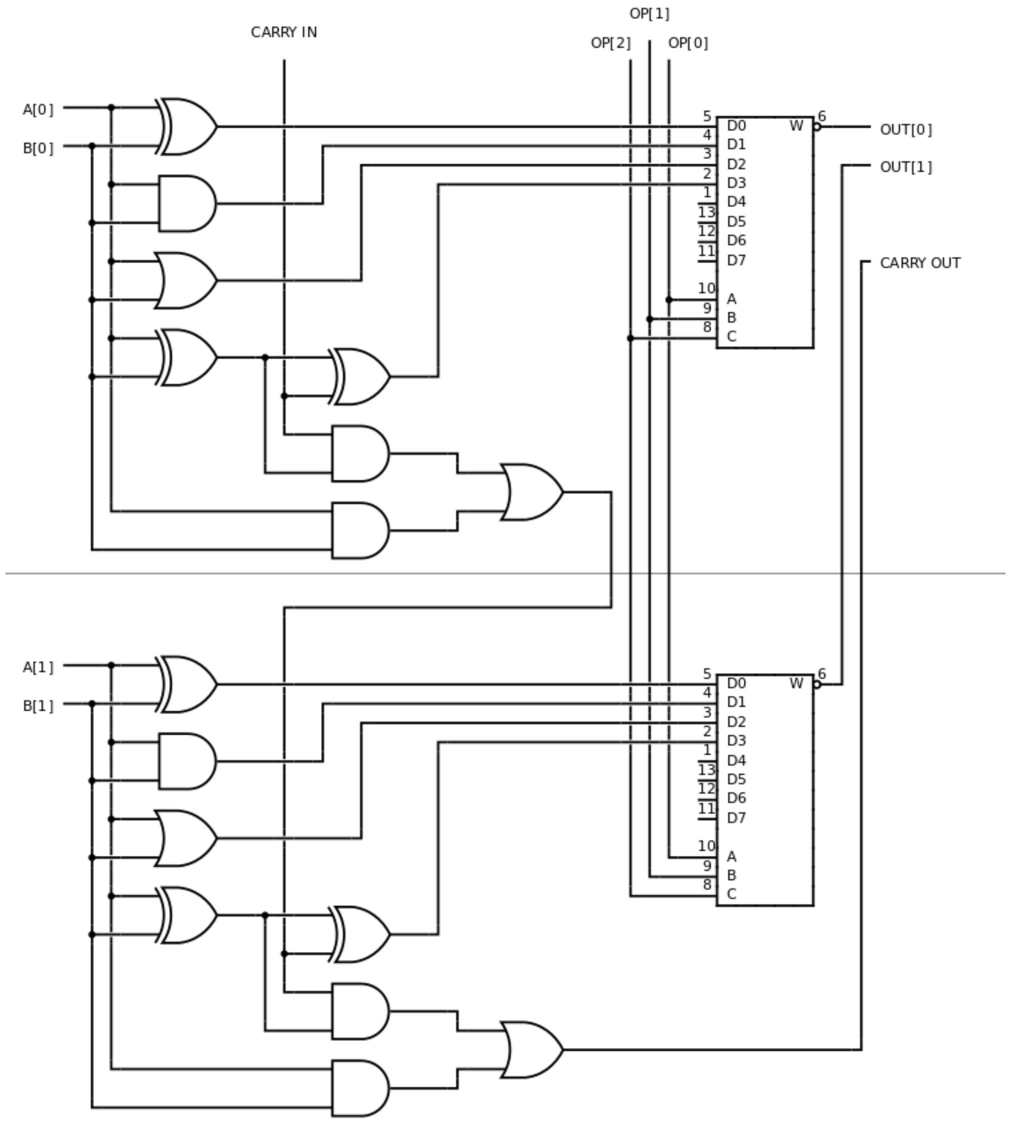
ALU units typically need to be able to perform the basic logical operations (AND, OR) and the addition operation. The inclusion of inverters on the inputs enables the same ALU hardware to perform the subtraction operation (adding an inverted operand), and the operations NAND and NOR.

A basic ALU design involves a collection of "ALU Slices", which each can perform the specified operation on a single bit. There is one ALU slice for every bit in the operand.

ALU Slice

Example: 2-Bit ALU

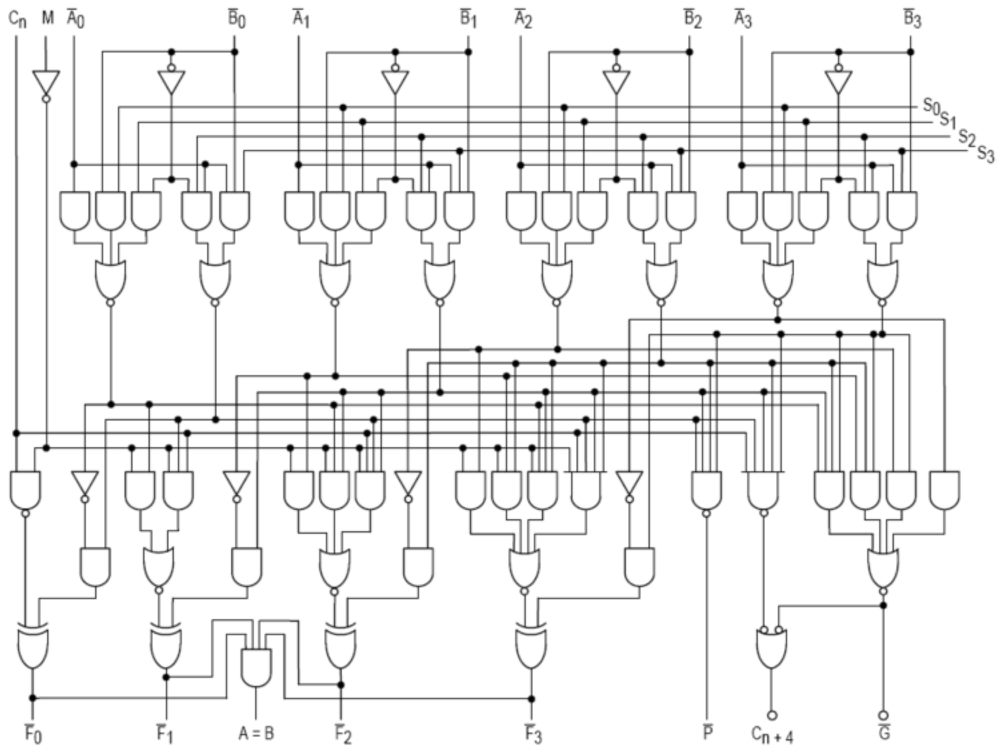
This is an example of a basic 2-bit ALU. The boxes on the right hand side of the image are multiplexers and are used to select between various operations: OR, AND, XOR, and addition.



Notice that all the operations are performed in parallel, and the select signal ("OP") is used to determine which result to pass on to the rest of the datapath. Notice that the carry signal, which is only used for addition, is generated and passed out of the ALU for every operation, so it is important that if we aren't performing addition that we ignore the carry flag.

Example: 4-Bit ALU

Here is a circuit diagram of a 4 bit ALU.



Additional Operations

Logic and addition are some of the easiest, but also the most common operations. For this reason, typical ALUs are designed to handle these operations specially, and other operations, such as multiplication and division, are handled in a separate module.

Notice also that the ALU units that we are discussing here are only for integer datatypes, not floating-point data. Luckily, once integer ALU and multiplier units have been designed, those units can be used to create floating-point units (FPU).

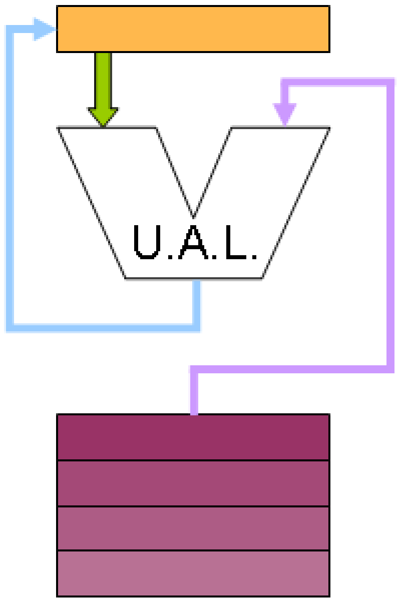
ALU Configurations

Once an ALU is designed, we need to define how it interacts with the rest of the processor. We can choose any one of a number of different configurations, all with advantages and disadvantages. Each category of instruction set architecture (ISA) -- stack, accumulator, register-memory, or register-register-load-store -- requires a different way of connecting the ALU. [2] In all images below, the orange represents memory structures internal to the CPU (registers), and the purple represents external memory (RAM).

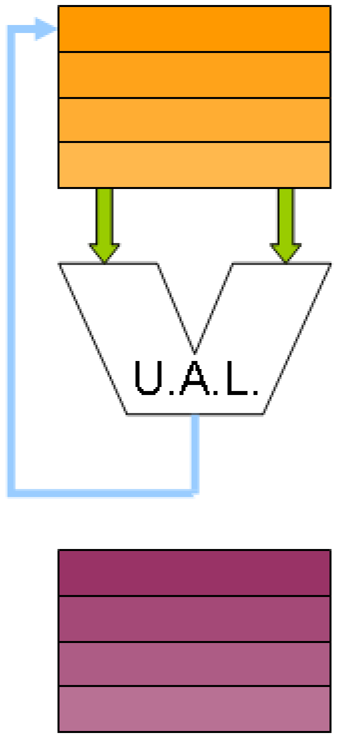
Accumulator

An accumulator machine has one special register, called the accumulator. The accumulator stores the result of every ALU operation, and is also one of the operands to every instruction. This means that our ISA can be less complicated, because instructions only need to specify one operand, instead of two operands and a destination. Accumulator architectures have simple ISAs and are typically very fast, but additional software needs to be written to load the accumulator with proper values. Unfortunately, accumulator machines are difficult 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



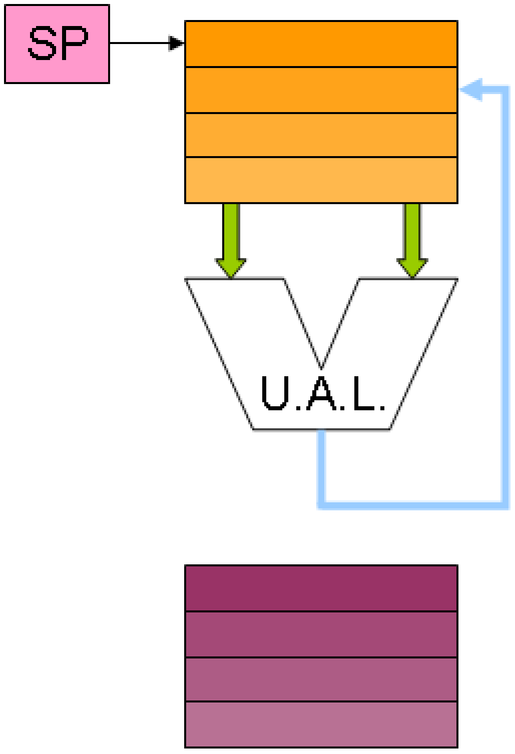
One of the more common architectures is a Register-to-register architecture, also called a 3 register operand machine. In this configuration, the programmer can specify both source operands, and a destination register. Unfortunately, the ISA needs to be expanded to include fields for both source operands and the destination operands. This requires longer instruction word lengths, and it also requires additional effort (compared to the accumulator) to write results back to the register file after execution. This write-back step can cause synchronization issues in pipelined processors (we will discuss pipelining later).

Register Stack

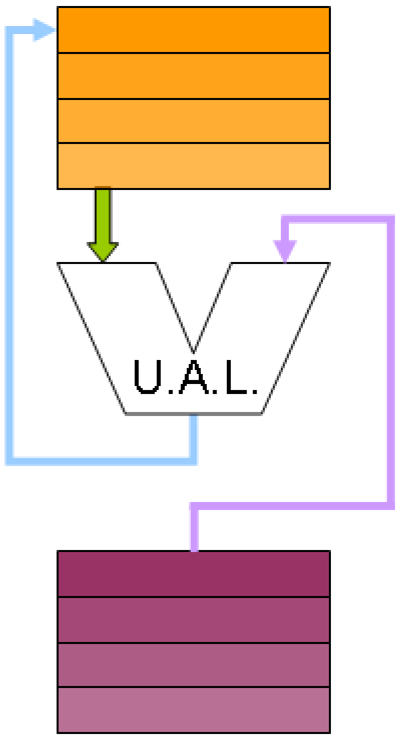
A register stack is like a combination of the Register-to-Register and the accumulator structures. In a register stack, the ALU reads the operands from the top of the stack, and the result is pushed onto the top of the stack. Complicated mathematical operations require decomposition into Reverse-Polish form, which can be difficult for programmers to use. However, many computer language compilers can produce reverse-polish notation easily because of the use of binary trees to represent instructions internally. Also, hardware needs to be created to implement the register stack, including PUSH and POP operations, in addition to hardware to detect and handle stack errors (pushing on a full stack, or popping an empty stack).

The benefit comes from a highly simplified ISA. These machines are called "0-operand" or "zero address machines" because operands don't need to be specified, because all operations act on specified stack locations.

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



Register-and-Memory



One complicated structure is a Register-and-Memory structure, like that shown at left. In this structure, one operand comes from a register file, and the other comes from external memory. In this structure, the ISA is complicated because each instruction word needs to be able to store a complete memory address, which can be very long. In practice, this scheme is not used directly, but is typically integrated into another scheme, such as a Register-to-Register scheme, for flexibility.

Some CISC architectures have the option of specifying one of the operands to an instruction as a memory address, although they are typically specified as a register address.

Complicated Structures

There are a number of other structures available, some of which are novel, and others are combinations of the types listed above. It is up to the designer to decide exactly how to structure the microprocessor, 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 a modified Register-to-Register structure for integer operations. All 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 result is stored in the first operand register. The downside to this is that the instruction words are not uniform in length, which means that the instruction fetch and decode modules of the processor need to be very complex.

A typical 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. Each operation can specify two register operands, and a third destination register. The downside is that memory reads need to be made in 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**.

References

1. CPU designers have used a variety of names for the arithmetic logic unit, including "ALU", "integer execution unit", and "E-box". Paul V. Bolotoff. "Functional Principles of Cache Memory" (http://alasir.com/articles/cache_principles/cache_hierarchy.html) 2007.
2. "Instruction Set Principles: Basic ISA Classes" (http://users.encs.concordia.ca/~tahar/coen6741/notes/Chapter2-4p.pdf) by Dr. Sofiène Tahar

- Digital Circuits/ALU
- Electronics/ALU