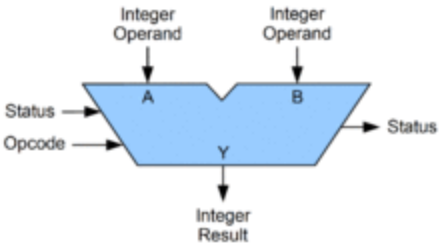
WIKIPEDIA

Arithmetic logic unit

In computing, an **arithmetic logic unit** (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.[1][2][3] This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. It is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs).[4]



A symbolic representation of an ALU and its input and output signals, indicated by arrows pointing into or out of the ALU, respectively. Each arrow represents one or more signals. Control signals enter from the left and status signals exit on the right; data flows from top to bottom.

The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed; the ALU's output is the result of the performed operation. In many designs, the ALU also has status inputs or outputs, or both, which convey information about a previous operation or the current operation, respectively, between the ALU and external status registers.

Contents

Signals

An ALU has a variety of input and output nets, which are the electrical conductors used to convey digital signals between the ALU and external circuitry. When an ALU is operating, external circuits apply signals to the ALU inputs and, in response, the ALU produces and conveys signals to external circuitry via its outputs.

Data

A basic ALU has three parallel data buses consisting of two input operands (*A* and *B*) and a result output (*Y*). Each data bus is a group of signals that conveys one binary integer number. Typically, the A, B and Y bus widths (the number of signals comprising each bus) are identical and match the native word size of the external circuitry (e.g., the encapsulating CPU or other processor).

Opcode

The *opcode* input is a parallel bus that conveys to the ALU an operation selection code, which is an enumerated value that specifies the desired arithmetic or logic operation to be performed by the ALU. The opcode size (its bus width) determines the maximum number of different operations the ALU can perform; for example, a four-bit opcode can specify up to sixteen different ALU operations. Generally, an ALU opcode is not the same as a machine language opcode, though in some cases it may be directly encoded as a bit field within a machine language opcode.

Status

Outputs

The status outputs are various individual signals that convey supplemental information about the result of the current ALU operation. General-purpose ALUs commonly have status signals such as:

- *Carry-out*, which conveys the carry resulting from an addition operation, the borrow resulting from a subtraction operation, or the overflow bit resulting from a binary shift operation.
- *Zero*, which indicates all bits of Y are logic zero.
- *Negative*, which indicates the result of an arithmetic operation is negative.
- *Overflow*, which indicates the result of an arithmetic operation has exceeded the numeric range of Y.
- *Parity*, which indicates whether an even or odd number of bits in Y are logic one.

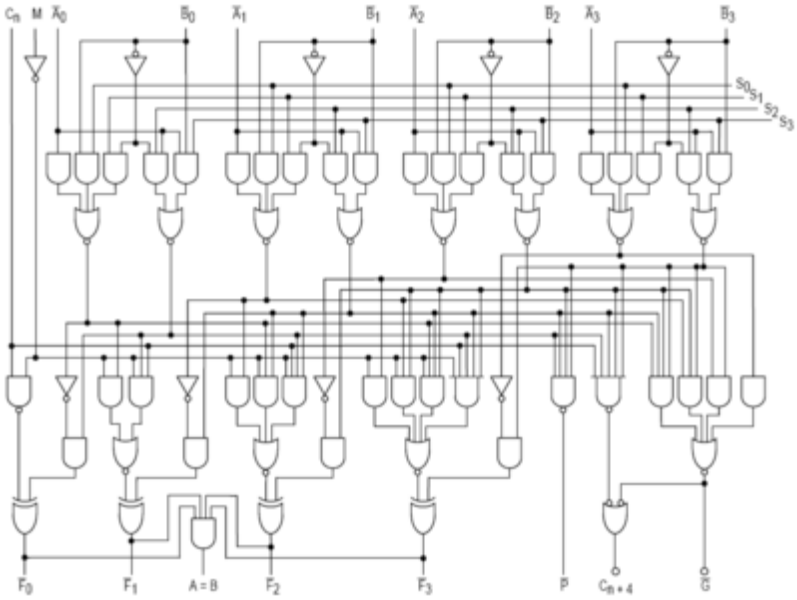
Upon completion of each ALU operation, the status output signals are usually stored in external registers to make them available for future ALU operations (e.g., to implement multiple-precision arithmetic) or for controlling conditional branching. The collection of bit registers that store the status outputs are often treated as a single, multi-bit register, which is referred to as the "status register" or "condition code register".

Inputs

The status inputs allow additional information to be made available to the ALU when performing an operation. Typically, this is a single "carry-in" bit that is the stored carry-out from a previous ALU operation.

Circuit operation

An ALU is a combinational logic circuit, meaning that its outputs will change asynchronously in response to input changes. In normal operation, stable signals are applied to all of the ALU inputs and, when enough time (known as the "propagation delay") has passed for the signals to propagate through the ALU circuitry, the result of the ALU operation appears at the ALU outputs. The external circuitry connected to the ALU is responsible for ensuring the stability of ALU input signals throughout the operation, and for allowing sufficient time for the signals to propagate through the ALU before sampling the ALU result.



The combinational logic circuitry of the 74181 integrated circuit, which is a simple four-bit ALU

In general, external circuitry controls an ALU by applying signals to its inputs. Typically, the external circuitry employs sequential logic to control the ALU operation, which is paced by a clock signal of a sufficiently low frequency to ensure enough time for the ALU outputs to settle under worst-case conditions.

For example, a CPU begins an ALU addition operation by routing operands from their sources (which are usually registers) to the ALU's operand inputs, while the control unit simultaneously applies a value to the ALU's opcode input, configuring it to perform addition. At the same time, the CPU also routes the ALU result output to a destination register that will receive the sum. The ALU's input signals, which are held stable until the next clock, are allowed to propagate through the ALU and to the destination register while the CPU waits for the next clock. When the next clock arrives, the destination register stores the ALU result and, since the ALU operation has completed, the ALU inputs may be set up for the next ALU operation.

Functions

A number of basic arithmetic and bitwise logic functions are commonly supported by ALUs. Basic, general purpose ALUs typically include these operations in their repertoires:[1][2][3][5]

Arithmetic operations

- *Add*: A and B are summed and the sum appears at Y and carry-out.
- *Add with carry*: A, B and carry-in are summed and the sum appears at Y and carry-out.
- *Subtract*: B is subtracted from A (or vice versa) and the difference appears at Y and carry-out. For this function, carry-out is effectively a "borrow" indicator. This operation may also be used to

compare the magnitudes of A and B; in such cases the Y output may be ignored by the processor, which is only interested in the status bits (particularly zero and negative) that result from the operation.

- *Subtract with borrow*: B is subtracted from A (or vice versa) with borrow (carry-in) and the difference appears at Y and carry-out (borrow out).
- *Two's complement (negate)*: A (or B) is subtracted from zero and the difference appears at Y.
- *Increment*: A (or B) is increased by one and the resulting value appears at Y.
- *Decrement*: A (or B) is decreased by one and the resulting value appears at Y.
- *Pass through*: all bits of A (or B) appear unmodified at Y. This operation is typically used to determine the parity of the operand or whether it is zero or negative, or to load the operand into a processor register.

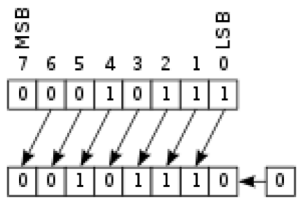
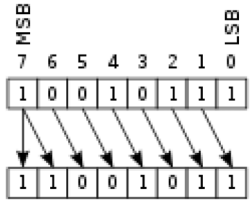
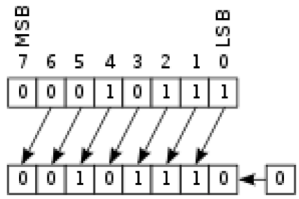
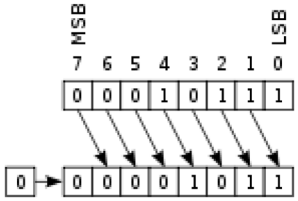
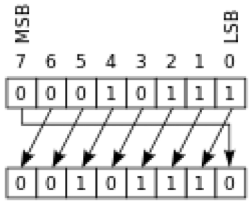
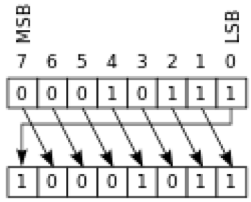
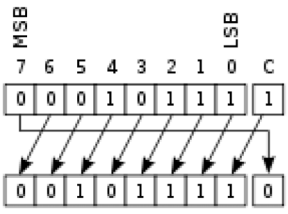
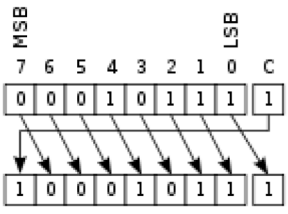
Bitwise logical operations

- *AND*: the bitwise AND of A and B appears at Y.
- *OR*: the bitwise OR of A and B appears at Y.
- *Exclusive-OR*: the bitwise XOR of A and B appears at Y.
- *Ones' complement*: all bits of A (or B) are inverted and appear at Y.

Bit shift operations

ALU shift operations cause operand A (or B) to shift left or right (depending on the opcode) and the shifted operand appears at Y. Simple ALUs typically can shift the operand by only one bit position, whereas more complex ALUs employ barrel shifters that allow them to shift the operand by an arbitrary number of bits in one operation. In all single-bit shift operations, the bit shifted out of the operand appears on carry-out; the value of the bit shifted into the operand depends on the type of shift.

Bit shift examples for an eight-bit ALU

Type	Left	Right
Arithmetic shift		
Logical shift		
Rotate		
Rotate through carry		

- *Arithmetic shift*: the operand is treated as a two's complement integer, meaning that the most significant bit is a "sign" bit and is preserved.
- *Logical shift*: a logic zero is shifted into the operand. This is used to shift unsigned integers.
- *Rotate*: the operand is treated as a circular buffer of bits so its least and most significant bits are effectively adjacent.

- *Rotate through carry*: the carry bit and operand are collectively treated as a circular buffer of bits.

Applications

Multiple-precision arithmetic

In integer arithmetic computations, **multiple-precision arithmetic** is an algorithm that operates on integers which are larger than the ALU word size. To do this, the algorithm treats each operand as an ordered collection of ALU-size fragments, arranged from most-significant (MS) to least-significant (LS) or vice versa. For example, in the case of an 8-bit ALU, the 24-bit integer `0x123456` would be treated as a collection of three 8-bit fragments: `0x12` (MS), `0x34`, and `0x56` (LS). Since the size of a fragment exactly matches the ALU word size, the ALU can directly operate on this "piece" of operand.

The algorithm uses the ALU to directly operate on particular operand fragments and thus generate a corresponding fragment (a "partial") of the multi-precision result. Each partial, when generated, is written to an associated region of storage that has been designated for the multiple-precision result. This process is repeated for all operand fragments so as to generate a complete collection of partials, which is the result of the multiple-precision operation.

In arithmetic operations (e.g., addition, subtraction), the algorithm starts by invoking an ALU operation on the operands' LS fragments, thereby producing both a LS partial and a carry out bit. The algorithm writes the partial to designated storage, whereas the processor's state machine typically stores the carry out bit to an ALU status register. The algorithm then advances to the next fragment of each operand's collection and invokes an ALU operation on these fragments along with the stored carry bit from the previous ALU operation, thus producing another (more significant) partial and a carry out bit. As before, the carry bit is stored to the status register and the partial is written to designated storage. This process repeats until all operand fragments have been processed, resulting in a complete collection of partials in storage, which comprise the multi-precision arithmetic result.

In multiple-precision shift operations, the order of operand fragment processing depends on the shift direction. In left-shift operations, fragments are processed LS first because the LS bit of each partial—which is conveyed via the stored carry bit—must be obtained from the MS bit of the previously left-shifted, less-significant operand. Conversely, operands are processed MS first in right-shift operations because the MS bit of each partial must be obtained from the LS bit of the previously right-shifted, more-significant operand.

In bitwise logical operations (e.g., logical AND, logical OR), the operand fragments may be processed in any arbitrary order because each partial depends only on the corresponding operand fragments (the stored carry bit from the previous ALU operation is ignored).

Complex operations

Although an ALU can be designed to perform complex functions, the resulting higher circuit complexity, cost, power consumption and larger size makes this impractical in many cases. Consequently, ALUs are often limited to simple functions that can be executed at very high speeds (i.e., very short propagation delays), and the external processor circuitry is responsible for performing complex functions by orchestrating a sequence of simpler ALU operations.

For example, computing the square root of a number might be implemented in various ways, depending on ALU complexity:

- *Calculation in a single clock*: a very complex ALU that calculates a square root in one operation.
- *Calculation pipeline*: a group of simple ALUs that calculates a square root in stages, with intermediate results passing through ALUs arranged like a factory production line. This circuit can accept new operands before finishing the previous ones and produces results as fast as the very complex ALU, though the results are delayed by the sum of the propagation delays of the ALU stages. For more information, see the article on instruction pipelining.
- *Iterative calculation*: a simple ALU that calculates the square root through several steps under the direction of a control unit.

The implementations above transition from fastest and most expensive to slowest and least costly. The square root is calculated in all cases, but processors with simple ALUs will take longer to perform the calculation because multiple ALU operations must be performed.

Implementation

An ALU is usually implemented either as a stand-alone integrated circuit (IC), such as the 74181, or as part of a more complex IC. In the latter case, an ALU is typically instantiated by synthesizing it from a description written in VHDL, Verilog or some other hardware description language. For example, the following VHDL code describes a very simple 8-bit ALU:

```
entity alu is
port (  -- the alu connections to external circuitry:
  A  : in signed(7 downto 0);   -- operand A
  B  : in signed(7 downto 0);   -- operand B
  OP : in unsigned(2 downto 0); -- opcode
  Y  : out signed(7 downto 0));  -- operation result
end alu;

architecture behavioral of alu is
begin
 case OP is  -- decode the opcode and perform the operation:
 when "000" =>  Y <= A + B;    -- add
 when "001" =>  Y <= A - B;    -- subtract
 when "010" =>  Y <= A - 1;    -- decrement
 when "011" =>  Y <= A + 1;    -- increment
 when "100" =>  Y <= not A;    -- 1's complement
 when "101" =>  Y <= A and B; -- bitwise AND
 when "110" =>  Y <= A or B;  -- bitwise OR
 when "111" =>  Y <= A xor B; -- bitwise XOR
 when others => Y <= (others => 'X');
 end case;
end behavioral;
```

History

Mathematician John von Neumann proposed the ALU concept in 1945 in a report on the foundations for a new computer called the EDVAC.[6]

The cost, size, and power consumption of electronic circuitry was relatively high throughout the infancy of the information age. Consequently, all serial computers and many early computers, such as the PDP-8, had a simple ALU that operated on one data bit at a time, although they often presented a wider word

size to programmers. One of the earliest computers to have multiple discrete single-bit ALU circuits was the 1948 Whirlwind I, which employed sixteen of such "math units" to enable it to operate on 16-bit words.

In 1967, Fairchild introduced the first ALU implemented as an integrated circuit, the Fairchild 3800, consisting of an eight-bit ALU with accumulator.[7] Other integrated-circuit ALUs soon emerged, including four-bit ALUs such as the Am2901 and 74181. These devices were typically "bit slice" capable, meaning they had "carry look ahead" signals that facilitated the use of multiple interconnected ALU chips to create an ALU with a wider word size. These devices quickly became popular and were widely used in bit-slice minicomputers.

Microprocessors began to appear in the early 1970s. Even though transistors had become smaller, there was often insufficient die space for a full-word-width ALU and, as a result, some early microprocessors employed a narrow ALU that required multiple cycles per machine language instruction. Examples of this includes the popular Zilog Z80, which performed eight-bit additions with a four-bit ALU.[8] Over time, transistor geometries shrank further, following Moore's law, and it became feasible to build wider ALUs on microprocessors.

Modern integrated circuit (IC) transistors are orders of magnitude smaller than those of the early microprocessors, making it possible to fit highly complex ALUs on ICs. Today, many modern ALUs have wide word widths, and architectural enhancements such as barrel shifters and binary multipliers that allow them to perform, in a single clock cycle, operations that would have required multiple operations on earlier ALUs.

ALUs can be realized as mechanical, electro-mechanical or electronic circuits[9] and, in recent years, research into biological ALUs has been carried out[10][11] (e.g., actin-based).[12]

See also

- Adder (electronics)
- Address generation unit
- Load–store unit
- Binary multiplier
- Execution unit

References

1. A.P.Godse; D.A.Godse (2009). "3". *Digital Logic Design* (https://books.google.com/books?id=-M0nkJY95GgC&pg=RA10-PA2). Technical Publications. pp. 9–3. ISBN 978-81-8431-738-1.
2. *Leadership Education and Training (LET) 2: Programmed Text* (https://books.google.com/books?id=2DSRtmn3Ri8C&pg=PA371). Headquarters, Department of the Army. 2001. pp. 371–.
3. A.P.Godse; D.A.Godse (2009). "Appendix". *Digital Logic Circuits* (https://books.google.com/books?id=6hjTpx_Whf8C&pg=RA14-PA1). Technical Publications. pp. C–1. ISBN 978-81-8431-650-6.
4. "1. An Introduction to Computer Architecture - Designing Embedded Hardware, 2nd Edition [Book]" (https://www.oreilly.com/library/view/designing-embedded-hardware/0596007558/ch01.html). *www.oreilly.com*. Retrieved 2020-09-03.
5. Horowitz, Paul; Winfield Hill (1989). "14.1.1". *The Art of Electronics* (2nd ed.). Cambridge University Press. pp. 990-. ISBN 978-0-521-37095-0.

6. Philip Levis (November 8, 2004). "Jonathan von Neumann and EDVAC" (https://web.archive.org/web/20150923211408/http://www.cs.berkeley.edu/~christos/classics/paper.pdf) (PDF). *cs.berkeley.edu*. pp. 1, 3. Archived from the original (http://www.cs.berkeley.edu/~christos/classics/paper.pdf) (PDF) on September 23, 2015. Retrieved January 20, 2015.
7. Lee Boysel (2007-10-12). "Making Your First Million (and other tips for aspiring entrepreneurs)" (https://web.archive.org/web/20121115072151/http://inst-tech.engin.umich.edu/leccap/view/ece-inv-lectures/1036). *U. Mich. EECS Presentation / ECE Recordings*. Archived from the original (http://inst-tech.engin.umich.edu/leccap/view/ece-inv-lectures/1036) on 2012-11-15.
8. Ken Shirriff. "The Z-80 has a 4-bit ALU. Here's how it works." (http://www.righto.com/2013/09/the-z-80-has-4-bit-alu-heres-how-it.html) 2013, righto.com
9. Reif, John H. (2009), Meyers, Robert A. (ed.), "Mechanical Computing: The Computational Complexity of Physical Devices" (https://doi.org/10.1007/978-0-387-30440-3_325), *Encyclopedia of Complexity and Systems Science*, New York, NY: Springer, pp. 5466–5482, doi:10.1007/978-0-387-30440-3_325 (https://doi.org/10.1007%2F978-0-387-30440-3_325), ISBN 978-0-387-30440-3, retrieved 2020-09-03
10. Lin, Chun-Liang; Kuo, Ting-Yu; Li, Wei-Xian (2018-08-14). "Synthesis of control unit for future biocomputer" (https://doi.org/10.1186/s13036-018-0109-4). *Journal of Biological Engineering*. **12** (1): 14. doi:10.1186/s13036-018-0109-4 (https://doi.org/10.1186%2Fs13036-018-0109-4). ISSN 1754-1611 (https://www.worldcat.org/issn/1754-1611). PMC 6092829 (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6092829). PMID 30127848 (https://pubmed.ncbi.nlm.nih.gov/30127848).
11. Gerd Hg Moe-Behrens. "The biological microprocessor, or how to build a computer with biological parts" (https://www.researchgate.net/publication/261257537_The_biological_microprocessor_or_how_to_build_a_computer_with_biological_parts).
12. Das, Biplab; Paul, Avijit Kumar; De, Debashis (2019-08-16). "An unconventional Arithmetic Logic Unit design and computing in Actin Quantum Cellular Automata" (https://doi.org/10.1007/s00542-019-04590-1). *Microsystem Technologies*. doi:10.1007/s00542-019-04590-1 (https://doi.org/10.1007%2Fs00542-019-04590-1). ISSN 1432-1858 (https://www.worldcat.org/issn/1432-1858).

Further reading

- Hwang, Enoch (2006). *Digital Logic and Microprocessor Design with VHDL* (http://faculty.lasierra.edu/~ehwang/digitaldesign). Thomson. ISBN 0-534-46593-5.
- Stallings, William (2006). *Computer Organization & Architecture: Designing for Performance* (http://williamstallings.com/COA/COA7e.html) (7th ed.). Pearson Prentice Hall. ISBN 0-13-185644-8.

External links