CSC 230 - Summer 2018

Architecture I

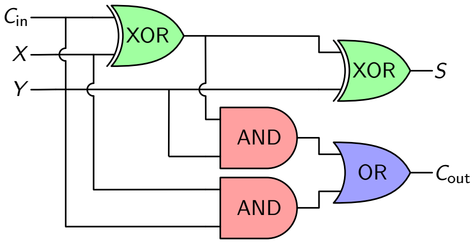
Bill Bird

Department of Computer Science
University of Victoria

May 21, 2018

Building a Calculator (1)

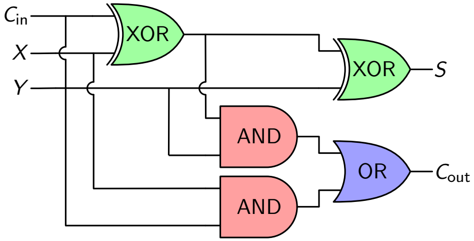
C_{in}	X	Y	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Computer processors as we know them evolved from digital logic circuits, which are built from basic binary operations like AND, OR, NOT and XOR.

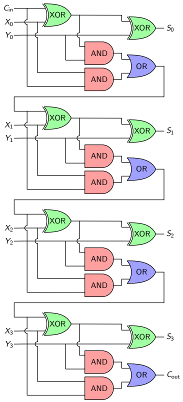
Building a Calculator (2)

C_{in}	X	Y	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



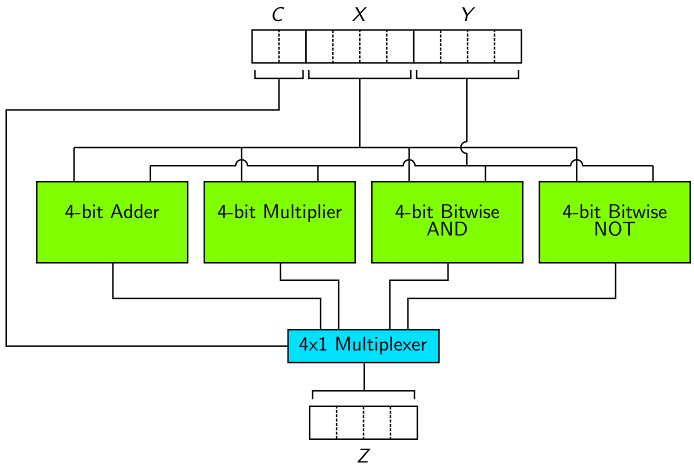
The circuit above is a **full adder**, which adds two bits X and Y to a carry-in bit C_{in}, producing two output bits S and C_{out}.

Building a Calculator (3)



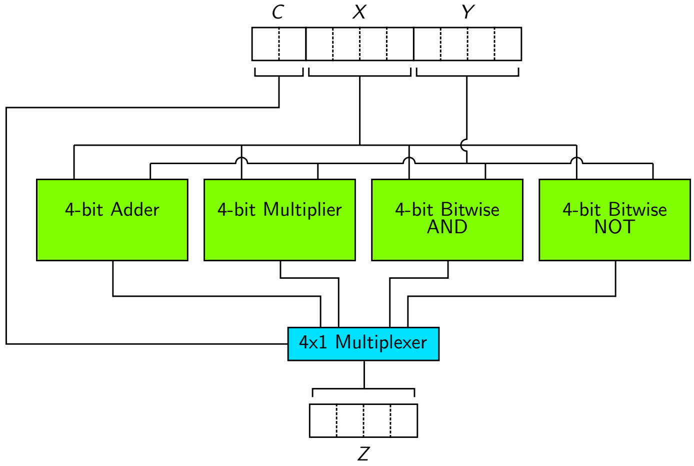
By chaining together the 1-bit full adder circuits, it is possible to add binary values with any number of bits. The circuit above adds two 4-bit operands X and Y to produce a 4-bit sum S.

Building a Calculator (4)



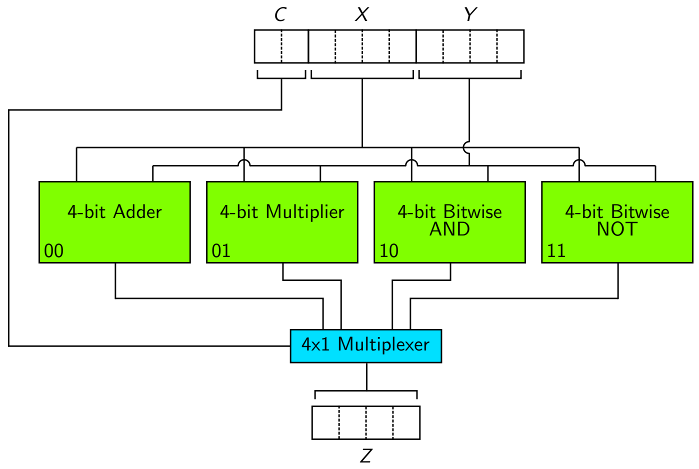
Circuits for arithmetic and bitwise operations can be constructed for any number of bits.

Building a Calculator (5)



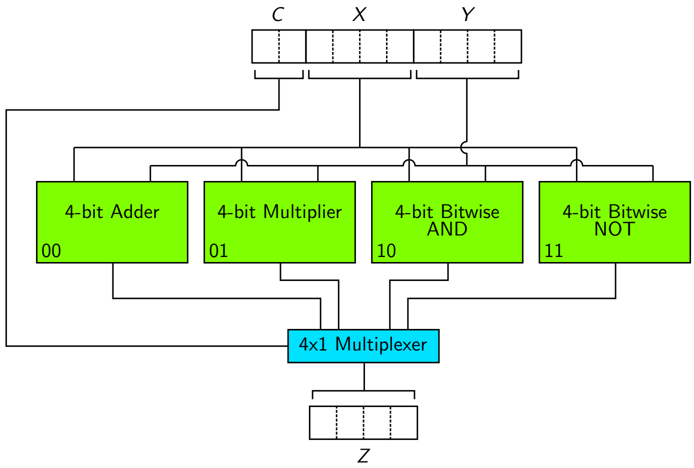
It is also possible to construct switching circuitry which chooses an operation based on some of the input bits.

Building a Calculator (6)



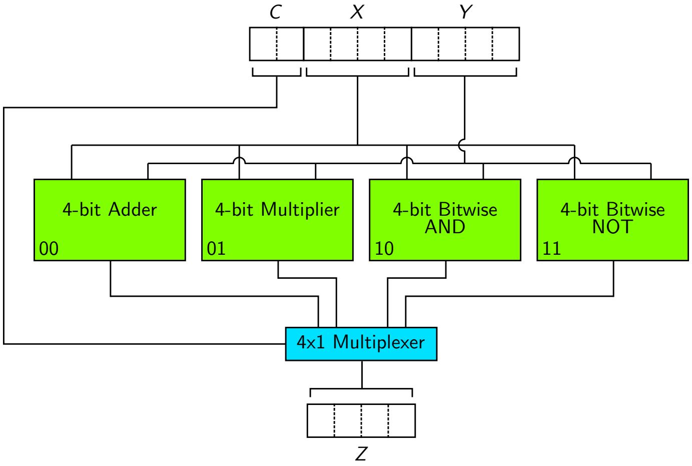
A simple calculator can be built from arithmetic circuits and switching circuits.

Building a Calculator (7)



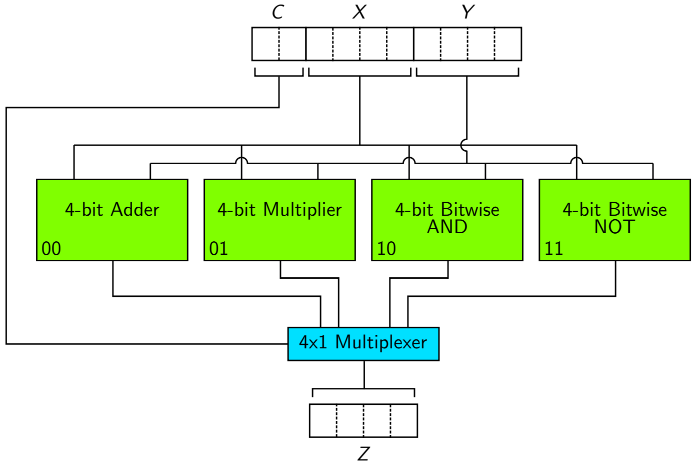
The circuit in the diagram above takes an 8-bit input, with 2 bits for control and two 4-bit operands X and Y, and performs one of $4 = 2^2$ operations, selected by the two control bits.

Building a Calculator (8)



The result is a 4-bit binary value produced by one of the four computation circuits.

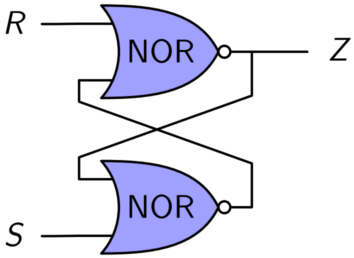
Building a Calculator (9)



This is not a computer since it is a linear process: an input is provided and an output is produced, but there is no facility for memory or programming.

Memory Circuits (1)

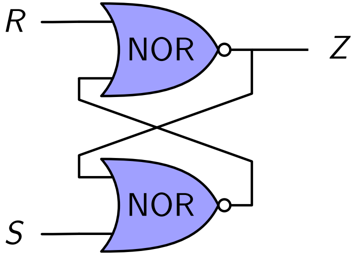
Input			**Output**
R	S	Z_{old}	Z_{new}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



It is possible to construct memory circuits from digital logic gates by using feedback loops. The circuit above is an **SR latch**: When S and R are 0, the bit Z is 'remembered' indefinitely. Momentarily setting R or S to 1 sets Z to 0 or 1 (respectively).

Memory Circuits (2)

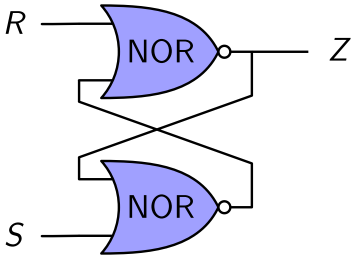
Input			Output
R	S	Z_{old}	Z_{new}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



(Digital logic circuits are covered in CSC 355; you do not have to know how to use them in this course)

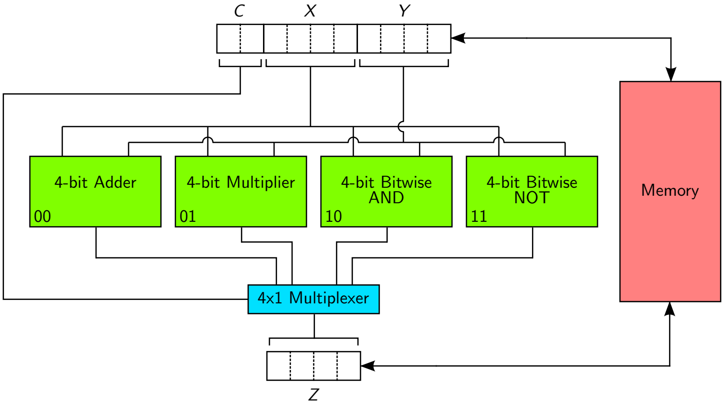
Memory Circuits (3)

Input			**Output**
R	S	Z_{old}	Z_{new}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



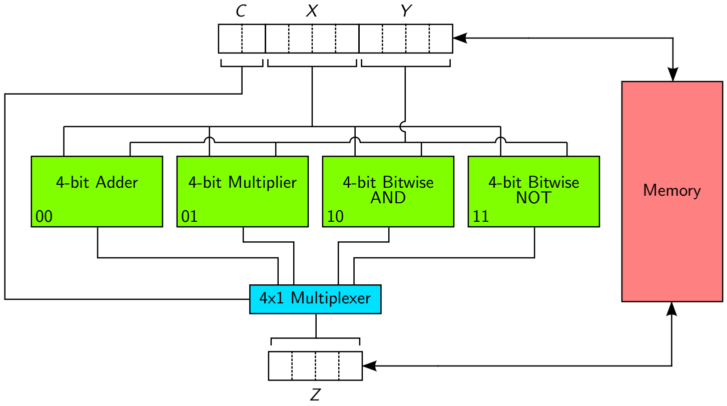
Combining memory circuits with the calculator style processing circuitry from earlier results in a **programmable computer**.

Memory Circuits (4)



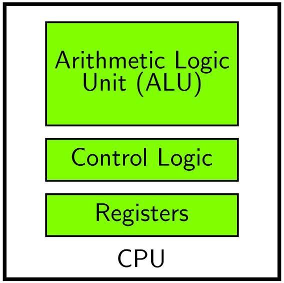
Using memory circuits to store operands (**data**) allows complicated sequences of operations to be performed, and using memory to store control information (**code**) allows the sequence of operations to be easily modified.

Memory Circuits (5)



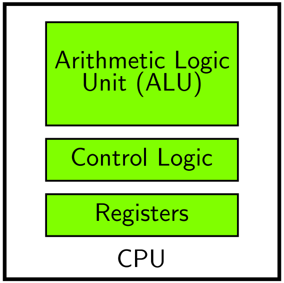
A computer system in which the program can be represented in memory is called a **stored program architecture**.

Stored Program Architectures (1)



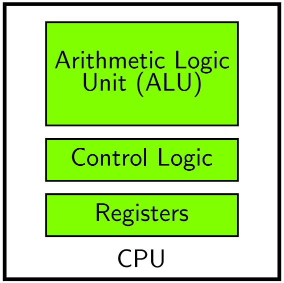
Fundamentally, a CPU can consist of an **arithmetic logic unit** (ALU), containing all of the calculation circuitry (arithmetic, bit-wise operations, comparisons, etc.), **control logic** to choose which execution path to follow and a small number of internal memory locations called **registers**.

Stored Program Architectures (2)



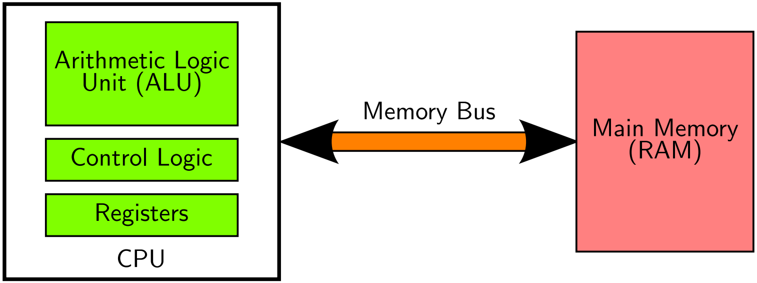
Registers are used to store the operand values for calculations, as well as the control bits for which operation to use and the results of operations.

Stored Program Architectures (3)



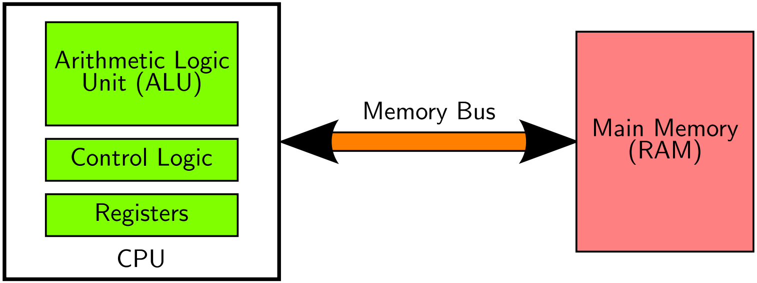
We will see that one of the key internal differences between CPUs is actually the register organization, since most CPUs have a fairly consistent set of basic arithmetic and control operations.

Stored Program Architectures (4)



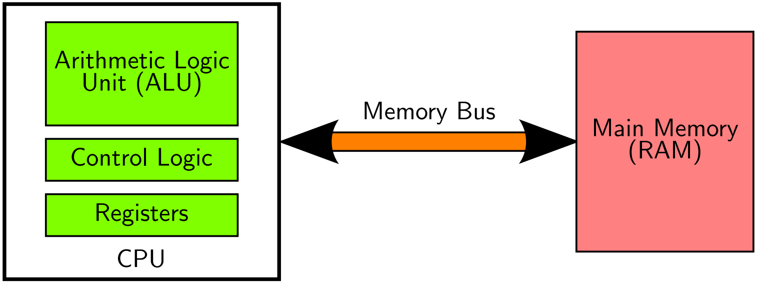
The processor is not capable of operating in isolation, since the internal registers are not intended to store an entire program (just to provide space for a few operations). The addition of a larger array of **main memory** allows large programs (and the associated data) to be stored and executed

Stored Program Architectures (5)



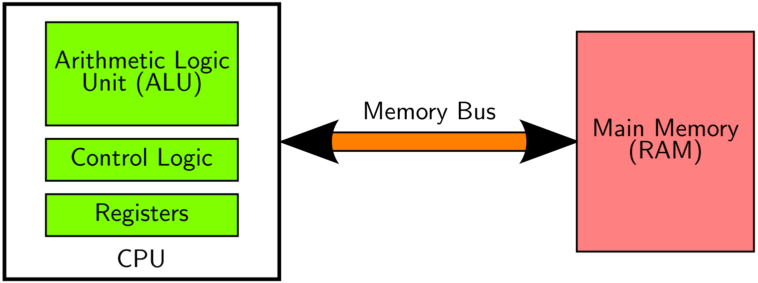
Even if the memory is physically on the same chip as the CPU (as is the case in some embedded architectures), we consider it to be a distinct and separate component, connected by an interlink called a **bus**.

Stored Program Architectures (6)



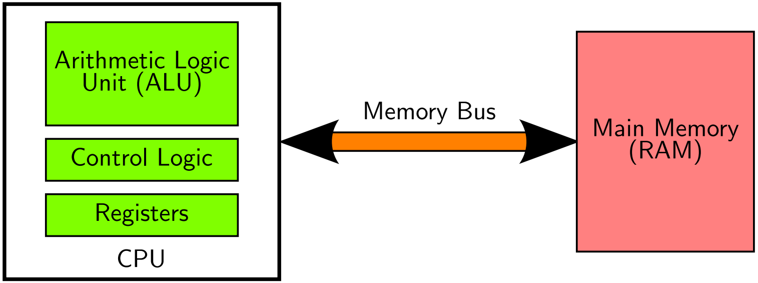
The main memory is used to store information for the running program, but also the program itself (in the form of binary machine code instructions).

Stored Program Architectures (7)



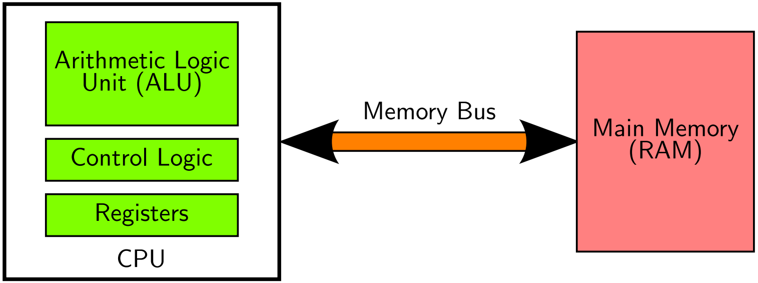
The program itself is retrieved, one instruction at a time, from memory. Processors usually keep track of where in memory the current instruction is using a special register called the **program counter** (PC).

Stored Program Architectures (8)



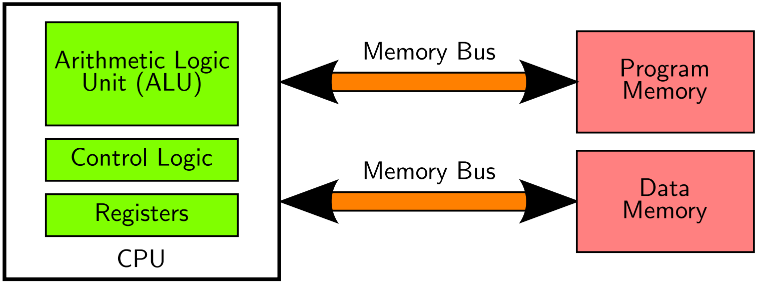
This form of stored program computer, in which the program itself is represented as data, is called a **von Neumann architecture**.

Stored Program Architectures (9)



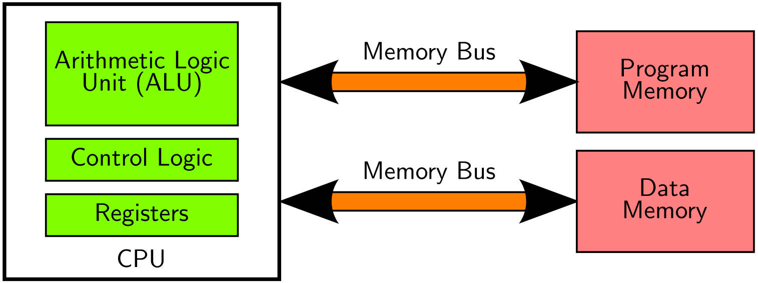
In a von Neumann architecture, since the program is indistinguishable from data, it is easy for a program to read (or modify) its own code while it is running or to interleave code and data.

Stored Program Architectures (10)



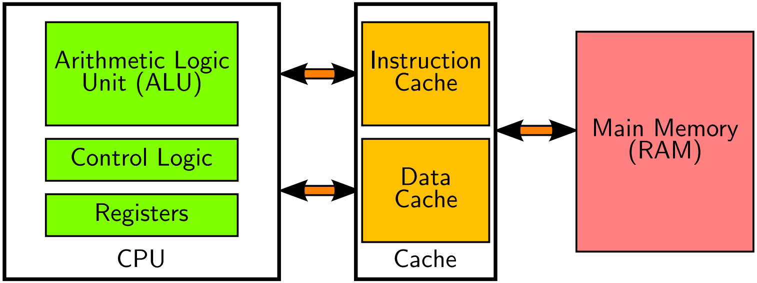
Some architectures use separate memory banks for the program and the data. These architectures are usually called **Harvard architectures**, after the Harvard Mark I, an early stored program computer.

Stored Program Architectures (11)



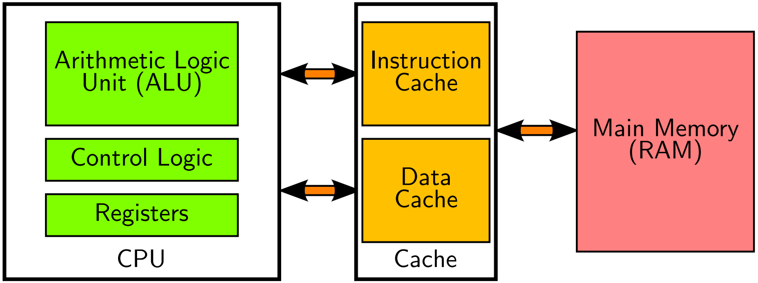
Harvard architectures have the advantage of allowing simultaneous access to instructions and data, since the two use independent memory busses. However, maintaining two independent memory systems can be inefficient and costly.

Stored Program Architectures (12)



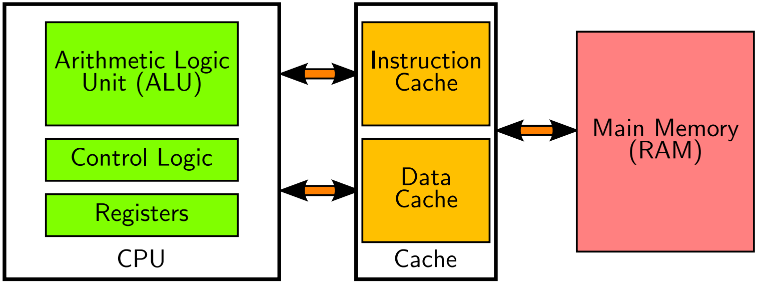
Computers running x86-based architectures are usually treated as von Neumann machines, which is a reasonable abstraction, since both programs and data are stored in main memory.

Stored Program Architectures (13)



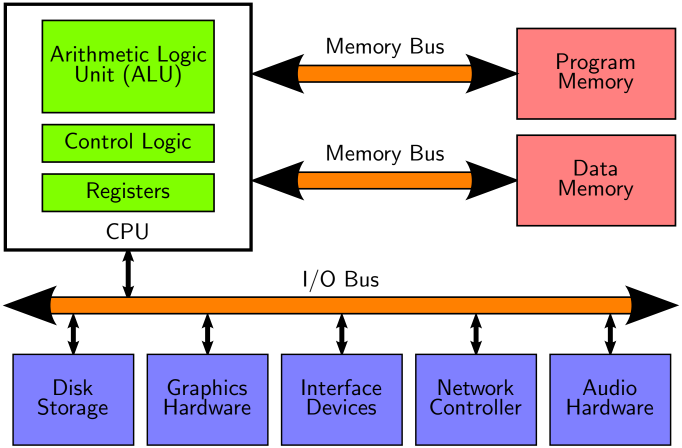
However, x86-based architectures actually combine elements of both the von Neumann and Harvard models, since the CPU contains separate **cache** structures to store instructions and data, even though both are ultimately loaded from the same bank of memory.

Stored Program Architectures (14)



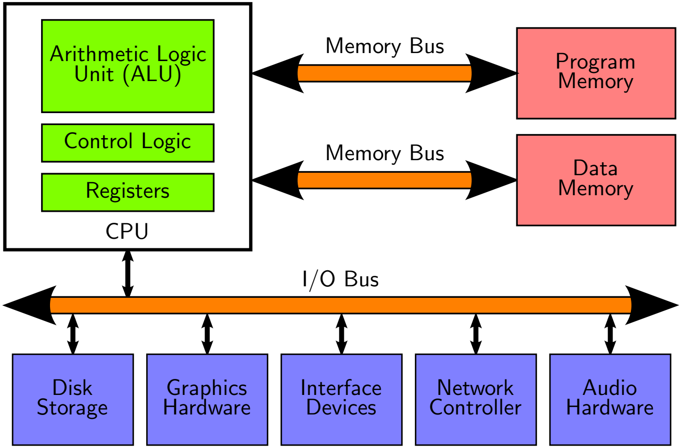
A cache consists of a small amount of fast memory, which can help to improve performance when the main memory is slow (but large and inexpensive). We will cover caching near the end of the course.

Peripherals (1)



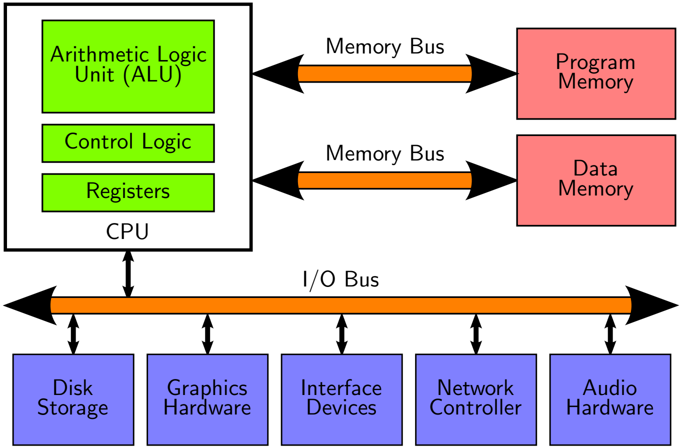
Systems normally consist of more than just a processor and memory, and peripheral devices can interact with CPUs in various ways.

Peripherals (2)



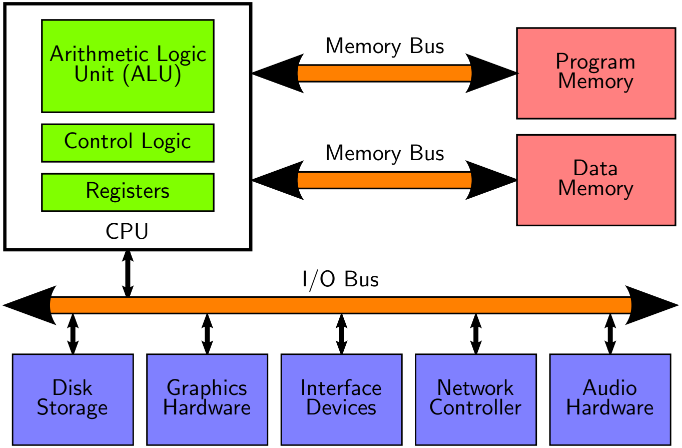
One possible model is **port-based** I/O, in which I/O devices are linked to the CPU by a dedicated **I/O bus**, and each device is identified by a particular I/O port.

Peripherals (3)



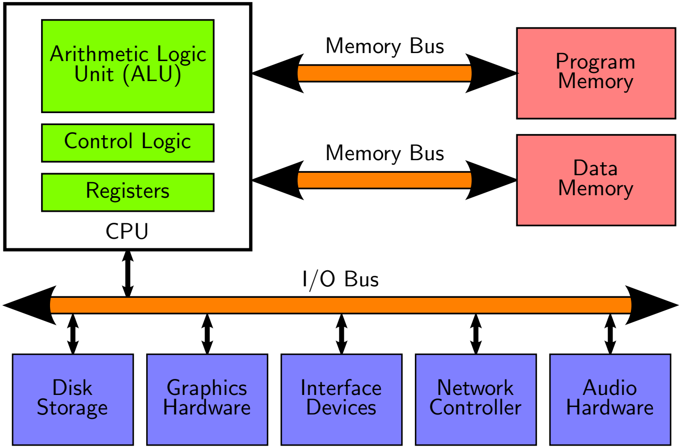
To interact with a device, the CPU sends or receives data from its I/O port. There can be issues related to synchronization between different devices and the CPU.

Peripherals (4)



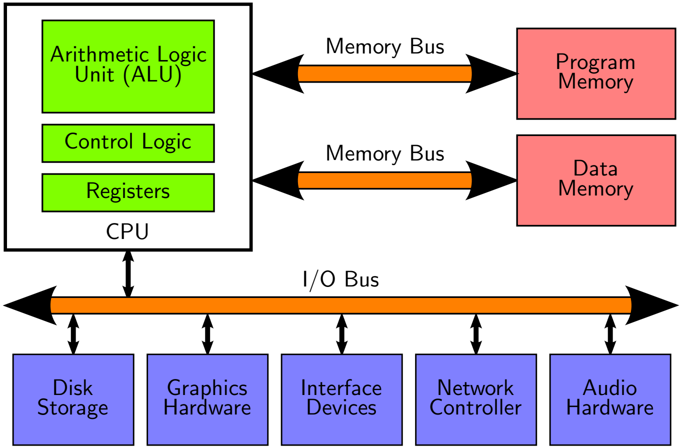
Interactions between peripherals and the CPU can be mediated in various ways.

Peripherals (5)



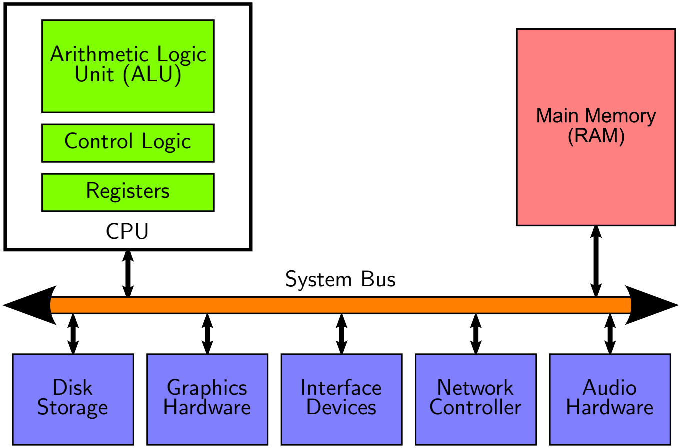
A passive option is **polling**, where the I/O device is queried for information (and unless a query occurs, no information is sent to the CPU).

Peripherals (6)



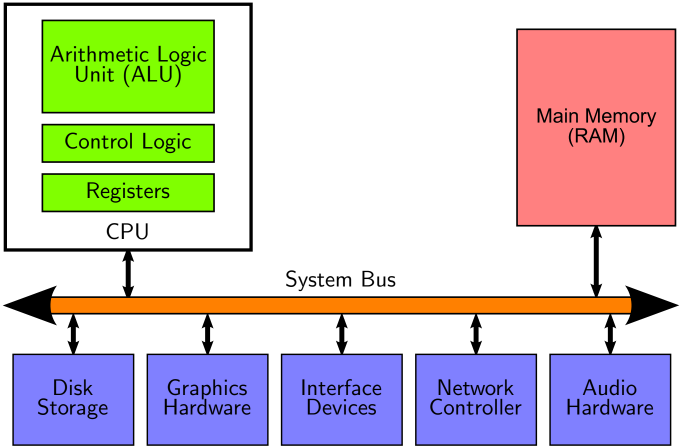
An active option is **interrupt processing**, where the I/O device can trigger an event inside the CPU (an **interrupt**) to signal that attention is needed.

Peripherals (7)



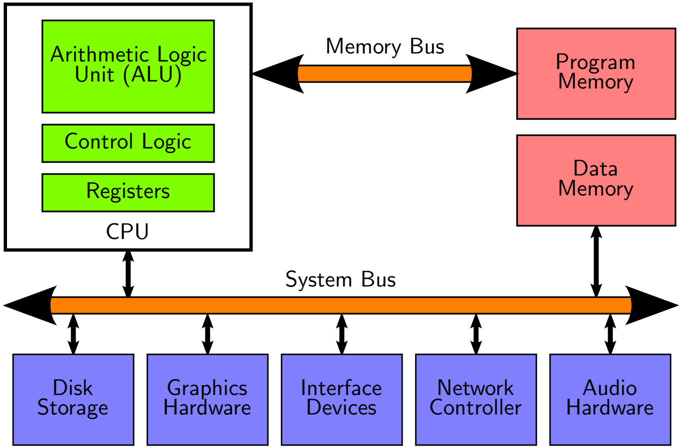
An alternative model for peripheral interaction is **memory mapping**, in which all devices and the main memory share a common **system bus**.

Peripherals (8)



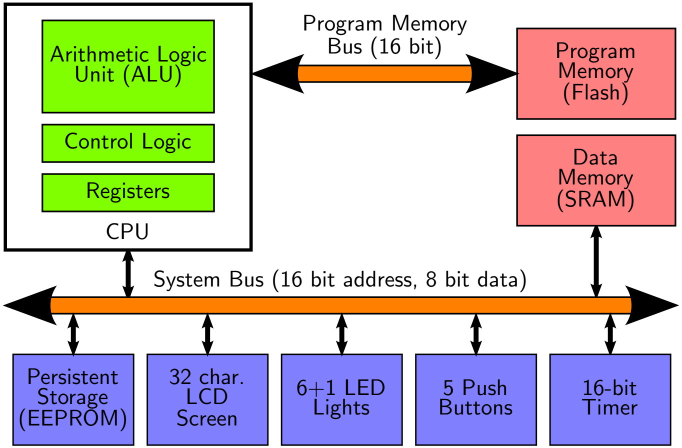
Each device (or memory unit) is identified based on its memory address, allowing the CPU to communicate with devices in the same way it communicates with memory.

Peripherals (9)



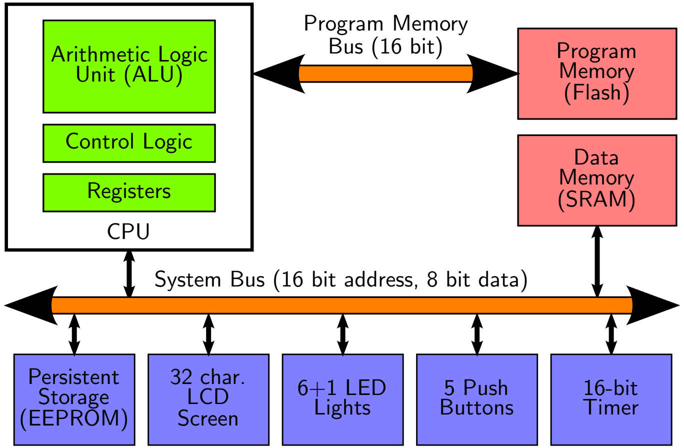
In Harvard architectures, it's possible for only one of the memories to lie on the system bus.

The AVR architecture (1)



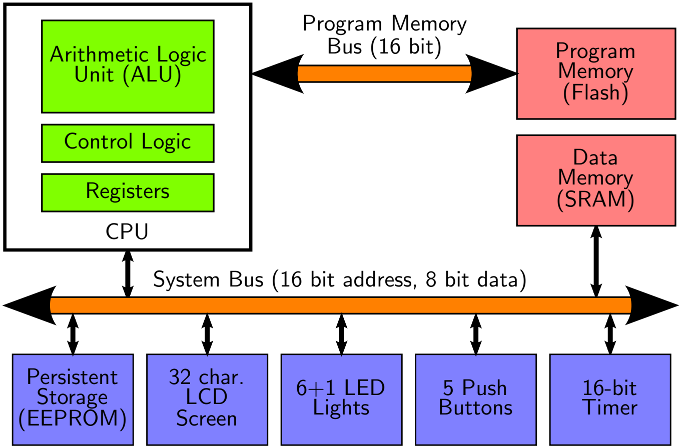
For this course, we will focus on the AVR architecture by Atmel, using the ATmega2560 processor.

The AVR architecture (2)



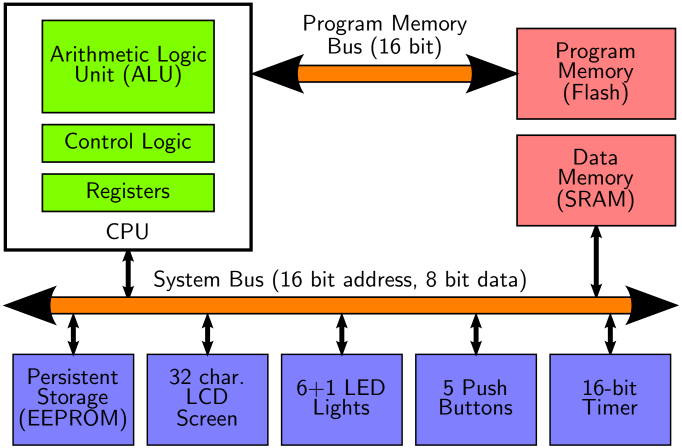
AVR is a Harvard architecture, which allows the program memory and data memory to be conveniently separated into different physical media.

The AVR architecture (3)



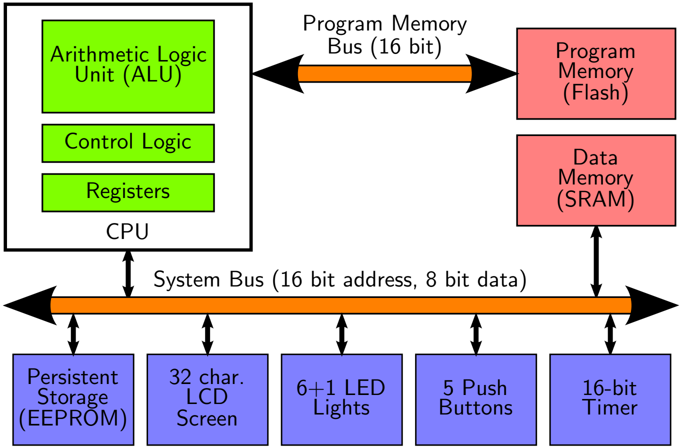
Flash memory, which is non-volatile and fast to read, but slow to write and has a limited number of write cycles, is used for the program memory.

The AVR architecture (4)



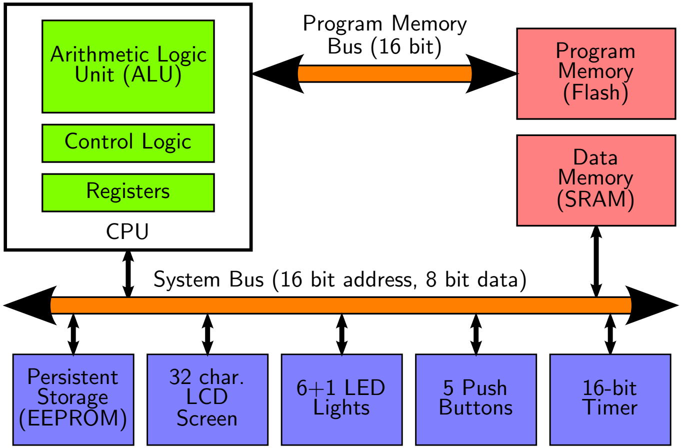
SRAM, which is fast to read and write but volatile (loses its data when power is lost) is used for data memory.

The AVR architecture (5)



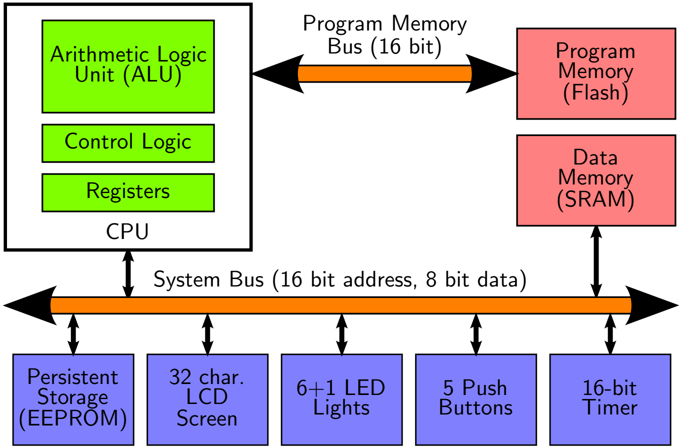
A third non-volatile memory bank called EEPROM is also available as a peripheral, but we will not be using it.

The AVR architecture (6)



Our AVR devices are equipped with an LCD screen, six LED lights (plus an extra LED light which is covered by the screen) and five push buttons.

The AVR architecture (7)



Internally, the device also has an onboard timer chip.