Interfacing Memory To a Microprocessor Bus

In order to design the interface, the timing specifications for both the memory and microprocessor must be satisfied. When writing to memory, the setup and hold time specifications for the memory must be satisfied, and when reading from memory, the setup and hold time specifications for the microprocessor bus must be satisfied. If the memory is slow, it may be necessary to insert wait states in the bus cycle. We design the interface between a 486 bus and a small static RAM memory, and then we write code to test the interface timing. We use the static RAM and 486 bus interface models that we have already developed. Figure shows how the bus interface is connected to the static RAM memory.

The memory consists of four static RAM chips, and each RAM chip contains 2^{15} = 32,768 8-bit words. The four chips operate in parallel to give a memory that is 32 bits wide. Data bits 31-24 are connected to the first chip, bits 23-16 to the second chip, etc. The lower 15 bits of the address bus are connected in parallel to all four chips. The system includes a memory controller that generates WE and CS signals for the memory and returns a Rdy signal to the bus interface to indicate completion of a read or write bus cycle. We will assign the address range 0 through 32,767 to the static RAM. In general, a complete 486 system would have a large dynamic RAM memory and I/O interface cards connected to the address and data busses. To allow for expansion to the system, we use an address decoder, so the static RAM is selected only for the specified address range. When an address in the range 0 to 32,767 is detected, the decoder outputs CSI = 1, which activates the memory controller.

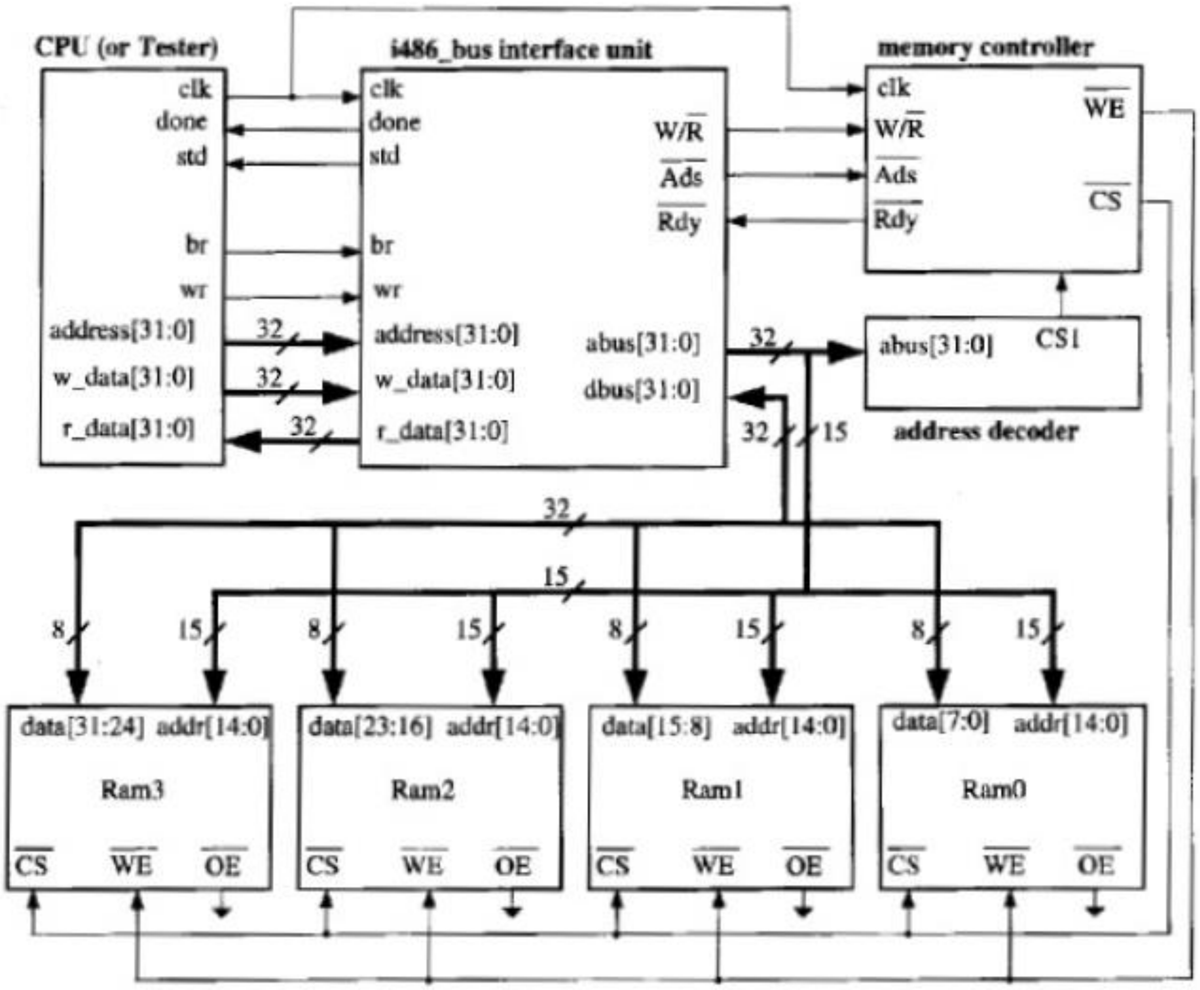


Fig 5.14 Interfacing Memory to a Microprocessor Bus

Design of a UART

A UART (Universal Asynchronous Receiver and Transmitter) is a device allowing the reception and transmission of information, in a serial and asynchronous way

A UART allows the communication between a computer and several kinds of devices (printer, modem, etc), interconnected via an RS-232 cable

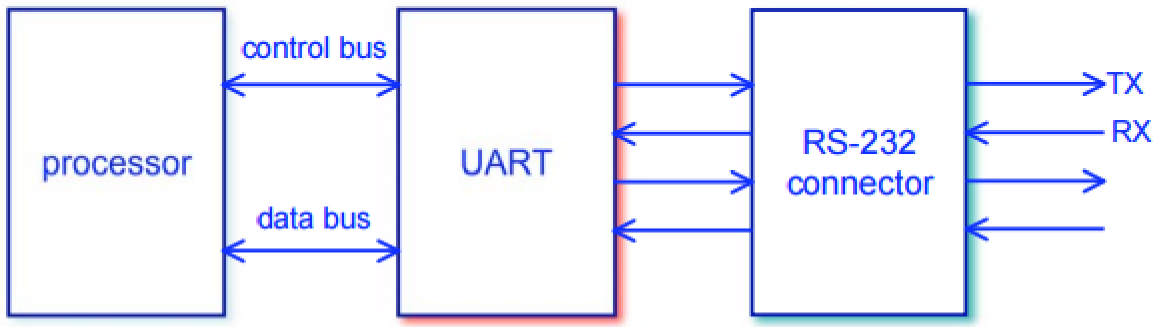


Fig 5.15 Block Diagram

Serial transmission

Data transmission is made by the UART in a serial way, by 11-bit blocks:

- a 0 bit marks the starting point of the block
- eight bits for data
- one parity bit
- a 1 bit marking the end of the block
- The transmission and reception lines should hold a 1 when no data is transmitted

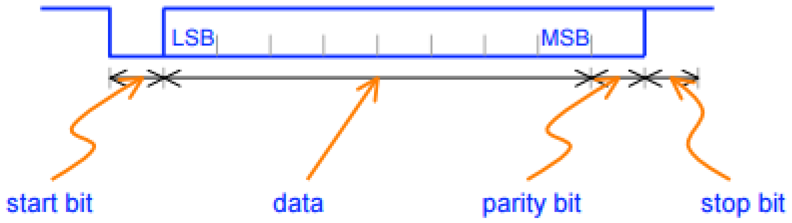


Fig 5.16 Data Transmission in UART

- The first transmitted bit is the LSB (least significant bit)
- The parity bit is set to 1 or 0, depending on the number of 1's transmitted: if even parity is used, this number should be even; if odd parity is used, this number should be odd. If the chosen parity is not respected in the block, a transmission error should be detected
- The transmission speed is fixed, measured in bauds

Implementation

A possible entity for the UART is:

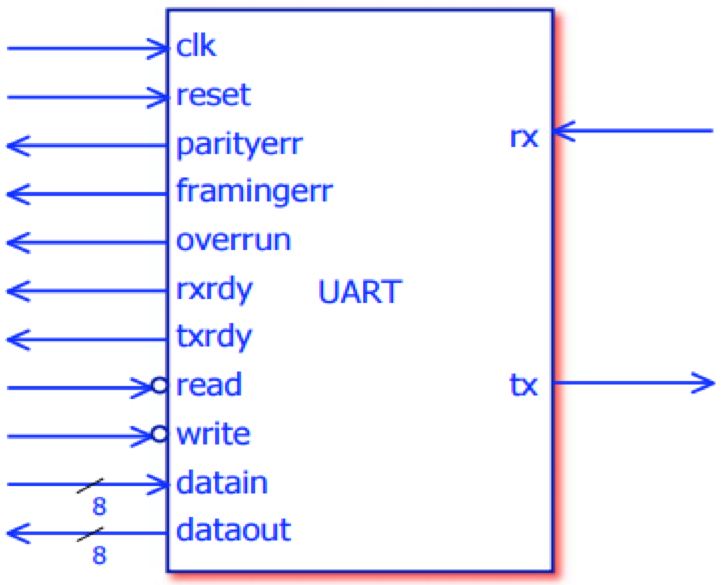


Fig 5.17 UART entity

The meaning of signals is:

• parityerr: error during the block reception

• framingerr: format error during the block reception

• overrun: a new data is arrived before reading of the precedent data

• rxrdy: a new data is arrived and it's ready for reading

• txrdy: a new data is ready for sending

• read: reading of the receiver's data is activated

• write: writing of the emitter's data is activated

• data: 8-bit data, read or written

• tx: output bit

• rx: input bit

The UART can be divided into two modules: the receiver and the emitter

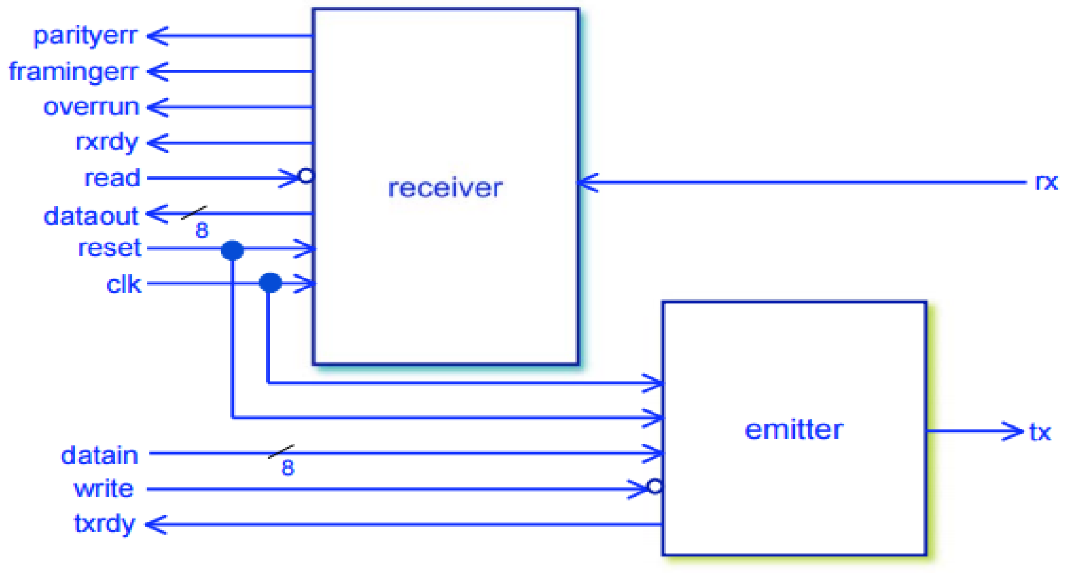


Fig 5.18 UART Module

Data emission:

• to test if signal txrdy is active. If yes, a 8-bit data can be written in the emitter

• to place the 8-bit data in the input and to active the write signal

• the UART sends the 8 bits, via the tx signal. During transmission, the txrdy signal should be inactive

• at the end of the emission, txrdy should be active again et tx set to 1

Data reception:

• the 8 bits of information arrive in a serial way, at any moment, via the rx signal. The starting point is given par a 0 value of rx

• the UART places the 8 bits in a parallel way over dataout, and announces their availability setting rxrdy active

• the information reading is made active with the read signal

Static RAM Memory

Random Access Memory (RAM) is a form of computer data storage. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. In contrast, with other direct-access data storage media such as hard disks, CD-RWs, DVD-RWs and the older drum memory, the time required to read and write data items varies significantly depending on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement.

RAM contains multiplexing and de-multiplexing circuitry, to connect the data lines to the addressed storage for reading or writing the entry. Usually more than one bit of storage is accessed by the same address, and RAM devices often have multiple data lines and are said to be '8-bit' or '16-bit' etc. devices. In today's technology, random-access memory takes the form of integrated circuits. RAM is normally associated with volatile types of memory (such as DRAM memory modules), where stored information is lost if power is removed, although many efforts have been made to develop non-volatile RAM chips. Other types of non-volatile memories exist that allow random access for read operations, but either do not allow write operations or have other kinds of limitations on them. These include most types of ROM and a type of flash memory called *NOR-Flash*.

The two widely used forms of modern RAM are static RAM (SRAM) and dynamic RAM (DRAM). In SRAM, a bit of data is stored using the state of a six transistor memory cell. This form of RAM is more expensive to produce, but is generally faster and requires less dynamic power than DRAM. In modern computers, SRAM is often used as cache memory for the CPU. DRAM stores a bit of data using a transistor and capacitor pair, which together comprise a DRAM memory cell. The capacitor holds a high or low charge (1 or 0, respectively), and the transistor acts as a switch that lets the control circuitry on the chip read the capacitor's state of charge or change it. As this form of memory is less expensive to produce than static RAM, it is the predominant form of computer memory used in modern computers.

Both static and dynamic RAM are considered *volatile*, as their state is lost or reset when power is removed from the system. By contrast, read-only memory (ROM) stores data by permanently enabling or disabling selected transistors, such that the memory cannot be altered. Writeable variants of ROM (such as EEPROM and flash memory) share properties of both ROM and RAM, enabling data to persist without power and to be updated without requiring special equipment. Static Random Access Memory (Static RAM or SRAM) is a type of RAM that holds data in a static form, that is, as long as the memory has power. Unlike dynamic RAM, it does not need to be refreshed. SRAM stores a bit of data on four transistors using two cross-coupled inverters. The two stable states characterize 0 and 1. During read and write operations another two access transistors are used to manage the availability to a memory cell. To store one memory bit it requires six metal-oxide-semiconductor field- effect transistors (MOFSET). The term *static* differentiates SRAM from DRAM (*dynamic* random access memory) which must be periodically refreshed. SRAM is faster and more expensive than DRAM; it is typically used for CPU cache while DRAM is used for a computer's main memory.

Verilog Code for Static RAM Memory:

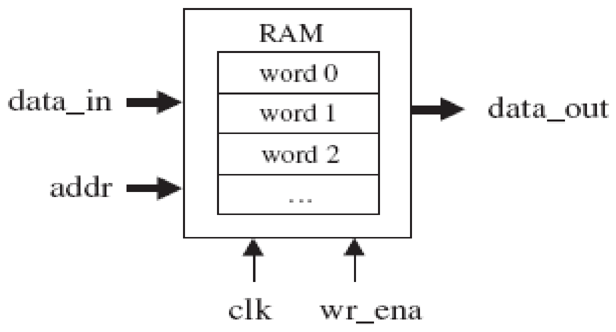
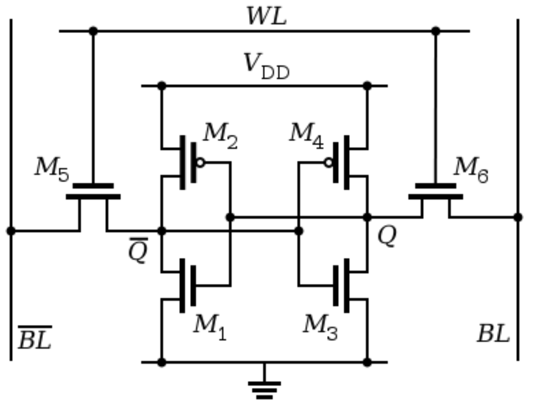


Fig 5.19 Static RAM Memory

```
module ram(clk, wr_en, data_in, addr, data_out);
  input clk;
  input wr_en;
  input [7:0] data_in;
  input [3:0] addr;
  output [7:0] data_out;
        reg [7:0] data_out;
        reg [7:0]mem [0:15];
        always@(posedge(clk),wr_en,data_in,addr)
```

```
            if(clk)
            begin
            if(wr_en)
            mem[addr]=data_in;
            else
            data_out=mem[addr];
            end
        endmodule
```

CMOS SRAM cell:



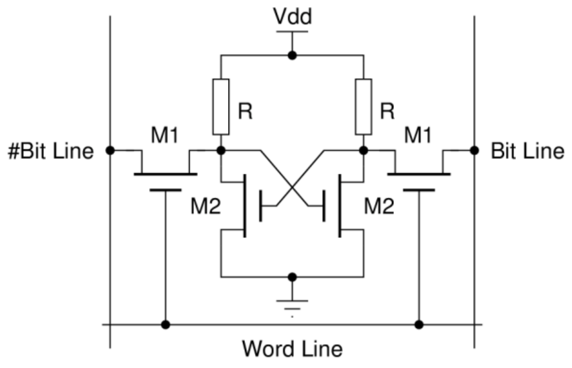


Fig 5.20 (a) Six-Transistor CMOS SRAM cell (b) Four Transistor SRAM cell.

A typical SRAM cell is made up of six MOSFETs. Each bit in an SRAM is stored on four transistors (M1, M2, M3, M4) that form two cross-coupled inverters. This storage cell has two stable states which are used to denote **0** and **1**. Two additional *access* transistors serve to control the access to a storage cell during read and write operations. In addition to such six-transistor (6T) SRAM, other kinds of SRAM chips use 4, 8, 10 (4T, 8T, 10T SRAM), or more transistors per bit. Four-transistor SRAM is quite common in stand-alone SRAM devices (as opposed to SRAM used for CPU caches), implemented in special processes with an extra layer of polysilicon, allowing for very high-resistance pull-up resistors.[8] The principal drawback of using 4T SRAM is increased static powerdue to the constant current flow through one of the pull-down transistors. This is sometimes used to implement more than one (read and/or write) port, which may be useful in certain types of video memoryand register files implemented with multi-ported SRAM circuitry.

Generally, the fewer transistors needed per cell, the smaller each cell can be. Since the cost of processing a silicon wafer is relatively fixed, using smaller cells and so packing more bits on one wafer reduces the cost per bit of memory.

Memory cells that use fewer than four transistors are possible – but, such 3T or 1T cells are DRAM, not SRAM (even the so-called 1T-SRAM).

Access to the cell is enabled by the word line (WL in figure) which controls the two *access* transistors M_5 and M_6 which, in turn, control whether the cell should be connected to the bit lines: BL and BL. They are used to transfer data for both read and write operations. Although it is not strictly necessary to have two bit lines, both the signal and its inverse are typically provided in order to improve noise margins.

During read accesses, the bit lines are actively driven high and low by the inverters in the SRAM cell. This improves SRAM bandwidth compared to DRAMs – in a DRAM, the bit line is connected to storage capacitors and charge sharing causes the bitline to swing upwards or downwards. The symmetric structure of SRAMs also allows for differential signaling, which makes small voltage swings more easily detectable. Another difference with DRAM that contributes to making SRAM faster is that commercial chips accept all address bits at a time. By comparison, commodity DRAMs have the address multiplexed in two halves, i.e. higher bits followed by lower bits, over the same package pins in order to keep their size and cost down.

The size of an SRAM with *m* address lines and *n* data lines is 2^m words, or $2^m \times n$ bits. The most common word size is 8 bits, meaning that a single byte can be read or written to each of 2^m different words within the SRAM chip. Several common SRAM chips have 11 address lines (thus a capacity of $2^m = 2{,}048 = 2k$ words) and an 8-bit word, so they are referred to as "2k × 8 SRAM".

SRAM Operation:

An SRAM cell has three different states: *standby* (the circuit is idle), *reading* (the data has been requested) or *writing* (updating the contents). SRAM operating in read mode and write modes should have "readability" and "write stability", respectively. The three different states work as follows:

Standby

If the word line is not asserted, the *access* transistors M_5 and M_6 disconnect the cell from the bit lines. The two cross-coupled inverters formed by M_1 – M_4 will continue to reinforce each other as long as they are connected to the supply.

Reading

In theory, reading only requires asserting the word line WL and reading the SRAM cell state by a single access transistor and bit line, e.g. M_6, BL. Nevertheless, bit lines are relatively long and have large parasitic capacitance. To speed up reading, a more complex process is used in practice: The read cycle is started by precharging both bit lines BL and BL, i.e., driving the bit lines to a threshold voltage (midrange voltage between logical **1** and **0**) by an external module. Then asserting the word line WL enables both the access transistors M_5 and M_6, which causes the bit line BL voltage to either slightly drop (bottom NMOS transistor M_3 is ON and top PMOS transistor M_4 is off) or rise (top PMOS transistor M_4 is on). It should be noted that if BL voltage rises, the BL voltage drops, and vice versa. Then the BL and BL lines will have a small voltage difference between them. A sense amplifier will sense which line has the higher voltage and thus determine whether there was **1** or **0** stored. The higher the sensitivity of the sense amplifier, the faster the read operation.

Writing

The write cycle begins by applying the value to be written to the bit lines. If we wish to write a **0**, we would apply a **0** to the bit lines, i.e. setting BL to **1** and BL to **0**. This is similar to applying a reset pulse to an SR-latch, which causes the flip flop to change state. A **1** is written by inverting the values of the bit lines. WL is then asserted and the value that is to be stored is latched in. This works because the bit line input-drivers are designed to be much stronger than the relatively weak transistors in the cell itself so they can easily override the previous state of the cross-coupled inverters. In practice, access NMOS transistors M_5 and M_6 have to be stronger than either bottom NMOS (M_1, M_3) or top PMOS (M_2, M_4) transistors. This is easily obtained as PMOS transistors are much weaker than NMOS when same sized. Consequently when one transistor pair (e.g. M_3 and M_4) is only slightly overriden by the write process, the opposite transistors pair (M_1 and M_2) gate voltage is also changed. This means that the M_1 and M_2transistors can be easier overriden, and so on. Thus, cross-coupled inverters magnify the writing process.

Design of Microcontroller CPU

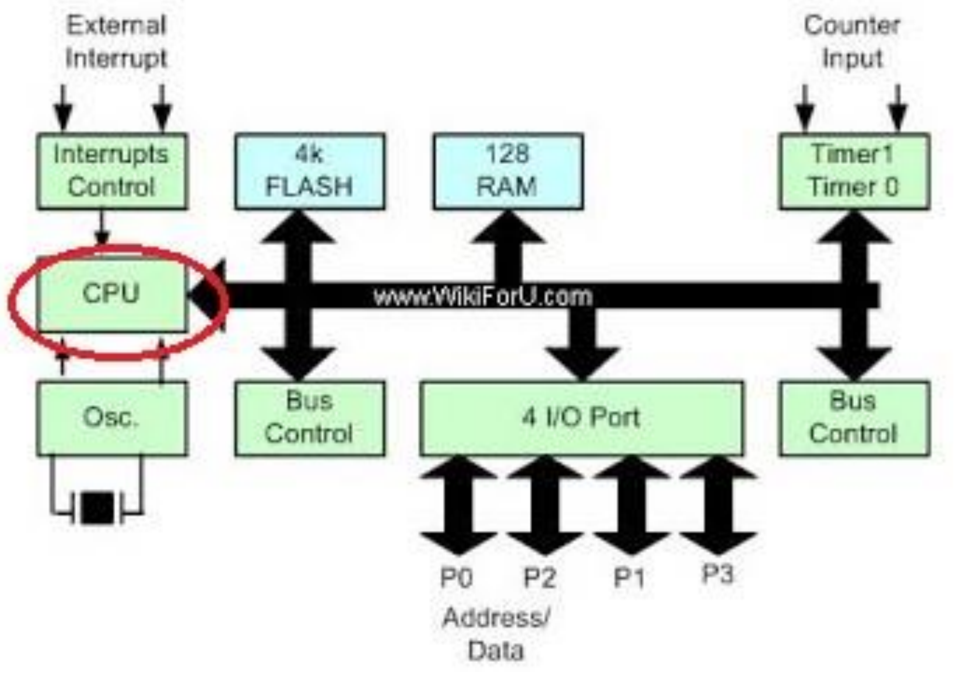


Fig 5.21 Basic Microcontroller Block Diagram

CPU is the brain of any processing device. It monitors and controls all operations that are performed in the Microcontroller.

8051 Internal Architecture:

ALU is main computational block of CPU of the Microcontroller. An arithmetic logic unit (ALU) is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integerbinary numbers. This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. An ALU is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers and controllers, FPUs, and graphics processing units (GPUs). A single CPU, FPU or GPU may contain multiple ALUs.

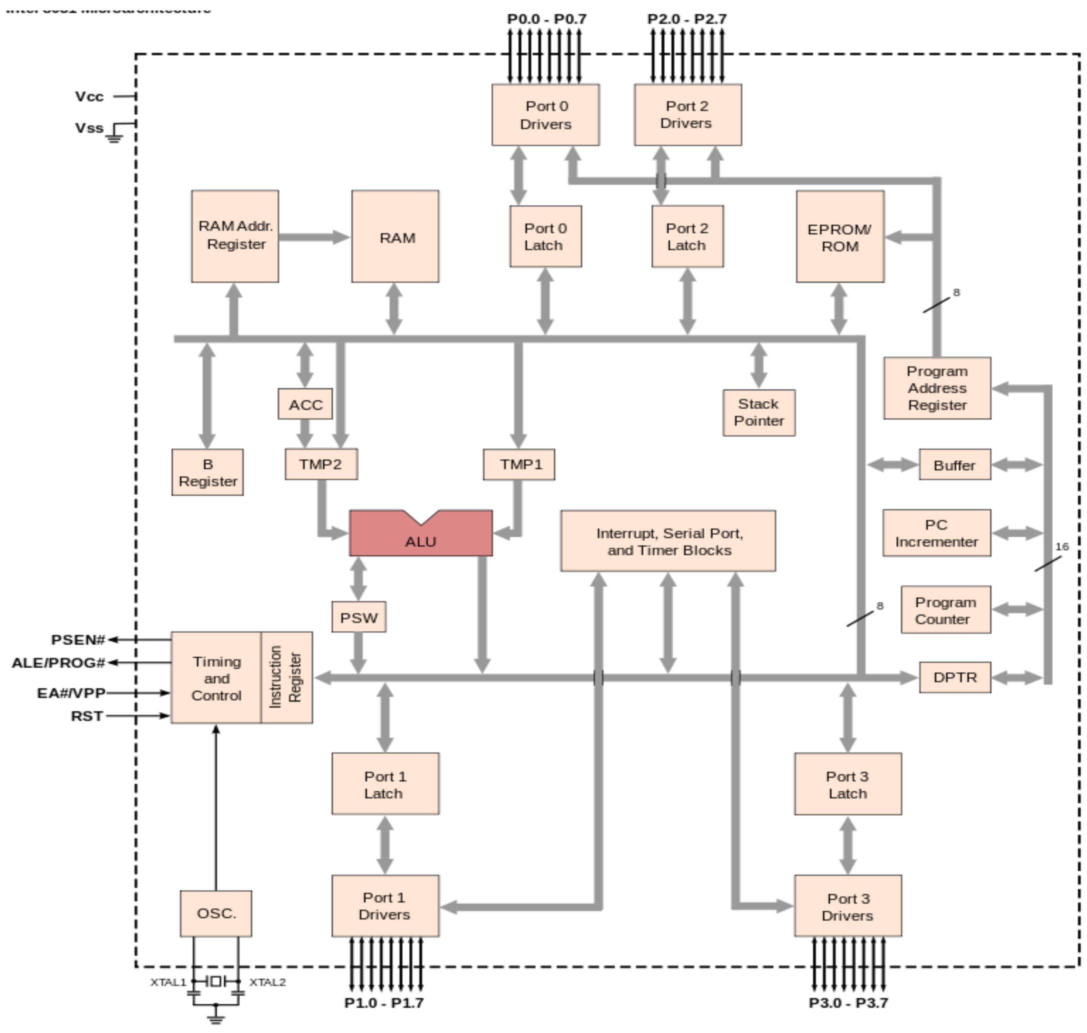


Fig 5.21 Intel 8051 Microcontroller Internal Architecture

An arithmetic-logic unit (ALU) is the part of a computer processor (CPU) that carries out arithmetic and logic operations on the operands in computer instruction words. In some processors, the ALU is divided into two units, an arithmetic unit (AU) and a logic unit (LU). Some processors contain more than one AU - for example, one for fixed-point operations and another for floating-point operations. (In personal computers floating point operations are sometimes done by a floating point unit on a separate chip called a numeric coprocessor.)

Typically, the ALU has direct input and output access to the processor controller, main memory (random access memory or RAM in a personal computer), and input/output devices. Inputs and outputs flow along an electronic path that is called a bus. The input consists of an instruction word (sometimes called a machine instruction word) that contains an operation code (sometimes called an "op code"), one or more operands, and sometimes a format code. The operation code tells the ALU what operation to perform and the operands are used in the operation. (For example, two operands might be added together or compared logically.) The format may be combined with the op code and tells, for example, whether this is a fixed-point or a floating-point instruction. The output consists of a result that is placed in a storage register and settings that indicate whether the operation was performed successfully. (If it isn't, some sort of status will be stored in a permanent place that is sometimes called the machine status word.)

In general, the ALU includes storage places for input operands, operands that are being added, the accumulated result (stored in an accumulator), and shifted results. The flow of bits and the operations performed on them in the subunits of the ALU is controlled by gated circuits. The gates in these circuits are controlled by a sequence logic unit that uses a particular algorithm or sequence for each operation code. In the arithmetic unit, multiplication and division are done by a series of adding or subtracting and shifting operations. There are several ways to represent negative numbers. In the logic unit, one of 16 possible logic operations can be performed - such as comparing two operands and identifying where bits don't match.

Verilog Code for ALU:

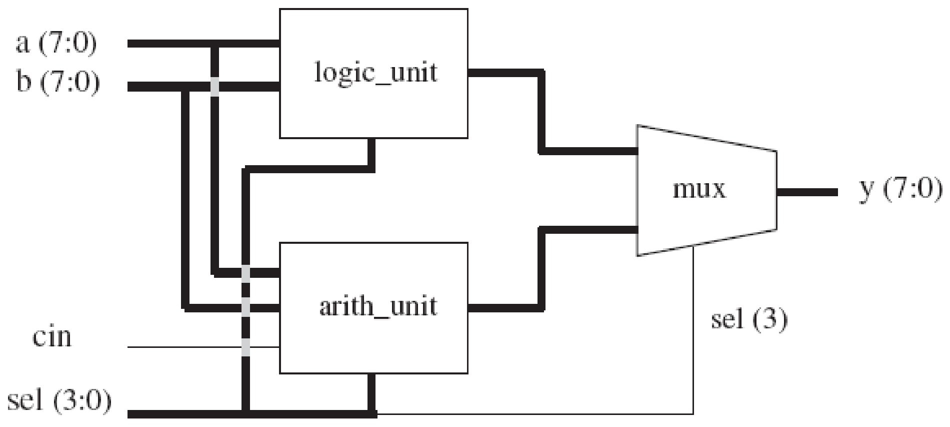


Fig 5.22 Arithmetic-Logic Unit

TRUTH TABLE:

sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

```
module alu(a, b, cin, s, y);
   input [7:0] a;
   input [7:0] b;
   input cin;
   input [3:0] s;
   output [7:0] y;
          reg [7:0]y;

          always @ (a or b or s)
          begin
          case(s)
          4'b0000: y=a;
          4'b0001: y=a+1;
          4'b0010: y=a-1;
          4'b0011: y=b;
          4'b0100: y=b+1;
          4'b0101: y=b-1;
          4'b0110: y=a+b;
          4'b0111: y=a+b+cin;
          4'b1000: y=~a;
          4'b1001: y=~b;
          4'b1010: y=a&b;
          4'b1011: y=a|b;
          4'b1100: y=~(a&b);
          4'b1101: y=~(a|b);
          4'b1110: y=a^b;
          4'b1111: y=~(a^b);
          endcase
          end
          endmodule
```