Feb.

2021

A GUIDE TO COMPUTER ARCHITECTURE

PEARL

AN RWU PROJECT

PREPARED BY

RUDRAN CHAKRABORTY

SOHAIB SAEED

KANDESH SOMVANSHI

HOCHSCHULE RAVENSBURG-WEINGARTEN

Table of Contents

Acknowledgement

We would like to express our special thanks of gratitude to our professor Dr. Markus Pfeil who gave us the golden opportunity to do this wonderful project as well as guided us in every step throughout the six months of the duration of the project.

Additionally, we would also like to thank our university Hochschule Ravensburg-Weingarten for accepting our project proposal.

List of Figures

1. Scope

An Electrical engineer and software developer's life revolves mostly around the different microcontrollers, microprocessors and at times different sensors. In most cases the jobs are done simply by writing a program in C/C++, Java, or other Higher-level languages. But for many, the actual working of the microcontroller or the microprocessor remains a mystery. In many applications, understanding the internal working of a microcontroller not only improves the performance of the device that the engineer wants to build but also solves many underlying issues that might pave the way in building more complex and efficient hardware designs in the future.

The purpose of this project is to understand the simple yet intuitive working of microcontrollers and processors. This project is aimed not only at engineering students but also hobbyists and computer enthusiasts who are eager to learn about computers and encourages them to build one on their own.

To fulfill this goal the project is divided into four parts:

- Part I: Focuses on building up the foundation by providing some basic yet important understanding of the computer architecture. Readers will learn about different components and blocks that forms the computer.

- Part II: Focuses on building a simple microcontroller in an FPGA using VHDL by using the concepts that the reader has learnt so far in Part I.

- Part III: Focuses on creating different instructions for the microcontroller as well as building a custom assembler using python to program the microcontroller.

- Part IV: Focuses on writing assembly programs using a custom assembler to perform simple arithmetic calculations such as multiplying numbers and finally controlling sensors and other hardware using our own microcontroller that we have built.

Part I

2. Computer architecture

Every computer, from simple calculators to smart watches to high-end computers to even super computers, all have a specific architecture depending on the application it performs. Computer architecture are specific rules that the computer follows to execute instructions that the user gives. Hence the internals of each computers are built in a specific way to perform specific tasks that the user requires. As a matter of fact, there is no so-called *best* computer architecture design as this depends on the user's requirements. This means that anyone can built their custom computer with their own architecture if it performs the way the user intends it to.

However, tech giants over the years have developed different computer designs and through performance tests and efficiency, came up with specific architecture types that works best in most of the applications that the computers might generally be used for.

2.1 CISC based architecture

Microcontrollers that are built under CISC or *Complex Instruction Set Computers* are usually preferred for applications where the user wants to perform complicated tasks by minimising the number of written codes needed to perform such tasks. This means that the computer must perform all the steps internally, that are needed to complete the operation. This increases power consumption as additional hardware is required to perform these complex operations which increases the number of execution cycles as well. However, with the help of complex instructions the user can save huge amounts of development time and production costs as the amount of space needed in memory is reduced.

For example, let us say we want to multiply two numbers which are stored in memory location *x* and *y* and store the result back in *z*. Using CISC based computers this operation can be reduced to a single instruction:

$$MUL\ z,\ y,\ x$$

2.2 RISC based architecture

Microcontrollers that are built under RISC or *Reduced Instruction Set Computers*, tend to have simple instruction sets. They generally have instructions to perform basic operations such as moving a byte of data from one memory location to another or performing simple arithmetic operations such as addition or subtractions. Thus, the number of different instruction that the computer understands or can execute are reduced. This reduces the hardware required to perform each instruction which reduces the power consumption as well. RISC based computers typically uses only few clock cycles to execute an instruction thus the efficiency of the computer increases drastically. However, this comes with a cost: Complex operations such as multiplying two numbers now requires a lot of codes which increases the memory consumption.

For example, to perform the same multiplication operation in RISC based architecture the following code algorithm can be followed:

```
    lda y
    subi 0
    jmz zero
loop:
    lda z
    add x
    sta z
    lda y
    subi 1
    sta y
    jmz end
    jmp loop
zero:
    ld z,  0
end:
    jmp end
```

As seen from the above example, it is clear that the instructions in RISC based architecture are simple and easy to understand and less hardware is needed to perform those operations, however, more RAM space is required.

3. Simplified computer

Each type of computer architecture has its own advantages and disadvantages. On one hand CISC based computers helps the user to perform complex operation with less memory usages but consumes more power as complex hardware are needed, on the other hand RISC based computers uses more RAM but uses simple instructions to perform operations and uses less hardware. Modern day computers try to use advantages provided by both kinds.

A computer architecture can be simplified to a very basic level composed of only registers for storing data and combinational blocks to perform arithmetic and logical operations.

Let us assume a computer consisting of 3 bytes of RAM, 2 registers and one arithmetic logic unit.

Now if the computer wants to add the number stored in location *x* with *y* and store the result in location *z*, the computer must take the following steps:

1. Take the number or data stored in location *x* and copy it to The *A* register.
2. Take the number stored in location y and copy it to The B register.
3. Add the contents of register *A* with register *B* and store the result in ALU register.

4. Finally copy the result in ALU and put it to address location z.

The computer will not be able to perform a simple command such as addition directly, rather would take multiple steps involving registers and memory transfers before the actual addition. However, these procedures get more complex as the number of registers and operations that the computer can perform increases. To deal with complex operations, every computer has a control unit. The main purpose of the control unit is to perform operations such as ADD, SUB, MOV and many more by breaking down into simple intermediate steps. These intermediate steps are called computer micro-codes and are very specific to the hardware as well as the architecture of the computer. But before going into more details into how these micro-codes work, it is first important to understand how each individual component such as RAM and registers work.

3.1 Computer register

The fundamental building blocks of every computer are storage components called *flip-flops*. A flip-flop can store 1 bit at a time. But they can be cascaded with multiple flip-flops to form what is called a register. Even though there are no limits to the size of registers, but most early computers had 8-bit registers in general and some special registers which were double the size. However, most modern computers have 32 or 64-bit registers. The main purpose of registers is to store data that the user feeds into the computer. In general, a computer programmer has access to almost every register that the computer has, but there are certain registers reserved only to the computer to perform some tasks.

Every register has at least 2 control signals. Control signals are signals which are used to control the behaviour of a component such as *Read* or *Write* functionality. When the *Read* signal is activated, the register would read the value from the source and store it. If the *Write* signal is activated, the register would write to a sink whatever value it had stored before. Thus, by simply controlling the control signals, data can be *read* or *written* whenever it is needed.

3.2 Computer memory: RAM

Another important storage element in a computer is the memory. The functionality of a memory component such as RAM is the same as that of registers. The key difference is that the registers can hold only a byte (8-bits for 8-bit registers) of data but a computer memory such as RAM can hold multiple bytes of data at a time. This number varies in every computer but most microcontroller has internal RAM up to 1 or 2 Kb (1024 to 2048 bytes). Computers that have higher RAM capacities are generally considered to run faster because they can perform operations on a large amount of data at a time.

RAM can be of two types: *static* and *dynamic*. Even though both types of RAM perform similarly, the difference lies in their hardware designs.

Static RAM types can retain their memory contents without the need of any external refresh circuitry if power is kept on. They do so by storing the value of each bit in transistors, and generally to store a single bit of data, as many as 6 to 7 transistors are required. Thus, *static*

RAMs are very expensive and are only used in microcontrollers or as an internal memory in micro-processors.

Dynamic RAM types on the other hand stores a single bit of data as a charge in a capacitor and thus, are less expensive to manufacture in large amounts. However, one drawback of dynamic RAM is that it requires a refresh circuit that will re-charge the capacitors regularly so that they can keep storing their memory contents. Thus, dynamic RAM types are mostly used as external memory devices such as external RAM for a computer.

3.2.1 Working of a RAM unit: memory address register

In the previous section we have discussed the functionality of a RAM unit. In this section we will have an in-depth discussion on the working of a RAM unit. Along with the *Read* and *Write* control signal, the RAM also has an *address* line. Depending on the size of the RAM, the address line can have 8 to 16-bit values. A RAM in general consist of multiple 8-bit registers which are connected in parallel. Thus, each of these 8-bit registers can store different data in its memory independent to each other. So, to read or write a data into a specific register in the RAM, some sort of signal is needed to activate only that register of the RAM. This is done with the help of an address line. The purpose of the address line is to point to a specific register of the RAM at a time. Thus, if a RAM has a size of 256 bytes, it means that the user can access each byte of the RAM by feeding a specific value in the *address* line.

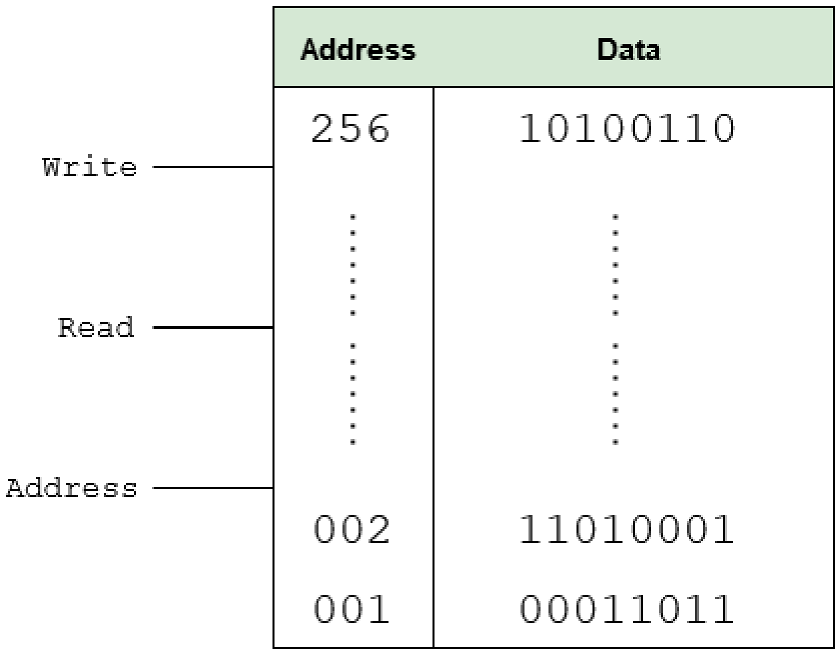


Figure 3.2.1-1 RAM register contents

The above diagram shows how each register in the RAM contains a specific data and each of this data can be accessed by providing a value in the address line corresponding to that specific register of the RAM.

3.3 Program counter

When a program is loaded for execution into the RAM of a computer, each line of the machine code is stored in each address location of the RAM as a byte of data. But during the program execution phase, the computer needs to keep track of the line of code that it is going to execute. This is achieved with the help of the program counter.

A simple program counter has three control signals called *counter enable, counter out* and *counter input*. Every time a line of code is executed, the program counter increments by 1 and points to the next line of the code that needs to be executed. The new value of the program counter is then loaded into the *address* line of the *memory address register* to fetch the next instruction from the RAM. Normally the program counter would increment its value by 1, but in some cases depending on the instruction fed by the user, the program counter can also be loaded with a different value. This is done by activating the *counter input* signal. The working of the program counter will be explained in detail in the later sections.

3.4 Instruction decoder

When an instruction is fed into a computer, the computer is required to understand what the instruction really means. This is done with the help of an instruction decoder. Every computer has a lookup tables (LUTs) that comprises of all the instruction that it supports. These LUTs could be stored in the ROM, or sometimes through external hardware or extender circuits. When the computer fetches a new instruction from RAM, it sends it first to the instruction decoder logic, where it uses the LUT to decode the meaning of the instruction. Once it is decoded, the computer can take appropriate steps to execute the instruction. Thus, a typical computer has three cycles: *Fetch* → *Decode* → *Execute*. Most modern computers try to perform a *prefetch* where the computer fetches and decodes the next instruction that must be executed even before the current instruction has been executed. This helps to reduce the overall clock cycles needed to execute each instruction which increases the overall speed of the computers dramatically.

3.5 Arithmetic logic units

Every computer needs to perform some arithmetic or logical operations with the data stored in its registers or RAM. In computing, an arithmetic logic unit is a combinational digital circuit that performs arithmetic and bitwise logical operations. Most *ALUs* have two data inputs called operands and a control signal that would indicate the type of operation that is needed to be performed by the *ALU*. In many hardware designs the *ALU* also has status or input/output flags along with the result. Flags are important to every computer architecture as they indicate the status of the previous arithmetic or logical operation.

3.6 Block diagram of a Simplified computer

So far, we have discussed about the basic modules that are required to build a computer. In this section we would introduce a simplified computer block diagram that would be able to execute simple instructions such as moving values from a memory location to registers and performing a simple addition operation.

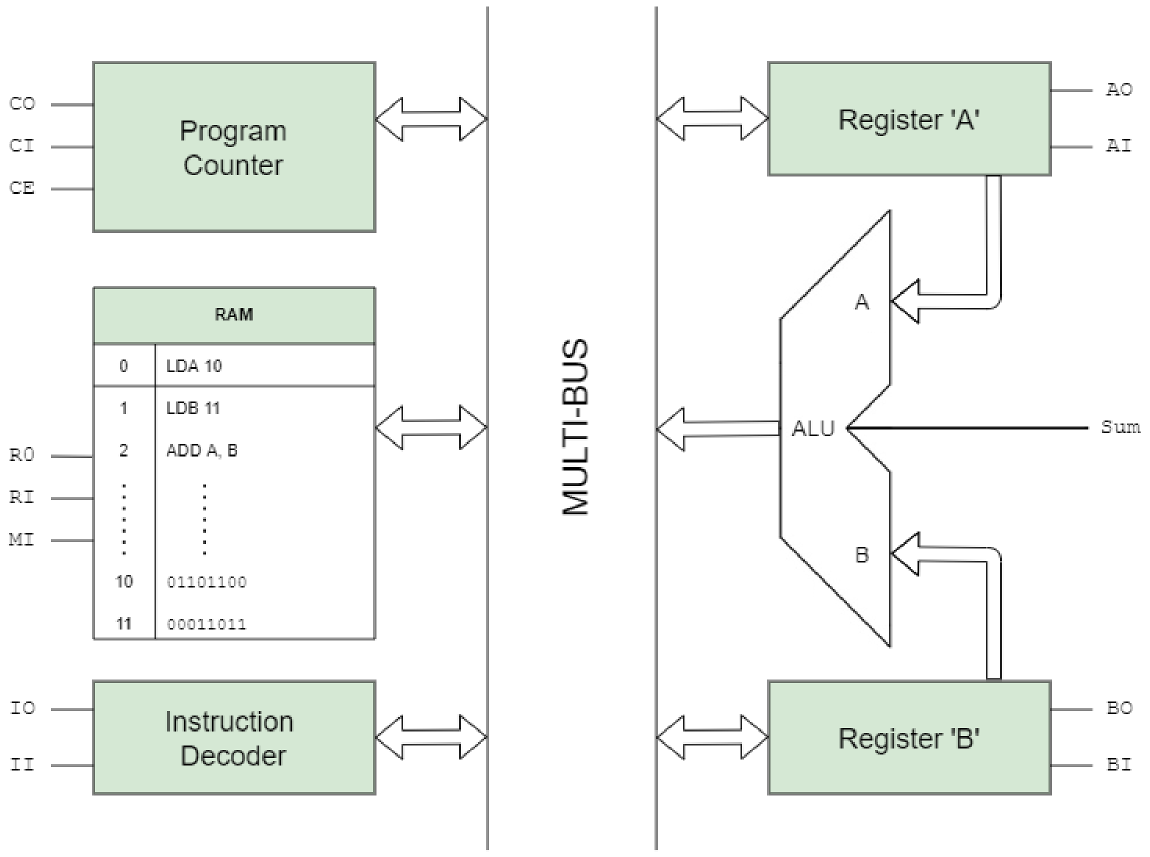


Figure 3.6-1 Simplified Computer block diagram

In the above block diagram, we have the following components:

- Program counter that keeps track of the current line of the program that is to be executed.
- RAM module to store the program.
- Instruction decoder to decode the instructions from the RAM.
- Two Registers to store some values.
- An ALU to computer arithmetic operations.

To perform the addition operation let us assume that the user wrote the following program:

$$\begin{gathered}LD\ A\ 10\\ LD\ B\ 11\\ ADD\ A,\ \ B\\ STR\ A\ 12\end{gathered}$$

- The first line of the above assembly program gets a value from memory location 10 in the RAM and loads it into the *A* register.
- The seconds line gets a value from memory location 11 in the RAM and loads it into the *B* register.
- The third line performs an addition operation between the two values stored in *A* and *B* and stores the result in the *A* register.
- The last line stores the result from *A* register back to memory location 12 in the RAM.

Let us assume that the above program would be stored in RAM starting from memory address location 0 in the form of machine code (binary). When the above program is loaded into the computer, it is loaded in the same binary form. This binary form of representation of each instruction (*Opcode*) is already provided by the computer architecture designer to the software developers. However, for the sake of this example we would assume the following binary representation of each instruction to be the following:

RAM address	*Opcode (Instruction)*	*Binary*
0000	*LD A*	*0110*
0001	*LD B*	*0111*
0010	*ADD A, B*	*1100*
0011	*STR A*	*0001*

Similarly, the binary representation of the operand would look like:

RAM address	*Operand*	*Binary*
0000	*10*	*1010*
0001	*11*	*1011*
0010	*-*	*0000*
0011	*12*	*1100*

It is to be noted that the Opcode for every instruction that the computer supports should have a unique binary representation which the computer architecture designer has the liberty to choose. From the table, the ADD A, B instruction takes no operand thus, the binary representation of operand is set to 0000. Thus, if we combine both the Opcodes and the Operands, the final binary representation is called the Machine code.

RAM address	*Machine Code*
0000	*0110 1010*
0001	*0111 1011*
0010	*1100 0000*
0011	*0001 1100*

As discussed earlier, every computer must go through with the fetch cycle regardless of the instruction. During this cycle, the computer fetches the instruction which is in RAM and sends it to the instruction decoder module to decode the instruction. To do so, the computer needs to set the correct control signals of the modules to transfer the data from one module to another. These sets of signals are called *Microcode* and are normally stored in the control unit of the computer as a sequence of words or bytes. The arrangements of the bits of the control signal depends on the computer architecture designer. The following table shows the control word sequence of our *Simplified computer*.

CW	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
	1 : Active **0 : Inactive**												

Now let us look at how the computer would execute the above program.

3.6.1 Fetch cycle

The fetch cycle comprises of three different states. In most modern computers, these three cycles are sometimes merged to one or two clock cycles, but for easier understanding, we would split them into individual clock cycles.

3.6.1.1 Address fetch state

The first state of the fetch cycle is called the *Address fetch state* because the address stored in the program counter is transferred to the memory address register MAR. During this state, the *CO* and *MI* control signals are active, and all the other control signals are inactive. The *CO* signal outputs the program counter value in the *Address-Data-Bus*. When the *MI* signal is activated, the memory address register would read whatever value is on the bus at that moment and in this case, it loads the program counter value. The *Control word* for the address fetch state is shown below.

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
1.	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**

3.6.1.2 Load instruction state

The second state of the fetch cycle is called the *Load instruction state*. Once the *memory address register* has loaded the address, it automatically retrieves the content of that address location from the RAM. In the Load instruction state, this RAM content is loaded to the instruction decoder register where the content is split into two nibbles. The upper nibble which is the Opcode goes to the control unit where it is decoded, and the lower nibble is stored to be written into the *Address-Data-Bus* if needed. During this state, the *RO* and *II* control signals are active. The *RO* signal outputs the content of the RAM to the bus and the *II* signal loads this content into its register. The *Control word* for the Load instruction state is shown below.

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
2.	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0

3.6.1.3 Increment counter state

The last state of the fetch cycle is called the *Increment counter state*. When the last instruction is already loaded to the instruction decoder register where it is ready to be executed, it is required to increment the program counter by 1 so that it would point to the next address location of the RAM. During this state, only *CE* is active which increments the program counter. The control word for the load instruction state is shown below.

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
3.	0	0	0	0	0	0	0	0	0	0	**1**	0	0

The Control word sequence for the complete fetch cycle is shown below.

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
1.	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**
2.	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0
3.	0	0	0	0	0	0	0	0	0	0	**1**	0	0

For every machine instruction, the fetch cycle is similar and at the end of the fetch cycle, the computer has already decoded the current instruction and would initiate the *Execution cycle*.

3.6.2 Execution cycle

The next states after the *fetch cycle* is the execution cycle. The control words of an execution cycle depend on the instruction that is being executed. For example, the *LD A* instruction would load a value from the memory location to the *A* register and on the other hand, an *ADD* instruction would add the contents of the *B* register with to the *A* register. Another key difference between the fetch and the execution cycle is the number of clock cycles required to complete the process. The fetch cycle regardless of the type of instruction, would always take three clock cycles to execute completer the fetch cycle process, and on the other hand, the execution cycle might require more than 1 clock cycle depending on the complexity of the instruction.

In the following section we would discuss the *control routines* of the instructions in the above assembly program.

3.6.2.1 *LD A:* instruction routine

The *LD A* instruction as discussed earlier, loads a value from the memory location, which is specified by the operand, to the *A* register. At the end of the fetch cycle, the operand value, in this case the address location, is stored in the lower nibble of the instruction register. This value must be loaded into the *MAR* to retrieve the contents of the RAM in that location. The control word for this step is as as follows:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
4.	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0

The control word activates the *IO* and *MI* signal. The *IO* signal puts the content of the lower nibble of the instruction decoder register into the Bus and the *MI* register reads the value from the Bus. This, process of transferring data or signals from one register or module to another is commonly called register transfer level (RTL).

The next step of the control routine is to take the content of the RAM and transfer it to the *A* register. The control word for this step is as follows:

Clock cycle	*ALU*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
5.	0	0	0	**1**	0	0	0	0	**1**	0	0	0	0

At the end of this cycle, the content of the RAM is now transferred to the *A* register. The following table shows the complete micro-code for the *LD A* instruction:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
Fetch cycle													
1.	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**
2.	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0
3.	0	0	0	0	0	0	0	0	0	0	**1**	0	0
Execution cycle													
4.	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0
5.	0	0	0	**1**	0	0	0	0	**1**	0	0	0	0

From the above table we can see that the *LD A* instruction would require 5 clock cycles to execute. This is commonly called the execution time of an instruction and is measured in *clocks per instruction* or *cycles per instruction (cpi)*. Thus, the execution time of instructions determines the CPU performance.

3.6.2.2 *LD B:* instruction routine

The *LD B* instruction routine is similar to the *LD A* instruction except for clock cycle 5 were the content of the RAM is transferred to the *B* register instead of the *A* register.
The following table shows the complete micro-code for the *LD B* instruction:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
Fetch cycle													
1.	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**
2.	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0
3.	0	0	0	0	0	0	0	0	0	0	**1**	0	0
Execution cycle													
4.	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0
5.	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0

3.6.2.3 *ADD A, B:* instruction routine

The *ADD A, B* instruction adds the data stored in *B* register to the *A* register and stores the result back to the *A* register. This arithmetic addition is performed in the *ALU* register which stores the result temporarily which is then moved back to the *A* register. The control word for the *ADD A, B* instruction is given below:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
4.	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0

The control word activates the *SUM* and *AI* signal. The *SUM* signal adds the two numbers in the registers and outputs the result in the bus.

The following table shows the complete micro-code for the *ADD A, B* instruction:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
Fetch cycle													
1.	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**
2.	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0
3.	0	0	0	0	0	0	0	0	0	0	**1**	0	0
Execution cycle													
4.	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0

3.6.2.4 STR *A:* instruction routine

The *STR A* instruction is similar to the *LD A* instruction with the exception that the *STR A* instruction stores the value in *A* register back to the memory location specified by the operand. The address location is stored in the lower nibble of the instruction decoder register and is loaded to the *MAR* register which then points to the desired memory location. The corresponding micro-code is given below:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
4.	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0

Once the *MAR* points to the correct address in the *RAM*, the value stored in *A* register is loaded to the *RAM*.

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
5.	0	0	0	0	**1**	0	0	0	0	**1**	0	0	0

The complete micro-code for *STR A* instruction is shown below:

Clock cycle	*SUM*	*IO*	*II*	*AI*	*AO*	*BI*	*BO*	*MI*	*RO*	*RI*	*CE*	*CI*	*CO*
Fetch cycle													
1.	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**
2.	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0
3.	0	0	0	0	0	0	0	0	0	0	**1**	0	0
Execution cycle													
4.	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0
5.	0	0	0	0	**1**	0	0	0	0	**1**	0	0	0

3.7 Program execution time

It is important to measure the performance of a computer such as the response time to an input. The performance of a computer depends on many factors - one such factor is the execution time of the program. The worst-case execution time (WCET) of a program is the maximum length of time the task could take to execute a program. This can be calculated by adding the total number of clock cycles needed to execute an instruction. Since the duration of the fetch cycle is the same for all instructions, the total execution time is proportional to the number of instructions used in the program and the duration of the execution cycle. The following formula gives us the execution time as a function of clock cycles.

$$X_T = n \cdot F_T + \sum_{i=1}^{n} \delta_i$$

Where, $n = number\ of\ instructions$

$F_T = fetch\ cycle\ duration\ of\ one\ instruction$

$\delta = execution\ cycle\ duration$

For example, the above program has a total of 4 instructions. The execution time of the program is given by:

$$X_T = 4 \cdot 3 + \sum \delta\ (ld\ a; ld\ b; add\ a, b; str\ a)$$

$$X_T = 12 + 2 + 2 + 1 + 2$$

$$X_T = 19$$

Part II

4. Microcontroller design in FPGA using VHDL

In *Part I* we have discussed some of the main components of a simplified computer with its functionality and execution processes. In *Part II* we would like to design a microcontroller with a customised RISC architecture that would be able to control sensors and other hardware. The design would be implemented and tested on both *Spartan 3E Development kit* and *Altera Cyclone 10LP based cyc1000* FPGA devices. The design UCF/pin-planner files would be included for both types of devices.

4.1 Design block diagram

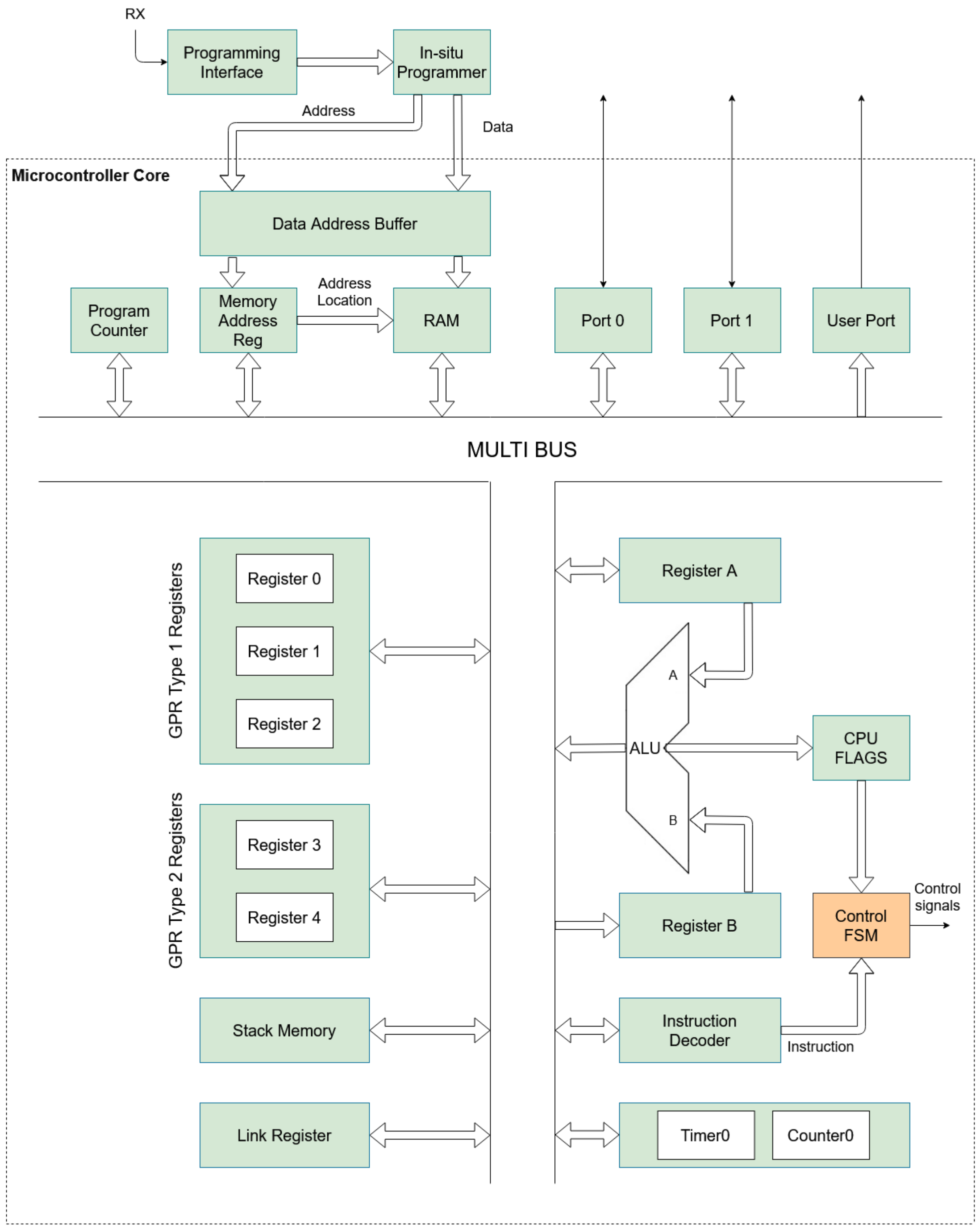


Figure 4.1-1 pearl computer architecture

The microcontroller features the following components:

1. Three General purpose 8-bit registers of type1.
2. Two General purpose 8-bit registers of type2.
3. Two General purpose I/O bit addressable ports.
4. One Output port.
5. 256 Bytes of RAM.
6. 32 bytes of stack memory.
7. Constant generator module.
8. 8-bit Program counter.
9. 8-bit Link register.
10. 8-bit ALU with status flag register.
11. 8-bit Accumulator with programmable carry bit.
12. 8-bit B register directly connected to ALU.
13. One 16-bit Timer/Counter with 8 selectable prescale.
14. Serial UART programming interface.
15. ISP with data address buffer for fast and easier programming.
16. 16-bit Data-address Multiplexed bus.

4.2 Block descriptions

4.2.1 GPR type1

The *GPR type1* are the main 8-bit registers of the microcontroller. In total the microcontroller consists of 3 type1 registers namely *R0*, *R1* and *R2*. The type1 registers can be used as two separate 4-bit registers where the lower nibble of the register is accessed by *RxA* while the upper nibble can be accessed by *RxB*. The type1 registers can not only perform all the arithmetic and logic operations that the microcontroller features but can also be used as address pointers to a memory location in the RAM. The block diagram of a type1 register is shown below:

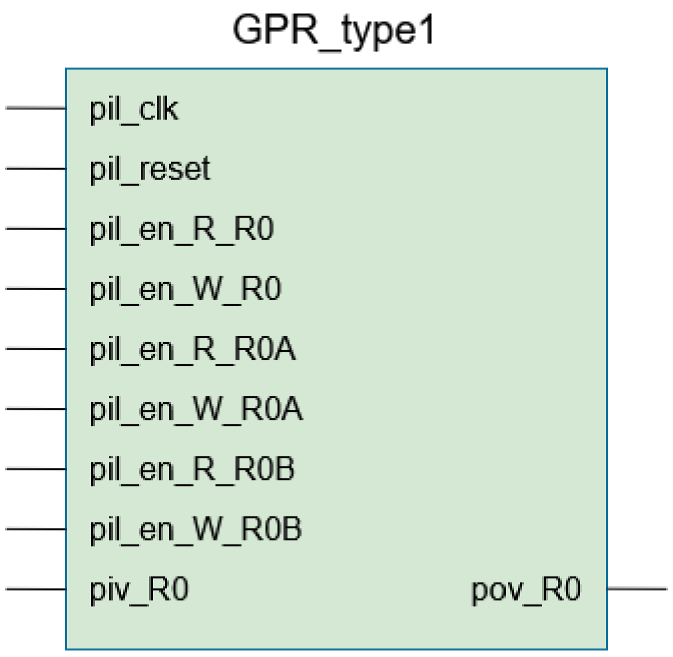


Figure 4.2.1-1 GPR Type1 block diagram

The type1 registers has 6 control signals for reading and writing data to the bus. To read data from the bus, a logic ‘1’ signal is provided to the specific *pil_en_R_Rx* signal which then stores the contents of the bus into the register and while writing, a logic ‘1’ signal to the specific *pil_en_W_Rx* signal is provided. The contents stored in the register can be cleared with the asynchronous reset button.

4.2.2 GPR type2

GPR type2 registers are strictly 8-bit registers, meaning that they cannot be used as two separate 4-bit registers like the type1 registers. The microcontroller consists of 2 type2 registers namely *R3* and *DM/R4*. These registers are used internally by the microcontroller for special arithmetic operations such as Multiplications and Divisions, where the *DM* (Division-Multiplication) register is used to store the result while *R3* stores one of the operands during the multiplication operation. Other than performing special operations, the *R3* register can also be used to perform other arithmetic and logical operations. The block diagram of GPR type2 register is shown below:

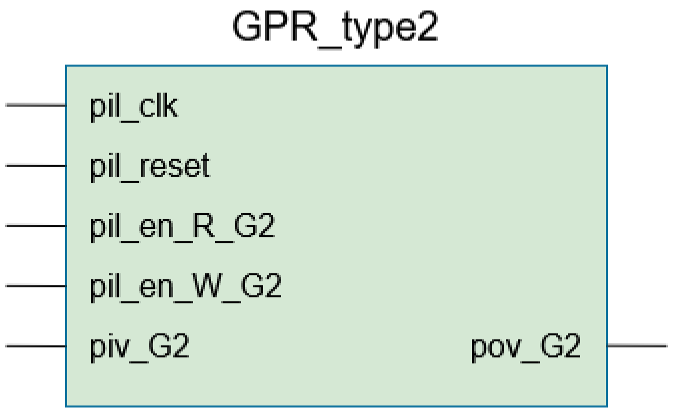


Figure 4.2.2-1 GPR Type2 block diagram

The type2 registers have 2 control signals for reading and writing to the bus, respectively. Like the type1 register, the contents of the type2 registers can be cleared with the asynchronous reset button.

4.2.3 General purpose user port

The microcontroller features of two General purpose input and output ports. Each *GPIO* port consists of 8 bi-directional pins that can be configured both as inputs and outputs and are bit addressable. The block diagram of the user port is shown below:

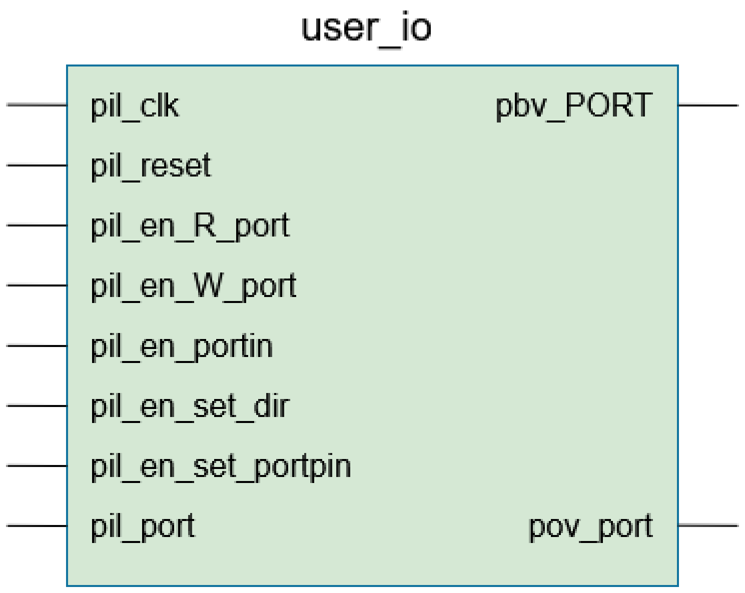


Figure 4.2.3-1 General purpose user port block diagram

The user port has 5 control signals. Upon reset both the user ports are configured as outputs and default to low. When the *pil_en_set_dir* control signal is activated, each of the port pins can be configured as inputs and outputs depending on the value read from the bus.

4.2.4 Output port

Along with the two general purpose input and output ports, the microcontroller also has one *Output port*. Unlike the user ports, the output port pins cannot be configured as inputs, and is mainly used to control the on-board LEDs for debugging purposes of a program. The block diagram of the output port is shown below:

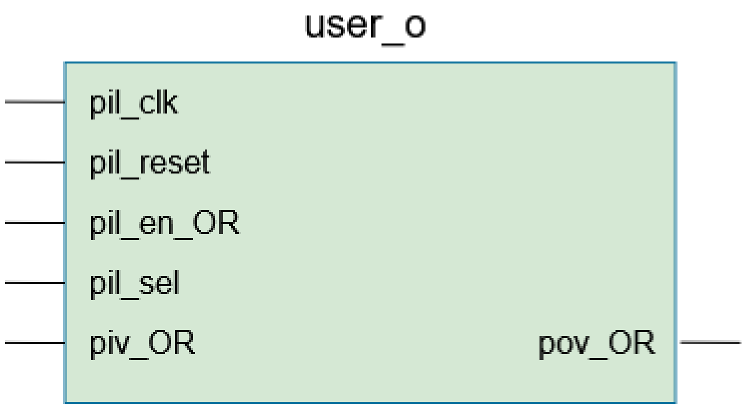


Figure 4.2.4-1 Output port block diagram

4.2.5 MAR and RAM unit

The *RAM* and *MAR* module together form the memory unit of the microcontroller. The memory address register can address up to 256 memory cells inside the RAM. Each memory cell consists of 16-bit wide words of which the most significant byte represents the opcode while the least significant byte represents the optional operand. The block diagram of the memory unit is shown below:

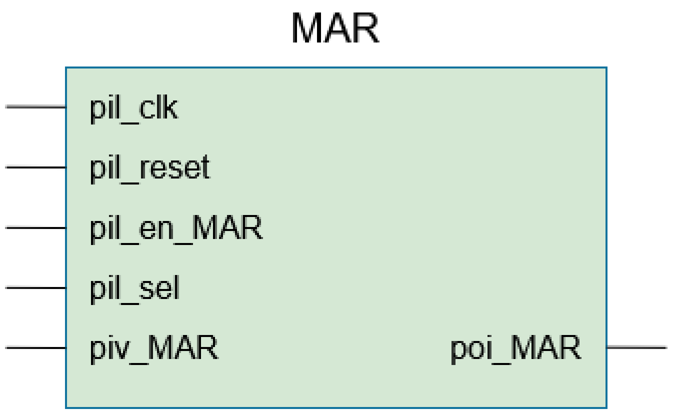


Figure 4.2.5-1 MAR block diagram

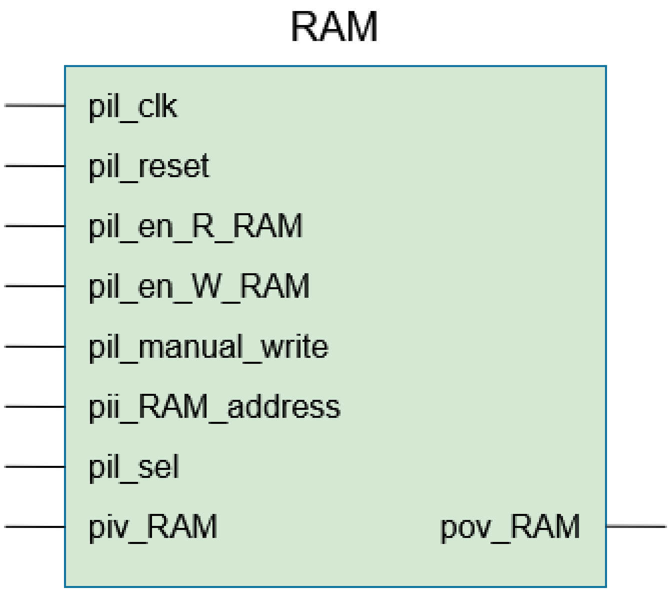


Figure 4.2.5-2 RAM unit block diagram

During the programming mode, the user must activate the *pil_sel* signal with an external button which then configures the memory unit to load the machine code into the *RAM*. During this phase, the input line *piv_MAR* as well as the *piv_RAM* is connected to the address and data output of the *Data-address-buffer,* respectively. Once the program is uploaded in the RAM, the user must deactivate the pil_sel signal which then connects the piv_MAR and piv_RAM inputs to the microcontroller's bus.

During the program execution phase, when the *pil_en_MAR* signal is activated, the memory address register reads the contents on the bus which is then sent to the *RAM* module which then retrieves the data stored in the corresponding memory location. When *pil_en_R_RAM* signal is activated, the RAM module reads the content of the bus and stores it in the memory location pointed by the MAR. If the *pil_en_W_RAM* signal is activated, the RAM module writes the contents of the memory location into the bus.

4.2.6 Stack memory unit

Along with 256x16 words of Program memory, the microcontroller also has 32 bytes of *LIFO Stack memory.* The stack memory is used during the subroutine call to store the subroutine return address.

To push data into the stack, *pil_en_SM_R* signal must be activated which then reads and stores the content of the bus. Once a value is pushed into the stack, the *stack pointer* is incremented by 1 which then points to the next available memory address in the stack. During stack overflow, the stack pointer resets to 0 and the data in the stack is overwritten. To reset the stack pointer, the asynchronous reset must be activated, however, the contents of the stack memory are retained until overwritten. When *pil_en_SM_W* signal is activated, the content of the memory location which the current value of the stack pointer points to is written to the bus and the stack pointer is decremented by 1.

The block diagram of the Stack memory unit is shown below:

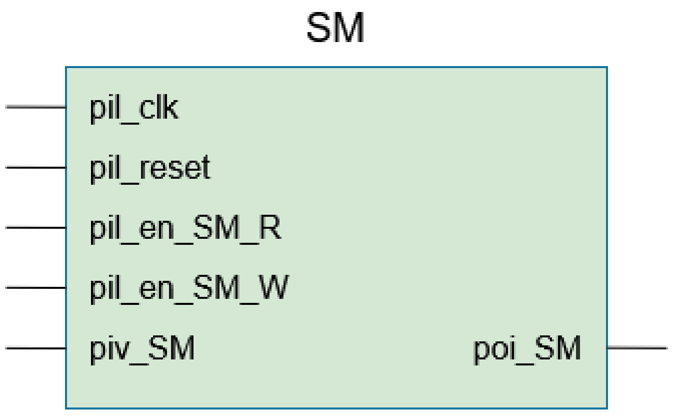


Figure 4.2.6-1 Stack memory block diagram

4.2.7 Constant generator module

The constant generator module is used to store commonly used constants such as 0, 1 and other powers of 2. These are used by the microcontroller to perform arithmetic and logical operations along with bit-wise operations involving the user I/O ports. Each of the constants can be accessed by activating the corresponding control signals. The following table shows the relation between the constants along with the control signals:

Constant	**CGB2**	**CGB1**	**CGB0**	**CG1**	**CG0**
0	0	0	0	0	**1**
1	0	0	0	**1**	0
2	0	0	**1**	0	0
4	0	**1**	0	0	0
8	0	**1**	**1**	0	0
16	**1**	0	0	0	0
32	**1**	0	**1**	0	0
64	**1**	**1**	0	0	0
128	**1**	**1**	**1**	0	0

The block diagram of the constant generator is shown below:

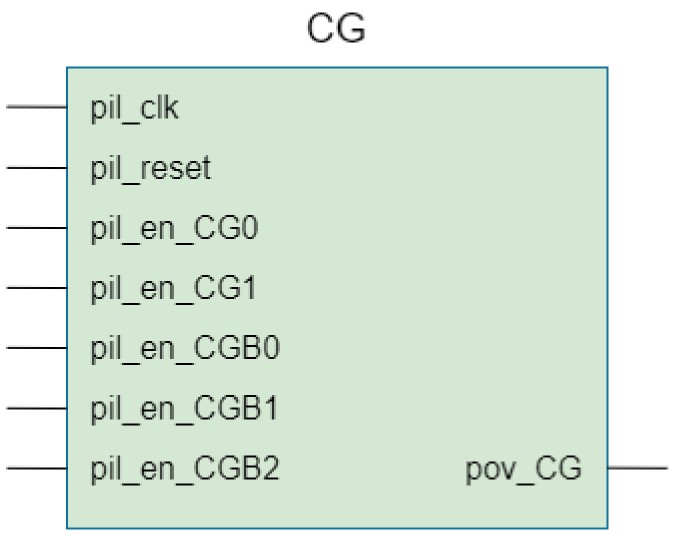


Figure 4.2.7-1 Constant generator block diagram

4.2.8 Program counter

The microcontroller has an 8-bit program counter which keeps track of the next program line to be executed. After the end of every fetch cycle, the program counter is incremented by 1 which points to the next line of the program. The block diagram of the program counter is shown below:

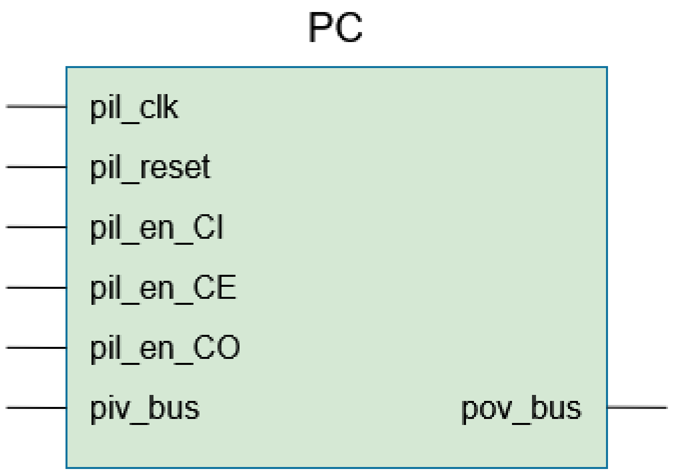


Figure 4.2.8-1 Program counter block diagram

To increment the program counter, the control signal *pil_en_CE* is activated which increments it by 1. During branching or returning from subroutine, *pil_en_CI* signal is activated which loads the program counter with the branching address from the bus.

4.2.9 Link register

The program link register is an 8-bit special purpose register whose main purpose is to hold a particular address. This is useful when the program wants to jump from a subroutine without changing the return address stored in the stack. The address stored in the link register can be moved to the program counter thereby pointing to a new memory location. Like R3 and DM registers, the block architecture of the link register also falls under the same architecture of the GPR type2 registers.

4.2.10 Arithmetic logic unit

The microcontroller features an 8-bit *ALU* module which can perform bitwise arithmetic operations such as addition, subtraction as well as various logical operations. The contents of the *A* and *B* register are fed into the ALU which then performs operations depending on the control signals which are activated. The block diagram of the ALU is shown below:

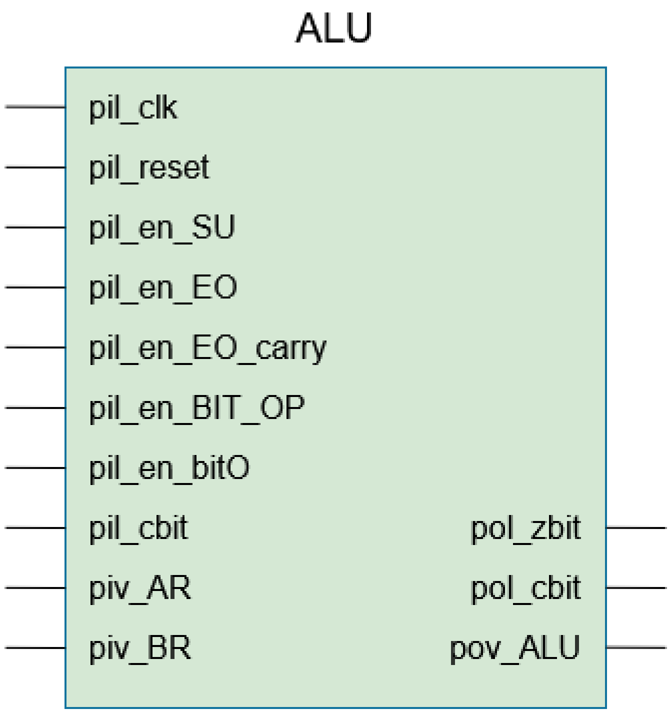


Figure 4.2.10-1 Arithmetic logic unit block diagram

4.2.11 CPU status flag register

The status flag register indicates the state of the microcontroller after the execution of an operation. The status flag can raise 4 different flags namely *carry, zero, timer flag and the counter overflow flag* depending on the last performed operation. The block diagram of the status flag register is shown below:

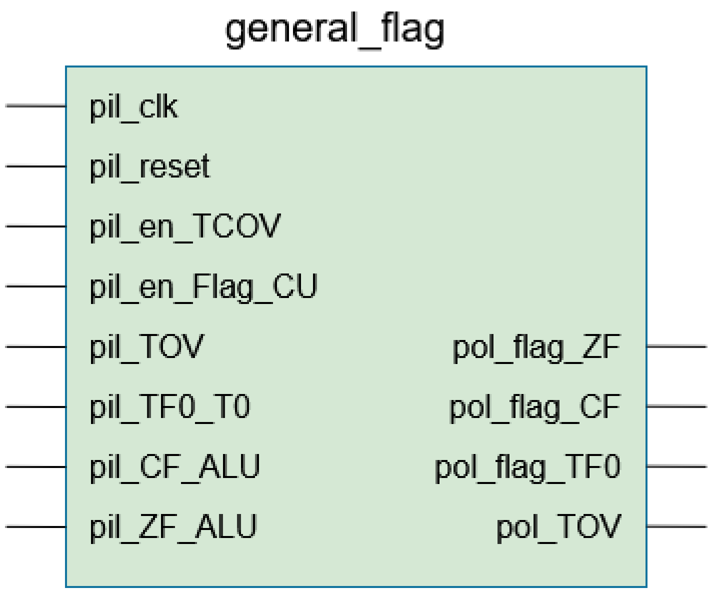


Figure 4.2.11-1 CPU status flag register block diagram

4.2.12 Accumulator register

The accumulator is the main register of the microcontroller and is involved in every arithmetic or logical operation performed by the microcontroller and stores the intermediate result of an operation. The accumulator register also has an additional carry-bit (not to be confused with the carry flag), which is used to perform operations involving accumulator shifts and can be set or cleared by the user. The block diagram of the accumulator register is shown below:

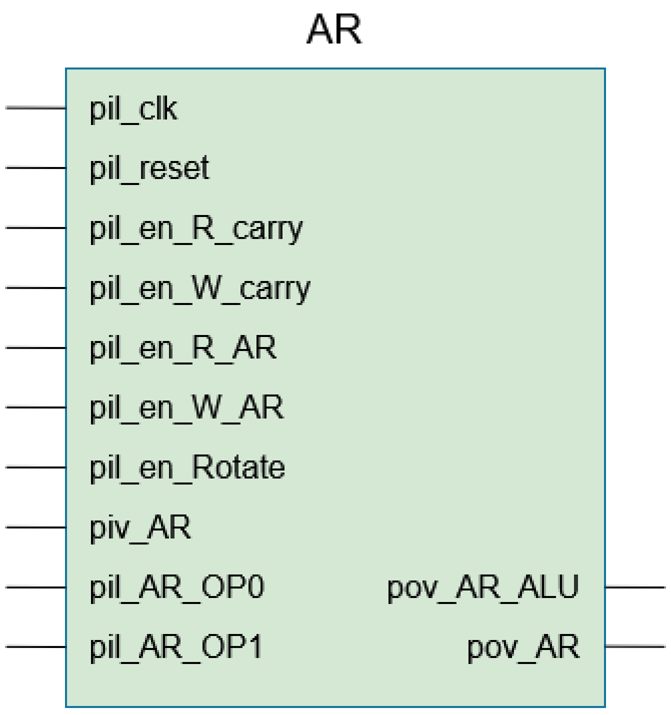


Figure 4.2.12-1 Accumulator register block diagram

4.2.13 B register

The B register is another special register which is directly connected to the ALU to store one of the operands during an operation. The microcontroller uses the B register internally and is not accessible by the user. The block diagram of the B register is shown below:

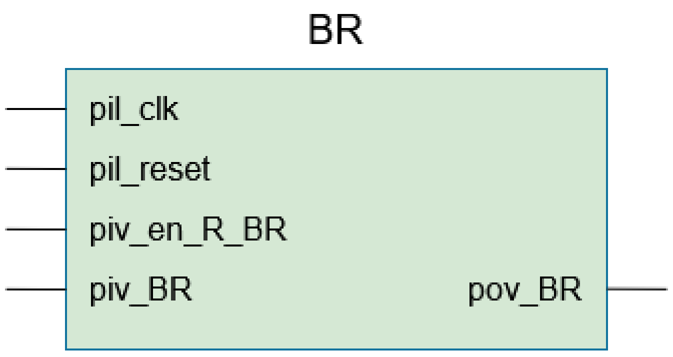


Figure 4.2.13-1 B register block diagram

4.2.14 Timer-counter register

The microcontroller has one 16-bit timer register and can be used as regular timer or as event counter. Since the microcontroller can handle 8 bits at a given moment, the 16-bit wide timer register is split into two 8-bit registers called *THx* which holds the upper byte and *TLx* which holds the lower byte. The block diagram of the timer register is shown below:

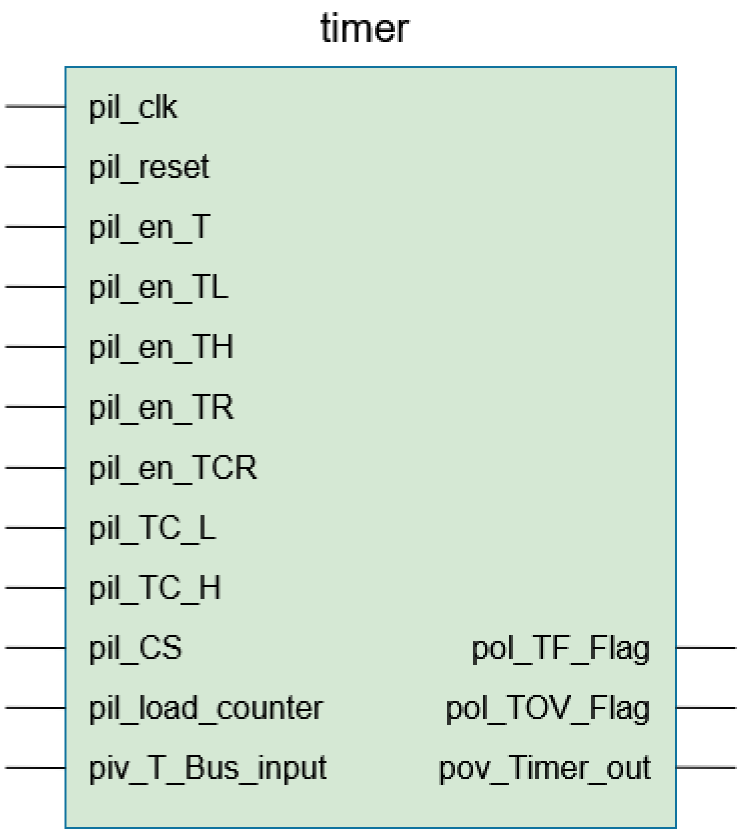


Figure 4.2.14-1 CPU Timer register block diagram

The Timer module also allows clock dividing by selecting between 8 prescale settings. This is done by activating the *pil_CS* control signal which then reads the three least significant bits from the bus and stores into its internal register *sv_CS_storage*. The following table shows the prescale according to the value stored in sv_CS_storage.

sv_CS_storage	**Prescale**
0	1
1	64
2	128
3	512
4	1024
5	2048
6	4096
7	8192

4.2.14.1 Timer mode

The Timer-counter module can be used as a timer to create required delays. This is done by loading a value between 0 – 65535 which is given by the formula shown below, into the TH0 and TL0 register by activating the *pil_en_TH* and *pil_en_TL* control signals, respectively. To start the timer, control signal *pil_en_TR* must be activated which enables counter *si_TC*. Once the si_TC counter value matches with the combined value stored in TH0 and TL0 registers, the *TF* flag is raised which resets the counter to 0.

$$CR = \frac{sys_{clk}}{prescale * f} - 1$$

Where $sys_{clk} = base\ clock\ of\ the\ FPGA.$

$f = desired\ frequency.$

4.2.14.2 Auto reload counter mode

The Timer-counter module can also be used as an event counter. This is done by activating the control signal *pil_en_TCR* which starts the counter with the initial value stored in TH0 and TL0 registers. Once the counter reaches the maximum value of 65535, the *TOV* flag is raised and the counter is reloaded with the initial value stored in TH0 and TL0. When using the Timer-counter register in counter mode, the value stored in the counter can be read by activating control signals *pil_TC_L (for lower byte)* and *pil_TC_H (for upper byte)*.

4.2.15 Serial UART programmer Interface

The microcontroller is equipped with an UART serial interface at 9600 baud-rate to upload the. hex programming file from the host computer into the microcontroller. The block diagram of the serial interface is shown below:

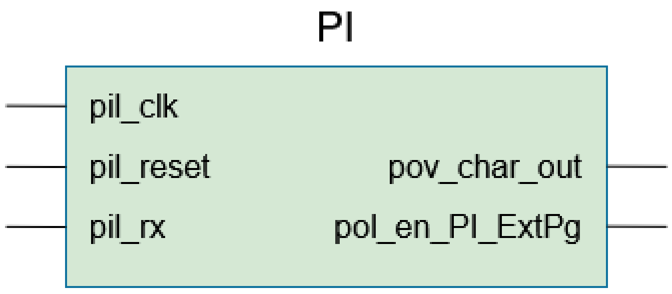


Figure 4.2.15-1 Serial programmer interface

The programmer interface module is a 5 state FSM. The st*_pr_state* would remain in *idle_st* until it detects a start condition, i.e., *pil_rx* goes low for one baud period and would go to the data receive state. In this state the microcontroller would receive 8 data bits after which it goes to the *send_byte_st*. In the send_byte_st, *pol_en_PI_ExtPrg* signal goes high which enables the ISP and sends the received byte of data.

4.2.16 In-situ Programmer and .hex file

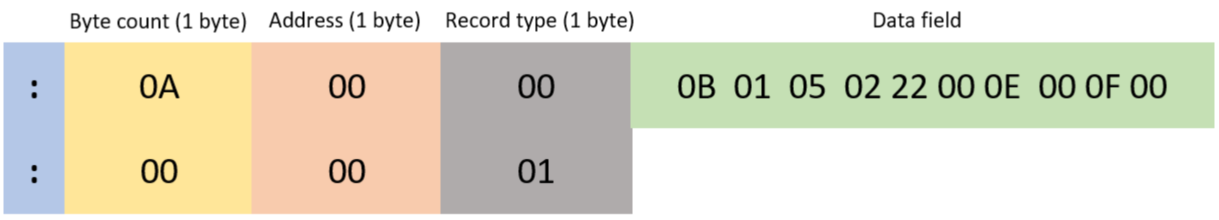
When the microcontroller is set to programming mode, the serial interface module receives a stream of data bytes (.hex file) from the host computer and sends it to the *In-situ Programmer (ISP)*. This stream of data bytes contains the machine codes in hex and the information regarding where to place these machine codes into the RAM.

A hex file is a file format that conveys binary information in ASCII text form. Each line of a hex file represents, memory address or other values depending on their position and type in the line. Each text line in a file contains hexadecimal numbers and is called a *record*.

A record consists of 5 parts and appears in order from left to right:

1. *Start code*: A start code represents the start of a new line. It takes up one character and is represented with a colon ':'.
2. *Byte count*: A byte count consists of two hexadecimal character i.e., 1 byte of length. It indicates the number of bytes in the data field. The data field contains the machine code of the program. The maximum byte count is 16 (0x10) per line.
3. *Address*: The address field is two hexadecimal character wide and represents the start of the memory address location. The physical address of the following data bates are calculated by adding 1 for every 2nd data byte to the previously established base address.
4. *Record type*: The record type consists of two hexadecimal digits and can be either 00 or 01. A record type of 00 represents that the line contains the data field and base address locations. A record type of 01 represents the end of file (EOF) and anything that is present after the record type is ignored by the *ISP*.
5. *Data*: The Data field contains the machine code representation of the program. Every two consecutive data bytes represents the 2-byte machine code where the first byte contains the opcode of the instruction and the second byte contains the optional operand.

The following diagram shows the format of the hex file.



	Byte count (1 byte)	Address (1 byte)	Record type (1 byte)	Data field
:	0A	00	00	0B 01 05 02 22 00 0E 00 0F 00
:	00	00	01	

Figure 4.2.16-1 Hex file format example

The purpose of the ISP is to decode this information and send the correct data to the *Data Address Buffer* which then programs the RAM. When the *pil_sel* signal is activated, the microcontroller enters the programming mode, and the ISP module is ready to receive the hex formatted data stream from the Serial interface module. The ISP consists of 4 states. In the *byte_count_st*, the ISP decodes the number of data byte that the line has and stores it in the *si_data_length*. The next two states are the *address_st* and the *record_st* which stores the base address of the line and the file type, respectively. The final state is the *data_st* where the module receives the data bytes and forms a 16-bit words with two consecutive bytes of data which are sent to the Data address buffer module. The block diagram of the ISP module is shown below.

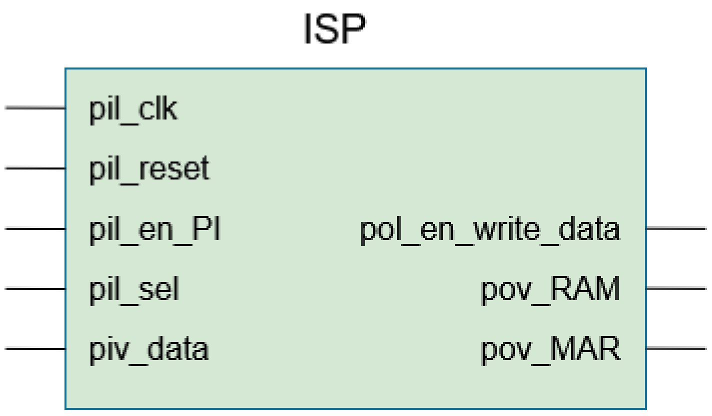


Figure 4.2.16-2 In-situ programmer

4.2.17 Multiplexed Bus module

The microcontroller features a 16-bit Bus module. The bus module has 16 inputs which are coming from different components of the microcontroller along with the control word input which is coming from the control FSM. The bus module acts as a buffer by outputting one of the inputs depending on the control signal activated. The block diagram of the bus module is shown below.

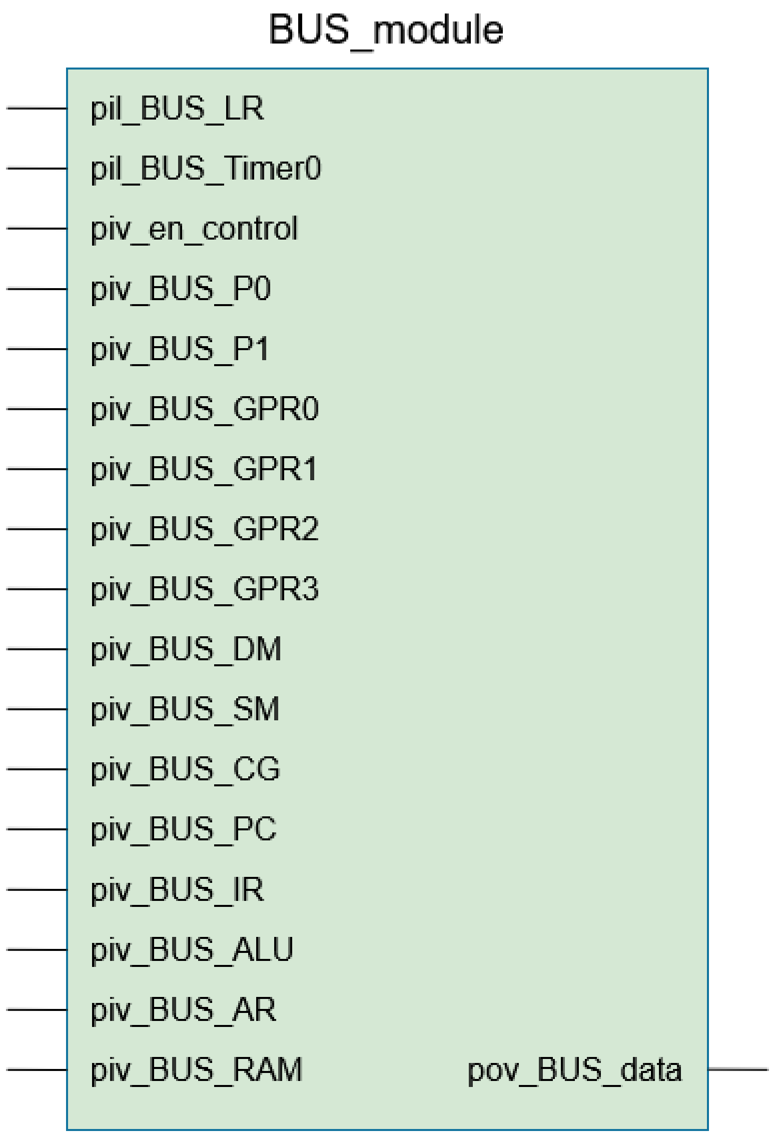


Figure 4.2.17-1 Bus module block diagram

4.2.18 Control FSM

The *Control FSM* is the heart of the microcontroller and governs the data transfer sequence from one module to another. The control FSM has a 75-bit bus which connects the control signals of the rest of computer. This control word determines how the registers would react to the next clock edge. For instance, when *pil_en_W_R0* and *pil_en_R_R1* is activated, the data stored in register0 would be transferred to register1 in the next clock edge.

The control FSM has 35 states out of which 5 states are used during the fetch cycle starting from *CU_fetch0a_st* till *CU_fetch2_st* and the rest starting from *CU_0a_*st till *CU_idle2_*st are used during the execution cycle. During the fetch cycle the control FSM activates the control signals according to the fetch cycle table in section 3.6.1.3. Each of the steps requires two clock cycles in compliance with setup time for the register transfer except for the 3rd step which is done in one clock cycle. Once the fetch cycle is completed, the control FSM receives the opcode from the instruction decoder register. The opcode is then decoded during the execution cycle and the corresponding control signals are activated at every step.

The block diagram of the control FSM is shown below.

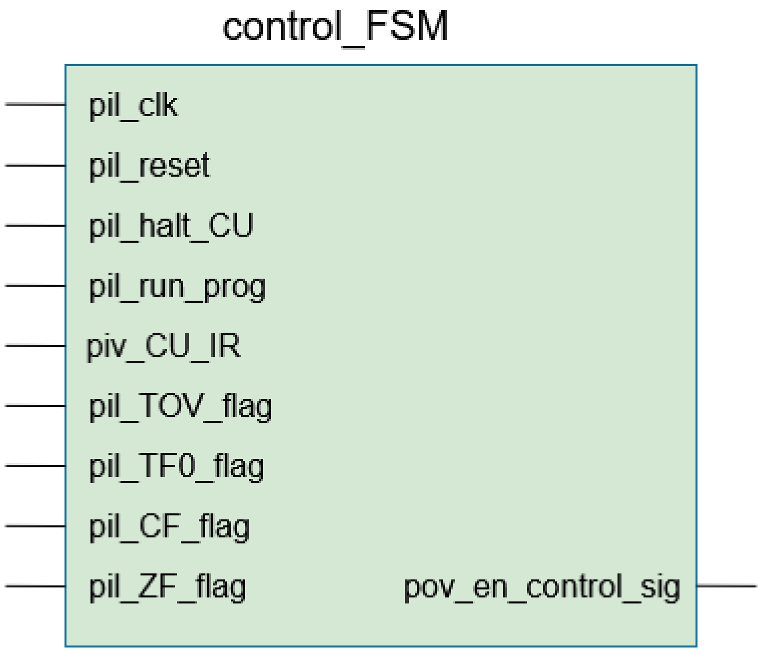


Figure 4.2.18-1 Control FSM block diagram

5. Microprogram

The control FSM sends out control words during each fetch and execution steps. These words are like directions telling the rest of the computer what to do. Since it produces a small step in the data processing, each of these control words are often called *microinstructions*. On the other hand, instructions such as *LD A, ST A, ADD* etc. are sometimes referred to as *macroinstructions*. Each macroinstruction is made up of multiple microinstructions. For example, LD A instruction consists of the microinstructions in section 3.6.2.

The control FSM stores these microinstructions in the form of an array. When the control FSM enters the execution cycle, it decodes the instruction opcode and accesses the corresponding microinstructions from the control FSM array. This process of writing microcode is called microprogramming.

The following sections shows the microinstructions for some of the commonly used Instruction.

5.1 LDA *address* (0x01)

The *LDA* instruction loads a data from the memory address location to the *A* register. The LDA instruction takes one operand which specifies the address location from which the data would be loaded. The microinstruction and the instruction duration are shown below.

Microinstruction:

Control FSM state	**Microinstruction**	
State_0a	*MI (20)*	*IO (17)*
State_1a	*RO (18)*	*AI (15)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

Where, $X_{IT} = Instruction\ execution\ time$

$F_T = fetch\ cycle\ duration$

$n = number\ of\ microinstruction\ cycles$

The fetch cycle duration F_T has a fixed value of 5 for the microcontroller.

$$X_{IT} = 5 + 2 \cdot 2$$

$$X_{IT} = 9$$

5.2 ADD/SUB *address* (0x02, 0x03)

The *ADD* instruction loads a data from the memory address location specified by the operand and adds it to the *A* register. The ADD operation is carried out by loading the data, which is being added, into the *B* register. The arithmetic addition is carried by the ALU unit which holds the result temporarily before transferring the result back to the *A* register.

The *SUB* instruction works the same way as the ADD instruction except that the on *State_2a*, control signal *SU* is also activated, which allows the ALU to perform arithmetic subtractions.

The microinstruction and the instruction duration are shown below.

Microinstruction for ADD:

Control FSM state	**Microinstruction**		
State_0a	*MI (20)*	*IO (17)*	-
State_1a	*RO (18)*	*BI (11)*	-
State_2a	*AI (15)*	*EO (13)*	*FI (06)*

Microinstruction for SUB:

Control FSM state	**Microinstruction**			
State_0a	*MI (20)*	*IO (17)*	-	-
State_1a	*RO (18)*	*BI (11)*	-	-
State_2a	*AI (15)*	*EO (13)*	*FI (06)*	*SU (12)*

Duration for ADD and SUB:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 3$$

$$X_{IT} = 11$$

5.3 OUT (0x0e)

The *OUT* instruction outputs the data stored in *A* register to the *Output port*. This instruction is mainly used for program debugging purposes and is directly connected to inbuilt LEDs in the FPGA development board. However, for *Spartan 3E* starter kit, this port can also be used as a general-purpose output port. The OUT instruction takes no additional operands for execution. The microinstruction and the instruction duration are shown below.

Microinstruction:

Control FSM state	**Microinstruction**	
State_0a	*AO (14)*	*OI (10)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.4 HLT (0x0f)

The *HLT* instruction is used to stop the microcontroller from executing any instruction that follows it. The HLT instruction activates the *pil_halt* control signal which then stops the Control FSM from executing. This instruction can also be used for debugging purposes or in scenarios where the program needs to be executed only once. Like the OUT instruction, the HLT instruction also takes no operands for execution. The microinstruction and the instruction duration are shown below.

Microinstruction:

Control FSM state	Microinstruction	
State_0a	*HLT (21)*	-

Duration:

$$X_{IT} = F_T + 1 \cdot n$$

$$X_{IT} = 5 + 1 \cdot 1$$

$$X_{IT} = 6$$

5.5 JMP *address* (0x06)

The *JMP* instruction performs an unconditional jump. It works by transferring the flow of execution of the program by changing the value in the Program counter. The program counter holds the address of the next instruction in the RAM. This address is replaced by the operand of the JMP instruction. Thus, during the next fetch cycle, the program counter points to the *branched* address. The microinstruction and the instruction duration are shown below.

Microinstruction:

Control FSM state	Microinstruction	
State_0a	*IO (17)*	*CI (07)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.6 JMC *address* (0x07)

The *JMC* instruction performs a conditional jump. The JMC instruction is performs the same way as the unconditional JMP instruction. In fact, the microinstruction for both the instructions are same as well. The only difference is that for the JMC instruction, the program execution jumps to the address specified by the operand only if the *carry* flag is set from the previous arithmetic or logical operation. The microinstruction and the instruction duration are shown below.

Microinstruction:

Control FSM state	**Microinstruction**	
State_0a (CF = 1)	*IO (17)*	*CI (07)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.7 JMZ *address* (0x08)

Like the JMC instruction the *JMZ* instruction is also a conditional jump instruction. The program execution jumps to the address specified by the operand only if the *zero* flag is set from the previous arithmetic or logical operation. The microinstruction and the instruction duration are shown below.

Microinstruction:

Control FSM state	**Microinstruction**	
State_0a (ZF = 1)	*IO (17)*	*CI (07)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.8 MOV A, R_ (0x10 – 0x1f)

The *MOV A, R_* instruction transfers a copy of the data stored in the GPR specified by *'_'* to the *A* register. The microinstruction for *MOV A, R1* and the instruction duration is shown below.

Control FSM state	Microinstruction	
State_0a	*AI (15)*	*GPROO (02)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.9 ADD/SUB R_ (0x20 – 31)

Unlike the ADD instruction which adds a data from a memory location to the *A* register, the *ADD R_* instruction adds the data stored in general-purpose register specified by '_' to the *A* register. The data stored in the general-purpose register remains unchanged while the result of the operation is stored in the *A* register.

The *SUB* instruction works the same way as the ADD instruction except that the on *State_1a*, control *SU* is also activated, which allows the ALU to perform arithmetic subtractions.

The microinstruction and the instruction duration for ADD/SUB R0 are shown below.

Microinstruction for ADD R0:

Control FSM state	Microinstruction		
State_0a	*GPROO (02)*	*BI (11)*	-
State_1a	*AI (15)*	*EO (13)*	*FI (06)*

Microinstruction for SUB R0:

Control FSM state	Microinstruction			
State_0a	*GPROO (02)*	*BI (11)*	-	-
State_1a	*AI (15)*	*EO (13)*	*FI (06)*	*SU (12)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 2$$

$$X_{IT} = 9$$

5.10 SET/CLR P0/1._ (0x32, 0x33, 0xb9, 0xba)

The *SET P0/1._* instruction sets the bit specified by the '_' of the *general purpose user io* port to *HIGH*. When the *SET/CLR* instruction is used as for bit addressing the user I/O pins, it takes additional operand (2^n) which tells the bit position of the n^{th} pin. For instance, machine code for SET P0.2 instruction would take *0x04* (2^2) as an operand.

During the execution cycle, in *State*_0a, the current state of all the pins that are configured as output are read and loaded in the *A* register. In *State*_1a, the operand value is loaded to the *B* register. In the next state, a bitwise *OR* operation is performed which preserves the old states of the other pins except for pin '_' which is set HIGH.

The *CLR P0/1._* instruction on the other hand sets the pin specified by '_' to *LOW*. This is done by activating the *pil_en_BIT_OP (CLR)* control signal in the ALU register which performs a bitwise *AND* operation with the complemented value of the *B* register.

The microinstruction and the instruction duration are shown below.

Microinstruction for SET P0._:

Control FSM state	Microinstruction		
State_0a	*IO (17)*	*BI (11)*	-
State_1a	*PORT0O (34)*	*AI (15)*	-
State_2a	*PBO (36)*	*AI (15)*	-
State_3a	*AO (14)*	*PORT0I (35)*	-

Microinstruction for CLR P0._:

Control FSM state	Microinstruction		
State_0a	*IO (17)*	*BI (11)*	-
State_1a	*PORT0O (34)*	*AI (15)*	-
State_2a	*PBO (36)*	*AI (15)*	*CLR (37)*
State_3a	*AO (14)*	*PORT0I (35)*	-

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 4$$

$$X_{IT} = 13$$

5.11 DJNZ R_, *address* (0x51 – 0x59, 0x96)

The *DJNZ R_, address* instruction is used to perform conditional loop operations. During the execution of the DJNZ instruction, if the data stored in the R_ register is *non-zero*, the content of the register is decremented by 1 and the program counter is loaded with the new address specified by the operand thus performing *branching*. If the register value is zero, the program flow breaks from the loop and executes the next instruction that follows. The microinstruction and the instruction duration for the DJNZ R0, address instruction is shown below.

Microinstruction for DJNZ R0, *address*:

Control FSM state	Microinstruction			
State_0a	*GPR0O (02)*	*AI (15)*	-	-
State_1a	*CG0O (43)*	*BI (11)*	-	-
State_2a	*EO (13)*	*FI (06)*	-	-
State_3a (ZF = 0)	*IO (17)*	*CI (07)*	-	-
State_4a	*CG1O (44)*	*BI (11)*	-	-
State_5a	*AI (15)*	*EO (13)*	*SU (12)*	*FI (06)*
State_6a	*AO (14)*	*GPR0I (05)*		

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 7$$

$$X_{IT} = 19$$

5.12 CALL/RET *address* (0x5a, 0x5b)

The *CALL address* along with the *RET* instruction is used to perform unconditional subroutine branch operations. Before branching to the address specified by the operand, the address of the next instruction that follows the subroutine call is linked on the stack. To return from the subroutine, the *RET* instruction must be used which retrieves the last linked address from the stack and loads it into the program counter. The microinstruction and the instruction duration for *CALL address* and *RET* instruction is shown below.

Microinstruction for CALL *address*:

Control FSM state	Microinstruction	
State_0a	*CO (08)*	*SMI (48)*
State_1a	*IO (17)*	*CI (07)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 2$$

$$X_{IT} = 9$$

Microinstruction for RET:

Control FSM state	Microinstruction	
State_0a	*SMO (49)*	*CI (07)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.13 DDRP0/1 *value* (0x62 – 0xb5)

The *DDRP0/1 value* instruction loads the value specified by the operand into the *Data direction register* which configures the direction of the data flow of the pins for the general-purpose user ports. To configure a pin as *input*, a logic ‘1’ must be loaded into the DDR register into its corresponding bit position, on the other hand a logic ‘0’ must be loaded to configure the pin as an *output*. During power up or when the asynchronous reset button is pressed, all the pins of the user ports are configured as outputs. The microinstruction and the instruction duration of the DDRP0 instruction is shown below.

Microinstruction for DDRP0 *value*:

Control FSM state	Microinstruction	
State_0a	*IO (17)*	*P0DIR (50)*

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 1$$

$$X_{IT} = 7$$

5.14 JB/JNB P0/1._, *address* (0x63 – 0x66)

The *JB P0/1._, address* instruction performs a conditional branching to the address specified by the second operand if the pin specified by '_' is *HIGH*. The JB/JNB instruction takes two 8-bit values as operands where the first 8-bit value represents the pin number while the second represents the branching address location. The JB/JNB instruction takes two successive memory cells in the RAM where the second memory cell stores the branch address location.

During the execution of the JB P0._ instruction, the state of the pin specified by '_' is checked and a branching is performed if the pin is at a logic '1'. The JNB instruction performs a branching if the pin is at a logic '0'. Like the other branching statement except DJNZ, the data stored in the *A* register prior to the execution of the JB/JNB instruction remains unchanged. The microinstruction and the instruction duration for *JB/JNB P0.2, address* instruction is shown below.

Microinstruction for JB P0.1, *address*:

Control FSM state	Microinstruction		
State_0a	*AO (14)*	*SMI (48)*	-
State_1a	*IO (17)*	*BI (11)*	*P0INR (52)*
State_2a	*AI (15)*	*P0IN (51)*	-
State_3a	*EO (13)*	*SU (12)*	*FI (06)*
State_4a (ZF = 1) *(ZF = 0)*	*MI (20)* *CE (09)*	*CO (08)* -	- -
State_5a	*RO (18)*	*CI (07)*	-
State_6a	*AI (15)*	*SMO (49)*	-

Microinstruction for JNB P0.1, *address*:

Control FSM state	Microinstruction		
State_0a	*AO (14)*	*SMI (48)*	-
State_1a	*IO (17)*	*BI (11)*	*P0INR (52)*
State_2a	*AI (15)*	*P0IN (51)*	-
State_3a	*EO (13)*	*SU (12)*	*FI (06)*
State_4a (ZF = 0) *(ZF = 1)*	*MI (20)* *CE (09)*	*CO (08)* -	- -
State_5a	*RO (18)*	*CI (07)*	-
State_6a	*AI (15)*	*SMO (49)*	-

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 7$$

$$X_{IT} = 19$$

5.15 MOV P0/1._, C (0x99 – 0xbb)

The *MOV P0/1._, C* instruction loads the current bit stored in the carry bit of the *A* register into the general-purpose user port pin specified by '_'. The instruction takes an additional operand (2^n) which tells the bit position of the n^{th} pin. This instruction when combined with the rotate instruction is very helpful when transmitting serial data to a sensor, host computer or other devices using only one pin. It is to be noted that the carry bit of the *A* register is different to the carry flag of the microcontroller and the main purpose of the carry bit is to simplify the use of serial communication. When using the MOV P0/1._, C instruction, the data stored in the *A* register prior to the execution of the instruction remains unchanged. The microinstruction and the instruction duration of MOV P0.2, C instruction is shown below.

Microinstruction for MOV P0.2, *C*:

Control FSM state	Microinstruction			
State_0a	*AO (14)*	*SMI (48)*	-	-
State_1a	*IO (17)*	*AI (15)*	-	-
State_2a	*AROP1 (59)*	*ROT (57)*	-	-
State_3a	*CG0O (43)*	*BI (11)*	-	-
State_4a	*EO (13)*	*SU (12)*	*BI (11)*	*FI (06)*
State_5a	*AI (15)*	*PORT0O (34)*	-	-
State_6a	*IO (17)*	*BI (11)*		-
State_7a	*PBO (36)*	*PORT0I (35)*	-	-
State_8a	*CLR (37)*	*PBO (36)*	*PORT0I (35)*	-
State_9a	*SMO (49)*	*AI (15)*	-	-

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 10$$

$$X_{IT} = 25$$

5.16 MOV C, P0/1._ (0x9a – 0xbc)

When a user port pin is configured as an input, the MOV C, P0/1._ instruction can be used to read the pin state and store it into the carry bit of the *A* register. This instruction is particularly useful when combined with the rotate instruction to read serial data from sensors or other devices with only one pin. When using the MOV C, P0/1._ instruction, the data stored in the *A* register prior to the execution of the instruction remains unchanged. The microinstruction and the instruction duration of MOV C, P0.2 instruction is shown below.

Microinstruction for MOV C, P0.2:

Control FSM state	Microinstruction		
State_0a	*AO (14)*	*SMI (48)*	-
State_1a	*P0INR (52)*	*IO (17)*	-
State_2a	*P0IN (51)*	*AI (15)*	-
State_3a	*AROP1 (59)*	*AROP0 (58)*	*ROT (57)*
State_4a	*CYI (60)*	*AO (14)*	-
State_5a	*AI (15)*	*SMO (49)*	-

Duration:

$$X_{IT} = F_T + 2 \cdot n$$

$$X_{IT} = 5 + 2 \cdot 6$$

$$X_{IT} = 17$$

6. Setting up the FPGA

The microcontroller architecture design would be tested on a *Spartan 3E starter kit*, *intel MAX 1000* board and intel *CYC 1000 board*. The following sections describes the configuration of the different types of devices in details.

6.1 Xilinx Spartan 3E starter kit

The microcontroller uses 29 I/O peripherals out of which 16 are bi-directional pins for the general-purpose user port, 8 are used for the output port and the remaining 5 pins are all input pins including the system clock. The microcontroller requires an optimal $f_{sys} = 35\ MHz$ for its functioning. However, the FPGA has an inbuilt oscillator at 50MHz which would be used as the clock source of the microcontroller. The starter kit on its own does not have enough GPIO pins that can be used by the microcontroller. However, the starter kit does support a *100-pin FX2 expansion connector* which provides an additional 43 I/O pins for the microcontroller to use.

6.1.1 Project setup

Once a new project is created, Xilinx stores all the source files that are necessary for the project into the main project directory by default. However, it is a recommended that a new folder *(rtl)* is created with the necessary source files. This not only makes the project directory more organized but also facilitates easier code portability. The microcontroller requires the following source files in the rtl folder to function properly.

1. pearl_top.vhd
2. programmer_inteface.vhd
3. isp.vhd
4. core_top.vhd
5. da_buffer.vhd
6. mar.vhd
7. iram.vhd
8. accumulator.vhd
9. b_reg.vhd
10. GP_Register0.vhd (GPR type1)
11. GP_Register3.vhd (GPR type2)
12. alu.vhd
13. Information_Register.vhd
14. program_header.vhd
15. user_o.vhd
16. user_io.vhd
17. multibus.vhd
18. Flag_Register.vhd
19. timer_counter.vhd
20. constant_generator.vhd
21. stack_mem.vhd
22. control_fsm.vhd

The following package files must be included as well.

1. accumulator_p.vhd
2. alu_p.vhd
3. control_fsm_p.vhd

6.1.2 UCF location constraints

For *fitter* and *routing* implementation a .ucf must be created.

The following table provides the UCF constraints for the microcontroller, including the I/O pin assignments and the I/O standards used.

Node name	**Direction**	**Board location**	**Fitter location**	**I/O standard**
Pil_reset	Input	SW0	L13	LVTTL - pullup
Pil_clk	Input	CLK_50MHz	C9	LVCMOS33
Pil_sel	Input	SW3	N17	LVTTL - pullup
Pil_run_prog	Input	SW2	H18	LVTTL - pullup
Pil_rx	Input	FX2_IO <12>	E8	LVTTL
*Pov_OR <0>	Output	LED0	F12	LVTTL
* Pov_OR <1>	Output	LED1	E12	LVTTL
* Pov_OR <2>	Output	LED2	E11	LVTTL

* Pov_OR <3>	Output	LED3	F11	LVTTL
* Pov_OR <4>	Output	LED4	C11	LVTTL
* Pov_OR <5>	Output	LED5	D11	LVTTL
* Pov_OR <6>	Output	LED6	E9	LVTTL
* Pov_OR <7>	Output	LED7	F9	LVTTL
Pbv_P1 <0>	Bidir	FX2_IO <21>	A13	LVTTL
Pbv_P1 <1>	Bidir	FX2_IO <22>	B13	LVTTL
Pbv_P1 <2>	Bidir	FX2_IO <23>	A14	LVTTL
Pbv_P1 <3>	Bidir	FX2_IO <24>	B14	LVTTL
Pbv_P1 <4>	Bidir	FX2_IO <25>	C14	LVTTL
Pbv_P1 <5>	Bidir	FX2_IO <26>	D14	LVTTL
Pbv_P1 <6>	Bidir	FX2_IO <27>	A16	LVTTL
Pbv_P1 <7>	Bidir	FX2_IO <28>	B16	LVTTL
Pbv_P0 <0>	Bidir	FX2_IO <1>	B4	LVTTL
Pbv_P0 <1>	Bidir	FX2_IO <2>	A4	LVTTL
Pbv_P0 <2>	Bidir	FX2_IO <3>	D5	LVTTL
Pbv_P0 <3>	Bidir	FX2_IO <4>	C5	LVTTL
Pbv_P0 <4>	Bidir	FX2_IO <5>	A6	LVTTL
Pbv_P0 <5>	Bidir	FX2_IO <6>	B6	LVTTL
Pbv_P0 <6>	Bidir	FX2_IO <7>	E7	LVTTL
Pbv_P0 <7>	Bidir	FX2_IO <8>	F7	LVTTL

* The output port which is connected to the on-board LEDs are also available in the FX2 connector from board location FX2_IO <13> to FX2_IO <20>. Thus, the port can not only be used for debugging but also as a special purpose output port.

The final .ucf file is shown below.

```
NET "pil_reset" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP;
NET "pil_clk" LOC = "C9" | IOSTANDARD = LVCMOS33;

NET "pil_sel" LOC = "N17" | IOSTANDARD = LVTTL | PULLUP;
NET "pil_run_prog" LOC = "H18" | IOSTANDARD = LVTTL | PULLUP;

# IO 12

NET "pil_rx" LOC = "E8" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6;

# Output Register

NET "pov_OR<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<6>" LOC = "E9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pov_OR<7>" LOC = "F9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;


# PORT1 Register IO 21 22 23 24 25 26 27 28 -> (0) (1) (2) (3) (4) (5) (6) (7)

NET "pbv_P1<0>" LOC = "A13" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<1>" LOC = "B13" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<2>" LOC = "A14" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<3>" LOC = "B14" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<4>" LOC = "C14" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<5>" LOC = "D14" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<6>" LOC = "A16" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P1<7>" LOC = "B16" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;

# PORT0 Register IO 1 2 3 4 5 6 7 8 -> (0) (1) (2) (3) (4) (5) (6) (7)

NET "pbv_P0<0>" LOC = "B4" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<1>" LOC = "A4" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<2>" LOC = "D5" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<3>" LOC = "C5" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<4>" LOC = "A6" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<5>" LOC = "B6" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<6>" LOC = "E7" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "pbv_P0<7>" LOC = "F7" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
```

6.1.3 Design Synthesis and summary

Once the project file contains all the necessary source file according to section 6.1.1, the *pearl_top.vhd* file must be set as the *Top Module* for design synthesis. The design should synthesise with no errors and a few warnings which can be ignored. The device utilisation along with synthesis timing and summary are shown below.

Device utilisation summary:

Selected device: 3s500efg320-4

Number of slices	2122	out of 4656	45 %
Number of slice Flip Flops	1834	out of 9312	19 %
Number of 4 input LUTs	3192	out of 9312	34 %
Number used as logic	2936		
Number used as RAMs	256		
Number of IOs	29		
Number of bonded IOBs	29	out of 232	12 %
Number of GCLKs	1	out of 24	4 %

Timing Summary:

Speed grade: -4

Minimum period: 27.536ns (Maximum Frequency: 36.316MHz)

Minimum input arrival time before clock: 6.626ns

Maximum output required time after clock: 4.394ns

As seen from the Timing summary, the maximum Frequency that the design can achieve with the current architecture is 36.3 MHz. As discussed earlier, the Starter kit comes with a base clock frequency of 50MHz and to achieve the necessary clock frequency, two procedures can be used.

1. Using clock divider: To achieve a frequency which is lower than the base clock frequency, a clock divider module can be used. However, using a clock division technique could introduce unnecessary jitters in the design which might affect the overall functionality of the design.

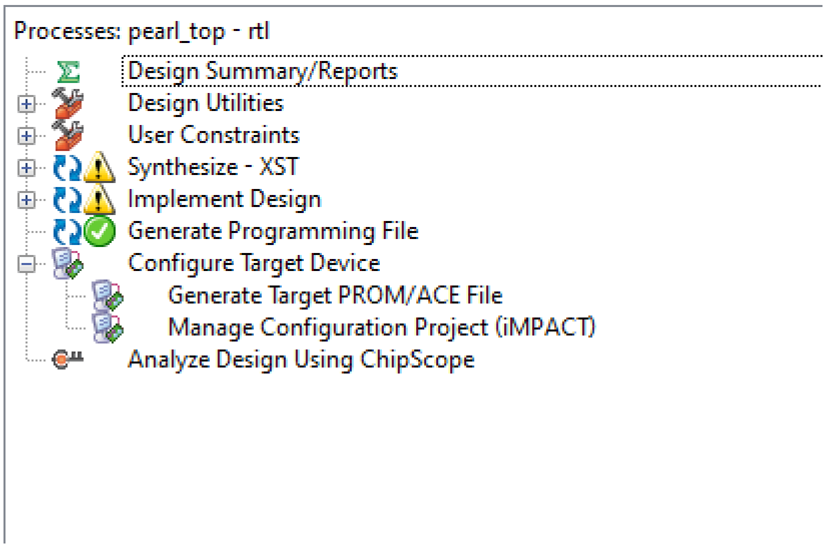
2. Using phase-locked loop: Phase-locked loop or pll can be used to achieve the necessary system clock for the microcontroller. However, creating a pll for Xilinx Spartan 3E is a bit complicated for beginners.

This timing summary is critical and must be considered when using the design in different conditions or using it for longer durations of time. Thus, for testing and demonstrating purposes, the timing report can be ignored.

6.1.4 PROM configuration

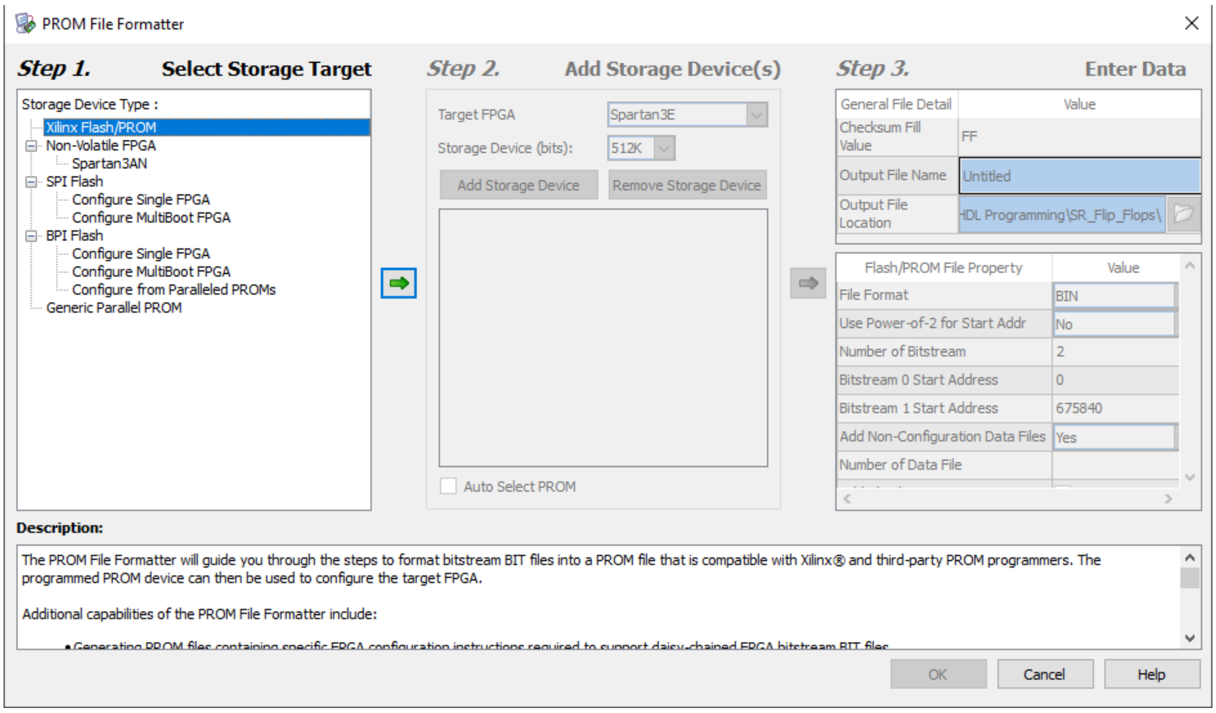
Once the power to the FPGA is turned off, the hardware configuration is lost. However, programming the *Flash PROM* with the hardware design automatically configures the FPGA every time when it is powered on. It is recommended to test the design properly before programming the Flash PROM as it would reconfigure the existing design. To configure the Flash PROM the following steps must be followed.

1. In the *Processes* pane, double click on *Configure Target Device*.



Processes: pearl_top - rtl
- Design Summary/Reports
- Design Utilities
- User Constraints
- Synthesize - XST
- Implement Design
- Generate Programming File
- Configure Target Device
  - Generate Target PROM/ACE File
  - Manage Configuration Project (iMPACT)
- Analyze Design Using ChipScope

2. In the *ISE iMPACT* tool double click *Create PROM File*. This would open the *PROM File Formatter* wizard.



PROM File Formatter

Step 1. Select Storage Target

Storage Device Type :

- Xilinx Flash/PROM
- Non-Volatile FPGA
  - Spartan3AN
- SPI Flash
  - Configure Single FPGA
  - Configure MultiBoot FPGA
- BPI Flash
  - Configure Single FPGA
  - Configure MultiBoot FPGA
  - Configure from Paralleled PROMs
- Generic Parallel PROM

Step 2. Add Storage Device(s)

Target FPGA: Spartan3E

Storage Device (bits): 512K

Add Storage Device | Remove Storage Device

Auto Select PROM

Step 3. Enter Data

General File Detail	Value
Checksum Fill Value	FF
Output File Name	Untitled
Output File Location	HDL Programming\SR_Flip_Flops\

Flash/PROM File Property	Value
File Format	BIN
Use Power-of-2 for Start Addr	No
Number of Bitstream	2
Bitstream 0 Start Address	0
Bitstream 1 Start Address	675840
Add Non-Configuration Data Files	Yes
Number of Data File	

Description:

The PROM File Formatter will guide you through the steps to format bitstream BIT files into a PROM file that is compatible with Xilinx® and third-party PROM programmers. The programmed PROM device can then be used to configure the target FPGA.

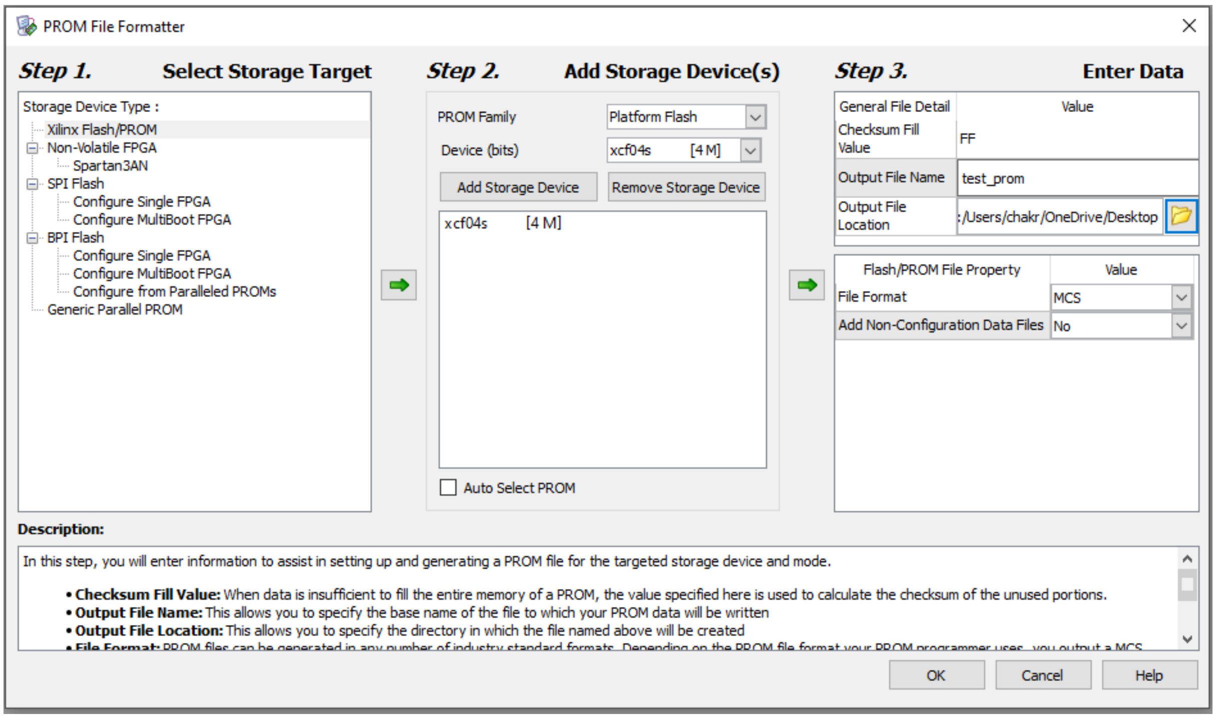
Additional capabilities of the PROM File Formatter include:

• Generating PROM files containing specific FPGA configuration instructions required to support daisy-chained FPGA bitstream BIT files

OK | Cancel | Help

3. In *Step 1* of the wizard select *Xilinx Flash/PROM* from *Storage Device Type* drop down option and click the next arrow button to proceed to the next step.
4. In *Step 2*, from the *PROM Family* drop down menu select *Platform Flash* if not selected automatically. From *Device* drop down menu select *xcf04s [4 M]* option. Then click *Add Storage Device* button. This will add the device selected earlier into the text field below. Click next to proceed.

5. In *Step 3*, under the *Output File Name*, enter the name of the PROM file and select the location where it would be stored. Under the *Flash/PROM File Property* select *MCS* as the file format. Then Click OK to save and a new UI will open.



PROM File Formatter

Step 1. Select Storage Target

Storage Device Type :
- Xilinx Flash/PROM
- Non-Volatile FPGA
  - Spartan3AN
- SPI Flash
  - Configure Single FPGA
  - Configure MultiBoot FPGA
- BPI Flash
  - Configure Single FPGA
  - Configure MultiBoot FPGA
  - Configure from Paralleled PROMs
- Generic Parallel PROM

Step 2. Add Storage Device(s)

PROM Family	Platform Flash
Device (bits)	xcf04s [4 M]

Add Storage Device | Remove Storage Device

xcf04s [4 M]

Auto Select PROM

Step 3. Enter Data

General File Detail	Value
Checksum Fill Value	FF
Output File Name	test_prom
Output File Location	:/Users/chakr/OneDrive/Desktop

Flash/PROM File Property	Value
File Format	MCS
Add Non-Configuration Data Files	No

Description:

In this step, you will enter information to assist in setting up and generating a PROM file for the targeted storage device and mode.

- **Checksum Fill Value:** When data is insufficient to fill the entire memory of a PROM, the value specified here is used to calculate the checksum of the unused portions.
- **Output File Name:** This allows you to specify the base name of the file to which your PROM data will be written
- **Output File Location:** This allows you to specify the directory in which the file named above will be created
- **File Format:** PROM files can be generated in any number of industry standard formats. Depending on the PROM file format your PROM programmer uses, you output a MCS

OK | Cancel | Help

6. Click Ok to open the File browser manager. From the browser manager select the *pearl_top.bit* file from the project folder which was generated after running the fitter and click open.

7. Click No when the *Add another device* dialog box pops up.

8. Click *Operations* from the menu lists and click on *Generate* to generate the PROM file.

9. Click the Close button to close the window (No need to save).

10. Double click on the *Configure Target Device* and double click on *Boundary Scan*.

11. Right click on the window and click on *Initialize Chain*.

12. The *pearl_top.bit* file in the device and assign the PROM file to the *xccf04s* on the JTAG chain. Right click on the PROM icon, then click *Assign new Configuration File*. Select the generated prom file and click ok.

13. To start the programming right click on the PROM icon. Even though when the Program button is clicked, the previous contents of the PROM is erased first and then the new PROM file is uploaded into the device, still it is a good practice to *Erase* the PROM file manually and then click *Program* button.

6.2 Intel cyc1000 / max1000 evaluation board

The *CYC1000* is a customizable IoT / Maker Board for evaluation and development. It is built around the cyclone 10 LP FPGA, which is optimized for low cost and power in very small package which makes it suitable for bread boarding and testing.

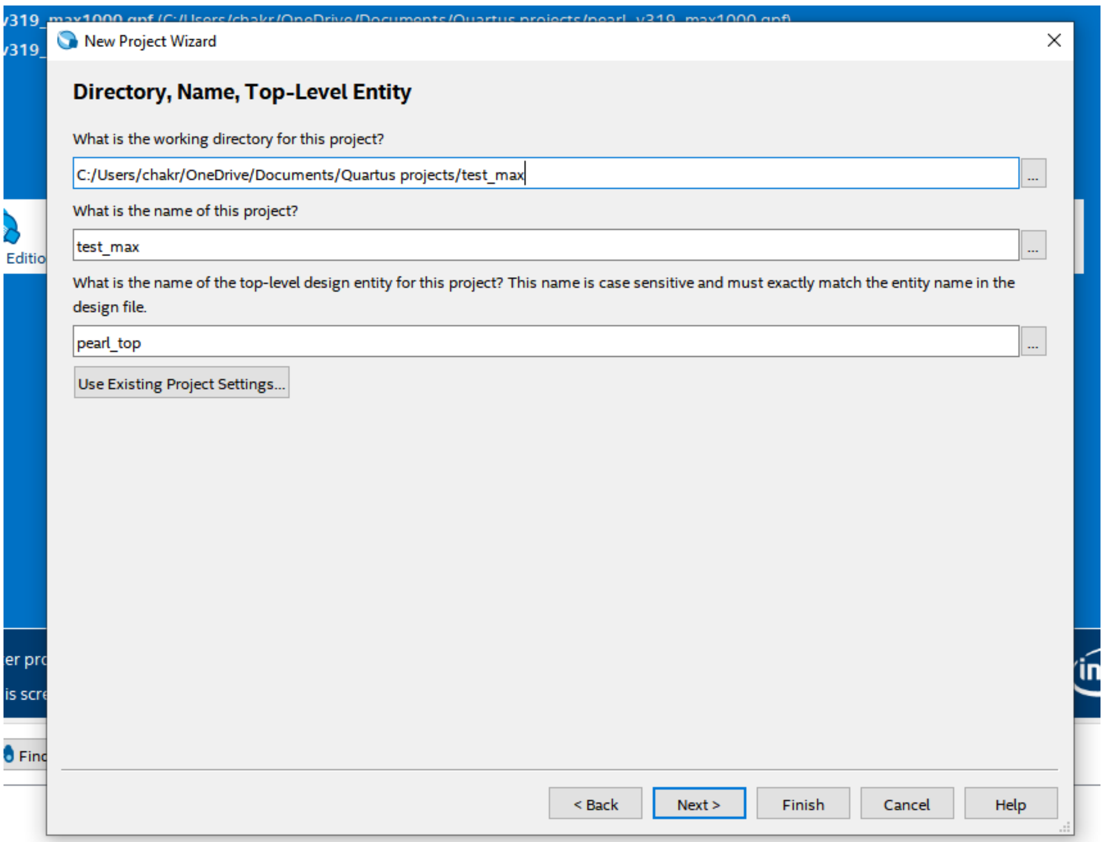
The cyc1000 and the max1000 boards both come with a base clock of 12MHz. Since the architecture can run at $f_{sys} = 35\ MHz$, *altera phased-locked loop (pll)* would be used to achieve the desired frequency.

For cyc1000 and max1000 evaluation boards, Quartus Prime Lite 18.1 or above would be used.

6.2.1 Project setup

Once the Quartus Prime Application is open, the following instructions must be followed to create a project.

1. Click *New Project Wizard* button from the Home window or from *File* menu list. This will load the introduction page.
2. Click Next and enter the working directory, and the name of the project. Enter *pearl*_top in the top-lever entity name field. The top-level entity name can be changed later from *settings* option if not entered now.


New Project Wizard

Directory, Name, Top-Level Entity

What is the working directory for this project?

C:/Users/chakr/OneDrive/Documents/Quartus projects/test_max

What is the name of this project?

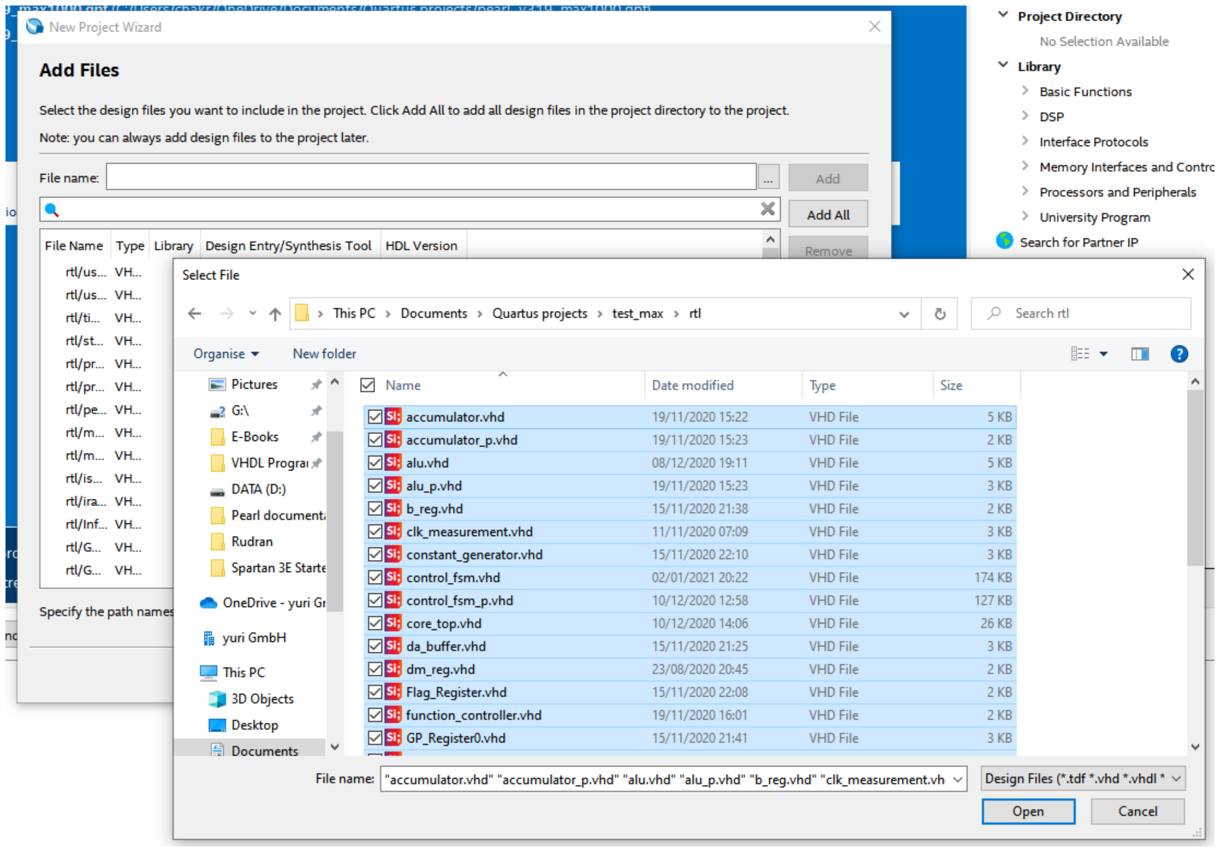
test_max

What is the name of the top-level design entity for this project? This name is case sensitive and must exactly match the entity name in the design file.

pearl_top

3. Click Next. Select Empty Project and click Next to proceed.

4. Copy the *rtl* folder containing all the necessary vhdl files (same as the Xilinx project rtl folder) into the project directory. Once the folder is copied, in the *Add Files* window click the browse button and add all the files from the rtl folder. Click Next to proceed.



New Project Wizard

Add Files

Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

Note: you can always add design files to the project later.

File name:

File Name	Type	Library	Design Entry/Synthesis Tool	HDL Version
rtl/us...	VH...			
rtl/us...	VH...			
rtl/ti...	VH...			
rtl/st...	VH...			
rtl/pr...	VH...			
rtl/pr...	VH...			
rtl/pe...	VH...			
rtl/m...	VH...			
rtl/m...	VH...			
rtl/is...	VH...			
rtl/ira...	VH...			
rtl/Inf...	VH...			
rtl/G...	VH...			
rtl/G...	VH...			

Specify the path names

Project Directory
No Selection Available
Library
Basic Functions
DSP
Interface Protocols
Memory Interfaces and Contrc
Processors and Peripherals
University Program
Search for Partner IP

Select File

This PC > Documents > Quartus projects > test_max > rtl

Name	Date modified	Type	Size
accumulator.vhd	19/11/2020 15:22	VHD File	5 KB
accumulator_p.vhd	19/11/2020 15:23	VHD File	2 KB
alu.vhd	08/12/2020 19:11	VHD File	5 KB
alu_p.vhd	19/11/2020 15:23	VHD File	3 KB
b_reg.vhd	15/11/2020 21:38	VHD File	2 KB
clk_measurement.vhd	11/11/2020 07:09	VHD File	3 KB
constant_generator.vhd	15/11/2020 22:10	VHD File	3 KB
control_fsm.vhd	02/01/2021 20:22	VHD File	174 KB
control_fsm_p.vhd	10/12/2020 12:58	VHD File	127 KB
core_top.vhd	10/12/2020 14:06	VHD File	26 KB
da_buffer.vhd	15/11/2020 21:25	VHD File	3 KB
dm_reg.vhd	23/08/2020 20:45	VHD File	2 KB
Flag_Register.vhd	15/11/2020 22:08	VHD File	2 KB
function_controller.vhd	19/11/2020 16:01	VHD File	2 KB
GP_Register0.vhd	15/11/2020 21:41	VHD File	3 KB

File name: "accumulator.vhd" "accumulator_p.vhd" "alu.vhd" "alu_p.vhd" "b_reg.vhd" "clk_measurement.vh

Design Files (*.tdf *.vhd *.vhdl *

5. In the *Family, Device* window, select the family name from the drop-down list; Cyclone 10 LP for cyc1000 and MAX 10 for max1000 boards.

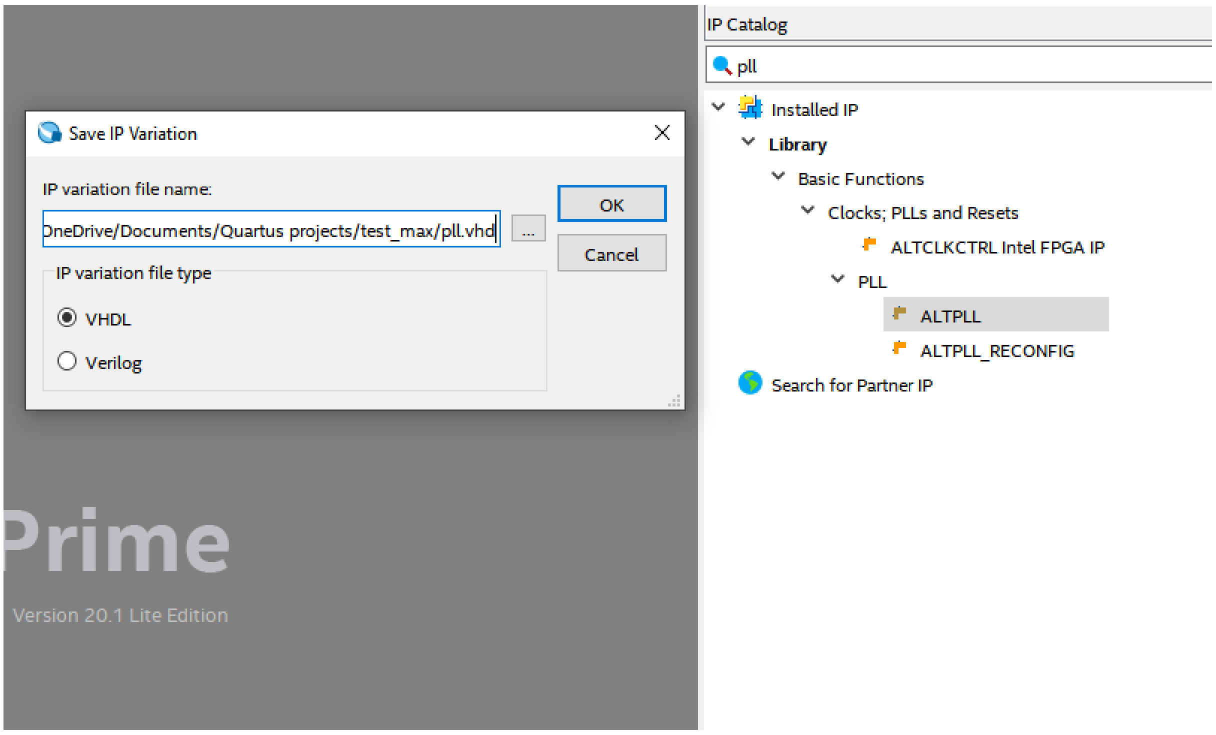
6. In the Name Filter enter 10CL025YU256C8G for cyc1000 and 10M16SAU169C8G for max1000 board. Select the Name from the available device list and Click Next.

7. In the *EDA Tool Settings* window click Finish to complete the project setup.

6.2.2 Adding Phased-locked loop

To add the pll module into the project follow the following steps.

1. On the *IP Catalog* Pane enter ‘pll’ and double click on *ALTPLL* from the pll list.
2. On the dialog box select VHDL and enter the directory path where the pll would be saved and add ‘/pll.vhd’ at the end and Click OK.



Save IP Variation

IP variation file name:

DneDrive/Documents/Quartus projects/test_max/pll.vhd

IP variation file type

VHDL [✓]

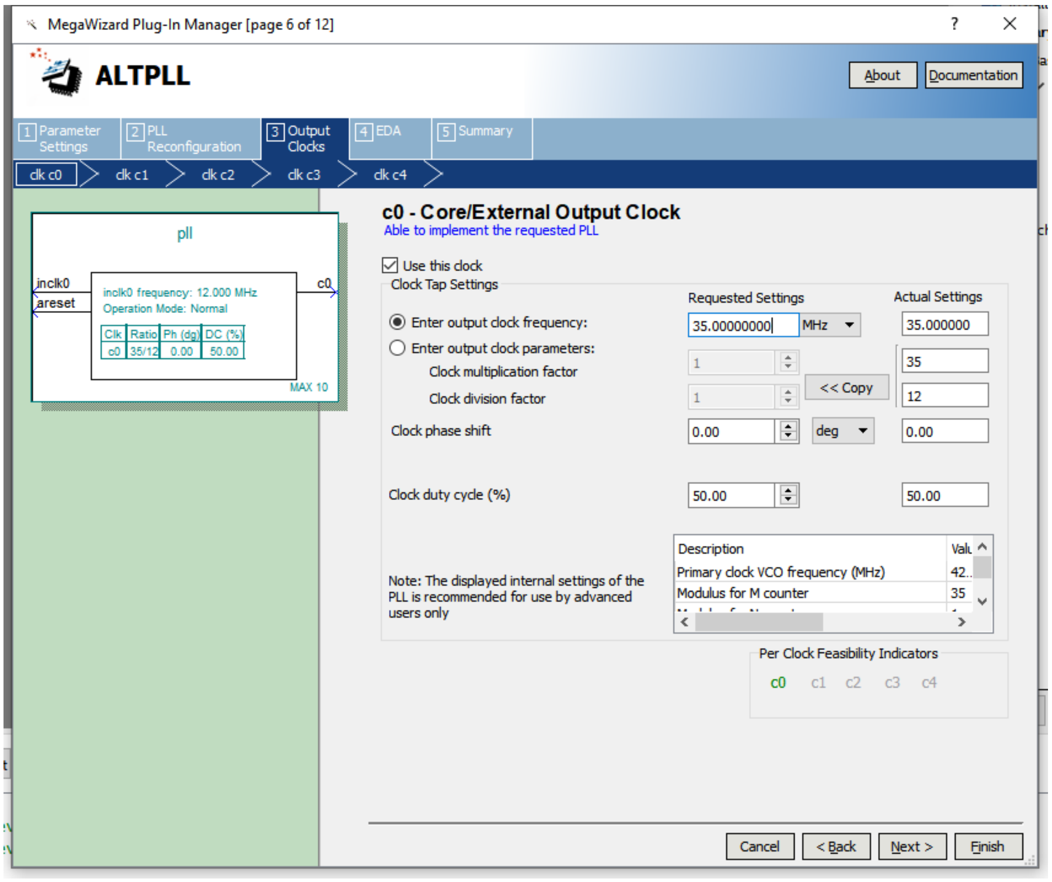
Verilog

IP Catalog

pll

- Installed IP
  - Library
    - Basic Functions
      - Clocks; PLLs and Resets
        - ALTCLKCTRL Intel FPGA IP
        - PLL
          - ALTPLL
          - ALTPLL_RECONFIG
- Search for Partner IP

3. In the *ALTPLL* wizard change the *inclk0 input* to 12.000 MHz and click Next to proceed.
4. Deselect the *'locked' output* option and select the *'areset' input* option and click Next.
5. Click Next in the following pages till *c0- Core/External Output Clock* page.
6. Select *Enter output clock frequency* and enter 35.000 MHz into the field next to it and Click Next.



MegaWizard Plug-In Manager [page 6 of 12]

ALTPLL

inclk0 frequency: 12.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	35/12	0.00	50.00

c0 - Core/External Output Clock

Able to implement the requested PLL

☑ Use this clock

Clock Tap Settings

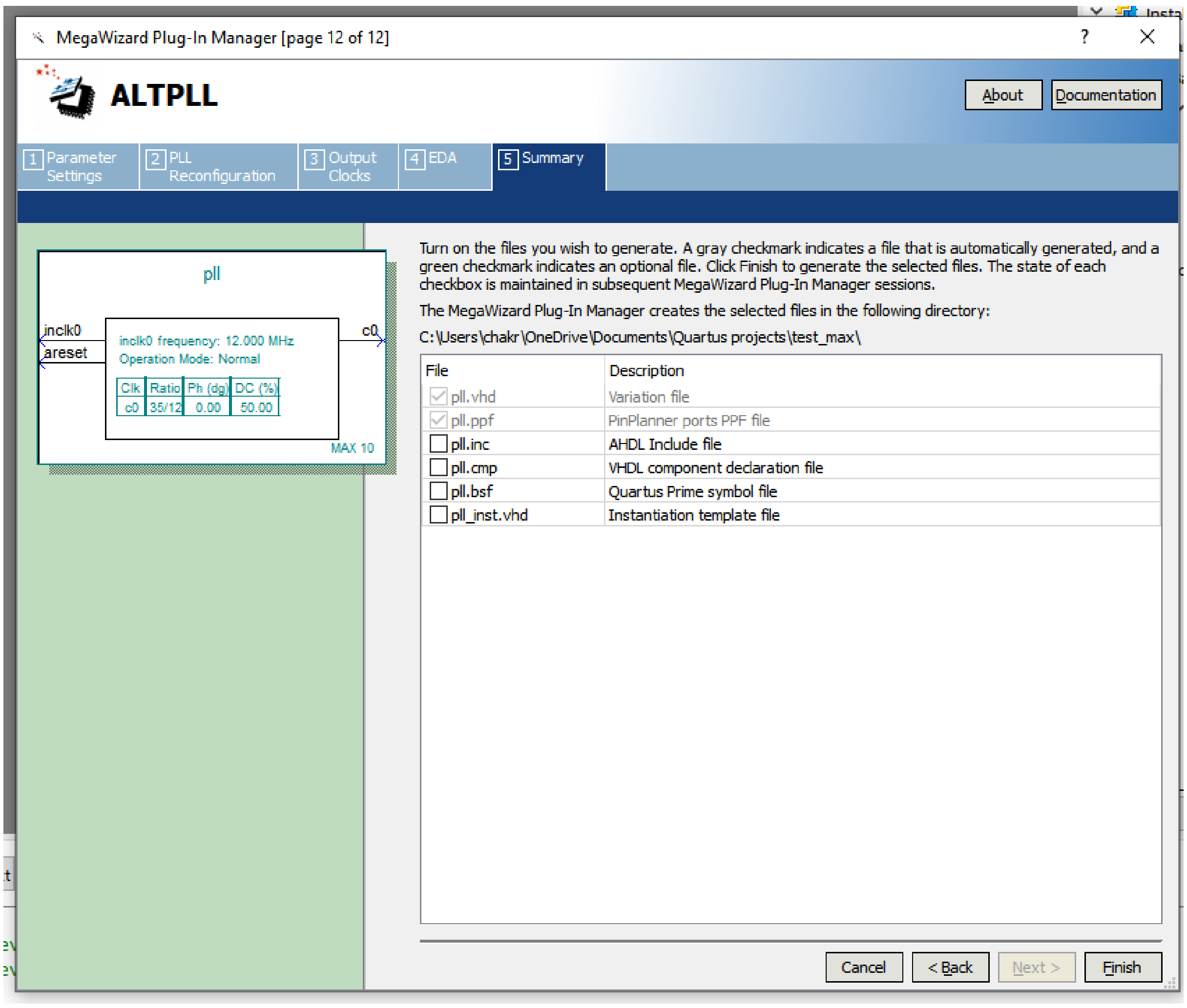
	Requested Settings		Actual Settings
[✓] Enter output clock frequency:	35.00000000	MHz	35.000000
Enter output clock parameters:			
Clock multiplication factor	1	<< Copy	35
Clock division factor	1		12
Clock phase shift	0.00	deg	0.00
Clock duty cycle (%)	50.00		50.00

Note: The displayed internal settings of the PLL is recommended for use by advanced users only

Description	Valu
Primary clock VCO frequency (MHz)	42..
Modulus for M counter	35

Per Clock Feasibility Indicators

7. Click Next in the following pages till the summary page. Deselect *pll.cmp option*. Click Finish to create the pll.



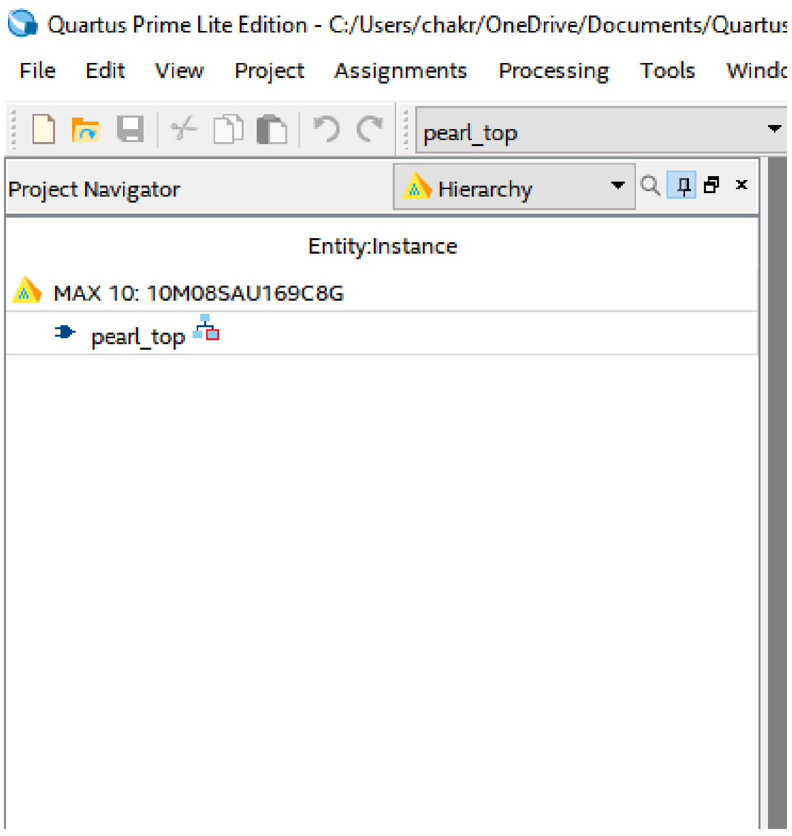
Turn on the files you wish to generate. A gray checkmark indicates a file that is automatically generated, and a green checkmark indicates an optional file. Click Finish to generate the selected files. The state of each checkbox is maintained in subsequent MegaWizard Plug-In Manager sessions.

The MegaWizard Plug-In Manager creates the selected files in the following directory:

C:\Users\chakr\OneDrive\Documents\Quartus projects\test_max\

File	Description
✓ pll.vhd	Variation file
✓ pll.ppf	PinPlanner ports PPF file
☐ pll.inc	AHDL Include file
☐ pll.cmp	VHDL component declaration file
☐ pll.bsf	Quartus Prime symbol file
☐ pll_inst.vhd	Instantiation template file

8. The *pll.qip* should be automatically added to the project. If not, then click on the project menu and then click on *Add/Remove files in project* and add the pll.qip file manually. The pll.qip file should be visible under the file option in the *Project Navigator* pane. Any future changes to the pll IP core could be done form the *IP Component* option in the *Project Navigator* pane.


Quartus Prime Lite Edition - C:/Users/chakr/OneDrive/Documents/Quartus
File Edit View Project Assignments Processing Tools Windo
pearl_top
Project Navigator
Hierarchy
Entity:Instance
MAX 10: 10M08SAU169C8G
pearl_top

6.2.3 Adding pll component in top-level

In the rtl folder, the old pearl_top.vhd file can be replaced with the modified pearl_top.vhd file. To add the pll component manually in the top-level, the following changes must be made in the *pearl_top.vhd* file only.

1. Add the following lines of code in the architecture of the pearl_top.vhd file:

```
sl_nrst <= not pil_reset;

inst_pll : entity work.pll
        port map(
                areset => sl_nrst,
                inclk0 => pil_clk_12MHz,
                c0     => sl_clk_c0
                  );
```

2. Declare the following signals under the signal declaration part of the architecture.

```
signal sl_nrst : std_logic;
signal sl_clk_c0 : std_logic;
```

3. In the module instantiation of the *Program interface*, *ISP,* and *core_architecture_top* replace the previous named association of *pil_clk and pil_reset* signals with the *sl_clk_c0* and *sl_nrst* signals, respectively. For instance,

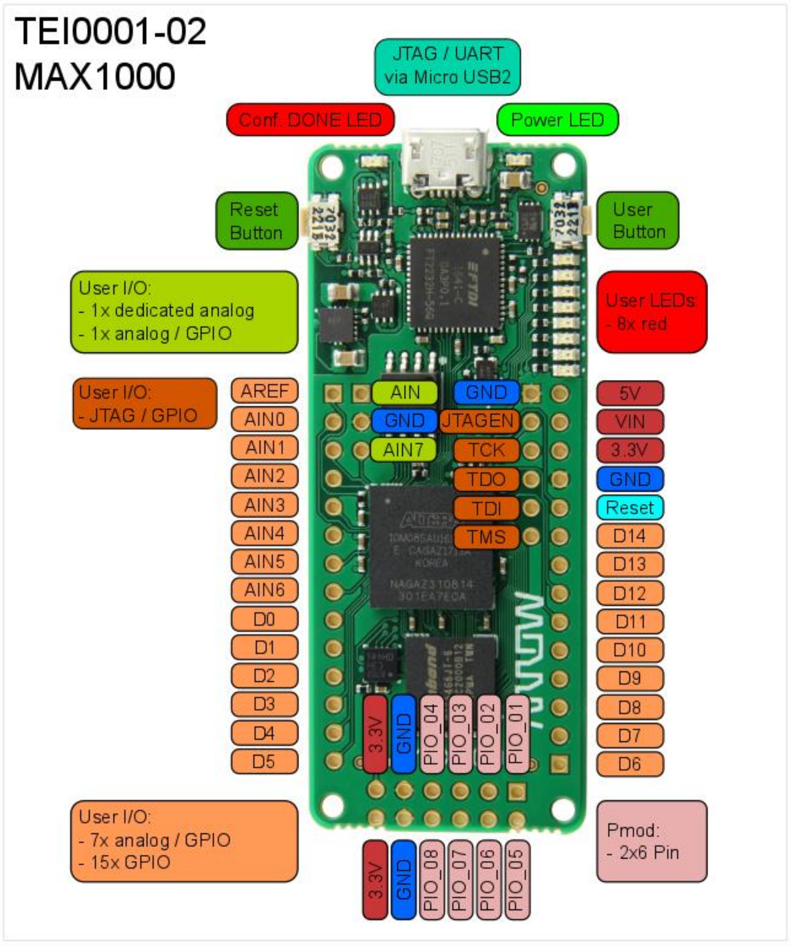
```
inst_Programmer_inteface : entity work.PI
        generic map(
                gc_bd     => gc_bd,
                gc_clk_fz => gc_clk_fz
        )
        port map(
                pil_clk            => sl_clk_c0,
                pil_reset          => sl_nrst,
                pil_rx             => pil_rx,
                pol_en_PI_ExtPg    => sl_PI_ISP,
                pov_char_out       => sv_char_out
        );
```

4. Change the *gc_clk_fz* frequency to 35 MHz in the generic declaration of the pearl_top entity.

6.2.4 I/O Pin Assignment

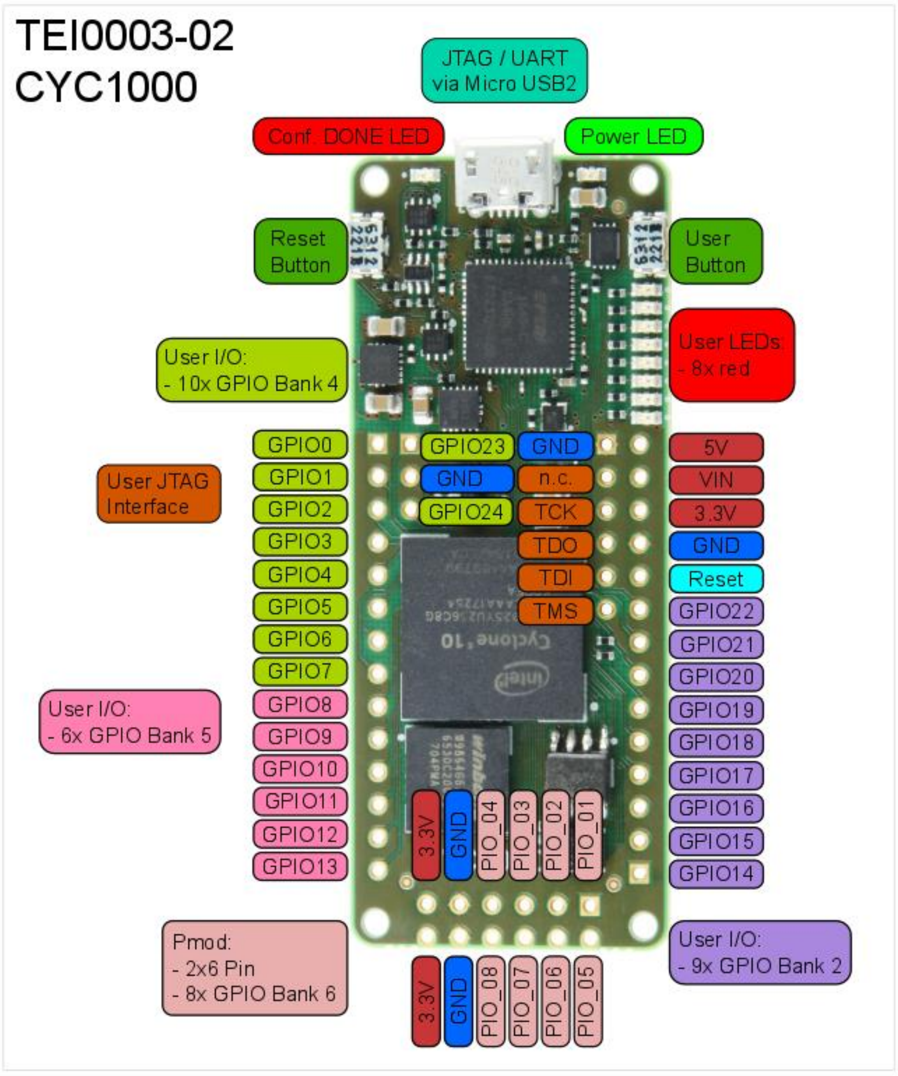
The following table provides the *Pin Planner* for the microcontroller which includes the I/O pin assignments and the I/O standards used.

MAX1000 Evaluation board:



Node name	Direction	Board location	Fitter location	I/O standard
Pil_reset	Input	User Button	E6	3.3 Schmitt Trigger
Pil_clk_12MHz	Input	CLK12M	H6	3.3 - LVTTL
Pil_sel	Input	PIO_07	K2	3.3 - LVTTL
Pil_run_prog	Input	PIO_05	N3	3.3 - LVTTL
Pil_rx	Input	PIO_06	N2	3.3 - LVTTL
Pov_OR <0>	Output	LED1	A8	3.3 - LVTTL
Pov_OR <1>	Output	LED2	A9	3.3 - LVTTL
Pov_OR <2>	Output	LED3	A11	3.3 - LVTTL
Pov_OR <3>	Output	LED4	A10	3.3 - LVTTL
Pov_OR <4>	Output	LED5	B10	3.3 - LVTTL
Pov_OR <5>	Output	LED6	C9	3.3 - LVTTL
Pov_OR <6>	Output	LED7	C10	3.3 - LVTTL
Pov_OR <7>	Output	LED8	D8	3.3 - LVTTL
Pbv_P1 <0>	Bidir	D1	K10	3.3 - LVTTL
Pbv_P1 <1>	Bidir	D2	H5	3.3 - LVTTL
Pbv_P1 <2>	Bidir	D3	H4	3.3 - LVTTL
Pbv_P1 <3>	Bidir	D4	J1	3.3 - LVTTL
Pbv_P1 <4>	Bidir	D5	J2	3.3 - LVTTL
Pbv_P1 <5>	Bidir	D6	L12	3.3 - LVTTL
Pbv_P1 <6>	Bidir	D7	J12	3.3 - LVTTL
Pbv_P1 <7>	Bidir	D8	J13	3.3 - LVTTL
Pbv_P0 <0>	Bidir	AIN0	E1	3.3 - LVTTL
Pbv_P0 <1>	Bidir	AIN1	C2	3.3 - LVTTL
Pbv_P0 <2>	Bidir	AIN2	C1	3.3 - LVTTL
Pbv_P0 <3>	Bidir	AIN3	D1	3.3 - LVTTL
Pbv_P0 <4>	Bidir	AIN4	E3	3.3 - LVTTL
Pbv_P0 <5>	Bidir	AIN5	F1	3.3 - LVTTL
Pbv_P0 <6>	Bidir	AIN6	E4	3.3 - LVTTL
Pbv_P0 <7>	Bidir	D0	H8	3.3 - LVTTL

CYC1000 Evaluation board:



Node name	Direction	Board location	Fitter location	I/O standard
Pil_reset	Input	User Button	N6	3.3 - LVTTL
Pil_clk_12MHz	Input	CLK12M	M2	3.3 - LVTTL
Pil_sel	Input	PIO_07	B16	3.3 - LVTTL
Pil_run_prog	Input	PIO_05	D15	3.3 - LVTTL
Pil_rx	Input	PIO_06	C15	3.3 - LVTTL
Pov_OR <0>	Output	LED1	M6	3.3 - LVTTL
Pov_OR <1>	Output	LED2	T4	3.3 - LVTTL
Pov_OR <2>	Output	LED3	T3	3.3 - LVTTL
Pov_OR <3>	Output	LED4	R3	3.3 - LVTTL
Pov_OR <4>	Output	LED5	T2	3.3 - LVTTL
Pov_OR <5>	Output	LED6	R4	3.3 - LVTTL
Pov_OR <6>	Output	LED7	N5	3.3 - LVTTL
Pov_OR <7>	Output	LED8	N3	3.3 - LVTTL
Pbv_P1 <0>	Bidir	D1	L15	3.3 - LVTTL
Pbv_P1 <1>	Bidir	D2	L16	3.3 - LVTTL
Pbv_P1 <2>	Bidir	D3	K15	3.3 - LVTTL

Pbv_P1 <3>	Bidir	D4	K16	3.3 - LVTTL
Pbv_P1 <4>	Bidir	D5	J14	3.3 - LVTTL
Pbv_P1 <5>	Bidir	D6	N2	3.3 - LVTTL
Pbv_P1 <6>	Bidir	D7	N1	3.3 - LVTTL
Pbv_P1 <7>	Bidir	D8	P2	3.3 - LVTTL
Pbv_P0 <0>	Bidir	AIN0	R12	3.3 - LVTTL
Pbv_P0 <1>	Bidir	AIN1	T13	3.3 - LVTTL
Pbv_P0 <2>	Bidir	AIN2	R13	3.3 - LVTTL
Pbv_P0 <3>	Bidir	AIN3	T14	3.3 - LVTTL
Pbv_P0 <4>	Bidir	AIN4	P14	3.3 - LVTTL
Pbv_P0 <5>	Bidir	AIN5	R14	3.3 - LVTTL
Pbv_P0 <6>	Bidir	AIN6	T15	3.3 - LVTTL
Pbv_P0 <7>	Bidir	D0	N16	3.3 - LVTTL

6.2.5 Design Synthesis and summary

The design should compile without any errors and critical warnings for both the evaluation boards. The warnings that are present can be ignored as well. The device utilisation and timing summary for both the evaluation boards are shown below.

Device utilisation summary for CYC1000:

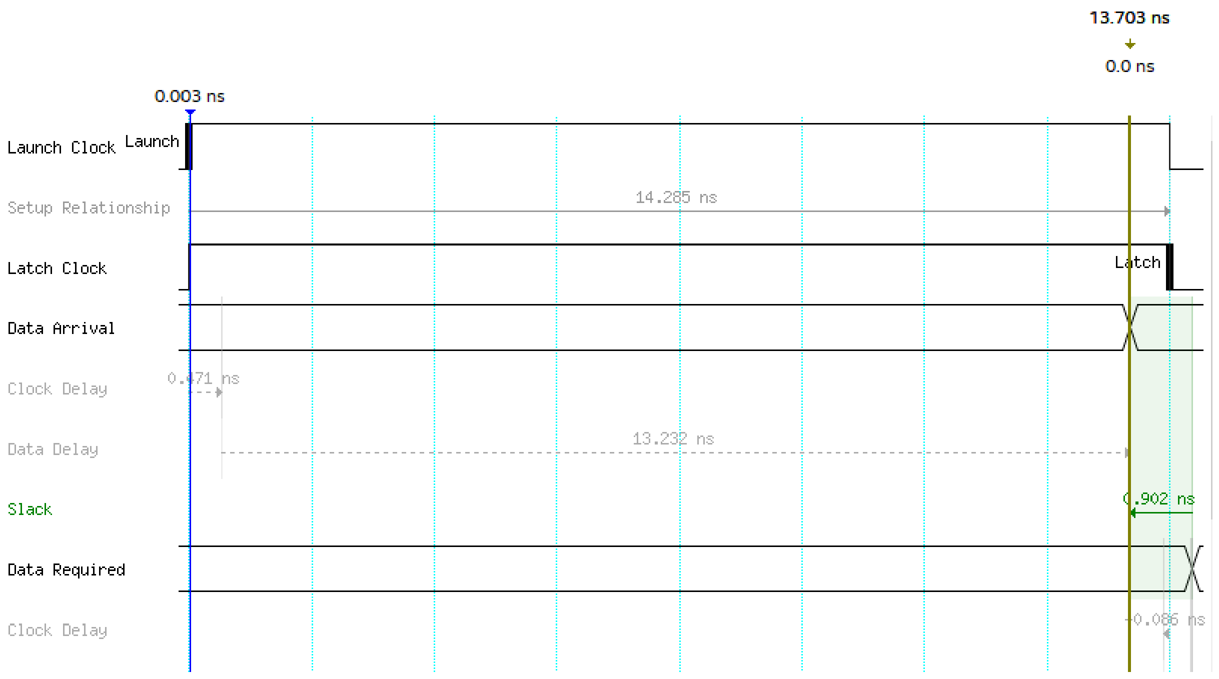
Selected device: 10CL025YU256C8G

Total logic elements	7205	out of 24,624	29 %
Total registers	846		
Total pins	29	out of 151	19 %
Total virtual pins	0		
Total memory bits	5120	out of 608,256	< 1 %
9 bit elements	0	out of 132	0 %
Total PLLs	1	out of 4	25 %

Timing Summary:

Maximum Clock Frequency $Fmax = 37.36\ MHz, Restricted\ Fmax = 37.36\ MHz.$

The setup waveform for signals is given below.



Device utilisation summary for MAX1000:

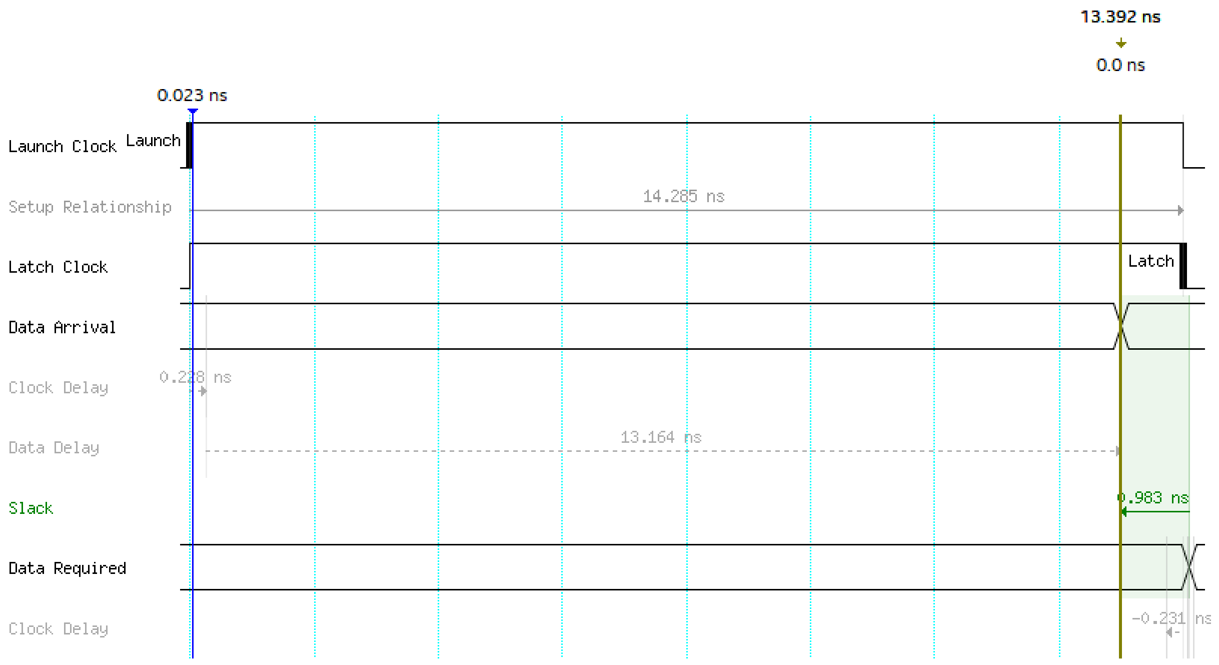
Selected device: 10M16SAU169C8G

Total logic elements	7297	out of 15,840	46 %
Total registers	846		
Total pins	30	out of 130	23 %
Total virtual pins	0		
Total memory bits	5120	out of 562,176	< 1 %
9 bit elements	0	out of 90	0 %
Total PLLs	1	out of 1	100 %

Timing Summary:

Maximum Clock Frequency $Fmax = 37.59\ MHz$, $Restricted\ Fmax = 37.59\ MHz$.

The setup waveform for signals is given below.



Part III

7. Assembly Language

In *Part II* we have finally built our microcontroller design in FPGA using VHDL. The microcontroller can store up to 256 words of machine codes. Even though it is possible to manually upload the machine code of any program, it would be very tedious and slow to deal with 0s and 1s to program the microcontroller. Thus, we would need to develop an assembly language that would provide the mnemonic codes for the machine code instruction which would make programming faster and easier for the end user. However, the assembly language program that would be written in the host computer by the user must be translated to machine code by a tool called the *assembler*. In *Part III* we will create our own assembly language along with its assembler and create a basic *graphical user interface (GUI)* application for the microcontroller using python.

7.1 Custom assembler design

As discussed earlier, an assembler is used to convert an assembly language program to its equivalent object code. The input to an assembler is a source code written in assembly language using mnemonic codes and the output is the object code which in our case is the *.hex* file containing the machine code for the microcontroller. The flowchart for the assembler is shown below.

Source file → read file → Assemble → Form machine code → Form Hex file → Deliver hex file

7.1.1 Scanning and parsing

The basic design of an assembler depends on the architecture of the microcontroller and the instruction sets that it supports. For instance, for the *LDA $200* instruction, the assembler must convert the opcode LDA to its machine code which is *00000001 (0x01)* as well as the operand $200 which converts to *11001000 (0xC8)*. In general, the assembler must perform the following tasks:

1. Scanning the source file (tokenizing)
2. Assigning machine addresses to labels
3. Handle assembler directives such as .WORD which tell the assembler to load a string into the RAM
4. Parsing (validating the instructions)
5. Create a machine coded table from the microcontroller *Instruction set*

The assembler performs all the tasks in different stages of the process. The stages are defined below:

***Stage 1*:**

The first and the most important stage for an assembler is to create the symbol lists and forward references which includes assigning addresses to the instruction by creating a list, separating opcodes and optional operands and storing them in their corresponding list, storing labels and their corresponding addresses in a subroutine list, creating lists of constants declared by the user for referencing and finally creating lists that holds addresses and data for assembler directives which in our case is the *.WORD* directive.

Stage 2:

The second stage of an assembler is to translate the information stored in the lists that were created in the first stage to machine code for the microcontroller. The assembler would also use references of symbols and constants that were declared in the first stage for decoding the data and addresses.

7.1.1.1 Assembler stage 1 algorithm template

The algorithm for Stage 1 of the assembler is shown below.

```
Start
  Open file
    code_length = 0
    for each instruction:
      remove whitespaces from beginning and the end
      if the line is not empty and is not commented:
        if the first character is not a start of direct addressing*:
          code_length += 1
          if comment lines are present at the end and the line does not
          include .word directive:
            remove comment lines from the end
          if the line is a constant declaration and does not include .word
          directive:
            add the constant in the constant list
            code_length -= 1
            go back to the start of the loop and read the next line
          if the line has ‘:’ indicating a label and does not include .word
          directive:
            add the label in the subroutine list along with the address
            if instruction immediately follows the label:
              save the instruction
            else:
              code_length -= 1
              go back to the start of the loop and read the next line
          if the line includes .word directive:
            save the string in a variable
            if .word directive contains parameters (address, length):
              add the starting address in descending order and the length
              of the string in constant list
```

```
   convert each character of the string stored in the variable
   to ASCII and add it in the word list
   code_length -= 1
   go back to the start of the loop and read the next line
if the line includes ‘(‘ indicating constant operand value
operation:
   perform operations depending on the symbols used and concat
   the result with the opcode
if the line includes ‘”’ indicating the constant operand to be
and ASCII character:
   convert the character to ASCII and concat the result with the
   opcode
store the opcode present in the mnemonic code
if the opcode is mov or add or sub or push or pop or mul or div
or rda or addi or subi or jb or jnb or addc or addic or subb or
or subib:
   split the mnemonic code with ‘ ‘ or ‘,’ as a separator
   if the words of the mnemonic code match with any of the
   constant names already declared in the constant list:
      replace the words that matches with the names with the
      values of the constants
   if the operand of the instruction does not include ‘x’
   indicating hexadecimal number, ‘$’ indicating integer
   number and ‘b’ indicating binary number and the mnemonic code
   does not include either ‘jb’ or ‘jnb’:
      replace all the white spaces present in the mnemonic code
      which makes it a single instruction which does not take any
      operand values
   else:
      sperate the opcode from the operand
   if the mnemonic code includes ‘p0.’ and does not include
   either ‘jb’ or ‘jnb’:
      replace the value (n) that follows the ‘p0.’ with the
```

```
            exponential value $2^n$
          if the **mnemonic code includes** ‘p1.’ **and does not include**
          **either ‘jb’ or ‘jnb’**:
            replace the value (n) that follows the ‘p1.’ with the
            exponential value $2^n$
        if the line **includes ‘”’** after the previous conditional
        statement:
          convert the character to ASCII and concat the result with the
          opcode
        if the **opcode is inc** or **dec** or **djnz**:
          replace the whitespaces present in the opcode
        if the **opcode is anl** or **orl** or **cpl**:
          replace all the whitespaces of the mnemonic code which makes
          it a single instruction
          if the **opcode is cpla**:
             concat 0xff at the end of the mnemonic code
        if the **opcode is set** or **clr**:
          if the **operand** is **not t0e, t0, tr0, tcr0e** and **tc0l**:
            if the **operand** is **in constant list**:
              replace the name with the value of the constant
            replace the value (n) with the exponential value $2^n$
         else:
           replace all the whitespaces of the mnemonic code which makes
           it a single instruction
      add the **opcode** in **opcode list**
      add the **operand** in the **operand list**
    if the first character **is ‘[‘**:
      separate the address and the data and convert them into binary
      add the address in **direct_address list**
      add the data in the **direct_data list**
   if the instruction has no operand then add **“NO OPER”** in **operand list**
 close file and End
```

7.1.1.2 Assembler stage 2 algorithm template

The algorithm for Stage 2 of the assembler is shown below.

```
Start
  for each address in range of code_length:
    convert the integer address to 16-bit binary and store it in variable
    byte_address
    add the lower byte to address_low list
    add the upper byte to address_high list
    if $byte\_address^{th}$ element of opcode list match with the pre-defined
    CODE_DICTIONARY:
      store the corresponding binary representation in variable
      temp_machine_code
    if $byte\_address^{th}$ element of operand list is not “NO OPER”:
      if $byte\_address^{th}$ element of operand list exists in constant list:
        store the corresponding value in variable temp
        if the first character of temp is ‘x’ indicating hexadecimal number
        or ‘$’ indicating integer or ‘b’ indicating binary or ‘”’ indicating
        ASCII character:
          replace the value in temp with the corresponding binary
          representation
      else if $byte\_address^{th}$ element of operand list exists in subroutine
      list:
        concat temp_machine_code with the corresponding binary value (in
        this case is the address location of the subroutine)
      else if the first character of temp is ‘x’ indicating hexadecimal
      number or ‘$’ indicating integer or ‘b’ indicating binary or ‘”’
      indicating ASCII character:
        replace the value in temp with corresponding binary representation
    else:
      concat temp_machine_code with 00000000
    add first 8 characters of temp_machine_code to machinecode_high list
    add next 8 characters of temp_machine_code to machinecode_low list
End
```

7.1.2 Assembler report file

After building the machine code and address lists, the assembler creates a report file that handles any of the exceptions that might have occurred while building the lists. Exceptions such as undefined instructions or wrong data types used while declaring constants are handled by the assembler. This is done by using the **try** and **except** statements of python. For instance, during string to integer conversion, the assembler expects all integer literals inside the string. If instead the assembler receives a character, an exception would occur. This exception can be handled, and an error message can be reported to the programmer.

The report file contains the following information:

1. Date and time stamps at the time of compiling the program
2. Total code memory used by the program in bytes and percentage
3. Opcodes and their corresponding addresses without any comment lines
4. Microcontroller memory map showing the machine codes including the address location both in binary and hexadecimal
5. The contents of the *.hex* file

The report file is created with the same name as the source file and is saved in the working directory.

7.1.3 Program .hex file

The program .hex file is generated by the assembler once the source code has successfully compiled without any errors. The format of the hex file is described in *section 4.2.16*. Some of the rules that were followed while creating the hex file is shown below.

1. The maximum *Byte count* of each line or segment of the hex file is 16
2. ' ' is used as a separator between bytes that are written in the hex file

The template to form the *hex file* from the *machine code list* is shown below.

```
Start
  segments = compute number of segments (lines) with $ceil\,(code\_length * 2/16)$
  declare a segment_bytecount list
  for x in range of segments:
    if more than one segment (more than 16 bytes) is present:
      add 16 to the segment_bytecount list
    else:
      add the number of remaining bytes to the segment_bytecount list
  create and open a new hex file with the source file name
  for segment_index in range of segments (lines):
    write to the hex file the number of bytes in the line
    (segment_bytecount[segment_index]), and base_address (8 * segment_index)
    of the line
    write to the hex file each byte from the machinecode_high list followed
    by machinecode_low list
  if direct_address_list and world is not empty:
    write to the hex file the corresponding bytes from each list
  close hex file
End
```

Part IV

8. Example programs

In this chapter we would program the microcontroller with the custom assembler that we have created in the previous chapter. The programs would be tested mostly on *MAX1000 evaluation board* because of its smaller dimensions. The schematics and board connections would also be provided.

8.1 LED blink

The LED blink program turns the 8 on-board LEDs which are connected to the *output port* of the microcontroller ON for one second and then OFF for one second repeatedly.

8.1.1 Schematic diagram

The board schematic is shown below.

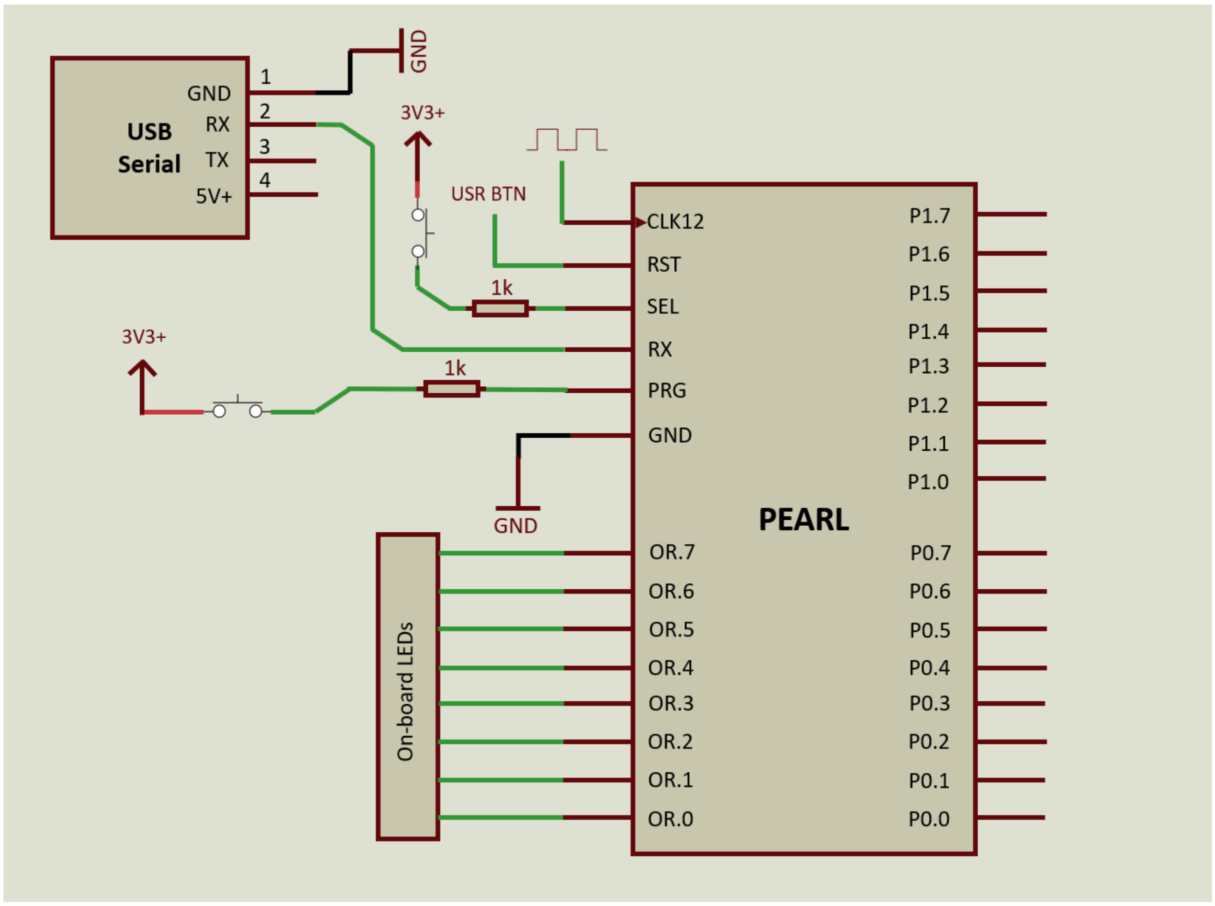


Figure 8.1.1-1 Blink LED Schematic

8.1.2 Assembly code

```
;;; fsys = 35 MHz

compare_th0 = x85
```

```
        compare_tl0 = x84
        prescale_1024 = $4
__init__:
        ;;; set Timer 0 for 1 sec delay
        ;;; compare value = fsys / (1 * 1024)
        mov th0 compare_th0
        mov tl0 compare_tl0
        mov cs0 prescale_1024
        set t0e
        mov r0 xff  ;; R0 = 1111 1111
blink:
        mov a, r0   ;; A = R0
        cpl a        ;; complement A i.e. 0000 0000 and so on...
        out          ;; Display result of A into OUTPUT Port built in LEDs
        mov r0, a    ;; store the current value of A in R0
        call delay   ;; add delay
        jmp blink

delay:
        set tr0      ;; start Timer
loop:
        jnbtf0 loop  ;; wait till 1 sec
        clr t0       ;; stop and clear timer
        ret
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 15 bytes, 5.86%
```

8.1.3 Result

After uploading the code in the microcontroller, the 8 onboard LEDs turns ON and OFF every second.

LEDs OFF:

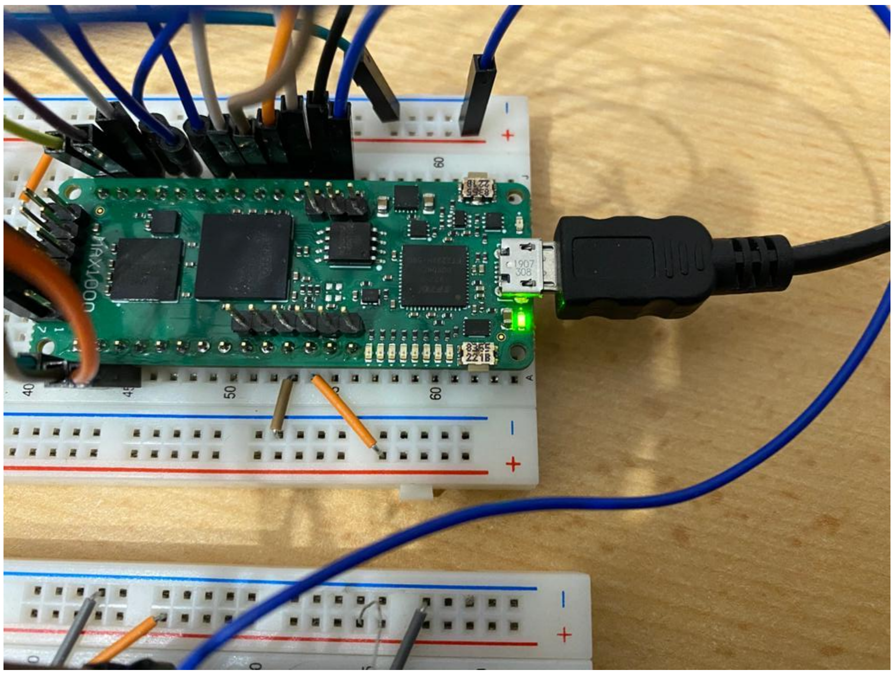


Figure 8.1.3-1 Onboard LEDs OFF

LEDs ON:

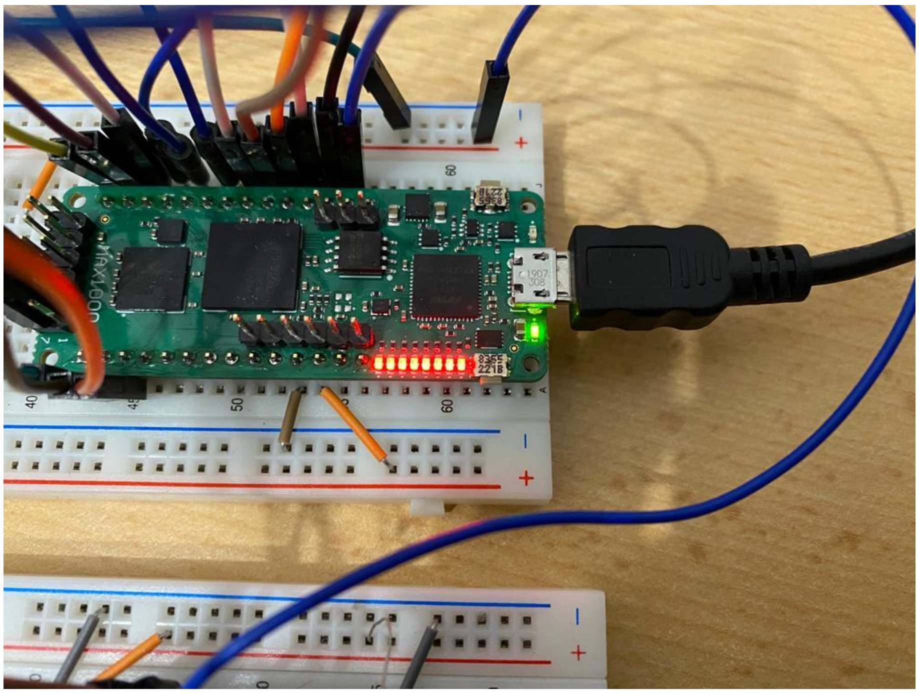


Figure 8.1.3-2 Onboard LEDs ON

8.2 Toggle LED with button

The Toggle LED button program toggles the LED connected at *user port P0.2* whenever the button at *user port P0.6* is pressed. To remove button bouncing, a delay of *200ms* is added between each button reading phases.

8.2.1 Schematic diagram

The board schematic is shown below.

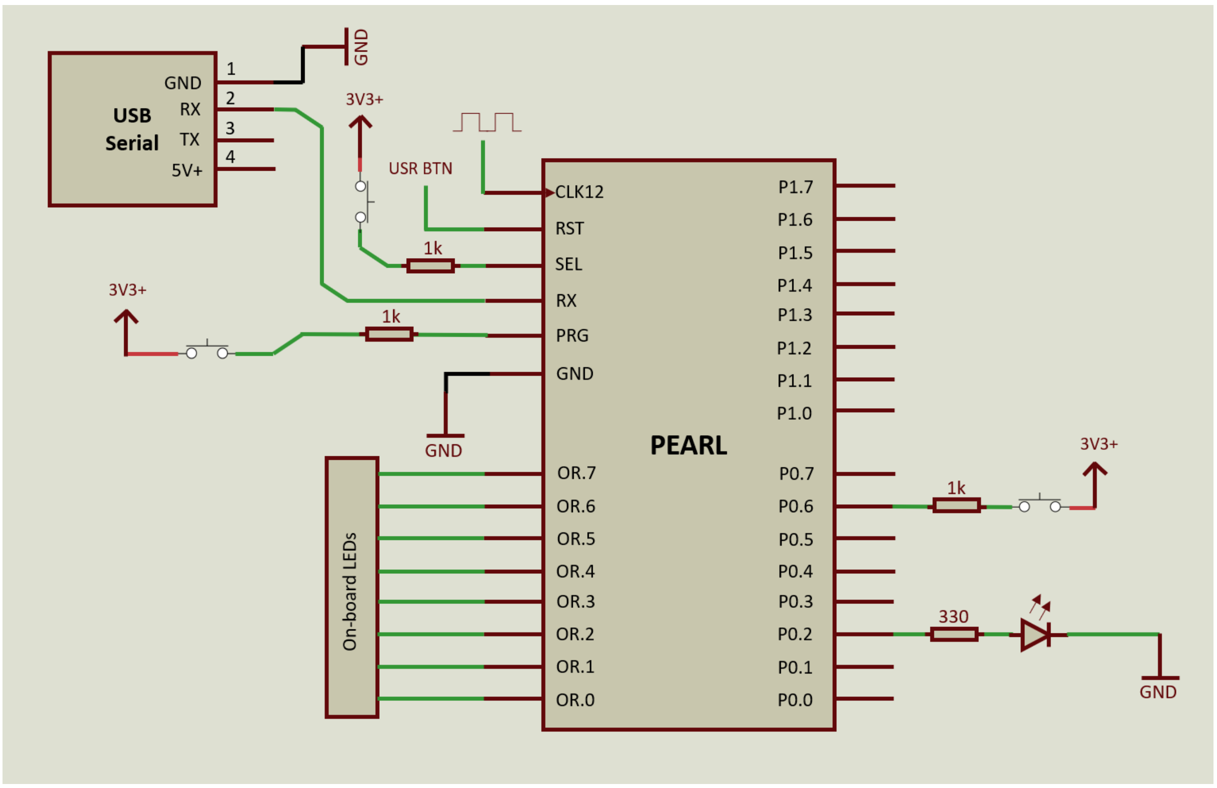


Figure 8.2.1-1 Toggle LED Schematic

8.2.2 Assembly code

```
        LED = P0.2

        btn = P0.6


        ;; fsys = 35 MHz

        compare_th0 = x35

        compare_tl0 = x68

        prescale_512 = $3

        btn_pin = x40

        LED_STATE = r0


__inti__:

        ;; set Timer 0 for 200ms delay
```

```
      mov th0 compare_th0
      mov tl0 compare_tl0
      mov cs0 prescale_512
      set t0e

      ddrp0 btn_pin     ;; set pin 6 of Port0 as input
      mov LED_STATE $0  ;; intial LED state is 0; OFF
wait:
      jb btn toggle
      jmp wait          ;; wait till button is pressed
toggle:
      mov a, LED_STATE  ;; load the LED state to A register
      cmpi $0           ;; compare with 0
      jmz setLed        ;; check if zero flag is set
clrLed:
      mov LED_STATE $0  ;; set LED state to 0
      clr LED           ;; LED OFF
      call debounce     ;; add debounce delay
      jmp wait
setLed:
      mov LED_STATE $1  ;; set LED state to 1
      set LED           ;; LED ON
      call debounce     ;; add debounce delay
      jmp wait
debounce:
      set tr0       ;; start Timer
delay:
      jnbtf0 delay  ;; wait for 200 ms
      clr t0        ;; stop and clear timer
      ret
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 24 bytes, 9.38%
```

8.2.3 Result

After uploading the code in the microcontroller, when the button at P0.6 is pressed, the external LED connected at P0.2 toggles.

LED ON with button press:

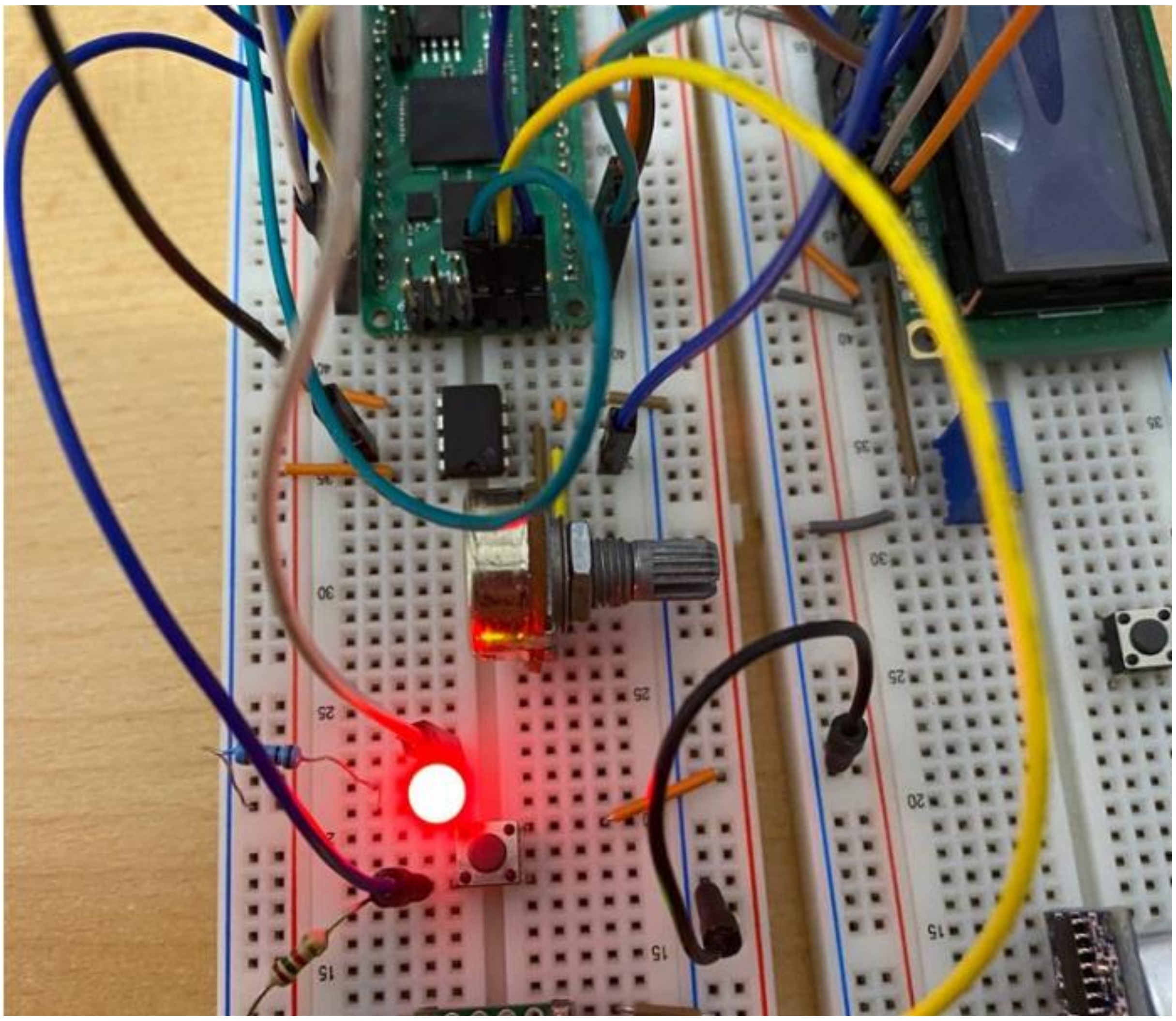


Figure 8.2.3-1 External LED ON

LED OFF with button press:

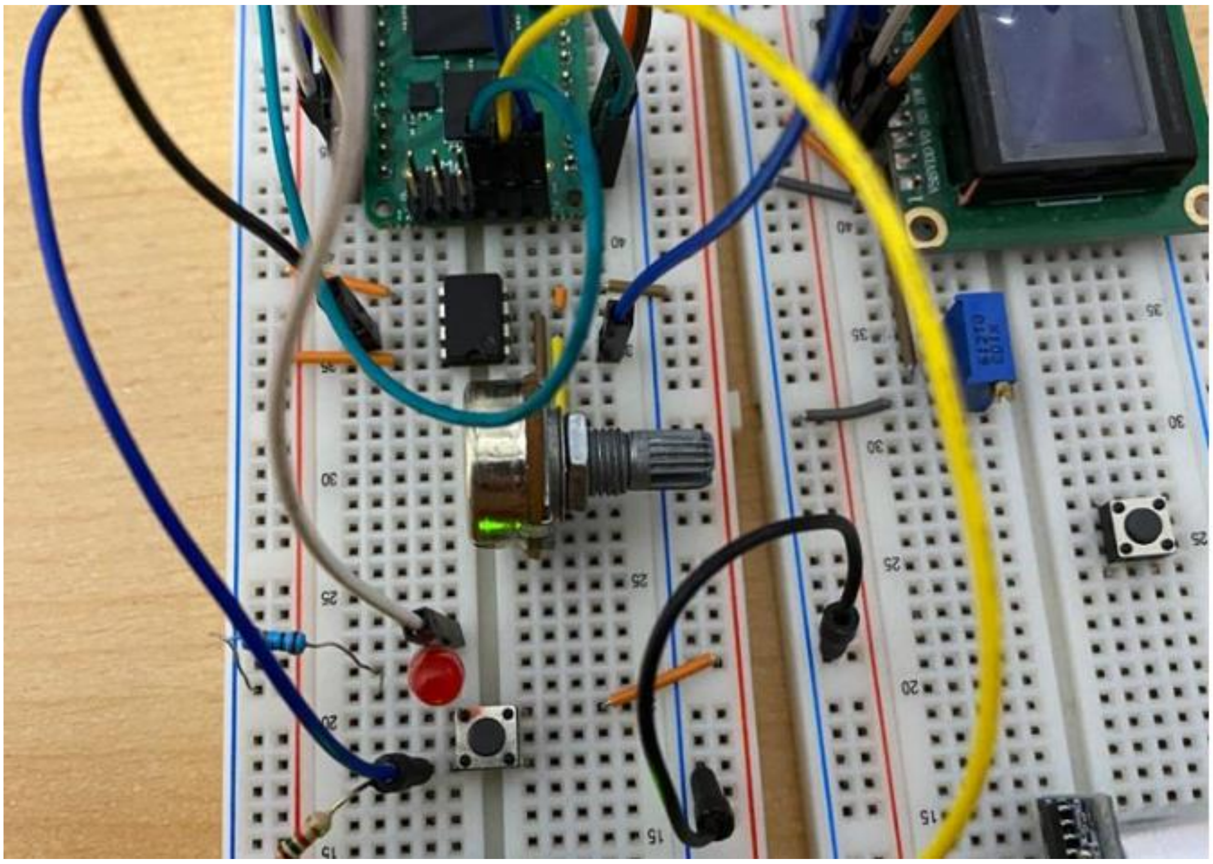


Figure 8.2.3-2 External LED OFF

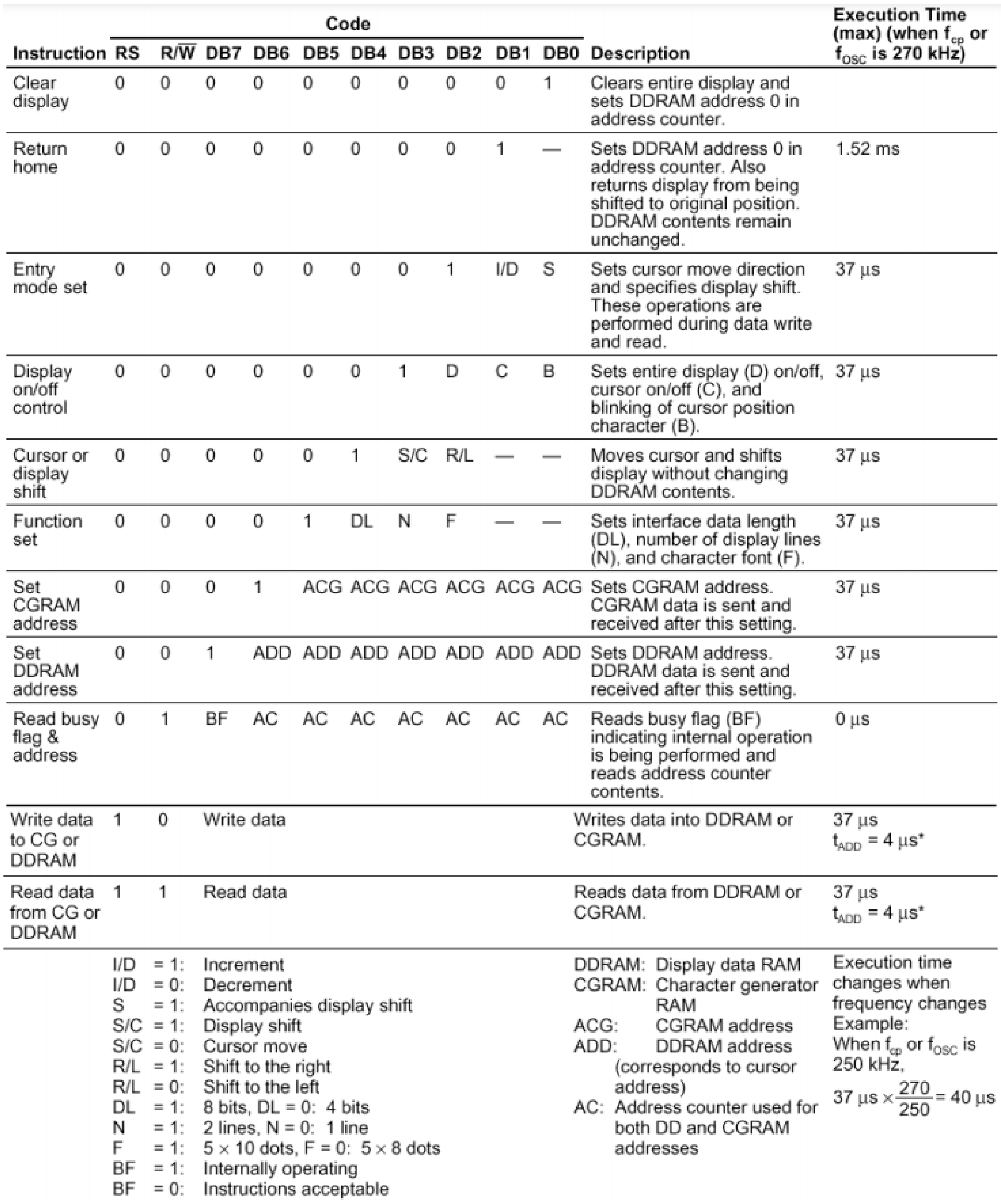
8.3 Controlling 1602 LCD

One of the most commonly used displays with microcontroller projects is the 1602 character-type LCD module based on HD44780U controller. It can display wide range of alphanumeric characters along with programmable symbols. The addition of the controller makes the LCD module easy to program as the programmer must control only the *RW, RS* and *E* signals and data while the actual displaying of the characters are taken care by the display controller module. The pin description of the LCD module is shown below.

Pin Symbol	Description
VSS	Power supply ground
VDD	Positive power supply. Ideally +5V
VO	Contrast setting. 4.7k to 10k resister pulled to ground
RS	**User control**, Logic 1 : LCD data mode (for write and read) Logic 0 : LCD instruction mode
RW	**User control**, Logic 1 : Read data from LCD Logic 0 : Write data to LCD
E	**User control**, LCD enable. A high to low pulse executes instruction
D0	**User control**, Data 0
D1	**User control**, Data 1
D2	**User control**, Data 2
D3	**User control**, Data 3
D4	**User control**, Data 4

D5	**User control**, Data 5
D6	**User control**, Data 6
D7	**User control**, Data 7
A	Anode: positive back light. Ideally +5V with 1k resister in series
K	Cathode: Connected to ground

The LCD instruction table is shown below.



	Code											Execution Time
Instruction	RS	$R/\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	(max) (when f_{cp} or f_{OSC} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*

I/D	= 1:	Increment
I/D	= 0:	Decrement
S	= 1:	Accompanies display shift
S/C	= 1:	Display shift
S/C	= 0:	Cursor move
R/L	= 1:	Shift to the right
R/L	= 0:	Shift to the left
DL	= 1:	8 bits, DL = 0: 4 bits
N	= 1:	2 lines, N = 0: 1 line
F	= 1:	5 × 10 dots, F = 0: 5 × 8 dots
BF	= 1:	Internally operating
BF	= 0:	Instructions acceptable

DDRAM:	Display data RAM
CGRAM:	Character generator RAM
ACG:	CGRAM address
ADD:	DDRAM address (corresponds to cursor address)
AC:	Address counter used for both DD and CGRAM addresses

Execution time changes when frequency changes
Example:
When f_{cp} or f_{OSC} is 250 kHz,

$$37\ \mu s \times \frac{270}{250} = 40\ \mu s$$

Figure 8.3-1 LCD Instruction table

Depending upon the available number of free pins in the microcontroller, the 1602 LCD module can be used either in 8-bit mode or in 4-bit / half-byte mode. In this example we would discuss both modes.

8.3.1 LCD 8-bit mode

In 8-bit mode, all the Data pins from D0-D7 are used to interface with the LCD module. Data and instructions are sent to the LCD module 8-bit at a time, thus requires less time and less lines of code to program the LCD module. As seen from *Figure 8.3-1*, during LCD initialisation i.e., when the *Clear display* and *Return home* instructions are used, the LCD requires a lot of time to execute the instructions. However, polling and reading the *Busy Flag (BF)* eliminates the need of Timer 0.

8.3.1.1 Schematic diagram

The board schematic is shown below.

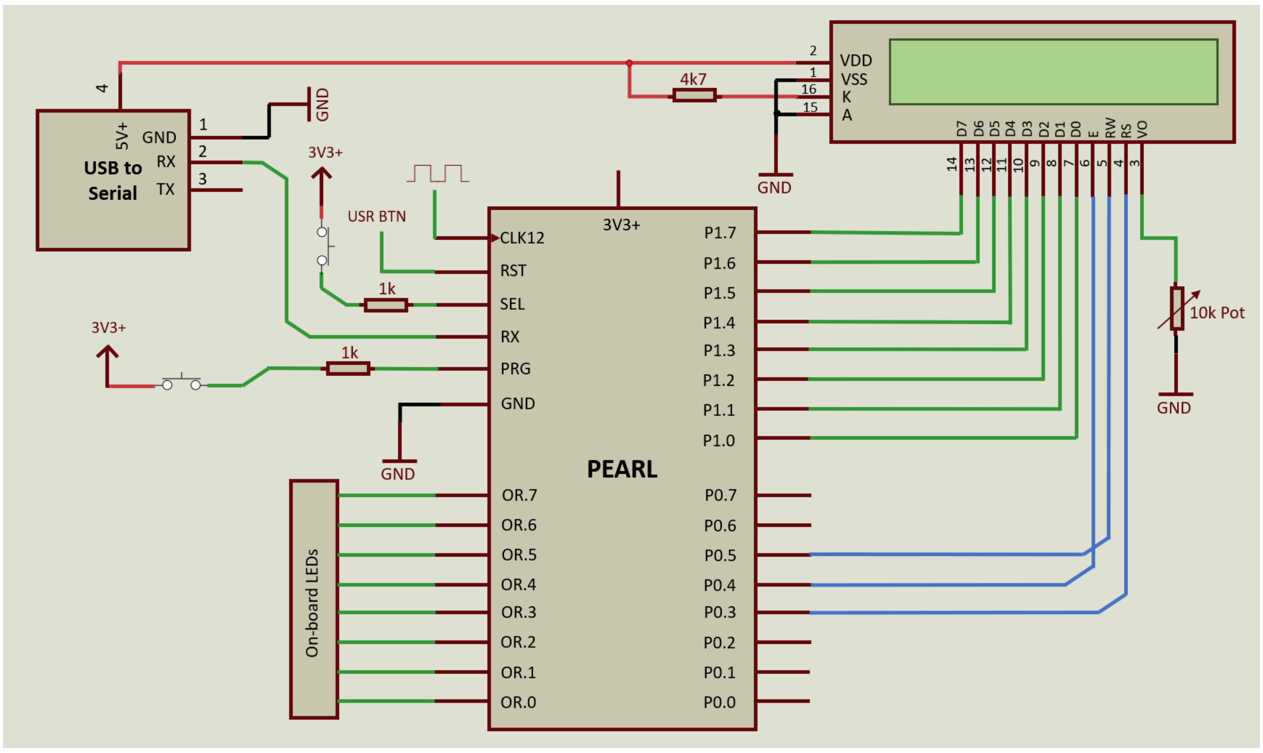


Figure 8.3.1.1-1 16x2 LCD module in 8-bit mode Schematic

8.3.1.2 Assembly code

```
;;;;;;       1602 LCD Display control; Reading the BF     ;;;;;;
;;; RS --> P0.3 E --> P0.4 R/W --> P0.5  D7 - D0 --> P1  ;;;



DATA = P1

LCD_RS = P0.3

LCD_E = P0.4

LCD_RW = P0.5


LCD_BF = P1.7

BF_PIN = x80


my_word:

      .word @ ADDRESS, DATA_LEN "Hello, world!"

display_setup:

      DDRP0 $0

      DDRP1 $0                  ;; set initial direction to outputs


      MOV P0 $0                 ;; which makes LCD_RS '0' i.e ready to send

                                ;; instructions to the display

      MOV R2 x30                ;; full byte mode Data Pins are D7 - D0

      CALL set_instruction

      MOV R0 ADDRESS         ;; store the address of first character i.e. 'H'

      MOV R1 (DATA_LEN - 1)  ;; store lenght of string - 1

      MOV R2 $1             ;; clear display

      CALL set_instruction

      MOV R2 $12            ;; display on, cursor off, blink off

      CALL set_instruction

send_characters:

      SET LCD_RS            ;; write data to LCD

      MOV A @R0             ;; load A with value from address pointed by R0

      MOV R2, A
```

```
        CALL set_data
        DEC R0 $1               ;; R0 has the address of the next character of the
                                ;; string
        DJNZ R1, send_characters  ;; check if all charactes are sent; if not then
                                  ;; continue sending
        CLR LCD_RS
        HLT
set_instruction:
set_data:
        MOV A, R2
        MOV DATA, A
execute_instruction:
        SET LCD_E          ;; E HIGH
check_BF:
        CLR LCD_RS         ;; RS LOW --> Instruction mode
        SET LCD_RW         ;; R/W HIGH  --> Read mode
        DDRP1 BF_PIN       ;; Setting D7/BF as input to read the busy flag
lcd_busy:
        JB LCD_BF lcd_busy ;; BF = 1 --> LCD is busy ;; BF = 0 --> LCD finished
executing
        DDRP1 $0           ;; Set all bits as output
        CLR LCD_E          ;; E LOW, i.e. finished execution cycle
        CLR LCD_RW         ;; R/W LOW --> Write mode
        RET
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 44 bytes, 17.19%
```

8.3.1.3 Result

The above program when executed, prints “8-bit Mode!!” on the LCD screen.

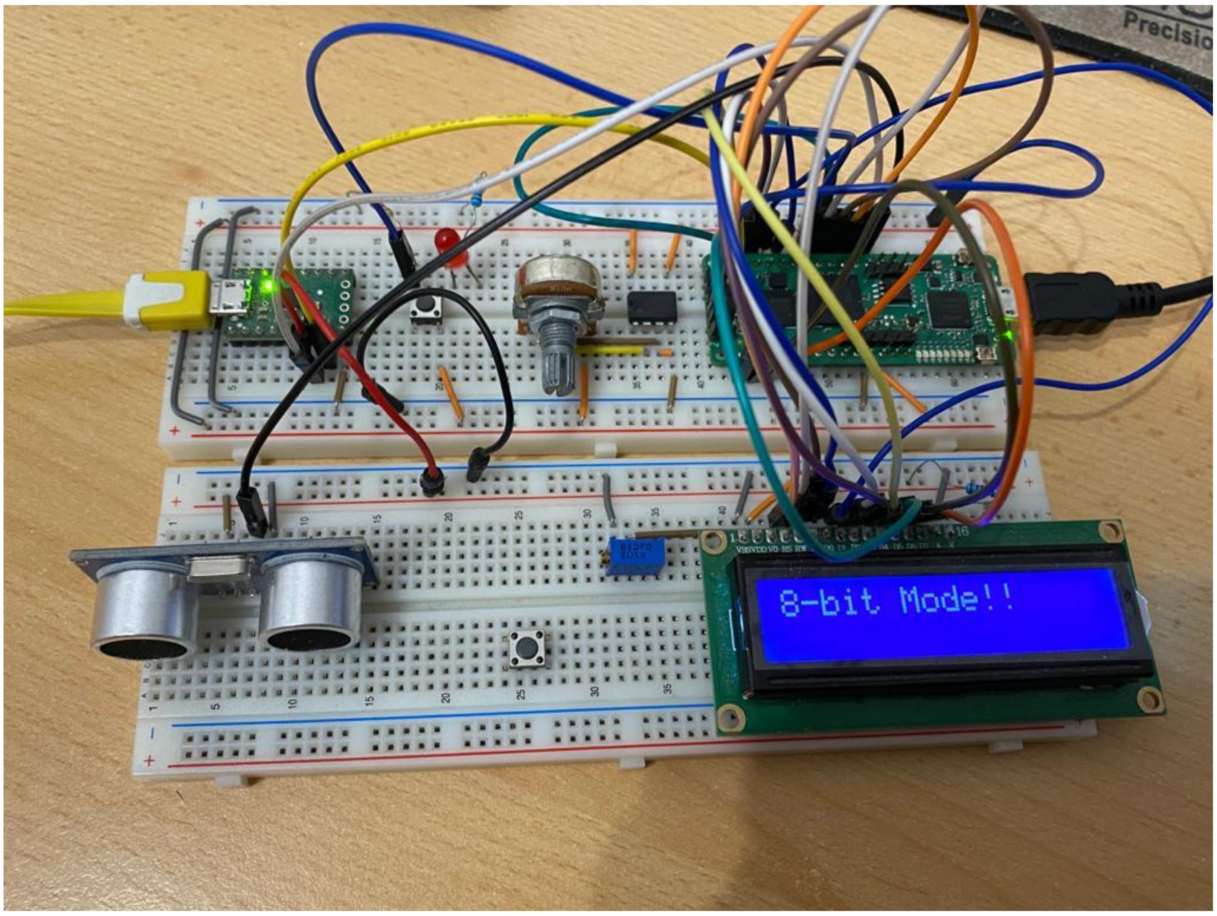


Figure 8.3.1.3-1 16x2 LCD module operated in 8-bit mode

8.3.2 LCD Half-byte mode

In 4-bit / half-byte mode, only 4 data pins from D4 to D7 are used to interface with the LCD module. The data and instructions are sent 4-bit at a time where the Most significant nibble is sent first followed by the Least significant nibble. The advantage of using the LCD module in half-byte mode is that only 4 bits of data pins are used by the microcontroller leaving the rest of the pins to control other hardware peripherals. However, programming the LCD in half-byte mode is relatively complicated and requires many lines of codes. The operation steps and commands are shown below.

Initialisation for 4-bit mode

1. Wait for 15ms

RS	RW	D7	D6	D5	D4
0	0	0	0	1	1

2. Wait for at least 5ms

RS	RW	D7	D6	D5	D4
0	0	0	0	1	1

3. Wait for at least 5ms

RS	RW	D7	D6	D5	D4
0	0	0	0	1	1

4. Wait for at least 5ms

RS	RW	D7	D6	D5	D4
0	0	0	0	1	0

Operations using 4-bit mode

5. Wait at least 100µs or poll Busy Flag

RS	RW	D7	D6	D5	D4
0	0	0	0	1	0

RS	RW	D7	D6	D5	D4
0	0	N	F	X	X

This command sets number of lines and font size of the display

6. Wait for 100µs or poll Busy Flag

RS	RW	D7	D6	D5	D4
0	0	0	0	0	0

RS	RW	D7	D6	D5	D4
0	0	0	0	0	1

This command clears the screen.

7. Wait for 5ms or poll Busy Flag

RS	RW	D7	D6	D5	D4
0	0	0	0	0	0

RS	RW	D7	D6	D5	D4
0	0	1	1	0	0

This command turns the display ON, cursor OFF and blink OFF.

8. Wait for 5ms or poll Busy Flag and Write Data to the LCD module.

8.3.2.1 Schematic diagram

The board schematic is shown below.

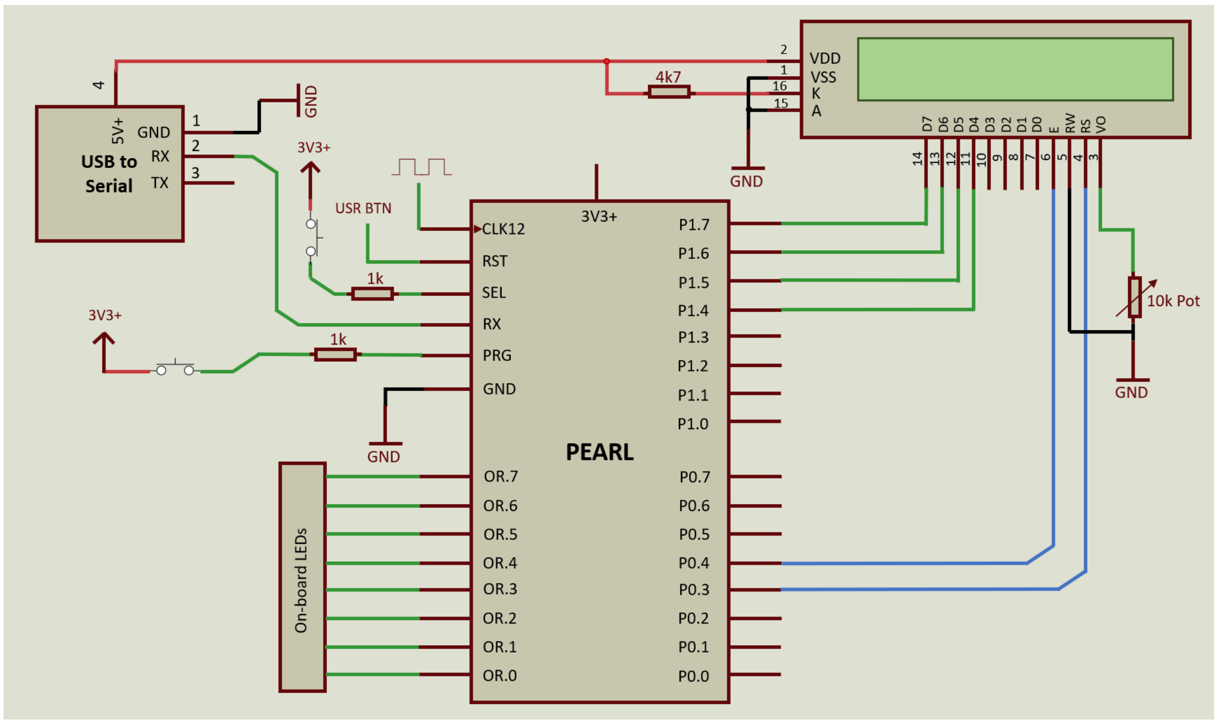


Figure 8.3.2.1-1 16x2 LCD module in half-byte mode Schematic

8.3.2.2 Assembly code

```
;;;;; controlling 1602 using half byte mode       ;;;;;

;;; RS --> P0.3 E --> P0.4. D7 to D4 --> P1.7 to P1.4. ;;;


;;; fsys = 35 MHz

compare_th0 = x20

compare_tl0 = x0b

prescalse_64 = $1


DATA = P1

LCD_RS = P0.3

LCD_E = P0.4
```

```
my_word:
      .word @ my_data, data_len "Hello, world!"
__init__:
      ;; set Timer 0 for 15 ms delay
      MOV TH0 compare_th0
      MOV TL0 compare_tl0
      MOV CS0 prescalse_64
      SET T0E
      DDRP0 $0
      DDRP1 $0
      MOV P0 $0            ;; makes LCD_RS '0' i.e ready to send instructions to
                           ;; the display
      MOV R0 my_data       ;; R0 stores the address of my_data
      MOV R1 (data_len - 1) ;; stores the lenght of my_data
      CALL delay
display_set_up:
      MOV R2 x30                 ;; 8 bit mode
      CALL set_instruction
      CALL delay
      CALL set_instruction
      CALL delay
      CALL set_instruction
      CALL delay
      MOV R2 x20            ;; half byte mode
      CALL set_instruction
      CALL delay
      ;;; function set command
      MOV R2 x20               ;; Function set command : 1 line, 5x8 Font size
      CALL send_instructions
      MOV R2 x01               ;; clear display
      CALL send_instructions
      MOV R2 x0c               ;; display ON, cursor off, blink off
      CALL send_instructions
data_write_Mode:
       SET LCD_RS   ;; write data to LCD
send_characters:
       MOV A @R0                  ;; write data to LCD
       MOV R2, A                  ;; load A with value from address pointed by R0
       CALL set_data
       CALL shift_LSB
       CALL set_data
       DEC R0 $1                 ;; R0 has the address of the next character of the
                                 ;; string
       DJNZ R1, send_characters   ;; check if all charactes are sent; if not then
                                 ;; continue sending
       CLR LCD_RS
       HLT
send_instructions:
       CALL set_instruction
       CALL shift_LSB
       CALL set_instruction
       RET
set_instruction:
set_data:
       ;; mask lower 4 bits of Port 1
       PUSH R2
       PUSH R1
       MOV R2A $0
       MOV A, DATA
       MOV R1, A
       MOV R1B $0
       MOV A, R2
       ORL R1
       MOV DATA, A
       POP R1
       POP R2
execute_instruction:
       SET LCD_E
delay:
        SET TR0
loop:
        JNBTF0 loop
        CLR T0
        CLR LCD_E
        RET
shift_LSB:
        MOV R3 $3
shift:
        MOV A, R2
        MOV C $0
        RLC
        MOV R2, A
        DJNZ R3, shift
        RET
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 77 bytes, 30.08%
```

8.3.2.3 Result

The above program prints “4-bit Mode!!” on the LCD screen.

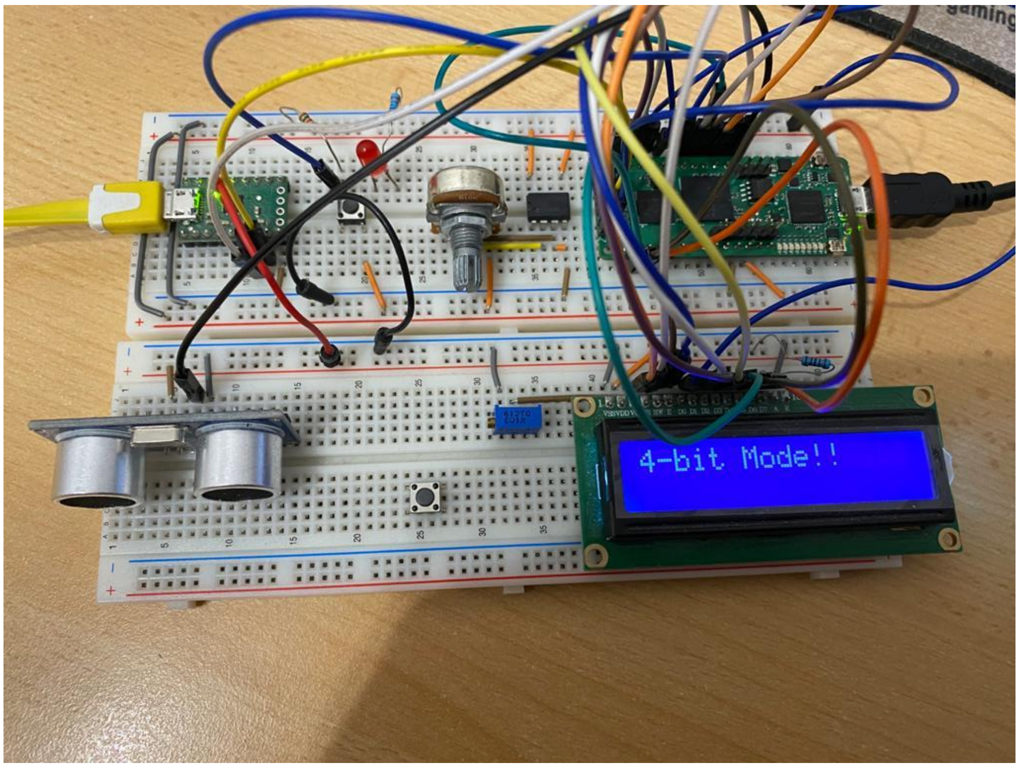


Figure 8.3.2.3-1 16x2 LCD module operated in half-byte mode

8.4 Serial 0831 A/D Converter

The microcontroller architecture does not feature any A/D conversion capabilities on its own. Thus, to measure sensor readings that uses analog signals, it is necessary to use an external A/D converter. Although any A/D converter could be used with the microcontroller, it is recommended to use serial A/D converters as it would require fewer pins to interface with the microcontroller instead of parallel A/D converters which uses 8 or more data pins depending on the resolution of the device.

The *ADC0831 series converters* are 8-bit successive approximation A/D converters with serial I/O data channel. The pin description of the ADC is shown below.

Pin Symbol	Description
n CS	**User control**, Active low chip select pin to enable the ADC
Vin (+)	Analog input positive
Vin (-)	Analog input ground
GND	ADC common ground
VCC	ADC + 3V to 5V
CLK	**User control**, ADC 10kHz to 400kHz clock during serial interface
D0	**User control**, Serial data channel 0
Vref	Reference voltage for ADC, ideally +3V to 5V

When the CS pin of the A/D converter is pulled low, the analog to digital conversion starts. The setup time or the time needed by the A/D converter to convert the analog input value to a valid 8-bit digital value is 250ns. After which the serial data at D0 can be sampled at every falling edge of the CLK line. The ADC D0 channel sends the data bits MSB first. The Timing diagram of ADC0831 is shown below.

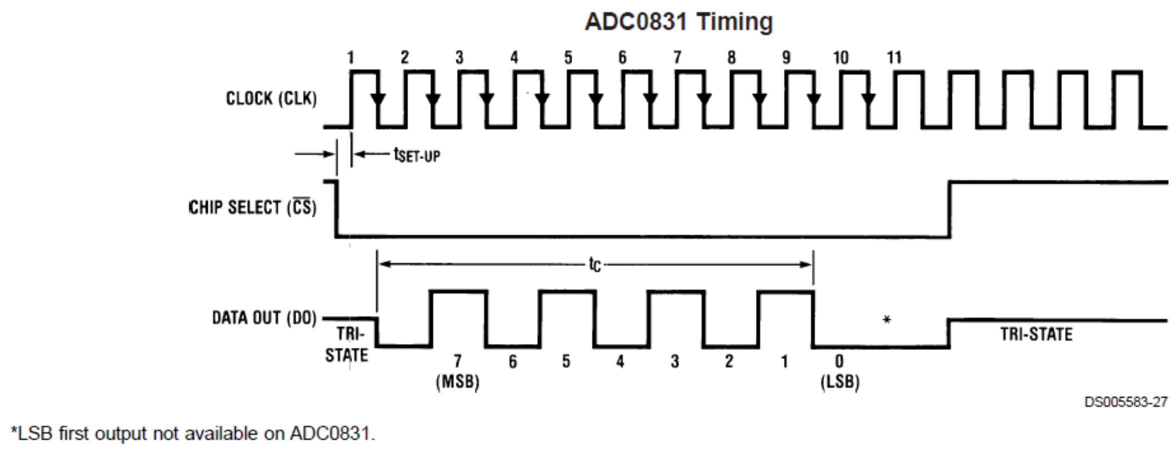


Figure 8.4-1 0831 Serial ADC timing diagram

8.4.1 Schematic diagram

The board schematic is shown below.

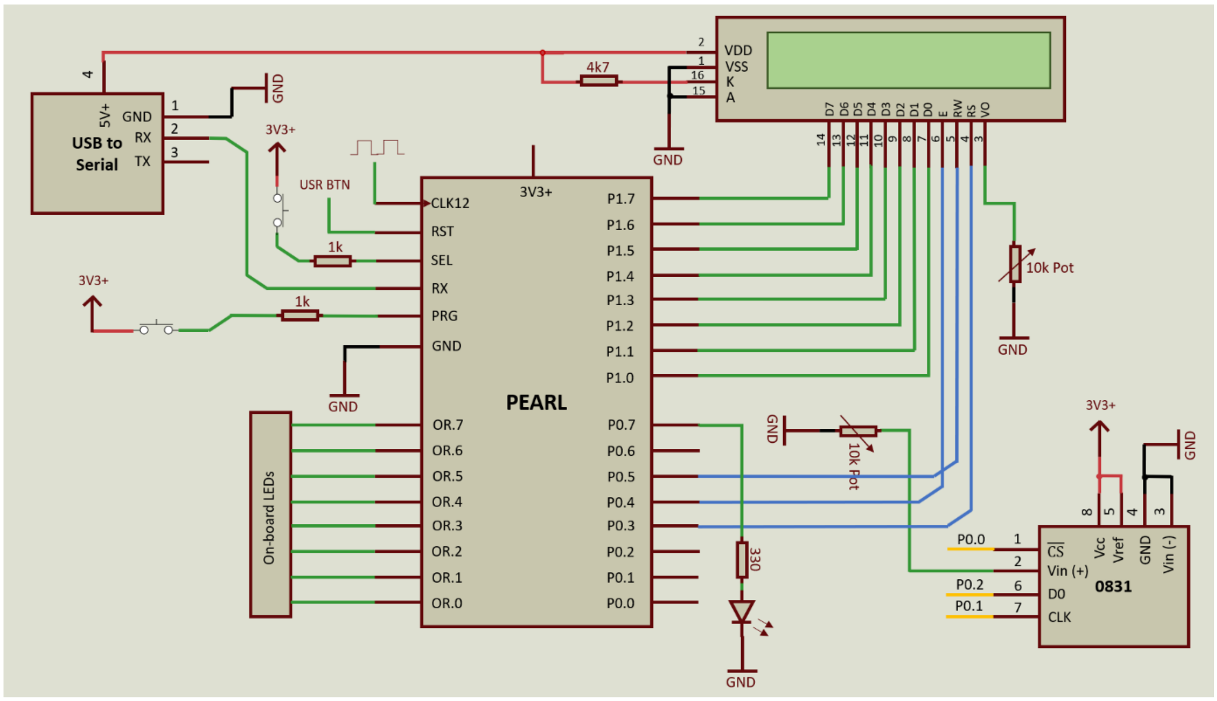


Figure 8.4.1-1 0831 Single channel serial ADC Schematic

8.4.2 Assembly code

```
;;        ADC reading and displaying in 1602 lcd
;; RS --> P0.3 E --> P0.4 D7 - D0 --> P1
;; ADC_CS = P0.0, ADC_CLK = P0.1, ADC_D0 = P0.2


digit_9 = $57       ;; ascii value of digit 9
decimal_numbers = $9
digit_len_counter_address = $200
;;; fsys = 35 MHz
delay_th0 = x0a
delay_tl0 = xae
prescale_64 = $1
;;; fsys = 35 MHz 10 µs delay
ADC_TH0 = x01
ADC_TL0 = x5e
```

```
ADC_CS0 = $0
ADC_read_delay = $20

ADC_nCS  = P0.0
ADC_CLK = P0.1
ADC_DO  = P0.2

;;
LED = P0.6
THRESHOLD = $210

;; Display
LCD_RS = P0.3
LCD_E = P0.4
LCD_RW = P0.5
LCD_BF = P1.7
BF_BIT = b10000000

[$200 : $0]
[$210 : $60]      ;; set threshold to value

__init__:
      ;; set Timer 0 for 5 ms delay
      MOV TH0 delay_th0
      MOV TL0 delay_tl0
      MOV CS0 prescale_64
      SET T0E
      MOV R1 $10
      DDRP1 $0
      DDRP0 $4
      CALL set_registers
set_display:
      MOV R2 x30         ;; full byte mode Data Pins are D7 - D0
      CALL set_instruction
      CLR LCD_RS      ;; LCD instruction mode
      MOV R2 $1           ;; clear display,
      CALL set_instruction
return_home:
      CLR LCD_RS
      MOV R2 $2           ;; cursor return home
      CALL set_instruction
      MOV R2 $12          ;; R2 stores the value of the data pins D7 - D0 12 -->
                          ;; sets display, blink off, cursor off
      CALL set_instruction
      SET LCD_RS      ;; LCD write mode
      SET ADC_nCS     ;; disable ADC
read_ADC:
start_conversion:
      ;; set Timer 0 for 5 µs delay
      MOV TH0 ADC_TH0
      MOV TL0 ADC_TL0
      MOV CS0 ADC_CS0
      SET T0E


      PUSH R0
      PUSH R1
      MOV R0A $8               ;; number of bits that must be received
      MOV R1 $0                ;; R1 stores the ADC value
      CLR ADC_nCS              ;; enable ADC
      CALL delay
      CALL pulse_clk
read_analog:
      CALL pulse_clk
      MOV A, R1                 ;; load the current ADC value to the A register
      MOV C, ADC_DO             ;; read the current value from the ADC D0 pin and
store it in Accumulator carry
      RLC                      ;; rotate A left with carry
      MOV R1, A                ;; store the result back to A
      DJNZ R0A, read_analog    ;; check if all 8 bits received
end_conversion:
      SET ADC_nCS              ;; disable ADC
      CALL delay
      MOV A, R1
      POP R1
      POP R0
      MOV TH0 delay_th0
      MOV TL0 delay_tl0
      MOV CS0 prescale_64
      SET T0E
led_controller:
      CMP THRESHOLD      ;; comparing with value stored in addres location THRESHOLD
      PUSH A
      JMC led_on          ;; if ADC value > 60 turn on LED
      CLR LED
      JMP continue
led_on:
      SET LED
continue:
      POP A
find_remainder:
      MOV DM $0
      DIV R1                 ;; extracting each digits of ADC value
      MOV R2, A              ;; store the remainder i.e. One's place of ADC value
decode_digit:
      MOV A, R2
      SUB R3
      JMZ store_number       ;; check if the digit matches with R3
      DEC R0 $1             ;; else decrement the ASCII value of and store the next
digit
      DJNZ R3, decode_digit ;; decrement the digit
stop:
      HLT
set_instruction:
      MOV A, R2
execute_instruction:
      MOV P1, A
      SET LCD_E
check_BF:
      CLR LCD_RS                ;; RS LOW --> Instruction mode
      SET LCD_RW                ;; R/W HIGH  --> Read mode
      DDRP1 BF_BIT       ;; Setting D7/BF as input to read the busy flag
lcd_busy:
      JB LCD_BF lcd_busy  ;; BF = 1 --> LCD is busy BF = 0 --> LCD finished executing
      DDRP1 $0                  ;; Set all bits to out mode
      CLR LCD_E                 ;; E LOW, i.e. finished execution cycle
      CLR LCD_RW                ;; R/W LOW --> Write mode
      RET
delay:
      SET TR0
loop:
      JNBTF0 loop
      CLR T0
      RET
store_number:
      LDA digit_len_counter_address      ;; load the number of digits in the ADC
value
      ADDI $1                           ;; increment the number of digits
      STA digit_len_counter_address      ;; store the number of digits in the ADC
value
      MOV A, R0
      PUSH A                             ;; save the ASCII values of the digits in
the stack
      MOV A, DM                         ;; load A with the result of the division
      SUBI $0
      CALL set_registers
      JMZ display_number                        ;; check if result is 0 i.e. if all
the digits of ADC value has been extracted
      MOV A, DM
      JMP find_remainder
set_registers:
      MOV R0 digit_9                                  ;;  R0  stores  ascii  value  of
digit 9
      MOV R3 decimal_numbers
      RET
display_number:
      SET LCD_RS
      POP A                         ;; retrieve the ASCII values of the digits
from the stack
      CALL execute_instruction
      LDA digit_len_counter_address
      SUBI $1
      STA digit_len_counter_address
      JMZ  read_delay                        ;;  check  if  all  the  digits  have  been
displayed
      JMP display_number
read_delay:
      MOV R2 $1
add_mask:
      SET LCD_RS
      ldi " "
      CALL execute_instruction
      DJNZ R2, add_mask
      PUSH R0
      MOV R0 ADC_read_delay
read_delay_loop:
      CALL delay
      DJNZ R0, read_delay_loop          ;; produces a reading delay of 20 * 5 ms
      POP R0
      JMP return_home
pulse_clk:
      SET ADC_CLK
      CALL delay
      CLR ADC_CLK
      RET
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 117 bytes, 45.70%
```

8.4.3 Result

The 0831 serial ADC converts the analog input coming from the 10k potentiometer to 8-bit digital value. When the digital value is equal or more than 60, the LED at P0.6 is turned on which simulates an alarm turning on.

ADC value < 60:

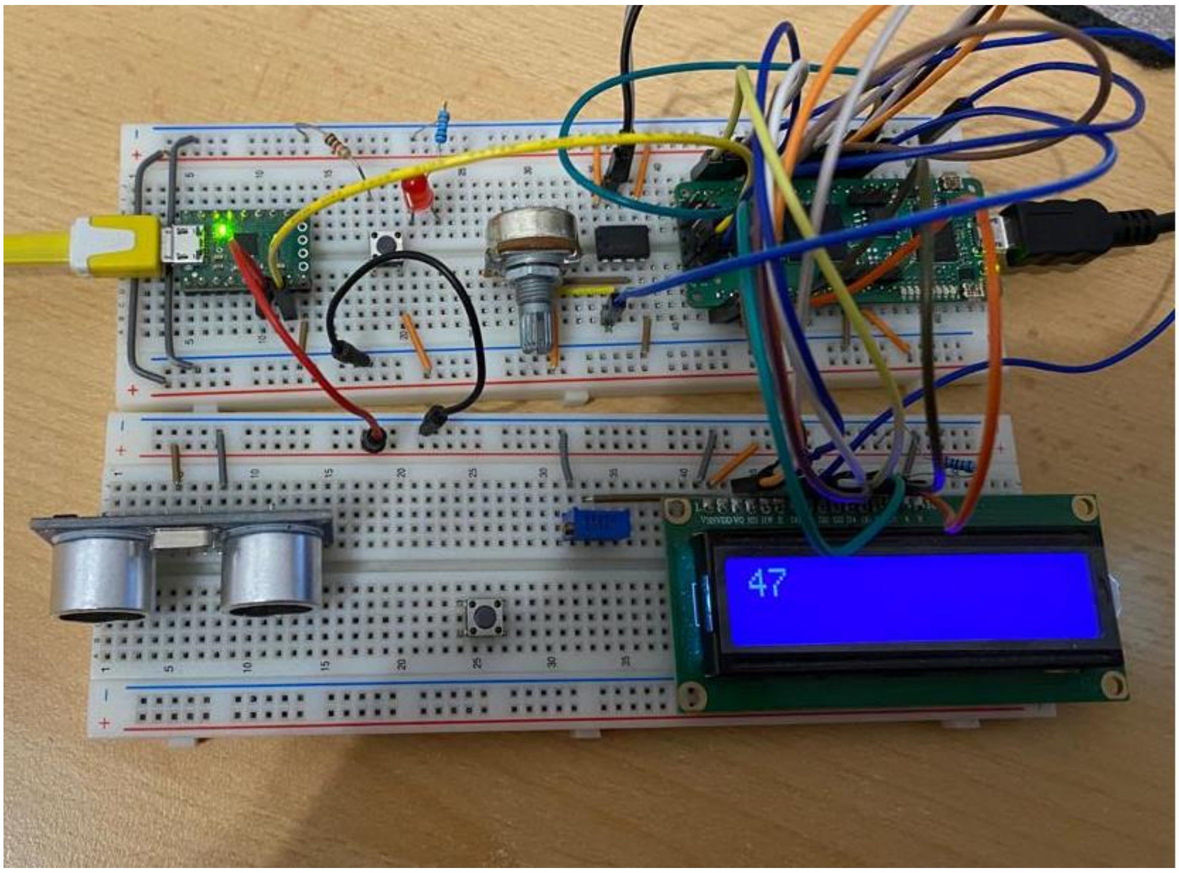


Figure 8.4.3-1 ADC output shown in LCD module

ADC value >= 60:

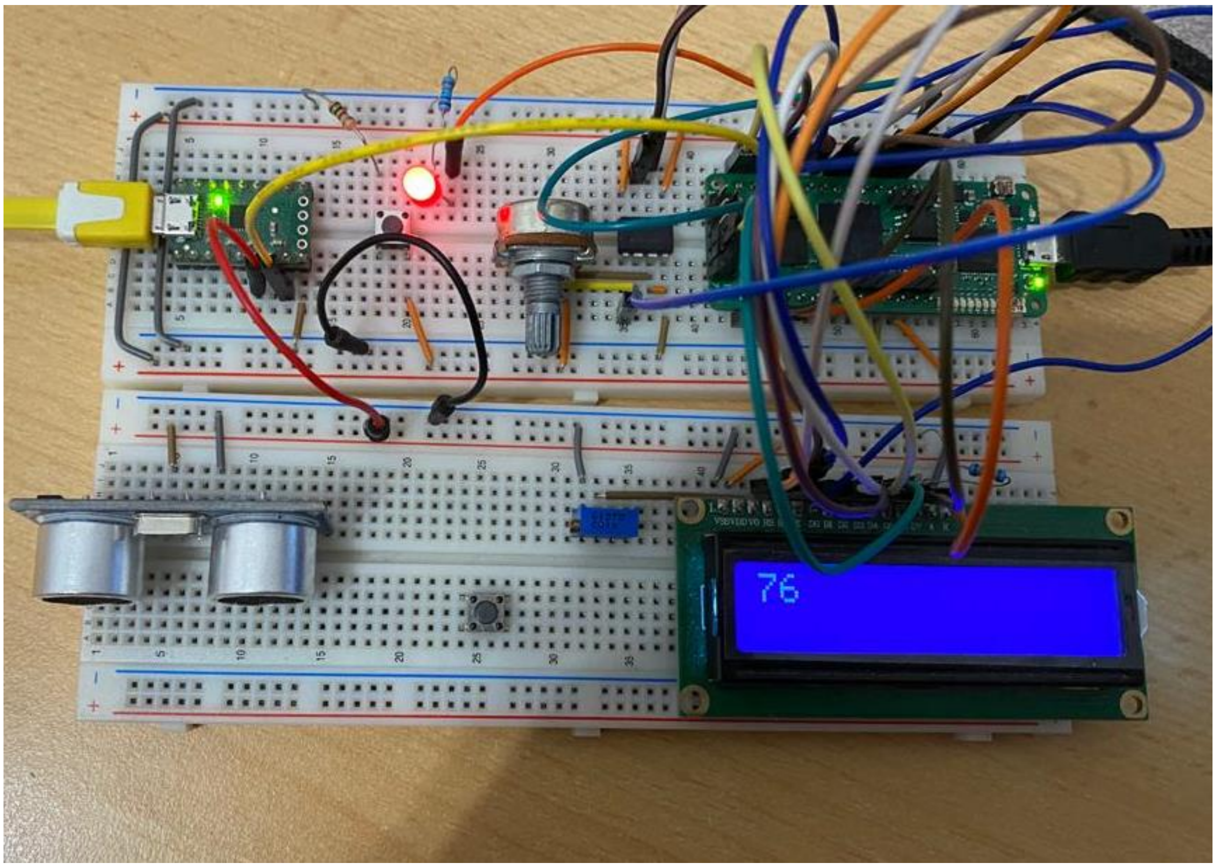


Figure 8.4.3-2 External LED turns ON when ADC value is greater or equal to 60

8.5 Distance measurement using HC-SR04 Ultrasonic sensor

The HC-SR04 ultrasonic sensor module provides 2cm to 400 cm non-contact measurements with good accuracies. The complete module includes both the ultrasonic transmitter and the receiver. The pin description of HC-SR04 ultrasonic sensor module is shown below.

Pin Symbol	**Description**
Vcc	Sensor +5V
Trig	**User control**, Trigger pin
Echo	**User control**, Echo pin
GND	Sensor ground

When a 10µs pulse is sent to the Trigger pin of the ultrasonic sensor, the transmitter module sends 8 pulses of ultrasonic sound at 40 kHz and pulls the echo pin HIGH. Upon hitting the target object, the Echo is received by the receiver module which then pulls the Echo pin LOW indicating an end of conversion. The width of the Echo pulse is proportional to the distance between the object and the sensor and can be used to calculate the approximate distance between the object and the sensor.

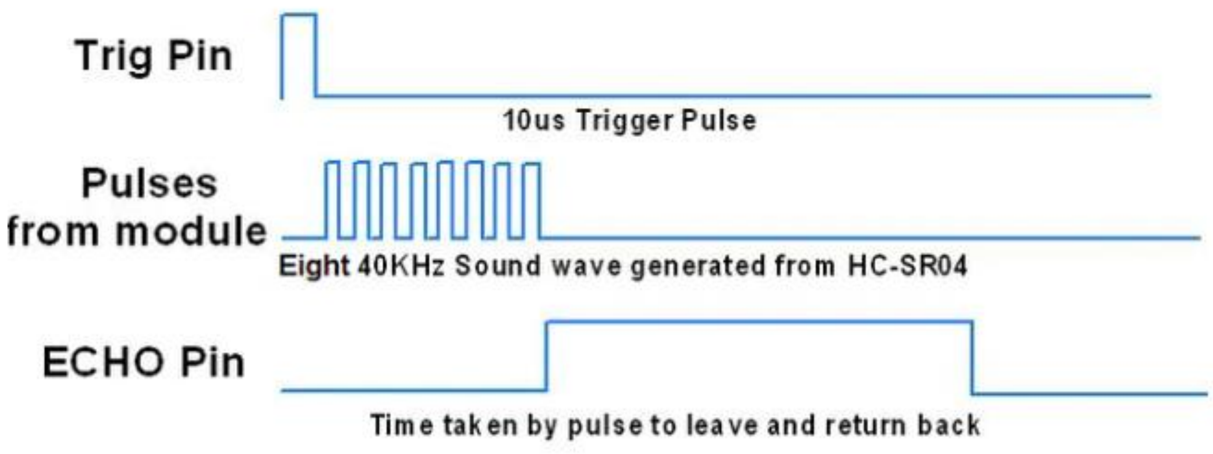

Trig Pin
10us Trigger Pulse
Pulses from module
Eight 40KHz Sound wave generated from HC-SR04
ECHO Pin
Time taken by pulse to leave and return back

Figure 8.5-2 HC-SR04 Ultrasonic sensor timing diagram

8.5.1 Schematic diagram

The board schematic is shown below.

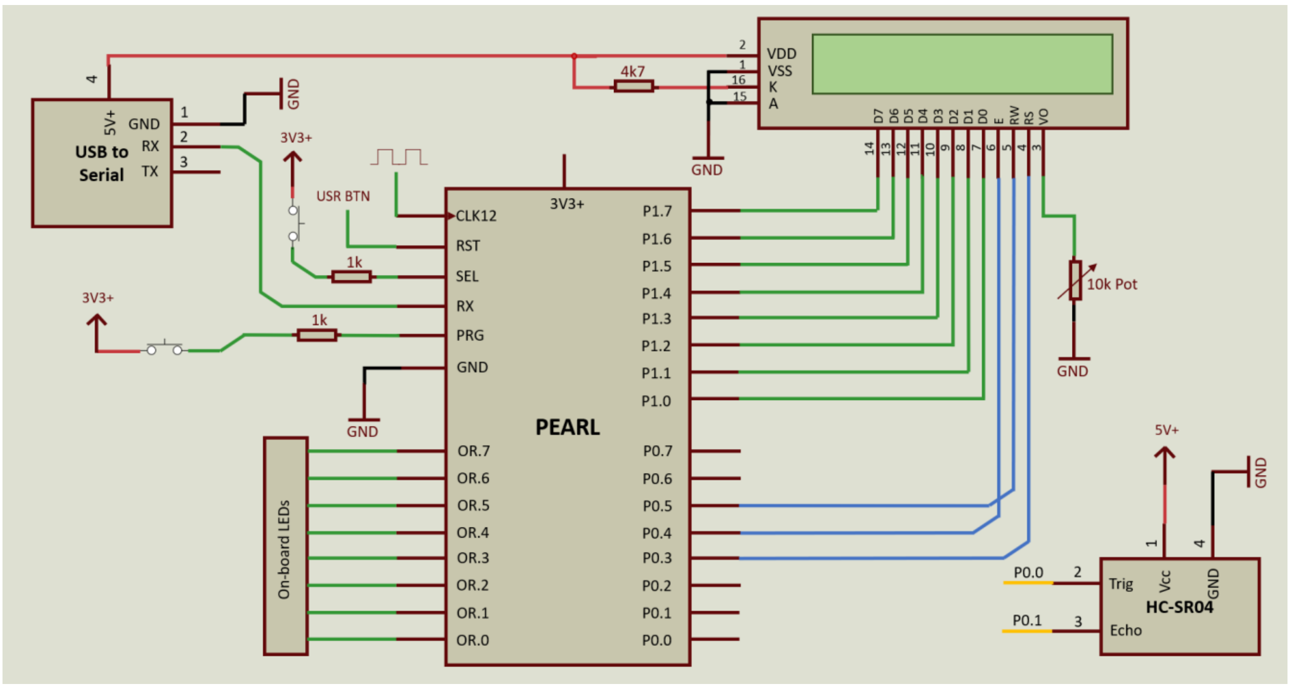

USB to Serial
5V+
GND
RX
TX
GND
3V3+
USR BTN
1k
1k
3V3+
4k7
VDD
VSS
K
A
GND
D7 D6 D5 D4 D3 D2 D1 D0 E RW RS VO
10k Pot
GND
PEARL
3V3+
CLK12
RST
SEL
RX
PRG
GND
OR.7 OR.6 OR.5 OR.4 OR.3 OR.2 OR.1 OR.0
P1.7 P1.6 P1.5 P1.4 P1.3 P1.2 P1.1 P1.0
P0.7 P0.6 P0.5 P0.4 P0.3 P0.2 P0.1 P0.0
On-board LEDs
GND
5V+
GND
P0.0
P0.1
Trig
Echo
Vcc
GND
HC-SR04

Figure 8.5.1-1 HC-SR04 Ultrasonic sensor Schematic

8.5.2 Assembly code

```
;;       HC-SR04 Ultrasonic sensor reading
;; RS --> P0.3 E --> P0.4 RW --> P0.5 D7 - D0 --> P1  ECHO --> P0.1 TRIG --> P0.0
;; 1 mm = 5.8 µs : 35 MHz / (1 / 5.8 µs) = 203
;; Final distance (mm) = Counter / (203 * 1)

;;; Display
digit_9 = $57      ;; ascii value of digit 9
decimal_numbers = $9
digit_len_counter_address = $210
;;; fsys = 35 MHz
display_th0 = x0a
display_tl0 = xae
display_cs0 = $1

LCD_DATA = P1
LCD_RS = P0.3
LCD_E = P0.4
LCD_RW = P0.5
LCD_BF = P1.7
BF_BIT = b10000000
;; HCSR-04 sensor
TRIG  = P0.0
ECHO = P0.1

echo_pin = $2
;;; fsys = 35 MHz
us_10_th0 = x01
us_10_tl0 = x5e
divisor_address = $200
;;; fsys = 35 MHz
divisor1 = $203
divisor2 = $1
```

```
dividend_h = $201
dividend_l = $202
result_h = $203
result_l = $204
sensor_read_delay = $25

[$210 : $0]
my_word:
      .word @ ADDRESS, DATA_LEN "Distance : "
__init__:
      MOV TH0 display_th0
      MOV TL0 display_tl0
      MOV CS0 display_cs0
      SET T0E
      MOV R1 $10
      DDRP0 ( $0 | echo_pin ) ;; setting echo pin as input
      DDRP1 $0
      CALL set_registers
set_display:
      MOV R2 x30         ;; full byte mode Data Pins are D7 - D0
      CALL set_instruction
clear_display:
      CLR LCD_RS
      MOV R2 $1          ;; clear display
      CALL set_instruction
return_home:
      CLR LCD_RS
      MOV R2 $2          ;; cursor return home
      CALL set_instruction
      MOV R2 $12         ;; sets display, blink off, cursor off
      CALL set_instruction
      SET LCD_RS         ;; write data to LCD
read_usensor:
      ;; initialise counter 0
      MOV TH0 $0
```

```
        MOV TL0 $0
        SET T0E
        SET TC0L
timer_init:
        ;; set 10 µs delay for trigger pulse
        MOV TH0 us_10_th0
        MOV TL0 us_10_tl0
        MOV CS0 $0
        SET T0E
send_trigger:
        SET TRIG
        CALL delay
        CLR TRIG
;; receive_phase
wait_for_echo_High:
        JNB ECHO wait_for_echo_High     ;; wait for echo pin to go HIGH indicating
                                         ;; start of counter
counter_init:
        SET TCR0E                       ;; start counter 0
wait_for_echo_Low:
        JB ECHO wait_for_echo_Low
        CLR T0                          ;; stop and reset counter 0
        mov a, tc0l                     ;; load lower byte of the counter
        sta dividend_l                  ;; store it in address dividend_l
        mov a, tc0h                     ;; load upper byte of the counter
        sta dividend_h                  ;; store it in address dividend_h
        ldi divisor1                    ;; load 10 in A register
        sta divisor_address             ;; store it in the divisor_address
        MOV R2 $1                       ;; R2 stores the number of divisions
                                        ;; that must be performed
U16int_DIV:
        ;; unsigned integer division using consecutive subtraction with carry
        ldi $0
        sta result_h
        sta result_l
```

```
continue:
      lda dividend_l
      sub divisor_address
      sta dividend_l
      lda dividend_h
      subib $0                    ;; subraction with carry
      sta dividend_h
      jmc increment_result  ;; if division was possible then increment the result
      lda result_l
      sta dividend_l
      lda result_h
      sta dividend_h
      ldi divisor2         ;; load 29 in A register
      sta divisor_address   ;; store it in the divisor_address
      djnz R2, U16int_DIV
      lda result_l
wait_1sec:
      MOV TH0 display_th0
      MOV TL0 display_tl0
      MOV CS0 display_cs0
      SET T0E
      CALL delay
      jmp display_character
increment_result:
      lda result_l
      addi $1
      sta result_l
      lda result_h
      addic $0
      sta result_h
      jmp continue
display_character:
      push a
      push r1
      push r0
      MOV R0 ADDRESS          ;; R0 stores the address of my data
      MOV R1 DATA_LEN         ;; stores the lenght of ADDRESS
      DEC R1 $1              ;; len - 1
send_characters:
      SET LCD_RS
      MOV A @R0
      MOV R2, A
      CALL set_instruction
      DEC R0 $1
      DJNZ R1, send_characters
      pop r0
      pop r1
      pop a
find_remainder:
      MOV DM $0
      DIV R1
      MOV R2, A
decode_digit:
      MOV A, R2
      SUB R3
      JMZ store_number
      DEC R0 $1
      DJNZ R3, decode_digit
set_instruction:
      MOV A, R2
execute_instruction:
      MOV LCD_DATA, A
      SET LCD_E
check_BF:
      CLR LCD_RS         ;; RS LOW --> Instruction mode
      SET LCD_RW         ;; R/W HIGH  --> Read mode
      DDRP1 BF_BIT       ;; Setting D7/BF as input to read the busy flag
lcd_busy:
      JB LCD_BF lcd_busy ;; BF = 1 --> LCD is busy BF = 0 --> LCD finished executing
      DDRP1 $0           ;; Set all bits to out mode
```

```
        CLR LCD_E           ;; E LOW, i.e. finished execution cycle
        CLR LCD_RW          ;; R/W LOW --> Write mode
        RET
delay:
        SET TR0
loop:
        JNBTF0 loop
        CLR T0
        RET
store_number:
        LDA digit_len_counter_address
        ADDI $1
        STA digit_len_counter_address
        MOV A, R0
        PUSH A
        MOV A, DM
        SUBI $0
        CALL set_registers
        JMZ display_number
        MOV A, DM
        JMP find_remainder
set_registers:
        MOV R0 digit_9 ;; R0 stores ascii value of digit 9
        MOV R3 decimal_numbers ;; stores the lenght of ADDRESS
        RET
display_number:
        SET LCD_RS
        POP A
        CALL execute_instruction
        LDA digit_len_counter_address
        SUBI $1
        STA digit_len_counter_address
        JMZ read_delay
        JMP display_number
read_delay:
        MOV R2 $1
add_mask:
        SET LCD_RS
        ldi " "
        CALL execute_instruction
        DJNZ R2, add_mask
        PUSH R0
        MOV R0 sensor_read_delay
read_delay_loop:
        CALL delay
        DJNZ R0, read_delay_loop
        POP R0
        JMP return_home
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 158 bytes, 61.72%
```

8.5.3 Result

The HC-SR04 ultrasonic sensor measures the distrance between the sensor and the object.

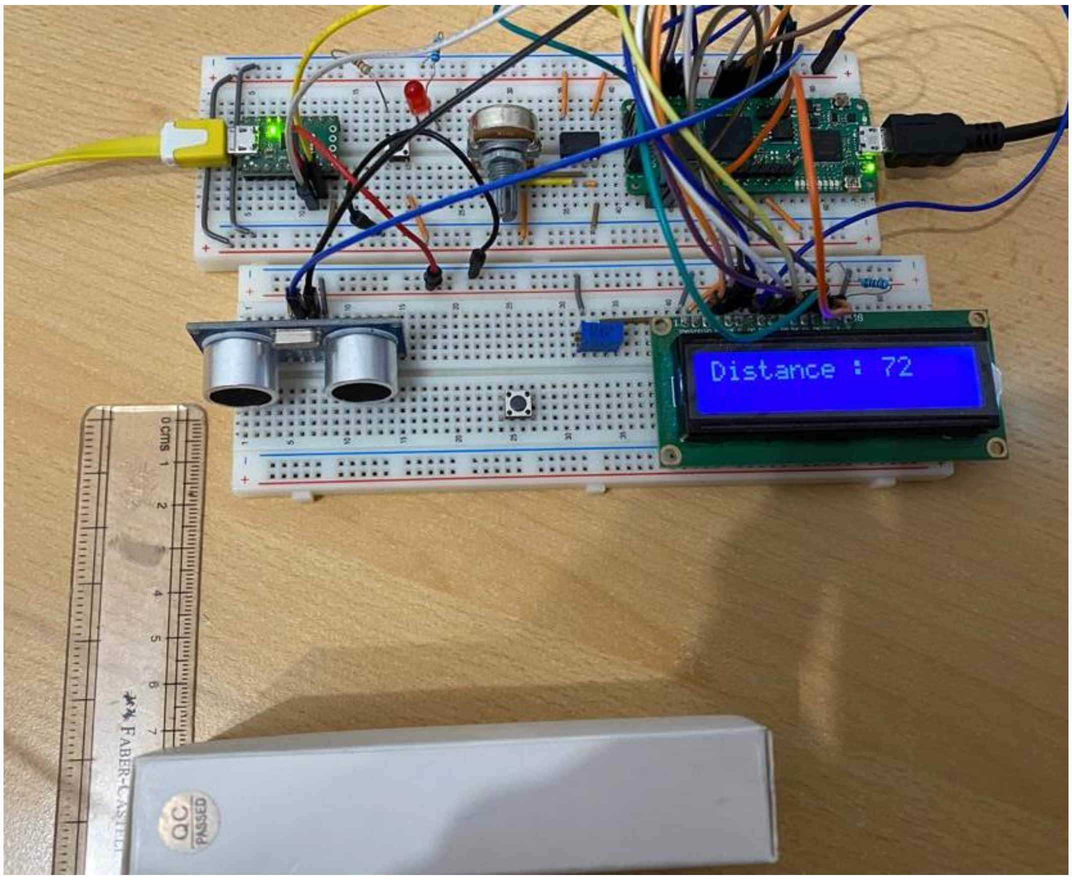


Figure 8.5.3-1 Distance measured between the sensor and the object

8.6 UART Interface programming

The microcontroller core does not have an UART interface built in. However, an UART interface would allow the programmer to send useful information such as register contents or sensor values to the host computer. A *Universal asynchronous receiver-transmitter (UART)* sends data bits one by one, from the least significant to the most significant, framed by the start and stop bits. The data framing diagram is shown below.

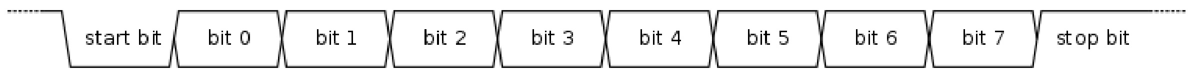


Figure 8.6-1 UART bit transmission sequence

For our microcontroller design, the transmission speed would be limited to 9600 and 38400.

8.6.1 Schematic diagram

The board schematic is shown below.

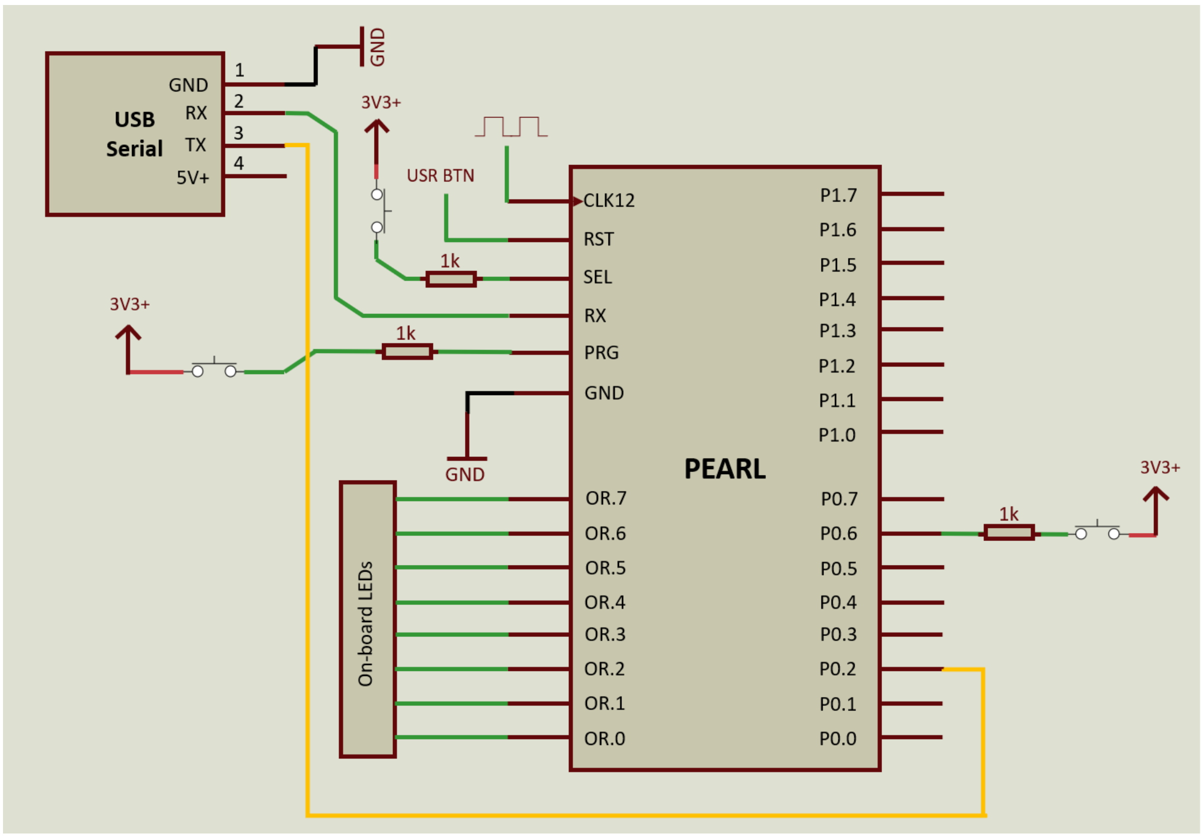


Figure 8.6.1-1 UART communication Schematic

8.6.2 Assembly code

```
        .word @ ADDRESS, len "Hello, world!!"
        .word @ , len1 ""

        tx = p0.2
        btn = p0.6

        ;;; fsys = 50 MHz
        ; tx_delay_th = x0a ;;half baud rate
        ; tx_delay_tl = x2c
        ; baud_th = x14  ;; baud 9600
        ; baud_tl = x58
        ; tx_delay_th = x02;;half baud rate
        ; tx_delay_tl = x8b
        ; baud_th = x05  ;; baud 38400
        ; baud_tl = x16
        ;;; fsys = 35 MHz
        tx_delay_th = x07 ;;half baud rate
        tx_delay_tl = x1f
        baud_th = x0e  ;; baud 9600
        baud_tl = x3e
        prescale = $0
        btn_out = b01000000
        tx_out = x0
        ddrp0 ($0 | btn_out | tx_out)
__init__:
        mov r3 (len + len1 +$2 - $1) ;; (r0 = len + len1 + CR + LF - 1)
        mov a, r3
        mov r0 (ADDRESS - len1 - len)
        ldi x0d
        mov @r0 a                    ;; add CR at the end of the string
        dec r0 $1
        ldi x0a
```

```
      mov @r0 a                    ;; add LF at the end of the string
      mov r0 ADDRESS
init_serial:
      call load_character
      set tx                  ;; pulling tx line high i.e. idle
      mov r1 $7               ;; R1 holds the number of characters from 7 to 0
      ;; setting Timer 0 to produce a delay to reach the middle of the bit
      mov th0 tx_delay_th
      mov tl0 tx_delay_tl
      mov cs0 prescale
      set t0e
      call delay
start:
      clr tx                   ;; start condition
      call delay
      ;; setting Time 0 to produce delay for 9600 baud
      mov th0 baud_th
      mov tl0 baud_tl
      set t0e
send_data:
      mov a, r2
      mov c $0
      rrc
      mov tx, c                 ;; send each bit
      call delay
      mov r2, a
      djnz r1, send_data   ;; loop till all bits are sent
      set tx                    ;; stop condition
      call delay
      dec r0 $1                 ;; decrement string pointer
      djnz r3, init_serial ;; loop though all characters of the string
wait:
      jnb btn wait         ;; wait for button press to resend the string
      ;; debouncing delay
      mov th0 b01001100
```

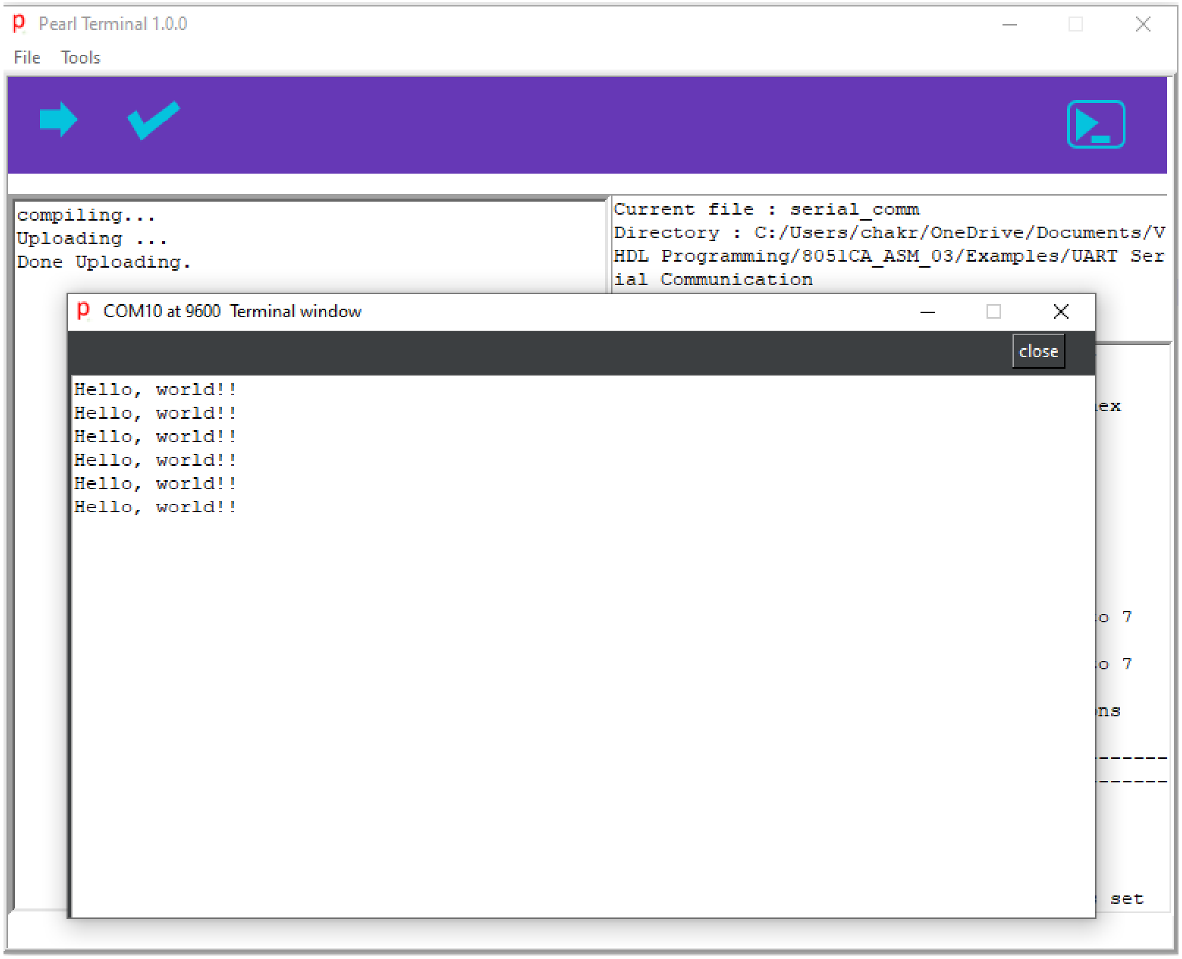
```
        mov tl0 b01001011
        mov cs0 $3
        set t0e
        call delay
        jmp __init__
load_character:
        mov a @r0
        mov r2, a
        ret
delay:
        set tr0
loop:
        jnbtf0 loop
        clr t0
        ret
```

Report file RAM information:

```
Memory available : 256 bytes,  Memory used : 63 bytes, 24.61%
```

8.6.3 Result

The microcontroller sends the string “Hello, world!!” to the host computer whenever the button at P0.6 is pressed and can be monitored using the *pearl interface terminal*.



```
compiling...
Uploading ...
Done Uploading.
```

```
Current file : serial_comm
Directory : C:/Users/chakr/OneDrive/Documents/V
HDL Programming/8051CA_ASM_03/Examples/UART Ser
ial Communication
```

```
Hello, world!!
Hello, world!!
Hello, world!!
Hello, world!!
Hello, world!!
Hello, world!!
```

Figure 8.6.3-1 Data received by pearl terminal window

Pearl assembly instruction set

Mnemonic Instruction		**Opcode**	**Clock cycle**	**Addressing**	**Description**
ADD	*address*	0x02	11	Direct	A 🡐 A + @*address*
	R0A	0x20	9	Register	A 🡐 A + R0A
	R0B	0x21	9	Register	A 🡐 A + R0B
	R0	0x22	9	Register	A 🡐 A + R0
	R1A	0x26	9	Register	A 🡐 A + R1A
	R1B	0x27	9	Register	A 🡐 A + R1B
	R1	0x28	9	Register	A 🡐 A + R1
	R2A	0x2c	9	Register	A 🡐 A + R2A
	R2B	0x2d	9	Register	A 🡐 A + R2B
	R2	0x2e	9	Register	A 🡐 A + R2
	R3	0x97	9	Register	A 🡐 A + R3
ADDC	*address*	0xc8	11	Direct	A 🡐 A + @*address* + C
	R0	0xc3	9	Register	A 🡐 A + R0 + C
	R1	0xc4	9	Register	A 🡐 A + R1 + C
	R2	0xc5	9	Register	A 🡐 A + R2 + C
	R3	0xc6	9	Register	A 🡐 A + R3 + C
ADDI	*immediate*	0xa3	9	Immediate	A 🡐 A + Immediate
ADDIC	*immediate*	0xc7	9	Immediate	A 🡐 A + Immediate + C
ANL	R0	0x83	11	Register	A 🡐 A and R0
	R1	0x84	11	Register	A 🡐 A and R1
	R2	0x85	11	Register	A 🡐 A and R2
	R3	0x86	11	Register	A 🡐 A and R3
CLR	P0.x	0x33	13	Port 0 Bit	P0.x 🡐 0
	P1.x	0xba	13	Port 1 Bit	P1.x 🡐 0
	T0	0x3a	7	Timer 0 Bit	T0 🡐 0 (Timer/Counter stop)
CMP	*address*	0xbf	11	Direct	Compare @*address* with A
CMPI	*immediate*	0xbe	9	Immediate	Compare Immediate value with A
CPL	A	0x8b	11	Register	A 🡐$\overline{A}$
CALL	*Address*	0x5a	9	Direct	Subroutine jump to *address*
DDRP0	*Immediate*	0x62	7	Immediate	Data direction register for Port 0
DDRP1	*Immediate*	0xb5	7	Immediate	Data direction register for Port 1
DEC	R0A *immediate*	0x48	13	Register	R0A 🡐 R0A – Immediate
	R0B *immediate*	0x49	13	Register	R0B 🡐 R0B – Immediate
	R0 *immediate*	0x4a	13	Register	R0 🡐 R0 – Immediate
	R1A *immediate*	0x4b	13	Register	R1A 🡐 R1A – Immediate
	R1B *immediate*	0x4c	13	Register	R1B 🡐 R1B – Immediate
	R1 *immediate*	0x4d	13	Register	R1 🡐 R1 – Immediate

Mnemonic Instruction		Opcode	Clock cycle	Addressing	Description
	R2A *immediate*	0x4e	13	Register	R2A ← R2A – Immediate
	R2B *immediate*	0x4f	13	Register	R2B ← R2B – Immediate
	R2 *immediate*	0x50	13	Register	R2 ← R2 – Immediate
	R3 *immediate*	0x95	13	Register	R3 ← R3 – Immediate
DIV	R0	0x7a	[1]	Register	A ← A / R0; DM ← A % R0
	R1	0x7c	[1]	Register	A ← A / R1; DM ← A % R1
	R2	0x7e	[1]	Register	A ← A / R2; DM ← A % R2
DJNZ	R0A, *address*	0x51	19	Register	Decrement R0A. Jump to *address* if not zero
	R0B, *address*	0x52	19	Register	Decrement R0B. Jump to *address* if not zero
	R0, *address*	0x53	19	Register	Decrement R0. Jump to *address* if not zero
	R1A, *address*	0x54	19	Register	Decrement R1A. Jump to *address* if not zero
	R1B, *address*	0x55	19	Register	Decrement R1B. Jump to *address* if not zero
	R1, *address*	0x56	19	Register	Decrement R1. Jump to *address* if not zero
	R2A, *address*	0x57	19	Register	Decrement R2A. Jump to *address* if not zero
	R2B, *address*	0x58	19	Register	Decrement R2B. Jump to *address* if not zero
	R2, *address*	0x59	19	Register	Decrement R2. Jump to *address* if not zero
	R3, *address*	0x96	19	Register	Decrement R3. Jump to *address* if not zero
HLT	-	0x0f	7	-	CPU Halt
INC	R0A *immediate*	0x3f	13	Register	R0A ← R0A + Immediate
	R0B *immediate*	0x40	13	Register	R0B ← R0B + Immediate
	R0 *immediate*	0x41	13	Register	R0 ← R0 + Immediate
	R1A *immediate*	0x42	13	Register	R1A ← R1A + Immediate
	R1B *immediate*	0x43	13	Register	R1B ← R1B + Immediate
	R1 *immediate*	0x44	13	Register	R1 ← R1 + Immediate
	R2A *immediate*	0x45	13	Register	R2A ← R2A + Immediate

Mnemonic Instruction		Opcode	Clock cycle	Addressing	Description
	R2B *immediate*	0x46	13	Register	R2B 🡐 R2B + Immediate
	R2 *immediate*	0x47	13	Register	R2 🡐 R2 + Immediate
	R3 *immediate*	0x94	13	Register	R3 🡐 R3 + Immediate
JMP	*address*	0x06	7	Direct	Jump to *address*
JMC	*address*	0x07	7	Direct	Jump to *address* if carry set
JMZ	*address*	0x08	7	Direct	Jump to *address* if zero set
JBTF0	*address*	0x37	7	Direct	Jump to *address* if Timer 0 Compare Match Flag is set
JNBTF0	*address*	0x38	7	Direct	Jump to *address* if Timer 0 Compare Match Flag is not set
JB	P0.x, *address*	0x63	19	Direct	Jump to *address* if P0.x Bit is set
	P1.x, *address*	0x65	19	Direct	Jump to *address* if P1.x Bit is set
JNB	P0.x, *address*	0x64	19	Direct	Jump to *address* if P0.x Bit is not set
	P1.x, *address*	0x66	19	Direct	Jump to *address* if P1.x Bit is not set
JT0V	*address*	0xcc	7	Direct	Jump to *address* if Counter 0 Overflow Flag is set
JNT0V	*address*	0xcd	7	Direct	Jump to *address* if Counter 0 Overflow Flag is not set
LDA	*address*	0x01	9	Direct	A 🡐 @*address*
LDI	*immediate*	0x05	7	Immediate	A 🡐 Immediate
MOV	A, R0A	0x0c	7	Register	A 🡐 R0A
	A, R0B	0x0d	7	Register	A 🡐 R0B
	A, R0	0x10	7	Register	A 🡐 R0
	A, R1A	0x15	7	Register	A 🡐 R1A
	A, R1B	0x16	7	Register	A 🡐 R1B
	A, R1	0x17	7	Register	A 🡐 R1
	A, R2A	0x1c	7	Register	A 🡐 R2A
	A, R2B	0x1d	7	Register	A 🡐 R2B
	A, R2	0x1e	7	Register	A 🡐 R2
	A, R3	0x74	7	Register	A 🡐 R3
	A, DM / R4	0x77	7	Register	A 🡐 DM
	A @R0	0x8c	9	Indirect	A 🡐 @R0; pointer to an address
	A @R1	0x8d	9	Indirect	A 🡐 @R1; pointer to an address
	A @R2	0x8e	9	Indirect	A 🡐 @R2; pointer to an address
	A @R3	0x8f	9	Indirect	A 🡐 @R3; pointer to an address
	A, C	0x9c	7	Carry Bit	A (0) 🡐 C
	A, P0	0x34	7	Port 0	A 🡐 P0
	A, P1	0xb8	7	Port 1	A 🡐 P1
	A, TC0L	0xc9	7	Timer 0	A 🡐 TC0L; counter 0 Low byte
	A, TC0H	0xca	7	Timer 0	A 🡐 TC0H; counter 0 High byte
	A, PC	0xd0	7	Register	A 🡐 PC; program counter

Mnemonic Instruction		Opcode	Clock cycle	Addressing	Description
	A, LR	0xd3	7	Register	A ← LR; link register
	R0, A	0x11	7	Register	R0 ← A
	R1, A	0x18	7	Register	R1 ← A
	R2, A	0x1f	7	Register	R2 ← A
	R3, A	0x75	7	Register	R3 ← A
	DM / R4, A	0x78	7	Register	DM ← A
	@R0, A	0x90	9	Indirect	@R0 ← A
	@R1, A	0x91	9	Indirect	@R1 ← A
	@R2, A	0x92	9	Indirect	@R2 ← A
	@R3, A	0x93	9	Indirect	@R3 ← A
	C, A	0x9d	7	Carry Bit	C ← A (0)
	P0, A	0x35	7	Port 0	P0 ← A
	P1, A	0xb7	7	Port 1	P1 ← A
	TH0, A	0xc0	7	Timer 0	TH0 ← A
	TL0, A	0xc1	7	Timer 0	TL0 ← A
	CS0, A	0xc2	7	Timer 0	CS0 ← A
	PC, A	0xcf	7	Register	PC ← A
	LR, A	0xd2	7	Register	LR ← A
	P0.x, C	0x99	25	Port 0 Bit	P0.x ← C
	P1.x, C	0xbb	25	Port 1 Bit	P1.x ← C
	C, P0.x	0x9a	17	Port 0 Bit	C ← P0.x
	C, P1.x	0xbc	17	Port 1 Bit	C ← P1.x
	R0A *immediate*	0x09	7	Immediate	R0A ← Immediate
	R0B *immediate*	0x0a	7	Immediate	R0B ← Immediate
	R0 *immediate*	0x0b	7	Immediate	R0 ← Immediate
	R1A *immediate*	0x12	7	Immediate	R1A ← Immediate
	R1B *immediate*	0x13	7	Immediate	R1B ← Immediate
	R1 *immediate*	0x14	7	Immediate	R1 ← Immediate
	R2A *immediate*	0x19	7	Immediate	R2A ← Immediate
	R2B *immediate*	0x1a	7	Immediate	R2B ← Immediate
	R2 *immediate*	0x1b	7	Immediate	R2 ← Immediate
	R3 *immediate*	0x73	7	Immediate	R3 ← Immediate
	DM / R4 immediate	0x76	7	Immediate	DM ← Immediate
	C *immediate*	0x9e	7	Immediate	C ← Immediate

Mnemonic Instruction		Opcode	Clock cycle	Addressing	Description
	P0 *immediate*	0x36	7	Immediate	P0 ← Immediate
	P1 *immediate*	0xb6	7	Immediate	P1 ← Immediate
	TH0 *immediate*	0x3c	7	Immediate	TH0 ← Immediate
	TL0 *immediate*	0x3d	7	Immediate	TL0 ← Immediate
	CS0 *immediate*	0x3e	7	Immediate	CS0 ← Immediate
	LR *immediate*	0xd1	7	Immediate	LR ← Immediate
	LR, PC	0xd5	7	Immediate	LR ← PC
	PC, LR	0xd4	7	Immediate	PC ← LR
MUL	R0	0x79	[1]	Register	R3 * R0; A ← High byte DM ← Low byte
	R1	0x7b	[1]	Register	R3 * R1; A ← High byte DM ← Low byte
	R2	0x7d	[1]	Register	R3 * R2; A ← High byte DM ← Low byte
ORL	R0	0x17	9	Register	A ← A or R0
	R1	0x18	9	Register	A ← A or R1
	R2	0x19	9	Register	A ← A or R2
	R3	0x1a	9	Register	A ← A or R3
OUT	-	0x0e	7	Register	Output port ← A
PUSH	R0	0x5c	7	Stack	SM ← R0
	R1	0x5e	7	Stack	SM ← R1
	R2	0x60	7	Stack	SM ← R2
	R3	0x81	7	Stack	SM ← R3
	DM / R4	0x7f	7	Stack	SM ← DM
	A	0xa1	7	Stack	SM ← A
POP	R0	0x5d	7	Stack	R0 ← SM
	R1	0x5f	7	Stack	R1 ← SM
	R2	0x61	7	Stack	R2 ← SM
	R3	0x82	7	Stack	R3 ← SM
	DM / R4	0x80	7	Stack	DM ← SM
	A	0xa2	7	Stack	A ← SM
RDA	P0 *value*	0x9b	9	Register	A ← parallel digital read from Port 0 pins specified by *value*
	P1 *value*	0xbd	9	Register	A ← parallel digital read from Port 1 pins specified by *value*
RET	-	0x5b	7	Stack	Return from subroutine
RLC	-	0xa0	7	Register	Rotate A Left with carry bit
RRC	-	0x9f	7	Register	Rotate A Right with carry bit
SET	P0.x	0x32	13	Port 0 Bit	P0.x ← 1
	P1.x	0xb9	13	Port 1 Bit	P1.x ← 1
	T0E	0x39	7	Timer 0 Bit	Timer 0 Enable

Mnemonic Instruction		Opcode	Clock cycle	Addressing	Description
	TC0L	0xc7	7	Timer 0 Bit	Load Counter 0 trim value
	TCR0E	0xcb	7	Timer 0 Bit	Run Counter 0
	TR0	0x3b	7	Timer 0 Bit	Run Timer 0
STA	*address*	0x04	9	Direct	*@address* ← A
SUB	*address*	0x03	11	Direct	A ← A - *@address*
	R0A	0x23	9	Register	A ← A - R0A
	R0B	0x24	9	Register	A ← A - R0B
	R0	0x25	9	Register	A ← A - R0
	R1A	0x29	9	Register	A ← A - R1A
	R1B	0x2a	9	Register	A ← A - R1B
	R1	0x2b	9	Register	A ← A - R1
	R2A	0x2f	9	Register	A ← A - R2A
	R2B	0x30	9	Register	A ← A - R2B
	R2	0x31	9	Register	A ← A - R2
	R3	0x98	9	Register	A ← A - R3
SUBB	*address*	0xdb	17	Direct	A ← A - *@address* - C
	R0	0xd6	15	Register	A ← A - R0 – C
	R1	0xd7	15	Register	A ← A - R1 – C
	R2	0xd8	15	Register	A ← A - R2 – C
	R3	0xd9	15	Register	A ← A - R3 – C
SUBI	*Immediate*	0xa4	9	Immediate	A ← A – Immediate
SUBIB	*Immediate*	0xda	15	Immediate	A ← A – Immediate - C

[1] The number of clock cycles depends on the operand values.

Bibliography

Albert Paul Malvino, P., & Brown, J. A. (n.d.). *Digital Computer Electronics Third Edition.*

Eater, B. (n.d.). *Ben Eater*. Retrieved from https://eater.net/8bit

Mazidi, M. A., Mazadi, J. G., & McKinlay, R. D. (2009). *The 8051 Microcontroller And Embedderd Systems Using assembly and C Second Edition.* New Delhi, India: Dorling Kindersley (India) Pvt. Ltd., liscensees of Pearson Education in South Asia.