MINI PROJECT ON
HEALTH MONITORING SYSTEM USING 7-SEGMENT DISPLAY AND MICROCONTROLLER FOR THE PARTIAL FULFILMENT OF DSD-2 LAB

Roll No of Students:

1. 204101
2. 204102
3. 204103
4. 204104

Faculty:

1. Dr. Prithvi Pothupogu
2. Dr. Hanumanth Rao T V K

Department of Electronics and Communication Engineering

Year: 2022

Abstract:

As a generation grows up there is the wide necessity of technology to gain control over the activities and monitor interpret them, without a human existence all the time. One of them is the health monitor which can monitor the health condition of a person with parameters of heartbeat and temperature. As heartbeat rate ranges differ for individual age ranges so it's important to have manual control of heartbeat range so it makes observation much easier to interpret.

A health monitor considers parameters of heartbeat and temperature of a person. The temperature range is fixed for humans permissible and the heartbeat range is set for the individual based on their age. Inputs of temperature and heartbeat are fed to the microcontroller and logic will generate the display number through one of the outputs and alert signals will get generated by other outputs. There is an alert message generated when the temperature or heartbeat increases so that the well-wisher of the person can get an alert in case of emergency. They can be used for: -

1. Monitoring infant and pregnant women
2. An aged person has gone out for some other place alone
3. Patients at hospitals and home
4. Mountain climbers and sea divers

Segments of the project: -

1. Display of heartbeat and temperature.
2. Alarm signal generating when the range exceeds.
3. Entering the range of heartbeat.

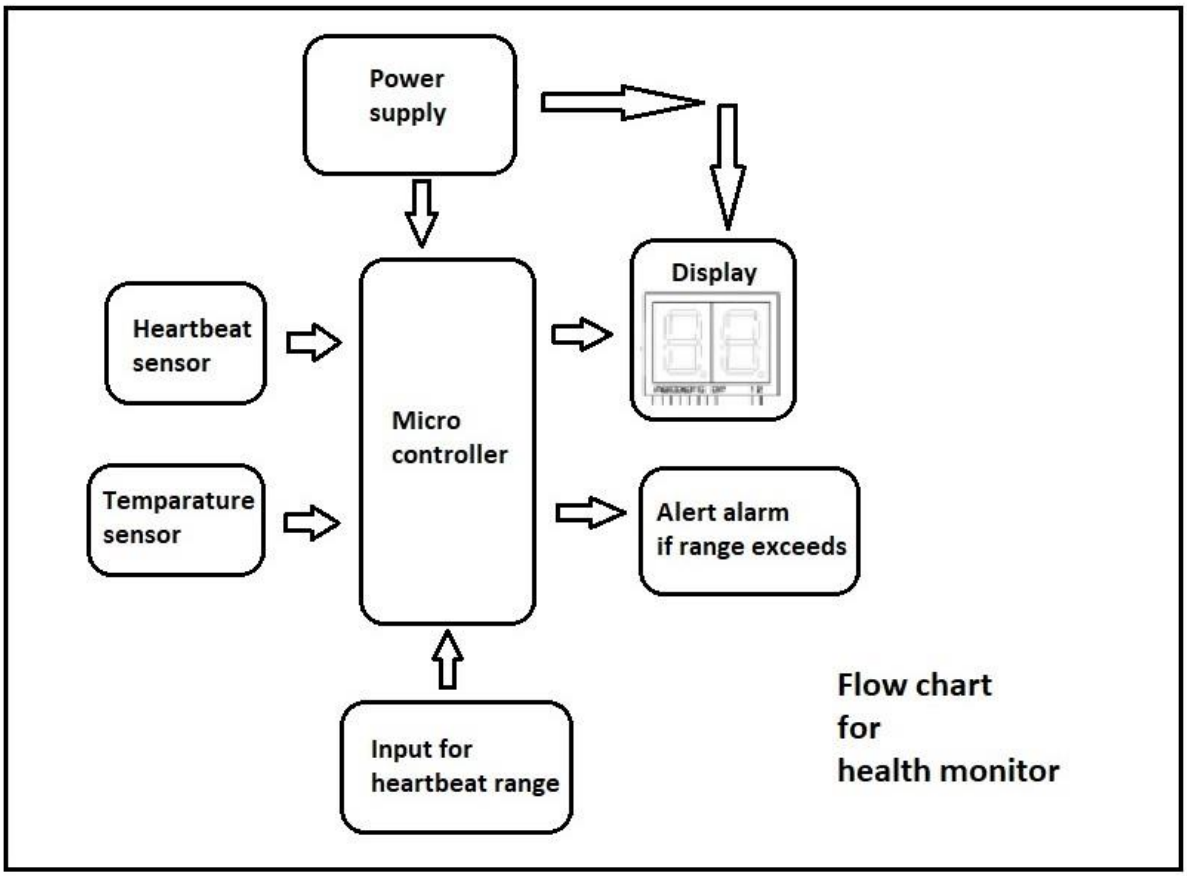
Languages used:

- Verilog

Tools required:

- XILINX-ISE -> Synthesis/coding.
- VIVADO -> Simulation.

Flow chart:



Hardware Circuit:

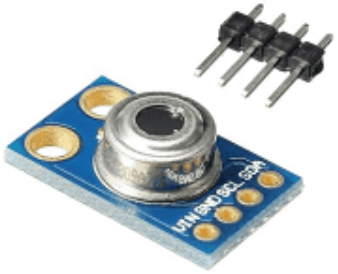
Theory:

Temperature Sensor

What are temperature Sensors?

A temperature sensor is a device, typically, a thermocouple or resistance temperature detector, that provides temperature measurement in a readable form through an electrical signal. A thermometer is the most basic form of a temperature sensor that is used to measure the degree of hotness and coolness.

Working:



The working of a temperature meter depends upon the voltage across the diode. The temperature change is directly proportional to the diode's resistance. The cooler the temperature, lesser will be the resistance, and vice-versa. The resistance across the diode is measured and converted into readable units of temperature (Fahrenheit, Celsius, Centigrade, etc.) and, displayed in numeric form over readout units.

In this project we will be using a non-contact temperature sensor to prevent contact disease transmission (MLX90614 ESF)

Heartbeat Sensor

An optical heart rate sensor measures pulse waves, which are changes in the volume of a blood vessel that occur when the heart pumps blood. Pulse waves are detected by measuring the change in volume using an optical sensor and green LED. Adopting an optical filter optimized for pulse wave detection in the sensor block minimizes the effects of ambient light such as red and infrared rays. This enables high quality pulse signals to be acquired, even outdoors.

Practical heartbeat Sensor examples are Heart Rate Sensor (Product No PC-3147). It consists of an infrared led and an LDR embedded onto a clip-like structure. The clip is attached to the organ (earlobe or the finger) with the detector part on the flesh.

Another example is TCRT1000, having 4 pins-

Pin1: To give the supply voltage to the LEDPin2 and 3 are grounded. Pin 4 is the output. Pin 1 is also the enable pin and pulling it high turns the LED on and the sensor starts working. It is embedded on a wearable device that can be worn on the wrist and the output can be sent wirelessly (through Bluetooth) to the computer for processing.

MICROCONTROLLER

WHAT IS A MICROCONTROLLER?

An integrated circuit consists of a microprocessor and memory and I/O interfaces designed to govern a specific operation in an embedded system

THE CORE ELEMENTS OF A MICROCONTROLLER ARE:

1) **PROCESSOR** -- It processes and responds to various instructions that direct the microcontroller's function. This involves performing the basic arithmetic, logic, and I/O operations.
2) **MEMORY**-- is used to store the data that the processor receives and uses to respond to instructions that it's been programmed to carry out. A microcontroller has two main memory types:

a) Program memory, used to store program code. Program memory is non-volatile memory, meaning it holds information over time without needing a power source.

b) Data memory, is used to store data variables. Data memory is volatile, Data gets lost once the power source gets disconnected.

3) **I/O peripherals** -- The input ports receive information and send it to the processor in the form of binary data. The processor receives that data and sends the necessary instructions to output devices that execute tasks external to the microcontroller.

INSTRUCTION SET ARCHITECTURE:

Each instruction is 12 bits.

There are 3 types of instructions by encoding:

1) M type: one operand is an accumulator and the other operand is from data memory; the result can be stored in the accumulator or the data memory entry.
2) **I type:** one operand is an accumulator and the other operand is an immediate number encoded in instruction; the result is stored in the accumulator.
3) S type: special instruction, no operand required.

These instructions can be grouped into 4 categories by functions:

1. ALU instruction: using ALU to compute results.

2. Unconditional branch: the GOTO instruction.

3. Conditional branch: the JZ, JC, JS, JO instruction.

4. Special instruction: the NOP.

M TYPE INSTRUCTIONS:

The general format of M type instruction:

001	d	Opcode(4bits)	Dmem index (4 bits)

VARIOUS INSTRUCTION:

Instruction mnemonics	**Function**	**Encoding** (binary)	**Status affected**	**Example** (meaning)
ADD	add a memory entry with an accumulator	001d_0000_aaaa	Z, C, S, O	0011_0000_0001 ADD, Acc, Acc, DMem [1] (Acc=Acc+DMem [1])
SUBAM	subtract accumulator by a memory entry	001d_0001_aaaa	Z, C, S, O	0011_0000_0001 SUBAM, Acc, Acc, DMem [0] (Acc=Acc – DMem [0])
MOVAM	move the value of the accumulator to a memory entry	0010_0010_aaaa	none	0011_0000_0001 (Dmem [0] =Acc)
MOVMA	move the value of a memory entry to the accumulator	0011_0011_aaaa	none	0011_0000_0001 (Acc=DMem [0])
AND	bitwise and a memory entry with accumulator	001d_0100_aaaa	Z	0011_0000_0001 (DMem [0] =DMem [0] AND Acc)
OR	Bitwise OR a memory entry with an accumulator	001d_0101_aaaa	Z	0010_0101_0000 OR DMem [0], Acc, DMem [0] (DMem [0] =DMem [0] OR Acc)
XOR	Bitwise XOR a memory entry with an accumulator	001d_0110_aaaa	Z	0010_0110_0000 XOR DMem [0], Acc, DMem [0] (DMem [0] =DMem [0] XOR Acc)
SUBMA	Subtract a memory entry by accumulator	001d_0111_aaaa	Z, C, S, O	0010_0001_0000 SUBMA DMem [0], DMem [0], Acc (DMem [0] =DMem [0] - Acc)
INC	Increment a memory entry	0010_1000_aaaa	Z, C, S, O	0011_1000_0000 INC DMem [0] (DMem [0] = DMem [0] + 1)
DEC	Decrement a memory entry	0010_1001_aaaa	Z, C, S, O	0011_1001_0000 DEC DMem [0] (DMem [0] = DMem [0] - 1)
ROTATEL	Circulative shift left a memory entry, by the number of bits specified by the accumulator	0010_1010_aaaa	none	0010_1010_0000 ROTATEL DMem [0]

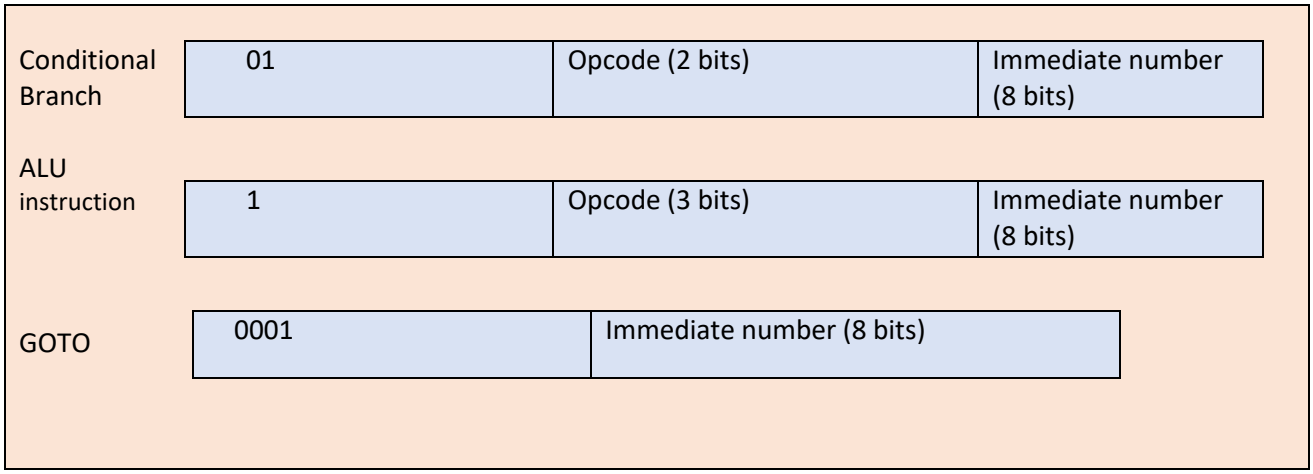
ROTATER	Circulative shift right a memory entry, by the number of bits specified by the accumulator	0010_1011_aaaa	None	0010_1011_0000 ROTATER DMem [0]
SLL	Shift a memory entry left, by the number of bits specified by the accumulator	0010_1100_aaaa	Z, C	0010_1100_0000 SLL DMem [0]
SRL	Shift a memory entry right, logical (fill 0), by the number of bits specified by the accumulator	0010_1101_aaaa	Z, C	0010_1101_0000 SRL DMem [0]
SRA	Shift a memory entry right, arithmetic (fill original MSB), by the number of bits specified by the accumulator	0010_1110_aaaa	Z, C, S	0010_1110_0000 SRA DMem [0]
COMP	Take 2's complement of a memory entry	0010_1111_aaaa	Z, C, S, O	0010_1111_0000 COMP DMem [0] (DMem [0] = -DMem [0])

Note that "aaaa" encodes the 4-bit address of data memory, and the "d" bit means the destination of the result, i.e., if d = 1, the result is written to Acc, otherwise, the result is written to the same memory location as the operand.

I TYPE INSTRUCTIONS:

I type instructions contain unconditional branch, conditional branch, and ALU instructions.

The general format of M type instruction is:



VARIOUS INSTRUCTION:

Instruction mnemonics	**function**	**encoding**	**Status affected**	**Example** (meaning)
GOTO	Unconditional branch	0001_xxxx_xxxx	none	0001_0000_0111 GOTO 7 (Goto the 8th instruction)
JZ	Jump to the instruction indexed by the immediate number, if the Z flag is 1	0100_xxxx_xxxx	None	0100_0000_0111 JZ 7 (Goto the 8th instruction if SR [3] ==1)
JC	Jump to the instruction indexed by the immediate number, if the C flag is 1	0101_xxxx_xxxx	None	0101_0000_0111 JC 7 (Goto the 8th instruction if SR [2] ==1)
JS	Jump to the instruction indexed by the immediate number, if the S flag is 1	0110_xxxx_xxxx	None	0101_0000_0111 JS 7 (goto the 8th instruction if SR[1]==1)
JO	Jump to the instruction indexed by the immediate number, if the O	0111_xxxx_xxxx	none	0101_0000_0111 JO 7 (Goto the 8th instruction if SR

	flag is 1			[0] ==1)

ADDI	Add accumulator with immediate number	1000_xxxxx_xxxx	Z, C, S, O	1000_0000_1000 ADDI Acc, Acc,8 (Acc=Acc + 8)
SUBAI	Subtract accumulator by immediate number	1001_xxxxx_xxxx	Z, C, S, O	1001_0000_1000 SUBAI Acc, Acc,8 (Acc=Acc - 8)
RSV	(Reserved, do nothing)	1010_xxxxx_xxxx	None	
MOVIA	Move immediate number to accumulator	1011_xxxxx_xxxx	None	1011_0000_0000 MOVIA Acc,0 (Acc=0)
ANDI	Bitwise AND accumulator with immediate number	1100_xxxxx_xxxx	Z	1100_0000_1111 ANDI Acc, Acc,0x0F (Acc = Acc AND 0x0F)
ORI	Bitwise OR accumulator with immediate number	1101_xxxxx_xxxx	Z	1101_1111_0000 ORI Acc, Acc,0xF0 (Acc = Acc OR 0XF0)
XORI	Bitwise XOR accumulator with immediate number	1110_xxxxx_xxxx	Z	1110_0000_1111 XORI Acc, Acc,0x0F (Acc = Acc XOR 0x0F)
SUBIA	Subtract accumulator by immediate number	1111_xxxxx_xxxx	Z, C, S, O	1111_0000_1000 SUBIA Acc,8, Acc (Acc = 8 - Acc)

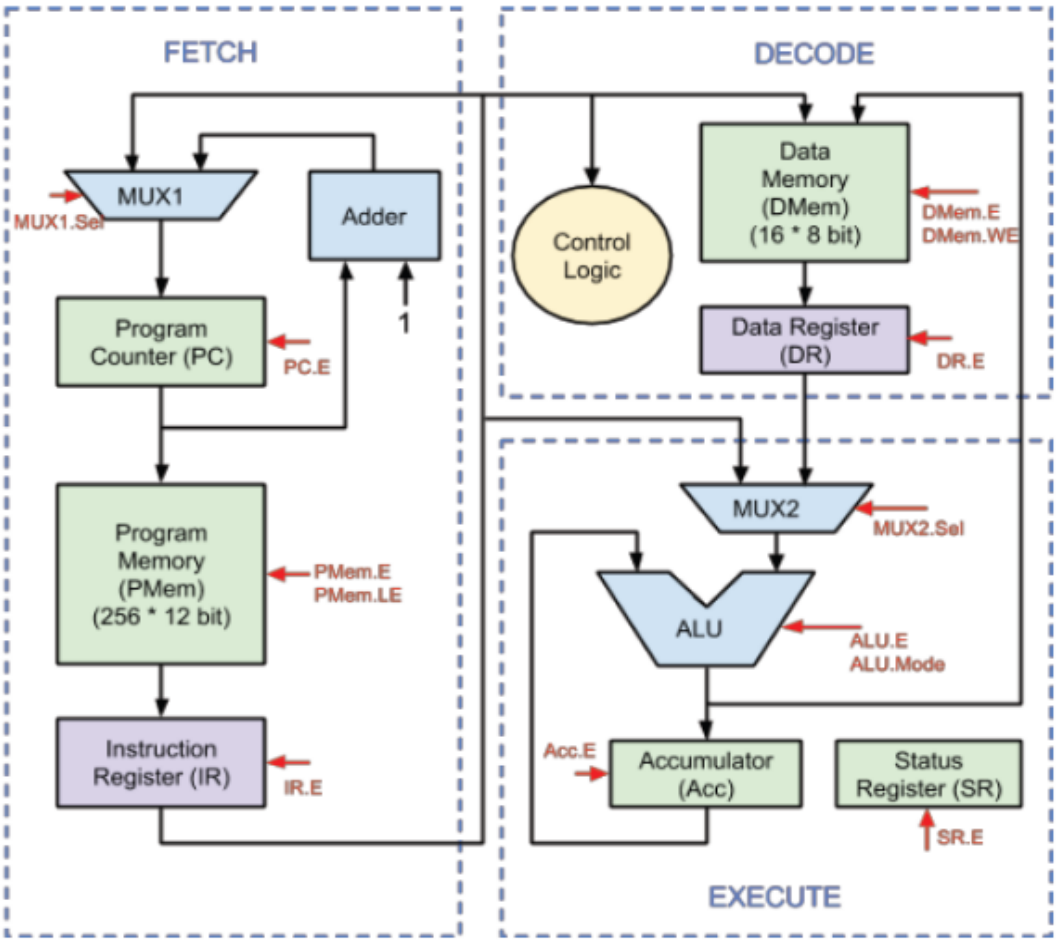
S TYPE INSTRUCTIONS:

The general format of **S** type instruction is:

S	0000	Opcode (8 bits)	type

There is only one S type instruction, i.e., the NOP instruction.

Instruction Mnemonics	**function**	**Encoding**	**Status affected**	**example**
NOP	no operation	0000_0000_0000_0000	none	NOP



The architecture of the microcontroller

The following two types of components hold programming context.

- Program counter, program memory, data memory, accumulator, status register. They are programmer visible registers and memories.
- Instruction registers and data register. They are programmer invisible registers.

The following two types of components are Boolean logic that does the actual computation work.

- ALU, MUX1, MUX2, Adder (blue boxes), used as a functional unit.
- Control Logic (yellow box), used to denote all control signals.

Each instruction needs 3 clock cycles to finish, i.e., FETCH stage, DECODE stage, and EXECUTE stage.

1. **LOAD (initial state):** load program to program memory.
2. **FETCH (first cycle):** fetch current instruction from program memory.
3. **DECODE (second cycle):** decode instruction to generate control logic, and read data memory for the operand.
4. **EXECUTE (of the third cycle):** execute the instruction.

Transitions:

1. LOAD → FETCH (initialization finish):
   Clear content of PC, IR, DR, Acc, SR; DMem is not required to be cleared.
2. FETCH → DECODE (rising edge of the second cycle):
   IR = PMem [PC]
3. DECODE → EXECUTE

0. DR = DMem [IR [3:0]] ;

4. EXECUTE → FETCH (rising edge of the first cycle and fourth cycle)
   a) For non-branch instruction, PC = PC + 1; for branch instruction, if branch is taken, PC = IR [7:0], otherwise PC = PC + 1;
   b) For ALU instruction, if the result destination is accumulator, Acc = ALU. Out; if the result destination is data memory, DMem [IR [3:0]] = ALU. Out
   c) For ALU instruction, SR = ALU. Status;

DESCRIPTION OF VARIOUS COMPONENTS OF MICROCONTROLLER:

1) Registers: The microcontroller has 3 programmers' visible registers:

1.1 Program Counter (8 bit, denoted as PC): contains the index of current executing instruction.

1.2 Accumulator (8 bit, denoted as Acc): holds result and 1 operand of the arithmetic or logic calculation.

1.3 Status Register (4 bit, denoted as SR): holds 4 status bits, i.e., Z, C, S, O.

a) Z (zero flag, SR [3]): 1 if the result is zero, 0 otherwise.

b) C (carry flag, SR [2]): 1 if carry is generated, 0 otherwise.

c) S (sign flag, SR [1]): 1 if the result is negative (as 2's complement), 0 otherwise.

d) O (overflow flag, SR [0]): 1 if result generates overflow, 0 otherwise.

The microcontroller has 2 programmer invisible registers (i.e., they cannot be manipulated by the programmer):

a) **Instruction Register (12 bit, denoted as IR):** contains the currently executing instruction.

b) **Data Register (8 bit, denoted as DR):** contains the operand read from data memory.

2) PROGRAM MEMORY:

The microcontroller has a 256-entry program memory that stores program instructions, denoted as PMem. Each entry is 12 bits, the ith entry is denoted as PMem[i]. The program memory has the following input/output ports.

- **Enable port (1 bit, input, denoted as PMem.E):** enable the device, i.e., if it is 1, then the entry specified by the address port will be read out, otherwise, nothing is readout.
- **Address port (8-bit, input, denoted as PMem.Addr):** specify which instruction entry is read out, connected to PC.
- **Instruction port (12-bit, output, denoted as PMem.I):** the instruction entry that is read out, connected to IR.
- 3 special ports are used to load the program to the memory, not used for executing instructions.
- **Load enables port (1 bit, input, denoted as PMem.LE):** enable the load, i.e., if it is 1, then the entry specified by the address port will be loaded with the value specified by the load instruction input port and the instruction port is supplied with the same value; otherwise, the entry specified by the address port will be read out on instruction port, and value on instruction load port is ignored.
- **Load address port (8-bit, input, denoted as PMem.LA):** specify which instruction entry is loaded.
- **Load instruction port (12-bit, input, denoted as PMem.LI):** the instruction that is loaded.

3) Data memory:

The microcontroller has a 16-entry data memory, denoted as DMem. Each entry is 8 bits, the ith entry is denoted as DMem[i]. The program memory has the following input/output ports.

. Enable port (1 bit, input, denoted as DMem.E): enable the device, i.e., if it is 1, then the entry specified by the address port will be read out or written in; otherwise nothing is read out or written in.

. Write enable port (1 bit, input, denoted as DMem.WE): enable the write, i.e., if it is 1, then the entry specified by the address port will be written with the value specified by the data input port and the data output port is supplied with the same value; otherwise, the entry specified by the address port will be read out on data output port, and value on data input port is ignored.

. Address port (4-bit, input, denoted as DMem.Addr): specify which data entry is read out, connected to IR [3:0].

. Data input port (8-bit, input, denoted as DMem.DI): the value that is written in, connected to ALU.Out.

. Data output port (8-bit, output, denoted as DMem.DO): the data entry that is read out, connected to MUX2.In1.

4) PC ADDER:

Program counter (PC) used to increment by PC by 1, in order to move at next instruction. It has the following port.

a) **Adder input port (8-bit, input, denoted as Adder.In):** connected to PC.

b) **Adder output port (8-bit, output, denoted as Adder.Out):** connected to MUX1.In2.

5) MUX1:

MUX1 is used to choose the source for updating PC. If the current instruction is not a branch or it is a branch but the branch is not taken, PC is incremented by 1; otherwise PC is set to the jumping target, i.e., IR [7:0]. It has the following port.

- **MUX1 input 1 port**: 8-bit, input, denoted as MUX1.In1, connected to IR [7:0].
- **MUX1 input 2 ports:** 8-bit, input, denoted as MUX1.In2, connected to Adder.Out.
- **MUX1 selection port**:1 bit, input, denoted as MUX1.Sel, connected to control logic.
- **MUX1 output port:** 8-bit, output, denoted as MUX1.Out, connected to PC.
- **MUX1 input 1 port**: 8-bit, input, denoted as MUX1.In1, connected to IR [7:0].
- **MUX1 input 2 ports:** 8-bit, input, denoted as MUX1.In2, connected to Adder.Out.
- **MUX1 selection port:** 1 bit, input, denoted as MUX1.Sel, connected to control logic.
- **MUX1 output port**: 8-bit, output, denoted as MUX1.Out, connected to PC.

6) MUX2:

MUX2 is used to choose the source for operand 2 of ALU. If the current instruction is M type, operand 2 of ALU comes from data memory; if the current instruction is I type, operand 2 of ALU comes from the instruction, i.e., IR [7:0]. It has the following port.

- **MUX2 input 1 port:** 8-bit, input, denoted as MUX2.In1, connected to IR [7:0].
- **MUX2 input 2 ports:** 8-bit, input, denoted as MUX2.In2, connected to DR.
- **MUX2 selection port:** 1 bit, input, denoted as MUX2.Sel, connected to control logic.
- **MUX2 output port:** 8-bit, output, denoted as MUX2.Out, connected to ALU.Operand2.

7) ALU

The arithmetic logic unit (ALU) is used to do the actual computation for the current instruction. It has the following port.

- **ALU operand 1 port:** 8-bit, input, denoted as ALU.Operand1, connected to Acc.
- **ALU operand 2 port:** 8-bit, input, denoted as ALU.Operand2, connected to MUX2.Out.
- **ALU enable port:** 1 bit, input, denoted as ALU.E, connected to control logic.
- **ALU mode port:** 4-bit, input, denoted as ALU.Mode, connected to control logic.
- **Current flags port**: 4-bit, input, denoted as ALU.CFlags, connected to SR.
- **ALU output port:** 8-bit, output, denoted as ALU.Out, connected to DMem. DI.
- **ALU flags port:** 4-bit, output, denoted as ALU.Flags, the Z (zero), C (carry), S (sign), O (overflow) bits, from MSB to LSB, connected to status register.

Various mode of ALU:

mode (binary)	mode (hex)	function	comments (instructions)
0000	0	Out = Operand1 + Operand2	
0001	1	Out = Operand1 - Operand2	
0010	2	Out = Operand1	for MOVAM
0011	3	Out = Operand2	for MOVMA and MOVIA
0100	4	Out = Operand1 AND Operand2	
0101	5	Out = Operand1 OR Operand2	
0110	6	Out = Operand1 XOR Operand2	
0111	7	Out = Operand2 - Operand1	
1000	8	Out = Operand2 + 1	
1001	9	Out = Operand2 - 1	
1010	A	Out = (Operand2 << Operand1 [2:0]) | (Operand2 >> 8 - Operand1 [2:0])	for ROTATEL
1011	B	Out = (Operand2 >> Operand1 [2:0]) | (Operand2 << 8 - Operand1 [2:0])	for ROTATER
1100	C	Out = Operand2 << Operand1 [2:0]	logical shift left
1101	D	Out = Operand2 >> Operand1 [2:0]	logical shift right
1110	E	Out = Operand2 >>> Operand1 [2:0]	arithmetic shift right
1111	F	Out = 0 - Operand2	2's complement

8) CONTROL UNIT DESIGN

Control signal is derived from the current state and current instruction. The control logic component is purely combinational. There are in total 12 control signals, listed as follows.

- **PC. E:** enable port of program counter (PC);
- **Acc.E:** enable port of accumulator (Acc);
- **SR. E:** enable port of status register (SR);
- **IR. E:** enable port of instruction register (IR);
- **DR. E:** enable port of data register (DR);
- **PMem.E:** enable port of program memory (PMem);
- **DMem.E:** enable port of data memory (DMem);
- **DMem.WE:** write enable port of data memory (DMem);
- **ALU.E:** enable port of ALU;
- **ALU.Mode**: mode selection port of ALU;
- **MUX1.Sel:** selection port of MUX1;
- **MUX2.Sel**: selection port of MUX2;

Table showing Generation of control signals for various operation:

Ins	Stage	PC .E	Acc. E	SR. E	IR. E	DR. E	PMem .E	DMem .E	DMem .WE	ALU. E	ALU. Mode	MUX 1.Sel	MUX 2.Sel
NOP (0000)	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	0	0	0	0	0	0	0	0	x	1	x
GOTO (0001)	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	0	0	0	0	0	0	0	0	x	0	x
ALU M Type (001x)	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
	DECODE	0	0	0	0	1	0	1	0	0	x	x	x
	EXECUTE	1	$IR[8]$	1	0	0	0	$\overline{IR[8]}$	$\overline{IR[8]}$	1	$IR[7:4]$	1	1
JZ, JC, JS, JO (01xx)	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	0	0	0	0	0	0	0	0	x	SR*	x
ALU I Type (1xxx)	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	1	1	0	0	0	0	0	1	$IR[10:8]$	1	0

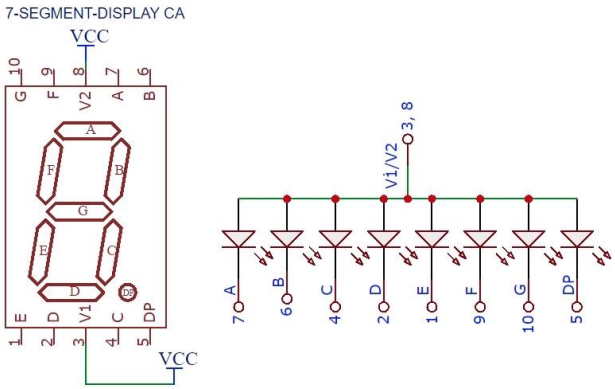
7-Segment Display

What is a 7- segment display?

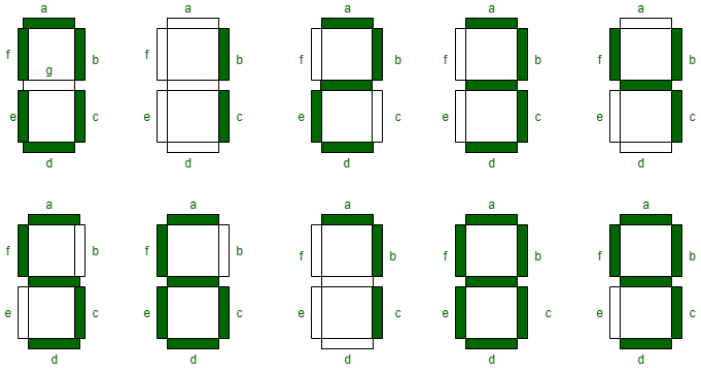
Seven segment display is an output display device that can display decimal numbers, alphabets and is an alternative to the more complex dot matrix displays. It is widely used in digital clocks, calculators, electronic meters, and other electronic devices that display numerical information. It consists of seven segments of light emitting diodes (LEDs) which is assembled in the pattern of numerical 8. LEDs emit light only when forward biased.

Working of seven segment displays:

The diode's are named from A-G as shown in the figure, seven anodes of the seven segments in a single LED are connected together to one common anode node while all cathodes are separate as shown in the following figure. DP-segment is to illuminate the dot, so we omit the DP-segment for now since it is not contributing to the number value of the seven-segment display.



To illuminate LED segments such as A-G, anodes of these segments need to be at the 'high' logic level and cathodes should be at the 'low' logic level. Thus, the common anode node should be high to activate a single seven-segment LED display. The logic levels of cathodes can be varying from low (illuminated) to high(un-illuminated) to show different decimal values on a single 7-segment LED display as follows.



As shown in the figure, we can display the numbers from zero to nine on the seven-segment LED display by turning on and off the seven segments of the single seven-segment LED display.

For example, to show number "1" on the LED, the segments A, F, G, E, and D are un-illuminated or their cathodes should be high, and B-C segments are illuminated or their cathodes should be low.

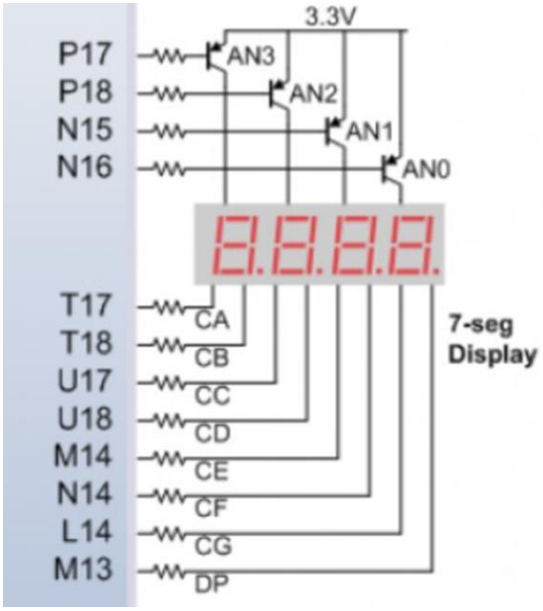
Truth Table

The truth table states which LEDs will be on and which will be at off in a 7-segment display for the decimal inputs from 0-9, here the high at cathode is taken as 1 and low at cathode is taken as 0

X	a	b	c	d	e	f	g
0 $(0000)_b$	0	0	0	0	0	0	1
1 $(0001)_b$	1	0	0	1	1	1	1
2 $(0010)_b$	0	0	1	0	0	1	0
3 $(0011)_b$	0	0	0	0	1	1	0
4 $(0100)_b$	1	0	0	1	1	0	0
5 $(0101)_b$	0	1	0	0	1	0	0
6 $(0110)_b$	0	1	0	0	0	0	0
7 $(0111)_b$	0	0	0	1	1	1	1
8 $(1000)_b$	0	0	0	0	0	0	0
9 $(1001)_b$	0	0	0	0	1	0	0

4-Digit 7-Segment Display

To display 4 different numbers on the 4-digit seven-segment LED display, we have to control the cathodes (CA-CG) of the four seven-segment LEDs separately by activating the four seven-segment LEDs at different times. For example, when we activate LED 1 by driving AN1 high and the other three LEDs (LED 2, LED 3, LED 4) are deactivated (AN2, AN3, and AN4 not driven), the cathode pattern (A-G) will be used for displaying numbers on LED 1. Similarly, LED 2 - LED 4 can be displayed by using the same way at different times.



Code:

```
module temperature_control_unit(
input [6:0]temp,
input reset,
output reg alarm);

initial begin
alarm=1'b0;
end
    always @ ( temp | reset )
    begin
    //if reset button is pushed to stop the alarm on
    if(reset)
        alarm=1'b0;
    //checking the temperature is in allowable range or not
    else if((temp<7'd97)|(temp>7'd100))
        alarm = 1'b1;
    else
        alarm = 1'b0 ;
    end
endmodule

module heart_beat(input [7:0]current_rate,ur,lr,input reset, output reg alarm);
    reg [7:0]upper_rate,lower_rate;
    //setting the ranges of heart beat
    always@*
    begin
    upper_rate=ur;  //uppre range of heart beat
    lower_rate=lr;  //lower range of heart beat
    end
    //checking the heatbeat to generate the alarm signal

    always@(current_rate or reset)
    begin
    //if reset button is pushed to stop the alarm on
    if(reset) begin
        alarm=1'b0;
        end
    else if((current_rate<lower_rate)|(current_rate>upper_rate))
        alarm=1'b1;
    else
        alarm=1'b0;
    end
endmodule
```

Code of Micro Controller:

```
`timescale 1ns / 1ps
module ALU(  input [7:0] Operand1,Operand2,
  input E,
  input [3:0] Mode,
  input [3:0] CFlags,
  output  [7:0] Out,
  output  [3:0] Flags
// the Z (zero), C (carry), S (sign),O (overflow) bits, from MSB to LSB, connected to status register
   );
  wire Z,S,O;
  reg CarryOut;
  reg [7:0] Out_ALU;
  always @(*)
  begin
  case(Mode)
  4'b0000: {CarryOut,Out_ALU} = Operand1 + Operand2;
  4'b0001: begin Out_ALU = Operand1 -  Operand2;
  CarryOut = !Out_ALU[7];
  end
  4'b0010: Out_ALU = Operand1;
  4'b0011: Out_ALU = Operand2;
  4'b0100: Out_ALU = Operand1 & Operand2;
  4'b0101: Out_ALU = Operand1 | Operand2;
  4'b0110: Out_ALU = Operand1 ^ Operand2;
  4'b0111: begin
  Out_ALU = Operand2 - Operand1;
  CarryOut = !Out_ALU[7];
  end
  4'b1000: {CarryOut,Out_ALU} = Operand2 + 8'h1;
  4'b1001: begin
  Out_ALU = Operand2 - 8'h1;
  CarryOut = !Out_ALU[7];
  end
  4'b1010: Out_ALU = (Operand2 << Operand1[2:0])| ( Operand2 >> (3'd8-Operand1[2:0]));
  4'b1011: Out_ALU = (Operand2 >> Operand1[2:0])| ( Operand2 << (3'd8-Operand1[2:0]));
  4'b1100: Out_ALU = Operand2 << Operand1[2:0];
  4'b1101: Out_ALU = Operand2 >> Operand1[2:0];
  4'b1110: Out_ALU = Operand2 >>> Operand1[2:0];
  4'b1111: begin
 Out_ALU = 8'h0 - Operand2;
 CarryOut = !Out_ALU[7];
 end
 default: Out_ALU = Operand2;
 endcase
 end
 assign O = Out_ALU[7] ^ Out_ALU[6];
 assign Z = (Out_ALU == 0)? 1'b1 : 1'b0;
 assign S = Out_ALU[7];
 assign Flags = {Z,CarryOut,S,O};
 assign Out = Out_ALU;
endmodule


module DMem( input clk,
     input E, // Enable port
     input WE, // Write enable port
     input [3:0] Addr, // Address port
     input [7:0] DI, // Data input port
     output [7:0] DO // Data output port
    );
```

```
reg [7:0] data_mem [15:0];
always @(posedge clk)
begin
 if(E==1 && WE ==1)
  data_mem[Addr] <= DI;
end
 assign DO = (E ==1 )? data_mem[Addr]:0;
endmodule

//PC Adder
module adder( input [7:0] In,
      output [7:0] Out
    );

assign Out = In + 1;
endmodule

module MUX1( input [7:0] In1,In2,
     input Sel,
     output [7:0] Out
    );

assign Out = (Sel==1)? In1: In2;
endmodule




module PMem( input clk,
   input E, // Enable port
  input [7:0] Addr, // Address port
  output [11:0] I, // Instruction port
  // 3 special ports are used to load program to the memory
  input LE, // Load enable port
  input[7:0] LA, // Load address port
  input [11:0] LI//Load instruction port
   );
 reg [11:0] Prog_Mem[255:0] ;
always @(posedge clk)
 begin
 if(LE == 1) begin
 Prog_Mem[LA] <= LI;
 end
 end
 assign I =  (E == 1) ?  Prog_Mem[Addr]: 0 ;
endmodule


module Control_Logic( input[1:0] stage,
  input [11:0] IR,
  input [3:0] SR,
 output reg PC_E,Acc_E,SR_E,IR_E,DR_E,PMem_E,DMem_E,DMem_WE,ALU_E,MUX1_Sel,MUX2_Sel,PMem_LE,
  output reg [3:0] ALU_Mode
    );
 parameter LOAD = 2'b00,FETCH = 2'b01, DECODE = 2'b10, EXECUTE = 2'b11;
 always @(*)
 begin
 PMem_LE = 0;
 PC_E = 0;
```

```
Acc_E = 0;
SR_E = 0;
IR_E = 0;
DR_E = 0;
PMem_E = 0;
DMem_E = 0;
DMem_WE = 0;
ALU_E  =0;
ALU_Mode = 4'd0;
MUX1_Sel = 0;
MUX2_Sel = 0;
if(stage== LOAD )
begin
PMem_LE = 1;
PMem_E = 1;
end
else if(stage== FETCH ) begin
IR_E = 1;
PMem_E = 1;
end
else if(stage== DECODE ) begin
if( IR[11:9] == 3'b001)
begin
DR_E = 1;
DMem_E = 1;
end
else
begin
DR_E = 0;
DMem_E = 0;
 end
 end
 else if(stage== EXECUTE )
 begin
 if(IR[11]==1) begin // ALU I-type
 PC_E = 1;
 Acc_E = 1;
 SR_E = 1;
 ALU_E = 1;
 ALU_Mode = IR[10:8];
 MUX1_Sel = 1;
 MUX2_Sel = 0;
 end
 else if(IR[10]==1) // JZ, JC,JS, JO
 begin
 PC_E = 1;
 MUX1_Sel = SR[IR[9:8]];
 end
 else if(IR[9]==1)
 begin
 PC_E = 1;
 Acc_E = IR[8];
 SR_E = 1;
 DMem_E = !IR[8];
 DMem_WE = !IR[8];
 ALU_E = 1;
 ALU_Mode = IR[7:4];
 MUX1_Sel = 1;
 MUX2_Sel = 1;
 end
```

```
 else if(IR[8]==0)
 begin
 PC_E = 1;
 MUX1_Sel = 1;
 end
 else
 begin
 PC_E = 1;
 MUX1_Sel = 0;
 end
 end
 end
endmodule



// The microcontroller (similar to Microchip PIC12, simplified extensively, not compatible in
// instruction set) in this project is a 3 cycle nonpipelined
module MicroController(input clk,rst
    );
  parameter LOAD = 2'b00,FETCH = 2'b01, DECODE = 2'b10, EXECUTE = 2'b11;
  reg [1:0] current_state,next_state;
  reg [11:0] program_mem[9:0];
// load instruction peripherals
  reg load_done;
  reg[7:0] load_addr;
  wire[11:0] load_instr;
//peripherals
  reg[7:0] PC,  DR, Acc;
  reg [11:0] IR;     // INSTRUCTION REGISTER

  reg [3:0]  SR;    // Z C S O
//enable signals
  wire PC_E,Acc_E,SR_E,DR_E,IR_E;
// clear signals
reg PC_clr,Acc_clr,SR_clr,DR_clr,IR_clr;
wire [7:0] PC_updated,DR_updated;
wire[11:0] IR_updated;
wire[3:0] SR_updated;
wire PMem_E,DMem_E,DMem_WE,ALU_E,PMem_LE,MUX1_Sel,MUX2_Sel;
wire [3:0] ALU_Mode;
wire [7:0] Adder_Out;
wire [7:0] ALU_Out,ALU_Oper2;
// LOAD instruction memory
initial
begin
$readmemb("program_me.dat", program_mem,0,9);
end
// ALU
ALU ALU_unit( .Operand1(Acc),
     .Operand2(ALU_Oper2),
   .E(ALU_E),
   .Mode(ALU_Mode),
   .CFlags(SR),
     .Out(ALU_Out),
     .Flags(SR_updated) // the Z (zero), C (carry), S (sign),O (overflow) bits, from MSB to LSB, connected to status register
 );
// MUX2
MUX1 MUX2_unit( .In2(IR[7:0]),.In1(DR),
  .Sel(MUX2_Sel),
  .Out(ALU_Oper2)
```

```
    );
// Data Memory
DMem DMem_unit( .clk(clk),
 .E(DMem_E), // Enable port
 .WE(DMem_WE), // Write enable port
 .Addr(IR[3:0]), // Address port
 .DI(ALU_Out), // Data input port
 .DO(DR_updated) // Data output port
   );
// Program memory
PMem PMem_unit( .clk(clk),
 .E(PMem_E), // Enable port
 .Addr(PC), // Address port
 .I(IR_updated), // Instruction port
 // 3 special ports are used to load program to the memory
 .LE(PMem_LE), // Load enable port
 .LA(load_addr), // Load address port
 .LI(load_instr)//Load instruction port
  );
// PC ADder
adder PC_Adder_unit( .In(PC),
  .Out(Adder_Out)
   );
// MUX1
MUX1 MUX1_unit( .In2(IR[7:0]),.In1(Adder_Out),
 .Sel(MUX1_Sel),
 .Out(PC_updated)
   );
// Control logic
Control_Logic Control_Logic_Unit( .stage(current_state),
 .IR(IR),
 .SR(SR),
 .PC_E(PC_E),
 .Acc_E(Acc_E),
 .SR_E(SR_E),
 .IR_E(IR_E),
 .DR_E(DR_E),
 .PMem_E(PMem_E),
 .DMem_E(DMem_E),
 .DMem_WE(DMem_WE),
 .ALU_E(ALU_E),
 .MUX1_Sel(MUX1_Sel),
 .MUX2_Sel(MUX2_Sel),
 .PMem_LE(PMem_LE),
 .ALU_Mode(ALU_Mode)
   );
// LOAD
always @(posedge clk)
begin
if(rst==1) begin   // reset the data in load instruction
load_addr <= 0;
load_done <= 1'b0;
end
else if(PMem_LE==1)
begin
load_addr <= load_addr + 8'd1;
if(load_addr == 8'd9)
begin
```

```
load_done <= 1'b1;
end
else
begin
load_done <= 1'b0;
end
end
end
assign load_instr = program_mem[load_addr];
// next state
always @(posedge clk)
begin
if(rst==1)
current_state <= LOAD;
else
current_state <= next_state;
end
always @(*)
begin
PC_clr = 0;
Acc_clr = 0;
SR_clr = 0;
DR_clr = 0;
IR_clr = 0;
case(current_state)
LOAD: begin
if(load_done==1) begin
next_state = FETCH;
PC_clr = 1;
Acc_clr = 1;

SR_clr = 1;
DR_clr = 1;
IR_clr = 1;
end
else
next_state = LOAD;
end
FETCH: begin
next_state = DECODE;
end
DECODE: begin
next_state = EXECUTE;
end
EXECUTE: begin
next_state = FETCH;
end
endcase
end
always @(posedge clk)
begin
if(rst==1)
begin
PC <= 8'd0;
Acc <= 8'd0;
SR <= 4'd0;
end
else
begin
```

```
 if(PC_E==1'd1)
 PC <= PC_updated;
 else if (PC_clr==1)
 PC <= 8'd0;
 if(Acc_E==1'd1)
 Acc <= ALU_Out;
 else if (Acc_clr==1)
 Acc <= 8'd0;
 if(SR_E==1'd1)
 SR <= SR_updated;
 else if (SR_clr==1)
 SR <= 4'd0;
 end
 end

 // 2 programmer invisible register
 always @(posedge clk)
 begin
 if(DR_E==1'd1)
 DR <= DR_updated;
 else if (DR_clr==1)
 DR  <= 8'd0;
 if(IR_E==1'd1)
 IR <= IR_updated;
 else if(IR_clr==1)
 IR <= 12'd0;
 end
endmodule
```

Code for 1-digit 7-segment controller

```
`timescale 1ns / 1ps
module bcd(a,b,c);
input [3:0]a;  // the number to be displayed
wire [3:0]a;
output [6:0]b; // for the patterns, in terms of 7-Seg "abcdefg"
reg [6:0]b;
output [3:0]c;
reg [6:0]c;
always @(a)
begin
if(a==0)begin
b = 7'b0000001;   // Here in common anode a/b/c/d/e/f are on i.e "0" and
                  //g is off i.e "1"
c = 4'b0000;      // for this pattern we have output as zero
end
else if(a==1)begin
b = 7'b1001111;
c = 4'b0001;  //for this pattern we have output as one
end
else if(a==2)begin
b = 7'b0010010;
c = 4'b0010;   //for this pattern we have output as two
end
else if(a==3)begin
b = 7'b0000110;
c = 4'b0011;   //for this pattern we have output as three
end
else if(a==4)begin
b = 7'b1001100;
c = 4'b0100;   //for this pattern we have output as four
end
else if(a==5)begin
b = 7'b0100100;
c = 4'b0101;  //for this pattern we have output as five
end
else if(a==6)begin
b = 7'b0100000;
c = 4'b0110;  //for this pattern we have output as six
end
else if(a==7)begin
b = 7'b0001111;
c = 4'b0111;  //for this pattern we have output as seven
end
else if(a==8)begin
b = 7'b0000000;
c = 4'b1000;  //for this pattern we have output as eight
end
else if(a==9)begin
b = 7'b0000100;
c = 4'b1001;  //for this pattern we have output as nine
end
```

```
else begin
b = 7'bxxxxxxx;
c = 7'bxxxxxxx;
end

end
endmodule
```

Code for 4-digit 7-segment Display:

```
`timescale 1ns / 1ps
// 4-digit seven-segment LED display controller on Basys  3 FPGA
module Seven_segment_LED_Display_Controller(
    input clock_100Mhz, // 100 Mhz clock source on Basys 3 FPGA
    input reset, // reset
    output reg [3:0] Anode_Activate, // anode signals of the 7-segment LED display
    output reg [6:0] LED_out//  patterns of the 7-segment LED display
    );
    reg [26:0] one_second_counter; // counter for generating 1 second clock enable
    wire one_second_enable;// one second enable for counting numbers
    reg [15:0] displayed_number; // 4-digit number to be displayed
    reg [3:0] LED_BCD;
    reg [19:0] refresh_counter; // 20-bit for creating 10.5ms refresh period or 380Hz refresh rate
             // the first 2 MSB bits for creating 4 LED-activating signals with 2.6ms digit period
    wire [1:0] LED_activating_counter;
                 // count     0    ->  1  ->  2  ->  3
               // activates    LED1    LED2   LED3   LED4
               // and repeat
    always @(posedge clock_100Mhz or posedge reset)
    begin
        if(reset==1)
            one_second_counter <= 0;
        else begin
            if(one_second_counter>=99999999)
                one_second_counter <= 0;
            else
                one_second_counter <= one_second_counter + 1;
        end
    end
    assign one_second_enable = (one_second_counter==99999999)?1:0;
    always @(posedge clock_100Mhz or posedge reset)
    begin
        if(reset==1)
            displayed_number <= 0;
        else if(one_second_enable==1)
            displayed_number <= displayed_number + 1;
    end
    always @(posedge clock_100Mhz or posedge reset)
    begin
        if(reset==1)
            refresh_counter <= 0;
        else
            refresh_counter <= refresh_counter + 1;
    end
```

```
    assign LED_activating_counter = refresh_counter[19:18];
    // anode activating signals for 4 LEDs,
    always @(*)
        case(LED_activating_counter)
        2'b00: begin
            Anode_Activate = 4'b0111;
            // activate LED1 and Deactivate LED2, LED3, LED4
            LED_BCD = displayed_number/1000;
            // the first digit of the 16-bit number
              end
        2'b01: begin
            Anode_Activate = 4'b1011;
            // activate LED2 and Deactivate LED1, LED3, LED4
            LED_BCD = (displayed_number % 1000)/100;
            // the second digit of the 16-bit number
              end
        2'b10: begin
            Anode_Activate = 4'b1101;
            // activate LED3 and Deactivate LED2, LED1, LED4
            LED_BCD = ((displayed_number % 1000)%100)/10;
            // the third digit of the 16-bit number
               end
        2'b11: begin
            Anode_Activate = 4'b1110;
            // activate LED4 and Deactivate LED2, LED3, LED1
            LED_BCD = ((displayed_number % 1000)%100)%10;
            // the fourth digit of the 16-bit number
               end
        endcase
    end
    always @(*)
    begin
        case(LED_BCD)
        // Cathode patterns of the 7-segment LED display
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
        4'b0011: LED_out = 7'b0000110; // "3"
        4'b0100: LED_out = 7'b1001100; // "4"
        4'b0101: LED_out = 7'b0100100; // "5"
        4'b0110: LED_out = 7'b0100000; // "6"
        4'b0111: LED_out = 7'b0001111; // "7"
        4'b1000: LED_out = 7'b0000000; // "8"
        4'b1001: LED_out = 7'b0000100; // "9"
        default: LED_out = 7'b0000001; // "0"
        endcase
    end
endmodule
```

Test Bench's:

```
module tb_tc();
reg [6:0]temp;
reg rst ;
wire alarm;
temperature_control_unit tcu(temp,rst,alarm);
initial begin
rst=0;
temp=7'd95; #10
temp=7'd97; #10
temp=7'd104; #10
rst =1'b0; #10
temp=7'd99; #10
temp=7'd100; #10
temp=7'd80; #10
rst=1'b0; #10
$stop;
end
endmodule
```

```
 module tb_hb();
    reg [7:0]cr,upper,lower;
    reg reset ;
    wire alarm;
    heart_beat hb(cr,upper,lower,reset,alarm);
    initial begin
    upper=8'd90; lower=8'd70; reset=0;
    cr=8'd72; #10
    cr=8'd95; #10
    cr=8'd65; #10
    reset=1;#10
    upper=8'd60; lower=8'd40; reset=0;
    cr=8'd35; #10
    cr=8'd50; #10
    cr=8'd75; #10
    reset=1;
    $stop;
end
endmodule
```

```
//test bench

module MCU_tb;
reg clk;
reg rst;
// Instantiate the Unit Under Test (UUT)
MicroController uut (
.clk(clk),
.rst(rst)
);
initial begin
// Initialize Inputs
rst = 1;
// Wait 100 ns for global reset to finish
#100;
rst = 0;
end
initial begin
clk = 0;
forever #10 clk = ~clk;
end
endmodule
```

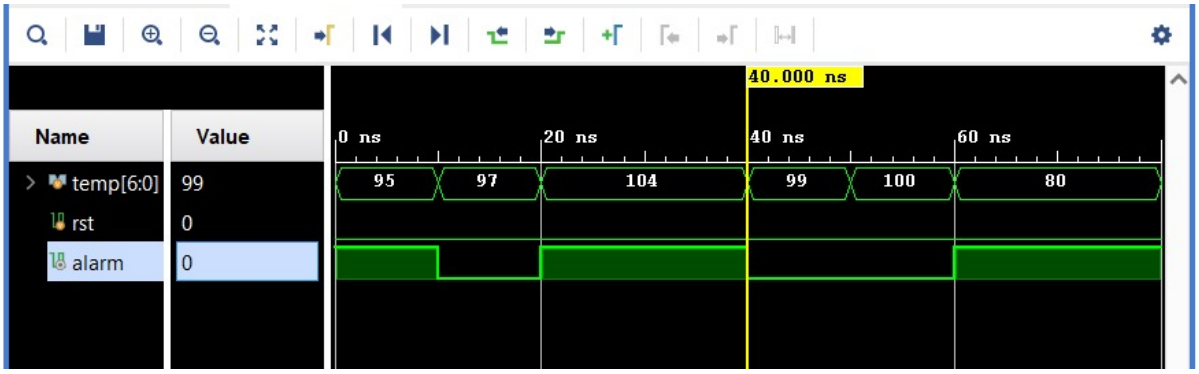
D:/Verilog Programming/n

```
1  0000_0000_0000
2  1011_0000_0101
3  1010_0000_0000
4  1000_0000_0111
5  1001_0000_0110
6  1111_0000_0111
7  1100_0000_0011
8  1101_0000_0101
9  1110_0000_0011
10 0001_0000_1001
```

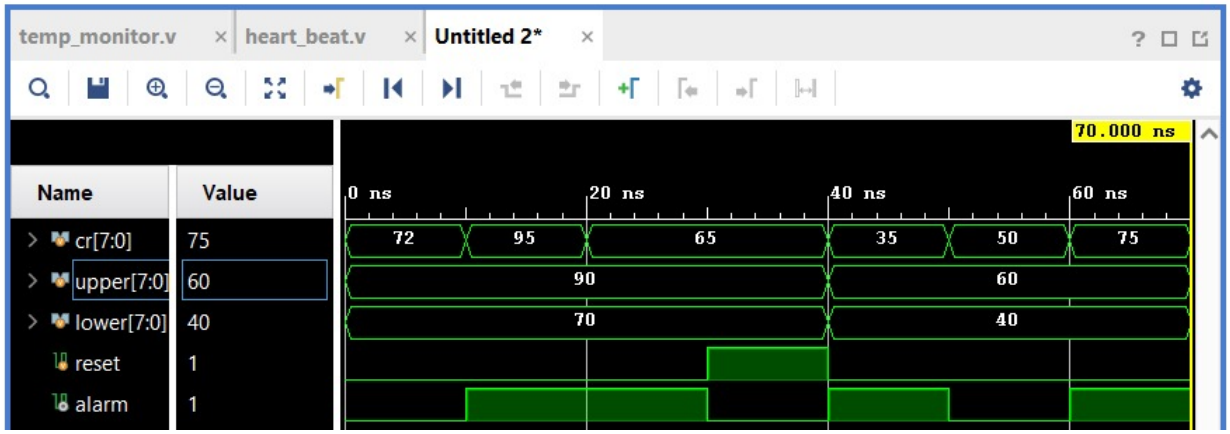
```
module BCD_to_7_B ();
reg [3:0]a;
wire [6:0]b;
wire [4:0]c;
integer i;
bcd b1(a,b,c);
initial
begin
for(i = 0;i < 10 ;i = i+1) //run loop for 0 to 9.
        begin
            a = i;
            #10; //wait for 10 ns
end
#10 $finish;
end
endmodule
```

Simulation outputs:

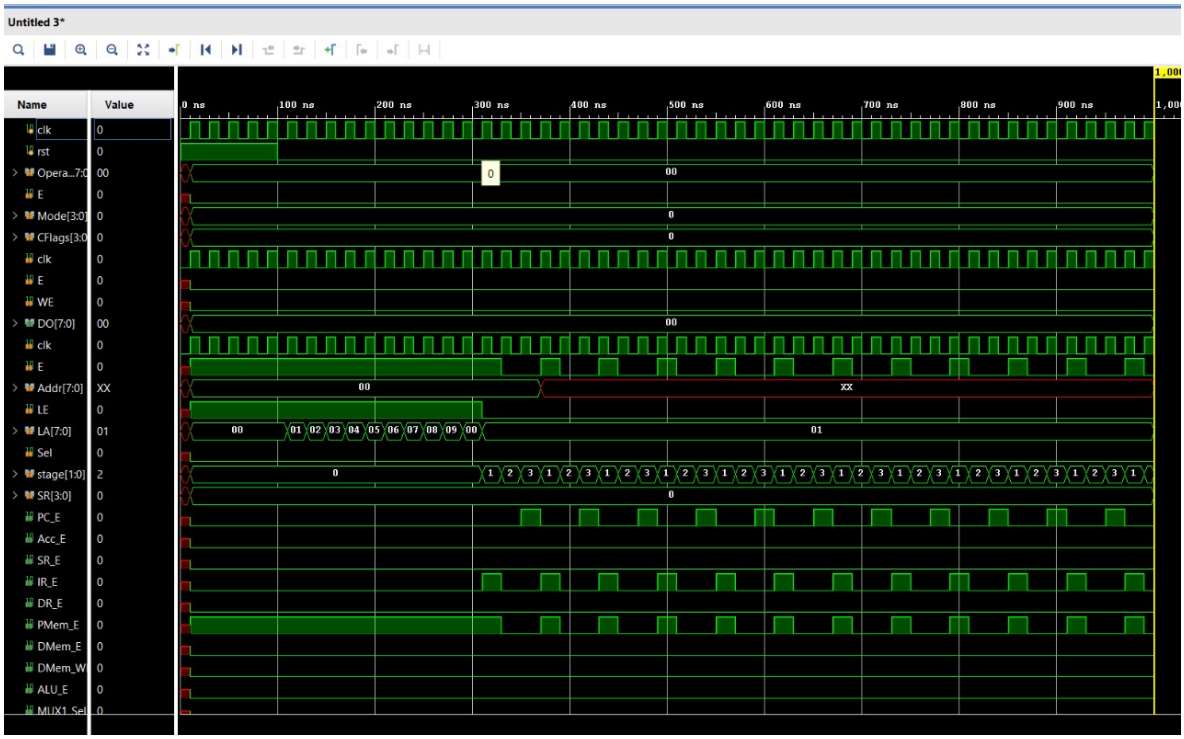
For temperature sensor



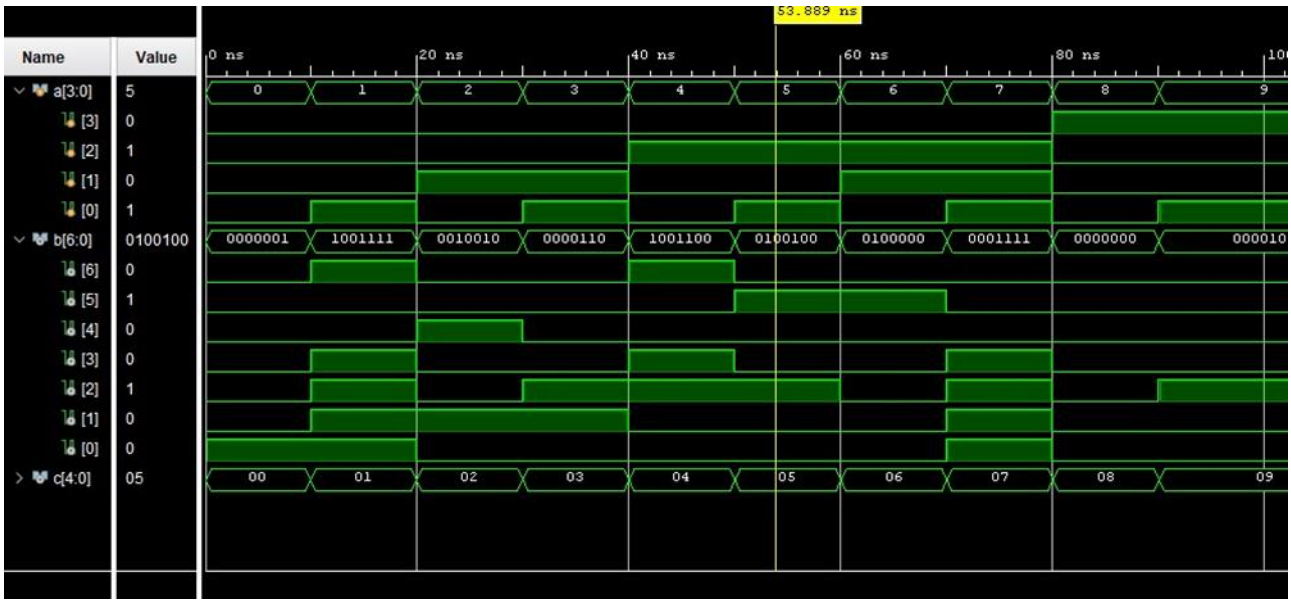
For Heart Beat sensor



For Micro Controller



For Seven Segment Display



Conclusion:

The successful implementation of the health monitoring system has been accomplished using the top-down approach. There are various components that

are being integrated. The complexed part of analysis is done over the implementation of micro-controller similar to Microchip PIC12. There are many advantages of implementing the hardware design using the micro-controller, they are:-

- The low time required for performing the operation
- It is easy to use, troubleshooting and system maintenance is simple
- At the same time, many tasks can be performed so the human effect can be saved
- The processor chip is very small and flexibility occurs
- Due to their higher integration, cost and size of the system is reduced

But the disadvantage comes with limitation of the no of instructions and its difficulty to interact with the higher power device directly. The display with the 4 digits and each digit is of seven segment. The display are used to instantly monitor the health condition of the user. There is no option of setting the temperature range of user as universal range is set though, where as different person have different level of heart beat which varies with age and the daily habits so that there is an user manual setting of the hear beat range is taken.

There are several advantages of the health monitor :-

- It can generate the alarm signal which can also trigger the message to the caretaker
- Need of human surveillance is reduced.
- Due to use of the microcontroller the observation is fast and cost effective.
- The device becomes more mobile.
- Instantaneous responses are generated.

References:

- Verilog Digital system design (Second Edition) by Z. Navabi
- https://nevonprojects.com/health-monitoring-system-using-7-segment-display-atmega-microcontroller/
- https://yg9120.github.io/2019/06/08/Control_Seven_Segment/
- https://www.egr.msu.edu/classes/ece480/capstone/spring15/group13/assets/app_note_john_foxworth.docx.pdf
- https://en.wikipedia.org/wiki/Microcontroller
- https://www.electronicshub.org/8051-microcontroller-instruction-set/
- https://www.youtube.com/watch?v=RwvUE6KutKs