Project 3: Modifying Microcontroller Architecture

by

Jordan Hayes

EGR 426 Integrated Circuit Systems Design
Section 10
Date Submitted: April 19, 2022
Instructor: Professor Parikh

Project Overview

The objective of the project is to design components of a microcontroller using behavioral functions and descriptions in VHDL. A rudimentary microcontroller architecture and implementation was provided. It is an 8-bit microprocessor, with a 9 bit RAM address (ADDR), having 512 locations in memory. There is an 8 bit data bus (DATA), two 8-bit registers (A and B), two 8-bit input and output ports, and a 3 bit Condition Code Register (CCR). A state diagram of the provision is shown below. Each command will have an RTL description of its registers' values during each state of execution.

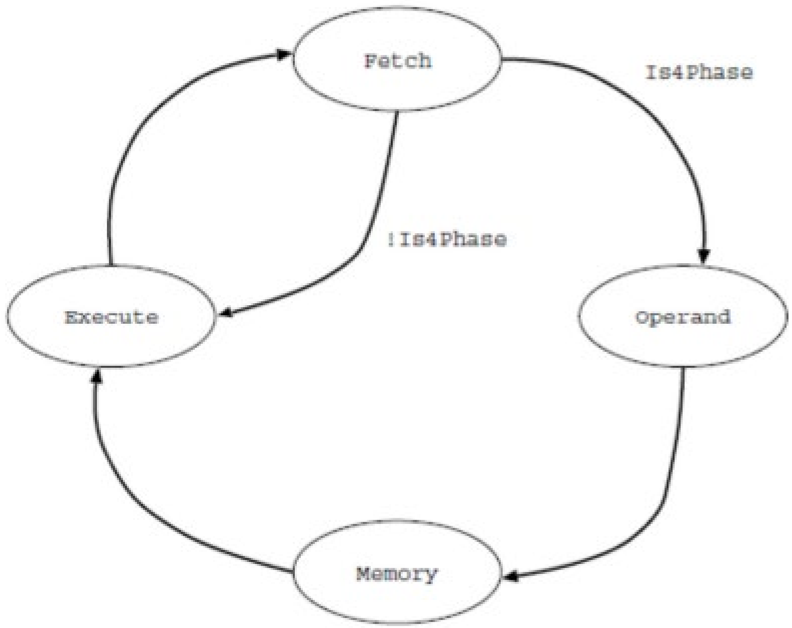


Figure 1: Controller State Machine

The figure shows that commands, stored in the IR register, can be either 2 or 4 phase. For demonstration purposes, the value of the PC counter was separated into the hundreds, tens, and ones place was output onto the left set of seven segments, the numbers stored and output in register A are shown on the first (leftmost) two seven-segments of the second set, and register B values are shown on the next two seven segments when output. A multiplexer (MUX) was designed to switch between enabling anodes of the segments every 1 ms, while the data set to be output to the segments was switched respectively.

Top-level Schematic

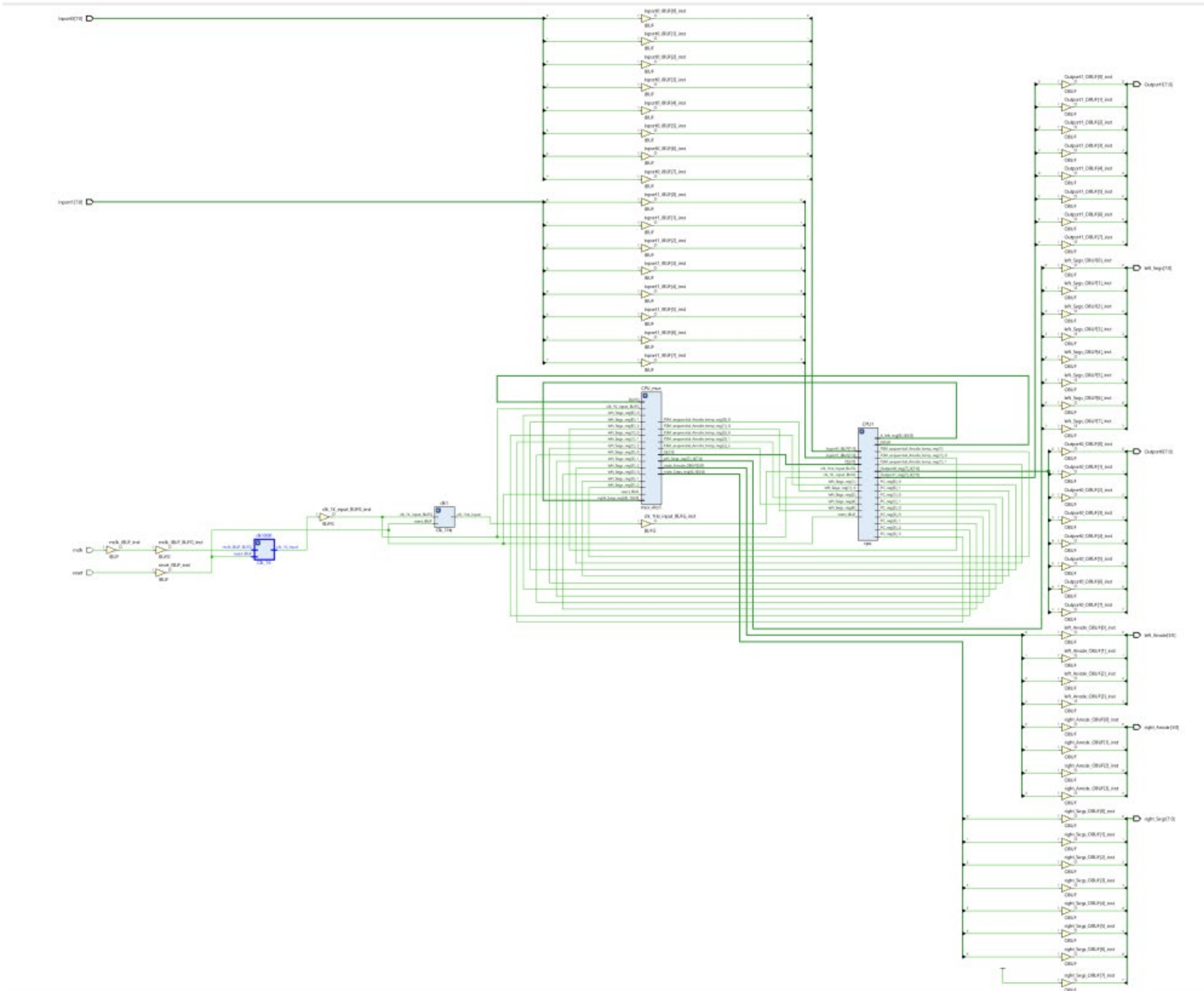


Figure 2: Top Level Schematic

A closer look at the designed portions is provided below.

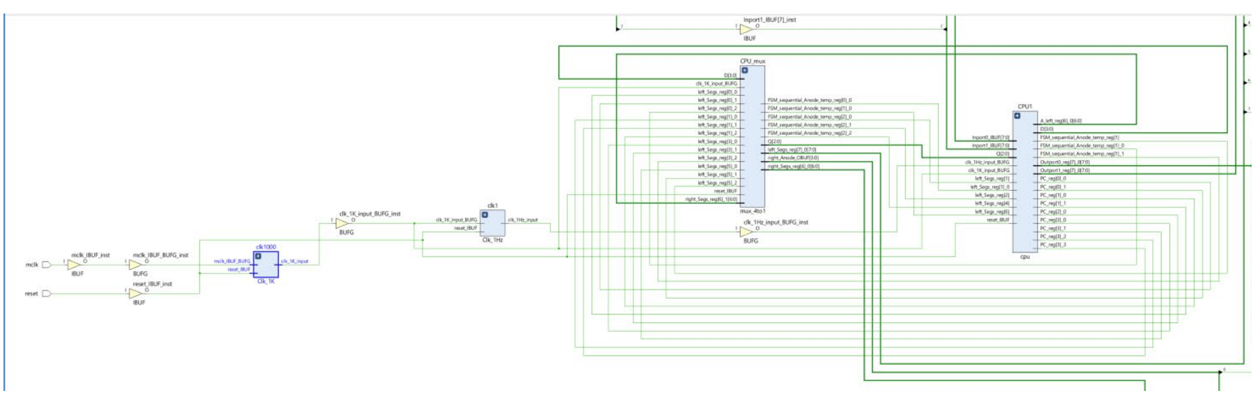


Figure 3: Top Level Schematic, Close Look

<u>Designed Functionality</u>

<u>BCD0 A/B: Binary Coded Demical</u>

This command is Binary Coded Decimal Out. The 2-phase instruction separates the data in DATA into two nibbles. The nibbles are decoded into an 8-bit number that when sent to the segments on the Boolean Board, the decimal representation of the number is shown. The command can set the data stored in the common register to be output on two of the seven segments. The last bit of the command determines which segments, A or B.

A snapshot of the simulated RAM memory is shown below.

Table 1: Snapshot of RAM BCD0

Adress Num	Op Code	Description	A_left	A_right	B_left	B_right
0	11110000	CLR A	0	0		
1	00000000	LOAD 14, A				
2	00001110	0x0E				
3	00010000	BCD0 A	0	1		
4	00000000	LOAD 15, A				
5	00001111	0x0F				
6	00010000	BCD0 A	1	5		
7	10100000	LSL A				
8	00010000	BCD0 A	2	-		
9	00000001	LOAD 16, B				
10	00010000	0x10, Page 0, 16				
11	00010001	BCD0 B	2	-	2	3
12	10000000	ADD A				
13	00010000	BCD0 A	4	-	2	3
14	00000001	0x01				
15	00010101	0x15				
16	00100011	0x23				

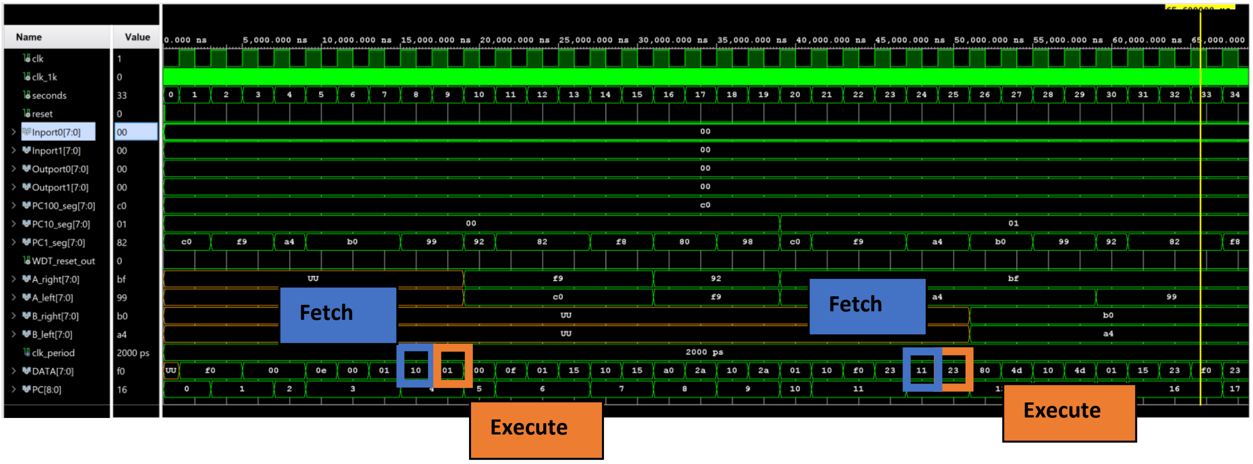


Figure 4: Annotated BCD0 Simulation

It can be seen from the figure that just before segments are activated, where *A_left* and *A_right* become defined, the DATA register receives the correct opcode for BCD (0x10), then DATA becomes the value stored in A the instruction before. The segments are known to be correct through testing and comparison with function values that correspond to the value received. When outputting the data from register B, the opcode becomes 0x11. 0x23 was stored in B, and is therefore output to *B_left* and *B_right,* respectively.

To make this functionality, a new opcode was designated for this command (0x10). When this opcode was encountered in the execute phase, a *BCD0_flag* was set high while DATA was set to A or B depending on IR(0).

```
------------------------------------------------------------------------
when "0001000" =>                         --BCD0 command either 0001000 for A
      if(IR(0) = '0') then                                    --0001001 for B
        DATA <= STD_LOGIC_VECTOR(A);
      else
        DATA <= STD_LOGIC_VECTOR(B);
      end if;

      BCD0_Write <= '1';
---------------------------------------------------
```

Figure 5: BCD0 Opcode

When *BCD_Write* is high, variables for the segment data to be selected by the MUX are set to the output of a function created to decode the nibbles specifically for the seven segments of the Boolean Board.

```vhdl
------------------------------------------------BCD0
if(BCD0_Write = '1') then
   if(IR(0) = '0') then
       A_left <= Decode_forSegs(DATA(7 downto 4));
       A_right <= Decode_forSegs(DATA(3 downto 0));
   else
       B_left <= Decode_forSegs(DATA(7 downto 4));
       B_right <= Decode_forSegs(DATA(3 downto 0));
   end if;

end if;
```

Figure 6: BCD0 Execute State, Decode Function Call

```vhdl
function Decode_forSegs(constant integer_4bit : STD_LOGIC_VECTOR(3 downto 0)) return STD_LOGIC_VECTOR is
variable dout_busToSeg : STD_LOGIC_VECTOR(7 downto 0);
begin
  case (integer_4bit) is
        when "0000" => dout_busToSeg := "11000000";        --0
        when "0001" => dout_busToSeg := "11111001";        --1
        when "0010" => dout_busToSeg := "10100100";        --2
        when "0011" => dout_busToSeg := "10110000";        --3
        when "0100" => dout_busToSeg := "10011001";        --4
        when "0101" => dout_busToSeg := "10010010";        --5
        when "0110" => dout_busToSeg := "10000010";        --6
        when "0111" => dout_busToSeg := "11111000";        --7
        when "1000" => dout_busToSeg := "10000000";        --8
        when "1001" => dout_busToSeg := "10011000";        --9
        when others => dout_busToSeg := "10111111";        --when more than 9, display a dash
    end case;

    return dout_busToSeg;
end function;
```

Figure 7: BCD0 Decoding Function

The function shows that when a number greater than decimal '9' is entered, the output on the segments will be a dash (-).

RTL

Fetch
IR = MEM[PC]

Execute
PC = PC + 1
BCD0_flag = 1
DATA = R
R_segs = Decode_forSegs(DATA)

<u>DEB0 R: Debounce 0/1 A/B</u>

This instruction may either take the form “DEB 0, R” or “DEB 1, R” where “R” stands for either register A or B. This instruction returns the debounced status of Input #0 bit 0 (for DEB 0) or Input #0 bit 1 (for DEB 1). If this bit has remained at a logic 0 for the last 3 seconds, the DEB instruction should set the target register (A or B) to 1, otherwise, the target register should be set to 0.

A snapshot of the simulated RAM memory is shown below.

Table 2: Snapshot of RAM DEB0 R

Adress Num	Op Code	Description	A_left	A_right	B_left	B_right
0	11110000	CLR A				
1	00000000	LOAD 24, A				
2	00001101	0x18 (24)				
3	00100000	DEB0 A				
4	00010000	BCD0 A	0	1		
5	00000001	LOAD 14, B				
6	00001110	0x0E				
7	11110000	CLR A				
8	11110000	CLR A				
9	11110000	CLR A				
10	00010001	BCD0 B				
11	00100011	DEB1 B				
12	00010001	BCD0 B			0	0
13	00000000	0x00				
14	00000000	0x00				

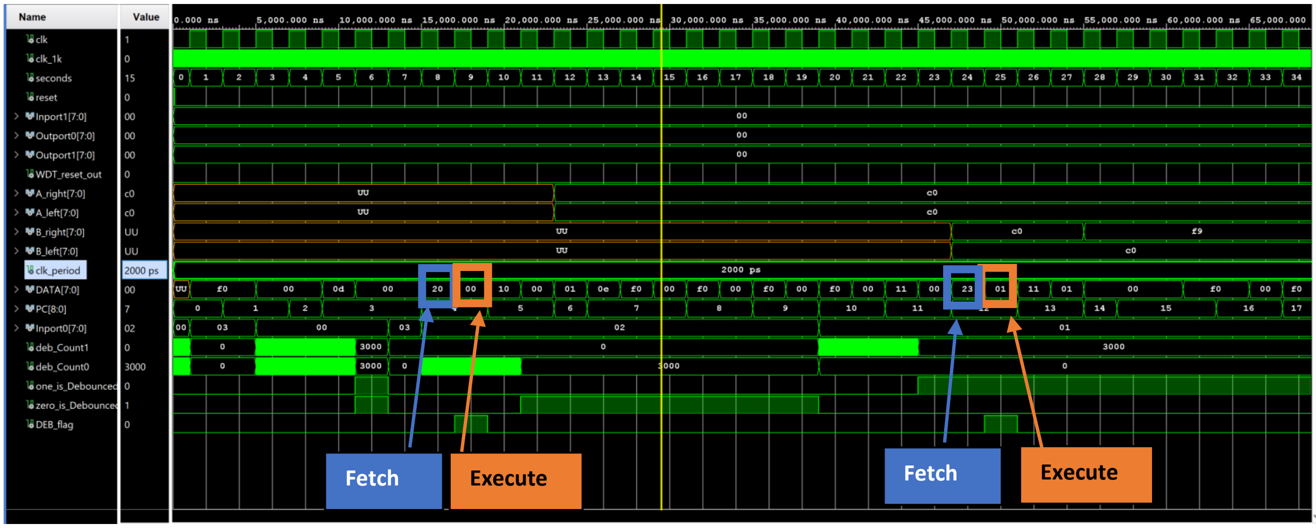


Figure 8: Annotated DEB Simulation

The simulation above shows the behavior of the function. At reset, when the inputs are both zero, the *deb_Count* timers accumulate. When switches are turned to logic '1', the timers reset to zero and do not accumulate. The designated opcode for the command is (0x20), received through the DATA vector. In the first DEB call, *Inport0(0)* is checked, but the switch was not yet debounced. DATA becomes 0x00 in the Execute state. BCD0 is called for the register and segment codes for zero are shown. The next time DEB is used, *Inport0(1)* is fully debounced. DATA becomes 0x01, and BCD0 B is called to output the result on the B segments. The simulation shows that the system will not output 1 for DEB unless the correct switch is checked and is fully debounced, both switches can be checked, and both registers can be assigned using different opcode values.

RTL

Fetch

IR = MEM[PC]
IR[]

Execute

PC = PC + 1
DEB_flag = 1
DATA = 0/1
R = DATA

BZ: Branch-if-Zero

This 4-phase instruction transfers execution to a new location in memory (given in the second byte, just like for LOAD/STOR) only if the Z bit is a logic 1. The Z bit is a part of the 3-bit CCR, and is set when the result of the ALU operation is 0. All 512 memory locations must be reachable with this instruction. The main objective for this function is to change PC, the value of the 9-bit register that stores the address of the next instruction to execute.

Table 3: Snapshot of RAM for BZ p

Adress Num	Op Code	Description	PC
0	11110000	CLR A	0
1	00000000	LOAD 21, A	1
2	00010101	0x15 (21)	2
3	00010000	BCD0 A	3
4	01100000	BZ, Page 0	4
5	00001110	0X0E, 14	5
6	00000000	load 22, A	6
7	00010110	0X16	
8	00010000	BCD0 A	
9	10100000	LSL A	
10	01100000	BZ	
11	00000001	0x01	
12	00010000	BCD0 A	
13	00000001	load 23, B	
14	00010111	0x17	14
15	00010001	BCD0 B	15
16	00011111	ADD A	16
17	00010000	BCD0 A	17
18	11110000	CLR A	18
19	01100001	BZ Page 1	19
20	00000001	BZ,	257
21	00000001	0x01	
22	00010101	0x01	
23	00100011	0x15 (21)	

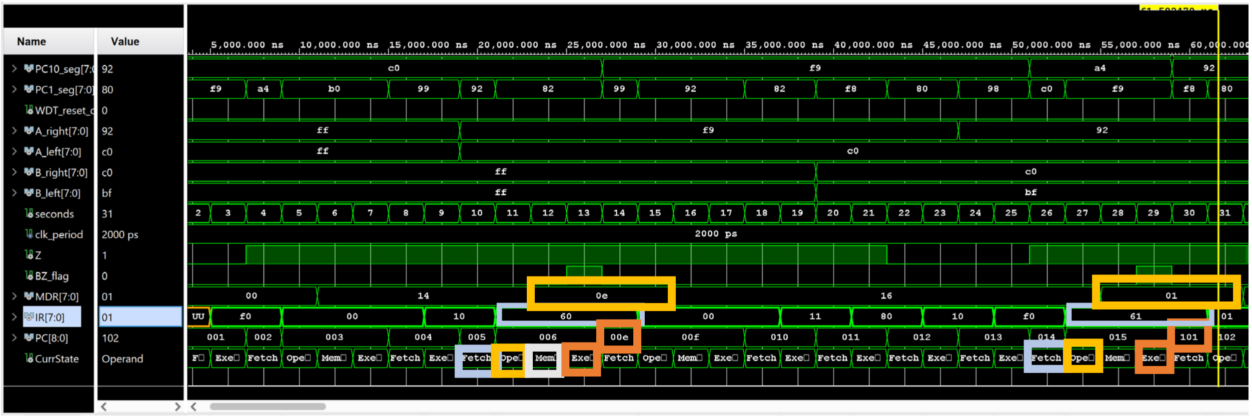


Figure 9: Annotated BZ Simulation, Branch, Page 0 and 1

It can be seen from the figure when PC = 5, the CPU is in Fetch, shown in light-blue. The opcode (0x60) is set to IR and the PC is incremented; the CPU goes to Operand, in yellow. In this state, MDR is set to the next value in memory. The Memory state is skipped for this instruction because the section of code is used for STOR commands (white).

```
case CurrState is
     when Fetch | Operand => DATA <= RAM_DATA_OUT;

     --skip MEM when branching, this is for STOR
     when Memory => if(not(IR(7 downto 2) = "011000")) then

                         if(IR(0) = '0') then
                              DATA <= STD_LOGIC_VECTOR(A);
                         else
                              DATA <= STD_LOGIC_VECTOR(B);
                          end if;

                    end if;

     when Execute => case IR(7 downto 1) is
                         when "1000000"
```

When the PC reaches the Execute state for this command, shown in orange, PC is reassigned to the operation shown below. This is due to the LSB of IR being the page reference for the Branch command, and MDR being the address of the page to be moved to. This only happens if the Z bit is high. Figure 9 also shows that the second page (Page 1) can be reached, as PC becomes too large to represent by only 8 bits. IR is set to 0x61 instead of 0x60, indicating page 1, and contributing a value of 256 to the new PC value. MDR is set to 0x01, so the result of the operation below is 257. The operation is shown below.

```
---------------------------------------------BZ
if((BZ_flag = '1') and (Z = '1')) then
        PC <= unsigned(IR(0) & MDR);
end if;
```

Instructions were changed so that the Z bit would no longer be set, and the change in PC never occurs even though the IR becomes the correct opcode and MDR is assigned. The CPU reaches the execute phase, sees the Z bit is low, and does not affect PC. The values during execution show this well.

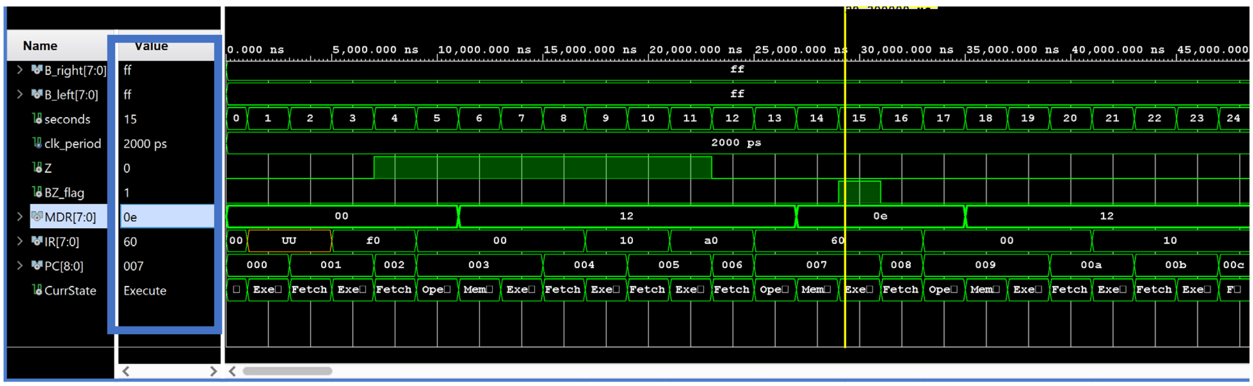
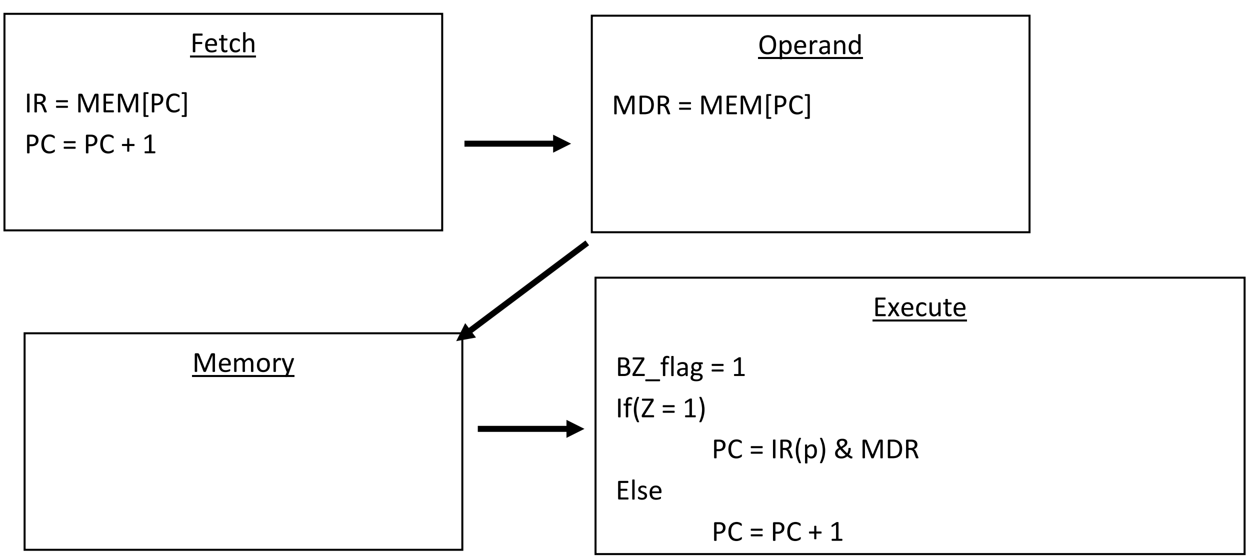


Figure 10: Annotated BZ Simulation, No Branch

RTL



<u>CLRWDT: Watchdog Reset</u>

This instruction takes no operands. If this instruction is not executed at least every 10 seconds (or more frequently), the processor's RESET signal is asserted then negated so that the processor begins executing instructions from address 0. This instruction requires the creation of a Watchdog (WDT) peripheral.

A snapshot of the simulated RAM memory is shown below.

Table 4: Snapshot of RAM for WDTCLR

Adress Num	Op Code	Description
0	11110000	CLR A
1	01000000	WDTCLR
2	00000000	LOAD 15, A
3	00001111	0x0F, Page 0, 15
4	01000000	WDTCLR
5	00010000	BCD0 A
6	00000001	LOAD 16, B
7	00010000	0x10, Page 0, 16
8	01000000	WDTCLR
9	00010001	BCD0 B
10	00010000	BCD0 A
11	00010001	BCD0 B
12	00010000	BCD0 A
13	00010001	BCD0 B
14	00010000	BCD0 A
15	00010110	BCD0 B
16	11111111	BCD0 A

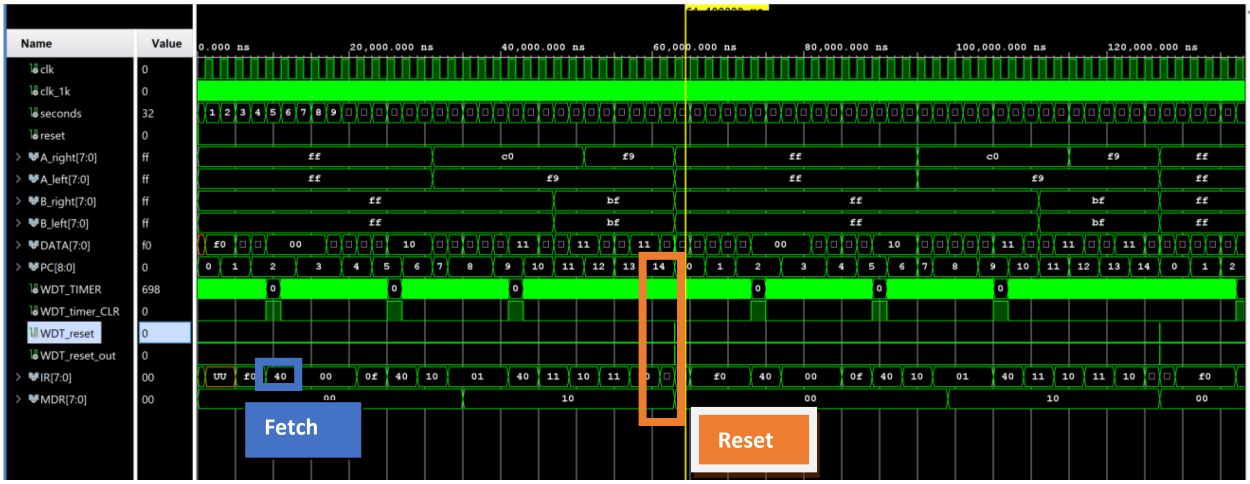


Figure 6: CLRWDT Simulation

This functionality was made possible by a free running clock, also running at 1 KHz. If *WDT_CLR* flag is not asserted, the WDT timer accumulates and is set to expire at 10000 cycles (10 seconds), trigger a reset, then will accumulate one more cycle and then reset itself. The simulation shows the *WDT_reset* being triggered at PC = 14. and all values returning to zero and default segment output.

```
--WDT_timer_CLR flag depends on IR, driven by other process. should be good.
--WDT_TIMER depends on CLK_1k and WDT_timer_CLR, and reset
process(clk_1K, reset, WDT_timer_CLR)
begin
    if(reset = '1' or WDT_timer_CLR = '1') then   --either reset will reset timer, so it will reset itself
        WDT_TIMER <= 0;

    elsif(clk_1K'event and clk_1K = '1') then
            if(WDT_TIMER = 10001) then            --we assume the controller is reset at time limit, so at the next cycle
              WDT_TIMER <= 0;                     --reset timer
            elsif(WDT_TIMER < 10001) then         --less than time limit + 1 and timer_CLR is not one, accumulate
                WDT_TIMER <= WDT_TIMER + 1;
            end if;
    end if;

end process;
```

The *WDT_reset* variable is triggered when this timer reaches 10 seconds.

```
--WTD_reset depends on WDT_TIMER
process(WDT_TIMER, reset)
begin

    if(WDT_TIMER = 10000) then                    --if WDT is not cleared and the timer has reached time limit
        WDT_reset <= '1';
    else
        WDT_reset<= '0';
    end if;
end process;
```

The reset is identical to the *reset* input by including the *WDT_reset* variable in the conditional

```
if(reset = '1' or WDT_reset = '1') then
   CurrState <= Fetch;
   PC <= (others => '0');
   IR <= (others => '0');
   MDR <= (others => '0');
   A <= X"01";
   B <= (others => '0');
   N <= '0';
   Z <= '0';
   V <= '0';
   Outport0 <= (others => '0');
   Outport1 <= (others => '0');
   A_left <= X"FF";
   A_right <= X"FF";
   B_left <= X"FF";
   B_right <= X"FF";
   --WDT_reset <= '0';
   temp := 0;
elsif(rising_edge(clk)) then
```

RTL

Fetch	Execute
IR = MEM[PC]	PC = PC + 1 WDT_CLR = 1 (WDT_TIMER = 0)

OR

Fetch	Execute
IR = MEM[PC]	WDT_CLR = 0 PC = 0 CCR = 0