Digital Design and Computer Architecture

Final Report- SOS '21

By Siddhi Bagul, 190070015

Mentor- Janeel Patel

1 Introduction

Digital circuits have become the core of any computational electric device. Since the computer can only understand the language of 'high' and 'low', that constraints the data(any kind of signal) to be in discrete levels. Thus, digital systems work on discrete space. Special circuits like Analog-to-Digital-Converters(ADC) and Digital-to-Analog-Converters(DAC) connects these two words of continuous and discrete space. Since, it would be cumbersome to understand both digital and analog aspects together, we work in abstractions. This abstraction has really helped in going from basic logic gates to any complex processor.

2 Digital systems

2.1 Number systems

Digital systems use binary number systems as it reduces logic levels to just 'high' and 'low'. Any number can be represented in binary number system. The most common IEEE standard method of representation is Two's complement. Two's complement encompasses both signed and unsigned numbers.

2.2 Logic gates

Logic gates are basic digital circuits which take one or more binary inputs and produce a binary output. These logic gate are implemented using transistors(MOS/ Bipolar Junction). Nowadays, CMOS transistors are widely used than BJTs as they draw zero input current. Though,TTL(Transistor-Transistor logic) still has applications in some areas.
The parameters such as $V_{IL}, V_{OL}, V_{IH}, V_{OH}$ are decided by looking at DC transfer characteristics of a transistor. For an unambiguous design, $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$
pMOS transistors are good at passing '1' and nMOS transistors are good at passing '0'. Thus, pMOS transistors are used for pull ups and nMOS transistors for pull down.

2.3 Combinational logic design

Combinational circuits: Outputs depend only on current inputs.
A circuit is combinational if:

1. Every circuit element is itself combinational.

2. Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.

3. The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

   Remark: While designing combinational circuits using VHDL, I encountered an error called "comninational loop". This occurs when there is a cyclic path in a circuit without having any sequential element in it.(for example, sum=sum+1 will give this error)

2.3.1 Logic minimisation

It is clear that output of combinational circuit can be represented by combination of inputs. Thus it gives us a logic equation. Minimising this logic equation affects directly to its gate level implementation. There are several methods to do this:

1. **Boolean Algebra**: Boolean Algebra method uses various axioms and theorems to minimise logic equation.

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2})$	De Morgan's Theorem

2. **Karnaugh Maps(K-maps)**:It is a graphical method of minimisation. K-maps work well for problems up to four variables. For more than 4 variables case, we need to use method of stacking which is bit cumbersome.

3. **Quine-McCluskey** This involves two steps. 1)Finding Prime implicants. 2) Use those prime implicants to find essential prime implicants, as well as other prime implicants that are necessary to cover the function. This method involves integer linear programming which is readily available in software packages like MATLAB.

2.3.2 Combinational building blocks, Timing and Glitches

The two major combinational building blocks are multiplexers and decoders. Multiplexers choose an output from several possible inputs based on the value of select signal. Decoder takes N inputs and gives away 2^N outputs. It asserts exactly one of its outputs depending upon input combination. We can implement any truth table with the help of MUX and decoders.

Any circuit takes time to change its voltage level from high to low (rising time) and low to high(falling time). This results in delay between input change and output change.Combinational logic has propagation and contamination delay. Propagation delay is maximum time between input change and final output change. Contamination delay is the minimum time between an input change and starting of output change. The speed of combinational circuit is limited by critical aka slowest path.The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path.

When single input transition causes multiple output transitions, it is called as a glitch. Glitches occur when there is the difference in propagation delay of two branches. Glitches won't cause any problem as long as we are waiting for enough propagation delay. We can avoid glitches by adding extra redundant gates. K-maps are useful to find 'Glitch' conditions. Glitches occur whenever there is transition across boundaries of two prime implicants.

2.4 Sequential Logic Design

Sequential Circuits: Output depends on both current and previous input values. Insertion of 'memory' device. It can remember prior inputs in the form of smaller information called the *state.state* is represented by a set of bits called the *state variables.*

2.4.1 Latches and Flip flops

These are the fundamental building blocks of the memory. These are bistable circuits with two stable states. Flip flops have clock input whereas latches do not. Difference between latches and flipflops can be understood by following example: In a D-Latch, state is updated continuously while clock is '1'. Whereas in D-Flipflop, state is updated at specific instant of time. Register is the bank of Flip-flops.

2.4.2 Asynchronous/ Synchronous Circuits

Asynchronous circuits has output directly fed back to inputs. This may run into *race conditions* where behaviour of circuit depends on which of two paths through logic gates is faster.Thus slightly non identical implementation of same circuit can cause different behavior. This problem is solved by inserting registers somewhere in the path. As register value changes only at the clock edge, state of the system is synchronised with the clock. The circuit is synchronous sequential circuit if it consists:

1. Every circuit element is either a register or a combinational circuit
2. At least one circuit element is a register.
3. All registers receive the same clock signal
4. Every cyclic path contains at least one register

Finite State Machines Synchronous sequential circuits with finite states can be drawn in the forms shown below. An FSM has a clk input and it advances to the next state on each clock edge. There are two types of FSM designs: 1) In Moore machines, outputs depend only upon the current state of machine. 2)In Melay machines, output depends on both current state and current input.

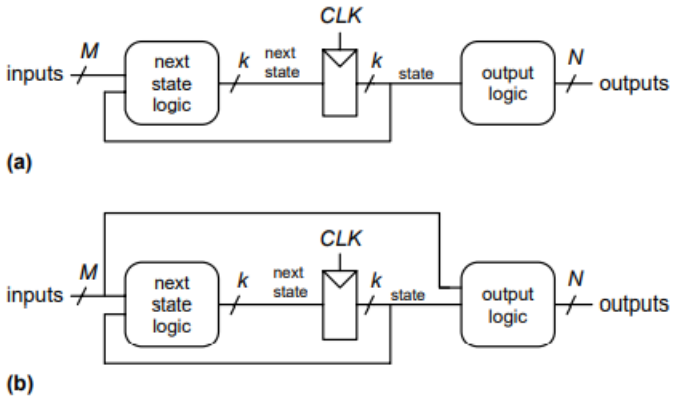


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

There are various steps while structurally implementing an FSM, such as State encoding, output encoding, state transition table etc. The more convenient and easy-to-go way is the behavioral implementation of an FSM. FSM has many applications like controller design, counters etc.

3 Architecture

Architecture of a computer is determined by its instruction set. Same architecture may have different hardware implementation. Instructions are assembled at multiple levels. For example, if we are writing in 'C' language, compiler firsts assembles the code in 'assembly language'. Then Assembler creates an object file which has machine code for each instruction. Linker links any library associated and creates a single executable program. Loader then loads it to memory.

There are various Instruction Set Architectures including ISA-32, RISC, X86 etc. The very general basic MIPS processor is based on IA-32 architecture. For our case, as we are trying to build our own microprocessor IITBProc, lets see this points with respect to IITBProc's ISA.

IITB-Proc is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The IITB-Proc is an 8-register, 16-bit computer system. It has 8 general-purpose registers (R0 to R7). PC points to the next instruction. All addresses are short word addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). This architecture uses condition code register which has two flags Carry flag (c) and Zero flag (z). The IITB-Proc is very simple, but it is general enough to solve complex problems. The architecture allows predicated instruction execution and multiple load and store execution. There are three machine-code instruction formats (R, I, and J type) and a total of 12 instructions. Here, R means register type, I means Immediate type and

J means branch instructions. The instructions are coded in 16 bit format as follows:

R Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Register B (RB) (3-bit)	Register C (RC) (3-bit)	Unused (1 bit)	Condition (CZ) (2 bit)

I Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Register C (RC) (3-bit)	Immediate (6 bits signed)

J Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Immediate (9 bits signed)

Instructions Encoding:

ADD:	00_00	RA	RB	RC	0	00
ADC:	00_00	RA	RB	RC	0	10
ADZ:	00_00	RA	RB	RC	0	01
ADI:	00_01	RA	RB	6 bit Immediate		
NDU:	00_10	RA	RB	RC	0	00
NDC:	00_10	RA	RB	RC	0	10
NDZ:	00_10	RA	RB	RC	0	01
LHI:	00_11	RA	9 bit Immediate			
LW:	01_00	RA	RB	6 bit Immediate		
SW:	01_01	RA	RB	6 bit Immediate		
~~LA:~~	~~01_10~~	~~RA~~				
~~SA:~~	~~01_11~~	~~RA~~				
BEQ:	11_00	RA	RB	6 bit Immediate		
JAL:	10_00	RA	9 bit Immediate offset			
JLR:	10_01	RA	RB	000_000		

RA: Register A

RB: Register B

RC: Register C

Instruction Description

Mnemonic	Name & Format	Assembly	Action
ADD	ADD (R)	add rc, ra, rb	Add content of regB to regA and store result in regC. *It modifies C and Z flags*
ADC	Add if carry set (R)	adc rc, ra, rb	Add content of regB to regA and store result in regC, if carry flag is set. *It modifies C & Z flags*
ADZ	Add if zero set (R)	adz rc, ra, rb	Add content of regB to regA and store result in regC, if zero flag is set. *It modifies C & Z flags*
ADI	Add immediate (I)	adi rb, ra, imm6	Add content of regA with Imm (sign extended) and store result in regB. *It modifies C and Z flags*
NDU	Nand (R)	ndu rc, ra, rb	NAND the content of regB to regA and store result in regC. *It modifies Z flag*
NDC	Nand if carry set (R)	ndc rc, ra, rb	NAND the content of regB to regA and store result in regC if carry flag is set. *It modifies Z flag*
NDZ	Nand if zero set (R)	ndc rc, ra, rb	NAND the content of regB to regA and store result in regC if zero flag is set. *It modifies Z flag*
LHI	Load higher immediate (J)	lhi ra, Imm	Place 9 bits immediate into most significant 9 bits of register A (RA) and lower 7 bits are assigned to zero.
LW	Load (I)	lw ra, rb, Imm	Load value from memory into reg A. Memory address is computed by adding immediate 6 bits with content of reg B. *It modifies flag Z.*

SW	Store (I)	sw ra, rb, Imm	Store value from reg A into memory. Memory address is formed by adding immediate 6 bits with content of red B.
LA	Load All (J)	lm ra	Load all registers (in a sequence of register, R0 to R7) Memory address is given in reg A. Registers are loaded from consecutive addresses.
SA	Store All (J)	sm, ra	Store all registers (in a sequence of register, R0 to R7). Memory address is given in reg A. Registers are stored to consecutive addresses.
BEQ	Branch on Equality (I)	beq ra, rb, Imm	If content of reg A and regB are the same, branch to PC+Imm, where PC is the address of beq instruction
JAL	Jump and Link (I)	jalr ra, Imm	Branch to the address PC+ Imm. Store PC into regA, where PC is the address of the jalr instruction
JLR	Jump and Link to Register (I)	jalr ra, rb	Branch to the address in regB. Store PC into regA, where PC is the address of the jalr instruction

(We have eliminated LA and SA instruction as those might be better to be implemented as pseudo instructions) As we can see through instruction format, IITBProc accesses memory through only Load/store instructions. Program counter is 16 bit register which points to instruction location in instr-memory. It also supports conditional branching as well as linking. Thus it comes handy to have subroutines in program code. As of now, we have seen only major properties of IITBProc. We will see hardware implementation of this instructions in micro-architecture part.

4 Microarchitecture

Microarchitecture of a processor describes actual hardware implementation of instructions set. Depending upon the of clk cycles each instruction takes, there are two types of such implementation:

1. Single Cycle Processor: Instruction executes in single cycle
2. Multi Cycle Processor: Instruction take multiple cycles to execute

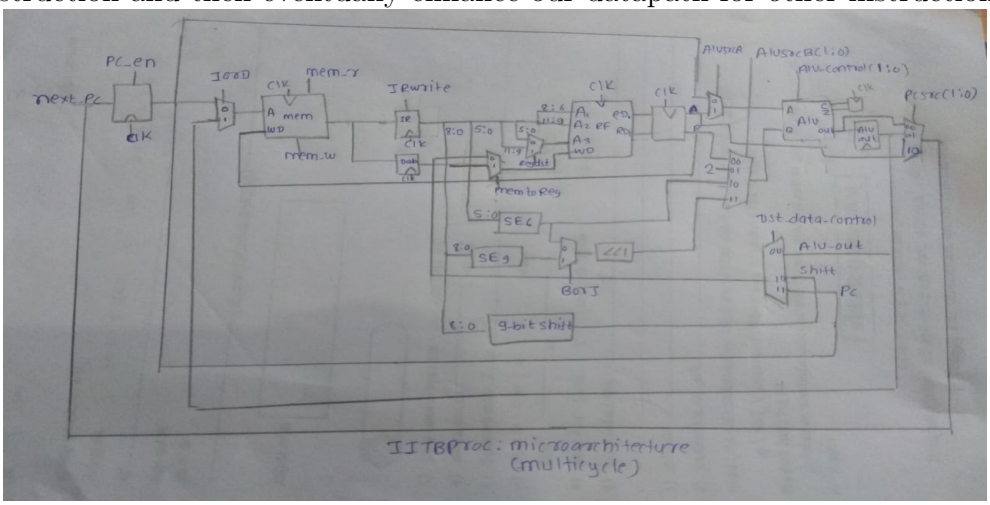
Any of above implementation requires same design steps:

1. Datapath designing: Datapath operates on word of data. It contains memory module, registers, register file, ALU and any other require hardware depending upon ISA.

2. Control unit: Control unit is controller FSM which controls the execution of instruction.

For our case, we are designing multicycle processor. But what does multicycle really mean while implementing it on hardware? The answer is we break our instruction in to stages. At each cycle, this stages are implemented. Thus depending upon instruction complexity, instruction takes number of cycles for its execution.

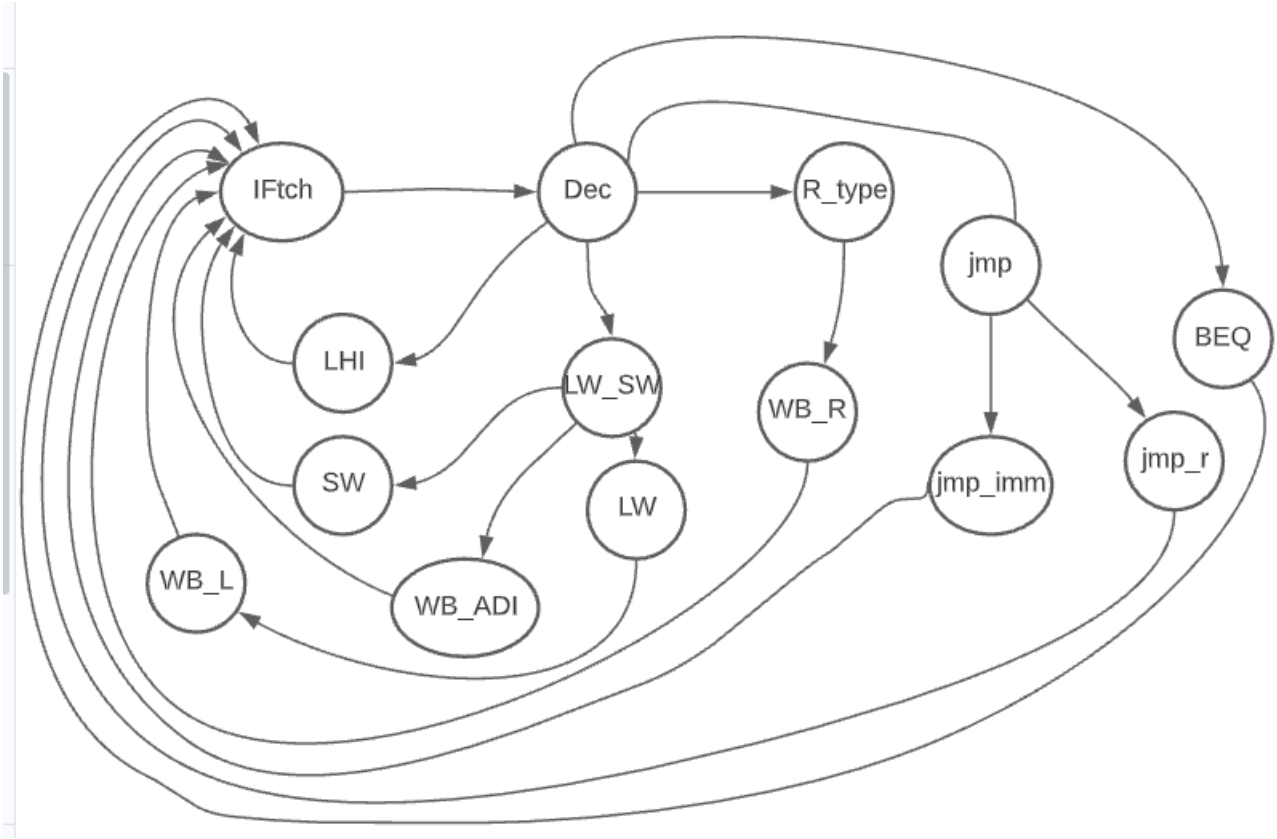
Datapath design for IITBProc: Highly inspired from MIPS multicycle datapath, we designed our own customised datapath. Generally, We first start with Load/store instruction and then eventually enhance our datapath for other instructions.



Controller FSM: We determined our FSM states according to our ISA. We have 14 states, which plays around control signals to give our datapath direction!

FSM states: IFtch, Dec, R_type,Jmp, jmp_imm, jmp_r, BEQ, WB_R, LHI, LW_SW, SW, LW, WB_L, WB_ADI

State diagram for controller FSM:



The code snippet for controller FSM is given here:

```
-- Controller FSM
  process(clk,ctrl_state)
  variable nxt_state:myfsm:=ctrl_state;

  begin

   case ctrl_state is
    when Iftch => V_AlusrcA:='0';
                  V_AlusrcB:="01";
                  V_ALU_control:="00";
                  v_dst_data_control:="00";
                  v_Pc_en:='1';
                  v_IorD:='0';
                  v_mem_w:='0';
                  v_mem_r:='1';
                  v_IRwrite:='1';
                  v_BorJ:='0';
                  v_rf_w:='0';
                  v_memtoReg:='0';
                  v_RegDst:='0';
                  v_PcSrc:="00";
                  nxt_state:= Dec;
```

```
when Dec => -- calculates PC+imm6bit*2

          v_AlusrcA:='0';
            v_AlusrcB:="11";
            v_ALU_control:="00";
            v_dst_data_control:="00";
            v_Pc_en:='0';
            v_IorD:='0';
            v_mem_w:='0';
            v_mem_r:='0';
            v_IRwrite:='0';
            v_BorJ:='0';
            v_rf_w:='0';
            v_memtoReg:='0';
            v_RegDst:='0';
            v_PcSrc:="00";
          case opcode is
           when "0010" | "0000"=> nxt_state:=R_type;
           when "0100" | "0101" | "0001" => nxt_state:=LW_SW;
           when "0011" => nxt_state:=LW_SW;
           when "1000" | "1001" => nxt_state:=Jmp;
           when "1100" => nxt_state := BEQ;
           when others => nxt_state:= Iftch;
          end case;

          -- FUll code can be found in github repo
```

5 Generic Building Blocks

At first I tried to implement IITBProc structurally. But due to time constraints and ease of doing shifted to behavioral modelling. Below are some building blocks that I created for structural implementation.

1. **Register**: Register is the collection of D-FF which stores any bit data and updates on each clock given En-signal is asserted

```
library ieee;
use ieee.std_logic_1164.all;

entity Reg_s is
port(clk,En:in std_logic;
     D: in std_logic_vector(15 downto 0);
     Q: out std_logic_vector(15 downto 0));
end entity Reg_s;
```

```
architecture behav of Reg_s is
signal Q_s:std_logic_vector(15 downto 0);
begin
process(clk)
begin
if(rising_edge(clk)) then
Q_s <=D;
end if;
end process;
Q<= Q_s;
end architecture behav;
```

2. **Regfile**: Regfile is collection of registers. It has a demux, registers and a mux.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity Regfile is
port (clk,rf_w:in std_logic;
A1,A2,A3: in std_logic_vector(2 downto 0);
D3:in std_logic_vector(15 downto 0);
D1,D2: out std_logic_vector(15 downto 0));
end entity Regfile;

architecture behav of Regfile is
type ramtype is array (7 downto 0) of std_logic_vector(15 downto 0);
signal mem: ramtype;
begin
-- A1,A2,A3 are reg field addresses
-- D1,D2 are read from registers
--D3 is 16bit write data
process(clk)
begin
if (rising_edge(clk)) then
 if(rf_w ='1') then mem(CONV_Integer(A3)) <= D3;
 end if;
 end if;
 end process;
 process(A1,A2)
 begin
 D1<= mem(Conv_Integer(A1));
 D2<= mem(Conv_Integer(A2));
 end process;
end architecture behav;
```

3. **shifter**: Shifter shifts the given bit data to a 16-bit vector according to specified shift amount.

```
library ieee;
use ieee.std_logic_1164.all;

entity Shifter is
generic (N:integer:= 9; sht_amt:integer:=7);
port(X: in std_logic_vector(N-1 downto 0);
Y:out std_logic_vector(15 downto 0));
end entity;

architecture behav of Shifter is
signal shifted_signal: std_logic_vector(15 downto 0);
begin
process(x)
begin
for i in 15 downto 0 loop
if( (i-sht_amt >= 0)) then
shifted_signal(i) <= X(i-sht_amt);
else
shifted_signal(i)<='0';
end if;
end loop;
end process;
Y<= shifted_signal;
end architecture behav;
```

4. **signextend**:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SignExt6 is
Port (X:in std_logic_vector(5 downto 0);
      Y:out std_logic_vector(15 downto 0) );
end SignExt6;

architecture Behavioral of SignExt6 is

begin
Y <= "0000000000" & X when X(5)= '0' else "1111111111" & X;

end Behavioral;
```

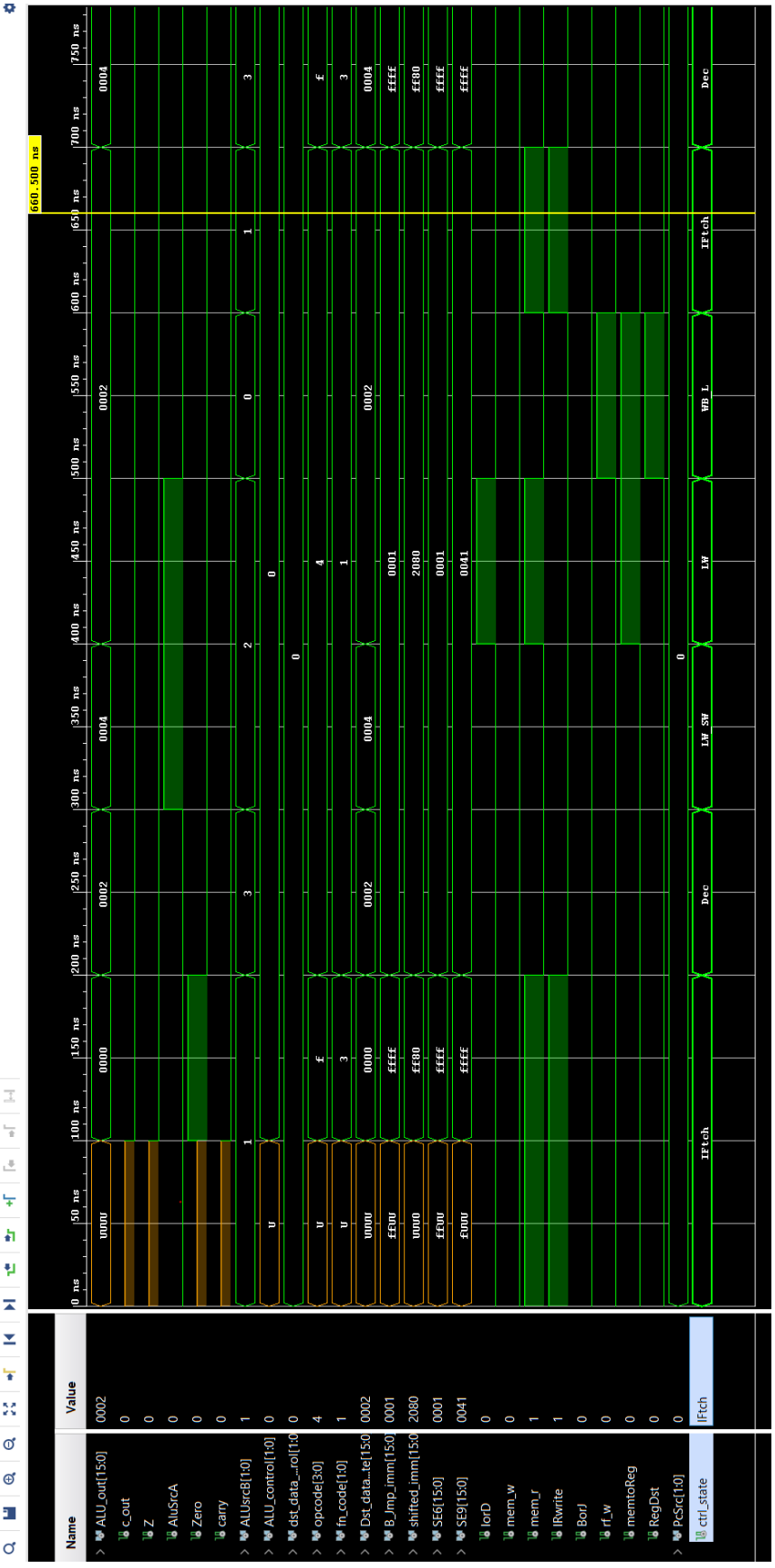
5. IorDmem:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.all;



entity IorD_mem is
Port (clk,mem_w:in std_logic;
mem_add: in std_logic_vector(4 downto 0);
mem_wdta: in std_logic_vector(15 downto 0);
mem_rdata:out std_logic_vector(15 downto 0));
end IorD_mem;

architecture Behavioral of IorD_mem is
type memory is array (31 downto 0) of std_logic_vector(15 downto 0);
signal i_mem: memory;
begin
process(clk)
begin
if(rising_edge(clk))then
if(mem_w='1') then i_mem(CONV_INTEGER(mem_add)) <=mem_wdta;
else
mem_rdata <=i_mem(CONV_INTEGER(mem_add));
end if;
end if;
end process;


end Behavioral;
```

The whole project can be found at `https://github.com/Sid6761/IITBProc_Behav`. Simulation result on xilinx vivado:

The image here shows implementation of LW instruction.


Name
Value
ALU_out[15:0]
0002
c_out
0
Z
0
AluSrcA
0
Zero
0
carry
0
ALUsrcB[1:0]
1
ALU_control[1:0]
0
dst_data...rol[1:0]
0
opcode[3:0]
4
fn_code[1:0]
1
Dst_data...te[15:0]
0002
B_jmp_imm[15:0]
0001
shifted_imm[15:0]
2080
SE6[15:0]
0001
SE9[15:0]
0041
IorD
0
mem_w
0
mem_r
1
IRwrite
1
BorJ
0
rf_w
0
memtoReg
0
RegDst
0
PCSrc[1:0]
0
ctrl_state
IFtch
660.500 ns
IFtch
Dec
LW SW
LW
WB L
IFtch
Dec