Report On The

CIOCCOLATO Machine

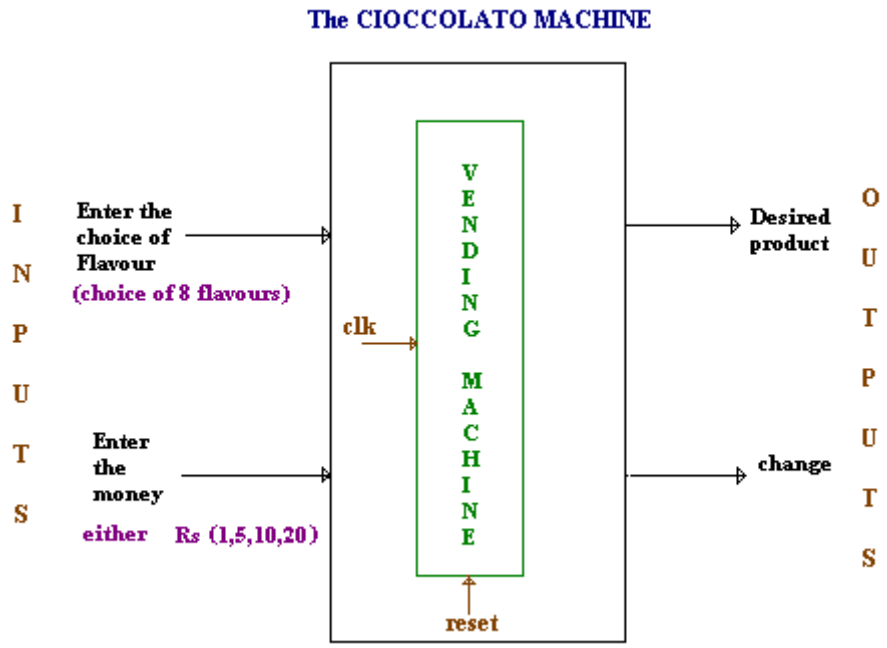
Design

by

1. Yogesh Kubal
2. Jinisha Vengurlekar
3. Shruti Patkar
4. Mansi Bhatt
5. Aditi Desai
6. Shreyas Gaonkar
7. Sunit Mehta

1. Understanding the problem:

- The Cioccolato Machine or the instant chocolate machine is required to be "re-designed".
- It is basically a vending machine which instantly prepares the desired chocolate flavor.



Input Buttons:

- At the input of the vending machine we have the select button which gives us a choice of 8 different chocolate flavors. And, the money denominations are accepted are Rs1, 5, 10, and 20.

- The following table shows the product details.

Sr.No.	Chocolate flavors	Price
1	Gianduja	Rs.1
2	Cou-ver-ture	Rs.3
3	Dark Chocolate	Rs.4
4	Sweet Chocolate	Rs.5
5	Chocolate Caramel	Rs.6
6	Milk Chocolate	Rs.8
7	White Chocolate	Rs.10
8	ChocolatePraline	Rs.20

Table 1: List of Products with prices

Output Buttons:

1) **Product**
   Vendee gets the desired product when
   (Money inserted) > **or** = (Price of the product)

2) (**Change**) = (Money inserted) – (Price of the product)
   (Money inserted) > (Price of the product)

The Cioccolato Machine unlike most other vending machines, prepares chocolate instantly. And for this part of the machine, we need to design a processor with the following specifications:-

- The *recipe for each type of chocolate* is given and using this *we obtain set of instruction sequences* for each reciepe. This gives us the *list of opcodes* that will be fed into the *instruction memory(IM).*

- The machine works on a slow *clock of 1Hz*.

- To prepare the recipes, we have *storages of chocolate* (that is available in *limited* quantity and hence its count is updated after every chocolate flavor is released.)
  We also have *infinite milk storage.*

- These are represented in main memory location as follows:

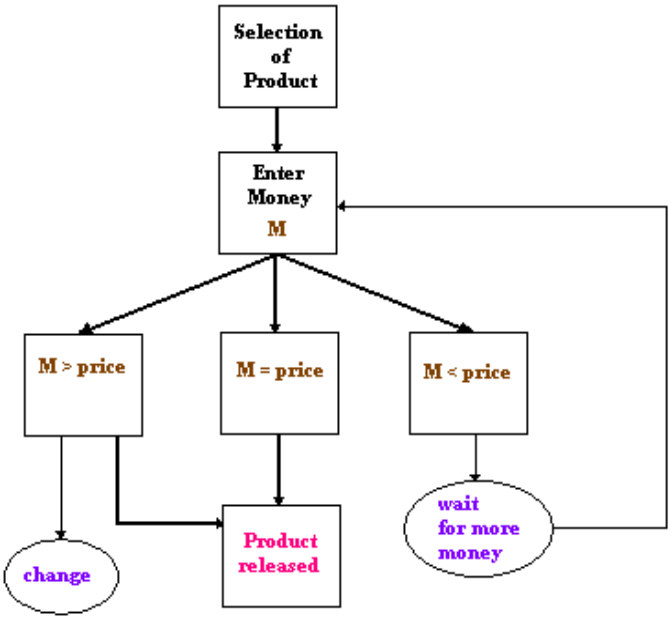
  - Main memory **000**: Updated count of **chocolate** parts available for use.
  - Main memory **001**: Store **milk** needed for recipes.

- We have 4 distinct registers and they are defines as follows:

  - R1: Mixing bucket (accumulator)
  - R2: Current available chocolate parts in machine for recipe
  - R3:One part of milk
  - R4: One part of chocolate.

2. Devising a plan: (Algorithm)

Vending machine:



Processor: *(line of action)*

1) In all, 8 instructions will be used in order to realize a the entire functioning of the machine. A detailed **datapath for each instruction** is first prepared.

2) A **combined datapath** is then prepared. A combined datapath is the one which is capable of execting all the instructions, given one at a time.

3) In order to increase the number of outputs in a given time i.e. the throughput, we use the concept of **pipelining.**

4) Upon obtaining the combined data path, it is **examine**d for various possible **hazards.**

5) Using the details of each recipe, we plan a sequence of instructions for each such that we try on getting minimum data hazards from the sequence.

6) **Each unit** required for the processor (memory, register file, ALU, control unit, hazard unit, buffers) is **designed and implemented in Verilog.**

7) All are **linked together** to get a **final code for the processor.**

❖ Data-Path For Each Instruction:

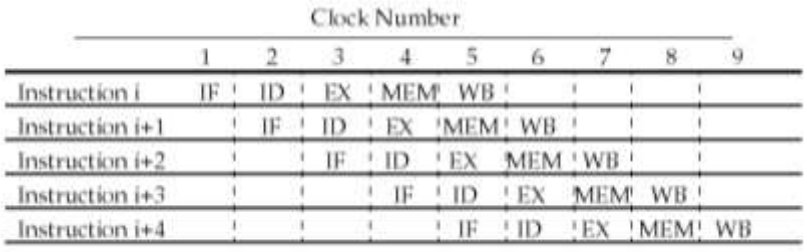
- General Instruction format:

INSTRUCTION	8-BIT INSTRUCTION FORMAT [7:0]								OPERATION
	OPCODE [7:5]			REG [4:3]		MEM [2:0]			
Add	1	0	0			---	---	---	Acc ← Acc + Reg
Sub	1	0	1			---	---	---	Acc ← Acc - Reg
Mul	1	1	0			---	---	---	{Reg,Acc} ← Acc * Reg
Store	1	1	1						[Mem] ← [Reg]
Load	0	1	0						[Reg] ← [Mem]
MemI	0	1	1						[Mem] ← 8'b1111_1111
RegI	0	0	1			I	M	M	[Reg] ← {5'b0,IMM}
EDP	0	0	0	---	---	---	---	---	

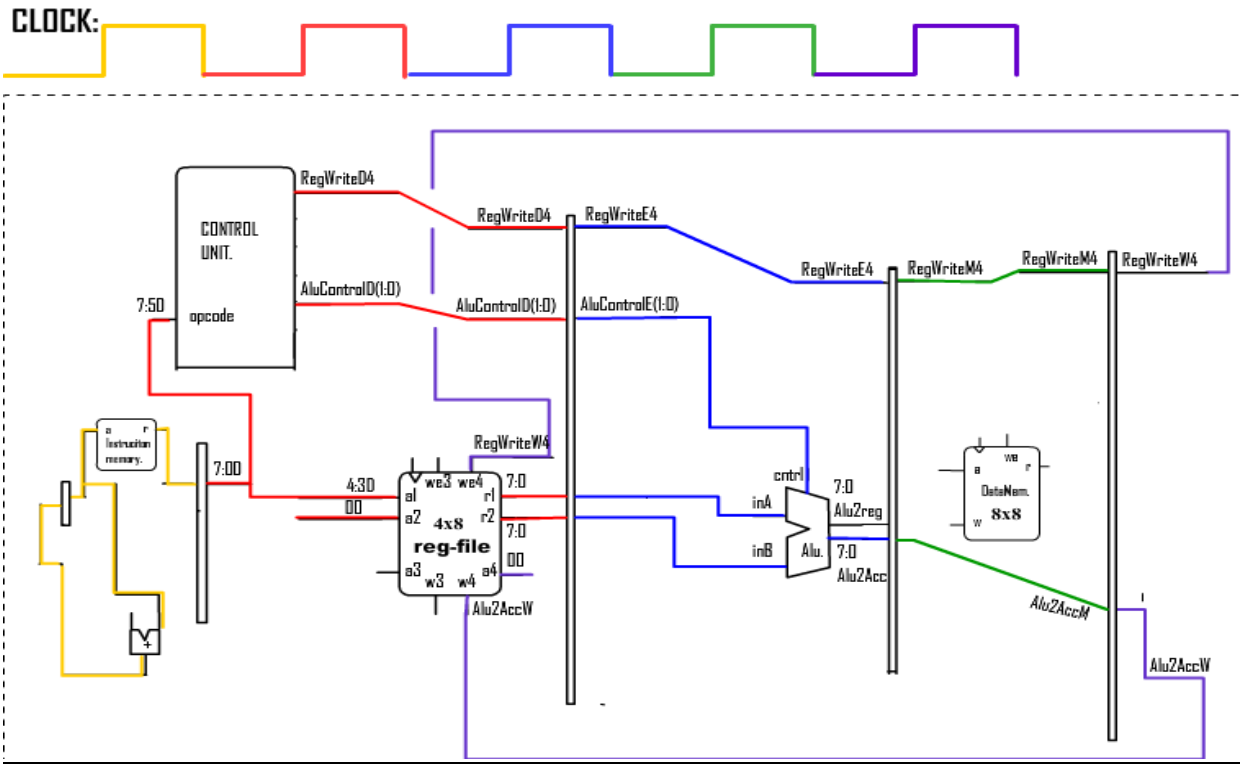
Table 2: General Instruction Format With Assigned Opcodes

❖ Pipelining:

- Pipelining is the ability to overlap execution of different instructions at the same time.It increases the throughput. Each step in a pipeline is called a pipe stage .
- there are five separate stages, we can have a pipeline in which one instruction is in each stage.
  - **a. Instruction Fetch Stage: (IF)**
    reads the next instruction from the memory
  - **b. Instruction Decode/Register Fetch Stage: (ID/RF)**
    reads the register file, determines the control signals for the rest of the pipeline stages, and selects the proper immediate values from the instruction.
  - **c. Execute Stage Stage: (Ex)**
    the ALU performs its calculation and branches are resolved.
  - **d. Memory Stage:**
    (Mem) read or write the data memory
  - **e. Write Back Stage: (WB)**
    write the register file

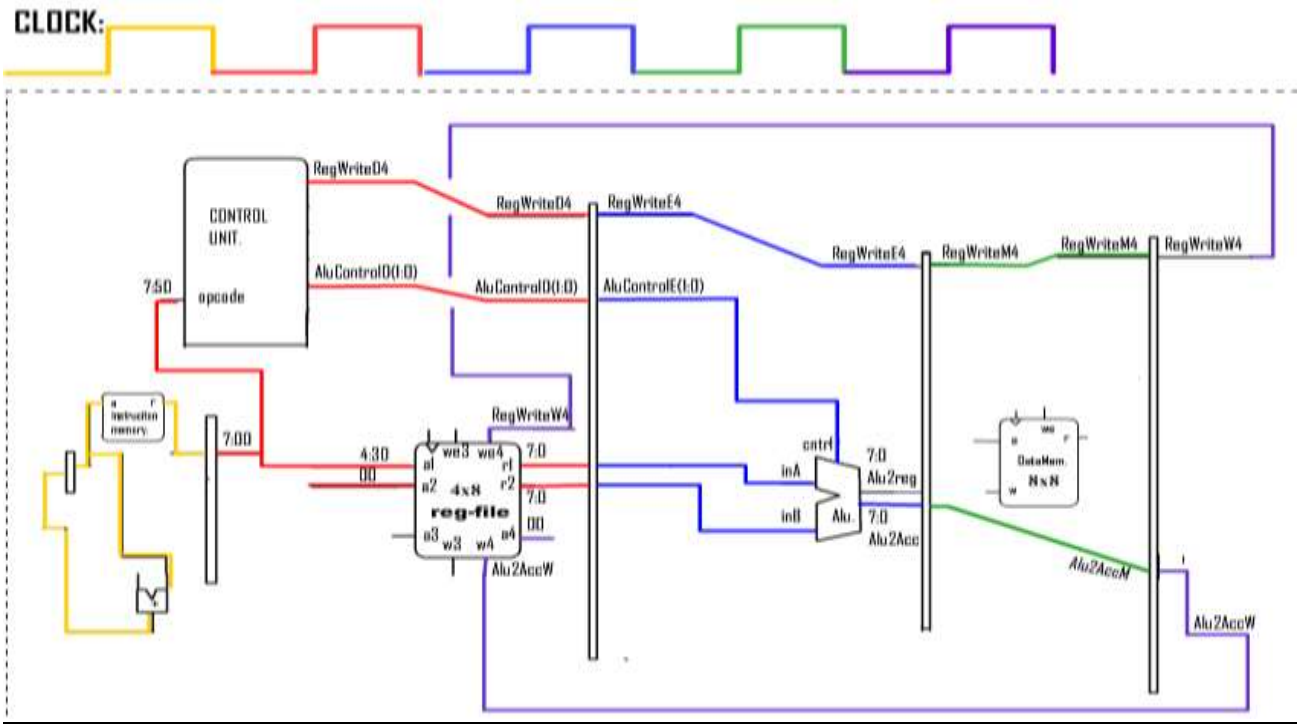


i. Add (Acc ← Acc + Reg)



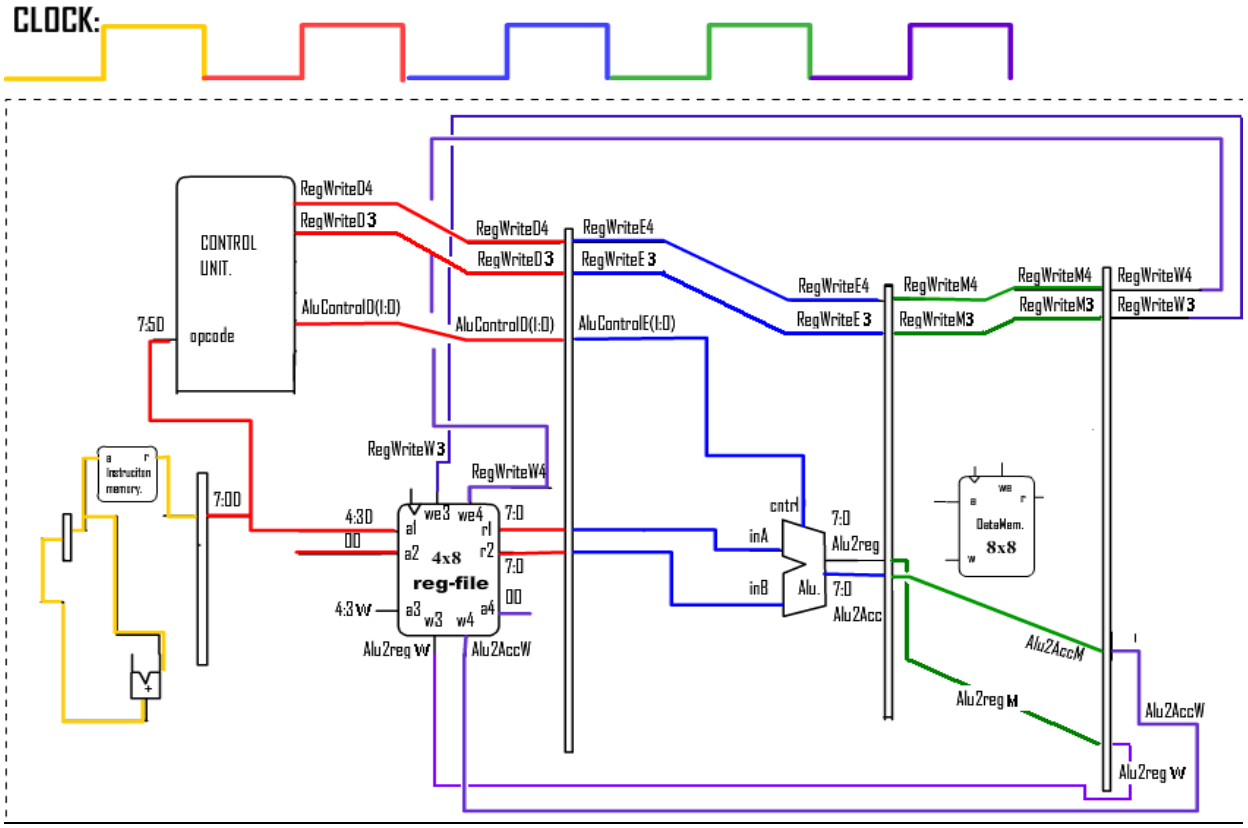
ID/RF stage		Exe stage (ALU)				Mem stage		WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p		Input r/w	Ctrl s/g used	Reg-File write stage	Ctrl s/g used
		in A	in B	Alu2 Acc	Alu2 Reg				
RegWrite4	[4:3] to a1	R1		result	---	---		w4 (a4 hardwired to 00 =>result stored in Acc)	RegWrite4
	00 to a2 (Acc)		R2						

ii. Sub (Acc ← Acc - Reg)



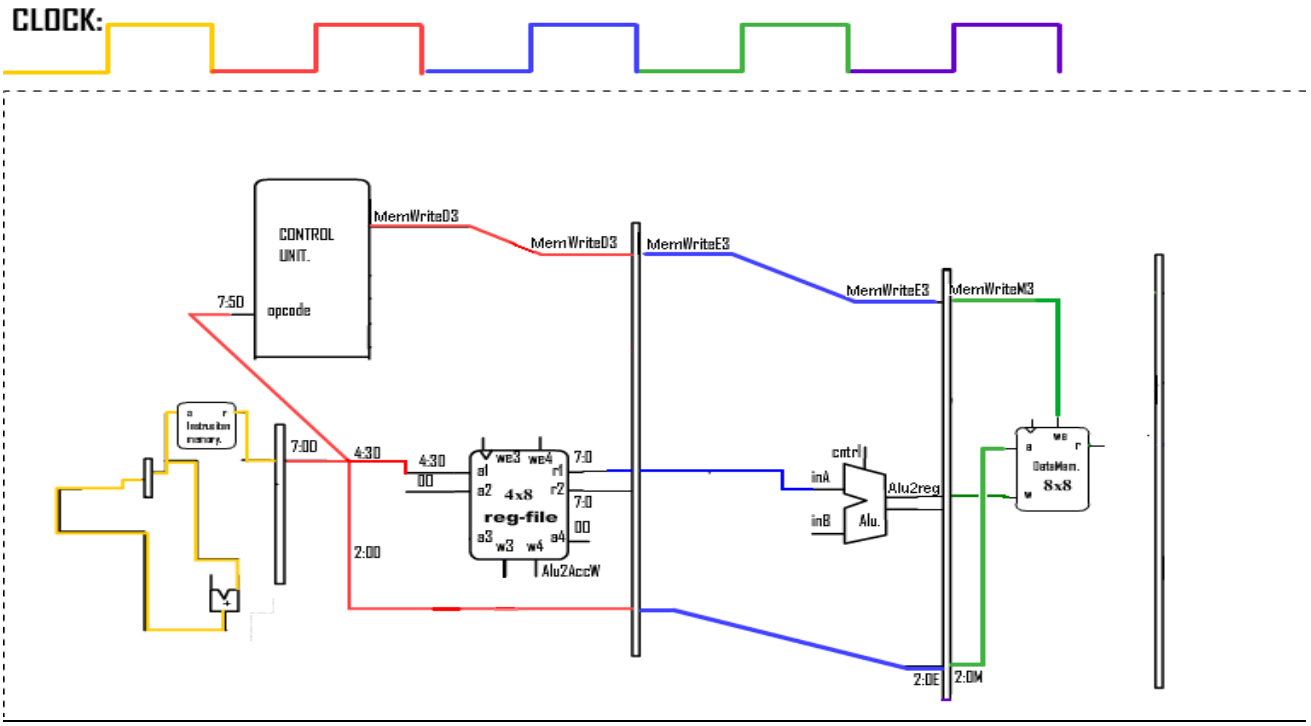
ID/RF stage		Exe stage (ALU)				Mem stage		WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p		Input r/w	Ctrl s/g used	Reg-File write stage	Ctrl s/g used
		in A	in B	Alu2 Acc	Alu2 Reg				
RegWrite4	[4:3] to a1	R1		result	---	---		w4 (a4 hardwired to 00 =>result stored in Acc)	RegWrite4
	01 to a2 (Acc)		R2						

iii. Mul (Acc ← Acc * Reg)



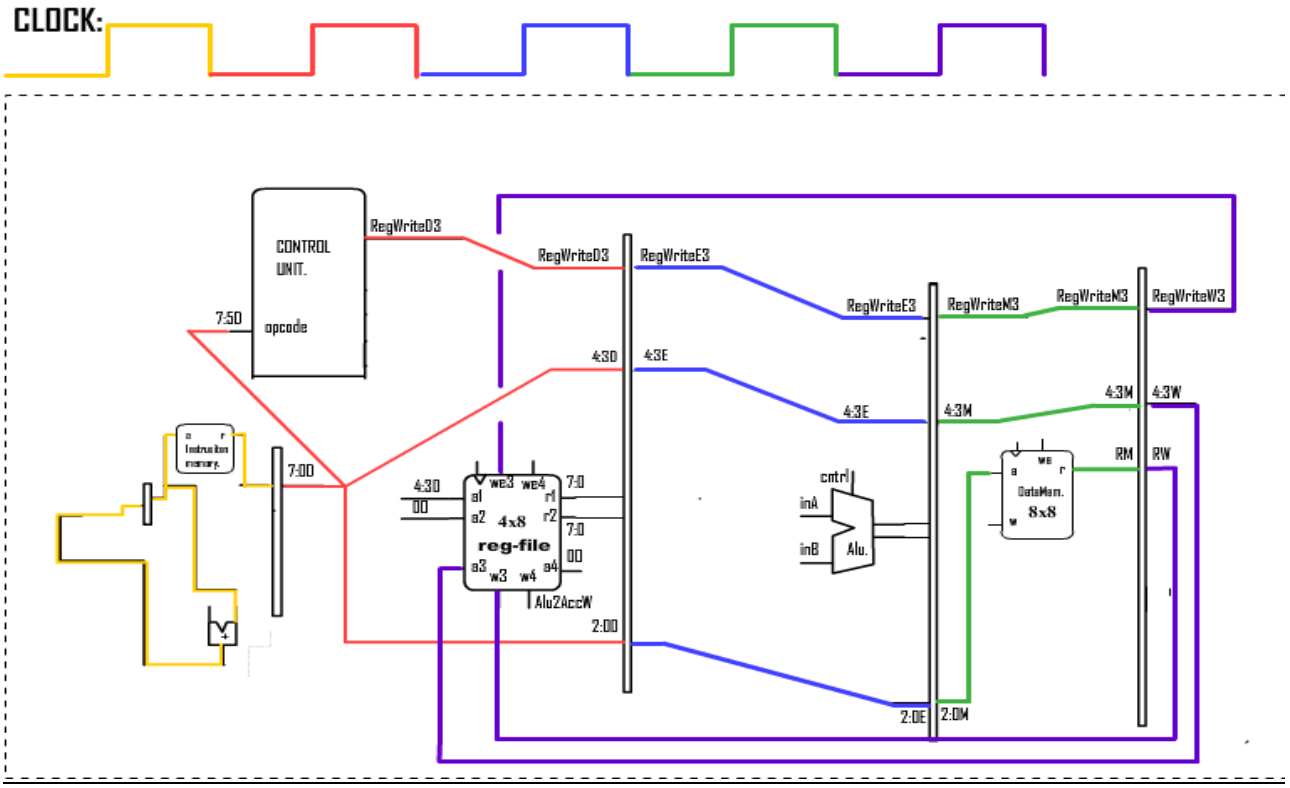
ID/RF stage		Exe stage (ALU)				Mem stage		WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p[15:0]		Input r/w	Ctrl s/g used	Reg-File write stage	Ctrl s/g used
		in A	in B	Alu2 Acc	Alu2 Reg				
RegWrite4	[4:3] to a1	R1		Result [7:0]	Result [15:8]	---	---	W4 (a4 hardwired to 00 =>Acc)	RegWrite4
RegWrite3	00 to a2 (Acc)		R2			---	---	W3 (a3 gets 4:3W)	RegWrite3

iv. Store [Mem] ← [Reg]



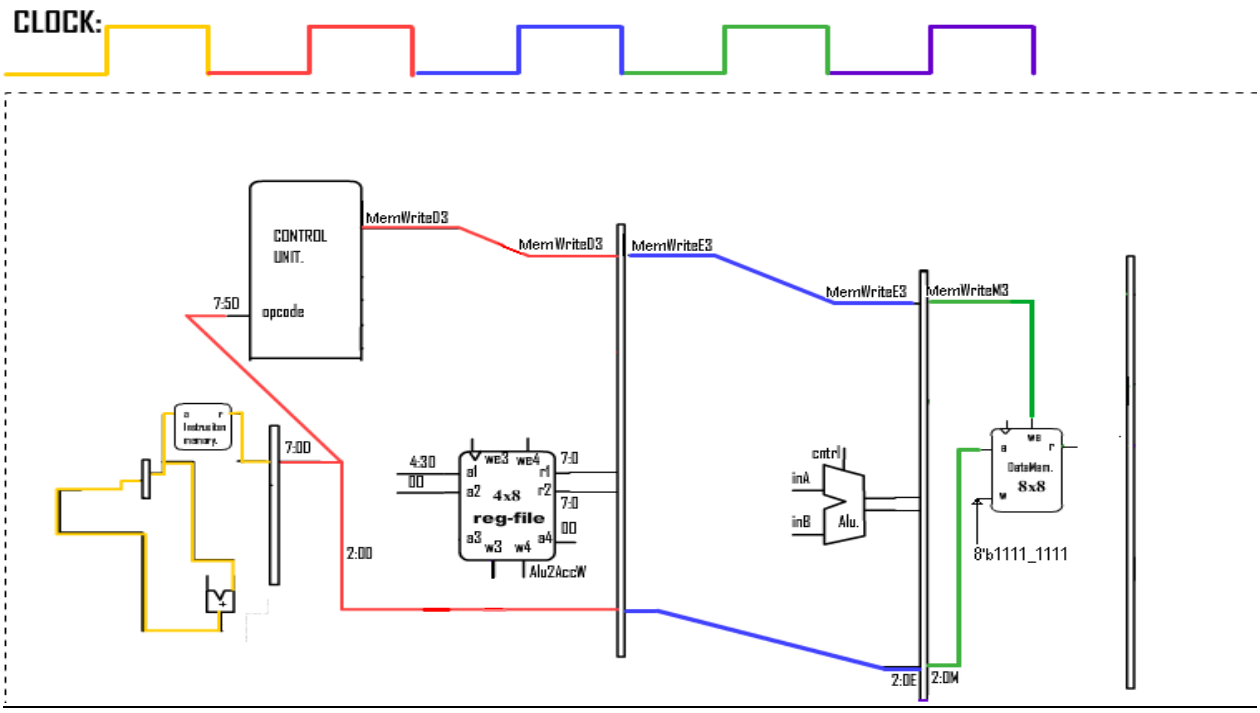
ID/RF stage		Exe stage (ALU)				Mem stage			WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p		Input r/w		Ctrl s/g used	Reg-File write stage	Ctrl s/g
		in A	in B	Alu2 Acc	Alu2Reg	a	w			
MemWrite D	[4:3] to a1	r1	--	---	Contents of r1	2:0M	Alu2 Reg	Mem WriteM	---	---

v. Load [Reg] ← [Mem]



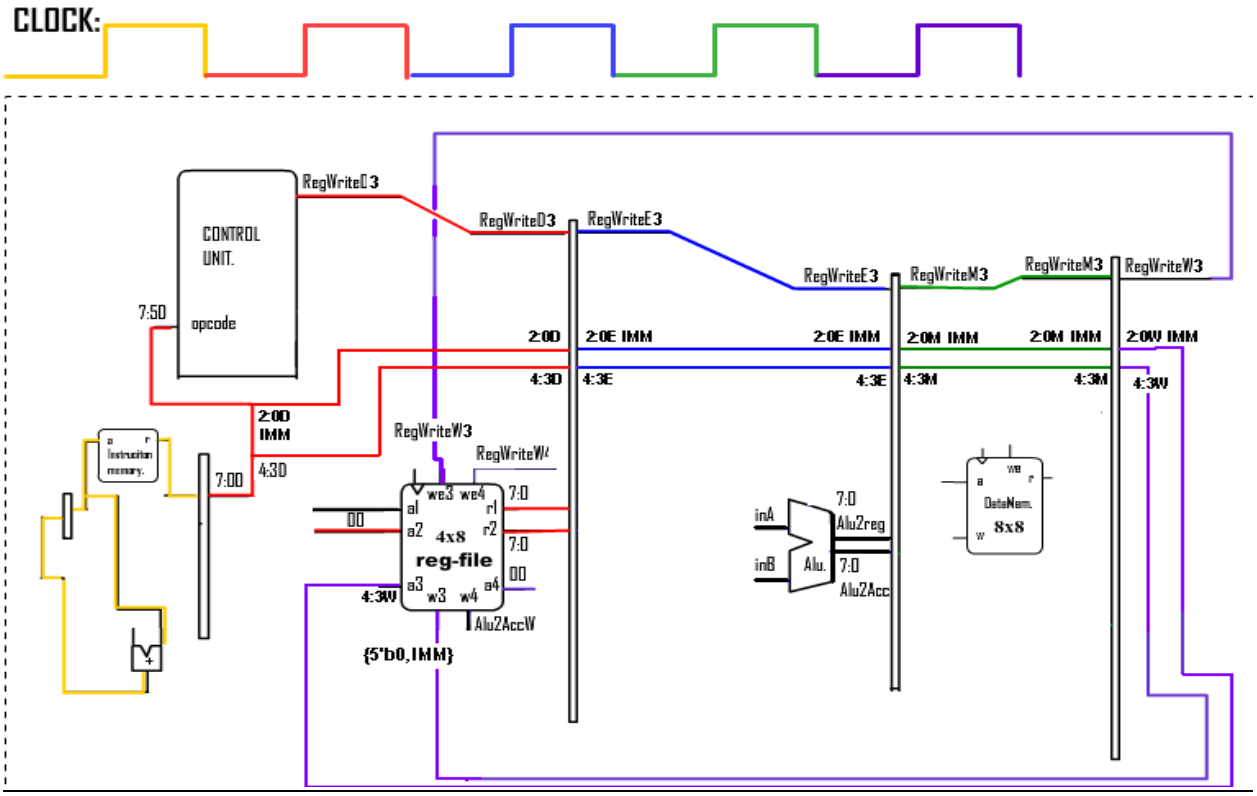
ID/RF stage		Exe stage (ALU)				Mem stage		WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p		Input r/w	Ctrl s/g used	Reg-File write stage	Ctrl s/g used
		in A	in B	Alu2 Acc	Alu2 Reg				
RegWrite3	---	---	---	---	---	2:0M (Data to be stored is obtained from r)	---	r to w3 (a3 4:3W)	RegWrite3

vi. MemI [Mem] ← 8'b1111_1111



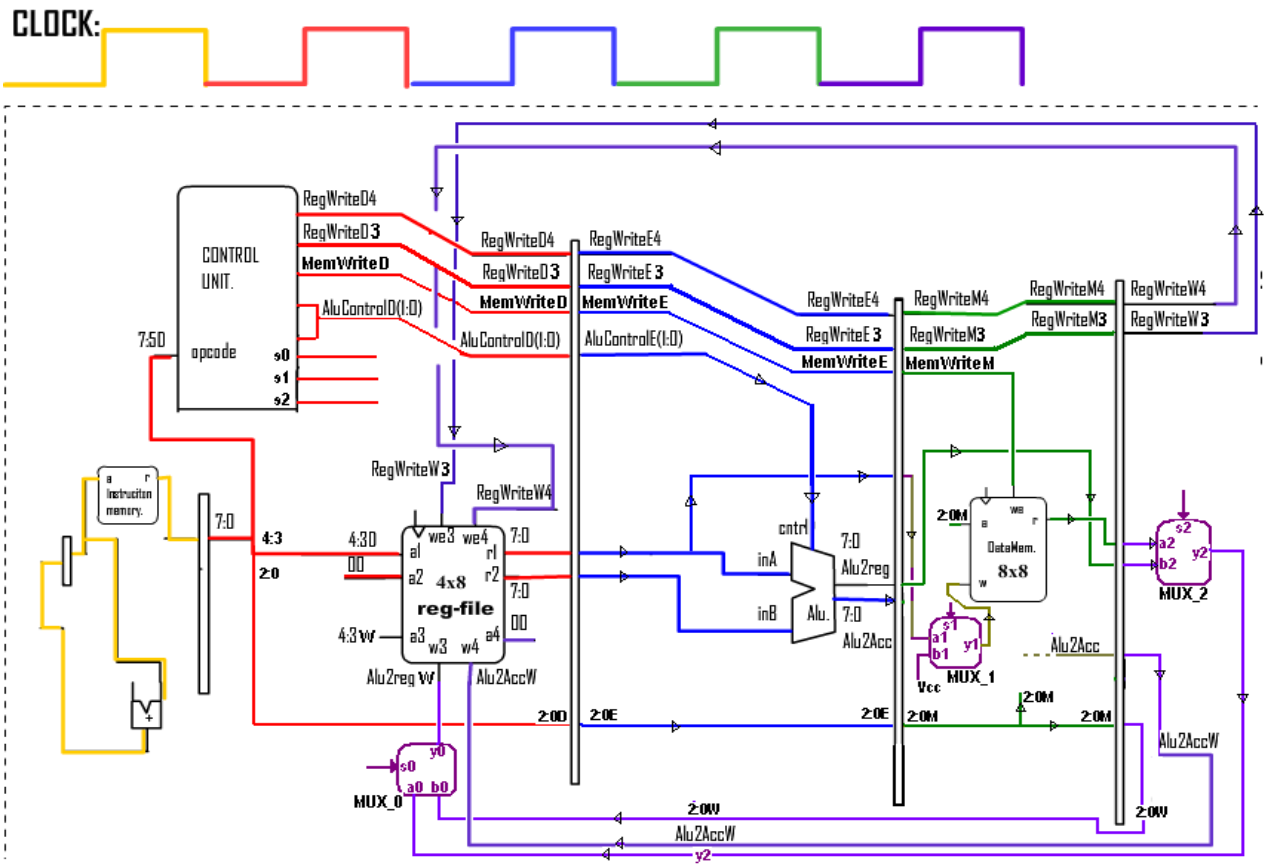
ID/RF stage		Exe stage (ALU)				Mem stage		WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p		Input r/w	Ctrl s/g used	Reg-File write stage	Ctrl s/g used
		in A	in B	Alu2 Acc	Alu2 Reg				
MemWrite	---	---	---	---	---	2:0M At w, 8'b1111_1111 is given as input	MemWrite	---	---

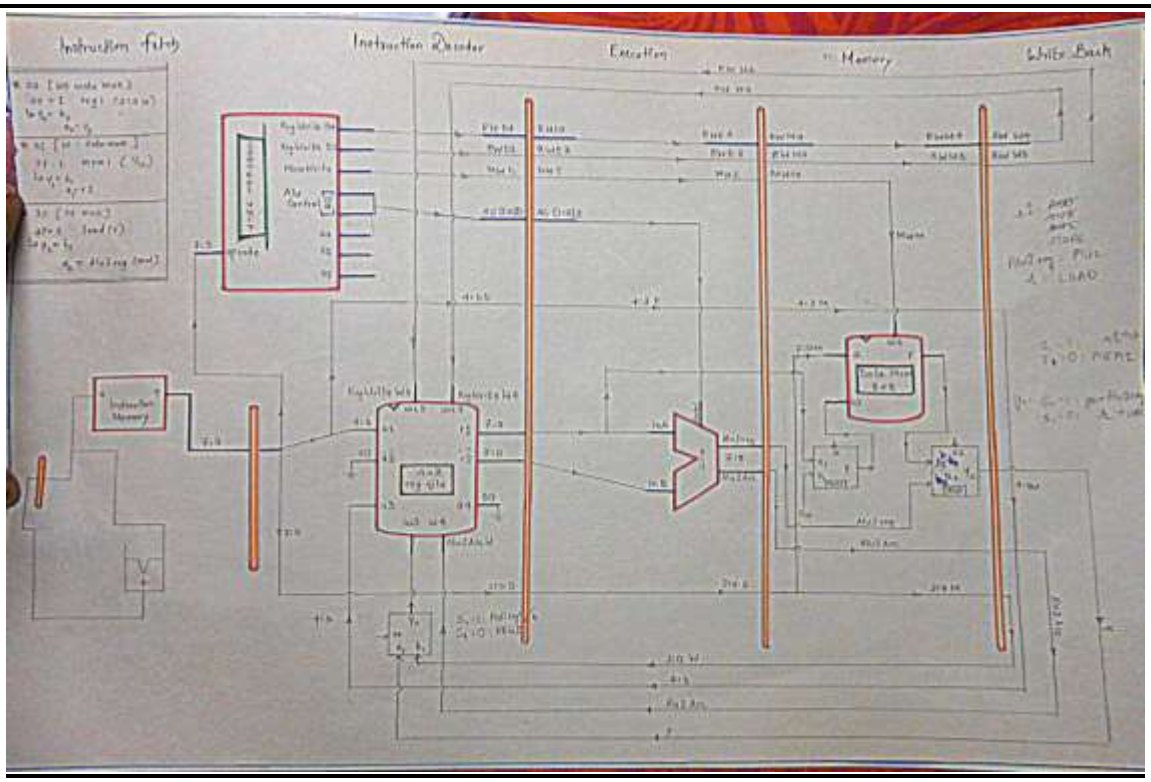
vii. RegI [Reg] ← {5'b0,IMM}



ID/RF stage		Exe stage (ALU)				Mem stage		WB stage	
Ctrl s/g generated [7:5]	Reg-File read stage	i/p		o/p		Input r/w	Ctrl s/g used	Reg-File write stage	Ctrl s/g used
		in A	in B	Alu2 Acc	Alu2 Reg				
RegWrite3	---	---	---	---	---	---	---	{5'b0,IMM(2:0)} to w3 (a3 4:3W)	RegWrite3

❖ **Combined Datapath:**





- The combined data path is capable of executing all the instructions given in pipelined fashion.

- **IF stage** has no modification (no additional hardware)

- **ID/RF stage:** (no additional hardware)
  Control Unit generate 8 control signals depenging on the opcode fed. They are RegWrite3/4, MemWrite, AluControl, s0,s1,s2. (s0,s1,s3 are select lines for additional Muxes used in combinig the data path)

  7:5 (opcode)goes to the control unit.
  4:3(register addr) directly goes to a1 of reg-file for add,su,mul,store. For load, it passes to next stage.
  2:0(memory addr) and 2:0IMM (immediate data of REGI) is passed to the next stage.

- **Exe Stage:** (no additional hardware)
  R1 operand is connected to inA (add,sub,mul,store) ;also passed to the the mux of next stage for Storing instruction in memeory.
  And r2 is connected to in B(add,sub,mul).

- **Mem Stage**
  Gets MemWrite signal from the control unit(for MemI, store). The multiplexer (MUX_1) used choses between 8'b1111_1111 and the input to be stored.
  Data read (r)during load instruction is passed to the next stage MUX_2.

- **WB stage**
  It has two muxes.
  MUX_2 which selects between r and Alu2reg using select line s2.
  MUX_0 which selects between output of MUX_2 and IMM (2:0) data that is used for REGI instruction.

❖ **Instruction Sequence for each Recipe:**

At the beginning

60	MEMI		000	store maximum available chocolate (limited)
61	MEMI		001	store maximum available milk (unlimited)
31	REGI	R3	001	one part of milk
39	REGI	R4	001	one part of chocolate
20	REGI	R1	000	make the contents of accumulator =zero
00	EDP			

Gianduja

(one part of chocolate and one part of milk together)

****

98	ADD	R4		Acc(R1)has R4 (i.e one part of chocolate);Acc=R4
90	ADD	R3		Acc=R4+R3
E2	STORE	R1	010	Acc has desired output and is now sent to 010
21	REGI	R1	001	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

Cou-ver-ture

(one part of chocolate , refill milk then add one part of milk together)

****

98	ADD	R4		Acc(R1)has R4 (i.e one part of chocolate);Acc=R4
61	MEMI		001	refill milk

90	ADD	R3		Acc=R4+R3
E2	STORE	R1	010	Acc has desired output and is now sent to 010
21	REGI	R1	001	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

Dark Chocolate

(two parts of chocolate and add one part of milk together)

****

98	ADD	R4		Acc(R1)has R4 (i.e one part of chocolate);Acc=R4
98	ADD	R4		Acc=R4+R4
90	ADD	R3		Acc=R4+R4+R3
E2	STORE	R1	010	Acc has desired output and is now sent to 010
22	REGI	R1	010	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

Sweet chocolate

(agitate one part of chocolate in a mixing bucket

then agitate one part of milk in the same)

****

90	ADD	R3	Acc=Acc+R3 (Acc←1)
D8	MUL	R4	{R4,Acc}←R4 * Acc

39	REGI	R4	001	
D0	MUL	R3		{R3,Acc}←R3 * Acc
31	REGI	R3	001	
E2	STORE	R1	010	Acc has desired output and is now sent to 010
21	REGI	R1	001	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

Chocolate Caramel

(agitate one part of chocolate

refill milk

then agitate chocolate again)

****

90	ADD	R3		Acc=Acc+R3 (Acc←1)
D8	MUL	R4		{R4,Acc}←R4 * Acc
61	MEMI		001	store maximum available milk (unlimited) i.e refill
D8	MUL	R4		{R4,Acc}←R4 * Acc
E2	STORE	R1	010	Acc has desired output and is now sent to 010
22	REGI	R1	010	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

Milk Chocolate

(agitate chocolate once

then agitate milk twice)

****

90	ADD	R3		Acc=Acc+R3 (Acc←1)
D8	MUL	R4		{R4,Acc}←R4 * Acc
D0	MUL	R3		{R3,Acc}←R3 * Acc
D0	MUL	R3		{R3,Acc}←R3 * Acc
E2	STORE	R1	010	Acc has desired output and is now sent to 010
21	REGI	R1	001	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

White Chocolate

(Add infinite quantity of milk

then add Chocolate)

****

41	LOAD	R1	001	Acc now has infinite quantity of milk
98	ADD	R4		Acc ←(infinite milk + one part of chocolate)
E2	STORE	R1	010	Acc has desired output and is now sent to 010
21	REGI	R1	001	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

Chocolate Praline

(Add infinite quantity of milk

refill the same

add chocolate)

****

41	LOAD	R1	001	Acc now has infinite quantity of milk
61	MEMI		001	store maximum available milk (unlimited) i.e refill
98	ADD	R4		Acc ←(infinite milk + one part of chocolate)
E2	STORE	R1	010	Acc has desired output and is now sent to 010
21	REGI	R1	001	Acc now has the chocolate counted that was used
48	LOAD	R2	000	R2 has the total chocolate available
00	EDP			
A8	SUB	R2		Acc=R2-Acc
E0	STORE	R1	000	current available chocolate counts updated at 000
00	EDP			

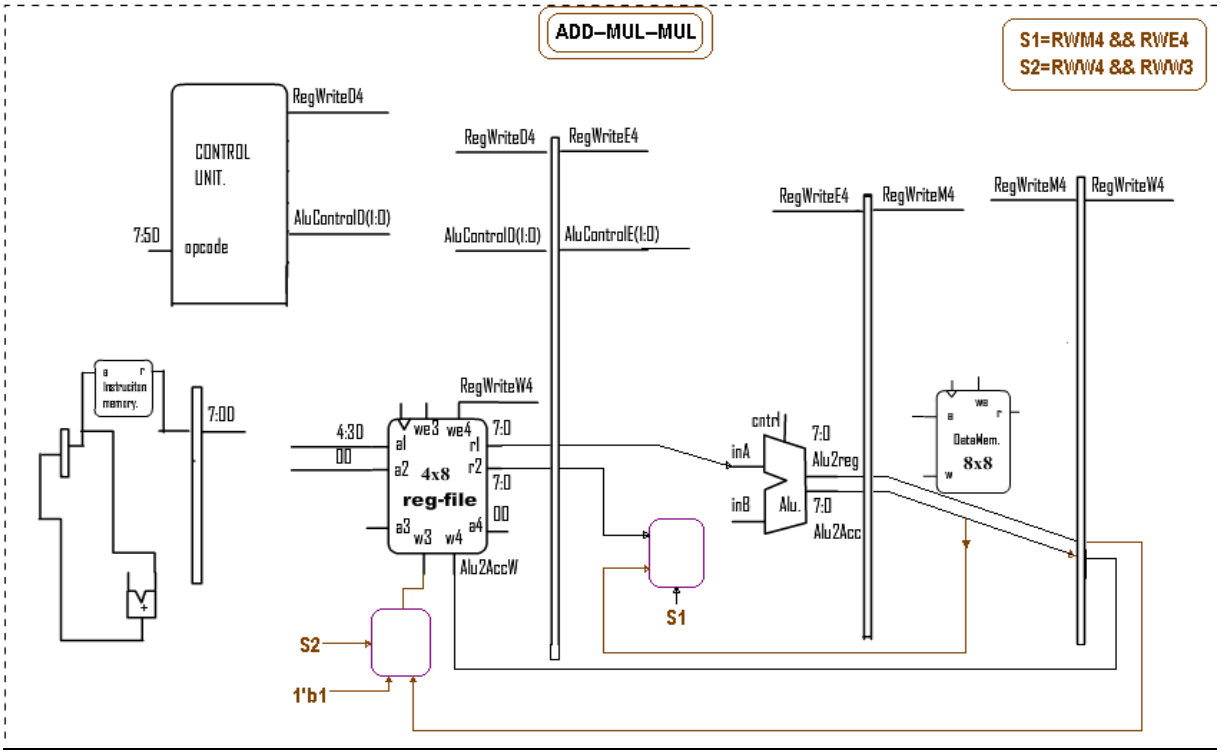
- **Hazards:**

  Structural Hazard: We have tried avoiding structural hazard by **using separate units.** For the adder in ALU and the adder for incrementing the Program Counter is different so that the resources are not blocked

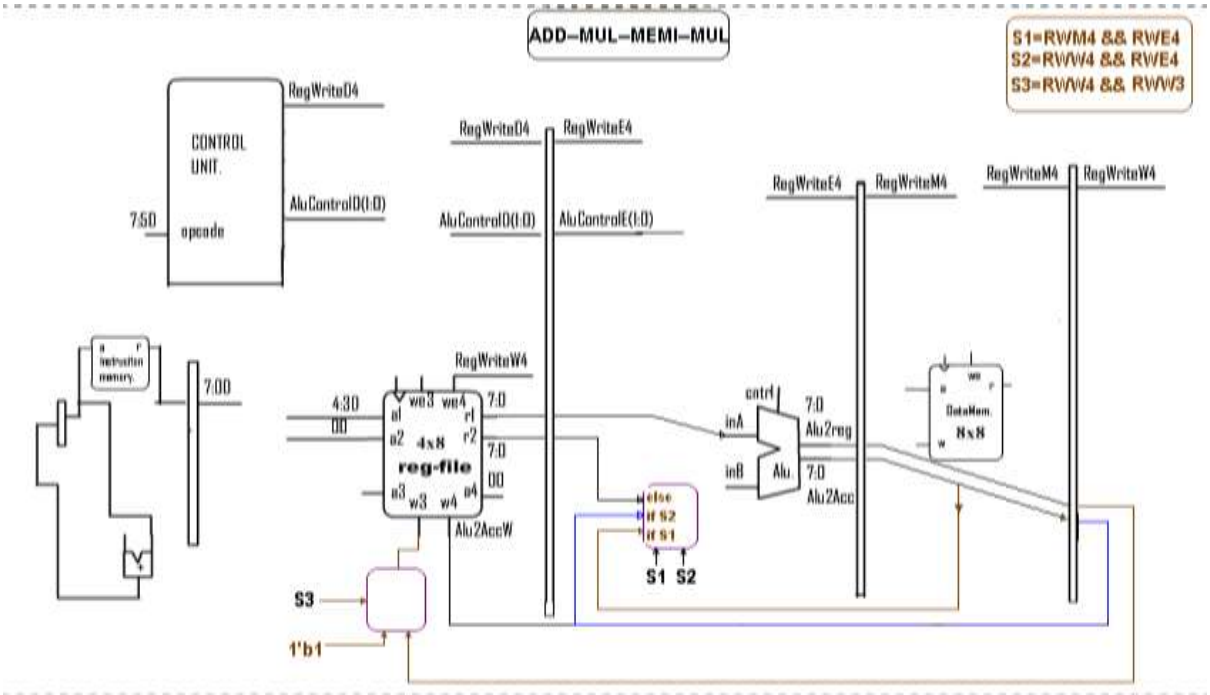
  Control Hazard: We have tried avoiding control hazard by **using the control signal to a unit when its function has to be performed** and **not when** the control signal is **generated.**
  For eg: in LOAD instruction, RegWrite3 is generated during ID stage but it is used only during WB stage when the data is written into the register file.

  Data Hazard: There are instances where the **results generated are immediately required as an operand** for the next instruction. This leads to data hazard. The data hazards that we encounter here are:-

1)Add—Mul—Mul



2)Add—Mul—MemI—Mul

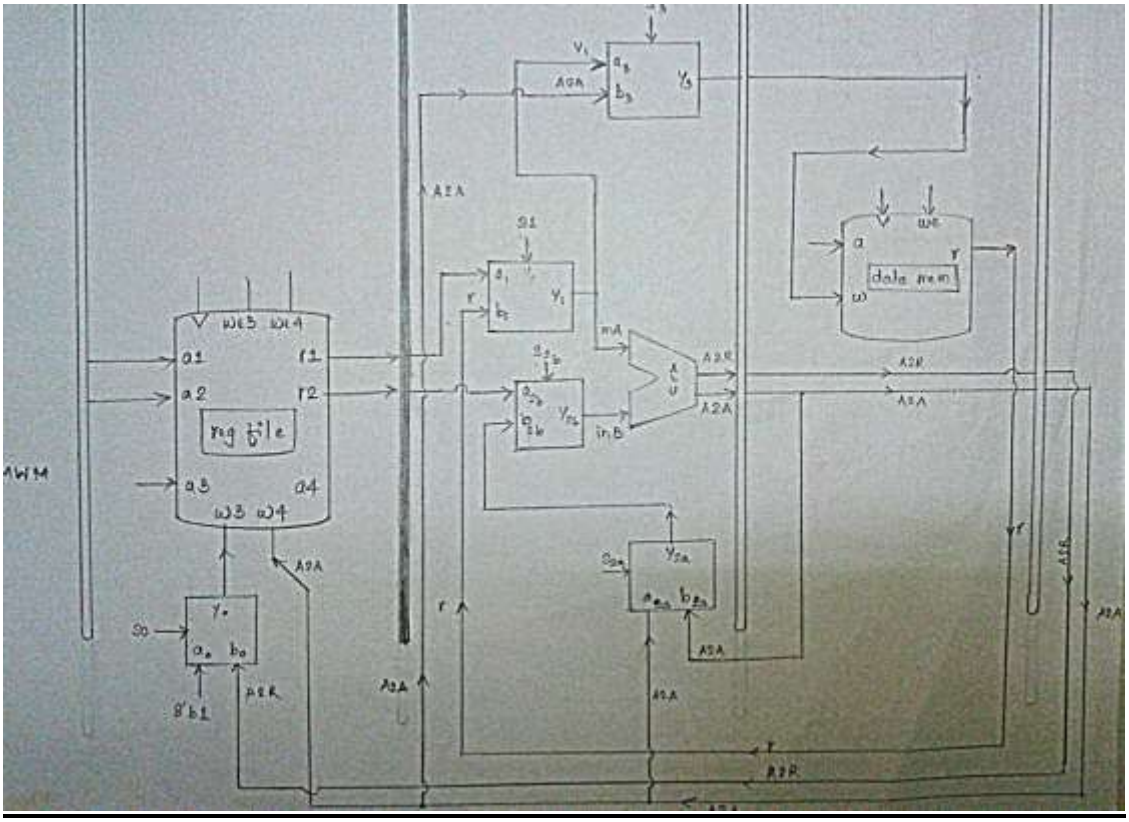


4)load—Add—Add (stalling i.e waiting till the result to be used is prodeuced)

5)load—Add-MemI—Add (stalling i.e waiting till the result to be used is prodeuced)

We get a solution for repitative arithematic hazards using data forwarding technique and for memory related instructions using stalling.

- **Combined Hazard Unit:**



Mux with select line S0

S0 =RWW4 && RWE4

Mux with select line S1

S1 =RWM3 && RWE4

Mux with select line S2a

S2a =RWW4 && RWE4 && MWM

Mux with select line S2b

S2b =RWM4 && MWE

3. Carrying out the plan:

- **Verilog code of indivual modules:**

```
module
Processor(clk,first,adder_out,D,rd1,rd2,alu2accE,alu2regE,alu2accM,alu2regM,alu2acc
W,alu2regW,rM,m1,inA,inB,MWM);
input clk;
input first;
output
[7:0]adder_out,D,rd1,rd2,alu2accE,alu2regE,alu2accM,alu2regM,alu2accW,alu2regW,r
M,m1,inA,inB;
output MWM;


//#########################################################################
#####################################################//
wire [7:0]rd1,rd2,re1,re2,rM;
wire [7:0]D;
wire [4:0]E,M,W;
wire sd0,sd1,sd2,se0,se1,se2,sm0,sm1,sm2,sw0,hold;
wire [7:0]e3,e4,m4,m2,m1,w2,w1;
wire [7:0]inA,inB,alu2accE,alu2regE,alu2accM,alu2regM,y,alu2accW;
wire RWD4,RWD3,MWD,RWE4,RWE3,MWE,RWM4,RWM3,MWM,RWW4,RWW3;
wire [1:0]ACD,ACE;
wire [7:0]adder_out,adder_in;
wire [7:0]b;
//#########################################################################
#####################################################//

/**************************************************************************
**************************************************************************
************/
Buff2 A0(adder_out,clk,1'b0,adder_in);
/**************************************************************************
**************************************************************************
********************/

Prog_counter PC(adder_out,clk,first);

InstructionMemory IM(adder_in,b);
```

```
/*****************************************************************************
*****************************************************************************
************/
Buff2 B0(b,clk,1'b0,D);
/*****************************************************************************
*****************************************************************************
********************/

Control_Unit CU(D[7:5],RWD4,RWD3,MWD,ACD,sd0,sd1,sd2,hold);

reg_file R(D[4:3],W[4:3],y,alu2accW,RWW3,RWW4,rd1,rd2,clk);

/*****************************************************************************
*****************************************************************************
************************/
Buff3
C0({RWD4,RWD3,MWD,ACD[1:0],sd0,sd1,sd2,D[4:3],rd1[7:0],rd2[7:0],D[2:0]},clk,1'
b0,{RWE4,RWE3,MWE,ACE[1:0],se0,se1,se2,E[4:3],re1[7:0],re2[7:0],E[2:0]});
/*****************************************************************************
*****************************************************************************
************************/

Mux2_1 E1(re1,rM,(RWM3&&RWE4),inA);
Mux2_1 E2(re2,e3,(RWM4&&RWE4),inB);
Mux2_1 E3(alu2accM,alu2accW,(RWW4&&RWE4&&MWM),e3);
Mux2_1 E4(inA,alu2accM,(RWM4&&MWE),e4);

Alu alu(inA,inB,ACE[1:0],alu2accE,alu2regE);

/*****************************************************************************
*****************************************************************************
************************/
Buff4
D0({RWE4,RWE3,MWE,se0,se1,se2,E[4:3],e4[7:0],alu2regE[7:0],alu2accE[7:0],E[2:0]
},clk,1'b0,{RWM4,RWM3,MWM,sm0,sm1,sm2,M[4:3],m4[7:0],alu2regM[7:0],alu2acc
M[7:0],M[2:0]});
/*****************************************************************************
*****************************************************************************
*********************************/
```

```
Mux2_1 M1(m4,8'b1111_1111,sm1,m1);

DataMemory DM(M[2:0],m1,clk,MWM,rM);

Mux2_1 M2(alu2regM,rM,sm2,m2);

/*************************************************************************
*************************************************************************
*********************************/
Buff5
E0({RWM4,RWM3,sm0,M[4:3],m2[7:0],alu2accM[7:0],M[2:0]},clk,1'b0,{RWW4,RW
W3,sw0,W[4:3],w2[7:0],alu2accW,W[2:0]});
/*************************************************************************
*************************************************************************
*********************************/

Mux2_1 W1(w2,{5'b0,W[2:0]},sw0,w1);
Mux2_1 W2(w1,8'b1,(RWW4&&RWW3),y);

/*************************************************************************
*************************************************************************
*************************************/

endmodule
```

```
module Buff2(
   input [7:0] a,
   input clk,
   input en,
   output reg[7:0] y
   );

always@(posedge clk)
begin
        if(!en)
                y<=a;
        else
                y<=y;
```

```
end
endmodule
```

```
module Prog_counter(addr_out,clk,first);

output[7:0]addr_out;
//input [7:0]addr_in;
input clk,first;

reg [7:0]addr_update;
wire [7:0]input_adder;
reg [7:0]addr_in;


always@(posedge clk)
begin
addr_update<=addr_out;
if(first)
addr_in<=8'b0;
end

mux2to1 M[7:0](input_adder,addr_in,addr_update,first);

full_adder  a(addr_out,,input_adder,8'b0000_0001,1'b0);

endmodule
```

```
module mux2to1(y,a,b,s);

output y;
input a,b,s;

assign y=((a&s)|(b&~s));

endmodule
```

```
module full_adder(y0,cout,p,q,cin
   );
output [7:0]y0;
```

```
output cout;
input [7:0]p,q;
input cin;

wire [7:0]c;

add1 add_0(y0[0],c[0],p[0],q[0],cin);
add1 add_1(y0[1],c[1],p[1],q[1],c[0]);
add1 add_2(y0[2],c[2],p[2],q[2],c[1]);
add1 add_3(y0[3],c[3],p[3],q[3],c[2]);
add1 add_4(y0[4],c[4],p[4],q[4],c[3]);
add1 add_5(y0[5],c[5],p[5],q[5],c[4]);
add1 add_6(y0[6],c[6],p[6],q[6],c[5]);
add1 add_7(y0[7],c[7],p[7],q[7],c[6]);


assign cout=c[7];                    //Gives final carry

endmodule
```

```
module add1(y2,c,p,q,cin
          );
output y2,c;
input p,q,cin;

wire T1,T2,T3;

xor(T1,p,q),
  (y2,cin,T1);
and(T3,T1,cin),
  (T2,p,q);
or(c,T3,T2);
endmodule
```

```
module InstructionMemory #(parameter N=8,M=8)(
//N:Instrumction Name Address Length
 //M: Instruction Lenth
 input [N-1:0]a,
 output [M-1:0]r
 );

 reg [M-1:0]rom[2**N-1:0];

 initial $readmemh("instructions.txt",rom);

 assign r=rom[a];

endmodule
```

```
module Buff2(
    input [7:0] a,
    input clk,
    input en,
    output reg[7:0] y
    );

always@(posedge clk)
begin
        if(!en)
                y<=a;
        else
                y<=y;
end
endmodule
```

```
module Control_Unit(opcode,RWD4,RWD3,MWD,AluControl,s0,s1,s2,hold);

input[2:0]opcode;
output reg[1:0]AluControl;
output reg RWD4,RWD3,MWD,s0,s1,s2,hold;

always@(opcode)
begin
        case(opcode)
```

```
3'b100://add
        begin
        RWD4<=1'b1;
        RWD3<=1'b0;
        MWD<=1'b0;
        AluControl<=2'b00;
        s0<=1'b0;
        s1<=1'b0;
        s2<=1'b0;
        hold<=1'b0;
        end
3'b101://sub
        begin
        RWD4<=1'b1;
        RWD3<=1'b0;
        MWD<=1'b0;
        AluControl<=2'b01;
        s0<=1'b0;
        s1<=1'b0;
        s2<=1'b0;
        hold<=1'b0;
        end
3'b110://mul
        begin
        RWD4<=1'b1;
        RWD3<=1'b1;
        MWD<=1'b0;
        AluControl<=2'b10;
        s0<=1'b0;
        s1<=1'b0;
        s2<=1'b0;
        hold<=1'b0;
        end
3'b111://store
        begin
        RWD4<=1'b0;
        RWD3<=1'b0;
        MWD<=1'b1;
        AluControl<=2'b11;
        s0<=1'b0;
```

```
            s1<=1'b0;
            s2<=1'b0;
            hold<=1'b0;
            end
    3'b010://load
            begin
            RWD4<=1'b0;
            RWD3<=1'b1;
            MWD<=1'b0;
            AluControl<=2'b11;
            s0<=1'b0;
            s1<=1'b0;
            s2<=1'b1;
            hold<=1'b0;
            end
    3'b011://memi
            begin
            RWD4<=1'b0;
            RWD3<=1'b0;
            MWD<=1'b1;
            AluControl<=2'b11;
            s0<=1'b0;
            s1<=1'b1;
            s2<=1'b0;
            hold<=1'b0;
            end
    3'b001://regi
            begin
            RWD4<=1'b0;
            RWD3<=1'b1;
            MWD<=1'b0;
            AluControl<=2'b11;
            s0<=1'b1;
            s1<=1'b0;
            s2<=1'b0;
            hold<=1'b0;
            end
    default://edp
            begin
            RWD4<=1'b0;
```

```
                RWD3<=1'b0;
                MWD<=1'b0;
                AluControl<=2'b11;
                s0<=1'b0;
                s1<=1'b0;
                s2<=1'b0;
                hold<=1'b1;
                end
        endcase
end


endmodule
```

```
module reg_file(a1,a3,w3,w4,we3,we4,r1,r2,clk);

input [1:0]a1,a3;
input we3,we4,clk;
input [7:0]w3,w4;
output reg[7:0]r1,r2;

reg[7:0]x[3:0];

initial
begin
x[0]=8'b0;
x[1]=8'b0;
x[2]=8'b1;
x[3]=8'b1;
end

always@(negedge clk)
        begin
                r1=x[a1];
                r2=x[2'b0];
        end

always@(posedge clk)
        begin
```

```
                if(we3)
                        x[a3]=w3;

                else if(we4)
                        x[2'b0]=w4;

        end

endmodule
```

```
module Buff3(
   input [28:0] a,
   input clk,
   input en,
   output reg[28:0] y
   );

always@(posedge clk)
begin
        if(!en)
                y<=a;
        else
                y<=y;
end
endmodule
```

```
module Mux2_1(
   input [7:0] a,
   input [7:0] b,
   input s,
   output reg[7:0] y
   );

always@(s,a,b)
begin
        if(s)
                y<=b;
        else
                y<=a;
end
```

```
endmodule
```

```
module Alu(a,b,s,a2a,a2r
   );
  input [7:0]a,b;
  input [1:0]s;
  output reg [7:0]a2a,a2r;

reg [7:0]temp;
reg [15:0]temp1,temp2;
wire [15:0]t1,t3;
wire cout;

always@(s,b)
 begin
  if (s[0]==0)
                temp=b;
  else
                temp=~b;
 end

always@(a,temp)
begin
temp1={8'b0,a[7:0]};
temp2={8'b0,temp[7:0]};
end

Adder16 m0(t1,,temp1,temp2,s[0]);
mul8 m2(a,b,t3);

always@(t1,t3,s)
 begin

        if(s==2'b10)
                begin
                        a2a=t3[7:0];
                        a2r=t3[15:8];
                end
```

```
        else if(s[1]==0)
                begin
                        a2a=t1[7:0];
                        a2r=t1[15:8];
                end

        else
            begin
                a2a=8'b0;
                a2r=8'b0;
            end

 end

endmodule
```

```
module Adder16(
   output [15:0] y,
          output cout,
   input [15:0] a,
   input [15:0] b,
   input cin
   );

wire T0,T1,T2;

Adder4 B1(y[3:0],T0,a[3:0],b[3:0],cin);
Adder4 B2(y[7:4],T1,a[7:4],b[7:4],T0);
Adder4 B3(y[11:8],T2,a[11:8],b[11:8],T1);
Adder4 B4(y[15:12],cout,a[15:12],b[15:12],T2);

endmodule
```

```
module Adder4(
   output[3:0]sum,
   output cout,
   input [3:0]a,
   input [3:0]b,
   input cin
```

```
    );

wire [2:0]y;
        Adder1 A0(sum[0],y[0],a[0],b[0],cin);
        Adder1 A1(sum[1],y[1],a[1],b[1],y[0]);
        Adder1 A2(sum[2],y[2],a[2],b[2],y[1]);
        Adder1 A3(sum[3],cout,a[3],b[3],y[2]);

endmodule
```

```
module Adder1(
    output sum,
    output cout,
    input a,
    input b,
    input cin
    );

wire y1,y2,y3;

        xor (y1,a,b);
        xor (sum,cin,y1);

        and (y2,a,b);
        and (y3,y1,cin);

        or (cout,y2,y3);

endmodule
```

```
module mul8(
    input [7:0] a,
    input [7:0] b,
    output [15:0] M
    );

wire[13:1]Ca; wire[13:2]Cb; wire[12:3]Cc; wire[11:4]Cd; wire[10:5]Ce; wire[9:6]Cf;
wire[8:7]Cg;
wire[13:2]Sa; wire[12:3]Sb; wire[11:4]Sc; wire[10:5]Sd; wire[9:6]Se; wire[8:7]Sf;
```

wire[26:0]Ta; wire[12:2]Tb; wire[11:3]Tc; wire[10:4]Td; wire[9:5]Te; wire[8:6]Tf; wire Tg;

HA A1(Ta[0],Ta[1],M[1],Ca[1]);
HA A2(Ta[2],Ta[3],Sa[2],Ca[2]);
HA A3(Ta[4],Ta[5],Sa[3],Ca[3]);
HA A4(Ta[6],Ta[7],Sa[4],Ca[4]);
HA A5(Ta[8],Ta[9],Sa[5],Ca[5]);
HA A6(Ta[10],Ta[11],Sa[6],Ca[6]);
HA A7(Ta[12],Ta[13],Sa[7],Ca[7]);
HA A8(Ta[14],Ta[15],Sa[8],Ca[8]);

FA B2(Sa[2],Tb[2],Ca[1],M[2],Cb[2]);
FA B3(Sa[3],Tb[3],Ca[2],Sb[3],Cb[3]);
FA B4(Sa[4],Tb[4],Ca[3],Sb[4],Cb[4]);
FA B5(Sa[5],Tb[5],Ca[4],Sb[5],Cb[5]);
FA B6(Sa[6],Tb[6],Ca[5],Sb[6],Cb[6]);
FA B7(Sa[7],Tb[7],Ca[6],Sb[7],Cb[7]);
FA B8(Sa[8],Tb[8],Ca[7],Sb[8],Cb[8]);
FA A9(Ta[16],Ta[17],Ca[8],Sa[9],Ca[9]);

FA C3(Sb[3],Tc[3],Cb[2],M[3],Cc[3]);
FA C4(Sb[4],Tc[4],Cb[3],Sc[4],Cc[4]);
FA C5(Sb[5],Tc[5],Cb[4],Sc[5],Cc[5]);
FA C6(Sb[6],Tc[6],Cb[5],Sc[6],Cc[6]);
FA C7(Sb[7],Tc[7],Cb[6],Sc[7],Cc[7]);
FA C8(Sb[8],Tc[8],Cb[7],Sc[8],Cc[8]);
FA B9(Sa[9],Tb[9],Cb[8],Sb[9],Cb[9]);
FA A10(Ta[19],Ta[18],Ca[9],Sa[10],Ca[10]);

FA D4(Sc[4],Td[4],Cc[3],M[4],Cd[4]);
FA D5(Sc[5],Td[5],Cc[4],Sd[5],Cd[5]);
FA D6(Sc[6],Td[6],Cc[5],Sd[6],Cd[6]);
FA D7(Sc[7],Td[7],Cc[6],Sd[7],Cd[7]);
FA D8(Sc[8],Td[8],Cc[7],Sd[8],Cd[8]);
FA C9(Sb[9],Tc[9],Cc[8],Sc[9],Cc[9]);
FA B10(Sa[10],Tb[10],Cb[9],Sb[10],Cb[10]);

```
FA A11(Ta[21],Ta[20],Ca[10],Sa[11],Ca[11]);

FA E5(Sd[5],Te[5],Cd[4],M[5],Ce[5]);
FA E6(Sd[6],Te[6],Cd[5],Se[6],Ce[6]);
FA E7(Sd[7],Te[7],Cd[6],Se[7],Ce[7]);
FA E8(Sd[8],Te[8],Cd[7],Se[8],Ce[8]);
FA D9(Sc[9],Td[9],Cd[8],Sd[9],Cd[9]);
FA C10(Sb[10],Tc[10],Cc[9],Sc[10],Cc[10]);
FA B11(Sa[11],Tb[11],Cb[10],Sb[11],Cb[11]);
FA A12(Ta[23],Ta[22],Ca[11],Sa[12],Ca[12]);

FA F6(Se[6],Tf[6],Ce[5],M[6],Cf[6]);
FA F7(Se[7],Tf[7],Ce[6],Sf[7],Cf[7]);
FA F8(Se[8],Tf[8],Ce[7],Sf[8],Cf[8]);
FA E9(Sd[9],Te[9],Ce[8],Se[9],Ce[9]);
FA D10(Sc[10],Td[10],Cd[9],Sd[10],Cd[10]);
FA C11(Sb[11],Tc[11],Cc[10],Sc[11],Cc[11]);
FA B12(Sa[12],Tb[12],Cb[11],Sb[12],Cb[12]);
FA A13(Ta[25],Ta[24],Ca[12],Sa[13],Ca[13]);

FA G7(Sf[7],Tg,Cf[6],M[7],Cg[7]);
FA G8(Sf[8],Cf[7],Cg[7],M[8],Cg[8]);
FA F9(Se[9],Cf[8],Cg[8],M[9],Cf[9]);
FA E10(Sd[10],Ce[9],Cf[9],M[10],Ce[10]);
FA D11(Sc[11],Cd[10],Ce[10],M[11],Cd[11]);
FA C12(Sb[12],Cc[11],Cd[11],M[12],Cc[12]);
FA B13(Sa[13],Cb[12],Cc[12],M[13],Cb[13]);
FA A14(Ta[26],Ca[13],Cb[13],M[14],M[15]);

and (M[0],a[0],b[0]);

and (Ta[0],a[0],b[1]);
and (Ta[1],a[1],b[0]);
and (Ta[2],a[2],b[0]);
and (Ta[3],a[1],b[1]);
and (Ta[4],a[3],b[0]);
and (Ta[5],a[2],b[1]);
and (Ta[6],a[4],b[0]);
and (Ta[7],a[3],b[1]);
and (Ta[8],a[5],b[0]);
```

```
and (Ta[9],a[4],b[1]);
and (Ta[10],a[6],b[0]);
and (Ta[11],a[5],b[1]);
and (Ta[12],a[7],b[0]);
and (Ta[13],a[6],b[1]);
and (Ta[14],a[7],b[1]);
and (Ta[15],a[6],b[2]);
and (Ta[16],a[7],b[2]);
and (Ta[17],a[6],b[3]);
and (Ta[18],a[7],b[3]);
and (Ta[19],a[6],b[4]);
and (Ta[20],a[7],b[4]);
and (Ta[21],a[6],b[5]);
and (Ta[22],a[7],b[5]);
and (Ta[23],a[6],b[6]);
and (Ta[24],a[7],b[6]);
and (Ta[25],a[6],b[7]);
and (Ta[26],a[7],b[7]);

and (Tb[2],a[0],b[2]);
and (Tb[3],a[1],b[2]);
and (Tb[4],a[2],b[2]);
and (Tb[5],a[3],b[2]);
and (Tb[6],a[4],b[2]);
and (Tb[7],a[5],b[2]);
and (Tb[8],a[5],b[3]);
and (Tb[9],a[5],b[4]);
and (Tb[10],a[5],b[5]);
and (Tb[11],a[5],b[6]);
and (Tb[12],a[5],b[7]);

and (Tc[3],a[0],b[3]);
and (Tc[4],a[1],b[3]);
and (Tc[5],a[2],b[3]);
and (Tc[6],a[3],b[3]);
and (Tc[7],a[4],b[3]);
and (Tc[8],a[4],b[4]);
and (Tc[9],a[4],b[5]);
and (Tc[10],a[4],b[6]);
and (Tc[11],a[4],b[7]);
```

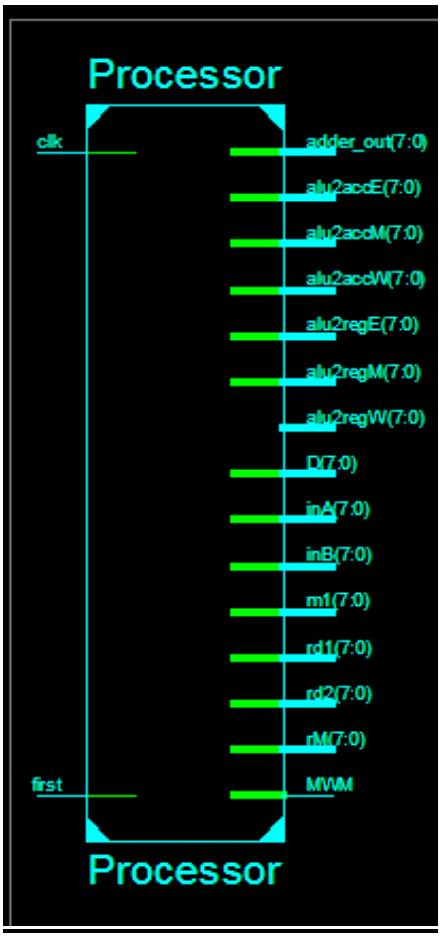
```
and (Td[4],a[0],b[4]);
and (Td[5],a[1],b[4]);
and (Td[6],a[2],b[4]);
and (Td[7],a[3],b[4]);
and (Td[8],a[3],b[5]);
and (Td[9],a[3],b[6]);
and (Td[10],a[3],b[7]);

and (Te[5],a[0],b[5]);
and (Te[6],a[1],b[5]);
and (Te[7],a[2],b[5]);
and (Te[8],a[2],b[6]);
and (Te[9],a[2],b[7]);

and (Tf[6],a[0],b[6]);
and (Tf[7],a[1],b[6]);
and (Tf[8],a[1],b[7]);

and (Tg,a[0],b[7]);

endmodule
```

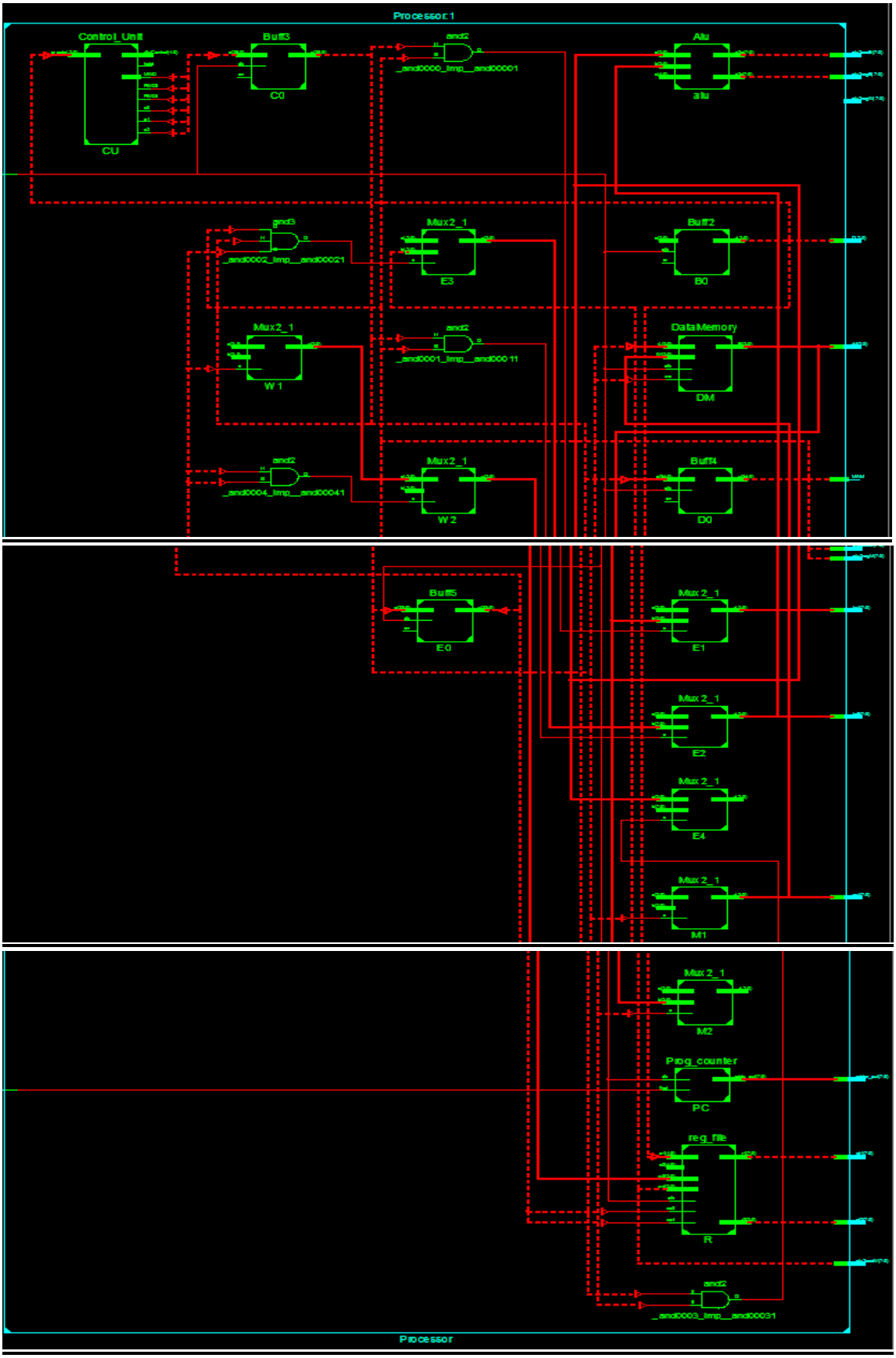
```
module Buff4(
   input [34:0] a,
   input clk,
   input en,
   output reg[34:0] y
   );

always@(posedge clk)
begin
        if(!en)
                y<=a;
        else
                y<=y;
end
endmodule
```

```
module Buff5(
```

```
    input [23:0] a,
    input clk,
    input en,
    output reg[23:0] y
    );

always@(posedge clk)
begin
        if(!en)
                y<=a;
        else
                y<=y;
end
endmodule
```

❖ **RTL Schematic of Main Processor:**




Processor1
Control_Unit
CU
Buff3
C0
and2
_and0000_lmp__and00001
Alu
alu
and3
_and0002_lmp__and00021
Mux2_1
E3
Buff2
B0
Mux2_1
W1
and2
_and0001_lmp__and00011
DataMemory
DM
and2
_and0004_lmp__and00041
Mux2_1
W2
Buff4
D0
Buff5
E0
Mux2_1
E1
Mux2_1
E2
Mux2_1
E4
Mux2_1
M1
Mux2_1
M2
Prog_counter
PC
reg_file
R
and2
_and0003_lmp__and00031
Processor

❖ **Testbench:**

```
module processor_tb;

    // Inputs
    reg clk;
    reg first;

    // Outputs
    wire [7:0] adder_out;
    wire [7:0] D;
    wire [7:0] rd1;
    wire [7:0] rd2;
    wire [7:0] alu2accE;
    wire [7:0] alu2regE;
    wire [7:0] alu2accM;
    wire [7:0] alu2regM;
    wire [7:0] alu2accW;
    wire [7:0] alu2regW;
    wire [7:0] rM;
    wire [7:0] m1;
    wire [7:0] inA;
    wire [7:0] inB;
    wire MWM;

    // Instantiate the Unit Under Test (UUT)
    Processor uut (
        .clk(clk),
        .first(first),
        .adder_out(adder_out),
        .D(D),
        .rd1(rd1),
        .rd2(rd2),
        .alu2accE(alu2accE),
        .alu2regE(alu2regE),
        .alu2accM(alu2accM),
        .alu2regM(alu2regM),
        .alu2accW(alu2accW),
        .alu2regW(alu2regW),
        .rM(rM),
```

```
                .m1(m1),
                .inA(inA),
                .inB(inB),
                .MWM(MWM)
        );

initial begin
                // Initialize Inputs
                clk = 0;
                first = 0;

                // Wait 100 ns for global reset to finish
                #100;

                // Add stimulus here

#20clk=~clk;
first = 1;

#20clk=~clk;
first = 1;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;
```

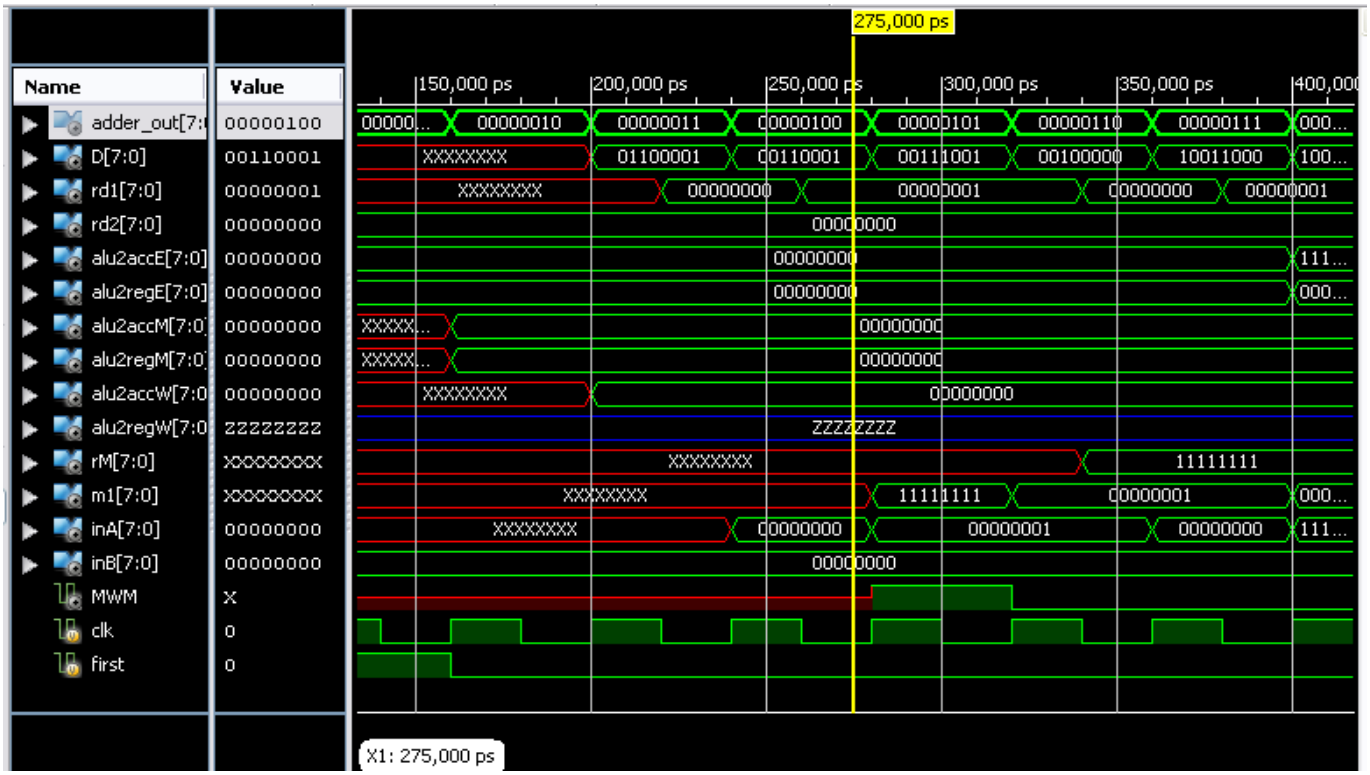
```
#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
```

```
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;

#20clk=~clk;
first = 0;
```

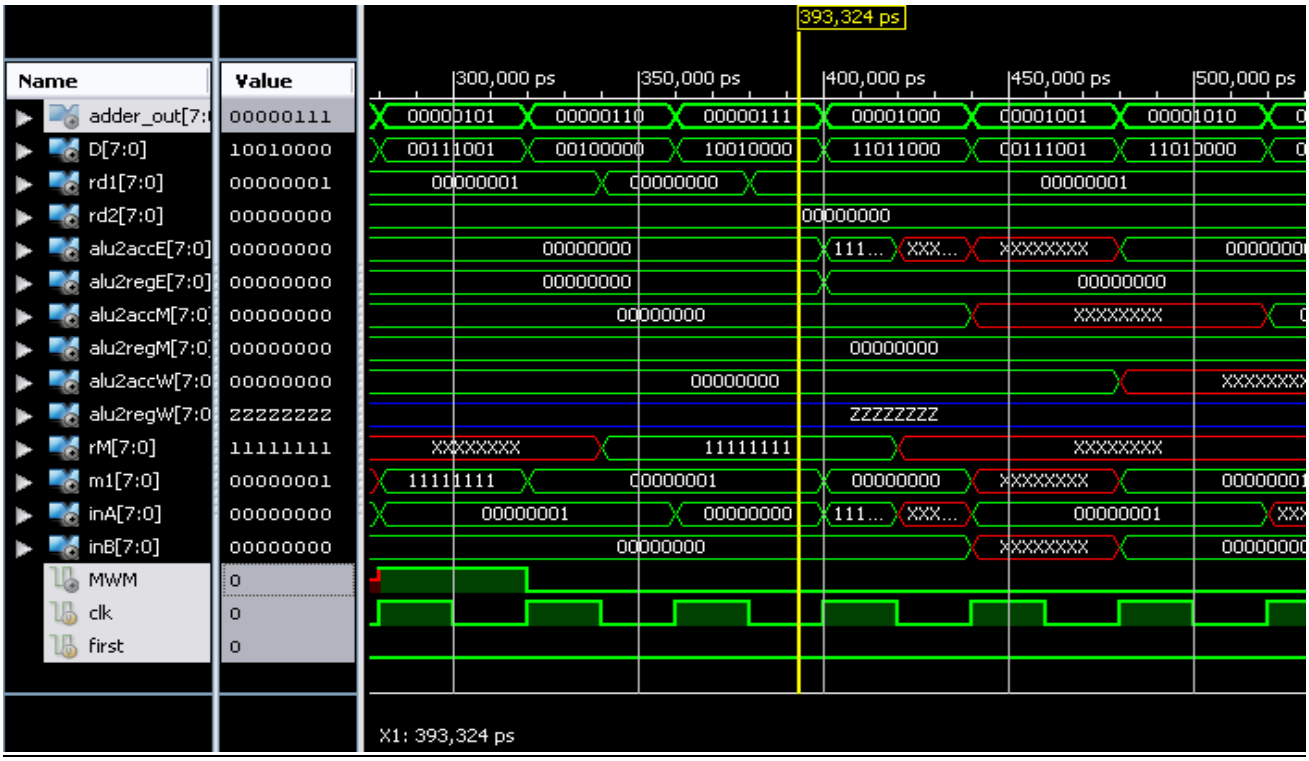
```
#20clk=~clk;
first = 0;
#100;


        end

endmodule
```

- **Test Signals:**

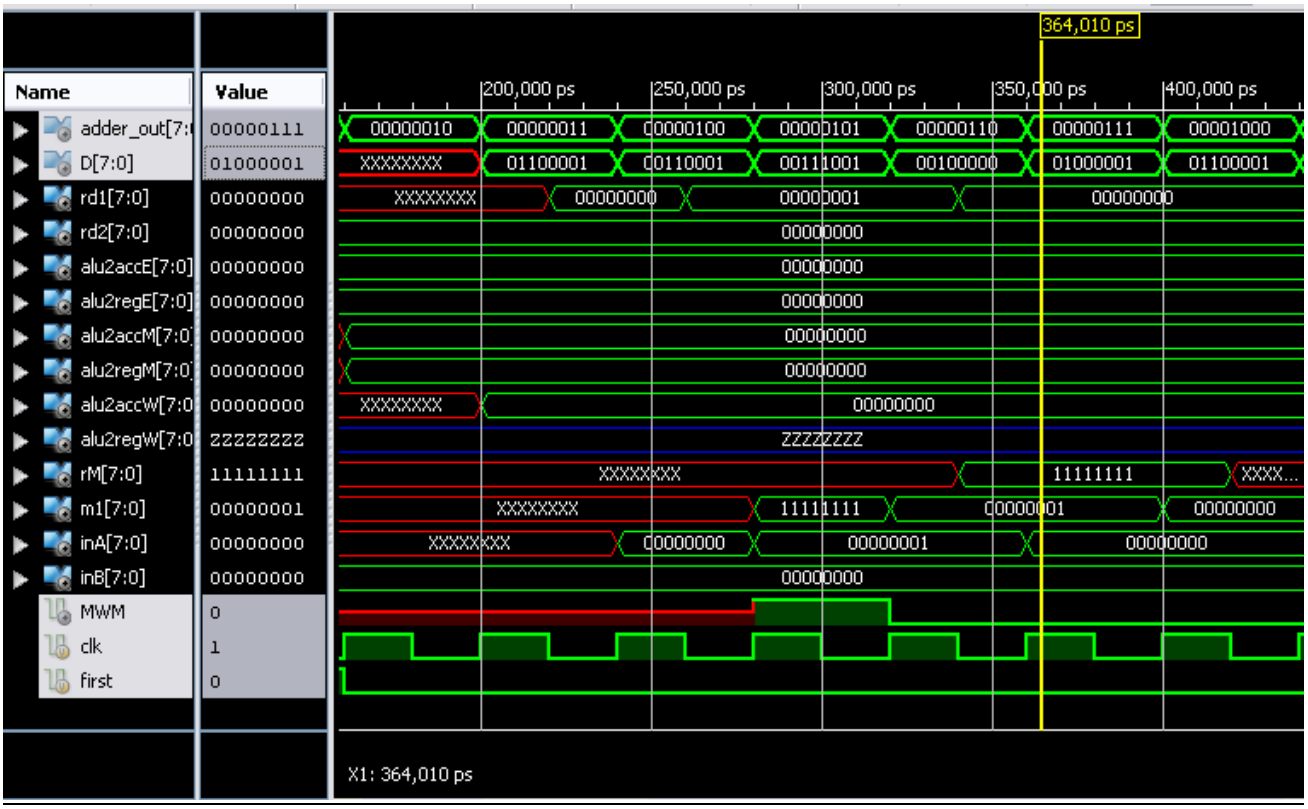
1) Test waveforms (**Gianduja**)



2) Test waveforms (**Sweet chocolate**)



3) Test waveforms (**Chocolate Praline**)



4. Looking Back:

In the final design, we are getting several warnings of the type

- sequential type is unconnected in block <D0>.
- FF/Latch <y_1> (without init value) has a constant value of 0 in block <C0>. This FF/Latch will be trimmed during the optimization process.

Although the processor logically seems correct, we have faced many problems while executing it.

1. The various blocks are not connected.
This is probably because an output which uses all the defined wires in the designed processor is not available. We've been trying to rectify the problem.

2. Connecting FSM and Processor.
The FSM for vending machine is designed so that it gives the address of the recipe as the output which is directly fed to the Program Counter. The PC then chooses the respective data from the text file and keeps incrementing it's value till it reaches a system-halt signal. (Here, EDP). But when using this model, we weren't sure on how to initialize the refilling of chocolate and milk (here, MEM 000 and 001).
The syntax of the text file for this operation is:
@address data
which gives errors (invalid in call of system task $readmemh).

3. EDP.
Here we have used EDP as an instruction where clk signal proceeds as usual but the all the units do not perform any of their function.
It is as good as introducing delay.

.

Synthesis Report

```
=========================================================================
HDL Synthesis Report
=========================================================
Macro Statistics
# RAMs                                                  : 1
 8x8-bit single-port RAM                               : 1
# ROMs                                                  : 1
 8x9-bit ROM                                           : 1
# Registers                                             : 13
 24-bit register                                       : 1
 29-bit register                                       : 1
 35-bit register                                       : 1
 8-bit register                                        : 10
# Multiplexers                                          : 1
 8-bit 4-to-1 multiplexer                              : 1
# Xors                                                  : 104
 1-bit xor2                                            : 56
    1-bit xor3                     : 48
=========================================================

Design Statistics
# IOs                              : 115
Cell Usage :
# BELS             : 13
=========================================================================
```