# EE309: Project Design Report

A 6-stage pipelined processor, with given ISA has been designed and implemented in VHDL. The architecture is also optimized for performance including hazard mitigation techniques

- MADE BY:
  Anugole Sai Gaurav, 170070008
  Parth Shettiwar, 17007
  Jayesh Choudhary, 170070038
  Vishesh Verma, 170070039

## Major Components used in our Data-path

1) **Instruction Memory**:

- Stores the instruction word (16 bit) as per the format specified.
- Outputs the word whenever the corresponding address location is given with the help of Instruction pointer

2) **Decoder:**

- Decodes output from Instruction word to operand/destination registers/ immediate values with corresponding sign extensions as required
- All its output stored in IDRR.

3) **Register File**:

- Contains 8 registers with 2 set of input and output read pins and 1 set of input and output write pin.
- Another port present to directly write data into R7 and one more to read from it since R7 stores the value of IP
- Required output values for follow up stage stored in RREX

4) **ALU1**:

- Solely used for incrementing the value of IP

5) **ALU2:**

- Performs addition, Subtraction and NAND operation according to Opcode and other conditions
- Also modifies the CCR containing carry and zero flag as and when required

6) **ALU3**:

- Performs addition solely for BEQ and JAL operations wherein incremented PC values needs to be updated when required

7) **Data Memory**:

- Stores data in location as per the given input address
- Also enabled to read data from memory corresponding to the given address

8) **LM_SM_block**:

- Solely, used for the purpose of Load and Store multiple instructions
- Includes a priority encoder that gives out the register address values corresponding to the bits set in instruction word to load/store.
- Inbuilt decoder to decide the number of cycles to be executed once a register is loaded into/ stored from the remaining bits left
- Inbuilt ALU to load/store values from increasing address locations in data memory

9) **Multiplexers**:

- 16 bit, 8 bit, 3 bit and 2 bit data multiplexers are designed that are mostly used in the datapath of both 4:1 and 2:1 types to assign one of multiple signals to another signal/port

10) **Branch Prediction Table**:

- Lookup-table to store the jump and branch type IPs whenever found during execution stage for 1st time
- 1-bit history additionally stored, used for prediction table

11) **Pipeline Registers**: Since, there are 6 stages, 5 pipeline registers are implemented. Each one is named corresponding the stages present on either side. These registers are flip-flops that output the values fed to the previous stage to the next during rising clock edge, making it the only synchronous entities in the entire processor.

**IFID:**

- IFID_IP: stores the current IP moving into the Decode stage from Fetch
- IFID_Mem_d: stores corresponding instruction word
- IFID_rst: Clears the content of the register when set
- IFID_flag: Used for branch prediction

**IDRR:**

- IDRR_PC_out: stores the current IP moving into the Register-read stage from decode
- IDRR_rst: Clears the content of the register when set
- IDRR_flag: Used for branch prediction
- IDRR_opcode: opcode obtained from decode logic
- IDRR11_9, IDRR8_6, IDRR5_3,  IDRR7_0: Corresponding bit streams from Instruction word

- IDRRse10_ir5_0, IDRRse7_ir8_0: Sign-extended corresponding bit streams
- IDRRls7_ir8_0: left-shifted bit stream

**RREX:**

- RREX_PC_out: stores the current IP moving into the Execute stage from Read Register
- RREX_rst: Clears the content of the register when set
- RREX_flag: Used for branch prediction
- RREX_dest_reg: stores address of the destination register of instruction currently to be executed
- RREX_r7_out: Stores the data of R7
- RREX_rf_d1, RREX_rf_d2: Stores the data of registers from RF_D1 and RF_D2
- RREX_opcode: opcode obtained from IDRR
- RREX11_9, RREX8_6, RREX5_3, RREX7_0, RREX_se10_ir5_0, RREX_se7_ir8_0, RREX_ls7_ir8_0: corresponding values from IDRR

**EXMA:**

- EXMA_PC_out: stores the current IP moving into the Memory access stage from write back
- EXMA_rst: Clears the content of the register when set
- EXMA_flag: Used for branch prediction
- EXMA_zero_prev, EXMA_carry_prev: bits of CCR register
- EXMA_opcode: opcode obtained from RREX
- EXMA_dest_reg, EXMA_r7_out, EXMA_rf_d1, EXMA_rf_d2: corresponding entities from RREX
- EXMA11_9, EXMA8_6, EXMA5_3, EXMA7_0, EXMA_ls7_ir8_0: corresponding values from RREX
- EXMA_alu2_out, EXMA_alu3_out: ALU2, ALU3 outputs

**MAWB:**

- MAWB_PC_out: stores the current IP moving into the write back stage from memory access
- MAWB_rst: Clears the content of the register when set
- MAWB_flag: Used for branch prediction
- MAWB_zero_prev, MAWB_carry_prev: corresponding bits from EXMA
- MAWB_dest_reg, MAWB_r7_out, MAWB_opcode, MAWB_rf_d1, MAWB_rf_d2: corresponding entities from EXMA
- MAWB11_9, MAWB8_6, MAWB5_3, MAWB7_0, MAWB_ls7_ir8_0, MAWB_alu2_out, MAWB_alu3_out: corresponding values from EXMA
- MAWB_data_mem_out: Output value form data memory

## HAZARDS:

### R7 Hazards

1) LW R7,Rx,Immediate - If we have load instruction with R7 as specified register then we have to change the control flow for which we can use forwarding from memory stage and flushing all the previous instructions.
2) LHI R7, ADD R7,Rx,Ry, Nand R7,Rx,Ry - In these cases, we are changing the PC and the updated value can be found out in execution stage so we are using the forwarding from execution stage and flushing the previous instruction. Also in carry and zero conditional cases, if the flags are not set then we don't need to forward the values as it won't be written back in the register file.
3) JLR Rx, R7 - In this case we have to put the data of register file in PC which can be found out in Register-read stage and so we can directly forward this value and flush the previous pipelines.

   TO IMPLEMENT ALL THESE HAZARD MITIGATION TECHNIQUES WE ARE USING MUX WHOSE CONTROL LOGICS ARE DEFINED BY THE OPCODES OF THE INSTRUCTION IN THE RESPECTIVE PIPELINES/STAGES.

### Dependencies without R7

1) If the operand register for new instructions are same as that of the destination register of  previous instruction, let it be in execution stage or memory stage or writeback stage, there will be errors and we can use forwarding directly from those stages. One thing to be noted is that in case of conditional execution like ADC, ADZ, NDC, NDZ if the respective flags are not set then there is no need for forwarding even if the destination register of previous instructions are same.

a) For all these forwarding we are using mux and the output is to be given to RREX_rf_d1 and RREX_rf_d2.
b) For ADD, ADC, ADZ, NDU, NDC, NDZ and ADI we are forwarding from ALU_2.
c) For LW we are forwarding from the d_mem_data.
d) For JAL and JLR instruction, forwarding is from ALU_3.

If LW is followed by addition or nand then we stall the pipeline for 1 cycle so that load reaches memory stage when add or nand is in register-read stage and then we do the forwarding from d_mem_data

**Branch Instructions:**

1) JLR - get the address in Execution stage, update R7 and hence IP and flush all the instructions before instruction
2) JAL - Branch Prediction Table
3) BEQ - Branch Prediction  Table and also evaluate at Execute stage, if branch was not taken flush the instructions till execution stage. Branch Prediction Table contains always the Taken Branch

**LMSM Hazards:**

In case of LMSM instruction when we get the opcode of LMSM in register-read stage, then disable :
ALU_1 to stop IP from incrementing
IDRR_en to stop values from entering the pipeline
RREX_en to wait and see if some values of register are changing in previous instruction.
Then we stall the pipeline for 3 cycles so as just to ensure that input values are not changing in the previous instructions (for example if IP is changing in previous instruction then there is no need to enter the LMSM block or if the value of registers are changing then we are taking the correct value inside the block. This is necessary since LMSM block is an independent entity). After entering the block the pipelines are enabled according to the state of FSM we are in.

R type instruction

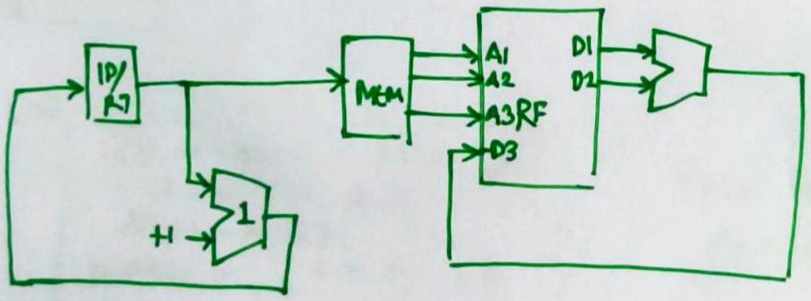
ADD , ADC , ADZ , NDU, NDC , NDZ

$$\boxed{\begin{array}{l} R7 \rightarrow ALU1\_A,\ MEM1\_A \\ +1 \rightarrow ALU1\_B \\ ALU1 \rightarrow R7 \\ MEM1\_D_{11-9} \rightarrow RF\_A1 \\ MEM1\_D_{8-6} \rightarrow RF\_A2 \\ RF\_D1 \rightarrow ALU2\_A \\ RF\_D2 \rightarrow ALU2\text{-}B \\ ALU2 \rightarrow RF\_D3 \\ MEM\_D_{5-3} \rightarrow RF\_A3 \end{array}}$$

Carry and zero flags are set in this case by ALU.

For conditional, we do the computati-on using opcode and carry flag combination. (DECODER STAGE)

For NAND, control of ALU changed to NAND function.

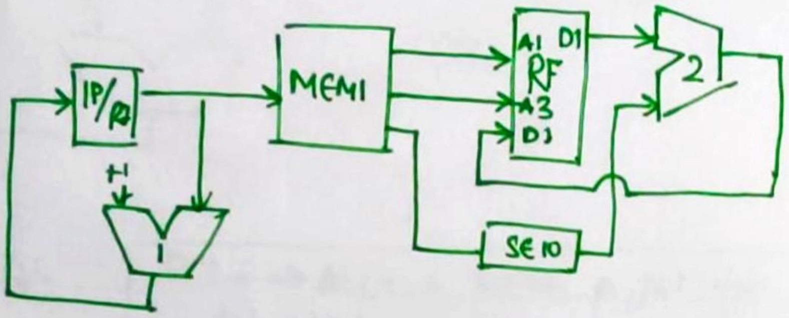


In execution stage we find if carry or zero flag is set or not and then we flush pipeline and take in next instruction.

ADI

R7 ⟶ ALU1A, MEM1_A
+1 ⟶ ALU1_B
ALU1 ⟶ R7
$MEMD_{11-9}$ ⟶ RF_A1
$MEM1\_D_{8-6}$ ⟶ RF_A3
$MEM1\text{-}D_{5-0}$ ⟶ SE10 ⟶ ALU2_B
RF_D1 ⟶ ALU2_A
ALU2 ⟶ RF_D3

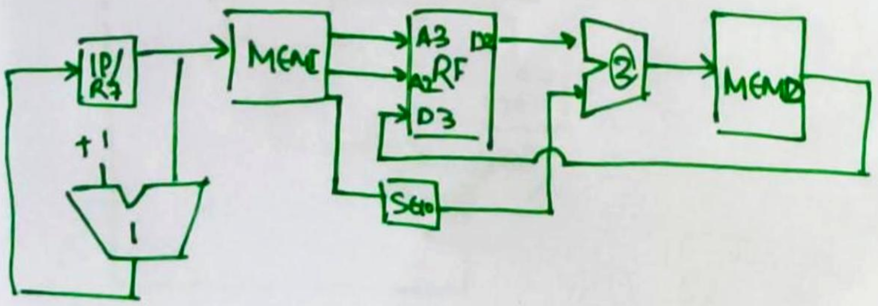
Carry and Zero flags updated here.



LW

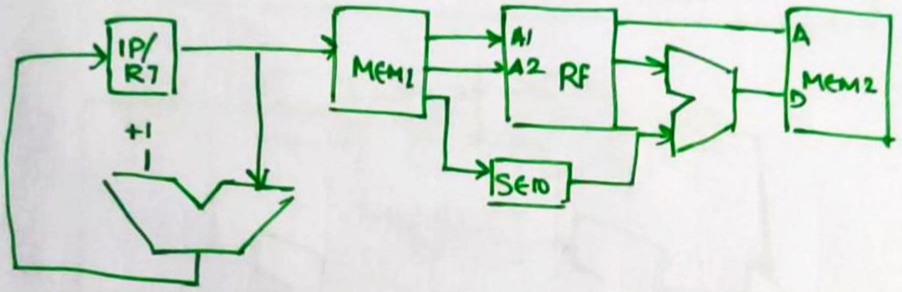
R7 ⟶ ALU1_A, MEM1_A
+1 ⟶ ALU1_B
ALU1 ⟶ R7
$MEM1\_D_{11-9}$ ⟶ RF_A3
$MEM1\_D_{8-6}$ ⟶ RF_A1
RF_D1 ⟶ ALU2_A
$MEMD_{5-0}$ ⟶ SE10 ⟶ ALU2_B
ALU2 ⟶ MEM2_A
MEM2_D ⟶ RF_D3

Zero flag modified here.



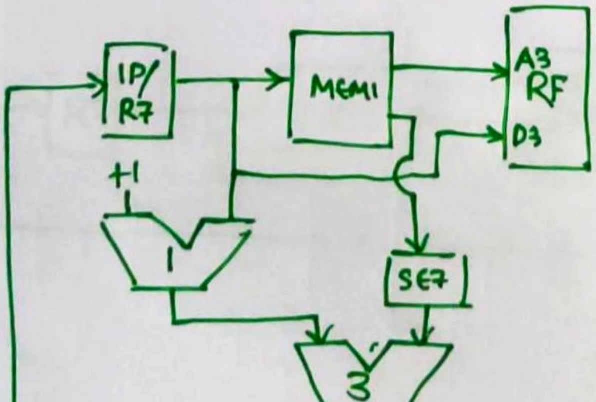
SW

$R7 \rightarrow ALU1\_A, MEM1\_A$
$+1 \rightarrow ALU1\text{-}B$
$ALU1 \rightarrow R7$
$MEM1\_D_{11\text{-}9} \rightarrow RF\_A1$
$MEM1\_D_{8\text{-}6} \rightarrow RF\_A2$
$RF\_D2 \rightarrow ALU2\text{-}A$
$MEM1D_{5\text{-}0} \rightarrow SE10 \rightarrow ALU2\_B$
$ALU2 \rightarrow MEM2\text{-}A$
$RF\_D1 \rightarrow MEM2\text{-}D$



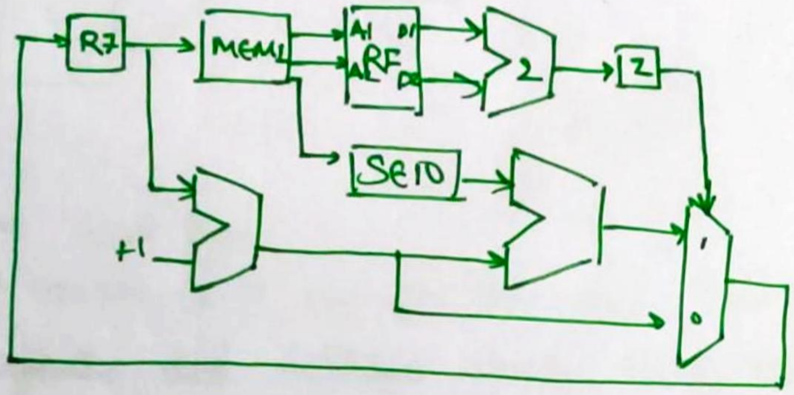
JAL

$R7 \rightarrow ALU1\_A, MEM1\_A, RF\_D3$
$+1 \rightarrow ALU1\_B$
$MEM1\text{-}D_{11\text{-}9} \rightarrow RF\_A3$
$MEM1\_D_{8\text{-}0} \rightarrow SE7 \rightarrow ALU3\text{-}B$
$ALU1 \rightarrow ALU3\_A$
$ALU3 \rightarrow R7$



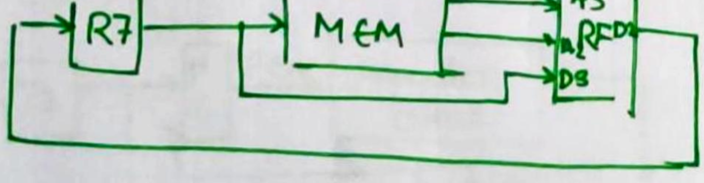
BEQ

R7 → ALU1_A, MEM1_A
+1 → ALU1_B
MEM1_$D_{11-9}$ → RF_A1
MEM1_$D_{8-6}$ → RF_A2
RF_D1 → ALU2_A
RF_D2 → ALU2_B
ALU1 → ALU3_A
MEM1_$D_{5-0}$ → SE10 → ALU3_B
IF ALU2-Z = 1 : ALU3 → R7
else : ALU1 → R7



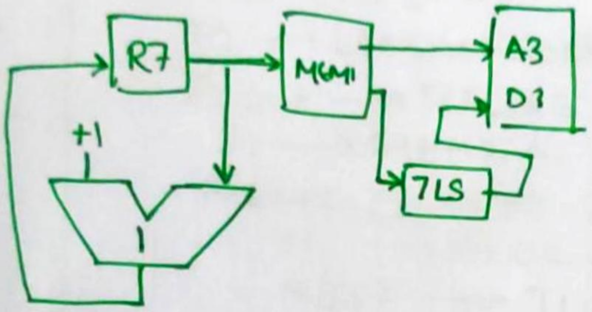
JLR

R7 → MEM1_A, RF_D3
MEM1 –$D_{11-9}$ → RF_A3
MEM1_$D_{8-6}$ → RF_A2
RF_D2 → R7



LHI

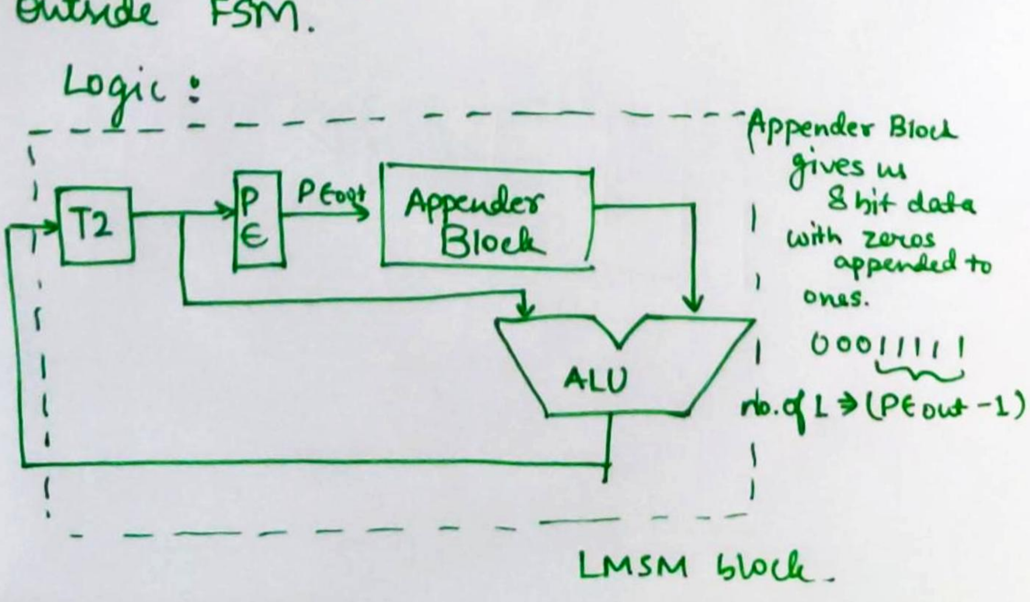
R7 ⟶ ALU1_A, MEM1_A
+1 ⟶ ALU1_B
ALU1 ⟶ R7
MEM1_D11-9 ⟶ RF_A3
MEM1_D8-0 ⟶ LS7 ⟶ RF_D3



LM and SM.

If opcode is of LM and SM, then the control signals are defined using FSM otherwise regular definition is there.
If we are in FSM then, pipelines are disabled as per in which stage we are and other signal are same as defined outside FSM.

Logic :

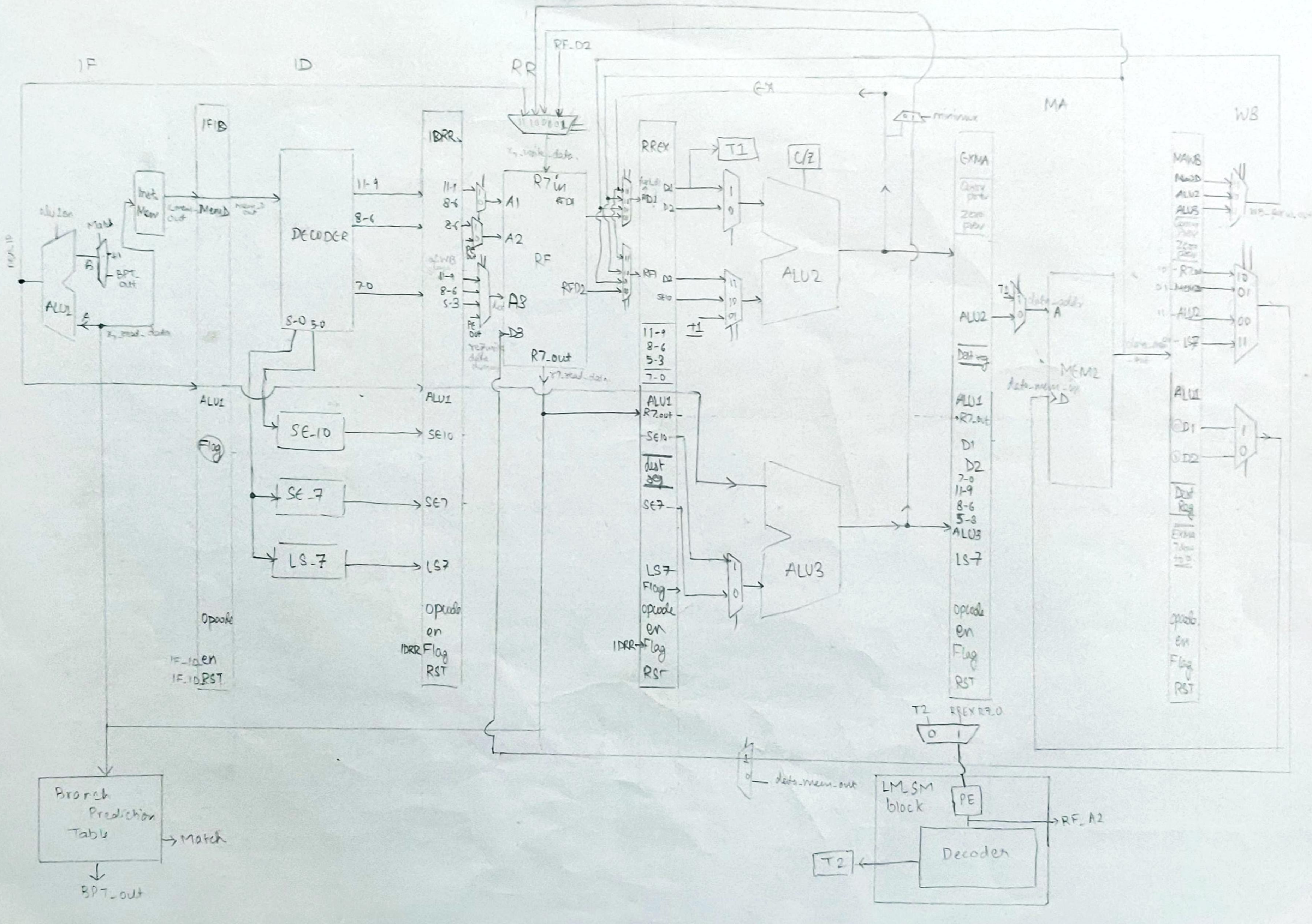


```
R7 → ALU1_A, MEM1_A
  +1 → ALU1_B
   ALU1 → R7
  MEM1_D11-9 → RF_A1
 RF_D1 → T1 , MEM1_D7-0 → T2
 while (T2 != 0)
  T2 → LMSM Block
  P_out → RF_A3
   T1 → MEM2_A, ALU4_A
   MEM2_D → RF_D3
      +1 → ALU2_B
      ALU2 → T1
     LMSM_out → T2
```

This is for LOAD MULTIPLE

In case of STORE MULTIPLE

```
P_out → RF_A2
RF_D2 → MEM2_D
```



IF
ID
RR
EX
MA
WB
ALU1
A
B
+1
BPT_out
Inst Mem
IFID
Mem_D
ALU1
Flag
Opcode
IF_ID en
IF_ID RST
DECODER
11-9
8-6
7-0
8-0
5-0
SE_10
SE_7
LS_7
IDRR
11-9
8-6
8-6
11-9
8-6
5-3
ALU1
SE10
SE7
LS7
opcode
en
IDRR Flag
RST
A1
A2
A3
D3
RF
R7 in
RD1
RD2
R7_out
RF_D2
RREX
D1
D2
11-9
8-6
5-3
7-0
ALU1
R7_out
SE10
dest reg
SE7
LS7
Flag
opcode
en
IDRR Flag
RST
T1
C/Z
ALU2
ALU3
+1
EXMA
Carry prev
Zero prev
ALU2
Dest reg
ALU1
R7_out
D1
D2
7-0
11-9
8-6
5-3
ALU3
LS7
opcode
en
Flag
RST
data_addr
A
D
MEM2
data_mem_out
MAWB
Mem_D
ALU2
ALU3
R7_out
Mem_D
ALU2
LS7
ALU1
D1
D2
Dest Reg
opcode
en
Flag
RST
T2
RREX R7_0
data_mem_out
LM_SM block
PE
Decoder
RF_A2
T2
Branch Prediction Table
Match
BPT_out