Low-Level Analysis and Implementation of a Pipelined Architecture

João Cardoso, 84096, MEEC

Abstract—**The motivation for this project is to analyze the internal operation of single-cycle and pipelined processors. We start by analyzing the control signals and explaining the operations that the processor performs. We go on to program the decode stage of the processor. Then, we consider the pipelined version, identifying sources for hazards, and making changes to the architecture in order to solve these hazards. Finally, we analyze the impact of these changes to the performance of the processor.**

Index Terms—**Processor, microarchitecture, RISC, VHDL.**

I. Introduction

THIS work consists in the development of a 32-bit RISC processor with a 5-stage pipeline (IF - Instruction Fetch, ID - Instruction Decode, EX - Execute, MEM - Memory and WB - Write Back), with 32-bit instructions and 16 32-bit general use registers (with register R0 stuck at value 0) using VHDL.

A simplified structure of the processor is shown in Figure 1, in its single-cycle form.

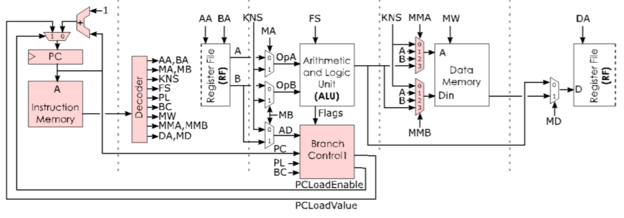


Fig. 1. Simplified diagram of the processor.

II. Datapath

The Datapath is divided into 5 sections:

A. Instruction Fetch

This stage controls the Instructions that are loaded into the Processor. The Program Counter (PC) is calculated and then used as the address to fetch an Instruction from Memory which in turn is passed to the next stage.

B. Instruction Decode

The input is decoded into several control signals that will control the other various stages and functional blocks, the meaning of each input is as follows:

AA, BA - Address of registers for operands A and B.

MA, MB - For the operands A and B of the ALU, choose whether the value comes from the Register File or from the Immediate (Constant).

KNS - Immediate Value extracted from the 32-bit Instruction Word. In this particular RISC ISA, It can range from 14,16 or 18 bits which then is extended to 32 bits (Extension Policy depends on Instruction).

KNSsel - bits that signal how to get the pieces of the Immediate from the Instruction word and how to extend it to 32 bits.

FS - bits that control the operation performed by the ALU.

PL - In the original architecture, it's 1 bit and it's used to signal the Branch Control if it's a Branch or not. Adapted to 2 bits to signal if it's a Branch,Jump or Not Branch.

BC - Branch Condition bits. It signals to the Branch Control which Flags to look to check if the condition is true, or false.

MW - Memory Write. If '1', Writes the specified Data to a specified Address. In this specific ISA, There's only 1 Store Instruction, so the Data IN will be the B operand and the Address comes from the ALU (Address calculation as it can be a sum of a Constant and a Register value).

MMA,MMB - Choses the operands for the Address and the Data to be stored.

DA - Destination Address. The register which the Data will be written on.

MD - Chooses the Data that will be written in the Destination Register.

C. Execute

This stage performs the instructions that require the ALU. The ALU flags (Z,N,C,O) are sent to the Branch Control Unit to check the conditions if it's a branch instruction. The Operations that the ALU performs depend on the value of FS, which is presented in the following table.

FS	ALU
0000	A + B
0001	A + B + 1
0010	A - B - 1
0011	A - B
0100	A and B
0101	A nand B
0110	A or B
0111	A nor B
1000	A xor B
1001	A xnor B
1010	Shift Left Logic B
1011	Shift Right Logic B
1100	Shift Left Arithmetic B
1101	Shift Right Arithmetic B
1110	Rotate Left B
1111	Rotate Right B

TABLE I
Instruction Decode Signals

D. Memory

This stage performs the memory operations, that is LD,LDI and ST.

E. Write-Back

This stage selects which Data to be written in the Register File and saves said Data.

III. Unsupported Instructions

After inspecting the table of instructions and looking at the μ-architecture, we can see that the Branch/Jump and Link instructions aren't implemented (BL,BIL,JL,JIL). Furthermore, in the VHDL code, it lacks the Decode Table and the Branch Control Logic.

To make it compliant with every instruction, Branch Control needs 2 new Outputs, PC Write Enable and PC+1. The first signal is 1 bit and if it's '1', Writing on registers is enabled, if '0', then disabled. PC+1 is then put in the MUX with MEM Data and ALU Data. To support this, MD is extended to 2 bits to chose one output for the 3 inputs.

Another change to the μ-architecture was adding 1 bit to PL so it can distinguish Branches from Jumps as it was mentioned in the previous section.

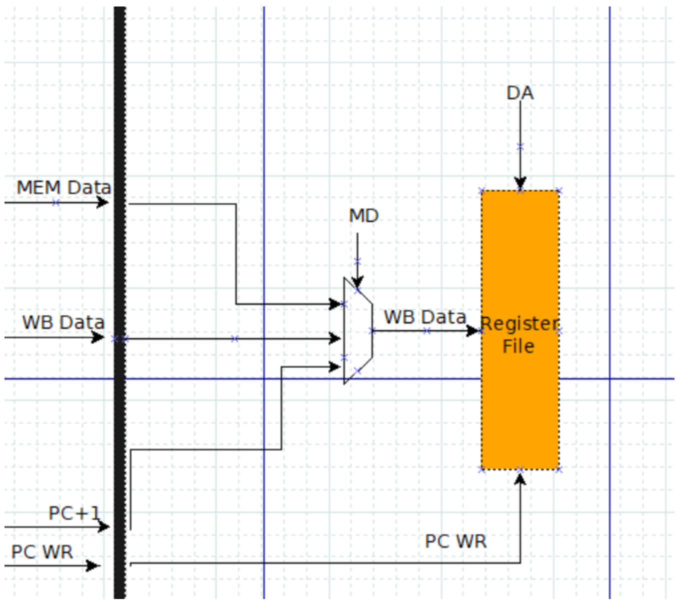


Fig. 2. Adaptation needed for the micro-architecture

IV. Programming the Decoder

For the correct implementation of all instructions, the decoder must be programmed accordingly. The table can be found in Appendix A.

Considerations taken:

• **dAA,dBA,dDA :** These signals chose the Register when the instruction doesn't specify it. The first 2 are always x"0" so when forwarding is implemented, all the instructions without R[SA] or R[SB] are put as R[0]. The Destination Register constant is always x"0" except in Branches and Jumps where it's R(15).

• **PL :** PL(1) means it's a branch or not and PL(0) states the type (Branch or Jump), so it's always "00" except for Branches which is "10" and in jumps it's "11".

• **MMA,MMB :** Both are defaulted to 0 except in Load and Store instructions in which it selects the correct inputs.

• **MW :** Default value is '0' unless it's a store.

• **FS :** It's always x"3" in Branch/Jump instructions to perform the subtraction which generates the flags.

V. Pipelined Execution

As Single Cycle goes, the work done in the previous sections is enough and it holds up after simulating some Assembly Code but to increase performance, pipelined execution is needed. That means the processor can hold 5 instructions in parallel and the operating frequency can be increased further.

The drawback is that there's a need for some changes in the /mu-architecture to support pipelining due to Hazards. There can be 3 types of Hazards:

• **Data Hazards -** Hazards related to the Writing or Reading of Data. Read after Write is the specific Data hazard tackled in this project. It occurs when the processor tries to read from a register which hasn't written the result to yet. To fix this, Forwarding is needed.

	1	2	3	4	5	6	7	8	9
ADDI R1,R0,300	IF	ID	EX	MEM	WB				
ADDI R2,R1,100		IF	ID	EX	MEM	WB			
ST 0(R1),R2			IF	ID	EX	MEM	WB		
ADDI R1,R1,R2				IF	ID	EX	MEM	WB	
JI R1,0					IF	ID	EX	MEM	WB

Fig. 3. Example of Assembly code with several Data Hazards

In Figure 3, there's a Read after Write Hazard in every instruction except the first one. In the 2nd instruction, R1 needs to be forwarded from MEM to the EX stage. In the Store instruction, R1 needs to be forwarded to EX and R2 needs to be forwarded from WB to the MEM stage.In the last instruction, there's a need to forward R1 from MEM to the EX stage, but instead of the ALU, it's to the branch Control. The last possible Read after Write Hazard in this processor is the Loads which will be treated differently than the others hazard.

• **Control Hazards -** Hazards related to the Branch execution. The Branch validation is done in the EXE stage, so when the validation is made, 2 instructions are in the pipeline already (IF and ID). In Figure 4, the 2 following ADDI's after the Branch are executed and that's the Control Hazard as if the condition is correct, it should jump to the store instruction.

	1	2	3	4	5	6	7	8	9	10
ADDI R1,R0,300	IF	ID	EX	MEM	WB					
SUB R2,R2,R1		IF	ID	EX	MEM	WB				
B.GE R2,R0,3			IF	ID	EX	MEM	WB			
ADDI R2,R1,R2				IF	ID	EX	MEM	WB		
ADDI R1,R0,100					IF	ID	EX	MEM	WB	
ST 0(R2) , R1						IF	ID	EX	MEM	WB

Fig. 4. Example of Assembly code with a Control Hazard

Figure 7.

(when AA/BA /= ’0’) and MW=’0’	MA(2:0)	MB(2:0) (when PL(1)=’0’)	Branch Forwarding(2:0) (when PL(1)=’1’)
ID_AA=EX_DA	10	’0’ & MBSel(0)	’0’ & MBSel(0)
ID_AA=MEM_DA	11	’0’ & MBSel(0)	’0’ & MBSel(0)
ID_BA=EX_DA	’0’ & MASel(0)	10	10
ID_BA=MEM_DA	’0’ & MASel(0)	11	11
MW=’1’			
ID_BA=EX_DA	0 & MASel(0)	’0’ & MBSel(0)	–
ID_BA=MEM_DA	0 & MASel(0)	’0’ & MBSel(0)	–

TABLE II
MODIFYING SIGNALS TO ACCOMMODATE FORWARDING

• **Structural Hazards -** Hazards related to the μ- architecture’s lack of resources, that is, 2 instructions need to use the same hardware at the same time.As this project uses a simple in-order 5 stage pipeline, this type of hazard is not present.

A. Solving the Hazards

To solve the previously mentioned hazards, several logic has to be implemented.

• **Data Hazards- ALU and Branch Control** In the ID stage, EX DA, MEM DA and EX MD are used to check if there’s a need to forward any Data to the EX. The Table II has the changes to the MA,MB and a new Signal Branch Forwarding that controls the AD input of the Branch Control Unit. MA is unaffected by MW and PL unlike MB .

The Write Back is done on the negative edge to avoid additional Logic. Frequency isn’t affected as the propagation time of the WB isn’t comparable to the other stages.

• **Data Hazards- MEM (Store)** For Store Instructions, the value that will be written in the Data Memory might not be available for other instructions to use, so Forwarding is needed to the EX/MEM stage. In the following figure it explains the Forwarding block that will become the Data IN of the Data Memory.

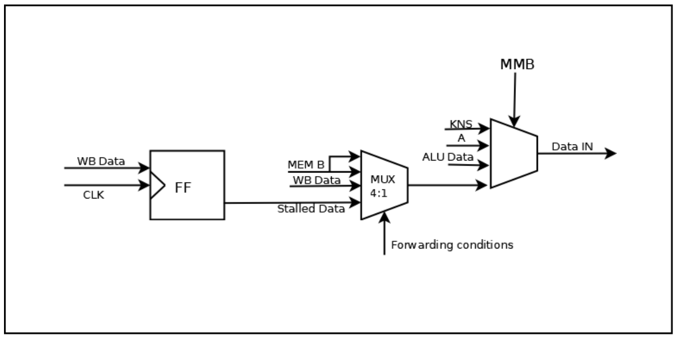


Fig. 5. Forwarding Block for Store Instructions

• **Data Hazards- MEM (Loads)** Forwarding for Load’s brings a much higher propagation time as tp=t(EX)+t(MEM) so performance would decrease instead of increasing.To avoid this hazard,Stalls are introduced so the Data can pass the pipeline to the Write Back Stage. This is done in the ID stage as it checks if the Registers to be Read are the same that the Load will write on and that EX MD="01", if true, it stalls the pipeline for 1 clock cycle. For the pipeline to not stall indefinitely,The PC’s of the different stages are compared, if they are equal, it stops the stalling. The Logic added is in

• **Control Hazards** To fix this hazard type, Branch Not taken policy was used, that is, While the Branch Control hasn’t checked the flags, it will load the PC+1 and PC+2 instruction to the pipeline, when the condition is checked, if it’s true, the IF-ID pipeline is reset to x"00000000" (NOP) and in the ID-EX pipeline MW,PC WR and PL are reset so no data is written. If the branch isnt taken, there’s no performance penalty.

In the Figure 8 , An example assembly code that was run by the processor. The Forwarding is highlighted. A few remarks:

• **ADDI R1,R1,R2 in Clock Cycle 5** - No need to Forward Data as the WB is in the negative Edge.

• **ADD R6,R4,R5 in Clock Cycle 11** - Introduced Stall due to Data not being avaiable to be forwarded yet by the LDI.

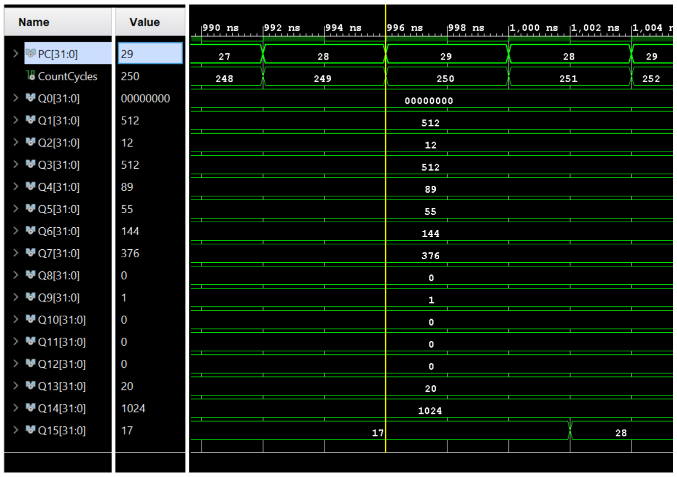
• **B.NEQ in Clock Cycle 16** - Deleted the 2 prior Instructions as it was a miss prediction.

VI. PERFORMANCE

The objective of Pipelining is to re-use existing Hardware resources to accelerate the throughput of the CPU. From the Assembly program used in the Lab to validate the CPU, we get that the SC version, did it in 163 clock cycles, while the Pipelined version completed it in 250 clock cycles (Figure 6).

The frequency for the Pipelined Version is around 3 times as fast.

$$Speedup = \frac{163/f_{SC}}{251/(3*f_{SC})} = \frac{163}{83.66} = 1.9480 \qquad (1)$$



Name	Value
PC[31:0]	29
CountCycles	250
Q0[31:0]	00000000
Q1[31:0]	512
Q2[31:0]	12
Q3[31:0]	512
Q4[31:0]	89
Q5[31:0]	55
Q6[31:0]	144
Q7[31:0]	376
Q8[31:0]	0
Q9[31:0]	1
Q10[31:0]	0
Q11[31:0]	0
Q12[31:0]	0
Q13[31:0]	20
Q14[31:0]	1024
Q15[31:0]	17

Fig. 6. Pipelined Simulated Version

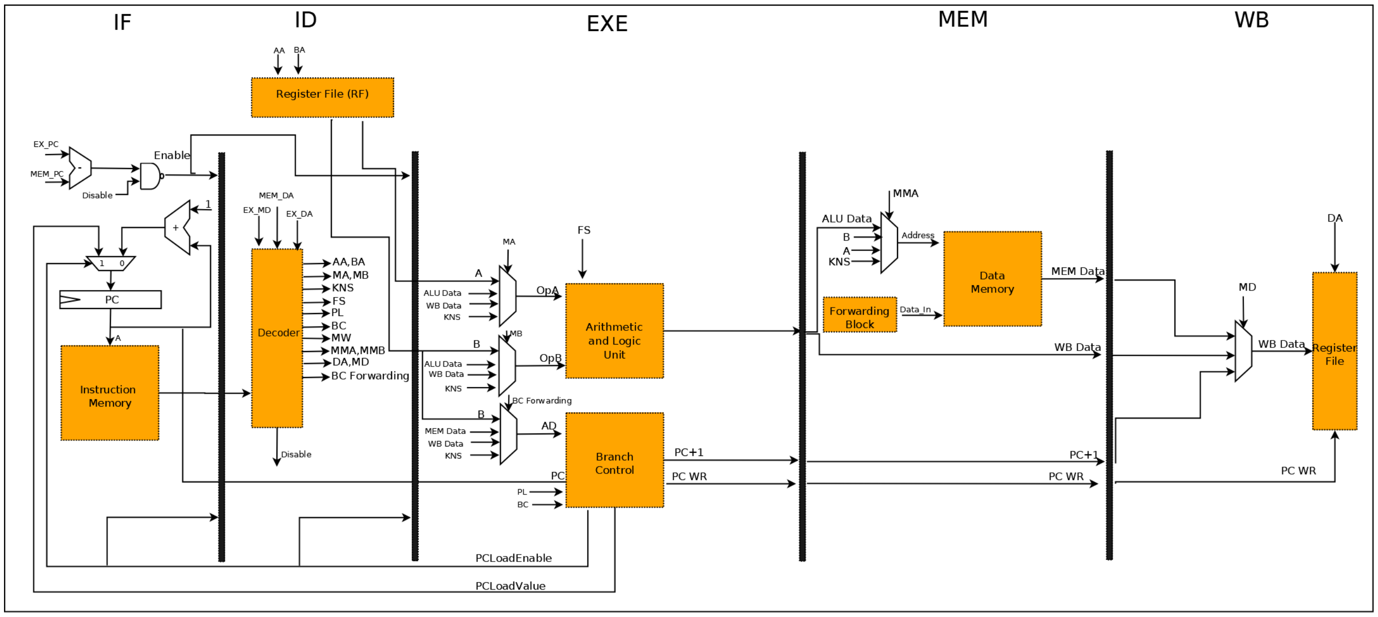
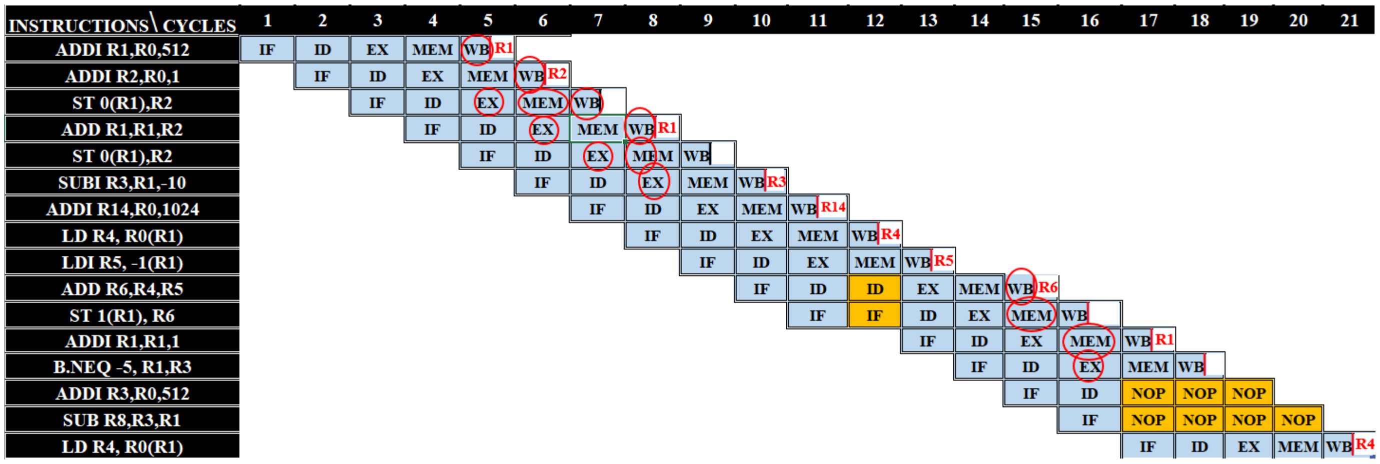


Fig. 7. Pipelined Version of the processor



INSTRUCTIONS\ CYCLES	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
ADDI R1,R0,512	IF	ID	EX	MEM	WB R1																
ADDI R2,R0,1		IF	ID	EX	MEM	WB R2															
ST 0(R1),R2			IF	ID	EX	MEM	WB														
ADD R1,R1,R2				IF	ID	EX	MEM	WB R1													
ST 0(R1),R2					IF	ID	EX	MEM	WB												
SUBI R3,R1,-10						IF	ID	EX	MEM	WB R3											
ADDI R14,R0,1024							IF	ID	EX	MEM	WB R14										
LD R4, R0(R1)								IF	ID	EX	MEM	WB R4									
LDI R5, -1(R1)									IF	ID	EX	MEM	WB R5								
ADD R6,R4,R5										IF	ID	ID	EX	MEM	WB R6						
ST 1(R1), R6											IF	IF	ID	EX	MEM	WB					
ADDI R1,R1,1													IF	ID	EX	MEM	WB R1				
B.NEQ -5, R1,R3														IF	ID	EX	MEM	WB			
ADDI R3,R0,512															IF	ID	NOP	NOP	NOP		
SUB R8,R3,R1																IF	NOP	NOP	NOP	NOP	
LD R4, R0(R1)																	IF	ID	EX	MEM	WB R4

Fig. 8. Example Assembly Code with All Hazards taken care of

The Speedup in performance is significant with resources used being mainly the Pipeline Registers.

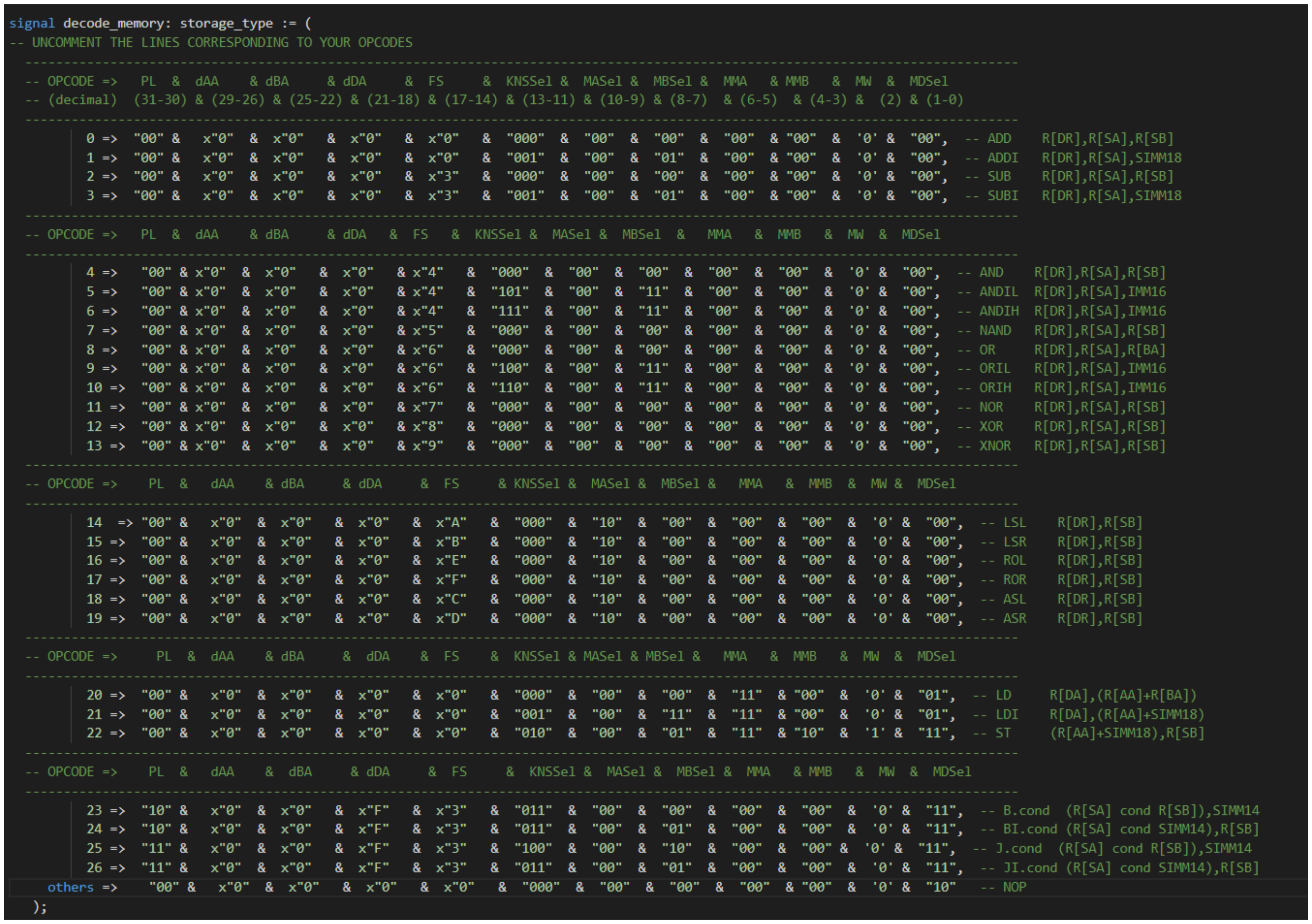
Performance for a "Simple" CPU, this will be as good as it gets, to further increase performance, other "tricks"would be needed like Issuing several instructions per clock cycle and having several and independent execution ports. Of course, those improvements would bring several Structural Hazards and other Data Hazards like Write after Write.

Improving Branch prediction will also improve performance, but with this shallow pipeline, it would be waste of resources (performance per area return would be low for this μ-architecture).

VII. Conclusion

In this project, the objective was to learn and to explore a simple Pipelined RISC CPU. It's fundamental to understand these concepts because most if not all CPU's designed today are RISC(Intel and AMD's μ-architectures are RISC based using a CISC ISA) and have the same concept as this simple CPU. Today, one of the most used μ-architectures, the ARM A53 (From IoT to low powered core in today's System on Chips) uses a similar in-order design but obviously much deeper pipeline and 2 execution ports. But that doesn't mean the resemblance isn't there .So this project helps us grasp the design difficulties of today's CPU.

Appendix A
Instruction Decoder Table



```
signal decode_memory: storage_type := (
-- UNCOMMENT THE LINES CORRESPONDING TO YOUR OPCODES
  ------------------------------------------------------------------------------------------------------------------
  -- OPCODE =>   PL  &  dAA    & dBA     & dDA     &  FS     &  KNSSel &  MASel &  MBSel &  MMA   & MMB   &  MW  &  MDSel
  -- (decimal)  (31-30) & (29-26) & (25-22) & (21-18) & (17-14) & (13-11) & (10-9) & (8-7)  & (6-5)  & (4-3) &  (2) & (1-0)
  ------------------------------------------------------------------------------------------------------------------
        0 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"0"   &  "000"  &  "00"  &  "00"  &  "00"  & "00"  &  '0' &  "00",  -- ADD    R[DR],R[SA],R[SB]
        1 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"0"   &  "001"  &  "00"  &  "01"  &  "00"  & "00"  &  '0' &  "00",  -- ADDI   R[DR],R[SA],SIMM18
        2 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"3"   &  "000"  &  "00"  &  "00"  &  "00"  & "00"  &  '0' &  "00",  -- SUB    R[DR],R[SA],R[SB]
        3 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"3"   &  "001"  &  "00"  &  "01"  &  "00"  & "00"  &  '0' &  "00",  -- SUBI   R[DR],R[SA],SIMM18
  ------------------------------------------------------------------------------------------------------------------
  -- OPCODE =>   PL  &  dAA    & dBA     & dDA   &  FS   &  KNSSel &  MASel &  MBSel  &   MMA   &  MMB   &  MW  &  MDSel
  ------------------------------------------------------------------------------------------------------------------
        4 =>   "00" & x"0"  &  x"0"   &  x"0"   & x"4"   &  "000"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- AND    R[DR],R[SA],R[SB]
        5 =>   "00" & x"0"  &  x"0"   &  x"0"   & x"4"   &  "101"  &  "00"  &  "11"  &  "00"  &  "00"  &  '0' &  "00",  -- ANDIL  R[DR],R[SA],IMM16
        6 =>   "00" & x"0"  &  x"0"   &  x"0"   & x"4"   &  "111"  &  "00"  &  "11"  &  "00"  &  "00"  &  '0' &  "00",  -- ANDIH  R[DR],R[SA],IMM16
        7 =>   "00" & x"0"  &  x"0"   &  x"0"   & x"5"   &  "000"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- NAND   R[DR],R[SA],R[SB]
        8 =>   "00" & x"0"  &  x"0"   &  x"0"   & x"6"   &  "000"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- OR     R[DR],R[SA],R[BA]
        9 =>   "00" & x"0"  &  x"0"   &  x"0"   & x"6"   &  "100"  &  "00"  &  "11"  &  "00"  &  "00"  &  '0' &  "00",  -- ORIL   R[DR],R[SA],IMM16
        10 =>  "00" & x"0"  &  x"0"   &  x"0"   & x"6"   &  "110"  &  "00"  &  "11"  &  "00"  &  "00"  &  '0' &  "00",  -- ORIH   R[DR],R[SA],IMM16
        11 =>  "00" & x"0"  &  x"0"   &  x"0"   & x"7"   &  "000"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- NOR    R[DR],R[SA],R[SB]
        12 =>  "00" & x"0"  &  x"0"   &  x"0"   & x"8"   &  "000"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- XOR    R[DR],R[SA],R[SB]
        13 =>  "00" & x"0"  &  x"0"   &  x"0"   & x"9"   &  "000"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- XNOR   R[DR],R[SA],R[SB]
  ------------------------------------------------------------------------------------------------------------------
  -- OPCODE =>    PL  &   dAA    & dBA     & dDA     &  FS     & KNSSel &  MASel &  MBSel &   MMA   &  MMB  &  MW &  MDSel
  ------------------------------------------------------------------------------------------------------------------
        14  => "00" &   x"0"  &  x"0"   &  x"0"   &  x"A"   &  "000"  &  "10"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- LSL    R[DR],R[SB]
        15 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"B"   &  "000"  &  "10"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- LSR    R[DR],R[SB]
        16 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"E"   &  "000"  &  "10"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- ROL    R[DR],R[SB]
        17 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"F"   &  "000"  &  "10"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- ROR    R[DR],R[SB]
        18 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"C"   &  "000"  &  "10"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- ASL    R[DR],R[SB]
        19 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"D"   &  "000"  &  "10"  &  "00"  &  "00"  &  "00"  &  '0' &  "00",  -- ASR    R[DR],R[SB]
  ------------------------------------------------------------------------------------------------------------------
  -- OPCODE =>     PL  &  dAA    & dBA     &  dDA    &  FS    &  KNSSel & MASel & MBSel &   MMA   &  MMB   &  MW  &  MDSel
  ------------------------------------------------------------------------------------------------------------------
        20 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"0"   &  "000"  &  "00"  &  "00"  &  "11"  & "00"  &  '0' &  "01",  -- LD     R[DA],(R[AA]+R[BA])
        21 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"0"   &  "001"  &  "00"  &  "11"  &  "11"  & "00"  &  '0' &  "01",  -- LDI    R[DA],(R[AA]+SIMM18)
        22 =>  "00" &   x"0"  &  x"0"   &  x"0"   &  x"0"   &  "010"  &  "00"  &  "01"  &  "11"  & "10"  &  '1' &  "11",  -- ST     (R[AA]+SIMM18),R[SB]
  ------------------------------------------------------------------------------------------------------------------
  -- OPCODE =>    PL  &   dAA    &  dBA     & dDA     &  FS     &  KNSSel &  MASel &  MBSel &  MMA   & MMB   &  MW  &  MDSel
  ------------------------------------------------------------------------------------------------------------------
        23 =>  "10" &   x"0"  &  x"0"   &  x"F"   &  x"3"   &  "011"  &  "00"  &  "00"  &  "00"  &  "00"  &  '0' &  "11",  -- B.cond  (R[SA] cond R[SB]),SIMM14
        24 =>  "10" &   x"0"  &  x"0"   &  x"F"   &  x"3"   &  "011"  &  "00"  &  "01"  &  "00"  &  "00"  &  '0' &  "11",  -- BI.cond (R[SA] cond SIMM14),R[SB]
        25 =>  "11" &   x"0"  &  x"0"   &  x"F"   &  x"3"   &  "100"  &  "00"  &  "10"  &  "00"  &  "00" &  '0' &  "11",  -- J.cond  (R[SA] cond R[SB]),SIMM14
        26 =>  "11" &   x"0"  &  x"0"   &  x"F"   &  x"3"   &  "011"  &  "00"  &  "01"  &  "00"  &  "00"  &  '0' &  "11",  -- JI.cond (R[SA] cond SIMM14),R[SB]
    others =>   "00" &   x"0"  &  x"0"   &  x"0"   &  x"0"   &  "000"  &  "00"  &  "00"  &  "00"  & "00"  &  '0' &  "10"   -- NOP
  );
```

Fig. 9. Instruction Decode Table

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.