EE 275

MINI PROJECT - 2

Last Name: Bhutiani

First Name: Akshat

ID: 014538238

Last Name: Nigam

First Name: Amiraj

ID: 01451202

Last Name: Sahu

First Name: Soumya

ID: 014537666

Date: 04/23/2020

Project video link :https://drive.google.com/file/d/1OPk64jlnUf7xD_CiCc2uSaNvPT25ooWZ/view

MIPS ISA DOT PRODUCT IMPLEMENTATION WITH FORWARD CHAINING

Introduction

The aim of this project is to design a Python, Verilog, C/C++ code to implement forward chaining in the design. This model will showcase how using the Forward Chaining methodology could enhance the effectiveness of the design. This technique utilizes the potential of the hardware Architecture to its maximum to give optimum speed and better time management as an output. A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

The MIPS architecture often faces various hazards: Structural Hazard, Data Hazard, Control Hazards. These hazards may introduce a stall in the architectural pipeline. Stalls could be due to cache miss or hazards in the pipeline. These stalls are good to avoid a hazard but do not serve the purpose of pipelining which is to increase the throughput of the design. Operand forwarding is a technique to get optimized design by controlling the hazards and preventing stalls in the different cycles of pipelining. The idea behind forwarding is simple: if a register $r has not yet been updated with its ”new value” by the time a later instruction needs the value of $r, the data can be forwarded if it is available at or before the time it is needed. This involves extra hardware in the form of wires, hazard-detection, additional MUXes, etc.

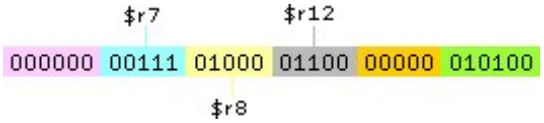
This paper provides a simple example of enabling forward chaining in the structure to prevent pipelining hazards and stalls in different clock cycles/stages. We recognize this by analyzing a simple code in python that functions without forward chaining then we will modify this code to run with faster speed by introducing forward chaining to it. This new process will be gauged by watching the CPU performance during run-time and execution of both the design codes.

MIPS Architecture

INSTRUCTION SET ARCHITECTURE

The MIPS instruction set consists of about 111 total instructions, each represented in 32 bits. An example of a MIPS instruction is below:

add $r12, $r7, $r8



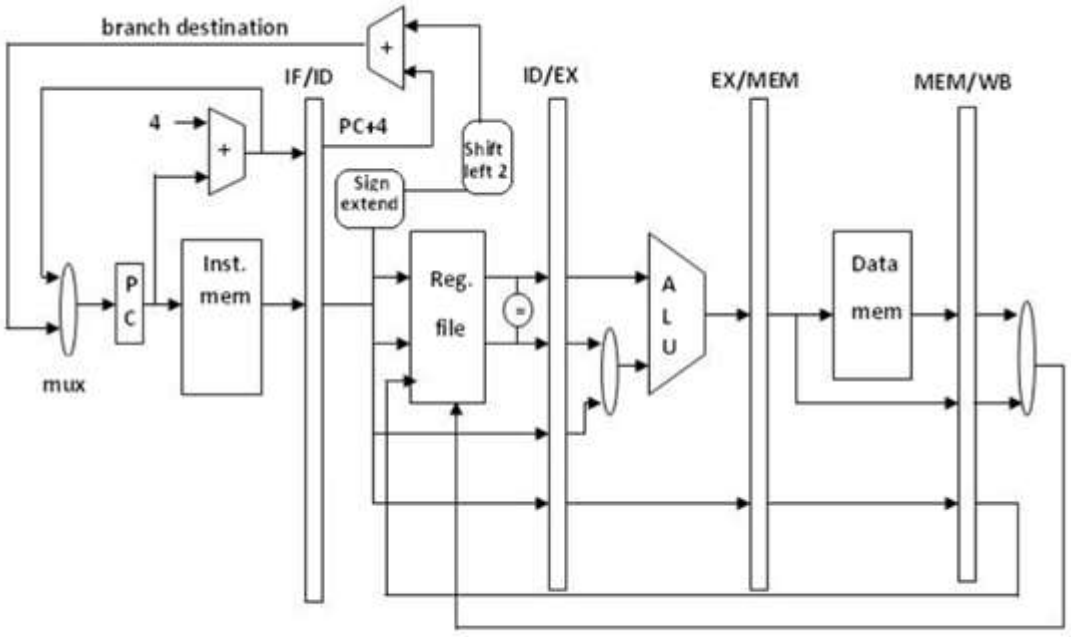
Above is the assembly (left) and binary (right) representation of a MIPS addition instruction. The instruction tells the processor to compute the sum of the values in registers 7 and 8 and store the result in register 12. The dollar signs are used to indicate operation on a register. The colored binary representation on the right illustrates the 6 fields of a MIPS instruction. The processor identifies the type of instruction by the binary digits in the first and last fields. In this case, the processor recognizes that this instruction is an addition from the zero in its first field and the 20 in its last field.
The operands are represented in the blue and yellow fields, and the desired result location is presented in the fourth (purple) field. The orange field represents the shift amount, something that is not used in an addition operation.

Types of MIPS Instruction –

Type of Instruction	**Format (in Bits)**					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

R Type instruction means Register type instruction. It is used to move data in between the registers.
I type instruction means Immediate type instruction. It is used to move numbers directly into the registers.
For exmaple - $ addi r29, #4
This adds the number 4 to the contents of register 29. After this instruction, r29 = r29 + 4.

There are five stages in MIPS pipeline:
IF: Instruction fetch from the MEMORY
ID: Instruction Decode and register READ
Ex: Execute operation or calculate address
MEM: Access Memory operand
WB: Write result back to the register



The MIPS architecture.

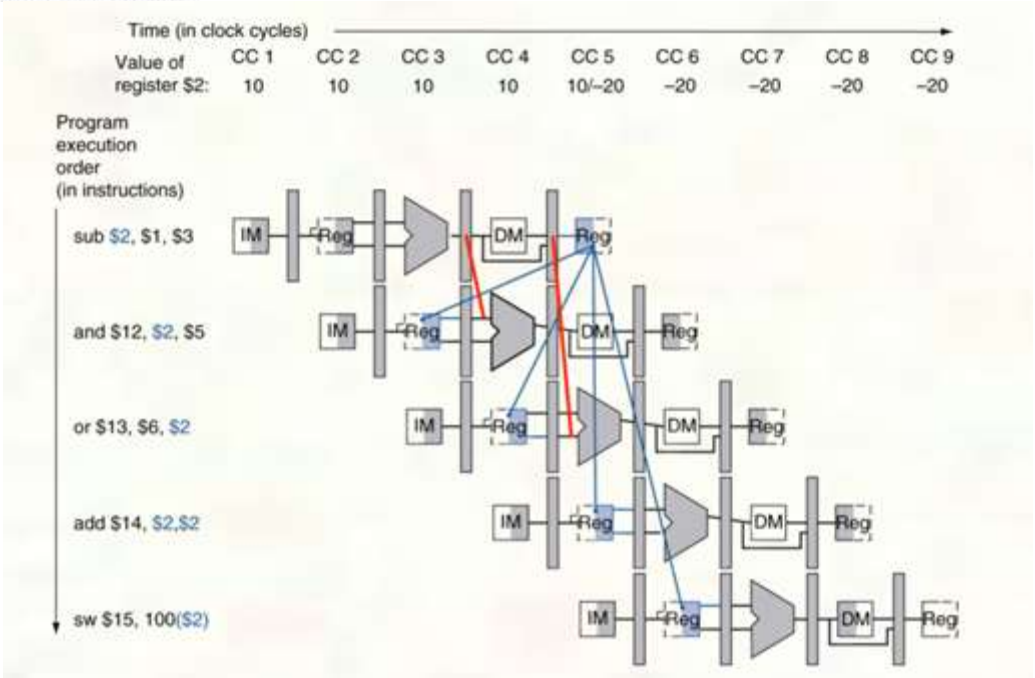


Figure 1. Forward Chaining Logic flow

Forward Chaining

Without forwarding our example will execute correctly with stalls:

	1	2	3	4	5	6	7	8	9
ADD **R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB R4, R5, **R1**		IF	**stall**	**stall**	**ID**$_{sub}$	EX	MEM	WB	
AND R6, **R1,** R7			**stall**	**stall**	IF	**ID**$_{and}$	EX	MEM	WB

Using those forwarding paths the code sequence can be executed without stall

	1	2	3	4	5	6	7
ADD **R1**, R2, R3	IF	ID	**EX**$_{add}$	**MEM**$_{add}$	WB		
SUB R4, R5, **R1**		IF	ID	**EX**$_{sub}$	MEM	WB	
AND R6, **R1,** R7			IF	ID	**EX**$_{and}$	MEM	WB

The first forwarding is for the value of **R1** from **EX**$_{add}$ to **EX**$_{sub}$.

The second forwarding is also for the value of **R1** from **MEM**$_{add}$ to **EX**$_{and}$.

This code now can be executed without stalls.

Forwarding can be generalized to include passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Implementation & Analysis

Analysis:

1)The original code is written without considering forward chaining, There is a data hazard RAW(Read after write) observed in the 5th (mul $R2 $R2 $R4)and the 3rd (LW $R2 0($R3))instruction which is due to the dependency of the contents of the register R2 , the contents of R2 register are available only after the memory write operation whereas the 5th instruction requires it in its decode stage in order to perform the multiplication operation.

Program

INDEX	INSTRUCTIONS
1	addu $r1 $r0 $r0 ; result = 0
2	LOOP: beq $r7 $r0 done
3	lw $r2 0($r3)
4	lw $r4 0($r5)
5	mul $r2 $r2 $r4
6	addu $r1 $r1 $r2
7	addiu $r3 $r3 #4
8	addiu $r5 $r5 #4
9	addiu $r7 $r7 #-1
10	j loop
11	DONE: jr $r31

Implementation:

This data hazard can be solved by using the value of $r2 and applying it directly to the $r5 instead of writing it back.

According to next analysis the stall occurs between the instructions 5th (MUL $R2 $R2 $R4) and 6th (ADDU $R1 $R1 $R2), this is the RAW(Read after Write hazard) as the result of the multiplication in the 5th instruction will only be available till the write-back is performed on the register R2, but we need it right in the decode stage for the execution of the 6th instruction.

Implementation:

The result of multiplication of the registers R2 and R1 is available after the execution/ALU stage of the 5th instruction and thus it can directly be written during the rising edge of the clock cycle of the execution stage and read by the 6th instruction in the falling edge of the same clock cycle.

The stall no 3 is occurring due to the data dependencies for R7 between the two instructions addiu $r7 $r7 #-1 and beq $r7 $r0 Done .There is a RAW data hazard observed here as the second instruction branch if R7 equal to R0 requires the value of R7 register in its decode stage whereas the data is available only on the writeback stage of the add unsigned instruction.

Implementation:The data dependencies here can be resolved by data forwarding form the ALU/Memory read buffer of the - addiu $r7 $r7 #-1 to the ALU stage of the next instruction

beq $r7 $r0 Done

Tabular Representation of the instructions in Cycle:

Table 1. Without Forward Chaining

INSTRUCTION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	21	22	23
addu $r1 $r0 $r0 ; result = 0	F	D	E	M	W																			
LOOP																								
beq $r7 $r0 done		F	D	E	M	W																		
lw $r2 0($r3)			F	D	E	M	W																	
lw $r4 0($r5)				F	D	E	M	W																
mul $r2 $r2 $r4					F	.	.	.	D	E	M	W												
addu $r1 $r1 $r2									F	.	.	.	D	E	M	W								
addiu $r3 $r3 #4													F	D	E	M	W							
addiu $r5 $r5 #4														F	D	E	M	W						
addiu $r7 $r7 #-1															F	D	E	M	W					
j loop																F	D	E	M	W				
DONE																								
jr $r31																								
	N	E	X	T		I	T	E	R	A	T	I	O	N		S	T	A	R	T				
beq $r7 $r0 done																	F	.	.	D	E	M	W	
lw $r2 0($r3)																								
lw $r4 0($r5)																								

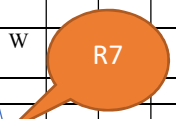
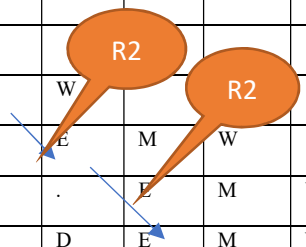


Table 2. With Data Forwarding

INSTRUCTIONS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
addu $r1 $r0 $r0 ; result = 0	F	D	E	M	W											
LOOP																
beq $r7 $r0 done		F	D	E	M	W										
lw $r2 0($r3)			F	D	E	M	W									
lw $r4 0($r5)				F	D	E	M	W								
mul $r2 $r2 $r4					F	D	.	E	M	W						
addu $r1 $r1 $r2						F	D	.	E	M	W					
addiu $r3 $r3 #4							F	D	E	M	W					
addiu $r5 $r5 #4								F	D	E	M	W				
addiu $r7 $r7 #-1									F	D	E	M	W			
j loop										F	D	E	M	W		
DONE																
jr $r31																
		N	E	X	T		I	T	E	R	A	T	I	O	N	
beq $r7 $r0 done											F	D	E	M	W	
lw $r2 0($r3)																
lw $r4 0($r5)																



CODE & OUTPUT WITHOUT FORWARDING -

Code -

```
print("Mini Project 3 - MIPS ISA Dot Product Implementation without Forwarding")

id1_36 = "014538238"

id2_36 = "236750450"

r0_36 = 0

r1_36 = 0 #USED FOR STORING THE RESULT

r2_36 = [] #FIRST VECTOR

r4_36 = [] #SECOND VECTOR

r3_36 = 0*1 #NEXT ELEMENT POINTER IN R2

r5_36 = 0*1 #NEXT ELEMENT POINTER IN R4

r7_36 = int(len(id1_36)) #STORE THE LENGTH OF A VECTOR

stalls_36 = 0

for var in id1_36:

    r2_36.append(var) #STORE 1st VECTOR

for var in id2_36:

    r4_36.append(var) #STORE SECOND VECTOR

def done():

    global stalls_36

    print ("Results and Analysis")


   print ("Dot Product Without Forwarding")

    print ("Two vectors are", id1_36, "and", id2_36)

    stalls_36 = stalls_36 + 3 #3 INSTRUCTION DELAY

    print ("Dot Product Result=", r1_36, "stalls", stalls_36)

    exit()

#START OF THE CODE
```

```
r1_36 = (r0_36 + r0_36) #ADDU $R1 $R0 $R0 ; result = 0


while(1):
  if r7_36 == r0_36: #BEQ $R7 $R0 DONE ; done looping
    done()
  else:
    if(r2_36): #Fetch  #LW $R2 0($R3) ; load a element
     True
     if(r2_36[r3_36])!="": #Decode LW $R2 0($R3) ; load a element
       if(r4_36):          #Fetch  lw $R4 0($R5) ; load b element
         True
    if len(r2_36)>=0:       #Execute lW $R2 0($R3) ; load a element
      if(r4_36[r5_36])!="": #Decode  lW $R4 0($R5) ; load b element
        if (r2_36):        #Fetch mul $R2 $R2 $R4 ;
         True
    if(r2_36[r3_36]):       #Memory  lW $R2 0($R3) ; load a element
      if len(r4_36) >= 0:   #Execute lW $R4 0($R5) ; load b element
        stalls_36 = stalls_36 +1 #
        pass                #NOP due to STALL
    if(r2_36[r3_36]):       #Write  lW $R2 0($R3) ; load a element
      m_v_12 = float(r2_36[r3_36])
      m_v_12 = m_v_12 + 0
      if float(r4_36[r5_36])>=0: #Memory lW $R4 0($R5);load b element
        stalls_36 = stalls_36+1
        pass                  #NOP due to STALL
    if(r4_36[r5_36]): #Write  lW $R4 0($R5) ; load b element
```

```
  l_v_12 = float(r4_36[r5_36]
  l_v_12 = l_v_12 + 0
  stalls_36 = stalls_36 + 1
  pass                    #NOP due to STALL
if(r4_36[r5_36])!="": #Decode MUL $R2 $R2 $R4 ;
  if (r2_36):       #Fetch addu $R1 $R1 $R2
    True
if len(r4_36)>= 0:    #Execute #MUL $R2 $R2 $R4 ;
  stalls_36 = stalls_36+1
  pass              #NOP due to STALL
if float(r4_36[r5_36])>= 0: #Memory mul $r2 $r2 $r4 ;
  stalls_36 = stalls_36+1
  pass                  #NOP due to STALL
if (r4_36[r5_36]):   #Write  MUL $R2 $R2 $R4
  r2_36[r3_36] = float(r2_36[r3_36])*float(r4_36[r5_36])
  stalls_36 = stalls_36 +1
  pass              #NOP due to STALL
if (r4_36[r5_36])!="":    #Decode ADDU $r1 $r1 $R2
  if(r2_36[r3_36])!="":   #Execute ADDU $R1 $R1 $R2
    r1_36 = r1_36 + float(r2_36[r3_36]) #Memory addu $R1 $R1 $R2
    r1_36 = r1_36                    #Write addu $R1 $R1 $R2
r3_36 = r3_36 + 1 #Increment counter addiu $R3 $R3 #4
r5_36 = r5_36 + 1 #Increment counter addiu $R5 $R5 #4
#r7_36=r7_36- 1
if(r7_36):    #Fetch  beq $R7 $R0 done ; done looping?
   stalls_36 = stalls_36 + 1
```

```
        pass       #NOP due to STALL
    if(r7_36)!= "": #Decode beq Rr7 R0 done ; done looping?
        stalls_36 = stalls_36 + 1
        pass        #NOP due to STALL
    if len(str(r7_36))>=0: #Execute BEQ $R7 $R0 done ; done looping?
      if (r7_36 == r7_36):                                    print (r2_36[0:r3_36])
         r7_36 = r7_36 - 1 #Write and Decrement
counter addiu $R7 $R7 #-1
```

Output -

```
Mini Project 3 - MIPS ISA Dot Product Implementation without Forwarding
[0.0]
[0.0, 3.0]
[0.0, 3.0, 24.0]
[0.0, 3.0, 24.0, 35.0]
[0.0, 3.0, 24.0, 35.0, 15.0]
[0.0, 3.0, 24.0, 35.0, 15.0, 0.0]
[0.0, 3.0, 24.0, 35.0, 15.0, 0.0, 8.0]
[0.0, 3.0, 24.0, 35.0, 15.0, 0.0, 8.0, 15.0]
[0.0, 3.0, 24.0, 35.0, 15.0, 0.0, 8.0, 15.0, 0.0]
Results and Analysis
Dot Product Without Forwarding
Two vectors are 014538238 and 236750450
Dot Product Result= 100.0 stalls 75
```

CODE & OUTPUT WITH FORWARDING -

Code -

```
print("Mini Project 3 - MIPS ISA Dot Product Implementation with Forwarding")
ID1_36 = "014538238"
ID2_36 = "236750450"
R0_36=0
R1_36=0 #stores the sum
R2_36=[] #stores the fist vector
R4_36=[] #stores the second vector
R3_36=0*1 #point at next element in R2_36
R5_36=0*1 #point at next element in R4_36
R7_36=int(len(ID1_36)) #stores the length of vector which reduces after every cycle by one
ps_12=0
for var in ID1_36:
    R2_36.append(var) #1st vector stored
for var in ID2_36:
    R4_36.append(var) #2nd vector stored
def done():
    global ps_12
    print("RESULTS AND ANALYSIS")
    print("Dot Product Using Forwarding")
    print("Two vectors are", ID1_36,"and",ID2_36)
    ps_12 = ps_12 + 1 #1 stalls due to beq $r7 $r0 done ; done  looping? Control Hazard
    print("Dot Product Result=", R1_36, "stalls",ps_12)
```

```
   exit()
#code starts here
R1_36=(R0_36 + R0_36)                    #addu $r1 $r0 $r0 ; result = 0
while(1):
 if R7_36==R0_36:                        #beq $r7 $r0 done ; done looping?
   done()
 else:
   if(R2_36):                      #Fetch  lw $r2 0($r3) ; load a elem
      True
   if(R2_36[R3_36])!="":                    #Decode lw $r2 0($r3) ; load a elem
      if(R4_36):                   #Fetch  lw $r4 0($r5) ; load b elem
         True
   if len(R2_36)>=0:                  #Execute lw $r2 0($r3) ; load a elem
      if(R4_36[R5_36])!="":                 #Decode  lw $r4 0($r5) ; load b elem
         if(R2_36):                #Fetch mul $r2 $r2 $r4 ;
            True
   if(R2_36[R3_36]):                   #Memory  lw $r2 0($r3) ; load a elem
      if len(R4_36)>=0:                #Execute lw $r4 0($r5) ; load b elem
         if(R4_36[R5_36])!="":             #Decode mul $r2 $r2 $r4 ;
            if(R2_36):             #Fetch addu $r1 $r1 $r2
               True
   if(R2_36[R3_36]):                   #Write  lw $r2 0($r3) ; load a elem
      m_v_12 = float(R2_36[R3_36])
      m_v_12=m_v_12+0
      if float(R4_36[R5_36])>=0:            #Memory lw $r4 0($r5) ; load b elem
         ps_12=ps_12+1
```

```
        pass              #NOP due to STALL
if(R4_36[R5_36]):                    #Write  lw $r4 0($r5) ; load b elem
    l_v_12 = float(R4_36[R5_36])
    l_v_12=l_v_12+0
    if len(R4_36)>=0:                #Execute #mul $r2 $r2 $r4 ;
        if(R4_36[R5_36])!="":              #Decode addu $r1 $r1 $r2
         True
if float(R4_36[R5_36])>=0:                #Memory mul $r2 $r2 $r4 ;
        ps_12=ps_12+1
        pass              #NOP due to STALL
if(R4_36[R5_36]):                    #Write  mul $r2 $r2 $r4
    if(R2_36[R3_36])!="":                #Execute addu $r1 $r1 $r2
        True
    R2_36[R3_36] = float(R2_36[R3_36])*float(R4_36[R5_36])
    R1_36=R1_36 + float(R2_36[R3_36])          #Memory addu $r1 $r1 $r2
    R1_36=R1_36                  #Write addu $r1 $r1 $r2
    print(R2_36[0:R3_36])
    #print R1_36
R3_36=R3_36+1                        #Increment counter addiu $r3 $r3 #4
R5_36=R5_36+1                        #Increment counter addiu $r5 $r5 #4
R7_36=R7_36-1
```

Output -

```
Python 3.7.7 (bundled)
>>> %Run Withforwarding.py
 Mini Project 3 - MIPS ISA Dot Product Implementation with Forwarding
 []
 [0.0]
 [0.0, 3.0]
 [0.0, 3.0, 24.0]
 [0.0, 3.0, 24.0, 35.0]
 [0.0, 3.0, 24.0, 35.0, 15.0]
 [0.0, 3.0, 24.0, 35.0, 15.0, 0.0]
 [0.0, 3.0, 24.0, 35.0, 15.0, 0.0, 8.0]
 [0.0, 3.0, 24.0, 35.0, 15.0, 0.0, 8.0, 15.0]
 RESULTS AND ANALYSIS
 Dot Product Using Forwarding
 Two vectors are 014538238 and 236750450
 Dot Product Result= 100.0 stalls 19
```

Conclusion

In this project, we have successfully implemented the MIPS ISA Dot Product with Forward Chaining. While implementing this project, we understood the architecture of MIPS. We also learned the various types of instructions present in MIPS. Designing hazard detection functions to detect hazards was a critical task and was implemented successfully. We also designed forwarding functions. We have also implemented pipelining in this project. We thank Dr, JeongHee Kim for this opportunity.

Link to project is below:

https://drive.google.com/file/d/1OPk64jlnUf7xD_CiCc2uSaNvPT25ooWZ/view

References

[1]Lecture notes/Prof.JeongHee Kim/SJSU SanJose, CA./Spring-2020 /EE-275

[2] D. A. Patterson, J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 2014, ISBN 978-0-12-407726-3.

[3] B. Parhami, \Real Arithmetic" in Computer Arithmetic - Algorithms and Hardware DeSign36s, 2nd Ed. Oxford U. Press, 2010.

[4] S. P. Ritpurkar, M. N. Thakare, G. D. Korde, "Design and simulation of 32-Bit RISC architecture based on MIPS using VHDL", *Advanced Computing and Communication Systems 2015 International Conference on*, pp. 1-6, 2015.

[5] M. N. Topiwala, N. Saraswathi, "Implementation of a 32-bit MIPS based RISC processor using Cadence", *Advanced Communication Control and Computing Technologies (ICACCCT) 2014 International Conference on*, pp. 979-983, 2014.