Department of Electrical Engineering

EE-275- ADVANCED COMPUTER ARCHITECTURE

PROJECT # 3

MIPS ISA DOT PRODUCT PROGRAMMING USING "FORWARD CHAINING"

Submitted By: Nikitha Matasugur
Anirudh Ashrit
Meeta Griglani

TABLE OF CONTENTS

1. Abstract

The goal of this project is to write a code that can perform a dot product of 2 input vectors on MIPS Instruction Set Architecture. The project also compares the results with the forwarding technique in a pipelined CPU. The code is written and simulated in Python. The most important objective of the project is to implement and analyze the design using forwarding technique also known as forward chaining to avoid data hazards. Forwarding optimization technique is implemented to resolve data hazards in CPUs. Pipelining can improve the performance of the processor by increasing the throughput, the average instructions completed per clock cycle, but introduces data hazards. Forwarding is a method of exploiting instruction-level parallelism (ILP). In the project, the performance comparative analysis on number of stalls is done for the dot product with and without forwarding.

2. Introduction

2.1 RISC

RISC (Reduced Instruction Set Computer) is a philosophy introduced by Cocke IBM, Patterson, Hennessy in the 1980s. The main idea of this philosophy is to keep the instruction set small with highly-optimized instructions and simple through fixed instruction length(eg:32-bit), limiting the number of addressing modes and operations to make the hardware simpler hence easy to build and test as compared to the other philosophy that insists of a more specialized set of instructions like CISC(Complex Instruction Set Computer) philosophy. In RISC, the software will do complicated operations by separating the operations into several simple instructions instead of adding in more complex hardware to perform the operation in one instruction. The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. Examples of processors implemented based on RISC philosophy are SUN's, UltraSparc, ARM's ARM11, Motorola's PowerPC, MIPS etc.

2.2 MIPS

MIPS, which stands for Microprocessor without interlocked Pipeline Stages, is computer architecture developed by David A. Patterson based on the RISC philosophy. In the early 80s, Professor John L. Hennessy started the development of MIPS processors with his graduate students in Stanford University. The concept is to create a faster processor by using simple instructions with a great compiler for instruction scheduling and also pipelining the hardware so that each stage of the hardware can run independently on different instructions at the same time to fully utilize the processor time.

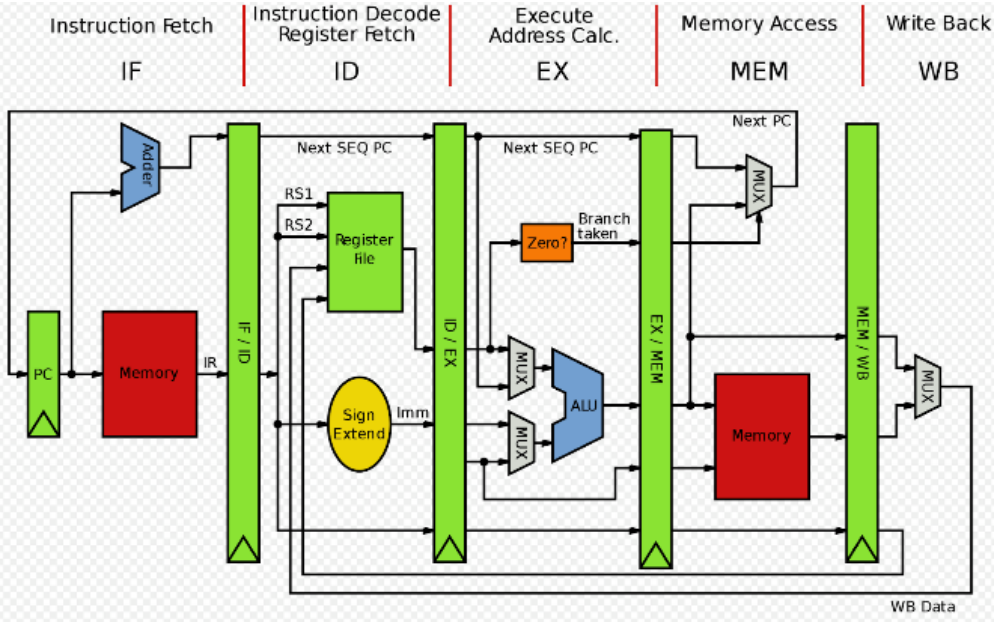


Fig. 1 MIPS Architecture

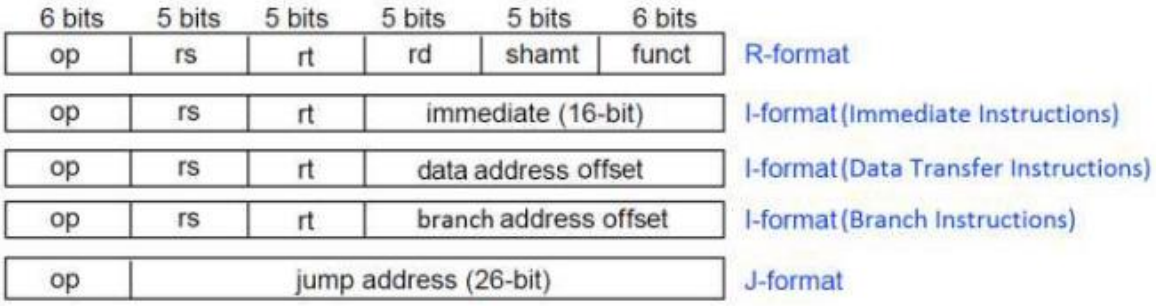


Fig. 2 MIPS Instruction Format

2.3 MIPS Internal operation

Instruction Fetch phase fetches the next instruction. First, it sends the contents of the PC register, which contains the address for the next instruction, to the instruction memory. The instruction memory will then respond by sending the correct instruction.

Instruction Decode phase has two main tasks: to calculate the next PC and fetch the operands for the current instruction. There are three possibilities for the next PC.

1. For the most common case, for all "normal" (meaning instructions that are not branches or jumps), we must simply calculate PC+4 to get the address of the next instruction.
2. For jumps and branches (if the branch is taken), we might also have to add some immediate value (the branch offset) to the PC. This branch offset is encoded in the instruction itself.
3. For jr and jalr instructions, we need to use the value of a register instead.

Execution phase executes the instruction. Any calculations necessary are done in this phase. These calculations are all done by the ALU (Arithmetic and Logic Unit). The ALU needs two operands. These either come from the ID phase and thus in turn from the register file (or PC+4 for jump and link instructions), or one operand comes from the ID phase and the other comes from the instruction register to supply an immediate value.

Memory access phase is to store operands into memory or load operands from memory. So, for all instructions except loads and stores, the Memory Access simply passes on the value from the ALU on to the WB stage. For loads and stores, the Memory Access needs to send the effective address calculated by the ALU to memory.

Write Back phase, finally, is a very simple one. It simply takes the output of the Memory access phase and sends it back to the write back phase to store the result into the destination register.

2.4 MIPS Instruction Formats and Addressing modes

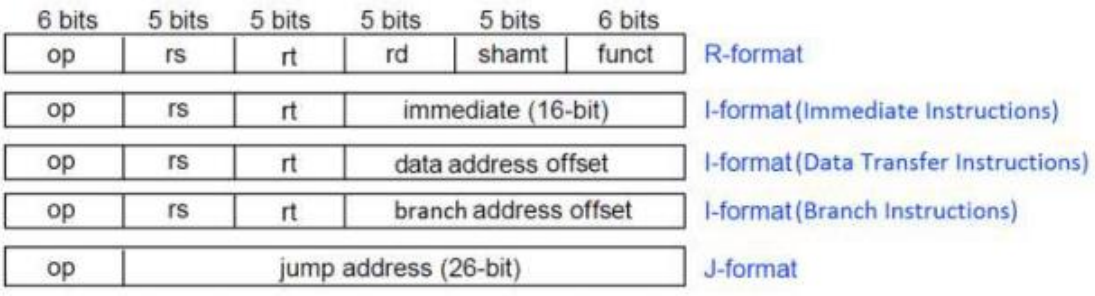
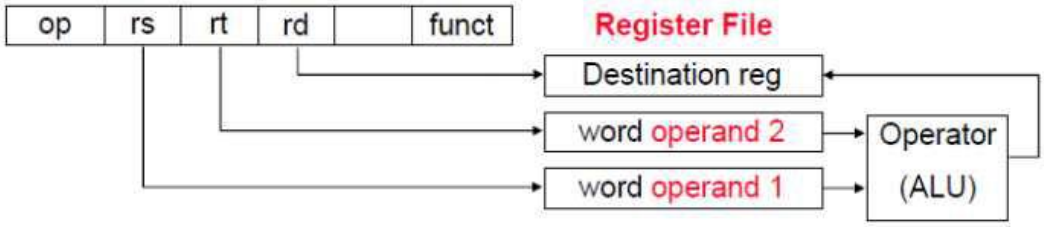
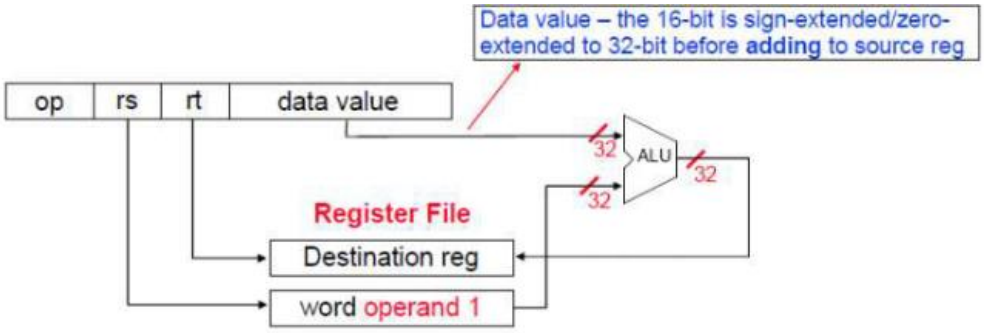


Fig. 3 MIPS Instruction Format

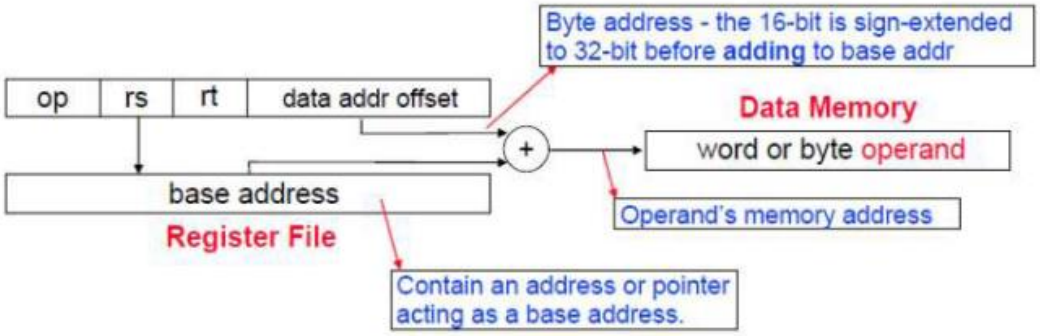
1. **Register addressing**: Perform operation on source and target register and store the result into destination register.



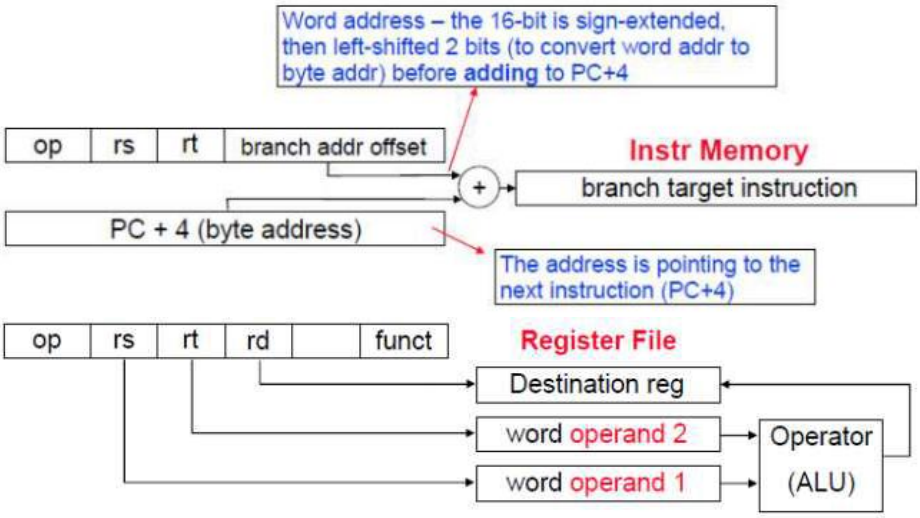
2. **Immediate addressing**: Perform operation on source register and immediate and store the result into target register.



3. **Displacement addressing**: Perform operation on source register and immediate, the result is then used as address to access the data memory to load/store data to/from target register.



4. **PC - relative addressing**: Perform operation on source and target register to determine next PC condition, the immediate is used as address offset for next PC.



2.5 MIPS Forwarding technique

Pipelining creates data hazards which changes the program and violates the pipeline correctness. This is called stalling the pipeline. We can fix it by forwarding the data within the pipeline. The hazards are of three types, all which are explained below.

1. Data hazards: A Data Hazard is a condition where either the source or the destination operands of instructions are not available when needed in the pipeline. Data hazards always exist in a processor designed based on the pipeline approach
2. Structural Hazards occur when two (or more) instructions require the use of a given hardware resource at the same time.
3. Control Hazards occur when a delay in the availability of instruction or the memory address needed to fetch the instruction.

The Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.

Forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation,

control logic selects the forwarded result as the ALU input rather than the value read from the register file.

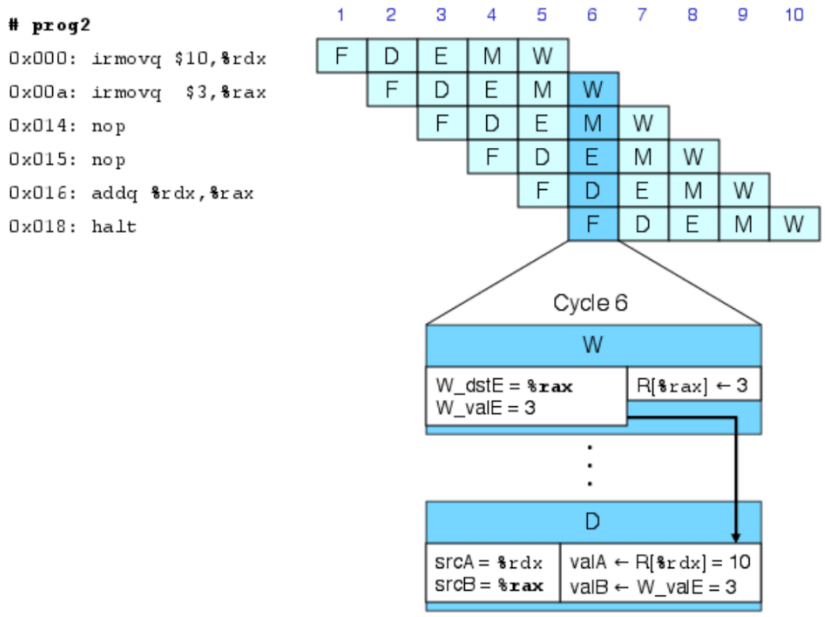


Fig. 4 Data Forwarding Example

Code:

Without Forwarding:

```
# When values in Reg6_46 and Reg0_46 become equal it displays final output

def meeta():

    global stall_withoutforwarding

    print ('\n')

    print ('Student IDs are', ANI_46, 'and', NIK_46)

    stall_withoutforwarding = stall_withoutforwarding + 3  # 3 INSTRUCTION DELAY

    print ('Dot Product: ', Reg1_46)

    print ('Stalls Without Forwarding:', stall_withoutforwarding)

    print ('This is Dot Product of given inputs Without Forwarding\n')

    exit()


# INPUTS

ANI_46 = '014535846'

NIK_46 = '014489852'


Reg0_46 = 0       # First register in MIPS is 0

Reg1_46 = 0       # Sum of the values are stored
```

```
Reg2_46 = []        # Reg2_46 and Reg3_46 store the first and second input

Reg3_46 = []

Reg4_46 = 0 * 1     # Reg4_46 and Reg5_46 indicate the next value

Reg5_46 = 0 * 1



Reg6_46 = int(len(ANI_46))

stall_withoutforwarding = 0



for value in ANI_46:

   Reg2_46.append(value)



for value in NIK_46:

   Reg3_46.append(value)



# Execution of Logic



Reg1_46 = Reg0_46 + Reg0_46  # Adds the values in Register0

while 1:

   if Reg6_46 == Reg0_46:  # If value in Reg6_46 and R0 are equal then it'll produce output by calling function

      meeta()

   else:

      if Reg2_46:

         True



      if Reg2_46[Reg4_46] != '':

         if Reg3_46:

            True



      if len(Reg2_46) >= 0:

         if Reg3_46[Reg5_46] != '':

            if Reg2_46:

               True



      if Reg2_46[Reg4_46]:

         if len(Reg3_46) >= 0:

            stall_withoutforwarding = stall_withoutforwarding + 1

            pass  # Stall occurs and hence No Operation (NOP)



      if Reg2_46[Reg4_46]:

         temp_46 = float(Reg2_46[Reg4_46])

         temp_46 = temp_46 + 0

         if float(Reg3_46[Reg5_46]) >= 0:

            stall_withoutforwarding = stall_withoutforwarding + 1

            pass



      if Reg3_46[Reg5_46]:

         temp1_46 = float(Reg3_46[Reg5_46])

         temp1_46 = temp1_46 + 0

         stall_withoutforwarding = stall_withoutforwarding + 1

         pass
```

```
    if Reg3_46[Reg5_46] != '':
        if Reg2_46:
            True

    if len(Reg3_46) >= 0:
        stall_withoutforwarding = stall_withoutforwarding + 1
        pass

    if float(Reg3_46[Reg5_46]) >= 0:
        stall_withoutforwarding = stall_withoutforwarding + 1
        pass

    if Reg3_46[Reg5_46]:
        Reg2_46[Reg4_46] = float(Reg2_46[Reg4_46]) * float(Reg3_46[Reg5_46])
        stall_withoutforwarding = stall_withoutforwarding + 1
        pass

    if Reg3_46[Reg5_46] != '':
        if Reg2_46[Reg4_46] != '':
            Reg1_46 = Reg1_46 + float(Reg2_46[Reg4_46])
            Reg1_46 = Reg1_46
    Reg4_46 = Reg4_46 + 1
    Reg5_46 = Reg5_46 + 1

    if Reg6_46:
        stall_withoutforwarding = stall_withoutforwarding + 1
        pass
    if Reg6_46 != '':
        stall_withoutforwarding = stall_withoutforwarding + 1
        pass

    if len(str(Reg6_46)) >= 0:
        if Reg6_46 == Reg6_46:
            Reg6_46 = Reg6_46 - 1
    print (Reg2_46[0:Reg4_46])
```

With Forwarding:

```
# Function to produce final result when values in Reg0_46 and Reg6_46 become equal

def meeta():
    global stall_withforwarding
    print ('\n')
    print("The Given Inputs are ", ANI_46,"and",NIK_46)
    stall_withforwarding = stall_withforwarding + 1
    print("Dot Product: ", Reg1_46)
    print ("Stalls With Forwarding",stall_withforwarding)
    print("This is Dot Product Using Forwarding to Reduce Stalls")
    exit()
```

```
# INPUTS
ANI_46 = "014535846"
NIK_46 = "014489852"

Reg0_46=0 # First register in MIPS is 0

Reg1_46=0 # Sum of the values are stored

Reg2_46=[] # Reg2_46 and Reg3_46 store the first and second input
Reg3_46=[]

Reg4_46=0*1 # Reg4_46 and Reg5_46 indicate the next value
Reg5_46=0*1

Reg6_46 = int(len(ANI_46))

stall_withforwarding=0

for value in ANI_46:
    Reg2_46.append(value)

for value in NIK_46:
    Reg3_46.append(value)

# Execution of Logic
Reg1_46=(Reg0_46 + Reg0_46)

while(1):
 if Reg6_46==Reg0_46:
    meeta()
 else:
    if(Reg2_46):
        True

    if(Reg2_46[Reg4_46])!="":
        if(Reg3_46):
            True

    if len(Reg2_46)>=0:
        if(Reg3_46[Reg5_46])!="":
            if(Reg2_46):
                True

    if(Reg2_46[Reg4_46]):
        if len(Reg3_46)>=0:
            if(Reg3_46[Reg5_46])!="":
                if(Reg2_46):
                    True

    if(Reg2_46[Reg4_46]):
        temp_46 = float(Reg2_46[Reg4_46])
        temp_46 = temp_46 + 0
        if float(Reg3_46[Reg5_46]) >= 0:
            stall_withforwarding=stall_withforwarding+1
```

```
        pass    # Stall occurs and hence No Operation
(NOP)
```

```
    if(Reg3_46[Reg5_46]):
        temp1_46 = float(Reg3_46[Reg5_46])
        temp_46 = temp1_46 + 0
        if len(Reg3_46) >= 0:
            if(Reg3_46[Reg5_46])!="":
             True

    if float(Reg3_46[Reg5_46])>=0:
            stall_withforwarding=stall_withforwarding+1
            pass
```

```
        if(Reg3_46[Reg5_46]):
            if(Reg2_46[Reg4_46])!="":
                True
            Reg2_46[Reg4_46] =
float(Reg2_46[Reg4_46])*float(Reg3_46[Reg5_46])
            Reg1_46 = Reg1_46 + float(Reg2_46[Reg4_46])
            Reg1_46 = Reg1_46
            print(Reg2_46[0:Reg4_46])

        Reg4_46=Reg4_46+1                    #Increment
counter addiu $r3 $r3 #4
        Reg5_46=Reg5_46+1                    #Increment
counter addiu $r5 $r5 #4
        Reg6_46=Reg6_46-
```

Simulation Results:

Without Forwarding:

```
[Running] python -u "d:\EE\Sem 2\275\HWs\Mini project 2\VScode\.vscode\dotprodwf.py"
[0.0]
[0.0, 1.0]
[0.0, 1.0, 16.0]
[0.0, 1.0, 16.0, 20.0]
[0.0, 1.0, 16.0, 20.0, 24.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0, 64.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0, 64.0, 20.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0, 64.0, 20.0, 12.0]

Student IDs are 014535846 and 014489852
Dot Product:  202.0
Stalls Without Forwarding: 75
This is Dot Product of given inputs Without Forwarding
```

With Forwarding:

```
[Running] python -u "d:\EE\Sem 2\275\HWs\Mini project 2\VScode\.vscode\dotprod.py"
[]
[0.0]
[0.0, 1.0]
[0.0, 1.0, 16.0]
[0.0, 1.0, 16.0, 20.0]
[0.0, 1.0, 16.0, 20.0, 24.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0, 64.0]
[0.0, 1.0, 16.0, 20.0, 24.0, 45.0, 64.0, 20.0]


The Given Inputs are  014535846 and 014489852
Dot Product:  202.0
Stalls With Forwarding 19
This is Dot Product Using Forwarding to Reduce Stalls

[Done] exited with code=0 in 0.214 seconds
```

Conclusion:

The dot product is successfully implemented and simulated in Python. The design is verified, and the number of stalls is compared with and without using forwarding technique. The obtained results shows the number of stalls in the design without forwarding is “” and with adding forwarding technique the number of stalls significantly reduced to the value of “”. This shows the importance of Forwarding technique to prevent hazards in the MIPS architecture designs.

References:

1. Patterson, David. Computer Architecture A Quantitative Approach. 2ed. s.l. :Morgan Kaufmann.
2. Hennessy, John L. and Patterson, David A. Computer Organization and Design : The Hardware/Software Interface. 4th. San Francisco : Morgan Kaufmann
3. https://www.geeksforgeeks.org/computer-organization-and-architecture-pipelining-set-2-dependencies-and-data-hazard/
4. https://www.cs.umd.edu/users/meesh/411/Pipelining_Lecture.pdf