|  |  |
| --- | --- |
| Implementation of MIPS Instruction set architecture  *Dot product of two vectors* | Abstract  *MIPS architecture often faces 3 types of pipeline hazards -Structural, Data, Control. Data forwarding) is an optimization in pipelined CPUs to limit performance deficits which occur due to pipeline stalls. A data hazard can lead to a pipeline stall when the current operation has to wait for the results of an earlier operation which has not yet finished. Forwarding is a method of exploiting instruction-level parallelism (ILP). In this report we optimize the performance of a given benchmark program and make an empirical analysis about the stalls required without forwarding and with forwarding*  Hemanta Ingle  *MS EE (Id-013765336), San-Jose State University, CA* |

**Contents-**

1. Introduction
2. Pipelining and Data forwarding
3. Implementation and Analysis
4. Data flow and Algorithm
5. Code Description
6. Appendix- Code and Results /Simulation and Verification
7. Biodata
8. Conclusion

1. References
2. **Introduction**

The goal of this project is to design and write a Verilog/ C/C++ code capable of performing dot product of two vectors. MIPS architecture often faces 3 types of pipeline hazards -Structural, Data, Control. Data forwarding) is an optimization in pipelined CPUs to limit performance deficits which occur due to pipeline stalls. A data hazard can lead to a pipeline stall when the current operation has to wait for the results of an earlier operation which has not yet finished. Forwarding is a method of exploiting instruction-level parallelism (ILP). In this report we optimize the performance of a given benchmark program and make an empirical analysis about the stalls required without forwarding and with forwarding

Implementing of a code in a programming language lets us understand the working of the architecture and hence we can realize a efficient architecture to overcome the current challenges in the data processing. Forwarding is a kind of optimization of the available resources which lead to a more time efficient processing of the instructions without any or with minimal stalls thus saving a lot of computational power of a processor when we think a architecture with billions of transistors implementing a logic.

This paper provides methods to identify and resolve every possible data hazard that might arise during execution phase within the range of the basic MIPS core instruction set. The techniques used to resolve data hazard in this paper are data forwarding and pipeline stages stalling. When data hazard arises, it is first resolve by using data forwarding. If the problem persists, we use pipeline stages stalling then only follow by another data forwarding to resolve the data hazard. This combination will reduce the impact of data hazard on the processor throughput, instead of only using the pipeline stages stalling. This paper delivers a comprehensive analysis and the development of the data hazard resolving blocks that are able to resolve data hazard arises.

1. **Pipelining and Data Forwarding**

Data hazards always exist in a processor designed based on pipeline approach. It can cause computational error when multiple instructions are overlapped during its execution which involve accessing the processor's system registers, (e.g. Register File (RF), interrupt controller (CPO) registers and HILO register). Although data hazard exists in pipeline processor and it require extra hardware to resolved, the high performance achieved by pipelined processor still outweighs its counter parts and remains a popular choice in processor design.

Data hazard occurs due to data dependency. Data dependency is distinguishing into 4 types, Read-after-Read (RAR), Write-after-Read (WAR), Read-after-Write (RAW) and Write-after-Write (WAW). 3 out of 4 will lead to data hazard. Both WAR and WAW dependencies will never occur when the pipeline processor only allows system registers to be updated at a specific stage. We observed that RAW dependency causes majority of the data hazards. In RISC32 convention, Register File (RF) will only be updated at the fifth (Write-Back (WB)) stage. Since read (ID stage) and write (WB stage) to RF is occurred in two different stages, an instruction may attempt to read RF before it is updated with the latest data. This condition can be resolved in several ways, generally classified into software and hardware approach. The software approach relies on compiler to reorder the user code or insert a delay slot to resolve the combinations of instructions that might produce data hazard. Resolving data hazard using software approach is less complex but will affect the processor throughput more severely than the hardware approach [6]. Also, code reordering might not be efficient since it depends on the nature of the program flow. On the other hand, hardware approach based on forwarding scheme does not affect the processor throughput (except the load-use case which we will cover later). The result may forward to the next instruction without wasting any processor cycle.

The paper aims to generalize the concept of data hazard based on the RF which we have further extended it into interrupt controller unit (CPO) registers and multiplier HILO register. The extended version provides comprehensive verification coverage for the data hazard based on the RISC32 processor and is useful for the micro-architecture development. Our work also includes miscellaneous scenario: the data hazard related to the unconditional branch (*jal* and *jalr*) with *$ra* register (return address register).

This paper is organized as follows: Section 2 describes the background of our work. Section 3 describes the processor micro-architecture requirements prior to resolve data hazards. Section 4 and 5 describe the process of resolving data hazard using forwarding and interlock scheme respectively. Lastly, Section 6 concludes the finding of our paper.

The problem with [data hazards](http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dataHaz.html), introduced by this sequence of instructions can be solved with a simple hardware technique called ***forwarding***. 

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| ADD | **R1**, R2, R3 | IF | ID | EX | MEM | **WB** |  |  |
| SUB | R4, R5, **R1** |  | IF | **IDsub** | EX | MEM | WB |  |
| AND | R6, **R1,** R7 |  |  | IF | **IDand** | EX | MEM | WB |

If the result can be moved from where the ADD produces it (EX/MEM register), to where the SUB needs it (ALU input latch), then the need for a stall can be avoided.   
Using this observation, forwarding works as follows:

The ALU result from the EX/MEM register is always ***fed back*** to the ALU input latches.   
If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the 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.

|  |  |
| --- | --- |
|  | **Forwarding of results to the ALU requires the additional of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.**  The paths correspond to a forwarding of: (a) the ALU output at the end of EX, (b) the ALU output at the end of MEM, and (c) the memory output at the end of MEM. |

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** | **IDsub** | EX | MEM | WB |  |
| AND | R6, **R1,** R7 |  |  | **stall** | **stall** | IF | **IDand** | EX | MEM | WB |

As our example shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started three cycles earlier. Forwarding can be arranged from MEM/WB latch to ALU input also.  Using those forwarding paths the code sequence can be executed without stalls: 

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| ADD | **R1**, R2, R3 | IF | ID | **EXadd** | **MEMadd** | WB |  |  |
| SUB | R4, R5, **R1** |  | IF | ID | **EXsub** | MEM | WB |  |
| AND | R6, **R1,** R7 |  |  | IF | ID | **EXand** | MEM | WB |

The first forwarding is for value of **R1** from **EXadd** to **EXsub** .   
The second forwarding is also for value of **R1** from **MEMadd**to **EXand**.   
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.

1. **Implementation and analysis**

The pipeline processor, RISC32, which we have developed, is based on 5-cycle instruction execution, which corresponds to 5 hardware stages: Instruction Fetch (IF), Instruction Decode and Register File Read (ID), Execution or address calculation (EX), Data Memory Access (MEM) and Write Back (WB).

The data forwarding in MIPS architecture requires us to analyze the data dependencies in a given program here a set of instructions in order to evaluate the benchmark performance of a processor is provided.

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

**1)Decoding the Program and deciding the stalls-**

STALL1-

In the below fig. below the implementation of the program without forwarding is provided. 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.

**SOLUTION-**

The data of register R2 is available in the memory stage as it reads the data from the effective address 0($R3) from the memory location so we can forward it right away without waiting for the write back to the R2 register.

**STALL2-**

The second 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 execution of the 6th instruction.

**SOLUTION-**

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 falling edge of the same clock cycle.

The next instruction doesn’t have any data dependencies and so there is no any hazard present.

**STALL3**-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.

**Solution-**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

For demonstration purpose, the first vector should be 9 single digits taken from the 9-digit student ID and the second vector the first vector added digit-wise by 2. Note that 9 added by 2 is “wrapped around” to 1. For example, for eg. If the ID is 012345678, the first vector is (0,1,2,3,4,5,6,7,8) and the second vector (2,3,4,5,6,7,8,9,0), making dot product to 0×2+1×3+2×4+3×5+4×6+5×7+6×8+7×9+8×0=3+8+15+24+ 35+48+63=196.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | R2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| lw $r4 0($r5) |  |  |  | F | D | E | M | W |  |  |  |  | R2 |  |  |  |  |  |  |  |  |  |  |  |
| 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 |  | R7 |  |  |  |
| 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) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| .  .  . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Fig3.1-without data forwarding

Fig.3.2- 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 |  | R2 |  |  |  |  |  |  |  |  |
| lw $r2 0($r3) |  |  | F | D | E | M | W |  | R2 |  |  |  |  |  |  |  |
| 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) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ……. |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Fig. 3.4- Execution after last iteration Without Forwarding

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INST | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| addu $r1 $r0 $r0 ; result = 0 | F | D | E | M | W |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LOOP |  |  |  |  |  |  |  |  |  |  | Control Hazard Delay |  |  |  |  |  |  |  |
| 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 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| jr $r31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Fig. 3.5 -Execution after last iteration with data forwarding

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INST | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| addu $r1 $r0 $r0 ; result = 0 | F | D | E | M | W |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LOOP |  |  |  |  |  |  |  |  | Control Hazard Delay |  |  |  |  |  |  |  |  |  |
| 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 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| jr $r31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. **Data flow and algorithm**

**1)Program Analysis-**

dot\_product:

addu $r1 $r0 $r0 ; // R0+R0 🡪R1 ===R1 🡪0

loop:

beq $r7 $r0 done ; // go to done : if R7==R0=0 ===R7=No of Iterations

lw $r2 0($r3) ; // Load 1st element from vector (R3+0) to R2 ===R2🡪 1

lw $r4 0($r5) ; // Load 1st element from vector (R5+0) to R4 ===R4🡪 5

mul $r2 $r2 $r4 ; // R4\*R2🡪 R2

addu $r1 $r1 $r2 ; // R2+R1🡪R1

addiu $r3 $r3 #4 // Increment to next location

addiu $r5 $r5 #4 // “ ”

addiu $r7 $r7 #-1 //Decrement the iteration register by 1

j loop //jump to loop:

done:

jr $r31 // Jump to return address

Fig 4.1-Block diagram representation of the code

|  |
| --- |
| 0🡪1  4🡪2  8🡪3  12🡪4 |

R2

R3=

R1

Result

R1

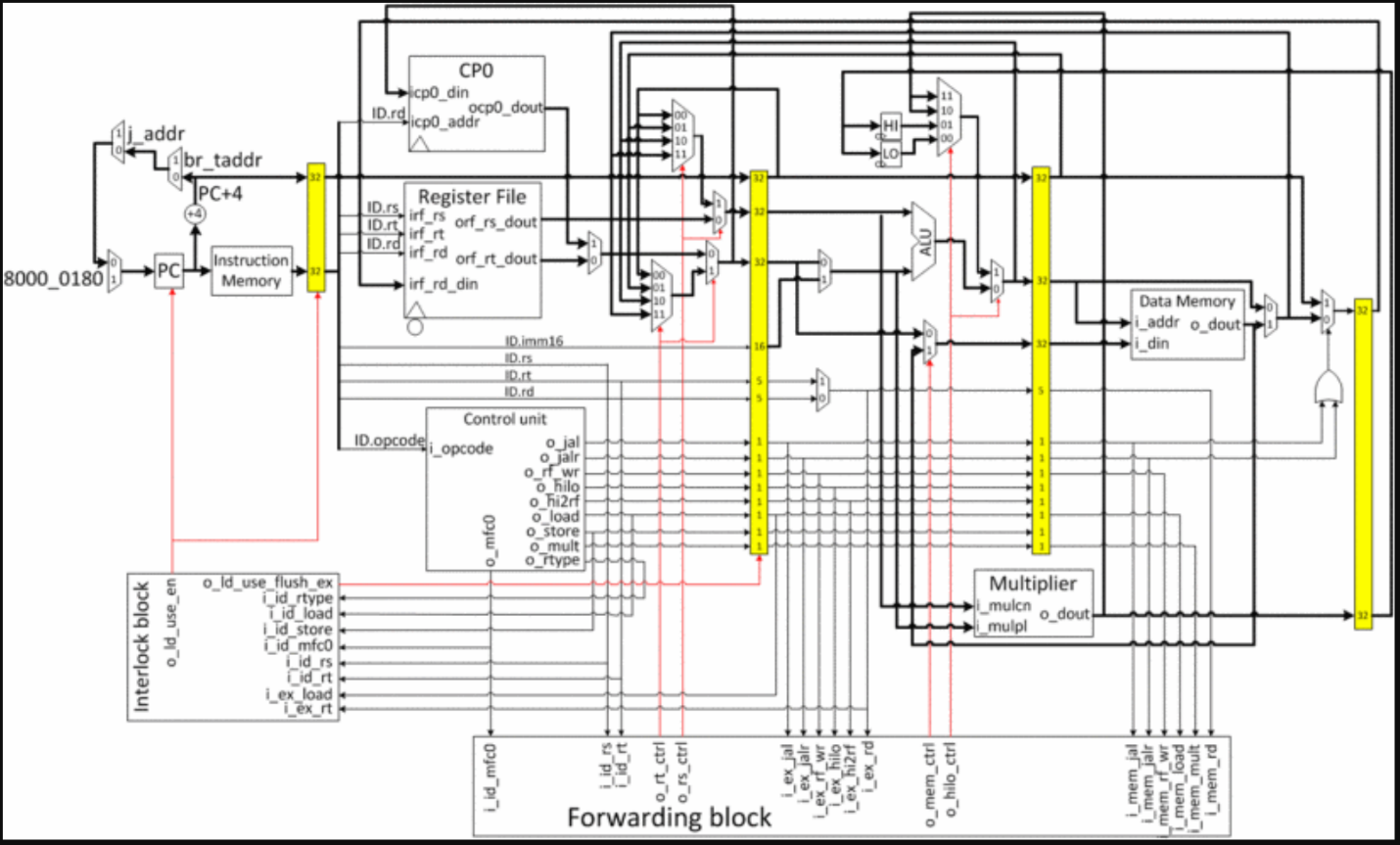
R2

|  |
| --- |
| 16🡪5  20🡪6  24🡪7  28🡪8 |

R4

R5=

**2)Pipelined Processor Architecture-**



1. **Code Description-**

For demonstration purpose, the first vector should is 9 single digits taken from the 9-digit student ID and the second vector the first vector added digit-wise by 2. Note that 9 added by 2 is “wrapped around” to 1. For example, for eg. If the ID is 012345678, the first vector is (0,1,2,3,4,5,6,7,8) and the second vector (2,3,4,5,6,7,8,9,0), making dot product to 0×2+1×3+2×4+3×5+4×6+5×7+6×8+7×9+8×0=3+8+15+24+ 35+48+63=196.

When forwarding is not used then r4 register data is required by next instruction “mul $r2 $r2 $r4”, but it is still not written back by previous instruction “lw $r4 0($r5)”, therefore next instruction “mul $r2 $r2 $r4” has to stall for two cycles. Whereas when forwarding is used then same contents can be passed directly to decode stage of “next instruction mul $r2 $r2 $r4” from M/W buffer, this results in only one stall cycle.

Similarly, previous instruction “mul $r2 $r2 $r4” cannot pass contents of register r2 since they are yet not written back, which results in two stall cycles for next instruction “addu $r1 $r1 $r2. Whereas when forwarding is used then data from Execution stage (ALU-Memory Buffer) is transferred to decode stage of next instruction. This saves two stall cycles and no stalls are encountered during entire cycle.

It should be noted that without forwarding given MIPS instruction set takes

Total No. of stalls= (Stalls /iteration\* no. of iterations) +3stalls

= (8\*9) +3=75 Stall Clock cycles

whereas with forwarding technique same code is executed in stalls only.

= (2\*9) +1=19 Stall Clock Cycles

1. **Appendix Code and Results-**

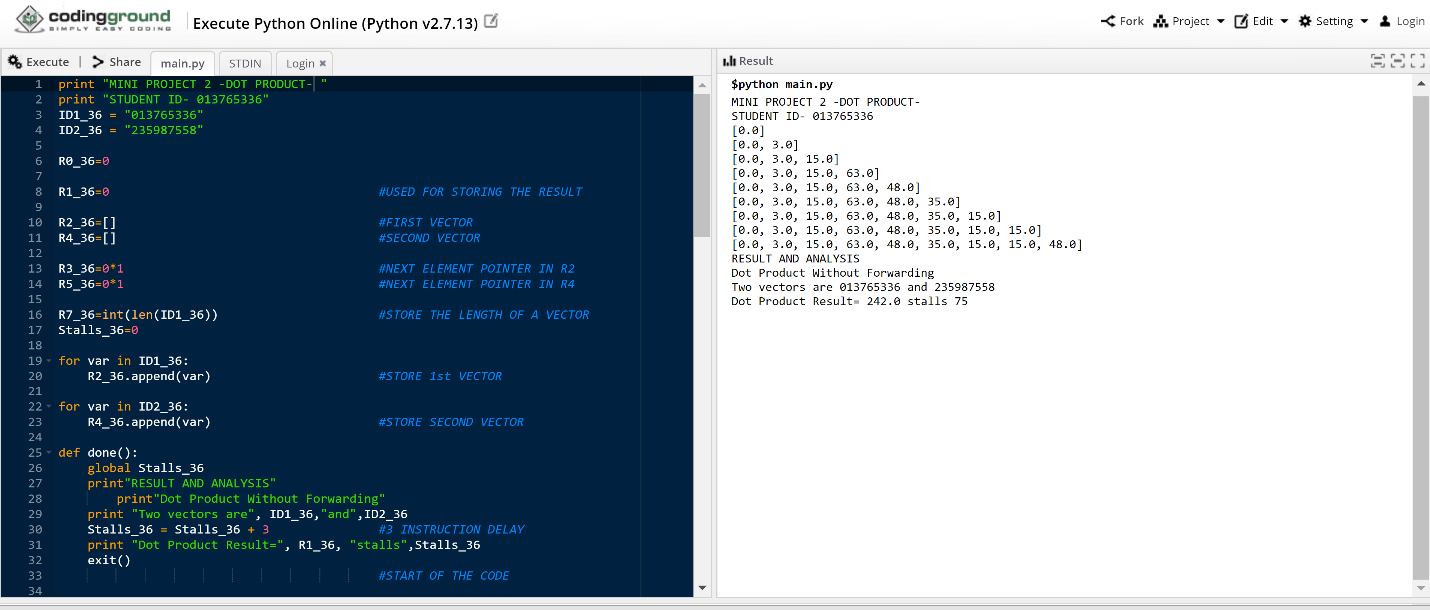
****

Fig.5.1- Results for program without data forwarding

CODE-

print "MINI PROJECT 2 -DOT PRODUCT- "

print "STUDENT ID- 013765336"

ID1\_36 = "013765336"

ID2\_36 = "235987558"

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"RESULT 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): #Memory BEQ $R7 $R0 done ; done looping?

R7\_36=R7\_36-1 #Write and Decrement counter addiu $R7 $R7 #-1

print R2\_36[0:R3\_36]

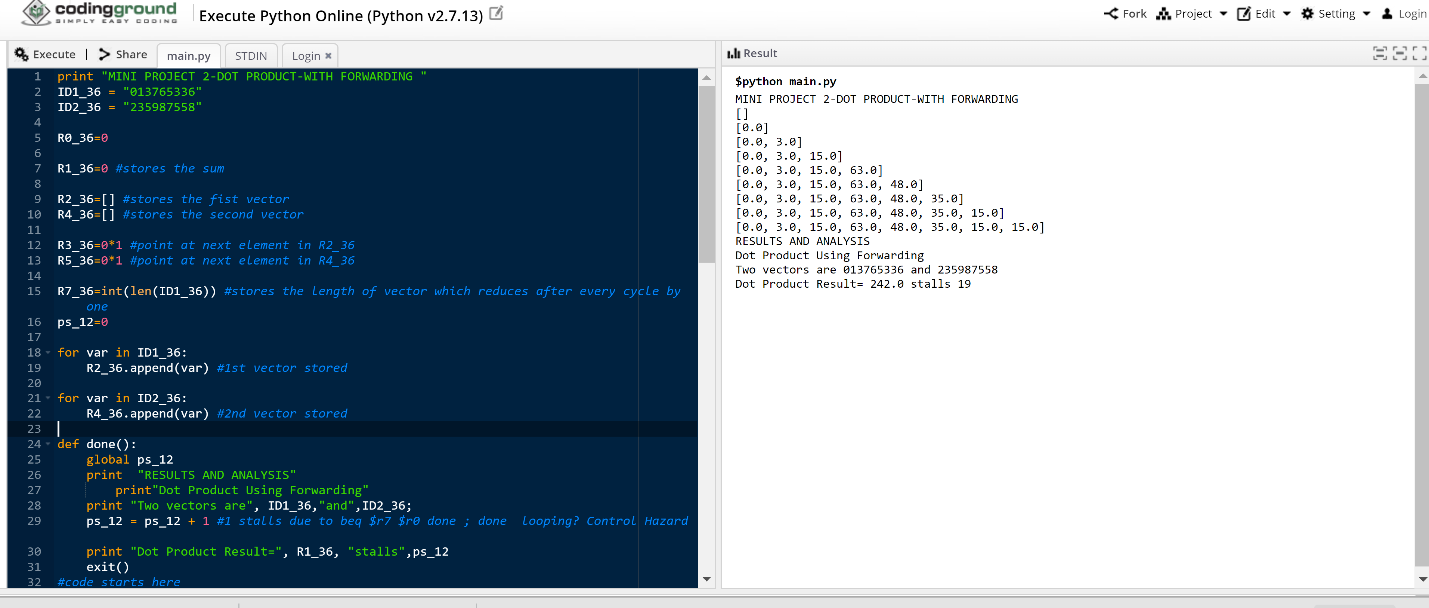


Fig. 5.2-Results for program with forwarding

CODE-

print "MINI PROJECT 2-DOT PRODUCT-WITH FORWARDING "

ID1\_36 = "013765336"

ID2\_36 = "235987558"

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

1. **BIODATA-**

My name is Hemanta Ingle and I am pursuing Master of Electrical Engineering with specialization in Embedded Systems at San Jose State University, California. I am interested in Self-Driving vehicles and IOT devices, as a part of curriculum I opted for courses like Embedded System Design, Advance Computer Architecture, Machine Learning and Semiconductor Devices. I have also authored a paper on Autonomous car for university campus with application of Artificial Intelligence and Machine Learning. I am working on an IOT device for assisting firefighters in their tasks. My inclination for IOT devices is from undergraduate studies and my work Experience.

Working as a Service Engineer in Siraga, India, provided me with a thorough understanding of application of Process Automation in LPG bottling industries which made me move up in my professional life as I gained more knowledge day by day. I have also worked as a Project Engineer at PAM Systems, India, which enhanced my Experience in Industrial Automation that led in successful completion of 5 major projects for automation in LPG bottling plants using PLC, SCADA, HMI. As a Project Engineer, I developed my skills like Production Management, Vendor Assignment, Project Management and Team Management.

I have also successfully completed projects like Data Acquisition Robot, Power Factor Correction, Traffic Signal using FPGA as a part of my undergraduate degree in Electronics and Telecommunication at SGGSIE&T, India. As a part of curriculum, I choose courses like Digital System Design, Digital VLSI, Analog Devices, Digital Signal Processing. The academic courses made me a quick learner and enhanced qualities like time management, punctuality, and honesty. I have also completed diploma in Industrial Electronics in 2008, during the course I learned courses like Industrial Drives, Opto-Electronics, Power electronics, Control Systems which gave me a good understanding about the processes used in industry for manufacturing.

I aim to work in the field of Embedded Systems which can provide a good future to Autonomous Vehicles making traveling safer, more enjoyable and eco-friendlier for benefiting coming generations.

1. **CONCLUSION-**

In this paper, we presented a thorough analysis on data hazard of 32-bit MIPS ISA compatible 5-stage pipeline processor and the overall resolving scheme to handle basic MIPS core instruction set data hazards that might arise. The analysis shown here can serve as a reference for MIPS ISA compatible pipeline processor developers to eliminate all the data hazards. All the data hazards should be resolved prior to further development as the data correctness is extremely important.

We intended to develop a pipeline processor for Internet-of-Things (IoT) in future, which is mostly going to handle large amount of data, including data collection from sensors, data aggregation and data transmission to another device. Therefore, data correctness is critical to IoT and hence we provide this paper as the useful information to resolve basic MIPS core instruction set data hazard that might arises in processor level.

1. **References**

[1]Lecture notes/Prof.Chang Choo/SJSU SanJose, CA./Spring-2019 /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.