THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

A Fault Tolerant Error Detection and Verification Approach

By

Srikanth Bolisetty

Major Professor

Dr.Lois Hawkes

A project submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Master of Science

Degree Awarded

Fall 2004

The members of the committee approve the Masters project of Srikanth Bolisetty defended on _____

_____

Dr.Lois Hawkes

Supervising Professor

_____

Dr.Robert van Engelen

Committee Member

_____

Dr.Daniel Schwartz

Committee Member

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to Dr. Lois Hawkes for the invaluable guidance, support and advise throughout my project and graduate studies here at Florida State University. Without her patience and encouragement, this project would have been difficult to complete. I would also like to thank the other members of my committee, Dr. Robert van Engelen and Dr. Daniel Schwartz. Their assistance in this endeavor and throughout my graduate studies will always be appreciated. Thanks are due to all my family members, friends and well-wishers for their constant encouragement, patience and support.

Table of Contents

Abstract……..05

1.0 Introduction……..06

1.1 Outline…….07

2.0 Microprocessor faults……..08

2.1 Fault tolerant techniques…….09

2.2 Prior and Related work……09

2.3 New Reliability Challenges…….12

2.4 Traditional Microprocessor Verification Methods……..13

3.0 Dynamic Verification……..15

3.1 DIVA Architecture……..15

3.2 Discussion……19

4.0 Modified DIVA with a predictor…….20

5.0 Simulation Setup…….24

5.1 Simulation Results……..25

6.0 Conclusion……..27

6.1 Suggestions and Improvement……27

References……28

Appendix A…….30

Appendix B…….31

Appendix C…….49

Appendix D…….69

Abstract:

There are three important attributes of a modern microprocessor namely reliability, performance and cost. Reliability is the most important attribute of a microprocessor followed by performance and cost [14]. It's unrealistic to think that hardware is always perfect especially when you consider the fact that there are over 42 million transistors in the latest Pentium 4 processor released by Intel, or to assume that every single transistor will work as expected for an infinite amount of time. All three attributes are inter-related. When you tend to increase the reliability of the microprocessor, the performance comes down. In an effort to increase sales, most of the chip manufacturing companies like Intel, AMD etc. have concentrated on increasing the performance of the processor by making faster chips without giving much importance to the reliability aspect of the processor. One recent example of faulty processor design is the Intel Pentium processor that had a bug in some of the floating-point divide units, which caused erroneous results in floating point divide operations [1]. This was actually a functional design error, which occurred at the design stage of the chip. Another reliability example is the MIPS1000 processor, which had implementation problems. In these cases, the manufactures had to replace the processors at a great cost. So, an important issue concerning the microprocessor architects at present is to build a fast, high-performance processor without compromising on reliability. This project analyzes some fault tolerance (FT) techniques and their design tradeoffs and also proposes a new FT technique. This new approach tries to minimize the errors that can occur in the core microprocessor.

1.0 Introduction:

To increase the performance of a microprocessor, it is necessary that the chip execute instructions at a faster rate. One of the factors that influence the processor speed is the chip size. In order to make faster chips, the size of the chip has to be reduced. The smaller the size, the faster the chip because now the electrons have to travel less distance within the chip. A processor may have transient execution errors, which occur due to a number of problems like noise related faults, natural radiation interference, increased complexity etc. which occur due to the finer feature size of the circuits on the chip that effect the reliability of the microprocessor. Traditional fault tolerance techniques can address these problems, but at a great cost. Moreover these techniques can only address permanent or transient faults. They cannot address design faults since if the design fault occurs in each of the redundant components, then the redundant components will agree on the wrong execution. There are no specific fault tolerance techniques that can address these design, operational, permanent and transient faults without much performance loss. With this background, this project evaluates a fault-tolerance technique called *dynamic verification* proposed by T.Austin [14] where the execution of the processor is verified by re-executing all the instructions with a simple and cheap checker. This project also tries to improve upon *dynamic verification* by adding a data predictor before the checker so that all the instructions need not be re-executed and this promises to address design, operational, permanent and transient faults without much performance loss. The design tradeoff between error recovery and performance is analyzed through simulations.

1.1 Outline:

This document is organized into the following sections. Section 2 discusses briefly the faults that occur in microprocessors, fault tolerant and verification techniques, prior work done in this area and new reliability challenges. Section 3 introduces the concept of dynamic verification, the DIVA[1] architecture, its advantages and disadvantages. Section 4 introduces modified DIVA with a predictor, simulation setup, and a brief discussion. Section 5 concludes this paper with personal annotations on suggestions and improvements and also future work that can be done in this field.

[1] Dynamic Instruction Verification Architecture

2.0 Microprocessor faults:

There are a whole lot of challenges that must be overcome to build a high performance but reliable microprocessor. There are many sources of errors that can reduce the reliability of the microprocessor, which can occur during design, verification and manufacturing stages. These faults in general can be classified into three categories as *Design faults, Manufacturing faults,* and *Operational faults* [2]. *Design faults* occur due to human error in the design or the specification of the system component. This results in the microprocessor producing incorrect results for certain specific inputs. *Manufacturing faults* occur due to improper fabrication. This can result in open circuits or improper doping in the CMOS transistors. *Operational faults* occur due to the effect of the environmental conditions. These operational faults can further be sub-divided into *Permanent, Intermittent* and *Transient faults* [3]. *Permanent faults* occur consistently due to internal failure of the microprocessor. These can occur due to Electrometal migration or due to hot electrons, which can damage the chip permanently. *Intermittent faults* do not appear consistently like permanent faults. They occur due to extreme operating conditions like low voltage, high temperature, high frequency, and high electrical noise. *Transient faults* occur randomly. These occur when alpha, gamma particles or other cosmic radiation strike the logic circuits (transistors). When these high-energy external particles strike the transistors, they deposit or remove sufficient charge from the transistors that can turn on or off that particular transistor resulting in a logical error. The frequency of occurrence of transient faults is comparatively low when compared with other faults. However, diminishing chip size, low power supply, higher clock frequency, (factors necessary to improve the performance of the microprocessor) tend to increase the

occurrence of transient faults. In the future, portable and mobile computers are being expected to be widely used for important business and commercial transactions. Also the microprocessors that are used in space ships (they have a higher probability of occurrence of transient faults due to space exposure) need to have very high reliability. Therefore, it becomes necessary that these high performance, fast microprocessors are fault tolerant.

2.1 Fault tolerant techniques

It is important that we not only design the microprocessor chips correctly but also have some sort of fault tolerance inside the chip. Fault tolerance has two components, error detection and recovery. Fault tolerance in general is achieved by *Information redundancy, Time redundancy and Space redundancy* [6]. *Information redundancy* is used to preserve the integrity of the data by using error correction or detection encoding schemes. This requires additional overhead for encoding and checking logic. *Time redundancy* is achieved by executing the same instruction multiple times in the same functional unit at different times. This decreases the performance of the microprocessor considerably. Also permanent hardware faults cannot be detected since multiple executions of the same instruction in a faulty functional unit can still yield the same erroneous result. *Space redundancy* is achieved by executing the same instruction in multiple functional units at the same time. It requires additional functional units to achieve reliability through majority voting. This increases the sizes of the chip and thereby again reduces the performance.

2.2 Prior and Related work

Microprocessor architects have always concentrated the design effort on performance, giving little effort on the reliability factor. Only recently (as early as 1998) have the

architects realized the importance of a fully fault – tolerant microprocessor and have pioneered research in this direction. Prior to 1998, only traditional fault tolerant techniques have been used in the design of microprocessors [4]. These techniques can address reliability challenges but only at the cost of performance. Hardware redundancy was used to prevent logic errors. The Pentium Pro processor family has a provision to connect two processors in master/checker configuration to check for logical errors. The results of the processors were compared to detect any errors. If an error was detected, the instruction was re-executed. However this approach resulted in each instruction being executed twice (once in each processor) therefore resulting in 100% logic overhead thus reducing its overall performance greatly. There are other hardware redundancy techniques like duplicating select logic circuits or including error-checking logic in the functional units. IBM's G5 processor duplicates certain selected logic circuits. This approach does not reduce performance as significantly as in the Pentium Pro approach but it increases the circuit overhead by 35%, which is undesirable [5]. Recently many techniques have been proposed to recover from transient and other faults.

Rotenberg [7] suggests a time redundancy fault tolerant approach called Active *stream/Redundant stream Simultaneous Multithreading* (AR-SMT) in which two instructions streams, A- stream and R – stream are dynamically scheduled and executed. It makes use of software redundancy for error correction. The R-stream trails the A-stream. The results of the A- steam are stored in a delay buffer and which is used to compare with the results from the R-stream. The instruction is committed if both the A-stream and R-stream results match, else re-executed. This approach provides recovery from transient faults and some restricted permanent faults and also optimizes

performance. Karthik (Slipstream processors) [16] further improved upon this method in which the leading stream (A-stream) executes a reduced version of the original program. This is done by removing some of the ineffectual instructions that do not effect the program execution from the original code. The R-stream executes the full version while the A-stream executes the reduced version. This approach improves the performance of the overall program and also provides better fault tolerance by being able to recover from transient and permanent faults. One overhead in this technique is to correctly determine the ineffectual instructions, for the A-stream which proves to be a difficult task. The weakness with this approach is the design complexity involved in dynamic scheduling and also its incapability to detect permanent faults. Tamir and Tremblay [8] proposed using FIFO queues to check point after every few cycles of execution. Should a fault be detected, the cycles are recovered from the last checkpoint and re-executed. Franklin [9] proposed a time redundancy technique in which spare resources in the processor are used to duplicate the execution of the instructions. Ray [10] proposed a technique in which an instruction creates multiple redundant threads. The results from the duplicated threads are verified before the instruction is committed. Mendelson [11] proposed a technique to detect transient faults in the memory look up table. He suggests executing the instructions twice at the execution stage and then verifying before the instruction is committed. This approach requires minimum hardware however it does not cover all transient and permanent faults. Nicolaids [12] proposed a technique in which data is duplicated at the circuit level by introducing a delay element between the combinational logic and pipeline registers so that the data is latched twice at different times. Lai [13] proposed a technique called Ditto processor in which all instructions are

re-fetched and re-decoded to protect all pipeline stages from transient errors. It uses limited extra hardware and also optimizes performance. All the above proposed techniques blindly duplicate the execution of instructions at decode stage or at the commit stage. This results in many unnecessary instructions (when branch misprediction occurs) being duplicated that consume a lot of hardware resources. Also these techniques fail to recover from faults that occur at the front end of the pipeline. These techniques in general do not provide complete fault coverage and also degrade the overall performance of the microprocessor.

2.3 New Reliability Challenges:

With the number of transistors increasing rapidly inside a processor chip along with diminishing noise margins and fast clocks, it is necessary that the architects address these new reliability challenges, which are accompanied with finer features and denser technology in order make fast and cost effective processors.

Increased Complexity:

As the size of the chip keeps decreasing (.09 µm and lower) millions of transistors are being packed more densely. This makes it more difficult to ensure reliability and fault tolerance.

Noise-Related Faults:

Noise related faults occur due to crosstalk. Crosstalk occurs between circuits due to increased inductance and capacitance because of densely (closely) packed circuits. Noise related faults also occur due to low voltage and high frequency, which gives less noise margins.

Heat-Related Faults:

It is estimated that temperatures inside powerful microchips can rise up to 140 – 160 Fahrenheit [17]. These extreme-operating conditions inside the processor can result in electrometal migration or hot electrons resulting in permanent faults thus shortening the life of the transistors and circuits inside the processor.

Natural Radiation Interference:

It is estimated that on average, at least one cosmic ray hits a computer affecting its computation every ten years. This may not seem to be very important given the frequency but after applying a few rounds of Moore's law, we might end up rebooting our computers frequently because of some cosmic rays creating a fault in the registry file.

Natural radiation interference occurs with electronic circuits when gamma and alpha particles strike the transistors. Shielding using thick, dense materials can stop gamma rays but this incurs additional overhead. Alpha particles can be stopped using thin shields but it should be ensured that this shielding is very pure, otherwise the shielding itself would be an additional source of alpha particles [13].

2.4 Traditional Microprocessor Verification Methods:

To ensure high reliability, architects employ functional verification and electrical verification. Functional verification is done during the design stage. It is done to verify the correctness of the processor at the logic level. It means that given a set of valid inputs in the starting state, the next state is valid i.e. the transition is made to a valid state. It is difficult to check all transitions from one state to another state because of immense test space size. This is usually done with the help of simulation based testing. A model of the processor is simulated for numerous test cases and the results are compared with the

expected results. Unfortunately, many design errors slip through the simulation testing due to the immense size of the test space. Electrical verification is done at the design stage and at the fabrication state. It is done to verify the correctness of the logical implementation in silicon. The transistors and circuits are stress tested at extreme conditions like low voltage, high frequency, high temperature and slow process. Some of the implementation errors may slip through the electrical verification. For example, an infrequently used path may slip through the test.

3.0 Dynamic Verification [14, 15]:

Dynamic verification is used to supplement the traditional verification methods. It recovers functional and electrical faults including both permanent and transient faults without considerable loss in performance. The DIVA[1] checker architecture has two processors, the DIVA core processor and the DIVA checker processor. The DIVA core is the speculative Out-Of-Order processor while the DIVA checker is a simple reliable, fast In-Order execution processor. The core fetches, decodes, executes and stores the results in the Reorder Buffer (ROB). It does not commit instructions. The checker has a checker stage to verify the correctness of the core and the commit stage to write the results back to the memory. The DIVA also has a watchdog timer to detect faults that have stalled the processor. The *watchdog timer* is set to the maximum latency of any single instruction. If the timer expires due to a fatal error the instruction is re-executed. The DIVA technique makes two assumptions,

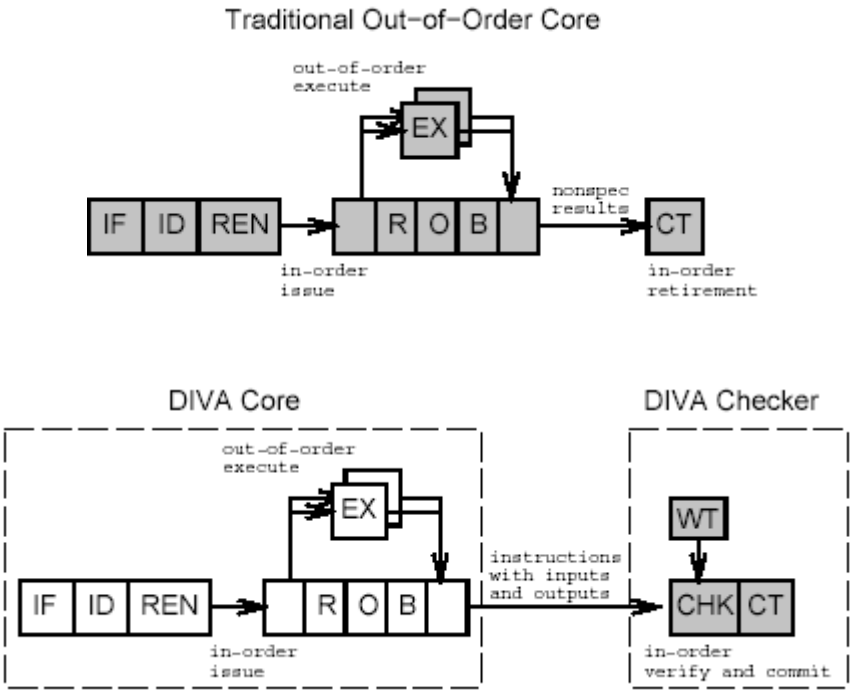
1. There will be no memory and register related faults. It is assumed that the registers and memory employ error detection and correction codes like Reed-Solomon code, Hamming code etc.
2. There will be no errors between the core and checker communication. Here also, it is assumed that appropriate coding techniques are employed to detect and correct errors that occur during communication.

3.1 DIVA Architecture

In the core processor, the instructions are issued in-order but executed out-of-order. Once the core processor executes each instruction, it sends the instruction, operands (inputs to the instruction) and the result to the checker in program order. In the checker, the

[1] Dynamic Instruction Verification Architecture

instructions are verified and committed in-order. The checker has two parallel and independent pipelines for verification, the *CHKcomp* and *CHKcomm* [14, 15]. CHKcomp pipeline ensures that the result computed by the functional unit is correct and the CHKcomm pipeline ensures that the register and memory communication with the instruction in the core processor occurred correctly i.e. when the operands are decoded and the respective register and memory locations are accessed. If both the pipelines report that there was no error, then the instruction is committed and written to the architectural state.



Figures Austin [14,15].

IF – Instruction fetch

ID - Instruction decode

REN – Register rename

ROB – Reorder buffer

EX – Execute stage

CT – Commit stage

DIVA Checker Architecture:

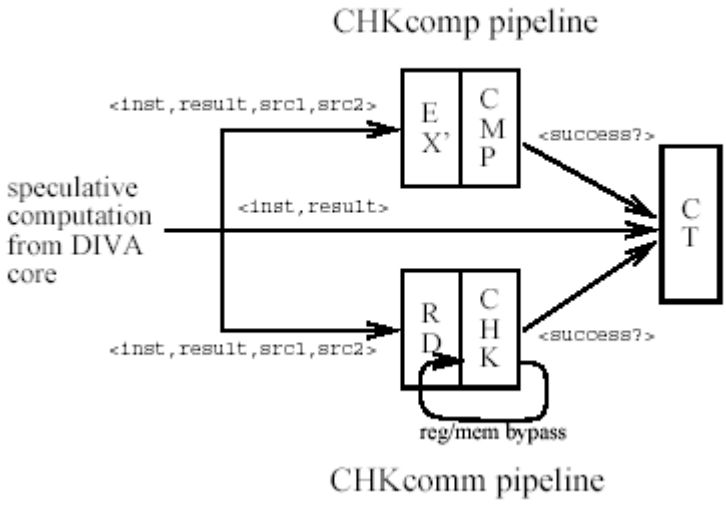


Figure Austin [14, 15].

EX – Execute stage

RD – Read stage

CMP – Compare

CHK – Check

CT – Commit Stage

CHKcomp Pipeline:

The CHKcomp pipeline has two stages, *EX stage* and *CMP stage*. In the EX stage, the checker independently re-executes the instruction using the operands (inputs) provided by the core processor. The EX stage takes the same number of cycles that the instruction

takes to execute in the respective functional unit in the core processor. In the CMP stage the computed result by the EX stage is compared with the stored result of the instruction provided by the core. The CMP stage takes one cycle to compare the results. If they differ, an exception is raised and the core processor result is replaced by the EX stage value.

CHKcomm Pipeline:

The CHKcomm pipeline has two stages, *RD stage* and *CHK stage*. In the RD stage, the checker reads the operand values from the memory and register. The RD stage takes one cycle to read the operands from the architectural state of the processor. In the CHK stage, the operand values in the RD are compared with the stored operand values provided by the core processor. The CHK stage takes one cycle to compare the operands, which it has read with that of the operands supplied by the core. Should they differ, the operand value in the RD stage is restored.

Since, the core delivers pre-computed operands to the checker pipeline, there are no instruction dependencies to resolve and hence the checker does not need extra cycles to resolve if there are any dependencies between the instructions, thus making the checker fast and simple. If both pipelines report correct operations then the instruction is committed. The whole process is repeated for every instruction. The success of this verification is dependent on the accurate and correct implementation of a robust checker. The checker should be simple, and latency insensitive. The checker only verifies the execution of the core and it does not speed up any computation, thus it is simple. Since the checker executes the instructions in program order and does not worry about resolving dependencies between instructions, it is latency insensitive i.e. it does not need

extra cycles to resolve dependencies. Should the checker malfunction, then the whole idea of dynamic verification is itself defeated thus making it infeasible. Since, this is a design only on paper, it would be interesting to actually construct a checker using VLSI technology and implement it.

Advantages of DIVA:

1. The checker does the electrical and functional verification and hence the core does not have the burden of this verification.
2. There is no possibility of transient faults and noise–related faults in the checker, since the checker uses large transistors and hence has higher voltage and frequency margins than the core. Thus making it more resistant to noise-related, heat-related and natural radiation interference.
3. The voltage and frequency margins can be relaxed in the core.
4. The core processor is much simplified by eliminating some of the infrequently used paths.
5. It has high error coverage since all the instructions are re-executed redundantly.
6. It provided better fault recovery than other techniques.

3.2 Discussion:

The motivation for this project was to find a cost effective solution to recover from transient and permanent faults with minimal impact on performance. One solution to recover from transient faults would be to use shielding and large transistors but this implies large surface and insulated regions. This project analyzes whether transient faults cause irreparable damage and if we can recover from these faults. It also discusses if the

checker can keep up with the speed of the core processor. Since the core delivers pre-computed inputs and results to the checker, instruction and data dependencies are eliminated in the checker. Also, the branch predictions are resolved in the core. These reasons make the checker simple and fast.

Limitations of DIVA:

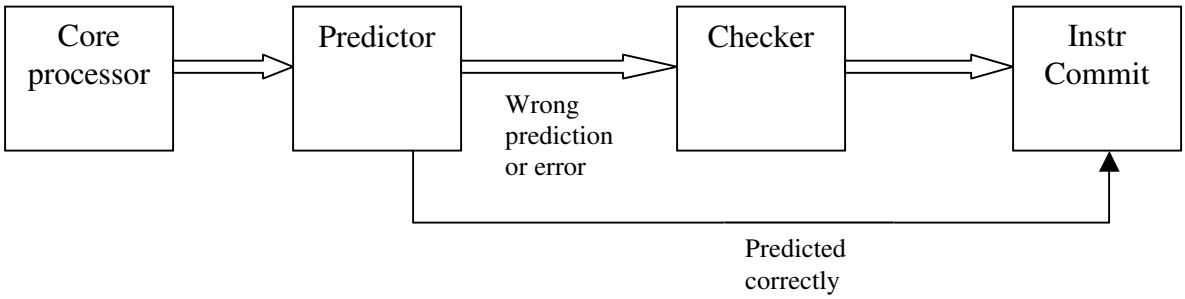
1. Not every instruction is in error. The fact that every instruction is re–executed redundantly by the checker may lead to some other higher costs.
2. Because of high redundancy, there is large power consumption, which is undesirable.
3. There is a possibility of bandwidth congestion for the checker, which in turn can lead to core slowdown.

Considering the costs involved in re-executing every single instruction, it becomes interesting to examine if we really need to re-execute every single instruction. By slightly modifying the design of DIVA architecture [14], we can introduce a data predictor that predicts the results of instructions. If the prediction is correct, then the instruction is not sent to the checker but can be committed directly by the core. Instead, if the prediction turns out to be wrong, then the instruction is sent to the checker for verification and then committed by the checker.

4.0 Modified DIVA with a predictor:

This system has a data predictor, which predicts the contents of each register. The data predictor filters the instructions from the core processor and then sends the remaining instructions to the checker for verification. Once the core processor finishes the execution of an instruction, it submits the results of its execution to the predictor. The data predictor

then compares this result with its own predicted value. If both the values match, then the instruction is not sent to the checker but can be directly committed. If a mismatch occurs between the values, then the instruction is sent to the checker for verification. Once the checker verifies the sanity of the instruction, it is committed. The performance of this system depends on the prediction. The better and more accurate the data predictor, the higher the performance is gained. So, it is really important that the data predictor works as accurately as possible for this system to succeed. The data predictor takes one cycle to predict the value and then another cycle to compare the values.



The advantage of having a data predictor before the checker is that those instructions whose values have been predicted correctly need not spend all those extra cycles in the checker (minimum extra of 5 cycles for add and higher for multiplication and other instructions) but instead directly go to the commit stage. And also the fact that fewer cycles translate to less power consumption and adds another advantage to this system. Working through the following example illustrates this,

Where [X] indicates register X being used.

		RegA	RegB	Dest Reg
1.	*add*	[7]	[2]	[4]
2.	*add*	[4]	[6]	[3]

3. *mult* [4] [5] [2]
4. *mult* [5] [1] [6]
5. *add* [1] [2] [3]
6. *halt*

Add takes two cycles and *Mult* takes five cycles to execute.

Consider the fact that the above code is executed without any errors.

Case 1: If this code is executed on an out-of order execution processor, it would take 17 cycles to give the final output.

Case 2: If the same code is executed on the same out-of-order processor but with the checker, it takes 27 cycles to give the final output.

Case 3: Again consider the same code is executed on the same out-of-order processor with the checker and a data predictor[1]. Let us assume, that our data predictor has correctly predicted the results of the 3rd and 5th instructions. Then these two instructions do not enter the checker stage but the remaining instructions enter the checker stage. In this case, the total number of cycles required to output the final result would be 22cycles. It should be noted here that number of cycles varies depending upon the accuracy of the predictor. It is important that we use a good predictor.

In the above scenario, where we have considered that there are no errors, the out-of-order, which takes 17 cycles to complete the execution, is clearly the winner. But this error-free scenario is unlikely and unrealistic in a real world situation.

Now, consider the situation where we introduce some errors.

This is the initial content of the registers before program execution.

[1] Predictor values have been hard coded. Different dynamic predictors can be used while actually implementing.

Register[0] = 0; Register[1] = 1; Register[2] = 2; Register[3] = 3; Register[4] = 4; Register[5] = 5; Register[6] = 6; Register[7] = 7.

This is the final content of the registers after program execution.

Register[0] = 0; Register[1] = 1; Register[2] = 45; Register[3] = 46; Register[4] = 9; Register[5] = 5; Register[6] = 5; Register[7] = 7.

Let us assume that some cosmic ray has hit our processor and altered the values of the results of 1st and 4th instructions to 50 and 100 respectively.

Case 1: In an out-of-order processor, it would take the same 17 cycles to execute but would produce wrong results since the instructions were not verified. So, the results would be a waste.

Case 2: In an out-of order processor with a checker, it would take 27 cycles to execute but since the checker detects an error, it replaces the results with the correct value and hence produces the correct output. Although, it takes more cycles than the out-of-order, it is worth it since we get the correct output.

Case 3: Now let us see if we can improve upon the performance of the checker with the addition of a predictor. Making the same assumption that the predictor predicts the results of the 3rd and 5th instructions correctly, it takes 22 cycles. Certainly, better than the checker without any prediction and of course it also produces the correct output. We see that this definitely improves upon DIVA in terms of performance.

5.0 Simulation Setup:

The core processor has three ALU units, three Mult units, and three load/store units. The latencies for the ALU units is two cycles, five cycles for Mult units and two cycles for load/store units. The instructions are issued in-order, executed out-of order and committed in-order. For this simulation, there is no cache, to reduce the complexity. The instructions are directly read from the memory. The Checker is not pipelined and has a single ALU unit, single Mult unit and a single load/store unit. In the checker the instructions are executed in program order. Only the add and mult instructions are studied in this simulation, although this can be easily extended to include all other instructions. A static predictor is used in the simulation although a dynamic predictor would be more realistic. Designing a suitable dynamic predictor would be complex and is beyond the scope of this project given the time constraints. One of the areas for future development in this project is to evaluate and design a suitable predictor.

Error injection:

Errors are injected into the simulation by changing the register contents of the source operands and also the results of the values in the functional units. Changing these data values should model the faults induced due to noise, heat and natural radiation. Also it should model the faults due to permanent errors that can occur in the functional units.

Predictor Cases:

Case 1: The result of the instruction is in error but the prediction predicts the correct result. In this case even though the predictor has the correct result, we do not know for sure that the predictor is correct. So, we treat this as an error and the instruction is sent to the checker.

Case 2: The result of the instruction is in error and the predictor also predicts the wrong result. In this case since the results do not match, the instruction is in error. It should be noted here that the erroneous result of the instruction and the wrong prediction might match in some cases. But this is a very unlikely situation and very rarely happens.

Case 3: The result of the instruction is correct but the prediction is wrong. In this case the results do not match. We do not know for sure that the result of the instruction is correct because we have not verified the result. Hence, the instruction is sent to the checker even though it has the correct result.

Case 4: The result of the instruction is not in error and the predictor also predicts the correct result. Here both the results match. In this case the instruction is not sent to the checker.

5.1 Simulation Results:

Error free cases (without any errors induced). The numbers represent the numbers of cycles taken by each different processor to execute the respective cases[1].

	Case1	Case2	Case3	Case4	Case5
Out-of-order	17	11	15	20	11
DIVA	27	16	20	39	16
DIVA with predictor[2]	22	16	16	25	12

[1] For Cases refer to the index

[2] Under the assumption that two instructions are predicted correctly.

Cases with errors induced

	Case 1	Case 2	Case 3	Case 4	Case 5	Error corrected??
Out-of-order	17	11	15	20	11	NO in all cases
DIVA	27	16	20	39	16	YES in all cases
DIVA with predictor	22	16	16	25	12	YES in cases 1,2,4,5 NO in case 3

Design Tradeoffs:

Introducing a predictor can be an overhead to the already complex system, but this tradeoff can be minimized or optimized by taking into account the following two reasons,

- A good predictor can offset this tradeoff when it has highly accurate predictions and hence fewer instructions enter the checker and thereby reduce the number of cycles and overhead of the checker.
- Considering the complexity involved in the core microprocessor and the checker system implementation, the introduction of the predictor into the system is negligible.

It should be accepted that the error detection with the introduction of the predictor is slightly vulnerable, because now not every instruction is verified by the checker. It should be noted that the only type of errors that may slip through are those where the erroneous result of the instruction and the wrong prediction match. But this is a very

unlikely case to occur given the probability of such an occurrence. These are the risks, which are worth taking given the performance gain, which we achieve.

6.0 Conclusion:

Functional verification and Electrical verification can detect most of the design errors but non – trivial faults and transient faults may skip through the verification process. Also, reliability poses a challenge in deep submicron fabrication technologies due to natural radiation interference and noise related faults. Dynamic verification [13,14] promises to counter these new reliability challenges. Dynamic verification also relieves the burden of functional and electrical verification from the core and instead the checker takes on this job of verification.

6.1 Suggestions and Improvement:

The DIVA technique can be extended to provide much more reliability by having three checkers instead of the single checker proposed originally. The three checkers can independently execute each instruction as the execution is completed and compare the results. If the results of at least two checkers match, then the instruction is committed. This may incur some overhead. So, a balance has to be made between accuracy and performance.

Future work:

Much work needs to be done to analyze and design a suitable data predictor. There are various predictors like stride-based predictor, repeat based predictor, which need to be tested and integrated with the system. As this is a concept still on paper, it would be interesting to actually design such a hardware system using very large scale integrated (VLSI) implementation.

References

[1] M. Blum and H. Wasserman, "Reflections on the Pentium Division Bug", Intel Corporation, Oct.1997.

[2] Chris Weaver and Todd Austin, "A Fault Tolerant Approach to microprocessor Design", Dependable Systems and Networks, July 2001.

[3] H. Al-Asaa, B.T.Murray and J.P. Hayes," Online BIST for Embedded Systems," IEEE Design and Test of Computers, pp.17-24, October – December 1998.

[4] A. Avizienis and Y.He, "Microprocessor entomology: a taxonomy of design faults in COTS miroprocessors", Dependable Computing for Critical Applications 7,1999,pp 3-23.

[5] Y. He et .al , "Assessment of the applicability of COTS microprocessors in high confidence in computing systems: a case study," Proceedings International Conference on Dependable Systems and Networks, 2000.

[6] Dr. Lois Hawkes, "Fault Tolerance and Reliability",
http://www.cs.fsu.edu/~hawkes/FT/ft_chap1.pdf, accessed on 6/1/2004.

[7] E.Rotenberg. "AR-SMT: A microacrchitectural approach to fault tolerance in microprocessors", In proceedings of the 29th Fault-Tolerant Computing Symposium, June 1999.

[8] Y.Tamir and M.Tremblay," High-performance Fault tolerant VLSI systems using micro rollback", IEEE Transactions on Computers, pages 47-48, Jan 1998.

[9] M. Franklin," A study of Time Redundant Fault Tolerance for Superscalar processors", Digest of papers in the 25th International Symposium on Fault-Tolerant Computing, Aug 1995,pp 207-215.

[10] Joydeep Ray, “Dual use of Superscalar Datapath for Transient-Fault Detection and Recovery”, In proceeding of 34th Microacrchitecture, December, 2001.

[11] A.Mendelson and N.Suri, “ Designing High-Performance and Reliable Superscalar Architectures”, Proc. Conference on Dependable Systems and Networks, 2000.

[12] M. Nicolaids, “ Time redundancy based soft error tolerance to rescue nanometer technologies”, Proceedings of 17th IEEE VLSI Test Symposium, 1999, pp. 86-94.

[13] Shih-Chang Lai, Shih-Lien Lu, Jih-kwon Peir, “ Ditto Processor”, International Conference on Dependable Systems and Networks (DSN'02) June 23 - 26, 2002 Washington, DC, USA. p. 525.

[14] T. Austin, “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design”, In proceeding of 32nd Microacrchitecture, November, 1999.

[15] T. Austin, “ DIVA: A Dynamic Approach to Microprocessor Verification” , The Journal of Instruction-Level Parallelism Volume 2, 2000.

[16] Karthik Sundaramoorthy, Zach Purser, Eric Rotenberg, “Slipstream Processors: Improving both Performance and Fault Tolerance”, 9th Int’l Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[17] http://www.cnn.com/2004/Tech/biztech/taiwan.heat.riu/index.html, cnn news, accessed on 6/1//2004.

Appendix 1: Explanation of cases

Case 1:

```
add 7 2 4
add 4 6 3
mult 4 5 2
mult 5 1 6
add 1 2 3
halt
```

Case 2:

```
add 7 1 4
add 3 4 2
add 5 6 7
halt
```

Case3:

```
add 2 2 3
add 4 3 5
add 4 5 6
add 3 5 1
halt
```

Case 4:

```
mult 2 2 3
mult 4 5 2
add 5 6 7
mult 2 3 1
add 3 6 2
add 6 5 4
mult 3 7 6
halt
```

Case5:

```
add 7 1 4
add 3 4 2
add 1 3 2
halt
```

Appendix B : Code for out-of-order execution processor.

```
/* Header Files */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>

#define NUMMEMORY 65536 /* Total of 2048 instructions of 32 bits */
#define ADD 0
#define NAND 1
#define LW 2
#define SW 3
#define BEQ 4
#define MULT 5
#define HALT 6
#define NOOP 7

#define NOOPINSTRUCTION 0x1c00000

#define ADD1 10
#define ADD2 20
#define ADD3 30
#define MUL1 40
#define MUL2 50
#define MUL3 60
#define LW1 70
#define LW2 80
#define LW3 90

/* Structures for defining the various pipelines in the processor */

typedef struct IFISStruct
{
 int instr;
}IFISType;

typedef struct ISEXStruct
{
 int instr;
}ISEXType;

typedef struct EXWBStruct
{
 int instr;
}EXWBType;

typedef struct WBENDStruct
{
 int instr;
}WBENDType;

typedef struct RESStruct
{
```

```
 int busy;
 int QJ,QK;
 int VJ,VK;
 int AluRe;
 int RegA,RegB,Dest;
}RESType;

typedef struct stateStruct
{
 int pc;
 int cycles;
 int vj1,vj2,vj3,vj4,vj5,vj6;
 int vk1,vk2,vk3,vk4,vk5,vk6;
 int wb1,wb2,wb3,wb4,wb5,wb6;
 int chk[2048];
 IFISType IFIS;
 ISEXType ISEX;
 EXWBType EXWB;
 WBENDType WBEND;
}stateType;

/* main function */
int main(int argc, char *argv[])
{
 FILE *inFilePtr;
 char *inputFile;
 stateType state,newState,current;
 RESType Add1,Add2,Add3,Mul1,Mul2,Mul3;
 int instrMem[NUMMEMORY],dataMem[NUMMEMORY];
 int inst = 0;
 int FP[8];
 int Register[8];
 int ARegister[8];
 int i;

 int p1,p2,p3,p4,p5,p6;
 int ct[2048];
 int ft[2048];
 int is[2048];
 int unit[2048];
 int regA[2048];
 int regB[2048];
 int exs[2048];
 int exe[2048];
 int wr[2048];
 int cmt[2048];
 int cmtchk[2048];

 int RA[2048];
 int RB[2048];
 int DS[2048];
 int RS[2048];
 int ins[2048];

 int c1=2,c2=2,c3=2;
 int c4=5,c5=5,c6=5;
 int k = 1;
```

```
/* Initialising the components in the reservation station */

  state.pc = 0;
  state.cycles = 1;
  state.IFIS.instr = NOOPINSTRUCTION;
  state.ISEX.instr = NOOPINSTRUCTION;
  state.EXWB.instr = NOOPINSTRUCTION;
  state.WBEND.instr = NOOPINSTRUCTION;

  Add1.busy = 0;
  Add1.QJ = -1;
  Add1.QK = -1;
  Add1.VJ = -1;
  Add1.VK = -1;

  Add2.busy = 0;
  Add2.QJ = -1;
  Add2.QK = -1;
  Add2.VJ = -1;
  Add2.VK = -1;

  Add3.busy = 0;
  Add3.QJ = -1;
  Add3.QK = -1;
  Add3.VJ = -1;
  Add3.VK = -1;

  Mul1.busy = 0;
  Mul1.QJ = -1;
  Mul1.QK = -1;
  Mul1.VJ = -1;
  Mul1.VK = -1;

  Mul2.busy = 0;
  Mul2.QJ = -1;
  Mul2.QK = -1;
  Mul2.VJ = -1;
  Mul2.VK = -1;

  Mul3.busy = 0;
  Mul3.QJ = -1;
  Mul3.QK = -1;
  Mul3.VJ = -1;
  Mul3.VK = -1;

 current.vj1 = 0;
 current.vk1 = 0;
 current.vj2 = 0;
 current.vk2 = 0;
 current.vj3 = 0;
 current.vk3 = 0;
 current.vj4 = 0;
 current.vk4 = 0;
 current.vj5 = 0;
 current.vk5 = 0;
 current.vj6 = 0;
 current.vk6 = 0;
```

```
 current.wb1 = 0;
 current.wb2 = 0;
 current.wb3 = 0;
 current.wb4 = 0;
 current.wb5 = 0;
 current.wb6 = 0;


  for (i=0;i<8;i++)
  {
   FP[i] = 0;
   Register[i] = i;
   ARegister[i] = i;
  }

  for(i=0;i<2048;i++)
  current.chk[i] = 0;


 /*----------Checking for Input arguments to the Simulator-----------*/
 if(argc!=2)
 {
  printf("\nError in usage:\n %s <Specify Input File>\n",argv[0]);
  exit(1);
 }

 inputFile = argv[1];
 inFilePtr = fopen(inputFile,"r");

 /*------------------Checking for Input Valid File----------------------
--*/
 if(inFilePtr == NULL)
 {
  printf("\n Failed to open %s, Specify correct File
name\n",inputFile);
  exit(1);
 }

 /*-----------------Reading instruction from file to memory--------*/
 while(fscanf(inFilePtr,"%d",&instrMem[inst])!=EOF)
 {
  dataMem[inst] = instrMem[inst];
  inst++;
 }
 printf("\n Number of Instructions read %d\n\n",inst);

 while(1)
 {
   // checking for the HALT instruction //
  if(Opcode(state.WBEND.instr) == HALT)


  {

  // printing the output of the simulation //
   if(cmtchk[inst - 1] == 1)
```

```
   {
    printf("\n
OUTPUT \n");
    printf("                                                                -
---------\n");
    printf("\nProgram Counter   Opcode  Inst Fetched  Inst Issued  Unit
Reg A  Reg B  ExStart  ExEnd  WriteBack    Commit\n");
    printf("-----------------------------------------------------------
-------------------------------------------------\n");

   for (i=0;i<inst;i++)
    {

printInstruction(i,instrMem[i],ct[i],ft[i],is[i+1],unit[i+1],regA[i+1],
regB[i+1],exs[i+1],exe[i+1],wr[i+1],cmt[i+1]);
    }


    for(i=0;i<8;i++)
    printf("\n Register[%d] = %d\n",i,ARegister[i]);
    printf("\n Total No of Cycles Executed : %d\n",state.cycles - 1);
    printf("\n");
    exit(0);
   }
   else
   {
    newState.pc = state.pc - 1;
      }
  }



  newState = state;
  newState.vj1 = current.vj1;
  newState.vk1 = current.vk1;
  newState.wb1 = current.wb1;
  newState.vj2 = current.vj2;
  newState.vk2 = current.vk2;
  newState.wb2 = current.wb2;
  newState.vj3 = current.vj3;
  newState.vk3 = current.vk3;
  newState.wb3 = current.wb3;
  newState.vj4 = current.vj4;
  newState.vk4 = current.vk4;
  newState.wb4 = current.wb4;
  newState.vj5 = current.vj5;
  newState.vk5 = current.vk5;
  newState.wb5 = current.wb5;
  newState.vj6 = current.vj6;
  newState.vk6 = current.vk6;
  newState.wb6 = current.wb6;

  for(i=0;i<inst;i++)
  newState.chk[i] = current.chk[i];

  newState.cycles++;
```

```
  /*----------------Instruction Fetch Stage-----------------------*/
  if (instrMem[state.pc] !=0)
  {
   newState.IFIS.instr = instrMem[state.pc];
//   printf("\n opcode = %d\n",Opcode(newState.IFIS.instr));
   ct[newState.pc] = state.pc;
   ft[newState.pc] = state.cycles;
   newState.pc++;
  }

 /*----------------Instruction Issue Stage-----------------------*/
 if (newState.IFIS.instr !=0)
 {
  newState.ISEX.instr = state.IFIS.instr;
  switch(Opcode(newState.ISEX.instr))
  {
   case ADD:
   {
            // checking if the ADD station is busy or not //

    if((Add1.busy == 1)&&(Add2.busy == 1)&&(Add3.busy == 1))
    {
     newState.pc = state.pc - 1;
    }
            // checking if the ADD1 station is busy or not //

    if(Add1.busy == 0)
    {
     Add1.busy = 1;
     Add1.RegA = field0(newState.ISEX.instr);
     Add1.RegB = field1(newState.ISEX.instr);
     Add1.Dest = field2(newState.ISEX.instr);
     is[state.pc] = state.cycles;
     p1 = state.pc;

     ins[p1] = newState.ISEX.instr;
     DS[p1] = Add1.Dest;

     unit[p1] = ADD1;
     FP[Add1.Dest] = ADD1;

     if(FP[Add1.RegA] == 0)
     {
      Add1.QJ = 0;
      Add1.VJ = Register[Add1.RegA];
      RA[p1] = Register[Add1.RegA];
      current.vj1 = 1;
      regA[p1] = state.cycles;
     }
     else
     {
      Add1.QJ = FP[Add1.RegA];
     }

     if(FP[Add1.RegB] == 0)
     {
      Add1.QK =  0;
```

```
  Add1.VK  = Register[Add1.RegB];
  RB[p1] = Add1.VK;
  current.vk1 = 1;
  regB[p1] = state.cycles;
 }
 else
 {
  Add1.QK = FP[Add1.RegB];
 }
}

    // checking if the ADD2 station is busy or not //


else if(Add2.busy == 0)
{
 Add2.busy = 1;
 Add2.RegA = field0(newState.ISEX.instr);
 Add2.RegB = field1(newState.ISEX.instr);
 Add2.Dest = field2(newState.ISEX.instr);

 is[state.pc] = state.cycles;
 p2 = state.pc;
 unit[p2] = ADD2;

 ins[p2] = newState.ISEX.instr;
 DS[p2] = Add2.Dest;


 FP[Add2.Dest] = ADD2;

 if(FP[Add2.RegA] == 0)
 {
  Add2.QJ = 0;
  Add2.VJ = Register[Add2.RegA];
  RA[p2] = Add2.VJ;
  regA[p2] = state.cycles;

  current.vj2 = 1;
 }
 else
 {
  Add2.QJ = FP[Add2.RegA];
 }

 if(FP[Add2.RegB] == 0)
 {
  Add2.QK = 0;
  Add2.VK = Register[Add2.RegB];
  RB[p2] = Add2.VK;
  current.vk2 = 1;
  regB[p2] = state.cycles;
 }
 else
 {
  Add2.QK = FP[Add2.RegB];
 }
```

```
}
      // checking if the ADD3 station is busy or not //


 else if(Add3.busy == 0)
 {
  Add3.busy = 1;
  Add3.RegA = field0(newState.ISEX.instr);
  Add3.RegB = field1(newState.ISEX.instr);
  Add3.Dest = field2(newState.ISEX.instr);

  is[state.pc] = state.cycles;
  p3 = state.pc;
  unit[p3] = ADD3;

  ins[p3] = newState.ISEX.instr;
  DS[p3] = Add3.Dest;


  FP[Add3.Dest] = ADD3;

  if(FP[Add3.RegA] == 0)
  {
   Add3.QJ = 0;
   Add3.VJ = Register[Add3.RegA];
   RA[p3] = Add3.VJ;
   current.vj3 = 1;
   regA[p3] = state.cycles;
  }
  else
  {
   Add3.QJ = FP[Add3.RegA];
  }

  if(FP[Add3.RegB] == 0)
  {
   Add3.QK = 0;
   Add3.VK = Register[Add3.RegB];
   RB[p3] = Add3.VK;
   regB[p3] = state.cycles;
   current.vk3 = 1;
  }
  else
  {
   Add3.QK = FP[Add3.RegB];
  }
 }
 break;
}// end of ADD


           // if the instruction is MULT //
 case MULT:
{
 if((Mul1.busy == 1)&&(Mul2.busy == 1)&&(Mul3.busy == 1))
 {
  newState.pc = state.pc - 1;
```

```
}
        // checking if the MULT1 station is busy or not //

if(Mul1.busy == 0)
{
 Mul1.busy = 1;
 Mul1.RegA = field0(newState.ISEX.instr);
 Mul1.RegB = field1(newState.ISEX.instr);
 Mul1.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p4 = state.pc;
 unit[p4] = MUL1;
 FP[Mul1.Dest] = MUL1;

 ins[p4] = newState.ISEX.instr;
 DS[p4] = Mul1.Dest;



 if(FP[Mul1.RegA] == 0)
 {
  Mul1.QJ = 0;
  Mul1.VJ = Register[Mul1.RegA];
  RA[p4] = Mul1.VJ;
  current.vj4 = 1;
  regA[p4] = state.cycles;
 }
 else
 {
  Mul1.QJ = FP[Mul1.RegA];
 }

 if(FP[Mul1.RegB] == 0)
 {
  Mul1.QK = 0;
  Mul1.VK = Register[Mul1.RegB];
  RB[p4] = Mul1.VK;
  current.vk4 = 1;
  regB[p4] = state.cycles;
 }
 else
 {
  Mul1.QK =  FP[Mul1.RegB];
 }
}
        // checking if the MULT2 station is busy or not //

else if(Mul2.busy == 0)
{
 Mul2.busy = 1;
 Mul2.RegA = field0(newState.ISEX.instr);
 Mul2.RegB = field1(newState.ISEX.instr);
 Mul2.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p5 = state.pc;
 unit[p5] = MUL2;
 FP[Mul2.Dest] = MUL2;
```

```
 ins[p5] = newState.ISEX.instr;
 DS[p5] = Mul2.Dest;


 if(FP[Mul2.RegA] == 0)
 {
  Mul2.QJ = 0;
  Mul2.VJ = Register[Mul2.RegA];
  RA[p5] = Mul2.VJ;
  current.vj5 = 1;
  regA[p5] = state.cycles;
 }
 else
 {
  Mul2.QJ = FP[Mul2.RegA];
 }

 if(FP[Mul2.RegB] == 0)
 {
  Mul2.QK = 0;
  Mul2.VK = Register[Mul2.RegB];
  RB[p5] = Mul2.VK;
  current.vk5 = 1;
  regB[p5] = state.cycles;
 }
 else
 {
  Mul2.QK =  FP[Mul2.RegB];
 }
}

        // checking if the MULT3 station is busy or not //

else if(Mul3.busy == 0)
{
 Mul3.busy = 1;
 Mul3.RegA = field0(newState.ISEX.instr);
 Mul3.RegB = field1(newState.ISEX.instr);
 Mul3.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p6 = state.pc;
 unit[p6] = MUL3;
 FP[Mul3.Dest] = MUL3;

 ins[p6] = newState.ISEX.instr;
 DS[p6] = Mul3.Dest;

 if(FP[Mul3.RegA] == 0)
 {
  Mul3.QJ = 0;
  Mul3.VJ = Register[Mul3.RegA];
  RA[p6] = Mul3.VJ;
  current.vj6 = 1;
  regA[p6] = state.cycles;
 }
 else
 {
```

```
     Mul3.QJ = FP[Mul3.RegA];
    }

    if(FP[Mul3.RegB] == 0)
    {
     Mul3.QK = 0;
     Mul3.VK = Register[Mul3.RegB];
     RB[p6] = Mul3.VK;
     current.vk6 = 1;
     regB[p6] = state.cycles;
    }
    else
    {
     Mul3.QK = FP[Mul3.RegB];
    }
   }
   break;
  }// end of MUL

           // if the instruction is HALT //


 case HALT:
 {
  is[state.pc] = state.cycles;
  break;
 }
}// end of switch

  if(Add1.busy == 1)
  {
   if(Add1.QJ!=0)

   {
    if(FP[Add1.RegA] == 0)
    {
     Add1.VJ = Register[Add1.RegA];
     RA[p1] = Add1.VJ;
     regA[p1] = state.cycles;
     current.vj1 = 1;
     Add1.QJ = 0;
    }
   }
   if(Add1.QK != 0)
    {
    if(FP[Add1.RegB] == 0)
    {
     Add1.VK = Register[Add1.RegB];
     RB[p1] = Add1.VK;
     regB[p1] = state.cycles;
     current.vk1 = 1;
     Add1.QK = 0;
    }
   }
  }

  if(Add2.busy == 1)
```

```
{
 if(Add2.QJ != 0)
 {
  if(FP[Add2.RegA] == 0)
  {
   Add2.VJ = Register[Add2.RegA];
   RA[p2] = Add2.VJ;
   regA[p2] = state.cycles;
   current.vj2 = 1;
   Add2.QJ = 0;
  }
 }
 if(Add2.QK != 0)
 {
  if(FP[Add2.RegB] == 0)
  {
   Add2.VK = Register[Add2.RegB];
   RB[p2] = Add2.VK;
   regB[p2] = state.cycles;
   current.vk2 = 1;
   Add2.QK = 0;
  }
 }
}

if(Add3.busy == 1)
{
 if(Add3.QJ != 0)
 {
  if(FP[Add3.RegA] == 0)
  {
   Add3.VJ = Register[Add3.RegA];
   RA[p3] = Add3.VJ;
   regA[p3] = state.cycles;
   current.vj3 = 1;
   Add3.QJ = 0;
  }
 }
 if(Add3.QK != 0)
 {
  if(FP[Add3.RegB] == 0)
  {
   Add3.VK = Register[Add3.RegB];
   RB[p3] = Add3.VK;
   regB[p3] = state.cycles;
   current.vk3 = 1;
   Add3.QK = 0;
  }
 }
}

if(Mul1.busy == 1)
{
 if(Mul1.QJ != 0)
 {
  if(FP[Mul1.RegA] == 0)
  {
```

```
   Mul1.VJ = Register[Mul1.RegA];
   RA[p4] = Mul1.VJ;
   regA[p4] = state.cycles;
   current.vj4 = 1;
   Mul1.QJ = 0;
  }
 }
 if(Mul1.QK != 0)
 {
  if(FP[Mul1.RegB] == 0)
  {
   Mul1.VK = Register[Mul1.RegB];
   RB[p4] = Mul1.VK;
   regB[p4] = state.cycles;
   current.vk4 = 1;
   Mul1.QK = 0;
  }
 }
}

if(Mul2.busy == 1)
{
 if(Mul2.QJ != 0)
 {
  if(FP[Mul2.RegA] == 0)
  {
   Mul2.VJ = Register[Mul2.RegA];
   RA[p5] = Mul2.VJ;
   regA[p5] = state.cycles;
   current.vj5 = 1;
   Mul2.QJ = 0;
  }
 }
 if(Mul2.QK != 0)
 {
  if(FP[Mul2.RegB] == 0)
  {
   Mul2.VK = Register[Mul2.RegB];
   RB[p5] = Mul2.VK;
   regB[p5] = state.cycles;
   current.vk5 = 1;
   Mul2.QK = 0;
  }
 }
}

if(Mul3.busy == 1)
{
 if(Mul3.QJ != 0)
 {
  if(FP[Mul3.RegA] == 0)
  {
   Mul3.VJ = Register[Mul3.RegA];
   RA[p6] = Mul3.VJ;
   regA[p6] = state.cycles;
   current.vj6 = 1;
   Mul3.QJ = 0;
```

```
    }
   }
   if(Mul3.QK != 0)
   {
    if(FP[Mul3.RegB] == 0)
    {
     Mul3.VK = Register[Mul3.RegB];
     RB[p6] = Mul3.VK;
     regB[p6] = state.cycles;
     current.vk6 = 1;
     Mul3.QK = 0;
    }
   }
  }
}

/*----------------Instruction Execute Stage-----------------------*/
if (newState.ISEX.instr !=0)
{
 newState.EXWB.instr = state.ISEX.instr;

 if((newState.vj1 == 1)&&(newState.vk1 == 1))
 {
  exs[p1] = state.cycles - 1;
  c1--;
  if(c1 == 0)
  {
   current.vj1 = 0;
   current.vk1 = 0;
   exe[p1] = state.cycles;
   Register[Add1.Dest] = Add1.VJ + Add1.VK;
   Add1.AluRe = Add1.VJ + Add1.VK;
   current.wb1 = 1;
   c1 = 2;
  }
 }

 if((newState.vj2 == 1)&&(newState.vk2 == 1))
 {
  exs[p2] = state.cycles - 1;
  c2--;
  if(c2 == 0)
  {
   current.vj2 = 0;
   current.vk2 = 0;
   exe[p2] = state.cycles;
   Register[Add2.Dest] = Add2.VJ + Add2.VK;
   Add2.AluRe = Add2.VJ + Add2.VK;
   current.wb2 = 1;
   c2 = 2;
  }
 }

 if((newState.vj3 == 1)&&(newState.vk3 == 1))
 {
  exs[p3] = state.cycles - 1;
  c3--;
```

```
 if(c3 == 0)
 {
  current.vj3 = 0;
  current.vk3 = 0;
  exe[p3] = state.cycles;
  Register[Add3.Dest] = Add3.VJ + Add3.VK;
  Add3.AluRe = Add3.VJ + Add3.VK;
  current.wb3 = 1;
  c3 = 2;
 }
}

if((newState.vj4 == 1)&&(newState.vk4 == 1))
{
 exs[p4] = state.cycles - 4;
 c4--;
 if(c4 == 0)
 {
  current.vj4 = 0;
  current.vk4 = 0;
  exe[p4] = state.cycles;
  Register[Mul1.Dest] = Mul1.VJ * Mul1.VK;
  Mul1.AluRe = Mul1.VJ * Mul1.VK;
  current.wb4 = 1;
  c4 = 5;
 }
}

if((newState.vj5 == 1)&&(newState.vk5 == 1))
{
 exs[p5] = state.cycles - 4;
 c5--;
 if(c5 == 0)
 {
  current.vj5 = 0;
  current.vk5 = 0;
  exe[p5] = state.cycles;
  Register[Mul2.Dest] = Mul2.VJ * Mul2.VK;
  Mul2.AluRe = Mul2.VJ * Mul2.VK;
  current.wb5 = 1;
  c5 = 5;
 }
}

if((newState.vj6 == 1)&&(newState.vk6 == 1))
{
 exs[p6] = state.cycles - 4;
 c6--;
 if(c6 == 0)
 {
  current.vj6 = 0;
  current.vk6 = 0;
  exe[p6] = state.cycles;
  Register[Mul3.Dest] = Mul3.VJ * Mul3.VK;
  Mul3.AluRe = Mul3.VJ * Mul3.VK;
  current.wb6 = 1;
  c6 = 5;
```

```
  }
 }

}
/*----------------Instruction Write Stage-----------------------*/
if (newState.EXWB.instr !=0)
{
 newState.WBEND.instr = state.EXWB.instr;

 if(newState.wb1 == 1)
 {
  current.wb1 = 0;
  Add1.busy = 0;
  FP[Add1.Dest] = 0;
  wr[p1] = state.cycles;
  current.chk[p1] = 1;
 }

 if(newState.wb2 == 1)
 {
  current.wb2 = 0;
  Add2.busy = 0;
  FP[Add2.Dest] = 0;
  wr[p2] = state.cycles;
  current.chk[p2] = 1;
 }

 if(newState.wb3 == 1)
 {
  current.wb3 = 0;
  Add3.busy = 0;
  FP[Add3.Dest] = 0;
  wr[p3] = state.cycles;
  current.chk[p3] = 1;
 }

 if(newState.wb4 == 1)
 {
  current.wb4 = 0;
  Mul1.busy = 0;
  FP[Mul1.Dest] = 0;
  wr[p4] = state.cycles;
  current.chk[p4] = 1;
 }

 if(newState.wb5 == 1)
 {
  current.wb5 = 0;
  Mul2.busy = 0;
  FP[Mul2.Dest] = 0;
  wr[p5] = state.cycles;
  current.chk[p5] = 1;
 }

 if(newState.wb6 == 1)
 {
  current.wb6 = 0;
```

```
   Mul3.busy = 0;
   FP[Mul3.Dest] = 0;
   wr[p6] = state.cycles;
   current.chk[p6] = 1;
  }
 }


  state = newState;
 }// end of while
}// end of main

// Function to extract the operands from the instructions //

int field0 (int instruction)
{
 return((instruction>>19) & 0x7);
}

int field1 (int instruction)
{
 return((instruction>>16) & 0x7);
}

int field2 (int instruction)
{
 return(instruction & 0xFFFF);
}

int Opcode (int instruction)
{
 return(instruction>>22);
}

// print function to output the result of the simulation //

 printInstruction(int j, int instr,int ct1,int ft1,int is1, int
unit1,int regA1,int regB1,int exs1,int exe1,int wr1,int cmt1)
{
 char opcodeString[10];
 char Unit[10];

 if(Opcode(instr) == ADD)
 {
  strcpy(opcodeString,"Add");
 }
 else if(Opcode(instr) == NAND)
 {
  strcpy(opcodeString,"Nand");
 }
 else if(Opcode(instr) == MULT)
 {
  strcpy(opcodeString,"Mult");
 }
 else if(Opcode(instr) == LW)
 {
  strcpy(opcodeString,"Load");
```

```
 }
 else if(Opcode(instr) == SW)
 {
  strcpy(opcodeString,"Store");
 }
 else if(Opcode(instr) == BEQ)
 {
  strcpy(opcodeString,"Branch");
 }
 else if(Opcode(instr) == HALT)
 {
  strcpy(opcodeString,"Halt");
 }
 else if(Opcode(instr) == NOOP)
 {
  strcpy(opcodeString,"Noop");
 }
 else
 {
  strcpy(opcodeString,"Data");
 }
 if (unit1 == ADD1)
 strcpy(Unit,"ALU1");
 else if(unit1 == ADD2)
 strcpy(Unit,"ALU2");
 else if(unit1 == ADD3)
 strcpy(Unit,"ALU3");
 else if(unit1 == MUL1)
 strcpy(Unit,"MULT1");
 else if(unit1 == MUL2)
 strcpy(Unit,"MULT2");
 else if(unit1 == MUL3)
 strcpy(Unit,"MULT3");
 else
 strcpy(Unit,"N/A");

  printf("\n        %d         %4s          %d            %d       %s
%d      %d        %d       %d        %d          %d
   \n\n",ct1,opcodeString,ft1,is1,Unit,regA1,regB1,exs1,exe1,wr1,cmt1);
}
```

Appendix C: Code for dynamic verification processor

```
            // Header files //
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>

#define NUMMEMORY 65536 /* Total of 2048 instructions of 32 bits */
#define ADD 0
#define NAND 1
#define LW 2
#define SW 3
#define BEQ 4
#define MULT 5
#define HALT 6
#define NOOP 7

#define NOOPINSTRUCTION 0x1c00000

#define ADD1 10
#define ADD2 20
#define ADD3 30
#define MUL1 40
#define MUL2 50
#define MUL3 60
#define LW1 70
#define LW2 80
#define LW3 90

            // Defining the structures for the pipeline stages //
typedef struct IFISStruct
{
 int instr;
}IFISType;

typedef struct ISEXStruct
{
 int instr;
}ISEXType;

typedef struct EXWBStruct
{
 int instr;
}EXWBType;

typedef struct WBENDStruct
{
 int instr;
}WBENDType;

typedef struct RESStruct
{
 int busy;
 int QJ,QK;
 int VJ,VK;
```

```
 int AluRe;
 int RegA,RegB,Dest;
}RESType;

typedef struct stateStruct
{
 int pc;
 int cycles;
 int vj1,vj2,vj3,vj4,vj5,vj6;
 int vk1,vk2,vk3,vk4,vk5,vk6;
 int wb1,wb2,wb3,wb4,wb5,wb6;
 int CHKcommA,CHKcommB,CHKcmp;
 int chk[2048];
 IFISType IFIS;
 ISEXType ISEX;
 EXWBType EXWB;
 WBENDType WBEND;
}stateType;

            // start of main function //
int main(int argc, char *argv[])
{
 FILE *inFilePtr;
 char *inputFile;
 stateType state,newState,current;
 RESType Add1,Add2,Add3,Mul1,Mul2,Mul3;
 int instrMem[NUMMEMORY],dataMem[NUMMEMORY];
 int inst = 0;
 int FP[8];
 int Register[8];
 int ARegister[8];
 int i;

 int p1,p2,p3,p4,p5,p6;
 int ct[2048];
 int ft[2048];
 int is[2048];
 int unit[2048];
 int regA[2048];
 int regB[2048];
 int exs[2048];
 int exe[2048];
 int wr[2048];
 int cmt[2048];
 int cmtchk[2048];
 int CHKcomm[2048];

 int CHKRA[2048];
 int CHKRB[2048];
 int CHKRS[2048];
 int RA[2048];
 int RB[2048];
 int DS[2048];
 int RS[2048];
 int ins[2048];
 int CHKcp[2048];
 int c1=2,c2=2,c3=2;
```

```
int c4=5,c5=5,c6=5;
int compA = 2,compM = 5;
int k = 1;

 state.pc = 0;
 state.cycles = 1;
 state.IFIS.instr = NOOPINSTRUCTION;
 state.ISEX.instr = NOOPINSTRUCTION;
 state.EXWB.instr = NOOPINSTRUCTION;
 state.WBEND.instr = NOOPINSTRUCTION;

           // Initializing the components in a reservation station //

 Add1.busy = 0;
 Add1.QJ = -1;
 Add1.QK = -1;
 Add1.VJ = -1;
 Add1.VK = -1;

 Add2.busy = 0;
 Add2.QJ = -1;
 Add2.QK = -1;
 Add2.VJ = -1;
 Add2.VK = -1;

 Add3.busy = 0;
 Add3.QJ = -1;
 Add3.QK = -1;
 Add3.VJ = -1;
 Add3.VK = -1;

 Mul1.busy = 0;
 Mul1.QJ = -1;
 Mul1.QK = -1;
 Mul1.VJ = -1;
 Mul1.VK = -1;

 Mul2.busy = 0;
 Mul2.QJ = -1;
 Mul2.QK = -1;
 Mul2.VJ = -1;
 Mul2.VK = -1;

 Mul3.busy = 0;
 Mul3.QJ = -1;
 Mul3.QK = -1;
 Mul3.VJ = -1;
 Mul3.VK = -1;

current.vj1 = 0;
current.vk1 = 0;
current.vj2 = 0;
current.vk2 = 0;
current.vj3 = 0;
current.vk3 = 0;
current.vj4 = 0;
current.vk4 = 0;
```

```
current.vj5 = 0;
current.vk5 = 0;
current.vj6 = 0;
current.vk6 = 0;

current.wb1 = 0;
current.wb2 = 0;
current.wb3 = 0;
current.wb4 = 0;
current.wb5 = 0;
current.wb6 = 0;

current.CHKcommA = 0;
current.CHKcommB = 0;




 for (i=0;i<8;i++)
 {
  FP[i] = 0;
  Register[i] = i;
  ARegister[i] = i;
 }

 for(i=0;i<2048;i++)
 current.chk[i] = 0;


/*----------Checking for Input arguments to the Simulator-----------*/
if(argc!=2)
{
 printf("\nError in usage:\n %s <Specify Input File>\n",argv[0]);
 exit(1);
}

inputFile = argv[1];
inFilePtr = fopen(inputFile,"r");

/*------------------Checking for Input Valid File--------------------
--*/
if(inFilePtr == NULL)
{
 printf("\n Failed to open %s, Specify correct File
name\n",inputFile);
 exit(1);
}

/*-----------------Reading instruction from file to memory--------*/
while(fscanf(inFilePtr,"%d",&instrMem[inst])!=EOF)
{
 dataMem[inst] = instrMem[inst];
 inst++;
}
printf("\n Number of Instructions read %d\n\n",inst);
```

```
 while(1)
 {
              // Checking for the HALT instruction //
  if(Opcode(state.WBEND.instr) == HALT)
  {
     if(cmtchk[inst -1] == 1)
   {

            // printing the output of simulation //
    printf("\n
OUTPUT \n");
    printf("                                                                -
---------\n");
    printf("\nCounter   Opcode  Int Fetch  Int Issu    Unit     Reg A
Reg B  ExStart  ExEnd  WrBack    Checker     Commit\n");
    printf("-----------------------------------------------------------
-------------------------------------------------------------\n");

   for (i=0;i<inst;i++)
    {

printInstruction(i,instrMem[i],ct[i],ft[i],is[i+1],unit[i+1],regA[i+1],
regB[i+1],exs[i+1],exe[i+1],wr[i+1],CHKcomm[i+1],cmt[i+1]);
    }


    for(i=0;i<8;i++)
    printf("\n Register[%d] = %d\n",i,ARegister[i]);
    printf("\n Total No of Cycles Executed : %d\n",state.cycles);
    printf("\n");
    exit(0);
   }
   else
   {
    newState.pc = state.pc - 1;
     }
  }

  newState = state;
  newState.vj1 = current.vj1;
  newState.vk1 = current.vk1;
  newState.wb1 = current.wb1;
  newState.vj2 = current.vj2;
  newState.vk2 = current.vk2;
  newState.wb2 = current.wb2;
  newState.vj3 = current.vj3;
  newState.vk3 = current.vk3;
  newState.wb3 = current.wb3;
  newState.vj4 = current.vj4;
  newState.vk4 = current.vk4;
  newState.wb4 = current.wb4;
  newState.vj5 = current.vj5;
  newState.vk5 = current.vk5;
  newState.wb5 = current.wb5;
  newState.vj6 = current.vj6;
  newState.vk6 = current.vk6;
```

```
 newState.wb6 = current.wb6;


 newState.CHKcommA = current.CHKcommA;
 newState.CHKcommB = current.CHKcommB;
 newState.CHKcmp = current.CHKcmp;

 for(i=0;i<inst;i++)
 newState.chk[i] = current.chk[i];

 newState.cycles++;

 /*----------------Instruction Fetch Stage-----------------------*/
 if (instrMem[state.pc] !=0)
 {
  newState.IFIS.instr = instrMem[state.pc];
  ct[newState.pc] = state.pc;
  ft[newState.pc] = state.cycles;
  newState.pc++;
 }

/*----------------Instruction Issue Stage-----------------------*/
if (newState.IFIS.instr !=0)
{
 newState.ISEX.instr = state.IFIS.instr;
 switch(Opcode(newState.ISEX.instr))
 {
  case ADD:
  {
           // checking if the ADD station is busy or not //

   if((Add1.busy == 1)&&(Add2.busy == 1)&&(Add3.busy == 1))
   {
    newState.pc = state.pc - 1;
   }
   if(Add1.busy == 0)
   {
           // checking if the ADD1 station is busy or not //

    Add1.busy = 1;
    Add1.RegA = field0(newState.ISEX.instr);
    Add1.RegB = field1(newState.ISEX.instr);
    Add1.Dest = field2(newState.ISEX.instr);
    is[state.pc] = state.cycles;
    p1 = state.pc;

    ins[p1] = newState.ISEX.instr;
    DS[p1] = Add1.Dest;

    unit[p1] = ADD1;
    FP[Add1.Dest] = ADD1;

    if(FP[Add1.RegA] == 0)
    {
     Add1.QJ = 0;
     Add1.VJ = Register[Add1.RegA];
     RA[p1] = Register[Add1.RegA];
```

```
  current.vj1 = 1;
  regA[p1] = state.cycles;
 }
 else
 {
  Add1.QJ = FP[Add1.RegA];
 }

 if(FP[Add1.RegB] == 0)
 {
  Add1.QK =  0;
  Add1.VK  = Register[Add1.RegB];
  RB[p1] = Register[Add1.RegB];
  current.vk1 = 1;
  regB[p1] = state.cycles;
 }
 else
 {
  Add1.QK = FP[Add1.RegB];
 }
}
        // checking if the ADD2 station is busy or not //

else if(Add2.busy == 0)
{
 Add2.busy = 1;
 Add2.RegA = field0(newState.ISEX.instr);
 Add2.RegB = field1(newState.ISEX.instr);
 Add2.Dest = field2(newState.ISEX.instr);

 is[state.pc] = state.cycles;
 p2 = state.pc;
 unit[p2] = ADD2;

 ins[p2] = newState.ISEX.instr;
 DS[p2] = Add2.Dest;


 FP[Add2.Dest] = ADD2;

 if(FP[Add2.RegA] == 0)
 {
  Add2.QJ = 0;
  Add2.VJ = Register[Add2.RegA];
  RA[p2] = Register[Add2.RegA];
  regA[p2] = state.cycles;

  current.vj2 = 1;
 }
 else
 {
  Add2.QJ = FP[Add2.RegA];
 }

 if(FP[Add2.RegB] == 0)
 {
  Add2.QK = 0;
```

```
   Add2.VK = Register[Add2.RegB];
   RB[p2] =  Register[Add2.RegB];
   current.vk2 = 1;
   regB[p2] = state.cycles;
  }
  else
  {
   Add2.QK = FP[Add2.RegB];
  }
}
           // checking if the ADD3 station is busy or not //

 else if(Add3.busy == 0)
 {
  Add3.busy = 1;
  Add3.RegA = field0(newState.ISEX.instr);
  Add3.RegB = field1(newState.ISEX.instr);
  Add3.Dest = field2(newState.ISEX.instr);

  is[state.pc] = state.cycles;
  p3 = state.pc;
  unit[p3] = ADD3;

  ins[p3] = newState.ISEX.instr;
  DS[p3] = Add3.Dest;


  FP[Add3.Dest] = ADD3;

  if(FP[Add3.RegA] == 0)
  {
   Add3.QJ = 0;
   Add3.VJ = Register[Add3.RegA];
   RA[p3] = Add3.VJ;
   current.vj3 = 1;
   regA[p3] = state.cycles;
  }
  else
  {
   Add3.QJ = FP[Add3.RegA];
  }

  if(FP[Add3.RegB] == 0)
  {
   Add3.QK = 0;
   Add3.VK = Register[Add3.RegB];
   RB[p3] = Add3.VK;
   regB[p3] = state.cycles;
   current.vk3 = 1;
  }
  else
  {
   Add3.QK = FP[Add3.RegB];
  }
 }
 break;
}// end of ADD
```

```
        // if the instruction is MULT //

case MULT:
{
        // checking if the MULT station is busy or not //

 if((Mul1.busy == 1)&&(Mul2.busy == 1)&&(Mul3.busy == 1))
 {
  newState.pc = state.pc - 1;
 }

        // checking if the MULT1 station is busy or not //

 if(Mul1.busy == 0)
 {
  Mul1.busy = 1;
  Mul1.RegA = field0(newState.ISEX.instr);
  Mul1.RegB = field1(newState.ISEX.instr);
  Mul1.Dest = field2(newState.ISEX.instr);
  is[state.pc] = state.cycles;
  p4 = state.pc;
  unit[p4] = MUL1;
  FP[Mul1.Dest] = MUL1;

  ins[p4] = newState.ISEX.instr;
  DS[p4] = Mul1.Dest;



  if(FP[Mul1.RegA] == 0)
  {
   Mul1.QJ = 0;
   Mul1.VJ = Register[Mul1.RegA];
   RA[p4] = Mul1.VJ;
   current.vj4 = 1;
   regA[p4] = state.cycles;
  }
  else
  {
   Mul1.QJ = FP[Mul1.RegA];
  }

  if(FP[Mul1.RegB] == 0)
  {
   Mul1.QK = 0;
   Mul1.VK = Register[Mul1.RegB];
   RB[p4] = Mul1.VK;
   current.vk4 = 1;
   regB[p4] = state.cycles;
  }
  else
  {
   Mul1.QK =  FP[Mul1.RegB];
  }
 }

        // checking if the MULT2 station is busy or not //
```

```
else if(Mul2.busy == 0)
{
 Mul2.busy = 1;
 Mul2.RegA = field0(newState.ISEX.instr);
 Mul2.RegB = field1(newState.ISEX.instr);
 Mul2.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p5 = state.pc;
 unit[p5] = MUL2;
 FP[Mul2.Dest] = MUL2;
 ins[p5] = newState.ISEX.instr;
 DS[p5] = Mul2.Dest;


 if(FP[Mul2.RegA] == 0)
 {
  Mul2.QJ = 0;
  Mul2.VJ = Register[Mul2.RegA];
  RA[p5] = Mul2.VJ;
  current.vj5 = 1;
  regA[p5] = state.cycles;
 }
 else
 {
  Mul2.QJ = FP[Mul2.RegA];
 }

 if(FP[Mul2.RegB] == 0)
 {
  Mul2.QK = 0;
  Mul2.VK = Register[Mul2.RegB];
  RB[p5] = Mul2.VK;
  current.vk5 = 1;
  regB[p5] = state.cycles;
 }
 else
 {
  Mul2.QK =  FP[Mul2.RegB];
 }
}
           // checking if the MULT3 station is busy or not //

else if(Mul3.busy == 0)
{
 Mul3.busy = 1;
 Mul3.RegA = field0(newState.ISEX.instr);
 Mul3.RegB = field1(newState.ISEX.instr);
 Mul3.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p6 = state.pc;
 unit[p6] = MUL3;
 FP[Mul3.Dest] = MUL3;

 ins[p6] = newState.ISEX.instr;
 DS[p6] = Mul3.Dest;
```

```
    if(FP[Mul3.RegA] == 0)
    {
     Mul3.QJ = 0;
     Mul3.VJ = Register[Mul3.RegA];
     RA[p6] = Mul3.VJ;
     current.vj6 = 1;
     regA[p6] = state.cycles;
    }
    else
    {
     Mul3.QJ = FP[Mul3.RegA];
    }

    if(FP[Mul3.RegB] == 0)
    {
     Mul3.QK = 0;
     Mul3.VK = Register[Mul3.RegB];
     RB[p6] = Mul3.VK;
     current.vk6 = 1;
     regB[p6] = state.cycles;
    }
    else
    {
     Mul3.QK = FP[Mul3.RegB];
    }
   }
   break;
  }// end of MUL

           // Checking if the instruction is HALT //


 case HALT:
 {
  is[state.pc] = state.cycles;
  break;
 }
}// end of switch

  if(Add1.busy == 1)
  {
   if(Add1.QJ!=0)

   {
    if(FP[Add1.RegA] == 0)
    {
     Add1.VJ = Register[Add1.RegA];
     RA[p1] = Add1.VJ;
     regA[p1] = state.cycles;
     current.vj1 = 1;
     Add1.QJ = 0;
    }
   }
   if(Add1.QK != 0)
    {
    if(FP[Add1.RegB] == 0)
    {
```

```
   Add1.VK = Register[Add1.RegB];
   RB[p1] = Add1.VK;
   regB[p1] = state.cycles;
   current.vk1 = 1;
   Add1.QK = 0;
  }
 }
}

if(Add2.busy == 1)
{
 if(Add2.QJ != 0)
 {
  if(FP[Add2.RegA] == 0)
  {
   Add2.VJ = Register[Add2.RegA];
   RA[p2] = Add2.VJ;
   regA[p2] = state.cycles;
   current.vj2 = 1;
   Add2.QJ = 0;
  }
 }
 if(Add2.QK != 0)
 {
  if(FP[Add2.RegB] == 0)
  {
   Add2.VK = Register[Add2.RegB];
   RB[p2] = Add2.VK;
   regB[p2] = state.cycles;
   current.vk2 = 1;
   Add2.QK = 0;
  }
 }
}

if(Add3.busy == 1)
{
 if(Add3.QJ != 0)
 {
  if(FP[Add3.RegA] == 0)
  {
   Add3.VJ = Register[Add3.RegA];
   RA[p3] = Add3.VJ;
   regA[p3] = state.cycles;
   current.vj3 = 1;
   Add3.QJ = 0;
  }
 }
 if(Add3.QK != 0)
 {
  if(FP[Add3.RegB] == 0)
  {
   Add3.VK = Register[Add3.RegB];
   RB[p3] = Add3.VK;
   regB[p3] = state.cycles;
   current.vk3 = 1;
   Add3.QK = 0;
```

```
  }
 }
}

if(Mul1.busy == 1)
{
 if(Mul1.QJ != 0)
 {
  if(FP[Mul1.RegA] == 0)
  {
   Mul1.VJ = Register[Mul1.RegA];
   RA[p4] = Mul1.VJ;
   regA[p4] = state.cycles;
   current.vj4 = 1;
   Mul1.QJ = 0;
  }
 }
 if(Mul1.QK != 0)
 {
  if(FP[Mul1.RegB] == 0)
  {
   Mul1.VK = Register[Mul1.RegB];
   RB[p4] = Mul1.VK;
   regB[p4] = state.cycles;
   current.vk4 = 1;
   Mul1.QK = 0;
  }
 }
}

if(Mul2.busy == 1)
{
 if(Mul2.QJ != 0)
 {
  if(FP[Mul2.RegA] == 0)
  {
   Mul2.VJ = Register[Mul2.RegA];
   RA[p5] = Mul2.VJ;
   regA[p5] = state.cycles;
   current.vj5 = 1;
   Mul2.QJ = 0;
  }
 }
 if(Mul2.QK != 0)
 {
  if(FP[Mul2.RegB] == 0)
  {
   Mul2.VK = Register[Mul2.RegB];
   RB[p5] = Mul2.VK;
   regB[p5] = state.cycles;
   current.vk5 = 1;
   Mul2.QK = 0;
  }
 }
}

if(Mul3.busy == 1)
```

```
  {
   if(Mul3.QJ != 0)
   {
    if(FP[Mul3.RegA] == 0)
    {
     Mul3.VJ = Register[Mul3.RegA];
     RA[p6] = Mul3.VJ;
     regA[p6] = state.cycles;
     current.vj6 = 1;
     Mul3.QJ = 0;
    }
   }
   if(Mul3.QK != 0)
   {
    if(FP[Mul3.RegB] == 0)
    {
     Mul3.VK = Register[Mul3.RegB];
     RB[p6] = Mul3.VK;
     regB[p6] = state.cycles;
     current.vk6 = 1;
     Mul3.QK = 0;
    }
   }
  }
}

/*----------------Instruction Execute Stage-----------------------*/
if (newState.ISEX.instr !=0)
{
 newState.EXWB.instr = state.ISEX.instr;

 if((newState.vj1 == 1)&&(newState.vk1 == 1))
 {
  exs[p1] = state.cycles - 1;
  c1--;
  if(c1 == 0)
  {
   current.vj1 = 0;
   current.vk1 = 0;
   exe[p1] = state.cycles;
   Register[Add1.Dest] = Add1.VJ + Add1.VK;
   Add1.AluRe = Add1.VJ + Add1.VK;
   RS[p1] = Add1.VJ + Add1.VK;
   current.wb1 = 1;
   c1 = 2;
  }

 if((newState.vj2 == 1)&&(newState.vk2 == 1))
 {
  exs[p2] = state.cycles - 1;
  c2--;
  if(c2 == 0)
  {
   current.vj2 = 0;
   current.vk2 = 0;
   exe[p2] = state.cycles;
   Register[Add2.Dest] = Add2.VJ + Add2.VK;
```

```
  Add2.AluRe = Add2.VJ + Add2.VK;
  RS[p2] = Add2.VJ + Add2.VK;
  current.wb2 = 1;
  c2 = 2;
 }

if((newState.vj3 == 1)&&(newState.vk3 == 1))
{
 exs[p3] = state.cycles - 1;
 c3--;
 if(c3 == 0)
 {
  current.vj3 = 0;
  current.vk3 = 0;
  exe[p3] = state.cycles;
  Register[Add3.Dest] = Add3.VJ + Add3.VK;
  Add3.AluRe = Add3.VJ + Add3.VK;
  RS[p3] = Add3.VJ + Add3.VK;
  current.wb3 = 1;
  c3 = 2;
 }
}

if((newState.vj4 == 1)&&(newState.vk4 == 1))
{
 exs[p4] = state.cycles - 4;
 c4--;
 if(c4 == 0)
 {
  current.vj4 = 0;
  current.vk4 = 0;
  exe[p4] = state.cycles;
  Register[Mul1.Dest] = Mul1.VJ * Mul1.VK;
  Mul1.AluRe = Mul1.VJ * Mul1.VK;
  RS[p4] = Mul1.VJ * Mul1.VK;
  current.wb4 = 1;
  c4 = 5;
 }
}

if((newState.vj5 == 1)&&(newState.vk5 == 1))
{
 exs[p5] = state.cycles - 4;
 c5--;
 if(c5 == 0)
 {
  current.vj5 = 0;
  current.vk5 = 0;
  exe[p5] = state.cycles;
  Register[Mul2.Dest] = Mul2.VJ * Mul2.VK;
  Mul2.AluRe = Mul2.VJ * Mul2.VK;
  RS[p5] = Mul2.VJ * Mul2.VK;
  current.wb5 = 1;
  c5 = 5;
 }
}
```

```
 if((newState.vj6 == 1)&&(newState.vk6 == 1))
 {
  exs[p6] = state.cycles - 4;
  c6--;
  if(c6 == 0)
  {
   current.vj6 = 0;
   current.vk6 = 0;
   exe[p6] = state.cycles;
   Register[Mul3.Dest] = Mul3.VJ * Mul3.VK;
   Mul3.AluRe = Mul3.VJ * Mul3.VK;
   RS[p6] = Mul3.VJ * Mul3.VK;
   current.wb6 = 1;
   c6 = 5;
  }

}
/*----------------Instruction Write Stage-----------------------*/
if (newState.EXWB.instr !=0)
{
 newState.WBEND.instr = state.EXWB.instr;

 if(newState.wb1 == 1)
 {
  current.wb1 = 0;
  Add1.busy = 0;
  FP[Add1.Dest] = 0;
  wr[p1] = state.cycles;
  current.chk[p1] = 1;
 }

 if(newState.wb2 == 1)
 {
  current.wb2 = 0;
  Add2.busy = 0;
  FP[Add2.Dest] = 0;
  wr[p2] = state.cycles;
  current.chk[p2] = 1;
 }

 if(newState.wb3 == 1)
 {
  current.wb3 = 0;
  Add3.busy = 0;
  FP[Add3.Dest] = 0;
  wr[p3] = state.cycles;
  current.chk[p3] = 1;
 }

 if(newState.wb4 == 1)
 {
  current.wb4 = 0;
  Mul1.busy = 0;
  FP[Mul1.Dest] = 0;
  wr[p4] = state.cycles;
  current.chk[p4] = 1;
 }
```

```
  if(newState.wb5 == 1)
  {
   current.wb5 = 0;
   Mul2.busy = 0;
   FP[Mul2.Dest] = 0;
   wr[p5] = state.cycles;
   current.chk[p5] = 1;
  }

  if(newState.wb6 == 1)
  {
   current.wb6 = 0;
   Mul3.busy = 0;
   FP[Mul3.Dest] = 0;
   wr[p6] = state.cycles;
   current.chk[p6] = 1;
  }
 }

/***************Checker Stage*********************/

  if(newState.chk[k] == 1)
  {
   /* Checker CHKcomm State*/

   CHKcomm[k] = state.cycles;
   CHKRA[k] = ARegister[field0(ins[k])];
   CHKRB[k] = ARegister[field1(ins[k])];


   if(RA[k] == CHKRA[k])
   {
    current.CHKcommA = 1;
     }
   else
   {
    current.CHKcommA = 1;
    printf("\n Error Detected\n");
    printf("\n The Core value is: %d.The Correct Value is: %d.The Error
is fixed.\n",RA[k],CHKRA[k]);
   }

  //  RB[3] = 1000;// introducing errors //

   if(RB[k] == CHKRB[k])
   {
    current.CHKcommB = 1;
   }
   else
   {
    current.CHKcommB = 1;
    printf("\n Error Detected\n");
    printf("\n The Core value is: %d.The Correct Value is: %d.The Error
is fixed.\n",RB[k],CHKRB[k]);
   }
```

```
/*****************CHKcomp State*****************/
 switch(Opcode(ins[k]))
 {
  case ADD:
  {
   compA--;
   if (compA == 0)
   {
    CHKRS[k] = CHKRA[k] + CHKRB[k];
    if(CHKRS[k] == RS[k])
    {
     CHKcp[k] = state.cycles;
     current.CHKcmp = 1;
     compA = 3;
    }
    else
    {
     printf("\n ERROR\n");
     compA = 3;
    }
   }
    break;
   }

  case MULT:
  {
   compM--;
   if(compM == 0)
   {
    CHKRS[k] = CHKRA[k] * CHKRB[k];
    if(CHKRS[k] == RS[k])
    {
     CHKcp[k] = state.cycles;
     current.CHKcmp = 1;
     compM = 5;
    }
    else
    {
     printf("\n ERROR\n");
     compM = 5;
    }
   }
    break;
  }
    }
}
/********commit stage*****************************/

 if(newState.CHKcmp == 1)
 {
  state.cycles++;
  switch(Opcode(ins[k]))
  {
   case ADD:
   {
    ARegister[DS[k]] = CHKRA[k] + CHKRB[k];
```

```
      current.CHKcmp = 0;
      cmt[k] = state.cycles;
      cmtchk[k] = 1;
      k++;
      break;
     }

     case MULT:
     {
      ARegister[DS[k]] = CHKRA[k] * CHKRB[k];

      current.CHKcmp = 0;
      cmt[k] = state.cycles;
      cmtchk[k] = 1;
      k++;
      break;
     }
    }
   }// end of if CHKcommA and B

  state = newState;
 }// end of while
}// end of main

      // Functions for extracting the operands from the instructions //

int field0 (int instruction)
{
 return((instruction>>19) & 0x7);
}

int field1 (int instruction)
{
 return((instruction>>16) & 0x7);
}

int field2 (int instruction)
{
 return(instruction & 0xFFFF);
}

int Opcode (int instruction)
{
 return(instruction>>22);
}


            // print function to output the result of the simulation //

 printInstruction(int j, int instr,int ct1,int ft1,int is1, int
unit1,int regA1,int regB1,int exs1,int exe1,int wr1,int CHKcomm1,
int cmt1)
{
 char opcodeString[10];
 char Unit[10];

 if(Opcode(instr) == ADD)
```

```
 {
  strcpy(opcodeString,"Add");
 }
 else if(Opcode(instr) == NAND)
 {
  strcpy(opcodeString,"Nand");
 }
 else if(Opcode(instr) == MULT)
 {
  strcpy(opcodeString,"Mult");
 }
 else if(Opcode(instr) == LW)
 {
  strcpy(opcodeString,"Load");
 }
 else if(Opcode(instr) == SW)
 {
  strcpy(opcodeString,"Store");
 }
 else if(Opcode(instr) == BEQ)
 {
  strcpy(opcodeString,"Branch");
 }
 else if(Opcode(instr) == HALT)
 {
  strcpy(opcodeString,"Halt");
 }
 else if(Opcode(instr) == NOOP)
 {
  strcpy(opcodeString,"Noop");
 }
 else
 {
  strcpy(opcodeString,"Data");
 }
 if (unit1 == ADD1)
 strcpy(Unit,"ALU1");
 else if(unit1 == ADD2)
 strcpy(Unit,"ALU2");
 else if(unit1 == ADD3)
 strcpy(Unit,"ALU3");
 else if(unit1 == MUL1)
 strcpy(Unit,"MULT1");
 else if(unit1 == MUL2)
 strcpy(Unit,"MULT2");
 else if(unit1 == MUL3)
 strcpy(Unit,"MULT3");
 else
 strcpy(Unit,"N/A");

  printf("\n %2d         %4s     %2d         %2d       %2s      %2d
%2d     %2d      %2d      %2d       %2d           %2d

\n\n",ct1,opcodeString,ft1,is1,Unit,regA1,regB1,exs1,exe1,wr1,CHKcomm1,
cmt1);
}
```

Appendix D: Code for dynamic verification processor with a data predictor.

```
            // Header files //
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>

#define NUMMEMORY 65536 /* Total of 2048 instructions of 32 bits */
#define ADD 0
#define NAND 1
#define LW 2
#define SW 3
#define BEQ 4
#define MULT 5
#define HALT 6
#define NOOP 7

#define NOOPINSTRUCTION 0x1c00000

#define ADD1 10
#define ADD2 20
#define ADD3 30
#define MUL1 40
#define MUL2 50
#define MUL3 60
#define LW1 70
#define LW2 80
#define LW3 90

            // Defining the structures for the pipeline stages //
typedef struct IFISStruct
{
 int instr;
}IFISType;

typedef struct ISEXStruct
{
 int instr;
}ISEXType;

typedef struct EXWBStruct
{
 int instr;
}EXWBType;

typedef struct WBENDStruct
{
 int instr;
}WBENDType;

typedef struct RESStruct
{
 int busy;
 int QJ,QK;
 int VJ,VK;
```

```
 int AluRe;
 int RegA,RegB,Dest;
}RESType;

typedef struct stateStruct
{
 int pc;
 int cycles;
 int vj1,vj2,vj3,vj4,vj5,vj6;
 int vk1,vk2,vk3,vk4,vk5,vk6;
 int wb1,wb2,wb3,wb4,wb5,wb6;
 int CHKcommA,CHKcommB,CHKcmp;
 int chk[2048];
 IFISType IFIS;
 ISEXType ISEX;
 EXWBType EXWB;
 WBENDType WBEND;
}stateType;

            // start of main function //
int main(int argc, char *argv[])
{
 FILE *inFilePtr;
 char *inputFile;
 stateType state,newState,current;
 RESType Add1,Add2,Add3,Mul1,Mul2,Mul3;
 int instrMem[NUMMEMORY],dataMem[NUMMEMORY];
 int inst = 0;
 int FP[8];
 int Register[8];
 int ARegister[8];
 int i;

 int p1,p2,p3,p4,p5,p6;
 int ct[2048];
 int ft[2048];
 int is[2048];
 int unit[2048];
 int regA[2048];
 int regB[2048];
 int exs[2048];
 int exe[2048];
 int wr[2048];
 int cmt[2048];
 int cmtchk[2048];
 int CHKcomm[2048];

 int CHKRA[2048];
 int CHKRB[2048];
 int CHKRS[2048];
 int RA[2048];
 int RB[2048];
 int DS[2048];
 int RS[2048];
 int ins[2048];
 int CHKcp[2048];
 int c1=2,c2=2,c3=2;
```

```
int c4=5,c5=5,c6=5;
int compA = 2,compM = 5;
int k = 1;

 state.pc = 0;
 state.cycles = 1;
 state.IFIS.instr = NOOPINSTRUCTION;
 state.ISEX.instr = NOOPINSTRUCTION;
 state.EXWB.instr = NOOPINSTRUCTION;
 state.WBEND.instr = NOOPINSTRUCTION;

           // Initializing the components in a reservation station //

 Add1.busy = 0;
 Add1.QJ = -1;
 Add1.QK = -1;
 Add1.VJ = -1;
 Add1.VK = -1;

 Add2.busy = 0;
 Add2.QJ = -1;
 Add2.QK = -1;
 Add2.VJ = -1;
 Add2.VK = -1;

 Add3.busy = 0;
 Add3.QJ = -1;
 Add3.QK = -1;
 Add3.VJ = -1;
 Add3.VK = -1;

 Mul1.busy = 0;
 Mul1.QJ = -1;
 Mul1.QK = -1;
 Mul1.VJ = -1;
 Mul1.VK = -1;

 Mul2.busy = 0;
 Mul2.QJ = -1;
 Mul2.QK = -1;
 Mul2.VJ = -1;
 Mul2.VK = -1;

 Mul3.busy = 0;
 Mul3.QJ = -1;
 Mul3.QK = -1;
 Mul3.VJ = -1;
 Mul3.VK = -1;

current.vj1 = 0;
current.vk1 = 0;
current.vj2 = 0;
current.vk2 = 0;
current.vj3 = 0;
current.vk3 = 0;
current.vj4 = 0;
current.vk4 = 0;
```

```
current.vj5 = 0;
current.vk5 = 0;
current.vj6 = 0;
current.vk6 = 0;

current.wb1 = 0;
current.wb2 = 0;
current.wb3 = 0;
current.wb4 = 0;
current.wb5 = 0;
current.wb6 = 0;

current.CHKcommA = 0;
current.CHKcommB = 0;




 for (i=0;i<8;i++)
 {
  FP[i] = 0;
  Register[i] = i;
  ARegister[i] = i;
 }

 for(i=0;i<2048;i++)
 current.chk[i] = 0;


/*----------Checking for Input arguments to the Simulator-----------*/
if(argc!=2)
{
 printf("\nError in usage:\n %s <Specify Input File>\n",argv[0]);
 exit(1);
}

inputFile = argv[1];
inFilePtr = fopen(inputFile,"r");

/*------------------Checking for Input Valid File--------------------
--*/
if(inFilePtr == NULL)
{
 printf("\n Failed to open %s, Specify correct File
name\n",inputFile);
 exit(1);
}

/*-----------------Reading instruction from file to memory--------*/
while(fscanf(inFilePtr,"%d",&instrMem[inst])!=EOF)
{
 dataMem[inst] = instrMem[inst];
 inst++;
}
printf("\n Number of Instructions read %d\n\n",inst);
```

```
 while(1)
 {
              // Checking for the HALT instruction //
  if(Opcode(state.WBEND.instr) == HALT)
  {
     if(cmtchk[inst -1] == 1)
   {

            // printing the output of simulation //
    printf("\n
OUTPUT \n");
    printf("                                                                -
---------\n");
    printf("\nCounter   Opcode  Int Fetch  Int Issu    Unit     Reg A
Reg B  ExStart  ExEnd  WrBack    Checker     Commit\n");
    printf("-----------------------------------------------------------
-------------------------------------------------------------\n");

   for (i=0;i<inst;i++)
    {

printInstruction(i,instrMem[i],ct[i],ft[i],is[i+1],unit[i+1],regA[i+1],
regB[i+1],exs[i+1],exe[i+1],wr[i+1],CHKcomm[i+1],cmt[i+1]);
    }


    for(i=0;i<8;i++)
    printf("\n Register[%d] = %d\n",i,ARegister[i]);
    printf("\n Total No of Cycles Executed : %d\n",state.cycles);
    printf("\n");
    exit(0);
   }
   else
   {
    newState.pc = state.pc - 1;
     }
  }

  newState = state;
  newState.vj1 = current.vj1;
  newState.vk1 = current.vk1;
  newState.wb1 = current.wb1;
  newState.vj2 = current.vj2;
  newState.vk2 = current.vk2;
  newState.wb2 = current.wb2;
  newState.vj3 = current.vj3;
  newState.vk3 = current.vk3;
  newState.wb3 = current.wb3;
  newState.vj4 = current.vj4;
  newState.vk4 = current.vk4;
  newState.wb4 = current.wb4;
  newState.vj5 = current.vj5;
  newState.vk5 = current.vk5;
  newState.wb5 = current.wb5;
  newState.vj6 = current.vj6;
  newState.vk6 = current.vk6;
```

```
 newState.wb6 = current.wb6;


 newState.CHKcommA = current.CHKcommA;
 newState.CHKcommB = current.CHKcommB;
 newState.CHKcmp = current.CHKcmp;

 for(i=0;i<inst;i++)
 newState.chk[i] = current.chk[i];

 newState.cycles++;

 /*----------------Instruction Fetch Stage-----------------------*/
 if (instrMem[state.pc] !=0)
 {
  newState.IFIS.instr = instrMem[state.pc];
  ct[newState.pc] = state.pc;
  ft[newState.pc] = state.cycles;
  newState.pc++;
 }

/*----------------Instruction Issue Stage-----------------------*/
if (newState.IFIS.instr !=0)
{
 newState.ISEX.instr = state.IFIS.instr;
 switch(Opcode(newState.ISEX.instr))
 {
  case ADD:
  {
           // checking if the ADD station is busy or not //

   if((Add1.busy == 1)&&(Add2.busy == 1)&&(Add3.busy == 1))
   {
    newState.pc = state.pc - 1;
   }
   if(Add1.busy == 0)
   {
           // checking if the ADD1 station is busy or not //

    Add1.busy = 1;
    Add1.RegA = field0(newState.ISEX.instr);
    Add1.RegB = field1(newState.ISEX.instr);
    Add1.Dest = field2(newState.ISEX.instr);
    is[state.pc] = state.cycles;
    p1 = state.pc;

    ins[p1] = newState.ISEX.instr;
    DS[p1] = Add1.Dest;

    unit[p1] = ADD1;
    FP[Add1.Dest] = ADD1;

    if(FP[Add1.RegA] == 0)
    {
     Add1.QJ = 0;
     Add1.VJ = Register[Add1.RegA];
     RA[p1] = Register[Add1.RegA];
```

```
  current.vj1 = 1;
  regA[p1] = state.cycles;
 }
 else
 {
  Add1.QJ = FP[Add1.RegA];
 }

 if(FP[Add1.RegB] == 0)
 {
  Add1.QK =  0;
  Add1.VK  = Register[Add1.RegB];
  RB[p1] = Register[Add1.RegB];
  current.vk1 = 1;
  regB[p1] = state.cycles;
 }
 else
 {
  Add1.QK = FP[Add1.RegB];
 }
}
        // checking if the ADD2 station is busy or not //

else if(Add2.busy == 0)
{
 Add2.busy = 1;
 Add2.RegA = field0(newState.ISEX.instr);
 Add2.RegB = field1(newState.ISEX.instr);
 Add2.Dest = field2(newState.ISEX.instr);

 is[state.pc] = state.cycles;
 p2 = state.pc;
 unit[p2] = ADD2;

 ins[p2] = newState.ISEX.instr;
 DS[p2] = Add2.Dest;


 FP[Add2.Dest] = ADD2;

 if(FP[Add2.RegA] == 0)
 {
  Add2.QJ = 0;
  Add2.VJ = Register[Add2.RegA];
  RA[p2] = Register[Add2.RegA];
  regA[p2] = state.cycles;

  current.vj2 = 1;
 }
 else
 {
  Add2.QJ = FP[Add2.RegA];
 }

 if(FP[Add2.RegB] == 0)
 {
  Add2.QK = 0;
```

```
   Add2.VK = Register[Add2.RegB];
   RB[p2] =  Register[Add2.RegB];
   current.vk2 = 1;
   regB[p2] = state.cycles;
  }
  else
  {
   Add2.QK = FP[Add2.RegB];
  }
}
           // checking if the ADD3 station is busy or not //

 else if(Add3.busy == 0)
 {
  Add3.busy = 1;
  Add3.RegA = field0(newState.ISEX.instr);
  Add3.RegB = field1(newState.ISEX.instr);
  Add3.Dest = field2(newState.ISEX.instr);

  is[state.pc] = state.cycles;
  p3 = state.pc;
  unit[p3] = ADD3;

  ins[p3] = newState.ISEX.instr;
  DS[p3] = Add3.Dest;


  FP[Add3.Dest] = ADD3;

  if(FP[Add3.RegA] == 0)
  {
   Add3.QJ = 0;
   Add3.VJ = Register[Add3.RegA];
   RA[p3] = Add3.VJ;
   current.vj3 = 1;
   regA[p3] = state.cycles;
  }
  else
  {
   Add3.QJ = FP[Add3.RegA];
  }

  if(FP[Add3.RegB] == 0)
  {
   Add3.QK = 0;
   Add3.VK = Register[Add3.RegB];
   RB[p3] = Add3.VK;
   regB[p3] = state.cycles;
   current.vk3 = 1;
  }
  else
  {
   Add3.QK = FP[Add3.RegB];
  }
 }
 break;
}// end of ADD
```

```
        // if the instruction is MULT //

case MULT:
{
        // checking if the MULT station is busy or not //

 if((Mul1.busy == 1)&&(Mul2.busy == 1)&&(Mul3.busy == 1))
 {
  newState.pc = state.pc - 1;
 }

        // checking if the MULT1 station is busy or not //

 if(Mul1.busy == 0)
 {
  Mul1.busy = 1;
  Mul1.RegA = field0(newState.ISEX.instr);
  Mul1.RegB = field1(newState.ISEX.instr);
  Mul1.Dest = field2(newState.ISEX.instr);
  is[state.pc] = state.cycles;
  p4 = state.pc;
  unit[p4] = MUL1;
  FP[Mul1.Dest] = MUL1;

  ins[p4] = newState.ISEX.instr;
  DS[p4] = Mul1.Dest;



  if(FP[Mul1.RegA] == 0)
  {
   Mul1.QJ = 0;
   Mul1.VJ = Register[Mul1.RegA];
   RA[p4] = Mul1.VJ;
   current.vj4 = 1;
   regA[p4] = state.cycles;
  }
  else
  {
   Mul1.QJ = FP[Mul1.RegA];
  }

  if(FP[Mul1.RegB] == 0)
  {
   Mul1.QK = 0;
   Mul1.VK = Register[Mul1.RegB];
   RB[p4] = Mul1.VK;
   current.vk4 = 1;
   regB[p4] = state.cycles;
  }
  else
  {
   Mul1.QK =  FP[Mul1.RegB];
  }
 }

        // checking if the MULT2 station is busy or not //
```

```
else if(Mul2.busy == 0)
{
 Mul2.busy = 1;
 Mul2.RegA = field0(newState.ISEX.instr);
 Mul2.RegB = field1(newState.ISEX.instr);
 Mul2.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p5 = state.pc;
 unit[p5] = MUL2;
 FP[Mul2.Dest] = MUL2;
 ins[p5] = newState.ISEX.instr;
 DS[p5] = Mul2.Dest;


 if(FP[Mul2.RegA] == 0)
 {
  Mul2.QJ = 0;
  Mul2.VJ = Register[Mul2.RegA];
  RA[p5] = Mul2.VJ;
  current.vj5 = 1;
  regA[p5] = state.cycles;
 }
 else
 {
  Mul2.QJ = FP[Mul2.RegA];
 }

 if(FP[Mul2.RegB] == 0)
 {
  Mul2.QK = 0;
  Mul2.VK = Register[Mul2.RegB];
  RB[p5] = Mul2.VK;
  current.vk5 = 1;
  regB[p5] = state.cycles;
 }
 else
 {
  Mul2.QK =  FP[Mul2.RegB];
 }
}
           // checking if the MULT3 station is busy or not //

else if(Mul3.busy == 0)
{
 Mul3.busy = 1;
 Mul3.RegA = field0(newState.ISEX.instr);
 Mul3.RegB = field1(newState.ISEX.instr);
 Mul3.Dest = field2(newState.ISEX.instr);
 is[state.pc] = state.cycles;
 p6 = state.pc;
 unit[p6] = MUL3;
 FP[Mul3.Dest] = MUL3;

 ins[p6] = newState.ISEX.instr;
 DS[p6] = Mul3.Dest;
```

```
    if(FP[Mul3.RegA] == 0)
    {
     Mul3.QJ = 0;
     Mul3.VJ = Register[Mul3.RegA];
     RA[p6] = Mul3.VJ;
     current.vj6 = 1;
     regA[p6] = state.cycles;
    }
    else
    {
     Mul3.QJ = FP[Mul3.RegA];
    }

    if(FP[Mul3.RegB] == 0)
    {
     Mul3.QK = 0;
     Mul3.VK = Register[Mul3.RegB];
     RB[p6] = Mul3.VK;
     current.vk6 = 1;
     regB[p6] = state.cycles;
    }
    else
    {
     Mul3.QK = FP[Mul3.RegB];
    }
   }
   break;
  }// end of MUL

           // Checking if the instruction is HALT //


 case HALT:
 {
  is[state.pc] = state.cycles;
  break;
 }
}// end of switch

  if(Add1.busy == 1)
  {
   if(Add1.QJ!=0)

   {
    if(FP[Add1.RegA] == 0)
    {
     Add1.VJ = Register[Add1.RegA];
     RA[p1] = Add1.VJ;
     regA[p1] = state.cycles;
     current.vj1 = 1;
     Add1.QJ = 0;
    }
   }
   if(Add1.QK != 0)
    {
    if(FP[Add1.RegB] == 0)
    {
```

```
   Add1.VK = Register[Add1.RegB];
   RB[p1] = Add1.VK;
   regB[p1] = state.cycles;
   current.vk1 = 1;
   Add1.QK = 0;
  }
 }
}

if(Add2.busy == 1)
{
 if(Add2.QJ != 0)
 {
  if(FP[Add2.RegA] == 0)
  {
   Add2.VJ = Register[Add2.RegA];
   RA[p2] = Add2.VJ;
   regA[p2] = state.cycles;
   current.vj2 = 1;
   Add2.QJ = 0;
  }
 }
 if(Add2.QK != 0)
 {
  if(FP[Add2.RegB] == 0)
  {
   Add2.VK = Register[Add2.RegB];
   RB[p2] = Add2.VK;
   regB[p2] = state.cycles;
   current.vk2 = 1;
   Add2.QK = 0;
  }
 }
}

if(Add3.busy == 1)
{
 if(Add3.QJ != 0)
 {
  if(FP[Add3.RegA] == 0)
  {
   Add3.VJ = Register[Add3.RegA];
   RA[p3] = Add3.VJ;
   regA[p3] = state.cycles;
   current.vj3 = 1;
   Add3.QJ = 0;
  }
 }
 if(Add3.QK != 0)
 {
  if(FP[Add3.RegB] == 0)
  {
   Add3.VK = Register[Add3.RegB];
   RB[p3] = Add3.VK;
   regB[p3] = state.cycles;
   current.vk3 = 1;
   Add3.QK = 0;
```

```
  }
 }
}

if(Mul1.busy == 1)
{
 if(Mul1.QJ != 0)
 {
  if(FP[Mul1.RegA] == 0)
  {
   Mul1.VJ = Register[Mul1.RegA];
   RA[p4] = Mul1.VJ;
   regA[p4] = state.cycles;
   current.vj4 = 1;
   Mul1.QJ = 0;
  }
 }
 if(Mul1.QK != 0)
 {
  if(FP[Mul1.RegB] == 0)
  {
   Mul1.VK = Register[Mul1.RegB];
   RB[p4] = Mul1.VK;
   regB[p4] = state.cycles;
   current.vk4 = 1;
   Mul1.QK = 0;
  }
 }
}

if(Mul2.busy == 1)
{
 if(Mul2.QJ != 0)
 {
  if(FP[Mul2.RegA] == 0)
  {
   Mul2.VJ = Register[Mul2.RegA];
   RA[p5] = Mul2.VJ;
   regA[p5] = state.cycles;
   current.vj5 = 1;
   Mul2.QJ = 0;
  }
 }
 if(Mul2.QK != 0)
 {
  if(FP[Mul2.RegB] == 0)
  {
   Mul2.VK = Register[Mul2.RegB];
   RB[p5] = Mul2.VK;
   regB[p5] = state.cycles;
   current.vk5 = 1;
   Mul2.QK = 0;
  }
 }
}

if(Mul3.busy == 1)
```

```
  {
   if(Mul3.QJ != 0)
   {
    if(FP[Mul3.RegA] == 0)
    {
     Mul3.VJ = Register[Mul3.RegA];
     RA[p6] = Mul3.VJ;
     regA[p6] = state.cycles;
     current.vj6 = 1;
     Mul3.QJ = 0;
    }
   }
   if(Mul3.QK != 0)
   {
    if(FP[Mul3.RegB] == 0)
    {
     Mul3.VK = Register[Mul3.RegB];
     RB[p6] = Mul3.VK;
     regB[p6] = state.cycles;
     current.vk6 = 1;
     Mul3.QK = 0;
    }
   }
  }
}

/*----------------Instruction Execute Stage-----------------------*/
if (newState.ISEX.instr !=0)
{
 newState.EXWB.instr = state.ISEX.instr;

 if((newState.vj1 == 1)&&(newState.vk1 == 1))
 {
  exs[p1] = state.cycles - 1;
  c1--;
  if(c1 == 0)
  {
   current.vj1 = 0;
   current.vk1 = 0;
   exe[p1] = state.cycles;
   Register[Add1.Dest] = Add1.VJ + Add1.VK;
   Add1.AluRe = Add1.VJ + Add1.VK;
   RS[p1] = Add1.VJ + Add1.VK;
   current.wb1 = 1;
   c1 = 2;
  }

 if((newState.vj2 == 1)&&(newState.vk2 == 1))
 {
  exs[p2] = state.cycles - 1;
  c2--;
  if(c2 == 0)
  {
   current.vj2 = 0;
   current.vk2 = 0;
   exe[p2] = state.cycles;
   Register[Add2.Dest] = Add2.VJ + Add2.VK;
```

```
  Add2.AluRe = Add2.VJ + Add2.VK;
  RS[p2] = Add2.VJ + Add2.VK;
  current.wb2 = 1;
  c2 = 2;
 }

if((newState.vj3 == 1)&&(newState.vk3 == 1))
{
 exs[p3] = state.cycles - 1;
 c3--;
 if(c3 == 0)
 {
  current.vj3 = 0;
  current.vk3 = 0;
  exe[p3] = state.cycles;
  Register[Add3.Dest] = Add3.VJ + Add3.VK;
  Add3.AluRe = Add3.VJ + Add3.VK;
  RS[p3] = Add3.VJ + Add3.VK;
  current.wb3 = 1;
  c3 = 2;
 }
}

if((newState.vj4 == 1)&&(newState.vk4 == 1))
{
 exs[p4] = state.cycles - 4;
 c4--;
 if(c4 == 0)
 {
  current.vj4 = 0;
  current.vk4 = 0;
  exe[p4] = state.cycles;
  Register[Mul1.Dest] = Mul1.VJ * Mul1.VK;
  Mul1.AluRe = Mul1.VJ * Mul1.VK;
  RS[p4] = Mul1.VJ * Mul1.VK;
  current.wb4 = 1;
  c4 = 5;
 }
}

if((newState.vj5 == 1)&&(newState.vk5 == 1))
{
 exs[p5] = state.cycles - 4;
 c5--;
 if(c5 == 0)
 {
  current.vj5 = 0;
  current.vk5 = 0;
  exe[p5] = state.cycles;
  Register[Mul2.Dest] = Mul2.VJ * Mul2.VK;
  Mul2.AluRe = Mul2.VJ * Mul2.VK;
  RS[p5] = Mul2.VJ * Mul2.VK;
  current.wb5 = 1;
  c5 = 5;
 }
}
```

```
 if((newState.vj6 == 1)&&(newState.vk6 == 1))
 {
  exs[p6] = state.cycles - 4;
  c6--;
  if(c6 == 0)
  {
   current.vj6 = 0;
   current.vk6 = 0;
   exe[p6] = state.cycles;
   Register[Mul3.Dest] = Mul3.VJ * Mul3.VK;
   Mul3.AluRe = Mul3.VJ * Mul3.VK;
   RS[p6] = Mul3.VJ * Mul3.VK;
   current.wb6 = 1;
   c6 = 5;
  }

}
/*----------------Instruction Write Stage-----------------------*/
if (newState.EXWB.instr !=0)
{
 newState.WBEND.instr = state.EXWB.instr;

 if(newState.wb1 == 1)
 {
  current.wb1 = 0;
  Add1.busy = 0;
  FP[Add1.Dest] = 0;
  wr[p1] = state.cycles;
  current.chk[p1] = 1;
 }

 if(newState.wb2 == 1)
 {
  current.wb2 = 0;
  Add2.busy = 0;
  FP[Add2.Dest] = 0;
  wr[p2] = state.cycles;
  current.chk[p2] = 1;
 }

 if(newState.wb3 == 1)
 {
  current.wb3 = 0;
  Add3.busy = 0;
  FP[Add3.Dest] = 0;
  wr[p3] = state.cycles;
  current.chk[p3] = 1;
 }

 if(newState.wb4 == 1)
 {
  current.wb4 = 0;
  Mul1.busy = 0;
  FP[Mul1.Dest] = 0;
  wr[p4] = state.cycles;
  current.chk[p4] = 1;
 }
```

```
  if(newState.wb5 == 1)
  {
   current.wb5 = 0;
   Mul2.busy = 0;
   FP[Mul2.Dest] = 0;
   wr[p5] = state.cycles;
   current.chk[p5] = 1;
  }

  if(newState.wb6 == 1)
  {
   current.wb6 = 0;
   Mul3.busy = 0;
   FP[Mul3.Dest] = 0;
   wr[p6] = state.cycles;
   current.chk[p6] = 1;
  }
 }

/*************Level 1 Checker*******************/

  if((RA[k] == 1)||(RA[k] == 2)||(RA[k] == 3)||(RA[k] == 4)||(RA[k] ==
5)||(RA[k] == 6)||(RA[k] == 7)||(RA[k] == 8)|| (RA[k] == 9))
  {
   lA[k] = 1;current.CHKcmp = 1;
  }
  else
  {
   lA[k] = 0;current.chk[k] = 1;
  }

  if((RB[k] == 1)||(RB[k] == 2)||(RB[k] == 3)||(RB[k] == 4)||(RB[k] ==
5)||(RB[k] == 6)||(RB[k] == 7)||(RB[k] == 8)|| (RB[k] == 9))
  {
   lB[k] = 1;current.CHKcmp = 1;
  }
  else
  {
   lB[k] = 0;current.chk[k] = 1;
  }

/***************Checker Stage*********************/

  if(newState.chk[k] == 1)
  {
   /* Checker CHKcomm State*/

   CHKcomm[k] = state.cycles;
   CHKRA[k] = ARegister[field0(ins[k])];
   CHKRB[k] = ARegister[field1(ins[k])];


   if(RA[k] == CHKRA[k])
   {
    current.CHKcommA = 1;
     }
```

```
   else
   {
    current.CHKcommA = 1;
    printf("\n Error Detected\n");
    printf("\n The Core value is: %d.The Correct Value is: %d.The Error
is fixed.\n",RA[k],CHKRA[k]);
   }

  //  RB[3] = 1000;// introducing errors //

   if(RB[k] == CHKRB[k])
   {
    current.CHKcommB = 1;
   }
   else
   {
    current.CHKcommB = 1;
    printf("\n Error Detected\n");
    printf("\n The Core value is: %d.The Correct Value is: %d.The Error
is fixed.\n",RB[k],CHKRB[k]);
   }

  /*****************CHKcomp State*****************/
   switch(Opcode(ins[k]))
   {
    case ADD:
    {
     compA--;
     if (compA == 0)
     {
      CHKRS[k] = CHKRA[k] + CHKRB[k];
      if(CHKRS[k] == RS[k])
      {
       CHKcp[k] = state.cycles;
       current.CHKcmp = 1;
       compA = 3;
      }
      else
      {
       printf("\n ERROR\n");
       compA = 3;
      }
     }
      break;
     }

    case MULT:
    {
     compM--;
     if(compM == 0)
     {
      CHKRS[k] = CHKRA[k] * CHKRB[k];
      if(CHKRS[k] == RS[k])
      {
       CHKcp[k] = state.cycles;
       current.CHKcmp = 1;
       compM = 5;
```

```
      }
      else
      {
       printf("\n ERROR\n");
       compM = 5;
      }
     }
      break;
    }
      }
  }
  /********commit stage*****************************/

   if(newState.CHKcmp == 1)
   {
    state.cycles++;
    switch(Opcode(ins[k]))
    {
     case ADD:
     {
      ARegister[DS[k]] = CHKRA[k] + CHKRB[k];

      current.CHKcmp = 0;
      cmt[k] = state.cycles;
      cmtchk[k] = 1;
      k++;
      break;
     }

     case MULT:
     {
      ARegister[DS[k]] = CHKRA[k] * CHKRB[k];

      current.CHKcmp = 0;
      cmt[k] = state.cycles;
      cmtchk[k] = 1;
      k++;
      break;
     }
    }
   }// end of if CHKcommA and B

  state = newState;
 }// end of while
}// end of main

      // Functions for extracting the operands from the instructions //

int field0 (int instruction)
{
 return((instruction>>19) & 0x7);
}

int field1 (int instruction)
{
 return((instruction>>16) & 0x7);
}
```

```
int field2 (int instruction)
{
 return(instruction & 0xFFFF);
}

int Opcode (int instruction)
{
 return(instruction>>22);
}


            // print function to output the result of the simulation //

 printInstruction(int j, int instr,int ct1,int ft1,int is1, int
unit1,int regA1,int regB1,int exs1,int exe1,int wr1,int CHKcomm1,
int cmt1)
{
 char opcodeString[10];
 char Unit[10];

 if(Opcode(instr) == ADD)
 {
  strcpy(opcodeString,"Add");
 }
 else if(Opcode(instr) == NAND)
 {
  strcpy(opcodeString,"Nand");
 }
 else if(Opcode(instr) == MULT)
 {
  strcpy(opcodeString,"Mult");
 }
 else if(Opcode(instr) == LW)
 {
  strcpy(opcodeString,"Load");
 }
 else if(Opcode(instr) == SW)
 {
  strcpy(opcodeString,"Store");
 }
 else if(Opcode(instr) == BEQ)
 {
  strcpy(opcodeString,"Branch");
 }
 else if(Opcode(instr) == HALT)
 {
  strcpy(opcodeString,"Halt");
 }
 else if(Opcode(instr) == NOOP)
 {
  strcpy(opcodeString,"Noop");
 }
 else
 {
  strcpy(opcodeString,"Data");
 }
```

```
 if (unit1 == ADD1)
 strcpy(Unit,"ALU1");
 else if(unit1 == ADD2)
 strcpy(Unit,"ALU2");
 else if(unit1 == ADD3)
 strcpy(Unit,"ALU3");
 else if(unit1 == MUL1)
 strcpy(Unit,"MULT1");
 else if(unit1 == MUL2)
 strcpy(Unit,"MULT2");
 else if(unit1 == MUL3)
 strcpy(Unit,"MULT3");
 else
 strcpy(Unit,"N/A");

  printf("\n %2d         %4s     %2d         %2d       %2s      %2d
%2d     %2d      %2d      %2d       %2d           %2d

\n\n",ct1,opcodeString,ft1,is1,Unit,regA1,regB1,exs1,exe1,wr1,CHKcomm1,
cmt1);
}
```