UNIVERSITY OF CALIFORNIA, SAN DIEGO

CSE 240A - Graduate Computer Architecture

Project Final Report

Costas Zarifis

December 18, 2014

1 INTRODUCTION

In this report we describe the implementation of a MIPS R10000 simulator and we provide the tests that verify our simulator's proper operation. The afore-mentioned tests are a collection of both the class-wide suite of test trace files and additional trace files that attack some more edge cases. All these tests are described in this document and the actual output of the simulator will be included in the submitted zip file. Naturally it would be impossible to include the pipeline stages for bigger trace files in this report as it would be impossible to verify that our simulator has the proper behaviour, therefore the HTML files including the WEB UI generated by our simulator are included for each test.

The web UI includes the following "structures" of our simulator:

- The pipeline's timeline
- The mapping between logical and physical registers
- The active list
- The busy-bit-table
- The integer queue
- The FP queue

- The address queue
- And finally the trace file that was executed

The user can easily verify the proper use of the simulator by checking the output of each one of these structures in each cycle.

2 Implementation Details

The simulator was implemented in Python 2.7.8, with the help of the pandas library . Pandas was mainly used to keep the consistency of the stages for each instruction and to enable a better visual UI. It's basically an in-memory relational database that stores both the output of the queues, register maps and the actual stages that each instruction has passed through. Pandas is included in the Anaconda collection which is a set of the most widely used scientific python packages. In the next sections and on the included README file we provided information on how to install and run our simulator. If the resources of the system we plan to run our simulator on are limited, Miniconda would be a better candidate since it doesn't include all the packages but it provides the option to install only the packages that are required and to omit all the others.

After the simulator finishes the execution, the contents of the Pandas Dataframes are exported into the file system as CSV files and then parsed again, and outputted as HTML tables. Bootstrap was used to create a visual interface that allows an easier navigation between the different structures of the simulator by using an action bar at the top. Since, bootstrap and jQuery are necessary we thought it would be a better idea to use content delivery networks (CDNs) instead of actually installing all the required web libraries in our system as this would make running and testing our simulator in other systems a lot harder, besides the only requirement in our case is Internet access.

2.1 Design Details

Since some parts of the simulator were not explicitly described in full detail in the papers we had to figure out a way to implement them without causing any discrepancies to the normal operation of the processor. Additionally, the project description included some simplifications and the simulator was implemented accordingly.

2.1.1 Clock Cycles

The output of the simulator had to explicitly show the clock cycles along with an identifier of each stage that each instruction was processed by. To accomplish that we followed the instructions provided by the project description. For each stage a class was generated that, among others, included two functions:

1. Calc()
2. Edge()

These two functions obviously a have different functionality for each unit but some common elements were that during the Calc() we process the instruction that was forwarded by the previous unit or that was pulled from a queue, and during the Edge() we increase the clock counter, update the dataframes with the newer values for the current unit and perhaps execute some more processing that is supposed to happen right before the edge.

Our main class also has these two functions that in turn serially call all the corresponding functions of each unit, bellow is a snippet that shows this particular point of the main function and the actual body of the edge function.

```
def edge(self, df, dfMap, IfStage, IdStage,
             IiStage, FPADD1Stage, FPADD2Stage,
             FPADD3Stage, WBStage, ALU1Stage,
             FPMUL1Stage, FPMUL2Stage, FPMUL3Stage,
             ALU2Stage, AStage, LSStage):

    # calling the edge function of each stage
    IfStage.edge(df, dfMap)
    IdStage.edge(df, dfMap, self.ActiveList)
    IiStage.edge(df, dfMap, self.ActiveList)
    FPADD1Stage.edge(df, dfMap, self.ActiveList)
    FPADD2Stage.edge(df, dfMap, self.ActiveList)
    FPADD3Stage.edge(df, dfMap, self.ActiveList)
    FPMUL1Stage.edge(df, dfMap, self.ActiveList)
    FPMUL2Stage.edge(df, dfMap, self.ActiveList)
    FPMUL3Stage.edge(df, dfMap, self.ActiveList)
    ALU1Stage.edge(df, dfMap, self.ActiveList)
    ALU2Stage.edge(df, dfMap, self.ActiveList)
    AStage.edge(df, dfMap, self.ActiveList)
    LSStage.edge(df, dfMap, self.ActiveList)
    WBStage.edge(df, dfMap, self.ActiveList)

def main():

    # more code
    ...
    for i in range(clocks):
        m.clc = i+1
        m.calc(df, args, IfStage, IdStage, IiStage,
               FPADD1Stage, FPADD2Stage, FPADD3Stage,
               WBStage, ALU1Stage, FPMUL1Stage,
               FPMUL2Stage, FPMUL3Stage, ALU2Stage,
               AStage, LSStage)
        m.edge(df, dfMap, IfStage, IdStage, IiStage,
               FPADD1Stage, FPADD2Stage, FPADD3Stage,
```

```
            WBStage, ALU1Stage, FPMUL1Stage,
            FPMUL2Stage, FPMUL3Stage, ALU2Stage,
            AStage, LSStage)
...
# more code
```

2.1.2 Physical VS Logical Registers

Since MIPS R10K is an out-of-order superscalar, it needs to be able to execute instructions out of order. The method that was used for determining register dependencies in order to enable such functionality is named register renaming. What the processor actually does is that it maps logical registers, which are the registers that initially appear in the trace file, to physical ones. Physical registers are the ones that appear in the register files. If an instruction uses as a "source" a logical register that has been used previously as a "destination", it might cause a RAW hazard and the two logical instructions must be mapped to the same physical one, if this register is not written (or forwarded) by the time the following instruction is about to get executed the second instruction will have to stall. If however this logical register is used as a "destination" on both these hypothetical instructions (WAW - not a "real" dependency) this logical register has to be mapped to more than one physical registers thus making sure that if no other dependencies exist between these instructions the second one will be able to execute before the first one, or even simultaneously.

The paper states that there are separate FP and integer physical registers, but the project description stated that, we were expected to use the same set for both types of instructions. This simplified the implementation of the register file, busy bit table, register map tables and so on, but it created a minor issue that had to be resolved. According to the paper there is a forwarding path from the end of the second (and third) stage of the floating point execution unit to the beginning of the first one. In order to avoid unnecessary stalls that would arise due to this simplification we had to create such forwarding paths that lead to the beginning of the first stage of all the other execution units regardless of the type of the register. The test cases that are included in this project clearly show both these details.

2.1.3 Fetch Stage

The fetch stage is a part of the front-end. Since MIPS R10K is a superscalar it is expected to fetch multiple instructions per cycle thus minimizing the CPI if the right instruction mix is fetched. The right instruction mix in this case is a set of instructions that if no dependencies exist between them they can execute in parallel thus minimizing the time required for their execution. By default the Fetch unit fetches 4 instructions at a time but an argument can be used when running the simulator to modify the number of fetched instructions per cycle. Since, however there is a limitation on the input ports of the pipeline registers and the queues used by the processor this number has to be limited to at most 4.

According to the paper, the fetch unit fetches the instructions and it pushes them to the decode unit, if however the pipeline register that exists between these two stages fills up the fetching stops. This register is essentially a queue that can store up to 8 instructions.

Additionally, if a mispredicted branch occurs, the branch execution stage (ALU1) uses a flag (that includes the line of the mispredicted branch) to let the fetch stage know that it has to refetch all the instructions bellow this branch and set the mispredict bit to 0. This is obviously another simplification that was made to simulate mispredicted branches without actually having to calculate the values of each register, more on that will follow when we analyze the ALU1 unit.

2.1.4 Decode Stage

This is perhaps the more sophisticated and hardest to implement stage. The general concept is that given a set of instructions, this unit has to check if there is enough space in the active list and then decode the input instructions. During this stage the logical registers are mapped to physical ones, therefore the busy bit tables, the register mapping and the free list will get updated.

Additionally according to the Professor and the TAs right before the edge of this stage, the decoded instructions will enter the corresponding queues. If a queue is full the decoded instruction that was supposed to enter this queue will remain in a buffer inside this stage and will be pushed on a later cycle to the corresponding queue when it is verified that there is enough space. The other instructions that were fetched along with the one that has to be stalled will be pushed to the corresponding queue if there is enough space. This scenario will be covered by one of the test cases that will follow. Its importance is significant since it adds a novelty to the OoO superscalars we have analyzed in this class. By enabling this out of order insertions to the corresponding queues we also enable the out of order issuing that makes this processor more efficient when compared to Tomasulo and other OoO processors we analyzed.

The execution timeline indicates with an "ID" an instruction that is currently into the decode stage if it is waiting to get decoded (mapped to physical registers etc) or it is waiting to get added to a queue. A quick glance at the register mapping and the corresponding queue on our UI will provide a better idea on what exactly is going on and why such a stall occurred.

2.1.5 Issue Stage

The issue stage is also one of the most important units. This is the unit that will feed instructions to the corresponding execution units. It has to deal with the order that instructions are supposed to come out of each queue and it has to push the appropriate instruction to the corresponding execution unit. However, there is a limitation at this point, this unit can only issue up to 4 instructions per cycle, according to the paper. On the execution timeline that is included for each test case, the issue symbol (II) indicates that the corresponding instructions are inside the issue stage (therefore inside the queues) but only if during the next stage the actual execution takes place will be considered as "issued" and removed from the queue (Load and Store Instructions however are not pulled out of the queue at this step)

2.1.6 Floating Point Units

The Floating Point Units consists of three individual execution units for the floating point adder and three unites for the floating point multiplier. After the end of the second individual

execution unit the destination register has the final value but it is not ready to be written back to the register file as it has not yet finished packing. However the value can be forwarded to the beginning of every other execution unit (we mentioned the reason why we had to implement this when covering the physical and logical registers). The same forwarding path exists at the end of the third stage. At this point the destination has the final value, it has finished packing but it hasn't been written back to the register file.

Between the three execution units pipeline registers were used to push the instruction that was processed by the previous unit to the next one. Since the individual execution units are unique they are expected to behave accordingly. If we skim through the pipeline timeline we will notice that it is impossible to have the same individual execution unit executing 2 or more instructions simultaneously.

2.1.7 Integer Unit

The integer unit is pretty similar to the floating point unit. Some differences are that the integer unit consists of two units: ALU1 and ALU2 and each of these units basically consist of just one stage, instead of three separate ones which was the case for the FP units we described before. Both ALU1 and ALU2 can execute integer operations but only ALU1 can execute branches. We will analyze what happens in the case of a branch in the next subsection. These execution units retrieve the instructions that were pushed by the issue stage and execute the actual instruction. At the end of each of these stages there is a forwarding path that leads to the beginning of each execution unit. Therefore if a RAW dependency exists the next instruction it doesn't have to wait until the commit stage of the current one in order for it to begin executing.

2.1.8 Branches

Branches are executed by the ALU1 execution unit. Each branch instruction that appears on a trace file is supposed to include a mispredict bit. When a branch is decoded, a full copy of the active list will be created and stored inside the branch stack. At the decode stage there is no way of knowing if the current branch will be correctly predicted or not. This is known when we reach the ALU1 stage. If this bit is 1 that means that this branch is mispredicted if it's 0 it means the instruction is correctly predicted, so it will terminate properly, and the execution of the following instructions will continue without any discrepancies. If the branch is mispredicted a flag is created and sent to the fetch unit. The fetch unit will stop fetching new instructions, it will go back to the mispredicted branch, modify the mispredict bit and resume fetching instructions from that point on.

2.1.9 Loads & Stores

The execution pipeline of loads and stores consists of two separate stages. The first stage is used to calculate the address. In order for the address to be calculated the corresponding source register has to be available. So in this case we could have RAW dependencies on the registers causing this stage to stall exactly as it occurs on other units. A load store operation is supposed to leave the address queue only when the instruction is graduated. Memory dependencies also matter when it comes to loads and stores. If we store the value of a register

to a specific address we have to wait before executing a load instruction that accesses this exact address. Our simulator handles these cases correctly as we will see in the following test cases.

3 Running Instructions

As we mentioned this simulator was implemented in Python. Specifically Python version 2.7.8 was used during development on a Linux machine and it is guaranteed that the application will work under such a configuration. A cloud 9 VM with the latest version of this simulator will also be shared for easier evaluation.

If you want to install and run the simulator on a seperate machine Python 2.7.8 and pandas dataframes must be present. For instructions on how to install those follow the instructions provided on the readme file or on this blog post (http://blog.zarifis.info/installing-anaconda-numpy-scipy-pandas-etc/).

After installing the appropriate dependencies, we are ready to run the simulator. The main python file is on the directory src/run.py. To see the different input arguments type:

```
python run.py -h
```

This outputs the following:

```
usage: run.py [-h] [-in FILENAME] [-out OUTPUT] [-issue ISSUE]

optional arguments:
  -h, --help    show this help message and exit
  -in FILENAME  input file path
  -out OUTPUT   output file path
  -issue ISSUE  total number of instructions that can get issued (default is 4)
```

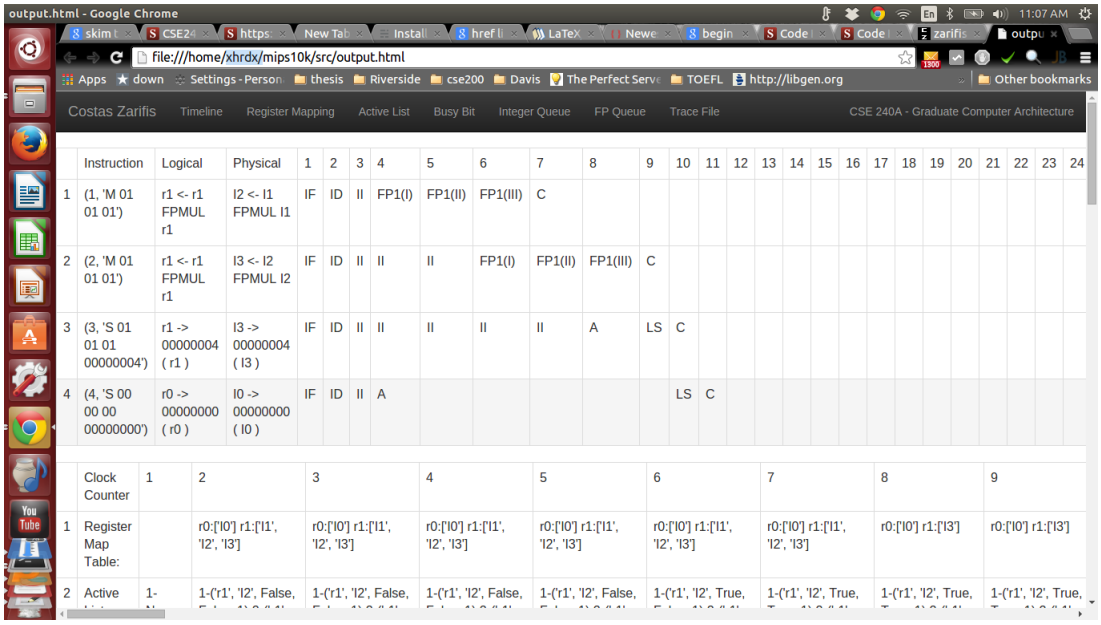
the input file path is the path of the benchmark we want to run and the output is the name of the output HTML file that will be generated. The default value of the output file is output.html.

This is an example of how to run a sample trace file:

```
python run.py -in benchmarks/test1.txt
```

For each one of the tests we provide the exact command that is used to run the simulator.

This will generate an HTML file with the UI shown bellow:



	Instruction	Logical	Physical	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	(1, 'M 01 01 01')	r1 <- r1 FPMUL r1	I2 <- I1 FPMUL I1	IF	ID	II	FP1(I)	FP1(II)	FP1(III)	C																	
2	(2, 'M 01 01 01')	r1 <- r1 FPMUL r1	I3 <- I2 FPMUL I2	IF	ID	II	II	II	FP1(I)	FP1(II)	FP1(III)	C															
3	(3, 'S 01 01 01 00000004')	r1 -> 00000004 (r1)	I3 -> 00000004 (I3)	IF	ID	II	II	II	II	II	A	LS	C													
4	(4, 'S 00 00 00 00000000')	r0 -> 00000000 (r0)	I0 -> 00000000 (I0)	IF	ID	II	A						LS	C													

	Clock Counter	1	2	3	4	5	6	7	8	9
1	Register Map Table:		r0:['I0'] r1:['I1', 'I2', 'I3']	r0:['I0'] r1:['I1', 'I2', 'I3']	r0:['I0'] r1:['I1', 'I2', 'I3']	r0:['I0'] r1:['I1', 'I2', 'I3']	r0:['I0'] r1:['I1', 'I2', 'I3']	r0:['I0'] r1:['I1', 'I2', 'I3']	r0:['I0'] r1:['I3']	r0:['I0'] r1:['I3']
2	Active	1-	1-('r1', 'I2', False,	1-('r1', 'I2', False,	1-('r1', 'I2', False,	1-('r1', 'I2', False,	1-('r1', 'I2', True,	1-('r1', 'I2', True,	1-('r1', 'I2', True,	1-('r1', 'I2', True,

4 Conventions

These are the stage naming conventions:

Instruction Fetch	IF
Instruction Decode	ID
Instruction Issue	II
Floating Point Execution Unit 1 stage I	FP1(I)
Floating Point Execution Unit 1 stage II	FP1(II)
Floating Point Execution Unit 1 stage III	FP1(III)
Floating Point Execution Unit 2 stage I	FP2(I)
Floating Point Execution Unit 2 stage II	FP2(II)
Floating Point Execution Unit 2 stage III	FP2(III)
Address Calculation	A
Integer Execution Unit 1	ALU1
Integer Execution Unit 2	ALU2
Commit & Write back	C
Branch Misprediction Bubble	X
Accessing Address	LS

Notice that commit and write back have the same symbol, since they both happen in the same stage, the other visual components provide access to the active queue and it can be verified that an instruction has been committed. Also the busy bit table can be used to verify that the registers of an instruction have been written back. Additionally the branch misprediction bubble is represented with the X

5 TESTING

This section provides a description of the test file, what we are trying to accomplish with each one and instructions on how to execute each one. To follow the reasoning please open the corresponding html file. The test files are also present on this URL: (http://zarifis.info/CSE240A/test%20results/)

5.1 TEST S1 - NO STALLS/DEPENDENCIES FOR LOGICAL REGISTER 0

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_s1.txt -out test_s1
```

This will generate a test_s1.html file with the output of the simulator

5.1.1 GENERAL CONCEPT

The general concept for this test is that whenever it comes to logical register r0 no renaming has to be performed, we always assign the physical register I0. Additionally, since this register always has value equal to zero no stalling will be caused. Open the corresponding test_s1.html file to view the timeline of this test.

5.2 TEST S5 - MAXIMUM COMMIT OF 4 INSTRUCTIONS

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_s5.txt -out test_s5
```

This will generate a test_s5.html file with the output of the simulator

5.2.1 GENERAL CONCEPT

As we've mentioned this is an OoO processor but it is important that the developer notices that the instruction execute serially even if under the hood the instructions execute out of order. When an instruction finishes execution it will have to wait until the previous instruction commits before it commits as well.

This test is used to verify that the maximum number of commits that can happen in a single cycle is 4. If more instructions wait to get committed they have to wait until the next cycle. This is a hardware limitation and the execution timeline can be seen at file test_s5.html . As we see the last instruction has to wait one cycle before committing.

5.3 TEST B1 - NO PARALLEL BRANCH EXECUTION

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_b1.txt -out test_b1
```

This will generate a test_b1.html file with the output of the simulator

5.3.1 General concept

This test is used to verify multiple key elements that were described earlier. Firstly, it shows that when a RAW dependency exists the forwarding path between the end of the 2nd floating point unit and the beginning of the 1st one forwards the value to the next instruction. Therefore the second instruction doesn't have to wait until the 1st one commits in order to begin executing. Secondly, it shows that the issuing correctly happens when all dependencies have been resolved. Thirdly it shows that due to the fact that according to the project description we have a common register file that both floating point and integer operations share, we had to implement the afore-mentioned forwarding path between different kind of instructions (FP and integer). Therefore the branch doesn't have to wait until the FP instruction above gets committed. Finally and most importantly it shows that only ALU1 can execute branches. Even though ALU2 is free the second branch waits unil the previous one finishes execution before it starts executing.

5.4 Test B6 - mispredict rollbacks

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_b6.txt -out test_b6
```

This will generate a test_b6.html file with the output of the simulator

5.4.1 General concept

This tests one of the hardest to implement features of this simulator, the branch misprediction. When a branch is found to be mispredicted while being executed inside the ALU1 unit, the previous instructions will finish without any issues, but the next ones will have to be cancelled on the next cycle, what the X signifies on the pipeline is that the corresponding units flush the instructions that were currently about to be executed in this unit. Obviously the refetched instructions will be executed and their commit will happen at the same time or after the instructions before the branch commit. Notice that on Sergio's 4 issue execution pipeline the floating point operation takes 4 cycles instead of 3 which is a mistake and it pushes the execution of all the next instructions one cycle after, that's why our timelines are slightly different.

5.5 Test LS1 - Stores execute strictly in order

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_ls1.txt -out test_ls1
```

This will generate a test_ls1.html file with the output of the simulator

5.5.1 General concept

As Sergio mentions on his website, MIPS R10K frees instructions from the Address queue only after they have committed. The reason for this is that store instructions have to be executed in order and also that when stores execute in order to irreversible changes will take place in the corresponding memory addresses.

Notice that the last store instruction calculates the address during the 4th cycle but it actually accesses it much later when it's safe to do. Also the 3rd instruction calculates the address during the 8th cycle because of a RAW dependency that occurred.

5.6 Test LS4 - Dependency Matrix

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_ls4.txt -out test_ls4
```

This will generate a test_ls4.html file with the output of the simulator

5.6.1 General concept

This test is used to show the proper use of the dependency matrix and it verifies that the order of the execution of load store instructions is maintained.

5.7 Test T1 - Branch Misprediction Fest

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_t1.txt -out test_t1
```

This will generate a test_t1.html file with the output of the simulator

5.7.1 General concept

This test is used to show the proper execution of multiple mispredicted branches. The trace file that is used includes only 3 mispredicted branches, sadly after 3 branches my machine runs out of memory. I am confident that it should work with more than 3 mispredicted branches and if tested on a better machine it should return the correct output.

5.8 Test T5 - floating point vector loop

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/test_t5.txt -out test_t5
```

This will generate a test_t5.html file with the output of the simulator

5.8.1 General concept

This test is used to show how the MIPS R10K would execute a real loop. Once again I had to limit the number of lines for this trace file because my machine ran out of memory, the provided output includes the execution of 115 lines of the trace file.

5.9 Personal Test - Fortunate Instruction Mix - Trace File

To run this test execute the following command from the src directory:

```
python run.py -in benchmarks/mytest.txt -out mytest
```

This will generate a mytest.html file with the output of the simulator

5.9.1 General concept

As we mentioned during the submission of the previous part of the project:

After executing a series of FP instructions with read after write dependencies, the floating point queue gets filled up. During the next cycle, the Fetch stage feeds the decode stage with 4 integer instructions. If the active list is empty and the integer queue is also empty (which will be the case in this test) the 4 instructions are going to get pushed to the integer list and get issued during the next stage even if the FP queue is full. As a result the integer instructions get issued and executed before the floating points instructions.

According to the paper, the professor and one of the TAs both the insertion into the active queue and the insertion to the corresponding queue happen during the decode stage. Therefore if one of those operations cannot happen, the instruction will remain in the decode stage, without blocking the following instructions.

That brings us to line 19. This line includes a floating point operation, however since the FP queue is full at that point, the decoded instruction has to wait before entering the queue until the 7th cycle, because at that point a previous instruction leaves the queue. Notice that the active list has the new instruction on the first appearance of the decode stage but the instruction actually enters the queue on the 7th cycle (both the active list and the FP list are available on the submitted HTML file). The same procedure is followed by instruction 20 but it has to wait even more until an instruction leaves the queue.

Notice however that the following four instractions are integer operations. Since there are no dependencies these instructions are issued without any delays and are executed two at a time by the corresponding ALU units. There are only two ALU units so these instructions obviously cannot get executed all at the same time.

The trace file is the following:

Listing 1: Fortunate Instruction Mix - Trace File

```
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
A 01 01 01
I 02 02 01
I 02 02 01
I 02 02 01
I 02 02 01
```

6 TRADE-OFFS

In this section we provide various insight that was gained after modifing the simulator to explore the following factors:

6.1 STRICT IN-ORDER MEMORY ACCESSES VS IN-ORDER ADDRESS RESOLUTION AND OUT-OF-ORDER MEMORY EXECUTION

Out of order memory execution accomplishes a much better execution time when data dependencies stall previous (load) instructions that would only read from the same address. If a store instruction appears however this such out of order execution is not possible, but a forwarding path that pushes the data directly to the newer load instruction would lead to a better CPI.

6.2 REORDER BUFFER SIZE (I.E. ACTIVE LIST)

A larger active list could definitely help when long RAW dependencies exist between instructions. Such instructions cause the queues and the actual active list to fill up. If the size of the active list was bigger more instructions would be able to get pushed to it and then get pushed to a queue that has space so that they would get issued earlier instead of waiting for the queues which they wouldn't get into anyway, to finish.

6.3 MAXIMUM NUMBER OF SIMULTANEOUS COMMITS

Obviously the fact that only 4 commits are allowed per cycle significantly increases the CPI on many tests that we included. If a higher number of commits was allowed the throughput would be higher and the performance increased.

6.4 MAXIMUM NUMBER OF DISPATCHES PER CYCLE

If the number of dispatches was increased by one and the circuits enabled the issuing of 5 instructions at the same time to execution units this would improve the performance provided that we had the right instruction mix (2 integer instructions and 2 floating point instruction and 1 load/store instruction). After this point though the bottleneck comes from the execution units themselves. Even if we had more instructions ready to be dispatched the fact that all our execution units would be full would be the actual limitation, thus the performance would not be increased.

6.5 NUMBER OF PHYSICAL REGISTERS

If the current setup remained as is and the only thing that changed was the number of physical registers, it would make no difference. Even if the entire pipeline was busy executing instructions the number of physical registers would be enough for every logical register to get mapped to a physical one.

6.6 NUMBER OF RESERVATION STATIONS OF EACH KIND (E.G. INTEGER, FLOATING POINT AND ADDRESS QUEUES)

This would definitely have improved the performance of integer and floating point operations but it would make no difference for loads and stores. A possible reason for a FP or integer queue to get filled up is obviously a bad instruction mix (always fetching FPADDs for example) with RAW dependencies. After this queue has been filled up new instructions have to wait for a previous instruction to commit before they enter the queue, therefore if the new instruction is of the same type (FPADD in this case), it has to wait regardless of whether or not a RAW dependency occurs. If the queue was bigger it would have been able to enter the queue and get issued faster (if no dependencies occurred) thus the throughput would be better. This is not the case for address queues though, since this is an actual FIFO instructions can't leave the queue if the previous instructions haven't been popped, so it wouldn't make a difference.

7 BILL OF MATERIALS

Since this was an implementation of the MIPS R10K simulator and not an actual implementation on the hardware level it's hard to calculate the exact bill of materials for it. An obvious correspondence of materials to parts of the implementation however is the following.

Each variable that was used as an argument for the calc and edge functions is a pipeline register, if statements and the variables that take place in them can be considered as the control unit of the processor that consists of multiplexers and logic gates. The active list, the

branch stack and queue classes correspond to the actual active list, branch stack and queues. The same applies to the register map tables, free list and all the similar classes that simulate the operation of real ones.