# COL216 Assignment 2 Report

Aaveg Jain, Tejas Anand

April 2023

## 1 Overview

### 1.1 Part 1

In this assignment we were required to build a MIPS Architecture Simulator which simulates a pipelined processor of various kinds, namely

- 5 stage pipeline
- 5 stage pipeline with forwarding
- 7-9 stage pipeline
- 7-9 stage pipeline with forwarding

### 1.2 Part 2

In the $2^{nd}$ Part of the assignment, we had to think of 3 different branch prediction strategies and test them on the given sequence of branchtraces. The strategies were -

- Maintaining a table of saturating counters for different $pc's$
- Maintaining a global history register of branches
- Combining both the strategies by using a table of histories for all $pc's$ along with their counters in another table

## 2 Design Decisions

### 2.1 Some common design decisions for 5 stage pipeline

- First we made structs to group control signals together according to the stage they are used in. We made struct WBctr for the WB stage, struct MEMctr for the MEM stage, struct EXctr for the EX stage. All of the attributes have the exact same signficance as written in the textbook, except "branch" which signifies whether the instruction is a branch instruction or not.

- Then we made a struct PipelineReg, which will act as pipeline registers between various stages of the pipeline. It has attributes int at_PC, int PC_new, WBctr* WB, MEMctr* MEM, EXctr* EX, int ALURESULT, int RegReadData1, int RegReadData2, int RS, int RT, int RD, bool zero. All of the attributes won't be needed for each pipeline register.

- In the function executePipelined, we created 4 instances of the struct PipelineReg, and initialized them properly, setting the PC attributes negative. In the while loop, we only start execution of any stage after checking the PC is non negative. So since all stages won't be there during initial cycles, this ensures no unnecessary execution takes place during initial cycles.

- We perform our execution cycle wise. In each cycle 5 stages happen namely (in order) Write-Back, Mem, Ex, Id, If. This ensures that Write-Back happens in first half of the clock cycles. After each stage, we change the attributes of the next Pipeline Reg by the values of the previous Pipeline Reg. For example, after Mem Stage, data stored in EX/MEM Pipeline Reg is transferred to MEM/WB Pipeline Reg. This mimics data flowing in the MIPS Processor. Additionally, jump and branch happen after ID and EX stages respectively.

### 2.2 5 Stage Pipeline

- In case of branch and jump, we allow instructions to be fetched until we change the PC in the EX and ID stages respectively. When we change the PC, we flush the previous instructions fetched by changing the PC negative in the latches occurring before that stage, which flushes the previously fetched instructions.

- Stalls are checked in the ID stage, and any dependency which might be present in the EX and MEM stages is stalled till it reaches the WB stage. Since the WB occurs in first half cycle, the correct value can be read in the ID stage. They are implemented by disabling the PC and the previous stages pipeline registers from updating and sending a bubble down the pipeline by setting the at_PC of the pipeline register to be negative.

### 2.3 5 Stage Pipeline With Forwarding

- There are three aspects of forwarding considered - forwarding from WB and MEM to EX stage for handling dependency of ALU operations, forwarding of WB to MEM stage for handling dependency of sw instructions, and finally stall hazard detection in ID stage, to handle dependency of ALU operations on a lw instruction, which gets the required value only till the MEM stage.

- Stalls are implemented as mentioned in previous part, whereas forwarding is implemented by saving values to be forwarded in additional variables.

### 2.4 Some common design decisions for 7-9 stage pipeline

- As in the case of 5 stage pipeline, we make structs of control signals, grouping them according to the stage they are used in - EX, DM1, DM2, WB stages. Again we create a struct for the pipeline registers and created 8 instances of it for the 9 stages.

- The attributes of the new pipeline register struct are the same as before, along with an additional attribute "is_9_stage" for checking whether we need 7 or 9 stages.
- PC being negative holds the same significance as before, and in this case we assume that RW and RR stages take an entire cycle unlike the previous case where they took half cycles.
- Execution procedure is same as above, with the stages happening in order - RW, DM2, DM1, EX, RR, ID2, ID1, IF2, IF1 for a 9 stage instruction, and DM2 and DM1 ommited for 7 stage instruction.
- If a command is a 7 stage, then it goes directly from the EX stage to the RW stage by updating the DM2/RW pipeline register attributes and setting the at_PC of the EX/DM1 pipeline register negative.
- If a 7 stage command comes after a 9 stage command, we stall two cycles to ensure FIFO. We also dont allow two instructions to be in RW in same cycle, even if only one is using the write port. Additionally the jump and beq instructions are implemented in the same manner as above, with the branch happening after the EX stage and jump after the ID2 stage.
- In IF1 we update our PC, in IF2 we fetch the new instruction, in ID1 we check for stalls caused due to a 7 stage command after a 9 stage, in ID2 we set the control signals, in RR we read the register values and in EX stage we perform our ALU operation. In DM1 we perform memory read (lw), in DM2 we perform memory write(sw) and in RW we perform our writeback.

### 2.5 7-9 Stage Pipeline

- In case of branch and jump commands, we perform the same procedure of flushing as above. Stalls are due to data dependency are checked in the RR stage and are implemented as follows - We check all active stages after RR for data dependency and store the required values for checking stalls in a vector of a struct "check". We then check all the stages in the RR stage using this vector, most recent stage first, and stall as soon as our stall condition is satisfied.
- To insert a stall in the pipeline, a bubble is sent in the next stage by making PC negative, and all the previous stages' pipeline registers' write enable is disabled to prevent them from being changed. PC is also kept the same.

### 2.6 7-9 Stage Pipeline With Forwarding

- According to required specifications, forwarding has been done only from the DM2/RW pipeline register. In all the stages - RR, EX, DM1, DM2 forwarding check has been used to see if forwarding is required. (in DM1, DM2 for the sw command, and the rest for the ALU operations).
- Stalls, if required, are checked in the RR stage. Only the EX, DM1 stages are checked for dependencies for stalls, since if the required value is in RW it can be forwarded, and if it is in DM2, it can be forwarded to ALU from RW in next cycle. Stalls are implemented as in above part, where we disable the write enable for the previous pipeline registers and keep the PC same.

### 2.7 Saturating Branch Predictor

In this strategy, we had a table of size $2^{14}$ for each distinct class of $pc's$ having distinct last 14 bits. Each entry of the table was a saturating 2 bit counter that got incremented/decremented depending upon the branch was taken or not. We predicted *taken* whenever the counter for a *pc* is 11 or 10, *not taken* whenever the counter for a *pc* is 00 or 01.

### 2.8 Global Branch History Register

In this strategy, we had a *history* bitset, which denoted the outcomes of the last 2 branches for any *pc*. We also had a history table, which contained 2-bit counters for each of the 4 possible histories. For Prediction, we would index into the history table using the history bitset, check the counter, if it is 11 or 10 predict *taken*, otherwise predict *not taken*. For updation, we would increment the counter in the history table of a given history depending upon the branch corresponding to that was actually taken, and afterwards update the *history*

### 2.9 Branch History Register with Saturating Counters

In this strategy, not only we had a global history and a global history table, but also a *local history* table of size $2^{14}$ containing the local histories of all the $pc's$ having the same least significant 14 bits. Then corresponding to all the 4 possible histories of a given pc, we have 4 counters in the *combination* table. So the size of the combination table is $2^{16}$.
The prediction strategy is a bit different in this case, we first check the local prediction by obtaining the local history by indexing into the *local history* table using the pc, then we index into the *combination* table using the *pc* and the obtained local history to check the value of the counter.
If the counter is predicting strongly taken or strongly not taken, *i.e* it is 00 or 11, we follow the counter, otherwise we follow the prediction of the global history.
For updation, depending upon if the branch was taken or not, we first increment/decrement the counters in the *combination table* and *global table*, and then update the histories in *local history table* and the *global history* bitset.

## 3 Benchmarking

The test cases uploaded on github were used as the instruction sets on which the code was tested.

- Instruction Set 1

```
addi $1, $0, 2
addi $2, $0, 1
sw $1 , 1023($2)
add $1,$1,$1
sub $2 , $1, $2
lw $3, 1021($2)
```

- Instruction Set 2

```
addi $1, $0, 2
addi $2, $0, 1
beq $1 , $2 , one
sw $1 , 1023($2)
addi $2 , $2 , 1
bne $1 , $2 , one
beq $1 , $2 , two
one: addi $2 , $2 , -1
two: lw $3 , 1022($2)
```

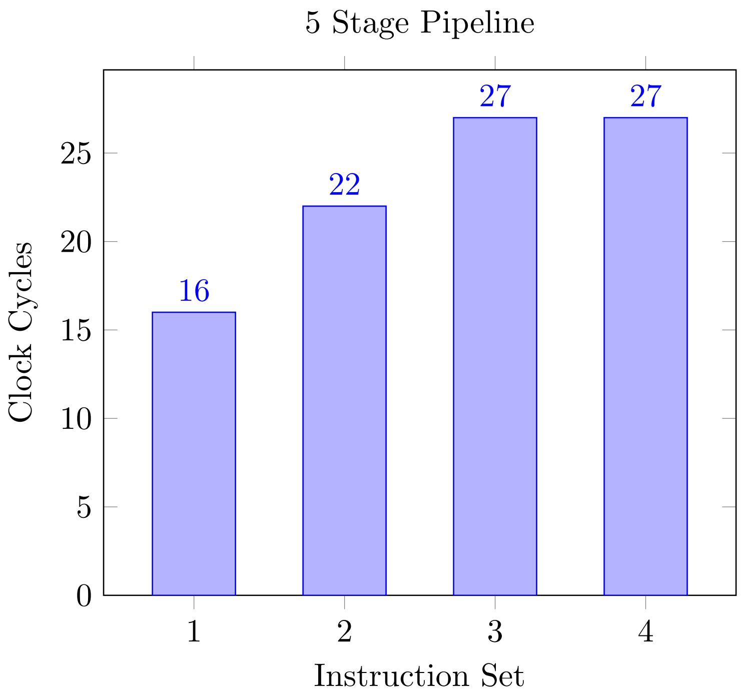
- Instruction Set 3

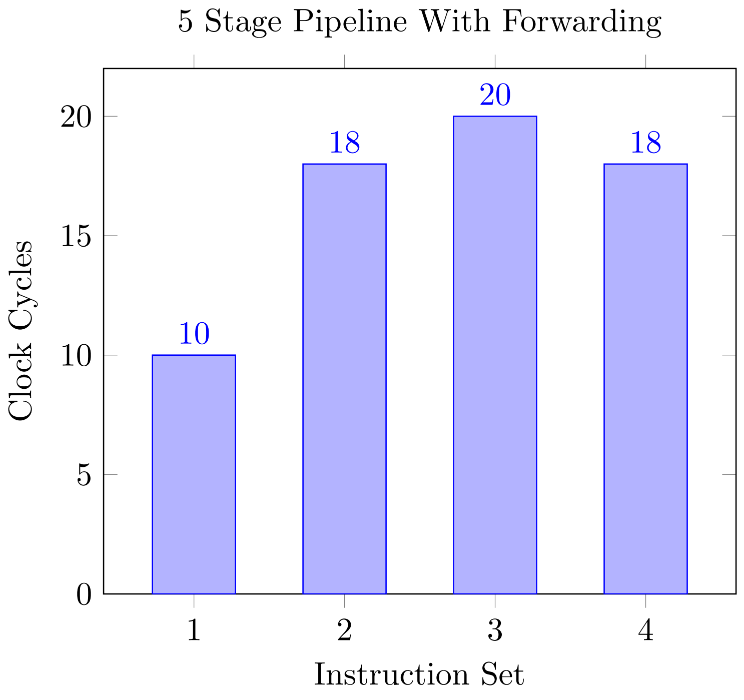
```
addi $1, $0, 2
addi $2, $0, 1
sw $1 , 1002($1)
sw $2 , 1006($1)
add $3, $2, $1
sub $4, $2, $1
sw $3 , 1011($2)
sw $4 , 1015($2)
lw $5 , 1002($1)
lw $6 , 1006($1)
lw $7 , 1010($1)
lw $8 ,  1014($1)
add $7, $7, $8
add $6, $6, $7
add $5, $6, $5
```

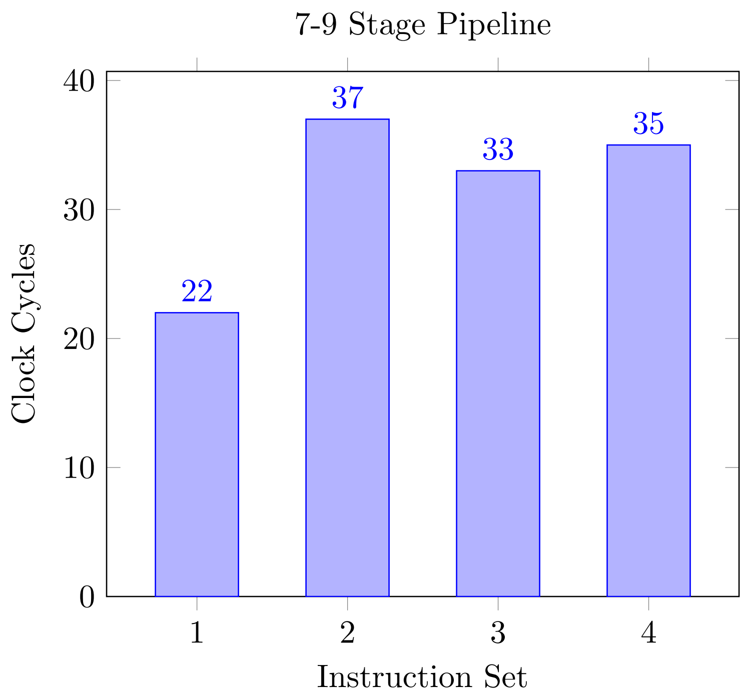
- Instruction Set 4

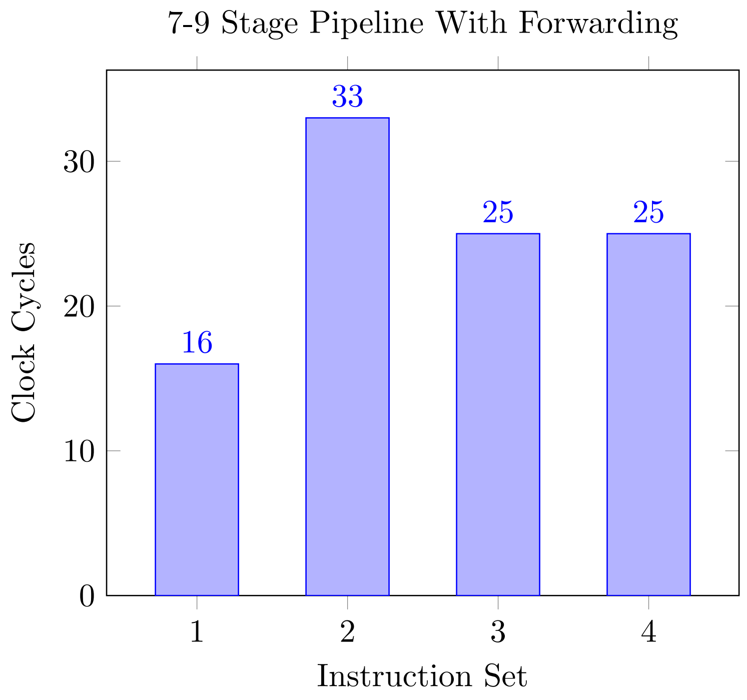
```
addi $t1, $0, 200
addi $t8, $0, 42
sw $t8, 8($t1)
addi $t3, $0, 3
addi $t4, $0, 12
addi $t7, $0, 100
sw $t7, 4($t1)
lw $t0, 4($t1)
add $t2, $t3, $t4
mul $t5, $t0, $t2
sw $t5, 4($t1)
lw $t6, 8($t1)
add $s0, $t6, $t5
```

### 3.1 Plots


5 Stage Pipeline
Clock Cycles
Instruction Set
16
22
27
27
0
5
10
15
20
25
1
2
3
4


5 Stage Pipeline With Forwarding
Clock Cycles
Instruction Set
10
18
20
18
0
5
10
15
20
1
2
3
4


7-9 Stage Pipeline
Clock Cycles
Instruction Set
0
10
20
30
40
1
2
3
4
22
37
33
35


7-9 Stage Pipeline With Forwarding
Clock Cycles
Instruction Set
0
10
20
30
1
2
3
4
16
33
25
25

### 3.2 Comments

- In case of all Instruction sets, we see an obvious decrease in the number of clock cycles if we compare code with forwarding for both 5 stage and 7-9 stage pipelines.

- If we see the % decrease in the number of clock cycles, we see that this decrease is minimum for Instruction set 2, as it has the highest proportion of branch instructions, which makes bypassing relatively less effective.

| Instruction Set | % Decrease in 5 Stage Pipelining | % Decrease in 7-9 Stage Pipelining |
|---|---|---|
| 1 | 37.5 | 27.27 |
| 2 | 18.18 | 12.12 |
| 3 | 25.92 | 24.24 |
| 4 | 33.33 | 28.57 |

Table 1: Improvement with Pipelining

### 3.3 Branch Prediction Accuracies

We Tested the Branch Prediction Accuracies of all the 3 strategies (Saturating, Branch History, Combination) with various initializations of the counters on branchtrace.txt . Here are their accuracies

| Initialization | % Saturating | % Global History Register | % Saturating + Global History |
|---|---|---|---|
| 00 | 79.01 | 70.43 | 72.62 |
| 01 | 83.92 | 70.62 | 80.47 |
| 10 | 87.95 | 70.98 | 80.10 |
| 11 | 86.67 | 71.16 | 80.83 |

Table 2: Branchtrace accuracies

Global History Strategy probably has less accuracy as compared to Saturating Counter Strategy because Global History makes predictions not dependent on the *pc* we are talking about.

## 4 Acknowledgements

- We would like to acknowledge that we used starter code from this repository written by Sayam Sethi.

- We used this book for our design decisions.