COL216 Assignment-2 Report

Amaiya Singhal
2021CS50598

Aanya Khurana
2021CS10084

General Implementation

- We have implemented each stage as a struct with attributes specifying its inputs and the control signals that we have made.
- All the functions that were present in the starter code, namely `add`, `sub`, `mul`,`slt`, `addi`, `bne`, `beq`, `j`, `lw` **and** `sw` have been implemented for all the parts.
- The termination condition for the pipeline is when every instruction has executed (exited the `Write` stage) and the pipeline is empty (all the stages are unoccupied).
- Control signals for each of the stages are used to determine whether that stage has to stall or if the stage is in a no-operation state. Apart from these there are some global control signals as well that are used by several stages.
- Separate functions are made for each of the stages which are run at every clock cycle (First clock cycle for the `write` stage and Second clock cycle for the other stages).
- Testing was done on several instruction pairs and all the public test case files available on Github.
- Design Decisions for all the pipelines - 5 stage, 5 stage with bypass, 7-9 stage and 7-9 stage with bypass have been written along with other implementation details, plots and tables in the following pages.
- Comparisons between these plots along with justifications have been provided.
- For the branch predictors, we have provided the accuracy tables along with implementation details for the functions `predict` and `update` for each of the 3 `structs`.
- For the branch predictors, we have also analysed the trends observed depending on both the predictor as well as the input file.
- We have compiled and explained our observations with justifications for different cases in branch prediction.
- The work split is mentioned at the end of the report.

PART A : 5 Stage without bypass

Implementation

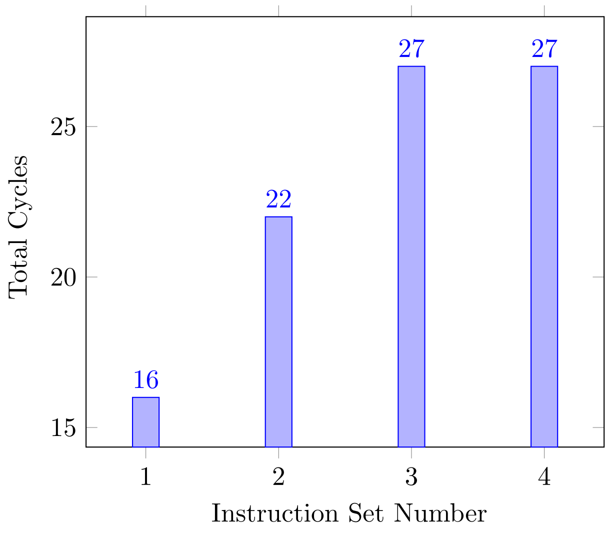
- In this part we implemented the standard 5 stage pipeline without bypassing.
- We made 5 stages using the struct that we had created.
- For every clock cycle, all the stages perform their respective functions. `WB` runs in the first half clock cycles whereas all the other stages run in the second clock cycle.
- Control signals were used to tell the stages when there is a stall and when there is a no-operation state.
- Since there is no bypassing in this case, we had to introduce stall cycles in the `ID` stage to wait until the correct values are written in the register/memory.

Testing

- We did a lot of testing on several instruction sets (all provided in the PDF + a large variety of new test cases including a lot of branches, loops, lw/sw instructions, data dependencies) and verified the results with manual calculations.
- Final testing was done on the 4 public test case files available on Github and the sample.asm file provided. Outputs were compared with peers and verified manually as well.
- The plot below shows the variation of the number of clock cycles taken for each instruction set.

Observations

Instruction Set	Test1	Test2	Test3	Test4	`sample.asm`
Clock Cycles	16	22	27	27	89



PART B : 5 Stage with Bypass

Implementation

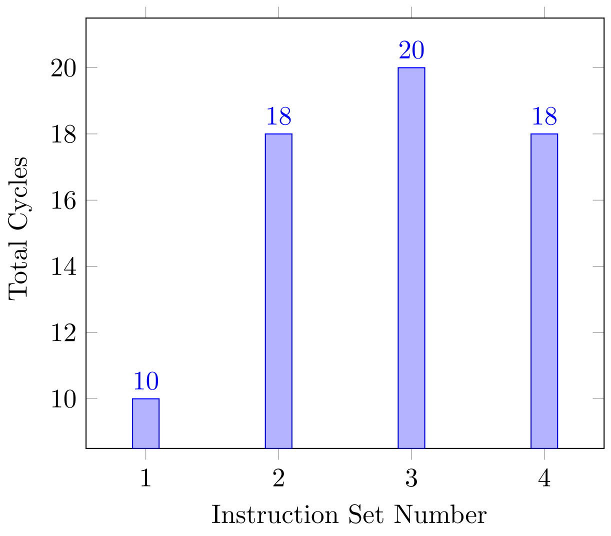
- In this part we implemented the standard 5 stage pipeline with bypassing.
- The stages are the same as PART A and work in a very similar manner.
- More control signals had to be introduced to keep track of data dependencies to avoid unnecessary stalls.
- We created a data structure to use as a latch where we update the computed values from the `EX` or the `MEM` stages (depending on the type of instruction) to be used by the further stages so they do not have to wait for the register values to get updated.
- Since there is bypassing in this case, there are less stalls and the total number of cycles is less than or equal to the number of cycles taken in Part A.

Testing

- We did a lot of testing on several instruction sets and verified the results with manual calculations similar to Part A.
- Final testing was done on the 4 public test case files available on Github and the sample.asm file provided.
- The plot below shows the variation of the number of clock cycles taken for each instruction set.

Observations

Instruction Set	Test1	Test2	Test3	Test4	`sample.asm`
Clock Cycles	10	18	20	18	88



PART C (a) : 7-9 Stage without Bypass

Implementation

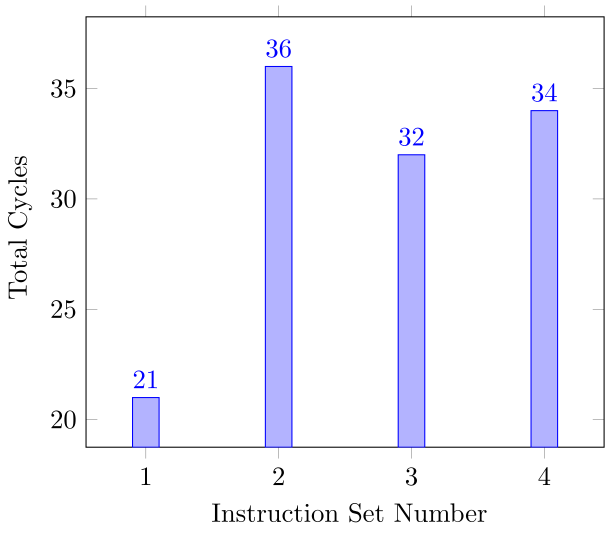
- In this part we implemented the 7-9 stage pipeline without bypassing.
- The initial stages `IF1`,`IF2`, `ID1`, `ID2`, `RR` are common for both the 7 and 9 stage pipelines.
- The 7 stage pipeline further has a `EX` and a `WB` stage.
- The 9 stage pipeline further has `EX`, `MEM1`, `MEM2` and `WB` stages.
- Since there is no bypassing in this case, we had to introduce stall cycles in the `ID` stage to wait until the correct values are written in the register/memory.
- Both the `WB` stages have a common write port hence there is a stall that occurs when both the 7 stage and 9 stage instructions are in their `WB` stages in the same clock cycle.
- In case a 9 stage instruction is followed by a 7 stage one, the `EX` stage stalls until the 9 stage instruction enters its `WB` stage so that the `write` operation of the 7 stage does not occur before that of the 9 stage. This is important to ensure that instructions are executed in the correct order (in-order implementation).

Testing

- We did a lot of testing on several instruction sets and verified the results with manual calculations similar to Part A.
- Final testing was done on the 4 public test case files available on Github and the sample.asm file provided.
- The plot below shows the variation of the number of clock cycles taken for each instruction set.

Observations

Instruction Set	Test1	Test2	Test3	Test4	`sample.asm`
Clock Cycles	21	36	32	34	141



PART C (b) : 7-9 Stage with Bypass

Implementation

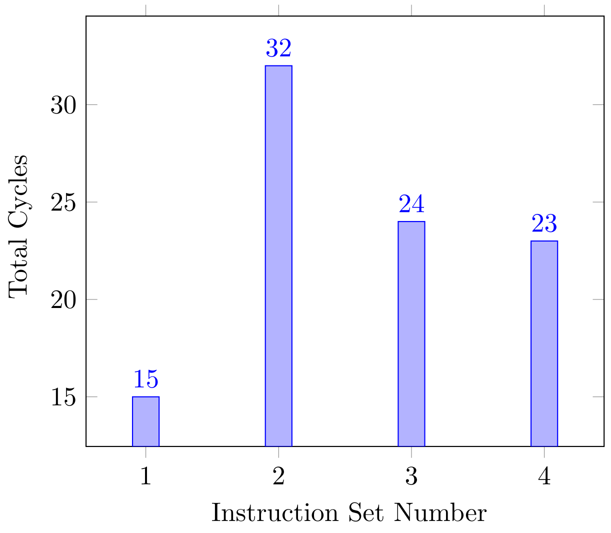
- In this part we implemented the 7-9 stage pipeline with bypassing.
- The stages are the same as in Part C (a).
- Since there is bypassing in this case, we had to introduce more control signals to ensure no unnecessary stalls are taking place.
- Since we have bypassing in this case, we have implemented a data structure that acts as a latch from where the stages can access the data without waiting for the correct value to be written in the registers/memory.
- In case a 9 stage instruction is followed by a 7 stage one, the `EX` stage stalls until the 9 stage instruction enters its `WB` stage so that the `write` operation of the 7 stage does not occur before that of the 9 stage. This is important to ensure that instructions are executed in the correct order.
- Stalls take place in the `RR` stage until the correct value is written into a latch from where the `EX` stage can access and do the calculations.

Testing

- We did a lot of testing on several instruction sets and verified the results with manual calculations similar to Part A.
- Final testing was done on the 4 public test case files available on Github and the sample.asm file provided.
- The plot below shows the variation of the number of clock cycles taken for each instruction set.

Observations

Instruction Set	Test1	Test2	Test3	Test4	`sample.asm`
Clock Cycles	15	32	24	23	140



PART D : Report with Cycle Comparison Plots

Comparisons Across the Plots

- The general trends for pipelines without bypassing V/S pipelines with bypassing are very clearly shown in the observations listed above. For all test cases, 5-stage and 7-9 stage pipelines with bypassing are faster than without bypassing: this is because we do not have to wait for writes into registers and intermediate values (stored in latches) can be forwarded into the latches of ALU/MEM stages for execution in later instructions, thus reducing the number of stalls.

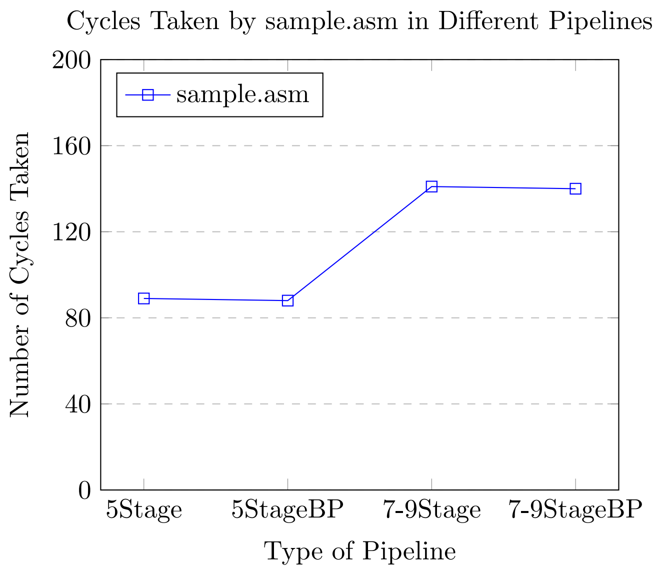
- The 7-9 stage pipeline takes more cycles than the 5 stage pipeline, due to the added number of stages. This difference is more pronounced in the case of instruction sets with more load-word/store-word instructions that use 9 stages due to memory accesses.

- The difference between the 5-stage pipeline (no bypassing) and 7-9 stage pipeline (with bypassing) is minor compared to other differences in these 4 cases. This difference is dependent on the instruction set: there is a tradeoff between the reduction in cycles due to forwarding and the increase in cycles due to added stages.

- Thus 7-9 stage pipelines (without bypassing) are the slowest and 5-stage pipelines (with bypassing) are the fastest. The exact magnitude of difference varies from instruction set to instruction set, and depends on the type, number, sequencing and branching in instructions.

- The comparitive plot drawn below shows the number of cycles taken per pipeline for the given sample.asm file. We observe that bypassing does not cause a significant difference due to the absence of data dependencies in the instructions in the loop, whereas there is a significant difference in the cycles taken by the 5-stage and 7-9 stage pipelines due to the looped instructions.

Plotting from data:



PART E : Branch Prediction

Implementation for `SaturatingBranchPredictor`:

- Corresponding to each branch, a 2-bit saturating counter is maintained which is hashed to the 14 least significant bits out of 32 (in the branch address) can be accessed in a table of 2^{14} entries.
- The prediction (taken V/S not taken) and updation of the counter solely depend on the bits of the counter.
- This training model is extremely efficient in the context of the assignment, indicating that the prediction of taken V/S not taken for a specific branch has a significant correlation to the history of that branch.
- We are accessing the counter of a specific branch, contrary to a general BHR in the second case (which may be true for sets of branches) and thus the accuracy is significantly better.

Implementation for `BHRBranchPredictor`:

- In this method, the Branch History Register (a 2-bit counter) is maintained and there is a one-one mapping from the BHR to a 2-bit saturating counter (stored in a table of 2^2 entries).
- The prediction depends on the counter corresponding to a given value of BHR (as in the case of the SaturatingBranchPredictor) and the updation includes both updating the counter corresponding to the BHR as well as updating the BHR itself, as the BHR captures the values of taken/not taken for the previous two instances in which the branch was called.
- The accuracy is lowest among these three as the predictor is trained on simply the 2-bit history of generic branches in the programme, not the history of a specific branch.

Implementation for `SaturatingBHRBranchPredictor`:

- This predictor is a combination of the elements of the first two, wherein counters corresponding to each of the 4 possible values of BHR are maintained per branch.
- The combination vector (size = 2^{16}) stores the 4 counters corresponding to each PC (14 bit), one for each value of BHR, and given a specific value (0/1/2/3) and branch PC, the counter corresponding to that BHR (a bitset generated from the value) for that PC is fetched from the vector. Prediction and updates follow the same logic as the BHR and Saturating predictors; the bits of the counter are used to make the prediction, and both the counter and BHR are updated accordingly.
- The accuracy is nearly same as that of the Saturating predictor (extremely high), as the counter uses both components of information to make predictions, and the extremely low difference can be attributed to the poor training of the predictor as implied by the size of the data (548 branches) compared to the number of entries in the vector (2^{16}).

Comparison between the different implementations

Accuracy Table

Initial Value	`SaturatingBranchPredictor`	`BHRBranchPredictor`	`SaturatingBHRBranchPredictor`
00 (0)	79.0146%	71.5328%	75.9124%
01 (1)	83.9416%	72.0803%	81.2044%
10 (2)	87.9562%	72.6277%	87.4088%
11 (3)	86.6788%	72.8102%	85.5839%

Comparison and Observations

- We tested the three predictors on different trace files, and observed that as the number of branches increases, the accuracy of the `BHRBranchPredictor` becomes apparently quite low compared to the accuracies of the `SaturatingBranchPredictor` and the `SaturatingBHRBranchPredictor`.
- For the given input file, `SaturatingBHRBranchPredictor` shows marginally higher accuracy than the `SaturatingBranchPredictor`, and the margin increases with the size of the data input on which the predictors are trained.
- These trends can be explained as follows: the more data is trained on BOTH the saturating counters, the more is their probability of giving correct predictions for specific branches, and the `BHRBranchPredictor` is very non-specific so the accuracy may not increase with the size of the output.
- While comparing the accuracies of the three predictors for the branchtrace file, we observed that initialising the BHR/counter with higher values (2 and 3) yielded better results than 0 and 1.
- This can be explained by the data in the given file: initially, a very large fraction of branches are taken, and patterns become more random in the subsequent branches, which explains why initialing the counter or BHR with the "taken" or "strongly taken" states yields better results.
- Upon varying the nature of these patterns in the input data files, we observed logical changes in the accuracies of each predictor for different initialising states. Moreover, as the taken/not taken values of branches became more and more periodic in nature (patterns like 001001. . . , 01100110. . . ..), the accuracy of the `BHRBranchPredictor` increased significantly, as the BHRs could train their counters with remarkable efficiency.
- Accuracy is a function of both the input and predictor: the larger is the input, the better is the predictor trained on the data.
- Comparative accuracies among different predictors for the same data depend on the nature of the data (the sequences of branches and their taken/not taken values) whereas comparative accuracies among different data sets for one predictor depend both on the nature and size of the data sets, as well as the functionality of the predictor itself.
- As the sizes of the testing input files became larger and larger (more data to train the models), accuracies for the `SaturatingBHRBranchPredictor` and the `SaturatingBranchPredictor` began touching 92-93%, while the accuracy for the `BHRBranchPredictor` stayed at roughly 73% (in the general case).

Work Split between the Partners

The work split between the partners is 50-50.