COL216 Assignment 2

Kushagra Gupta (2021CS50592)
Parth Patel (2021CS10550)

April 15, 2023

§1 Cycle Comparison Plots

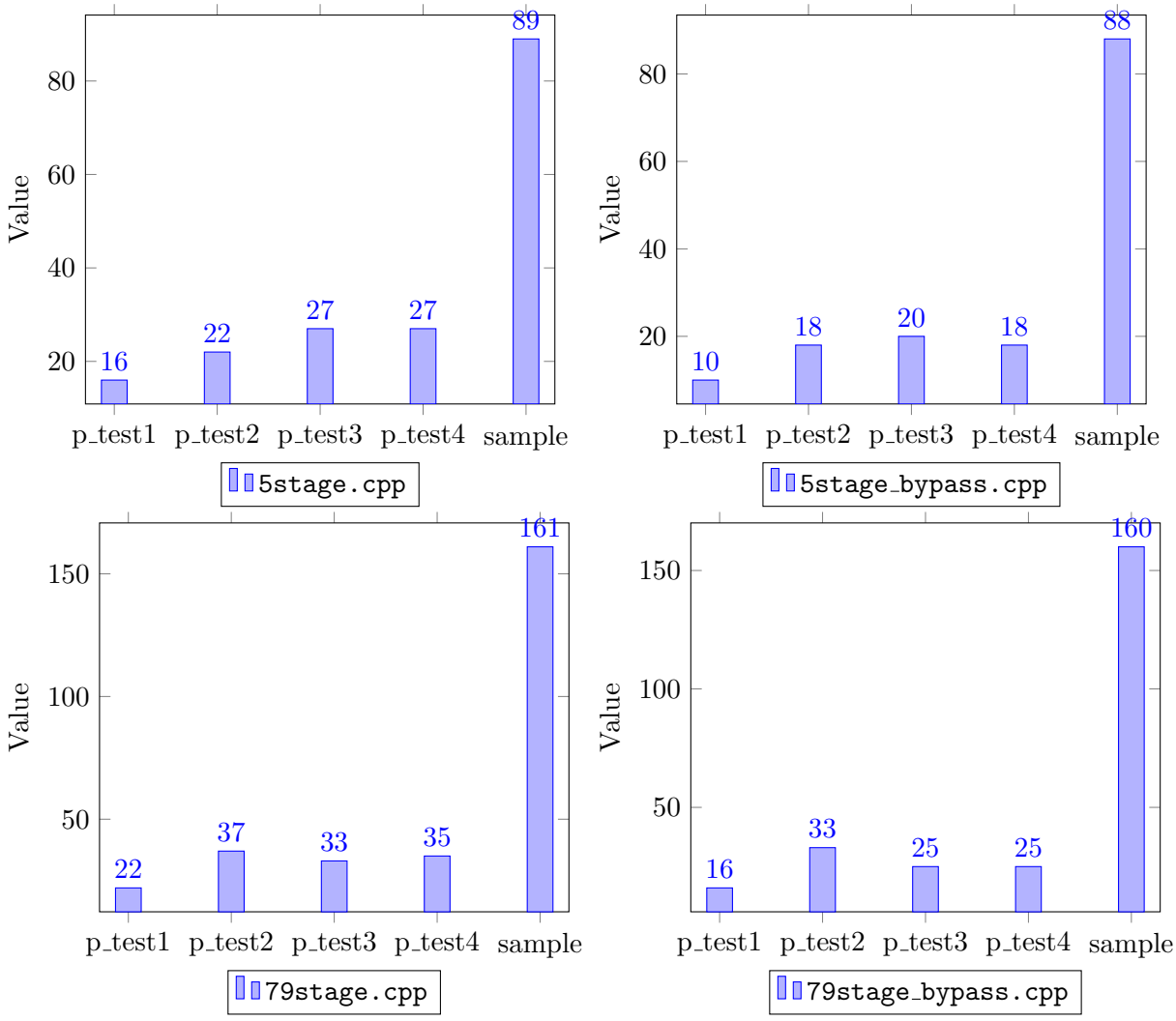
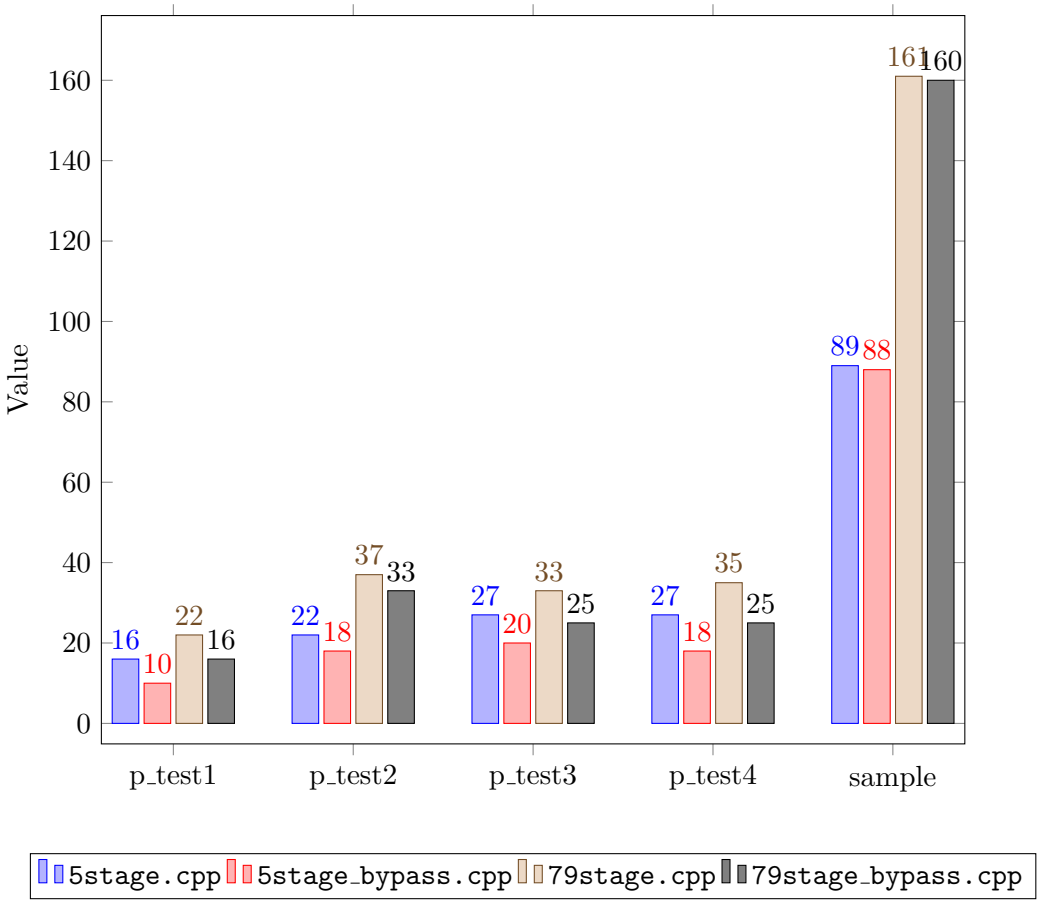


Figure 1: 4 side-by-side bar graphs for the 5 and 7-9 stage pipeline



§1.1 Analysis of the cycle comparision plots

We ran the given 5 files (sample.asm, public_test1.asm, public_test2.asm, public_test3.asm, public_test4.asm) and observed the no.of cycles taken as we vary the type and size the of input files and size of input files to obtain the above plots.

§2 Our Pipelined Design

§2.1 5 stage without forwarding

Our design for the 5 stage pipeline involved use of 5 structs representing the `IF, ID, EX, MEM and WB` stages whose instances `S1, S2, S3, S4 and S5` are created.
The control flow is managed by using the while loop for the clock cycles and idle markers allow execution of different structs in one cycle. All the 5 stages are run parallelly in 1 cycle, starting from S5 to S1.
Continue statements allow us to represents stalls, and subsequently the further instructions below in the pipeline are also stalled by means of our design. The `idle` markers are set to 1, as soon as we do not encounter any more instructions, and we make use of an `acquired` array for dependencies.

§2.2 5 stage with forwarding

The forwarding design changes slightly that we assign `INT_MIN` to the dependent instructions and forward them, and consequently we fetch them, by storing the produced

outputs in an array `Latches` along with the register numbers.

§2.3 7-9 stage without forwarding

Our design for the 9 stage pipeline involved use of 9 structs representing the `IF1, IF2, Dec1, Dec2, ID, EX, MEM1, MEM2 and WB` stages whose instances `S1, S2, S3, S4 and S5, S6, S7, S8 and S9` are created. (S10 is an additional instance to store values at end of clock cycles.

§2.4 7-9 stage with forwarding

The forwarding design changes slightly that we assign `INT_MIN` to the dependent instructions and forward them, and consequently we fetch them, by storing the produced outputs in an array `Latches` along with the register numbers.

§2.5 Design Decisions for the 7-9 stage Pipeline

1) The memory store in sw is done in the second MEM2, and similar fetch for the lw instruction. So the forwarding from lw instruction can be done in the second half of the memory stage.

2) We have assumed that the sw takes 9 stages to execute. So, it will pass the last WB stage, and hence even other instruction cannot do WB in its same cycle. This would only be possible in a 7-8-9 stage pipeline. Our design agrees with the Program order execution.

§3 Branch Predictors

§3.1 Designing Saturating Branch Predictor

Each entry in `table` is a 2-bit saturation counter. We map each branch to an element or entry of `table` by using the modulo operator to get the index of `table`, which is `pc%table.size()`

To `update` the predictor, two inputs are given, i.e. `pc` and `taken`. Since `pc` can't exceed `table.size()` (i.e. `1<<14`), we take modulo to map the branch to a `counter` in the `table`. We then increment or decrement the `counter` depending on whether the branch is `taken` or not.

To `predict` the outcome of a branch, we check whether `table[pc%table.size()]` is greater than 1 (i.e. either `10 or 11`).

§3.2 Designing Branch History Register (BHR) Predictor

`bhr` is a global register that stores the values for the last two branches. `bhr` can have four possible values, and each of the possible values is mapped to `bhrTable`. Each of the four elements of the `bhrTable` is a saturating counter.

To `update` the predictor, two inputs are given, i.e. `pc` and `taken`. Since `bhr` is a

global register, we don't require `pc`. We simply increment or decrement counter corresponding to `bhr` in the `bhrTable`. Also, we change the `bhr` value depending upon the value of `taken`.

To `predict` the outcome of a branch, we check whether `bhrTable[bhr.to_ulong()]` is greater than 1 (i.e. either `10 or 11`).

§3.3 Designing Combined Predictor

Each branch is mapped to an entry of `table` consisting of `1<<14` entries. An entry of the `table` denotes a counter from `00` to `11`. Each element of this `table` is mapped to 4 entries of `combination`. Each element of `combination` is itself a saturation counter.

To `update` the predictor, two inputs are given, i.e. `pc` and `taken`. Since `pc` can't exceed `table.size()` (i.e. `1<<14`), we take modulo to map the branch to an element in the `table`. Let `x` be the corresponding index of `combination` which we compute and store. Then, depending upon the branch, if it is `taken` or `not taken`, update the `table` entry. And finally update `combination[x]` by incrementing or decrementing it depending upon the boolean `taken`.

To `predict` the outcome of a branch, we check whether `combination[x]` is greater than 1 (i.e. either `10 or 11`) where `x` is the same as defined in the previous paragraph.

§3.4 Accuracy analysis for all three predictors

Depending on the initialization of counters, we get different accuracy for each possible counter's initial value. Note - we have chosen `combination size` as `1<<16` for this table. We have documented the results for the `branchtrace.txt` file in the form of a table -

Initial Counter value	(2 bit) Saturating	BHR	(2 bit) Saturating BHR
0	79.0146%	71.5328%	70.073%
1	83.9416%	72.2628%	77.9197%
2	87.9562%	72.6277%	84.6715%
3	86.6788%	72.8102%	80.292%

The Analysis for 2 bit saturating BHR is done by assuming `combination` size is `1<<16`. The sample trace given only had around 500 branches because of which the accuracy of the combined predictor is little less than 2 bit saturating counter. If size of `combination` was choosed to be `1<<3` for the given sample trace then the accuracy would have been relatively better as illustrated in the following table -

Initial Counter value	`combination size 1<<16`	`combination size 1<<3`
0	70.073%	80.4745%
1	77.9197%	85.219%
2	84.6715%	86.1314%
3	80.292%	84.4891%

We also tried to test our branch predictor on much bigger trace (26.6 MB) and we got the accuracy of the combined branch predictor much better than 2 bit Saturating and BHR branch predictors. Note - we have chosen `combination size` as `1<<16` for this table

The results for this trace are documented in the following table -

Initial Counter value	(2 bit) Saturating	BHR	(2 bit) Saturating BHR
0	80.1646%	63.3784%	95.7163%
1	80.0648%	63.3784%	95.7195%
2	80.2073%	63.3785%	95.719%
3	80.2053%	63.3786%	95.7171%

§3.5 Results and observations

The 2 bit saturation counter Branch Predictor predicts by the "tendency" or "inertia" of the results of past.
The BHR Branch Predictor is less accurate than the other two. This is mainly because it has a Global register to store the history. So every branch will modify that same Global register (irrespective of the `pc`), and this will introduce interference and hence less accurate results than that by using Local registers to store history.

The combined Branch Predictor is a little complex and hence the accuracy or improvement from the Saturation Branch Predictor can only be seen when the number of branches to "train" are high (we tested for a large trace that we found on `github.com/saivittalb`). The implementation is very accurate because we are using many local registers to record history of branches instead of a single global register to store history for all the branches. This history of 2 bits concatenated with the branch number can be have much more information than that of the Saturating Branch Predictor alone.

§4 Token Distribution

Both team members made equal contributions and efforts. Hence a 50-50 split