INDIAN INSTITUTE OF TECHNOLOGY, DELHI

ASSIGNMENT DESIGN

Problem Statement
Memory Request Re-Ordering for optimal DRAM utilization.

ARNAV TULI | ENTRY NO. 2019CS10424

APAR AHUJA | ENTRY NO. 2019CS10465

Course - COL216 | Prof. Preeti Ranjan Panda

Compiled on April 10, 2021

COL216 - Assignment 4

Arnav Tuli | Apar Ahuja

April 10, 2021

Note: Much of the base design/approach was implemented in Assignment-3 and Minor and is still the same. Therefore, in this document we will only mention the new design decisions/approaches used, and some modifications that we made to the previous code.

1 Approach and Design

1.1 Input - Output

- **INPUT:** **main.cpp** takes input from a *.txt* file, which is given as a command line argument. It also takes the *ROW_ACCESS_DELAY*, *COL_ACCESS_DELAY* and *MODE* as additional command line arguments.
- **OUTPUT:** The output format follows the one specified in the problem statement, more precisely:
.   - Each clock cycle **activity** is displayed. This includes:
.   - Cycle **count**
.   - Specific **instruction** executed (like add $t0, $t0, $t1; etc.)
.   - Memory address of **instruction** executed
.   - Modified **registers** (with their values), if any
.   - **DRAM activity**:
.   - issuing request (store/load),
.   - starting to process a DRAM request,
.   - finish processing a DRAM request,
.   - row activation and/or row writeback,
.   - reading/writing operations
.   - Modified **memory** locations (with their values), if any
.   - At the end of program execution, relevant **statistics** are displayed. These include:
.   - Total number of **clock cycles**
.   - Number of times a particular instruction was executed (like add, addi, lw, sw etc.)
.   - Accessed/Modified memory locations with their values
.   - *ROW BUFFER* operations:
.   - Row **writeback**
.   - Row **activation** (buffer update)
.   - Data **write** (buffer update)
.   - Data **read**
.   **Note:** All integer values (register and memory) are displayed in both *decimal* and *hexadecimal* formats.
.   **Note:** *Simultaneous* clock cycle activity is shown in separate lines in the output for clarity.
.   **Note:** In case of error (overflow, syntactic, etc.), relevant error message is displayed and execution is stopped.

1.2 MIPS Memory (DRAM IMPLEMENTATION)

According to the problem specifications, the total memory available to MIPS program is 2^{20} bytes. The memory model that we are using is that of a **DRAM** (as specified). DRAM can be thought of as a 2-dimensional array of bytes, with **1024 rows** and **1024 columns**. In order to implement DRAM in our

code, we are using an **integer array** (each integer location represents a byte) of size 2^{20}. This way we are moulding the *2-D DRAM* into a *1-D array* in **row-major format**. For our design, we have partitioned this memory into two sections:- **Instruction memory** and **Data memory**. Instruction memory spans from address 0 to 699391 (basically first **683 rows** of DRAM), whereas, Data memory spans from address 699392 to 1048575 (basically the remaining **341 rows**).
The division is nearly 2:1, which is justifiable, as for every memory location a minimum of 2 instructions will be required (lw and sw). Hence, the division cannot be more optimal, while keeping the implementation general. Also, to account for partitioning the memory, a constant **DATA_START** (= 699392) is added to the address provided in lw and sw commands for smooth memory/value access.
Coming to the other part of our implementation, we have created a **ROW_BUFFER**, which is an integer array of size **1024**. This array basically models the buffer that acts as an interface between the main memory (DRAM) and the processor. Whenever, a **lw** or **sw** operation is to be executed, a DRAM request is issued, and contents of the DRAM (basically a **row**) gets activated and is copied to this buffer. Similarly, writeback operations are also performed from this buffer to DRAM when a new row is to be copied, or at the end of the program execution.
Note: Instruction and Data storage format in DRAM is the same as it was in Assignment-3, and hence is omitted from this section.

1.3 Design

Note: Only additions and modifications over previous design (Assignment-3) are being mentioned here.
- Global Variables:
. - **PC:** program counter that represents the memory position of next instruction to be executed.
. - **PARTITION:** represents the end of instructions stored in memory
. - **DATA_START:** represents the starting address of data section in memory (699392 by default)
. - **ROW_ACCESS_DELAY:** used to store the row access delay value (DRAM)
. - **COL_ACCESS_DELAY:** used to store the column access delay value (DRAM)
. - **MODE:** bool used to store the operating mode (true for non-block memory access, false otherwise)
. - **DRAM[1048576]:** memory array representing the MIPS main memory (DRAM) of size 2^{20} bytes
. (in row major format).
- structure Request: request data structure that stores all the information relating to a lw/sw request.
. - **addr:** this is the DRAM address (DATA SECTION) that the lw/sw wants to access
. - **row:** this is the DRAM row (DATA SECTION) that the lw/sw wants to access
. - **data_bus:** this is the bus that stores the value that needs to be written on the DRAM (**addr**)
. (in case of **sw** request)
. - **destination:** this is the integer code of the register on which write operation needs to be performed
. (in case of **lw** request)
. - **type:** string used to store type of request (”lw” or ”sw”)
. - **instruction:** string used to store the actual assembly instruction executed (sw/lw)
. - **PC:** this stores the memory address of the actual assembly instruction
- structure Queue: queue data structure that stores all pending DRAM access requests (lw or sw).
. - **MemToAdj:** hash map that maps given data memory row to its corresponding rows in **Adj**
. - **Pread:** hash map that stores the number of times value is being read from a given memory address
. (**key**). It also tells the latest queue row in which the read access is requested for that memory address
. (lw command).
. - **Pwrite:** hash map that stores the number of times value is being written to a given memory
. address/register (**key**). It also tells the latest queue row in which the write access is requested for that
. memory address/register (sw/lw respectively).
. - **reverseMap:** hash map that maps given integer to its corresponding register. eg. num_reg[8] = ”$t0”
. - **Adj:** vector of queue that is used to store all pending DRAM requests in a **queue**-like manner
. - **adjIndex:** stores the index of the **current** queue-row in **Adj** vector (0-indexed)
. - **isEmpty:** function that is used to check if **Adj** is empty or not (returns true if empty, false otherwise)
. - **BinarySearch:** function that implements binary search to find the index of least integer greater than
. or equal to **key** in a vector of integers

. - **addRequest:** adds a request to **Adj** in an optimal manner, appending it to the **topmost safest** queue.
. - **getRequest** function that is used to get the foremost request in the queue vector (**Adj**) for processing
. - **deleteRequest:** function that is used to delete the foremost request in **Adj** after its processing is
. complete
- structure REGI: base structure that helps in integrating memory with processor and registers
. - **num_reg:** hash map that maps given integer to its corresponding register. eg. num_reg[8] = "$t0"
. - **execution_stats:** function to print the relevant statistics at the end of execution (mentioned in **1.1**)
. - **instructions:** string vector used to store each and every instruction present in the input file
. - **ROW_BUFFER:** integer array of size 1024 that implements the **ROW_BUFFER** mentioned in **1.2**
. - **start** and **end:** used to store the starting and ending address of the memory row stored in buffer
. - **isEmpty:** bool used to store whether or not the buffer is empty (**true** if empty, **false** otherwise)
. - **memoryAddress:** integer vector used to store accessed memory addresses
. - **writebacks, copies, value_read, value_write:** store respective row buffer operation count
. - **doWriteback:** bool used to check whether or not a writeback should be done at the end of program
. - **loadBuffer:** function used to load a DRAM row into ROW_BUFFER
. - **raiseRequest:** function that raises a new DRAM request (either lw or sw) and adds it to the **Queue**
. - **NonBlockLW:** function that implements non-blocking memory access feature
. - **lw** and **sw:** (modified) to make use of **NonBlockLW** and **raiseRequest** functions.
. - **inPwrite:** function that checks for pending writes on registers (maximum 3 registers).
. - **executeIndependent:** executes independent instructions while waiting for DRAM response
. - **buffer:** function used to do an optional writeback at the end of the program.
- function simulator: (modified) now also keeps track of the DRAM request **queue**. DRAM keeps on
. processing new requests while the queue is non-empty
- function removeSpaces: removes trailing and leading white spaces from an instruction (string)
- function addToMemory: (modified) now also stores instruction (string) in the **instructions** vector (after removing redundant white spaces)

1.4 Pending Read and Pending Write

By the word **Pending**, we basically mean that an operation is yet to be performed by the DRAM, in case a request has been issued. Pending **write** on a **location** means that the processor is still waiting for the DRAM response, responsible for writing new data on that location. From this definition, we can see that the location can either be a memory address (DATA SECTION), or a register, as we can request DRAM access so as to write on a memory location (**sw**) or a register (**lw**). Pending **read** from a **location** means that the processor is still waiting for the DRAM response, responsible for reading data from that location. From this definition, we can see that the location can be a memory address (DATA SECTION), as we can request DRAM to read from a memory location (**lw**). Note that there is no notion of Pending read from a **register**, as the processor can readily access the register values, and need not wait for DRAM response to do that. So, it is assumed that the read operation (from a register) is performed by the processor in a single clock-cycle. (No pending read)

1.5 Notion of Safety: Safe Instructions

The processor fetches instructions from memory in a sequential manner, and checks if the instruction is **safe** or not. If it safe, then the command is executed, and PC is incremented/decremented appropriately, otherwise, the processor stops further execution until the instruction becomes safe to execute. The notion of safety for each instruction present in the ISA is given below:

- *add, sub, mul, slt:* These instructions involve three register parameters. Hence, executing them will mean **reading** data from two registers, performing some operation, and **writing** acquired data on the third register. Since, read/write operations need to be performed on the registers, it must be ensured that the data present in the registers is up-to-date, otherwise, the program output can be erroneous.

Hence, there should not be any **Pending write** on any of the three registers in question, for the instruction to be **safe** to execute.

- *beq, bne, addi:* These instructions involve two register parameters. Hence, executing them will mean **reading** data from one/two registers, performing some operation, and optionally **writing** acquired data on the second register (in case of addi). Since, read/write operations need to be performed on the registers, it must be ensured that the data present in the registers is up-to-date, otherwise, the program output can be erroneous. Hence, there should not be any **Pending write** on any of the two registers in question, for the instruction to be **safe** to execute.
- *j:* This instruction doesn't involve any register and data memory address, hence, it is always **safe** to execute. (there are no additional dependencies)
- *sw:* This instruction involves two register parameters, both of which are to be read from. Hence, executing (raising DRAM request) it will mean **reading** data from two registers, performing some operation (address calculation, waiting for DRAM response), and then **writing** acquired data at the given memory address. Since, read operation needs to be performed on the registers, it must be ensured that the data present in the registers is up-to-date, otherwise, wrong DRAM request may be issued. Hence, there should not be any **Pending write** on any of the two registers in question, for the instruction to be **safe** to execute (raise DRAM request).
- *lw:* This instruction involves two register parameters, one of which is to be read from, and other is to be written onto. Hence, executing (raising DRAM request) it will mean **reading** data from address register, performing some operation (address calculation, waiting for DRAM response), and then **writing** acquired data on the other register. Since, read operation needs to be performed on one of the registers, it must be ensured that the data present in that register is up-to-date, otherwise, wrong DRAM request may be issued. Hence, there should not be any **Pending write** on the address register (present inside parenthesis), for the instruction to be **safe** to execute (raise DRAM request).

Note: Instructions which are not safe are said to be **unsafe**. Generally dependent instructions are **unsafe** to execute. By dependent instruction, we mean any instruction in the ISA that uses a register (having a pending write) either as a source (read from) or as the target (write to). In other words, we consider dependent instructions to be **unsafe** for execution. The reasoning behind this choice is justifiable, as the execution needs to be sequential. Therefore, we should not access (read) the value of the register before we actually load some value into it using lw command. If we do not consider **read register** operation as unsafe, then we will end up reading erroneous values, as the actual value is still not loaded. Also, we should not write some value on to the register, as this can have a corrupting effect. For e.g. rather than overwriting the loaded value with some other value, we can end up overwriting some other value with the loaded value. Hence, dependent instructions need to be unsafe for our implementation to be correct and consistent with the actual program semantics. One exception to this is the **lw** command itself, which is not considered to be unsafe, when there is a pending write on the target register. This is because, the queue-insertion algorithm takes care of this dependency while adding the request to the queue. In other words, it is ensured that the program semantics remain unchanged.

1.6 Queue Reordering IMPLEMENTATION

Request and **Queue** data structure have already been described in section **1.3**. In this section, we describe our reordering algorithm that is used in request-insertion step. Since, a DRAM request is raised when PC encounters either a sw/lw command, we describe the insertion algorithm for each of them separately:
Store Word request: Main components of a store word request are memory **addr**ess (DATA SECTION), memory **row** and **data_bus**. data_bus consists of the data that was read from the desired register, and which needs to be written at the given memory address (DATA SECTION). Now, there are 3 cases that can occur:

- **row** is **not** present in **MemToAdj** map. In this case, we need to add **row** to **MemToAdj**, and create a new queue-row containing the given request and append it to **Adj** vector.
- **row** is present in **MemToAdj** map, but there are **no** pending read/write on **addr**. In this case, we determine the queue-row (in **Adj**) using **MemToAdj** map, and append the request to the corresponding queue, thereby, bypassing following queue-rows (if any). This way, we have optimally reordered the queue-vector, and ensured that DRAM requests accessing same memory row are processed together.
- **row** is present in **MemToAdj** map, but there **are** pending read/write on **addr**. In this case, we determine the queue-row (in **Adj**) using **Pread** and **Pwrite** maps, with **addr** as the key. These map return the index of the queue-rows in **Adj**, where a read/write request to same memory address has been issued. In order to ensure that our implementation is correct, we choose the maximum of the two possible locations, and append the request to the corresponding queue, thereby, bypassing following queue-rows (if any). This way, we have optimally reordered the queue-vector, and ensured that DRAM requests accessing same memory row are processed together, while maintaining the semantics of the program. (basically, we cannot interchange processing order of two write commands, nor can we interchange the processing order of a read and write command).

Load Word request: Main components of a load word request are memory **addr**ess (DATA SECTION), memory **row** and **destination**. destination consists of the integer code of the register onto which the write operation needs to be performed. Now, there are 3 cases that can occur:

- **row** is **not** present in **MemToAdj** map. In this case, we need to add **row** to **MemToAdj**, and create a new queue-row containing the given request and append it to **Adj** vector.
- **row** is present in **MemToAdj** map, but there are **no** pending writes on **addr** or on **destination**. In this case, we determine the queue-row (in **Adj**) using **MemToAdj** map, and append the request to the corresponding queue, thereby, bypassing following queue-rows (if any). This way, we have optimally reordered the queue-vector, and ensured that DRAM requests accessing same memory row are processed together.
- **row** is present in **MemToAdj** map, but there **are** pending writes on **addr** or on **register**. In this case, we determine the queue-row (in **Adj**) using **Pwrite** map, with **addr** and/or **destination** as the key. These map return the index of the queue-rows in **Adj**, where a write request to same memory address or same register has been issued. In order to ensure that our implementation is correct, we choose the maximum of the two possible locations. In case, the queue-row doesn't correspond to memory **row**, we find the first next queue-row corresponding to **row** by using **binary search** on **MemToAdj[row]** vector, and append the request to the corresponding queue (in **Adj**), thereby, bypassing following queue-rows (if any). This way, we have optimally reordered the queue-vector, and ensured that DRAM requests accessing same memory row are processed together, while maintaining the semantics of the program. (basically, we cannot interchange processing order of two write commands, nor can we interchange the processing order of a read and write command. However, we can interchange the processing order of two read commands provided they do not have a write command in between, hence, we do not consider pending read on **addr**, even though it may also be there. For example, two **lw** instructions acting on the same **addr**, but different registers can be reordered, without changing the program output).

1.7 Non-Blocking Memory Access IMPLEMENTATION

Store Word: When we execute a store word operation, our interpreter raises a DRAM request. This **request** consists of memory address (DATA SECTION), memory row, data_bus value, type etc. So, basically, we have assumed that address calculation, data transfer to data bus and raising request takes place in a single clock cycle. The request will either wait or get processed from next clock cycle onwards based on its position in queue vector. Apart from this usual execution procedure, we have also implemented a non-blocking strategy. So, what our design decision is that while waiting for the DRAM response, the processor will continue executing subsequent instructions. This simultaneous execution continues till it reaches an **unsafe** instruction or it got the response from DRAM. In case, it encounters an unsafe instruction, the processor stops further execution and awaits DRAM response before executing it. Also, as we have already stored the register's value in **data_bus**, we need not worry about register contents being overwritten in subsequent instructions and consequently memory corruption.

Load Word: When we execute a load word operation, our interpreter raises a DRAM request. This **request** consists of memory address (DATA SECTION), memory row, destination register code, type etc. So, basically, we have assumed that address calculation and raising request takes place in a single clock cycle. The request will either wait or get processed from next clock cycle onwards based on its position in queue vector. Apart from this usual execution procedure, we have also implemented a non-blocking strategy. So, what our design decision is that while waiting for the DRAM response, the processor will continue executing subsequent instructions. This simultaneous execution continues till it reaches an **unsafe** instruction or it got the response from DRAM. In case, it encounters an unsafe instruction, the processor stops further execution and awaits DRAM response before executing it.

1.8 Strengths and Weaknesses of our implementation

Strengths:

- Our implementation of queue-reordering is optimal in the sense that given a set of DRAM requests (raised in a sequential order), our **insertion algorithm** inserts the request into the queue in such a manner that the final reordering requires the least number of row writebacks/activation. Hence, given any MIPS code, our design efficiently reorders the requests and requires the least number of clock cycles, given the ROW_ACCESS_DELAY and COL_ACCESS_DELAY. This optimal nature of our reordering holds for the assumptions that we have taken regarding safety of an instruction, which have already been mentioned in section **1.5**.
- Another strength of our approach is that we have made use of data bus while raising request for a **sw** instruction. Due to this, there is actually **no** notion of **Pending read** from a register. In other words, the processor will never have to wait to write a value onto the register, which was used in a previous **sw** instruction. Hence, the clock cycle count reduces further.
- We have implemented non-blocking memory access, hence, processor can execute **safe** instructions while waiting for the DRAM response. This also includes raising multiple DRAM requests, while DRAM is still processing one. Hence, the processor doesn't stop its processing when it encounters a lw/sw after another lw/sw, rather it carries out its usual execution, given that the instruction is safe. This feature in turn greatly reduces the clock cycle count.
- Also, one more advantage of our queue implementation is that the reordering takes place only at the time of insertion of a request into the queue. In other words, we do not reorder the queue every clock cycle depending on the instruction fetched by the processor. Due to this reason, DRAM does not have to wait for reordering to take place before it can start processing. It can fetch a request from queue and start processing it, as soon as it is available. This in turn reduces the **idle time** of DRAM, and lowers down execution time.

Hence, our implementation effectively tries to tackle the increased runtime, due to slow sw/lw DRAM response, by reordering and non-blocking memory access.

Weaknesses:

- Our approach of reordering DRAM requests using a **Queue** data structure has its own share of weaknesses. The most prominent one being the **memory requirement**. Normally, a DRAM implementation without any queue of requests will only require 2^{20} bytes of memory, which is basically the DRAM available to us. However, now we also require an **additional** data storage mechanism (between processor and DRAM), that will actually store all the DRAM requests (and **dependency data**) that are raised while DRAM is still processing a particular request. Also, each **request** has its own data components, which include- memory address, row, data_bus, destination, etc. So, if there are **n** DRAM requests (sw/lw) that are raised and present in the queue at a given point in time, then the additional data memory required of our implementation will be $\mathbf{O(n)}$. This is in **contrast** with the **naive** implementation of DRAM access without forming queue of requests, which has a space efficiency of $\mathbf{O(1)}$.

- Apart from this, there is a **complexity-cycle count** trade-off that takes place in our implementation. In order to find an optimal reordering of DRAM requests at run-time, we have developed a **queue-insertion algorithm**, that has an element insertion time complexity of $\boldsymbol{\Theta(\log n)}$ in the worst case, as we may use binary search on a vector of size $\boldsymbol{\Theta(n)}$ during insertion step (lw). This is in contrast with $\mathbf{O(1)}$ time complexity of insertion in a normal queue, without reordering.

- Now, as it can take more time to insert a request in the queue, it may so be the case that the current **clock period** is not sufficient for complete execution of a sw/lw command (just the request part). Hence, we may have to **increase** the clock period in order to accomodate this. This in turn will make all the instructions slower. This increase in runtime will be more pronounced, when the program only consists of normal instructions. However, in case of lw/sw instructions, due to optimal reordering of DRAM requests at runtime, the total clock cycle count of a MIPS program is also decreased, therefore, total runtime should also decrease (as expected decrease in cycle count is much more than increase in clock period). Hence, this is another plausible drawback of our implementation.

- Another minor weakness of our implementation is that the **sw** instruction becomes unsafe to execute if there is a pending write on the source register (read from). This could have been avoided, if we had not introduced the notion of data bus, and reordered the queue accordingly. However, having a data bus has its own **advantage**, therefore, we went ahead with this approach.

Note:
- **Space** issue can be resolved if we have access to some auxiliary memory (other than DRAM), which can act as an intermediate between processor and DRAM.
- **Time** issue can be resolved even without increasing clock period, as we can make raising request (lw) a **two-cycle** operation. This way, we will not end up penalizing all other instructions as well.
- **Reordering** queue every clock cycle may give better result on some particular cases, but not in general. This is because the ROW_ACCESS_DELAYS and COL_ACCESS_DELAYS are generally big enough that decrease in cycle count due to dynamic reordering is over compensated by increasing row writebacks/activation. Also, increase in DRAM **idle time** in another drawback.

1.9 Algorithm

Note: The algorithm related to parsing, storing instructions in the memory (DRAM) and executing instructions other than sw/lw is the same as in Assignment-3, and is therefore, not mentioned in this section. Here, we are only mentioning the changes done to sw/lw instructions and simulator function.

-simulator:
. - Check if **PC** < **PARTITION**. If it is, fetch an instruction from memory, otherwise, stop execution
. - Check the fetched instruction. If it is **safe**, then execute it, otherwise, stop execution temporarily
. - Check the instruction to be executed. If it is other than sw/lw, then execute it normally
. (as in Assignment-3), otherwise, **raise** a DRAM request and add it to the queue-vector
. - Check if the DRAM queue is empty or not. If it is empty, then repeat above procedure as long as
. **PC** < **PARTITION**. Otherwise, enable DRAM processing and non-blocking execution of instructions
. - Once queue becomes empty, repeat above procedure as long as **PC** < **PARTITION**.

-NonBlockLW:
. - Pop a DRAM request from queue-vector and start processing it.
. - Check if the row buffer is empty or not
. - If it is empty, activate the row and then perform column access
. - Else, check if the required row is already there in the row buffer
. - If the row is already there, then just do column access

. - Else, writeback the row onto DRAM, and perform row activation and column access
. - Meanwhile, if MODE is **true** execute subsequent **safe** instructions, and stop when an **unsafe**
. instruction is encountered. If MODE is **false**, then await DRAM response
. - Return from this function when DRAM response is received
- Print relevant statistics at every clock cycle
- Perform a row writeback operation at the end if required
- Print execution statistics

1.10 Raising Errors

Note: Only extra errors/warnings raised are mentioned here (those not raised in Assignment-3).
- If insufficient command line arguments are provided, we raise *Insufficient number of arguments* error
- If the delay value provided is negative, we raise *Negative Delay* error
- If the delay value provided is not an integer, we raise *Invalid Delay* error
- If operating mode value is not an integer, we raise *Invalid Mode* error
- If extra command line arguments are provided, we raise *Extra Arguments* warning

2 Testing Strategy

- A total of **60+ test cases** were generated and used as a part of our extensive testing strategy.
- Types of Test Cases Used:
. - **QueueCheck:** These test cases include all the corner cases our DRAM Queue can encounter during
. execution. These include checking for pending read/write and finding the optimal row in *Adj*.
. - **DRAM:** These test cases involve various cases varying from 1 - 2 lw/sw instructions up to 30 sw/lw
. instructions. These also check complex DRAM memory access implementation (row writeback, row
. activation, same row access and column access)
. - **Overlapping:** These include cases where register overlap in consecutive instructions causing blocking.
. Such programs run very efficiently on our implementation due to non-blocking implementation.
. - **Non - Overlapping:** These include cases with no register overlap in consecutive instructions. Such
. programs run very efficiently in general due to no rate-determining register dependencies.
. - **DifferentDelays:** Contains various cases of sizes 1 through 4. These also check the DRAM
. implementation, but with different values of ROW_ACCESS_DELAY and COL_ACCESS_DELAY
. - **Sparse:** These test cases check our non-block memory access implementation.
. These test cases have sparse sw/lw instructions, hence, CPI is close to 1.
. - **Dense:** These test cases check our non-block memory access implementation.
. These test cases have dense sw/lw instructions and other **unsafe** instruction, hence, CPI increases.
. - **Relational:** Here only relational instructions - bne/beq/slt/j are used for testing.
. - **Mixed:** Mixed testing for all instructions add/addi/sub/mul/bne/beq/slt/j/sw/lw.
. - **SamplePrograms:** These are miscellaneous programs like finding factorial, finding a^b, finding
. maximum element in an array and finding sum of all elements in an array. These test cases involve all
. instructions present in the ISA (implemented by us in Assignment-3), and also uses loops, therefore,
. these act as good test cases to test our implementation, especially the non-block memory access part.
. - **SampleTestCases:** These are sample variants of the preliminary test cases provided to us for
. the Assignment 4 and Minor. For example we reversed/alternated the sw/lw instructions.
. - **ErrorsAndWarnings:** These are just some extra errors and warnings (in addition to the ones that
. are already raised in Assignment-3 code). These mainly relate to the invalid format of command line
. - **Corner:** These include some corner cases like lw/sw with $zero register, and some others involving
. queue dynamics.

3 Result

Our code **passed all test cases**. Relevant error messages(if any) were displayed for invalid inputs.