4DM4 Project Report:
ROB and LSQ Reordering Implementation

Johnnathan Gershkovich - 400408809
Joshua Umansky - 400234265
Shanzeb Immad - 400382928
Yuxi Qin - 400385757

Group Member	Contributions
Johnnathan Gershkovich	● LSQ/ROB class debugging and implementation ● Overall debugging ● SH file creation for output data
Joshua Umansky	● LSQ/ROB class creation & implementation ● ProcessTxBuf, ProcessRxBuf, Step function updates ● LSQ and Step report sections
Shanzeb Immad	● Report implementation ● Debugging ● Aided with logical implement of the STEP function
Yuxi Qin	● Report formatting ● General code debugging

Tx Path

The transmission path is the mechanism where memory requests or data is transmitted to the cache or memory by the CPU. In the case of this project, the transmission is done with a FIFO queue referred to in the code as m_txFIFO. This queue is only worked with when the LSQ needs to push a memory into the cache. A function was created within the LSQ.cc which appends the oldest memory of the LSQ to the txFIFO queue. This function is used within the CpuCoreGenerator file within the function that processes the tx buffer.

It pushes the oldest memory to ensure the memory is processed in program order to avoid data hazards. The LSQ is prompted to push only when the FIFO queue is empty to avoid multiple pushes of the same instruction, this allows each instruction to be pushed to the cache one time only. The fifo is designed to hold one instruction until it is executed by the CPU, once empty it relays the message that it is ready for the next request where the LSQ can safely push the next oldest instruction.

Rx Path

The overall purpose of the Receive Path (Rx path) is to make sure the CPU is handling the memory operations and to appropriately integrate it back within the pipeline. In this path, both the ROB and LSQ have a function that works almost identically to receive from the cache. The only difference is that the LSQ only compares the response instruction with the oldest instruction. Since the ROB does not function with in order programming and just keeps track of all instructions being

processed, the response message can realistically be anywhere in the buffer, that is why the ROB rxFromCache uses a for loop to search for the identical instruction.

The function these helper functions are utilized in is the ProcessRxBuf which is responsible for handling received memory responses. These memory responses are stored in the m_rxFIFO queue to be used within the code. The oldest memory from the queue is obtained and popped to where it is then searched for in the LSQ and ROB where it is set to a state of READY. The number of requests is decremented and the logger is updated. Finally a check is done to see if the request is completed and the next schedule is made.

LSQ

The LSQ was implemented as a class, which made use of a vector to mimic the queue nature, while also allowing random access to each individual request. This class has two private parameter variables (maximum entries and number of entries) that are updated to keep track of the state of the LSQ buffer. The LSQ has several functions that were required in order to operate correctly, outside of some helper functions that were declared, the primary functions include step, commit, allocate and ldFwd (pushToCache and rxFromCache have both been previously discussed).

The step function simply calls the retire function (which in turn removes any instruction that has been marked as ready by the commit function). When dealing with the LSQ, the instructions can be retired out of order, therefore the step function will loop through the entire LSQ vector, and remove any that have already received the response from the memory controller, or any that were handled by the LdFwd function.

The commit function sets the provided instruction to be ready, and is called whenever the state of an instruction call needs to be updated.

The allocate function adds a new memory request to the LSQ vector, and increments the amount of num_entries private variable in order to track the size of the LSQ. The allocate function is called every step (In the main CpuCoreGenerator.cc file) when there is a valid memory instruction ready to be allocated.

The LdFwd instruction takes two instructions as input values, and will shift the data from one to another. This is done in order to optimize the case where a read instruction would be added to the LSQ, which already has a write instruction to the same address location.

ROB

The ROB was also implemented as a class, very similar to the LSQ as it utilizes a vector. This design allows the flexibility for managing and storing instructions to track and eventually retire. The class has three private parameter variables, MAX_ENTRIES, num_entries, IPC and a vector to store memory requests that are being tracked. The functions in this class are fairly similar to the LSQ containing major functions such as step, allocate, retire, commit and rxFromCache.

The step function loops through the ROB and checks if the oldest entries are ready to be retired, it continues retiring until an entry is not yet ready. Allocate basically adds a new entry into the ROB along with increasing the counter keeping track of how many entries are being tracked. The retire function just removes the oldest entry and decrements the same counter. rxFromCache was discussed in the RxPath section. The last major function was commit, which served the purpose of manually setting the state to ready of the instruction in the input parameter.

Step Function

The step function which occurs every interval dt, in the main CpuCoreGenerator is the driving force behind the entire simulation. This function calls both the ROB and LSQ step functions, it

processes the existing cpu_FIFO functions (ProcessTxBuf and ProcessRxBuf) and also reads new instructions from the trace file.

With the updated trace file format, the file reading needed to be updated in order to correctly parse the data (Figure 1). The three separate parts are saved, and converted accordingly to the correct data types. As the trace file is read, the memReq object is updated with the type information, and address which is then staged for addition to the ROB and LSQ. In addition to the memory request that was created, there is an amount of compute instructions that are to be processed in between each memory request. These compute instructions are created before the memory instruction of each line, and stall logic was implemented in order to prevent a memory instruction from being added to the ROB and LSQ before the compute instructions are done being added. This was done by using the variable "stall_counter_compute", which both functions as a boolean (0 or not 0) and as a location to save the value of the iterator i when the compute instructions are being added to the ROB (Figure 2).

```
if (getline(m_bmTrace,fline) && stall_counter_compute == 0 && stall_counter_mem == 0) {
 m_newSampleRdy    = true;
 size_t pos        = fline.find(" ");
 size_t pos2 = fline.rfind(" ");
 std::string compute_instructions_s = fline.substr(0, pos); //number of compute instructions
 std::string address = fline.substr(pos + 1, pos2 - pos - 1); //address
 std::string type = fline.substr(pos2 + 1, 1); //type
```

Figure 1: Snippet of the step function that parses the data from a trace file

```
for(i; i < compute_instructions; i++){
  //create a new memReq for each computer instrcution, and allocate to ROB
  CpuFIFO::ReqMsg newReq = m_cpuMemReq;
  newReq.msgId = IdGenerator::nextReqId();
  newReq.type = CpuFIFO::REQTYPE::COMPUTE;
  newReq.addr = 0;
  if(m_rob->canAccept()){
    m_rob->allocate(newReq);
  }
  else{
    stall_counter_compute = i;
    break;
  }
  if(i == compute_instructions - 1){
    stall_counter_compute = 0;
  }
}
```

Figure 2: Snippet of the step function that handles the compute instruction addition to the ROB

After the compute instructions have been handled, then the memory request can be added to both the ROB and LSQ, if at any point either the LSQ or ROB is full, the variable stall_counter_mem is set to 1, meaning that a memory stall has occurred, preventing a new line in the trace from being read. If a stall occurs, the request to put into the ROB and LSQ is rerun on the next step.

If the request is a store, and both queues can take the new memory request, it is simply allocated, as there is no dependency requirement. If the request is a read, before it is allocated, the LSQ is searched to see if a load forward can occur, in this case the address of any previous write requests is checked, and if the address is equal to the new request, the data is forwarded using the ldFwd function (Figure 3). Otherwise the memory request is added to the LSQ and ROB, and all stall

variables are set to zero in order to read the next line of the trace file.

```
int loadFWD = 0;
for(int i = m_lsq->getNumEntries() - 1; i >= 0; i--){
  if(m_lsq->lsq_q[i].addr == m_cpuMemReq.addr && m_lsq->lsq_q[i].type == CpuFIFO::REQTYPE::WRITE){
    //need to forward the data
    m_lsq->ldFwd(m_cpuMemReq, m_lsq->lsq_q[i]);
    loadFWD = 1;
    break;
  }
}
if(loadFWD == 1){
  if(m_rob->canAccept()){
    m_rob->allocate(m_cpuMemReq);
    stall_counter_mem = 0;
  }
  else{
    stall_counter_mem == 1;
  }
}
```

Figure 3: Load forwarding logic from step function

Finally, the ROB and LSQ step functions are called, and they individually manage any instructions that can be retired, therefore freeing locations in both queues.

Results

We created a new file in order to automate running the different parameters that needed to be sweeped. The three parameters included ROB_Size (Figure 4), IPC (Figure 5) and the LSQ_Size (Figure 6) which respectively had the initial values of 32, 4 and 8. When one parameter was subject to sweeping, the others maintained their initial values. The following shows what values each parameter was swept as, ROB_SIZE: [1, 2, 4, 8, 6, 32], IPC: [1, 2, 4, 8] and LSQ_Size: [1, 2, 4, 8, 16, 32].

As can be seen from the graphs below, the values of all three parameters have a minor effect on the overall performance of any trace file. The most noticeable change can be observed when increasing the size of the LSQ or ROB buffers to the value of 32, as this allows for more instructions to potentially be completed out of order, therefore speeding up the general flow of the program. This does agree with the hypothesis as this scenario is unique, as there is no data dependency for any compute instructions, and the number of compute instructions is vastly greater than the number of memory instructions. In our simulation, compute instructions are immediately committed in the ROB, and retired subsequently in the next cycle, therefore freeing the values of more often. This trends towards showing that our simulation is not memory bound, as the trend is evident in both the ROB and LSQ size varying simulations.

The IPC values do slightly decrease as the value increases (Table 1), which supports the suggestion earlier that allowing more instructions to operate out of order does increase the performance of the program overall.

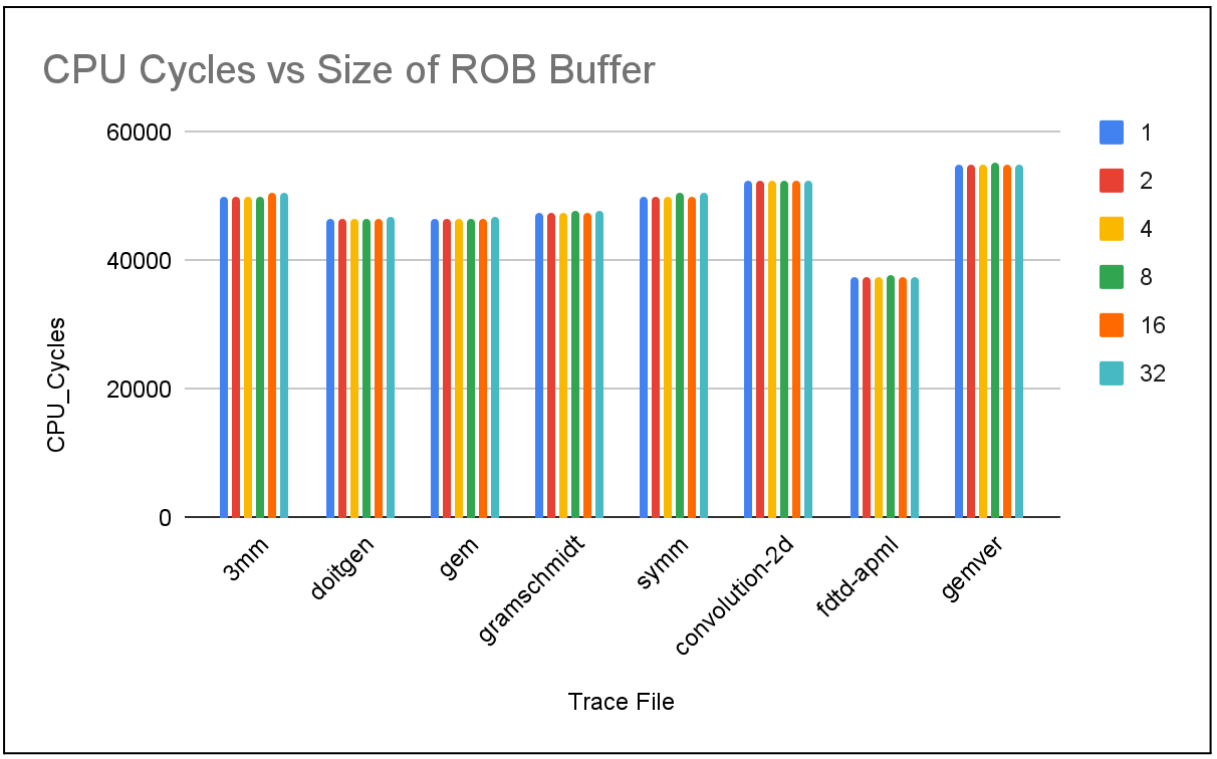

Figure 4: Change in CPU_Cycles by varying size of ROB buffer

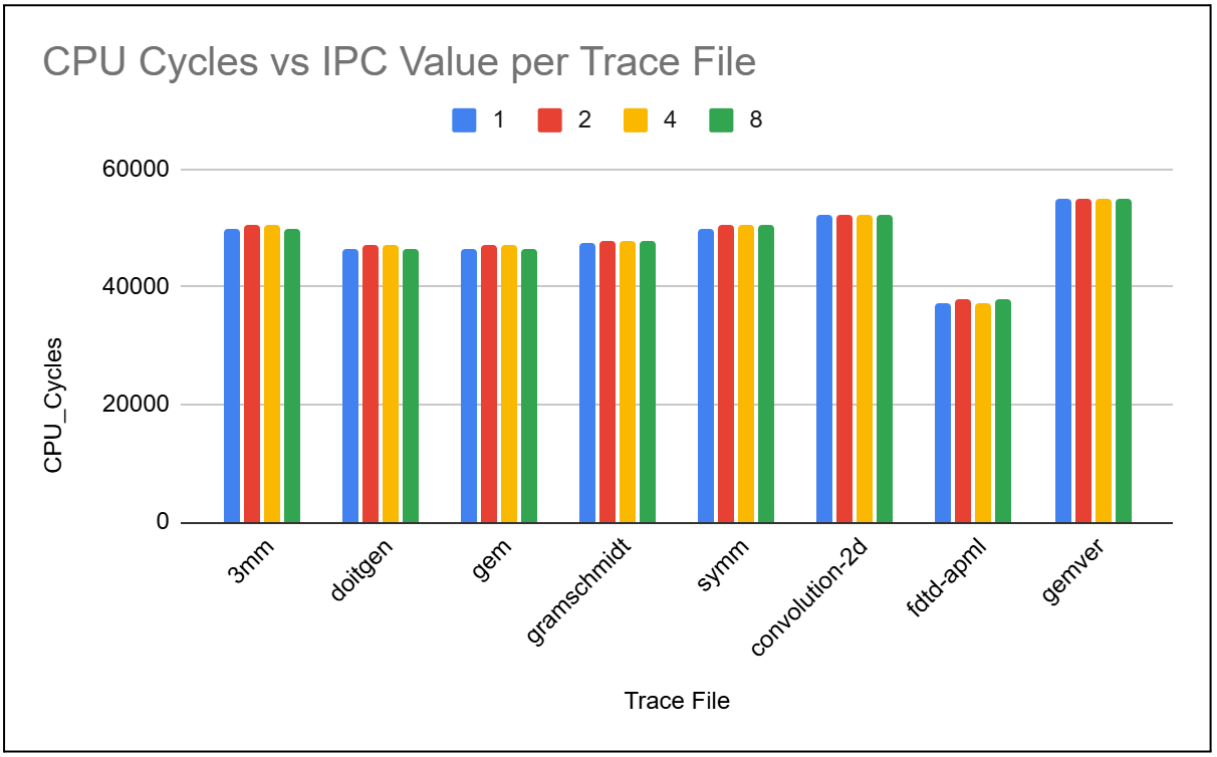

Figure 5: Change in CPU Cycles by varying size of IPC

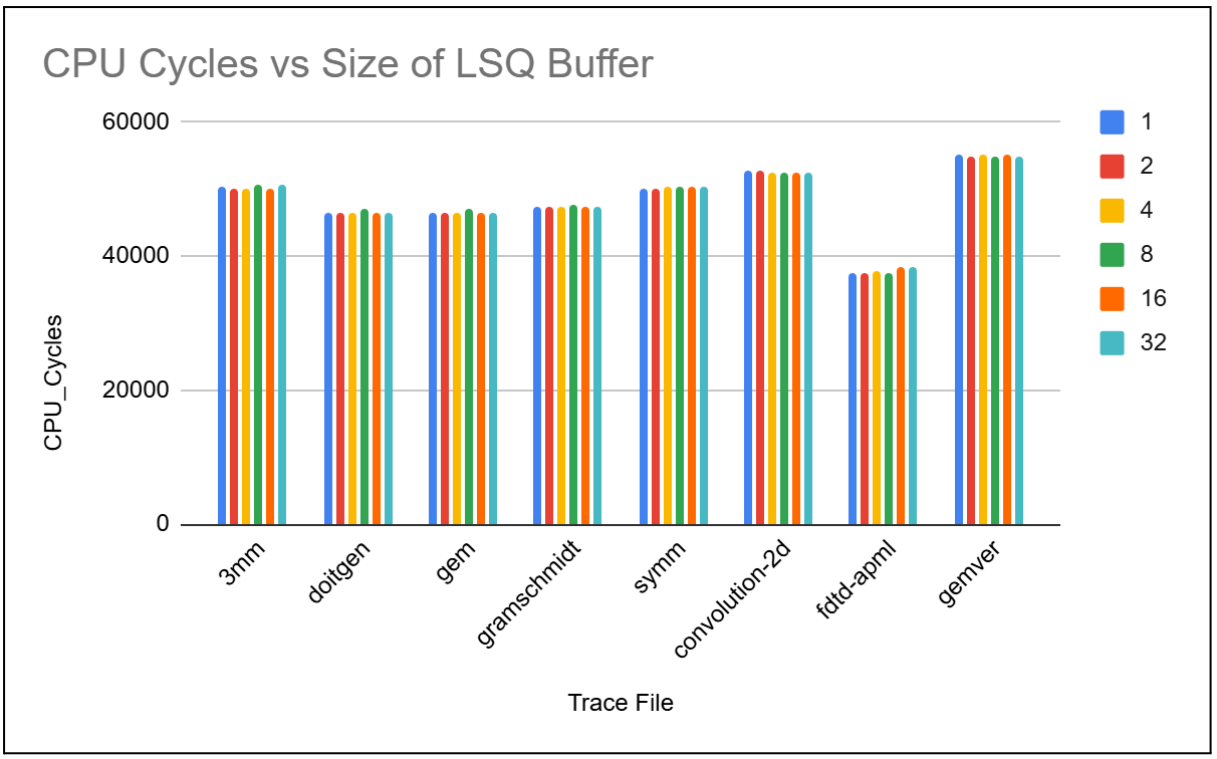

Figure 6: Change in CPU Cycles by varying size of LSQ Buffer

IPC	3mm	doitgen	gem	gramschmidt	symm	convolution-2d	fdtd-apml	gemver
1	50009	46469	46469	47372	50006	52384	37368	55002
2	50532	46992	46992	47639	50529	52384	37891	55002
4	50532	46992	46992	47639	50529	52384	37368	55002
8	50009	46469	46469	47639	50529	52384	37891	55002

Table 1: Values of cpu cycles for varying IPC value