Name : Yossef Yakobi

ID :301752267

Name : David Meriin

ID : 304472038

# Speculative Tomasulo Simulator

* This simulator is a simulator of a speculative tomasulo cpu. The project is written in java. in order to run the project enter the following command in the command line :

**java -jar sim.jar cfg.txt memin.txt memout.txt regint.txt regout.txt trace.txt**

* + **cfg.txt -**Input file that is theconfiguration filefor the simulator
  + memin.txt - Input file that holds 1024 lines of 8 digit hex values, that represent the main memory before starting the run.
  + memout.txt - Output file that holds 1024 lines of 8 digit hex values, that represent the main memory after the run has finished.
  + regint.txt - Output file that holds 16 lines of the decimal value of all integer registers
  + regout.txt - Output file that holds 16 lines of the float value of all float registers
  + **trace.txt -** Output file that holds traces of all operations that were issued in the run.
* The simulator runs on all instruction loaded in main memory starting from pc = 0.
* The simulator runs until it commits a halt or a not supported operation. If stores are still in progress, they will finish.
* Each cycle has 5 stages ( actions performed in each stage do not influence other stages in the same cycle ) :
  + Fetch
  + Issue
  + Execute
  + Write CDB
  + Commit
* BTB - Implements a random LRU mechanism .
* For store operations in ROB, the "Value" field will hold the float register ID, since when committing store operations the current value of the float register is used.

## Stages

* Fetch –
  + Queue of maximum 16 instructions. When queue is full , the new instruction is not enqueued and PC is not incremented.
  + Search for the pc in the BTB . Update Pc if found., else increment by 4.
* Issue
  + If decoded a non valid command , set the opcode to "not supported". If the operation would be committed it'd be treated as if it is was halt.
  + Peek and decode the instruction . If there’s room in the reservation station and ROB dequeue from instruction queue and push to corresponding station. When adding the instruction to a reservation station, go to Register status table and take the value to Vj/Vk or robAddress to Qj/Qk ( depending on opcode and value amiability ).
  + Insert the instruction to the ROB.
* Execution –
  + For all FUs (function units) check 2 conditions, and if they comply, start execution for the new operation. The conditions are :
    - There’s a new operation which is ready, meaning needed values are available.
    - The appropriate FU Is ready, meaning no operation has started on this FU, at this cycle yet.
  + Hold for each FU an array of counters, which will reflect state of alu for each operation that use it and how many cycles remain for operation to finish.
  + For an existing running alu, continue the op and decrement the counter of this op.
  + On last execution cycle, do the requested operation.
  + If 2 operations that require the same FU are ready, the following order is applied :
    - For the integer ALU
      * Integer operations
      * Store address calculation
      * Load address calculation
    - For the Float Add/Sub FU – lowest index in the reservation station starts.
    - For the Float Multiply FU – lowest index in the reservation station starts.
    - For The memory unit
      * Load operations.
      * Store operations.

If the operations are of the same kind, operation with lower index in the reservation station will be executed.

* Write result
  + Write the result to the rob table in the relevant rob index.
  + Notify all reservation station tables that ROBx has a value now which is y.
  + Delete the row from its corresponding reservation station.
  + When rob is notified on branch/jump then Check BTB.:
    - If pc is in BTB and branch is taken or pc is not in BTB and branch Is not taken - do nothing.
    - Else
      * Delete all rows after branch op in ROB until tail is reached.
      * For each rob row deleted :
        + Delete all reservation rows that are stored in any reservation stations, where the target rob id is the rob being deleted.
        + Update Float/integer status tables . If there is a register that waits for the result of the rob being deleted, search for a valid rob that is not deleted and writes to the same register. The register will be updated with the new rob id.
      * Flush instruction queue.
      * If pc is in BTB and branch isn’t taken then - Erase row from BTB.
      * If pc is not in BTB and branch is taken then - add row to BTB.
* Commit
  + Commit the head of the ROB only if it ready to be committed.
  + For non store operations just update register tables if needed.
  + For store write to memory.
    - Rob head is advanced, but the write can still be active for the next few cycles.
  + After commit is done increment head.
  + If committing Halt/Not supported operations, set halt to true.
    - In this case next cycles will only finish ongoing store operations and then exit the application.

## Classes

* **Commit :**
  + **Public static functions:**
    - Run() – the main function of the commit stage as described above, it commits the head of the Rob table if the head is ready and operate according to the head operation. If the operation is store then it starts write the data to the main memory, and if it not store operation it updates the int/fp register status tables.
    - hasMoreStoreToCommit() – Boolean function that return true if there is at least one store in progress and false otherwise.
    - handleStoreOfCommits() – the function is responsible for handling the stores operations that already started, it advances each store one step in the memory pipe and when it finishes the pipeline it writes the result to the main memory.
* **Config :**This class holds the information from the cfg.txt input file
  + **Public members :**
    - IntDelay;
    - AddDelay;
    - MulDelay;
    - MemDelay;
    - RobEntries;
    - AddNrReservation;
    - MulNrReservation;
    - IntNrReservation;
    - MemNrLoadBuffers;
    - MemNrStoreBuffers;
* **Execution:**
  + **Public static function:**
    - Run() - the main function of the Execution stage as described above, In the Execution stage we calculate the results of the operation in the Function Units of the processor In each cycle we start in each FU at most one operation, and advanced one step in the pipe the operations that already start using it. Because of the fact that at most one operation starts in each cycle for each FU, at most one operation can finish in each FU each cycle. When operation is done in the current cycle we save the appropriate row and the calculated result,   
      so we will be able to write this result to the appropriate ROB row and other reservation stations in the write result (write CDB) stage.
  + **Public final static member :**
    - BRANCH\_NOT\_TAKEN = -1   
      the result for not taken branch.
  + **default static members**
    - Int AluIntResult – Holds the result of the integer operation that finish in the previous cycle.
    - float LdResult – Holds the result of the load operation that finish in the previous cycle.
    - int StResult - Holds the result of the store address calculation that finish in the previous cycle.
    - float FpAddResult - Holds the result of the Fp add/sub operation that finish in the previous cycle.
    - float FpMulResult - Holds the result of the Fp multiply operation that finish in the previous cycle.
    - FpReserveRow ReadyFpAddRow – holds the row of the appropriate operation that finished in the previous cycle
    - FpReserveRow ReadyFpMulRow – holds the row of the appropriate operation that finished in the previous cycle
    - IntegerReserveRow ReadyIntRow – holds the row of the appropriate operation that finished in the previous cycle
    - MemBufferRow ReadyLdRow – holds the row of the appropriate operation that finished in the previous cycle
    - MemBufferRow ReadyStRow – holds the row of the appropriate operation that finished in the previous cycle
    - int ReadyIntRowIndex – holds the row’s index in the reservation station of the appropriate operation that finished in the previous cycle
    - int ReadyLdRowIndex – holds the row’s index in the load buffer of the appropriate operation that finished in the previous cycle
    - int ReadyStRowIndex – holds the row’s index in the store buffer of the appropriate operation that finished in the previous cycle.
    - int ReadyFpMulRowIndex – holds the row’s index in the reservation station of the appropriate operation that finished in the previous cycle.
    - int ReadyFpAddRowIndex – holds the row’s index in the reservation station of the appropriate operation that finished in the previous cycle.
* **Fetch:**
  + **Public static function:**
    - Run() - the main function of the Fetch stage as described above, if there is a place in the instruction queue then reads the next instruction from memory, check in the BTB if the instruction is a branch and if it taken. And then it fills the instruction container and enq the instruction to the queue.  
      Finally it increments the pc for the next instruction to fetch by the result from the BTB.
* **FileHandler:**
  + **Public static functions :**
    - Config ReadConfig(string cfgFile) – the function gets the cfg file path, it reads the file and fill out config object with the appropriate parameters and return the config object.
    - Int[] ReadMainMem(string MemIn) – the function gets the memin file path, it reads the file and fill out an int array with the data and returns it.
    - WriteMemOut(int[] memOut, string fileName) – the function gets an int array memOut and filename, and it writes the data from memOut into the file.
    - writeRegInt ( int[] regInt, string FileName) – the function gets an int array regInt and filename and reads the data from the array and writes the data into the file.
    - writeRegOut(float[] regFp, string FileName) - the function gets a float array regFp and filename and reads the data from the array and writes the data into the file.
    - WriteTraceToFile(Trace trace, string FileName) – the function gets a trace and filename and reads the data from the trace and writes the data into the file.
* **FpRegStatus :**
  + **default members :**
    - Float Value – holds the register value.
    - Short Rob – holds the row id in the ROB table.
* **FpReserveRow :**
  + **Public function :**
    - GetOpcode() – returns the operation code of the row.
  + **default members:**
    - Int ID – holds the trace ID.
    - float Vj – holds the register value.
    - float Vk – holds register value.
    - int Qj – holds the row id in the ROB table which we get the source from.
    - int Qk – holds the row id in the ROB table which we get the source from.
    - int ROB – holds the row id in the ROB table which we writes the result to.
    - Boolean Busy – holds information if the row is busy or not.
    - Int PC – holds the operation PC.
    - Int Address – holds address for branch and memory operations.
    - Boolean Taken – holds information if the branch predicted as taken or not.
* **InstructionContainer:**
  + **public members:**
    - int Instruction – holds the instruction.
    - boolean Taken – holds information if the branch predicted as taken or not.
    - int ID – holds the trace id.
    - int PC – holds the instruction pc.
* **IntegerReserveRow :**
  + **Public function :**
    - GetOpcode() – returns the operation code of the row.
  + **default members:**
    - Int ID – holds the trace ID
    - int Vj – holds the register value
    - int Vk – holds register value
    - int Qj – holds the row id in the ROB table which we get the source from.
    - int Qk – holds the row id in the ROB table which we get the source from.
    - int ROB – holds the row id in the ROB table which we writes the result to.
    - Boolean Busy – holds information if the row is busy or not
    - Int PC – holds the operation PC
    - Int Address – holds address for branch and memory operations.
    - Boolean Taken – holds information if the branch predicted as taken or not
* **IntRegStatus :**
  + **default members :**
    - int Value – holds the register value.
    - Short Rob – holds the row id in the ROB table.
* **Issue :**
  + **Public static function:**
    - Run() - the main function of the Issue stage as described above, the function peek the instruction from the queue decodes it and check the opcode if it valid if does it operate according to the opcode. If it unknown, halt or jump command it insert it direct to the rob table. Else it check if there is a place in the appropriate reservation station/memory buffer if there is it deq the operation from the queue and insert it to the reservation station/memory buffer.
* **MemBufferRow :**
  + **default members:**
    - Int ID – holds the trace ID
    - int Vj – holds the register value
    - int Vk – holds register value
    - int Qj – holds the row id in the ROB table which we get the source from.
    - int ROB – holds the row id in the ROB table which we writes the result to.
    - Boolean Busy – holds information if the row is busy or not
    - Int Address – holds address for branch and memory operations.
* **OpCodes :**
  + **Public final static members one per each operation code:**
    - public static final byte LD\_OPCODE = 0
    - public static final byte ST\_OPCODE = 1
    - public static final byte JUMP\_OPCODE = 2
    - public static final byte BEQ\_OPCODE = 3
    - public static final byte BNE\_OPCODE = 4
    - public static final byte ADD\_OPCODE = 5
    - public static final byte ADDI\_OPCODE = 6
    - public static final byte SUB\_OPCODE = 7
    - public static final byte SUBI\_OPCODE = 8
    - public static final byte ADD\_S\_OPCODE = 9
    - public static final byte SUB\_S\_OPCODE = 10
    - public static final byte MULT\_S\_OPCODE = 11
    - public static final byte HALT\_OPCODE = 12
    - public static final byte NOT\_SUPPORTED = 13
  + **Public static functions :**
    - isOpSetToFloat(byte opcode) – Boolean function that returns true if the opcode is FP operation and false otherwise.
    - isOpSetToInt(byte opcode) – Boolean function that returns true if the opcode is Int operation and false otherwise.
* **ResvStatHandler:**
  + **Public functions:**
    - IsResvStatFull () – boolean function, it returns true if the appropriate reservation station/memory buffer is full and false otherwise.
    - AddRowToResvStat() – gets reservation station/memory buffer and a row and add the row to the reservation station/memory buffer if it succeed it returns the row number and if it fails to add the row it returns -1.
    - removeRowFromRestStatByRobID – gets reservation station/memory buffer and a row and remove the row from it.
* **RobQueue:**
  + **Public functions:**
    - Add(RobRow row) – gets a row and add it to the rob table if it not full and returns the row index in the array. If the rob queue is full it returns -1.
    - Delete(int index) – gets an index and delete the row in that index from the rob queue.
    - FlushAfter(int index) – gets index and delete all the rows from the next index and until we get to the head.
    - Increment(int index) – gets index and return the next index in the rob queue,  
      if the rob size is less than the maxRobSize then it returns index + 1, else it returns 0.
    - Decrement(int index) - gets index and return the previous index in the rob queue,  
      if the index is bigger than zero then it returns index - 1, else it returns QueueMaxSize - 1.
  + **Protcted members:**
    - RobRow[] queue – an array of RobRow.
    - int QueueMaxSize – holds the max size of the Rob table as it given in the cfg.txt file input.
    - int head – holds the index of the queue’s head.
    - int tail – holds the index of the queue’s tail.
* **RobRow:**
  + **Public function:**
    - GetOpCode() – returns the rob row operation code
  + **default members:**
    - Int ID – holds the trace ID.
    - Int Destination – holds the register destination which we writes the value to.
    - Object Value – holds the operation result value.
    - Boolean Ready –set to true only when value is ready. For all opcodes except store commit when ready is true. For store make sure ready and destination is not null then commit.
* **Sim:**
  + **The class that holds the main function.**
    - Main(String[] args) – while halt is not true, do all the stages by the order of :  
      Fetch, Issue, Execution, Write CDB, Commit. When we halt set to true the loop ends it check if there are stores operations that already start it finishes them, and then it writes the results to the appropriate files.
* **Trace:**
  + **Public function:**
    - TraceRecord GetRecord(int id) - return the TraceRecord that match the given id.
    - Int AddRecord(int instruction) – gets instruction create new record for it with unique id and return the id.
* **TraceRecord:**
  + **Public members:**
    - int ID – holds the trace id.
    - String Instruction – holds the instruction.
    - int CycleIssued – holds the information of the cycle that the instruction has been issued in.
    - public int CycleExeuctedStart – holds the information of the cycle that the instruction execute has been started in.
    - public int WriteCdb – holds the information of the cycle that the instruction has been written to the CDB in.
    - public int CycleCommit – holds the information of the cycle that the instruction has been commited in.
    - public int CycleFetch– holds the information of the cycle that the instruction has been fetched in.
* **Utils:**
  + **Public function:**
    - int AddressToRowNum(int address) – gets an address from the file and return the appropriate address for the main memory array.
    - Init(Config conf) gets config object and initiate the static fields from the config data.
    - PC – holds the current pc.
    - IntRegStatusTable– an array of IntRegStatus in length of 16 that represent the integer registers R0 – R15 status.
    - FpStatusTable– an array of FpRegStatus in length of 16 that represent the float registers F0 – F15 status.
    - Int Cycle counter – holds the current cycle number.
    - UnitCounter static object (not defined actually in global) – arrays of countersone array for each reservation station/memory buffer with same length.
    - RobTable – an array of RobRow that Implemented as a cyclic queue with head and tail pointer in a given size from cfg file.
    - InstructionsQueue - queue in size of 16 all from instructionContainer Type
    - Integer reservation station array – of given size
    - FP adders reservation station array – of given size
    - FP multipliers reservation station array – of given size
    - Load Buffer array – of given size
    - Store Buffer array – of given size
    - BTB HashMap<int,int> – size 16. Dictionary that has the pc as the key and the pc predicted as the value.
    - Main Memory – 1024 rows of 32 bit.
    - boolean Halt - set to false by default.
    - boolean MemInUse set to false by default.
* **WriteCDB**
  + **Public static function:**
    - Run() - the main function of the Issue stage as described above, the function is responsible for writing the results of the operation, that finished their execution in the previous cycle, to the appropriate Rob table’s row and to all the reservation station/memory buffer that waits for this Rob row. It then delete the row from the reservation station/memory buffer.

If in the write CDB we get for branch operation results that show a miss predicted branch we flush the Rob Table from this row until the head, delete those rows from the reservation stations/memory buffer, update the int/fp registers status tables, flush the instruction queue and update the PC and the BTB table according to the branch result.

Structures:

**BtbRow** :

* PC
* Predicted PC

**IntegerReserveRow**

* Opcode – 4 bit
* ID - int
* Vj- 32 bit
* Vk – 32 btit
* Qj - int
* Qk - int
* ROB - int
* Busy bit - bool
* PC/Address - int
* Taken/NotTaken (bool)

**FpReserveRow**

* Opcode
* ID - int
* Vj - FP
* Vk- FP
* Qj
* Qk
* ROB
* Busy bit

**Fp Status**

* Value - FP
* Rob - short

**Int status**

* Value – int
* Rob - short

**ROB**

* Opcode – 4 bit
* ID - int
* Destination - int
* Value – Object(FP/int/null)
* Ready – (bool) set to true only when value is ready . For all opcodes except store commit when ready is true. For store make sure ready and destination is not null then commit.

**TraceRecord**

* ID
* Instruction (string that represent 8 bytes in hex)
* Cycle\_issued
* Cycle\_exeucted\_start
* Write\_cdb
* Cycle\_commit

**Trace**

* **ArrayList of TraceRecord**
* Id – int , starts at 0 . ID generator for the instructions .

**Functions:**

* **Int AddRecord(int Insturction ) – Returns ID.**
* **TraceRecord GetRecord(ID).**

**FileHandler**

**Functions**

* **Config ReadConfig(string cfgFile)**
* **Int[] ReadMainMem(string MemIn)**
* **WriteMemOut(int[] memOut, string fileName)**
* **writeRegInt ( int[] regInt, string FileName)**
* **writeRegOut(float[] regFp, string FileName)**
* **WriteTraceToFile(Trace trace, string FileName)**

**UnitCounter (static)**

* **MemCounter**
* **IntCounter**
* **FpAddCounter**
* **FpMulCounter**

**InstructionContainer**

* **Instruction(int)**
* **Taken (bool)**

**RobQueue**

* **Array of rob**
* **Add,delete(location), flushAfter – all use head and tails pointer on a cyclic array.**

**Utils**

* **Function : static public int AddressToRowNum(int address)**
* **Function : static public void Init(Config conf)**
* PC
* Array of int registers status(16)
* Array of float registers status (16)
* Cycle counter (int)
* UnitCounter static object (not defined actually in global)
* Array of Rob in the given size. Implement a cyclic queue with head and tail pointer.
* Queue of instructions (16) – all instructionContainer Type
* Integer reservation station array – of given size
* FP adders reservation station array – of given size
* FP multipliers reservation station array – of given size
* BTB HashMap<int,int> – size 16. Dictionary that has the pc as the key and the pc predicted as the value.
* Main Memory – 1024 rows of 32 bit.
* Halt = 0 (bool) . 0 by default.

**Main**

1. GetConfig and initialize
2. Main loop while(!halt):
   1. Fetch
   2. Issue
   3. Execute
   4. Write CDB
   5. Commit