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 a counter, which will reflect state of alu and how many cycles remain to finish.
  + For an existing running alu , continue the op and decrement the counter.
  + 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 ALU
      * Add operations
      * Multiply operations.
    - 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 :**
* **Config :**
* **Execution:**
* **Fetch:**
* **FileHandler:**
* **FpRegStatus :**
* **FpReserveRow :**
* **InstructionContainer:**
* **IntegerReserveRow :**
* **IntRegStatus :**
* **Issue :**
* **MemBufferRow :**
* **OpCodes :**
* **ResvStatHandler:**
* **RobQueue**
* **RobRow**
* **sim**
* **Trace**
* **TraceRecord**
* **Utils**
* **WriteCDB**

Structure:

**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

**Config**

* **IntDelay**
* **AddDelay**
* **MulDelay**
* **MemDelay**
* **RobEntries**
* **AddNrReservatopm**
* **MulNrReservatopm**
* **IntNrReservatopm**
* **MemNrLoadBuffers**
* **MemNrStoreBuffers**

**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**

**Utils**

* **Function : static public int AddressToRowNum(int address)**
* **Function : static public void Init(Config conf)**

**InstructionContainer**

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

**RobQueue**

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

Global

* 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