**APEX Simulator for Out of Order Processor supporting speculative execution using register renaming**

The APEX processor is designed as an out-of-order execution processor, which allows for dynamic execution of instructions to optimize the usage of CPU cycles and improve throughput. It includes various features like out of order execution which enhances processing speed by executing instructions as resources become available, rather than following strictly to program order, the other main feature is register renaming where in it reduces pipeline stalls due to data dependencies by allowing multiple physical registers to represent a single architectural register which in turn eliminates all the anti and output dependencies leaving only flow dependency, the other feature is speculative execution which improves branch prediction accuracy and reduces the penalty of branch mispredictions by speculatively executing instructions ahead of branch resolutions.

**Design and Distribution of work:**

* Initially, we’ve spent time together on understanding the requirement document by discussing when encountered any misclarifications.
* Then we made sure each one of us accurately understood what happens in each stage of the pipeline , if not we’ve discussed and understood all the flow in the given datapath.
* After achieving a thorough understanding of the pipeline and its datapath, we’ve started writing structs for all regular arrays (IQ,BQ) and circular FIFO arrays (ROB, LSQ), then written pseudo code for all the operations on these structs like establishing entries into Issue queue for every dispatching instruction and establishing entry in BQ for branch instructions and how to add and delete entries from circular FIFO arrays like ROB and LSQ and made there entries as valid by setting the valid bit else invalid.
* Then we made sure all the operations made by the respective teammate is correct by carefully analyzing the pseudo code written by each teammate.
* Then we’ve broken down the coding part for all the operations that need to be done by each teammate and we’ve ensured each teammate has equal work.

**Contribution of Team members**:

1.Rakesh Palle – Out of Order design, Implemented LSQ, MAU/D-Cache, ROB, AFU, forwarding tags in the forwarding bus to wake up the instructions.

2. Satya Sai Visalakshi Ayyalasomayajula – Register to register renaming (RAT), PRF, CC register file , ARF, commit ARF

3. FNU Rashmika Alishetty- Integrated BTB for Branching and speculation , implemented IQ,BQ and BFU, IntFU,MulFU

**Given DataPath Overview and our implementation technique:**

* There are 4 FU’s , IntFU implements all integer operations other multiply which has single cycle latency, Multiplier FU, MulFU that computes product of 2 source registers for the MUL operation, it is not pipelined and has 3 cycle latency, and a dedicated FU called AFU which computes the memory addresses for loads and stores instructions and target addresses for all the branch instructions like BZ,BNZ,BP,BNP and unconditional instructions like JUMP,JALR then in the same cycle it computes the addresses and puts in the respective forwarding buses ,thus delivers to LSQ for loads and stores and to BQ for branch instructions and JUMP,JALR and for LOAP,STOREP the increment to the source register by 4 is also performed in parallel to the memory address calculation and we have taken a separate bus for it to deliver them to LSQ, and the last FU is BFU wherein we have implemented the actual control flow for all the branch instructions and corrected the path if misprediction happened and flushed the instructions along the mis predicted path and also restored the physical free registers, rename table and CC flags free register along the mis predicted path.
* MAU (Memory Access Unit) implements the memory operations which follows the program order using the LSQ and ROB, so when the heads of LSQ and ROB are same we deliver to MAU to complete a memory operation which take 2 full cycles whether its load or store, we write the data retrieved from memory into the destination physical register for loads and for stores we just update the memory which also takes 2 full cycles.
* There are 2 separate queues called BQ(Branch Queue) for all conditional and unconditional instructions like BZ,BNZ,BP,BNP,JUMP and JALR and IQ for all other instrucions, the instructions wait until all of its source operands becomes valid and if more than one instruction becomes ready for issuing, we’ve choosen based on the dispatch time, the earliest instrucions wake’s up first and gets issued by choosing the min dispatch time between the valid instructions.
* The LSQ maintains the dispatch order of all the loads and stores and in the case of misspeculation, rollbacks are easy with the implementation of BIS(Branch Instruction Queue) to maintain the program order of dispatched branch instructions, so when branch instructions is encountered, we insert it in the top of the BIS and points to the respective entry in the ROB and viceversa i.e ROB entry also points to the entry in the BIS , so when mispredicted instruction is encountered, we flush the instructions along the mispredicted path until the mispredicted branch and then take the actual path.