**Design Document APEX pipeline Simulator**

**Submitted by**:

Ankush Arora

Purva Myakal

Divyaraj Vaghela

**EXECUTING INSTRUCTIONS**:

To change input file, change the filename char pointer below which is at the beginning of simulator.c source code file.

char \*filename="Input1.txt";

**DESIGN**:

**Data Structures used**:

1. Enum operation

Enumeration members are the different instructions possible.

1. Struct inst

Structure of an instruction which contains the instruction operation code, 3 arguments/operands, their valid bits and their address, architectural register for destination and program counter value.

This structure is used by each stage to hold current instruction and also used by IQ, LSQ and ROB to hold instructions in buffer.

1. Flag variables

Flag variables are initialized and used throughout the program to manage and implement the simulator.

**For Register Renaming:**

* AL[] list (size 16)
* PRF – status bit (size 16)
* renamed array (size 16)
* rename table (entries 9)
* Free list (size 16)
* physical register array (16)

**Re-ordered Buffer**: At dispatching (renaming time), an instruction is assigned an entry at the tail of the reorder buffer (ROB). Instructions are committed in program order, thus the ROB stores precise state.

**LSQ:** At dispatching, if instruction is LOAD or STORE, it is stored in LSQ. LOAD executes when it is at LSQ's head. STORE executes when it is at ROB's and LSQ's head. When data is broad-casted, it is forwarded to LSQ too, tag-based matching is done.

**IQ:** At dispatching, if instruction is anything except LOAD and STORE, it is stored in IQ. Instruction is issued as and when its operands are ready. When data is broad-casted, it is forwarded to IQ too, tag-based matching is done.

**Files**:

1. simulator.c contains all the source code for this APEX pipeline simulator.

**Menu Functions**:

1. Initialize()

Resets variables, flags, structures and flushes out any instructions if present in all the pipeline stages as well as reads the input file instructions into an instruction array.

1. Simulate (<number>)

Takes number of cycles as input parameter and simulates the APEX pipeline for those many number of cycles by calling the functions for each stage, i.e., at the end of simulate, <number> of cycles will be completed.

1. Display()

Displays information about the instruction (if any) present in each stage at the time of function call to Display function. Also displays the state of each register and state of first 100 memory locations. It also shows the contents of the architectural and physical registers, the rename table, the IQ, LSQ and ROB, the free list of registers and all structures relevant to renaming, including the tag and result value broadcasted etc.

**Other functions**:

1. Func\_free()

Function to free the current pointer for all the structures using free function and set the pointers to NULL.

1. Data\_forwarding()

Calls IQ\_handling() and LSQ\_handling().

1. Extract\_inst()

Called from D/RF 1 to separate op code and various operands and their value and address fields.

1. Rollback()

To rollback in case of branch mis-prediction using ROB to flush instructions from ROB tail and correct renaming structures.

**Functions for stages**:

1. Commit()

This is an additional logical stage to commit introduced just to reset the dependent pointer and set the register status bit back to valid for the register which was set invalid but now processed completely out of write back stage.

1. Write\_back()

If there was no instruction in memory stage in previous cycle then set the write\_back stage pointer to null, i.e., no instruction in write\_back stage now. Otherwise copy instruction from memory to write back and update program counter value in this stage to point to the instruction now present.

New Register values are written to registers at this stage (apart from STORE, HALT and branching instructions).

1. Execute\_mem()

Similar to write back stage, instruction if present in execution stage in previous cycle is copied to memory stage and program counter value updated.

Memory references out of memory range are captured for LOAD and STORE computations on memory.

1. Execute()

Check if any instruction in decode stage in previous cycle that needs to be moved to execute stage now. Also check for any STALLS in pipeline in this stage so as to not move a instruction out of decode stage into execute if the instruction in decode needs to wait for any previous instruction to write back first. This thus introduces a STALL of 2 cycles till the instruction is committed at the end of write back stage.

On the basis of operation code in instruction, corresponding functions are called to perform any calculation required. For example, adding of operand 2 and 3 in case of an ADD operation. This is similar to what ALU does in a pipeline in EX stage implemented via a switch case having function calls to appropriate FUs (different functions for different operations). Three FUs are added in execution for Integer, Multiplication and Memory.

1. Execute\_int()

It will pick instruction from IQ entries. Except Mul, Load and Store instructions, all other instructions are being executed here. It has single cycle latency.

1. Execute\_mem1()

Execute\_mem are pipelined functions, so there will be different instructions in all these three stages. We are also maintaining status of data structure used for register renaming. This will pick instruction from head of LSQ.

1. Execute\_mem2()

It will simply process the instruction from execute\_mem1 through execute\_mem3. We are checking for validations here too.

1. Execute\_mem3()

Here, we are updating ROB entry. This is final stage of memory execution. It will propagate instruction to write back stage

1. IQ\_handlng()

It would check for physical register status and it will be used for forwarding purpose. IQ is maintained here.

1. LSQ\_handling()

It would check for physical register status and it will be used for forwarding purpose. LSQ is maintained here.

1. Decode\_stage1()

Check if instruction in fetch stage in previous cycle is ready to move to decode stage now, i.e., instruction should be present and there must be no stall currently in pipeline. Decode is running here in two stages.

1. Decode\_stage2()

In this stage, we are checking if the sources are valid or not. If sources are not valid then we are stalling that particular instruction in this stage. Also, to copy latest value of dependent register from write back so that updated value is passed to execute in next cycle when dependency is removed via commit stage.

1. Fetch\_stage1()

This first stage of simulator will check for ROB slot and pick new instruction from the file to process and propagate through fetch\_stage2.

1. Fetch\_stage2()

It would simply propagate the instruction through it and instruction goes into decode stage.

**Functions for instructions**:

1. Func\_add()

ADD instruction – register to register operation which adds values of operand 2 and operand 3 and returns result

1. Func\_sub()

SUB instruction – register to register operation which subtracts operand 3 from operand 2 and returns result

1. Func\_movc()

MOVC instruction – register to register operation which moves value of operand 2 to operand 1

1. Func\_mov ()

MOV instruction – register to register operation which moves value of operand 2 to operand 1

1. Func\_mul()

MUL instruction – register to register operation which multiplies operand 2 and operand 3 and returns result

1. Func\_and()

AND instruction – register to register operation which performs logical AND on operand 2 and operand 3 and returns result

1. Func\_or()

OR instruction – register to register operation which performs logical OR on operand 2 and operand 3 and returns result

1. Func\_exor()

EX-OR instruction – register to register operation which performs logical EX-OR on operand 2 and operand 3 and returns result

1. Func\_load()

LOAD instruction – memory operation which adds values of operand 2 and operand 3 and returns result which will later be used to compute memory address value required for the LOAD instruction.

1. Func\_STORE()

STORE instruction - memory operation which adds values of operand 2 and operand 3 and returns result which will later be used to write into that memory address value for the STORE instruction.

1. Func\_bz()

Branch if Zero instruction – control flow instruction which branches flow to the relative program counter address passed as an argument to the instruction if the result to previous instruction was zero (arg1 value).

The instructions in fetch and decode when the branch instruction in execution stage is realized to be taken are flushed and new instructions from the new program counter address read into fetch stage from next cycle, i.e., a STALL of 2 cycles.

1. Func\_bnz()

Branch if Non Zero instruction – control flow instruction which branches flow to the relative program counter address passed as an argument to the instruction if the result to previous instruction was non zero (arg1 value).

The instructions in fetch and decode when the branch instruction in execution stage is realized to be taken are flushed and new instructions from the new program counter address read into fetch stage from next cycle, i.e., a STALL of 2 cycles.

1. Func\_jump()

JUMP instruction – control flow instruction which branches flow to the absolute program counter computed as sum of the arguments to the function (operands reg + lit value).

The instructions in fetch and decode when the branch instruction in execution stage is realized to be taken are flushed and new instructions from the new program counter address read into fetch stage from next cycle, i.e., a STALL of 2 cycles.

1. Func\_bal()

BAL instruction – control flow instruction which branches flow to the absolute program counter computed as sum of the arguments to the function (operands reg + lit value) and saves the address of next instruction after BAL in register X.

The instructions in fetch and decode when the branch instruction in execution stage is realized to be taken are flushed and new instructions from the new program counter address read into fetch stage from next cycle, i.e., a STALL of 2 cycles.

1. Func\_halt()

HALT instruction - flow instruction which stops execution when the HALT instruction reaches the write back stage and flushes all instructions from pipeline.

**WHAT EACH MEMBER DID**:

**Purva Myakal** – data forwarding, register renaming, branching and branch prediction.

**Ankush Arora** – multiple FUs and branch prediction.

**Divyaraj Vaghela** – IQ, LSQ, ROB , register renaming and design document.

Testing done by all.

**OUTPUT**:

