CYCLE ACCURATE SIMULATOR OF DYNAMICALLY SCHEDULED PROCESSOR IMPLEMENTING THE TOMASULO ALGORITHM WITH REORDER BUFFER

ECE-563 Project-2 Report

- Vijayalakshmi Parthiyoor Sundar (200201162)

DATA STRUCTURES

Reorder Buffer:

The implementation of ROB was done using a circular FIFO using an array. Given that the ROB commits in-order and must maintain the correct indexing, a circular FIFO made it easier to keep track. The data structure of ROB consists of the instruction, a ready bit, a misprediction bit, the destination, and the value and memory latency.

```
struct robT{
    dynInstructPT   dInstP;
    bool            ready;
    bool            misPred;
    unsigned        dest;
    unsigned        value;
    uint32_t        memLatency;


    robT(){
        dInstP      = NULL;
        ready       = false;
        dest        = UNDEFINED;
        value       = UNDEFINED;
        memLatency = 0;
    }

    ~robT(){
    }
};
     Fifo<robT> rob;
```

Reservation Stations and Load Buffers:

The data structure for the reservation stations and load buffer consists of the instruction with the value of the source registers, it also consists of the bits which depict if the values are ready. The structure consists of the tags of each of the source registers and the entry index of the instruction in ROB. It consists of the address, and an ID to keep track, with an “inExec” bit which checks if the instruction has already been pushed to execution unit. The reservation station was implemented as a vector.

```
157 //Data structure for Reservation Station
158 struct resStationT{
159     dynInstructPT   dInstP;
160     unsigned        vj;
161     bool            vjR;
162     unsigned        vk;
163     bool            vkR;
164     unsigned        qj;
165     unsigned        qk;
166     unsigned        tagD;
167     unsigned        addr;
168     int             id;
169     bool            inExec;
170
171     resStationT(){
172         vjR         = true;
173         vkR         = true;
174         vj          = UNDEFINED;
175         vk          = UNDEFINED;
176         qj          = UNDEFINED;
177         qk          = UNDEFINED;
178         tagD        = UNDEFINED;
179         addr        = UNDEFINED;
180         inExec      = false;
181     }
182 };
183
184       vector <resStationT*>  resStation[RS_TOTAL];
185
```

Register File:

To implement Floating-point registers, a structure **"fpFileT" and "gprFileT"** are declared with its members as value, busy and tag. The value stores the value of the respective register, while the busy indicates if the register is being used as a destination, the tag is the entry number of the instruction in the reorder buffer. The general-purpose register file was then implemented in the similar way.

```
struct gprFileT{
    int             value;
    int             busy;
    int             tag;
};

struct fpFileT{
    float           value;
    int             busy;
    int             tag;
};
```

SALIENT ASPECTS OF THE CODE

1. To model the multiple computational address units and the bypass logic, a separate lane was implemented which is used by store and bypassed load in the execute stage. The load that is not bypassed uses the memory unit during the execute stage, hence the latency is modelled in the execute stage.

```
if ( !resP->inExec && resP->vjR && resP->vkR && instReady ){
   int execUnit   = opcodeToExUnit(resP->dInstP->opcode);
   int numLanes   = execFp[execUnit].numLanes;
   bool isMem     = execUnit == MEMORY;

   //TODO: check
   if( is_store || bypassReady ){
      // Go in bypass lane
      resP->inExec                                  = true;
      execWrLaneT lane;
      lane.payloadP                                 = resP;
      lane.outputReady                              = is_load && bypassReady;
      lane.output                                   = (is_load && bypassReady) ? bypassValue : UNDEFINED;
      bypassLane.push_back( lane );
   }
   else{
      // Try to go in regular lanes
      for(int laneId = 0; laneId < numLanes; laneId++){
         //checking for free execution units
```

2. Reservation stations and load buffers as a collapsing buffer using vectors helped dispatching instructions in program order.

3. A function that finds the nearest store preceding the load and checks if it's conflicting by checking if the address has been computed, or if it is the same address as the load instruction.

```
//checking for conflicting store with a load instruction
bool sim_ooo::isConflictingStore(int loadTag, unsigned memAddress, bool& bypassReady, uint32_t& bypassValue){
   bool conflict               = false;
   bypassReady                 = false;
   for(int i = 0; i < rob.getCount(); i++){
      //getting the current tag
      int tag                  = rob.genIndex(i);
      //Get the ROB entry
      robT* robEntryP          = rob.peekNth(i);

      //checking if the opcode is store
      if( robEntryP->dInstP->is_store ){
         //if the store is not complete (a.k.a ??), then there is a conflict
         if( robEntryP->dest == UNDEFINED ){
            conflict           = true;
            bypassReady        = false;
         }
         //if store is complete and match the address, no conflict.
         //values are stored from this store to load temporarily
         else if(robEntryP->dest == memAddress){
            conflict           = !robEntryP->ready; // TODO: not conflicting if rob is ready
            bypassReady        = robEntryP->ready;  // bypass is ready if rob is ready
            bypassValue        = robEntryP->value;
         }
      }
      if(tag == loadTag)
         //returns the most recent conflict entry
         return conflict;
   }
   ASSERT(true, "tag == loadTag not found!");
   return conflict;
}
```

SELF GRADING

Testcase	Points	Comments
Testcase 1	6	The test case and reference output fully match
Testcase 2	6	The test case and reference output fully match
Testcase 3	6	The test case and reference output fully match
Testcase 4	6	The test case and reference output fully match
Testcase 5	6	The test case and reference output fully match
Testcase 6	6	The test case and reference output fully match
Testcase 7	6	The test case and reference output fully match
Testcase 8	6	The test case and reference output fully match
Testcase 9	6	The test case and reference output fully match
Testcase 10	6	The test case and reference output fully match

The code passes all testcases and everything works.

Std=c++11 was added in the makefile.