Chuck Richardson

ECE 563 Microprocessor Architecture

ECE 563 Project 2-Cycle-Accurate Simulator of a Dynamically Scheduled Processor Implementing Tomasulo Algorithm with Reorder Buffer

1. Objective

The objective of this project is to implement a dynamically scheduled processor using Tomasulo's algorithm with a reorder buffer. This processor consists of 4 stages, issue, execute, write result, and commit. Issue checks for structural hazards such as not having a free reservation station or respective load-store buffer, as well as renaming register if necessary. Before entering the execution stage one monitors the CDB(Common Data Bus) and checks if the operands are available. Execute performs the operations necessary whether it be computing the address for a LOAD or STORE, Arithmetic Floating and Integer Operations such as ADD, MULT, and SUB. Write result broadcasts this value to the appropriate reorder buffer value entry, thusly freeing a reservation station. The commit stage then writes the appropriate value to the register or memory from the reorder buffer. This processor also contains reservation stages to be processed by each of the respective functional units whether it be adder, multiplier.

2. Data Structures

2.1. Reorder Buffer

The reorder buffer was done using a circular FIFO implementation. This circular FIFO implementation contains the instruction, ready bit, misprediction bit, destination, value, and memory latency. This is shown in Figure 1.

```
221 struct reorder_buf_t{
222    dynamic_instruct_pointer   dyn_instruction_p      ;
223    bool                       ready                  ;
224    bool                       miss_prediction        ;
225    unsigned                   dest                   ;
226    unsigned                   value                  ;
227    unsigned                   mem_latency            ;
228
```

Figure 1: Reorder Buffer Data Structure

2.2. Register Files

The register files were implemented using a structure of fp_file_t and int_file_t which contain a unsigned value to hold the value written to the register file, busy to check if the register is being written to, and tag corresponding to the entry number in the reorder buffer.

```
//Integer General-Purpose Registers
struct int_fileT{
    int             value;
    int             busy;
    int             tag;
};

//Floating-Point General-Purpose Registers
struct fp_fileT{
    float           value;
    int             busy;
    int             tag;
};
```

Figure 2: Register File Data Structures

2.3. Reservation Stations/Load Buffers

The reservation stations and load buffers were implemented. This is shown in Figure 3.

```
//Reservation Station Data Structure
struct reservation_station_t{
    dynamic_instruct_pointer    dyn_instruction_p       ;
    unsigned                    vj                      ;
    bool                        vj_ready                ;
    unsigned                    vk                      ;
    bool                        vk_ready                ;
    unsigned                    qj                      ;
    unsigned                    qk                      ;
    unsigned                    tag_res_stat            ;
    unsigned                    addr                    ;
    int                         id                      ;
    bool                        pushed_2_exec           ;
```

Figure 3: Reservation Station Data Structure

3. Salient Aspects of Code

3.1. Register Renaming

Shown below in Figure 4 is the function for register renaming as well as receiving values from the reorder buffer from previous instructions.

```
unsigned sim_ooo::register_rename(unsigned reg, bool is_floating, unsigned& tag, bool& ready){
   unsigned value         = UNDEFINED;
   ready                  = true;
   tag                    = UNDEFINED;
   if(register_busy_check(reg, is_floating)) {
      tag                              = get_reg_tag(reg, is_floating);
      reorder_buf_t* rob_pointer       = rob.peekIndex(tag);
      if(rob_pointer->ready){
         value            = rob_pointer->value;
         tag              = UNDEFINED;
      }
      else{
         ready            = false;
      }
   }
   else{
      value               = read_register(reg, is_floating);
   }
   return value;
}
```

Figure 4: Register Rename Function

3.2. Conflicting Store

This function is used to find a conflicting store before a load. This checks if its conflicts by checking if the address is the same address as the load instruction, therefore producing a conflict.

```
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++){
      int tag                          = rob.genIndex(i);
      reorder_buf_t* robEntryP         = rob.peekNth(i);

      if( robEntryP->dyn_instruction_p->is_store ){
         if( robEntryP->dest == UNDEFINED ){
            conflict           = true;
            bypassReady        = false;
         }
         else if(robEntryP->dest == memAddress){
            conflict           = !robEntryP->ready;
            bypassReady        = robEntryP->ready;
            bypassValue        = robEntryP->value;
         }
      }
      if(tag == loadTag)
         return conflict;
   }
   return conflict;
}
```

4. Self-Grading

Test Case	Points	Comment
Test Case 1	5	Test Case Log and Reference Output Fully Match
Test Case 2	5	Test Case Log and Reference Output Fully Match
Test Case 3	5	Test Case Log and Reference Output Fully Match
Test Case 4	5	Test Case Log and Reference Output Fully Match
Test Case 5	5	Test Case Log and Reference Output Fully Match
Test Case 6	5	Test Case Log and Reference Output Fully Match
Test Case 7	5	Test Case Log and Reference Output Fully Match
Test Case 8	5	Test Case Log and Reference Output Fully Match
Test Case 9	5	Test Case Log and Reference Output Fully Match
Test Case 10	5	Test Case Log and Reference Output Fully Match

5. Conclusion

In conclusion, all testcases match and the Cycle-Accurate Simulator of a Dynamically Scheduled Processor was implemented correctly using Tomasulo algorithm.

6. Makefile Additions

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