ECE552 Lab 3 Report Judy Hanwen Shen 999735764 Rahul Chandan 999781801
Dynamic Scheduling with Tomasulo

Total Number of Cycles with Tomasulo

Running the following traces with –max:insn 1000000 gave the following total number of cycles to run using the Tomasulo architecture:

Benchmark Program	Total Number of Cycles
Gcc.eio	1697731
Go.eio	1713569
Compress.eio	1831248

Code Description

In general, the code in tomasulo.c is intended to simulate, at a high level, the functionality of the Tomasulo architecture specified for this lab with the end goal of finding the cycles per instruction if the code were to run on the actual architecture, with dependency checking and limited reservation stations and functional units. The below functions are implemented in reverse order, in order to preserve the per cycle capabilities of the Tomasulo architecture. The number of retired instructions counter is incremented when instructions are identified as traps or branches, or when the CDB has a non-NULL value.

Fetch

This function recursively checks instructions in the trace until the queue has free space and it finds a non-trap instruction. If the queue is full, the function returns. If the fetched instruction is a trap instruction, the function increments the fetch_index as well as the retired instruction count, and makes a recursive call (this effectively skips and 'retires' the trap instruction, as we were instructed to do). The function returns if the queue has space and it finds a non-trap function.

Fetch to Dispatch

This function calls fetch, which is guaranteed to return either if the queue is full, or if the fetch index is a non-trap instruction. For this reason, we recheck if the queue is full, and if we have not loaded all required instructions, we append the instruction to the queue and increment the fetch index. The instruction is assigned its dispatch cycle and the queue size is incremented.

Dispatch to Issue

This function’s main purpose is to find room for the instruction at the top of the instruction queue in an appropriate reservation station for its op-code. If the top of the queue is NULL, we break.

Otherwise, if the instruction at the top of the queue is a floating point instruction, we check if there is room in a floating point reservation station and push the instruction in. Same for integer instructions (including loads and stores). If there is no room, we stall until a reservation station frees up. If the instruction is not of either of these types, it must be a branch, so we skip dependency checking and dequeue the instruction from the top of the stack. Dependency checking consists of checking the map table entries for all the input and output dependencies of the floating point or integer instruction. If an input dependency exists, the register dependency from the map table is copied into the instruction metadata. All output dependencies from the instruction are also copied into the map table. After assigning its issue cycle, the instruction is dequeued from the instruction buffer.

Issue to Execute

In issue, we identify the oldest and second oldest instructions in the integer reservation stations without active dependencies and push them into any free spaces in the integer functional units in order of oldest to youngest. The same is done with the floating point reservation stations and functional units. If the functional units are all full then we stall.

Execute to CDB

The oldest instruction that has been in its functional unit for the required number of latency cycles (i.e. current cycle - execute cycle = required latency) is sent to the CDB, regardless if it's a floating point or integer instruction. If no instruction is finished executing, stall this stage. Otherwise, the reservation station corresponding to the instruction chosen to be sent to CDB is cleared, and all input dependencies on its return value are cleared, and its CDB cycle is assigned.

CDB to Retire

This function simply clears the CDB every cycle.

Is Simulation Done

If the number of retired instructions is equal to the number of instructions to be completed, then return true. Otherwise, return false.the simulatio

Testing

To test our code, we first used 10 instructions in the gcc benchmark added print statements at every stage to track the progress of each instruction and observe the status of reservation stations and functional units. At the fetch stage, we printed the indices of trap instructions and record the instruction as retired. At the fetch to dispatch stage, we print out the instructions being added to the queue and the queue position changes as instructions are removed from the queue as well as whether the reservation stations are full. At the dispatch to execute stage, we print out any dependencies, function unit states and instruction to be executed. In the execute to

CDB stage we print out instruction retired and registers cleared. In the retire stage we print out instructions to be retired and number of total instructions retired. Please refer to our commented out print statements for more details.

There were some bugs that we found only by running more instructions with the other benchmarks. Please refer to the following section for how we noticed and fixed them.

Toughest Bugs

Even though we encountered many segmentation faults while compiling the code, the two toughest bugs were 1)Self-register dependency and 2) two output register broadcasting.

When we tested our code, it would occasionally become stuck. We looked in our code and realized were assigning output dependencies before checking input dependencies. This means that if an instruction had the same input and output register, the instruction would first place a pointer in the map table to itself to assign the output dependency and then read the map table to store the input dependency as itself. This instruction was never ready to execute. To fix this, we simply had to check input dependencies before assigning output dependencies.

Another bug in our code became stuck when running 500 instructions for the go.eio benchmark. This was an interesting bug because the first benchmark ran fine with the original version of our code. We used GDB to break the program where it stalled and check the register values. We noticed that even though the reservation station for integers were completely full the functional units for integers were empty. We then noticed that the instructions all had dependencies on instructions that had already executed or retired. Then we realized that in the dispatch to issue stage, we recorded both output registers r_out[0] and r_out[1] in the map table but in execute to cdb we only cleared r_out[0]. We assumed initially that instructions would only use 1 output registered based on the examples given in class. However, since we were not broadcasting both output registers instructions looked at the map table and recorded input dependencies on instructions that had already executed. To fix this bug, we cleared r_out[0] and r_out[1] from the map table when an instruction retired.

Statement of Work Completed

We coded the functions together and bugged together. Rahul wrote the code description and benchmark sections of the report and Judy wrote the Bugs and Testing portion of the report.