Georgia Institute of Technology

ECE 6100: 2014(Conte)

Report: Tomasulo Algorithm Pipelined Processor

By: Harshit Jain

GT ID: 903024992

1. Introduction

The project comprises of a simulator for an out-of -order superscalar processor that uses Tomasulo algorithm. The simulator is used to find the appropriate number of functional units, fetch rate and result buses for each benchmarks. The project has been completed using C language. The tools used were Eclipse IDE on a Linux operating system (Ubuntu). GDB was used for code debugging and GIT is used for version control. Apart from this a python based script is used to automate the simulations and find the best solution for benchmarks.

2. Implementation: Source Code

2.1 Data structures

The project uses C++ standard template libraries to use existing data structures and methods associated with these data structures. The data structures used are:

i. Vector: A temporary fetch queue is constructed as a vector. As the fetch_rate changes so does the limiting size of the fetch queue vector.
ii. Deque: A Deque is used to realize dispatch queue. Dispatch queue acts as a FIFO where dispatched instruction are pushed at the end of the queue and scheduled instructions are popped out from front.
iii. List: A list is used to realize schedule queue. The schedule queue size is limited by the number of functional units provided.
iv. Map: A local hash map to store completed instructions with cycle values to print on stdout.

Apart from this structures are used as elements of above mentioned data structure. Fetch queue and dispatch queue share the same data structure. Whereas, schedule queue consist of another data structure which includes src ready flags and tag fields along with multiple status field about the instruction states.

2.2 Control Flow

The setup_proc sets the global variables like fetch_rate, sizes of functional units and result bus size. The function run_proc simulates the Tomasulo algorithm.

Steps on how the code executes is given below:

i. In its prologue the function sets the local variables like clock, fetch_instruction and various complete flags to default value.
ii. The complete algorithm resides in a while loop which runs till scheduling queue is empty after the fetching began.
iii. The program increments the clock at the very beginning indicating the present clock cycle for that particular iteration.
iv. At first the algorithm looks for any completed instructions in the output map which is ready to be printed. If found, the instructions are printed on stdout and subsequently deleted from the map.
v. Secondly, if any instruction was marked for deletion (markfordelete flag in schedule queue) in schedule queue then it is deleted in this cycle, freeing schedule queue with the deleted instruction's spot. Also the program frees the complete result bus is this segment.
vi. Thirdly, if any instruction which was completed in the previous cycle, i.e. completed execution and did status update in previous cycle. The instruction's markfordelete flag is set in schedule queue and this instruction is deleted in the next cycle.
vii. Then, the algorithm again checks schedule queue for any instructions that can occupy the freed result bus. The necessary check is being made on the schedule queue to find out the instructions which were the first to enter with least tag number to occupy the result bus. After occupying the result bus, the instruction does update the schedule queue instructions by clearing any dependencies and also updating register file with busy =0 and reg_tag =-1, also it frees the functional unit.
viii. Then, the algorithm consists of the execute segment where the all the ready to fire instructions in the schedule queue (instructions with no register dependencies) are fired on the availability of functional units. All the executed instructions are marked as fired and occupy the scoreboard.
ix. After this the scheduler gets instructions from dispatch queue if there are available slots in schedule queue. After this a check is performed on the instructions with no dependencies and they are marked as ready to fire.
x. The dispatch_queue gets the fetched instructions from fetch_queue which were fetched in the previous cycle.

xi. The fetch_queue fetches instructions as per fetch rate from the trace file.

xii. The fetching continues till every instruction is read from file, dispatching till dispatch queue gets empty and scheduling till there is no inst in scheduling queue.

xiii. In the end the calculated stats are updated to desired variables.

3. Simulation Results

The simulation is performed using a python script. The script runs for the given four traces and runs tests for every possible value of fetch rate, functional unit numbers and result bus size as given in the project description. The script consists of 2 iterations for fetch rate (4,8) , 2 for k0 functional unit, 2 for k1 functional unit, 2 for k2 functional unit and 9 for result bus. In total 144 tests are run for every trace. The script reads the output log of every test and preserves only those which pass the 95% IPC criteria given in the project description. Hence, we are left with only a few groups of logs which pass our requirement of minimum IPC. The name of the log contain the fetch_rate, functional unit and result bus size values for that log and we chose only those logs with minimum hardware in this group.

After running this script, the results for every trace was found to be same:

Answer: F =4, K0 = 2, K1= 2, K2 = 2 and R = 3.