Tomasulo-Softcore Library

Yihao Liu & Stephen Longofono

November 4, 2018

Introduction

This document outlines the Tomasulo-Softcore library, a Python library written to model a single-core software CPU based on Tomasulo's Algorithm. The source code is presented as and open-source library, and is available on Github.com under the MIT license. This library was developed as an academic exercise for the graduate-level computer architecture course at the University of Pittsburgh in Pennsylvania, U.S.A.

The library demonstrates all of the major facets of the algorithm, including register renaming, out-of-order execution, memory disambiguation, speculative execution.

Please reference the Github repository for the most up-to-date code, instructions, and test cases. The repository is hosted at:
https://www.github.com/SLongofono/Tomasulo-Softcore.git/

1 Building & Execution

1.1 System Requirements

The library was developed for Python 3.6+. No external libraries were used to develop the core library. The project was developed and tested in a Linux environment, in a directory with write and execute privileges. Note that the output files will be placed in the same directory as the input files passed in; in order to correctly generate an output file, the user must have write privileges to the directory with the input files.

Included in the repository is a shell script used to generate UML diagrams from the class hierarchy. If you choose to modify or extend the library, the script can be run again from the root directory of the repository to generate fresh images of the UML class diagrams. Running this script requires Pylint 2.2.1+, but is not otherwise necessary for the use of the core library.

1.2 Build Instructions

There are no build instructions for the library, as it is a standalone Python module. The entry point to execution is Tomasulo.py. To start the library and print usage instructions, simply invoke your Python interpreter with Tomasulo.py as an argument from the command line.

1.3 Running Test Inputs

All test inputs are passed as command line arguments. The test inputs all follow a rigid and ordered format, as demonstrated by the file "example_input.txt" in the "testCases/" directory of the repository root. To execute the library on a test input file, type:

```
<Python3.6> Tomasulo.py <filepath>
```

where Python3.6 is the CLI invoker for your Python 3 interpreter, and filepath is the absolute filepath of the desired test input.

1.4 System Outputs

The library will output execution information in two ways; during execution, progress and the contents of various system components are displayed to **stdout**. Upon completion, an output file will be written to the directory containing the input file. The output file will be the same name as the input file, with "_output" appended just before the file extension.

2 Verification & Testing

In order to verify a proper and complete implementation of Tomasulo's algorithm, we developed a series of progressively more complex test inputs. All of the test cases used are provided in the repository under the "testCases/"

directory. For convenience, they are repeated here along with a short description of which feature they verify.

2.1 simple_test1.txt

This test was designed to test a pair of independent integer ALU instructions, as a verification that our system could correctly process both types of Integer ALU instructions in isolation. It also serves as a verification of the general flow of the program, excluding the memory stage.

Input

```
  # of rs Cycles in EX  Cycles in Mem # of FUs
Integer adder 2 1    1
FP adder  3 3    1
FP multiplier 2 20     1
Load/store unit 3 1 4 1
ROB entries = 128
CDB buffer entries = 1
R1=10,R2=20,R3=30,F2=30.1
Mem[4]=1,Mem[8]=2,Mem[12]=3.4

ADD R1,R2,R3
SUB R4,R2,R3
```

Output

```
==============================Instruction Completion Table===================

ID | IS  EX  MEM  WB  COM

0 | 0  1  None  2  3

1 | 1  2  None  3  4


==================================Integer ARF=================================
```

```
R0: 0 R1: 50

R2: 20 R3: 30

R4: -10 R5: 0

R6: 0 R7: 0

R8: 0 R9: 0

R10: 0 R11: 0

R12: 0 R13: 0

R14: 0 R15: 0

R16: 0 R17: 0

R18: 0 R19: 0

R20: 0 R21: 0

R22: 0 R23: 0

R24: 0 R25: 0

R26: 0 R27: 0

R28: 0 R29: 0

R30: 0 R31: 0


=================================Floating Point ARF===========================

F0: 0.000000 F1: 0.000000

F2: 30.100000 F3: 0.000000
```

```
F4: 0.000000 F5: 0.000000

F6: 0.000000 F7: 0.000000

F8: 0.000000 F9: 0.000000

F10: 0.000000 F11: 0.000000

F12: 0.000000 F13: 0.000000

F14: 0.000000 F15: 0.000000

F16: 0.000000 F17: 0.000000

F18: 0.000000 F19: 0.000000

F20: 0.000000 F21: 0.000000

F22: 0.000000 F23: 0.000000

F24: 0.000000 F25: 0.000000

F26: 0.000000 F27: 0.000000

F28: 0.000000 F29: 0.000000

F30: 0.000000 F31: 0.000000
```

2.2   simple_test2.txt

This test was designed to test a pair of independent floating point ALU instructions, as a verification that our system could correctly process both types of floating point ALU instructions in isolation. It also serves as a verification of the general flow of the program, excluding the memory stage.

Input

Output

2.3   simple_test3.txt

This test was designed to test a pair of independent floating point multiplier instructions, as a verification that our system could correctly process floating point multiplier instructions in isolation.

Input

Output

2.4   simple_test4.txt

This test case was designed to test the load and store instructions in isolation. It also serves as a verification of the general flow of the program, excluding the branch unit.

Input

Output

2.5   simple_test5.txt

This test case was designed to test the branch unit with a single branch instruction. It serves to test the simplest functions of the branch unit, including storing copies of the RAT and correct branch prediction.

Input

Output

2.6   simple_test6.txt

This test case was designed to test corner cases of the system configuration. This test will issue instructions to saturate the capacity of all functional units by filling the output buffers and the reservation stations. The goal is to prove that the library can make forward progress in the face of a demanding load.

Input

Output

2.7   simple_test7.txt

This test case was designed to verify the absence of WAW hazards, a behavior that the algorithm was designed to remove.

Input

Output

2.8 simple_test8.txt

This test case was designed to verify the absence of WAR hazards, a behavior that the algorithm was designed to remove.

Input

Output

2.9 simple_test9.txt

This test case was designed to verify that the system is capable of handling RAW dependencies. Several truly dependent instructions are issued for each of the types of functional units, to test the ability to complete execution as prescribed by Tomasulo's algorithm.

Input

Output

2.10 simple_test10.txt

This test case was designed to verify correct operation of the memory unit. Specifically, it tests that the memory unit is able to disambiguate dependencies among loads and stores, and ensure both correctness and precision of execution in the memory unit.

Input

Output

2.11   simple_test11.txt

This test case was designed to induce failure (exceptions) in the case of illegal instructions. This includes writing to read-only registers, malformed instruction names, malformed register names, and improper types in the instruction input.

Input

Output

2.12   complex_test1.txt

This test case was designed to test a more true-to-life program, involving all the instructions in the ISA. Dependencies exist among the instructions, and there are a sufficient number of instructions to induce a few stalls due to architecture.

Input

Output

2.13 complex_test2.txt

This test case was designed to test the ability of the branch unit to handle a failed speculation in isolation. The predicted branch target is a group of long-latency instructions, ensuring that the branch unit is able to properly recover and entirely remove the speculative results from the data path.

Input

Output

2.14 complex_test3.txt

This test case was designed to further test the ability of the branch unit to handle failed speculation. Two mis-predictions in a row are induced, in order to prove that the branch unit can properly back out speculations from two execution paths.

Input

Output

2.15 complex_test4.txt

This test case was designed to further test the ability of the branch unit to handle failed speculation. Three branches are used in succession, with the first and third mis-predicted and the middle branch correctly predicted. We chose this test because we are sadomasochists.

Input

Output

2.16   complex_test5.txt

This test case was designed to test the limits of the behavior of both the branch unit and the system. A dependency is induced in the branch condition, followed by many instructions at the predicted target. The branch condition is mis-predicted, and the result is a more onerous recovery than in previous tests.

Input

Output

2.17   complex_test6.txt

This test case was designed to test our model of system memory (RAM). Here, we write a series of patterns to memory to verify that loads and stores are executing in the correct order, and that data is written to the correct location.

Input

Output

2.18   complex_test7.txt

This final test case was designed to show that our system can do real work. We execute a recursive version of a Fibonacci numbers procedure to induce a deep chain of RAW dependencies.

Input

Output

3 Division of Labor

3.1 Responsibilities

We developed our code and test cases together. Stephen was more experienced with Python, so for productivity's sake he wrote much of the code. Yihao helped by developing the general flow of the program, writing effective test cases, identifying areas for improvement in the design, and general debugging.

3.2 Code Contribution

Yihao developed the floating-point functional units and the memory unit. Stephen developed the other main components of the library. Stephen wrote the first pass of the five execution stages, and then worked with Yihao to debug them as each test case was visited. More specific details on the day-to-day code contributions are available in the commit history of the Github repository.

3.3 Test Contribution

Stephen wrote the simpler test cases as a litmus test for the first pass of the data path. Stephen helped come up with the ideas for the more exhaustive tests. Yihao wrote the bulk of the test input files. Both team members completed all tests by hand to verify that our expected results were consistent.

3.4 Meetings & Participation

Since there were only two members on our team, it was easy to work together and hold productive meetings. We spent several weekends coding side-by-

side, and maintained effective communication throughout the project.