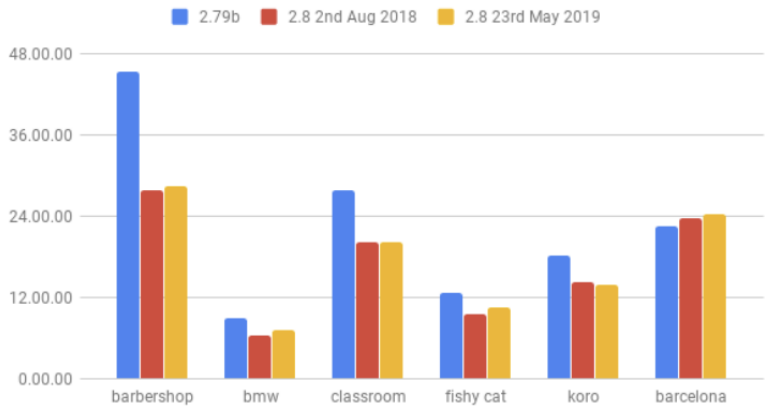
Lab 1: Performance and BenchMarking

CPE 333 - W24



Abstract

In this lab, you will design a set of benchmarks and measure their performance on the OTTER (our 233 processor). The set of benchmarks (e.g. matrix multiplication and addition) are representative of many important applications that are used in systems today. This lab will give you more practice with assembly programming, the RISCV ISA, and the basic multi-cycle processor design. After implementing and testing your benchmarks, for the second part of the lab you will measure the performance of the multi-cycle processor (with a 1-cycle latency memory and a variable latency memory). Learning Objectives:

- Understand basics of an instruction set architecture
- Understand a basic multicycle processor microarchitecture
- Abstraction levels, including register-transfer-level modeling
- Design principles including modularity, hierarchy, and encapsulation
- Design patterns including control/datapath split

1 Introduction

Performance, energy, power, reliability are key attributes when designing computers. Accurately measuring and comparing different computers is critical for designers. The performance of a computer is inversely proportional to is execution time, the total time required for a computer to complete a task (including disk accesses, memory accesses, I/O, OS overhead, CPU execution time, etc.). For many computers, designers may also be care about throughput or bandwidth, which is the total amount of work done in a given amount of time. Execution time (performance) is typically measured in seconds. The smaller the execution time, the faster the computer. To evaluate two computer systems, a user would simply compare the execution time of a workload on the two computers (a workload is a set of representative programs that would typically run on a given computer). In most cases actual user workloads are not used, instead designers choose a set of benchmarks (i.e. programs specifically chosen to measure performance) – that will comprise the workload and hopefully accurately predict the performance of the actual workloads. Benchmarks play an important role in computer architecture, because in order to make the common case fast (we will discuss more on the principle later –also known as Amdahl's Law), we need to know which case is common.

2 Assignment

1. Install (if you don't already have it the RISC-V toolchain).

   For **WSL**: "Using WSL instead of Virtualbox.pdf"

   For **Docker Installation**. Some Students last quarter had problems installing this toolchain so they created a Docker for this project you can find the instructions here: `https://github.com/realrusssobti/Painless-RISCV` Note: You have sample code in C and the compiled assembly in file tools/Example Otter Programs.tgz

   Note 2: Once you have the Toolchain installed then you can compile the c code into assembly (.s). Place you assembly ('.s' or '.S') source files in the src directory. Ensure that you do not have multiple source files with the same name (e.g. 'main.c' and 'main.s'). Run 'make' to build the software. This produces a build directory, containing:

- - object files (‘.o‘) produced from your source code
- - a ‘mem.txt‘ file which is the memory image for use in the simulator
- - a ‘program.dump‘ file which is the full disassembled program output.
- - and some intermediate ‘.bin‘ and ‘.elf‘ files which you can ignore.

You will need to load the x.txt file into the otter memory to get the performance evaluation

2. Develop three benchmark assembly programs:

   - matadd: Matrix addition. Compute the addition of two MxN matrices.
   - mul: Multiplication. Multiply two numbers together.
   - matmul: Matrix multiplication. Computer the multiplication of two NxN matrices

   Assume the matrices have an equal number of columns and rows. **Note**: I have posted C code that you can use for matmul on canvas matmul.tgz and a samples initialized matrix on file datasets.zip. You need to install the Risc-V toolChain on your and compile the provided C code and data into RISC-V machine code (as explained in previous step). You can create your own assembly code BUT careful with memory address initialization , it needs to match the otter to work correctly.

3. Setup the baseline OTTER (multi-cycle) and verify the processor with the $test_all.s$ program on Canvas. You can use your OTTER design from 233 or the reference design posted on Canvas ($OTTER - multicyle - 1 - cycle - memory.zip$).

   - Measure the performance of the $test_all$ program (e.g. to cycle through all instruction test cases, the total number of cycles).
   - Measure the performance of your benchmarks from part 1 on the OTTER, for matrices of size (3x3, 10x10, and 50x50).

3 Deliverables

:

1. List of Group Names and a table with what work was performed by each member of the group
2. Short paragraph that provides an overview for all of your designs.
3. Answers to all questions, including bar graphs (and tables) showing your measurements.
4. HDL and other code for all designs