430 Course Project:
Project Team

Developed by:
Jacob Kenney, 48610961
David Shriver, 07007741

Wiki Page:
http://projectteam.wikia.com/wiki/ProjectTeam_Wiki

Completed:
April 23, 2015

Abstract:

In this project we were challenged to design a pipelined RISC processor with hardware-based data hazard detection and resolution, static branch prediction, and set-associative instruction and data caches. To accomplish this goal, we relied on the following five ideas: simplicity favors regularity, a good design demands compromise, the common case should be fast, utilization of locality, and parallelism. This document goes in depth into the design of our processor and the reasons for our decisions as they relate back to these main ideas. We start by giving an introduction to the concepts that we needed to understand to accomplish our goal. We then describe our design and decisions, and finally, we provide an evaluation of our design.

Introduction:

RISC ISAs:

Reduced Instruction Set Architecture (RISC) is a CPU design strategy that favors simple instructions that can be executed in only a few cycles. The two key characteristics of RISC instruction sets are that each instruction fits in a single word of memory and all operands of arithmetic or logic operations must either be in a processor register, or given explicitly within the instruction word (Hamacher, 2012, p.34). Our Instruction Set is a RISC style design, with 24-bit word instructions. We keep the instructions simple by having the first 8 bits of each instruction have equivalent meaning. This allows us to parse each instruction almost identically. Because our ISA is a RISC design, we also only perform operations on registers or registers and immediate values, depending on the instruction. Our ISA is explained in more depth in the Design section.

Pipelines and Hazards:

Pipelining allows us to increase the number of instructions per cycle by allowing us to execute instructions at the same time and exploit parallelism. This is possible in our processor by executing five instructions at the same time, each in a different stage, and propagating each instruction down the pipeline. Because we are executing multiple instructions at once, we can run into hazards. These include data dependencies and control dependencies. Data dependencies arise when one instruction needs to read from a register before a previous instruction has time to write back to that register (Hamacher, 2012, pp.197-198). We can alleviate some of these dependencies by forwarding values between stages before they get written back. Others require the processor to stall until the value is available. A control dependency occurs when we have a conditional branch instruction because we don't know which path we should load the next instruction from.

Branch predictors:

A branch predictor predicts which path a conditional branch will take to try and reduce the cost of a mis-prediction. Our design uses a static branch predictor that always predicts the same path will be taken, but branch predictors exist that look at what path previous branches took.

Memory hierarchy:

Memory in a computer is organized in a hierarchy. In the processor, at the lowest level we have the registers, that can load or write data to the primary cache (L1 cache). The primary cache can read or write to a secondary cache (L2 cache). This cache is the highest level memory in the processor, and accesses its data from the main memory. The main memory is followed by the magnetic disk or secondary memory (Hamacher, 2012, p.288). In this hierarchy, the further we get from the processor,

the slower it is to access the memory. Because of this, cache design is very important to processor speed. In our design, we have two L1 caches, one for instructions and one for data.

Design:

System Overview:

Our design is a basic RISC style design. It is 5 stages and fully pipelined with hazard detection and resolution. Our instruction set uses 24-bit instructions and utilizes sixteen 16-bit registers. It uses static branch prediction on branches by predicting "not taken". It also features instruction and data caches.

Pipeline:

Our pipeline is five stages. The stages are: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (M), and write back (WB). We also have full hazard detection and resolution. We have full forwarding for when values are needed before they get written back to the register file. The pipeline can also be stalled for certain hazards, for instance, when an arithmetic operation uses a value from a memory access in the immediately preceding instruction. We can also stall the entire processor on a cache miss, allowing us to load the data from memory. On certain hazards, such as a branch mispredict, we must also flush some stages of the processor that have invalid instructions.

Branch Prediction:

Our branch prediction scheme is to always predict that a branch will not be taken. We chose this design because we thought it was simpler to implement. On a branch mis-prediction we flush the IF stage and set the PC to the correct value. This leads to a no-op "bubble" that wastes one cycle. On a correct prediction we do not lose any cycles.

Caches:

Both of our caches are two way set associative and are 512 bytes each. Each cache contains only one set, with two entries (because it is two way set associative). Each entry has a block size of 256 bytes (128 16-bit words). We chose this size because each entry can hold an entire string. The offset is specified by the lowest 7 bits. Because the size of our main memory is only 256 words, we only need one bit for the tag. We only store two bits of meta data, the tag, and whether the entry is valid. If the tag matches the address we are accessing but the entry is invalid then we must load from the main memory. When we have a cache miss, we stall the processor and prefetch all 256 words of memory. This means that a cache miss is expensive, but every memory access in the rest of the program only takes a single cycle. This makes the common case as fast as possible. Also, because we do not have any write memory instructions, we do not implement a write-back policy.

Instruction Set Architecture:

Because the processor was designed specifically to be able to find the longest common substring, our RISC ISA only required a few simple instructions used to implement the algorithm: add, add immediate, load word, subtraction, and branch. Although others were included in the design, these were the only ones used and are therefore the only ones excessively tested. Every instruction is comprised of an opcode that specifies the type of instruction along with a condition code that specifies any conditions for it to execute with. With these 24-bit instructions, there are three types we can have:

Register-Register (R-type) instructions, Data Transfer (D-type) instructions, and Branch (B-type) instructions.

R-Type:

Our R-type instructions include the add and subtraction and are structured as follows:

23...20	19...16	15...12	11...8	7...4	3...1	0
Op	Cond	Rs	Rt	Rd	Opx	Set Bit

Where the top row is the bits it is represented by, and the bottom row is the components of the instructions. The opx code for R-type instructions is utilized to tell the ALU specifically what instruction is being done, beings many R-type instructions share the same opcode. The set bit is used to indicated whether the given instruction should test the condition flags or update them during its execution. If the set bit is set to one, the given instruction will update the condition flags within the processor based on the result the instruction returns. If the set bit is not set, the previous values continue to exist in the flag register.

Below, the two R-type instructions and their opcode and opx code are listed:

Instruction	Add	Subtraction
Op (in Hex)	A	A
Opx (in Hex)	4	0

D-Type:

Our D-type instructions include the add immediate and load word and are structured as follows:

23...20	19...16	15...12	11...8	7...1	0
Op	Cond	Rs	Rt	Immediate	Set Bit

The set bit behaves in the exact same way in D-type instructions as it does in R-type. The difference between R-type and D-type instructions is the ability for D-type instructions to use an immediate value for the second input.

Below lists our D-type instructions and their opcode:

Instruction	Load Word	Add Immediate
Op (in Hex)	8	C

B-Type:

The only B-type instruction we used is the branch, structured as follows:

23...20	19...16	15...0
Op	Cond	Label

The label for B-type instructions marks where the program counter should move to when a branch is calculated. The label is specified as an offset to the current instruction address.

Below lists our B-type instruction and its opcode:

Instruction	Branch
Op (in Hex)	0

Conditional Execution of Instructions (Cond):

Every instruction we implemented for the processor has a set of bits (19...16) which can be used for conditional execution. That is, an instruction will only execute if, based on the given condition code, the corresponding condition flag(s) are set. Conditional execution allows for us to expand the amount of instructions we can do based off a basic instruction. Branch was incredibly reliant on the use of conditional execution because it allowed for a branch to be based off a wide array of possible comparisons (ex: branch equal, branch not equal, branch greater than, etc.). Condition codes we used are listed below:

Code (in Hex)	Meaning (abbreviation)
1	Always (al)
A	Greater or equal (ge)
B	Less than (lt)
D	Less than or equal (le)
E	Equal (eq)
F	Not equal (ne)

Evaluation:

We utilized ModelSim heavily in our debugging process and to verify that the programs we were using were correct. When we first began, we started by testing individual components using ModelSim-Altera. As we progressed and got a basic processor design, we also started testing on the Altera DE2 board, mainly using ModelSim for testing. It was extremely useful in visualizing the different signals within our processor to find mistakes. When we started, we only used short programs to test so we could make sure that all cases worked. We then slowly started building our LCS program, first by finding the lengths of the strings and viewing a simulation in ModelSim to verify that the program was actually doing what we wanted. We eventually built an algorithm that worked in both ModelSim and ran correctly on the DE2 board.

Conclusion & Suggestions:

Through the project, we reused our knowledge from Computer Organization to implement a basic processor and then manage to go further, causing it to be more abstract than before. This involved implementing a fully working pipeline with data hazard detection and forwarding, a static branch

predictor, and implementing two cachse, one for instructions and the other for data. Through the tedious process of implementing these features, we improved on our skills to read a gate level simulation and think outside of the box to help resolve issues when trying to implement our RISC processor. We managed to successfully obtain a working processor capable of finding the least common substring that implemented all the requirements following the five criteria: simplicity favors regularity, a good design demands compromise, the common case should be fast, utilization of locality, and parallelism. As for suggestions towards the TA and professor, we greatly would have benefitted from more information being given on how to actually design a cache for the processor, it was probably the part we struggled with the most. We think it would have been nice to discuss the project in class a bit more, or have a lab or recitation for the class.

References:

Hamacher, V. Carl. *Computer Organization and Embedded Systems*. 6th ed. New York, NY: McGraw-Hill, 2012. Print.