|  |
| --- |
| Department of computer science & Engineering  University of Nebraska—Lincoln |
| RISC Based Processor |
| Processor Architecture and Hardware Design Project |
|  |
| **Conner Bell, Joshua Behlen, Peyton Owens** |
| **11/28/2017** |

|  |
| --- |
| This document will detail the design and architecture of a simple processor using a concurrent hardware description language, and Reduced Instruction Set instructions |

Contents

[1. Introduction 3](#_Toc497320003)

[1.1 Purpose of this Document 3](#_Toc497320004)

[1.2 Definitions, Acronyms, Abbreviations 3](#_Toc497320005)

[1.2.1 Definitions 3](#_Toc497320006)

[1.2.2 Abbreviations & Acronyms 3](#_Toc497320007)

[2. Component Description 3](#_Toc497320008)

[2.1 Processor Design 3](#_Toc497320009)

[2.1.1 Processor Testing Strategy 3](#_Toc497320010)

[2.2 Current abilities 4](#_Toc497320011)

[2.3 Speed overview 4](#_Toc497320012)

[2.3.1 Speed Testing Strategy 4](#_Toc497320013)

[2.4 Componentry 4](#_Toc497320014)

[3 Group Experience 4](#_Toc497320015)

# Introduction

RISC (reduced instruction set computer) processors are designed to have a very simplified and highly structured set of instructions, favoring low clock cycle instructions that can be performed as quickly as possible to maintain efficiency. RISC also uses fixed instruction length to allow for pipelining. Pipelining is an architectural principle that allows for 1 instruction execution per clock cycle due the ability of an instruction to be at each stage of processing after the initial 5 stages. RISC is the computing method of processors for smaller devices, as the minimal load on the processor per clock cycle increases power efficiency. The processor described in this document is a 5 stage RISC based processor that uses a custom instruction set with an instruction length of 24 bits. The purpose designing this processor is to gain an understanding of low level computer processes, and to become familiar with the data path that data within your computer takes in order to achieve certain objectives. A hardware description language (VHDL) using a concurrent programming paradigm is used in tandem with a field programable logic array in order to achieve a simulated processor architecture that can be implemented using gate-level logic.

## Purpose of this Document

This document will detail the design and architecture of a simple processor using a concurrent hardware description language, and Reduced Instruction Set Computer instructions.

## Definitions, Acronyms, Abbreviations

### Definitions

RISC – Reduced instruction Set Computer

FPGA – Field Programmable Logic Array

Arithmetic logic unit – A circuit used to perform arithmetic and logic operations.

Register – A component/entity that loads/accepts data when specified, and holds the data for later use

Clock – The wave form, or switching bit value that dictates when the processor components are active, as well as the speed of the processor as a whole.

Decoder – A device that takes an inputs and outputs multiple unique values based on different combinations of bit values.

Instruction Register – Register for holding the instruction that is currently being operated.

Multiplexer – A device that selects one of several inputs and forwards that to a single output.

### Abbreviations & Acronyms

ALU – Arithmetic logic unit

IAG – Instruction Address Generator

MIF – Memory Initialization File

IO – Input/Output

IR – Instruction Register

Mux – Multiplexer

PS – Program Status

xor – Exclusive Or

b – Branch to the location specified following this command

bal – Branch and link, it’s an assembly command stores address of next instruction in 1st operator and then jumps to the address in the 2nd operator.

cmp – An assembly command that performs a comparison between two arguments by signed subtraction of 2ndarg – 1starg

jr – jump register

lw – load word (actually a 24 bit value)

sw – store word (actually a 24 bit value)

j – jump

addi – add immediate value

subi – subtract immediate value

add – add registers

sub – subtract registers

# Architecture and Testing

The following section will entail the components contained within the processor at each point, the speed overview of the processor at each point, and the capabilities of the processor at each phase of its design.

## 2.1 Processor Overview

This processor uses a RISC based instruction set, requiring 5 stages, or clock cycles, for each instruction. The stages are: Instruction Fetch, Loading, Arithmetic/Execution, Memory, and Write Back. Instructions are fetched from a memory initialization file; necessary flags are modified in stage 2; any arithmetic is executed in stage 3; data is loaded from/stored to memory in stage 4 for some instructions types; and finally, any new data is written back to a register if necessary. This design allows for conditional execution of instructions as well memory mapped IO. Further detail of the design and architecture can be found in section 2.4.

## Capabilities

Phase 2 was the first phase that allowed for the use of instructions and data output. The processor was capable of executing the following basic R-type instructions during this phase:

|  |  |  |
| --- | --- | --- |
| Instruction | Example | Description |
| add | add r2,r3,r4 | stores the value of r3 + r4 into r2 |
| sub | sub r2,r3,r4 | stores the value of r3 - r4 into r2 |
| and | and r2,r3,r4 | stores the bit string produced by r3 and r4 into r2 |
| or | or r2,r3,r4 | stores the bit string produced by r3 or r4 into r2 |
| xor | xor r2,r3,r4 | stores the bit string produced by r3 xor r4 into r2 |

In phase 3, due to the addition of a memory interface, the following commands were able to be executed in addition to those in phase 2:

|  |  |  |
| --- | --- | --- |
| Instruction | Example | Description |
| jr | jr r15 | jumps to the memory address stored in register 15 |
| cmp | cmp r2,r4 | compares (subracts) r2 and r4 |
| lw | lw r2,(0)r4 | loads the contents of the address of r4 + an offset into r2 |
| sw | sw r2,(0)r4 | stores the contents of r2 into the address of r4 + an offset |
| addi | addi r2,r0,8 | stores the value of r0 + 8 into r2 |
| subi | subi r2,r2,8 | stores the value of r2 - 8 into r2 |
| b | b LOOP | branches to the location of the LOOP label relative to the current address |
| bal | bal LOOP | branches to the location of the LOOP label relative to the current address, and stores the next instruction as the return address in r15 |

In phase 4, conditional execution of instructions was added, which is explained further in part 2.4. Due to the addition of an IO memory interface, the following IO inputs and outputs were able to be loaded and stored to memory (red LEDs were implemented in phase 5):

|  |  |
| --- | --- |
| IO input | Data In |
|  | 1010 |
|  | 0000001101 |
| Data Out | IO output |
| 01011100 |  |
| 0000011101 |  |
| 1001111 |  |

In phase 5, 3 additional 7-segment hex displays were implemented, along with the red LEDs above, and one additional instruction:

|  |  |  |
| --- | --- | --- |
| Instruction | Example | Description |
| j | j LOOP | jumps to the memory location of LOOP |

### Testing

The testing for phase 2 was conducted by hard coding instructions into a .DO file used by the simulation software, and observing the waveform/data outputs of each instruction. These basic R-type instructions were simple to verify, as the byte code for the output was displayed in binary using the simulation software.

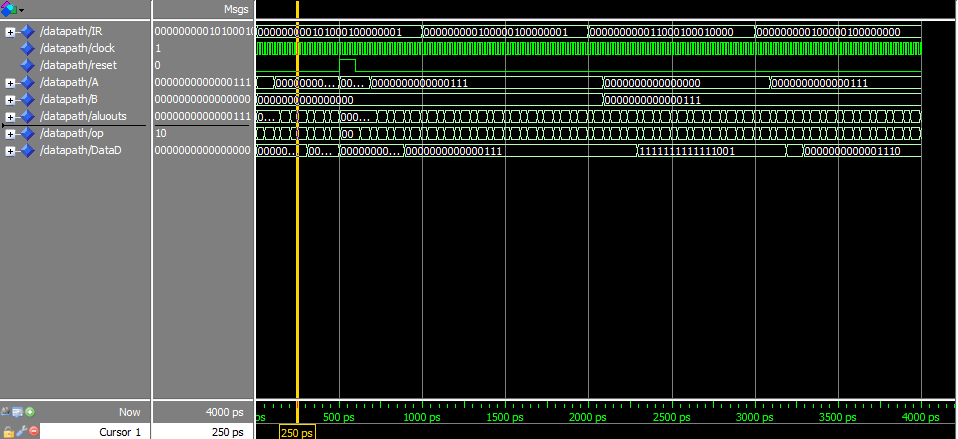


Figure 1: A graphical representation of the functioning of the processor as time progresses.

For phase 3, to test the memory and instructions simultaneously, junk data was inserted into memory, which was to be passed over using branch and jump instructions, and values were stored into memory, and loaded into a different register; if not junk data was observed in the output, and if the loaded data matched the stored data, then it was confirmed that the memory interface and instructions were working properly.

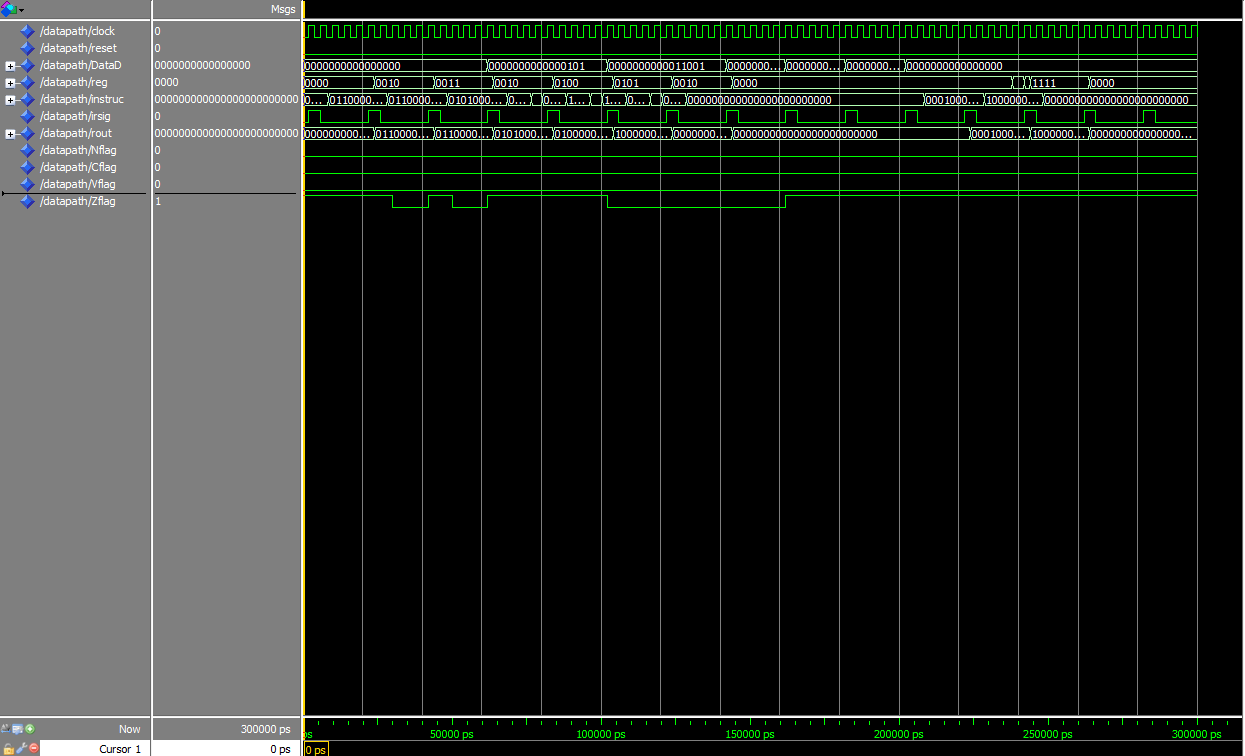


Figure 2: A graphical representation of the functioning of the processor as time progresses.

For phases 4 and 5, testing was conducted initially by hard coding input data, and storing that data into the output IO to verify that data would be displayed properly. Next, to test the input IO, pins on the board were mapped to the input described in VHDL code, at which point the initial testing was repeated using live input data from the IO with the same values as the hard-coded data for a comparison of behavior.

## Speed overview

During phase 2, the observed highest clock speed of the processor is at 500 GHz. This was achieved only due to the simplicity of the instructions at the time.

During phase 3, the observed highest clock speed of the processor is around 1 GHz. This large drop in speed was due to the delay in loading and storing of data in memory.

During all subsequent phases, the clock speed was limited to 50 MHz due to the specifications of the FPGA board, which was now in use, and due to the delay when storing and loading from memory exclusive to the IO components.

### Speed Testing

Speed was tested by increasing the clock speed by increasingly small increments to the determine the exact speed at which the processor would cease to function properly.

## Componentry/Architecture

Initially the processor consisted of an ALU for arithmetic operations on data within registers, a register bank for data to be stored, and control unit that controlled the many flags and enabling signals needed for proper functioning of the processor. The ALU operated using gate-level arithmetic, taking two values as inputs as well as a carry-in bit in the case that a subtraction was performed; Because of the way values are converted into their negative equivalent using 2’s compliment, it is necessary that the carry in bit be used to add one to the inverted value of the pseudo negative input. Many intermediate registers where used in this phase, and carried through to phase IV, for capturing data to be used in subsequent clock cycles. During phase 2, instructions were hard-coded into the processor due to a lack of a memory interface.

In phase 3, a memory interface was added, which uses virtual memory to store instructions as well as data. This block took in a memory address, and two signals dictating whether the data was being written or read from the address, producing a single piece of data that could be selected from among other values with a mux to be stored back to a register. An instruction address generator was also added to that would provide the address for the next instruction. The default behavior is to increment the address by 3 bytes (24 bits) every 5 clock cycles, in order to achieve sequential instruction behavior. This could be altered with a branch or jump command, which used a mux to select a different value to be added to the current memory address, or specified an address with data that would be passed through the IAG as the new address. These components allowed a MIF to be used, and instructions to be pulled from memory automatically, meaning that a program could be written in byte code and run on the processor at this phase. The outputs of this program could be observed using a gate-level simulation tool to view the waveforms produced by the processor.

Phase 4 saw the addition of an exclusive IO memory interface that could be used to store data to LEDs and 7-segment hex displays, or load data from switches and buttons. With the addition of IO, output data could be observed directly on the FPGA boards, and simulations were no longer necessary. The IO used consisted of a 7-segment hex displays, 8 green LEDs, 4 buttons, and 10 switches. Conditional instruction execution was added to the control unit at this stage. These conditions were checked using 4 flags that were set by the control unit based on the outputs of prior instructions. For example, if two equal value registers are compared using the cmp instruction, which will subtract the registers from one another, a Z flag (zero flag) will be set to ‘1’; this would allow the control unit to identify when a conditional instruction will be executed.

In phase 5, three more 7-segment hex displays, and 10 red LEDs were added to the IO memory interface. During this phase, additional commands such as jump were added, and the ripple-carry adder component of the ALU was replaced with a carry look-ahead adder. The carry look-ahead adder uses simple logic to propagate the carry-in bit across each of its four full adders with isolated logic, in a way that is non-reliant on the sum of the previous adder. In this way, the carry bit can be determined before the sum, which is much more efficient when calculating sums of large numbers. In contrast, the ripple-carry adder must wait for the sum and carry bit of the previous adder component to correctly determine the value of the sum. This phase also entailed creating an assembly program, which was translated into byte code using the assembler developed for this project, that highlighted all instructions and IO performance that the processor is capable of. This program is described in detail in section 2.5.

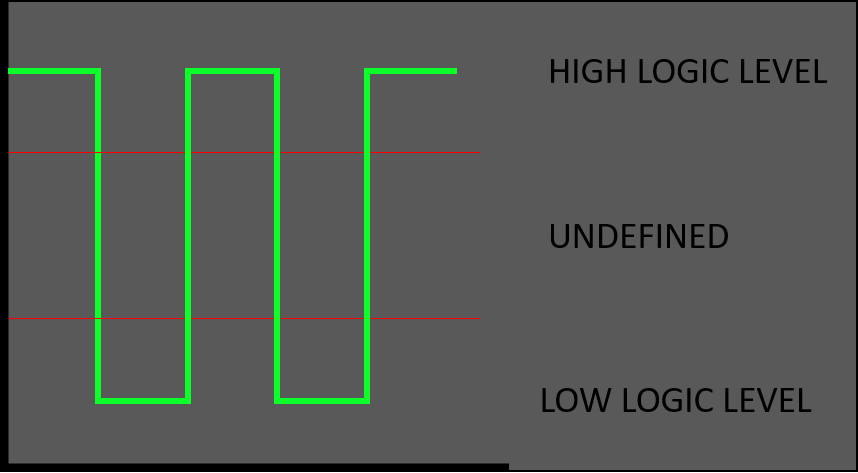
## Assembly and Assembler

In order to translate the assembly into byte code for the MIF, a two-pass assembler program was written in Python 3, which parsed the assembly code line by line, tokenizing each instruction into their command, register values, and offset values. The assembler would then use these tokens to look up the corresponding bit values for each token in one of four different dictionaries; these dictionaries were associate with op codes, opx values, register values, and instruction type. The memory location of each instruction, starting from word 1, or byte 4, was incremented once for each instruction, and printed out in the MIF beside the byte code for the instruction. When a label was read into the assembler, the memory location of the label was stored into a new dictionary with the label as its corresponding key, and printed in plain text next to the byte code, with an ascii indicator for later use. All of this was written to an intermediate file. A second pass of the assembler would look for the ascii indicator (character ‘?’), and use the plain text label to look up the binary memory location in the dictionary, which it would then replace the label in the intermediate file with; this new file would be saved as the final MIF. Due to the logic needed in the assembler, the assembly programs had to be written in a highly structured format. Once translated into a MIF, the MIF could be loaded into memory on the board, and the translated assembly instructions could be processed as byte code by the processor, proceeding the data path as needed.

# Challenges and Failures

Initially, the challenges our group experienced were setting up files in GitHub so that all project members have equal access to the files at any given time. GitHub is a very capable software, but does have a learning curve that must be overcome.

After working on phase 3 for a few days, the phase was almost complete, and debugging had begun; It was realized that incorrect or extraneous data was being directed to the output in the simulation. This led the team to believe that there were fundamental issues with the operation of the processor. After many hours of debugging to no avail, the current version of phase 3 was scrapped, and restarting using the final version of phase 2. This allowed the team to organize the code and have a preconceived notion of what possible issues could arise. Phase 3 was re-implemented making sure that all components were being mapped correctly. Once this was complete, the same extraneous data was observed. It was then realized that this data was to be expected, and since the write enabling flag was not set to ‘1’ during these extraneous data outputs, the data was not actually affecting the processors functions. Because of this, the team is unaware if restarting phase 3 was necessary or not, but this misunderstanding/challenge was a major set back in developing the processor.

An issue during phases 4 and 5 was indeterminate values leading to stray signals being output to the IO. This was observed by arbitrary IO being activated when no data was being written, or by partially lit LEDs on the board. Indeterminate values arise when there is an undefined value being passed to a signal or output, meaning that the logic can no longer be interpreted correctly. These issues can also arise from different race conditions if the processor’s design allows for such conditions, or from noise causing unexpected fluctuations in signals. The following is a graphical representation on what is meant by an indeterminate value: