**ULTRA-LOW POWER RISC-V SPECIAlized ISA PROCESSOR**

**FOR IOT applications**

Undergraduate graduation project report submitted in partial fulfillment of the requirements for the

Degree of Bachelor of Science of Engineering

in

The Department of Electronic & Telecommunication Engineering

University of Moratuwa.

Supervisor: Group Members:

Dr. Ajith Pasqual Maduwantha.A.A.S.D 120377E

Rathnayake R.L.W 120537V

Rathnayaka I.M 120532C

Jayawickrama A.C 120275N

April 2016

Approval

Approval of the Department of Electronic & Telecommunication Engineering

…….…………………………………………….

Head, Department of Electronic & Telecommunication Engineering

This is to certify that I/we have read this project and that in my/our opinion it is fully adequate, in scope and quality, as an Undergraduate Graduation Project.

Supervisor: Dr. Ajith Pasqual

Signature: …………………………………

Date: ……………………………………….

Declaration

This declaration is made on January 5, 2017.

**Declaration by Project Group**

We declare that the dissertation entitled *Project Name* and the work presented in it are our own. We confirm that:

* this work was done wholly or mainly in candidature for a B.Sc. Engineering degree at this university,
* where any part of this dissertation has previously been submitted for a degree or any other qualification at this university or any other institute, has been clearly stated,
* where we have consulted the published work of others, is always clearly attributed,
* where we have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely our own work,
* we have acknowledged all main sources of help,
* parts of this dissertation have been published. (see List of Publications).

……………………….. ………………………………………

Date Maduwantha A.A.S.D 120377E

………………………………………

Rathnayake R.L.W 120537V

………………………………………

Rathnayaka I.M 120532C

………………………………………

Jayawickrama A.C 120275N

**Declaration by Supervisor**

I/We have supervised and accepted this dissertation for the submission of the degree.

……………………………………….. ………………………………

Dr. Ajith Pasqual Date

Abstract

Ultra-Low Power RISC-V Specialized ISA Processor for IOT Applications

Group Members: Maduwantha A.A.S.D, Rathnayaka I.M, Rathnayake R.L.W, Jayawickrama A.C

Supervisor: Prof. Dr. Ajith Pasquel

This report describes the development of our processor titled Ultra-Low Power RISC-V base ISA processor for IOT in satisfaction of our Final Year Project at the Department of Electronic and Telecommunication Engineering, University of Moratuwa. The difference between the processor we are designing and a standard PC or laptop processor is, that they vary highly in their capacities. Our processor does not include the capacity to run an operating system nor does highly complicated tasks. This is developed solely to deliver a limited range of functions, but to do them while consuming a low power and a lower surface area.

Initial digging into the present-day architecture brought up two options. We can continue our work on Intelligent Transportation Systems (ITS) or Health-care. We decided that our focus should be Health Care facilities out of the two options. This way our product will help design equipment that will be worn or used by people with medical conditions and ease their lives. We aim to design and develop a low power low area processor which can deliver almost the same performance as the existing processors. Nevertheless, the design will perform the original IOT functions.

Most of the processors used in embedded applications are not designed targeting deeply embedded systems. The difference between those and our processor is that this processor is targeting deeply embedded systems. In the content that follows, you will explore the variety of design alternatives available and now a step by step development of a low power low area processor is done based on the RISC-V ISA.

To our teachers, family and friendsAcknowledgments

No achievement noteworthy is an individual effort. Many people have guided us along the way in making this project idea a reality.

Firstly we would like express our heartfelt gratitude to our Project Supervisor and Final Year Project coordinator Dr A. Pasqual, for the enormous support extended to us. It was a great pleasure to work along with someone of your calibre and great expertise.

We also appreciate the guidance and expertise provided by Prof. Rohan Munasinghe, Head of Department and providing us with permission to access the servers and other valuable equipment that were used in the design stages. We must also express our gratitude for the knowledge and support extended to us by Dr, Jayathu Samarawickrama for providing us with the basic knowledge required for the project during the past semesters. The idea and support given to us during the evaluation sessions is also appreciated.

Our sincere gratitude goes to our project sponsors ParaQum Technologies (Pvt) Ltd, specially Engineers Mr Kalana De Silva, Mr Dinushan Vishwajith, Mr Namitha Liyanage for providing us with guidance and helping out in difficult circumstances we faced in the project.

We would also like to thank all the academic staff members, non-academic staff members of the Department of Electronic and Telecommunication Engineering University of Moratuwa.

Contents

[Introduction 1](#_Toc480182310)

[Objectives 2](#_Toc480182311)

[Review of Literature 3](#_Toc480182312)

[PULPino 32-BIT RISC-V Processor 3](#_Toc480182313)

[Sodor Processors 4](#_Toc480182314)

[Method of Investigation 5](#_Toc480182315)

[Principal results of the investigation 5](#_Toc480182316)

[Summary 6](#_Toc480182317)

[Chapter 2 7](#_Toc480182318)

[Alternative methods 7](#_Toc480182319)

[Verilog (HDL) 7](#_Toc480182320)

[Chisel 8](#_Toc480182321)

[Alternative methods in designing 9](#_Toc480182322)

[Cache 9](#_Toc480182323)

[SRAM 9](#_Toc480182324)

[Pipeline Stages 9](#_Toc480182325)

[Summary 10](#_Toc480182326)

[Chapter 3 11](#_Toc480182327)

[Methodology 11](#_Toc480182328)

[Core 12](#_Toc480182329)

[Fetch stage 13](#_Toc480182330)

[Execute stage 13](#_Toc480182331)

[Write-Back stage 14](#_Toc480182332)

[Decoder 14](#_Toc480182333)

[Arithmetic Logic Unit (ALU) 15](#_Toc480182334)

[Reg-File (Register Stack) 16](#_Toc480182335)

[Branch Code Generator 16](#_Toc480182336)

[Program Counter (PC) 16](#_Toc480182337)

[Instruction Register (IR) 16](#_Toc480182338)

[Memory 17](#_Toc480182339)

[Chisel Tool Chain 17](#_Toc480182340)

[Improving Power Efficiency through Design 18](#_Toc480182341)

[Clock Gating 19](#_Toc480182342)

[Memory Subsystems 20](#_Toc480182343)

[Summary 21](#_Toc480182344)

[Chapter 4 22](#_Toc480182345)

[ISA 22](#_Toc480182346)

[Risc-v Compiler 25](#_Toc480182347)

[Determining specifications of the design 25](#_Toc480182348)

[Top Level Design 25](#_Toc480182349)

[Detailed Design of the Processor Core 26](#_Toc480182350)

[Coding 26](#_Toc480182351)

[IMPLEMENTATION 27](#_Toc480182352)

[Results Obtained 29](#_Toc480182353)

[Summary 36](#_Toc480182354)

[Discussion 37](#_Toc480182355)

[Conclusion 39](#_Toc480182356)

List of Figures

[Figure 1: Block diagram of the processor 10](#_Toc480134888)

[Figure 2: Simple Flow chart of the Processor core 11](#_Toc480134889)

[Figure 3: Three stage pipeline design 12](#_Toc480134890)

[Figure 4: Decoding Operation 14](#_Toc480134891)

[Figure 5: Addition Simulation Waveform 27](#_Toc480134892)

[Figure 6: If Condition Simulation Waveform 28](#_Toc480134893)

[Figure 7: Max Value Simulation Waveform 31](#_Toc480134894)

[Figure 8: Sample Healthare Wearable code Testbench Wavaform 33](#_Toc480134895)

List of Tables

[Table 1: Frequency Comparison between existing implementations 4](#_Toc480134896)

[Table 2: Area and power Comparison between existing implementations 4](#_Toc480134897)

[Table 3: ISA of the processor 22](#_Toc480134898)

[Table 4: DC Compiler Results 26](#_Toc480134899)

Chapter 1

This chapter includes the introduction to the project, objectives of the project and the initial preparations done in order to take up the project.

## Introduction

RISC-V is a new instruction set architecture (ISA) that has been originally designed to support computer architecture research and education and is now set to become a standard open architecture for industry implementations under the governance of the RISC-V Foundation, University of California, Berkeley.

The aim of the project is to design and develop an ultra-low power RISC-V ISA processor with suitable extensions to specifically designed for IOT applications. The difference between the processor we are designing and a standard PC or laptop processor is, that they vary highly in their capacities. Our processor does not include the capacity to run an operating system nor does highly complicated tasks. This is developed solely to deliver a limited range of functions, but to do them while consuming low power and a lower surface area.

Most of the processors used in embedded applications are not designed targeting deeply embedded systems. The difference between those and our processor is that this processor is targeting deeply embedded systems.

Most of the existing products are bounded by patents and other restrictions. But since RISC-V architecture is an open source our processor will beat the cost margin by a huge amount.

Initial digging in to the present day architecture brought up two options.

We can continue our work on Intelligent Transportation Systems(ITS) or Health-care wearable device.

We decided that our focus should be Health Care facilities out of the two options. This way our product will help design equipment that will be worn or used by people with medical conditions and ease their lives. We aim to design and develop a low power low area processor which can deliver almost the same performance as the existing processors. Nevertheless, the design will perform the original IOT functions.

### 

### Objectives

The project we are trying to undertake has a vast scope and a lot of people that could benefit by it. Therefore, it was important that we clearly identified and noted down what our goals are. We discovered the ultimate goal to be as follows.

* + Develop an ASIC design with low power and low area requirements
  + Design is intended to be used in IOT (Healthcare Wearable) applications
  + Implement a prototype design in an FPGA as a proof of the design

Chisel language is used in the design of the architecture. After running it through the chisel toolchain we can generate the Verilog code of the design. We implemented the prototype on ZYBO (ZYNQ – 7010) board and checked the power and area requirements of the final design using the QOR report generated from RTL Synthesis tool.

Figure 1: Healthcare Wearable

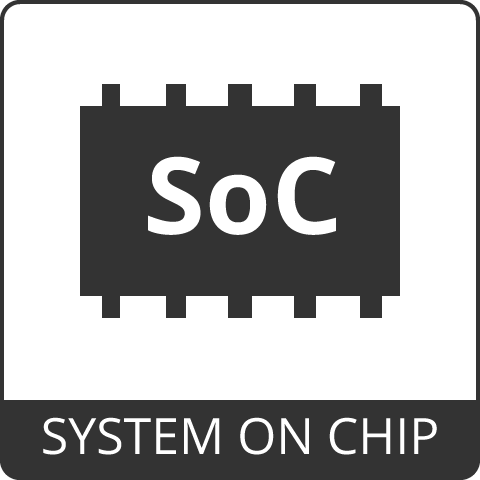
**Scope**

Figure 2: System on Chip

As explained earlier, the potential users are huge and the project work could also spread in a wide range of fields there by catering to various aspects of wearables and related industries. Nevertheless, it is important to identify what specific area is being addressed by our work. We agreed to limit the scope to the following areas given the time and resource constraints we have for the project.

* Optimizing RISC V open source ISA to support IOT devices
* Optimizing area and power requirements

The development process could be broken down into the following sub sections.

* + Use Verilog programming for the high-level design
  + Implement the RTL design
  + Derive the ASIC design using the tool chain

## Review of Literature

As of now, only a very few IOT processors are available that are implemented using RISC-V ISA. However, there are two designs, both done by well-known universities, that have significant similarities to what we are intending to do. Those designs are,

* PULPino 32-BIT RISC-V processor by **ETH Zurich** and **University of Bologna**
* **Sodor processor implementations by University of Berkeley**

Furthermore, both of these designs are open source.

### Image result for pulpinoPULPino 32-BIT RISC-V Processor

This design was done by the Parallel Ultra Low Power (PULP) project team at **ETH Zurich** Integrated Systems Lab (IIS) and **University of Bologna** Micrel Lab using System Verilog. They have previously implemented four PULP processor designs using RISC-V. This design uses several intellectual properties from those designs to implement a simple, single core, low power processor. To achieve this, they have made several changes to the previous implementations. A brief overview of the PULPino design is given below.

* Microcontroller-style platform
* Focused on core
* No caches, no memory hierarchy, no DMA

Cache memory and complex memory hierarchies have been removed to reduce area and power requirements since an IOT processor does not necessarily need them. Furthermore, the clock gating is used to put the processor core on a “sleep” mode while an event unit is responsible for “waking up” the core when needed. This is also done to improve power budget of the design.

### Image result for sodor processorSodor Processors

Figure 3: Sodor Processor

This family of educational processors are being designed by Christopher Celio who is a PhD student at UC Berkeley. Constructing Hardware in Scala Embedded Language (Chisel) is used in the designs which also comes from UC Berkeley. Unlike PULPino these designs are intended to be used as tools in education instead of being used in commercial applications.

Hence the designs are not complex and sophisticated to the level of PULPino. Several versions of Sodor designs have been released until now. The following section gives an overview of releases up to now.

* stage -1 (essentially an ISA simulator)
* stage-2 (demonstrates pipelining in Chisel)
* stage-3 (uses sequential memory)
* stage-4 (can toggle between fully bypassed or fully interlocked)
* "bus"-based micro-coded implementation

These designs prioritize improving performance of the processor instead of improving area and power requirements. Adapting this for the IOT processor will require significant modifications including revising the architecture.

We have done a comparison of the frequency, power and are requirements of the existing IoT specialized processors to get an idea about the target frequency, area and power requirements of our processor.

Table : Frequency Comparison between existing implementations

|  |  |  |
| --- | --- | --- |
| Product | Processor Core | Frequency |
| Toshiba TZ1000 | ARM®Cortex®-M4F | 48MHz |
| Leopard Gecko MCU | ARM®Cortex®-M3 | 50MHz |
| Research level fitness belt | Synopsys ARC-EM4 | With floating point extension– 10MHz  Without floating point extension– ~100Mhz |
| ATMEL ATSAMG51 | ARM® Cortex®-M4 | 48MHz |

Table : Area and power Comparison between existing implementations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Synopsys Arc 601 core | ARM Cortex M4 | PULPino | RISC-V Rocket |
| Technology | 130nm | 65nm | 65nm | 45nm |
| Gate Count | 12K | 65K | 41.5K | - |
| Power Consumption (μW/MHz) | 8.0 | 23.2 | 30.21 | 34 |
| Silicon Area (mm^2) | 0.01 | 0.062 | 0.06 | 0.14 |

### Method of Investigation

Since the project is focused on achieving power and area requirements better than existing implementations, a detailed analysis about existing processor designs is needed to be done to identify the ways to improve area and power requirements of the processor.

### Principal results of the investigation

After considering results of the investigation we realized that PULPino has huge differences with what we are targeting to do, because PULPino is designed to run an operating system. Since they’re targeting low power and low area they have removed the cache memory and use a single core design. Sodor processor is mainly designed for educational purposes. They have not considered the power consumption of the processor. They have designed different processors with different number of pipeline stages. After studying those design, we were able to decide the number of pipeline stages that we are going to include in our design.

We have decided to go for a single core, three stage pipeline design with clock frequency of 50MHz and the power rating of 13µW/MHz.

An overview of the chapters to follow is given below

Chapter 2 provides a detailed description on the design and architecture of the RISC-V processor which is implementing.

Chapter 3 provides how we apply the selected methodologies in our design and what are the results we have obtained so far.

Chapter 4 presents the discussion and conclusion. the chapter discusses the practical problems encountered while designing and testing, the significance of the results obtained and the future work to be carried out for the final implementation.

### Summary

This chapter discussed the methods of investigation done for starting the project and evaluates on the predefined stages of the project. It brings forth and overview of the task assigned to us with related literature overviews.

# Chapter 2

This chapter discusses about the alternative methods that could be followed in designing the processor which is the final aim of this project. Alternative methods of design are considered.

## Alternative methods

There are two alternative approaches to the designing Ultra-Low Power RISC-V Base ISA Processor for IOT apart from using chisel programming, which we intend to use. Those are,

* Alternative methods in implementing languages
* Alternative methods in designing

### Verilog (HDL)

Verilog is a hardware description language, which describe the circuits in textural format and which ease to describe circuits. This hardware description language is intended to use for verification in functional, timing for test analysis for logic synthesis. In addition, this is an IEEE standard language of Hardware description languages, therefore using this language as our implementing language will be better than other languages.

Using Verilog than other languages in hardware description criteria is better because this is the most familiar with gate level in circuits. It is very easy to use for functional verification, timing verification because after going through gate level synthesis. Most tools for functional verification and timing verification has been design-gate-level synthesis from commonly Verilog. Therefor if we use Verilog to implement our design will be very easy to continue our project in development phases and in future developments and modifications in other groups intended to use our design.

In addition, we need to consider complexity of the implementing language when using it to a more complex design. Because when we are in development phase we will be able to do our changes in the design very easily and simply. Using directly Verilog into our project will complex our design accordingly. Therefore, we were searching a language which ease implementing our design also which can be translated into Verilog very easily. Using that language, we should be able to implement and develop our design without touching complexity.

### Chisel

Chisel is an open-source hardware constructing language developed at UC Berkeley. This used to design advanced and more complex designs which is specific to RISC architecture. Also chisel supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. Using chisel, we can simplex our design with using Verilog for implementing too.

There are lots of important benefits of using chisel in our project. Chisel uses Scala language through coding therefore we can use object oriented concepts through our project which will ease implementation our design. In addition, it supports multiple clock domains, included with sizeable standard library including floating point units, and Scala supports layering of domain specific designs, algebraic construction and wiring, also highly parameterizable using metaprogramming with Scala. The important thing in chisel is best community support which is connected through google forums, because using like this tool as our project without good familiarizing with chisel tool we must need a community support as chisel is open-source language.

As it is explained earlier when using chisel language for this implementation we can reduce complexity of the design, also in chisel we can check functionality using Scala C++ model. Using object oriented programming methods in the implementation reduces the amount of coding required in high level coding and we can easily convert these codes in to Verilog which we are intended to use in our project. While checking functionality of Verilog behavioral coding.

## Alternative methods in designing

### Cache

Cache memory is a small-sized type of volatile computer memory that provides high-speed instruction access in our project to the processor, which will reduce instruction access time in huge amount.

Storing instruction which is relevant to program we are intended to run in the processor we can get a good through put.

There are some unexpected drawbacks in using cache memory in a system which are there will be unpredictable latency which slows the embedded system also using a cache memory the cost of manufacturing also will be very higher.

### SRAM

Using a SRAM in our project also we can decrease latency, not like in cache memory, SRAM doesn’t have any unpredictable latencies and gives the same access time always when fetching instruction through it. Therefore, using a SRAM in our project, we can get a great through put than using a cache.

### Pipeline Stages

This is the main decision in the implementation before designing the processor, because we cannot change the number of pipeline stages as we need after the design is finalized. In addition, when choosing number of pipeline stages, it depends on maximum target frequency of our design. We also referred research level processor called Sodor and it has been developed to multiple pipeline stages like one stage, two stages, three stages and five stages. We had to refer lots of present cores which were used in embedded designs. Finally, we could choose a number of pipeline stages as three which is suitable for every requirement in our design.

Increasing the pipeline stages, we can achieve high frequency in our implementation but for an embedded system high frequency is not a requirement, as our targets are and power minimization increasing frequency those needs are violated. We included decoding stage into execute stage, if it was individual stage alone in the design compared to three stage pipeline design could not achieve a big difference, therefore it is useless to increase another pipeline stage and complex the design further more.

Decreasing the number of pipeline stages will reduce maximum achievable frequency, therefore we need to decrease the pipeline stages very carefully, if it reduces the maximum achievable frequency blow than our target values will not give a better final implementation according to our design. We also had to merge execute and write back stages into one stage but we cannot predict how long it will take to complete write back stage in each instruction. Therefore, it has to hold at execute stage until the write back stage is completed, that is not efficient in an embedded design therefore we decided to implement three stages execute and write back individual two stages and instruction fetch stage.

When selecting pipeline stages, we have to consider lots of factors discussed in the earlier part. After going through our project design criteria and requirements, targets in our design finally could finalize a number of pipeline stages as three stages.

FETCH

EXECUTE

WRITE BACK

### Summary

This chapter discussed about the design aspects and the alternative design strategies that could have been undertaken and the changes that would have happened if they were done.

# Chapter 3

This chapter discusses the methodology followed in the designing the RISC-V low power low area processor. This section provides a detailed description on the design and architecture of the processor which we are implementing through this project.

## Methodology

We choose our reference architecture as RISC-V architecture and RISC-V Embedded ISA as our reference ISA with some modifications.

We had to modify RISC-V architecture and RISC-V Based Embedded ISA according to our design. RISC-V Embedded ISA have 37 instructions and we get rid of seven instructions which are six CSR instructions and FENC.I instruction which are not needed in our design. We also do not include instructions from C and M standard instructions.

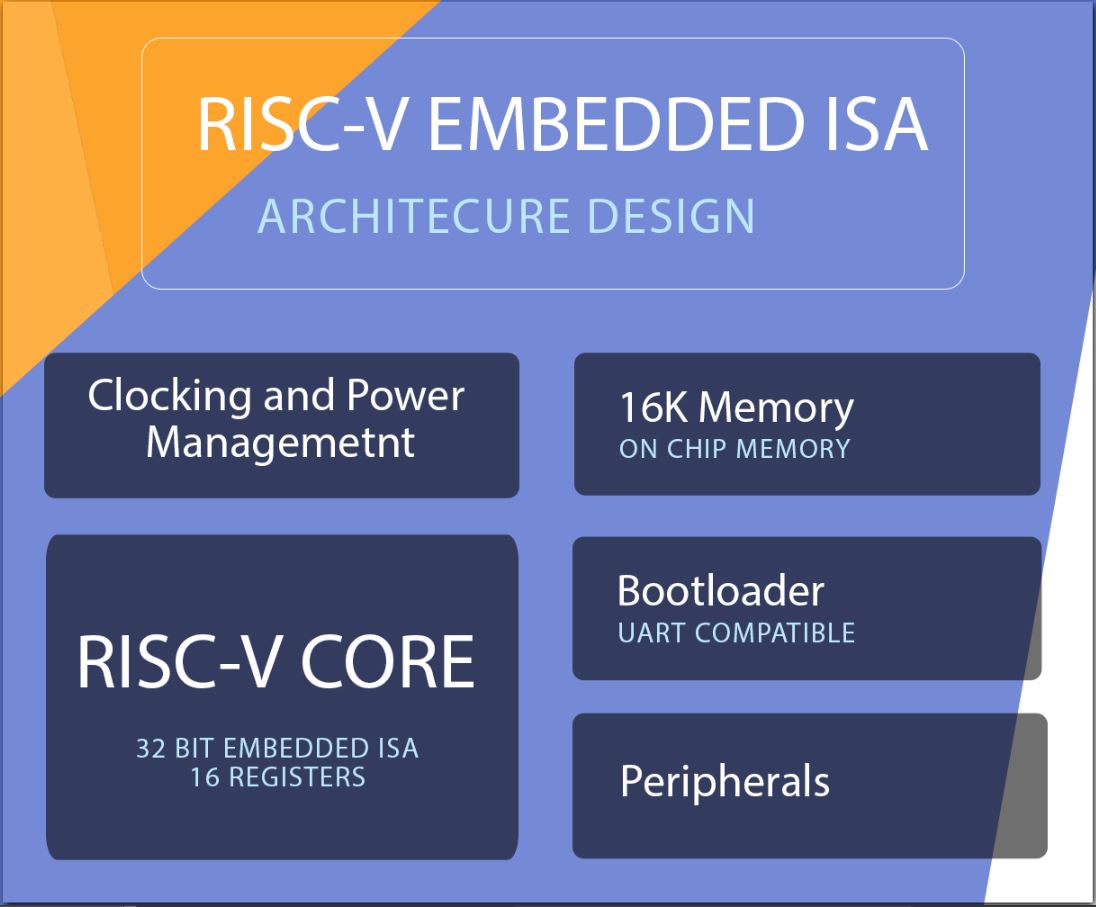


Figure : Block diagram of the processor

### Core

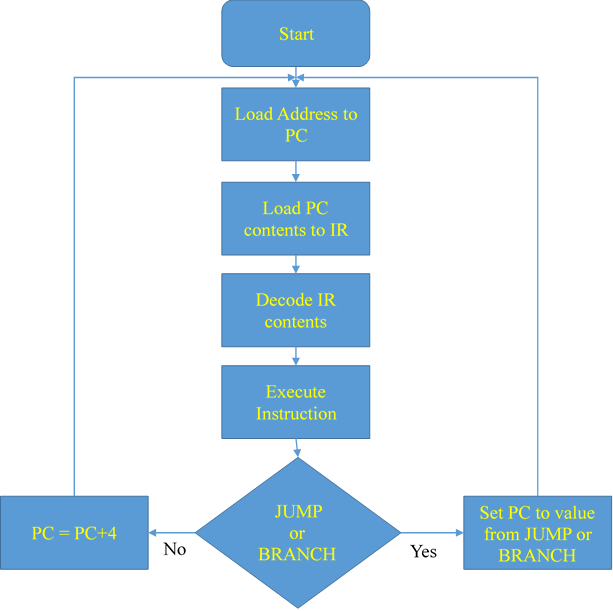


Figure : Simple Flow chart of the Processor core

This is the main processing unit of our design. In the core, we have implemented a three stage pipeline design

* Fetch Stage
* Execute Stage
* Write Back Stage

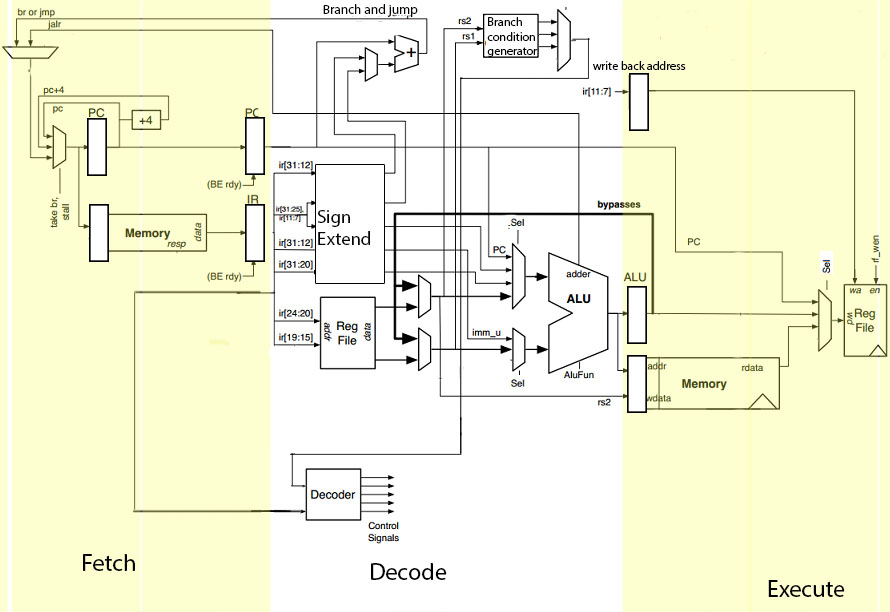


Figure : Three stage pipeline design

### Fetch stage

In this stage, the next instruction is fetched from the memory address that is currently stored in the program counter (PC), and stored in the instruction register (IR). At the end of the fetch operation, the PC points to the next instruction or points to an appropriate address if the current instruction is JUMP or BRANCH, that will be read at the next cycle.

### Execute stage

In this stage, the main processing part is done such as ALU operations, Instruction decoding, Register Stack Read, Branch condition generation and also J Type, U Type, S Type, B Type, I Type extensions are all done. Also, ALU inputs are according to instruction decoded by the decoder.

### Write-Back stage

In this stage Memory read and write, Registry stack write is done according to current instruction. Here we used a Dynamic random access memory (DRAM) which takes dynamic time intervals to read and write operations therefore we had to design a method to hold the other two stages until the Memory operations are done, otherwise two memory write or read operations can be overlapped and give garbage values in future instructions.

There are main modules in the core,

* Decoder
* Arithmetic and Logic Unit (ALU)
* Reg File (Register Stack)
* Branch Code Generator
* Program Counter (PC)
* Instruction Register (IR)

### Decoder

Decoder is the main unit and the center of the control path of the core. The decoder takes the instruction from the Instruction Register, decode it using opcode and other function bits of the instruction and send control signals to ALU, Register File, Memory and all other components.

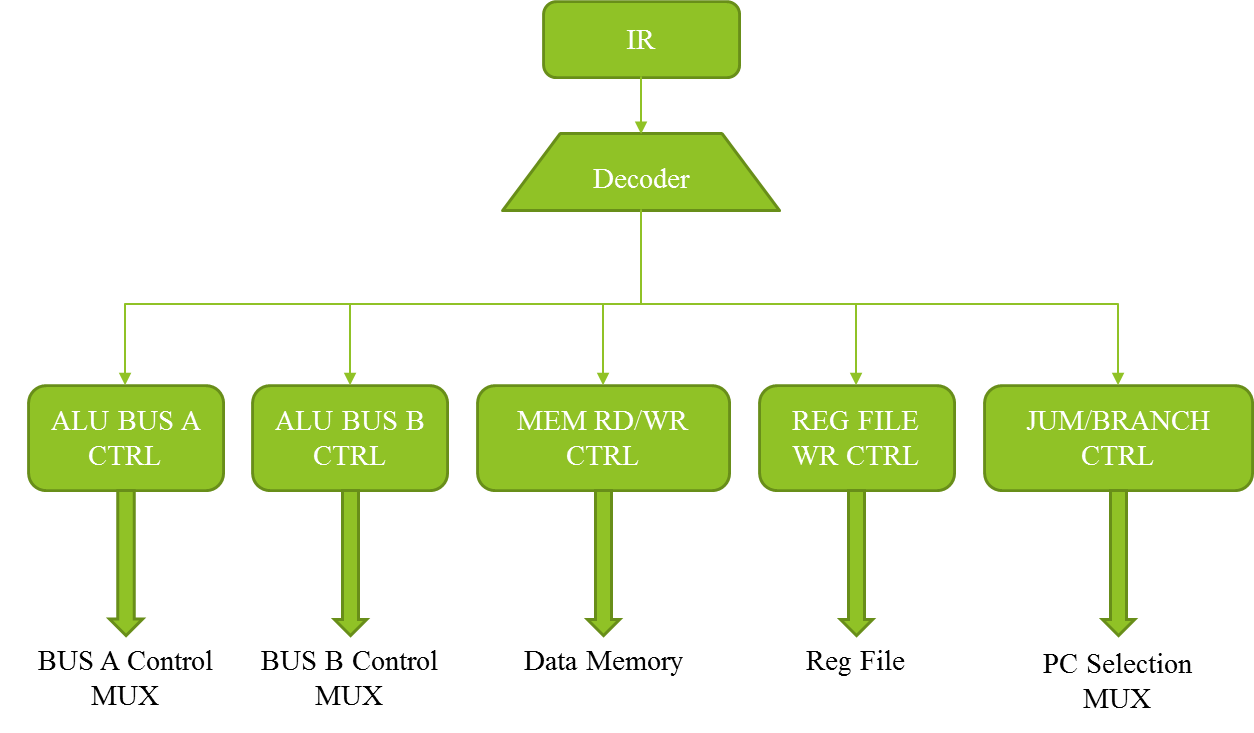


Figure : Decoding Operation

### Arithmetic Logic Unit (ALU)

This is the main module in the core after the decoder. There are ten instructions in our designed ISA as ALU instructions.

* ADD – Addition is done from sources and least significant bits are written to the destination ignoring overflows
* SUB – Subtraction is done from sources and least significant bits are written to the destination ignoring overflows
* SLL – Shifts the value in rs1 logically left by shift amount is held on rs2 lower five bits
* SLT – Performs signed less than operation with rs1 and rs2, write 1 to rd if rs1 < rs2 otherwise 0
* SLTU – Performs unsigned less than operation with rs1 and rs2, write 1 to rd if rs1 < rs2 otherwise 0
* XOR – Performs XOR operation between rs1 and rs2 registers logically
* SRL – Shift the value in rs1 logically right by shift amount is held on rs2 lower five bits
* SRA – Shifts the value in rs1 arithmetically right by shift amount is held on rs2 lower five bits
* OR – Performs OR operation between rs1 and rs2 registers logically
* AND – Performs AND operation between rs1 and rs2 registers logically

In the ALU operations using ten different ALU instructions all arithmetic operations and logical operations are done according to the current instruction which is executing at execute stage.

### Reg-File (Register Stack)

Reg File stores the data used in arithmetic and control operations and the outputs of those operations. Reg File read and write addresses are taken from the instruction stored in IR. It can give data stored in it to the ALU for arithmetic operations, check Branch status and to store the observed results to the data memory. To perform any operation using the data stored in the data memory user has to take those data to the register and process them afterwards. Since we follow RISC-V Embedded ISA in our design, Register Stack consists of sixteen 32 bit registers. Two registers can be read simultaneously and can write the results to only one register at a time.

### Branch Code Generator

In the Branch Code Generator, take decisions according to current instruction and rs1, rs2, there are three-bit output to indicate comparison between rs1 and rs2, if rs1 and rs2 equals set one bit high like that rs1 is less than rs2 then set another bit high and unsigned values of rs1 less than rs2 will set the other bit high.

### Program Counter (PC)

A program counter is a register in the processor that contains the address of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 4 or change the value accordingly if the instruction is JUMP or BRANCH. After each instruction is fetched, the program counter points to the next instruction in the sequence.

### Instruction Register (IR)

Instruction register (IR) is the part of the core that holds the instruction currently being executed or decoded. Each instruction to be executed is loaded into the instruction register which holds it while it is decoded, prepared and ultimately executed, which can take several steps.

### On-Chip Memory

This is the memory that is directly accessed by the RISCV processor core. To keep up with the speed of the processor core this memory needs to be able to be accessed within one clock cycle. Thus, the on-chip memory is an SRAM rather than a DRAM. In the early stages of the project we have considered several alternative approaches to this memory requirement. The main alternatives we had were implementing a single level cache memory or using a scratchpad memory.

The required processor design is intended to be used in hard real time applications. The variable memory access latency, which is inevitable in cache memory systems is thoroughly undesired in this type of designs as the memory should meet the worst-case memory access time constraint rather than the average-case memory access time constraint. This makes the cache memory system infeasible for our design.

However, scratchpad memory does not have this problem. Thus, it is desired in hard real time applications. Therefore, the on-chip memory in this design is basically a simple scratchpad memory. However, as the project progressed we have realized that a properly size determined small on-chip memory alone can accommodate all instructions and data in all our test cases. Thus, in our design we use only the scratchpad memory to store data and instructions. Furthermore, we use a boot loader that uses Universal Asynchronous Receiver/Transmitter (UART) communication to load instructions and data to the scratchpad from outside the SoC.

Following section describes the specifications of the memory. On-chip memory is completely implemented in Chisel and is fully parameterized. Changing the memory size, word size, bus widths, number of access ports etc. as well as choosing combinational read or sequential read options is parameterized to allow quickly generating different variations of the memory. Since the processor core and the memory are in the same clock domain, memory request ports use ready valid handshake while memory response ports use valid IO interfaces.

There are three versions of the design.

Version 1 – Memory supports all basic functionalities. Supports word, half word, half word unsigned, byte and byte unsigned access modes for memory load and store. Same memory block is shared between instruction and data memories while two separate access ports are used for them.

Version 2 – Separate memory blocks are used for instruction and cache memories to avoid dual port memory which is expensive.

Version 3 – Uses a memory consisting of multiple same sized memory blocks (number of blocks is also parameterized) to reduce power consumption of the memory via clock gating. Separate blocks have separate clock signals. The clock signal to a particular memory block is provided only when the address decoder decides that the memory block is accessed.

### Chisel Tool Chain

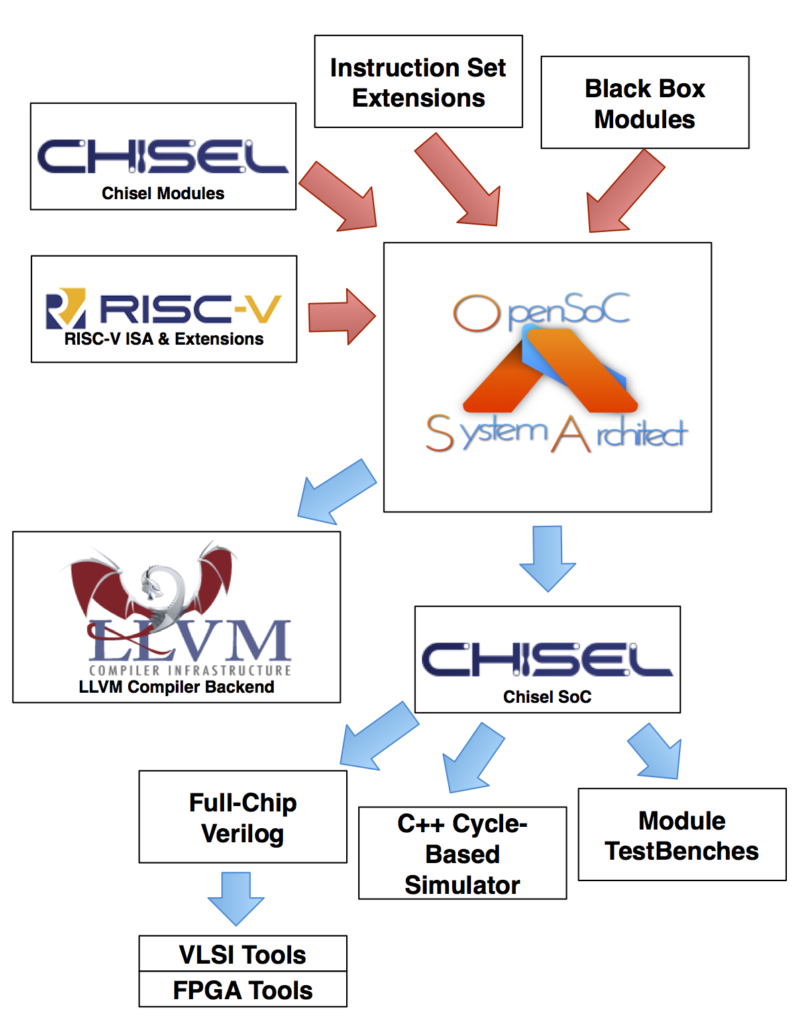


Figure : Chisel Tools

In the first phase of the project chisel has been installed to our server because we need to code our implementation in chisel and compile to Verilog. In chisel tool chain, there are three main parts,

* Chisel: A front-end circuit generator that implements an embedded Scala circuit DSL which compiles into a FIRRTL file
* FIRRTL: An extensible multi-pass compiler that lowers high level FIRRTL into lower FIRRTL or to Verilog
* FIRRTL Interpreter: A Scala level simulator that executes low FIRRTL.

Scala is the based language used in Chisel. We have to import chisel libraries to use chisel commands in our design. After writing Chisel program we had to write Scala test harness for test each generated output from the Chisel tool chain. After executing our Chisel program using Chisel tool chain we can get Verilog Behavioral model and an equivalent C++ model. Therefore, we can do hardware level behavioral verification without much effort.

In our design, we used leaf level module design using Chisel and the top-level design using Verilog for area and power optimization processes. For leaf cells, we verified the functionality by the C++ model generated by Chisel. After doing the top-level design we have verified the functionality and timing accuracy using Verilog test benches.

## Improving Power Efficiency through Design

The methodology followed to reach the aims of the project are discussed in detail below. Since our design is an improvement on the existing RISC-V architecture it was necessary that we focus on ensuring lower power and area values. The first and foremost idea is to reduce the gate count. It is well known that more significant power reductions are possible if optimizations on higher levels of abstraction are made like the architectural and algorithmic level than at the circuit or gate level. This provides the required motivation for the developers to focus on the development of new architectural level power analysis tools. DC compiler is used to obtain the gate count of the designs developed by us. There are also equations that provide relationship between the area and the design and the area of a NAND2 gate, that provides a value for the number of gate count. Die area is however easily calculated and provided to us in the synthesis report provided by DC compiler. The ASIC design is what is used as the basis for the calculation of the area.

We include an Event Unit in our design. This will ensure the sleep, wake operations of the IC are handled correctly. The functionality of this is in a way similar to interrupts in micro-controllers. However, it is more advanced than that.

### Clock Gating

Clock tree consume more than 50 % of dynamic power. The components of this power are:

1) Power consumed by combinatorial logic whose values are changing on each clock edge

2) Power consumed by flip-flops and

3) The power consumed by the clock buffer tree in the design.

It is a good design idea to turn off the clock when it is not needed. Automatic clock gating is supported by modern EDA tools. They identify the circuits where clock gating can be inserted. This is where our design falls in.

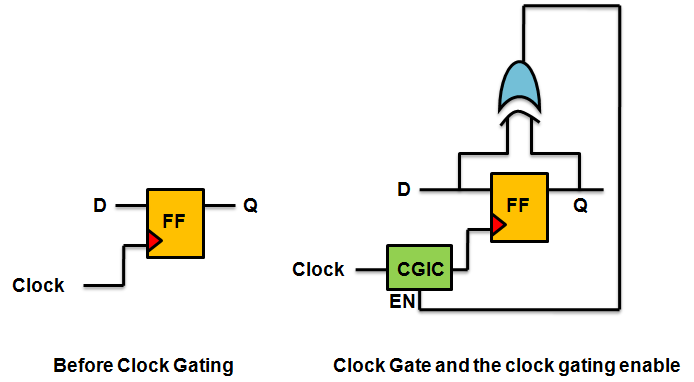


Figure : Clock Gating

RTL clock gating works by identifying groups of flip-flops which share a common enable control signal. Traditional methodologies use this enable term to control the select on a multiplexer connected to the D port of the flip-flop or to control the clock enable pin on a flip-flop with clock enable capabilities. RTL clock gating uses this enable signal to control a clock gating circuit which is connected to the clock ports of all of the flip-flops with the common enable term. Therefore, if a bank of flip-flops which share a common enable term has RTL clock gating implemented, the flip-flops will consume zero dynamic power as long as this enable signal is false.

There are two types of clock gating styles available. They are:

1) Latch-based clock gating

2) Latch-free clock gating.

The latch-free clock gating style uses a simple AND or OR gate (depending on the edge on which flip-flops are triggered). Here if enable signal goes inactive in between the clock pulse or if it multiple times then gated clock output either can terminate prematurely or generate multiple clock pulses. This restriction makes the latch-free clock gating style inappropriate for our single-clock flip-flop based design.

The latch-based clock gating style adds a level-sensitive latch to the design to hold the enable signal from the active edge of the clock until the inactive edge of the clock. Since the latch captures the state of the enable signal and holds it until the complete clock pulse has been generated, the enable signal need only be stable around the rising edge of the clock, just as in the traditional ungated design style.

The latch-based clock gating style adds a level-sensitive latch to the design to hold the enable signal from the active edge of the clock until the inactive edge of the clock. Since the latch captures the state of the enable signal and holds it until the complete clock pulse has been generated, the enable signal need only be stable around the rising edge of the clock, just as in the traditional ungated design style.

### Memory Subsystems

The other method is to partition the data memory and access them/ power them in a sectioned manner to avoid wastage of power.

Unlike regular design registers that only dissipate dynamic power when they are written, memories dissipate dynamic power when they are either read or written. As such, removing redundant reads or writes can result in significant dynamic power reduction.

Static power is the power that is consumed when the memory stays alive, and dynamic power is consumed every time we attempt to write or read from it. Out of the two, dynamic power is the bigger one. Static power is comparatively smaller. To ensure the functionality of the memory, the part of memory must only be put to sleep when that part is not being accessed. It could also be necessary that the memory exits the sleep mode some number of cycles before the operation, in order to ensure proper functionality takes place. Removing redundant memory accesses and utilizing the low power modes in embedded memories requires sequential analysis of the design across multiple pipeline stages.

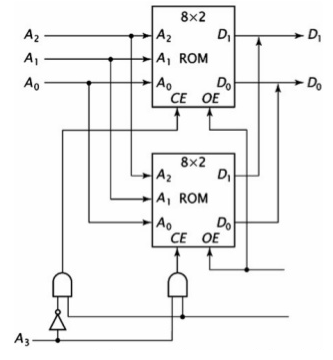


Figure : Memory Subsystems

### Summary

In this chapter, we have included the detailed design of the architecture we have implemented in our processor and the details of the design specifications of the processor. In addition, the methods we have used for power optimization is included in here too.

# Chapter 4

This chapter explains the architecture chosen for the project and the changes we have done to it in order to develop the final design.

## ISA

Result of initial stage of the project was finalizing the Instruction Set Architecture (ISA) for the processor that is being designed. Finalized ISA is based on the RISC-V Embedded ISA (RV32E). RV32E ISA contains 47 instructions and only supports M and C user-level standard extensions. Except ten of these instructions, the rest is used in the finalized ISA. None of the instructions in the supported two extensions are included.

RISCV-Compiler can be configured to compile a C code only using the Base ISA instructions. Since we have implemented all the instructions that are used in deeply embedded systems, the compiled codes should work in our processor without any problem. The instructions we omitted won’t affect the data path and control path of the core.

Table below shows the ISA of the processor, made up of 37 instructions.

Table : ISA of the processor

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| LUI | Loads immediate Value to Register |
| AUIPC | Stores PC value to Register |
| JAL | Unconditional jump |
| JALR | Jump anywhere in a 32-bit absolute address range |
| BEQ | Conditional Branch |
| BNE |
| BLT |
| BGE |
| BLTU |
| BGEU |
| LB | Load data from Memory |
| LH |
| LW |
| LBU |
| LHU |
| SB | Store Data to Memory |
| SH |
| SW |

|  |  |
| --- | --- |
| ADDI | Immediate ALU Instructions |
| SLTI |
| SLTIU |
| XORI |
| ORI |
| ANDI |
| SLLI |
| SRLI |
| SRAI |
| ADD | Register ALU Instructions |
| SUB |
| XOR |
| SRL |
| SRA |
| OR |
| AND |
| SLL |
| SLT |
| SLTU |

This ISA only supports the machine-mode (M-mode) operation. Which is not a problem since deeply embedded processors does not run any operating systems.

## Risc-v Compiler

After the ISA was finalized, we initiated the RISC-V tool chain and Newlib cross-compiler in a Linux operating system environment. Then that was used to compile sample C program codes to obtain corresponding assembly codes. By comparing those codes with RISC-V assembly programmer’s handbook in RISV-V version specification 2.1, we obtained the corresponding RISC-V instructions.

We used the results of this procedure to find out which instructions were used and how frequently those instructions were used.

## Determining specifications of the design

Since the design should target a specific area of Internet of Things (IoT), we have decided to choose the field of healthcare wearable devices. Then the specifications of a processor suitable for such application had to be decided. For this task, we considered the specifications of processors currently used in healthcare wearable devices as well as other factors like expected battery life and cost.

Processor specification of this stage simply included maximum clock frequency and power requirement. Thus, we have determined rough figures of 50MHz clock frequency and 13µW/MHz power consumption.

## Top Level Design

Then we made a top level abstract design of the processor to suffice the specifications. After some research, we decided not to include cache memory in the design since the unpredictable memory latency given by a design with cache memory is undesirable in real time systems. Next, we considered the possibility to include a scratchpad memory in place of the cache memory. Ultimately, we have rejected that approach as infeasible because the need to include custom instructions to manage scratchpad memory will require us to make our own compiler and ISA simulator. Thus, we finalized a flat memory hierarchy with on chip dual port instruction and data RAMs.

Furthermore, we have included an event unit in the design to put the processor core to sleep using clock gating to reduce dynamic power consumption when the processor is not active.

## Detailed Design of the Processor Core

Some of the details of processor core is directly determined by the ISA. We had to determine the rest before we started. Considering that the maximum clock speed of the processor is just 50MHz we decided to use a simple instruction pipeline with a small number of pipelining stages. After studying the 5-stage basic RISC-V pipeline we started the design with a 2-stage (fetch and execute) pipeline.

However soon we realized that the execution stage has a much higher setup time compared to the fetch stage. Since this will drastically limit the maximum clock frequency of the design we decided to split the execution stage into two stages and use a 3-stage pipeline (fetch, execute and write back).

Since we do not use concurrent execution we face only true dependency (read after write) as data hazards. As our pipeline has only 3 stages have decided to use operand forwarding (by register file bypasses) to solve that.

## Coding

We did the Register Transfer Level (RTL) design using Chisel language. We used Scala test harnesses to verify our designs, by testing the clock cycle accurate C++ output of the source files generated by the Chisel tool chain with Scala test benches. Since as of now there is no way to directly synthesize Chisel source code we generated Verilog output from the tool chain and use Xilinx Vivado 2016.2 for synthesis. We also used Verilog test benches to catch any problems we might miss with C++ emulations.

The top-level design was done using Verilog HDL, since it’s a familiar language for us and the power and area optimizations can be easily done in the top level using Verilog.

After designing the core module, we have designed the memory module.

…….

Then we designed the bootloader to send the compiled code to our processor. This is a hardcoded bootloader that uses UART communication to receive the code. When a certain pin configuration is setup the bootloader changed the status to instruction writing stage. And the instructions we feed through UART port will be write to the instruction memory of the processor. When we change the ststus to execute program bootloader resets the core and other modules and start executing the code written to the instruction memory.

The clocking and power management module is developed partially. The memory is divided into blocks and this unit sends signals to which block should be wake in each clock cycle. We haven’t implemented the sleep and wake functions of the processor yet.

## IMPLEMENTATION

we used a Xilinx Zybo FPGA development board to implement the design on an FPGA for final demonstration. We have used debug cores to probe several locations of the processor to demonstrate that the desired values in each clock cycle is in those locations.

We have implemented the design in FPGA and run it at a 50MHz frequency. We were able to run simple codes that are mostly used in IoT devices such as addition, multiplication, arrays, sorting, finding maximum and minimum values, etc. And also, we were able to run a sample code that was written for a healthcare device successfully.

Figure 11: Xilinx Zybo FPGA

Using the results obtained by running these sample codes we calculated that our processor should be running at 39MIPS at the frequency of 50MHz.

Furthermore, we have used an RTL synthesis tool to generate Quality of Result (QOR) report to find out area and power requirements of the design. The power requirements are not 100% correct, because we have not done the floor planning and place and route of the design.

The power and area data extracted from QOR reports generated by Synopsys DC Compiler is given below. We have run the tests at a 50MHz clock and we used Global Foundries 60nm technology.

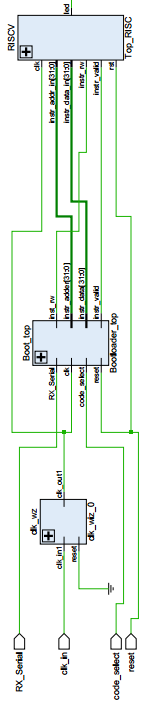


Figure : Generated System Block Design

Table : DC Compiler Results

|  |  |
| --- | --- |
| Global Operating Voltage | 1.2V |
| Hierarchical Port Count | 2963 |
| Leaf Cell Count | 2104 |
| Combinational Cell Count | 1856 |
| Sequential Cell Count | 248 |
| Total Peak Power | 34.6929mW |

## Results Obtained

Some of the sample C codes we have tested and the results we have obtained are shown below

1. Simple addition

Take two integer values and add them together

#include <stdio.h>

int main(){

int a,b,c;

a = 1;

b = 2;

c = a+b;

}

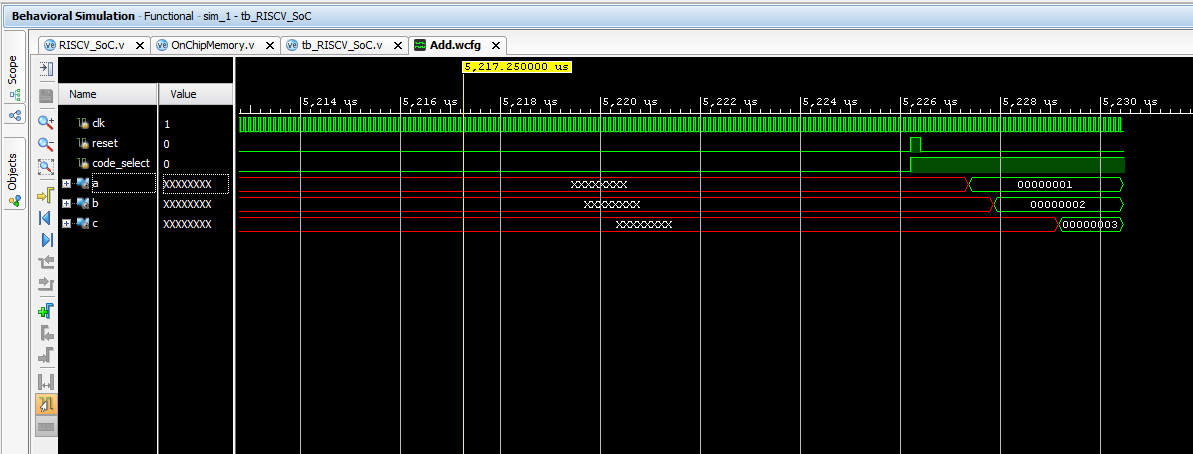


Figure : Addition Simulation Waveform

1. If Condition checker

Take two values and check whether those two are equal and perform operation

#include <stdio.h>

int main(){

int a,b,c;

a = 5;

b = 10;

if(a==b){

c = 20;

}

}



Figure : If Condition Simulation Waveform

1. Sorting

Sort ten integers stored in the data memory using selection sort algorithm.

#include <stdio.h>

void sort(int \*array);

int main()

{

int array[10];

int j = 0;

for(int i=0;i<10;i++){

array[i] = \*((int\*)0x64 + j);

j = j + 4;

}

sort(array);

j = 0;

for(int i=0;i<10;i++){

\*((int\*)0x64 + i) = array[i];

j = j + 4;

}

while(1);

}

void sort(int \*array){

int c = 0;

int d = 0;

int position = 0;

int swap = 0;

for ( c = 0 ; c < 9 ; c++ )

{

position = c;

for ( d = c + 1 ; d < 10 ; d++ )

{

if ( array[position] > array[d] )

position = d;

}

if ( position != c )

{

swap = array[c];

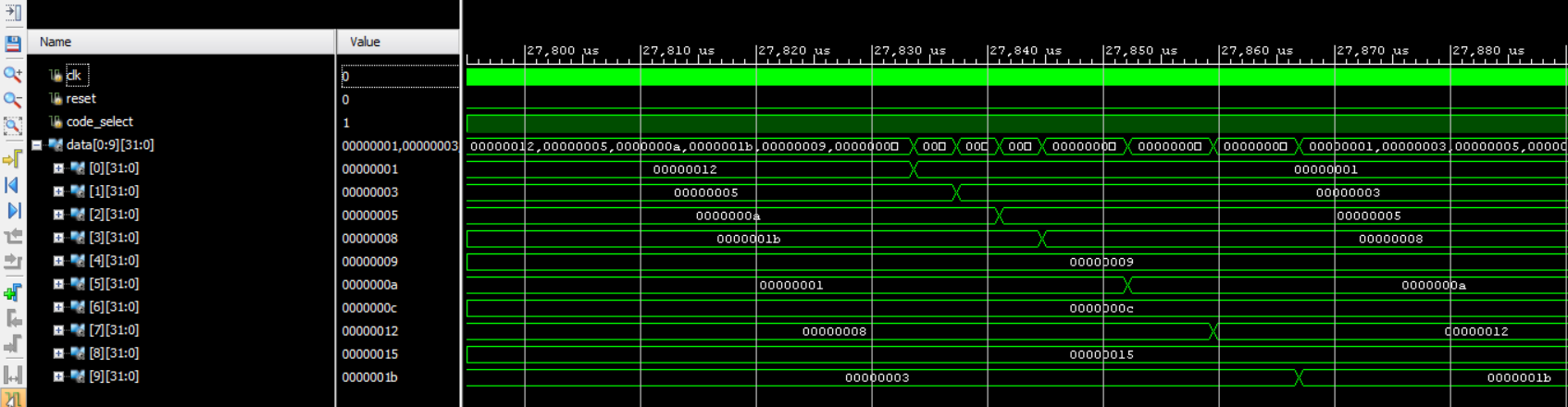
array[c] = array[position];

array[position] = swap;

}

}

}



1. Max

Find the maximum value from a ten-integer array

#include<stdio.h>

int main(){

int array[10] = {1,2,6,5,2,3,9,0,8,7};

int i = 0;

int max = array[0];

for(i = 0; i< 10; i++){

if(max < array[i]){

max = array[i];

}

}

}

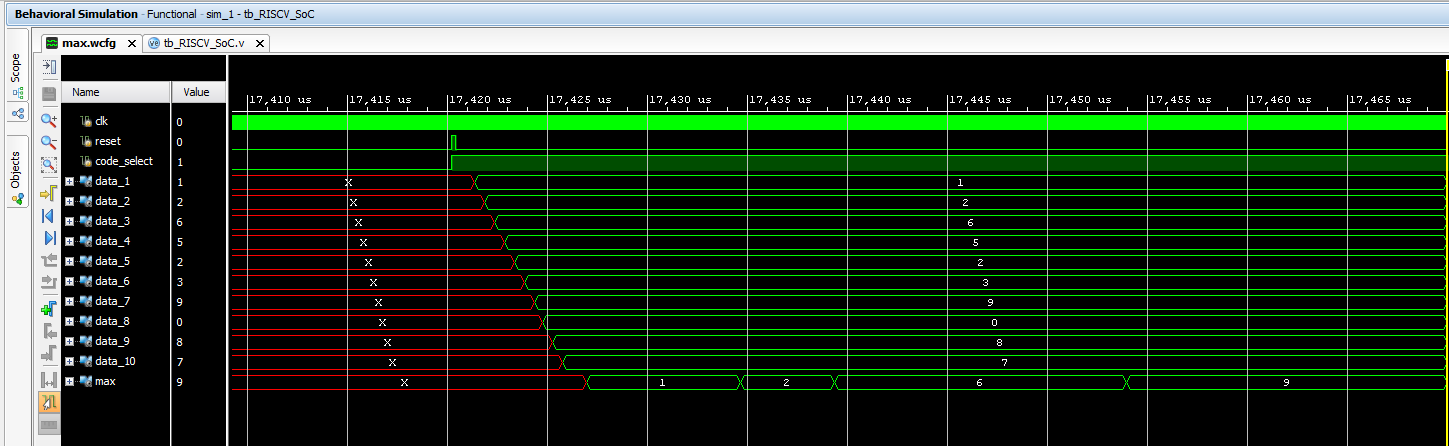


Figure : Max Value Simulation Waveform

1. Healthcare Wearable Code

Simple example code used in healthcare wearables

#include<stdio.h>

int main(){

int heartRate[10] = {100,125,136,145,120,85,92,103,85,82};

int Temp[10] = {30,32,29,36,38,40,39,41,28,30};

int peakHeartRate = 495;

int cardioHeartRate = 407;

int fatburnHeartRate = 291;

int hypothermiaTemp = 105;

int normalTemp = 111;

int feverTemp = 114;

int HyperpyrexiaTemp = 123;

int T\_HeartRate = 0;

int T\_Temp = 0;

int i = 0;

int j = 0;

int status = 0; //0 - Normal, 1 - Unhealthy, 2 - Excercising, 3 – Sleeping

while(1){

for(i = 0; i < 8; i++){

T\_HeartRate = heartRate[i] + heartRate[i+1] + heartRate[i+2];

T\_Temp = Temp[i] + Temp[i+1] + Temp[i+2];

if((T\_HeartRate > cardioHeartRate) && (T\_Temp > normalTemp)){

status = 2;

}else if((T\_HeartRate > fatburnHeartRate) && (T\_Temp > normalTemp)){

status = 2;

}else if((T\_HeartRate > fatburnHeartRate) && (T\_Temp < normalTemp)){

}else if((T\_HeartRate > fatburnHeartRate) && (T\_Temp < normalTemp)){

status = 0;

}else if(T\_Temp > HyperpyrexiaTemp){

status = 1;

}else if((T\_HeartRate > peakHeartRate) && (T\_Temp < hypothermiaTemp)){

status = 1;

}else if((T\_HeartRate < fatburnHeartRate) && (T\_Temp < hypothermiaTemp)){

status = 3;

}else{

status = 0;

}

for(j = 0; j < 500000; j++);

}

}

}

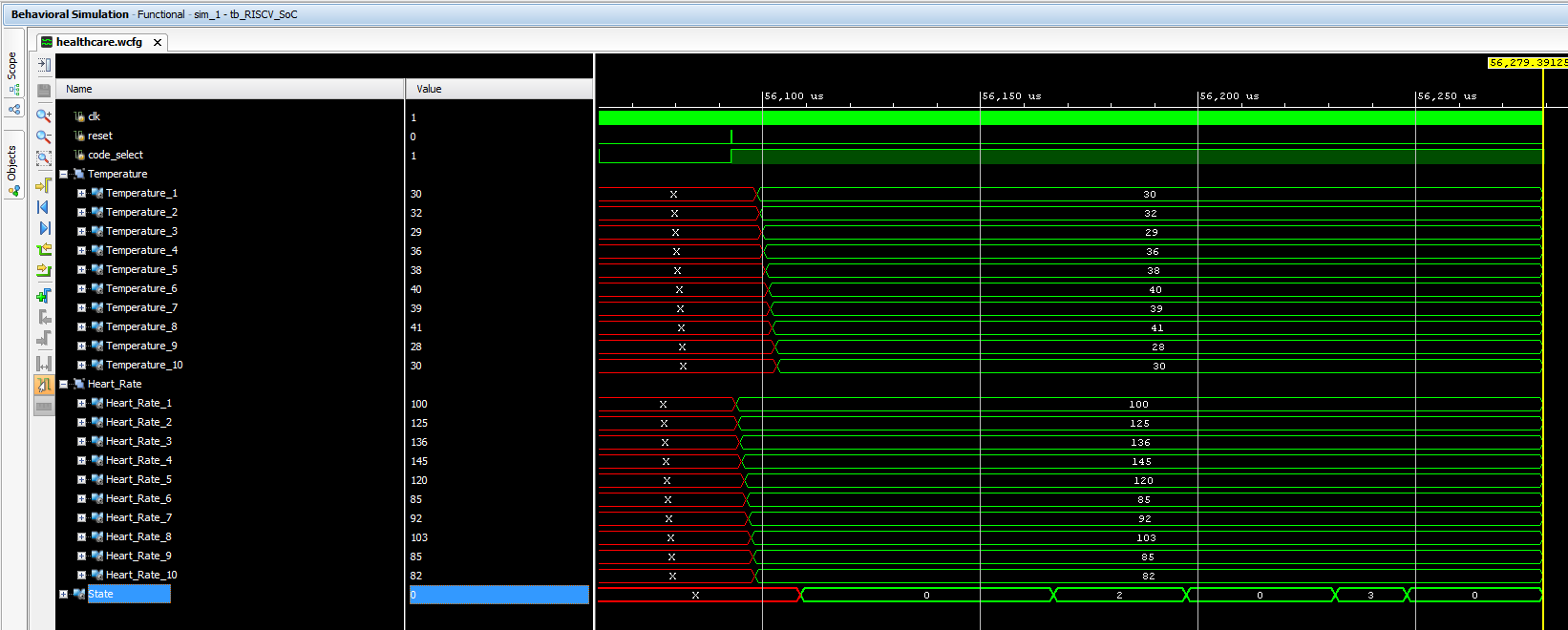


Figure : Sample Healthare Wearable code Testbench Wavaform

### Summary

In this chapter, we have discussed what we have done throughout the time of the project, what we have achieved while deigning the processor and what we have implemented in order to achieve our goals.

Chapter 5

This chapter presents the overall discussion of the project. Included in this a general look back at the work already done and what is yet to be completed.

## Discussion

We started off by observing the RISC-V Embedded ISA (RV32E) which contains 47 instructions. We ended up using all of those instructions in our finalized ISA but 10. Our instruction set consists of 37 instructions. The design ISA will only support machine-mode operation. This will not be problematic since deeply embedded processors do not run any operating systems.

Having finalized the ISA, we initiated the RISC-C tool chain and Newlib cross compiler in a Linux Operating System environment. Newlib cross-compiler was then used to compile sample C program codes to obtain corresponding assembly codes.

In order to obtain the RISC-V instructions required, RISC-V assembly the programmer’s handbook version specification 2.1 was used as a reference. Then it was crucial that out design met the area and power constraints better than the level of existing smart wearable designs in the health care sector. Since we had already selected that the design should cater to users in the specific area of Internet of Things (IoT) only, it was decided to choose the field of healthcare wearable devices.

We had to go through a phase of extensive research in order to determine what are the existing implementations and their power consumption figure and area of processors. The wearable industry, which includes medical devices, fitness and health gadgets, and infotainment devices like [virtual reality gamer goggles](http://www.informationweek.com/mobile/virtual-reality-still-goofy-after-all-these-years/d/d-id/1319350) and smartwatches primarily used for communication, is expected to grow to 170 million unit shipments in the year of 2016, driving $20 billion in global manufacturer sales.

Samsung, Huawei and HTC are among the pioneers among manufacturers. The all-time favourite movie series Star Trek had a medical tricorder, to monitor and check any symptom of the person wearing it anytime of the day. This is what inspired the scientific community to look for real-world applications of a concept like it.

Wikipedia defines a medical device as “Any instrument, apparatus, appliance, software, material, or other article—whether used alone or in combination, including the software intended by its manufacturer to be used specifically for diagnostic and/or therapeutic purposes and necessary for its proper application—intended by the manufacturer to be used for human beings for the purpose of the following activities”

* Diagnosis, prevention, monitoring, treatment, or alleviation of disease;
* Diagnosis, monitoring, treatment, alleviation, or compensation for an injury or handicap;
* Investigation, replacement, or modification of the anatomy or of a physiological process;
* Control of conception; and which does not achieve its principal intended action in or on the human body by pharmacological, immunological, or metabolic means, but which may be assisted in its function by such means

The existing designs of such health care devices were analysed, taking in to consideration the processing power, battery consumption, cost of manufacturing were analysed.

The results lead to determining which instructions should be used and how frequently they will be used.

Our research and discussion done with experts in the field lead to us deciding what are the most commonly used Internet of Things (IoT) Health Care Devices, and then we tabulated the design aspects of them. We wrote down the area and power consumption figures of those devices, their clock frequencies etc. We decided by thorough analysis that our design must deliver a minimum of at least 50MHz clock frequency and 13µW/MHz power consumption.

The power ratings and clock frequency anticipated for the design will enable it to be used in most common Internet of Things Health Care devices in the market at a low power than they are consuming now.

## Conclusion

The current progress of the work stands proof of the fact that the design is feasible and can be finalized within a short period of time. The evaluations of the project suggested that we should follow the newsletters of the RISC mailing lists and we found colleagues who are very keen on working with designs and solving problems related to RISC architecture.

We had immense help from them and the discussions in the forum broadened our horizons of the knowledge we have in processor design. The design is finalized and successfully implemented a working prototype on an FPGA. No scratch pad or caches will be used since they will require more power and area.

Memory will be accessed via clock gating making it possible to save power. Pipelining will be used. 3 stages of pipelining were chosen as ideal because it provided us with the required performance without draining the battery power or showing excessive amounts of power consumption. The existing design with pipelining were also taken in to consideration in doing this.

Register Transfer Level (RTL) designs of leaf level modules were done using Chisel language. Verilog was used to develop test benches to catch any problems we might miss with C++ emulations. Top-level design was done using Verilog. Timing and functional verification was done using Vivado test benches. Xilinx Zybo FPGA was used for the final demonstration to demonstrate the working prototype of the processor. Since we were able to run the design in 50MHz clock in an FPGA, we can confirm that the ASIC design will be run at 50MHz clock without any issues.

The work has not brought about any issues we could not address. Simple problems we faced during coding were solved by working together and we have implemented a functional design. We plan to improve the design by finishing the partially implemented modules like power management module and the peripherals.

Bibliography

Biegel, F. E. a. M., 2000. *Power Reduction Through RTL Clock Gating.* SNUG: San Jose.

Gharachorloo, S. V. A. a. K., 1996. *Shared Memory Consistency Models: A.* s.l.:IEEE Computer 29.12 (Dec. 1996).

Hill, S. V. A. a. M. D., 1990. *Weak Ordering&Mdash;a New Definition.* s.l., s.n., p. 2–14.

IEEE, 1985. IEEE standard for floating-point arithmetic.

IEEE, 2008. IEEE standard for floating-point arithmetic.

Ltd, A., 2008. *ARMv6-M Architecture Reference Manual.* s.l.:s.n.

Ltd, A., 2014. *ARM Architecture Reference Manual.* ARMv7-A and ARMv7-R Edition ed. s.l.:s.n.

Maynard, M., 1996. *lpha Architecture Handbook.* s.l.:Digital Equipment Corporation.

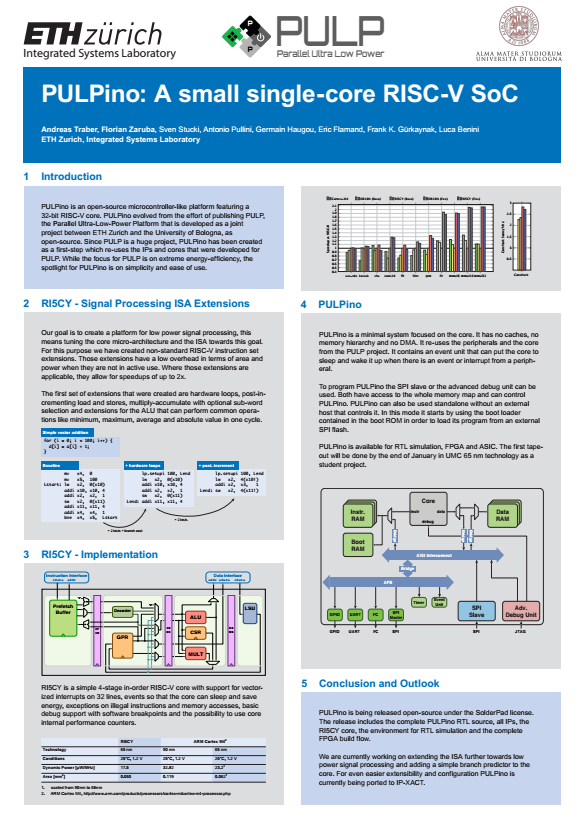
Patterson, J. L. H. a. D. A., 2002. *Computer Architecture: A Quantitative Approach.* 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

S´equin., D. A. P. a. C. H., 1981. *RISC I: A Reduced Instruction Set VLSI Computer.* Minneapolis, Minnesota, USA: ISCA.

Waterman, A., 2011. *Improving energy efficiency and reducing code size with RISC-V compressed.* University of California, Berkeley: s.n.

Appendix

* PULPino



* Clock Gating

