



# UCL

---

## Design of novel processor architectures with photonic interconnect networks

---

AUTHOR: SREETHYAN ARAVINTHAN

UNIVERSITY COLLEGE LONDON

SUPERVISOR: PROFESSOR ROBERT KILLEY

SECOND ASSESSOR: DR ZHIXIN LIU

April 21, 2020

A BEng Project Final Report

# DECLARATION

I have read and understood the College and Department's statements and guidelines concerning plagiarism.

I declare that all material described in this report is all my own work except where explicitly and individually indicated in the text. This includes ideas described in the text, figures and computer programs.

Name: Sreethyan Aravinthan .....

Signature:  .....

Date: 21/04/2020.....

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Work carried out . . . . .	2
1.2	Remaining Work . . . . .	4
<b>2</b>	<b>Goals and Objective</b>	<b>5</b>
<b>3</b>	<b>Theory and Analytical Bases for the Work</b>	<b>6</b>
3.1	Methods of transmitting data from source to destination . . . . .	6
3.1.1	Broadcast-and-select . . . . .	6
3.1.2	Routing and Wavelength Assignment (RWA) . . . . .	7
3.1.3	Chosen method . . . . .	7
3.2	Network Topology . . . . .	7
3.2.1	Ring topology . . . . .	8
3.2.2	Star network - Coupler . . . . .	8
3.2.3	Star network - Router . . . . .	9
3.2.4	Spine-leaf Topology . . . . .	10
3.2.5	Chosen Topology . . . . .	10
3.3	Optical technologies . . . . .	10
3.4	Transceiver design for the photonic network . . . . .	11
3.4.1	Transmitter for the photonic network . . . . .	12
3.4.2	Receiver for the photonic network . . . . .	13
3.5	Processor Architecture . . . . .	13
3.5.1	CISC . . . . .	14
3.5.2	RISC . . . . .	14
3.5.3	Chosen processor architecture . . . . .	14
3.6	Node architecture . . . . .	14
3.6.1	GPP and CP . . . . .	14
3.6.2	GPP with interrupts . . . . .	15
3.6.3	Chosen node architecture . . . . .	15

<b>4</b>	<b>Technical Method</b>	<b>16</b>
4.1	Overall node architecture . . . . .	16
4.2	General Purpose Processor (GPP) . . . . .	18
4.2.1	Instruction Set Architecture . . . . .	18
4.2.2	Instruction format in machine language . . . . .	20
4.2.3	Microarchitecture of GPP . . . . .	22
4.2.4	Assembler . . . . .	25
4.3	Communications Processor (CP) . . . . .	26
4.3.1	Communication protocol . . . . .	26
4.3.2	Transferring data from GPP to CP . . . . .	28
4.3.3	Retrieving data from the CP to GPP . . . . .	28
4.3.4	Microarchitecture of CP . . . . .	29
4.4	Microarchitecture of Microprocessor . . . . .	35
4.5	Testing . . . . .	35
<b>5</b>	<b>Results and Analysis</b>	<b>39</b>
5.1	Results from Simulation . . . . .	39
5.2	Synthesis . . . . .	55
<b>6</b>	<b>Conclusion</b>	<b>57</b>
	<b>Appendices</b>	<b>62</b>
<b>A</b>	<b>GPP</b>	<b>63</b>
<b>B</b>	<b>CP</b>	<b>107</b>
<b>C</b>	<b>Testbenches</b>	<b>133</b>
<b>D</b>	<b>Assembly programs used for testing</b>	<b>141</b>
<b>E</b>	<b>Assembler code</b>	<b>145</b>

## **Abstract**

Data traffic has been exponentially increasing within data centres and high performance computers over the years. In order to accommodate future traffic growth, alternative technologies need to be investigated, as the currently-used electronic packet switching approaches lead to bottlenecks constricting network throughput. Wavelength division multiplexing (WDM) and space division multiplexing (SDM) are two complementary optical technologies to increase link throughputs. Using wavelength and space domains to route signals optically between nodes also offers high flexibility and throughput and low latencies, avoiding the constrictions of electronic packet switched networks. This report describes the specification, design, implementation and testing of a novel microprocessor design, designed to carry out compute operations and communicate and exchange data with a large number of other such compute nodes (on the order of thousands of nodes). The interconnect network considered was a high capacity optical broadcast-and-select interconnect network with a simple ring topology, based on combined WDM and SDM technologies. The microarchitecture of the microprocessor was developed in SystemVerilog and verified using ModelSim. Parts of the microprocessor were experimentally implemented and tested using a field programmable gate array (FPGA). All the simulation and experimental results were successful.

# Chapter 1

## Introduction

At the core of many services provided by big companies like Google or Amazon [1] are data centres which contain thousands of servers. Over the years there has been an increase in data traffic and to support future demands, the data centre networks interconnecting the servers should be scaleable to support bandwidth and low latency that will be required [2]. Current intra-data centre networks implementations use optical fibre links with electronic switching and have a multi-layer/hierarchical design [2, 3]. In this design the bottom layer contains switches which help connect to the servers or racks; the top layers contain switches that help distribute the traffic [3]. FatTree is an example of such a design [4].

High performance computers are used for more complex computational tasks such as quantum mechanics, drug discovery or climate modelling [5, 6]. A number of techniques to improve high performance computer's performance have been proposed, including employing different computer architectures and increasing parallelism [7]. Supercomputers that use multiple processors working in parallel use Message Passing Interface (MPI). However, the limitations of electronic packet switching leads to the network being congested and packets being lost [6], thus resulting in performance loss in high performance computers.

The data traffic within data centres and high performance computers has been exponentially increasing over the years. In the near future there will be an increase in the number of servers and compute nodes, and the bisection bandwidth for the interconnect networks within data centres and high performance computers will be on the order of hundreds of Tb/s. The ability to switch and route such large data volumes will start to exceed the capabilities of the electronic packet switches which are currently used. Hence to achieve the performance required data centres and high performance computers should not rely on current electronic interconnects as these will not be sufficient without latency, increased complexity and inefficiency

[8].

In order to tackle the problems faced by electronic packet switching, optical routing is a promising technology. It can provide higher bandwidth for transmitting, parallel transmission innately (WDM) and low crosstalk [8]. Since data can be sent on multiple different wavelengths in a single optical fibre, this data must be split according to receive the correct data. There are two main methods Arrayed Waveguide Grating (AWG) and optical tunable bandpass filter. An AWG takes the multiplexed signal at the transmitter and will demultiplex the signal into the individual wavelength that made up the signal. An optical tunable bandpass filter receives a control signal which selects which wavelength to pass through. In this project it is assumed that an optical bandpass filter is used. There are many types of optical bandpass filter, including semiconductor-based devices, ferroelectric liquid crystal Fabry Perot filters, micro machined device and acoustic-optic tunable filters [9]. The fastest tuning filters are semiconductor-based, allowing rapid tuning on the nanosecond time scale.

This project is focused on the design of a novel microprocessor, where tens of thousands of such processors can be interconnected with a high capacity optical broadcast-and-select interconnect network with a simple ring topology, using multiple spatial channels and wavelengths, which is known as space and wavelength division multiplexing (SWDM). Using these two technologies, and the broadcast-and-select operating principle, in which each node has its own unique wavelength-/spatial channel on which to transmit data, a large scale network can be implemented, for example, 10,000 nodes, using 100 spatial channels and 100 wavelengths, with each compute node having its own unique SWDM channel, allowing, in principle, almost instantaneous data transmission between any node pair. By avoiding the congestion and delays arising from electronic packet switches sharing low bandwidth communication channels, this concept exploits the massive capacity of space and wavelength division multiplexing to simplify the routing and scheduling of data packets between nodes, increasing throughput and reducing latency.

## 1.1 Work carried out

Optical routing for data centres and high performance computing has been the focus of much previous research. One proposed approach is a WDM-based Reconfigurable Hierarchical Optical Data Center Architecture (RHODA) [3]. This architecture is a two-level hierarchical and reconfigurable data centre network architecture, designed to tackle varying traffic patterns. This is achieved by using optical space switches and Wavelength Selective Switches (WSSs). It has been

shown that this particular architecture outperforms other architectures such as FatTree and WaveCube. Another approach is a novel Optical Switching Architecture for data centre networks which is discussed in Ref. [1]. The architecture described here aims to be flexible for different traffic patterns and the results from the tests carried out show that this architecture can provide a high bisection bandwidth for the set of traffic patterns.

Optically-routed networks utilise spatial and wavelength domains to route signals at intermediate nodes. A major challenge in maximising the throughput of such networks is optimising the routing wavelength assignment (RWA) algorithm. One such algorithm is based on the Intelligent Water Drops (IWD) [10]. This algorithm was designed for a WDM network and can achieve a lower blocking probability when compared to fixed routing and adaptive shortest routing. Another algorithm was designed for static traffic which will help tackle the RWA assignment issue [11]. The algorithm presented was a new genetic algorithm and the results from the simulation showed that the algorithm performed a bit better than the standard techniques for routing and wavelength assignments.

In contrast to optical routing described above, the method used for transmitting data from the source to destination nodes considered in this project is broadcast-and-select. The destination node uses a tunable receiver, and the challenge with this technique is receiver conflict, with multiple source nodes attempting to transmit to the same destination node simultaneously. This may lead to packets being lost. A proposed solution for this is to use receivers with a conflict algorithm that is based on a learning automata [12]. This solution is proposed for a WDM star network and the results of this implementation show that there is an improvement in performance under the specified protocols [13, 14].

This report focuses on the design of microprocessors suitable for operating in the SWDM broadcast-and-select network. The design philosophy followed was that of RISC. RISC processors employ simple hardware and more complex software to achieve the same tasks as Complex Instruction Set Computer (CISC) processors, but with lower cost and power consumption. An example of a RISC processor is the Microprocessor without Interlocked Pipelined Stages (MIPS) processor. Ref. [15] describes the design of a single clock cycle MIPS processor in VHDL. This work covered the design of the MIPS processor and the simulations of the top-level modules. Another single cycle RISC processor which is based on the open-source RISC-V (RV32I) instruction set architecture, with an example of the microarchitecture designed in Verilog described in [16]. The particular processor design described in this paper is modular and can be easily extended. These properties are



very useful for the system that was designed in this project. Both microarchitectures vary but do have similar implementations. However, the microarchitecture design for MIPS [15] is simpler than the microarchitecture for the RISC-V [16].

## **1.2 Remaining Work**

This report introduces the operating principles of the SWDM broadcast-and-select data centre interconnect network, and the specifications of the microprocessors controlling the transmission and reception of data in such networks. It then describes the detailed design, implementation and testing of a RISC microprocessor which was developed for this application in this project, followed by the results from testing the microprocessor.

# Chapter 2

## Goals and Objective

The main goal of the project was to specify, design, implement and test a micro-processor system which can carry out standard arithmetic and logical operations and conditional and unconditional branching, which is Turing-complete (i.e. able to carry out any program), and is able to communicate with and transfer data to and from a large numbers of other such processors working in parallel, over a space- and wavelength-division multiplexed broadcast-and-select optical ring network.

The detailed steps to be taken to achieve this goal were as follows:

- A multi-processor system architecture was to be proposed, based on a novel high-throughput SWDM broadcast-and-select ring interconnect network.
- The optical transceiver design and its interface to the microprocessor were to be proposed.
- The operating protocols of the network were to be specified, including the operation of the control and data planes.
- A Turing-complete Reduced Instruction Set Computer (RISC) processor architecture was to be specified, based on existing designs, in particular the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture
- The microarchitecture, with single instruction per clock cycle operation, was to be designed, implemented in SystemVerilog and its operation simulated
- The code was to be synthesised and loaded onto an Field Programmable Gate Array (FPGA) and then the FPGA was to be tested to confirm the processor operates as designed

# Chapter 3

## Theory and Analytical Bases for the Work

The aim of this project is to design and test a microprocessor for operation in intra-data centre photonic networks, in which thousands of processing nodes are interconnected with high throughput and low latency.

### 3.1 Methods of transmitting data from source to destination

There are two methods of transmitting data from source to destination: broadcast-and-select and routing and wavelength assignment (RWA). Both methods can behave as a single-hop network or a multi-hop network. A single-hop network transmits the data and it reaches the destination node without turning into an electronic form in-between [17]. However, in a multi-hop network the data will be converted into electronic form at the intermediate nodes [17].

#### 3.1.1 Broadcast-and-select

When setting up a system using broadcast-and-select, the data transmitted by a node will be received by all the nodes in the network. Each node can be equipped with one of the following: a tunable laser and fixed filter or fixed laser and tunable filter or tunable laser and filter. In such a network there is a control plane and a data plane, where there are multiple data channels. Typically the number of data channels is equal to the number of nodes in the network [17]. It is important that there will be no problems when data is received at the destination node on the data plane. An example of such an issue can be considered. If each node is equipped with a fixed laser and a tunable filter and if two nodes transmit to

the same node, then the receiver node can only receive the data from one of the nodes. Thus the data from the other node is lost. In order to mitigate this issue a protocol is implemented on the control plane.

### **3.1.2 Routing and Wavelength Assignment (RWA)**

In this method of transmitting, a node is given a route and a wavelength to transmit on. This wavelength should be fixed from source to destination. However, if wavelength converters are used then different wavelengths will be used throughout the journey from source to destination. For such a problem an aim could be to reduce the number of wavelengths used from source to destination [18]. Another aim instead can be to increase the number of links for a set number of wavelengths from source to destination [18]. This problem can be solved by looking at the routing and wavelength assignments separately [18]. Thus reducing the complexity of the problem.

### **3.1.3 Chosen method**

In this project broadcast-and-select was chosen as the method for transmitting data between the source and destination. This was chosen as we can leverage the very wide bandwidth of SWDM to achieve low complexity node interconnection avoiding congestion at intermediate nodes which occurs with wavelength routing at intermediate nodes.

## **3.2 Network Topology**

Many topologies exist to implement a communications network, including ring, star, mesh and spine-leaf. Each of these topologies have their own advantages and disadvantages. For example, some are better for Local Area Network (LAN) and some might be better for Wide Area Network (WAN). Below is an explanation as to how these topologies work.

### 3.2.1 Ring topology

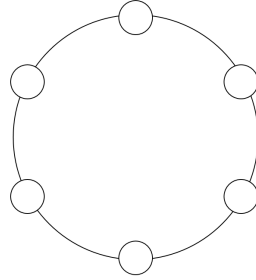


Figure 3.1: Ring topology

Figure 3.1 shows how the nodes on a network are connected in a ring topology. In this topology each node is connected to two other nodes only. Connecting each node in such matter allows a packet sent by one node to be received by all other nodes, in a broadcast-and-select operation. The packets can travel either clockwise or anticlockwise. This is referred to as unidirectional. However, if packets travel in this manner and a link is broken then the packet cannot reach all the nodes. Due to this packets can also travel in both directions (bidirectional), and protection switching detects failure and switches to reception of the signals circulating in the opposite direction.

### 3.2.2 Star network - Coupler

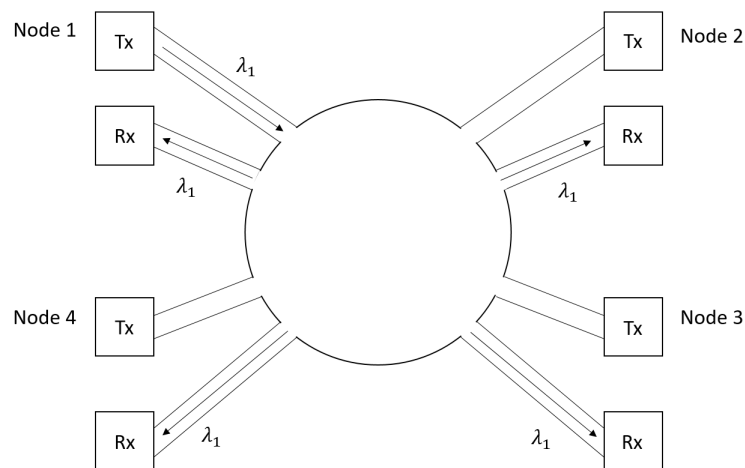


Figure 3.2: Star network with a coupler

Figure 3.2 shows how the nodes on a network are connected in a star network where the centre is a  $N \times N$  coupler. In this set up, the coupler distributes the data from each node to all the other nodes in the network, independently of the transmitted wavelength. The receiver at each node has a tunable filter which is used to tune to the wavelength that the data was transmitted at.

### 3.2.3 Star network - Router

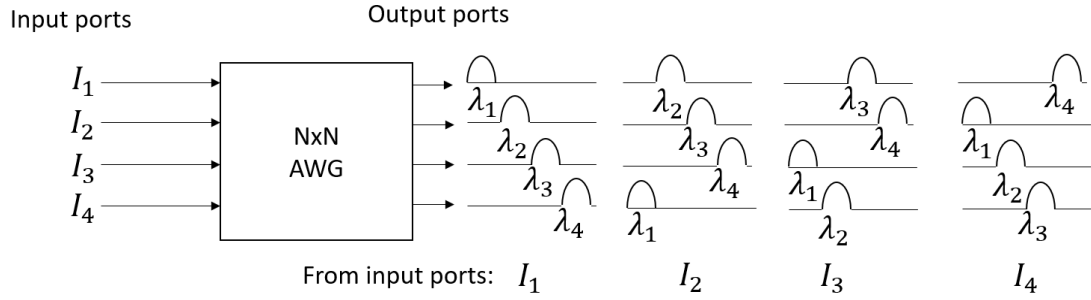


Figure 3.3: Cyclic AWG

For an  $N \times N$  Arrayed Waveguide Grating (AWG) wavelength router there are  $N$  input ports and  $N$  output ports. Each of the input ports have WDM signals on  $N$  different wavelengths [19]. From a single input port one wavelength is passed to each output port [19]. The connections from each input port to each output port are made based on the wavelength, as shown in the figure [19].

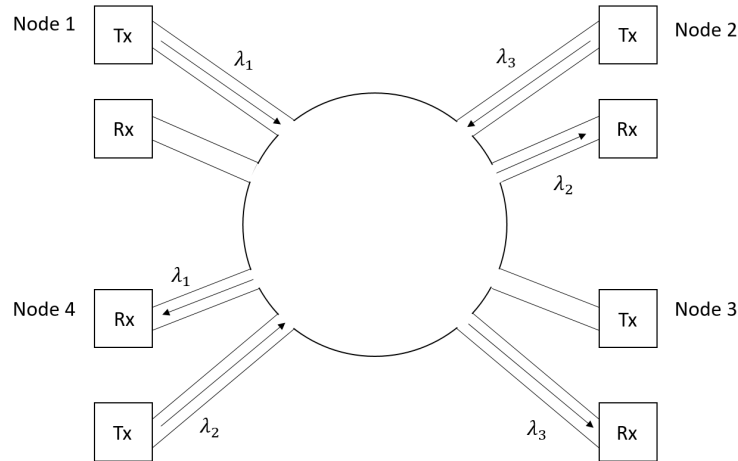


Figure 3.4: Star network with a router

Figure 3.4 shows how the nodes on a network are connected in a star network in which the centre operates as a wavelength router. In this set up the router will receive the data at a certain wavelength and routes the data to the receiver that can accept data at the same wavelength. This can be done via an AWG hub router [20]. The transmitter at each node will have a tunable laser which will allow the packets to be sent with different wavelengths.

### 3.2.4 Spine-leaf Topology

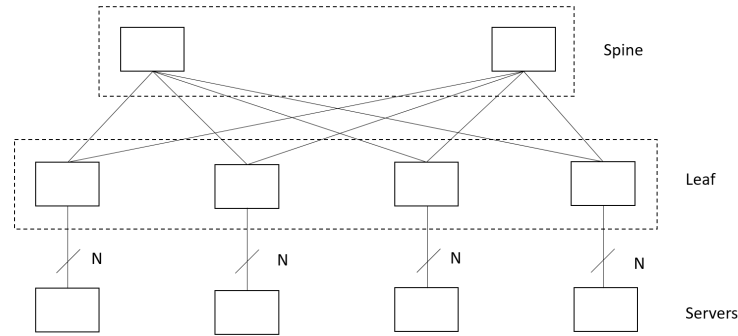


Figure 3.5: Spine-leaf topology

Figure 3.5 shows how the nodes on a network are connected in a spine-leaf topology. In this topology there are two layers. The spine layer and the leaf layer. In the leaf layer each switch is connected to multiple servers. Each switch in the spine layer is interconnected with each switch in the leaf layer. This way any server can communicate to any other server.

### 3.2.5 Chosen Topology

In this project the ring topology was chosen as the method to connect all the nodes together. This was because this network can be bi-directional. Therefore, if there is a break in network then the opposite direction can be used to send the data.

## 3.3 Optical technologies

WDM technology allows multiple optical signals with different wavelengths to be multiplexed onto a single optical fiber and then demultiplex this signal at the receiver to recover the individual wavelengths that were transmitted.

SDM technology allows multiple optical data signals to be transmitted and received simultaneously because each signal will be given a separate spatial channel. This can be achieved using, for example, multiple optical fibres, each with a single core, or using one or more multi-core fibres.

Combining the above two technologies, and allocating a unique SWDM channel for transmission from each processing node, a network with 100 spatial channels and 100 wavelengths realised which allows the interconnection of up to 10,000 nodes, with each node on the network will have a dedicated wavelength and spatial channel.

### 3.4 Transceiver design for the photonic network

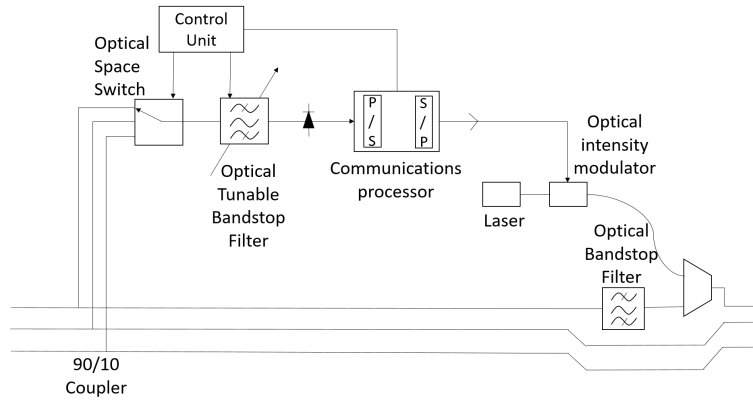


Figure 3.6: Design for photonic transceiver. S/P - serial to parallel conversion. P/S - parallel to serial conversion

Figure 3.6 shows the design of the transceiver at a node for the photonic network. Below are explanations as to how the transmitter and receiver will be designed using the optical technologies explained above.



### 3.4.1 Transmitter for the photonic network

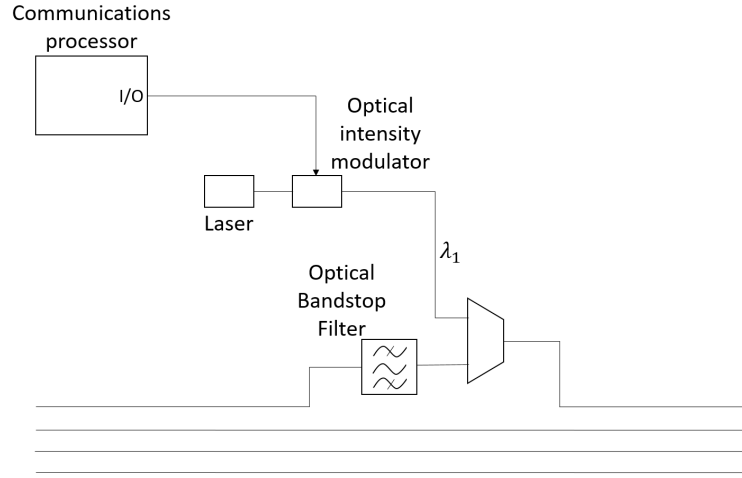


Figure 3.7: Design for photonic transmitter

Figure 3.7 shows the design of the transmitter at a node for the photonic network. The laser emits light at a specified wavelength, e.g.  $\lambda_1$  for this node. The output of the laser is connected to an optical intensity modulator. This will transmit the binary data that was given as an input. The output of the optical intensity modulator is varied between high and low levels, encoding the data as binary 1 and 0, respectively. This data on wavelength  $\lambda_1$  that has already circulated around the ring is removed using a band-stop filter. The output of the modulator is multiplexed with the other signals that pass by the node. The output of the multiplexer will then be going to the next node with all the other fibres.

### 3.4.2 Receiver for the photonic network

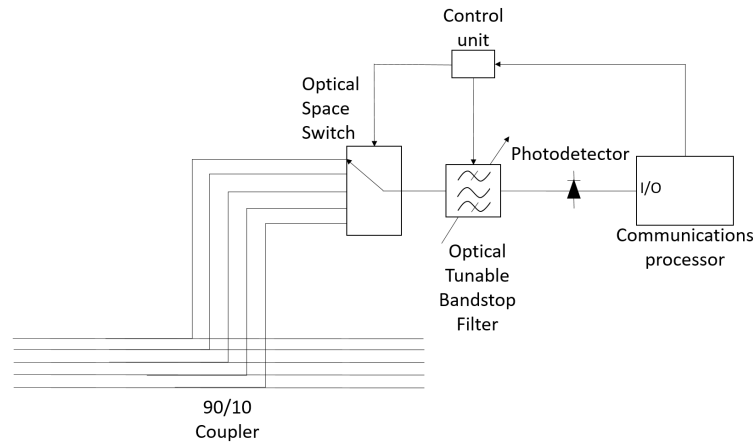


Figure 3.8: Design for photonic receiver

Figure 3.8 shows the design of the receiver at a node for the photonic network. All the fibres will be connected to the optical space switch via a 90/10 coupler. This means that only 10% of the power will be given to the optical space switch. The optical space switch has a control signal which comes from a control unit which is analog. This control signal selects which of the spatial channels will be passed to the output of the optical space switch. The output will be connected to the tunable optical band-pass filter. Since this filter is tunable it receives an analog control voltage from the control unit. This signal determines which wavelength is selected for reception at the node. The resulting signal is incident on a photodetector which converts the optical signal into an electrical signal. This electrical signal is connected to the communications processor which carries out serial to parallel conversion. The resulting data is the packet that has to be processed by the processor.

## 3.5 Processor Architecture

There are many types of computer architectures available. These include Complex Instruction Set Computer (CISC), RISC, One-Instruction Set Computer (OSIC) and Zero Instruction Set Computer (ZSIC). Each has its own advantages and disadvantages. However, the two processor architectures that were under consideration for this project were CISC and RISC. Below an explanation as to what each processor architecture is and which processor architecture was used for the project.

### **3.5.1 CISC**

This processor architecture consists of many instructions, where each instruction has different widths and each instruction does not take the same number of clock cycles. This is feasible since the hardware implementation is complicated for this processor architecture. Due to the large number of instructions available there are a lot of instructions that can access the memory (RAM). There are some instructions that can read and write to memory. In such architectures there are not many registers, but they are specialised. In addition to all this there are multiple addressing modes. Examples of CISC processor architectures are Intel x86, Zilog Z8000 and Freescale 9S12.

### **3.5.2 RISC**

This processor architecture does not have many instructions, however, each instruction's width is fixed and each instruction requires a single clock cycle to execute. This leads to a less complex hardware implementation. This means that more lines of code are required to achieve a similar program that will run on a CISC architecture. Due to the hardware complexity of RISC there are not many instructions that can access the memory and there are no instructions that can read and write to memory. Unlike CISC, this architecture provides a lot of general purpose registers but fewer addressing modes. Examples of RISC processor architectures are MIPS, ARM and MSP430.

### **3.5.3 Chosen processor architecture**

In this project the RISC processor architecture was chosen. This was due to the simplicity of this processor architecture which is easier and quicker to implement. The RISC architecture also has a better power performance than CISC.

## **3.6 Node architecture**

There were multiple approaches for the node architecture. However, the only two that seemed feasible were a GPP and CP linked together or a GPP that uses interrupts. Below an explanation as to what each node architecture is and which node architecture was used for the project.

### **3.6.1 GPP and CP**

The node composes of a GPP and a CP. The GPP is Turing complete. Thus this should be able to execute any program. The CP is responsible for pinging a node

to transmit data, if there is any data to send. In addition to this, this processor is responsible to respond to any pings such that data can be received. These pings are managed by the control plane and the data plane is responsible for transmitting and receiving data. The transmitted data is saved in the transmitter's RAM and the received data is saved in the receiver's RAM. The output of the GPP's RAM is connected to the input of the transmitter's RAM. Similarly, the output of the receiver's RAM is connected as one of the inputs to the GPP's RAM. These are the two links that are used to connect the GPP and CP.

### **3.6.2 GPP with interrupts**

In this node architecture the node has a GPP only, which is Turing complete. In this system the communications system is implemented via interrupts. This means that the workings of the control plane and data plane will be done via interrupts. For example a ping to transmit data will be done via an interrupt and the response to this ping will be received and the processor will register it via an interrupt. Then depending on the response the data will be transmitted. If the data is transmitted then this will be done via an interrupt as well. During these interrupts the main program will not be executing. Rather the interrupt service routine for each of these interrupts will be executing.

### **3.6.3 Chosen node architecture**

In this project the GPP and CP was chosen for the node architecture. This architecture has a dedicated CP which will be more efficient at transmitting/receiving data as this is the sole purpose of that processor. The GPP with interrupts will not be as efficient as there will be cycles being wasted that handle interrupts instead of executing the main code.

# Chapter 4

## Technical Method

### 4.1 Overall node architecture

As described in subsection 3.6.3 the node architecture chosen employs a Turing-complete general purpose processor (GPP) for compute and a separate communications processor (CP) to handle communications with other processor nodes in the network.

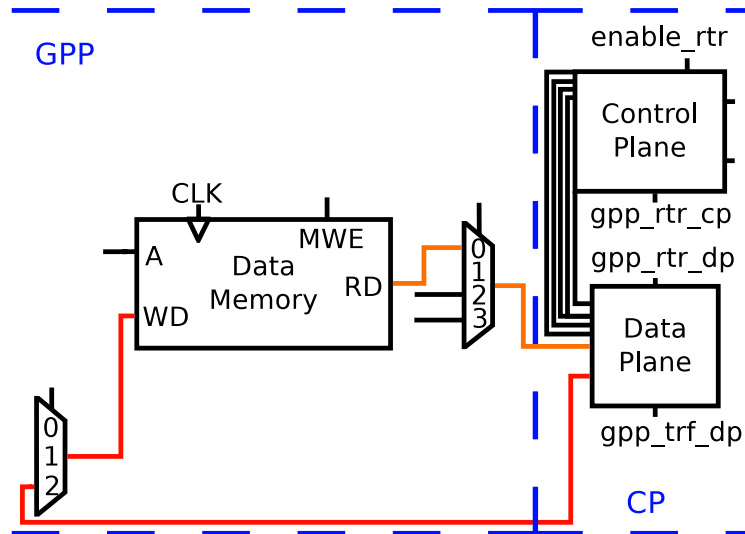


Figure 4.1: Data transfer between GPP and CP

Figure 4.1 shows the interconnection between the GPP and the CP for data transfer. The orange line indicates the data transfer from GPP to CP, provided the correct control signals are applied. The red line shows the transfer from CP to GPP, provided the correct control signals are applied. The remainder of this chap-

ter focuses on the implementation of the GPP and CP and their interface. A section exists which covers all the testing that was carried out to confirm that the system works as specified.

## 4.2 General Purpose Processor (GPP)

### 4.2.1 Instruction Set Architecture

Table 4.1: Instruction set architecture for the microprocessor that is going to be designed

Instruction	Name	Action	Opcode
ld regX, regY	load	regY = *regX	000000
str regX, regY	store	*regX = regY	000001
add regX, regY	add	regY = regX + regY	000010
sub regX, regY	sub	regY = regX - regY	000011
and regX, regY	and	regY = regX & regY	000100
or regX, regY	or	regY = regX — regY	000101
mov regX, regY	move	regY = regX	000110
cmp regX, regY	compare	regX == regY	000111
jz #label	jump if zero	if(ZF == 1) pc = #label	001000
jmp #label	unconditional jump	pc = #label	001001
movi regX, #immediate	move immediate	regX = #immediate	001010
addi regX, #immediate	add immediate	regX = regX + #immediate	001011
subi regX, #immediate	sub immediate	regX = regX - #immediate	001100
andi regX, #immediate	and immediate	regX = regX & #immediate	001101
ori regX, #immediate	or immediate	regX = regX — #immediate	001110
push regX	push	sp = sp + 1 && *sp = regX	001111
pop regX	pop	regX = *sp && sp = sp - 1	010000
call #label	call subroutine	pc = #label	010001
return	return from subroutine	pc = *sp	010010
jb #label	jump below	if(CF == 1) pc = #label	010011
jbe #label	jump below equal	if(CF == 1 or ZF == 1) pc = #label	010100

ja #label	jump above	if(CF == 0 and ZF == 0) pc = #label	010101
jae #label	jump above equal	if(CF == 0) pc = #label	010110
jg #label	jump greater	if(SF == OF and ZF == 0) pc = #label	010111
jge #label	jump greater equal	if(SF == OF) pc = #label	011000
jl #label	jump less	if(SF != OF) pc = #label	011001
jle #label	jump less equal	if(SF != OF or ZF == 1) pc = #label	011010
cbt	compare before transfer	if(trf == 1)	011011
trf	transfer	data_memory = tx_RAM	011100
pr	pause receiver	retrieve = 0	011101
cbr	compare before retrieving	if (rtr == 1)	011110
rtr	retrieve	rx_RAM = data_memory	011111
rr	resume retrieve	retrieve = 1	100000



Table 4.2.1 shows the instruction set architecture that was implemented for the GPP in this project. This set was designed to contain simple and/or basic instructions, not advanced/complex instructions. This microprocessor is classified as a Reduced Instruction Set Computer (RISC).

### 4.2.2 Instruction format in machine language

The equivalent machine code for the instructions specified in Table 4.2.1 will now be described. There are three types of instructions:

- Simple 16-bit instructions
- Immediate instructions
- Jump instructions

#### Simple 16-bit Instructions

These are instructions which take up a single address in ROM and are 16-bits wide. These instructions include ld, str, add, sub, and, or, mov, cmp, push, pop, return, cbr, trf, pr, cbr, rtr & rr.

For example the move instruction in assembly has the following format `mov regX, regY`. Starting from the Most Significant Bits (MSBs) the instruction is formatted with the opcode of `mov` followed by the machine code of operand 1 and then operand 2. In this scenario the opcode is 000110. The machine code of `regX` is XXXXX and `regY` is YYYYY. Since `regX` and `regY` can be any register available it has been assigned XXXXX and YYYYY respectively to refer to the general case. Therefore, the machine code equivalent of `mov regX, regY` is 000110XXXXXYYYYY. The opcode is 6-bits wide and both of the operands are 5-bits wide. Therefore, the instruction is 16-bits wide.

Not all of the instructions which are classed as simple 16-bit instructions have 2 operands. For example `push`, `pop`, etc. For these instructions the missing operand is given a default value of 00000, which is equivalent to the 1st register in the register file (`reg0`).

#### Immediate Instructions

These are instructions which take up two consecutive addresses in ROM and both are 16-bits wide. These instructions include `movi`, `addi`, `subi`, `andi`, & `ori`.

To understand the format of the instructions in machine code the following example can be considered `movi regX, #immediate`. The 6 MSBs of the lower address is equal to the opcode of the instruction. The 10 Least Significant Bits (LSBs) for this address is equal to the first operand repeated twice. The higher address takes the binary equivalent of the immediate provided by the programmer. This is because an immediate is just a number e.g. 10, -10. Therefore, this needs to be converted to binary and stored in that higher address. This shows that the immediate value can take a 16-bit value.

## **Jump Instructions**

These are instructions which take up two consecutive addresses in ROM and both are 16-bits wide. These instructions include `jz`, `jmp`, `call`, `jb`, `jbe`, `ja`, `jae`, `jg`, `jge`, `jl` & `jle`.

To understand the format of the instructions in machine code the following example can be considered `jmp #label`. The 6 MSBs of the lower address is equal to the opcode of the instruction. The 10 LSBs are equal to zero. The higher address is equal to the binary equivalent of the label. This value is the address that the code will jump to if the condition for the jump instruction is met.

### 4.2.3 Microarchitecture of GPP

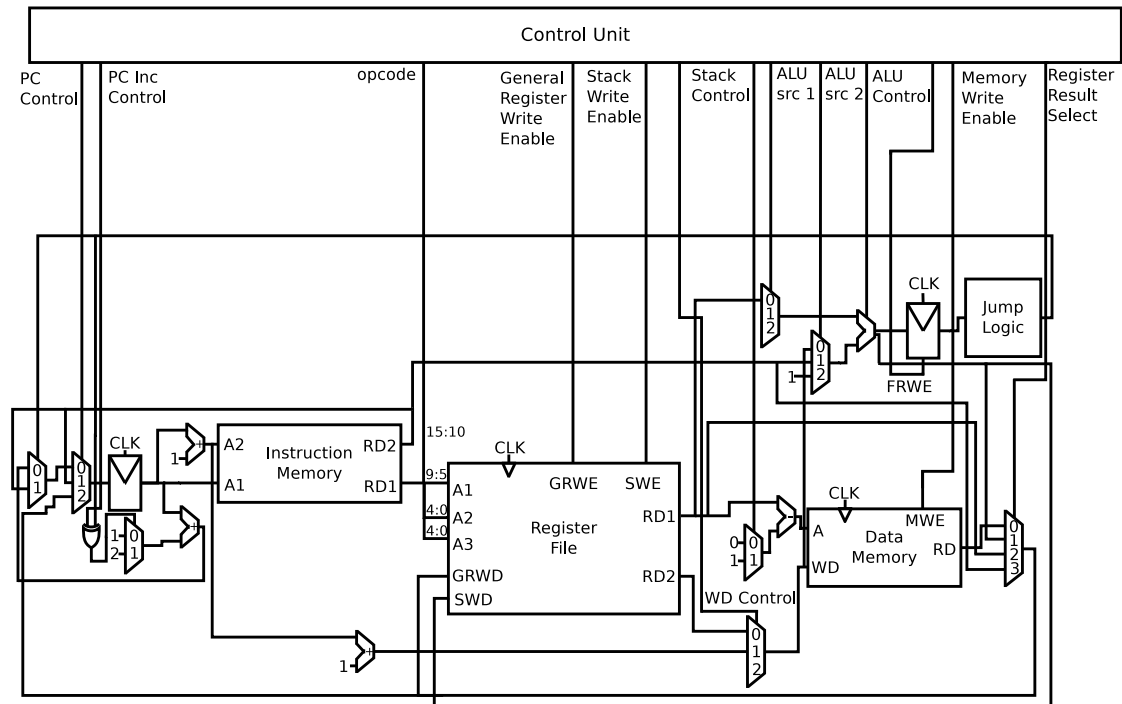


Figure 4.2: Microarchitecture for the GPP

The above microarchitecture is for the instruction set defined in Table 4.2.1, and is based on the microarchitecture for the single-cycle MIPS processor in [21]. Below, the lower-level modules that make up the GPP are highlighted and their operation explained.

#### Arithmetic Logic Unit (ALU)

The purpose of this component is to carry out basic arithmetic and logical operations which are add, sub, and & or. This component has three inputs in total. Two of the inputs are the data on which the operation will be applied on. The final input controls what operation to carry out. This component has a single output which is the result after applying the operation on the two data inputs. Refer to Listing A.1 in the appendix for the code.

#### ALU advanced

The purpose of this component is to do basic arithmetic and logical operations which are add, sub, and & or. In addition to this four flags are updated after each

operation which are useful for determining if a jump should take place. These flags are the zero flag, sign flag, overflow flag and carry flag. This component has three inputs in total. Two of the inputs are the data on which the operation will be applied on. The final input controls what operation to carry out. There are 5 outputs from this component. One of the outputs is the result after applying the operation on the two data inputs. The other four outputs correspond to the value of the flags after the operation has been carried out. Refer to Listing A.2 in the appendix for the code.

### **Control Unit**

This component will set the appropriate control signals depending on the opcode of the instruction. The input to this component is the opcode. The output are all the available control signals on the processor. Refer to Listing A.3 in the appendix for the code.

### **Data Memory**

This component is used to hold variables, register values during an interrupt or return addresses when a program is being executed. There are four inputs to this component which include a clock signal, address to read from or write to, data to be written and a write enable. There is a single output which corresponds to the data read from the address value at the address input. Refer to Listing A.4 in the appendix for the code.

### **Flags Register**

This is a register which has an enable signal. The value at the input will propagate if and only if it is a positive edge of a clock and the enable signal is high. Refer to Listing A.5 in the appendix for the code.

### **General Purpose Register File**

This component has an array of registers to which the program running on the GPP can read from or write to. These registers hold temporary values when doing some computation. There are two inputs which are addresses used to access one of the registers. The data at the addresses given by these two inputs are given as two outputs. Another input address exists to write to one of the registers at any given time and the data is given as an input. There is a write enable signal which allows the data to be written if it is high and it is the positive edge of the clock. There is also another data input and write enable signal. These two inputs are for the 1st register in the array of registers only as this is dedicated as the stack

pointer. In addition to this there is a clock signal. Refer to Listing A.6 in the appendix for the code.

## Instruction Memory

This component holds the machine code that will run on the GPP. This contains two inputs which are both addresses. There are two outputs which correspond to the data at the addresses given at the input. Refer to Listing A.7 in the appendix for the code.

## Jump Logic

This component is used to indicate if a jump should take place or not. The inputs will be the zero flag, overflow flag, sign flag and carry flag. In addition to this there will be other control signals which correspond to the different types of jumps that can take place. Including jump if zero and unconditional jump, this logic also supports jumps after comparing data as signed or unsigned numbers. Table 4.2 and Table ?? shows the conditions that have to be met by each jump instruction for unsigned and signed numbers respectively. The output of this module is binary indicating if a jump should take place or not. Refer to Listing A.8 in the appendix for the code. The logic is based off of the x86 architecture.

Instruction	Condition
jb (jump below)	carry flag = 1
jbe (jump below equal)	carry flag = 1 or zero flag = 1
ja (jump above)	carry flag = 0 or zero flag = 0
jae (jump above equal)	carry flag = 0

Table 4.2: Conditions that have to be met for the following jump instructions to take place when comparing unsigned numbers

Instruction	Condition
jg (jump greater)	sign flag = overflow flag and zero flag = 0
jge (jump greater equal)	sign flag = overflow flag
jl (jump less)	sign flag $\neq$ overflow flag
jle (jump less equal)	sign flag $\neq$ overflow flag or zero flag = 1

Table 4.3: Conditions that have to be met for the following jump instructions to take place when comparing signed numbers

## Multiplexer

This component will take N inputs which are M-bits wide and select one of the inputs via a control signal. Refer to Listing A.9 in the appendix for the code.

## Register

This component will behave as an asynchronous register. Refer to Listing A.10 in the appendix for the code.

## Datapath

This describes how the components should be connected together for data processing. Refer to Listing A.11 in the appendix for the code.

### 4.2.4 Assembler

An assembler was written (in the C language) for the general purpose processor. Its functionality is described below.

The assembler takes as many files as specified. This is useful if there are some functions which are used repeatedly in multiple different programs, e.g. division. These functions can be saved in a single file and can be included in different programs. All the input files are combined in the order in which the files were submitted and saved in a file called combined.asm.

After this the comments are removed from this file. In this standard the ';' is used as the delimiter for a comment. Once each line is parsed for comments and removed if present, the result is saved in a comment-free file called no\_comments.asm. Upon completing this the assembler will go through the code in no\_comments.asm and replace all the labels with their effective addresses. This is achieved by parsing the no\_comments.asm to find out all the labels present in the code and what the effective address's of each label is. Then the file is parsed again from the beginning by checking if the particular instruction has a label and if so replacing it with the effective address. This result is saved in a file called no\_labels.asm. If the line that is being parsed in the no\_comments.asm is a label (e.g. print: ) then ignore the line and proceed to the next line.

Once this is done the code that remains will be comment free with effective addresses for all labels that were present in the instructions. The final step is to simply go through each line and convert the opcode, operand, immediate values and effective addresses to machine code. Once this is done for each instruction it

should be saved in a file with extension .mem. This is important as the simulation software ModelSim can only load any machine language programs if it has the .mem file extension. Finally, the assembler gives all the resources back to the Operating System (OS) that were allocated during run-time and deletes all the temporary files that were created to convert the assembly code to machine code. Refer to E in the appendix for all the code that was used to design the assembler.

## 4.3 Communications Processor (CP)

### 4.3.1 Communication protocol

Each node is equipped with a control plane transceiver. The network employs one channel for the control plane, on which the transceivers can all transmit and receive, each using its control plane transceiver. The nodes share transmission time on this channel using a round-robin time division multiple access (TDMA) scheme, with each node's control plane receiver detecting all packets transmitted on this channel. Communicating on this control channel allows node-pairs to agree on the timing of data transmission. A source node can 'ping' another node to which it wishes to send a data packet (the destination node), and subsequently receives a response from that node indicating whether it is available to receive that data packet (this will depend on whether its data plane receiver is already engaged receiving data transmitted from elsewhere). If it is available, it switches its data plane receiver to the source node's channel, and the source node sends its packet. Due to the relatively small amount of information needed to be sent between nodes over the control channel, it is possible for a large number of nodes to share a relatively low bandwidth control plane using TDMA. The details of the protocol are explained in the remainder of this section.

To determine whether there is data to be sent by a node, the stack pointer of this node's CP transmitter RAM is compared to zero. If the stack pointer is zero then the RAM is empty and there is nothing to send. If the stack pointer is not zero then it is not empty and there is data to be sent.

Each packet in the TDMA control channel contains 32 bits. If there is data required to be sent from a source node, then the structure of this packet is such that the 16 MSBs is set to the destination node id and the 16 LSBs are equal to the source node id.

All other nodes in the network receive the packet, and the destination node with the id corresponding to the code in the packet takes appropriate action. It first

checks if it is already engaged in receiving data on the data plane. If this is the case, then on the control plane the destination node sends an acknowledgement packet with the source node id as the 16 MSBs and 0x0000 as the LSBs, indicating to the source node that it is busy and not currently available to receive its data. If the destination node is not busy, then on the control packet the 16 MSBs is equal to the source node id and the 16 LSBs is equal to 0xFFFF. In preparation for receiving the data, the destination node converts the 16-bit source node id code to analogue voltages (via lookup tables and digital-to-analogue converters) to the required values for the data receiver to select the source node's dedicated channel (through tuning the bandpass filter and setting the optical space switch). Note the source node id came with the packet that pinged the destination node.

The source node receives the response from the destination node. If the 16 LSBs of this packet are all equal to 0x0000, this indicates that the destination node is busy receiving data from another node. Therefore, the source node does not send the data on the data plane, but waits and tries again at the next opportunity. If the 16 LSBs are equal to 0xFFFF, indicating to the source node that the destination node is free to receive data, it waits for a period of time (a few microseconds) to allow the destination node receiver to select the correct spatial/wavelength channel, then sends the data on the data plane. The number of packets that it sends on the data plane is fixed to 5 - where each packet is 4 bytes. The destination node receives the data and it saves it in the CP's receiver RAM, to be transferred to the general purpose processor. A simple change to the programming of the CP can be carried out to adjust the size of the number of packets to achieve the optimum throughput for a given network or application.

The theoretical maximum possible frequency that the processor can operate at can be calculated. This is done by finding out the time taken for a packet to be transmitted and take the inverse to get the frequency. Using the fact that the transceiver rate is  $100 \text{ Gbs}^{-1}$  and each packet is 32-bits, the time taken can be calculated as follows

$$t_{32} = \frac{32}{100 \times 10^9} = 0.32 \text{ ns} \quad (4.1)$$

From this it is clear that the theoretical maximum possible frequency is  $f_{32} = 3.125 \times 10^9 \text{ Hz}$ . This means that the clock speed of the microprocessor can be set to  $3 \times 10^9 \text{ Hz}$  which is feasible by modern technology. From this the serial data rate is  $32 \times 3\text{GHz} = 96\text{Gbit/s}$ .



### 4.3.2 Transferring data from GPP to CP

A flag is output from the control plane which indicates if data can be transferred from the GPP to the CP. This flag is connected to one of the inputs of the ALU. The flag is set to 1, if the minimum number of clock cycles to transfer the data is present and if no data is being transmitted on the data plane. This second point is important because if data is being transmitted the stack pointer will get modified as the data is sent one after another. Therefore, in this case the stack pointer modified by the data plane. When data is transferred from the GPP, it will also be modify the stack pointer. Hence two resources will modify the value and the data will not be saved safely. Which is why it is important for the data plane not to transmit. If the conditions are not met then the flag is set to 0. To see if the flag is set the second input available at the ALU is used and the subtraction operation is carried out. This entire operation is done by a single instruction, cbt.

In addition to carrying out the above operation this instruction will set the appropriate flags. The only one that is important in this case is the zero flag. If this is set then data can be transferred. This is done via a single instruction called trf. Since fixed packets are being sent, which are of size five, the instruction needs to be called five times. If the zero flag is not set then do not transmit. To see if the zero flag has been set or not jump instructions can be used. Below is a sample assembly code indicating how this can be done. Note before running this code all the data needs to be pushed onto the GPP's RAM.

```
1  cbt    # compare to see if data can be transferred
2  jz transfer # if zero then data can be transferred
3  jmp exit  # if not then do not transfer
4 transfer:
5  trf
6  trf
7  trf
8  trf
9  trf
10 exit:
```

Listing 4.1: Sample assembly code showing how to transfer data between GPP and CP

### 4.3.3 Retrieving data from the CP to GPP

To achieve this the CP needs to be paused from saying yes to any pings during this period. This is done via a flag that will be controlled from the GPP. The instruction pr, does this job. After this it is important to check if the data plane is

receiving any data. Again for the same reason identified in subsection 4.3.2, data has to be written safely. The CP will output a flag to indicate this. This flag will be connected to one of the inputs of the ALU. The flag will be set if the data plane is receiving data. Otherwise it will be set to 0. To see if the flag is set the second input available at the ALU is used and the subtraction operation is carried out. This entire operation is done by a single instruction, cbr.

In addition to carrying out the above operation this instruction will set the appropriate flags. The only one that is important in this case is the zero flag. If this flag is set then the data plane is receiving data and the data should not be retrieved. If this flag is not set the data can be retrieved. This is done via a single instruction called rtr. After this the CP should be allowed to say yes to pings. This can be done using the rr instruction. Below is a sample assembly code indicating how this can be done.

```

1  pr # pause recieving on CP
2  cbr # instruction which sees if the data can be retrived
3  jz exit # if the result is zero then this indicates that the data plane is reciving
4  rtr
5  rtr
6  rtr
7  rtr
8  rtr
9  exit:
10 rr # resume receiving on CP

```

Listing 4.2: Sample assembly code showing how to retrieve data from the CP to the GPP

#### 4.3.4 Microarchitecture of CP

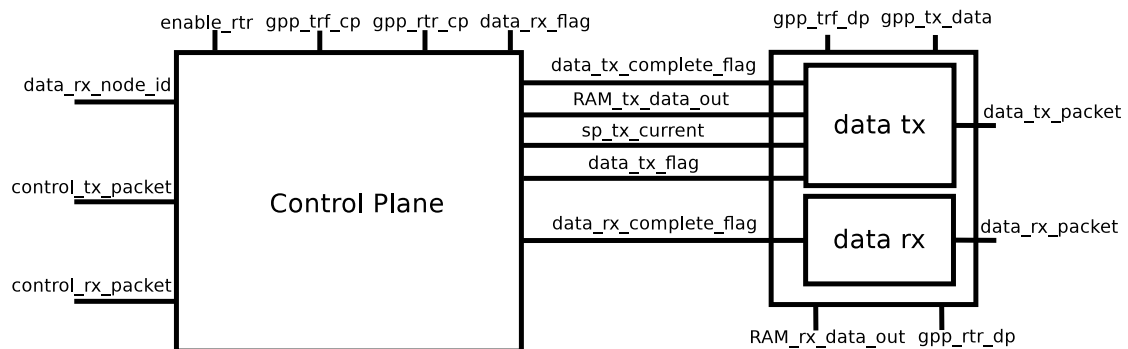


Figure 4.3: Microarchitecture of CP

Variable name	Explanation	Bit-width
control_rx_packet	This is the packet that will be received on the control plane	32
enable_rtr	This is a control signal that will allow the GPP to retrieve data from the CP if it is possible	1
gpp_rtr_cp	This is a control signal that pauses rx from GPP so that GPP can retrieve data from rx RAM	1
data_rx_packet	This is the packet that is received on the data plane	32
gpp_rtr_dp	This is a control signal which indicates the data plane to modify the stack pointer. Since the values are stored in a stack data structure, the value at the top of the stack is sent to the GPP from the DP receiver	1
gpp_trf_dp	This is a control signal which tells the data plane transmitter that data is going to be transferred from the GPP to the data plane transmitter	1
gpp_tx_data	This is the data that will be transferred from the GPP to the data plane transmitter RAM	16
control_tx_packet	This is the packet that is transmitted on the control plane	32
data_rx_node_id	This is the value that is sent to a control unit which sets the wavelength/spatial channel of the receiver	16

data_rx_flag	This is used to show the control plane if data is being received on the data plane. Thus the control plane will not say yes for another ping if set. This is also a flag that is used by the GPP to see if it can retrieve data from the data plane receiver	1
gpp_trf_cp	This is a flag that is used by the GPP to transfer data from the GPP RAM to the data plane transmitter RAM	1
RAM_rx_data_out	This shows the data that is outputted from the data plane receiver. It will be connected to the RAM of the GPP via a multiplexer	16
data_tx_packet	This is the packet that is transmitted on the data plane	32

Table 4.4: Explanation of the variable names in Figure 4.3

Figure 4.6 is an image of the microarchitecture for the CP. It shows how the control plane and the data plane are connected. Refer to Listing B.5 in the appendix for the top-level code. Below the lower-level modules that make up the CP will be highlighted and explained as to how they work.

### Control Plane

The control plane implements the protocol described in subsection 4.3.1 and the techniques described for transferring and retrieving data between the GPP and CP (subsection 4.3.2 and subsection 4.3.3). Since the description of the protocol is stated, a behavioural model was adopted when implementing this subsystem. Refer to Listing B.1 in the appendix for the code.

## Data Plane TX

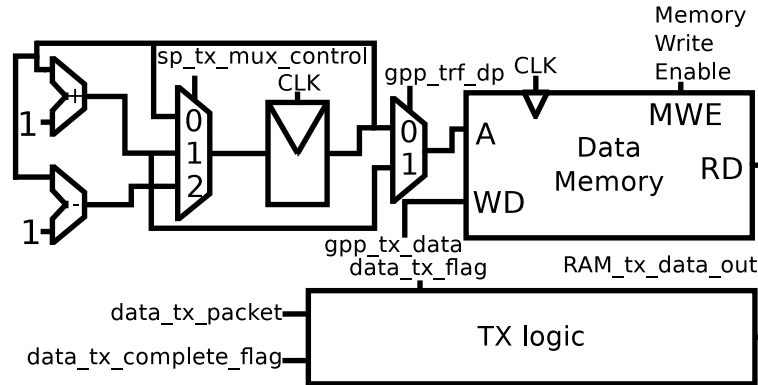


Figure 4.4: Microarchitecture of data plane transmitter

Variable name	Explanation	Bit-width
CLK	The clock for the system	1
gpp_trf_dp	This is a control signal which tells the data plane transmitter that data is going to be transferred from the GPP to the data plane transmitter	1
gpp_tx_data	This is the data that will be transferred from the GPP to the data plane transmitter RAM	16
data_tx_flag	This is a flag which indicates if the node is transmitting data on the data plane	1
data_tx_complete_flag	This flag will reset the data_tx_flag. The value for this is sent from the data plane transmitter	1
data_tx_packet	This is the packet that is transmitted on the data plane	32
RAM_tx_data_out	This input has the value equal to the top of the RAM of the data plane tx RAM	16

sp_tx_mux_control	This chooses the correct stack pointer value for the next clock cycle	2
-------------------	---	---

Table 4.5: Explanation of the variable names in Figure 4.4

Figure 4.4 shows the microarchitecture of the data plane transmitter. The TX logic component is a behavioural model that makes sure that only 5 packets are sent on the data plane. In addition the logic makes sure that a flag is set when the 5 packets are sent on the data plane. This allows the transmit flag from the control plane to be reset and send a ping if there is any data to be sent. The remaining components take care of controlling the stack pointer, this is when data is transferred by GPP or when data is sent on the data plane. Refer to Listing B.2 in the appendix for the code.

### Data Plane RX

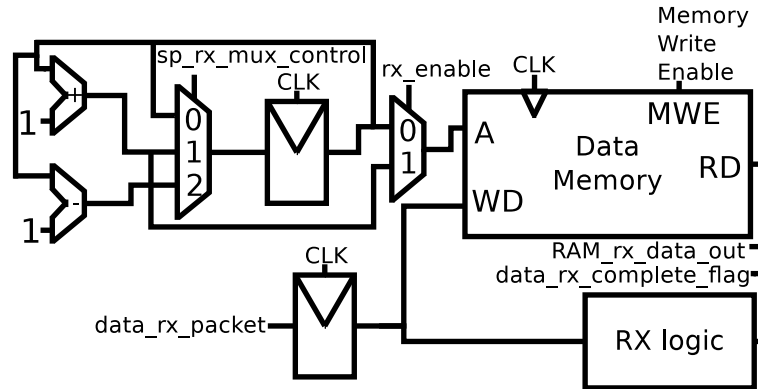


Figure 4.5: Microarchitecture of data plane receiver

Variable name	Explanation	Bit-width
CLK	The clock for the system	1
data_rx_packet	This is the packet that is received on the data plane	32

RAM_rx_data_out	This shows the data that is outputted from the data plane receiver. It will be connected to the RAM of the GPP via a multiplexer	16
data_rx_complete_flag	This flag will reset the data_rx_flag. The value for this is sent from the data plane receiver.	1
sp_rx_mux_control	This chooses the correct stack pointer value for the next clock cycle	2
rx_enable	This will chose the right stack pointer.	1

Table 4.6: Explanation of the variable names in Figure 4.5

Figure 4.5 shows the microarchitecture of the data plane receiver. The RX logic component is a behavioural model that makes sure that a flag is set once 5 packets are received. This allows the receive flag at the control plane to be reset and to accept a future ping. The remaining components take care of controlling the stack pointer, this is when data is retrieved by GPP or when data is received by the data plane receiver. Refer to Listing B.3 in the appendix for the code.

### Data Plane

This instantiates the data plane transmitter and receiver and creates the overall data plane system. Refer to Listing B.4 in the appendix for the code.

## 4.4 Microarchitecture of Microprocessor

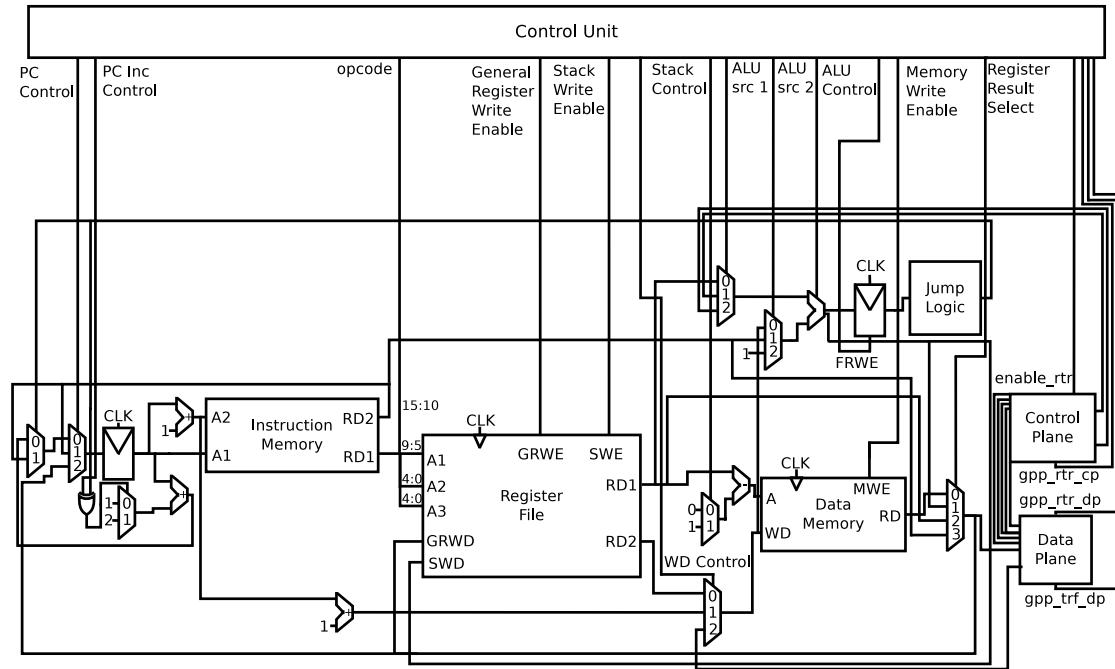


Figure 4.6: Microarchitecture for the specified instruction set

Figure 4.6 is an image of the microarchitecture for the microprocessor that was being designed for this project. It shows how the GPP and CP will be connected. Refer to Listing B.6 in the appendix for the top-level code.

## 4.5 Testing

### Simulation

ModelSim is used to carry out simulations by writing different testbenches.

### Low-Level Modules

The lower-level modules were tested by writing testbenches. The testbench makes sure if the module works as designed. Since these modules are quite simple they were tested for all possible inputs and the testbenches were designed to be self-checking with testvectors. This makes it easier to test the module as no manual checking had to be done on the results produced. The only exception to this



method of testing were the datapath and all the communications processor modules. For these modules a simple testbench was written which made the analysis of the results easier, like seeing if flags were set at appropriate times.

## GPP

To test the GPP a simple test bench was written. This testbench simply instantiated the GPP and reset it. Refer to Listing C.1 in the appendix for the testbench. When the GPP was simulated different assembly programs were written. Initially simple programs were written to check if the instructions work, like loops and conditional jumps. From this slightly more complicated programs were written and simulated. Below are the key programs that were simulated (complexity increases down the list):

- Use the move instruction to go through the Fibonacci sequence. Refer to Listing D.1 in the appendix for the code.
- Use the stack instructions to go through the Fibonacci sequence. Refer to Listing D.2 in the appendix for the code.
- Division - Instruction does not exist, therefore, a software implementation. Both the quotient and remainder will be saved in separate registers. Simulation was done twice. Once remainder is zero and once when remainder is not zero. Refer to Listing D.3 in the appendix for the code.
- Check if a number is a prime number or not. Refer to Listing D.4 in the appendix for the code.

**Note** Adding zero to the register that contains the result allows the result to be viewed on the timing diagram, provided one of the signals is the output of the GPP's ALU. The division and prime number algorithms have more instructions to execute. Therefore, by placing this add instruction in a loop and running the simulation to a later time, the results can easily be seen as they will always be repeating, provided the main algorithm has finished executing.

## CP

To test the CP two simple testbenches were written. One testbench was for checking the transmitter and the other was to check the receiver.

The transmitter testbench instantiated the CP. Default environment values like node id and maximum number of nodes in the network were set and the device was reset. A packet was transferred every clock cycle for 5 clock cycles. This was

to put data in the transmitter's RAM and to simulate transfer of data from GPP to CP. A delay was added to give the processor the opportunity to ping the node. The response to the ping was hard-coded in the testbench since the processor alone was simulated. From this response the processor either should send the packets or not. Refer to Listing C.2 in the appendix for the testbench.

The receiver testbench instantiated the CP. Default environment values like node id and maximum number of nodes in the network were set and the device was reset. A packet was hard-coded so that it was received on the control plane which acted like a ping. Then control signals were hard-coded to see if the data can be retrieved. This allowed the simulation of retrieving data from CP to GPP. Refer to Listing C.3 in the appendix for the code.

## **Microprocessor**

To test the microprocessor two simple testbenches were written. One testbench was for checking the transmitter and the other was to check the receiver.

The transmitter testbench instantiated the microprocessor. Default environment values like node id and maximum number of nodes in the network were set and the device was reset. An assembly program was written which pushed data onto the GPP's RAM and transfer that data to the transmitter's RAM. From this a ping was sent. The response to the ping was hard-coded in the testbench since the processor alone was simulated. From this response the processor either should send the packets or not. Refer to Listing C.4 and Listing D.5 in the appendix for the testbench and the assembly code respectively.

The receiver testbench instantiated the microprocessor. Default environment values like node id and maximum number of nodes in the network were set and the device was reset. A packet was hard-coded so that it was received on the control plane which acted as a ping. This data was stored in the receiver's RAM. During this time a program was running on the GPP. The instructions that the GPP executes should be to create a delay. Once the packet was received the assembly code retrieves the data from the receiver's RAM to the GPP's RAM. Refer to Listing C.5 and Listing D.6 in the appendix for the testbench and the assembly code respectively.

## **Synthesis**

The synthesis tool that is available from the Quartus prime software was used to produce a database that will configure the FPGA.

## **Low-Level Modules**

All the low-level modules were synthesised first. The SystemVerilog file used during simulation was also used during synthesis. Doing this allowed any errors to be removed at the start before building the system up. The on-board switches and buttons were used as the inputs to the modules. The on-board LEDs were used as the outputs of the modules.

## **GPP**

The same top-level module used to simulate the GPP was used during synthesis. To test the GPP the same assembly programs that were used during simulation was used again. The clock and reset were the two inputs to the system and the on-board buttons were used to apply the inputs. The output of the system was to be connected to the ALU. This would represent the result from the algorithm. The outputs were connected to the on-board LEDs to view the result.

## **CP & Microprocessor**

The CP and the microprocessor was not synthesised and tested. This is due to the limited resources available on the DE0-Nano FPGA. The limited resources were the General Purpose Inputs and Outputs (GPIOs) and the dedicated RAM & ROM.

# Chapter 5

## Results and Analysis

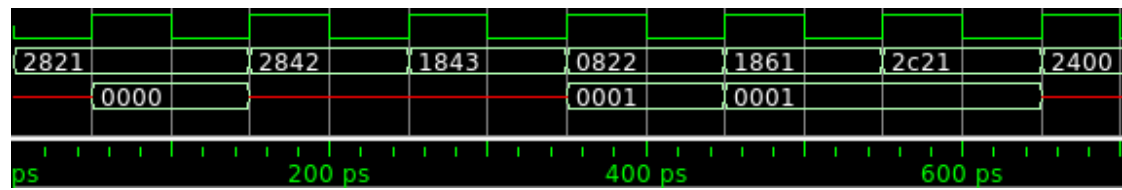
### 5.1 Results from Simulation

#### Lower-Level Modules

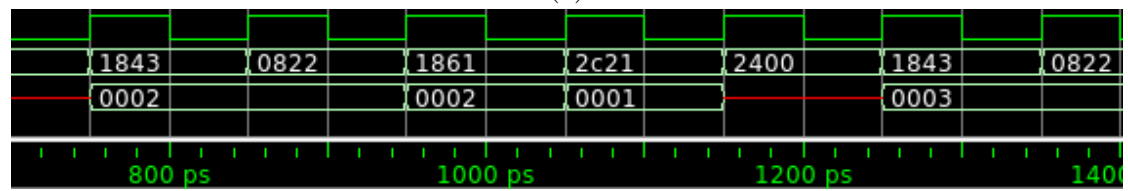
All the tests on the lower-level that were carried out were successful. The simulation results matched the expected behaviour.

# GPP

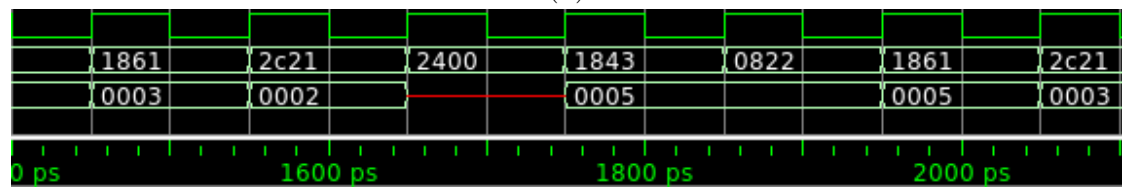
Fibonacci sequence - move instruction



(a)



(b)



(c)

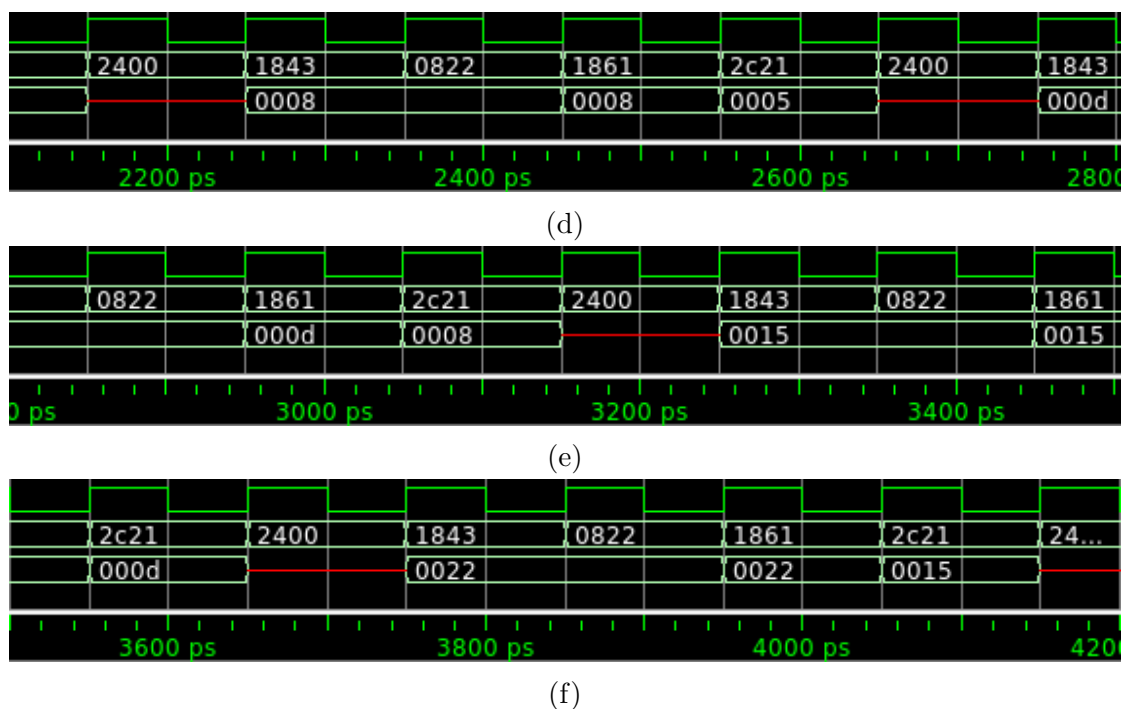
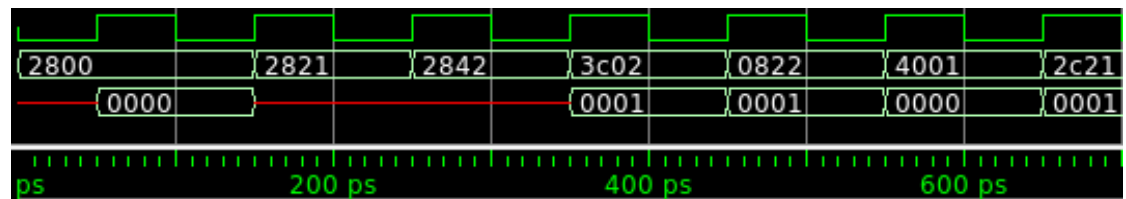


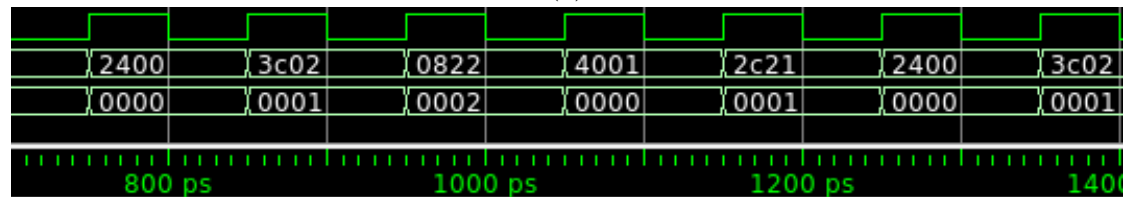
Figure 5.1: Shows the timing diagrams when executing the Fibonacci sequence program using the move instructions. Figure 5.1a is between 0 ps and 700ps. Figure 5.1b is between 700 ps and 1400ps. Figure 5.1c is between 1400 ps and 2100ps. Figure 5.1d is between 2100 ps and 2800ps. Figure 5.1e is between 2800 ps and 3500ps. Figure 5.1f is between 3500 ps and 4200ps.

Figure 5.1 shows the timing diagram when the GPP was executing the Fibonacci sequence algorithm using move instructions. The first signal is the clock signal. Second signal is the current instruction that is being executed in hexadecimal. The final signal is the output from the GPP's ALU in hexadecimal. In order to see if the GPP goes through the Fibonacci sequence it is important to remember that the add instruction allows the result to be viewed. The add instruction that was used in this simulation was `addi reg1, #0`. This particular instruction in machine code is 0010110000100001. This is 0x2C21 in hexadecimal. Whenever this instruction is seen then the output of the Fibonacci sequence is seen. Going through the timing diagram it is clear that the values are 0x0001 (1), 0x0001 (1), 0x0002 (2), 0x0003 (3), 0x0005 (5), 0x0008 (8), 0x000d (13) and 0x0015 (21). The results clearly indicate that the GPP was outputting the Fibonacci sequence.

## Fibonacci sequence - stack instructions



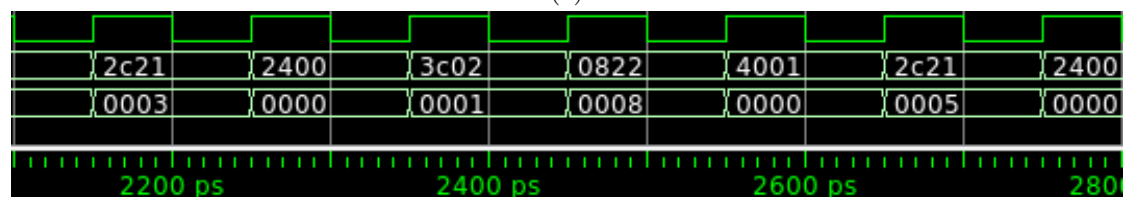
(a)



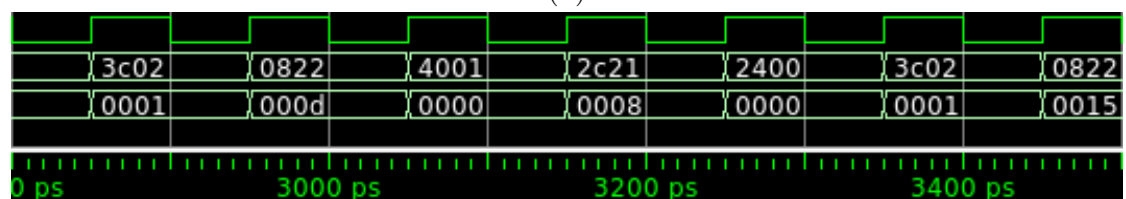
(b)



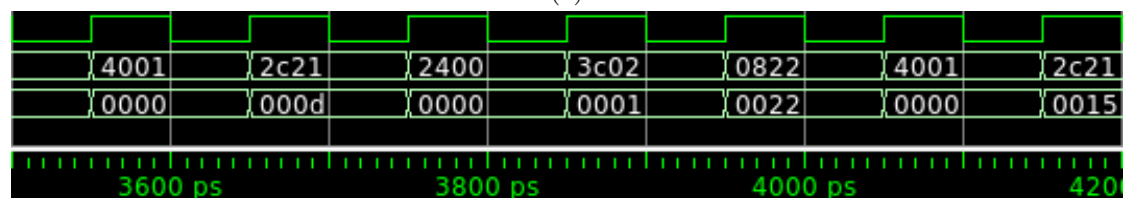
(c)



(d)



(e)



(f)

Figure 5.2: Shows the timing diagrams when executing the Fibonacci sequence program using the stack instructions. Figure 5.2a is between 0 ps and 700ps. Figure 5.2b is between 700 ps and 1400ps. Figure 5.2c is between 1400 ps and 2100ps. Figure 5.2d is between 2100 ps and 2800ps. Figure 5.2e is between 2800 ps and 3500ps. Figure 5.2f is between 3500 ps and 4200ps.

Figure 5.2 shows the timing diagram when the GPP was executing the Fibonacci sequence algorithm using stack instructions. As with the previous test, the first signal is the clock signal. Second signal is the current instruction that was being executed in hexadecimal. The final signal is the output from the GPP's ALU in hexadecimal. Using the add instruction that was used in this simulation, `addi reg1, #0`, the output from the GPP can be seen. This particular instruction in machine code is `0010110000100001` or `0x2C21` in hexadecimal. Whenever this instruction is seen then the output of the Fibonacci sequence can be observed. Going through the timing diagram it is clear that the values are `0x0001` (1), `0x0001` (1), `0x0002` (2), `0x0003` (3), `0x0005` (5), `0x0008` (8), `0x000d` (13) and `0x0015` (21). The results clearly indicate that the GPP was outputting the Fibonacci sequence.

## Division

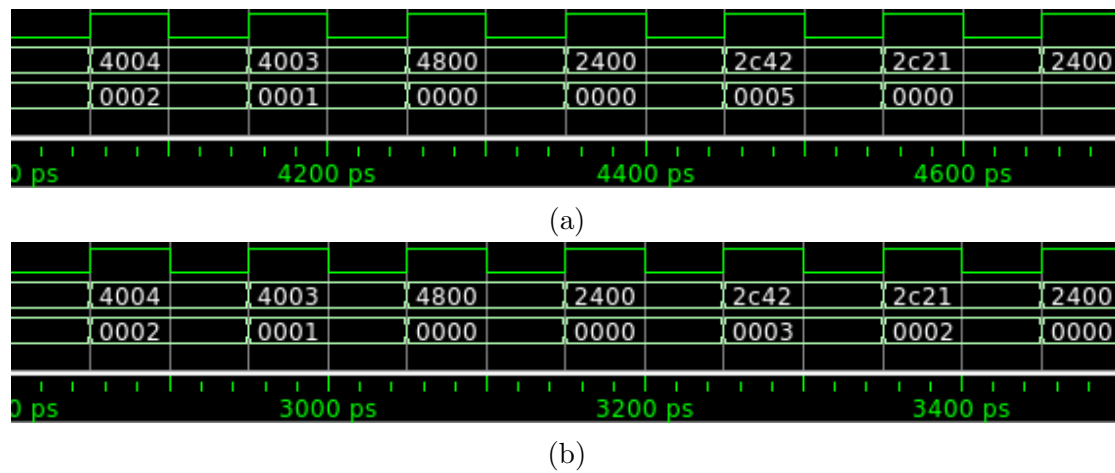


Figure 5.3: Shows the timing diagrams when executing the division algorithm. Figure 5.3a shows the case when the dividend is 10 and divisor is 2. Figure 5.3b shows the case when the dividend is 11 and divisor is 3

Figure 5.3 shows the timing diagram when the GPP was executing the division algorithm with two sets of dividends and divisors. As with the previous test, the first signal is the clock signal. The second signal is the current instruction that was being executed in hexadecimal. The final signal is the output from the GPP's ALU in hexadecimal. The timing diagram shown is not of the entire simulation. This is because before this time, the division algorithm was executing. Therefore, the results were not produced. This is not particularly useful but only useful when debugging to see where the error could have been. The division algorithm produces two results, quotient and remainder, which are saved in two separate registers. The



quotient is in reg2 and the remainder is in reg1. Therefore, to display these values two add instructions were used, which are add reg2, #0 and add reg1, #0. The two instructions in machine code are 0010110001000010 and 0010110000100001 or 0x2C42 and 0x2C21 in hexadecimal respectively. Whenever these instructions are seen the results from the division are seen. Looking at the timing diagram shown in Figure 5.3a it is clear that the quotient is 5 and remainder is 0. This is correct since ten divided by two gives a quotient 5 and a remainder 0. Also looking at the timing diagram shown in Figure 5.3b it is clear that the quotient is 3 and remainder is 2. This is also correct since eleven divided by three gives a quotient 3 and a remainder 2.

### Prime Number

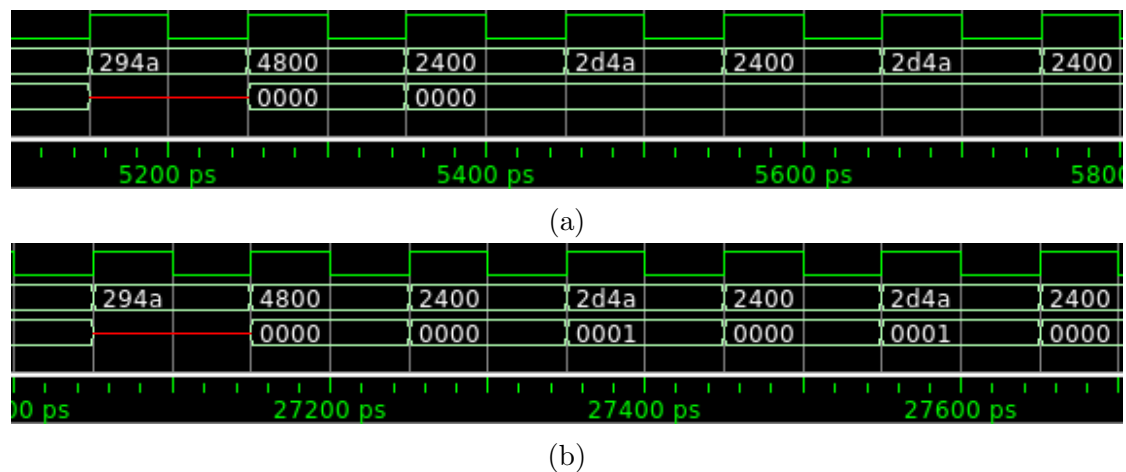


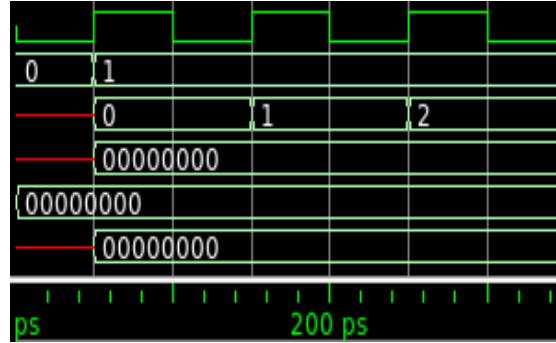
Figure 5.4: Shows the timing diagrams when executing the prime number algorithm. Figure 5.4a shows the case when checking if 10 is a prime number or not. Figure 5.4b shows the case when checking if 11 is a prime number or not

Figure 5.4 shows the timing diagram when the GPP was executing the prime number algorithm for two different cases. The first case is to check if 10 is a prime number or not and the second case is to check if 11 is a prime number or not. As with the previous test, the first signal is the clock signal. Second signal is the current instruction that is being executed in hexadecimal. The final signal is the output from the GPP's ALU in hexadecimal. Like before, the timing diagram shown is not of the entire simulation. The result was saved in a register as a flag. A 0 indicated that it was not a prime number and a 1 indicated that it was a prime number. Therefore, to display this value an add instruction was used add reg10, #0. This instruction in machine code is 0010110101001010 or 0x2D4A in

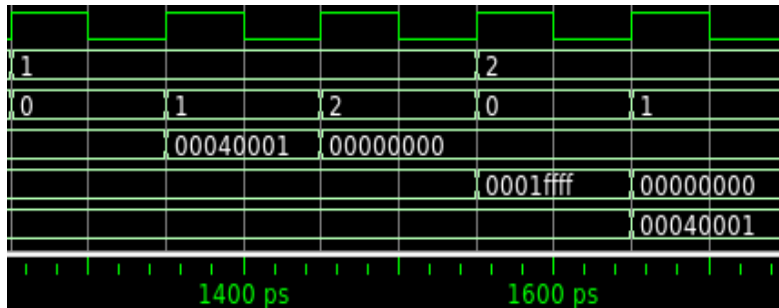
hexadecimal. Whenever this instruction is seen the result - as to whether the number is a prime number or not - is shown. It can be observed from the timing diagram shown in Figure 5.4a that it indicates that the value is 0. Therefore, stating that it is not a prime number, which is true for the number 10. Also looking at the timing diagram shown in Figure 5.4b it indicates that the value is 1. Therefore, stating that it is a prime number, which is true for the number 11.

## Communications Processor

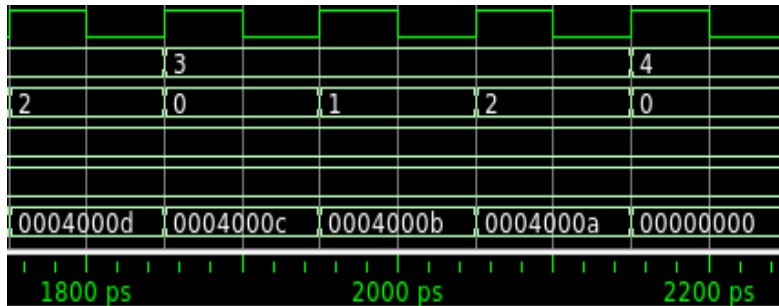
### Transmitter



(a)



(b)



(c)

Figure 5.5: Timing diagrams when communications processor works as a transmitter. Figure 5.5a shows the case when the transmitter's RAM is empty. Figure 5.5b and Figure 5.5c shows the case when the transmitter's RAM has contents. Figure 5.5b shows the beginning of the simulation in a specific scenario and Figure 5.5c shows the continuation of the simulation in the same scenario

Figure 5.5 shows the timing diagrams when the communications processor is controlling the data transmitter. The signals that are shown in the diagrams are as

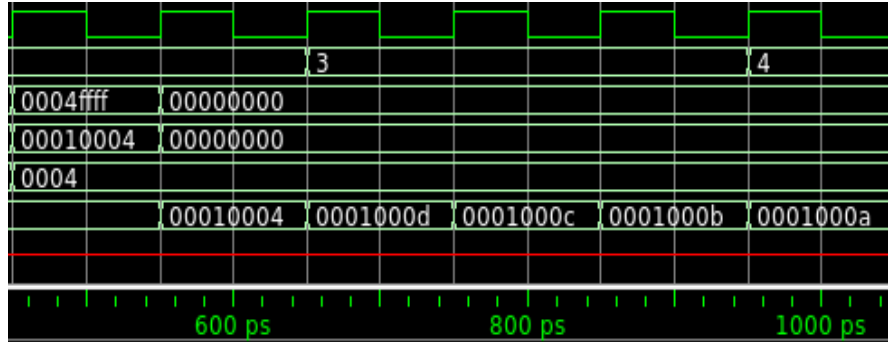
follows in the following order:

- Clock
- Slot
- Count - this indicates the number of clock cycles it has been in a certain slot
- Packet transmitted on the control plane
- Packet received on the control plane
- Packet transmitted on the data plane

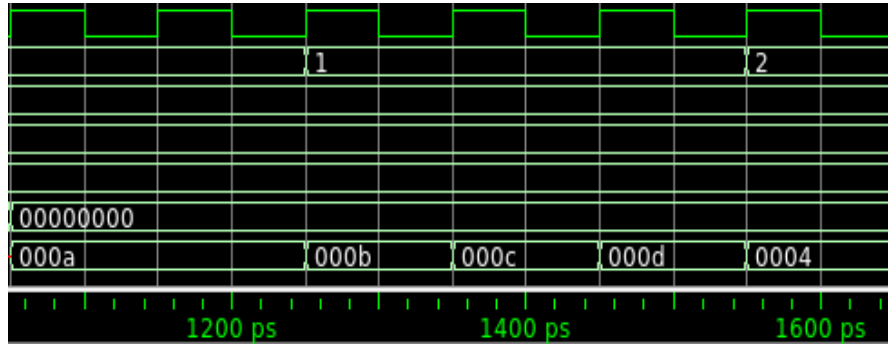
The node id was set to 1 and the maximum number of nodes in the network is set to 4. Figure 5.5a shows the timing diagram as soon as the communications processor has booted. From this timing diagram, it can be seen that the packet that was transmitted on the control plane by node 1 (this node) was 32x00000000. This is correct as there was no data in the transmitter's RAM. Hence there is no need to ping a node.

Figure 5.5b and Figure 5.5c are timing diagrams after data had been transferred to the transmitter's RAM. Since the slot and node id have been the same at the start, the node had to wait for its turn to ping a node. These diagrams begin when the slot is equal to the node id once again. Since the slot is equal to the node id and data was available to send, a packet should be sent on the control plane pinging the destination node. It can be seen from the timing diagram that a packet was sent on the control plane which was 0x00040001. The ping response that should have come from the destination node was hard-coded in the testbench. This allowed the node under test to receive a ping response. This packet was 0x0001FFFF. Since the LSBs are 0xFFFF, data can be sent on the data plane. Therefore, on the data plane a packet should be transmitted every clock cycle for 5 clock cycles. Looking at Figure 5.5b and Figure 5.5c the last signal clearly indicates this taking place. After this no data is sent on the data plane.

## Receiver



(a)



(b)

Figure 5.6: Timing diagrams when communications processor works as a receiver. Figure 5.6a shows the case when the receiver receives data. Figure 5.6b shows the simulation of data being retrieved

Figure 5.6 shows the timing diagrams when the communications processor is used as a receiver. The signals that are shown in the diagrams are as follows in the following order:

- Clock
- Slot
- Packet transmitted on the control plane
- Packet received on the control plane
- 16-bit value used to tune the data plane receiver to select a channel

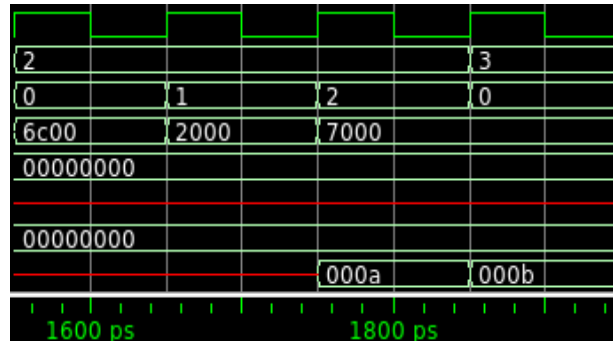
- Packet received on the data plane
- Data transferred from from CP to GPP

The node id was set to 1 and the maximum number of nodes in the network is set to 4. Figure 5.6a shows the timing diagram as soon as the node receives a ping. Since the testbench was designed such that no other ping took place before this, data will not be received on the data plane. Thus the LSBs of the ping response will be 0xFFFF. Along with this ping response a 16-bit value was set to the source node id so that the receiver can tune into that channel. These two values are seen in the timing diagram shown in Figure 5.6a. After this a packet should be received every clock cycle for 5 clock cycles where the 16 MSBs of the packet correspond to the node id. This also seen in the timing diagrams shown in Figure 5.6a.

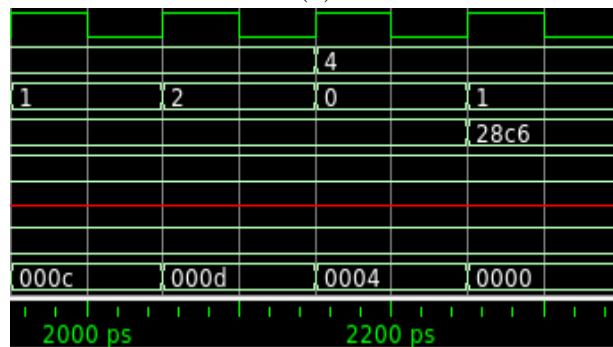
The next section describes the simulated transfer of data from CP to GPP. The appropriate control signals were set in the testbench. If these control signals are applied for 5 clock cycles all the data will be transferred to the GPP. This was clearly observed in the timing diagram shown in Figure 5.6b. The last signal shows the output of the RAM. The data plane receiver stores the data in a stack data structure. Thus the first packet that was transferred is the last packet that was received. The last packet that was transferred was the first packet that was received.

# Microprocessor

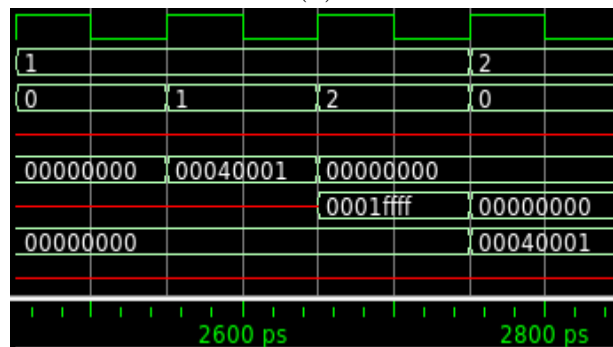
## Transmitter



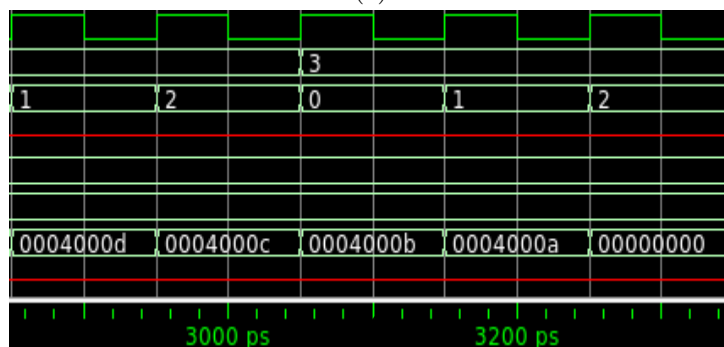
(a)



(b)



(c)



(d)

Figure 5.7: Timing diagrams when microprocessor works as a transmitter. Figure 5.7a and Figure 5.7b shows the case when data was being transferred from the GPP to the CP. Figure 5.7a shows the beginning of the simulation in a specific scenario and Figure 5.7b shows the continuation of the simulation in the same scenario. Figure 5.7c and Figure 5.7d shows the data being transmitted on the data plane as there is data to be sent. Figure 5.7c shows the beginning of the simulation in a specific scenario and Figure 5.7d shows the continuation of the simulation in the same scenario.

Figure 5.7 shows the timing diagrams when the microprocessor was used as a transmitter. The signals that are shown in the diagrams are as follows in the following order:

- Clock
- Slot
- Count - this indicates the number of clock cycles it has been in a certain slot
- Current instruction that is being executed in hexadecimal
- Packet transmitted on the control plane
- Packet received on the control plane
- Packet transmitted on the data plane
- Data transferred from GPP to CP

The node id is set to 1 and the maximum number of nodes in the network is set to 4.

Figure 5.7a and Figure 5.7b shows data process of data being transferred from GPP to CP. As discussed in subsection 4.3.2 the same procedure was carried out and the code specified in Listing D.5 was executed on the GPP. Therefore, before transferring the following two instructions were carried out, `cbt` and `jz #label`. The two instructions in machine code are 0110110000000000 and 0010000000000000 or 0x6C00 and 0x2000 in hexadecimal respectively. This was seen in Figure 5.7a where both instructions were carried out.

The simulation was set up such that no data was being transmitted on the data plane and the current slot was not equal to the node id. Thus data can be transferred from the GPP to CP as these were the conditions required for data transfer.

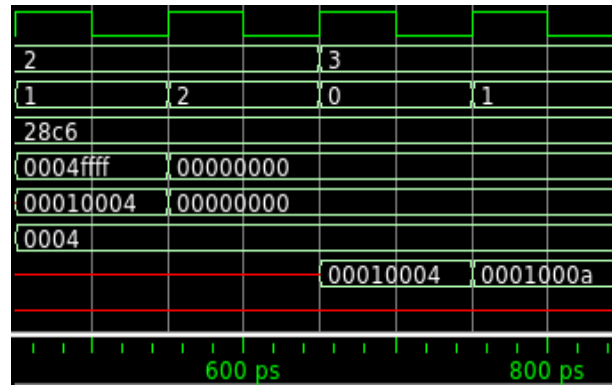


Figure 5.7a indicates that the current slot is equal to 2 and the packet that was transmitted on the data plane is 0x0000 (which also means no data transfer). The instruction to transfer data is trf which in machine code is 0111000000000000 or 0x7000 in hexadecimal. In Figure 5.7a it is clear that this instruction was executed at 1750ps. From this time a 16-bit value is transferred from the GPP to the CP every clock cycle for 5 clock cycles. The last signal in both Figure 5.7a and Figure 5.7b shows the data being transferred.

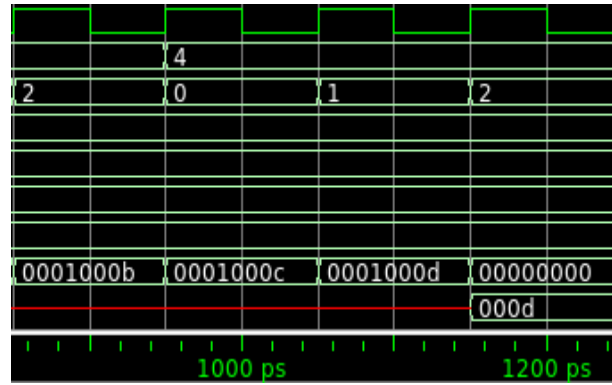
Figure 5.7c and Figure 5.7d are timing diagrams after data has been transferred to the transmitter's RAM. Since the slot and node id have been the same at the start, the node had to wait for its turn to ping a node. These diagrams begin when the slot is equal to the node id once again. Since the slot is equal to the node id and data was available to send, a packet should be sent on the control plane pinging the destination node. Looking at the diagram a packet was sent on the control plane which was 0x00040001.

The ping response was received by this node on the control plane which was hard-coded in the testbench. This packet was 0x0001FFFF. Since the LSBs are 0xFFFF, data can be sent on the data plane. Therefore, on the data plane a packet should be transmitted every clock cycle for 5 clock cycles. From Figure 5.7c and Figure 5.7d it can be observed that the last signal clearly indicates this taking place. After this no data is sent on the data plane.

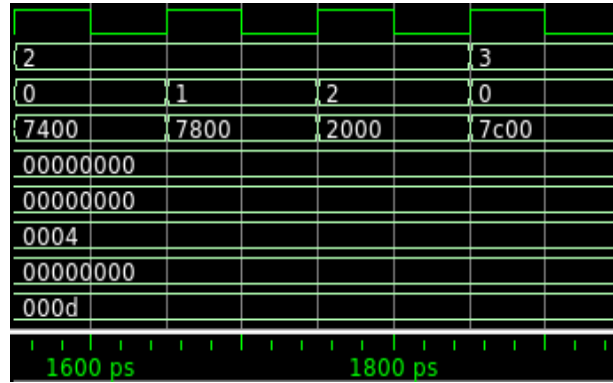
## Receiver



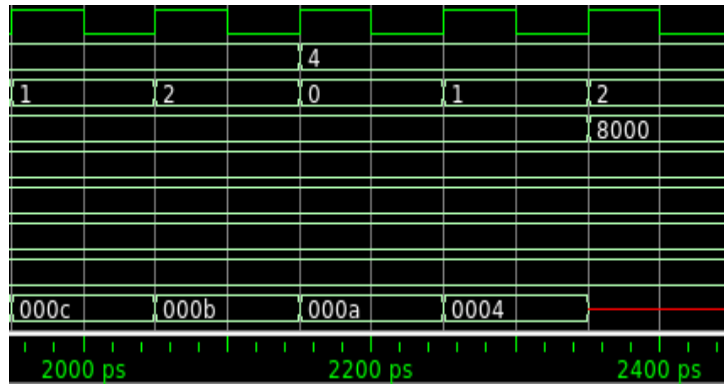
(a)



(b)



(c)



(d)

Figure 5.8: Timing diagrams when microprocessor works as a transmitter. Figure 5.8a and Figure 5.8b shows the case when data was being transferred from the GPP to the CP. Figure 5.8a shows the beginning of the simulation in a specific scenario and Figure 5.8b shows the continuation of the simulation in the same scenario. Figure 5.8c and Figure 5.8d shows the data being transmitted on the data plane as there is data to be sent. Figure 5.8c shows the beginning of the simulation in a specific scenario and Figure 5.8d shows the continuation of the simulation in the same scenario.

Figure 5.8 shows the timing diagrams when the microprocessor was used as a receiver. The signals that are shown in the diagrams are as follows in the following order:

- Clock
- Slot
- Count - this indicates the number of clock cycles it has been in a certain slot

- Current instruction that is being executed in hexadecimal
- Packet transmitted on the control plane
- Packet received on the control plane
- 16-bit value used to tune the data plane receiver to select a channel
- Packet received on the data plane
- Data transferred from CP to GPP

The node id was set to 1 and the maximum number of nodes in the network was set to 4.

Figure 5.8a and Figure 5.8b shows the timing diagram as soon as the node receives a ping. Since the testbench was designed such that no other ping took place before this, data will not be received on the data plane. Thus the LSBs of the ping response will be 0xFFFF. Along with this ping response a 16-bit value is set to the source node id so that the receiver can tune into that channel. These two values are seen in the timing diagram shown in Figure 5.8a. Following this, a packet should be received every clock cycle for 5 clock cycles where the 16 MSBs of the packet correspond to the node id. This also seen in the timing diagrams shown in Figure 5.8a and Figure 5.8b.

Figure 5.8c and Figure 5.8d shows data process of data being retrieved from CP to GPP. As discussed in subsection 4.3.3 the same procedure was carried out and the code specified in Listing D.6 is executed on the GPP. Therefore, before retrieving the data the following three instructions were carried out, pr, cbr and jz #label. The three instructions in machine code are 011101000000000000 , 0111100000000000 and 0010000000000000 or 0x7400, 0x7800 and 0x2000 in hexadecimal, respectively. This is seen in Figure 5.8c where these instructions were carried out. Since the simulation was set up such that no data is being received on the data plane, data can be retrieved from the CP to GPP. The instruction to retrieve data is rtr which in machine code is 0111110000000000 or 0x7C00 in hexadecimal. In Figure 5.7c it is clear that this instruction was executed at 1850ps. From this time a 16-bit value is transferred from the CP to the GPP every clock cycle for 5 clock cycles. The last signal in both Figure 5.8c and Figure 5.8d shows the data being transferred.

## 5.2 Synthesis

All the experimental tests that were carried out on the low-level modules and GPP were 100% in agreement with the behavioural simulation, which were in turn fully

compliant with the design specifications.

# Chapter 6

## Conclusion

The project was focused on the design of a novel microprocessor for data centre and high performance computing applications. It was designed to perform compute tasks and communicate with other nodes in an optical network designed to overcome the issues faced by current electronic switches. The nodes employ wavelength division multiplexing and space division multiplexing to achieve high link capacities and all-optical signal routing between nodes, enabling high throughput and low latency communications. Combining both technologies, each node can be assigned a dedicated SWDM channel on which to transmit. A broadcast-and-select network protocol was adopted due to the low complexity and avoidance of the congestion occurring at routing nodes in networks with shared channels. A ring topology was chosen as the network topology as it can be bi-directional, allowing protection switching in the case of link failures. A Reduced Instruction Set Computer design approach was adopted for the processor architecture as it was easier and quicker to implement when compared to the Complex Instruction Set Computer approach, and it also has the advantage of being more power efficient. The use of dedicated general purpose processor (for compute tasks) and communications processor to handle communications with other nodes was chosen as the node architecture, as this was considered to be more efficient than using a single processor with interrupts to handle the communications system. The latter method would lead to clock cycles being wasted, impacting the speed at which the main compute programs run.

The microarchitectures of the GPP and CP were implemented in SystemVerilog. This system was verified and tested in ModelSim. This was done by testing the low-level modules using testbenches. The top-level module for the GPP was tested by writing a number of assembly programs and confirming that the correct result was computed. The top-level module for the CP was tested by writing testbenches that included hard-coded values to check whether data could be correctly trans-

mitted and received. The node (GPP and CP together) was tested for transmitting and receiving data separately. In each case there was a separate assembly code and testbench with hard-coded values in the testbench. The tests also included confirming that data can be transferred from GPP and CP or retrieved from CP to GPP. All the tests carried out in ModelSim were successful. The GPP was implemented experimentally and tested on the DE0-Nano field programmable gate array. The same assembly programs as used in the simulations were used to experimentally test the GPP.

Future work on this project could include MATLAB simulations to investigate the performance of large-scale SWDM broadcast-and-select network operation using the novel processor designs developed in this project, and the implementation of multiple nodes, using FPGAs, and the construction and testing of a large scale network.

# Bibliography

- [1] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, “Osa: An optical switching architecture for data center networks with unprecedented flexibility,” *IEEE/ACM Transactions on Networking*, vol. 22, no. 2, pp. 498–511, 2013.
- [2] A. Celik, A. Al-Ghadhban, B. Shihada, and M.-S. Alouini, “Design and provisioning of optical wireless data center networks: A traffic grooming approach,” in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2018, pp. 1–6.
- [3] M. Xu, J. Diakonikolas, E. Modiano, and S. Subramaniam, “A hierarchical wdm-based scalable data center network architecture,” *arXiv preprint arXiv:1901.06450*, 2019.
- [4] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.
- [5] M. A. Taubenblatt, “Optical interconnects for high-performance computing,” *Journal of Lightwave Technology*, vol. 30, no. 4, pp. 448–457, 2011.
- [6] S. Yu, B. Guo, W. Li, Y. Zhou, X. Li, and S. Huang, “Optical circuit switching enabled reconfigurable hpc network for traffic pattern,” in *Optical Fiber Communication Conference*. Optical Society of America, 2018, pp. W4I–5.
- [7] K. Ishii and S. Namiki, “Toward exa-scale photonic switch system for the future datacenter,” in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2016, pp. 1–5.
- [8] R. Proietti, Z. Cao, C. J. Nitta, Y. Li, and S. B. Yoo, “A scalable, low-latency, high-throughput, optical interconnect architecture based on arrayed waveguide grating routers,” *Journal of Lightwave Technology*, vol. 33, no. 4, pp. 911–920, 2015.



- [9] D. Sadot and E. Boimovich, "Tunable optical filters for dense wdm networks," *IEEE Communications Magazine*, vol. 36, no. 12, pp. 50–55, 1998.
- [10] D. K. Tyagi, V. Chaubey, and P. Khandelwal, "Routing and wavelength assignment in wdm network using iwd based algorithm," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2016, pp. 1424–1429.
- [11] D. B. A. Teixeira, C. T. Batista, A. J. F. Cardoso, and J. d. S. Araújo, "A genetic algorithm approach for static routing and wavelength assignment in all-optical wdm networks," in *EPIA Conference on Artificial Intelligence*. Springer, 2017, pp. 421–432.
- [12] G. I. Papadimitriou and D. G. Maritsas, "Learning automata-based receiver conflict avoidance algorithms for wdm broadcast-and-select star networks," *IEEE/ACM transactions on networking*, vol. 4, no. 3, pp. 407–412, 1996.
- [13] G. Papadimitriou and D. Maritsas, "Self-adaptive random-access protocols for wdm passive star networks," *IEE Proceedings-Computers and Digital Techniques*, vol. 142, no. 4, pp. 306–312, 1995.
- [14] G. I. Papadimitriou and D. G. Maritsas, "Wdm passive star networks: a learning automata-based architecture," *Computer Communications*, vol. 19, no. 6-7, pp. 580–589, 1996.
- [15] M. B. I. Reaz, M. S. Islam, and M. S. Sulaiman, "A single clock cycle mips risc processor design using vhdl," in *ICONIP'02. Proceedings of the 9th International Conference on Neural Information Processing. Computational Intelligence for the E-Age (IEEE Cat. No. 02EX575)*. IEEE, 2002, pp. 199–203.
- [16] D. K. Dennis, A. Priyam, S. S. Virk, S. Agrawal, T. Sharma, A. Mondal, and K. C. Ray, "Single cycle risc-v micro architecture processor and its fpga prototype," in *2017 7th International Symposium on Embedded Computing and System Design (ISED)*. IEEE, 2017, pp. 1–5.
- [17] R. Ramaswami, "Multiwavelength lightwave networks for computer communication," *IEEE communications magazine*, vol. 31, no. 2, pp. 78–88, 1993.
- [18] H. Zang, J. P. Jue, B. Mukherjee *et al.*, "A review of routing and wavelength assignment approaches for wavelength-routed optical wdm networks," *Optical networks magazine*, vol. 1, no. 1, pp. 47–60, 2000.
- [19] J. Tippinit and W. Asawamethapant, " $N \times n$  cyclic awg with low and uniform insertion losses achieved by introducing array of tapered waveguides,"

- in *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE, 2015, pp. 1–6.
- [20] M. J. Spencer and M. A. Summerfield, “Wrap: A medium access control protocol for wavelength-routed passive optical networks,” *Journal of Lightwave Technology*, vol. 18, no. 12, pp. 1657–1676, 2000.
- [21] D. M. Harris and S. Harris, *Digital design and computer architecture*, 2013.

# Appendices

# Appendix A

## GPP

### ALU.sv

```
1  /*
    ↪ *****
    ↪
2  * File name
3    ALU.sv
4  * Description
5    This code will create an ALU which can take two inputs and add/sub/
6    and/or them.
7  * Parameters
8    WIDTH – This is the width of the input and output data.
9  * Inputs
10    src_a – 1st input
11    src_b – 2nd input
12    ALU_control – Control signal which determines which operation to carry out (add/sub/
    ↪ and/or)
13 * Outputs
14    ALU_out – The output result after applying the operation on the 2 inputs
15 * Author
16    Sreethyan Aravinthan (UCL)
17 *****/
18
19 module ALU
20 #(
21     parameter WIDTH = 1
22 )
23 (
24     input logic [WIDTH - 1:0]src_a,
25     input logic [WIDTH - 1:0]src_b,
26     input logic [1:0] ALU_Control,
27     output logic [WIDTH - 1:0]ALU_out
28 );
29
```

```

30 always_comb
31 begin
32     case(ALU_Control)
33         2'b00 : // add
34         begin
35             ALU_out = src_a + src_b;
36         end
37         2'b01 : // sub
38         begin
39             ALU_out = src_a - src_b;
40         end
41         2'b10 : // and
42         begin
43             ALU_out = src_a & src_b;
44         end
45         2'b11 : // or
46         begin
47             ALU_out = src_a | src_b;
48         end
49     endcase
50 end
51
52 endmodule

```

Listing A.1: ALU

### ALU\_advanced.sv

```

1  /*
   ↳ *****
   ↳
2  * File name
3      ALU_advanced.sv
4  * Description
5      This code will create an ALU which can take two inputs and add/sub/and/or them. The
   ↳ module set the following flags zero flag, carry flag, sign flag and overflow flag after the
   ↳ operation was carried out.
6  * Parameters
7      WIDTH – This is the width of the input and output data.
8  * Inputs
9      src_a – 1st input
10     src_b – 2nd input
11     ALU_control – Control signal which determines which operation to carry out (add/sub/
   ↳ and/or)
12 * Outputs
13     ALU_out – The output result after applying the operation on the 2 inputs
14     zero_flag – Indicates if the results was a zero
15     carry_flag – Indicates if the carry took place
16     sign_flag – Indicates if the Most Significant Bit (MSB) is set to 1
17     overflow_flag – Indicates if an overflow took place

```

```

18 * Author
19     Sreethyan Aravinthan (UCL)
20 *****/
21
22 module ALU_advanced
23 #(
24     parameter WIDTH = 1
25 )
26 (
27     input logic [WIDTH - 1:0]src_a,
28     input logic [WIDTH - 1:0]src_b,
29     input logic [1:0] ALU_Control,
30     output logic [WIDTH - 1:0]ALU_out,
31     output logic zero_flag,
32     output logic carry_flag,
33     output logic sign_flag,
34     output logic overflow_flag
35 );
36
37 // this holds the result after the operation, so that the carry flag can be set appropriately
38 logic [WIDTH:0] result;
39
40 // for all the operations set the zero and sign flag as follows
41 assign zero_flag = ALU_out == 0;
42 assign sign_flag = ALU_out[WIDTH - 1] == 1'b1;
43
44 always_comb
45 begin
46     case(ALU_Control)
47         2'b00 : // add
48             begin
49                 ALU_out = src_a + src_b;
50
51                 /**Setting the carry flag**/
52
53                 result = {1'b0, src_a} + {1'b0, src_b};
54                 carry_flag = result[WIDTH];
55
56                 /**Setting the overflow flag**/
57
58                 // if the two numbers are positive
59                 if(src_a[WIDTH - 1] == 1'b0 && src_b[WIDTH - 1] == 1'b0)
60                     begin
61                         // and if the result is negative
62                         if(ALU_out[WIDTH - 1] == 1'b1)
63                             begin
64                                 // set the overflow flag to 1
65                                 overflow_flag = 1'b1;
66                             end

```

```

67     else
68     begin
69         // set the overflow flaf to 0
70         overflow_flag = 1'b0;
71     end
72 end
73 // if the two numbers that are being added -ve
74 else if(src_a[WIDTH - 1] == 1'b1 && src_b[WIDTH - 1] == 1'b1)
75 begin
76     // the result is positive
77     if(ALU_out[WIDTH - 1] == 1'b0)
78     begin
79         // set overflow flag to 1
80         overflow_flag = 1'b1;
81     end
82     else
83     begin
84         // set overflow flag to 0
85         overflow_flag = 1'b0;
86     end
87 end
88 // if one is +ve and the other is -ve
89 else
90 begin
91     overflow_flag = 1'b0;
92 end
93 end
94 2'b01 : // sub
95 begin
96     ALU_out = src_a - src_b;
97
98     /**Setting the carry flag**/
99
100    result = {1'b0, src_a} - {1'b0, src_b};
101    carry_flag = result[WIDTH];
102
103    /**Setting the overflow flag**/
104
105    // if the value is +ve then -ve
106    if(src_a[WIDTH - 1] == 1'b0 && src_b[WIDTH - 1] == 1'b1)
107    begin
108        // and if the result is negative
109        if(ALU_out[WIDTH - 1] == 1'b1)
110        begin
111            // set the overflow flag to 1
112            overflow_flag = 1'b1;
113        end
114        else
115        begin

```

```

116         // set the overflow flaf to 0
117         overflow_flag = 1'b0;
118     end
119 end
120 // if the value is -ve then +ve
121 else if(src_a[WIDTH - 1] == 1'b1 && src_b[WIDTH - 1] == 1'b0)
122 begin
123     // the result is positive
124     if(ALU_out[WIDTH - 1] == 1'b0)
125     begin
126         // set overflow flag to 1
127         overflow_flag = 1'b1;
128     end
129     else
130     begin
131         // set overflow flag to 0
132         overflow_flag = 1'b0;
133     end
134 end
135 // if one is +ve and the other is -ve
136 else
137 begin
138     overflow_flag = 1'b0;
139 end
140 end
141 2'b10 : // and
142 begin
143     ALU_out = src_a & src_b;
144     result = 'b0;
145
146     /**Setting the carry flag**/
147     carry_flag = 1'bx;
148
149     /**Setting the overflow flag**/
150     overflow_flag = 1'bx;
151 end
152 2'b11 : // or
153 begin
154     ALU_out = src_a | src_b;
155     result = 'b0;
156
157     /**Setting the carry flag**/
158     carry_flag = 1'bx;
159
160     /**Setting the overflow flag**/
161     overflow_flag = 1'bx;
162 end
163 endcase
164 end

```



165 endmodule

Listing A.2: ALU\_advanced (ALU with addition features like flags)

### Control\_Unit.sv

```
1  /*
    ↪ *****
    ↪
2  * File name
3    Control_Unit.sv
4  * Description
5    This module takes the opcode and sets the values of the control signals.
6  * Parameters
7    NONE
8  * Inputs
9    opcode – This the 6 MSB of the instruction.
10 * Outputs
11    ALL THE CONTROL SIGNALS
12 * Author
13    Sreethyan Aravinthan (UCL)
14 *****/
15
16 module Control_Unit
17 (
18   input logic [5:0] opcode,
19   output logic pc_increment_control,
20   output logic [1:0] pc_control,
21   output logic general_register_write_enable,
22   output logic stack_write_enable,
23   output logic stack_control,
24   output logic [1:0] write_data_enable,
25   output logic [1:0] ALU_source_1,
26   output logic [1:0] ALU_source_2,
27   output logic [1:0] ALU_control,
28   output logic flags_write_enable,
29   output logic jump_zero_control,
30   output logic jump_below_control,
31   output logic jump_below_equal_control,
32   output logic jump_above_control,
33   output logic jump_above_equal_control,
34   output logic jump_greater_control,
35   output logic jump_greater_equal_control,
36   output logic jump_less_control,
37   output logic jump_less_equal_control,
38   output logic memory_write_enable,
39   output logic [1:0] general_register_result_select,
40   output logic enable_rtr,
41   output logic gpp_rtr_cp,
42   output logic gpp_rtr_dp,
```

```

43  output logic gpp_trf_dp
44  );
45  logic [30:0] controls;
46
47  always_comb
48  begin
49      pc_increment_control = controls[30];
50      pc_control = controls[29:28];
51      general_register_write_enable = controls[27];
52      stack_write_enable = controls[26];
53      stack_control = controls[25];
54      write_data_enable = controls[24:23];
55      ALU_source_1 = controls[22:21];
56      ALU_source_2 = controls[20:19];
57      ALU_control = controls[18:17];
58      flags_write_enable = controls[16];
59      jump_zero_control = controls[15];
60      jump_below_control = controls[14];
61      jump_below_equal_control = controls[13];
62      jump_above_control = controls[12];
63      jump_above_equal_control = controls[11];
64      jump_greater_control = controls[10];
65      jump_greater_equal_control = controls[9];
66      jump_less_control = controls[8];
67      jump_less_equal_control = controls[7];
68      memory_write_enable = controls[6];
69      general_register_result_select = controls[5:4];
70      enable_rtr = controls[3];
71      gpp_rtr_cp = controls[2];
72      gpp_rtr_dp = controls[1];
73      gpp_trf_dp = controls[0];
74  end
75
76  always_comb
77  begin
78      case(opcode)
79          6'b000000: // ld
80              begin
81                  controls[30] = 1'b0;
82                  controls[29:28] = 2'b00;
83                  controls[27] = 1'b1;
84                  controls[26] = 1'b0;
85                  controls[25] = 1'b0;
86                  controls[24:23] = 2'b00;
87                  controls[22:21] = 2'b00;
88                  controls[20:19] = 2'b00;
89                  controls[18:17] = 2'b00;
90                  controls[16] = 1'b0;
91                  controls[15] = 1'b0;

```

```

92     controls[14] = 1'b0;
93     controls[13] = 1'b0;
94     controls[12] = 1'b0;
95     controls[11] = 1'b0;
96     controls[10] = 1'b0;
97     controls[9] = 1'b0;
98     controls[8] = 1'b0;
99     controls[7] = 1'b0;
100    controls[6] = 1'b0;
101    controls[5:4] = 2'b00;
102    controls[3] = 1'b0;
103    controls[2] = 1'b0;
104    controls[1] = 1'b0;
105    controls[0] = 1'b0;
106    end
107
108    6'b000001: // str
109    begin
110        controls[30] = 1'b0;
111        controls[29:28] = 2'b00;
112        controls[27] = 1'b0;
113        controls[26] = 1'b0;
114        controls[25] = 1'b0;
115        controls[24:23] = 2'b00;
116        controls[22:21] = 2'b00;
117        controls[20:19] = 2'b00;
118        controls[18:17] = 2'b00;
119        controls[16] = 1'b0;
120        controls[15] = 1'b0;
121        controls[14] = 1'b0;
122        controls[13] = 1'b0;
123        controls[12] = 1'b0;
124        controls[11] = 1'b0;
125        controls[10] = 1'b0;
126        controls[9] = 1'b0;
127        controls[8] = 1'b0;
128        controls[7] = 1'b0;
129        controls[6] = 1'b1;
130        controls[5:4] = 2'b00;
131        controls[3] = 1'b0;
132        controls[2] = 1'b0;
133        controls[1] = 1'b0;
134        controls[0] = 1'b0;
135    end
136
137    6'b000010: // add
138    begin
139        controls[30] = 1'b0;
140        controls[29:28] = 2'b00;

```

```

141     controls[27] = 1'b1;
142     controls[26] = 1'b0;
143     controls[25] = 1'b0;
144     controls[24:23] = 2'b00;
145     controls[22:21] = 2'b00;
146     controls[20:19] = 2'b00;
147     controls[18:17] = 2'b00;
148     controls[16] = 1'b1;
149     controls[15] = 1'b0;
150     controls[14] = 1'b0;
151     controls[13] = 1'b0;
152     controls[12] = 1'b0;
153     controls[11] = 1'b0;
154     controls[10] = 1'b0;
155     controls[9] = 1'b0;
156     controls[8] = 1'b0;
157     controls[7] = 1'b0;
158     controls[6] = 1'b0;
159     controls[5:4] = 2'b01;
160     controls[3] = 1'b0;
161     controls[2] = 1'b0;
162     controls[1] = 1'b0;
163     controls[0] = 1'b0;
164 end
165
166 6'b000011: // sub
167 begin
168     controls[30] = 1'b0;
169     controls[29:28] = 2'b00;
170     controls[27] = 1'b1;
171     controls[26] = 1'b0;
172     controls[25] = 1'b0;
173     controls[24:23] = 2'b00;
174     controls[22:21] = 2'b00;
175     controls[20:19] = 2'b00;
176     controls[18:17] = 2'b01;
177     controls[16] = 1'b1;
178     controls[15] = 1'b0;
179     controls[14] = 1'b0;
180     controls[13] = 1'b0;
181     controls[12] = 1'b0;
182     controls[11] = 1'b0;
183     controls[10] = 1'b0;
184     controls[9] = 1'b0;
185     controls[8] = 1'b0;
186     controls[7] = 1'b0;
187     controls[6] = 1'b0;
188     controls[5:4] = 2'b01;
189     controls[3] = 1'b0;

```

```

190     controls[2] = 1'b0;
191     controls[1] = 1'b0;
192     controls[0] = 1'b0;
193 end
194
195 6'b000100: // and
196 begin
197     controls[30] = 1'b0;
198     controls[29:28] = 2'b00;
199     controls[27] = 1'b1;
200     controls[26] = 1'b0;
201     controls[25] = 1'b0;
202     controls[24:23] = 2'b00;
203     controls[22:21] = 2'b00;
204     controls[20:19] = 2'b00;
205     controls[18:17] = 2'b10;
206     controls[16] = 1'b1;
207     controls[15] = 1'b0;
208     controls[14] = 1'b0;
209     controls[13] = 1'b0;
210     controls[12] = 1'b0;
211     controls[11] = 1'b0;
212     controls[10] = 1'b0;
213     controls[9] = 1'b0;
214     controls[8] = 1'b0;
215     controls[7] = 1'b0;
216     controls[6] = 1'b0;
217     controls[5:4] = 2'b01;
218     controls[3] = 1'b0;
219     controls[2] = 1'b0;
220     controls[1] = 1'b0;
221     controls[0] = 1'b0;
222 end
223
224 6'b000101: // or
225 begin
226     controls[30] = 1'b0;
227     controls[29:28] = 2'b00;
228     controls[27] = 1'b1;
229     controls[26] = 1'b0;
230     controls[25] = 1'b0;
231     controls[24:23] = 2'b00;
232     controls[22:21] = 2'b00;
233     controls[20:19] = 2'b00;
234     controls[18:17] = 2'b11;
235     controls[16] = 1'b1;
236     controls[15] = 1'b0;
237     controls[14] = 1'b0;
238     controls[13] = 1'b0;

```

```

239     controls[12] = 1'b0;
240     controls[11] = 1'b0;
241     controls[10] = 1'b0;
242     controls[9] = 1'b0;
243     controls[8] = 1'b0;
244     controls[7] = 1'b0;
245     controls[6] = 1'b0;
246     controls[5:4] = 2'b01;
247     controls[3] = 1'b0;
248     controls[2] = 1'b0;
249     controls[1] = 1'b0;
250     controls[0] = 1'b0;
251 end
252
253 6'b000110: // mov
254 begin
255     controls[30] = 1'b0;
256     controls[29:28] = 2'b00;
257     controls[27] = 1'b1;
258     controls[26] = 1'b0;
259     controls[25] = 1'b0;
260     controls[24:23] = 2'b00;
261     controls[22:21] = 2'b00;
262     controls[20:19] = 2'b00;
263     controls[18:17] = 2'b00;
264     controls[16] = 1'b0;
265     controls[15] = 1'b0;
266     controls[14] = 1'b0;
267     controls[13] = 1'b0;
268     controls[12] = 1'b0;
269     controls[11] = 1'b0;
270     controls[10] = 1'b0;
271     controls[9] = 1'b0;
272     controls[8] = 1'b0;
273     controls[7] = 1'b0;
274     controls[6] = 1'b0;
275     controls[5:4] = 2'b10;
276     controls[3] = 1'b0;
277     controls[2] = 1'b0;
278     controls[1] = 1'b0;
279     controls[0] = 1'b0;
280 end
281
282 6'b000111: // cmp
283 begin
284     controls[30] = 1'b0;
285     controls[29:28] = 2'b00;
286     controls[27] = 1'b0;
287     controls[26] = 1'b0;

```

```

288     controls[25] = 1'b0;
289     controls[24:23] = 2'b00;
290     controls[22:21] = 2'b00;
291     controls[20:19] = 2'b00;
292     controls[18:17] = 2'b01;
293     controls[16] = 1'b1;
294     controls[15] = 1'b0;
295     controls[14] = 1'b0;
296     controls[13] = 1'b0;
297     controls[12] = 1'b0;
298     controls[11] = 1'b0;
299     controls[10] = 1'b0;
300     controls[9] = 1'b0;
301     controls[8] = 1'b0;
302     controls[7] = 1'b0;
303     controls[6] = 1'b0;
304     controls[5:4] = 2'b00;
305     controls[3] = 1'b0;
306     controls[2] = 1'b0;
307     controls[1] = 1'b0;
308     controls[0] = 1'b0;
309 end
310
311 6'b001000: // jz
312 begin
313     controls[30] = 1'b1;
314     controls[29:28] = 2'b00;
315     controls[27] = 1'b0;
316     controls[26] = 1'b0;
317     controls[25] = 1'b0;
318     controls[24:23] = 2'b00;
319     controls[22:21] = 2'b00;
320     controls[20:19] = 2'b00;
321     controls[18:17] = 2'b00;
322     controls[16] = 1'b0;
323     controls[15] = 1'b1;
324     controls[14] = 1'b0;
325     controls[13] = 1'b0;
326     controls[12] = 1'b0;
327     controls[11] = 1'b0;
328     controls[10] = 1'b0;
329     controls[9] = 1'b0;
330     controls[8] = 1'b0;
331     controls[7] = 1'b0;
332     controls[6] = 1'b0;
333     controls[5:4] = 2'b00;
334     controls[3] = 1'b0;
335     controls[2] = 1'b0;
336     controls[1] = 1'b0;

```

```

337     controls[0] = 1'b0;
338 end
339
340 6'b001001: // jmp
341 begin
342     controls[30] = 1'b0;
343     controls[29:28] = 2'b01;
344     controls[27] = 1'b0;
345     controls[26] = 1'b0;
346     controls[25] = 1'b0;
347     controls[24:23] = 2'b00;
348     controls[22:21] = 2'b00;
349     controls[20:19] = 2'b00;
350     controls[18:17] = 2'b00;
351     controls[16] = 1'b0;
352     controls[15] = 1'b0;
353     controls[14] = 1'b0;
354     controls[13] = 1'b0;
355     controls[12] = 1'b0;
356     controls[11] = 1'b0;
357     controls[10] = 1'b0;
358     controls[9] = 1'b0;
359     controls[8] = 1'b0;
360     controls[7] = 1'b0;
361     controls[6] = 1'b0;
362     controls[5:4] = 2'b00;
363     controls[3] = 1'b0;
364     controls[2] = 1'b0;
365     controls[1] = 1'b0;
366     controls[0] = 1'b0;
367 end
368
369 6'b001010: // movi
370 begin
371     controls[30] = 1'b1;
372     controls[29:28] = 2'b00;
373     controls[27] = 1'b1;
374     controls[26] = 1'b0;
375     controls[25] = 1'b0;
376     controls[24:23] = 2'b00;
377     controls[22:21] = 2'b00;
378     controls[20:19] = 2'b00;
379     controls[18:17] = 2'b00;
380     controls[16] = 1'b0;
381     controls[15] = 1'b0;
382     controls[14] = 1'b0;
383     controls[13] = 1'b0;
384     controls[12] = 1'b0;
385     controls[11] = 1'b0;

```



```

386     controls[10] = 1'b0;
387     controls[9] = 1'b0;
388     controls[8] = 1'b0;
389     controls[7] = 1'b0;
390     controls[6] = 1'b0;
391     controls[5:4] = 2'b11;
392     controls[3] = 1'b0;
393     controls[2] = 1'b0;
394     controls[1] = 1'b0;
395     controls[0] = 1'b0;
396 end
397
398 6'b001011: // addi
399 begin
400     controls[30] = 1'b1;
401     controls[29:28] = 2'b00;
402     controls[27] = 1'b1;
403     controls[26] = 1'b0;
404     controls[25] = 1'b0;
405     controls[24:23] = 2'b00;
406     controls[22:21] = 2'b00;
407     controls[20:19] = 2'b01;
408     controls[18:17] = 2'b00;
409     controls[16] = 1'b1;
410     controls[15] = 1'b0;
411     controls[14] = 1'b0;
412     controls[13] = 1'b0;
413     controls[12] = 1'b0;
414     controls[11] = 1'b0;
415     controls[10] = 1'b0;
416     controls[9] = 1'b0;
417     controls[8] = 1'b0;
418     controls[7] = 1'b0;
419     controls[6] = 1'b0;
420     controls[5:4] = 2'b01;
421     controls[3] = 1'b0;
422     controls[2] = 1'b0;
423     controls[1] = 1'b0;
424     controls[0] = 1'b0;
425 end
426
427 6'b001100: // subi
428 begin
429     controls[30] = 1'b1;
430     controls[29:28] = 2'b00;
431     controls[27] = 1'b1;
432     controls[26] = 1'b0;
433     controls[25] = 1'b0;
434     controls[24:23] = 2'b00;

```

```

435     controls[22:21] = 2'b00;
436     controls[20:19] = 2'b01;
437     controls[18:17] = 2'b01;
438     controls[16]  = 1'b1;
439     controls[15]  = 1'b0;
440     controls[14]  = 1'b0;
441     controls[13]  = 1'b0;
442     controls[12]  = 1'b0;
443     controls[11]  = 1'b0;
444     controls[10]  = 1'b0;
445     controls[9]   = 1'b0;
446     controls[8]   = 1'b0;
447     controls[7]   = 1'b0;
448     controls[6]   = 1'b0;
449     controls[5:4] = 2'b01;
450     controls[3]   = 1'b0;
451     controls[2]   = 1'b0;
452     controls[1]   = 1'b0;
453     controls[0]   = 1'b0;
454 end
455
456 6'b001101: // andi
457 begin
458     controls[30] = 1'b1;
459     controls[29:28] = 2'b00;
460     controls[27] = 1'b1;
461     controls[26] = 1'b0;
462     controls[25] = 1'b0;
463     controls[24:23] = 2'b00;
464     controls[22:21] = 2'b00;
465     controls[20:19] = 2'b01;
466     controls[18:17] = 2'b10;
467     controls[16] = 1'b1;
468     controls[15] = 1'b0;
469     controls[14] = 1'b0;
470     controls[13] = 1'b0;
471     controls[12] = 1'b0;
472     controls[11] = 1'b0;
473     controls[10] = 1'b0;
474     controls[9] = 1'b0;
475     controls[8] = 1'b0;
476     controls[7] = 1'b0;
477     controls[6] = 1'b0;
478     controls[5:4] = 2'b01;
479     controls[3] = 1'b0;
480     controls[2] = 1'b0;
481     controls[1] = 1'b0;
482     controls[0] = 1'b0;
483 end

```

```

484
485 6'b001110: // ori
486 begin
487     controls[30] = 1'b1;
488     controls[29:28] = 2'b00;
489     controls[27] = 1'b1;
490     controls[26] = 1'b0;
491     controls[25] = 1'b0;
492     controls[24:23] = 2'b00;
493     controls[22:21] = 2'b00;
494     controls[20:19] = 2'b01;
495     controls[18:17] = 2'b11;
496     controls[16] = 1'b1;
497     controls[15] = 1'b0;
498     controls[14] = 1'b0;
499     controls[13] = 1'b0;
500     controls[12] = 1'b0;
501     controls[11] = 1'b0;
502     controls[10] = 1'b0;
503     controls[9] = 1'b0;
504     controls[8] = 1'b0;
505     controls[7] = 1'b0;
506     controls[6] = 1'b0;
507     controls[5:4] = 2'b01;
508     controls[3] = 1'b0;
509     controls[2] = 1'b0;
510     controls[1] = 1'b0;
511     controls[0] = 1'b0;
512 end
513
514 6'b001111: // push
515 begin
516     controls[30] = 1'b0;
517     controls[29:28] = 2'b00;
518     controls[27] = 1'b0;
519     controls[26] = 1'b1;
520     controls[25] = 1'b0;
521     controls[24:23] = 2'b00;
522     controls[22:21] = 2'b00;
523     controls[20:19] = 2'b10;
524     controls[18:17] = 2'b00;
525     controls[16] = 1'b0;
526     controls[15] = 1'b0;
527     controls[14] = 1'b0;
528     controls[13] = 1'b0;
529     controls[12] = 1'b0;
530     controls[11] = 1'b0;
531     controls[10] = 1'b0;
532     controls[9] = 1'b0;

```

```

533     controls[8] = 1'b0;
534     controls[7] = 1'b0;
535     controls[6] = 1'b1;
536     controls[5:4] = 2'b00;
537     controls[3] = 1'b0;
538     controls[2] = 1'b0;
539     controls[1] = 1'b0;
540     controls[0] = 1'b0;
541 end
542
543 6'b010000: // pop
544 begin
545     controls[30] = 1'b0;
546     controls[29:28] = 2'b00;
547     controls[27] = 1'b1;
548     controls[26] = 1'b1;
549     controls[25] = 1'b1;
550     controls[24:23] = 2'b00;
551     controls[22:21] = 2'b00;
552     controls[20:19] = 2'b10;
553     controls[18:17] = 2'b01;
554     controls[16] = 1'b0;
555     controls[15] = 1'b0;
556     controls[14] = 1'b0;
557     controls[13] = 1'b0;
558     controls[12] = 1'b0;
559     controls[11] = 1'b0;
560     controls[10] = 1'b0;
561     controls[9] = 1'b0;
562     controls[8] = 1'b0;
563     controls[7] = 1'b0;
564     controls[6] = 1'b0;
565     controls[5:4] = 2'b00;
566     controls[3] = 1'b0;
567     controls[2] = 1'b0;
568     controls[1] = 1'b0;
569     controls[0] = 1'b0;
570 end
571
572 6'b010001: // call
573 begin
574     controls[30] = 1'b0;
575     controls[29:28] = 2'b01;
576     controls[27] = 1'b0;
577     controls[26] = 1'b1;
578     controls[25] = 1'b0;
579     controls[24:23] = 2'b01;
580     controls[22:21] = 2'b00;
581     controls[20:19] = 2'b10;

```

```

582     controls[18:17] = 2'b00;
583     controls[16]  = 1'b0;
584     controls[15]  = 1'b0;
585     controls[14]  = 1'b0;
586     controls[13]  = 1'b0;
587     controls[12]  = 1'b0;
588     controls[11]  = 1'b0;
589     controls[10]  = 1'b0;
590     controls[9]   = 1'b0;
591     controls[8]   = 1'b0;
592     controls[7]   = 1'b0;
593     controls[6]   = 1'b1;
594     controls[5:4] = 2'b00;
595     controls[3]   = 1'b0;
596     controls[2]   = 1'b0;
597     controls[1]   = 1'b0;
598     controls[0]   = 1'b0;
599 end
600
601 6'b010010: // return
602 begin
603     controls[30]  = 1'b0;
604     controls[29:28] = 2'b10;
605     controls[27]  = 1'b0;
606     controls[26]  = 1'b1;
607     controls[25]  = 1'b1;
608     controls[24:23] = 2'b00;
609     controls[22:21] = 2'b00;
610     controls[20:19] = 2'b10;
611     controls[18:17] = 2'b01;
612     controls[16]  = 1'b0;
613     controls[15]  = 1'b0;
614     controls[14]  = 1'b0;
615     controls[13]  = 1'b0;
616     controls[12]  = 1'b0;
617     controls[11]  = 1'b0;
618     controls[10]  = 1'b0;
619     controls[9]   = 1'b0;
620     controls[8]   = 1'b0;
621     controls[7]   = 1'b0;
622     controls[6]   = 1'b0;
623     controls[5:4] = 2'b00;
624     controls[3]   = 1'b0;
625     controls[2]   = 1'b0;
626     controls[1]   = 1'b0;
627     controls[0]   = 1'b0;
628 end
629
630 6'b010011: // jb

```

```

631 begin
632     controls[30] = 1'b1;
633     controls[29:28] = 2'b00;
634     controls[27] = 1'b0;
635     controls[26] = 1'b0;
636     controls[25] = 1'b0;
637     controls[24:23] = 2'b00;
638     controls[22:21] = 2'b00;
639     controls[20:19] = 2'b00;
640     controls[18:17] = 2'b00;
641     controls[16] = 1'b0;
642     controls[15] = 1'b0;
643     controls[14] = 1'b1;
644     controls[13] = 1'b0;
645     controls[12] = 1'b0;
646     controls[11] = 1'b0;
647     controls[10] = 1'b0;
648     controls[9] = 1'b0;
649     controls[8] = 1'b0;
650     controls[7] = 1'b0;
651     controls[6] = 1'b0;
652     controls[5:4] = 2'b00;
653     controls[3] = 1'b0;
654     controls[2] = 1'b0;
655     controls[1] = 1'b0;
656     controls[0] = 1'b0;
657 end
658
659 6'b010100: // jbe
660 begin
661     controls[30] = 1'b1;
662     controls[29:28] = 2'b00;
663     controls[27] = 1'b0;
664     controls[26] = 1'b0;
665     controls[25] = 1'b0;
666     controls[24:23] = 2'b00;
667     controls[22:21] = 2'b00;
668     controls[20:19] = 2'b00;
669     controls[18:17] = 2'b00;
670     controls[16] = 1'b0;
671     controls[15] = 1'b0;
672     controls[14] = 1'b0;
673     controls[13] = 1'b1;
674     controls[12] = 1'b0;
675     controls[11] = 1'b0;
676     controls[10] = 1'b0;
677     controls[9] = 1'b0;
678     controls[8] = 1'b0;
679     controls[7] = 1'b0;

```

```

680     controls[6] = 1'b0;
681     controls[5:4] = 2'b00;
682     controls[3] = 1'b0;
683     controls[2] = 1'b0;
684     controls[1] = 1'b0;
685     controls[0] = 1'b0;
686 end
687
688 6'b010101: // ja
689 begin
690     controls[30] = 1'b1;
691     controls[29:28] = 2'b00;
692     controls[27] = 1'b0;
693     controls[26] = 1'b0;
694     controls[25] = 1'b0;
695     controls[24:23] = 2'b00;
696     controls[22:21] = 2'b00;
697     controls[20:19] = 2'b00;
698     controls[18:17] = 2'b00;
699     controls[16] = 1'b0;
700     controls[15] = 1'b0;
701     controls[14] = 1'b0;
702     controls[13] = 1'b0;
703     controls[12] = 1'b1;
704     controls[11] = 1'b0;
705     controls[10] = 1'b0;
706     controls[9] = 1'b0;
707     controls[8] = 1'b0;
708     controls[7] = 1'b0;
709     controls[6] = 1'b0;
710     controls[5:4] = 2'b00;
711     controls[3] = 1'b0;
712     controls[2] = 1'b0;
713     controls[1] = 1'b0;
714     controls[0] = 1'b0;
715 end
716
717 6'b010110: // jae
718 begin
719     controls[30] = 1'b1;
720     controls[29:28] = 2'b00;
721     controls[27] = 1'b0;
722     controls[26] = 1'b0;
723     controls[25] = 1'b0;
724     controls[24:23] = 2'b00;
725     controls[22:21] = 2'b00;
726     controls[20:19] = 2'b00;
727     controls[18:17] = 2'b00;
728     controls[16] = 1'b0;

```

```

729     controls[15] = 1'b0;
730     controls[14] = 1'b0;
731     controls[13] = 1'b0;
732     controls[12] = 1'b0;
733     controls[11] = 1'b1;
734     controls[10] = 1'b0;
735     controls[9] = 1'b0;
736     controls[8] = 1'b0;
737     controls[7] = 1'b0;
738     controls[6] = 1'b0;
739     controls[5:4] = 2'b00;
740     controls[3] = 1'b0;
741     controls[2] = 1'b0;
742     controls[1] = 1'b0;
743     controls[0] = 1'b0;
744 end
745
746 6'b010111: // jg
747 begin
748     controls[30] = 1'b1;
749     controls[29:28] = 2'b00;
750     controls[27] = 1'b0;
751     controls[26] = 1'b0;
752     controls[25] = 1'b0;
753     controls[24:23] = 2'b00;
754     controls[22:21] = 2'b00;
755     controls[20:19] = 2'b00;
756     controls[18:17] = 2'b00;
757     controls[16] = 1'b0;
758     controls[15] = 1'b0;
759     controls[14] = 1'b0;
760     controls[13] = 1'b0;
761     controls[12] = 1'b0;
762     controls[11] = 1'b0;
763     controls[10] = 1'b1;
764     controls[9] = 1'b0;
765     controls[8] = 1'b0;
766     controls[7] = 1'b0;
767     controls[6] = 1'b0;
768     controls[5:4] = 2'b00;
769     controls[3] = 1'b0;
770     controls[2] = 1'b0;
771     controls[1] = 1'b0;
772     controls[0] = 1'b0;
773 end
774
775 6'b011000: // jge
776 begin
777     controls[30] = 1'b1;

```



```

778     controls[29:28] = 2'b00;
779     controls[27]  = 1'b0;
780     controls[26]  = 1'b0;
781     controls[25]  = 1'b0;
782     controls[24:23] = 2'b00;
783     controls[22:21] = 2'b00;
784     controls[20:19] = 2'b00;
785     controls[18:17] = 2'b00;
786     controls[16]   = 1'b0;
787     controls[15]   = 1'b0;
788     controls[14]   = 1'b0;
789     controls[13]   = 1'b0;
790     controls[12]   = 1'b0;
791     controls[11]   = 1'b0;
792     controls[10]   = 1'b0;
793     controls[9]    = 1'b1;
794     controls[8]    = 1'b0;
795     controls[7]    = 1'b0;
796     controls[6]    = 1'b0;
797     controls[5:4]  = 2'b00;
798     controls[3]    = 1'b0;
799     controls[2]    = 1'b0;
800     controls[1]    = 1'b0;
801     controls[0]    = 1'b0;
802 end
803
804 6'b011001: // j1
805 begin
806     controls[30] = 1'b1;
807     controls[29:28] = 2'b00;
808     controls[27]  = 1'b0;
809     controls[26]  = 1'b0;
810     controls[25]  = 1'b0;
811     controls[24:23] = 2'b00;
812     controls[22:21] = 2'b00;
813     controls[20:19] = 2'b00;
814     controls[18:17] = 2'b00;
815     controls[16]   = 1'b0;
816     controls[15]   = 1'b0;
817     controls[14]   = 1'b0;
818     controls[13]   = 1'b0;
819     controls[12]   = 1'b0;
820     controls[11]   = 1'b0;
821     controls[10]   = 1'b0;
822     controls[9]    = 1'b0;
823     controls[8]    = 1'b1;
824     controls[7]    = 1'b0;
825     controls[6]    = 1'b0;
826     controls[5:4]  = 2'b00;

```

```

827     controls[3] = 1'b0;
828     controls[2] = 1'b0;
829     controls[1] = 1'b0;
830     controls[0] = 1'b0;
831 end
832
833 6'b011010: // jle
834 begin
835     controls[30] = 1'b1;
836     controls[29:28] = 2'b00;
837     controls[27] = 1'b0;
838     controls[26] = 1'b0;
839     controls[25] = 1'b0;
840     controls[24:23] = 2'b00;
841     controls[22:21] = 2'b00;
842     controls[20:19] = 2'b00;
843     controls[18:17] = 2'b00;
844     controls[16] = 1'b0;
845     controls[15] = 1'b0;
846     controls[14] = 1'b0;
847     controls[13] = 1'b0;
848     controls[12] = 1'b0;
849     controls[11] = 1'b0;
850     controls[10] = 1'b0;
851     controls[9] = 1'b0;
852     controls[8] = 1'b0;
853     controls[7] = 1'b1;
854     controls[6] = 1'b0;
855     controls[5:4] = 2'b00;
856     controls[3] = 1'b0;
857     controls[2] = 1'b0;
858     controls[1] = 1'b0;
859     controls[0] = 1'b0;
860 end
861
862 6'b011011: // cbt
863 begin
864     controls[30] = 1'b0;
865     controls[29:28] = 2'b00;
866     controls[27] = 1'b0;
867     controls[26] = 1'b0;
868     controls[25] = 1'b0;
869     controls[24:23] = 2'b00;
870     controls[22:21] = 2'b01;
871     controls[20:19] = 2'b10;
872     controls[18:17] = 2'b01;
873     controls[16] = 1'b1;
874     controls[15] = 1'b0;
875     controls[14] = 1'b0;

```

```

876     controls[13] = 1'b0;
877     controls[12] = 1'b0;
878     controls[11] = 1'b0;
879     controls[10] = 1'b0;
880     controls[9] = 1'b0;
881     controls[8] = 1'b0;
882     controls[7] = 1'b0;
883     controls[6] = 1'b0;
884     controls[5:4] = 2'b00;
885     controls[3] = 1'b0;
886     controls[2] = 1'b0;
887     controls[1] = 1'b0;
888     controls[0] = 1'b0;
889 end
890
891 6'b011100: // trf
892 begin
893     controls[30] = 1'b0;
894     controls[29:28] = 2'b00;
895     controls[27] = 1'b0;
896     controls[26] = 1'b1;
897     controls[25] = 1'b1;
898     controls[24:23] = 2'b00;
899     controls[22:21] = 2'b00;
900     controls[20:19] = 2'b10;
901     controls[18:17] = 2'b01;
902     controls[16] = 1'b0;
903     controls[15] = 1'b0;
904     controls[14] = 1'b0;
905     controls[13] = 1'b0;
906     controls[12] = 1'b0;
907     controls[11] = 1'b0;
908     controls[10] = 1'b0;
909     controls[9] = 1'b0;
910     controls[8] = 1'b0;
911     controls[7] = 1'b0;
912     controls[6] = 1'b0;
913     controls[5:4] = 2'b00;
914     controls[3] = 1'b0;
915     controls[2] = 1'b0;
916     controls[1] = 1'b0;
917     controls[0] = 1'b1;
918 end
919
920 6'b011101: // pr
921 begin
922     controls[30] = 1'b0;
923     controls[29:28] = 2'b00;
924     controls[27] = 1'b0;

```

```

925     controls[26] = 1'b0;
926     controls[25] = 1'b0;
927     controls[24:23] = 2'b00;
928     controls[22:21] = 2'b00;
929     controls[20:19] = 2'b00;
930     controls[18:17] = 2'b00;
931     controls[16] = 1'b0;
932     controls[15] = 1'b0;
933     controls[14] = 1'b0;
934     controls[13] = 1'b0;
935     controls[12] = 1'b0;
936     controls[11] = 1'b0;
937     controls[10] = 1'b0;
938     controls[9] = 1'b0;
939     controls[8] = 1'b0;
940     controls[7] = 1'b0;
941     controls[6] = 1'b0;
942     controls[5:4] = 2'b00;
943     controls[3] = 1'b1;
944     controls[2] = 1'b1;
945     controls[1] = 1'b0;
946     controls[0] = 1'b0;
947 end
948
949 6'b011110: // cbr
950 begin
951     controls[30] = 1'b0;
952     controls[29:28] = 2'b00;
953     controls[27] = 1'b0;
954     controls[26] = 1'b0;
955     controls[25] = 1'b0;
956     controls[24:23] = 2'b00;
957     controls[22:21] = 2'b10;
958     controls[20:19] = 2'b10;
959     controls[18:17] = 2'b01;
960     controls[16] = 1'b1;
961     controls[15] = 1'b0;
962     controls[14] = 1'b0;
963     controls[13] = 1'b0;
964     controls[12] = 1'b0;
965     controls[11] = 1'b0;
966     controls[10] = 1'b0;
967     controls[9] = 1'b0;
968     controls[8] = 1'b0;
969     controls[7] = 1'b0;
970     controls[6] = 1'b0;
971     controls[5:4] = 2'b00;
972     controls[3] = 1'b0;
973     controls[2] = 1'b0;

```

```

974     controls[1] = 1'b0;
975     controls[0] = 1'b0;
976 end
977
978 6'b011111: // rtr
979 begin
980     controls[30] = 1'b0;
981     controls[29:28] = 2'b00;
982     controls[27] = 1'b0;
983     controls[26] = 1'b1;
984     controls[25] = 1'b0;
985     controls[24:23] = 2'b10;
986     controls[22:21] = 2'b00;
987     controls[20:19] = 2'b10;
988     controls[18:17] = 2'b00;
989     controls[16] = 1'b0;
990     controls[15] = 1'b0;
991     controls[14] = 1'b0;
992     controls[13] = 1'b0;
993     controls[12] = 1'b0;
994     controls[11] = 1'b0;
995     controls[10] = 1'b0;
996     controls[9] = 1'b0;
997     controls[8] = 1'b0;
998     controls[7] = 1'b0;
999     controls[6] = 1'b1;
1000    controls[5:4] = 2'b00;
1001    controls[3] = 1'b0;
1002    controls[2] = 1'b0;
1003    controls[1] = 1'b1;
1004    controls[0] = 1'b0;
1005 end
1006
1007 6'b100000: // rr
1008 begin
1009     controls[30] = 1'b0;
1010     controls[29:28] = 2'b00;
1011     controls[27] = 1'b0;
1012     controls[26] = 1'b0;
1013     controls[25] = 1'b0;
1014     controls[24:23] = 2'b00;
1015     controls[22:21] = 2'b00;
1016     controls[20:19] = 2'b00;
1017     controls[18:17] = 2'b00;
1018     controls[16] = 1'b0;
1019     controls[15] = 1'b0;
1020     controls[14] = 1'b0;
1021     controls[13] = 1'b0;
1022     controls[12] = 1'b0;

```

```

1023     controls[11] = 1'b0;
1024     controls[10] = 1'b0;
1025     controls[9] = 1'b0;
1026     controls[8] = 1'b0;
1027     controls[7] = 1'b0;
1028     controls[6] = 1'b0;
1029     controls[5:4] = 2'b00;
1030     controls[3] = 1'b1;
1031     controls[2] = 1'b0;
1032     controls[1] = 1'b0;
1033     controls[0] = 1'b0;
1034 end
1035
1036 default:
1037 begin
1038     controls[30] = 1'b0;
1039     controls[29:28] = 2'b00;
1040     controls[27] = 1'b0;
1041     controls[26] = 1'b0;
1042     controls[25] = 1'b0;
1043     controls[24:23] = 2'b00;
1044     controls[22:21] = 2'b00;
1045     controls[20:19] = 2'b00;
1046     controls[18:17] = 2'b00;
1047     controls[16] = 1'b0;
1048     controls[15] = 1'b0;
1049     controls[14] = 1'b0;
1050     controls[13] = 1'b0;
1051     controls[12] = 1'b0;
1052     controls[11] = 1'b0;
1053     controls[10] = 1'b0;
1054     controls[9] = 1'b0;
1055     controls[8] = 1'b0;
1056     controls[7] = 1'b0;
1057     controls[6] = 1'b0;
1058     controls[5:4] = 2'b00;
1059     controls[3] = 1'b0;
1060     controls[2] = 1'b0;
1061     controls[1] = 1'b0;
1062     controls[0] = 1'b0;
1063 end
1064
1065 endcase
1066 end
1067
1068 endmodule

```

Listing A.3: Control\_Unit

**Data\_Memory.sv**

```

1  /*
   ↪ *****
   ↪
2  * File name
3    Data_Memory.sv
4  * Description
5    This module describes RAM.
6  * Parameters
7    WIDTH – This is the WIDTH of the input/output data and the address used to read/
   ↪ write.
8  * Inputs
9    clk – The clock for the system.
10
11    memory_write_enable – This is a control signal that indicates if the data at the input
   ↪ should be written or not.
12
13    address_rw – This is the address that is used to read/write.
14
15    data_in – This is the data that will be written to the memory at the address given at
   ↪ the input.
16 * Outputs
17    data_out – This is the data that is read from the address at the input.
18 * Author
19    Sreethyan Aravinthan (UCL)
20 *****/
21
22 module Data_Memory
23 #(
24     parameter WIDTH = 1
25 )
26 (
27     input logic clk,
28     input logic memory_write_enable,
29     input logic [WIDTH - 1:0]address_rw,
30     input logic [WIDTH - 1:0]data_in,
31     output logic [WIDTH - 1:0]data_out
32 );
33
34     logic [WIDTH - 1:0]RAM[2**WIDTH - 1:0];
35
36     always_comb
37     begin
38         data_out = RAM[address_rw];
39     end
40
41     always_ff@(posedge clk)
42     begin
43         if(memory_write_enable == 1'b1)
44             begin

```

```

45     RAM[address_rw] = data_in;
46 end
47 end
48
49 endmodule

```

Listing A.4: Data\_Memory

### Flags\_Register.sv

```

1  /*
    ↪ *****
    ↪
2  * File name
3      Flags_Register.sv
4  * Description
5      This module is a register with an enable signal. This is for the flags.
6  * Parameters
7      NONE
8  * Inputs
9      clk — The clock for the system.
10
11      rst — Signal to reset the system to the default values.
12
13      flags_reg_write_enable — This is an enable signal that allows the the contents of the
    ↪ register to be updated if this is set and it is the positive edge of the clock.
14
15      d — This is the input data to the flags register.
16 * Outputs
17     q — This is the output data to the flags register.
18 * Author
19     Sreethyan Aravinthan (UCL)
20 *****/
21
22 module Flags_Register
23 (
24     input logic clk,
25     input logic rst,
26     input logic flags_reg_write_enable,
27     input logic d[3:0],
28     output logic q[3:0]
29 );
30
31     logic zero[3:0];
32
33     always_comb
34     begin
35         zero[3] = 1'b0;
36         zero[2] = 1'b0;
37         zero[1] = 1'b0;

```



```

38     zero[0] = 1'b0;
39 end
40
41 // asynchronus register
42 always_ff@(posedge clk, posedge rst)
43 begin
44     // if reset is 1 then set the output to 0
45     if(rst == 1)
46         begin
47             q <= zero;
48         end
49     // else set the output to the input value
50     else if(flags_reg_write_enable == 1)
51         begin
52             q <= d;
53         end
54     end
55 endmodule

```

Listing A.5: Flags\_Register

### General\_Purpose\_Register\_File.sv

```

1  /*
   ↳ *****
   ↳
2  * File name
3     General_Purpose_Register_File.sv
4  * Description
5     This module describes a register file.
6  * Parameters
7     ADDR_WIDTH_RF – This indicates the width of the address.
8
9     DATA_WIDTH – This is the width of the data written or read.
10 * Inputs
11     clk – The clock for the system.
12
13     general_register_write_enable – This control signal will write the data given by
   ↳ general_register_write_data at the address indicated by address_3 if set to 1 and it is
   ↳ the positive edge of the clock.
14
15     stack_write_enable – This control signal will write the data given by
   ↳ stack_register_write_data at the address indicated by address_3 if set to 1 and it is the
   ↳ positive edge of the clock.
16
17     address_1 – This indicates which register to be accesed in the register file.
18
19     address_2 – This indicates which register to be accesed in the register file.
20

```

```

21     address_3 – This indicates which register to be accessed in the register file.
22
23     general_register_write_data – This is the data that can be written to any register apart
    ↪ from the 1st register in the register file.
24
25     stack_register_write_data – This is the data that can be written to only the 1st register
    ↪ in the register file.
26 * Outputs
27     read_data_1 – This is the data that is read from the register file at address_1.
28
29     read_data_2 – This is the data that is read from the register file at address_2.
30 * Author
31     Sreethyan Aravinthan (UCL)
32 *****/
33
34 module General_Purpose_Register_File
35 #(
36     parameter ADDR_WIDTH_RF = 1,
37     parameter DATA_WIDTH = 1
38 )
39 (
40     input logic clk,
41     input logic general_register_write_enable,
42     input logic stack_write_enable,
43     input logic [ADDR_WIDTH_RF - 1:0] address_1,
44     input logic [ADDR_WIDTH_RF - 1:0] address_2,
45     input logic [ADDR_WIDTH_RF - 1:0] address_3,
46     input logic [DATA_WIDTH - 1:0] general_register_write_data,
47     input logic [DATA_WIDTH - 1:0] stack_register_write_data,
48     output logic [DATA_WIDTH - 1:0] read_data_1,
49     output logic [DATA_WIDTH - 1:0] read_data_2
50 );
51
52 // regisiter file
53 logic [DATA_WIDTH - 1:0] registers[2**ADDR_WIDTH_RF - 1:0];
54
55 // data read from address pointed by address_1 and address_2
56 always_comb
57 begin
58     read_data_1 = registers[address_1];
59     read_data_2 = registers[address_2];
60 end
61
62 // write data if enable signal is set and positive edge of the clock
63 always_ff@(posedge clk)
64 begin
65     if(general_register_write_enable == 1'b1)
66     begin
67         registers[address_3] <= general_register_write_data;

```

```

68     end
69     if(stack_write_enable == 1'b1)
70     begin
71         registers[0] <= stack_register_write_data;
72     end
73 end
74
75 endmodule

```

Listing A.6: General\_Purpose\_Register\_File

### Instruction\_Memory.sv

```

1  /*
   ↳ *****
   ↳
2  * File name
3      Instruction_Memory.sv
4  * Description
5      This module describes a 2-port ROM.
6  * Parameters
7      ADDR_WIDTH_IM – This is the width of the address field.
8
9      INSTR_WIDTH – This is the width of the instruction.
10
11     FILE_NAME – This is the name of the file that will be loaded and set as default values.
12 * Inputs
13     address_1 – This is the address that is wished to be accessed.
14
15     address_2 – This is the address that is wished to be accessed.
16 * Outputs
17     read_data_1 – This is the value at address_1.
18
19     read_data_2 – This is the value at address_2.
20 * Author
21     Sreethyan Aravinthan (UCL)
22 *****/
23
24 module Instruction_Memory
25 #(
26     parameter ADDR_WIDTH_IM = 1,
27     parameter INSTR_WIDTH = 1,
28     parameter FILE_NAME = "out.mem"
29 )
30 (
31     input logic [ADDR_WIDTH_IM - 1:0]address_1,
32     input logic [ADDR_WIDTH_IM - 1:0]address_2,
33     output logic [INSTR_WIDTH - 1:0]read_data_1,
34     output logic [INSTR_WIDTH - 1:0]read_data_2
35 );

```

```

36
37 logic [INSTR_WIDTH - 1:0]ROM[2**ADDR_WIDTH_IM - 1:0];
38
39 initial
40 begin
41     $readmemb(FILE_NAME, ROM);
42 end
43
44 always_comb
45 begin
46     read_data_1 = ROM[address_1];
47     read_data_2 = ROM[address_2];
48 end
49
50 endmodule

```

Listing A.7: Instruction\_Memory

### Jump\_Logic.sv

```

1  /*
   ↳ *****
   ↳
2  * File name
3      Jump_Logic.sv
4  * Description
5      This module describes how the flags for the jump instructions should be set so that the
   ↳ instructions work.
6  * Parameters
7      NONE
8  * Inputs
9      INPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO THE
   ↳ DATAPATH
10 * Outputs
11     OUTPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO
   ↳ THE DATAPATH
12 * Author
13     Sreethyan Aravinthan (UCL)
14 *****/
15
16 module Jump_Logic
17 (
18     // control signals
19     input logic jump_zero_control,
20     input logic jump_below_control,
21     input logic jump_below_equal_control,
22     input logic jump_above_control,
23     input logic jump_above_equal_control,
24     input logic jump_greater_control,
25     input logic jump_greater_equal_control,

```

```

26 input logic jump_less_control,
27 input logic jump_less_equal_control,
28 // flags
29 input logic zero_flag,
30 input logic carry_flag,
31 input logic sign_flag,
32 input logic overflow_flag,
33 output logic jump_logic_out
34 );
35
36 logic jz_out;
37 logic jb_out;
38 logic jbe_out;
39 logic jbe_1_out;
40 logic ja_out;
41 logic ja_1_out;
42 logic jae_out;
43 logic jae_1_out;
44 logic jg_out;
45 logic jg_1_out;
46 logic jg_2_out;
47 logic jg_3_out;
48 logic jge_out;
49 logic jge_1_out;
50 logic jl_out;
51 logic jl_1_out;
52 logic jle_out;
53 logic jle_1_out;
54 logic jle_2_out;
55
56 // instantiate logic for jump zero instruction
57 and jz(jz_out, jump_zero_control, zero_flag);
58
59 // instantiate logic for jump below instruction
60 and jb(jb_out, jump_below_control, carry_flag);
61
62 // instantiate logic for the jump below equal instruction
63 or jbe_1(jbe_1_out, carry_flag, zero_flag);
64 and jbe(jbe_out, jump_below_equal_control, jbe_1_out);
65
66 // instantiate logic for jump above instruction
67 nand ja_1(ja_1_out, carry_flag, zero_flag);
68 and ja(ja_out, jump_above_control, ja_1_out);
69
70 // instantiate logic for jump above equal instruction
71 not jae_1(jae_1_out, carry_flag);
72 and jae(jae_out, jump_above_equal_control, jae_1_out);
73
74 // instantiate logic for jump greater instruction

```

```

75 not jg_1(jg_1_out, zero_flag);
76 xnor jg_2(jg_2_out, sign_flag, overflow_flag);
77 and jg_3(jg_3_out, jg_1_out, jg_2_out);
78 and jg(jg_out, jump_greater_control, jg_3_out);
79
80 // instantiate logic for jump greater equal instruction
81 xnor jge_1(jge_1_out, sign_flag, zero_flag);
82 and jge(jge_out, jump_greater_equal_control, jge_1_out);
83
84 // instantiate logic for jump less instruction
85 xor jl_1(jl_1_out, sign_flag, zero_flag);
86 and jl(jl_out, jump_less_control, jl_1_out);
87
88 // instantiate logic for jump less equal instruction
89 xor jle_1(jle_1_out, sign_flag, carry_flag);
90 or jle_2(jle_2_out, zero_flag, jle_1_out);
91 and jle(jle_out, jump_less_equal_control, jle_2_out);
92
93 or selector(jump_logic_out, jz_out, jb_out, jbe_out, ja_out, jae_out, jg_out, jge_out, jl_out,
    ↪ jle_out);
94
95 endmodule

```

Listing A.8: Jump Logic

## Multiplexer.sv

```

1 /*
    ↪ *****
    ↪
2 * File name
3   Multiplexer.sv
4 * Description
5   This module describes how a multiplexer works.
6 * Parameters
7   NUM_OF_CONTROL_SIGNALS – Number of control signals
8
9   WIDTH – Number of bits for each input in the multiplexer
10 * Inputs
11   control_signals – Used to select a certain input data.
12
13   data – Set of inputs put sent into the multiplexer.
14 * Outputs
15   multiplexer_out – Output data choosen from the set of inputs.
16 * Author
17   Sreethyan Aravinthan (UCL)
18 *****/
19
20 module Multiplexer
21 #(

```

```

22 parameter NUM_OF_CONTROL_SIGNALS = 1,
23 parameter WIDTH = 1
24 )
25 (
26 input logic [NUM_OF_CONTROL_SIGNALS - 1:0]control_signals,
27 input logic [WIDTH - 1:0]data[2**NUM_OF_CONTROL_SIGNALS - 1:0],
28 output logic [WIDTH - 1:0]multiplexer_out
29 );
30
31 // combinational logic
32 always_comb
33 begin
34 // select the correct data
35 multiplexer_out = data[control_signals];
36 end
37
38 endmodule

```

Listing A.9: Multiplexer

## Register.sv

```

1 /*
   ↳ *****
   ↳
2 * File name
3   Register.sv
4 * Description
5   This module describes how a register.
6 * Parameters
7   WIDTH – Number of bits for the input data.
8 * Inputs
9   clk – The clock for the system.
10
11   rst – Signal to reset the system to the default values.
12
13   d – This is the input to the register.
14 * Outputs
15   q – This is the output of the register.
16 * Author
17   Sreethyan Aravinthan (UCL)
18 *****/
19
20 module Register
21 #(
22   parameter WIDTH = 1
23 )
24 (
25   input logic clk, // clock signal
26   input logic rst, // reset signal

```

```

27 input logic [WIDTH - 1:0] d, // input signal
28 output logic [WIDTH - 1:0] q // output signal
29 );
30
31 // asynchronous register
32 always_ff@(posedge clk, posedge rst)
33 begin
34     // if reset is 1 then set the output to 0
35     if(rst == 1)
36     begin
37         q <= 0;
38     end
39     else // else set the output to the input value
40     begin
41         q <= d;
42     end
43 end
44
45 endmodule

```

Listing A.10: Register

## Datapath.sv

```

1 /*
   ↳ *****
   ↳
2 * File name
3     Datapath.sv
4 * Description
5     This module describes how the components of the datapath are connected.
6 * Parameters
7     NONE
8 * Inputs
9     clk — The clock for the system.
10
11     rst — Signal to reset the system to the default values.
12
13     OTHER INPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT
   ↳ TO THE DATAPATH
14 * Outputs
15     OUTPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO
   ↳ THE DATAPATH
16 * Author
17     Sreethyan Aravinthan (UCL)
18 *****/
19
20 ‘include "ALU.sv"
21 ‘include "ALU.advanced.sv"
22 ‘include "Multiplexer.sv"

```



```

23 'include "Register.sv"
24 'include "General_Purpose_Register_File.sv"
25 'include "Flags_Register.sv"
26 'include "Jump_Logic.sv"
27
28 module Datapath
29 (
30     input logic clk,
31     input logic rst,
32     // Instruction memory
33     output logic [15:0]address_1,
34     output logic [15:0]address_2,
35     input logic [15:0]read_address_1,
36     input logic [15:0]read_address_2,
37     // Data memory
38     output logic [15:0]address_rw,
39     output logic [15:0]data_in,
40     input logic [15:0]data_out,
41     // Control signals
42     input logic pc_increment_control,
43     input logic [1:0] pc_control,
44     input logic general_register_write_enable,
45     input logic stack_write_enable,
46     input logic stack_control,
47     input logic [1:0] write_data_enable,
48     input logic [1:0] ALU_source_1,
49     input logic [1:0] ALU_source_2,
50     input logic [1:0] ALU_control,
51     input logic flags_write_enable,
52     input logic jump_zero_control,
53     input logic jump_below_control,
54     input logic jump_below_equal_control,
55     input logic jump_above_control,
56     input logic jump_above_equal_control,
57     input logic jump_greater_control,
58     input logic jump_greater_equal_control,
59     input logic jump_less_control,
60     input logic jump_less_equal_control,
61     input logic [1:0] general_register_result_select,
62     // Communications Processor Signals
63     input logic [15:0] RAM_rx_data_out, // gpp
64     input logic data_rx_flag, // gpp
65     input logic gpp_trf_cp, // gpp
66     output logic [15:0] gpp_tx_data // gpp
67 );
68
69 // constants
70 localparam [15:0] zero = 16'h0000;
71 localparam [15:0] one = 16'h0001;

```

```

72  localparam [15:0] two = 16'h0002;
73
74  // signals
75  logic [15:0] branch_mux_data[1:0];
76  logic [15:0] branch_mux_out;
77  logic [15:0] general_register_result_select_out;
78  logic [15:0] pc_control_data_mux [3:0];
79  logic [15:0] next_PC;
80  logic branch_result;
81  logic pc_increment_choose_out;
82  logic [15:0] pc_increment_mux_out;
83  logic [15:0] pc_increment_data_mux [1:0];
84  logic [15:0] ALU_out;
85  logic [15:0] WD_control_data_mux[3:0];
86  logic [15:0] ALU_source_2_data_mux[3:0];
87  logic [15:0] ALU_source_2_mux_out;
88  logic zero_flag;
89  logic carry_flag;
90  logic sign_flag;
91  logic overflow_flag;
92  logic [15:0] general_register_result_select_data_mux [3:0];
93  logic [15:0] reg_read_data_1;
94  logic [15:0] stack_control_data_mux [1:0];
95  logic [15:0] stack_control_mux_out;
96  logic flags_in [3:0];
97  logic flags_out [3:0];
98
99  logic [15:0] ALU_source_1_data_mux[3:0];
100 logic [15:0] ALU_source_1_mux_out;
101
102 always_comb
103 begin
104     branch_mux_data[1] = read_address_2;
105
106     pc_control_data_mux[0] = branch_mux_out;
107     pc_control_data_mux[1] = read_address_2;
108     pc_control_data_mux[2] = general_register_result_select_out;
109     // pc_control_data_mux[3] = ;
110
111     pc_increment_data_mux[0] = one;
112     pc_increment_data_mux[1] = two;
113
114     ALU_source_2_data_mux[0] = data_in;
115     ALU_source_2_data_mux[1] = read_address_2;
116     ALU_source_2_data_mux[2] = one;
117     // ALU_source_2_data_mux
118
119     general_register_result_select_data_mux[0] = data_out;
120     general_register_result_select_data_mux[1] = ALU_out;

```

```

121 general_register_result_select_data_mux[2] = reg_read_data_1;
122 general_register_result_select_data_mux[3] = read_address_2;
123
124 stack_control_data_mux[0] = zero;
125 stack_control_data_mux[1] = one;
126
127 flags_in[3] = zero_flag;
128 flags_in[2] = carry_flag;
129 flags_in[1] = sign_flag;
130 flags_in[0] = overflow_flag;
131
132 WD_control_data_mux[2] = RAM_rx_data_out;
133
134 ALU_source_1_data_mux[0] = reg_read_data_1;
135 ALU_source_1_data_mux[1] = gpp_trf_cp;
136 ALU_source_1_data_mux[2] = data_rx_flag;
137 // ALU_source_1_data_mux_out[3] =
138
139 gpp_tx_data = general_register_result_select_out;
140 end
141
142 // instantiating branch multiplexer
143 Multiplexer #(1, 16) branch_mux(branch_result, branch_mux_data, branch_mux_out);
144
145 // instantiating pc control multiplexer
146 Multiplexer #(2, 16) pc_control_mux(pc_control, pc_control_data_mux, next_PC);
147
148 // instantiate pc register
149 Register #(16) pc_register(clk, rst, next_PC, address_1);
150
151 // instantiate adder to see the next instruction
152 ALU #(16) pc_adder(address_1, one, 2'b00, address_2);
153
154 // instantiate xor for which will be the control for choosing if the increment is 1 or 2
155 xor pc_increment_choose(pc_increment_choose_out, branch_result, pc_increment_control);
156
157 // instantiate pc increment multiplexer
158 Multiplexer #(1, 16) pc_increment_mux(pc_increment_choose_out,
    ↪ pc_increment_data_mux, pc_increment_mux_out);
159
160 // instantiate pc increment adder
161 ALU #(16) pc_increment_adder(address_1, pc_increment_mux_out, 2'b00, branch_mux_data
    ↪ [0]);
162
163 // instantiate register file
164 General_Purpose_Register_File #(5, 16) register_file_0(clk, general_register_write_enable,
    ↪ stack_write_enable, read_address_1[9:5], read_address_1[4:0], read_address_1[4:0],
    ↪ general_register_result_select_out, ALU_out, reg_read_data_1, WD_control_data_mux[0]);
165

```

```

166 // instantiate stack control multiplexer
167 Multiplexer #(1, 16) stack_control_mux(stack_control, stack_control_data_mux,
    ↪ stack_control_mux_out);
168
169 // instantiate a subber for correct stack access
170 ALU #(16) stack_access_subber(reg_read_data_1, stack_control_mux_out, 2'b01, address_rw);
171
172 // instantiate WD control multiplexer
173 Multiplexer #(2, 16) WD_control_mux(write_data_enable, WD_control_data_mux,
    ↪ data_in);
174
175 // instantiate adder for return address
176 ALU #(16) return_address_adder(address_2, one, 2'b00, WD_control_data_mux[1]);
177
178 // instantiate ALU source 1 multiplexer
179 Multiplexer #(2, 16) ALU_source_1_mux(ALU_source_1, ALU_source_1_data_mux,
    ↪ ALU_source_1_mux_out);
180
181 // instantiate ALU source 2 multiplexer
182 Multiplexer #(2, 16) ALU_source_2_mux(ALU_source_2, ALU_source_2_data_mux,
    ↪ ALU_source_2_mux_out);
183
184 // instantiate ALU
185 ALU_advanced #(16) ALU_0(ALU_source_1_mux_out, ALU_source_2_mux_out, ALU_control,
    ↪ ALU_out, zero_flag, carry_flag, sign_flag, overflow_flag);
186
187 // instantiate zero register
188 Flags_Register flag_reg(clk, rst, flags_write_enable, flags_in, flags_out);
189
190 // instantiate jump logic
191 Jump_Logic jump_logic_0(jump_zero_control, jump_below_control, jump_below_equal_control,
    ↪ jump_above_control, jump_above_equal_control, jump_greater_control,
    ↪ jump_greater_equal_control, jump_less_control, jump_less_equal_control, flags_out[3],
    ↪ flags_out[2], flags_out[1], flags_out[0], branch_result);
192
193 // instantiate general register result selector
194 Multiplexer #(2, 16) general_register_result_select_mux(general_register_result_select,
    ↪ general_register_result_select_data_mux, general_register_result_select_out);
195
196 endmodule

```

Listing A.11: Datapath

## GPP.sv

```

1 /*
    ↪ *****
    ↪
2 * File name
3   GPP.sv

```

```

4 * Description
5     This module instantiates the Datapath and control unit and connects them together.
6 * Parameters
7     NONE
8 * Inputs
9     clk – The clock for the system.
10
11     rst – Signal to reset the system to the default values.
12
13     OTHER INPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT
14     ↪ TO THE DATAPATH
15 * Outputs
16     OUTPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO
17     ↪ THE DATAPATH
18 * Author
19     Sreethyan Aravinthan (UCL)
20 *****/
21 module GPP
22 (
23     input logic clk,
24     input logic rst,
25     // Instruction memory
26     output logic [15:0]address_1,
27     output logic [15:0]address_2,
28     input logic [15:0]read_address_1,
29     input logic [15:0]read_address_2,
30     // Data memory
31     output logic [15:0]address_rw,
32     output logic [15:0]data_in,
33     input logic [15:0]data_out,
34     // Control Signals
35     output logic memory_write_enable,
36     output logic enable_rtr,
37     output logic gpp_rtr_cp,
38     output logic gpp_rtr_dp,
39     output logic gpp_trf_dp,
40     // Communications Processor Signals
41     input logic [15:0] RAM_rx_data_out, // gpp
42     input logic data_rx_flag, // gpp
43     input logic gpp_trf_cp, // gpp
44     output logic [15:0] gpp_tx_data // gpp
45 );
46
47 // Control signals
48 logic pc_increment_control;
49 logic [1:0] pc_control;
50 logic general_register_write_enable;
51 logic stack_write_enable;

```

```

51 logic stack_control;
52 logic [1:0] write_data_enable;
53 logic [1:0] ALU_source_1;
54 logic [1:0] ALU_source_2;
55 logic [1:0] ALU_control;
56 logic flags_write_enable;
57 logic jump_zero_control;
58 logic jump_below_control;
59 logic jump_below_equal_control;
60 logic jump_above_control;
61 logic jump_above_equal_control;
62 logic jump_greater_control;
63 logic jump_greater_equal_control;
64 logic jump_less_control;
65 logic jump_less_equal_control;
66 logic [1:0] general_register_result_select;
67
68 Datapath datapath_processor(clk,
69     rst,
70     address_1,
71     address_2,
72     read_address_1,
73     read_address_2,
74     address_rw,
75     data_in,
76     data_out,
77     pc_increment_control,
78     pc_control,
79     general_register_write_enable,
80     stack_write_enable,
81     stack_control,
82     write_data_enable,
83     ALU_source_1,
84     ALU_source_2,
85     ALU_control,
86     flags_write_enable,
87     jump_zero_control,
88     jump_below_control,
89     jump_below_equal_control,
90     jump_above_control,
91     jump_above_equal_control,
92     jump_greater_control,
93     jump_greater_equal_control,
94     jump_less_control,
95     jump_less_equal_control,
96     general_register_result_select,
97     RAM_rx_data_out, // gpp
98     data_rx_flag, // gpp
99     gpp_trf_cp, // gpp

```

```

100         gpp.tx.data); // gpp
101
102
103     Control_Unit cu_processor (read_address_1[15:10],
104         pc.increment_control,
105         pc.control,
106         general_register_write_enable,
107         stack_write_enable,
108         stack_control,
109         write_data_enable,
110         ALU_source_1,
111         ALU_source_2,
112         ALU_control,
113         flags_write_enable,
114         jump_zero_control,
115         jump_below_control,
116         jump_below_equal_control,
117         jump_above_control,
118         jump_above_equal_control,
119         jump_greater_control,
120         jump_greater_equal_control,
121         jump_less_control,
122         jump_less_equal_control,
123         memory_write_enable,
124         general_register_result_select,
125         enable_rtr,
126         gpp_rtr_cp,
127         gpp_rtr_dp,
128         gpp_trf_dp);
129
130 endmodule

```

Listing A.12: GPP

# Appendix B

## CP

### Control\_Plane.sv

```
1  /*
    ↳ *****
    ↳
2  * File name
3      comms_processor.sv
4  * Description
5      This module describes how the control plane works.
6  * Parameters
7      NONE
8  * Inputs
9      clk — The clock for the system.
10
11      rst — Signal to reset the system to the default values.
12
13      node_id — This will be the id that will be given to the node in the system. NOTE:
    ↳ EACH NODE MUST HAVE A DIFFERENT ID OTHERWISE THERE WILL BE
    ↳ ISSUES IN TX/RX.
14
15      max_node — This shows the maximum number of nodes in the ring topology.
16
17      data_rx_complete_flag — This flag will reset the data_rx_flag. The value for this is sent
    ↳ from the data plane reciever.
18
19      control_rx_packet — This is the packet that will be recieved on the control plane.
20
21      data_tx_complete_flag — This flag will reset the data_tx_flag. The value for this is sent
    ↳ from the data plane transmitter.
22
23      RAM_tx_data_out — This input has the value equal to the top of the RAM of the data
    ↳ plane tx RAM.
24
```



```

25     sp_tx_current – This has the value of the stack pointer from the data plane tx RAM.
    ↪ This is used to see if the data plane tx RAM is empty or not.
26
27     enable_rtr – This is a control signal that will allow the GPP to retrieve data from the CP
    ↪ if it is possible.
28
29     gpp_rtr_cp – This is a control signal that pauses rx from gpp so that gpp can retrieve
    ↪ data from rx RAM.
30 * Outputs
31     control_tx_packet – This is the packet that is transmitted on the control plane.
32
33     data_rx_node_id – This is the value that is sent to a control unit which sets the
    ↪ wavelength/spatial channel of the receiver.
34
35     data_tx_flag – This is a flag which indicates if the node is transmitting data on the data
    ↪ plane.
36
37     data_rx_flag – This is used to show the control plane if data is being received on the
    ↪ data plane. Thus the control plane will not say yes for another ping if set. This is also
    ↪ a flag that is used by the GPP to see if it can retrieve data from the data plane receiver
    ↪ . This will be connected to the ALU_src_1.
38
39     gpp_trf_cp – this is a flag that is used by the GPP to transfer data from the GPP RAM
    ↪ to the data plane transmitter RAM. This will be connected to the ALU_src_1.
40 * Author
41     Sreethyan Aravinthan (UCL)
42 *****/
43
44 module Control_Plane
45 (
46     input logic clk,
47     input logic rst,
48     input shortint node_id,
49     input shortint max_node,
50     input logic data_rx_complete_flag, // ***
51     input logic [31:0] control_rx_packet,
52     input logic data_tx_complete_flag, // ***
53     input logic [15:0] RAM_tx_data_out, // ***
54     input logic [15:0] sp_tx_current, // ***
55     input logic enable_rtr,
56     input logic gpp_rtr_cp,
57     output logic [31:0] control_tx_packet,
58     output logic [15:0] data_rx_node_id,
59     output logic data_tx_flag, // ***
60     output logic data_rx_flag,
61     output logic gpp_trf_cp
62 );
63
64 // COMMENT ALL THE CODE AND MAKE SURE ALL THE VARIABLE NAMES ARE

```

```

65 // SNESIBLE. FURTHERMORE THE VAIRABLES THAT ARE LINKED BETWEEN
    ↳ THE
66 // MODULES SHOULD HAVE THE SAME NAME TO AVOID CONFUSION
67
68 // MAKE SURE THE CORRECT CONTROL SIGNALS AND LOGIC IS APPLIED TO
    ↳ THE
69 // SYSTEM. BELOW IS HOW TO DO IT FOR BOTH THE TX AND RX
70 //
71 // THEN MAKE THE LINK BETWEEN GPP AND COMMS.PROCESSOR
72 //
73 // THEN CREATE THE INSTRUCTIONS THAT ARE NECESSARY
74
75 // HOW SHOULD THE DATA FROM THE GPP RAM BE PLACED INTO THE TX
    ↳ RAM
76 // MAKE PHYSICAL CONNECTION BETWEEN GPP RAM AND TX RAM
77 // OUTPUT FROM THE CONTROL PLANE IF THERE ARE ENOUGH CLOCK
    ↳ CYCLES TO
78 // THIS RESULT SHOULD BE CONNECTED TO ALU SRC 1
79 // UPON CHECKING IF IT IS OK TO TRANSMIT THEN TRANSFER THE DATA
80 // ELSE DO NOT TRANSMIT AND TRY AGAIN LATER
81 // NEED TO CREATE TWO INSTRUCTIONS
82 // 1ST INSTR: TO COMPARE SEE IF THE NUMBER OF CYCLES ARE ENOUGH
83 // 2ND INSTR: TRANSFER THE DATA FROM THE GPP RAM TO TX RAM
84
85 /*
86 * transfer_func:
87 *   cbt # compare the value to see if data cane be transferred
88 *   jnz exit # if the result is not equal to zero then data cannot be
89 *             transfered
90 *   # transfer 5 packet data
91 *   trf
92 *   trf
93 *   trf
94 *   trf
95 *   trf
96 * exit:
97 *   return
98 */
99
100 // HOW SHOULD THE DATA FROM THE RX RAM O THE GPP RAM BE
    ↳ PLACED
101 // MAKE THE CONNECTIONS BETWEEN THE GPP AND THE DATA RX
102 // SET CONTROL SIGNAL FROM THE GPP TO THE CONTROL PLANE
103 // THEN THIS WILL MAKE SURE NOT TO SET THE RX FLAG IF A PING COMES
104 // THEN COMPARE THE RX FLAG OUTPUT FROM THE CONTROL PLANE TO SEE
    ↳ IF
105 // THE DATA CAN BE RETRIEVED
106 // IF THE COMPARISION LEADS TO THE CONCLUSION SUCH THAT IT CANNOT
    ↳ BE

```

```

107 // RETRIVED NOW THEN DO NOT TRY TO RETRIVE NOW
108 // IF THE COMPARISON LEADS TO THE CONCLUSION SUCH THAT THE DATA
    ↳ CAN BE
109 // RETRIVED FROM THE DATA PLANE THEN RETRIVE THE PACKET
110 // 4 INSTRUCTIONS NEED TO BE DESIGNED
111 // 1ST INSTR: SET THE CONTROL SIGNAL TO PAUSE THE RECIEVE IN THE
112 // CONTROL PLANE
113 // 2ND INSTR: CHECK IF THE DATA CAN BE RETRIEVED FROM THE RX RAM
114 // 3RD INSTR: RETRIEVE THE DATA FROM THE RX RAM TO THE GPP RAM
115 // 4TH INSTR: RESUME RECEIVING THE DATA ON THE CONTROL PLANE
116
117 /*
118 * pr # instruction 'pause recieving' this stops the control
119 * plane from saying yes if it receives a ping
120 * cbr # instruction which sees if the data can be retrived
121 * # instruction is called 'can be retrived'
122 * # Remember the data_rx_flag is connected to the
123 * # gpp. specifically the ALU src 1
124 * jz exit # if the result is zero then this indicates that the
125 * data plane is reciving. therefore do not retrieve the
126 * data.
127 * rtr
128 * rtr
129 * rtr
130 * rtr
131 * rtr
132 *exit:
133 * rr # instruction 'resume receving'
134 * return
135 */
136
137 // THINGS TO DO
138 // PART 1
139
140 // go through the design of the system and find out hot the control
141 // signals from gpp should be connected with the control and data
142 // plane
143 // ADD CONTROL SIGNAL FROM GPP
144 // OUTPUT THE DATA_RX_FLAG TO SEE IF THE GPP CAN GO AND RETIRVE THE
145 // DATA FROM RAM
146 // OUTPUT THE CURRENT SLOT AND TX_FLAG(ALREADY DONE AS
    ↳ DATA_TX_FLAG)
147 // BOTH WILL BE USED TO SEE IF THE GPP CAN PUT DATA INTO THIS RAM
148 // THIS IS DONE BY SEEING IF THE NUMBER OF CLOCK CYCLES IS GREATER
149 // THAN THE MINIMUM NUMBER OF CYCLES REQUIRED TO TRANSFER THE
    ↳ DATA
150
151 // variables
152

```

```

153 // the system is designed such that each slot on the control plane is
154 // equal to n*clock_period. Therefore to count to n the variable count
155 // exists which increases with each clock cycle and resets to 0 when
156 // it hits the max value such that the count*clock_period ==
157 // n*clock_period
158 logic [1:0] count;
159 // This variable holds the current slot on the control plane. By
160 // default it will equal to 1. However, it will increment to the
161 // maximum number of nodes in the network. therefore the type of this
162 // variable is shortint as this is a 16bit value and our system aims
163 // to hold 2^10 nodes
164 shortint slot;
165 // this is a flag and it is used so that a ping is not made more than
166 // once in the allocated slot on the control plane for a specific node
167 logic control_tx_flag;
168 // defining constants which are used for the ping response
169 localparam [15:0] all_zero = 16'h0000;
170 localparam [15:0] all_one = 16'hFFFF;
171 shortint num_of_slots = 0;
172 logic gpp_rtr = 1'b0;
173
174 always_comb
175 begin
176     if(enable_rtr == 1'b1)
177         begin
178             gpp_rtr = gpp_rtr_cp;
179         end
180 end
181
182 // increment counter for each clock cycle
183 // 100 Gb/s transcieverer. Find the number of bits sending (32 bits). Math. Make as short as
184 //    ↪ possible. nut limited by clock period of processor
185 // 32 / 100*10^9 = 3.2*10^-10s: it takes this long to send a packet
186 // therefore it will take 3.125*10^9 Hz
187 // set clock speed to 3*10^9 Hz
188 // let 1 slot to be 3 clock cycle
189
190 // this logic deals with setting the value of the current slot. thus
191 // indicating if the curent node can transmit on the control plane or not
192 // provided reset is set to 0
193 always_ff @(posedge clk)
194 begin
195     // this logic sets the default values of count and slot
196     // provided reset is asserted
197     if(rst == 1'b1)
198         begin
199             count <= 2'b00;
200             slot <= 1;
201         end

```

```

201 else
202 begin
203 // this means one slot time has occurred
204 if(count == 2)
205 begin
206 // if the slot is equal to max number of nodes in the
207 // network then reset the slot to 1
208 if(slot == max_node)
209 begin
210 slot <= 1;
211 end
212 // otherwise just increment the slot by one to
213 // show that the next node can now transmit
214 else
215 begin
216 slot <= slot + 1;
217 end
218 // reset count to 0
219 count <= 2'b00;
220 end
221 // increment the count value
222 else
223 begin
224 count <= count + 1;
225 end
226 end
227 end
228
229 // control plane logic
230 // provided reset is set to 0
231 always_ff @(posedge clk)
232 begin
233 // this sets the default values of all the variables that is used
234 // within the control plane
235 // provided reset is set to 1
236 if(rst == 1'b1)
237 begin
238 control_tx_packet <= 32'b0;
239 data_rx_flag <= 1'b0;
240 data_tx_flag <= 1'b0;
241 control_tx_flag <= 1'b0;
242 data_rx_node_id <= 16'h0000;
243 end
244 else
245 begin
246 // if the slot is equal to the node id then it means that this
247 // node can now transmit on the control plane a packet
248 // to another node saying that it wishes to transmit
249 // data and wants to see if it is free

```

```

250  if(slot == node_id)
251  begin
252      // provided the stack is not empty
253      // and in this slot a transmission has not taken place
254      // and provided the data plane is not tx to another
255      // node
256      if(sp_tx.current != 1'b0 && control_tx_flag == 1'b0 && data_tx_flag == 1'b0)
257      begin
258          // send a packet consisting of the dest_id
259          // and src_id
260          control_tx_packet <= {RAM_tx_data_out, node_id};
261          control_tx_flag <= 1'b1;
262      end
263      // if it does not meet the above conditions then send
264      // out 32'b0
265      else
266      begin
267          control_tx_packet <= 32'b0;
268      end
269  end
270  // if the slot is not equal to the node it then it means that
271  // this node cannot ping another node but other nodes
272  // may ping this node regarding data transfer
273  // the response to this should be on the control plane
274  else
275  begin
276      // reset the control_tx_flag for later
277      control_tx_flag <= 1'b0;
278      // compare the 16 most significant bits to the node id
279      // if equal then the packet is meant for this node
280      // if the packet is not meant for this node then drop
281      // it
282      // provided the lower 16 bits are not 0x0000 and
283      // 0xFFFF this means that this node is being pinged
284      // for a data transfer
285      if((control_rx_packet[31:16] == node_id) && (control_rx_packet[15:0] != all_zero) && (
↪ control_rx_packet[15:0] != all_one))
286      begin
287          // if the data plane is not receiving and the gpp does not want to rtr the data from
↪ RAM
288          // the node can accept data on the data plane
289          if((data_rx_flag == 1'b0) && (gpp_rtr_cp == 1'b0))
290          begin
291              // tell the data plane which node is
292              // receiving the data
293              data_rx_node_id <= control_rx_packet[15:0];
294              // set the flag which tells the data
295              // plane that it can transmit
296              data_rx_flag <= 1'b1;

```

```

297         // tell the node that asked that it
298         // can transmit to this node on the
299         // data plane
300         control_tx_packet <= {control_rx_packet[15:0], all_one};
301     end
302     // if the data plane is already recieving data
303     // or if the flag is set from the gpp
304     // then
305     // from another node then tell the
306     // current node not now
307     else // if(data_rx_flag == 1'b1 || gpp_rtr == 1'b1)
308     begin
309         control_tx_packet <= {control_rx_packet[15:0], all_zero};
310     end
311 end
312 // if the packet was not meant for the node then drop
313 // it and let the control_tx_packet equal to 0
314 else
315 begin
316     control_tx_packet <= 32'b0;
317 end
318 end
319
320 // this will reset the data_rx_flag if all the data has been received
321 // the signal data_rx_complete_flag will go high from the data plane
322 // indicating that the local flag for rx goes to zero
323 // this is provided the rst is low
324 if(data_rx_complete_flag == 1'b1)
325 begin
326     data_rx_flag <= 1'b0;
327 end
328
329 // this is the logic for ping response
330 // provided reset is set to 0
331
332 // provided the packet is meant for this node
333 // and provided the packet has all zeros or all ones for the
334 // lower 16 bits then this is packet is a ping response
335 if(control_rx_packet[31:16] == node_id && (control_rx_packet[15:0] == all_zero ||
↪ control_rx_packet[15:0] == all_one))
336 begin
337     // if the reponse is one then the requested node is
338     // not busy and it can accept data on the data
339     // plane
340     if(control_rx_packet[15:0] == all_one)
341     begin
342         data_tx_flag <= 1'b1;
343     end
344 end

```

```

345
346 // this will reset the data_tx_flag if all the data has been
347 // transmitted
348 // the signal data_tx_complete_flag will go high from the data plane
349 // indicating that the local flag for tx goes to zero
350 // this is provided the rst is low
351 if(data_tx_complete_flag == 1'b1)
352 begin
353     data_tx_flag <= 1'b0;
354 end
355
356 end
357 end
358
359 // this logic is used to see if data can be transfered from the GPP to CP
360 always_comb
361 begin
362     // check if data is being transmitted
363     if(data_tx_flag == 1'b0)
364     begin
365         // logic to calculate the number of slots depending on
366         // the current slot, node id & max node in network
367         if(node_id < slot)
368         begin
369             num_of_slots = max_node - slot + node_id - 1;
370         end
371         else if(node_id > slot)
372         begin
373             num_of_slots = node_id - slot - 1;
374         end
375         else // if(node_id == slot)
376         begin
377             num_of_slots = 0;
378         end
379     end
380     // if not then set the flag to 0
381     else
382     begin
383         gpp_trf_cp = 1'b0;
384         num_of_slots = 0;
385     end
386
387     // if the number of slots is greater than or equal to
388     // 2 then set the flag as there are enough
389     // clock cycles
390     if(num_of_slots >= 2)
391     begin
392         gpp_trf_cp = 1'b1;
393     end

```



```

394     else
395     begin
396         gpp_trf_cp = 1'b0;
397     end
398 end
399
400 endmodule

```

Listing B.1: Control\_Plane

### data\_plane\_tx.sv

```

1  /*
    ↳ *****
    ↳
2  * File name
3      data_plane_tx.sv
4  * Description
5      This module describes how the data plane tx works.
6  * Parameters
7      NONE
8  * Inputs
9      clk — The clock for the system.
10
11      rst — Signal to reset the system to the default values.
12
13      gpp_trf_dp — This is a control signal which tells the data plane transmitter that data is
    ↳ going to be transferred from the GPP to the DP transmitter.
14
15      gpp_tx_data — This is the data that will be transferred from the GPP to the DP
    ↳ transmitter RAM.
16
17      node_id — This will be the id that will be given to the node in the system. NOTE:
    ↳ EACH NODE MUST HAVE A DIFFERENT ID OTHERWISE THERE WILL BE
    ↳ ISSUES IN TX/RX.
18
19      data_tx_flag — This is a flag which indicates if the node is transmitting data on the data
    ↳ plane.
20 * Outputs
21      data_tx_complete_flag — This flag will reset the data_tx_flag. The value for this is sent
    ↳ from the data plane transmitter.
22
23      data_tx_packet — This is the packet that is transmitted on the data plane.
24
25      RAM_tx_data_out — This input has the value equal to the top of the RAM of the data
    ↳ plane tx RAM.
26
27      sp_tx_current — This has the value of the stack pointer from the data plane tx RAM.
    ↳ This is used to see if the data plane tx RAM is empty or not.
28 * Author

```

```

29      Sreethyan Aravinthan (UCL)
30      *****/
31      'include "ALU.sv"
32      'include "Data_Memory.sv"
33      'include "Multiplexer.sv"
34      'include "Register.sv"
35
36      module data_plane_tx
37      (
38          input logic clk,
39          input logic rst,
40          input logic gpp_trf_dp,
41          input logic [15:0] gpp_tx_data,
42          input shortint node_id,
43          input logic data_tx_flag,
44          output logic data_tx_complete_flag,
45          output logic [31:0] data_tx_packet,
46          output logic [15:0] RAM_tx_data_out,
47          output logic [15:0] sp_tx_current
48      );
49
50      // this is used to see if the number of data packets sent is equal to
51      // the fixed packet length
52      logic [2:0] count;
53      // this holds the current dest node that all the packets need to be
54      // transmitted to
55      logic [15:0] current_dest_node;
56      // this is a flag that is used to transmit a special packet to the
57      // destination node
58      // this special packet consists of the destination node and the src
59      // node id
60      logic first_flag;
61      // this is used to add to subtract from the stack pointer
62      localparam [15:0] one = 16'h0001;
63      // this will hold the data that is fed into the sp multiplexer
64      // there are three values
65      // same value as earlier
66      // sp + 1
67      // sp - 1
68      logic [15:0] sp_tx_mux_data [3:0];
69      // this will hold the next sp value that will be the input to the
70      // register
71      logic [15:0] sp_tx_next;
72      // this contains the sp plus 1
73      logic [15:0] sp_tx_current_plus_one;
74      // this contains the sp minus 1
75      logic [15:0] sp_tx_current_minus_one;
76      // this will be the control signals of the sp multiplexer
77      logic [1:0] sp_tx_mux_control;

```

```

78 // this will contain the RAM addresses that can be accessed and are
79 // the inputs to the multiplexer for the RAM address choice
80 logic [15:0] RAM_tx_address_mux_data [1:0];
81 // This is RAM address that is currently being accessed
82 logic [15:0] RAM_address;
83
84 // this combinational logic sets the values for all the multiplexer
85 // data
86 always_comb
87 begin
88 // stack pointer multiplexer data being set
89 sp_tx_mux_data[0] = sp_tx_current;
90 sp_tx_mux_data[1] = sp_tx_current_plus_one;
91 sp_tx_mux_data[2] = sp_tx_current_minus_one;
92
93 // set the RAM address multiplexer data
94 RAM_tx_address_mux_data[0] = sp_tx_current;
95 RAM_tx_address_mux_data[1] = sp_tx_current_plus_one;
96 end
97
98 // wire up the control signal for the stack pointer multiplexer
99 assign sp_tx_mux_control = {data_tx_flag, gpp_trf_dp};
100 // 0 0 – same
101 // 0 1 – plus 1
102 // 1 0 – minus 1
103
104 // instantiate multiplexer to choose the next stack pointer
105 Multiplexer #(2, 16) sp_tx_mux(sp_tx_mux_control, sp_tx_mux_data, sp_tx_next);
106
107 // instantiate register for stack pointer tx
108 Register #(16) sp_tx_reg(clk, rst, sp_tx_next, sp_tx_current);
109
110 // ALU for stack pointer increment to point to the latest data that
111 // has been added
112 ALU #(16) sp_increment(sp_tx_current, one, 2'b00, sp_tx_current_plus_one);
113
114 // ALU for stack pointer decrement. this is needed if the data is
115 // retrived from the RAM
116 ALU #(16) sp_decrement(sp_tx_current, one, 2'b01, sp_tx_current_minus_one);
117
118 // multiplexer for choosing the most appropriate address
119 Multiplexer #(1, 16) RAM_tx_address_mux(gpp_trf_dp, RAM_tx_address_mux_data,
    ↪ RAM_address);
120
121 // instantiate RAM modules
122 Data_Memory #(16) RAM_tx(clk, gpp_trf_dp, RAM_address, gpp_tx_data, RAM_tx_data_out)
    ↪ ;
123
124 // need to set a variable and not depend on the data_tx_flag_out

```

```

125 // not good due to the clock cycles required to reset it as well
126
127
128 // get the destination node from the top of
129 // the node
130 always_comb
131 begin
132 // provided the data plane is not transmitting get the value
133 if(data_tx_flag == 1'b0)
134 begin
135     current_dest_node = RAM_tx_data_out;
136 end
137 else
138 begin
139     current_dest_node = current_dest_node;
140 end
141 end
142
143 // logic for what packet should be transmitted on each clock edge on
144 // the data plane tx
145 // provided the reset state is 0
146 always_ff @(posedge clk)
147 begin
148 // if the system is in the reset state
149 // then transmit a packet with all zeros and set all variables and
150 // flags to 0
151 // provided the reset state is 1
152 if(rst == 1'b1)
153 begin
154     data_tx_packet <= 32'h0000;
155     count <= 3'b000;
156     first_flag <= 1'b0;
157     data_tx_complete_flag <= 1'b0;
158 end
159 else
160 begin
161 // if the tx flag has been set then it means the
162 // system can now transmit the packet
163 if(data_tx_flag == 1'b1)
164 begin
165 // this logic is used for the first packet
166 // that is sent
167 // it is special as it sends the current nodes
168 // id in the packet which will be useful for
169 // the rx processor
170 if(first_flag == 1'b0)
171 begin
172 // set the flag so that the next time
173 // the correct packet is sent

```

```

174     first_flag <= 1'b1;
175     // send this packet in this clock
176     // cycle where it contains the dest
177     // node id and the current node id
178     data_tx_packet <= {current_dest_node, node_id};
179 end
180 // if the flag is set then the 1st packet is
181 // sent
182 else
183 begin
184     // send the packet containing the
185     // follwoing format
186     // destination node id and data at the
187     // top of stack
188     data_tx_packet <= {current_dest_node, RAM_tx_data_out};
189
190     // if the count is equal to 3 then all
191     // the packets have been sent
192     // this is because in the
193     // system design we have
194     // decided to use fix packet
195     // length
196     if(count == 3)
197     begin
198         // rest all the flags
199         first_flag <= 1'b0;
200         count <= 0;
201     end
202     // if the count does not equal to
203     // 3 then all the packets have
204     // not been sent to the rx
205     // node
206     else
207     begin
208         count <= count + 1;
209     end
210     if(count == 2)
211     begin
212         // set this flag to reset the
213         // flag at the control plane
214         data_tx_complete_flag <= 1'b1;
215     end
216 end
217 end
218 // if the transmit signal is not given then
219 else
220 begin
221     data_tx_packet <= 32'h0000;
222 end

```

```

223
224 // logic for sending out a pulse for one clock cycle after a packet
225 // has been transmitted
226 // provided the system is not in the reset state
227 if(data_tx_complete_flag == 1'b1)
228 begin
229 // reset the flag
230 data_tx_complete_flag <= 1'b0;
231 end
232 end
233 end
234
235 endmodule

```

Listing B.2: data\_plane\_tx

### data\_plane\_rx.sv

```

1 /*
   ↳ *****
   ↳
2 * File name
3     data_plane_rx.sv
4 * Description
5     This module describes how the data plane rx works.
6 * Parameters
7     NONE
8 * Inputs
9     clk – The clock for the system.
10
11     rst – Signal to reset the system to the default values.
12
13     data_rx_packet – This is the packet that is received on the data plane.
14
15     node_id – This will be the id that will be given to the node in the system. NOTE:
   ↳ EACH NODE MUST HAVE A DIFFERENT ID OTHERWISE THERE WILL BE
   ↳ ISSUES IN TX/RX.
16
17     gpp_rtr_dp – This is a control signal which indicates the data plane to modify the stack
   ↳ pointer. Since the values are stored in a stack data structure, the value at the top of
   ↳ the stack is sent to the GPP from the DP reciever.
18 * Outputs
19     data_rx_complete_flag – This flag will reset the data_rx_flag. The value for this is sent
   ↳ from the data plane reciever.
20
21     RAM_rx_data_out – This shows the data that is outputted from the data plane receiver.
   ↳ It will be connected to the RAM of the GPP via a multiplexer.
22 * Author
23     Sreethyan Aravinthan (UCL)
24 *****/

```

```

25
26 'include "ALU.sv"
27 'include "Data_Memory.sv"
28 'include "Multiplexer.sv"
29 'include "Register.sv"
30
31 module data_plane_rx
32 (
33     input logic clk,
34     input logic rst,
35     input logic [31:0] data_rx_packet,
36     input shortint node_id,
37     input logic gpp_rtr_dp,
38     output logic data_rx_complete_flag,
39     output logic [15:0] RAM_rx_data_out
40 );
41
42 // RAM will be part of the data plane
43 // there will be two lots of RAMs
44 // RAM1: for transmitting data
45 // RAM2: for recieving data
46
47 // variables
48 // this is a control signal that allows data to be written to the RAM
49 // if the received packet is for this node
50 logic rx_enable;
51 // this holds the next sp value. Also is the input for the register
52 logic [15:0] sp_rx_next;
53 // this is the current sp value and it is the output of the register
54 logic [15:0] sp_rx_current;
55 // this will hold current sp value - one
56 logic [15:0] sp_rx_current_plus_one;
57 // this will hold current sp value + one
58 logic [15:0] sp_rx_current_minus_one;
59 // this is used to add to subtract from the stack pointer
60 localparam [15:0] one = 16'h0001;
61 // this is the data that will be written into the RAM
62 logic [15:0] data_rx_current;
63 // this is the RAM address that will be accessed
64 logic [15:0] RAM_address;
65 // this will hold the data that is fed into the sp multiplexer
66 // there are three values
67 // same value as earlier
68 // sp + 1
69 // sp - 1
70 logic [15:0] sp_rx_mux_data [3:0];
71 // this will be the control signals of the sp multiplexer
72 logic [1:0] sp_rx_mux_control;
73 // this will contain the RAM addresses that can be accessed and are

```

```

74 // the inputs to the multiplexer for the RAM address choice
75 logic [15:0] RAM_rx_address_mux_data [1:0];
76 // this is used to see if the number of data packets received is equal to
77 // the fixed packet length
78 logic [2:0] count = 3'b000;
79
80 // wire up the control signal for the stack pointer multiplexer
81 assign sp_rx_mux_control = {gpp_rtr_dp, rx.enable};
82
83 // this combinational logic sets the values for all the multiplexer
84 // data
85 always_comb
86 begin
87 // stack pointer multiplexer data being set
88 sp_rx_mux_data[0] = sp_rx_current;
89 sp_rx_mux_data[1] = sp_rx_current_plus_one;
90 sp_rx_mux_data[2] = sp_rx_current_minus_one;
91
92 // set the RAM address multiplexer data
93 RAM_rx_address_mux_data[0] = sp_rx_current;
94 RAM_rx_address_mux_data[1] = sp_rx_current_plus_one;
95 end
96
97 // instantiate multiplexer to choose the next stack pointer
98 Multiplexer #(2, 16) sp_rx_mux(sp_rx_mux_control, sp_rx_mux_data, sp_rx_next);
99
100 // instantiate register for stack pointer rx
101 Register #(16) sp_rx_reg(clk, rst, sp_rx_next, sp_rx_current);
102
103 // ALU for stack pointer increment to point to the latest data that
104 // has been added
105 ALU #(16) sp_increment(sp_rx_current, one, 2'b00, sp_rx_current_plus_one);
106
107 // ALU for stack pointer decrement. this is needed if the data is
108 // retrived from the RAM
109 ALU #(16) sp_decrement(sp_rx_current, one, 2'b01, sp_rx_current_minus_one);
110
111 // multiplexer for choosing the most appropriate address
112 Multiplexer #(1, 16) RAM_rx_address_mux(rx.enable, RAM_rx_address_mux_data,
    ↪ RAM_address);
113
114 // instantiate RAM modules
115 Data_Memory #(16) RAM_rx (clk, rx.enable, RAM_address, data_rx_current,
    ↪ RAM_rx_data_out);
116
117 // instantiate register for stack pointer rx
118 Register #(16) data_rx_reg(clk, rst, data_rx_packet[15:0], data_rx_current);
119
120 // this logic deals with how to deal with a packet that is recieved

```



```

121 // provided the reset state is 0
122 always_ff @(posedge clk)
123 begin
124 // this logic will deal with what the default values should be
125 // provided reset is set to 1
126 if(rst == 1'b1)
127 begin
128     rx_enable <= 1'b0;
129     count <= 3'b000;
130     data_rx_complete_flag <= 1'b0;
131 end
132 else
133 begin
134 // if the destination id on the rx packet is equal to the node
135 // id then the packet is for this node
136 if(data_rx_packet[31:16] == node_id)
137 begin
138     // save in RAM and change stack pointer value
139     // set the rx_enable flag to 1
140     rx_enable <= 1'b1;
141     // if count is equal to 4 – indicates the entire data
142     // is sent from the source node
143     if(count == 3'b100)
144     begin
145         // reset count
146         count <= 3'b000;
147         // set this flag to 1 thus allowing the the
148         // rx_flag at the control plane to be reset
149         data_rx_complete_flag <= 1'b1;
150     end
151     // if the count is not equal to 4 then the number of
152     // packets recieved is not right
153     else
154     begin
155         // increment count
156         count <= count + 1;
157     end
158 end
159 // if the dest id does not match the node id then set
160 // rx_enable to 0 preventing a write into RAM
161 else
162 begin
163     rx_enable <= 1'b0;
164 end
165 // this logic will deal with ensuring that the data_rx_complete_flag
166 // is set to 1 for only 1 clock cycle if it is set
167 // provided the reset state is 0
168 if(data_rx_complete_flag == 1'b1)
169 begin

```

```

170     // reset the flag
171     data_rx_complete_flag <= 1'b0;
172 end
173 end
174 end
175
176 endmodule

```

Listing B.3: data\_plane\_rx

### data\_plane.sv

```

1  /*
    ↳ *****
    ↳
2  * File name
3    data_plane.sv
4  * Description
5    This module instantiates the data plane tx & rx.
6  * Parameters
7    NONE
8  * Inputs
9    clk — The clock for the system.
10
11    rst — Signal to reset the system to the default values.
12
13    data_rx_packet — This is the packet that is received on the data plane.
14
15    node_id — This will be the id that will be given to the node in the system. NOTE:
    ↳ EACH NODE MUST HAVE A DIFFERENT ID OTHERWISE THERE WILL BE
    ↳ ISSUES IN TX/RX.
16
17    gpp_rtr_dp — This is a control signal which indicates the data plane to modify the stack
    ↳ pointer. Since the values are stored in a stack data structure, the value at the top of
    ↳ the stack is sent to the GPP from the DP receiver.
18
19    gpp_trf_dp — This is a control signal which tells the data plane transmitter that data is
    ↳ going to be transferred from the GPP to the DP transmitter.
20
21    gpp_tx_data — This is the data that will be transferred from the GPP to the DP
    ↳ transmitter RAM.
22
23    data_tx_flag — This is a flag which indicates if the node is transmitting data on the data
    ↳ plane.
24 * Outputs
25    data_rx_complete_flag — This flag will reset the data_rx_flag. The value for this is sent
    ↳ from the data plane receiver.
26
27    RAM_rx_data_out — This shows the data that is outputted from the data plane receiver.
    ↳ It will be connected to the RAM of the GPP via a multiplexer.

```

```

28
29     data_tx_complete_flag – This flag will reset the data_tx_flag. The value for this is sent
    ↪ from the data plane transmitter.
30
31     data_tx_packet – This is the packet that is transmitted on the data plane.
32
33     RAM_tx_data_out – This input has the value equal to the top of the RAM of the data
    ↪ plane tx RAM.
34
35     sp_tx_current – This has the value of the stack pointer from the data plane tx RAM.
    ↪ This is used to see if the data plane tx RAM is empty or not.
36 * Author
37     Sreethyan Aravinthan (UCL)
38 *****/
39
40 module data_plane
41 (
42     // signals for the data plane rx
43
44     input logic clk,
45     input logic rst,
46     input logic [31:0] data_rx_packet,
47     input shortint node_id,
48     input logic gpp_rtr_dp,
49     output logic data_rx_complete_flag, // ***
50     output logic [15:0] RAM_rx_data_out,
51
52     // signals for the data plane tx
53
54     input logic gpp_trf_dp,
55     input logic [15:0] gpp_tx_data,
56     input logic data_tx_flag, // ***
57     output logic data_tx_complete_flag, // ***
58     output logic [31:0] data_tx_packet,
59     output logic [15:0] RAM_tx_data_out, // ***
60     output logic [15:0] sp_tx_current // ***
61 );
62
63 // data plane tx
64 data_plane_tx dp_tx(clk,
65     rst,
66     gpp_trf_dp,
67     gpp_tx_data,
68     node_id,
69     data_tx_flag,
70     data_tx_complete_flag,
71     data_tx_packet,
72     RAM_tx_data_out,
73     sp_tx_current);

```

```

74
75 // data plane rx
76 data_plane_rx dp_rx(clk,
77     rst,
78     data_rx_packet,
79     node_id,
80     gpp_rtr_dp,
81     data_rx_complete_flag,
82     RAM_rx_data_out);
83 endmodule

```

Listing B.4: data\_plane

### comms\_processor.sv

```

1  /*
    ↳ *****
    ↳
2  * File name
3      comms_processor.sv
4  * Description
5      This module instantiates the control plane and data plane. This will be connected to the
    ↳ GPP.
6  * Parameters
7      NONE
8  * Inputs
9      clk — The clock for the system.
10
11      rst — Signal to reset the system to the default values.
12
13      node_id — This will be the id that will be given to the node in the system. NOTE:
    ↳ EACH NODE MUST HAVE A DIFFERENT ID OTHERWISE THERE WILL BE
    ↳ ISSUES IN TX/RX.
14
15      max_node — This shows the maximum number of nodes in the ring topology.
16
17      control_rx_packet — This is the packet that will be recieved on the control plane.
18
19      enable_rtr — This is a control signal that will allow the GPP to retrieve data from the CP
    ↳ if it is possible.
20
21      gpp_rtr_cp — This is a control signal that pauses rx from gpp so that gpp can retrieve
    ↳ data from rx RAM.
22
23      data_rx_packet — This is the packet that is received on the data plane.
24
25      gpp_rtr_dp — This is a control signal which indicates the data plane to modify the stack
    ↳ pointer. Since the values are stored in a stack data structure, the value at the top of
    ↳ the stack is sent to the GPP from the DP reciever.
26

```

```

27     gpp_trf_dp – This is a control signal which tells the data plane transmitter that data is
    ↪ going to be transferred from the GPP to the DP transmitter.
28
29     gpp_tx_data – This is the data that will be transferred from the GPP to the DP
    ↪ transmitter RAM.
30 * Outputs
31     control_tx_packet – This is the packet that is transmitted on the control plane.
32
33     data_rx_node_id – This is the value that is sent to a control unit which sets the
    ↪ wavelength/spatial channel of the reciever.
34
35     data_rx_flag – this is used to show the control plane if data is being received on the data
    ↪ plane. Thus the control plane will not say yes for another ping if set. This is also a
    ↪ flag that is used by the GPP to see if it can retrieve data from the data plane receiver.
    ↪ This will be connected to the ALU_src_1.
36
37     gpp_trf_cp – this is a flag that is used by the GPP to transfer data from the GPP RAM
    ↪ to the data plane transmitter RAM. This will be connected to the ALU_src_1.
38
39     RAM_rx_data_out – This shows the data that is outputted from the data plane receiver.
    ↪ It will be connected to the RAM of the GPP via a multiplexer.
40
41     data_tx_packet – This is the packet that is transmitted on the data plane.
42 * Author
43     Sreethyan Aravinthan (UCL)
44 *****/
45
46 module comms_processor
47 (
48     // commons signals
49     input logic clk,
50     input logic rst,
51     input shortint node_id,
52     // control signals
53     input shortint max_node,
54     input logic [31:0] control_rx_packet,
55     input logic enable_rtr, // cu
56     input logic gpp_rtr_cp, // cu
57     output logic [31:0] control_tx_packet,
58     output logic [15:0] data_rx_node_id,
59     output logic data_rx_flag, // gpp
60     output logic gpp_trf_cp, // gpp
61     // data signals
62     input logic [31:0] data_rx_packet,
63     input logic gpp_rtr_dp, // cu
64     output logic [15:0] RAM_rx_data_out, // gpp
65     input logic gpp_trf_dp, // cu
66     input logic [15:0] gpp_tx_data, // gpp
67     output logic [31:0] data_tx_packet

```

```

68 );
69
70 // connected signals
71 logic data_rx_complete_flag;
72 logic data_tx_complete_flag;
73 logic [15:0] RAM_tx_data_out;
74 logic [15:0] sp_tx_current;
75 logic data_tx_flag;
76
77 // instantiate control plane
78 Control_Plane cp(clk,
79     rst,
80     node_id,
81     max_node,
82     data_rx_complete_flag, // ***
83     control_rx_packet,
84     data_tx_complete_flag, // ***
85     RAM_tx_data_out, // ***
86     sp_tx_current, // ***
87     enable_rtr,
88     gpp_rtr_cp,
89     control_tx_packet,
90     data_rx_node_id,
91     data_tx_flag, // ***
92     data_rx_flag,
93     gpp_trf_cp);
94
95 // instantiate data plane
96 data_plane dp(clk,
97     rst,
98     data_rx_packet,
99     node_id,
100     gpp_rtr_dp,
101     data_rx_complete_flag, // ***
102     RAM_rx_data_out,
103     gpp_trf_dp,
104     gpp_tx_data,
105     data_tx_flag, // ***
106     data_tx_complete_flag, // ***
107     data_tx_packet,
108     RAM_tx_data_out, // ***
109     sp_tx_current); // ***
110
111 endmodule

```

Listing B.5: comms\_processor

## Computer.sv

```

1 /*

```

```

    ↪ *****
    ↪
2 * File name
3     Computer.sv
4 * Description
5     This module instantiates all the modules needed and creates the microprocessor.
6 * Parameters
7     NONE
8 * Inputs
9     clk – The clock for the system.
10
11     rst – Signal to reset the system to the default values.
12
13     node_id – This will be the id that will be given to the node in the system. NOTE:
    ↪ EACH NODE MUST HAVE A DIFFERENT ID OTHERWISE THERE WILL BE
    ↪ ISSUES IN TX/RX.
14
15     max_node – This shows the maximum number of nodes in the ring topology.
16
17     control_rx_packet – This is the packet that will be recieved on the control plane.
18
19     data_rx_packet – This is the packet that is received on the data plane.
20 * Outputs
21     control_tx_packet – This is the packet that is transmitted on the control plane.
22
23     data_rx_node_id – This is the value that is sent to a control unit which sets the
    ↪ wavelength/spatial channel of the reciever.
24
25     data_tx_packet – This is the packet that is transmitted on the data plane.
26 * Author
27     Sreethyan Aravinthan (UCL)
28 *****/
29
30 module Computer
31 (
32     input logic clk,
33     input logic rst,
34     input shortint node_id,
35     // control plane signals
36     input shortint max_node,
37     input logic [31:0] control_rx_packet,
38     output logic [31:0] control_tx_packet,
39     output logic [15:0] data_rx_node_id,
40     // data plane signals
41     input logic [31:0] data_rx_packet,
42     output logic [31:0] data_tx_packet
43 );
44
45     // Instruction memory

```

```

46 logic [15:0]address_1;
47 logic [15:0]address_2;
48 logic [15:0]read_address_1;
49 logic [15:0]read_address_2;
50 // Data memory
51 logic [15:0]address_rw;
52 logic [15:0]data_in;
53 logic [15:0]data_out;
54 // Control Signals
55 logic memory_write_enable;
56 logic enable_rtr;
57 logic gpp_rtr_cp;
58 logic gpp_rtr_dp;
59 logic gpp_trf_dp;
60 // Communications Processor Signals
61 logic [15:0] RAM_rx_data_out; // gpp
62 logic data_rx_flag; // gpp
63 logic gpp_trf_cp; // gpp
64 logic [15:0] gpp_tx_data; // gpp
65
66 GPP cpu(clk,
67         rst,
68         address_1,
69         address_2,
70         read_address_1,
71         read_address_2,
72         address_rw,
73         data_in,
74         data_out,
75         memory_write_enable,
76         enable_rtr,
77         gpp_rtr_cp,
78         gpp_rtr_dp,
79         gpp_trf_dp,
80         RAM_rx_data_out, // gpp
81         data_rx_flag, // gpp
82         gpp_trf_cp, // gpp
83         gpp_tx_data); // gpp
84
85 comms_processor cp (clk,
86                    rst,
87                    node_id,
88                    max_node,
89                    control_rx_packet,
90                    enable_rtr, // cu
91                    gpp_rtr_cp, // cu
92                    control_tx_packet,
93                    data_rx_node_id,
94                    data_rx_flag, // gpp

```



```

95         gpp_trf_cp, // gpp
96         data_rx_packet,
97         gpp_rtr_dp, // cu
98         RAM_rx_data_out, // gpp
99         gpp_trf_dp, // cu
100        gpp_tx_data, // gpp
101        data_tx_packet);
102
103    Instruction_Memory #(16, 16, "test.mem") im_computer (address_1,
104        address_2,
105        read_address_1,
106        read_address_2);
107
108    Data_Memory #(16) dm_computer (clk,
109        memory_write_enable,
110        address_rw,
111        data_in,
112        data_out);
113
114
115    endmodule

```

Listing B.6: Top-level module showing how the GPP CP RAM and ROM are linked together

# Appendix C

## Testbenches

### **gpp\_tb.sv**

```
1 module Computer_tb;
2
3   logic clk;
4   logic rst;
5
6   Computer dut(clk, rst);
7
8   initial
9   begin
10      clk = 0;
11      forever #50ps clk = ~clk;
12   end
13
14   initial
15   begin
16      rst = 1'b1;
17      #100ps;
18      rst = 1'b0;
19   end
20
21 endmodule
```

Listing C.1: Testbench for GPP

### **comms\_processor\_tx\_tb.sv**

```
1 module comms_processor_tb;
2
3   // commons signals
4   logic clk;
5   logic rst;
6   shortint node_id;
7   // control signals
```

```

8  shortint max_node;
9  logic [31:0] control_rx_packet;
10 logic enable_rtr;
11 logic gpp_rtr_cp;
12 logic [31:0] control_tx_packet;
13 logic [15:0] data_rx_node_id;
14 logic data_rx_flag;
15 logic gpp_trf_cp; // ALU src 1
16 // data signals
17 logic [31:0] data_rx_packet;
18 logic gpp_rtr_dp;
19 logic [15:0] RAM_rx_data_out;
20 logic gpp_trf_dp;
21 logic [15:0] gpp_tx_data;
22 logic [31:0] data_tx_packet;
23
24 comms_processor dut (clk,
25     rst,
26     node_id,
27     max_node,
28     control_rx_packet,
29     enable_rtr,
30     gpp_rtr_cp,
31     control_tx_packet,
32     data_rx_node_id,
33     data_rx_flag,
34     gpp_trf_cp, // ALU src 1
35     data_rx_packet,
36     gpp_rtr_dp,
37     RAM_rx_data_out,
38     gpp_trf_dp,
39     gpp_tx_data,
40     data_tx_packet);
41
42 initial
43 begin
44     clk = 0;
45     forever #50ps clk = ~clk;
46 end
47
48 initial
49 begin
50     // set constant values
51     node_id = 1;
52     max_node = 4;
53     // set default values
54     control_rx_packet = 32'h00000000;
55     enable_rtr = 1'b0;
56     gpp_rtr_cp = 1'b0;

```

```

57 data_rx_packet = 32'h00000000;
58 gpp_rtr_dp = 1'b0;
59 gpp_trf_dp = 1'b0;
60 gpp_tx_data = 16'h0000;
61 rst = 1'b1;
62 #100ps;
63 rst = 1'b0;
64 #100ps;
65
66 // TX testing
67 #100ps;
68 #100ps;
69 gpp_trf_dp = 1'b1;
70 gpp_tx_data = 16'h000A;
71 #100ps;
72 gpp_tx_data = 16'h000B;
73 #100ps;
74 gpp_tx_data = 16'h000C;
75 #100ps;
76 gpp_tx_data = 16'h000D;
77 #100ps;
78 gpp_tx_data = 16'h0004;
79 #100ps;
80 gpp_trf_dp = 1'b0;
81 #100ps;
82 #100ps;
83 #100ps;
84 #100ps;
85 #100ps;
86 #100ps;
87 #50ps;
88 control_rx_packet = 32'h0001FFFF;
89 #100ps;
90 control_rx_packet = 32'h00000000;
91 #100ps;
92 end
93
94 endmodule

```

Listing C.2: Testbench for communications processor transmitter

#### **comms\_processor\_rx\_tb.sv**

```

1 module comms_processor_tb;
2
3 // commons signals
4 logic clk;
5 logic rst;
6 shortint node_id;
7 // control signals

```

```

8  shortint max_node;
9  logic [31:0] control_rx_packet;
10 logic enable_rtr;
11 logic gpp_rtr_cp;
12 logic [31:0] control_tx_packet;
13 logic [15:0] data_rx_node_id;
14 logic data_rx_flag;
15 logic gpp_trf_cp; // ALU src 1
16 // data signals
17 logic [31:0] data_rx_packet;
18 logic gpp_rtr_dp;
19 logic [15:0] RAM_rx_data_out;
20 logic gpp_trf_dp;
21 logic [15:0] gpp_tx_data;
22 logic [31:0] data_tx_packet;
23
24 comms_processor dut (clk,
25     rst,
26     node_id,
27     max_node,
28     control_rx_packet,
29     enable_rtr,
30     gpp_rtr_cp,
31     control_tx_packet,
32     data_rx_node_id,
33     data_rx_flag,
34     gpp_trf_cp, // ALU src 1
35     data_rx_packet,
36     gpp_rtr_dp,
37     RAM_rx_data_out,
38     gpp_trf_dp,
39     gpp_tx_data,
40     data_tx_packet);
41
42 initial
43 begin
44     clk = 0;
45     forever #50ps clk = ~clk;
46 end
47
48 initial
49 begin
50     // set constant values
51     node_id = 1;
52     max_node = 4;
53     // set default values
54     control_rx_packet = 32'h00000000;
55     enable_rtr = 1'b0;
56     gpp_rtr_cp = 1'b0;

```

```

57 data_rx_packet = 32'h00000000;
58 gpp_rtr_dp = 1'b0;
59 gpp_trf_dp = 1'b0;
60 gpp_tx_data = 16'h0000;
61 rst = 1'b1;
62 #100ps;
63 rst = 1'b0;
64 #100ps;
65
66 // RX testing
67 #100ps;
68 #100ps;
69 #50ps;
70 control_rx_packet = 32'h00010004;
71 #100ps;
72 control_rx_packet = 32'h00000000;
73 data_rx_packet = 32'h00010004;
74 #100ps;
75 data_rx_packet = 32'h0001000D;
76 #100ps;
77 data_rx_packet = 32'h0001000C;
78 #100ps;
79 data_rx_packet = 32'h0001000B;
80 #100ps;
81 data_rx_packet = 32'h0001000A;
82 #100ps;
83 data_rx_packet = 32'h00000000;
84 enable_rtr = 1'b1;
85 gpp_rtr_cp = 1'b1;
86 #100ps;
87 gpp_rtr_dp = 1'b1;
88 control_rx_packet = 32'h00010004;
89 #100ps;
90 end
91
92 endmodule

```

Listing C.3: Testbench for communications processor receiver

### Computer\_tx\_tb.sv

```

1 module Computer_tx_tb;
2
3 logic clk;
4 logic rst;
5 shortint node_id;
6 shortint max_node;
7 logic [31:0] control_rx_packet;
8 logic [31:0] control_tx_packet;
9 logic [15:0] data_rx_node_id;

```

```

10 logic [31:0] data_rx_packet;
11 logic [31:0] data_tx_packet;
12
13 Computer dut (clk,
14     rst,
15     node_id,
16     max_node,
17     control_rx_packet,
18     control_tx_packet,
19     data_rx_node_id,
20     data_rx_packet,
21     data_tx_packet);
22
23 initial
24 begin
25     clk = 0;
26     forever #50ps clk = ~clk;
27 end
28
29 initial
30 begin
31     node_id = 1;
32     max_node = 4;
33     rst = 1'b1;
34     #100ps;
35     rst = 1'b0;
36     #100ps;
37     #100ps;
38     #100ps;
39     #100ps;
40     #100ps;
41     #100ps;
42     #100ps;
43     #100ps;
44     #100ps;
45     #100ps;
46     #100ps;
47     #100ps;
48     #100ps;
49     #100ps;
50     #50ps;
51     control_rx_packet = 32'h0001FFFF;
52     #100ps;
53     control_rx_packet = 32'h00000000;
54     #100ps;
55 end
56

```

57 **endmodule**

Listing C.4: Testbench for microprocessor transmitter

### **Computer\_rx\_tb.sv**

```
1 module Computer_rx_tb;
2
3   logic clk;
4   logic rst;
5   shortint node_id;
6   shortint max_node;
7   logic [31:0] control_rx_packet;
8   logic [31:0] control_tx_packet;
9   logic [15:0] data_rx_node_id;
10  logic [31:0] data_rx_packet;
11  logic [31:0] data_tx_packet;
12
13  Computer dut (clk,
14               rst,
15               node_id,
16               max_node,
17               control_rx_packet,
18               control_tx_packet,
19               data_rx_node_id,
20               data_rx_packet,
21               data_tx_packet);
22
23  initial
24  begin
25      clk = 0;
26      forever #50ps clk = ~clk;
27  end
28
29  initial
30  begin
31      node_id = 1;
32      max_node = 4;
33      rst = 1'b1;
34      #100ps;
35      rst = 1'b0;
36      #300ps;
37      control_rx_packet = 32'h00010004;
38      #100ps;
39      control_rx_packet = 32'h00000000;
40      #100ps;
41      data_rx_packet = 32'h00010004;
42      #100ps;
43      data_rx_packet = 32'h0001000A;
44      #100ps;
```



```
45 data_rx_packet = 32'h0001000B;  
46 #100ps;  
47 data_rx_packet = 32'h0001000C;  
48 #100ps;  
49 data_rx_packet = 32'h0001000D;  
50 #100ps;  
51 data_rx_packet = 32'h00000000;  
52 #100ps;  
53 end  
54  
55  
56 endmodule
```

Listing C.5: Testbench for microprocessor reciever

# Appendix D

## Assembly programs used for testing

### **fib\_normal.asm**

```
1  movi reg1, #0
2  movi reg2, #1
3  loop:
4  mov reg2, reg3
5  add reg1, reg2 ; reg2 = reg1 + reg2
6  mov reg3, reg1
7  addi reg1, #0
8  jmp loop
```

Listing D.1: Assembly code going through Fibonacci sequence using move instructions

### **fib\_stack.asm**

```
1  movi reg0, #0
2  movi reg1, #1
3  movi reg2, #1
4  loop:
5  push reg2
6  add reg1, reg2
7  pop reg1
8  addi reg2, #0
9  jmp loop
```

Listing D.2: Assembly code going through Fibonacci sequence using stack instructions

### **division.asm**

```
1  movi reg0, #0
2  movi reg1, #10
3  movi reg2, #2
```

```

4  call division
5  jmp end
6  division:
7  ; pass into reg1 the dividend
8  ; pass into reg2 the divisor
9  push reg3 ; use reg3 as a check therefore save value
10 push reg4 ; reg4 will hold the quotient
11 mov reg2, reg3 ; move the divisor
12 movi reg4, #0 ; set the quotient to zero initially
13 divide:
14 sub reg1, reg2 ; dividend_cur(reg2) = dividend_prev(reg1) - divisor(reg2)
15 mov reg2, reg1 ; dividend_prev(reg1) = dividend_cur(reg2)
16 mov reg3, reg2 ; dividend_cur(reg2) = reg3
17 addi reg4, #1 ; increment quotient by 1
18 cmp reg1, reg3 ; compare reg1 to reg3
19 jae divide ; if above or equal
20 jmp divide_end ; else quit the sub routine
21 divide_end:
22 mov reg4, reg2 ; move the quotient into reg2
23 ; reg1 contains the remainder
24 pop reg4
25 pop reg3 ; retrieve old value of reg3
26 return
27 end:
28 addi reg2, #0
29 addi reg1, #0
30 jmp end

```

Listing D.3: Assembly code for a software implmenation of division

### **prime\_number.asm**

```

1  movi reg0, #0 ; initialise the stack pointer
2  movi reg5, #11 ; value to determine if prime
3  call check_prime
4  jmp end
5  check_prime:
6  movi reg6, #2 ; this is the 1st number that will divide the to.be_prime_number
7  movi reg7, #1 ; move 1
8  sub reg5, reg7 ; this should hold the upper limit
9  movi reg8, #0
10 continue:
11 mov reg5, reg1 ; move dividend into reg1
12 mov reg6, reg2 ; move divisor into reg2
13 call division ; divide
14 cmp reg1, reg8 ; check if remainder is 0
15 jz not_prime ; if remainder is 0 then not a prime so flag output is 0
16 addi reg6, #1 ; increment for the next for the next divisor
17 cmp reg6, reg5 ; check if the divisor is equal to the dividend
18 jz end_search ; if the above result is 0 then end the search and flag it as zero

```

```

19  jmp continue ; else continue
20 not_prime:
21  movi reg10, #0
22  return
23 end_search:
24  movi reg10, #1
25  return
26 division:
27  ; pass into reg1 the dividend
28  ; pass into reg2 the divisor
29  push reg3 ; use reg3 as a check therefore save value
30  push reg4 ; reg4 will hold the quotient
31  mov reg2, reg3 ; move the divisor
32  movi reg4, #0 ; set the quotient to zero initially
33 divide:
34  sub reg1, reg2 ; dividend_cur(reg2) = dividend_prev(reg1) - divisor(reg2)
35  mov reg2, reg1 ; dividend_prev(reg1) = dividend_cur(reg2)
36  mov reg3, reg2 ; dividend_cur(reg2) = reg3
37  addi reg4, #1 ; increment quotient by 1
38  cmp reg1, reg3 ; compare reg1 to reg3
39  jae divide ; if above or equal
40  jmp divide_end ; else quit the sub routine
41 divide_end:
42  mov reg4, reg2 ; move the quotient into reg2
43  ; reg1 contains the remainder
44  pop reg4
45  pop reg3 ; retrieve old value of reg3
46  return
47 end:
48  addi reg10, #0
49  jmp end

```

Listing D.4: Assembly code for checking if a number is a prime number or not

#### sample\_tx.asm

```

1  movi reg0, #0
2  movi reg1, #10
3  movi reg2, #11
4  movi reg3, #12
5  movi reg4, #13
6  movi reg5, #4
7  push reg5
8  push reg4
9  push reg3
10 push reg2
11 push reg1
12 movi reg6, #0
13 movi reg6, #0
14 movi reg6, #0

```

```

15  movi reg6, #0
16  ; compare the value to see if data cane be transferred
17  cbt
18  ; if the result is not equal to zero then data cannot be transfered
19  jz tranf
20  jmp exit
21  ; transfer 5 packet data
22 tranf:
23  trf
24  trf
25  trf
26  trf
27  trf
28 exit:
29  movi reg6, #0

```

Listing D.5: Assembly code for microprocessor transmitter

#### **sample\_rx.asm**

```

1  movi reg0, #0
2  movi reg6, #0
3  movi reg6, #0
4  movi reg6, #0
5  movi reg6, #0
6  movi reg6, #0
7  movi reg6, #0
8  movi reg6, #0
9  movi reg6, #0
10 movi reg6, #0
11 movi reg6, #0
12 movi reg6, #0
13 movi reg6, #0
14 movi reg6, #0
15 movi reg6, #0
16 pr
17 cbr
18 jz exit
19 rtr
20 rtr
21 rtr
22 rtr
23 rtr
24 exit:
25 rr

```

Listing D.6: Assembly code for microprocessor reciever

# Appendix E

## Assembler code

**main.c**

```
1  /*
   ↪ *****
   ↪
2  * File name
3    main.c
4  * Description
5    This is the main program file for the assembler.
6  * Author
7    Sreethyan Aravinthan (UCL)
8  *****/
9
10
11 // standard header files
12 #include <stdio.h>
13 #include <string.h>
14
15 // personal header files
16 #include "assembly.h"
17 #include "preprocess.h"
18 #include "extra.h"
19
20 void combine_files(int argc, char * argv[], char out_file)
21 {
22     // open the combined file in write mode
23     FILE * combine = xfopen("combined.asm", "w");
24     // this will be the pointer of the read file
25     FILE * read_file;
26     // this will hold the current file that is read from the current file
27     char line[1000];
28     // go through all the files and combine into a single new file
29     for(int i = 1; i < argc - out_file; i++)
30     {
```

```

31 // open the file
32 read_file = xopen(argv[i], "r");
33 // read until EOF
34 while(fgets(line, 1000, read_file) != NULL)
35 {
36     // write to new combined file
37     fprintf(combine, "%s", line);
38 }
39 // close current file and then go to the next file if it exists
40 fclose(read_file);
41 }
42 // close the combined file
43 fclose(combine);
44 }
45
46 int main(int argc, char * argv[])
47 {
48     char * file_name;
49     char out_file = 1;
50
51     // check if three arguments are supplied
52     if(argc < 2)
53     {
54         printf("Too few arguements\nNeed atleast two arguements\n");
55         return 0;
56     }
57     else // one file assembly
58     {
59         // check to see if the output file is supplied
60         if(strstr(argv[argc - 1], ".mem") == NULL)
61         {
62             printf("No output file given\nSetting output file as out.mem\n");
63             out_file = 0;
64         }
65         // combine all input files
66         combine_files(argc, argv, out_file);
67     }
68     // remove the comments – creates a file called no_comments.asm
69     remove_comments("combined.asm");
70     // remove and replace with addresses
71     remove_label();
72     if(out_file == 1)
73     {
74         // convert from mnemonics to machine code
75         convert_to_machine_code(argv[argc - 1]);
76     }
77     else
78     {
79         // convert from mnemonics to machine code

```

```

80     convert_to_machine_code("out.mem");
81 }
82 // remove temporary files
83 remove("combined.asm");
84 remove("no_comments.asm");
85 remove("no_labels.asm");
86 return 0;
87 }

```

Listing E.1: This is the main program file that calls the appropriate functions to convert the assembly code to machine code

### preprocess.h

```

1  /*
    ↪ *****
    ↪
2  * File name
3      preprocess.h
4  * Description
5      Header file preprocess.h
6  * Author
7      Sreethyan Aravinthan (UCL)
8  *****/
9
10 /*
11  * Design Unit:
12  * File name:
13  * Description:
14  * Author:
15  * Version:
16 */
17
18 #ifndef _PREPROCESS_H_
19 #define _PREPROCESS_H_
20
21 #define NUM_OF_LABELS    100
22 #define NUM_OF_ADDRESSES 100
23 #define LENGTH_OF_LABEL  100
24 #define LENGTH_OF_INSTR  30
25 #define SPACE            32
26 #define TAB              9
27 #define NEW_LINE         10
28 #define UPPER_A          65
29 #define UPPER_Z          90
30 #define LOWER_A          97
31 #define LOWER_Z          122
32 #define UNDERSCORE       95
33 #define ZERO             48
34 #define NINE             57

```



```

35
36 typedef struct
37 {
38     // multi-array which holds all the labels in the assembly file
39     char labels[NUM_OF_LABELS][LENGTH_OF_LABEL];
40
41     // array which holds the address of each label
42     unsigned int label_address[NUM_OF_ADDRESSES];
43
44     // this indicates the number of labels/addresses that were filled
45     int index;
46 }labels_and_addresses_file;
47
48 // functions
49 void remove_comments(char * file_name);
50 void remove_label(void);
51
52 #endif

```

Listing E.2: This is the header file for all the preprocessing functions

### preprocess.c

```

1  /*
    ↪ *****
    ↪
2  * File name
3      preprocess.c
4  * Description
5      This file has functions which remove comments and labels were.
6  * Author
7      Sreethyan Aravinthan (UCL)
8  *****/
9
10 // standard header files
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14
15 // personal header files
16 #include "preprocess.h"
17 #include "extra.h"
18
19 // remove the comments from the code and create a new file
20 // lets call the file no_comments.asm
21 void remove_comments(char * file_name)
22 {
23     FILE * original_code = xfopen(file_name, "r");
24     FILE * no.comment_code = xfopen("no_comments.asm", "w");
25     char line[100]; // stores current line read

```

```

26 char * target = ";";
27 char * occur = NULL;
28
29 while(fgets(line, 100, original_code) != NULL)
30 {
31     occur = strstr(line, target);
32
33     // check to see if the pointer is not NULL indicating that this line has a ;
34     if(occur != NULL)
35     {
36         // remove the ;
37         // add a \n and \0
38         *occur = '\n';
39         *(occur + 1) = '\0';
40
41         // provided the format is not a tab then new line then save line in file
42         if(strcmp(line, " \n") != 0 && strcmp(line, "\n") != 0)
43         {
44             fprintf(no_comment_code, "%s", line);
45         }
46     }
47     // if the pointer is null then the line has no comment therefore write it in
48     else
49     {
50         fprintf(no_comment_code, "%s", line);
51     }
52 }
53
54 // close all the files
55 fclose(original_code);
56 fclose(no_comment_code);
57 }
58
59 void remove_label(void)
60 {
61     FILE * no_comment_code = xfopen("no_comments.asm", "r");
62     FILE * no_labels_code = xfopen("no_labels.asm", "w");
63     char line[LENGTH_OF_INSTR]; // hold the current line that was read from the file
64
65     // hold the instructions that will take two 16 bit values in instruction memory in a multi-
66     ↪ dimensional array
67     char two_space_instr[16][5];
68     // copy the values in
69     strcpy(two_space_instr[0], "jz");
70     strcpy(two_space_instr[1], "jmp");
71     strcpy(two_space_instr[2], "movi");
72     strcpy(two_space_instr[3], "addi");
73     strcpy(two_space_instr[4], "subi");
74     strcpy(two_space_instr[5], "andi");

```

```

74 strcpy(two_space_instr[6], "ori");
75 strcpy(two_space_instr[7], "call");
76 strcpy(two_space_instr[8], "jb");
77 strcpy(two_space_instr[9], "jbe");
78 strcpy(two_space_instr[10], "ja");
79 strcpy(two_space_instr[11], "jae");
80 strcpy(two_space_instr[12], "jg");
81 strcpy(two_space_instr[13], "jge");
82 strcpy(two_space_instr[14], "jl");
83 strcpy(two_space_instr[15], "jle");
84
85 // this will hold all the labels and their addresses
86 labels_and_addresses_file file_data;
87 // initialise the index field to 0
88 file_data.index = 0;
89
90 char * colon = ":";
91 char * occur = NULL;
92 // this holds the address of the current instruction
93 unsigned short int address = 0;
94
95 // find the labels and the corresponding address
96 while(fgets(line, LENGTH_OF_INSTR, no_comment_code) != NULL)
97 {
98     // check if the line has a colon
99     occur = strstr(line, colon);
100
101     // if the line does have a colon get the label
102     if(occur != NULL)
103     {
104         // place null terminator at the same place as colon
105         *occur = '\0';
106         // copy the label into the array
107         strcpy(file_data.labels[file_data.index], line);
108         // save the address of this label as well
109         file_data.label_address[file_data.index] = address;
110         // increment index for next label
111         file_data.index++;
112     }
113     // else calculate the next address
114     else
115     {
116         // flag to indicate if the address needs to be jumped by 2
117         char plus_2 = 0;
118
119         // check to see if the current instruction has an instruction which takes up 2 16 bit fields
120         for(int i = 0; i < 16; i++)
121         {
122             if(strstr(line, two_space_instr[i]) != NULL)

```

```

123     {
124         plus_2 = 1;
125         break;
126     }
127 }
128
129 // if yes add 2 or else add 1
130 if(plus_2 == 1)
131 {
132     address += 2;
133 }
134 else
135 {
136     address++;
137 }
138 }
139 }
140
141 // go back to the begining of the file
142 rewind(no_comment_code);
143
144 // get the line
145 // see if it has a :
146 // if yes do nothing
147 // if no (NULL) then check if it has a ,
148 // if yes then print line to file
149 // if no then the instruction can be jump, call, one operand(pop) or zero opernad
150 // ↪ instructions(cbt)
151 // get the instruction and compare it to the list of jump and call
152 // if no print the line to file
153 // if yes find the address of the label and replace it and write to file
154
155 char * comma = ",";
156 char has_label = 0;
157
158 char label_instr[11][5];
159 strcpy(label_instr[0], "jz");
160 strcpy(label_instr[1], "jmp");
161 strcpy(label_instr[2], "call");
162 strcpy(label_instr[3], "jb");
163 strcpy(label_instr[4], "jbe");
164 strcpy(label_instr[5], "ja");
165 strcpy(label_instr[6], "jae");
166 strcpy(label_instr[7], "jg");
167 strcpy(label_instr[8], "jge");
168 strcpy(label_instr[9], "jl");
169 strcpy(label_instr[10], "jle");
170
171 char ** words_in_line;

```

```

171 int num_of_words = 0;
172 int num_of_frees = 0;
173 char * shifted_line = NULL;
174
175 // remove all the labels and replace it with the address
176 // also remove all the labels that has a colon next to it
177 while(fgets(line, LENGTH_OF_INSTR, no_comment_code) != NULL)
178 {
179     shifted_line = shift_left_one(line);
180
181     occur = strstr(shifted_line, colon);
182
183     // this line is not a label
184     if(occur == NULL)
185     {
186         // find out if a comma exists in the line
187         occur = strstr(shifted_line, comma);
188
189         // this means that this is a two operand instruction and they do not have labels. Simply
190         ↪ print the line into the file
191         if(occur != NULL)
192         {
193             fprintf(no_labels_code, "%s", shifted_line);
194         }
195         // the instruction can be jump, call, one operand(pop) or zero operand instructions(cbt)
196         else
197         {
198             words_in_line = get_words_from_string(shifted_line, &num_of_words, &num_of_frees);
199             // go through the instructions and see if it has 2 space instructions
200             for(int i = 0; i < 16; i++)
201             {
202                 if(strcmp(words_in_line[0], label_instr[i]) == 0)
203                 {
204                     has_label = 1;
205                     break;
206                 }
207             }
208
209             // free the allocated memory
210             free_split(words_in_line, num_of_frees);
211
212             // this line of code does not have a label as an operand
213             if(has_label == 0)
214             {
215                 fprintf(no_labels_code, "%s", shifted_line);
216             }
217             // this line of code does have a label as an operand
218             else if(has_label == 1)
219             {

```

```

219     // get the words from the line of code
220     words_in_line = get_words_from_string(line, &num_of_words, &num_of_frees);
221     fprintf(no_labels_code, "%s ", words_in_line[0]);
222     // now go through all the labels and find the address and replace it with its address
223     for(int j = 0; j < file_data.index; j++)
224     {
225         if(strcmp(file_data.labels[j], words_in_line[1]) == 0)
226         {
227             fprintf(no_labels_code, "%d\n", file_data.label_address[j]);
228             break;
229         }
230     }
231     // free the allocated memory
232     free_split(words_in_line, num_of_frees);
233 }
234 has_label = 0;
235 }
236 }
237 }
238
239 fclose(no_comment_code);
240 fclose(no_labels_code);
241 }

```

Listing E.3: This is the source file for all the preprocessing functions

## assembly.h

```

1  /*
   ↳ *****
   ↳
2  * File name
3      assembly.h
4  * Description
5      Header file assembly.h
6  * Author
7      Sreethyan Aravinthan (UCL)
8  *****/
9
10 #ifndef _ASSEMBLY_H_
11 #define _ASSEMBLY_H_
12
13 #define NUM_OF_INSTR    33
14 #define MNEMONIC_LENGTH  10
15 #define OPCODE_LENGTH   6
16 #define NUM_OF_REG      32
17 #define REG_LENGTH      6
18
19 typedef struct
20 {

```

```

21 char mnemonic[NUM_OF_INSTR][MNEMONIC_LENGTH];
22 char * opcode[NUM_OF_INSTR];
23 }instructions;
24
25 typedef struct
26 {
27     char reg[NUM_OF_REG][REG_LENGTH];
28     char * operand[NUM_OF_REG];
29 }registers;
30
31 // functions
32 void convert_to_machine_code(char * output_file);
33 void get_list_of_instr(instructions * list_of_instr);
34 void get_list_of_regs(registers * list_of_regs);
35 void free_all_opcodes_and_operands(instructions * list_of_instr, registers * list_of_regs);
36 char * reg_operand_get(char * reg, registers * list_of_regs);
37 char * two_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions *
    ↪ list_of_instr);
38 void label_operands(char * code, FILE * machine_code, instructions * list_of_instr);
39 char * immediate_operands(char * code, FILE * machine_code, registers * list_of_regs,
    ↪ instructions * list_of_instr);
40 char * stack_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions *
    ↪ list_of_instr);
41 void return_instr();
42 void comms_instr(FILE * machine_code, char * instr, instructions * list_of_instr);
43 void clean_up(FILE * no_labels_code, FILE * machine_code, instructions * list_of_instr,
    ↪ registers * list_of_regs);
44
45 #endif

```

Listing E.4: This is the header file for all the functions that are responsible for converting the preprocessed assembly code into machine code

### assembly.c

```

1 /*
    ↪ *****
    ↪
2 * File name
3     assembly.c
4 * Description
5     This file converts the preprocessed file to machine code
6 * Author
7     Sreethyan Aravinthan (UCL)
8 *****/
9
10 // standard header files
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>

```

```

14
15 // personal header files
16 #include "assembly.h"
17 #include "extra.h"
18
19 void get_list_of_instr(instructions * list_of_instr)
20 {
21     // open file which contains all the instructions that can run on
22     // the processor
23     FILE * instructions_file = xfopen("/mnt/d/Documents/University/Year_3/Project/Solution/
        ↪ Assembler/Software/src/instructions.txt", "r");
24
25     // read from the file and store it in the array field called mnemonic
26     for(unsigned short int index = 0; index < NUM_OF_INSTR; index++)
27     {
28         fscanf(instructions_file, "%s", list_of_instr->mnemonic[index]);
29     }
30
31     // close the file
32     fclose(instructions_file);
33 }
34
35 void get_list_of_regs(registers * list_of_regs)
36 {
37     // open file which contains all the registers that are part of the processor
38     FILE * registers_file = xfopen("/mnt/d/Documents/University/Year_3/Project/Solution/
        ↪ Assembler/Software/src/registers.txt", "r");
39
40     // read from the file and store it in the array field called reg
41     for(unsigned int index = 0; index < NUM_OF_REG; index++)
42     {
43         fscanf(registers_file, "%s\n", list_of_regs->reg[index]);
44     }
45
46     // close the file
47     fclose(registers_file);
48 }
49
50 void free_all_opcodes_and_operands(instructions * list_of_instr, registers * list_of_regs)
51 {
52     // free the opcodes
53     for(int i = 0; i < NUM_OF_INSTR; i++)
54     {
55         free(list_of_instr->opcode[i]);
56         list_of_instr->opcode[i] = NULL;
57     }
58
59     // free the operands
60     for(int i = 0; i < NUM_OF_REG; i++)

```



```

61 {
62     free(list_of_regs->operand[i]);
63     list_of_regs->operand[i] = NULL;
64 }
65 }
66
67 void convert_to_machine_code(char * output_file)
68 {
69     // open the file called no_labels in read mode
70     FILE * no_labels_code = fopen("no_labels.asm", "r");
71
72     // open the file called machine code in write mode which contains the machine code
73     // that will run on the processor
74     FILE * machine_code = fopen(output_file, "w");
75
76     // create memory which holds the mnemonics and the corresponding opcodes
77     instructions * list_of_instr = (instructions *) xmalloc(sizeof(instructions));
78     // load mnemonics that are part of the processor
79     get_list_of_instr(list_of_instr);
80
81     // convert each opcode from decimal to binary and store in the opcode field
82     for(unsigned short int index = 0; index < NUM_OF_INSTR; index++)
83     {
84         list_of_instr->opcode[index] = convert_to_binary(index, 0, 6);
85     }
86
87     // create memory which holds the registers and the corresponding operands in binary format
88     registers * list_of_regs = (registers *)xmalloc(sizeof(registers));
89     // load the registers and get the operands for each registers
90     get_list_of_regs(list_of_regs);
91
92     // convert each operand from decimal to binary and store in the operand field
93     for(unsigned short int index = 0; index < NUM_OF_REG; index++)
94     {
95         list_of_regs->operand[index] = convert_to_binary(index, 0, 5);
96     }
97
98     char line[100];
99     int num_of_words = 0;
100    int num_of_frees = 0;
101    char ** words_in_line;
102    int instr = 0;
103
104    // get the current line
105    // get the words from it
106    // get the instruction and find out which one it is
107    // use that index for a switch case
108    // call the function that will convert to machine code
109

```

```

110 while(fgets(line, 100, no_labels_code) != NULL)
111 {
112     words_in_line = get_words_from_string(line, &num_of_words, &num_of_frees);
113
114     for(instr = 0; instr < NUM_OF_INSTR; instr++)
115     {
116         if(strcmp(list_of_instr->mnemonic[instr], words_in_line[0]) == 0)
117         {
118             break;
119         }
120     }
121
122     // if the mnemonic is not found then print an error message and exit
123     if(instr == NUM_OF_INSTR)
124     {
125         char * target = "\n";
126         char * occur = NULL;
127
128         occur = strstr(line, target);
129         *occur = '\0';
130
131         // print the error message
132         fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid mnemonic %s\n", line,
133             ↪ words_in_line[0]);
134         // free the words that were taken from the line
135         free_split(words_in_line, num_of_frees);
136         clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
137         // exit the program
138         exit(-1);
139     }
140
141     switch(instr)
142     {
143     case 0: // ld
144     case 1: // str
145     case 2: // add
146     case 3: // sub
147     case 4: // and
148     case 5: // or
149     case 6: // mov
150     case 7: // cmp
151     {
152         char * error_operand = two_operands(line, machine_code, list_of_regs, list_of_instr);
153         if(error_operand != NULL)
154         {
155             char * target = "\n";
156             char * occur = NULL;
157
158             occur = strstr(line, target);

```

```

158     *occur = '\0';
159
160     // print the error message
161     fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid operand %s\n", line,
    ↪ error_operand);
162     // free the memory
163     free(error_operand);
164     free_split(words_in_line, num_of_frees);
165     clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
166     // exit the program
167     exit(-1);
168 }
169 }
170 break;
171 case 8: // jz
172 case 9: // jmp
173 case 17: // call
174 case 19: // jb
175 case 20: // jbe
176 case 21: // ja
177 case 22: // jae
178 case 23: // jg
179 case 24: // jge
180 case 25: // jl
181 case 26: // jle
182     label_operands(line, machine_code, list_of_instr);
183     break;
184 case 10: // movi
185 case 11: // addi
186 case 12: // subi
187 case 13: // andi
188 case 14: // ori
189 {
190     char * error_operand = immediate_operands(line, machine_code, list_of_regs, list_of_instr);
191     if(error_operand != NULL)
192     {
193         char * target = "\n";
194         char * occur = NULL;
195
196         occur = strstr(line, target);
197         *occur = '\0';
198
199         // print the error message
200         fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid operand %s\n", line,
    ↪ error_operand);
201         // free the memory
202         free(error_operand);
203         free_split(words_in_line, num_of_frees);
204         clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);

```

```

205     // exit the program
206     exit(-1);
207 }
208 }
209 break;
210 case 15: // push
211 case 16: // pop
212 {
213     char * error_operand = stack_operands(line, machine_code, list_of_regs, list_of_instr);
214     if(error_operand != NULL)
215     {
216         char * target = "\n";
217         char * occur = NULL;
218
219         occur = strstr(line, target);
220         *occur = '\0';
221
222         // print the error message
223         fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid operand \"%s\n", line,
↪ error_operand);
224         // free the memory
225         free(error_operand);
226         free_split(words_in_line, num_of_frees);
227         clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
228         // exit the program
229         exit(-1);
230     }
231 }
232 break;
233 case 18: // return
234     return_instr(machine_code);
235     break;
236 case 27: // cbt
237 case 28: // trf
238 case 29: // pr
239 case 30: // cbr
240 case 31: // rtr
241 case 32: // rr
242     comms_instr(machine_code, words_in_line[0], list_of_instr);
243     break;
244 default:
245     break;
246 }
247 // free memory used for this line of code
248 free_split(words_in_line, num_of_frees);
249 }
250
251 clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
252 }

```

```

253
254 char * reg_operand_get(char * reg, registers * list_of_regs)
255 {
256     // hold the address where the binary format of the register is stored
257     char * operand;
258     // use this variable to index through the array of registers
259     int index = 0;
260     // find the register
261     for(index = 0; index < NUM_OF_REG; index++)
262     {
263         if(strcmp(reg, list_of_regs->reg[index]) == 0)
264         {
265             operand = list_of_regs->operand[index];
266             break;
267         }
268     }
269     // if index is NUM_OF_REG then the register does not exist
270     // therefore return NULL to indicate the register does not exist
271     if(index == NUM_OF_REG)
272     {
273         return NULL;
274     }
275     // return the address of the binary format of the register
276     return operand;
277 }
278
279 char * two_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions *
    ↪ list_of_instr)
280 {
281     // this will be a double pointer that holds the address of all the strings in an instruction
282     char ** indiv;
283     // this will hold the number of strings in the instruction
284     int num_of_data = 0;
285     // this will hold the number of frees
286     int num_of_frees = 0;
287     // get all the strings in the instruction
288     indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
289     // get the length of the 1st operand
290     int length = strlen(indiv[1]);
291     // add a null terminator one less than length of the string so that the comma is removed
292     indiv[1][length - 1] = '\0';
293
294     // print opcode
295     for(int i = 0; i < NUM_OF_INSTR; i++)
296     {
297         if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
298         {
299             fprintf(machine_code, "%s", list_of_instr->opcode[i]);
300             break;

```

```

301     }
302 }
303
304 // get the 1st operand
305 char * operand_1 = reg_operand_get(indiv[1], list_of_regs);
306 // get the 2nd operand
307 char * operand_2 = reg_operand_get(indiv[2], list_of_regs);
308
309 // if the 1st operand is NULL then the register does not exist
310 if(operand_1 == NULL)
311 {
312     char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[1]));
313     strcpy(error_operand, indiv[1]);
314     free_split(indiv, num_of_frees);
315     return error_operand;
316 }
317 // if the 2nd operand is NULL then the register does not exist
318 else if(operand_2 == NULL)
319 {
320     char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[2]));
321     strcpy(error_operand, indiv[2]);
322     free_split(indiv, num_of_frees);
323     return error_operand;
324 }
325
326 // print the 1st and 2nd operand and a newline
327 fprintf(machine_code, "%s", operand_1);
328 fprintf(machine_code, "%s", operand_2);
329 fprintf(machine_code, "\n");
330 // free the strings and the double pointer
331 free_split(indiv, num_of_frees);
332 // return NULL to indicate that it is successful
333 return NULL;
334 }
335
336 void label_operands(char * code, FILE * machine_code, instructions * list_of_instr)
337 {
338     char ** indiv;
339     int num_of_data = 0;
340     int num_of_frees = 0;
341     indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
342
343     // print opcode
344     for(int i = 0; i < NUM_OF_INSTR; i++)
345     {
346         if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
347         {
348             fprintf(machine_code, "%s", list_of_instr->opcode[i]);
349             break;

```

```

350     }
351 }
352 fprintf(machine_code, "000000000");
353 fprintf(machine_code, "\n");
354
355 int label = atoi(indiv[1]);
356 char * label_bin = convert_to_binary(label, 0, 16);
357 fprintf(machine_code, "%s\n", label_bin);
358
359 free(label_bin);
360 free_split(indiv, num_of_frees);
361 }
362
363 char * immediate_operands(char * code, FILE * machine_code, registers * list_of_regs,
    ↪ instructions * list_of_instr)
364 {
365     // this will be a double pointer that holds the address of all the strings in an instruction
366     char ** indiv;
367     // this will hold the number of strings in the instruction
368     int num_of_data = 0;
369     // this will hold the number of frees
370     int num_of_frees = 0;
371     // get all the strings in the instruction
372     indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
373     // get the length of the 1st operand
374     int length = strlen(indiv[1]);
375     // add a null terminator one less than length of the string so that the comma is removed
376     indiv[1][length - 1] = '\0';
377
378     // print opcode
379     for(int i = 0; i < NUM_OF_INSTR; i++)
380     {
381         if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
382         {
383             fprintf(machine_code, "%s", list_of_instr->opcode[i]);
384             break;
385         }
386     }
387
388     char * reg_operand = reg_operand_get(indiv[1], list_of_regs);
389     // if the reg operand is NULL then the register does not exist
390     if(reg_operand == NULL)
391     {
392         char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[1]));
393         strcpy(error_operand, indiv[1]);
394         free_split(indiv, num_of_frees);
395         return error_operand;
396     }
397     // print the reg_operand

```

```

398 fprintf(machine_code, "%s%s", reg_operand, reg_operand);
399 fprintf(machine_code, "\n");
400
401 // check if the immediate has a '-'
402 int i = 0;
403 for(i = 0; i < strlen(indiv[2]); i++)
404 {
405     if(indiv[2][i] == '-')
406     {
407         break;
408     }
409 }
410
411 // this will hold the address of the immediate value that will be extracted from the 2nd
412     ↪ operand
413 char * immediate;
414
415 // if i is 1 then it means that the immediate is a negative value
416 if(i == 1)
417 {
418     // increment by 1 so that we skip the '-'
419     i++;
420     // create memory on the heap
421     immediate = (char *)xmalloc(sizeof(char) * (strlen(indiv[2]) - 2));
422     int j = 0;
423     // copy data
424     for(j = 0; j < strlen(indiv[2]) - 2; j++, i++)
425     {
426         immediate[j] = indiv[2][i];
427     }
428     // place null terminator
429     immediate[j] = '\0';
430     // convert from string to integer
431     int immediate_int = atoi(immediate);
432     // convert the integer to binary format
433     char * immediate_bin = convert_to_binary(immediate_int, 1, 16);
434     // print the binary format of the immediate to the file
435     fprintf(machine_code, "%s\n", immediate_bin);
436     // free the immediate binary format
437     free(immediate_bin);
438 }
439 else
440 {
441     // set the i to 1 so that the data is copied after the # character
442     i = 1;
443     // create memory on the heap
444     immediate = (char *)xmalloc(sizeof(char) * (strlen(indiv[2]) - 1));
445     // copy the immediate
446     int j = 0;

```



```

446     for(j = 0; j < strlen(indiv[2]) - 1; j++, i++)
447     {
448         immediate[j] = indiv[2][i];
449     }
450     // place null terminator
451     immediate[j] = '\0';
452     // convert from string to integer
453     int immediate_int = atoi(immediate);
454     // convert from integer to binary format
455     char * immediate_bin = convert_to_binary(immediate_int, 0, 16);
456     // print the binary format of the immediate to the file
457     fprintf(machine_code, "%s\n", immediate_bin);
458     // free the immediate binary format
459     free(immediate_bin);
460 }
461
462 // free the immediate value that was copied
463 free(immediate);
464 // free the each opcode and operand from the current instruction
465 free_split(indiv, num_of_frees);
466
467 return NULL;
468 }
469
470 char * stack_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions *
    ↪ list_of_instr)
471 {
472     char ** indiv;
473     int num_of_data = 0;
474     int num_of_frees = 0;
475     indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
476     int length = strlen(indiv[1]);
477     indiv[1][length] = '\0';
478
479     // print opcode
480     for(int i = 0; i < NUM_OF_INSTR; i++)
481     {
482         if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
483         {
484             fprintf(machine_code, "%s", list_of_instr->opcode[i]);
485             break;
486         }
487     }
488
489     char * reg_operand = reg_operand_get(indiv[1], list_of_regs);
490     // if the reg operand is NULL then the register does not exist
491     if(reg_operand == NULL)
492     {
493         char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[1]));

```

```

494     strcpy(error_operand, indiv[1]);
495     free_split(indiv, num_of_frees);
496     return error_operand;
497 }
498
499 fprintf(machine_code, "00000");
500 fprintf(machine_code, "%s", reg_operand);
501 fprintf(machine_code, "\n");
502 free_split(indiv, num_of_frees);
503
504 return NULL;
505 }
506
507 void return_instr(FILE * machine_code)
508 {
509     fprintf(machine_code, "0100100000000000\n");
510 }
511
512 void comms_instr(FILE * machine_code, char * instr, instructions * list_of_instr)
513 {
514     // print opcode
515     for(int i = 0; i < NUM_OF_INSTR; i++)
516     {
517         if(strcmp(instr, list_of_instr->mnemonic[i]) == 0)
518         {
519             fprintf(machine_code, "%s", list_of_instr->opcode[i]);
520             break;
521         }
522     }
523
524     fprintf(machine_code, "0000000000\n");
525 }
526
527 void clean_up(FILE * no_labels_code, FILE * machine_code, instructions * list_of_instr,
528               ↪ registers * list_of_regs)
529 {
530     fclose(no_labels_code);
531     fclose(machine_code);
532     free_all_opcodes_and_operands(list_of_instr, list_of_regs);
533     free(list_of_instr);
534     free(list_of_regs);
535 }

```

Listing E.5: This is the source file for all the functions that are responsible for converting the preprocessed assembly code into machine code

## extra.h

```

1 /*
   ↪ *****

```

```

2 * File name
3     extra.h
4 * Description
5     Header file extra.h
6 * Author
7     Sreethyan Aravinthan (UCL)
8 *****/
9
10 #ifndef _EXTRA_H_
11 #define _EXTRA_H_
12
13 #define WIDTH 16
14
15 // functions
16 FILE * xopen(char * file_name, char * mode);
17 void * xmalloc(size_t size);
18 int positive_binary(unsigned short int immediate, char * binary_form, unsigned int index);
19 void twos_complement(char * binary_form, unsigned char width1);
20 char * convert_to_binary(unsigned short int immediate, char negative, unsigned char width);
21 char **get_words_from_string(const char *input_string, int *num_of_words, int *num_of_frees);
22 void free_split(char ** words, int num_of_frees);
23 char * shift_left_one(char * data);
24
25 #endif

```

Listing E.6: This is the header file for all helper functions that are used throughout the code

### extra.c

```

1 /*
2 *
3 * File name
4 *     extra.c
5 * Description
6 *     This file has extra tools that are used for converting to machine code
7 * Author
8 *     Sreethyan Aravinthan (UCL)
9 *****/
10
11 // standard header files
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15
16 // personal header files
17 #include "extra.h"

```

```

18 FILE * xfopen(char * file_name, char * mode)
19 {
20     // open the file
21     FILE * file = fopen(file_name, mode);
22     // if the file is NULL then exit gracefully
23     if(file == NULL)
24     {
25         fprintf(stderr, "File opening error");
26         exit(-1);
27     }
28     return file;
29 }
30
31 void * xmalloc(size_t size)
32 {
33     void * data = malloc(size);
34
35     // if the requested data is not given in other words returned NULL exit
36     if (data == NULL)
37     {
38         fprintf(stderr, "virtual memory exhausted");
39         exit(-1);
40     }
41
42     // return the data variable
43     return data;
44 }
45
46 void * xrealloc(void *ptr, size_t size)
47 {
48     void * data = realloc(ptr, size);
49
50     // if the requested data is not given in other words returned NULL exit
51     if (data == NULL)
52     {
53         fprintf(stderr, "virtual memory exhausted");
54         exit(-1);
55     }
56
57     // return the data variable
58     return data;
59 }
60
61
62 int positive_binary(unsigned short int immediate, char * binary_form, unsigned int index)
63 {
64     // check if the value is greater than 1
65     if(immediate > 1)
66     {

```

```

67         index = positive_binary(immediate/2, binary_form, index);
68     }
69
70     // if the modulus is 1 then set that index value to 1
71     // else 0
72     if(immediate % 2 == 1)
73     {
74         binary_form[index] = '1';
75     }
76     else
77     {
78         binary_form[index] = '0';
79     }
80     // increment index for the next binary number
81     index++;
82
83     // return index for the next binary number which was called in a recursive manner
84     return index;
85 }
86
87 void twos_complement(char * binary_form, unsigned char width)
88 {
89     // convert to 1s complement
90     for(int i = 0; i < width; i++)
91     {
92         if(binary_form[i] == '0')
93         {
94             binary_form[i] = '1';
95         }
96         else
97         {
98             binary_form[i] = '0';
99         }
100     }
101
102     // convert to 2s complement from 1s complement
103     for(int i = width - 1; i >= 0; i--)
104     {
105         if(binary_form[i] == '1')
106         {
107             binary_form[i] = '0';
108         }
109         else
110         {
111             binary_form[i] = '1';
112             break;
113         }
114     }
115 }

```

```

116
117 char * convert_to_binary(unsigned short int immediate, char negative, unsigned char width)
118 {
119     // this will contain the positive binary form of a number
120     char * pos_binary = (char *)malloc(sizeof(char) * width);
121     // this variable will hold the pointer to which the correct number of bits of the binary
122     ↪ number should be
123     char * pos_binary_form = (char *)malloc(sizeof(char) * width);
124     // set the length to 0 which hold the length of the initial positive version of the binary
125     ↪ number
126     int length = 0;
127     // get the positive version of the number passed in
128     length = positive_binary(immediate, pos_binary, 0);
129     // add null terminator
130     length++;
131     pos_binary[length] = '\0';
132     length--;
133
134     int i = 0;
135     int j = 0;
136
137     // create the positive binary number with the correct width
138     for(i = 0; i < width - length; i++)
139     {
140         pos_binary_form[i] = '0';
141     }
142
143     for(j = 0; j < length; j++, i++)
144     {
145         pos_binary_form[i] = pos_binary[j];
146     }
147
148     // if the flag for negative was set then get the 2s complement
149     if(negative == 1)
150     {
151         twos_complement(pos_binary_form, width);
152     }
153
154     // place null terminator
155     pos_binary_form[width] = '\0';
156     // free the earlier version of the positive binary number
157     free(pos_binary);
158
159     // printf("%s\n", pos_binary_form);
160
161     // return the pointer to the positive binary number
162     return pos_binary_form;
163 }

```

```

163 char **get_words_from_string(const char *input_string, int *num_of_words, int *num_of_frees)
164 {
165     // amount holds the number of words that can be taken from string
166     // intialy 5
167     unsigned int amount = 5;
168
169     // start with asumming that there are 5 words in the string
170     char **words = (char **)xmalloc(amount * sizeof(char *));
171
172     // this is the default size of each word
173     unsigned int word_size = 5;
174
175     // allocate each word size as 5 as default
176     for (int i = 0; i < amount; i++)
177     {
178         words[i] = (char *)xmalloc(word_size * sizeof(char));
179     }
180
181     // this is used to go through the string
182     unsigned int index = 0;
183     /* this is used to see how many words of the allocated amount has actually
184     been filled */
185     unsigned int amount_level = 0;
186     // this is used to see how many characters have been filled
187     unsigned int word_size_level = 0;
188
189     // this is a flag used to see if the end of a word has been reached
190     char clean = 0;
191
192     // until space or \n or tab is not reached
193     while(input_string[index] == ' ' || input_string[index] == '\n'
194           || input_string[index] == 9)
195     {
196         // increment index to skip ' ' or '\n'
197         index++;
198     }
199
200     while(input_string[index] != '\0')
201     {
202         if(word_size_level == word_size)
203         {
204             word_size += 5;
205             // create a new string of this size
206             words[amount_level] = (char *)xrealloc((void *)words[amount_level],
207             word_size * sizeof(char));
208         }
209
210         // until space or \n or tab is not reached
211         while(input_string[index] == ' ' || input_string[index] == '\n'

```

```

212         || input_string[index] == 9)
213     {
214         // increment index to skip ' ' or '\n'
215         index++;
216         clean = 1;
217     }
218
219     if(clean == 1)
220     {
221         // place null terminator at the end of string
222         words[amount_level][word_size_level] = '\0';
223         // reset flag
224         word_size_level = 0;
225
226         //amount_levelincrement for the next string
227         amount_level++;
228
229         /* if the amount filled is the same as the max available then
230         increase */
231         if(amount_level == amount)
232         {
233             // create a temp variable of char **
234             char ** temp = (char **)xmalloc((amount + 5) * sizeof(char *));
235
236             // create space for each word
237             for(int i = 0; i < amount + 5; i++)
238             {
239                 // this for the old data to be copied over so strlen used
240                 if(i < amount)
241                 {
242                     temp[i] = (char *)xmalloc((strlen(words[i]) + 1)
243                                             * sizeof(char));
244                 }
245                 else // this is for new data so default word_size
246                 {
247                     temp[i] = (char *)xmalloc(word_size * sizeof(char));
248                 }
249             }
250
251             // copy old data over
252             for(int i = 0; i < amount; i++)
253             {
254                 strcpy(temp[i], words[i]);
255             }
256
257             // clear old data
258             for(int i = 0; i < amount; i++)
259             {
260                 free(words[i]);

```



```

261         }
262
263         // clear words
264         free(words);
265
266         // assign words to the newly created memory
267         words = temp;
268         // NULL the initial pointer pointing to the newly created memory
269         temp = NULL;
270         // increment the total number of words for comparison next time
271         amount += 5;
272     }
273
274     // reset flag such next time a space, \n or tab is reached
275     clean = 0;
276 }
277
278 // assign current character
279 words[amount_level][word_size_level] = input_string[index];
280
281 // increment track and index for next character
282 word_size_level++;
283 index++;
284 }
285
286 // place \0 for last one
287 words[amount_level][word_size_level] = '\0';
288
289 // assign the num_of_words and num_of_frees for use of printing and freeing
290 *num_of_words = amount_level + 1;
291 *num_of_frees = amount;
292
293 // return words
294 return words;
295 }
296
297 void free_split(char ** words, int num_of_frees)
298 {
299     // free the individual strings
300     for(int i = 0; i < num_of_frees; i++)
301     {
302         free(words[i]);
303         words[i] = NULL;
304     }
305     // free the double pointer
306     free(words);
307     words = NULL;
308 }
309

```

```
310 char * shift_left_one(char * data)
311 {
312     int i;
313
314     for(i = 1; i < strlen(data); i++)
315     {
316         data[i - 1] = data[i];
317     }
318     data[i - 1] = '\0';
319     return data;
320 }
```

Listing E.7: This is the source file for all helper functions that are used throughout the code