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: A. Sreethyan ………………………………

Date: 21/04/2020…...……………………….

Contents

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 a 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 of the open-source RISC-V (RV321) 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 microprocessor 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

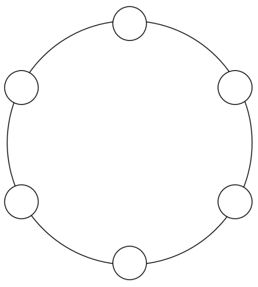


Figure 3.1: Ring topology

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

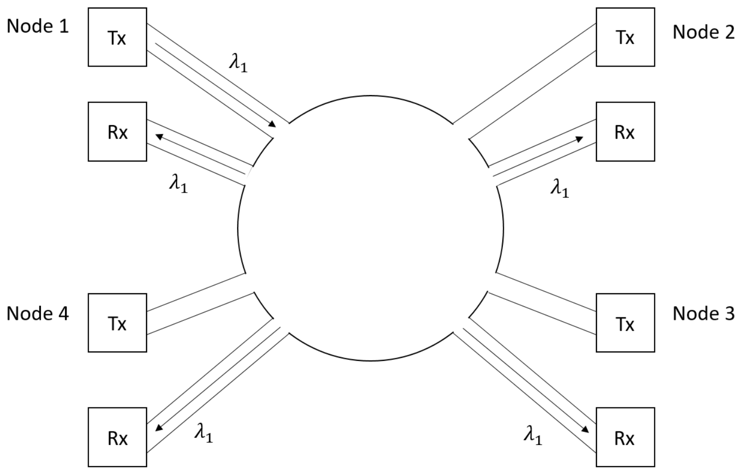


Figure 3.2: Star network with a coupler

Figure 3.2 shows how the nodes on a network are connected in a star network where the centre is a NxN 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

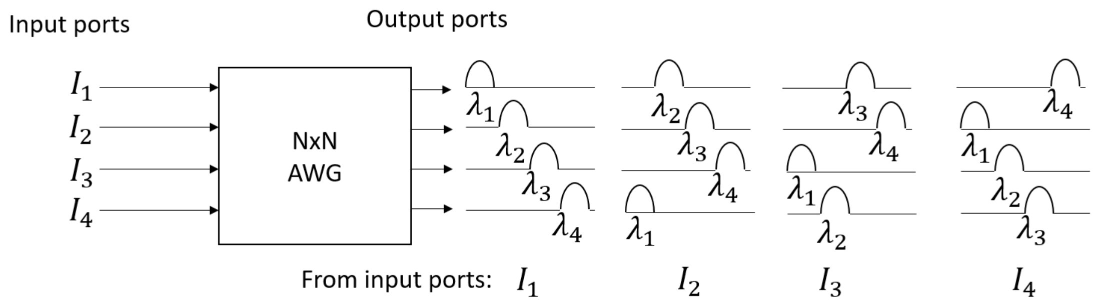

Input ports
Output ports
I_1
I_2
I_3
I_4
NxN
AWG
λ_1 λ_2 λ_3 λ_4
λ_2 λ_3 λ_4 λ_1
λ_3 λ_4 λ_1 λ_2
λ_4 λ_1 λ_2 λ_3
From input ports: I_1 I_2 I_3 I_4

Figure 3.3: Cyclic AWG

For an N×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].

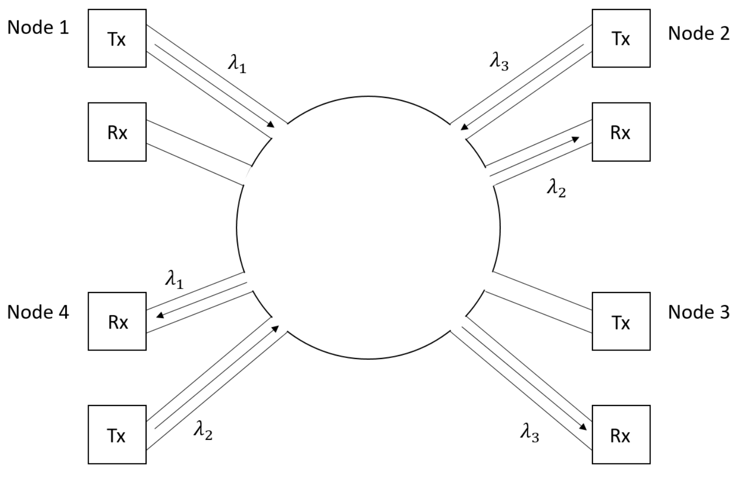


Figure 3.4: Star network with a router

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 accepts 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

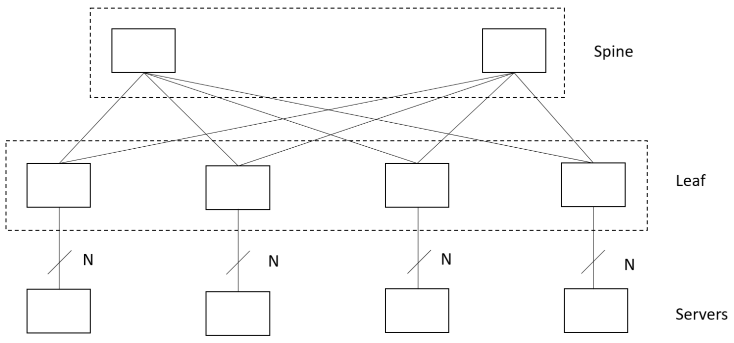


Figure 3.5: Spine-leaf topology

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

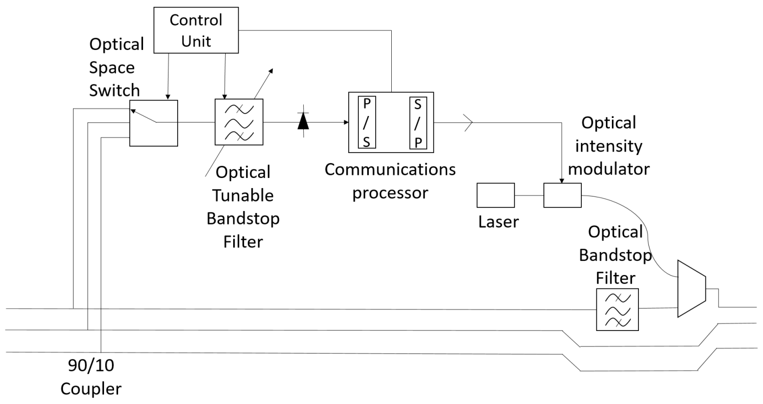


Figure 3.6: Design for photonic transceiver. S/P - serial to parallel conversion. P/S - parallel to serial conversion

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

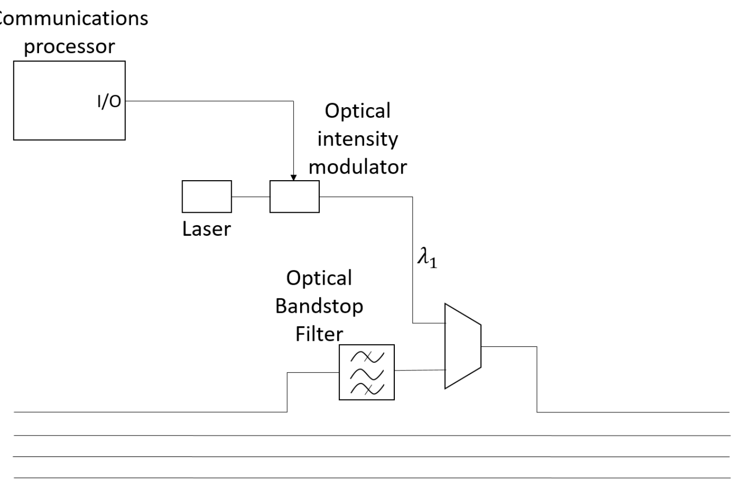


Figure 3.7: Design for photonic transmitter

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. λ_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 λ_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

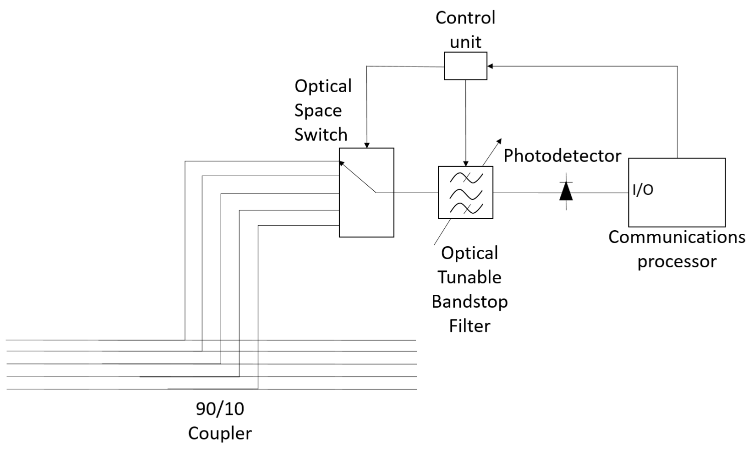


Figure 3.8: Design for photonic receiver

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.

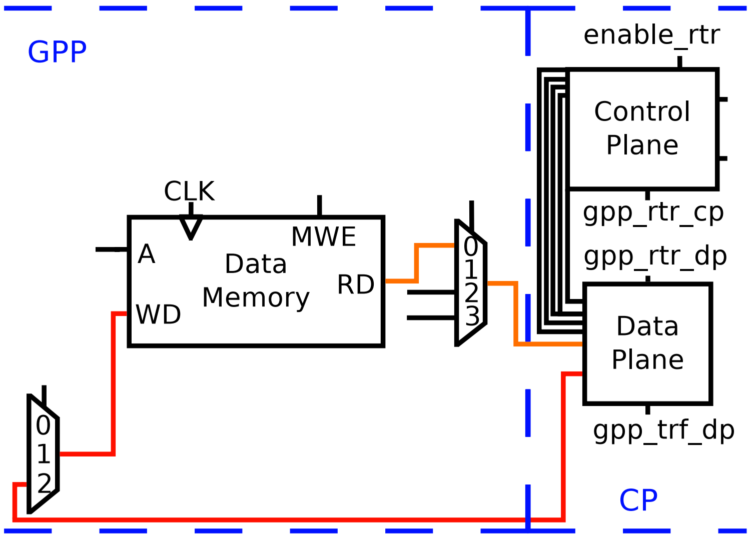


Figure 4.1: Data transfer between GPP and CP

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 chapter 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 recevier	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, cbt, 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

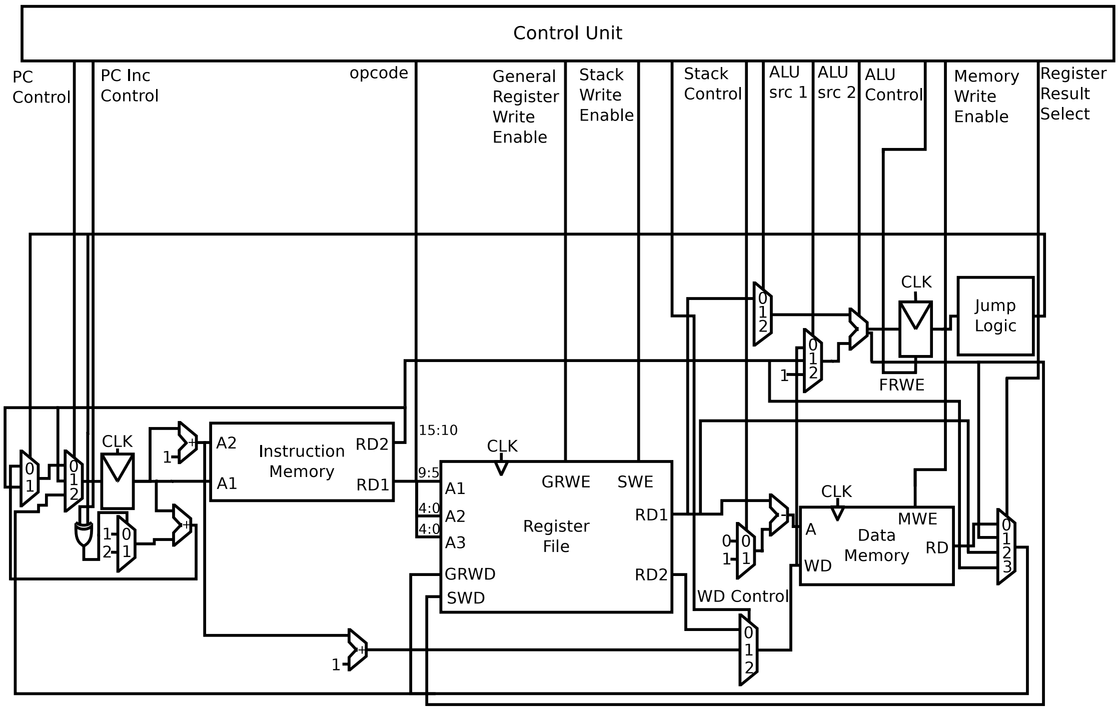


Figure 4.2: Microarchitecture for the GPP

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 form 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 ≠ overflow flag
jle (jump less equal)	sign flag ≠ 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 is 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 in 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 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 Gbs^{-1} and each packet is 32-bits, the time taken can be calculated as follows

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

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

```
  cbt      # compare to see if data can be transferred
  jz transfer # if zero then data can be transfered
  jmp exit  # if not then do not transfer
transfer:
  trf
  trf
  trf
  trf
  trf
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.

```
    pr  # pause recieving on CP
    cbr # instruction which sees if the data can be retrived
    jz exit # if the result is zero then this indicates that the data plane is reciving
    rtr
    rtr
    rtr
    rtr
    rtr
exit:
    rr  # resume receving 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

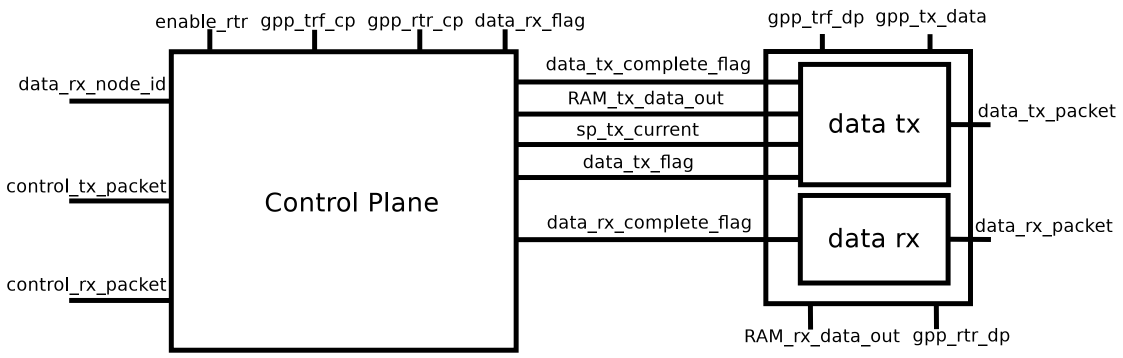


Figure 4.3: Microarchitecture of CP

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 retrive 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 retrive 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 reciever	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

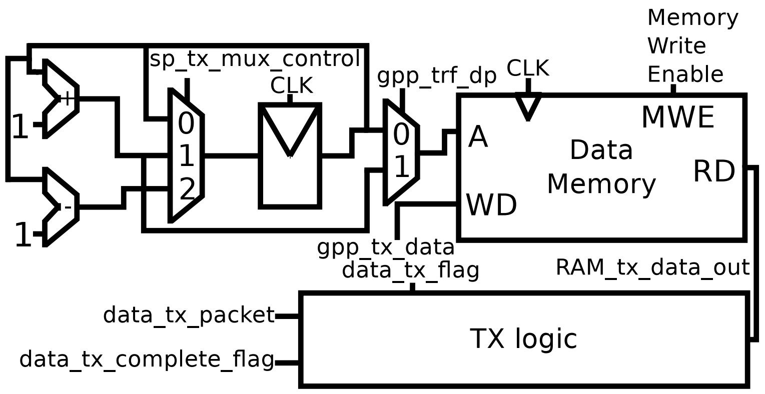


Figure 4.4: Microarchitecture of data plane transmitter

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

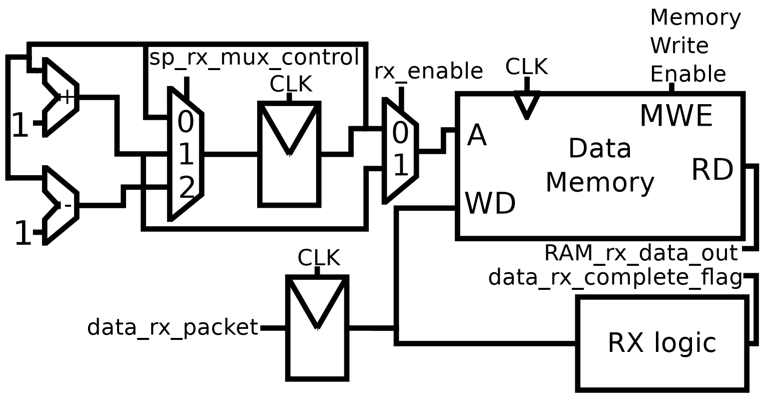


Figure 4.5: Microarchitecture of data plane receiver

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

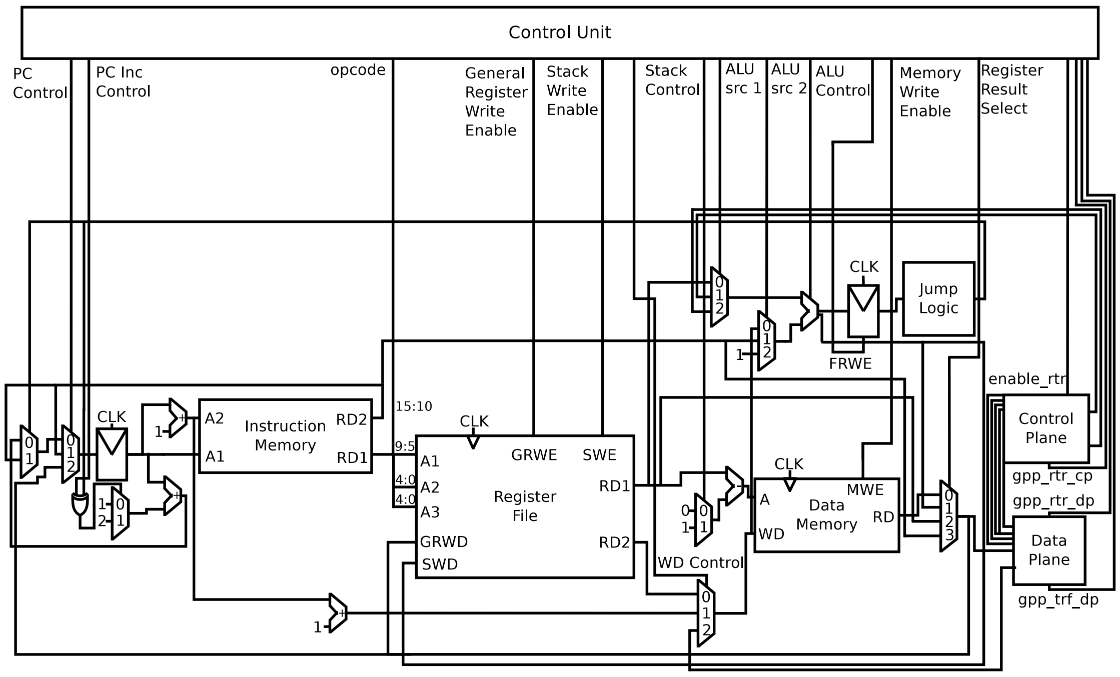


Figure 4.6: Microarchitecture for the specified instruction set

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

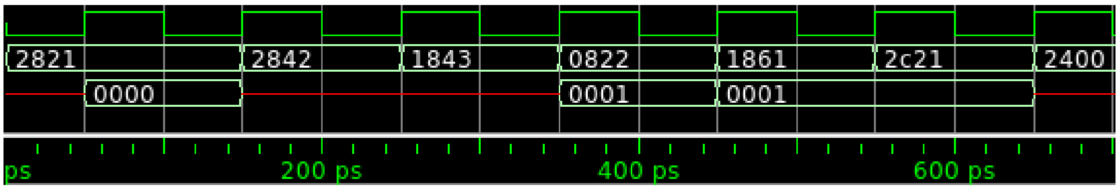
5.1 Results from Simulation

Lower-Level Modules

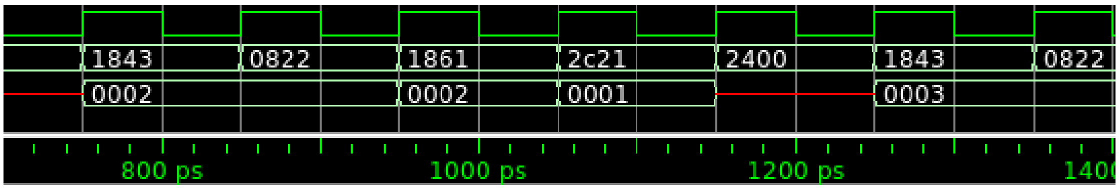
All the tests on the lower-level that were carried out were successful. The simulation results matched the expected behaviour.

GPP

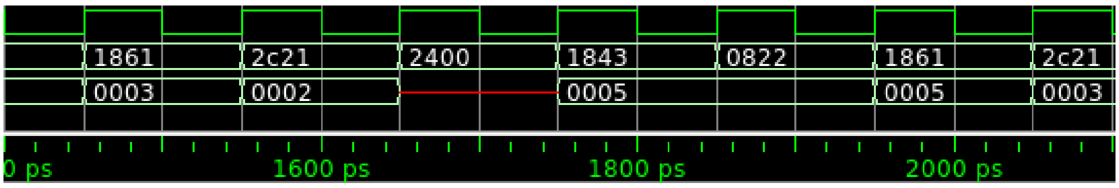
Fibonacci sequence - move instruction



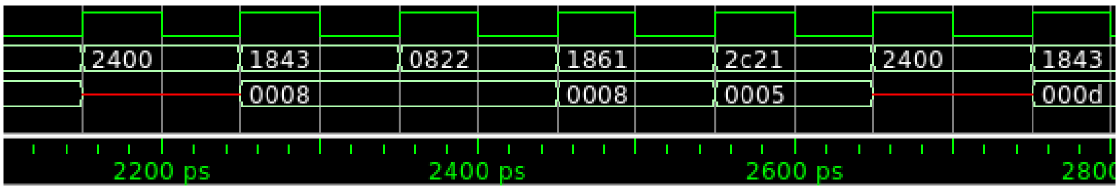
(a)



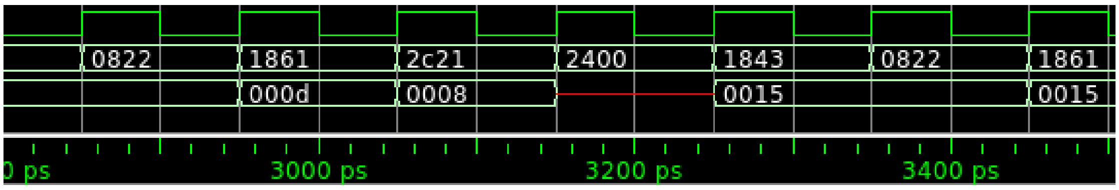
(b)



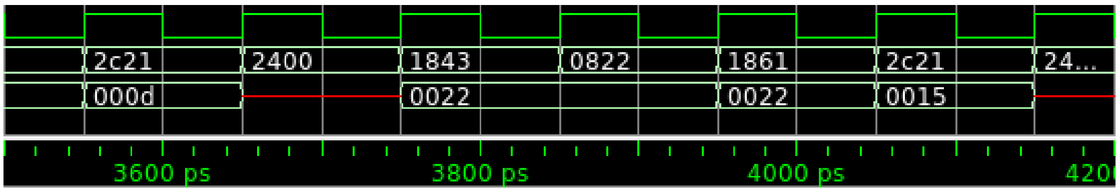
(c)



(d)



(e)

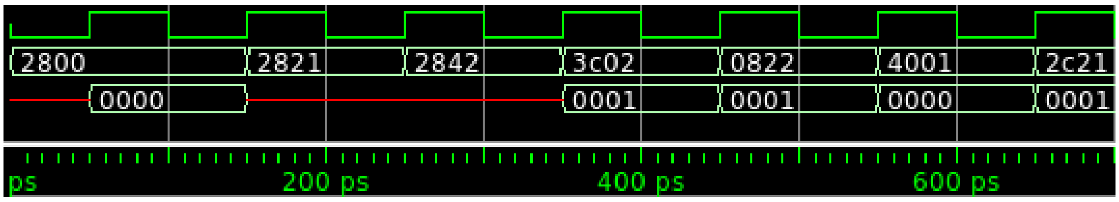


(f)

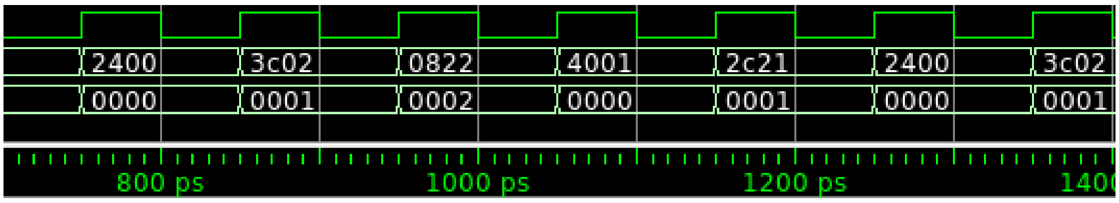
Figure 5.1: Shows the timing diagrams when executing the Fibonacci sequence program using the move instructions. Figure 5.1a is between 0 ps and 700ps. Figure 5.1b is between 700 ps and 1400ps. Figure 5.1c is between 1400 ps and 2100ps. Figure 5.1d is between 2100 ps and 2800ps. Figure 5.1e is between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

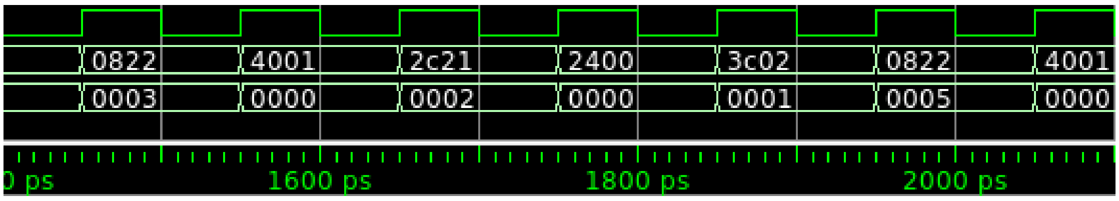
Fibonacci sequence - stack instructions



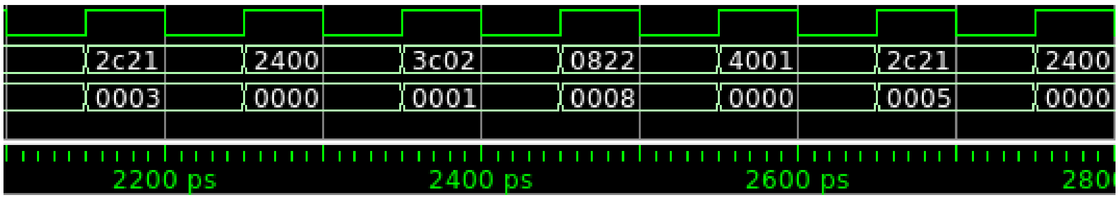
(a)



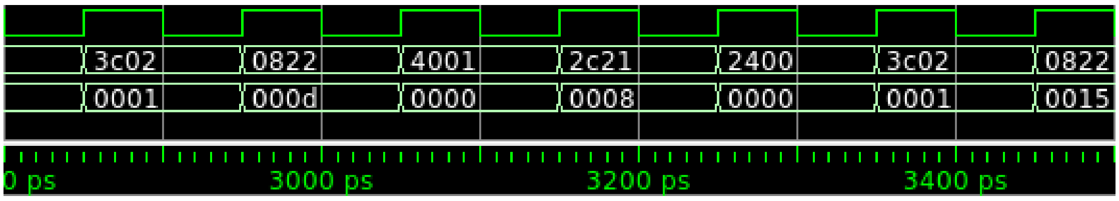
(b)



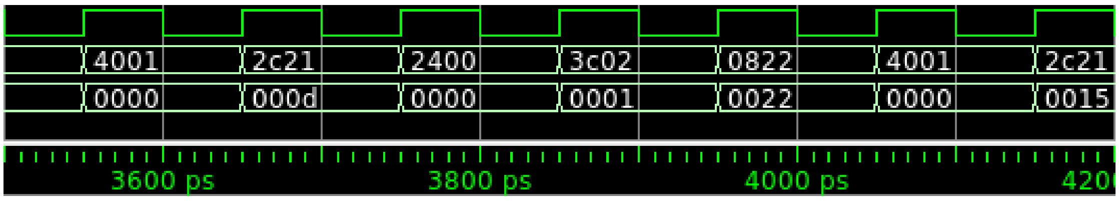
(c)



(d)



(e)

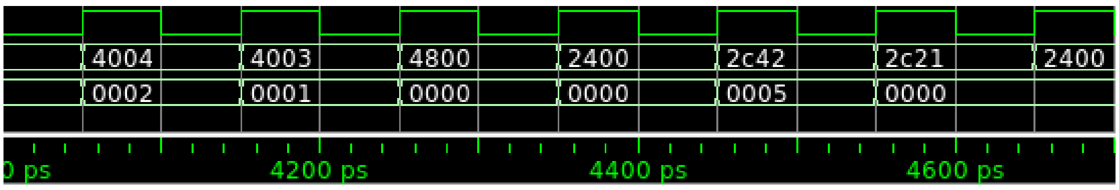


(f)

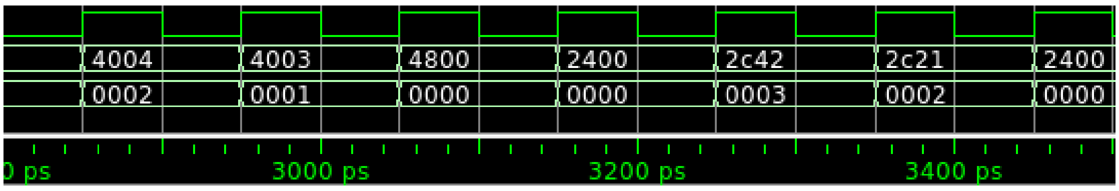
Figure 5.2: Shows the timing diagrams when executing the Fibonacci sequence program using the stack instructions. Figure 5.2a is between 0 ps and 700ps. Figure 5.2b is between 700 ps and 1400ps. Figure 5.2c is between 1400 ps and 2100ps. Figure 5.2d is between 2100 ps and 2800ps. Figure 5.2e is between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



(a)

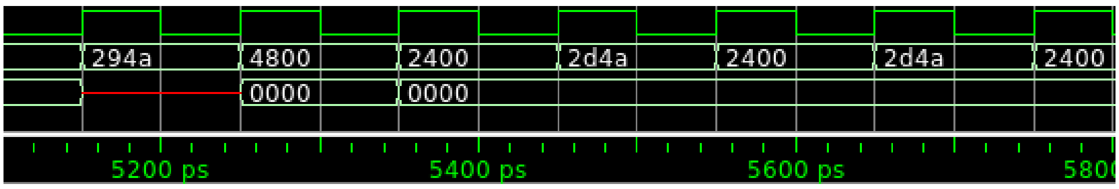


(b)

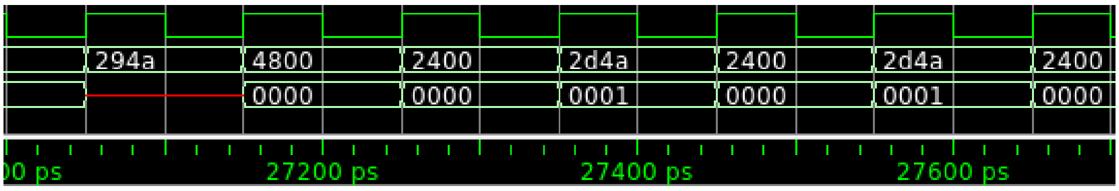
Figure 5.3: Shows the timing diagrams when executing the division algorithm. Figure 5.3a shows the case when the dividend is 10 and divisor is 2. Figure 5.3b shows the case when the dividend is 11 and divisor is 3

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



(a)



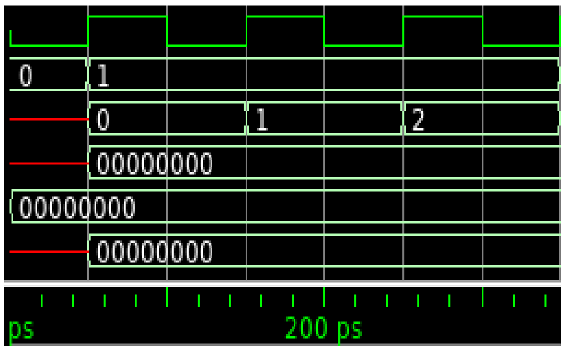
(b)

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

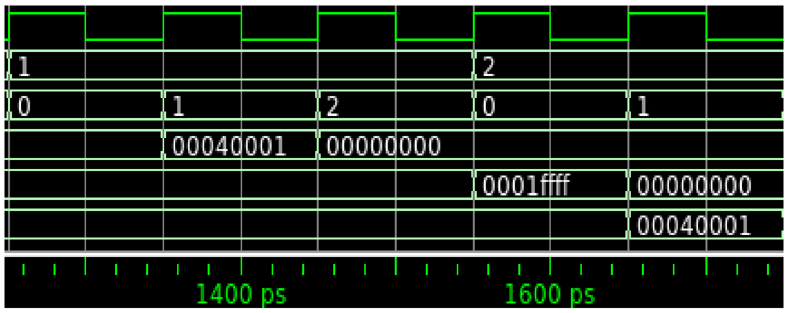
Figure 5.4 shows the timing diagram when the GPP was executing the prime number algorithm for two different cases. The first case is to check if 10 is a prime number or not and the second case is to check if 11 is a prime number or not. As with the previous test, the first signal is the clock signal. 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

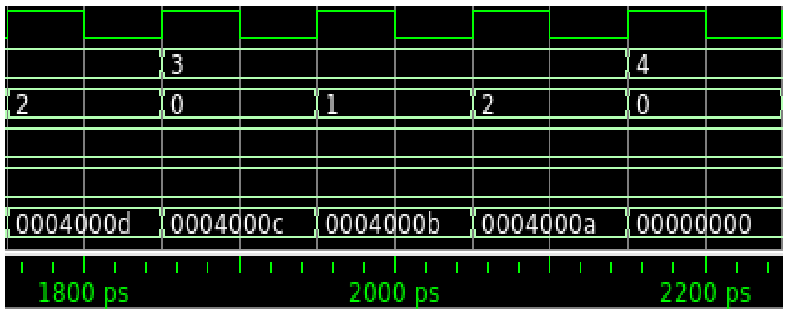
Transmitter



(a)



(b)



(c)

Figure 5.5: Timing diagrams when communications processor works as a transmitter. Figure 5.5a shows the case when the transmiter's RAM is empty. Figure 5.5b and Figure 5.5c shows the case when the transmiter'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

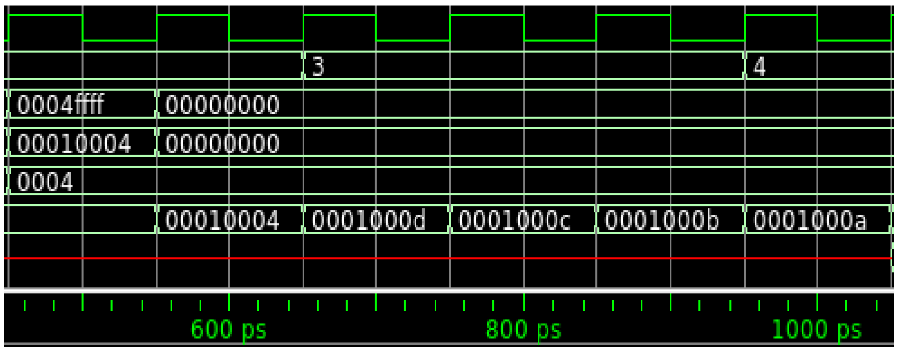
Figure 5.5 shows the timing diagrams when the communications processor is controlling the data transmitter. The signals that are shown in the diagrams are as follows in the following order:

- 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

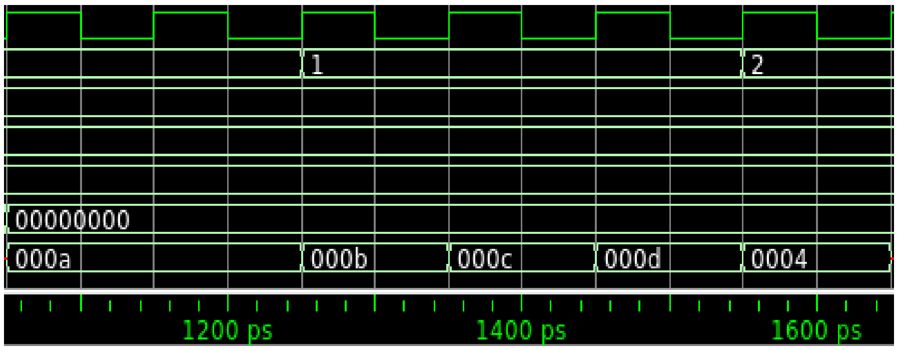
The node id was set to 1 and the maximum number of nodes in the network is set to 4. Figure 5.5a shows the timing diagram as soon as the communications processor has booted. From this timing diagram, it can be seen that the packet that was transmitted on the control plane by node 1 (this node) was 32x00000000. 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 it's 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 reciever to select a channel

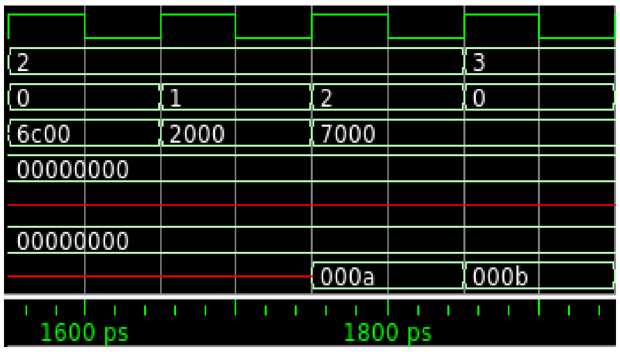
- Packet received on the data plane
- Data transferred from from CP to GPP

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.

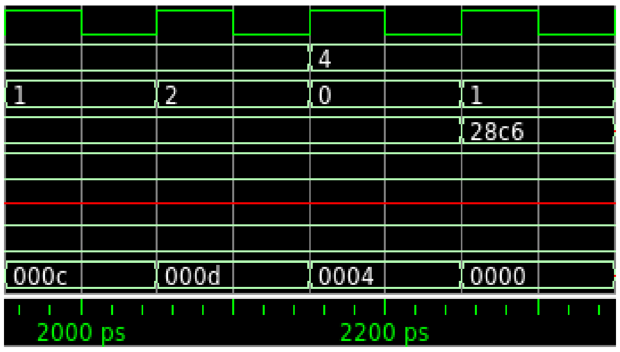
The next section describes the simulated transfer of data from CP to GPP. The appropriate control signals were set in the testbench. If these control signals are applied for 5 clock cycles all the data will be transferred to the GPP. This was clearly observed in the timing diagram shown in Figure 5.6b. 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

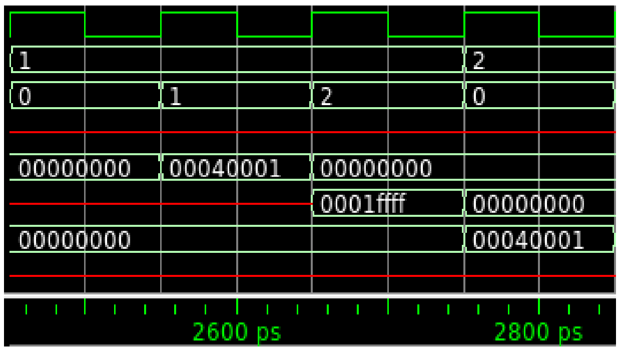
Transmitter



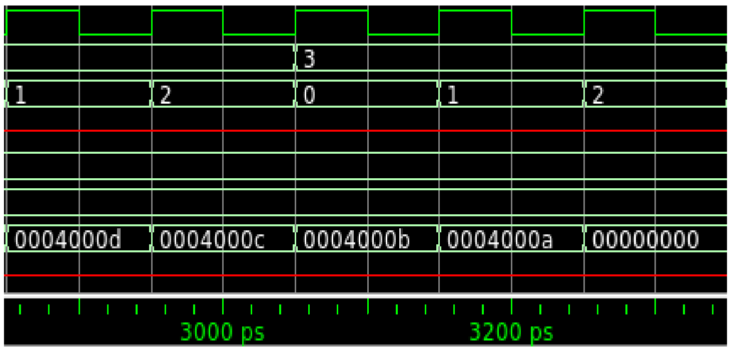
(a)



(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.

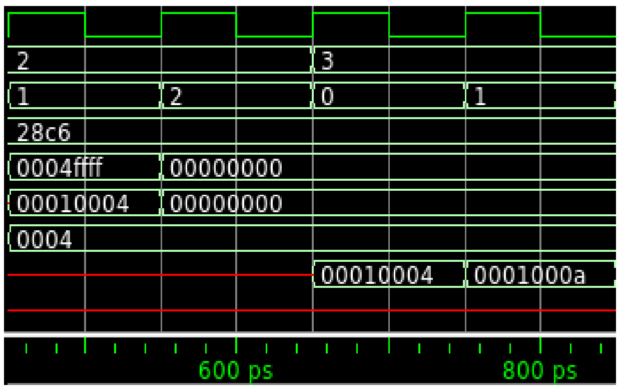
The simulation was set up such that no data was being transmitted on the data plane and the current slot was not equal to the node id. Thus data can be transferred from the GPP to CP as these were the conditions required for data transfer.

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.

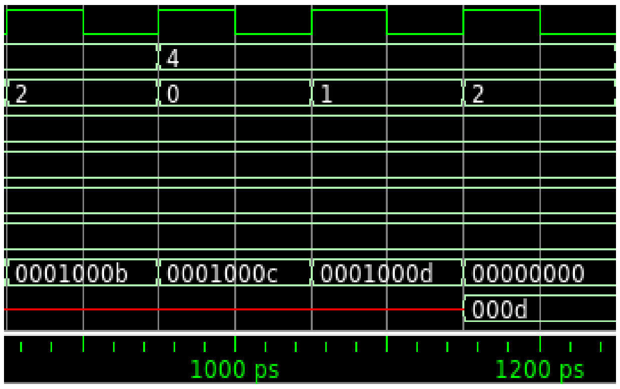
Figure 5.7c and Figure 5.7d are timing diagrams after data has been transferred to the transmitter's RAM. Since the slot and node id have been the same at the start, the node had to wait for its turn to ping a node. These diagrams begin when the slot is equal to the node id once again. Since the slot 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.

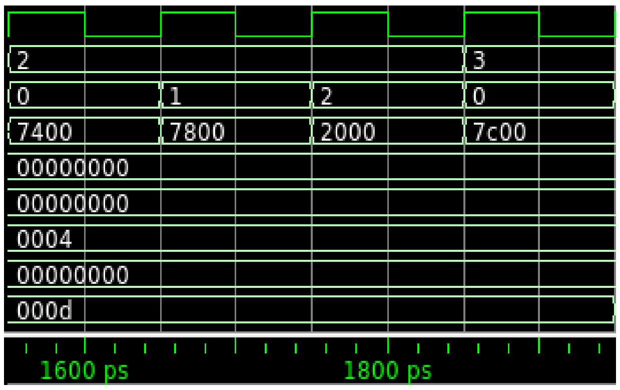
Receiver


2
3
1
2
0
1
28c6
0004ffff
00000000
00010004
00000000
0004
00010004
0001000a
600 ps
800 ps

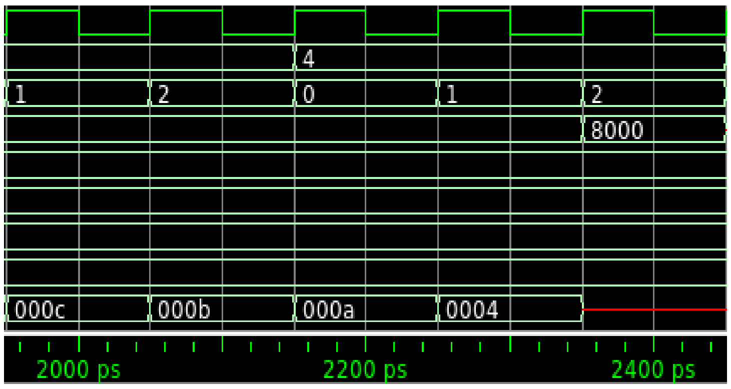
(a)


4
2
0
1
2
0001000b
0001000c
0001000d
00000000
000d
1000 ps
1200 ps

(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 reciever 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 0111010000000000 , 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 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 transmitted 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

```
/*********************************************************************************
* File name
	ALU.sv
* Description
	This code will create an ALU which can take two inputs and add/sub/
	and/or them.
* Parameters
	WIDTH - This is the width of the input and output data.
* Inputs
	src_a - 1st input
	src_b - 2nd input
	ALU_control - Control signal which determins which operation to carry out (add/sub/and/or)
* Outputs
	ALU_out - The output result after applying the operation on the 2 inputs
* Author
	Sreethyan Aravinthan (UCL)
*********************************************************************************/

module ALU
#(
  parameter WIDTH = 1
)
(
  input logic [WIDTH - 1:0]src_a,
  input logic [WIDTH - 1:0]src_b,
  input logic [1:0] ALU_Control,
  output logic [WIDTH - 1:0]ALU_out
);

```

```
always_comb
begin
  case(ALU_Control)
    2'b00 : // add
    begin
      ALU_out = src_a + src_b;
    end
    2'b01 : // sub
    begin
      ALU_out = src_a - src_b;
    end
    2'b10 : // and
    begin
      ALU_out = src_a & src_b;
    end
    2'b11 : // or
    begin
      ALU_out = src_a | src_b;
    end
  endcase
end

endmodule
```

Listing A.1: ALU

ALU_advanced.sv

```
/*********************************************************************************
* File name
        ALU_advanced.sv
* Description
        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.
* Parameters
        WIDTH - This is the width of the input and output data.
* Inputs
        src_a - 1st input
        src_b - 2nd input
        ALU_control - Control signal which determins which operation to carry out (add/sub/and/or)
* Outputs
        ALU_out - The output result after applying the operation on the 2 inputs
        zero_flag - Indicates if the results was a zero
        carry_flag - Indicates if the carry took place
        sign_flag - Indicates if the Most Significant Bit (MSB) is set to 1
        overflow_flag - Indicates if an overflow took place
```

```
* Author
        Sreethyan Aravinthan (UCL)
******************************************************************************/

module ALU_advanced
#(
  parameter WIDTH = 1
)
(
  input logic [WIDTH - 1:0]src_a,
  input logic [WIDTH - 1:0]src_b,
  input logic [1:0] ALU_Control,
  output logic [WIDTH - 1:0]ALU_out,
  output logic zero_flag,
  output logic carry_flag,
  output logic sign_flag,
  output logic overflow_flag
);

  // this holds the result after the operation, so that the carry flag can be set appropriately
  logic [WIDTH:0] result;

  // for all the operations set the zero and sign flag as follows
  assign zero_flag = ALU_out == 0;
  assign sign_flag = ALU_out[WIDTH - 1] == 1'b1;

  always_comb
  begin
    case(ALU_Control)
      2'b00 : // add
      begin
        ALU_out = src_a + src_b;

        /**Setting the carry flag**/

        result = {1'b0, src_a} + {1'b0, src_b};
        carry_flag = result[WIDTH];

        /**Setting the overflow flag**/

        // if the two numbers are positive
        if(src_a[WIDTH - 1] == 1'b0 && src_b[WIDTH - 1] == 1'b0)
        begin
          // and if the result is negative
          if(ALU_out[WIDTH - 1] == 1'b1)
          begin
            // set the overflow flag to 1
            overflow_flag = 1'b1;
          end
          else
          begin
            // set the overflow flaf to 0
            overflow_flag = 1'b0;
          end
        end
        // if the two numbers that are being added -ve
        else if(src_a[WIDTH - 1] == 1'b1 && src_b[WIDTH - 1] == 1'b1)
        begin
          // the result is positive
          if(ALU_out[WIDTH - 1] == 1'b0)
          begin
            // set overflow flag to 1
            overflow_flag = 1'b1;
          end
          else
          begin
            // set overflow flag to 0
            overflow_flag = 1'b0;
          end
        end
        // if one is +ve and the other is -ve
        else
        begin
          overflow_flag = 1'b0;
        end
      end
      2'b01 : // sub
      begin
        ALU_out = src_a - src_b;

        /**Setting the carry flag**/

        result = {1'b0, src_a} - {1'b0, src_b};
        carry_flag = result[WIDTH];

        /**Setting the overflow flag**/

        // if the value is +ve then -ve
        if(src_a[WIDTH - 1] == 1'b0 && src_b[WIDTH - 1] == 1'b1)
        begin
          // and if the result is negative
          if(ALU_out[WIDTH - 1] == 1'b1)
          begin
            // set the overflow flag to 1
            overflow_flag = 1'b1;
          end
          else
          begin
```

```
          // set the overflow flaf to 0
          overflow_flag = 1'b0;
        end
      end
      // if the value is −ve then +ve
      else if(src_a[WIDTH − 1] == 1'b1 && src_b[WIDTH − 1] == 1'b0)
      begin
        // the result is positive
        if(ALU_out[WIDTH − 1] == 1'b0)
        begin
          // set overflow flag to 1
          overflow_flag = 1'b1;
        end
        else
        begin
          // set overflow flag to 0
          overflow_flag = 1'b0;
        end
      end
      // if one is +ve and the other is −ve
      else
      begin
        overflow_flag = 1'b0;
      end
    end
    2'b10 : // and
    begin
      ALU_out = src_a & src_b;
      result = 'b0;

      /**Setting the carry flag**/
      carry_flag = 1'bx;

      /**Setting the overflow flag**/
      overflow_flag = 1'bx;
    end
    2'b11 : // or
    begin
      ALU_out = src_a | src_b;
      result = 'b0;

      /**Setting the carry flag**/
      carry_flag = 1'bx;

      /**Setting the overflow flag**/
      overflow_flag = 1'bx;
    end
  endcase
end
```

```
endmodule
```

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

Control_Unit.sv

```
/*
    ↪ ********************************************************************************
    ↪
* File name
        Control_Unit.sv
* Description
        This module takes the opcode and sets the values of the control signals.
* Parameters
        NONE
* Inputs
        opcode - This the 6 MSB of the instruction.
* Outputs
        ALL THE CONTROL SIGNALS
* Author
        Sreethyan Aravinthan (UCL)
********************************************************************************/

module Control_Unit
(
  input logic [5:0] opcode,
  output logic pc_increment_control,
  output logic [1:0] pc_control,
  output logic general_register_write_enable,
  output logic stack_write_enable,
  output logic stack_control,
  output logic [1:0] write_data_enable,
  output logic [1:0] ALU_source_1,
  output logic [1:0] ALU_source_2,
  output logic [1:0] ALU_control,
  output logic flags_write_enable,
  output logic jump_zero_control,
  output logic jump_below_control,
  output logic jump_below_equal_control,
  output logic jump_above_control,
  output logic jump_above_equal_control,
  output logic jump_greater_control,
  output logic jump_greater_equal_control,
  output logic jump_less_control,
  output logic jump_less_equal_control,
  output logic memory_write_enable,
  output logic [1:0] general_register_result_select,
  output logic enable_rtr,
  output logic gpp_rtr_cp,
  output logic gpp_rtr_dp,
  output logic gpp_trf_dp
);
  logic [30:0] controls;

  always_comb
  begin
    pc_increment_control = controls[30];
    pc_control = controls[29:28];
    general_register_write_enable = controls[27];
    stack_write_enable = controls[26];
    stack_control = controls[25];
    write_data_enable = controls[24:23];
    ALU_source_1 = controls[22:21];
    ALU_source_2 = controls[20:19];
    ALU_control = controls[18:17];
    flags_write_enable = controls[16];
    jump_zero_control = controls[15];
    jump_below_control = controls[14];
    jump_below_equal_control = controls[13];
    jump_above_control = controls[12];
    jump_above_equal_control = controls[11];
    jump_greater_control = controls[10];
    jump_greater_equal_control = controls[9];
    jump_less_control = controls[8];
    jump_less_equal_control = controls[7];
    memory_write_enable = controls[6];
    general_register_result_select = controls[5:4];
    enable_rtr = controls[3];
    gpp_rtr_cp = controls[2];
    gpp_rtr_dp = controls[1];
    gpp_trf_dp = controls[0];
  end

  always_comb
  begin
    case(opcode)
      6'b000000:  // ld
      begin
        controls[30]    = 1'b0;
        controls[29:28] = 2'b00;
        controls[27]    = 1'b1;
        controls[26]    = 1'b0;
        controls[25]    = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b00;
        controls[18:17] = 2'b00;
        controls[16]    = 1'b0;
        controls[15]    = 1'b0;
        controls[14]  = 1'b0;
        controls[13]  = 1'b0;
        controls[12]  = 1'b0;
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b00;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      6'b000001:  // str
      begin
        controls[30]  = 1'b0;
        controls[29:28] = 2'b00;
        controls[27]  = 1'b0;
        controls[26]  = 1'b0;
        controls[25]  = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b00;
        controls[18:17] = 2'b00;
        controls[16]  = 1'b0;
        controls[15]  = 1'b0;
        controls[14]  = 1'b0;
        controls[13]  = 1'b0;
        controls[12]  = 1'b0;
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b1;
        controls[5:4] = 2'b00;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      6'b000010:  // add
      begin
        controls[30]  = 1'b0;
        controls[29:28] = 2'b00;
```

```
      controls[27]   = 1'b1;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]   = 1'b1;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b01;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b000011:  // sub
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b1;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b01;
      controls[16]   = 1'b1;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b01;
      controls[3] = 1'b0;
```

```
  controls[2] = 1'b0;
  controls[1] = 1'b0;
  controls[0] = 1'b0;
end

6'b000100:  // and
begin
  controls[30]  = 1'b0;
  controls[29:28] = 2'b00;
  controls[27]  = 1'b1;
  controls[26]  = 1'b0;
  controls[25]  = 1'b0;
  controls[24:23] = 2'b00;
  controls[22:21] = 2'b00;
  controls[20:19] = 2'b00;
  controls[18:17] = 2'b10;
  controls[16]  = 1'b1;
  controls[15]  = 1'b0;
  controls[14]  = 1'b0;
  controls[13]  = 1'b0;
  controls[12]  = 1'b0;
  controls[11]  = 1'b0;
  controls[10]  = 1'b0;
  controls[9] = 1'b0;
  controls[8] = 1'b0;
  controls[7] = 1'b0;
  controls[6] = 1'b0;
  controls[5:4] = 2'b01;
  controls[3] = 1'b0;
  controls[2] = 1'b0;
  controls[1] = 1'b0;
  controls[0] = 1'b0;
end

6'b000101:  // or
begin
  controls[30]  = 1'b0;
  controls[29:28] = 2'b00;
  controls[27]  = 1'b1;
  controls[26]  = 1'b0;
  controls[25]  = 1'b0;
  controls[24:23] = 2'b00;
  controls[22:21] = 2'b00;
  controls[20:19] = 2'b00;
  controls[18:17] = 2'b11;
  controls[16]  = 1'b1;
  controls[15]  = 1'b0;
  controls[14]  = 1'b0;
  controls[13]  = 1'b0;
```

```
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b01;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b000110:  // mov
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b1;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]   = 1'b0;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b10;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b000111:  // cmp
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b0;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b01;
      controls[16]   = 1'b1;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b001000:  // jz
    begin
      controls[30]   = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b0;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]   = 1'b0;
      controls[15]   = 1'b1;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
```

```
        controls[0] = 1'b0;
      end

      6'b001001:  // jmp
      begin
        controls[30]   = 1'b0;
        controls[29:28] = 2'b01;
        controls[27]   = 1'b0;
        controls[26]   = 1'b0;
        controls[25]   = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b00;
        controls[18:17] = 2'b00;
        controls[16]   = 1'b0;
        controls[15]   = 1'b0;
        controls[14]   = 1'b0;
        controls[13]   = 1'b0;
        controls[12]   = 1'b0;
        controls[11]   = 1'b0;
        controls[10]   = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b00;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      6'b001010:  // movi
      begin
        controls[30]   = 1'b1;
        controls[29:28] = 2'b00;
        controls[27]   = 1'b1;
        controls[26]   = 1'b0;
        controls[25]   = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b00;
        controls[18:17] = 2'b00;
        controls[16]   = 1'b0;
        controls[15]   = 1'b0;
        controls[14]   = 1'b0;
        controls[13]   = 1'b0;
        controls[12]   = 1'b0;
        controls[11]   = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b11;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      6'b001011:  // addi
      begin
        controls[30]  = 1'b1;
        controls[29:28] = 2'b00;
        controls[27]  = 1'b1;
        controls[26]  = 1'b0;
        controls[25]  = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b01;
        controls[18:17] = 2'b00;
        controls[16]  = 1'b1;
        controls[15]  = 1'b0;
        controls[14]  = 1'b0;
        controls[13]  = 1'b0;
        controls[12]  = 1'b0;
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b01;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      6'b001100:  // subi
      begin
        controls[30]  = 1'b1;
        controls[29:28] = 2'b00;
        controls[27]  = 1'b1;
        controls[26]  = 1'b0;
        controls[25]  = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b01;
        controls[18:17] = 2'b01;
        controls[16]  = 1'b1;
        controls[15]  = 1'b0;
        controls[14]  = 1'b0;
        controls[13]  = 1'b0;
        controls[12]  = 1'b0;
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b01;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      6'b001101:  // andi
      begin
        controls[30]  = 1'b1;
        controls[29:28] = 2'b00;
        controls[27]  = 1'b1;
        controls[26]  = 1'b0;
        controls[25]  = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b01;
        controls[18:17] = 2'b10;
        controls[16]  = 1'b1;
        controls[15]  = 1'b0;
        controls[14]  = 1'b0;
        controls[13]  = 1'b0;
        controls[12]  = 1'b0;
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b01;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end
```

```

        6'b001110:  // ori
        begin
          controls[30]   = 1'b1;
          controls[29:28] = 2'b00;
          controls[27]   = 1'b1;
          controls[26]   = 1'b0;
          controls[25]   = 1'b0;
          controls[24:23] = 2'b00;
          controls[22:21] = 2'b00;
          controls[20:19] = 2'b01;
          controls[18:17] = 2'b11;
          controls[16]   = 1'b1;
          controls[15]   = 1'b0;
          controls[14]   = 1'b0;
          controls[13]   = 1'b0;
          controls[12]   = 1'b0;
          controls[11]   = 1'b0;
          controls[10]   = 1'b0;
          controls[9] = 1'b0;
          controls[8] = 1'b0;
          controls[7] = 1'b0;
          controls[6] = 1'b0;
          controls[5:4] = 2'b01;
          controls[3] = 1'b0;
          controls[2] = 1'b0;
          controls[1] = 1'b0;
          controls[0] = 1'b0;
        end

        6'b001111:  // push
        begin
          controls[30]   = 1'b0;
          controls[29:28] = 2'b00;
          controls[27]   = 1'b0;
          controls[26]   = 1'b1;
          controls[25]   = 1'b0;
          controls[24:23] = 2'b00;
          controls[22:21] = 2'b00;
          controls[20:19] = 2'b10;
          controls[18:17] = 2'b00;
          controls[16]   = 1'b0;
          controls[15]   = 1'b0;
          controls[14]   = 1'b0;
          controls[13]   = 1'b0;
          controls[12]   = 1'b0;
          controls[11]   = 1'b0;
          controls[10]   = 1'b0;
          controls[9] = 1'b0;
```

```
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b1;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010000:  // pop
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b1;
      controls[26]   = 1'b1;
      controls[25]   = 1'b1;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b10;
      controls[18:17] = 2'b01;
      controls[16]   = 1'b0;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010001:  // call
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b01;
      controls[27]   = 1'b0;
      controls[26]   = 1'b1;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b01;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b10;
      controls[18:17] = 2'b00;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b1;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010010:  // return
    begin
      controls[30]  = 1'b0;
      controls[29:28] = 2'b10;
      controls[27]  = 1'b0;
      controls[26]  = 1'b1;
      controls[25]  = 1'b1;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b10;
      controls[18:17] = 2'b01;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010011:  // jb
```

```
begin
  controls[30]   = 1'b1;
  controls[29:28] = 2'b00;
  controls[27]   = 1'b0;
  controls[26]   = 1'b0;
  controls[25]   = 1'b0;
  controls[24:23] = 2'b00;
  controls[22:21] = 2'b00;
  controls[20:19] = 2'b00;
  controls[18:17] = 2'b00;
  controls[16]   = 1'b0;
  controls[15]   = 1'b0;
  controls[14]   = 1'b1;
  controls[13]   = 1'b0;
  controls[12]   = 1'b0;
  controls[11]   = 1'b0;
  controls[10]   = 1'b0;
  controls[9] = 1'b0;
  controls[8] = 1'b0;
  controls[7] = 1'b0;
  controls[6] = 1'b0;
  controls[5:4] = 2'b00;
  controls[3] = 1'b0;
  controls[2] = 1'b0;
  controls[1] = 1'b0;
  controls[0] = 1'b0;
end

6'b010100:  // jbe
begin
  controls[30]   = 1'b1;
  controls[29:28] = 2'b00;
  controls[27]   = 1'b0;
  controls[26]   = 1'b0;
  controls[25]   = 1'b0;
  controls[24:23] = 2'b00;
  controls[22:21] = 2'b00;
  controls[20:19] = 2'b00;
  controls[18:17] = 2'b00;
  controls[16]   = 1'b0;
  controls[15]   = 1'b0;
  controls[14]   = 1'b0;
  controls[13]   = 1'b1;
  controls[12]   = 1'b0;
  controls[11]   = 1'b0;
  controls[10]   = 1'b0;
  controls[9] = 1'b0;
  controls[8] = 1'b0;
  controls[7] = 1'b0;
```

```
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010101:  // ja
    begin
      controls[30]  = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
      controls[26]  = 1'b0;
      controls[25]  = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b1;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010110:  // jae
    begin
      controls[30]  = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
      controls[26]  = 1'b0;
      controls[25]  = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b1;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b010111:  // jg
    begin
      controls[30]  = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
      controls[26]  = 1'b0;
      controls[25]  = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b1;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b011000:  // jge
    begin
      controls[30]  = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
      controls[26]  = 1'b0;
      controls[25]  = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b1;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b011001:  // jl
    begin
      controls[30]  = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
      controls[26]  = 1'b0;
      controls[25]  = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b1;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
```

```
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b011010:  // jle
    begin
      controls[30]   = 1'b1;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b0;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]   = 1'b0;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b1;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b011011:  // cbt
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b0;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b01;
      controls[20:19] = 2'b10;
      controls[18:17] = 2'b01;
      controls[16]   = 1'b1;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b011100:  // trf
    begin
      controls[30]  = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
      controls[26]  = 1'b1;
      controls[25]  = 1'b1;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b10;
      controls[18:17] = 2'b01;
      controls[16]  = 1'b0;
      controls[15]  = 1'b0;
      controls[14]  = 1'b0;
      controls[13]  = 1'b0;
      controls[12]  = 1'b0;
      controls[11]  = 1'b0;
      controls[10]  = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b0;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b0;
      controls[0] = 1'b1;
    end

    6'b011101:  // pr
    begin
      controls[30]  = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]  = 1'b0;
    controls[26]  = 1'b0;
    controls[25]  = 1'b0;
    controls[24:23] = 2'b00;
    controls[22:21] = 2'b00;
    controls[20:19] = 2'b00;
    controls[18:17] = 2'b00;
    controls[16]  = 1'b0;
    controls[15]  = 1'b0;
    controls[14]  = 1'b0;
    controls[13]  = 1'b0;
    controls[12]  = 1'b0;
    controls[11]  = 1'b0;
    controls[10]  = 1'b0;
    controls[9] = 1'b0;
    controls[8] = 1'b0;
    controls[7] = 1'b0;
    controls[6] = 1'b0;
    controls[5:4] = 2'b00;
    controls[3] = 1'b1;
    controls[2] = 1'b1;
    controls[1] = 1'b0;
    controls[0] = 1'b0;
  end

  6'b011110:  // cbr
  begin
    controls[30]  = 1'b0;
    controls[29:28] = 2'b00;
    controls[27]  = 1'b0;
    controls[26]  = 1'b0;
    controls[25]  = 1'b0;
    controls[24:23] = 2'b00;
    controls[22:21] = 2'b10;
    controls[20:19] = 2'b10;
    controls[18:17] = 2'b01;
    controls[16]  = 1'b1;
    controls[15]  = 1'b0;
    controls[14]  = 1'b0;
    controls[13]  = 1'b0;
    controls[12]  = 1'b0;
    controls[11]  = 1'b0;
    controls[10]  = 1'b0;
    controls[9] = 1'b0;
    controls[8] = 1'b0;
    controls[7] = 1'b0;
    controls[6] = 1'b0;
    controls[5:4] = 2'b00;
    controls[3] = 1'b0;
    controls[2] = 1'b0;
```

```
      controls[1] = 1'b0;
      controls[0] = 1'b0;
    end

    6'b011111:  // rtr
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b0;
      controls[26]   = 1'b1;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b10;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b10;
      controls[18:17] = 2'b00;
      controls[16]   = 1'b0;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
      controls[11]   = 1'b0;
      controls[10]   = 1'b0;
      controls[9] = 1'b0;
      controls[8] = 1'b0;
      controls[7] = 1'b0;
      controls[6] = 1'b1;
      controls[5:4] = 2'b00;
      controls[3] = 1'b0;
      controls[2] = 1'b0;
      controls[1] = 1'b1;
      controls[0] = 1'b0;
    end

    6'b100000:  // rr
    begin
      controls[30]   = 1'b0;
      controls[29:28] = 2'b00;
      controls[27]   = 1'b0;
      controls[26]   = 1'b0;
      controls[25]   = 1'b0;
      controls[24:23] = 2'b00;
      controls[22:21] = 2'b00;
      controls[20:19] = 2'b00;
      controls[18:17] = 2'b00;
      controls[16]   = 1'b0;
      controls[15]   = 1'b0;
      controls[14]   = 1'b0;
      controls[13]   = 1'b0;
      controls[12]   = 1'b0;
```

```
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b00;
        controls[3] = 1'b1;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

      default:
      begin
        controls[30]  = 1'b0;
        controls[29:28] = 2'b00;
        controls[27]  = 1'b0;
        controls[26]  = 1'b0;
        controls[25]  = 1'b0;
        controls[24:23] = 2'b00;
        controls[22:21] = 2'b00;
        controls[20:19] = 2'b00;
        controls[18:17] = 2'b00;
        controls[16]  = 1'b0;
        controls[15]  = 1'b0;
        controls[14]  = 1'b0;
        controls[13]  = 1'b0;
        controls[12]  = 1'b0;
        controls[11]  = 1'b0;
        controls[10]  = 1'b0;
        controls[9] = 1'b0;
        controls[8] = 1'b0;
        controls[7] = 1'b0;
        controls[6] = 1'b0;
        controls[5:4] = 2'b00;
        controls[3] = 1'b0;
        controls[2] = 1'b0;
        controls[1] = 1'b0;
        controls[0] = 1'b0;
      end

    endcase
  end

endmodule
```

Listing A.3: Control_Unit

Data_Memory.sv

```
/********************************************************************************
* File name
	Data_Memory.sv
* Description
	This module describes RAM.
* Parameters
	WIDTH - This is the WIDTH of the input/output data and the address used to read/write.
* Inputs
	clk - The clock for the system.

	memory_write_enable - This is a control signal that indicates if the data at the input should be written or not.

	address_rw - This is the address that is used to read/write.

	data_in - This is the data that will be written to the memory at the address given at the input.
* Outputs
	data_out - This is the data that is read from the address at the input.
* Author
	Sreethyan Aravinthan (UCL)
*******************************************************************************/

module Data_Memory
#(
  parameter WIDTH = 1
)
(
  input logic clk,
  input logic memory_write_enable,
  input logic [WIDTH - 1:0]address_rw,
  input logic [WIDTH - 1:0]data_in,
  output logic [WIDTH - 1:0]data_out
);

  logic [WIDTH - 1:0]RAM[2**WIDTH - 1:0];

  always_comb
  begin
    data_out = RAM[address_rw];
  end

  always_ff@(posedge clk)
  begin
    if(memory_write_enable == 1'b1)
    begin
```

```
      RAM[address_rw] = data_in;
    end
  end

endmodule
```

Listing A.4: Data_Memory

Flags_Register.sv

```
/*
   ↪ ********************************************************************************
   ↪
* File name
        Flags_Register.sv
* Description
        This module is a register with an enable signal. This is for the flags.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        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.

        d - This is the input data to the flags register.
* Outputs
        q - This is the output data to the flags register.
* Author
        Sreethyan Aravinthan (UCL)
*******************************************************************************/

module Flags_Register
(
  input logic clk,
  input logic rst,
  input logic flags_reg_write_enable,
  input logic d[3:0],
  output logic q[3:0]
);

  logic zero[3:0];

  always_comb
  begin
    zero[3] = 1'b0;
    zero[2] = 1'b0;
    zero[1] = 1'b0;
    zero[0] = 1'b0;
  end

  // asynchronus register
  always_ff@(posedge clk, posedge rst)
  begin
    // if reset is 1 then set the output to 0
    if(rst == 1)
    begin
      q <= zero;
    end
    // else set the output to the input value
    else if(flags_reg_write_enable == 1)
    begin
      q <= d;
    end
  end

endmodule
```

Listing A.5: Flags_Register

General_Purpose_Register_File.sv

```
/* ******************************************************************************
* File name
	General_Purpose_Register_File.sv
* Description
	This module describes a register file.
* Parameters
	ADDR_WIDTH_RF - This indicates the width of the address.

	DATA_WIDTH - This is the width of the data written or read.
* Inputs
	clk - The clock for the system.

	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.

	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.

	address_1 - This indicates which register to be accesed in the register file.

	address_2 - This indicates which register to be accesed in the register file.

```

```
      address_3 − This indicates which register to be accesed in the register file.

      general_register_write_data − This is the data that can be written to any register apart
    ↪ from the 1st register in the register file.

      stack_register_write_data − This is the data that can be written to only the 1st register
    ↪ in the register file.
* Outputs
      read_data_1 − This is the data that is read from the register file at address_1.

      read_data_2 − This is the data that is read from the register file at address_2.
* Author
      Sreethyan Aravinthan (UCL)
*****************************************************************************/

module General_Purpose_Register_File
#(
  parameter ADDR_WIDTH_RF = 1,
  parameter DATA_WIDTH = 1
)
(
  input logic clk,
  input logic general_register_write_enable,
  input logic stack_write_enable,
  input logic [ADDR_WIDTH_RF − 1:0] address_1,
  input logic [ADDR_WIDTH_RF − 1:0] address_2,
  input logic [ADDR_WIDTH_RF − 1:0] address_3,
  input logic [DATA_WIDTH − 1:0] general_register_write_data,
  input logic [DATA_WIDTH − 1:0] stack_register_write_data,
  output logic [DATA_WIDTH − 1:0] read_data_1,
  output logic [DATA_WIDTH − 1:0] read_data_2
);

  // regsiter file
  logic [DATA_WIDTH − 1:0]registers[2**ADDR_WIDTH_RF − 1:0];

  // data read from address pointed by address_1 and address_2
  always_comb
  begin
    read_data_1 = registers[address_1];
    read_data_2 = registers[address_2];
  end

  // write data if enable signal is set and positive edge of the clock
  always_ff@(posedge clk)
  begin
    if(general_register_write_enable == 1'b1)
    begin
      registers[address_3] <= general_register_write_data;
    end
    if(stack_write_enable == 1'b1)
    begin
      registers[0] <= stack_register_write_data;
    end
  end

endmodule
```

Listing A.6: General_Purpose_Register_File

Instruction_Memory.sv

```
/*
    ↪ *******************************************************************************
    ↪
* File name
        Instruction_Memory.sv
* Description
        This module describes a 2−port ROM.
* Parameters
        ADDR_WIDTH_IM − This is the width of the address field.

        INSTR_WIDTH − This is the width of the instruction.

        FILE_NAME − This is the name of the file that will be loaded and set as default values.
* Inputs
        address_1 − This is the address that is wished to be accessed.

        address_2 − This is the address that is wished to be accessed.
* Outputs
        read_data_1 − This is the value at address_1.

        read_data_2 − This is the value at address_2.
* Author
        Sreethyan Aravinthan (UCL)
*******************************************************************************/

module Instruction_Memory
#(
  parameter ADDR_WIDTH_IM = 1,
  parameter INSTR_WIDTH = 1,
  parameter FILE_NAME = "out.mem"
)
(
  input logic [ADDR_WIDTH_IM − 1:0]address_1,
  input logic [ADDR_WIDTH_IM − 1:0]address_2,
  output logic [INSTR_WIDTH − 1:0]read_data_1,
  output logic [INSTR_WIDTH − 1:0]read_data_2
);
```

```
  logic [INSTR_WIDTH − 1:0]ROM[2∗∗ADDR_WIDTH_IM − 1:0];

  initial
  begin
    $readmemb(FILE_NAME, ROM);
  end

  always_comb
  begin
    read_data_1 = ROM[address_1];
    read_data_2 = ROM[address_2];
  end

endmodule
```

Listing A.7: Instruction_Memory

Jump_Logic.sv

```
/*
    ↪ ********************************************************************************
    ↪
∗ File name
        Jump_Logic.sv
∗ Description
        This module describes how the flags for the jump instructions should be set so that the
    ↪ instructions work.
∗ Parameters
        NONE
∗ Inputs
        INPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO THE
    ↪  DATAPATH
∗ Outputs
        OUTPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO
    ↪ THE DATAPATH
∗ Author
        Sreethyan Aravinthan (UCL)
********************************************************************************/

module Jump_Logic
(
  // control signals
  input logic jump_zero_control,
  input logic jump_below_control,
  input logic jump_below_equal_control,
  input logic jump_above_control,
  input logic jump_above_equal_control,
  input logic jump_greater_control,
  input logic jump_greater_equal_control,
  input logic jump_less_control,
  input logic jump_less_equal_control,
  // flags
  input logic zero_flag,
  input logic carry_flag,
  input logic sign_flag,
  input logic overflow_flag,
  output logic jump_logic_out
);

  logic jz_out;
  logic jb_out;
  logic jbe_out;
  logic jbe_1_out;
  logic ja_out;
  logic ja_1_out;
  logic jae_out;
  logic jae_1_out;
  logic jg_out;
  logic jg_1_out;
  logic jg_2_out;
  logic jg_3_out;
  logic jge_out;
  logic jge_1_out;
  logic jl_out;
  logic jl_1_out;
  logic jle_out;
  logic jle_1_out;
  logic jle_2_out;

  // instantiate logic for jump zero instruction
  and jz(jz_out, jump_zero_control, zero_flag);

  // isntantiate logic for jump below instrucion
  and jb(jb_out, jump_below_control, carry_flag);

  // instantiate logic doe the jump below equal instruction
  or jbe_1(jbe_1_out, carry_flag, zero_flag);
  and jbe(jbe_out, jump_below_equal_control, jbe_1_out);

  // instantiate logic for jump above instruction
  nand ja_1(ja_1_out, carry_flag, zero_flag);
  and ja(ja_out, jump_above_control, ja_1_out);

  // instantiate logic for jump above equal instruction
  not jae_1(jae_1_out, carry_flag);
  and jae(jae_out, jump_above_equal_control, jae_1_out);

  // instantiate logic for jump greater instruction
  not jg_1(jg_1_out, zero_flag);
  xnor jg_2(jg_2_out, sign_flag, overflow_flag);
  and jg_3(jg_3_out, jg_1_out, jg_2_out);
  and jg(jg_out, jump_greater_control, jg_3_out);

  // instantiate logic for jump greater equal instruction
  xnor jge_1(jge_1_out, sign_flag, zero_flag);
  and jge(jge_out, jump_greater_equal_control, jge_1_out);

  // instantiate logic for jump less instruction
  xor jl_1(jl_1_out, sign_flag, zero_flag);
  and jl(jl_out, jump_less_control, jl_1_out);

  // instantiate logic for jump less equal instruction
  xor jle_1(jle_1_out, sign_flag, carry_flag);
  or jle_2(jle_2_out, zero_flag, jle_1_out);
  and jle(jle_out, jump_less_equal_control, jle_2_out);

  or selctor(jump_logic_out, jz_out, jb_out, jbe_out, ja_out, jae_out, jg_out, jge_out, jl_out,
     ↪ jle_out);

endmodule
```

Listing A.8: Jump_Logic

Multiplexer.sv

```
/*
   ↪ ********************************************************************************
   ↪
* File name
      Multiplexer.sv
* Description
      This module describes how a multiplexer works.
* Parameters
      NUM_OF_CONTROL_SIGNALS - Number of control signals

      WIDTH - Number of bits for each input in the multiplexer
* Inputs
      control_signals - Used to select a certain input data.

      data - Set of inputs put sent into the multiplexer.
* Outputs
      multiplexer_out - Output data choosen from the set of inputs.
* Author
      Sreethyan Aravinthan (UCL)
********************************************************************************/

module Multiplexer
#(
  parameter NUM_OF_CONTROL_SIGNALS = 1,
  parameter WIDTH = 1
)
(
  input logic [NUM_OF_CONTROL_SIGNALS − 1:0]control_signals,
  input logic [WIDTH − 1:0]data[2**NUM_OF_CONTROL_SIGNALS − 1:0],
  output logic [WIDTH − 1:0]multiplexer_out
);

  // combinational logic
  always_comb
  begin
    // select the correct data
    multiplexer_out = data[control_signals];
  end

endmodule
```

Listing A.9: Multiplexer

Register.sv

```
/*
    ↪ ********************************************************************************
    ↪
* File name
        Register.sv
* Description
        This module describes how a register.
* Parameters
        WIDTH − Number of bits for the input data.
* Inputs
        clk − The clock for the system.

        rst − Signal to reset the system to the default values.

        d − This is the input to the register.
* Outputs
        q − This is the output of the register.
* Author
        Sreethyan Aravinthan (UCL)
********************************************************************************/

module Register
#(
  parameter WIDTH = 1
)
(
  input logic clk,  // clock signal
  input logic rst,  // reset signal
  input logic [WIDTH - 1:0] d,  // input signal
  output logic [WIDTH -1:0] q // output signal
);

  // asynchronus register
  always_ff@(posedge clk, posedge rst)
  begin
    // if reset is 1 then set the output to 0
    if(rst == 1)
    begin
      q <= 0;
    end
    else  // else set the output to the input value
    begin
      q <= d;
    end
  end

endmodule
```

Listing A.10: Register

Datapath.sv

```
/*********************************************************************************
* File name
        Datapath.sv
* Description
        This module describes how the components of the datapath are connected.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        OTHER INPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO THE DATAPATH
* Outputs
        OUTPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO THE DATAPATH
* Author
        Sreethyan Aravinthan (UCL)
*******************************************************************************/

`include "ALU.sv"
`include "ALU_advanced.sv"
`include "Multiplexer.sv"
```

```
‘include ”Register.sv”
‘include ”General_Purpose_Register_File.sv”
‘include ”Flags_Register.sv”
‘include ”Jump_Logic.sv”

module Datapath
(
  input logic clk,
  input logic rst,
  // Instruction memory
  output logic [15:0]address_1,
  output logic [15:0]address_2,
  input logic [15:0]read_address_1,
  input logic [15:0]read_address_2,
  // Data memory
    output logic [15:0]address_rw,
    output logic [15:0]data_in,
    input logic [15:0]data_out,
  // Control signals
  input logic pc_increment_control,
  input logic [1:0] pc_control,
  input logic general_register_write_enable,
  input logic stack_write_enable,
  input logic stack_control,
  input logic [1:0] write_data_enable,
  input logic [1:0] ALU_source_1,
  input logic [1:0] ALU_source_2,
  input logic [1:0] ALU_control,
  input logic flags_write_enable,
  input logic jump_zero_control,
  input logic jump_below_control,
  input logic jump_below_equal_control,
  input logic jump_above_control,
  input logic jump_above_equal_control,
  input logic jump_greater_control,
  input logic jump_greater_equal_control,
  input logic jump_less_control,
  input logic jump_less_equal_control,
  input logic [1:0] general_register_result_select,
  // Communications Processor Signals
  input logic [15:0] RAM_rx_data_out, // gpp
  input logic data_rx_flag, // gpp
  input logic gpp_trf_cp, // gpp
  output logic [15:0] gpp_tx_data // gpp
);

  // constants
  localparam [15:0] zero = 16’h0000;
  localparam [15:0] one = 16’h0001;
  localparam [15:0] two = 16'h0002;

  // signals
  logic [15:0] branch_mux_data[1:0];
  logic [15:0] branch_mux_out;
  logic [15:0] general_register_result_select_out;
  logic [15:0] pc_control_data_mux [3:0];
  logic [15:0] next_PC;
  logic branch_result;
  logic pc_increment_choose_out;
  logic [15:0] pc_increment_mux_out;
  logic [15:0] pc_increment_data_mux [1:0];
  logic [15:0] ALU_out;
  logic [15:0] WD_control_data_mux[3:0];
  logic [15:0] ALU_source_2_data_mux[3:0];
  logic [15:0] ALU_source_2_mux_out;
  logic zero_flag;
  logic carry_flag;
  logic sign_flag;
  logic overflow_flag;
  logic [15:0] general_register_result_select_data_mux [3:0];
  logic [15:0] reg_read_data_1;
  logic [15:0] stack_control_data_mux [1:0];
  logic [15:0] stack_control_mux_out;
  logic flags_in [3:0];
  logic flags_out [3:0];

  logic [15:0] ALU_source_1_data_mux[3:0];
  logic [15:0] ALU_source_1_mux_out;

  always_comb
  begin
    branch_mux_data[1] = read_address_2;

    pc_control_data_mux[0] = branch_mux_out;
    pc_control_data_mux[1] = read_address_2;
    pc_control_data_mux[2] = general_register_result_select_out;
    // pc_control_data_mux[3] = ;

    pc_increment_data_mux[0] = one;
    pc_increment_data_mux[1] = two;

    ALU_source_2_data_mux[0] = data_in;
    ALU_source_2_data_mux[1] = read_address_2;
    ALU_source_2_data_mux[2] = one;
    // ALU_source_2_data_mux

    general_register_result_select_data_mux[0] = data_out;
    general_register_result_select_data_mux[1] = ALU_out;
```

```verilog
  general_register_result_select_data_mux[2] = reg_read_data_1;
  general_register_result_select_data_mux[3] = read_address_2;

  stack_control_data_mux[0] = zero;
  stack_control_data_mux[1] = one;

  flags_in[3] = zero_flag;
  flags_in[2] = carry_flag;
  flags_in[1] = sign_flag;
  flags_in[0] = overflow_flag;

  WD_control_data_mux[2] = RAM_rx_data_out;

  ALU_source_1_data_mux[0] = reg_read_data_1;
  ALU_source_1_data_mux[1] = gpp_trf_cp;
  ALU_source_1_data_mux[2] = data_rx_flag;
  // ALU_source_1_data_mux_out[3] =

  gpp_tx_data = general_register_result_select_out;
end

// instantiating branch multiplexer
Multiplexer #(1, 16) branch_multiplexer(branch_result, branch_mux_data, branch_mux_out);

// instantiating pc control multiplexer
Multiplexer #(2, 16) pc_control_multiplexer(pc_control, pc_control_data_mux, next_PC);

// instantiate pc register
Register #(16) pc_register(clk, rst, next_PC, address_1);

// instantiate adder to see the next instruction
ALU #(16) pc_adder(address_1, one, 2'b00, address_2);

// instantiate xor for which will be the control for choosing if the increment is 1 or 2
xor pc_increment_choose(pc_increment_choose_out, branch_result, pc_increment_control);

// instantiate pc increment multiplexer
Multiplexer #(1, 16) pc_increment_multiplexer(pc_increment_choose_out, pc_increment_data_mux, pc_increment_mux_out);

// instantiate pc increment adder
ALU #(16) pc_increment_adder(address_1, pc_increment_mux_out, 2'b00, branch_mux_data[0]);

// instantiate register file
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]);

```

```
  // instantiate stack control multiplexer
  Multiplexer #(1, 16) stack_control_multiplexer(stack_control, stack_control_data_mux,
     ↪ stack_control_mux_out);

  // instantiate a subber for correct stack access
  ALU #(16) stack_access_subber(reg_read_data_1, stack_control_mux_out, 2'b01, address_rw);

  // instantiate WD control multiplexer
  Multiplexer #(2, 16) WD_control_multiplexer(write_data_enable, WD_control_data_mux,
     ↪ data_in);

  // instantiate adder for return address
  ALU #(16) return_address_adder(address_2, one, 2'b00, WD_control_data_mux[1]);

  // instantiate ALU source 1 multiplexer
  Multiplexer #(2, 16) ALU_source_1_multiplexer(ALU_source_1, ALU_source_1_data_mux,
     ↪ ALU_source_1_mux_out);

  // instantiate ALU source 2 multiplexer
  Multiplexer #(2, 16) ALU_source_2_multiplexer(ALU_source_2, ALU_source_2_data_mux,
     ↪ ALU_source_2_mux_out);

  // instantiate ALU
  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);

  // instantiate zero register
  Flags_Register flag_reg(clk, rst, flags_write_enable, flags_in, flags_out);

  // instantiate jump logic
  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);

  // instantiate general register result selector
  Multiplexer #(2, 16) general_register_result_select_multiplexer(general_register_result_select,
     ↪ general_register_result_select_data_mux, general_register_result_select_out);

endmodule
```

Listing A.11: Datapath

GPP.sv

```
/*
   ↪ ****************************************************************************
   ↪
* File name
      GPP.sv
```

```
* Description
        This module instantiates the Datapath and control unit and connects them together.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        OTHER INPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO THE DATAPATH
* Outputs
        OUTPUTS ARE CONNECTED TO THE OTHER BLOCKS THAT CONNECT TO THE DATAPATH
* Author
        Sreethyan Aravinthan (UCL)
*****************************************************************************/

module GPP
(
  input logic clk,
    input logic rst,
    // Instruction memory
    output logic [15:0]address_1,
    output logic [15:0]address_2,
    input logic [15:0]read_address_1,
    input logic [15:0]read_address_2,
    // Data memory
    output logic [15:0]address_rw,
    output logic [15:0]data_in,
    input logic [15:0]data_out,
  // Control Signals
  output logic memory_write_enable,
  output logic enable_rtr,
  output logic gpp_rtr_cp,
  output logic gpp_rtr_dp,
  output logic gpp_trf_dp,
  // Communications Processor Signals
  input logic [15:0] RAM_rx_data_out, // gpp
  input logic data_rx_flag, // gpp
  input logic gpp_trf_cp, // gpp
  output logic [15:0] gpp_tx_data // gpp
);

    // Control signals
  logic pc_increment_control;
  logic [1:0] pc_control;
  logic general_register_write_enable;
  logic stack_write_enable;
  logic stack_control;
  logic [1:0] write_data_enable;
  logic [1:0] ALU_source_1;
  logic [1:0] ALU_source_2;
  logic [1:0] ALU_control;
  logic flags_write_enable;
  logic jump_zero_control;
  logic jump_below_control;
  logic jump_below_equal_control;
  logic jump_above_control;
  logic jump_above_equal_control;
  logic jump_greater_control;
  logic jump_greater_equal_control;
  logic jump_less_control;
  logic jump_less_equal_control;
  logic [1:0] general_register_result_select;

  Datapath datapath_processor(clk,
              rst,
              address_1,
              address_2,
              read_address_1,
              read_address_2,
              address_rw,
              data_in,
              data_out,
              pc_increment_control,
              pc_control,
              general_register_write_enable,
              stack_write_enable,
              stack_control,
              write_data_enable,
              ALU_source_1,
              ALU_source_2,
              ALU_control,
              flags_write_enable,
              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,
              general_register_result_select,
              RAM_rx_data_out, // gpp
              data_rx_flag, // gpp
              gpp_trf_cp, // gpp
              gpp_tx_data); // gpp


    Control_Unit cu_processor   (read_address_1[15:10],
              pc_increment_control,
              pc_control,
              general_register_write_enable,
              stack_write_enable,
              stack_control,
              write_data_enable,
              ALU_source_1,
              ALU_source_2,
              ALU_control,
              flags_write_enable,
              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,
              memory_write_enable,
              general_register_result_select,
              enable_rtr,
              gpp_rtr_cp,
              gpp_rtr_dp,
              gpp_trf_dp);

  endmodule
```

Listing A.12: GPP

Appendix B

CP

Control_Plane.sv

```
/*
↪ *****************************************************************************
↪
* File name
        comms_processor.sv
* Description
        This module describes how the control plane works.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        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.

        max_node - This shows the maximum number of nodes in the ring topology.

        data_rx_complete_flag - This flag will reset the data_rx_flag. The value for this is sent
    ↪ from the data plane reciever.

        control_rx_packet - This is the packet that will be recieved on the control plane.

        data_tx_complete_flag - This flag will reset the data_tx_flag. The value for this is sent
    ↪ from the data plane transmitter.

        RAM_tx_data_out - This input has the value equal to the top of the RAM of the data
    ↪ plane tx RAM.

```

```
        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.

        enable_rtr - This is a control signal that will allow the GPP to retrive data from the CP
     ↪  if it is possible.

        gpp_rtr_cp - This is a control signal that pauses rx from gpp so that gpp can retrive
     ↪ data from rx RAM.
* Outputs
        control_tx_packet - This is the packet that is transmitted on the control plane.

        data_rx_node_id - This is the value that is sent to a control unit which sets the
     ↪ wavelength/spatial channel of the reciever.

        data_tx_flag - This is a flag which indicates if the node is transmitting data on the data
     ↪ plane.

        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 retrive data from the data plane receiver
     ↪ . This will be connected to the ALU_src_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. This will be connected to the ALU_src_1.
* Author
        Sreethyan Aravinthan (UCL)
******************************************************************************/

module Control_Plane
(
  input logic clk,
  input logic rst,
  input shortint node_id,
  input shortint max_node,
  input logic data_rx_complete_flag,  // ***
  input logic [31:0] control_rx_packet,
  input logic data_tx_complete_flag,  // ***
  input logic [15:0] RAM_tx_data_out, // ***
  input logic [15:0] sp_tx_current, // ***
  input logic enable_rtr,
  input logic gpp_rtr_cp,
  output logic [31:0] control_tx_packet,
  output logic [15:0] data_rx_node_id,
  output logic data_tx_flag,  // ***
  output logic data_rx_flag,
  output logic gpp_trf_cp
);

  // COMMENT ALL THE CODE AND MAKE SURE ALL THE VARIABLE NAMES ARE
  // SNESIBLE. FURTHERMORE THE VAIRABLES THAT ARE LINKED BETWEEN
      ↪ THE
  // MODULES SHOULD HAVE THE SAME NAME TO AVOID CONFUSION

  // MAKE SURE THE CORRECT CONTROL SIGNALS AND LOGIC IS APPLIED TO
      ↪ THE
  // SYSTEM. BELOW IS HOW TO DO IT FOR BOTH THE TX AND RX
  //
  // THEN MAKE THE LINK BETWEEN GPP AND COMMS_PROCESSOR
  //
  // THEN CREATE THE INSTRUCTIONS THAT ARE NECESSARY

  // HOW SHOULD THE DATA FROM THE GPP RAM BE PLACED INTO THE TX
      ↪ RAM
  // MAKE PHYSICAL CONNECTION BETWEEN GPP RAM AND TX RAM
  // OUTPUT FROM THE CONTROL PLANE IF THERE ARE ENOUGH CLOCK
     ↪ CYCLES TO
  // THIS RESULT SHOULD BE CONNECTED TO ALU SRC 1
  // UPON CHECKING IF IT IS OK TO TRANSMIT THEN TRANSFER THE DATA
  // ELSE DO NOT TRANSMIT AND TRY AGAIN LATER
  // NEED TO CREATE TWO INSTRUCTIONS
  // 1ST INSTR: TO COMPARE SEE IF THE NUMBER OF CYCLES ARE ENOUGH
  // 2ND INSTR: TRANSFER THE DATA FROM THE GPP RAM TO TX RAM

  /*
  * transfer_func:
  *   cbt # compare the value to see if data cane be transferred
  *   jnz exit # if the result is not equal to zero then data cannot be
          transfered
    # transfer 5 packet data
    trf
    trf
    trf
    trf
    trf
  * exit:
    return
  */

        // HOW SHOULD THE DATA FROM THE RX RAM O THE GPP RAM BE
     ↪ PLACED
  // MAKE THE CONNECTIONS BETWEEN THE GPP AND THE DATA RX
  // SET CONTROL SIGNAL FROM THE GPP TO THE CONTROL PLANE
  // THEN THIS WILL MAKE SURE NOT TO SET THE RX FLAG IF A PING COMES
  // THEN COMPARE THE RX FLAG OUTPUT FROM THE CONTROL PLANE TO SEE
     ↪  IF
  // THE DATA CAN BE RETRIEVED
  // IF THE COMPARISION LEADS TO THE CONCLUSION SUCH THAT IT CANNOT
     ↪ BE
```

```
// RETRIVED NOW THEN DO NOT TRY TO RETRIVE NOW
// IF THE COMPARISON LEADS TO THE CONCLUSION SUCH THAT THE DATA
    ↪ CAN BE
// RETRIVED FROM THE DATA PLANE THEN RETRIVE THE PACKET
// 4 INSTRUCTIONS NEED TO BE DESIGNED
// 1ST INSTR: SET THE CONTROL SIGNAL TO PAUSE THE RECIEVE IN THE
// CONTROL PLANE
// 2ND INSTR: CHECK IF THE DATA CAN BE RETRIEVED FROM THE RX RAM
// 3RD INSTR: RETRIEVE THE DATA FROM THE RX RAM TO THE GPP RAM
// 4TH INSTR: RESUME RECEVING THE DATA ON THE CONTROL PLANE

/*
* pr  # instruction 'pause recieving' this stops the control
*   plane from saying yes if it receives a ping
* cbr # instruction which sees if the data can be retrived
*   # instruction is called 'can be retrived'
*   # Remember the data_rx_flag is connected to the
*   # gpp. specfically the ALU src 1
* jz exit # if the result is zero then this indicates that the
*   data plane is reciving. therefore do not retrive the
*   data.
* rtr
* rtr
* rtr
* rtr
* rtr
*exit:
* rr  # instruction 'resume receving'
* return
*/

// THINGS TO DO
// PART 1

// go through the design of the system and find out hot the control
// signals from gpp should be connected with the control and data
// plane
// ADD CONTROL SIGNAL FROM GPP
// OUTPUT THE DATA_RX_FLAG TO SEE IF THE GPP CAN GO AND RETIRVE THE
// DATA FROM RAM
// OUTPUT THE CURRENT SLOT AND TX_FLAG(ALREADY DONE AS
    ↪ DATA_TX_FLAG)
// BOTH WILL BE USED TO SEE IF THE GPP CAN PUT DATA INTO THIS RAM
// THIS IS DONE BY SEEING IF THE NUMBER OF CLOCK CYCLES IS GREATER
// THAN THE MINIMUM NUMBER OF CYCLES REQUIRED TO TRANSFER THE
    ↪ DATA

// variables

// the system is desgined such that each slot on the control plane is
// equal to n*clock_period. Therefore to count to n the variable count
// exists which increases with each clock cycle and resets to 0 when
// it hits the max value such that the count*clock_period ==
// n*clock_period
logic [1:0] count;
// This variable holds the current slot on the control plane. By
// default it will equal to 1. However, it will increment to the
// maximum number of nodes in the network. therefore the type of this
// variable is shortint as this is a 16bit value and our system aims
// to hold 2^10 nodes
shortint slot;
// this is a flag and it is used so that a ping is not made more than
// once in the allocated slot on the control plane for a specfic node
logic control_tx_flag;
// defining constants which are used for the ping response
localparam [15:0] all_zero = 16'h0000;
localparam [15:0] all_one = 16'hFFFF;
shortint num_of_slots = 0;
logic gpp_rtr = 1'b0;

always_comb
begin
  if(enable_rtr == 1'b1)
  begin
    gpp_rtr = gpp_rtr_cp;
  end
end

// increment counter for each clock cycle
// 100 Gb/s transcieverer. Find the number of bits sending (32 bits). Math. Make as short as possible. nut limited by clock period of processor
// 32 / 100*10^9 = 3.2*10^-10s: it takes this long to send a packet
// therefore it will take 3.125*10^9 Hz
// set clock speed to 3*10^9 Hz
// let 1 slot to be 3 clock cycle

// this logic deals with setting the value of the current slot. thus
// indicating if the curent node can transmit on the control plane or not
// provided reset is set to 0
always_ff @(posedge clk)
begin
  // this logic sets the default values of count and slot
  // provided reset is asserted
  if(rst == 1'b1)
  begin
    count <= 2'b00;
    slot <= 1;
  end
  else
  begin
    // this means one slot time has occured
    if(count == 2)
    begin
      // if the slot is equal to max number of nodes in the
      // network then reset the slot to 1
      if(slot == max_node)
      begin
        slot <= 1;
      end
      // otherwise just increment the slot by one to
      // show that the next node can now transmit
      else
      begin
        slot <= slot + 1;
      end
      // reset count to 0
      count <= 2'b00;
    end
    // increment the count value
    else
    begin
      count <= count + 1;
    end
  end
end

// control plane logic
// provided reset is set to 0
always_ff @(posedge clk)
begin
  // this sets the default values of all the variables that is used
  // within the control plane
  // provided reset is set to 1
  if(rst == 1'b1)
  begin
    control_tx_packet <= 32'b0;
    data_rx_flag <= 1'b0;
    data_tx_flag <= 1'b0;
    control_tx_flag <= 1'b0;
    data_rx_node_id <= 16'h0000;
  end
  else
  begin
    // if the slot is equal to the node id then it means that this
    // node can now transmit on the control plane a packet
    // to another node saying that it wishes to transmit
    // data and wants to see if it is free
```

```
if(slot == node_id)
begin
  // provided the stack is not empty
  // and in this slot a transmission has not taken place
  // and provided the data plane is not tx to another
  // node
  if(sp_tx_current != 16'b0 && control_tx_flag == 1'b0 && data_tx_flag == 1'b0)
  begin
    // send a packet consisting of the dest_id
    // and src_id
    control_tx_packet <= {RAM_tx_data_out, node_id};
    control_tx_flag <= 1'b1;
  end
  // if it does not meet the above conditions then send
  // out 32'b0
  else
  begin
    control_tx_packet <= 32'b0;
  end
end
// if the slot is not equal to the node it then it means that
// this node cannot ping another node but other nodes
// may ping this node regarding data transfer
// the response to this should be on the control plane
else
begin
  // reset the control_tx_flag for later
  control_tx_flag <= 1'b0;
  // compare the 16 most significant bits to the node id
  // if equal then the packet is meant for this node
  // if the packet is not meant for this node then drop
  // it
  // provided the lower 16 bits are not 0x0000 and
  // 0xFFFF this means that this node is being pinged
  // for a data transfer
  if((control_rx_packet[31:16] == node_id) && (control_rx_packet[15:0] != all_zero) && (control_rx_packet[15:0] != all_one))
  begin
    // if the data plane is not recieving and the gpp does not want to rtr the data from RAM
    // the node can accept data on the data plane
    if((data_rx_flag == 1'b0) && (gpp_rtr_cp == 1'b0))
    begin
      // tell the data plane which node is
      // receving the data
      data_rx_node_id <= control_rx_packet[15:0];
      // set the flag which tells the data
      // plane that it can transmit
      data_rx_flag <= 1'b1;
```

```verilog
          // tell the node that asked that it
          // can transmit to this node on the
          // data plane
          control_tx_packet <= {control_rx_packet[15:0], all_one};
        end
        // if the data plane is already recieving data
        // or if the flag is set from the gpp
        // then
        // from another node then tell the
        // current node not now
        else // if(data_rx_flag == 1'b1 || gpp_rtr == 1'b1)
        begin
          control_tx_packet <= {control_rx_packet[15:0], all_zero};
        end
      end
      // if the packet was not meant for the node then drop
      // it and let the control_tx_packet equal to 0
      else
      begin
          control_tx_packet <= 32'b0;
      end
    end

    // this will reset the data_rx_flag if all the data has been received
    // the signal data_rx_complete_flag will go high from the data plane
    // indicating that the local flag for rx goes to zero
    // this is provided the rst is low
    if(data_rx_complete_flag == 1'b1)
    begin
      data_rx_flag <= 1'b0;
    end

    // this is the logic for ping response
    // provided reset is set to 0

    // provided the packet is meant for this node
    // and provided the packet has all zeros or all ones for the
    // lower 16 bits then this is packet is a ping response
    if(control_rx_packet[31:16] == node_id && (control_rx_packet[15:0] == all_zero || control_rx_packet[15:0] == all_one))
    begin
      // if the reponse is one then the requested node is
      // not busy and it can accept data on the data
      // plane
      if(control_rx_packet[15:0] == all_one)
      begin
        data_tx_flag <= 1'b1;
      end
    end

    // this will reset the data_tx_flag if all the data has been
    // transmitted
    // the signal data_tx_complete_flag will go high from the data plane
    // indicating that the local flag for tx goes to zero
    // this is provided the rst is low
    if(data_tx_complete_flag == 1'b1)
    begin
      data_tx_flag <= 1'b0;
    end

  end
end

// this logic is used to see if data can be transfered from the GPP to CP
always_comb
begin
  // check if data is being transmitted
  if(data_tx_flag == 1'b0)
  begin
    // logic to calcualte the number of slots depending on
    // the current slot, node id & max node in network
    if(node_id < slot)
    begin
      num_of_slots = max_node - slot + node_id - 1;
    end
    else if(node_id > slot)
    begin
      num_of_slots = node_id - slot - 1;
    end
    else // if(node_id == slot)
    begin
      num_of_slots = 0;
    end
  end
  // if not then set the flag to 0
  else
  begin
    gpp_trf_cp = 1'b0;
    num_of_slots = 0;
  end

  // if the number of slots is greater that or equal to
  // 2 then set the flag as there are enough
  // clock cycles
  if(num_of_slots >= 2)
  begin
    gpp_trf_cp = 1'b1;
  end
```

```
    else
    begin
      gpp_trf_cp = 1'b0;
    end
  end

endmodule
```

Listing B.1: Control_Plane

data_plane_tx.sv

```
/*
    ↪ ********************************************************************************
    ↪
* File name
        data_plane_tx.sv
* Description
        This module describes how the data plane tx works.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        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.

        gpp_tx_data - This is the data that will be transferred from the GPP to the DP
    ↪ transmitter RAM.

        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.

        data_tx_flag - This is a flag which indicates if the node is transmitting data on the data
    ↪ plane.
* Outputs
        data_tx_complete_flag - This flag will reset the data_tx_flag. The value for this is sent
    ↪ from the data plane transmitter.

        data_tx_packet - This is the packet that is transmitted on the data plane.

        RAM_tx_data_out - This input has the value equal to the top of the RAM of the data
    ↪ plane tx RAM.

        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.
* Author
```

```
        Sreethyan Aravinthan (UCL)
*****************************************************************************/
`include "ALU.sv"
`include "Data_Memory.sv"
`include "Multiplexer.sv"
`include "Register.sv"

module data_plane_tx
(
  input logic clk,
  input logic rst,
  input logic gpp_trf_dp,
  input logic [15:0] gpp_tx_data,
  input shortint node_id,
  input logic data_tx_flag,
  output logic data_tx_complete_flag,
  output logic [31:0] data_tx_packet,
  output logic [15:0] RAM_tx_data_out,
  output logic [15:0] sp_tx_current
);

  // this is used to see if the number of data packets sent is equal to
  // the fixed packet length
  logic [2:0] count;
  // this holds the current dest node that all the packets need to be
  // transmitted to
  logic [15:0] current_dest_node;
  // this is a flag that is used to transmit a special packet to the
  // destination node
  // this special packet consists of the destination node and the src
  // node id
  logic first_flag;
  // this is used to add to subtract from the stack pointer
  localparam [15:0] one = 16'h0001;
  // this will hold the data that is fed into the sp multiplexer
  // there are three values
  // same value as earlier
  // sp + 1
  // sp − 1
  logic [15:0] sp_tx_mux_data [3:0];
  // this will hold the next sp value that will be the input to the
  // register
  logic [15:0] sp_tx_next;
  // this contains the sp plus 1
  logic [15:0] sp_tx_current_plus_one;
  // this contains the sp minus 1
  logic [15:0] sp_tx_current_minus_one;
  // this will be the control signals of the sp multiplexer
  logic [1:0] sp_tx_mux_control;
```

```
// this will contain the RAM addresses that can be accessed and are
// the inputs to the multiplexer for the RAM address choice
logic [15:0] RAM_tx_address_mux_data [1:0];
// This is RAM address that is currently being accessed
logic [15:0] RAM_address;

// this combinational logic sets the values for all the multiplexer
// data
always_comb
begin
  // stack pointer multiplexer data being set
  sp_tx_mux_data[0] = sp_tx_current;
  sp_tx_mux_data[1] = sp_tx_current_plus_one;
  sp_tx_mux_data[2] = sp_tx_current_minus_one;

  // set the RAM address multiplexer data
  RAM_tx_address_mux_data[0] = sp_tx_current;
  RAM_tx_address_mux_data[1] = sp_tx_current_plus_one;
end

// wire up the control signal for the stack pointer multiplexer
assign sp_tx_mux_control = {data_tx_flag, gpp_trf_dp};
// 0 0 - same
// 0 1 - plus 1
// 1 0 - minus 1

// instantiate multiplexer to choose the next stack pointer
Multiplexer #(2, 16) sp_tx_mux(sp_tx_mux_control, sp_tx_mux_data, sp_tx_next);

// instantiate register for stack pointer tx
Register #(16) sp_tx_reg(clk, rst, sp_tx_next, sp_tx_current);

// ALU for stack pointer increment to point to the latest data that
// has been added
ALU #(16) sp_increment(sp_tx_current, one, 2'b00, sp_tx_current_plus_one);

// ALU for stack pointer decrement. this is needed if the data is
// retrived from the RAM
ALU #(16) sp_decrement(sp_tx_current, one, 2'b01, sp_tx_current_minus_one);

// multiplexer for choosing the most appropriate address
Multiplexer #(1, 16) RAM_tx_address_mux(gpp_trf_dp, RAM_tx_address_mux_data,
    ↪ RAM_address);

// instantiate RAM modules
Data_Memory #(16) RAM_tx(clk, gpp_trf_dp, RAM_address, gpp_tx_data, RAM_tx_data_out)
    ↪ ;

// need to set a variable and not depend on the data_tx_flag_out
// not good due to the clock cycles required to reset it as well


// get the destination node from the top of
// the node
always_comb
begin
  // provided the data plane is not transmitting get the value
  if(data_tx_flag == 1'b0)
  begin
    current_dest_node = RAM_tx_data_out;
  end
  else
  begin
    current_dest_node = current_dest_node;
  end
end

// logic for what packet should be transmitted on each clock edge on
// the data plane tx
// provided the reset state is 0
always_ff @(posedge clk)
begin
  // if the system is in the reset state
  // then transmit a packet with all zeros and set all variables and
  // flags to 0
  // provided the reset state is 1
  if(rst == 1'b1)
  begin
    data_tx_packet <= 32'h0000;
    count <= 3'b000;
    first_flag <= 1'b0;
    data_tx_complete_flag <= 1'b0;
  end
  else
  begin
    // if the tx flag has been set then it means the
    // system can now transmit the packet
    if(data_tx_flag == 1'b1)
    begin
      // this logic is used for the first packet
      // that is sent
      // it is special as it sends the current nodes
      // id in the packet which will be useful for
      // the rx processor
      if(first_flag == 1'b0)
      begin
        // set the flag so that the next time
        // the correct packet is sent
        first_flag <= 1'b1;
        // send this packet in this clock
        // cycle where it contains the dest
        // node id and the current node id
        data_tx_packet <= {current_dest_node, node_id};
      end
      // if the flag is set then the 1st packet is
      // sent
      else
      begin
        // send the packet containing the
        // follwoing format
        // destination node id and data at the
        // top of stack
        data_tx_packet <= {current_dest_node, RAM_tx_data_out};

        // if the count is equal to 3 then all
        // the packets have been sent
        // this is because in the
        // system design we have
        // decided to use fix packet
        // length
        if(count == 3)
        begin
          // rest all the flags
          first_flag <= 1'b0;
          count <= 0;
        end
        // if the count does not equal to
        // 3 then all the packets have
        // not been sent to the rx
        // node
        else
        begin
          count <= count + 1;
        end
        if(count == 2)
        begin
          // set this flag to reset the
          // flag at the control plane
          data_tx_complete_flag <= 1'b1;
        end
      end
    end
    // if the transmit signal is not given then
    else
    begin
      data_tx_packet <= 32'h0000;
    end

    // logic for sending out a pulse for one clock cycle after a packet
    // has been transmitted
    // provided the system is not in the reset state
    if(data_tx_complete_flag == 1'b1)
    begin
      // reset the flag
      data_tx_complete_flag <= 1'b0;
    end
  end
end

endmodule
```

Listing B.2: data_plane_tx

data_plane_rx.sv

```
/*
   ↪ ***********************************************************************************
   ↪
* File name
      data_plane_rx.sv
* Description
      This module describes how the data plane rx works.
* Parameters
      NONE
* Inputs
      clk − The clock for the system.

      rst − Signal to reset the system to the default values.

      data_rx_packet − This is the packet that is received on the data plane.

      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.

      gpp_rtr_dp − This is a control signal which indicates the data plane to modify the stack
   ↪ ponter. 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.
* Outputs
      data_rx_complete_flag − This flag will reset the data_rx_flag. The value for this is sent
   ↪ from the data plane reciever.

      RAM_rx_data_out − This shows the data that is outputed from the data plane receiver.
   ↪ It will be connected to the RAM of the GPP via a multiplexer.
* Author
      Sreethyan Aravinthan (UCL)
***********************************************************************************/
```

```

‘include ”ALU.sv”
‘include ”Data_Memory.sv”
‘include ”Multiplexer.sv”
‘include ”Register.sv”

module data_plane_rx
(
  input logic clk,
  input logic rst,
  input logic [31:0] data_rx_packet,
  input shortint node_id,
  input logic gpp_rtr_dp,
  output logic data_rx_complete_flag,
  output logic [15:0] RAM_rx_data_out
);

  // RAM will be part of the data plane
  // there will be two lots of RAMs
  // RAM1: for transmitting data
  // RAM2: for recieving data

  // variables
  // this is a control signal that allows data to be written to the RAM
  // if the received packet is for this node
  logic rx_enable;
  // this holds the next sp value. Also is the input for the register
  logic [15:0] sp_rx_next;
  // this is the current sp value and it is the output of the register
  logic [15:0] sp_rx_current;
  // this will hold current sp value − one
  logic [15:0] sp_rx_current_plus_one;
  // this will hold current sp value − one
  logic [15:0] sp_rx_current_minus_one;
  // this is used to add to subtract from the stack pointer
  localparam [15:0] one = 16’h0001;
  // this is the data that will be written into the RAM
  logic [15:0] data_rx_current;
  // this is the RAM address that will be accessed
  logic [15:0] RAM_address;
  // this will hold the data that is fed into the sp multiplexer
  // there are three values
  // same value as earlier
  // sp + 1
  // sp − 1
  logic [15:0] sp_rx_mux_data [3:0];
  // this will be the control signals of the sp multiplexer
  logic [1:0] sp_rx_mux_control;
  // this will contain the RAM addresses that can be accessed and are
  // the inputs to the multiplexer for the RAM address choice
  logic [15:0] RAM_rx_address_mux_data [1:0];
  // this is used to see if the number of data packets received is equal to
  // the fixed packet length
  logic [2:0] count = 3'b000;

  // wire up the control signal for the stack pointer multiplexer
  assign sp_rx_mux_control = {gpp_rtr_dp, rx_enable};

  // this combinational logic sets the values for all the multiplexer
  // data
  always_comb
  begin
    // stack pointer multiplexer data being set
    sp_rx_mux_data[0] = sp_rx_current;
    sp_rx_mux_data[1] = sp_rx_current_plus_one;
    sp_rx_mux_data[2] = sp_rx_current_minus_one;

    // set the RAM address multiplexer data
    RAM_rx_address_mux_data[0] = sp_rx_current;
    RAM_rx_address_mux_data[1] = sp_rx_current_plus_one;
  end

  // instantiate multiplexer to choose the next stack pointer
  Multiplexer #(2, 16) sp_rx_mux(sp_rx_mux_control, sp_rx_mux_data, sp_rx_next);

  // instantiate register for stack pointer rx
  Register #(16) sp_rx_reg(clk, rst, sp_rx_next, sp_rx_current);

  // ALU for stack pointer increment to point to the latest data that
  // has been added
  ALU #(16) sp_increment(sp_rx_current, one, 2'b00, sp_rx_current_plus_one);

  // ALU for stack pointer decrement. this is needed if the data is
  // retrived from the RAM
  ALU #(16) sp_decrement(sp_rx_current, one, 2'b01, sp_rx_current_minus_one);

  // multiplexer for choosing the most appropriate address
  Multiplexer #(1, 16) RAM_rx_address_mux(rx_enable, RAM_rx_address_mux_data,
      ↪ RAM_address);

  // instantiate RAM modules
  Data_Memory #(16) RAM_rx (clk, rx_enable, RAM_address, data_rx_current,
      ↪ RAM_rx_data_out);

  // instantiate register for stack pointer rx
  Register #(16) data_rx_reg(clk, rst, data_rx_packet[15:0], data_rx_current);

  // this logic deals with how to deal with a packet that is recieved
  // provided the reset state is 0
  always_ff @(posedge clk)
  begin
    // this logic will deal with what the default values should be
    // provided reset is set to 1
    if(rst == 1'b1)
    begin
      rx_enable <= 1'b0;
      count <= 3'b000;
      data_rx_complete_flag <= 1'b0;
    end
    else
    begin
      // if the destination id on the rx packet is equal to the node
      // id then the packet is for this node
      if(data_rx_packet[31:16] == node_id)
      begin
        // save in RAM and change stack pointer value
        // set the rx_enable flag to 1
        rx_enable <= 1'b1;
        // if count is equal to 4 - indicates the entire data
        // is sent from the source node
        if(count == 3'b100)
        begin
          // reset count
          count <= 3'b000;
          // set this flag to 1 thus allowing the the
          // rx_flag at the control plane to be reset
          data_rx_complete_flag <= 1'b1;
        end
        // if the count is not equal to 4 then the number of
        // packets recieved is not right
        else
        begin
          // increment count
          count <= count + 1;
        end
      end
      // if the dest id does not match the node id then set
      // rx_enable to 0 preventing a write into RAM
      else
      begin
        rx_enable <= 1'b0;
      end
      // this logic will deal with ensuring that the data_rx_complete_flag
      // is set to 1 for only 1 clock cycle if it is set
      // provided the reset state is 0
      if(data_rx_complete_flag == 1'b1)
      begin
```

```
        // reset the flag
        data_rx_complete_flag <= 1'b0;
      end
    end
  end

endmodule
```

Listing B.3: data_plane_rx

data_plane.sv

```
/* ********************************************************************************
* File name
        data_plane.sv
* Description
        This module instantiates the data plane tx & rx.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        data_rx_packet - This is the packet that is received on the data plane.

        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.

        gpp_rtr_dp - This is a control signal which indicates the data plane to modify the stack ponter. 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.

        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.

        gpp_tx_data - This is the data that will be transferred from the GPP to the DP transmitter RAM.

        data_tx_flag - This is a flag which indicates if the node is transmitting data on the data plane.
* Outputs
        data_rx_complete_flag - This flag will reset the data_rx_flag. The value for this is sent from the data plane reciever.

        RAM_rx_data_out - This shows the data that is outputed from the data plane receiver. It will be connected to the RAM of the GPP via a multiplexer.

        data_tx_complete_flag - This flag will reset the data_tx_flag. The value for this is sent
      ↪ from the data plane transmitter.

        data_tx_packet - This is the packet that is transmitted on the data plane.

        RAM_tx_data_out - This input has the value equal to the top of the RAM of the data
      ↪ plane tx RAM.

        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.
* Author
        Sreethyan Aravinthan (UCL)
*****************************************************************************/

module data_plane
(
  // signals for the data plane rx

  input logic clk,
  input logic rst,
  input logic [31:0] data_rx_packet,
  input shortint node_id,
  input logic gpp_rtr_dp,
  output logic data_rx_complete_flag, // ***
  output logic [15:0] RAM_rx_data_out,

  // signals for the data plane tx

  input logic gpp_trf_dp,
  input logic [15:0] gpp_tx_data,
  input logic data_tx_flag, // ***
  output logic data_tx_complete_flag, // ***
  output logic [31:0] data_tx_packet,
  output logic [15:0] RAM_tx_data_out,  // ***
  output logic [15:0] sp_tx_current // ***
);

  // data plane tx
  data_plane_tx dp_tx(clk,
          rst,
          gpp_trf_dp,
          gpp_tx_data,
          node_id,
          data_tx_flag,
          data_tx_complete_flag,
          data_tx_packet,
          RAM_tx_data_out,
          sp_tx_current);

  // data plane rx
  data_plane_rx dp_rx(clk,
          rst,
          data_rx_packet,
          node_id,
          gpp_rtr_dp,
          data_rx_complete_flag,
          RAM_rx_data_out);
endmodule
```

Listing B.4: data_plane

comms_processor.sv

```
/*
    ********************************************************************************
* File name
        comms_processor.sv
* Description
        This module instantiates the control plane and data plane. This will be connected to the
    GPP.
* Parameters
        NONE
* Inputs
        clk - The clock for the system.

        rst - Signal to reset the system to the default values.

        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.

        max_node - This shows the maximum number of nodes in the ring topology.

        control_rx_packet - This is the packet that will be recieved on the control plane.

        enable_rtr - This is a control signal that will allow the GPP to retrive data from the CP
    if it is possible.

        gpp_rtr_cp - This is a control signal that pauses rx from gpp so that gpp can retrive
    data from rx RAM.

        data_rx_packet - This is the packet that is received on the data plane.

        gpp_rtr_dp - This is a control signal which indicates the data plane to modify the stack
    ponter. 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.

```

```
        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.

        gpp_tx_data - This is the data that will be transferred from the GPP to the DP
      ↪ transmitter RAM.
* Outputs
        control_tx_packet - This is the packet that is transmitted on the control plane.

        data_rx_node_id - This is the value that is sent to a control unit which sets the
      ↪ wavelength/spatial channel of the reciever.

        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 retrive data from the data plane receiver.
      ↪ This will be connected to the ALU_src_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. This will be connected to the ALU_src_1.

        RAM_rx_data_out - This shows the data that is outputed from the data plane receiver.
      ↪ It will be connected to the RAM of the GPP via a multiplexer.

        data_tx_packet - This is the packet that is transmitted on the data plane.
* Author
        Sreethyan Aravinthan (UCL)
*******************************************************************************/

module comms_processor
(
  // commons signals
  input logic clk,
  input logic rst,
  input shortint node_id,
  // control signals
  input shortint max_node,
  input logic [31:0] control_rx_packet,
  input logic enable_rtr, // cu
  input logic gpp_rtr_cp, // cu
  output logic [31:0] control_tx_packet,
  output logic [15:0] data_rx_node_id,
  output logic data_rx_flag, // gpp
  output logic gpp_trf_cp, // gpp
  // data signals
  input logic [31:0] data_rx_packet,
  input logic gpp_rtr_dp, // cu
  output logic [15:0] RAM_rx_data_out, // gpp
  input logic gpp_trf_dp, // cu
  input logic [15:0] gpp_tx_data, // gpp
  output logic [31:0] data_tx_packet
);

  // connected signals
  logic data_rx_complete_flag;
  logic data_tx_complete_flag;
  logic [15:0] RAM_tx_data_out;
  logic [15:0] sp_tx_current;
  logic data_tx_flag;

  // instantiate control plane
  Control_Plane cp(clk,
       rst,
       node_id,
       max_node,
       data_rx_complete_flag, // ***
       control_rx_packet,
       data_tx_complete_flag, // ***
       RAM_tx_data_out, // ***
       sp_tx_current, // ***
       enable_rtr,
       gpp_rtr_cp,
       control_tx_packet,
       data_rx_node_id,
       data_tx_flag,  // ***
       data_rx_flag,
       gpp_trf_cp);

  // instantiate data plane
  data_plane dp(clk,
          rst,
          data_rx_packet,
          node_id,
          gpp_rtr_dp,
          data_rx_complete_flag,  // ***
          RAM_rx_data_out,
          gpp_trf_dp,
          gpp_tx_data,
          data_tx_flag, // ***
          data_tx_complete_flag,  // ***
          data_tx_packet,
          RAM_tx_data_out,  // ***
          sp_tx_current); // ***

endmodule
```

Listing B.5: comms_processor

Computer.sv

```
/*
```

```
↪ ******************************************************************************
↪
* File name
      Computer.sv
* Description
      This module instantiates all the modules needed and creates the microprocessor.
* Parameters
      NONE
* Inputs
      clk − The clock for the system.

      rst − Signal to reset the system to the default values.

      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.

      max_node − This shows the maximum number of nodes in the ring topology.

      control_rx_packet − This is the packet that will be recieved on the control plane.

      data_rx_packet − This is the packet that is received on the data plane.
* Outputs
      control_tx_packet − This is the packet that is transmitted on the control plane.

      data_rx_node_id − This is the value that is sent to a control unit which sets the
    ↪ wavelength/spatial channel of the reciever.

      data_tx_packet − This is the packet that is transmitted on the data plane.
* Author
      Sreethyan Aravinthan (UCL)
******************************************************************************/

module Computer
(
  input logic clk,
  input logic rst,
  input shortint node_id,
  // control plane signals
  input shortint max_node,
  input logic [31:0] control_rx_packet,
  output logic [31:0] control_tx_packet,
  output logic [15:0] data_rx_node_id,
  // data plane signals
  input logic [31:0] data_rx_packet,
  output logic [31:0] data_tx_packet
);

  // Instruction memory
```

```
logic [15:0]address_1;
logic [15:0]address_2;
logic [15:0]read_address_1;
logic [15:0]read_address_2;
// Data memory
logic [15:0]address_rw;
logic [15:0]data_in;
logic [15:0]data_out;
// Control Signals
logic memory_write_enable;
logic enable_rtr;
logic gpp_rtr_cp;
logic gpp_rtr_dp;
logic gpp_trf_dp;
// Communications Processor Signals
logic [15:0] RAM_rx_data_out; // gpp
logic data_rx_flag; // gpp
logic gpp_trf_cp; // gpp
logic [15:0] gpp_tx_data; // gpp

GPP cpu(clk,
        rst,
        address_1,
        address_2,
        read_address_1,
        read_address_2,
        address_rw,
        data_in,
        data_out,
    memory_write_enable,
    enable_rtr,
    gpp_rtr_cp,
    gpp_rtr_dp,
    gpp_trf_dp,
    RAM_rx_data_out, // gpp
    data_rx_flag, // gpp
    gpp_trf_cp, // gpp
    gpp_tx_data); // gpp

comms_processor cp  (clk,
          rst,
          node_id,
          max_node,
          control_rx_packet,
          enable_rtr, // cu
          gpp_rtr_cp, // cu
          control_tx_packet,
          data_rx_node_id,
          data_rx_flag, // gpp
          gpp_trf_cp, // gpp
          data_rx_packet,
          gpp_rtr_dp, // cu
          RAM_rx_data_out, // gpp
          gpp_trf_dp, // cu
          gpp_tx_data, // gpp
          data_tx_packet);

 Instruction_Memory #(16, 16, "test.mem") im_computer  (address_1,
                          address_2,
                          read_address_1,
                          read_address_2);

 Data_Memory #(16) dm_computer (clk,
                memory_write_enable,
                address_rw,
                data_in,
                data_out);


endmodule
```

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

Appendix C

Testbenches

gpp_tb.sv

```
module Computer_tb;

  logic clk;
  logic rst;

  Computer dut(clk, rst);

  initial
  begin
    clk = 0;
    forever #50ps clk = ~clk;
  end

  initial
  begin
    rst = 1'b1;
        #100ps;
    rst = 1'b0;
  end

endmodule
```

Listing C.1: Testbench for GPP

comms_processor_tx_tb.sv

```
module comms_processor_tb;

  // commons signals
  logic clk;
  logic rst;
  shortint node_id;
  // control signals
```

```
shortint max_node;
logic [31:0] control_rx_packet;
logic enable_rtr;
logic gpp_rtr_cp;
logic [31:0] control_tx_packet;
logic [15:0] data_rx_node_id;
logic data_rx_flag;
logic gpp_trf_cp; // ALU src 1
// data signals
logic [31:0] data_rx_packet;
logic gpp_rtr_dp;
logic [15:0] RAM_rx_data_out;
logic gpp_trf_dp;
logic [15:0] gpp_tx_data;
logic [31:0] data_tx_packet;

comms_processor dut (clk,
          rst,
          node_id,
          max_node,
          control_rx_packet,
          enable_rtr,
          gpp_rtr_cp,
          control_tx_packet,
          data_rx_node_id,
          data_rx_flag,
          gpp_trf_cp, // ALU src 1
          data_rx_packet,
          gpp_rtr_dp,
          RAM_rx_data_out,
          gpp_trf_dp,
          gpp_tx_data,
          data_tx_packet);

initial
begin
  clk = 0;
  forever #50ps clk = ~clk;
end

initial
begin
  // set constant values
  node_id = 1;
  max_node = 4;
  // set defualt values
  control_rx_packet = 32'h00000000;
  enable_rtr = 1'b0;
  gpp_rtr_cp = 1'b0;
```

```
    data_rx_packet = 32'h00000000;
    gpp_rtr_dp = 1'b0;
    gpp_trf_dp = 1'b0;
    gpp_tx_data = 16'h0000;
    rst = 1'b1;
    #100ps;
    rst = 1'b0;
    #100ps;

    // TX testing
    #100ps;
    #100ps;
    gpp_trf_dp = 1'b1;
    gpp_tx_data = 16'h000A;
    #100ps;
    gpp_tx_data = 16'h000B;
    #100ps;
    gpp_tx_data = 16'h000C;
    #100ps;
    gpp_tx_data = 16'h000D;
    #100ps;
    gpp_tx_data = 16'h0004;
    #100ps;
    gpp_trf_dp = 1'b0;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #50ps;
    control_rx_packet = 32'h0001FFFF;
    #100ps;
    control_rx_packet = 32'h00000000;
    #100ps;
  end

endmodule
```

Listing C.2: Testbench for communications processor transmitter

comms_processor_rx_tb.sv

```
module comms_processor_tb;

  // commons signals
  logic clk;
  logic rst;
  shortint node_id;
  // control signals
```

```
shortint max_node;
logic [31:0] control_rx_packet;
logic enable_rtr;
logic gpp_rtr_cp;
logic [31:0] control_tx_packet;
logic [15:0] data_rx_node_id;
logic data_rx_flag;
logic gpp_trf_cp; // ALU src 1
// data signals
logic [31:0] data_rx_packet;
logic gpp_rtr_dp;
logic [15:0] RAM_rx_data_out;
logic gpp_trf_dp;
logic [15:0] gpp_tx_data;
logic [31:0] data_tx_packet;

comms_processor dut (clk,
          rst,
          node_id,
          max_node,
          control_rx_packet,
          enable_rtr,
          gpp_rtr_cp,
          control_tx_packet,
          data_rx_node_id,
          data_rx_flag,
          gpp_trf_cp, // ALU src 1
          data_rx_packet,
          gpp_rtr_dp,
          RAM_rx_data_out,
          gpp_trf_dp,
          gpp_tx_data,
          data_tx_packet);

initial
begin
  clk = 0;
  forever #50ps clk = ~clk;
end

initial
begin
  // set constant values
  node_id = 1;
  max_node = 4;
  // set defualt values
  control_rx_packet = 32'h00000000;
  enable_rtr = 1'b0;
  gpp_rtr_cp = 1'b0;
```

```
    data_rx_packet = 32'h00000000;
    gpp_rtr_dp = 1'b0;
    gpp_trf_dp = 1'b0;
    gpp_tx_data = 16'h0000;
    rst = 1'b1;
    #100ps;
    rst = 1'b0;
    #100ps;

    // RX testing
    #100ps;
    #100ps;
    #50ps;
    control_rx_packet = 32'h00010004;
    #100ps;
    control_rx_packet = 32'h00000000;
    data_rx_packet = 32'h00010004;
    #100ps;
    data_rx_packet = 32'h0001000D;
    #100ps;
    data_rx_packet = 32'h0001000C;
    #100ps;
    data_rx_packet = 32'h0001000B;
    #100ps;
    data_rx_packet = 32'h0001000A;
    #100ps;
    data_rx_packet = 32'h00000000;
    enable_rtr = 1'b1;
    gpp_rtr_cp = 1'b1;
    #100ps;
    gpp_rtr_dp = 1'b1;
    control_rx_packet = 32'h00010004;
    #100ps;
  end

endmodule
```

Listing C.3: Testbench for communications processor reciever

Computer_tx_tb.sv

```
module Computer_tx_tb;

  logic clk;
  logic rst;
  shortint node_id;
  shortint max_node;
  logic [31:0] control_rx_packet;
  logic [31:0] control_tx_packet;
  logic [15:0] data_rx_node_id;
  logic [31:0] data_rx_packet;
  logic [31:0] data_tx_packet;

  Computer dut  (clk,
          rst,
          node_id,
          max_node,
          control_rx_packet,
          control_tx_packet,
          data_rx_node_id,
          data_rx_packet,
          data_tx_packet);

  initial
  begin
    clk = 0;
    forever #50ps clk = ~clk;
  end

  initial
  begin
    node_id = 1;
    max_node = 4;
    rst = 1'b1;
        #100ps;
    rst = 1'b0;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #100ps;
    #50ps;
    control_rx_packet = 32'h0001FFFF;
    #100ps;
    control_rx_packet = 32'h00000000;
    #100ps;
  end

```

```
endmodule
```

Listing C.4: Testbench for microprocessor transmitter

Computer_rx_tb.sv

```
module Computer_rx_tb;

  logic clk;
  logic rst;
  shortint node_id;
  shortint max_node;
  logic [31:0] control_rx_packet;
  logic [31:0] control_tx_packet;
  logic [15:0] data_rx_node_id;
  logic [31:0] data_rx_packet;
  logic [31:0] data_tx_packet;

  Computer dut  (clk,
          rst,
          node_id,
          max_node,
          control_rx_packet,
          control_tx_packet,
          data_rx_node_id,
          data_rx_packet,
          data_tx_packet);

  initial
  begin
    clk = 0;
    forever #50ps clk = ~clk;
  end

  initial
  begin
    node_id = 1;
    max_node = 4;
    rst = 1'b1;
        #100ps;
    rst = 1'b0;
    #300ps;
    control_rx_packet = 32'h00010004;
    #100ps;
    control_rx_packet = 32'h00000000;
    #100ps;
    data_rx_packet = 32'h00010004;
    #100ps;
    data_rx_packet = 32'h0001000A;
    #100ps;
    data_rx_packet = 32'h0001000B;
    #100ps;
    data_rx_packet = 32'h0001000C;
    #100ps;
    data_rx_packet = 32'h0001000D;
    #100ps;
    data_rx_packet = 32'h00000000;
    #100ps;
  end


endmodule
```

Listing C.5: Testbench for microprocessor reciever

Appendix D

Assembly programs used for testing

fib_normal.asm

```
  movi reg1, #0
  movi reg2, #1
loop:
  mov reg2, reg3
  add reg1, reg2  ; reg2 = reg1 + reg2
  mov reg3, reg1
  addi reg1, #0
  jmp loop
```

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

fib_stack.asm

```
  movi reg0, #0
  movi reg1, #1
  movi reg2, #1
loop:
  push reg2
  add reg1, reg2
  pop reg1
  addi reg2, #0
  jmp loop
```

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

division.asm

```
  movi reg0, #0
  movi reg1, #10
  movi reg2, #2
  call division
  jmp end
division:
  ; pass into reg1 the dividend
  ; pass into reg2 the divisor
  push reg3 ; use reg3 as a check therefore save value
  push reg4 ; reg4 will hold the quotient
  mov reg2, reg3  ; move the divisor
  movi reg4, #0 ; set the quotient to zero initially
divide:
  sub reg1, reg2  ; dividend_cur(reg2) = dividend_prev(reg1) - divisor(reg2)
  mov reg2, reg1  ; dividend_prev(reg1) = dividend_cur(reg2)
  mov reg3, reg2  ; dividend_cur(reg2) = reg3
  addi reg4, #1 ; increment quotient by 1
  cmp reg1, reg3  ; compare reg1 to reg3
  jae divide  ; if above or equal
  jmp divide_end  ; else quit the sub routine
divide_end:
  mov reg4, reg2  ; move the quotient into reg2
  ; reg1 contains the remainder
  pop reg4
  pop reg3  ; retrive old value of reg3
  return
end:
  addi reg2, #0
  addi reg1, #0
  jmp end
```

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

prime_number.asm

```
  movi reg0, #0 ; initalise the stack pointer
  movi reg5, #11  ; value to determine if prime
  call check_prime
  jmp end
check_prime:
  movi reg6, #2 ; this is the 1st number that will divide the to_be_prime_number
  movi reg7, #1 ; move 1
  sub reg5, reg7  ; this should hold the upper limit
  movi reg8, #0
continue:
  mov reg5, reg1  ; move dividend into reg1
  mov reg6, reg2  ; move divisor into reg2
  call division ; divide
  cmp reg1, reg8  ; check if remainder is 0
  jz not_prime  ; if remainder is 0 then not a prime so flag output is 0
  addi reg6, #1 ; increment for the next for the next divisor
  cmp reg6, reg5  ; check if the divisor is equal to the dividend
  jz end_search ; if the above result is 0 then end the search and flag it as zero
  jmp continue  ; else continue
not_prime:
  movi reg10, #0
  return
end_search:
  movi reg10, #1
  return
division:
  ; pass into reg1 the dividend
  ; pass into reg2 the divisor
  push reg3 ; use reg3 as a check therefore save value
  push reg4 ; reg4 will hold the quotient
  mov reg2, reg3  ; move the divisor
  movi reg4, #0 ; set the quotient to zero initially
divide:
  sub reg1, reg2  ; dividend_cur(reg2) = dividend_prev(reg1) − divisor(reg2)
  mov reg2, reg1  ; dividend_prev(reg1) = dividend_cur(reg2)
  mov reg3, reg2  ; dividend_cur(reg2) = reg3
  addi reg4, #1 ; increment quotient by 1
  cmp reg1, reg3  ; compare reg1 to reg3
  jae divide  ; if above or equal
  jmp divide_end  ; else quit the sub routine
divide_end:
  mov reg4, reg2  ; move the quotient into reg2
  ; reg1 contains the remainder
  pop reg4
  pop reg3  ; retrive old value of reg3
  return
end:
  addi reg10, #0
  jmp end
```

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

sample_tx.asm

```
  movi reg0, #0
  movi reg1, #10
  movi reg2, #11
  movi reg3, #12
  movi reg4, #13
  movi reg5, #4
  push reg5
  push reg4
  push reg3
  push reg2
  push reg1
  movi reg6, #0
  movi reg6, #0
  movi reg6, #0
```

```
 movi reg6, #0
 ; compare the value to see if data cane be transferred
 cbt
 ; if the result is not equal to zero then data cannot be transfered
 jz tranf
 jmp exit
 ; transfer 5 packet data
tranf:
 trf
 trf
 trf
 trf
 trf
exit:
 movi reg6, #0
```

Listing D.5: Assembly code for microprocessor transmitter

sample_rx.asm

```
 movi reg0, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 movi reg6, #0
 pr
 cbr
 jz exit
 rtr
 rtr
 rtr
 rtr
 rtr
exit:
   rr
```

Listing D.6: Assembly code for microprocessor reciever

Appendix E

Assembler code

main.c

```
/*
    ↪ ********************************************************************************
    ↪
* File name
        main.c
* Description
        This is the main program file for the assembler.
* Author
        Sreethyan Aravinthan (UCL)
********************************************************************************/


// standard header files
#include <stdio.h>
#include <string.h>

// personal header files
#include "assembly.h"
#include "preprocess.h"
#include "extra.h"

void combine_files(int argc, char * argv[], char out_file)
{
  // open the combined file in write mode
  FILE * combine = xfopen("combined.asm", "w");
  // this will be the pointer of the read file
  FILE * read_file;
  // this will hold the current file that is read from the current file
        char line[1000];
  // go through all the files and combine into a single new file
  for(int i = 1; i < argc - out_file; i++)
  {
    // open the file
    read_file = xfopen(argv[i], "r");
    // read until EOF
    while(fgets(line, 1000, read_file) != NULL)
    {
      // write to new combined file
      fprintf(combine, "%s", line);
    }
    // close current file and then go to the next file if it exists
    fclose(read_file);
  }
  // close the combined file
  fclose(combine);
}

int main(int argc, char * argv[])
{
  char * file_name;
  char out_file = 1;

  // check if three arguments are supplied
  if(argc < 2)
  {
    printf("Too few arguements\nNeed atleast two arguements\n");
    return 0;
  }
  else  // one file assembly
  {
    // check to see if the output file is supplied
    if(strstr(argv[argc - 1], ".mem") == NULL)
    {
      printf("No output file given\nSetting output file as out.mem\n");
      out_file = 0;
    }
    // combine all input files
    combine_files(argc, argv, out_file);
  }
  // remove the comments - creates a file called no_comments.asm
  remove_comments("combined.asm");
  // remove and replace with addresses
  remove_label();
  if(out_file == 1)
  {
    // convert from mnemonics to machine code
    convert_to_machine_code(argv[argc - 1]);
  }
  else
  {
    // convert from mnemonics to machine code
    convert_to_machine_code("out.mem");
  }
  // remove temporary files
  remove("combined.asm");
  remove("no_comments.asm");
  remove("no_labels.asm");
  return 0;
}
```

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

preprocess.h

```
/* ******************************************************************************
* File name
        preprocess.h
* Description
        Header file preprocess.h
* Author
        Sreethyan Aravinthan (UCL)
******************************************************************************/

/*
 * Design Unit:
 * File name:
 * Description:
 * Author:
 * Version:
*/

#ifndef _PREPROCESS_H_
#define _PREPROCESS_H_

#define NUM_OF_LABELS      100
#define NUM_OF_ADDRESSES   100
#define LENGTH_OF_LABEL    100
#define LENGTH_OF_INSTR    30
#define SPACE        32
#define TAB        9
#define NEW_LINE     10
#define UPPER_A      65
#define UPPER_Z      90
#define LOWER_A      97
#define LOWER_Z      122
#define UNDERSCORE     95
#define ZERO       48
#define NINE       57
```

```

typedef struct
{
  // multi−array which holds all the labels in the assembly file
  char labels[NUM_OF_LABELS][LENGTH_OF_LABEL];

  // array which holds the address of each label
  unsigned int label_address[NUM_OF_ADDRESSES];

  // this indicates the number of labels/addresses that were filled
  int index;
}labels_and_addresses_file;

// functions
void remove_comments(char ∗ file_name);
void remove_label(void);

#endif
```

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

preprocess.c

```
/∗
    ↪ ****************************************************************************
    ↪
∗ File name
        preprocess.c
∗ Description
        This file has functions which remove comments and labels were.
∗ Author
        Sreethyan Aravinthan (UCL)
****************************************************************************/

// standard header files
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// personal header files
#include "preprocess.h"
#include "extra.h"

// remove the comments from the code and create a new file
// lets call the file no_comments.asm
void remove_comments(char ∗ file_name)
{
  FILE ∗ original_code = xfopen(file_name, "r");
  FILE ∗ no_comment_code = xfopen("no_comments.asm", "w");
  char line[100]; // stores current line read
  char * target = ";";
  char * occur = NULL;

  while(fgets(line, 100, original_code) != NULL)
  {
    occur = strstr(line, target);

    // check to see if the pointer is not NULL indicating that this line has a ;
    if(occur != NULL)
    {
      // remove the ;
      // add a \n and \0
      *occur = '\n';
      *(occur + 1) = '\0';

      // provided the format is not a tab then new line then save line in file
      if(strcmp(line, " \n") != 0 && strcmp(line, "\n") != 0)
      {
        fprintf(no_comment_code, "%s", line);
      }
    }
    // if the pointer is null then the line has no comment therefore write it in
    else
    {
      fprintf(no_comment_code, "%s", line);
    }
  }

  // close all the files
  fclose(original_code);
  fclose(no_comment_code);
}

void remove_label(void)
{
  FILE * no_comment_code = xfopen("no_comments.asm", "r");
  FILE * no_labels_code = xfopen("no_labels.asm", "w");
  char line[LENGTH_OF_INSTR]; // hold the current line that was read from the file

  // hold the instructions that will take two 16 bit values in instruction memory in a multi-dimmesional array
  char two_space_instr[16][5];
  // copythe values in
  strcpy(two_space_instr[0], "jz");
  strcpy(two_space_instr[1], "jmp");
  strcpy(two_space_instr[2], "movi");
  strcpy(two_space_instr[3], "addi");
  strcpy(two_space_instr[4], "subi");
  strcpy(two_space_instr[5], "andi");
  strcpy(two_space_instr[6], "ori");
  strcpy(two_space_instr[7], "call");
  strcpy(two_space_instr[8], "jb");
  strcpy(two_space_instr[9], "jbe");
  strcpy(two_space_instr[10], "ja");
  strcpy(two_space_instr[11], "jae");
  strcpy(two_space_instr[12], "jg");
  strcpy(two_space_instr[13], "jge");
  strcpy(two_space_instr[14], "jl");
  strcpy(two_space_instr[15], "jle");

  // this will hold all the labels and their addresses
  labels_and_addresses_file file_data;
  // initialise the index field to 0
  file_data.index = 0;

  char * colon = ":";
  char * occur = NULL;
  // this holds the address of the current instruction
  unsigned short int address = 0;

  // find the labels and the corresponding address
  while(fgets(line, LENGTH_OF_INSTR, no_comment_code) != NULL)
  {
    // check if the line has a colon
    occur = strstr(line, colon);

    // if the line does have a colon get the label
    if(occur != NULL)
    {
      // place null terminator at the same place as colon
      *occur = '\0';
      // copy the label into the array
      strcpy(file_data.labels[file_data.index], line);
      // save the address of this label as well
      file_data.label_address[file_data.index] = address;
      // increment index for next label
      file_data.index++;
    }
    // else calculate the next address
    else
    {
      // flag to indicate if the address needs to be jumped by 2
      char plus_2 = 0;

      // check to see if the current instruction has an instruction which takes up 2 16 bit fields
      for(int i = 0; i < 16; i++)
      {
        if(strstr(line, two_space_instr[i]) != NULL)
```

```
        {
          plus_2 = 1;
          break;
        }
      }

      // if yes add 2 or else add 1
      if(plus_2 == 1)
      {
        address += 2;
      }
      else
      {
        address++;
      }
    }
  }

  // go back to the begining of the file
  rewind(no_comment_code);

  // get the line
  // see if it has a :
  // if yes do nothing
  // if no (NULL) then check if it has a ,
  // if yes then print line to file
  // if no then the instruction can be jump, call, one operand(pop) or zero opernad
      ↪ instructions(cbt)
  // get the instruction and compare it to the list of jump and call
  // if no print the line to file
  // if yes find the address of the label and replace it and write to file

  char * comma = ",";
  char has_label = 0;

  char label_instr[11][5];
  strcpy(label_instr[0], "jz");
  strcpy(label_instr[1], "jmp");
  strcpy(label_instr[2], "call");
  strcpy(label_instr[3], "jb");
  strcpy(label_instr[4], "jbe");
  strcpy(label_instr[5], "ja");
  strcpy(label_instr[6], "jae");
  strcpy(label_instr[7], "jg");
  strcpy(label_instr[8], "jge");
  strcpy(label_instr[9], "jl");
  strcpy(label_instr[10], "jle");

  char ** words_in_line;
```

```
int num_of_words = 0;
int num_of_frees = 0;
char * shifted_line = NULL;

// remove all the labels and replace it with the address
// also remove all the labels that has a colon next to iti
while(fgets(line, LENGTH_OF_INSTR, no_comment_code) != NULL)
{
  shifted_line = shift_left_one(line);

  occur = strstr(shifted_line, colon);

  // this line is not a label
  if(occur == NULL)
  {
    // find out if a comma exists in the line
    occur = strstr(shifted_line, comma);

    // this means that this is a two operand instruction and they do not have labels. Simply
    ↪ print the line into the file
    if(occur != NULL)
    {
      fprintf(no_labels_code, "%s", shifted_line);
    }
    // the instruction can be jump, call, one operand(pop) or zero opernad instructions(cbt)
    else
    {
      words_in_line = get_words_from_string(shifted_line, &num_of_words, &num_of_frees);
      // go through the instructions and see if it has 2 space instructions
      for(int i = 0; i < 16; i++)
      {
        if(strcmp(words_in_line[0], label_instr[i]) == 0)
        {
          has_label = 1;
          break;
        }
      }

      // free the allocated memory
      free_split(words_in_line, num_of_frees);

      // this line of code does not have a label as an operand
      if(has_label == 0)
      {
        fprintf(no_labels_code, "%s", shifted_line);
      }
      // this line of code does have a label as an operand
      else if(has_label == 1)
      {
```

```c
          // get the words from the line of code
          words_in_line = get_words_from_string(line, &num_of_words, &num_of_frees);
          fprintf(no_labels_code,"%s ", words_in_line[0]);
          // now go through all the labels and find the address and replace it with its address
          for(int j = 0; j < file_data.index; j++)
          {
            if(strcmp(file_data.labels[j], words_in_line[1]) == 0)
            {
              fprintf(no_labels_code,"%d\n", file_data.label_address[j]);
              break;
            }
          }
          // free the allocated memory
          free_split(words_in_line, num_of_frees);
        }
        has_label = 0;
      }
    }
  }

  fclose(no_comment_code);
  fclose(no_labels_code);
}
```

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

assembly.h

```c
/* *******************************************************************************
* File name
        assembly.h
* Description
        Header file assembly.h
* Author
        Sreethyan Aravinthan (UCL)
*******************************************************************************/

#ifndef _ASSEMBLY_H_
#define _ASSEMBLY_H_

#define NUM_OF_INSTR     33
#define MNEMONIC_LENGTH    10
#define OPCODE_LENGTH    6
#define NUM_OF_REG       32
#define REG_LENGTH       6

typedef struct
{
  char mnemonic[NUM_OF_INSTR][MNEMONIC_LENGTH];
  char * opcode[NUM_OF_INSTR];
}instructions;

typedef struct
{
  char reg[NUM_OF_REG][REG_LENGTH];
  char * operand[NUM_OF_REG];
}registers;

// functions
void convert_to_machine_code(char * output_file);
void get_list_of_instr(instructions * list_of_instr);
void get_list_of_regs(registers * list_of_regs);
void free_all_opcodes_and_operands(instructions * list_of_instr, registers * list_of_regs);
char * reg_operand_get(char * reg, registers * list_of_regs);
char * two_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions * list_of_instr);
void label_operands(char * code, FILE * machine_code, instructions * list_of_instr);
char * immediate_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions * list_of_instr);
char * stack_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions *  list_of_instr);
void return_instr();
void comms_instr(FILE * machine_code, char * instr, instructions * list_of_instr);
void clean_up(FILE * no_labels_code, FILE * machine_code, instructions * list_of_instr, registers * list_of_regs);

#endif
```

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

assembly.c

```c
/* ***************************************************************************************
* File name
        assembly.c
* Description
        This file converts the preprocessed file to machine code
* Author
        Sreethyan Aravinthan (UCL)
*************************************************************************************/

// standard header files
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

// personal header files
#include "assembly.h"
#include "extra.h"

void get_list_of_instr(instructions * list_of_instr)
{
  // open file which contains all the instructions that can run on
  // the processor
  FILE * instructions_file = xfopen("/mnt/d/Documents/University/Year_3/Project/Solution/
      ↪ Assembler/Software/src/instructions.txt", "r");

  // read from the file and store it in the array field called mnemonic
  for(unsigned short int index = 0; index < NUM_OF_INSTR; index++)
  {
    fscanf(instructions_file, "%s", list_of_instr->mnemonic[index]);
  }

  // close the file
  fclose(instructions_file);
}

void get_list_of_regs(registers * list_of_regs)
{
  // open file which contains all the registers that are part of the processor
  FILE * registers_file = xfopen("/mnt/d/Documents/University/Year_3/Project/Solution/
      ↪ Assembler/Software/src/registers.txt", "r");

  // read from the file and store it in the array field called reg
  for(unsigned int index = 0; index < NUM_OF_REG; index++)
  {
    fscanf(registers_file, "%s\n", list_of_regs->reg[index]);
  }

  // close the file
  fclose(registers_file);
}

void free_all_opcodes_and_operands(instructions * list_of_instr, registers * list_of_regs)
{
  // free the opcodes
  for(int i = 0; i < NUM_OF_INSTR; i++)
  {
    free(list_of_instr->opcode[i]);
    list_of_instr->opcode[i] = NULL;
  }

  // free the operands
  for(int i = 0; i < NUM_OF_REG; i++)
  {
    free(list_of_regs->operand[i]);
    list_of_regs->operand[i] = NULL;
  }
}

void convert_to_machine_code(char * output_file)
{
  // open the file called no_labels in read mode
  FILE * no_labels_code = xfopen("no_labels.asm", "r");

  // open the file called machine code in write mode which contains the machine code
  // that will run on the processor
  FILE * machine_code = xfopen(output_file, "w");

  // create memory which holds the mnemonics and the coresponding opcodes
  instructions * list_of_instr = (instructions *) xmalloc(sizeof(instructions));
  // load mnemonics that are part of the processor
  get_list_of_instr(list_of_instr);

  // convert each opcode from decimal to binary and store in the opcode field
  for(unsigned short int index = 0; index < NUM_OF_INSTR; index++)
  {
    list_of_instr->opcode[index] = convert_to_binary(index, 0, 6);
  }

  // create memory which golds the registers and the coresponding operands in binary format
  registers * list_of_regs = (registers *)xmalloc(sizeof(registers));
  // load the registers and get the operands for each registers
  get_list_of_regs(list_of_regs);

  // convert each operand from decimal to binary and store in the operand field
  for(unsigned short int index = 0; index < NUM_OF_REG; index++)
  {
    list_of_regs->operand[index] = convert_to_binary(index, 0, 5);
  }

  char line[100];
  int num_of_words = 0;
  int num_of_frees = 0;
  char ** words_in_line;
  int instr = 0;

  // get the current line
  // get the words from it
  // get the instruction and find out which one it is
  // use that index for a switch case
  // call the function that will convert to machine code

```

```
while(fgets(line, 100, no_labels_code) != NULL)
{
  words_in_line = get_words_from_string(line, &num_of_words, &num_of_frees);

  for(instr = 0; instr < NUM_OF_INSTR; instr++)
  {
    if(strcmp(list_of_instr->mnemonic[instr], words_in_line[0]) == 0)
    {
      break;
    }
  }

  // if the mnemonic is not found then print an error message and exit
  if(instr == NUM_OF_INSTR)
  {
    char * target = "\n";
    char * occur = NULL;

    occur = strstr(line, target);
    *occur = '\0';

    // print the error message
    fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid mnemonic %s\n", line,
    ↪ words_in_line[0]);
    // free the words that were taken from the line
    free_split(words_in_line, num_of_frees);
    clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
    // exit the program
    exit(-1);
  }

  switch(instr)
  {
    case 0:   // ld
    case 1:   // str
    case 2:   // add
    case 3:   // sub
    case 4:   // and
    case 5:   // or
    case 6:   // mov
    case 7:   // cmp
    {
      char * error_operand = two_operands(line, machine_code, list_of_regs, list_of_instr);
      if(error_operand != NULL)
      {
        char * target = "\n";
        char * occur = NULL;

        occur = strstr(line, target);
```

```
            *occur = '\0';

            // print the error message
            fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid operand %s\n", line,
        ↪  error_operand);
            // free the memory
            free(error_operand);
            free_split(words_in_line, num_of_frees);
            clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
            // exit the program
            exit(-1);
          }
        }
          break;
        case 8:   // jz
        case 9:   // jmp
        case 17:  // call
        case 19:  // jb
        case 20:  // jbe
        case 21:  // ja
        case 22:  // jae
        case 23:  // jg
        case 24:  // jge
        case 25:  // jl
        case 26:  // jle
          label_operands(line, machine_code, list_of_instr);
          break;
        case 10:  // movi
        case 11:  // addi
        case 12:  // subi
        case 13:  // andi
        case 14:  // ori
        {
          char * error_operand = immediate_operands(line, machine_code, list_of_regs, list_of_instr);
          if(error_operand != NULL)
          {
            char * target = "\n";
            char * occur = NULL;

            occur = strstr(line, target);
            *occur = '\0';

            // print the error message
            fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid operand %s\n", line,
        ↪  error_operand);
            // free the memory
            free(error_operand);
            free_split(words_in_line, num_of_frees);
            clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
            // exit the program
            exit(−1);
          }
        }
          break;
        case 15:  // push
        case 16:  // pop
        {
          char ∗ error_operand = stack_operands(line, machine_code, list_of_regs, list_of_instr);
          if(error_operand != NULL)
          {
            char ∗ target = "\n";
            char ∗ occur = NULL;

            occur = strstr(line, target);
            ∗occur = '\0';

            // print the error message
            fprintf(stderr, "Instruction \"%s\" is not valid as it has an invalid operand %s\n", line,
        ↪   error_operand);
            // free the memory
            free(error_operand);
            free_split(words_in_line, num_of_frees);
            clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
            // exit the program
            exit(−1);
          }
        }
          break;
        case 18:  // return
          return_instr(machine_code);
          break;
        case 27:  // cbt
        case 28:  // trf
        case 29:  // pr
        case 30:  // cbr
        case 31:  // rtr
        case 32:  // rr
          comms_instr(machine_code, words_in_line[0], list_of_instr);
          break;
        default:
          break;
      }
      // free memory used for this line of code
      free_split(words_in_line, num_of_frees);
    }

    clean_up(no_labels_code, machine_code, list_of_instr, list_of_regs);
  }
```

```

char * reg_operand_get(char * reg, registers * list_of_regs)
{
  // hold the address where the binary format of the register is stored
  char * operand;
  // use this variable to index through the array of registers
  int index = 0;
  // find the register
  for(index = 0; index < NUM_OF_REG; index++)
  {
    if(strcmp(reg, list_of_regs->reg[index]) == 0)
    {
      operand = list_of_regs->operand[index];
      break;
    }
  }
  // if index is NUM_OF_REG then the register does not exist
  // therefore return NULL to indicate the register does not exist
  if(index == NUM_OF_REG)
  {
    return NULL;
  }
  // return the address of the binary format of the register
  return operand;
}

char * two_operands(char * code, FILE * machine_code, registers * list_of_regs, instructions *
     ↪ list_of_instr)
{
  // this will be a double pointer that holds the address of all the strings in an instruction
  char ** indiv;
  // this will hold the number of strings in the instruction
  int num_of_data = 0;
  // this will hold the number of frees
  int num_of_frees = 0;
  // get all the strings in the instruction
  indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
  // get the length of the 1st operand
  int length = strlen(indiv[1]);
  // add a null terminator one less than length of the string so that the comma is removed
  indiv[1][length - 1] = '\0';

  // print opcode
  for(int i = 0; i < NUM_OF_INSTR; i++)
  {
    if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
    {
      fprintf(machine_code, "%s", list_of_instr->opcode[i]);
      break;
    }
  }

  // get the 1st operand
  char * operand_1 = reg_operand_get(indiv[1], list_of_regs);
  // get the 2nd operand
  char * operand_2 = reg_operand_get(indiv[2], list_of_regs);

  // if the 1st operand is NULL then the register does not exisit
  if(operand_1 == NULL)
  {
    char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[1]));
    strcpy(error_operand, indiv[1]);
    free_split(indiv, num_of_frees);
    return error_operand;
  }
  // if the 2nd operand is NULL then the register does not exisit
  else if(operand_2 == NULL)
  {
    char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[2]));
    strcpy(error_operand, indiv[2]);
    free_split(indiv, num_of_frees);
    return error_operand;
  }

  // print the 1st and 2nd operand and a newline
  fprintf(machine_code, "%s", operand_1);
  fprintf(machine_code, "%s", operand_2);
  fprintf(machine_code, "\n");
  // free the strings and the double pointer
  free_split(indiv, num_of_frees);
  // return NULL to indicate that it is successul
  return NULL;
}

void label_operands(char * code, FILE * machine_code, instructions * list_of_instr)
{
  char ** indiv;
  int num_of_data = 0;
  int num_of_frees = 0;
  indiv = get_words_from_string(code, &num_of_data, &num_of_frees);

  // print opcode
  for(int i = 0; i < NUM_OF_INSTR; i++)
  {
    if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
    {
      fprintf(machine_code, "%s", list_of_instr->opcode[i]);
      break;
    }
  }
  fprintf(machine_code, "0000000000");
  fprintf(machine_code, "\n");

  int label = atoi(indiv[1]);
  char * label_bin = convert_to_binary(label, 0, 16);
  fprintf(machine_code, "%s\n", label_bin);

  free(label_bin);
  free_split(indiv, num_of_frees);
}

char * immediate_operands(char * code, FILE * machine_code, registers * list_of_regs,
    ↪ instructions * list_of_instr)
{
  // this will be a double pointer that holds the address of all the strings in an instruction
  char ** indiv;
  // this will hold the number of strings in the instruction
  int num_of_data = 0;
  // this will hold the number of frees
  int num_of_frees = 0;
  // get all the strings in the instruction
  indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
  // get the length of the 1st operand
  int length = strlen(indiv[1]);
  // add a null terminator one less than length of the string so that the comma is removed
  indiv[1][length - 1] = '\0';

  // print opcode
  for(int i = 0; i < NUM_OF_INSTR; i++)
  {
    if(strcmp(indiv[0], list_of_instr->mnemonic[i]) == 0)
    {
      fprintf(machine_code, "%s", list_of_instr->opcode[i]);
      break;
    }
  }

  char * reg_operand = reg_operand_get(indiv[1], list_of_regs);
  // if the reg operand is NULL then the register does not exisit
  if(reg_operand == NULL)
  {
    char * error_operand = (char *)xmalloc(sizeof(char) * strlen(indiv[1]));
    strcpy(error_operand, indiv[1]);
    free_split(indiv, num_of_frees);
    return error_operand;
  }
  // print the reg_operand
```

```c
fprintf(machine_code, "%s%s", reg_operand, reg_operand);
fprintf(machine_code, "\n");

// check if the immediate has a '-'
int i = 0;
for(i = 0; i < strlen(indiv[2]); i++)
{
  if(indiv[2][i] == '-')
  {
    break;
  }
}

// this will hold the address of the immediate value that will be extracted from the 2nd operand
char * immediate;

// if i is 1 then it means that the immediate is a negative value
if(i == 1)
{
  // increment by 1 so that we skip the '-'
  i++;
  // create memory on the heap
  immediate = (char *)xmalloc(sizeof(char) * (strlen(indiv[2]) - 2));
  int j = 0;
  // copy data
  for(j = 0; j < strlen(indiv[2]) - 2; j++, i++)
  {
    immediate[j] = indiv[2][i];
  }
  // place null terminator
  immediate[j] = '\0';
  // convert from string to integer
  int immediate_int = atoi(immediate);
  // convert the integer to binary format
  char * immediate_bin = convert_to_binary(immediate_int, 1, 16);
  // print the binary format of the immediate to the file
  fprintf(machine_code, "%s\n", immediate_bin);
  // free the immediate binary format
  free(immediate_bin);
}
else
{
  // set the i to 1 so that the data is copied after the # character
  i = 1;
  // create memory on the heap
  immediate = (char *)xmalloc(sizeof(char) * (strlen(indiv[2]) - 1));
  // copy the immediate
  int j = 0;
```

```
    for(j = 0; j < strlen(indiv[2]) − 1; j++, i++)
    {
      immediate[j] = indiv[2][i];
    }
    // place null terminator
    immediate[j] = '\0';
    // convert from string to integer
    int immediate_int = atoi(immediate);
    // convert from integer to binary format
    char ∗ immediate_bin = convert_to_binary(immediate_int, 0, 16);
    // print the binary format of the immediate to the file
    fprintf(machine_code, "%s\n", immediate_bin);
    // free the immediate binary format
    free(immediate_bin);
  }

  // free the immediate value that was copied
  free(immediate);
  // free the each opcode and operand from the current instruction
  free_split(indiv, num_of_frees);

  return NULL;
}

char ∗ stack_operands(char ∗ code, FILE ∗ machine_code, registers ∗ list_of_regs, instructions ∗
      ↪ list_of_instr)
{
  char ∗∗ indiv;
  int num_of_data = 0;
  int num_of_frees = 0;
  indiv = get_words_from_string(code, &num_of_data, &num_of_frees);
  int length = strlen(indiv[1]);
  indiv[1][length] = '\0';

  // print opcode
  for(int i = 0; i < NUM_OF_INSTR; i++)
  {
    if(strcmp(indiv[0], list_of_instr−>mnemonic[i]) == 0)
    {
      fprintf(machine_code, "%s", list_of_instr−>opcode[i]);
      break;
    }
  }

  char ∗ reg_operand = reg_operand_get(indiv[1], list_of_regs);
  // if the reg operand is NULL then the register does not exisit
  if(reg_operand == NULL)
  {
    char ∗ error_operand = (char ∗)xmalloc(sizeof(char) ∗ strlen(indiv[1]));
    strcpy(error_operand, indiv[1]);
    free_split(indiv, num_of_frees);
    return error_operand;
  }

  fprintf(machine_code, "00000");
  fprintf(machine_code, "%s", reg_operand);
  fprintf(machine_code, "\n");
  free_split(indiv, num_of_frees);

  return NULL;
}

void return_instr(FILE * machine_code)
{
  fprintf(machine_code, "0100100000000000\n");
}

void comms_instr(FILE * machine_code, char * instr, instructions * list_of_instr)
{
  // print opcode
  for(int i = 0; i < NUM_OF_INSTR; i++)
  {
    if(strcmp(instr, list_of_instr->mnemonic[i]) == 0)
    {
      fprintf(machine_code, "%s", list_of_instr->opcode[i]);
      break;
    }
  }

  fprintf(machine_code, "0000000000\n");
}

void clean_up(FILE * no_labels_code, FILE * machine_code, instructions * list_of_instr,
    ↪ registers * list_of_regs)
{
  fclose(no_labels_code);
  fclose(machine_code);
  free_all_opcodes_and_operands(list_of_instr, list_of_regs);
  free(list_of_instr);
  free(list_of_regs);
}
```

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

extra.h

```
/*
    ↪ ******************************************************************************
```

```
      ↪
* File name
        extra.h
* Description
        Header file extra.h
* Author
        Sreethyan Aravinthan (UCL)
*******************************************************************************/

#ifndef _EXTRA_H_
#define _EXTRA_H_

#define WIDTH   16

// functions
FILE * xfopen(char * file_name, char * mode);
void * xmalloc(size_t size);
int positive_binary(unsigned short int immediate, char * binary_form, unsigned int index);
void twos_complement(char * binary_form, unsigned char width1);
char * convert_to_binary(unsigned short int immediate, char negative, unsigned char width);
char **get_words_from_string(const char *input_string, int *num_of_words, int *num_of_frees);
void free_split(char ** words, int num_of_frees);
char * shift_left_one(char * data);

#endif
```

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

extra.c

```
/*
      ↪ *******************************************************************************
      ↪
*       File name
      extra.c
* Description
      This file has extra tools that are used for converting to machine code
* Author
      Sreethyan Aravinthan (UCL)
*******************************************************************************/

// standard header files
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// personal header files
#include "extra.h"

```

```
FILE * xfopen(char * file_name, char * mode)
{
  // open the file
  FILE * file = fopen(file_name, mode);
  // if the file is NULL then exit gracefully
  if(file == NULL)
  {
    fprintf(stderr, "File opening error");
    exit(-1);
  }
  return file;
}

void * xmalloc(size_t size)
{
    void * data = malloc(size);

    // if the requested data is not given in other words returned NULL exit
    if (data == NULL)
    {
        fprintf(stderr, "virtual memory exhausted");
        exit(-1);
    }

    // return the data variable
    return data;
}

void * xrealloc(void *ptr, size_t size)
{
    void * data = realloc(ptr, size);

    // if the requested data is not given in other words returned NULL exit
    if (data == NULL)
    {
        fprintf(stderr, "virtual memory exhausted");
        exit(-1);
    }

    // return the data variable
    return data;
}


int positive_binary(unsigned short int immediate, char * binary_form, unsigned int index)
{
  // check if the value is greater than 1
        if(immediate > 1)
        {
            index = positive_binary(immediate/2, binary_form, index);
        }

    // if the moduls is 1 then set that index value to 1
    // else 0
        if(immediate % 2 == 1)
        {
            binary_form[index] = '1';
        }
        else
        {
            binary_form[index] = '0';
        }
    // increment index for the next binary number
        index++;

    // return index for the next binary number which was called in a recursive manner
        return index;
}

void twos_complement(char * binary_form, unsigned char width)
{
        // convert to 1s complement
        for(int i = 0; i < width; i++)
        {
            if(binary_form[i] == '0')
            {
                binary_form[i] = '1';
            }
            else
            {
                binary_form[i] = '0';
            }
        }

        // convert to 2s complement from 1s complement
        for(int i = width - 1; i >= 0; i--)
        {
            if(binary_form[i] == '1')
            {
                binary_form[i] = '0';
            }
            else
            {
                binary_form[i] = '1';
                break;
            }
        }
}
```

```c

char * convert_to_binary(unsigned short int immediate, char negative, unsigned char width)
{
  // this will contain the positive binary form of a number
        char * pos_binary = (char *)malloc(sizeof(char) * width);
  // this variable will hold the pointer to which the correct number of bits of the binary number should be
        char * pos_binary_form = (char *)malloc(sizeof(char) * width);
        // set the length to 0 which hold the length of the initial postive version of the binary number
  int length = 0;
        // get the positive version of the number passed in
        length = positive_binary(immediate, pos_binary, 0);
        // add null terminator
        length++;
        pos_binary[length] = '\0';
        length--;

        int i = 0;
        int j = 0;

  // create the positive binary number with the correct width
        for(i = 0; i < width - length; i++)
        {
                pos_binary_form[i] = '0';
        }

        for(j = 0; j < length; j++, i++)
        {
                pos_binary_form[i] = pos_binary[j];
        }

        // if the flag for negative was set then get the 2s complement
        if(negative == 1)
        {
                twos_complement(pos_binary_form, width);
        }

  // place null terminator
  pos_binary_form[width] = '\0';
  // free the earlier version of the positive binary number
  free(pos_binary);

        // printf("%s\n", pos_binary_form);

  // return the pointer to the positive binary number
  return pos_binary_form;
}

```

```
char **get_words_from_string(const char *input_string, int *num_of_words, int *num_of_frees)
{
    // amount holds the number of words that can be taken from string
    // intialy 5
    unsigned int amount = 5;

    // start with asumming that there are 5 words in the string
    char **words = (char **)xmalloc(amount * sizeof(char *));

    // this is the defualt size of each word
    unsigned int word_size = 5;

    // allocate each word size as 5 as default
    for (int i = 0; i < amount; i++)
    {
        words[i] = (char *)xmalloc(word_size * sizeof(char));
    }

    // this is used to go through the string
    unsigned int index = 0;
    /* this is used to see how many words of the allocated amount has actually
    been filled */
    unsigned int amount_level = 0;
    // this is used to see how many characters have been filled
    unsigned int word_size_level = 0;

    // this is a flag used to see if the end of a word has been reached
    char clean = 0;

    // until space or \n or tab is not reached
    while(input_string[index] == ' ' || input_string[index] == '\n'
                                        || input_string[index] == 9)
    {
        // increment index to skip ' ' or '\n'
        index++;
    }

    while(input_string[index] != '\0')
    {
        if(word_size_level == word_size)
        {
            word_size += 5;
            // create a new string of this size
            words[amount_level] = (char *)xrealloc((void *)words[amount_level],
                                        word_size * sizeof(char));
        }

        // until space or \n or tab is not reached
        while(input_string[index] == ' ' || input_string[index] == '\n'
                                                    || input_string[index] == 9)
        {
            // increment index to skip ' ' or '\n'
            index++;
            clean = 1;
        }

        if(clean == 1)
        {
            // place null terminator at the end of string
            words[amount_level][word_size_level] = '\0';
            // reset flag
            word_size_level = 0;

            //amount_levelincrement for the next string
            amount_level++;

            /* if the amount filled is the same as the max available then
            increase */
            if(amount_level == amount)
            {
                // create a temp variable of char **
                char ** temp = (char **)xmalloc((amount + 5) * sizeof(char *));

                // create space for each word
                for(int i = 0; i < amount + 5; i++)
                {
                    // this for the old data to be copied over so strlen used
                    if(i < amount)
                    {
                        temp[i] = (char *)xmalloc((strlen(words[i]) + 1)
                                                  * sizeof(char));
                    }
                    else    // this is for new data so default word_size
                    {
                        temp[i] = (char *)xmalloc(word_size * sizeof(char));
                    }
                }

                // copy old data over
                for(int i = 0; i < amount; i++)
                {
                    strcpy(temp[i], words[i]);
                }

                // clear old data
                for(int i = 0; i < amount; i++)
                {
                    free(words[i]);
```

```c
        }

        // clear words
        free(words);

        // assign words to the newly created memory
        words = temp;
        // NULL the inital pointer pointing to the newly created memory
        temp = NULL;
        // increment the total number of words for comparison next time
        amount += 5;
      }

      // reset flag such next time a space, \n or tab is reached
      clean = 0;
    }

    // assign current character
    words[amount_level][word_size_level] = input_string[index];

    // increment track and index for next character
    word_size_level++;
    index++;
  }

  // place \0 for last one
  words[amount_level][word_size_level] = '\0';

  // assign the num_of_words and num_of_frees for use of printing and freeing
  *num_of_words = amount_level + 1;
  *num_of_frees = amount;

  // return words
  return words;
}

void free_split(char ** words, int num_of_frees)
{
 // free the individual strings
 for(int i = 0; i < num_of_frees; i++)
 {
  free(words[i]);
  words[i] = NULL;
 }
 // free the double pointer
 free(words);
 words = NULL;
}
```

```
char * shift_left_one(char * data)
{
  int i;

  for(i = 1; i < strlen(data); i++)
  {
    data[i - 1] = data[i];
  }
  data[i - 1] = '\0';
  return data;
}
```

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