3211 Group Assignment Report

Group number: 2

Team members:

Ava Williams, Dong Huang, Anran Li, Farnaz Tavakol

Contents

[1. Instruction Table 1](#_Toc70024729)

[2. Instructions Description 2](#_Toc70024730)

[2.1 Load 2](#_Toc70024731)

[2.2 BNE 2](#_Toc70024732)

[2.3 Store 2](#_Toc70024733)

[2. 4 Bit\_Flip 2](#_Toc70024734)

[2.5 ROL 3](#_Toc70024735)

[2.6 XOR 3](#_Toc70024736)

[2.7 Parity 4](#_Toc70024737)

[2.8 Set\_Sig 4](#_Toc70024738)

[3. Hardware Supported Operations 5](#_Toc70024739)

[3.1 PC Loading 5](#_Toc70024740)

[3.2 Special Register Loading 5](#_Toc70024741)

[4. Hardware Components 6](#_Toc70024742)

[4.1 ALU 6](#_Toc70024743)

[4.2 Instruction Memory 6](#_Toc70024744)

[4.3 Data Memory 6](#_Toc70024745)

[4.4 I/O Unit 7](#_Toc70024746)

[4.5 Data Forwarder 7](#_Toc70024747)

[4. ASIP Control 7](#_Toc70024748)

[4.1 Receive Signal 8](#_Toc70024749)

[4.2 Send Signal 8](#_Toc70024750)

[4.3 Operation Flag 8](#_Toc70024751)

[4.4 Busy Signal 8](#_Toc70024752)

[4.5 ASIP Handshake and Usage 8](#_Toc70024753)

[Receiving Data from Network to Host CPU 9](#_Toc70024754)

[Sending Data from Host CPU to Network 9](#_Toc70024755)

[5. Processor Performance 10](#_Toc70024756)

[5.1 Metrics 10](#_Toc70024757)

[5.2 Critical Path 10](#_Toc70024758)

[5. Appendix 11](#_Toc70024759)

[5.1 Assembly Code 11](#_Toc70024760)

# Instruction Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mnemonic | Description | Type | Registers | OPCODE | Extra notes |
| NOP | - | - | - | 0000 | - |
| LOAD | Load 2 Bytes from memory to Rd | I-type | Rd, Rt,addr | 0001 | Uses absolute address not offset |
| BNE | Branch to addr if Rs not equal to Rt | I-type | Rd, Rt, addr | 0010 | Uses absolute address not offset |
| STORE | Save 2 Bytes from Rs to memory | I-type | Rd, Rt, addr | 0011 | Uses absolute address not offset |
| BIT\_FLIP | Flip data in Rs according to secret key in Rd and save to Rt | R-type | Rd, Rs, Rt | 0100 | - |
| ROL | Rotate-left-shift Rs by Rt and save to Rd | R-type | Rd, Rs, Rt | 0101 | - |
| XOR | XOR Rs and save to Rd | R-type | Rd, Rs, Rt | 0110 | does not use Rt register |
| PARITY | Check parity bit of Rs with Rt and save result to Rd | R-type | Rd, Rs, Rt | 0111 | - |
| Set\_Sig | Set imm as status register value (status register is a special register separate from the main register block) | I-type | Rd, Rs, imm | 1000 | Set\_Sig only needs immediate input because it is using pre-defined signal. |

Table 1.1: ISA instruction description and their mnemonics

# 2. Instructions Description

## 2.1 Load

Instruction Type: I-type

Registers: Rd, Rt

Immediate: absolute address in data memory

Description: Loads two bytes from the absolute address that is passed to it as an immediate and saves the result to register Rd. ***Note that load used absolute address and register Rt is not used.***

|  |
| --- |
| Example:  .data  key: .word 0xAAAA  .text main: load$1, $0, 0 |

## 2.2 BNE

Instruction Type: I-type

Registers: Rd, Rt

Immediate: absolute address to jump to

Description: Will branch to the given absolute address if registers Rd and Rt are not equal.

***Note that BNE uses absolute address.***

|  |
| --- |
| Example  .data  Data1: .word 0xAAAA  Data2: .word 0xAAAB  .text  Main:  Load $1,$0,0  Load $2, $0,1  BNE $1,$2, attack  Attack:  # This is where BNE jumps to |

## 2.3 Store

Instruction type: I-type

Registers: Rd, Rt

Immediate: absolute address to store the data to in data memory

Description: will store the data in Rd register in the given address in data memory

***Note that store uses absolute address.***

|  |
| --- |
| Example  .data  Data1: .word 0xAAAA  Data2: .word 0xAAAB  .text  Main:  # This code is swapping the memory content  # by first loading them to registers and then storing them from the  # registers in their new location  Load $1,$0,0  Load $2, $0,1  Store $1, $0,1  Store $2, $0, 0 |

## 2. 4 Bit\_Flip

Instruction Type: R-type

Registers: Rd, Rs, Rt

Description: Uses the Rt register to flip selected bits in the Rs register. The Rt register is assumed to contain the secret key. The top 4 bits of the Rt register determine which blocks of 4-bits get flipped in the Rs register.

The bits of the Rs register will be flipped as such:

|  |  |
| --- | --- |
| Rs | 1111 1111 1111 1111 |
| Rt | **1**0**1**0 ... |
| Rd | **0000** 1111 **0000** 1111 |

|  |
| --- |
| Example  .data  Secret key: .word 0xAAAA # top 4 bits are : 1010  Data\_to\_flip: .word 0xAAAB  .text  Main:  # This code performs a bit flip on Data\_to\_flip by the top four bits of Secret key  # the expected result stored to offset 3 is 0x5A5B  Load $1,$0,0  Load $2, $0,1  Bit\_Flip $2, $2, $1  Store $2, $0, 3 |

## 2.5 ROL

Instruction Type: R-type

Registers: Rd, Rs, Rt

Description: rotate-left-shift data in register Rs by Rt and stores the value in Rd. The Rt register is assumed to contain the secret key. As mentioned, the first Byte of secret key will be used for Bit\_flip. The remaining 12 bits are then grouped into 3 bit blocks (r\_i) as shown in the table. Each D\_i block of data is then rotated-left-shift by r\_i bits.

Secret key:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bit\_flip | r1 | r2 | r3 | r4 |
| XXXX | 001 | 010 | 011 | 100 |

Data:

|  |  |  |  |
| --- | --- | --- | --- |
| D1 | D2 | D3 | D4 |
| 1100 | 0100 | 1000 | 1100 |

Output

|  |  |  |  |
| --- | --- | --- | --- |
| 0110 | 0001 | 0001 | 1100 |

|  |
| --- |
| Example  .data  Data\_to\_shift: .word 0xF29C  Secret\_key: word 0xC48C  .text  Load $1, $0, 0  Load $2, $0, 1  Rol $3, $2, $1  Store $4, $0, 3 |

## 2.6 XOR

Instruction Type: R-type

Registers: Rd, Rs, Rt

Description: The instruction performs the XOR operation on each 4-bit sized group. The resulting value is then stored in Rd. As the result is less than 16 bits, the upper 12 bits with be padded with 0s. The Rt register is unused in this operation.

Data:

|  |  |  |  |
| --- | --- | --- | --- |
| D1 | D2 | D3 | D4 |
| 1100 | 0100 | 1000 | 1100 |

Output

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 | 0000 | 0000 | 1100 |

## 2.7 Parity

Instruction Type: R-type

Registers: Rd, Rs, Rt

Description: generate the parity bit for Rs and store result into Rd. The Rt register is unused for this operation. The last bit of Rd will be used for storing the parity result with the upper 15 padded with 0. The design uses an even parity, such that 0 corresponds to an even number of 1s and 1 an odd.

Data1:

|  |  |  |  |
| --- | --- | --- | --- |
| D1 | D2 | D3 | D4 |
| 1100 | 0100 | 1000 | 1100 |

Output 1

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 | 0000 | 0000 | 0000 |

Data2:

|  |  |  |  |
| --- | --- | --- | --- |
| D1 | D2 | D3 | D4 |
| 1100 | 0100 | 1001 | 1100 |

Output2

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 | 0000 | 0000 | 0001 |

## 2.8 Set\_Sig

Instruction Type: I-Type

Register: special status register

Immediate: 4-bit immediate determining the state of the processor

Description: Instruction sets the status register by a given immediate.

The status bits are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 2 | 1 | 0 |
| **busy** | **attack** | **error** | **valid** |

**Busy**: asserted when the ASIP is busy processing data

**Attack**: asserted when a data packet from the network contains tampered data, i.e. the given tag does not match the generated tag

**Error:** asserted when a data packet from the host processor has encountered a transfer issue, i.e. the given parity bit does not match the generated parity.

**Valid:** asserted when data from the network is free of error. Valid data is sent through the ASIP output to the host processor.

|  |
| --- |
| Example  .data  Secret\_key: word 0xC48C  .text  # example data recieve and polling loop  Load $1, $0, 0  While:  set\_sig 0000 # clear all status bits  Bne $1, $0, do\_recv  Do\_recv:  set\_sig 1000 # set busy  # check parity  # generate tag  Set\_sig 1001 # set valid and busy  Bne $1, $0, while |

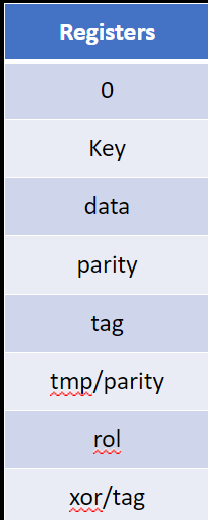
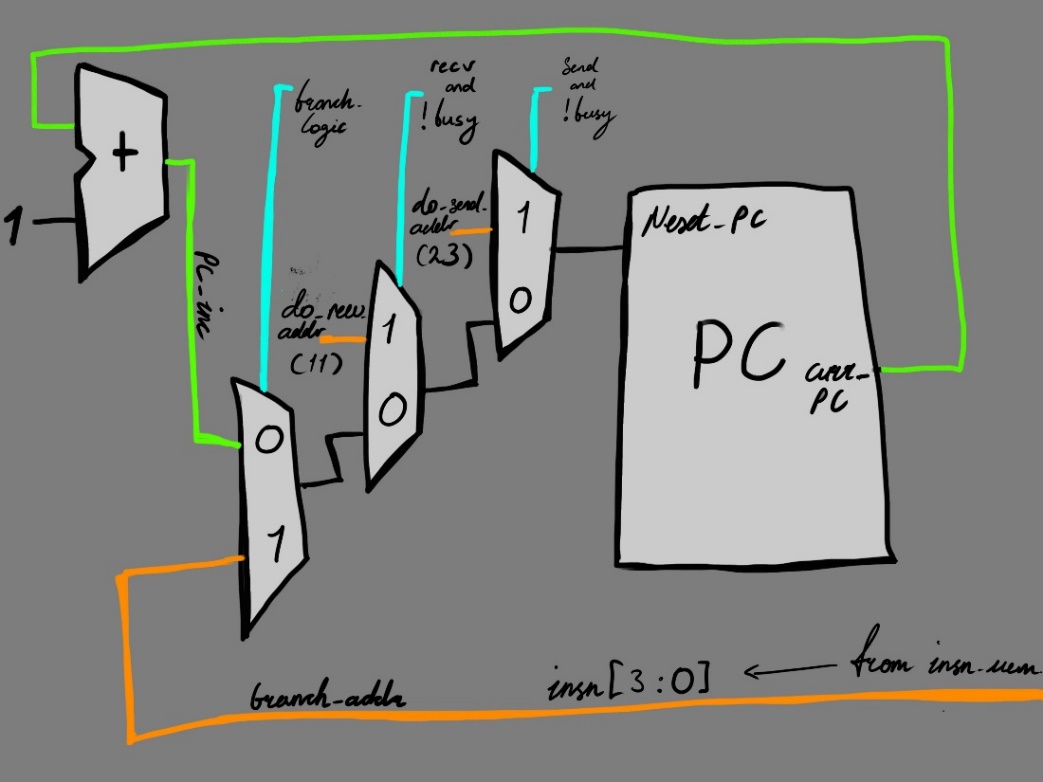
# 

# 3. Hardware Supported Operations

## 3.1 PC Loading

The ASIP uses hardware to load program counter. There are 3 multiplexers used to derive the next PC. The address of the receive function is 11 and the send function 23.

Picture 3.1 shows the block diagram for this component.



Picture 3.1: PC loading in hardware Picture 3.2: instruction memory special registers

## 3.2 Special Register Loading

The ASIP has 16 registers each 16 bit wide. The ASIP uses special Registes to store the data sent from network or CPU, tag, parity bit and the secret key. The following are the special registers.

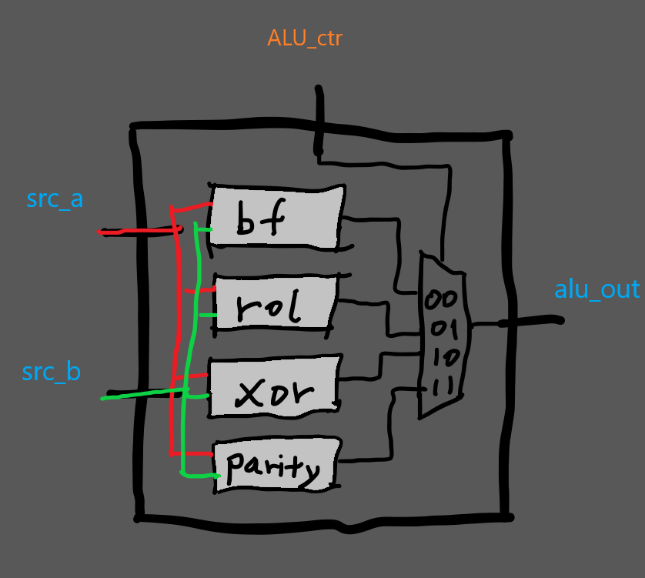
* $1: stores the secret key
* $2: stores data from network or CPU
* $3: stores parity bit
* $4: stores the tag of the input data

# 4. Hardware Components

## 4.1 ALU

The ALU uses the ALU\_ctr signal to control the output. The bit\_flip instruction is implemented by translating a 4-bit key flip value to 16 -bits and then performing a xor with the data input from Rs. ROL is implemented by concatenation. Parity is implemented by xor-ing together all bits.

|  |
| --- |
| **Data Memory** |
| Key |
| data |
| 0x000 | tag |



Sketch 4.1 ALU\_wrapper Sketch 4.3 memory management

## 4.2 Instruction Memory

The instruction memory consists of a 36-entry long array of 16-bit wide instructions. The reset signal is used to initialise the memory unit to its default values, as specified by the data\_memory component. A 6-bit address is used to read an instruction from memory which is then passed out of the component. The component is falling edge triggered. Implementation details can be found in instruction\_memory.vhd.

## 4.3 Data Memory

Data memory holds 16 entries size16-bits, address 0 is used to store the secret key value, address 1 is data being send to network, and address 2 is the tag for sent data (Sketch 4.3). All other memory entries are not used. Excluding the key value, all memory entries are initialised to 0 upon reset. Write is falling edge triggered and requires the write enable signal to also be asserted. Memory reads are continuous and are updated on a falling clock edge. See data\_memory.vhd for further information.

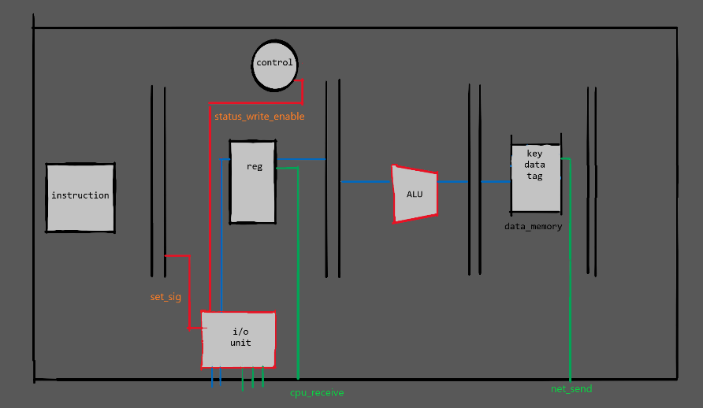
## 4.4 I/O Unit

I/O unit is used to process data received from host CPU and network. It keeps track of the status register which asserts the busy handshake signal with the network and host CPU, whist also providing alerts upon attack and error occurrences. Furthermore, the status register along with the send and receive input signals controls the hardware supported PC loading.

HDL design for I/O unit is found in io\_unit.vhd.

Status signal used for pc(colored in red lable with +) and output to host CPU (colored in black,lable with -)

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 2 | 1 | 0 |
| **Busy+** | **Attack-** | **Error-** | **Valid-** |



Sketch 4.4 for I/O unit datapath

## 4.5 Data Forwarder

# Data forwarder follows the implementation shown in the data slides, minus the Load Use Hazard (LUH), as this issue was not present in the firmware, it was decided to remove the functionality to preserve a faster datapath. Furthermore, the Data Forwarder only supports the forwarding of ALU components, branch and store instructions need two empty cycles between uses of the same register.

# 4. ASIP Control

## 4.1 Receive Signal

Receive signal is an external signal which is set by the testbench.

Receive signal is set if:

* If busy signal is 0
  + If operation flag is set to 1. (For information regarding operation flag read section 4.3)
    - If network is still transmitting data

## 4.2 Send Signal

Send signal is an external signal which is set by the testbench.

Send signal is set if:

* if busy signal is 0
  + If operation flag is set to 0. (For information regarding operation flag read section 4.3)
    - If CPU is still transmitting data.

## 4.3 Operation Flag

Operation flag is used to make sure the host program switches between receive and send operations and that neither is starving. Operation flag is set to 1 when a receive operation is to be run, and 0 for send. If either send or receive are not ready upon their corresponding turn, the operation bit is toggled.

## 4.4 Busy Signal

The busy signal is asserted when the ASIP is processing data. When it is de-asserted, the ASIP has finished processing data, and the host can check the output signals.

## 4.5 ASIP Handshake and Usage

The ASIP and the host communicate via a handshake process using the control signals recv, send, attack, and busy. Following are the steps in the handshake, and the behaviour of each control signal.

### Receiving Data from Network to Host CPU

|  |  |  |
| --- | --- | --- |
| Network | ASIP | Host CPU |
| Put input on network\_in. |  |  |
| Assert send to indicate to the ASIP to start processing. |  |  |
|  | Assert busy to indicate it is working. |  |
| It is now safe to remove input from network\_in. |  |  |
|  | If the tag does not match, assert attack for 3 clock cycles and stop, otherwise, skip to after ‘END ATTACK’. |  |
| If attack is asserted, then the input was corrupted. |  |  |
|  | De-assert busy. |  |
| **END ATTACK** | | |
|  | Assert valid. |  |
|  | Put output on network\_out for one clock cycle. | Retrieve output within one clock cycle. |
|  | De-assert busy and valid. |  |
| **END VALID** | | |

### Sending Data from Host CPU to Network

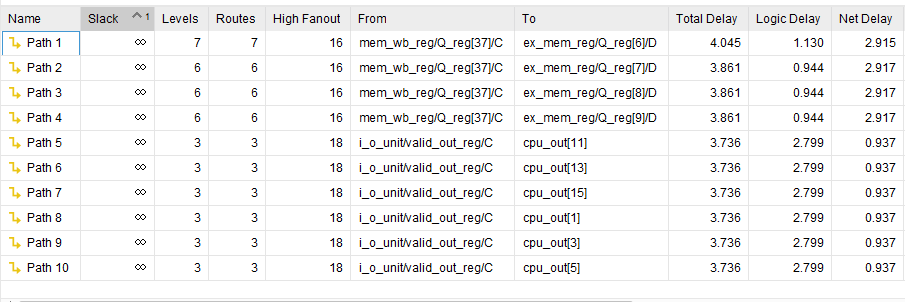
|  |  |  |
| --- | --- | --- |
| Host CPU | ASIP | Network |
| Put input on cpu\_in. |  |  |
| Assert recv to indicate to the ASIP to start processing. |  |  |
|  | Assert busy to indicate it is working. |  |
| It is now safe to remove input from cpu\_in. |  |  |
|  | If the parity check fails, assert error for 3 clock cycles, otherwise, skip to after ‘END ERROR’. |  |
| If error is asserted, then the data was corrupted. |  |  |
|  | De-assert busy. |  |
| **END ERROR** | | |
|  | De-assert busy. |  |
|  | Put output on network\_out until the next valid network\_out data. | Retrieve output before the next network send is available. |
| **END VALID** | | |

# 5. Processor Performance

## 5.1 Metrics

|  |  |
| --- | --- |
| Metric | Score |
| **Minimum Clock delay** | 4.045 ns |
| **Clock frequency** | 247 MHz |
| **Throughput (Upload | Download)** | 233.1 Mbps (116.6 Mbps) |
| **LUT usage** | 319 |
| **FF usage** | 421 |
| **Power usage** | 4.01 W |
| **Junction Temperature** | 43.4°C |

## 5.2 Critical Path



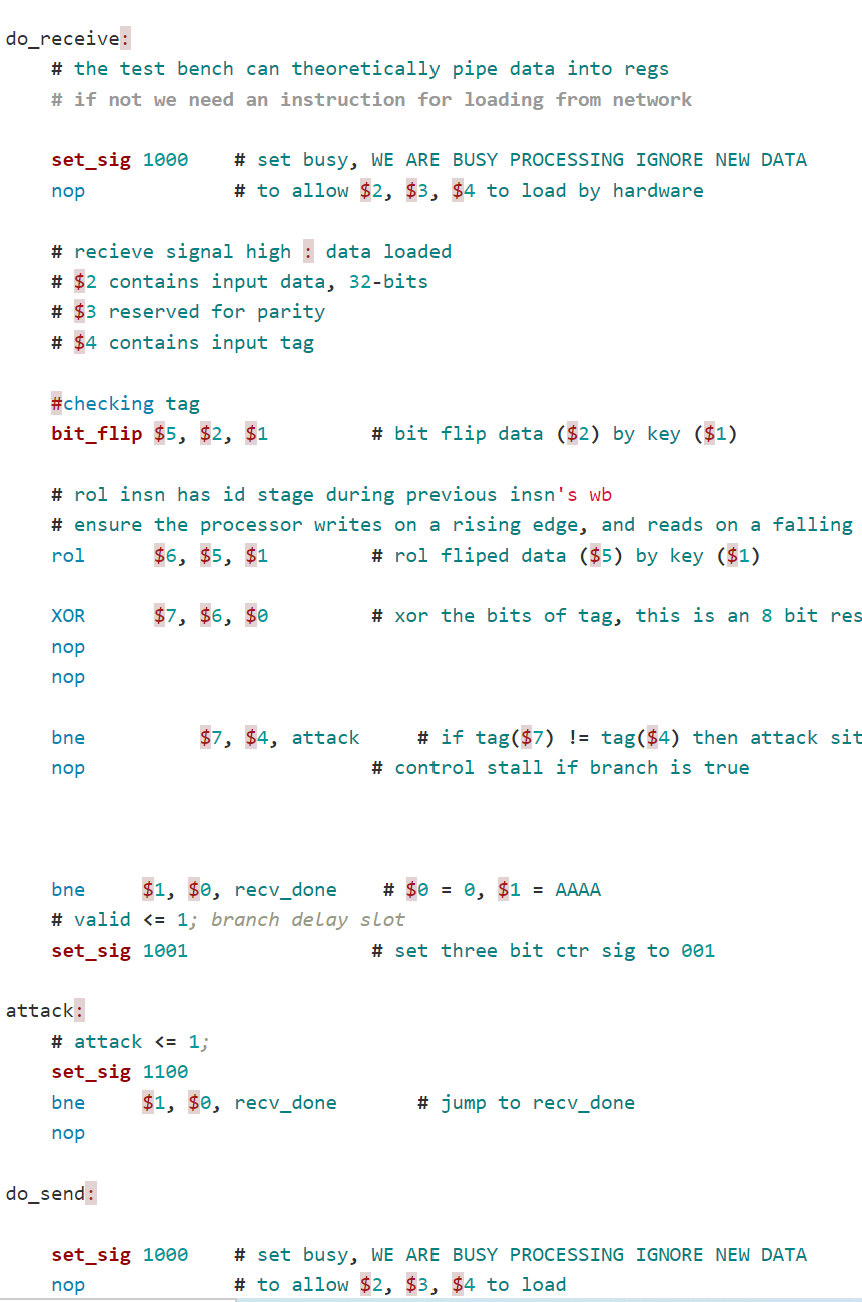
The design is limited by its critical path of 4.045ns delay. If this path were reduced or optimised, the delay for the design could be further reduced to 3.861ns resulting an improved clock frequency of 259MHz.

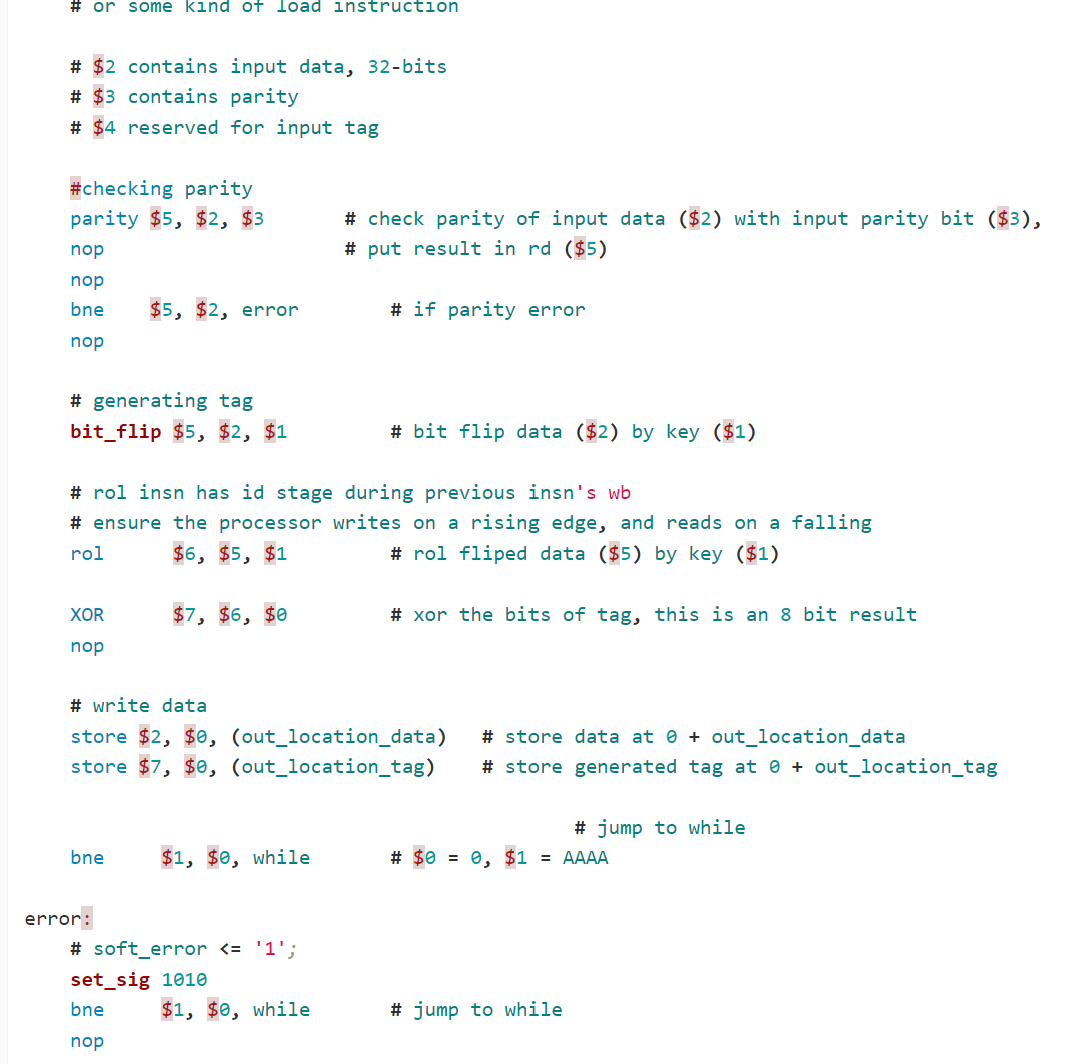
The longest path through the circuits is from the exit point of the MEM/WB stage register through the forwarding unit and ending at the EX/MEM register.

If the EX stage were to be split to two sections, with a new stage added for the forwarding unit, then the critical path could be reduced allowing for a shorter clock delay and a larger clock frequency. However, this would increase the length of the pipeline, thus slightly reducing the throughput of the processor

# 5. Appendix

## 5.1 Assembly Code





See linked: Processor Source code.