University College London

Department of Electronic and Electrical Engineering

# Performance characterisation of 8-bit RISC and OISC architectures

| *Author:* | *Supervisor:* | *Second Assessor:* |
|---|---|---|
| Mindaugas Jarmolovičius | Prof. Robert Killey | Dr. Ed Romans |
| zceemja@ucl.ac.uk | r.killey@ucl.ac.uk | e.romans@ucl.ac.uk |

**A BEng Project Final Report**

March 27, 2020

# Abstract

To be added

# Contents

# 1 Introduction

Since the 70s there has been a rise of many processor architectures that try to fulfil specific performance and power application constraints. One of more noticeable cases are ARM RISC (Reduced Instruction Set Computer) architecture being used in mobile devices instead of the more popular and robust x86 CISC (Complex Instruction Set Computer) architecture in favour of simplicity, cost and lower power consumption [1, 2]. It has been shown that in low power applications, such as IoTs (Internet of Things), OISC (One Instruction Set Computer) implementation can be superior in power and data throughput comparing to traditional RISC architectures [3, 4]. This project proposes to compare two novel RISC and OISC 8bit architectures and compare their performance, design complexity and efficiency.

## 1.1 Aims and Objectives

The project has three main objectives:

1. Design and build a RISC based processor.
2. Design and build an OISC based processor.
3. Design and perform a fair benchmark on both processors.

## 1.2 Supporting Theory

This section goes though supporting theory of RISC and OISC architectures, and their comparison.

Principal functions of general OISC architecture should have advantage in performance and power consumption while having lower transistor count. There are several theoretical models to implement a processor using only one instruction, most important are subtract and branch, MOVE and half adder architectures [25].

Some researches have proven benefits of subtract and branch architecture over RISC:

• Using OISC `SUBLEQ` (SUBtract and jump if Less or EQuial to zero) as a coprocessor for the MIPS-ISA processor to emulate the functionality of different classes shows desirable area/performance/power trade-offs [4].

• Comparing OISC `SUBLEQ` multicore to RISC achieves better performance and lower energy for streaming data processing [3].

Looking at OISC `MOVE` type, it has been researched since early 90s. It has been shown that `MOVE` can benefits of VLIW (very large instruction word) arrangement, classifying it as SIMO (single instruction, multiple operation) or SIMT (single instruction, multiple transports) architectures. Problem with all of these arrangement is that they exhibit poor or complex hardware utilization. OISC `MOVE` has been proposed as a design framework enabling lower complexity, better hardware utilization, and scalable performance [5]. In this framework a Transport Triggered Architecture (TTA) is described which describes how single instruction should transport data. To support theoretical benefits, a `MOVE32INT` TTA has been designed [6] and proven to be superior architecture to RISC. Using $1.6\mu m$ fabrication technology RISC achieved 20MHz clock with 20Mops/second, `MOVE32INT` implemented using SoGs (Sea of Gates) achieved 80MHz with 320Mops/second [7].

TTA framework as further used in other researches to implement Application-Specific Instruction Set Processors (ASIPs) to solve various problems. Some of the relevant examples are RSA calculation [8]; matrix inversion [9]; Fast Fourier Transform (FFT) [10]; IWEP, RC4 and 3DES encryption [11]; Parallel Finite Impulse Response (FIR) filter [12]; Low-Density Parity-Check (LDPC) encoding [13]; Software Defined Radio (SDR) [14]. One of the most recent researches use TTA architecture to solve Compressive Sensing algorithms. It showed 9 times of energy efficiency that

of FPGA implemented NIOS II processor, and theoretical 20 time energy efficiency that of ARM Cortex-A15 [15]. This particular research however, ARM Cortex-A15 uses 28nm Metal Gate CMOS technology, comparing to 60nm Silicon Gate CMOS technology used in Altera Cyclone IV-EP4CE115F29C7 FPGA which been used for implementing particular TTA. Both processor implementations cannot be directly compared.

Most of these researches show that TTA has greater power efficiency, higher clock frequency, lower logic resource count.

These benefits come with an expense, VLIW has bigger instruction word therefore bigger program size. TTA especially suffers from this due to redundant instructions. Some proposed solutions are variable length instructions and instruction templates, which reduced program size between 30% and 44%; [16, 17]; compression based on arithmetic coding [18]; and methods to remove redundant instructions [19]. Software is another difficulty as compiler need to take additional steps for data transportation optimisations. TTA software can be easily exploited however, to embed software pipelining and parallelism without need of extra hardware [20].

With proposed `MOVE` framework, hardware utilisation shown to be improved by reducing transition activity [21], reducing interconnects shown saving 13% of energy [22] on small scale. A novel architecture named SynZEN also showed a further improvements by using adaptable processing unit and simple control logic [23].

## 1.3 Project contents

Section 2 will go more in details behind motivation and project decisions based on Supporting Theory. Section 3 explains theory and result predictions. Section 4 explains both processor design choices and how each processor part is implemented on OISC and RISC processor. It also includes assembly design. In section 5, results will be discussed, including benchmark methods. Summary and conclusion of design and results can be found in section 6. Appendix in section 7 includes any other information such as both processor instruction set.

# 2 Goals and Objectives

This project can be classified as Design and Construction type, which explores alternative designs of processor architecture and microarchitecture. Main goals are:

1. Study and explore computer architectures, SystemVerilog and assembly languages.
2. Compare how well OISC `MOVE` architecture would perform in low performance microcontroller application comparing to equivalent and most commonly used RISC architecture.
3. View an alternative method of using OISC `MOVE` in a SISO (single instruction, single operation) structure, comparing to more commonly implemented TTAs VLIW architectures that are either SIMO or SIMT structure.

## 2.1 RISC Processor

As this is aimed for low power and performance applications it will be 8bit word processor with four general purpose registers, structure is similar to MIPS. RISC architecture will be mainly based on MIPS architecture explained in [24], except it this RISC processor would have 8bit databus and would have multiple optimisations related to 8bit limits. Some minimalistic ideas was also from [25].

## 2.2 OISC Processor

OISC `MOVE` has many benefits from VLIW and SIMO or SIMT design, however there is a lack of research investigating and comparing more general purpose OISC `MOVE` 8bit processor with short instruction word and SISO configuration. The main theory for building OISC architecture will be based on [25].

## 2.3 Design Criteria

In order for fair comparison between both architectures, a common design criteria:

- Minimal instruction size
- Minimalistic design
- 8bit data bus width
- 16bit ROM address width
- 24bit RAM address width
- 16bit RAM word size

When constructing these points, time and equipment resources were taken into consideration.

## 2.4 Benchmark

This benchmark include different algorithms that are commonly used in 8bit microcontrollers, IoT devices or similar low power microprocessor applications.

# 3 Theory and Analytical Bases

In this section differences in RISC and OISC are explained. It includes predictions and theory behind it.

This paper will be exploring is classical SISO processors. TTAs described section 1.2 are usually of type SIMT (single instruction, multiple transports) [7]; A middle between these two classes is SIMO type (single instruction, multiple operation).

## 3.1 RISC Processor

In this project, proposed RISC microarchitecture is mainly based on MIPS microachitecture [24]. Figure 3.1.1 represents simplified diagram of proposed RISC processor. In this architecture, program data travels from program memory to the control block where instruction is decoded. Then control

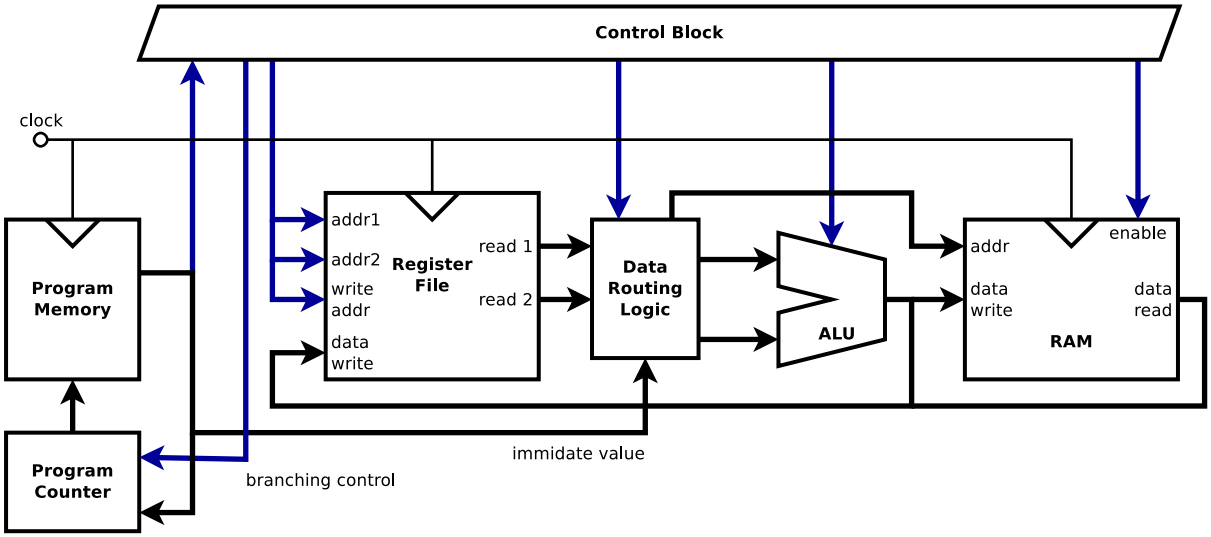


***Figure 3.1.1:*** *Abstract diagram of proposed RISC structure*

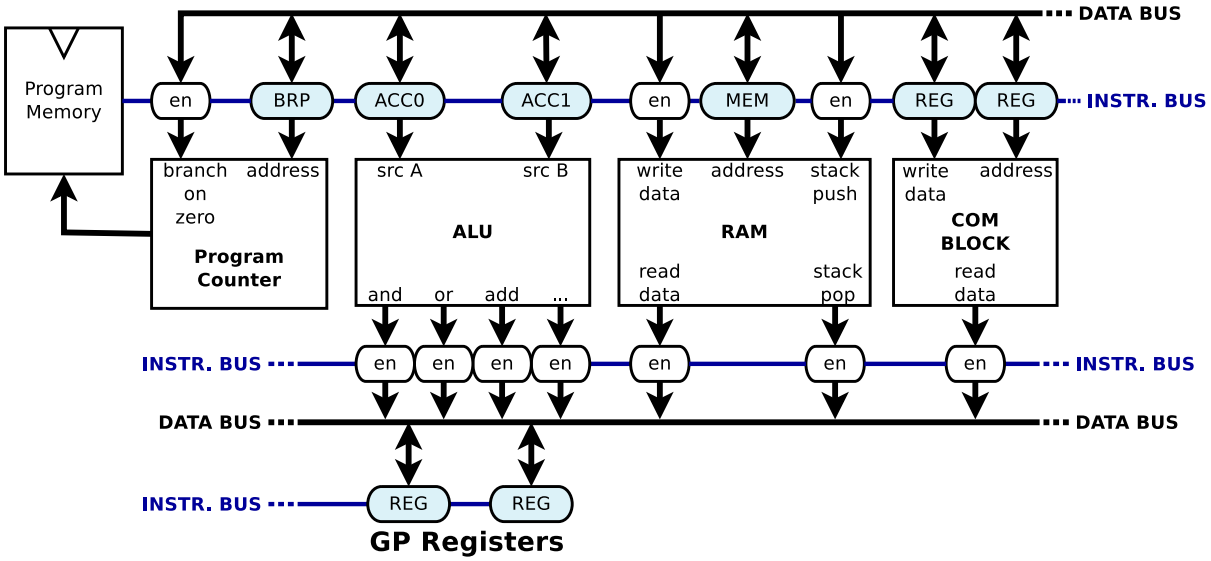


***Figure 3.1.2:*** *Abstract diagram of proposed OISC structure*

block further decides how data is directed in the datapath block. Such structure requires a complicated control block and additional data routing blocks. Depending on instruction, control block sets ALU, register file, memory operations and how data flows from one to other. Therefore, if non of blocks are bypassed, data can flow though every of these blocks, creating a log chain of combinational logic and increasing critical path. However this enables great flexibility allowing multiple operations to happen during single step, for example load value from register to memory, while address value is immediate offset by other register value using ALU. In order to increase performance of such processor, pipelining or multiple cores may be used.

### 3.1.1 Pipelining

$$T_c = t_{pcq} + t_{ROM} + t_{register} + t_{routing} + t_{ALU} + t_{RAM} + t_{setup} \quad (1)$$

Equation 1 shows maximum processor cycle period $T_c$ which depends on combina-

tional logic delay of every logic block, flip-flop time of propagation from clock to output of synchronous sequential circuit $t_{pcq}$ and flip-flop setup time $t_{setup}$.

$$T_{cp} = max \begin{pmatrix} t_{pcq} + t_{ROM} + t_{setup}, \\ t_{pcq} + t_{register} + t_{setup}, \\ t_{pcq} + t_{ALU} + t_{setup}, \\ t_{pcq} + t_{RAM} + t_{setup} \end{pmatrix} \quad (2)$$

Pipelinig separates each processor datapath block with a flip-flop. This changes combinational logic critical path this reducing cycle period. Pipelined processor cycle period $T_{cp}$ is represented in equation 2. Such modification could technically increase clock frequency by 2 or 3 times.

Pipelining, however, introduces other design complications. Instructions that depend on each other, for example operation $R = A + B + C$ needs to be executed in two steps, $t = A + B$ and $R = t + C$. Second step is depends and previous step result. Therefore additional logic is required to detect such dependencies and bypass datapath stages, or stall pipelining. Furthermore, breaching would also require stalling, temporary saving datapath stage and restoring it if needed when branching is concluded, or further branch prediction logic. Such dependency and branching issue requires hazards prevention logic which increases processor complexity and required resources.

### 3.1.2 Multiple cores

Multicore system is a solution to increase processor throughput by having multiple datapath and control logic instances, each running separate instructions. Cores share other resources such as RAM.

Multicore processor requires software adjustments as each processor core would execute separate programs. Therefore, some synchronisation between them is needed. A single additional core would also double the control and datapath blocks, substantially increasing resource requirements too. In addition, problems most often cannot be perfectly divided to parallel tasks due to some result dependencies between each task. Therefore, doubling processor core count would not likely result double the performance.

## 3.2 OISC Processor

Figure 3.1.2 represents simplified structure of OISC MOVE architecture. In simplest case, processor has a pair of buses - data and instruction. Instruction bus has source and destination address that connects two parts of processor via data bus. This mechanism allows data to flow around processor. Computation is accomplished by setting accumulators at destination addresses and taking computed values from source addresses. Other actions can be performed by at destination node for instance check value for branching or sending data to memory.

Needs to be finished.

## 3.3 Predictions

To be added.

# 4 Technical Method

This section describes methods and design choices used to construct two processors.

## 4.1 Machine Code

### 4.1.1 RISC

As the aim of instruction size to be as minimal as possible, RISC instruction decided to be 8bits with optional additional immediate value from 1 to 3 bytes. Immediate values are explained in section 4.7.

Decision was made to have instruction compose of operation code two operands - source/destination and source, which is similar to x86 architecture rather than MIPS. Three possible combinations of register address sizes are possible in such case from one to three bits. Two was selected as it allow having four general purpose registers which is sufficient for most applications, and allow four bits for operation code - allowing up to 16 instructions.

Due to small amount of available operation codes and not all instructions requiring two operands (for example `JUMP` instruction may not need any operands or could use one operand to have address offset), other two type instructions are added to the design - with one and zero operands. See figure 4.1.1. This enabled processor to have 45 different instructions while maintaining minimal instruction size. Final design has:

- **8** 2-operand instructions
- **32** 1-operand instructions
- **5** 0-operand instructions

Full list of RISC instructions are listed in table 7.1.1 in Appendix section.

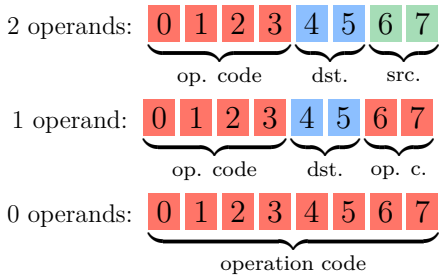


***Figure 4.1.1:*** *RISC instructions composition. Number inside box represents bit index. Destination (dst.) bits represents of source and destination register address.*

### 4.1.2 OISC

As OISC requires only a single instruction, composition of instruction mainly requires two parts - source and destination. To allow higher instruction flexibility a immediate bit has been added to replace source address by immediate value. Composition of finalised machine code is shown in figure 4.1.2.

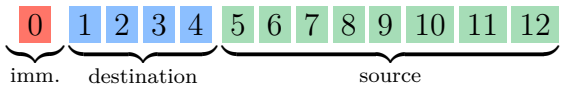


***Figure 4.1.2:*** *OISC instruction composition. Number inside box represents bit index.*

Decision was made to have source address to be eight bits to allow it be replaced with immediate value. Destination address was chosen to be as minimal as possible, leaving only four bits or 16 possible destinations. Final design has **15** destination and **41** source addresses. This is not the most space efficient design as 41 source addresses would require only six bits for address, wasting two bits every time non-immediate source is used.

Full list of OISC sources and destinations are listed in table 7.1.2 in Appendix section.

## 4.2 Data flow

### 4.2.1 RISC Datapath

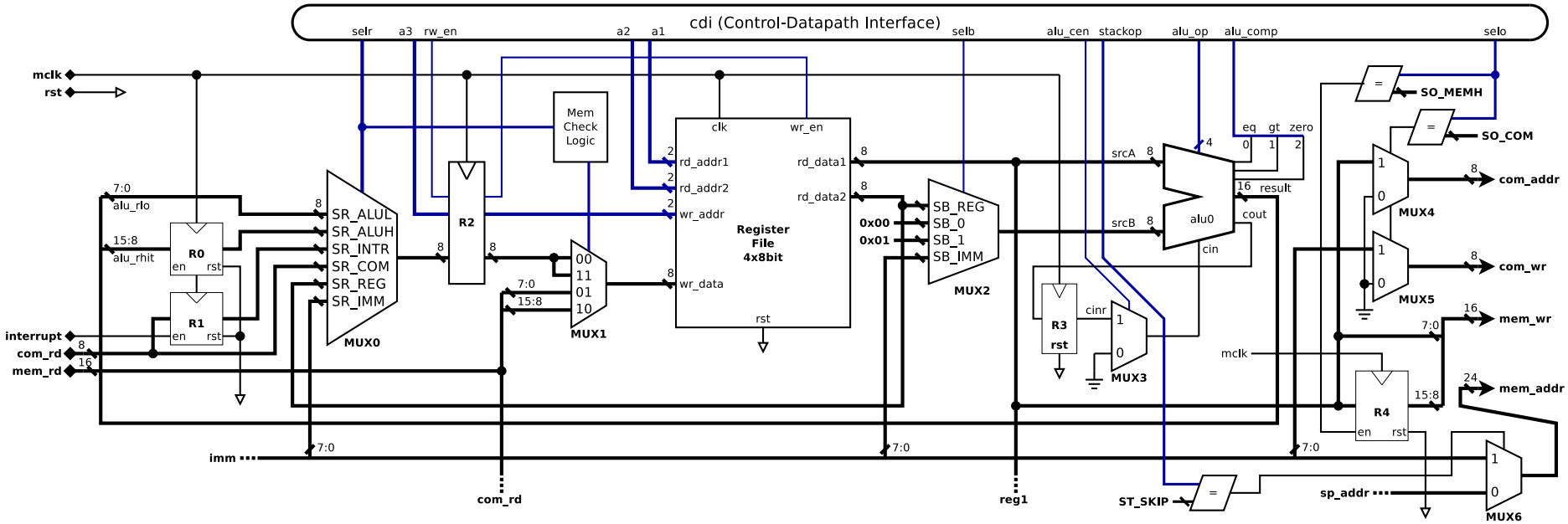


***Figure 4.2.1:*** *Digital diagram of RISC datapath*

Figure 4.2.1 above represents partial RISC datapath. Program counter, Stack pointer and Immediate Override logics are represented in figures 4.4.1, 4.3.1 and 4.7.1 respectively. CDI (Control-Data Interface) is HDL concept that connects datapath and control unit. Immediate value to datapath is provided by IMO block described in section 4.7.1.
Data to register file is selected and saved with *MUX0*. This data is delayed 1 cycle with *R2* to match timing that of data is taken from memory. If `LWLO` or `LWHI` is executed, *MUX1* select high or low byte from memory to read. To compensate for timing as value written to register file is delayed by 1 cycle, register file has internal logic that outputs *wr_data* to *rd_data1* or/and *rd_data2* immediately if *wr_en* is high and *rd_addr1* or/and *rd_addr2* matches *wr_addr*.

### 4.2.2 OISC Datapath

OISC datapath only consists of instruction and data buses, and small circuit that connect them to logic blocks that process the data. These logic blocks can represent ALU operation combinational logic, or any other part of a processor.

Figure 4.2.2 represents common destination circuit. It checks if particular block destination matches one on instruction bus, then enables latch and also sets flag to further logic.

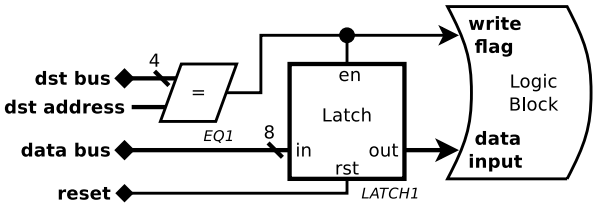


***Figure 4.2.2:*** *OISC processor data bus to destination connection logic*

Similarly Figure 4.2.3 represents source circuit connecting output of logic block. As logic block may only involve combinational logic a register is placed at the output of it. Buffer is used to connect data in register to data bus. This ensures that only one bus driver is present, ensuring do data collision.

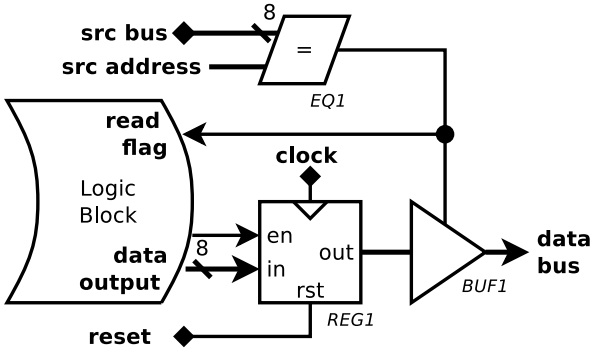


***Figure 4.2.3:*** *OISC processor data bus to source connection logic*

The general timing is designed so that the information at the source is immediately ready in data bus at rise of the processor clock. The source is connected to the destination connection where combinational logic is present.

### 4.2.3 OISC Datapath Implementation Problems

The complete implementation using latches for destination was not successful. Latches did not operate correctly when synthesised on FPGA. This issue might be caused by some timing problem between some source and destination logic combination. Exact cause was not resolved.

As a quick solution, latches at destination has been replaced with a clocked register that is triggered at opposite to source register clock edge (negative). This resolved this issue, however it effectively reduce period that data can propagate though logic blocks between source and destination by two.

## 4.3 Stack

This section describes RISC and OISC dedicated logic for stack pointer control. Stack pointer starts from the highest memory address value and "stacks" to lower memory address values. Both designs were simplified to only operate on two byte addresses, meaning that stack pointer has a constant `0xFF` value at least significant byte.

### 4.3.1 RISC Stack

RISC processor implements the stack pointer that is used in `PUSH`, `POP`, `CALL` and `RET` instructions. The stack pointer's initial address starts at the highest memory address (`0xFFFF`) and subtracts 1 when data is put to stack. Figure 4.3.1 represents the digital diagram for stack pointer. Note that the stack is only 16bit in size and the most significant byte is set to `0xFF`. The stack pointer circuit also supports *pc_halted* signal from program counter to prevent the stack pointer from being added by 1 twice during `RET` instruction.

One of the problems with the current stack pointer implementation is 8bit data stored in 16bit memory address, wasting a byte. This can be avoided by adding a high

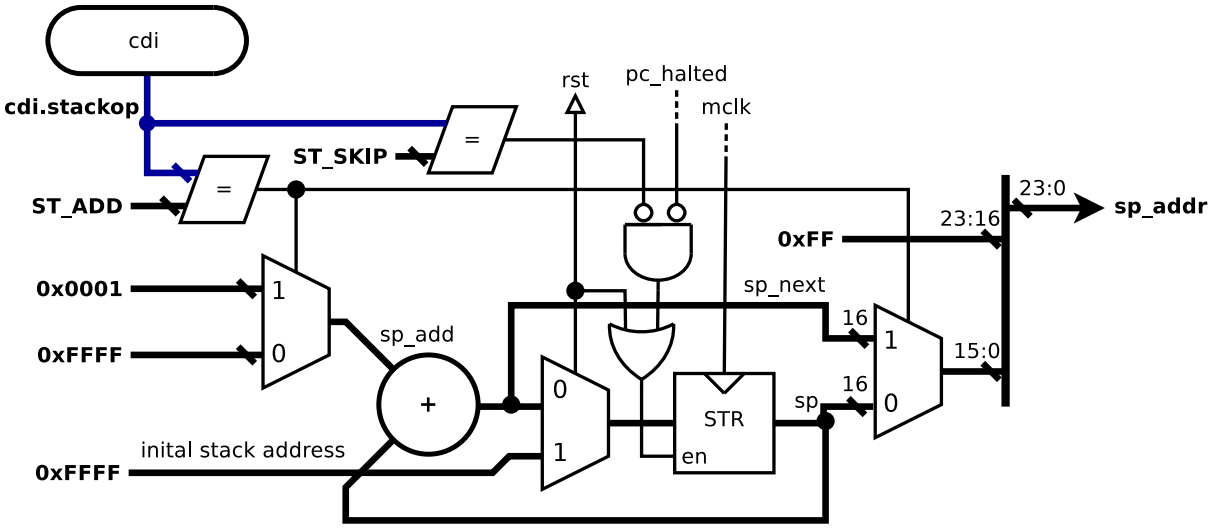


***Figure 4.3.1:*** *Digital diagram of RISC stack pointer logic*

byte register, however then it would cause problems when a 16bit program pointer is stored with `CALL` instruction. This can still be improved with a more complex circuit, or by using memory cache with 8bit data input. However with current implementation this does not affect processor comparison, it only increases stack size in memory.

### 4.3.2 OISC Stack

Stack pointer in OISC is very similar to RISC. In basic operation, when reset, push or pop flags are set, it changes the state of stack pointer by adding or subtracting its value by one, or resetting it to default. Logic diagram is shown in Figure 4.3.2

Logic diagram of stack control unique to OISC processor is shown in Figure 4.3.3. Push and pop flags are taken from source and destination logic. A cached value of last stored value is kept, so that it would be immediately available on source request. Pop flag is delayed by one cycle which ensures that once popped, lower stack value is written to cache during next cycle.

## 4.4 Program Counters

In this subsection, program counter and their differences will be described.

### 4.4.1 RISC Program Counter

Figure 4.4.1 represents the digital diagram for program counter. There are a few key features about this design: it can take values from memory for `RET` instruction; immediate value (*PC_IMM2* is shifted by 1 byte to allow `BEQ`, `BGT`, `BGE` instructions as first immediate byte used as ALU source B); can jump to interrupt address; produces

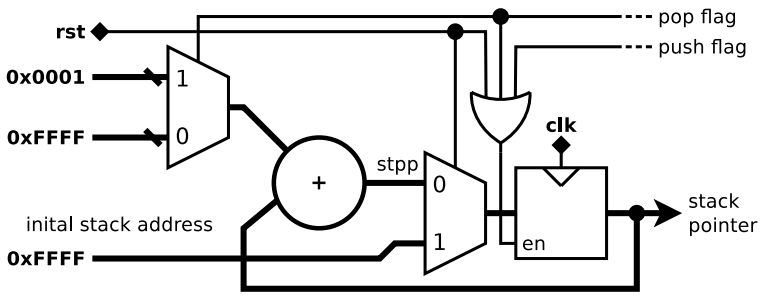


***Figure 4.3.2:*** *Digital diagram of OISC stack pointer logic*

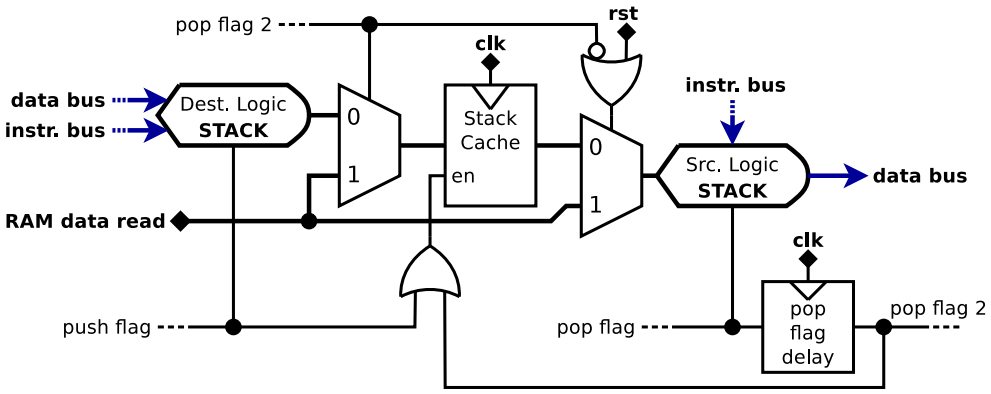


***Figure 4.3.3:*** *Digital diagram of OISC stack control logic*

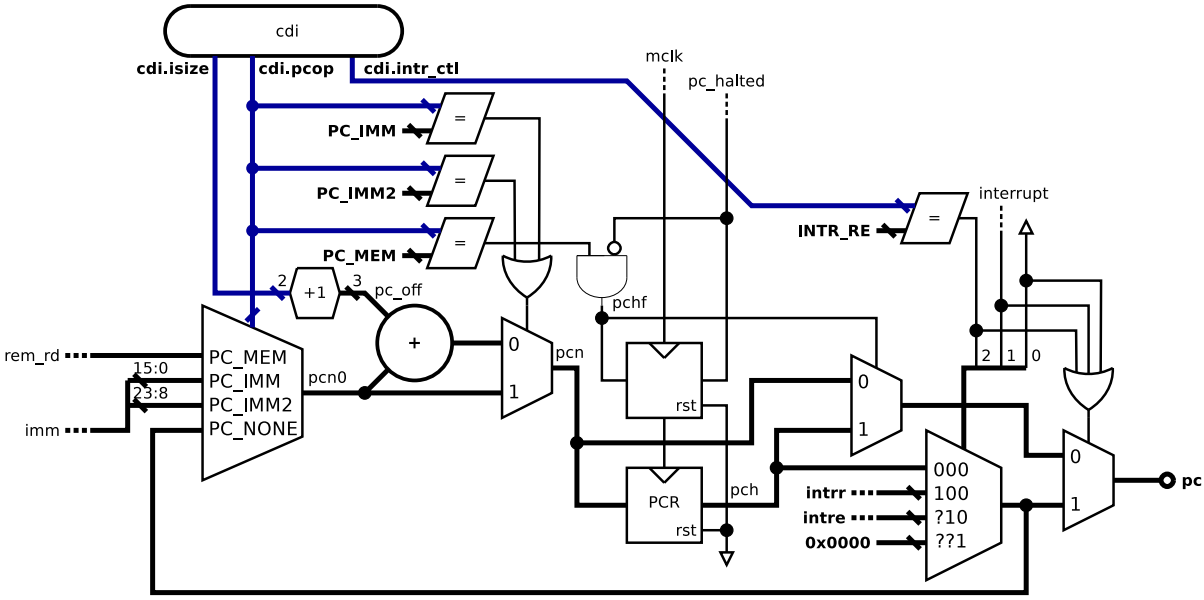


***Figure 4.4.1:*** *Digital diagram of RISC program counter*

*pc_halted* signal when memory is read (`RET` instruction takes 2 cycles, because cycle one fetches the address from stack and second cycle fetches the instruction from the instruction memory).

### 4.4.2 OISC Program Counter

OISC program counter is much simpler than RISC, as it does not have variable length instruction, delay flags instructions, or logic for selecting branch source address.

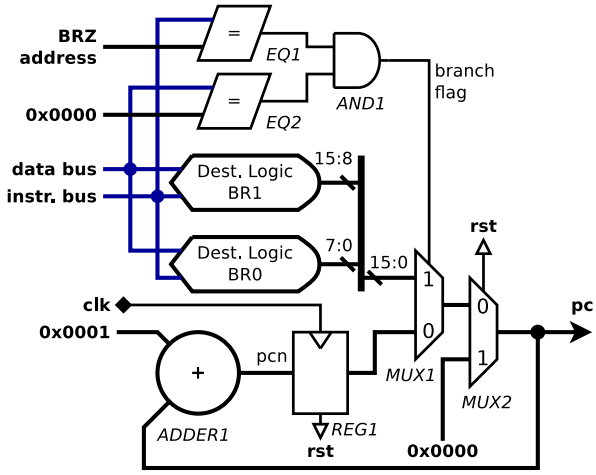


***Figure 4.4.2:*** *Digital diagram of OISC program counter*

Looking at Figure 4.4.2 bottom, the ba-

sic operation is to just add one to previous program counter with *ADDER1* and *REG1*, reset it to zero at reset with *MUX2*. Two destination logic blocks are used as accumulators to store branch address. Once instruction with BRZ destination is executed, *EQ2* check if data bus value is zero, which enables *MUX1* and overrides program counter to address stored in `BR0` and `BR1` accumulators. Unlike in RISC however, it requires three instructions to set new address and jump. Similarly, *CALL* and *RET* requires five and three instructions respectively. These RISC equivalent instructions are show in Listing 1.

***Listing 1:*** *OISC assembly code emulating RISC `JUMP`, `CALL` and `RET` instructions.*

```
%macro JUMP 1
  BR1 %1 @1
  BR0 %1 @0
  BRZ 0x00
%endmacro

%macro CALL 1
  BR1 %1 @1
  BR0 %1 @0
  STACK %%return @1
  STACK %%return @0
  BRZ 0x00
  %%return:
%endmacro

%macro RET 0
  BR0 STACK
  BR1 STACK
  BRZ 0x00
%endmacro
```

## 4.5 Arithmetic Logic Unit

This section will discuss ALU implementations of both processors. For fair comparison between OISC and RISC, ALU in both system will have the same capabilities described in table 4.5.1.

| Name | Description |
|---|---|
| ADD | Arithmetic addition (inc. carry) |
| SUB | Arithmetic subtraction (inc. carry) |
| AND | Bitwise AND |
| OR | Bitwise OR |
| XOR | Bitwise XOR |
| SLL | Shift left logical |
| SRL | Shift right logical |
| ROL | Shifted carry from previous SLL |
| ROR | Shifted carry from previous SRL |
| MUL | Arithmetic multiplication |
| DIV | Arithmetic division |
| MOD | Arithmetic modulus |

***Table 4.5.1:*** *Supported ALU commands for both processors*

### 4.5.1 OISC ALU

Due to the structure of OISC processor, ALU source A and B are two latches that are written into when `ALU0` or `ALU1` destination address is present. ALU sources are connected with every ALU operator and performed in single clock cycle. This value is stored in register so that it would immediately available in a next clock cycle as a source data. Figure 4.5.1 represents logic diagram of ALU with only addition and multiplication operators present. Note that output of *EQ3* is connected to enable of *REG3*, enabling output of carry to be only read after `ADD` source is requested. This previous source memory is also used for `SUB`, `ROL` and `ROR` operations. This allows processor to perform other operations such as store or load values, before accessing carry bit, or carried byte for `ROL` and `ROR` operations.

### 4.5.2 RISC ALU

RISC processor has very similar structure to OISC with two exceptions. Inputs to ALU comes from logic router that decided how to route data in datapath. Output buffers are replaced by one multiplexer that selects single output from all ALU operations. An-

other point is that RISC ALU output is 16bit, higher byte saved in "ALU high byte register" for `MUL`, `MOD`, `ROL` and `ROR` operations. This register is accessible with `GETAH` instruction.

## 4.6 Program Memory

This section describes how instruction memory (ROM) is implemented for both processors.

### 4.6.1 RISC Program Memory

In order to allow dynamic instruction size from one to four bytes a special memory arrangement is made. A system was required to access word (8bits) from memory and next three words. To achieve this four ROM blocks been utilised, each containing one fourth of sliced original data. Input address is offset by adders *ADDER1-3* and further divided by four by removing two least significant bits at **addr0-3**. Before concatenating output of each ROM block into final four bytes, ROM outputs **q0-3** are rearranged depending on **ar** signal. Note that *MUX1-4* each input is different, this may be better visualised with Verilog code in listing 2.

***Listing 2:*** *RISC sliced ROM memory multiplexer arrangement Verilog code*

```
case(ar)
  2'b00: data={q3,q2,q1,q0};
  2'b01: data={q0,q3,q2,q1};
  2'b10: data={q1,q0,q3,q2};
  2'b11: data={q2,q1,q0,q3};
endcase
```

### 4.6.2 OISC Program Memory

OISC instructions are fixed 13 bits, which causes different problems to RISC sliced memory - non-standard memory word size. To implement ROM in FPGA, Altera Cyclone IV M9K memory configurable blocks were used. Each blocks as 9kB of memory each allowing 1024x9bit configuration. Combining three of such blocks together yields 27bits if readable data in single clock cycle. To store instruction code to such configuration, pairs of instruction machine code sliced into three parts plus one bit for parity check, see figure 4.6.2. Circuit extract-

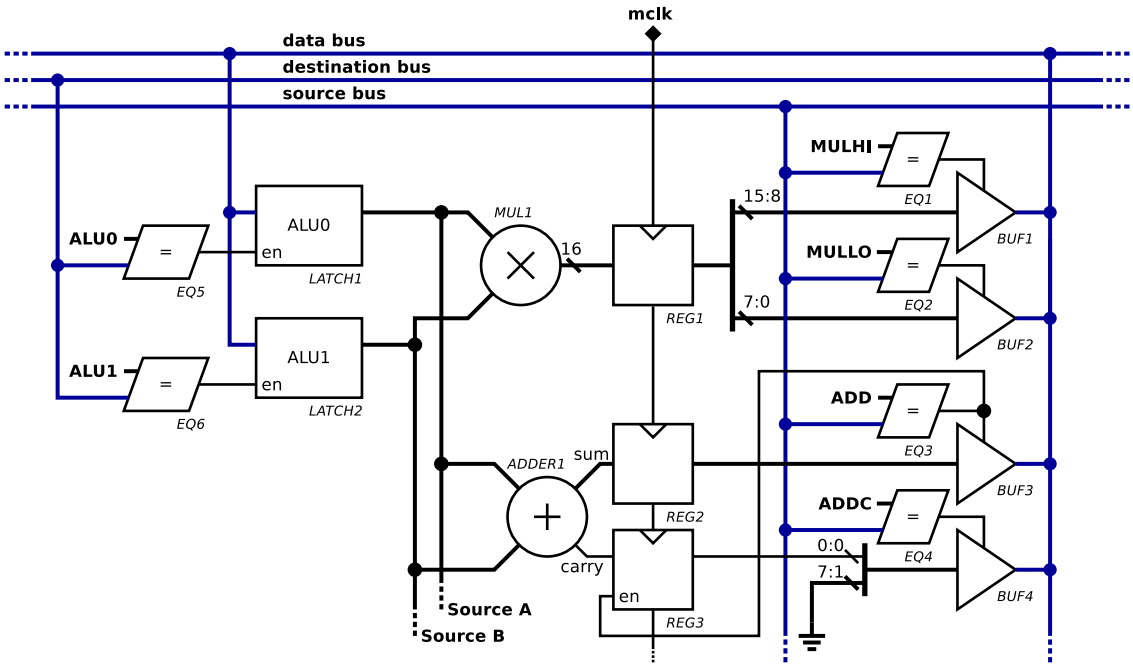


***Figure 4.5.1:*** *Digital diagram of OISC partial ALU logic*

ing each instruction is fairly simple, shown in figure 4.6.3.

## 4.7 Instruction decoding

This section describes RISC and OISC differences between instruction decoding and immediate value handling.

### 4.7.1 RISC

Already described in previous section 4.6, instruction from memory comes as 4 bytes. Least significant byte is sent to control block, other three bytes are sent to immediate override block (IMO) shown in figure 4.7.1. These three bytes are labelled as **immr**.

IMO block is a solution to change immediate value which enabled dynamically calculated memory pointers, branches dependant on register value or any other function that needs instruction immediate value been replaced by calculated register value. IMO is controlled by control block and **cdi.imoctl** signal, which is changed by `CI0`, `CI1` and `CI2` instructions. When signal is `0h`, this block is transparent connecting **immr** directly to **imm**. When any of `CI` instructions executed, one of IMO register is overridden by *reg1* value from register file. In order to override two or three bytes of immediate, `CI` instructions need to be executed in order. Only for one next instruction after last `CI` will have immediate bytes changed depending on what are values in *IMO* registers.
This circuit has two disadvantages:

1. Overriding immediate bytes takes one or more clock cycles,

2. At override, **immr** bytes are ignored therefore they are wasting instruction memory space.

Second point can be resolved by designing a circuit that would subtract the amount of overridden IMO bytes from *pc_off* signal (program counter offset that is dependant on i-size value) at the program counter, thus effectively saving instruction memory space. This solution however would introduce a complication with the assembler as additional checks would need to be done during compiling to check if IMO instruction are used.

### 4.7.2 OISC

OISC immediate value is set in instruction decoder shown in figure 4.7.2. Decoder operation is simple - instruction machine code is split into three parts as described in 4.1.2.

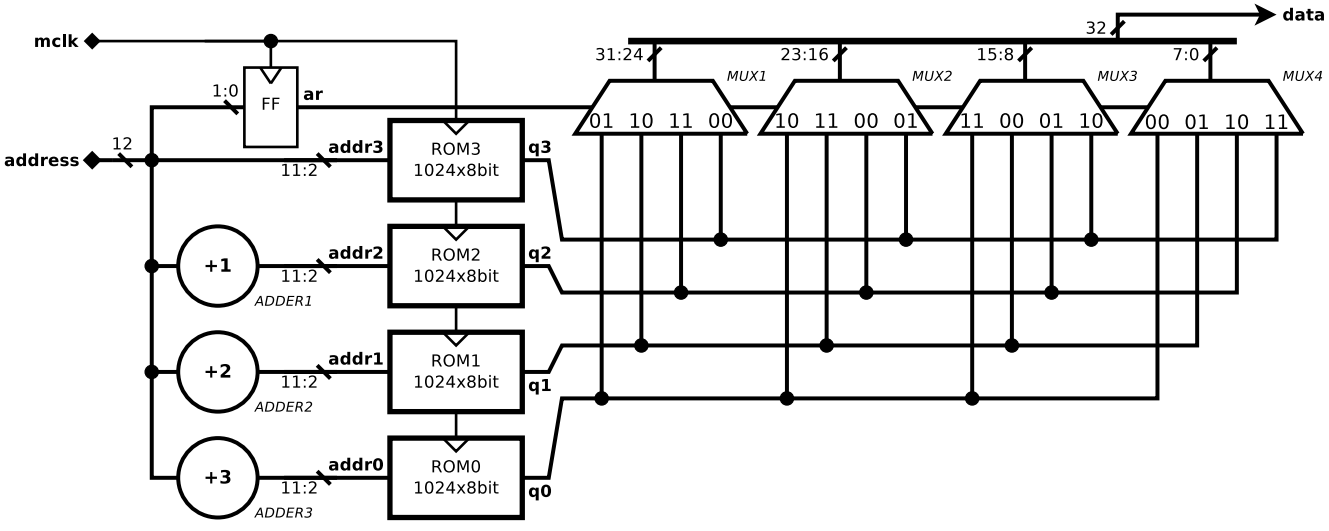


***Figure 4.6.1:*** *Digital diagram of RISC sliced ROM memory logic*

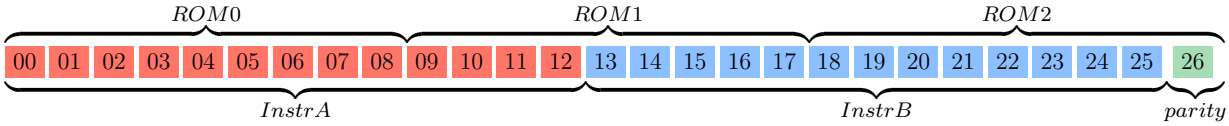


***Figure 4.6.2:*** *OISC three memory words composition. Number inside box represents bit index.*

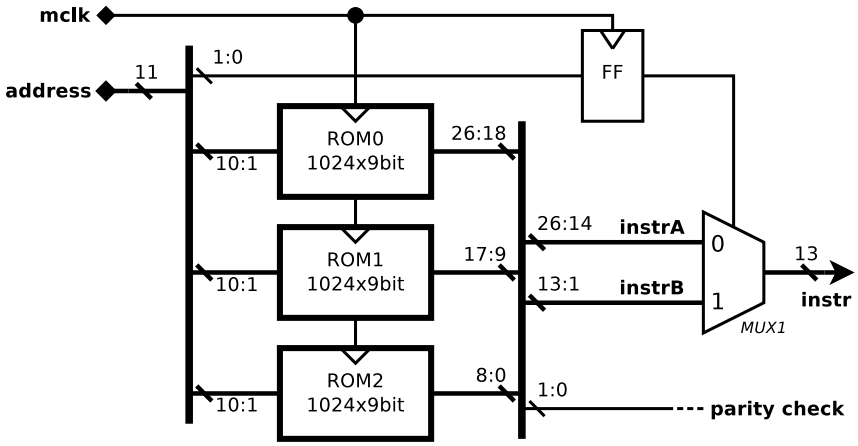


***Figure 4.6.3:*** *Digital diagram of OISC instruction ROM logic*

If instruction source address is `00h`, connect data bus with constant 0 via *MUX2*. If immediate bit is 1, set source address to `00h` (to make sure no other buffer source connects to data bus), and connect instruction source address (immediate value) to databus via *MUX2* and *BUF1*.

## 4.8 Assembly

There are two steps between assembly code and its execution on a processor. First it needs to be converted to binary machine code. Secondly, binary data needs to be sliced to different parts described in section 4.6. These slices also need to be converted into appropriate formats, as simulation, HDL synthesis and flashing memory directly to FPGA memory, all use different formats.

A universal assembler was implemented with python for both processors. Flowchart in figure 4.8.1 represents general structure of assembler process. It splits assembly file into three parts - sections, definitions and macros. Definitions are keywords mapped to values which are saved in global label dictionary. Macros are a chunk of assembly code and is used as templates.

There are only two sections implemented in assembler - `.text` and `.data`. Section `.text` contains all machine instructions which will be stored in program ROM memory. Section `.data` is used for global and static data, and it will be written into RAM memory. This section contains values such as strings and structures uninitialised data as labels which data is RAM memory location.

Section `.text` is processed line by line. Each has label, instruction name and instruction arguments. Label however is optional, if line contains it, label is saved to global label dictionary with program address. If line instruction name is a macro, line is replaces by macro and instruction arguments are used as macro arguments. Otherwise instruction name is decoded and stored in instruction list with original arguments.

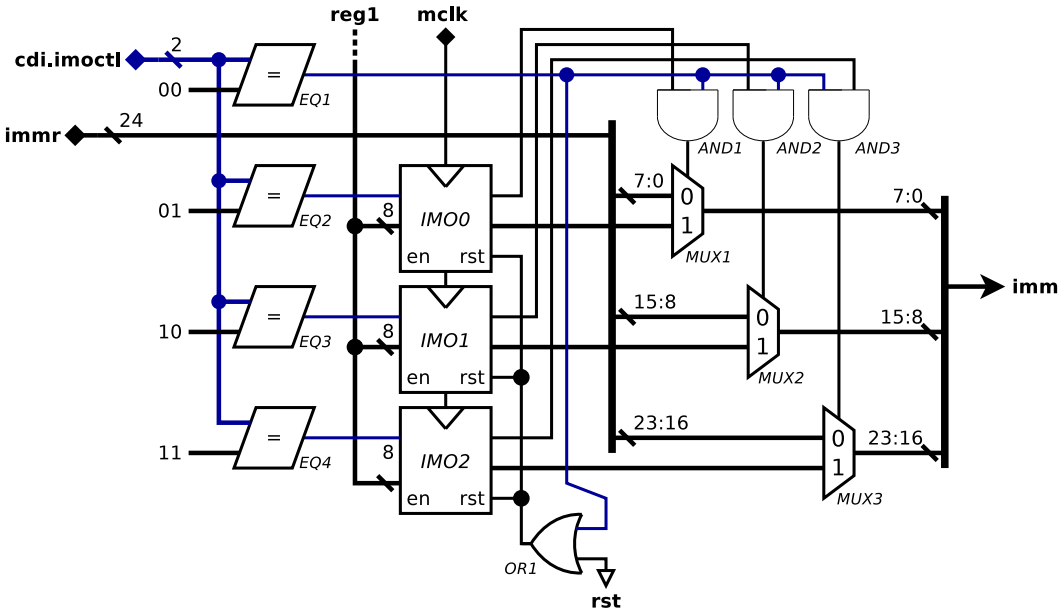


***Figure 4.7.1:*** *Digital diagram of RISC immediate override system*

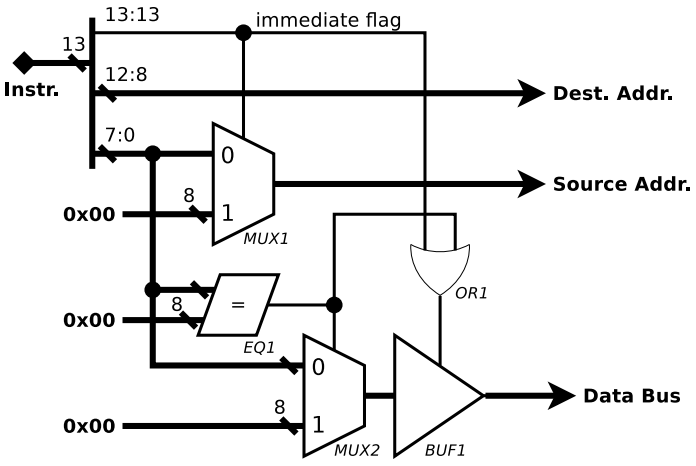


***Figure 4.7.2:*** *Digital diagram of OISC instruction decoder*

After all instruction lines are completed, each stored instruction arguments are processed, labels are replaced with binary values, any other processing is done such as addition by constant, byte selection, etc. Completed list is then saved as raw binary. Similarly, `.data` section labels also replaced and it is saved as binary data.

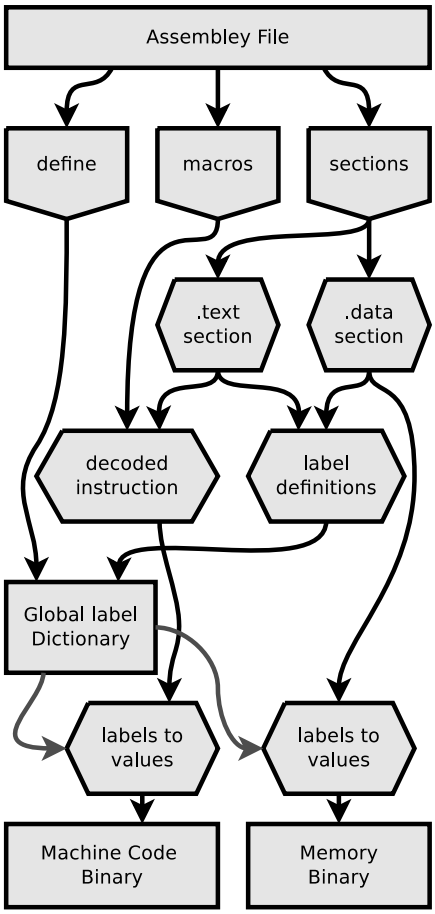


***Figure 4.8.1:*** *Flow chart of assember converting assembly code into machine code and memory binary.*

## 4.9 System setup

Description to be added. Include FPGA used, makefile, how programs are flashed into memory via JTAG and debugging tools

# 5 Results and Analysis

## 5.1 FPGA logic component composition

This subsection looks at test and its results to find how much FPGA logic components each processor takes and what is composition of each part.

Test was performed with Quartus synthesis tool and viewing flow summary report. This report includes synthesised design metrics including total logic elements, registers, memory bits and other FPGA resources. Test will only look at logic elements and registers. Total number of logic elements was found out by synthesising full processors, then commenting relevant parts of code, re-synthesising and viewing changes in total logic elements. Such method may not be the most accurate, because during HDL synthesis circuit is optimised an unused connections removed. This means that more logic may be not synthesised than intended.

There are four parts of each processor that will be tested:

1. **Common** - processor auxiliary logic that is used by both processors. It includes communication block with UART, RAM and PLL (Phase-Locked Loop, for master clock generation).

2. **ALU** - as described in section 4.5, both processors have slightly different implementation of ALU.

3. **Memory** - processors memory management, including stack.

4. **Other** - reminding logic of processor that was not analysed.

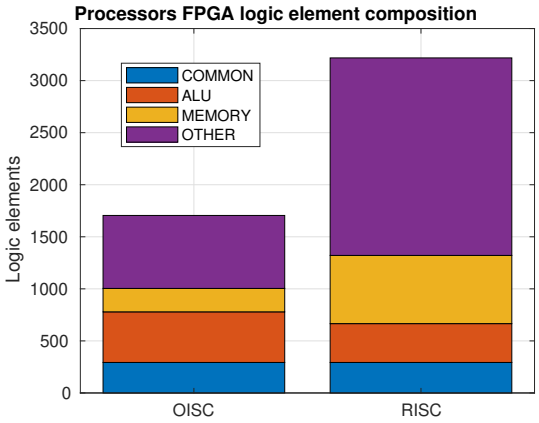


***Figure 5.1.1:*** *Bar graph of FPGA logic components taken by each processor.*

Results of a test are shown in figures 5.1.1 and 5.1.2. Common logic uses 293 logic elements and 170 registers. OISC uses 1705 logic elements, while RISC uses 3218. Excluding common logic, OISC takes 48.3% of RISC's logic elements.

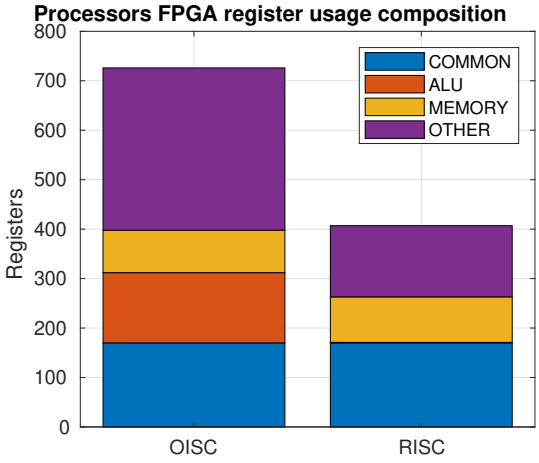


***Figure 5.1.2:*** *Bar graph of FPGA register resources taken by each processor.*

OISC uses 726 logic elements, while RISC uses 407. Excluding common logic, OISC uses 78.4% more registers than RISC.

Looking at composition, OISC ALU takes 30.2% more logic gates. Looking at figure 5.1.2, high number of OISC ALU registers can be observed which concludes, that higher resource usage is OISC ALU code include buffer logic.

Memory logic elements composition of OISC is only 34.4% of RISC's and 7% lower for register resources, comparing to RISC. This indicate that by removing memory logic for RISC, synthesis tool may removed also other parts of processor, possibly part of control block because it mostly contains combinational logic.

Other logic includes instruction decoding with ROM, register file, program counter. RISC exclusively has control block. Note that OISC uses only three ROM memory blocks whereas RISC uses four as explained in section 4.6, however this should make a minimal difference as M9K memory blocks are not included in FPGA logic element or register count. Comparing both processors, OISC has only 37% of other logic components to RISC, however it has 2.28 times more registers. This shows a logic component - register trade-off. OISC buffer and common registers logic that connects bus require many more registers whereas RISC uses combination logic in control block in order to control same data in datapath.

Much higher logic components in RISC can be also explained more complicated register file, ROM memory logic and program counter. All of these components has some additional logic for timing correction or additional functionality required by these blocks integration into datapath.

## 5.2 Power analysis

Power analysis was performed to analyse power consumption of both processors. This has been accomplished by connecting FPGA board to a laboratory power supply with 4V to an external power input. A shunt resistor of was used of 1.020Ω was connected in series to calculate current. Supply voltage and voltage across shut resistor were measured using oscilloscope with data sampling feature. Multiple tests have been performed with different processor configurations. Between each tests a period of about 5 minutes was given for FPGA to reach steady state.

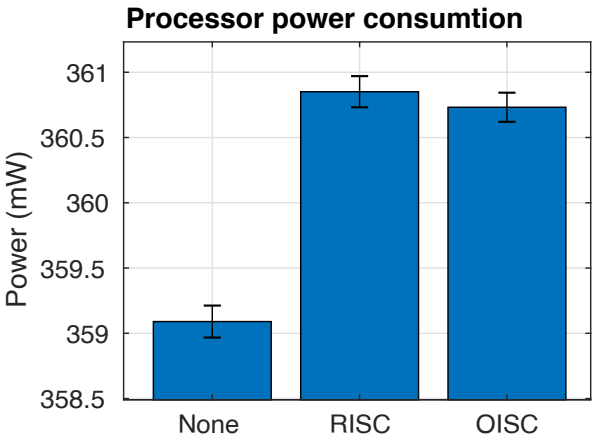


***Figure 5.2.1:*** *Measured power of processors when implemented on FPGA, running 16bit multiplication function in loop. None indicates auxiliary-only power.*

Figure 5.2.1 represents power results. Auxiliary power includes whole FPGA board, voltage regulators, and synthesised logic on FPGA required to support a processor (such as PLL, UART, Input/Output control, RAM). RISC and OISC bars in the graph indicate auxiliary power plus processor power, which means that the processor itself takes relatively small amount comparing to auxiliary power, about 0.5%.

During this test clock frequency of 1MHz was used. Due to equipment unavailability, further tests were not carried out to investigate power consumption at different frequencies.

### 5.2.1 Activity Factor

An activity factor could be also found using Equation 3 where $P$ is power, $C_{total}$ indicate total gate capacitance and $V_{DD}$ indicate voltage supplied to the transistors.

$$\alpha = \frac{P}{C_{total} \cdot f \cdot V_{DD}^2} \tag{3}$$

As $C_{total}$ and $V_{DD}$ are constants, measuring power at different frequencies allows finding activity factor. This value could be used to compare how much of a processor circuit is active. Further design improvements could be used to optimise power [10, 14, 21, 22].

## 5.3 Benchmark Programs

A number of and programs have been written to test both processors. These involve simple functions that could be commonly used in 8bit processors:

- **Printing:** Sends data to UART. It includes waiting until UART is available for transmission.
- **Printing unsinged integer:** Uses binary-coded decimal algorithm to convert 8 or 16bit binary value to decimal value and print it.
- **16bit multiplication:** Uses simple matrix multiplication.
- **16bit division:** Uses Long division algorithm to divide two 16bit numbers, result including a reminder.
- **16bit modulo:** Uses "Russian Peasant Multiplication" algorithm to perform Modulo operation with two 16bit numbers.
- **Prime number calculator:** Uses Sieve of Atkins algorithm to calculate primer number, operates on 16bit numbers and utilise 16bit multiplication and modulo functions.

### 5.3.1 Instruction composition

This test is performed to investigate instruction composition of each function to see how similar it is between RISC and OISC processors.

- **MOVE** - All instructions that move data around internal processor registers.
- **ALU** - Instructions that are used to perform ALU operation.
- **MEMORY** - Instructions that are required to send/retrieve data from system memory, except stack.
- **STACK** - Instructions that push/pop data from memory stack.

- **COM** - Instruction(s) that send/receive data from communication block.
- **BRANCH** - Instructions that are used to make program branching.
- **OTHER** - Any other instructions.

| Name | Instructions |
|---|---|
| MOVE | MOVE, CPY0, CPY1, CPY2, CPY3, CI0, CI1, CI2 |
| ALU | ADD, ADDI, SUB, SUBI, AND, ANDI, OR, ORI, XOR, XORI, DIV, MUL, ADDC, SUBC, INC, DEC, SLL, SRL, SRA, GETAH |
| MEMORY | LWLO, LWHI, SWLO, SWHI |
| STACK | PUSH, POP |
| COM | COM |
| BRANCH | BEQ, BGT, BGE, BZ, JUMP, CALL, RET |

***Table 5.3.1:*** *RISC processor instruction groups used in instruction composition test.*

| Name | Destination |
|---|---|
| MOVE | REG0, REG1 |
| ALU | ALU0, ALU1 |
| MEMORY | MEM0, MEM1, MEM2, MEMLO, MEMHI |
| STACK | STACK |
| COM | COMA, COMD |
| BRANCH | BR0, BR1, BRZ |

***Table 5.3.2:*** *OISC processor instruction desination groups used in instruction composition test*

| Name | Instructions |
|---|---|
| MOVE | ALU0, ALU1, REG0, REG1, PC0, PC1, NULL, IMMEDIATE |
| ALU | ADD, ADDC, SUB, SUBC, AND, OR, XOR, SLL, SRL, EQ, GT, GE, NE, LT, LE, MULLO, MULHI, DIV, MOD, ADC, SBC, ROL, ROR |
| MEMORY | MEM0, MEM1, MEM2, MEMLO, MEMHI |
| STACK | STACK |
| COM | COMA, COMD |
| BRANCH | BR0, BR1 |

***Table 5.3.3:*** *OISC processor instruction source groups used in instruction composition test*

Each function was ran on simulated processor, program counter and instruction been recorded into file at every cycle. File recording was done with SytemVerilog test bench, it started recording when program counter matched `.start` location and stopped when it matched `.done` location. Code shown in listings **??** and **??** enabled both location to be static, not depending on test function executed.

***Listing 3:*** *RISC assembly frame for executring tests*

```
setup:
JUMP .start
.done:
JUMP .done
.start:
; Setup values
; Call function
JUMP .done
```

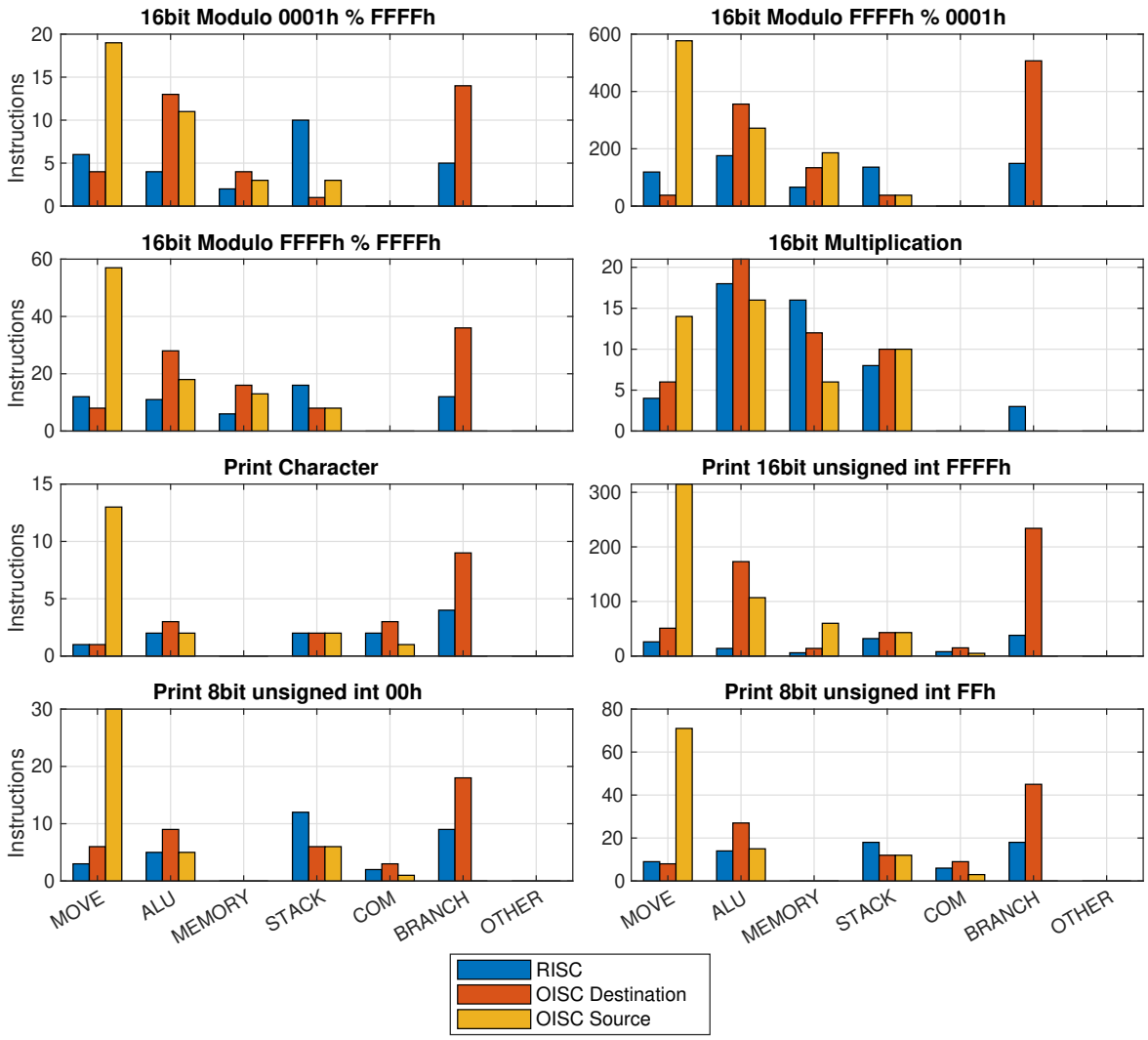


***Figure 5.3.1:*** *Graph of instruction composition for every benchmark program.*

***Listing 4:*** *OISC assembly frame for executring tests*

```
setup:
BR1 .start @1
BR0 .start @0
BRZ 0x00
.done:
BRZ 0x00
.start:
; Setup values
; Call function
BR1 .done @1
BR0 .done @0
BRZ 0x00
```

Each function recorded file then was further analysed and each instruction was grouped. Recorded program counter was used to find effective program space. This has been achieved by calculating unique instances of program counter and summing up instruction size for each of them. In RISC, dynamic instruction size has been taken into account.

From results in Figure 5.3.1 few key differences can be seen. Across every test, OISC has much more *BRANCH* destination and *MOVE* source groups. *BRANCH* group can be explained by emulated `CALL`, `RET` and `JUMP` instruction explained in section 4.4.2. High number of *MOVE* source group instructions may be explained by using immediate values as separate source,

where RISC uses instruction that integrate with immediate in instructions such as `ADDI`. In most cases *ALU* group instructions are also higher than for OISC comparing to RISC. This shows lower OISC ALU efficiency, mostly due to need to move data to septate accumulators.

### 5.3.2 Performance

This subsection investigates time and clock cycles to run benchmark programs. Simulation was sued to find a number of cycles required to execute each function. Note that prime number calculator was not simulated due to too complex dynamic nature of program.

Print 16bit decimal and modulo operation were executed with different arguments to show the worst and the best case scenarios as algorithms length depend on inputs. This is not the case for 16bit multiplication as this it has no branching.

Results are shown in Figure 5.3.2. In most cases, OISC requires around 55-67% more instruction, with some exceptions. These results can be better explained in following subsection 5.3.1.

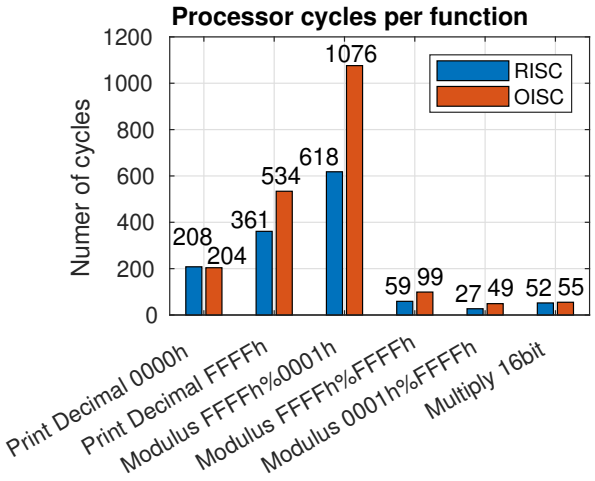


***Figure 5.3.2:*** *Simulated results of cycles that taken to perform function.*

Another set of benchmarks have been performed and on both processors once they been implemented on FPGA. Time taken for perform each set has been recorded. This have been done via UART connection, a single character was sent to indicate start and stop of benchmark. In order to void slight timing variation due low baud rate of UART, each benchmark was performed many iterations. Figure 5.3.3 represents results.

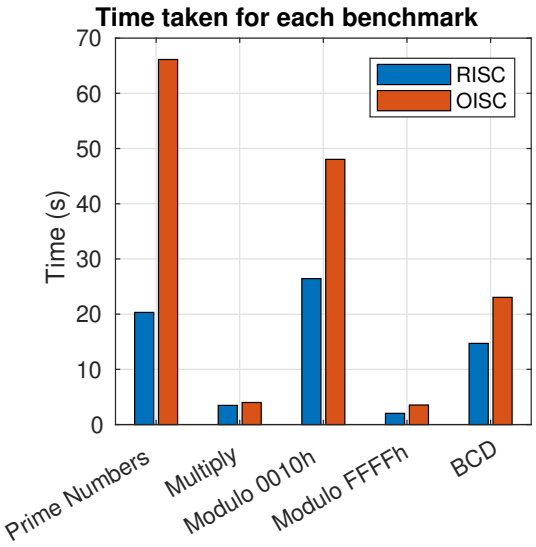


***Figure 5.3.3:*** *Time taken perform each benchmark on FPGA.*

Results indicate that on average OISC takes about 71% longer to execute same benchmark. This is close to results found with simulation. Prime number calculator have taken 3.26 times longer.

Benchmarks include:

- **Prime Numbers:** Calculate every prime number between 5 to $2^{16}$.
- **Multipy:** 16bit multiplication iterated 65536 times.
- **Modulo 0010h:** 16bit *0010h* modulo that operated on every number between 0 and 65536.
- **Modulo FFFFh:** 16bit *FFFFh* modulo that operated on every number between 0 and 65536.
- **BDC:** Encoded 16bit binary to ASCII decimal number without printing.

### 5.3.3 Program space

Figure 5.3.4 represents effective program size for each test function. Effective program size only includes instruction that been executed depending on argument, meaning that it does not fully represent complete function. A specific argument might cause branching and skipping some function code which would not be added to effective program size. In this test, the main objective is to look difference in instruction size required to execute the same function, therefore not representing full program size is not relevant.

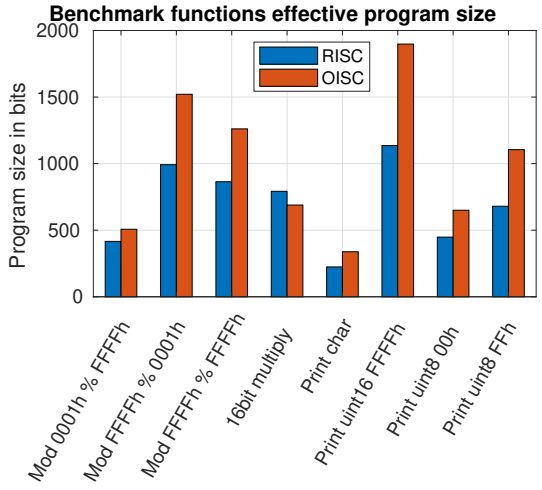


***Figure 5.3.4:*** *Bar graph showing effective size in bits each benchmark function is taking in program memeory.*

## 5.4 Maximum clock frequency

To find maximum clock frequency, processors were loaded with basic print string function an d 16bit multiplication. Then frequency was constantly increased until resulting output though UART was not correct.

In order to change clock frequency, three parameters were changed and HDL code resynthesised:

- **PLL frequency multiplier and divider:** PLL takes 50MHz clock that is sourced from crystal on FPGA board and converts it to master clock $f_{mclk}$. Multiplier and divider values are used to adjust $f_{mclk}$.
- **UART frequency divider:** Division value was calculated as $D = \left\lfloor \frac{f_{mclk}}{4f_{baud}} \right\rfloor$. UART rate was set to 9600 baud. UART module itself has four times oversample.

Frequency was changed in 5MHz increments.

Theoretical maximum frequency was found using Quartus Timing Analysis tool. Slow 1200mV 85°C model was used.

| | Theoretical | Actual |
|---|---|---|
| RISC | 114.08MHz | 75-70MHz |
| OISC | 64.68MHz | 45-40MHz |

***Table 5.4.1:*** *Theoretical and actual maximum frequencies of both processors.*

Theoretical and actual results show unexpected results shown in Table 5.4.1, RISC operated at about 40% higher maximum frequency than OISC.

As explained in Subsection 4.2.3, OISC logic blocks has about twice less time for data propagation. Keeping that in mind, and assuming that latch propagation and register setup periods are insignificant to critical path of OISC logic block, maximum OISC frequency could be double as high as, reaching 80-90MHz. This also assumes that there is no other part of processor would have limit. Further timing analysis needs to be carried out to confirm this.

## 5.5 Future work

Description to be added.

# 6 Conclusion

Description to be added.

# References


[1] T. Jamil. "RISC versus CISC". In: vol. 14. 3. 1995, pp. 13–16. DOI: 10.1109/45.464688.

[2] E. Blem, J. Menon, and K. Sankaralingam. "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures". In: 2013. DOI: 10.1109/hpca.2013.6522302.

[3] Minato Yokota, Kaoru Saso, and Yuko Hara-Azumi. "One-instruction set computer-based multicore processors for energy-efficient streaming data processing". In: 2017. DOI: 10.1145/3130265.3130318.

[4] Tanvir Ahmed et al. "Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC". In: 2015. DOI: 10.1109/euc.2015.23.

[5] H. Corporaal and H. Mulder. "MOVE: a framework for high-performance processor design". In: *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 1991, pp. 692–701. DOI: 10.1145/125826.126159.

[6] Henk Corporaal. *MOVE32INT: Architecture and Programmer's Reference Manual*. Tech. rep. 1994.

[7] H. Corporaal. "Design of transport triggered architectures". In: *Proceedings of 4th Great Lakes Symposium on VLSI*. 1994, pp. 130–135. DOI: 10.1109/GLSV.1994.289981.

[8] J. Hu et al. "A Novel Architecture for Fast RSA Key Generation Based on RNS". In: *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*. 2011, pp. 345–349. DOI: 10.1109/PAAP.2011.75.

[9] A. Burian, P. Salmela, and J. Takala. "Complex fixed-point matrix inversion using transport triggered architecture". In: *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. 2005, pp. 107–112. DOI: 10.1109/ASAP.2005.25.

[10] J. ádník and J. Takala. "Low-power Programmable Processor for Fast Fourier Transform Based on Transport Triggered Architecture". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 1423–1427. DOI: 10.1109/ICASSP.2019.8682289.

[11] P. Hamalainen et al. "Implementation of encryption algorithms on transport triggered architectures". In: *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*. Vol. 4. 2001, 726–729 vol. 4. DOI: 10.1109/ISCAS.2001.922340.

[12] P. Salmela et al. "Scalable FIR filtering on transport triggered architecture processor". In: *International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005.* Vol. 2. 2005, 493–496 Vol. 2. DOI: 10.1109/ISSCS.2005.1511285.

[13] B. Rister et al. "Parallel programming of a symmetric transport-triggered architecture with applications in flexible LDPC encoding". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2014, pp. 8380–8384. DOI: 10.1109/ICASSP.2014.6855236.

[14] J. Multanen et al. "Power optimizations for transport triggered SIMD processors". In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2015, pp. 303–309. DOI: 10.1109/SAMOS.2015.7363689.

[15] M. Safarpour, I. Hautala, and O. Silvén. "An Embedded Programmable Processor for Compressive Sensing Applications". In: *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 2018, pp. 1–5. DOI: 10.1109/NORCHIP.2018.8573494.

[16] J. Heikkinen et al. "Evaluating template-based instruction compression on transport triggered architectures". In: *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.* 2003, pp. 192–195. DOI: 10.1109/IWSOC.2003.1213033.

[17] J. Helkala et al. "Variable length instruction compression on Transport Triggered Architectures". In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. 2014, pp. 149–155. DOI: 10.1109/SAMOS.2014.6893206.

[18] J. Wei et al. "Program Compression Based on Arithmetic Coding on Transport Triggered Architecture". In: *2008 International Conference on Embedded Software and Systems Symposia*. 2008, pp. 126–131. DOI: 10.1109/ICESS.Symposia.2008.9.

[19] Su Wang et al. “An instruction redundancy removal method on a transport triggered architecture processor”. In: *Proceedings of the 2009 12th International Symposium on Integrated Circuits*. 2009, pp. 602–604.

[20] L. Jiang, Y. Zhu, and Y. Wei. “Software Pipelining with Minimal Loop Overhead on Transport Triggered Architecture”. In: *2008 International Conference on Embedded Software and Systems*. 2008, pp. 451–458. DOI: 10.1109/ICESS.2008.18.

[21] T. Pionteck et al. “Hardware evaluation of low power communication mechanisms for transport-triggered architectures”. In: *14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings.* 2003, pp. 141–147. DOI: 10.1109/IWRSP.2003.1207041.

[22] T. Viitanen et al. “Heuristics for greedy transport triggered architecture interconnect exploration”. In: *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2014, pp. 1–7. DOI: 10.1145/2656106.2656123.

[23] S. Hauser, N. Moser, and B. Juurlink. “SynZEN: A hybrid TTA/VLIW architecture with a distributed register file”. In: *NORCHIP 2012*. 2012, pp. 1–4. DOI: 10.1109/NORCHP.2012.6403142.

[24] David Money Harris and Sarah L Harris. *Digital design and computer architecture*. 2nd ed. Elsevier, 2013.

[25] William F Gilreath and Phillip A Laplante. *Computer Architecture: A Minimalist Perspective*. Kluwer Academic Publishers, 2003.

# 7 Appendix

## 7.1 Processor instruction set tables

***Table 7.1.1:*** *Instruction set for RISC processor. * Required immediate size in bytes*

| Instr. | Description | I-size * |
|---|---|---|
| | *2 register instructions* | |
| MOVE | Copy value from one register to other | 0 |
| ADD | Arithmetical addition | 0 |
| SUB | Arithmetical subtraction | 0 |
| AND | Logical AND | 0 |
| OR | Logical OR | 0 |
| XOR | Logical XOR | 0 |
| MUL | Arithmetical multiplication | 0 |
| DIV | Arithmetical division (inc. modulus) | 0 |
| | *1 register instructions* | |
| COPY0 | Copy intimidate to a register 0 | 1 |
| COPY1 | Copy intimidate to a register 1 | 1 |
| COPY2 | Copy intimidate to a register 2 | 1 |
| COPY3 | Copy intimidate to a register 3 | 1 |
| ADDC | Arithmetical addition with carry bit | 0 |
| ADDI | Arithmetical addition with immediate | 1 |
| SUBC | Arithmetical subtraction with carry bit | 0 |
| SUBI | Arithmetical subtraction with immediate | 1 |
| ANDI | Logical AND with immediate | 1 |
| ORI | Logical OR with immediate | 1 |
| XORI | Logical XOR with immediate | 1 |
| CI0 | Replace intimidate value byte 0 for next instruction | 1 |
| CI1 | Replace intimidate value byte 1 for next instruction | 1 |
| CI2 | Replace intimidate value byte 2 for next instruction | 1 |
| SLL | Shift left logical | 1 |
| SRL | Shift right logical | 1 |
| SRA | Shift right arithmetical | 1 |
| LWHI | Load word (high byte) | 3 |
| SWHI | Store word (high byte, reg. only) | 0 |
| LWLO | Load word (low byte) | 3 |
| SWLO | Store word (low byte, stores high byte reg.) | 3 |
| INC | Increase by 1 | 0 |
| DEC | Decrease by 1 | 0 |
| GETAH | Get ALU high byte reg. (only for MUL & DIV & ROL & ROR) | 0 |
| GETIF | Get interrupt flags | 0 |
| PUSH | Push to stack | 0 |
| POP | Pop from stack | 0 |
| COM | Send/Receive to/from com. block | 1 |
| BEQ | Branch on equal | 3 |
| BGT | Branch on greater than | 3 |

***Table 7.1.1:*** *Instruction set for RISC processor. * Required immediate size in bytes*

| Instr. | Description | I-size * |
|---|---|---|
| BGE | Branch on greater equal than | 3 |
| BZ | Branch on zero | 2 |
| *0 register instructions* | | |
| CALL | Call function, put return to stack | 2 |
| RET | Return from function | 0 |
| JUMP | Jump to address | 2 |
| RETI | Return from interrupt | 0 |
| INTRE | Set interrupt entry pointer | 2 |

***Table 7.1.2:*** *Instructions for OISC processor.*

| Name | Description |
|---|---|
| *Destination Addresses* | |
| ACC0 | Set ALU source A accumulator |
| ACC1 | Set ALU source B accumulator |
| BR0 | Set Branch pointer register (low byte) |
| BR1 | Set Branch pointer register (high byte) |
| BRZ | If source value is 0, set program counter to branch pointer |
| STACK | Push value to stack |
| MEM0 | Set Memory pointer register (low byte) |
| MEM1 | Set Memory pointer register (middle byte) |
| MEM2 | Set Memory pointer register (high byte) |
| MEMHI | Save high byte to memory at memory pointer |
| MEMLO | Save low byte to memory at memory pointer |
| COMA | Set communication block address register |
| COMD | Send value to communication block |
| REG0 | Set general purpose register 0 |
| REG1 | set general purpose register 1 |
| *Source Addresses* | |
| NULL | Get constant 0 |
| ALU0 | Get value at ALU source A accumulator |
| ALU1 | Get value at ALU source B accumulator |
| ADD | Get Arithmetical addition of ALU sources |
| ADDC | Get Arithmetical addition carry |
| ADC | Get Arithmetical addition of ALU sources and carry |
| SUB | Get Arithmetical subtraction of ALU sources |
| SUBC | Get Arithmetical subtraction carry |
| SBC | Get Arithmetical subtraction of ALU sources and carry |
| AND | Get Logical AND of ALU sources |
| OR | Get Logical OR of ALU sources |
| XOR | Get Logical XOR of ALU sources |
| SLL | Get ALU source A shifted left by source B |
| SRL | Get ALU source A shifted right by source B |
| ROL | Get rolled off value from previous SLL instance |
| ROR | Get rolled off value from previous SRL instance |

***Table 7.1.2:*** *Instructions for OISC processor.*

| Name | Description |
|---|---|
| MULLO | Get Arithmetical multiplication of ALU sources (low byte) |
| MULHI | Get Arithmetical multiplication of ALU sources (high byte) |
| DIV | Get Arithmetical division of ALU sources |
| MOD | Get Arithmetical modulus of ALU sources |
| EQ | Check if ALU source A is equal to source B |
| GT | Check if ALU source A is greater than source B |
| GE | Check if ALU source A is greater or equal to source B |
| NE | Check if ALU source A is not equal to source B |
| LT | Check if ALU source A is less than source B |
| LE | Check if ALU source A is less or equal to to source B |
| BR0 | Get Branch pointer register value (low byte) |
| BR1 | Get Branch pointer register value (high byte) |
| PC0 | Get Program counter value (low byte) |
| PC1 | Get Program counter value (high byte) |
| MEM0 | Get Memory pointer register value (low byte) |
| MEM1 | Get Memory pointer register value (middle byte) |
| MEM2 | Get Memory pointer register value (high byte) |
| MEMHI | Load high byte from memory at memory pointer |
| MEMLO | Load low byte from memory at memory pointer |
| STACK | Pop value from stack |
| ST0 | Get stack address value (low byte) |
| ST1 | Get stack address value (high byte) |
| COMA | Get communication block address register value |
| COMD | Read value from communication block |
| REG0 | Get value from general purpose register 0 |
| REG1 | Get value from general purpose register 1 |