University College London

Department of Electronic and Electrical Engineering

Performance characterisation of 8-bit RISC and OISC architectures

Author:
Mindaugas Jarmolovičius
zceemja@ucl.ac.uk
SN: 17139494

Supervisor:
Prof. Robert Killey
r.killey@ucl.ac.uk

Second Assessor:
Dr. Ed Romans
e.romans@ucl.ac.uk

A BEng Project Final Report

April 21, 2020

Contents

1 Abstract

One Instruction Set Computer (OISC), commonly implemented as Transport Triggered Architectures (TTAs) is a promising architecture that is successfully used in Application-Specific Instruction Set Processors (ASIPs) exploiting operation style parallelism, while keeping simplicity and flexibility. There is a lack of research in general purpose OISC with single data-instruction bus that could be used in lower power and performance comparable to an 8bit microcontroller using traditional Reduce Instruction Set Computer (RISC) architecture. This report describes the design, implementation and testing of two novel 8bit RISC and OISC processors, and investigates their characteristics and performance when implemented on FPGA. OISC required only a half of logic elements comparing to RISC, however it takes 71% longer to execute designed benchmark, showing that OISC would need more than one data-instruction bus to outperform RISC.

2 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 notable cases are ARM RISC architecture being used in mobile devices instead of the more popular 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 implementation can be superior in power and data throughput compared 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.

2.1 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.

2.2 Related Work

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

The 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 a single instruction, most important models are subtract and branch, MOVE and half-adder architectures [5].

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

- Using an OISC `SUBLEQ` (SUBtract and jump if Less or EQual to zero) as a coprocessor for the Microprocessor without Interlocked Pipelined Stages - Instruction Set Architecture (MIPS-ISA) processor to emulate the functionality of different classes shows desirable area/performance/power trade-offs [4].
- Comparing an OISC `SUBLEQ` multicore to a RISC achieves better performance and lower energy for streaming data processing [3].

Looking at the OISC `MOVE` type, it has been researched since early 90s. It has been shown that the OISC `MOVE` can benefit from a VLIW (very large instruction word) arrangement, classifying it as a SIMO (single instruction, multiple operation) or a SIMT (single instruction, multiple transports) architectures. The problem with all of these arrangements is that they exhibit poor or complex hardware utilization. OISC `MOVE` has been proposed as a design framework enabling a lower complexity, better hardware utilization, and a scalable performance [6]. In this framework a TTA is proposed which describes how a single instruction should transport the data. To support theoretical benefits, a `MOVE32INT` TTA has been designed [7] and proven to be a superior architecture to the RISC. Using a $1.6\mu m$ fabrication technology, RISC has achieved 20MHz clock with 20Mops/second, while `MOVE32INT` implemented using SoGs (Sea of Gates) achieved 80MHz with 320Mops/second [8].

The TTA framework was further used in other researches to implement ASIPs to solve various problems. Some relevant examples are RSA calculation [9]; matrix inversion [10]; Fast Fourier Transform (FFT) [11]; IWEP, RC4 and 3DES encryption [12]; Parallel Finite Impulse Response (FIR) filter [13]; Low-Density Parity-Check (LDPC) encoding [14]; Software Defined Radio (SDR) [15]. One of the most recent researches uses TTA architecture to solve Compressive Sensing algorithms. Research showed 9 times higher energy efficiency to that of FPGA implemented NIOS II processor, and theoretical 20 time energy efficiency compared to that of ARM Cortex-A15 [16]. In this particular research however, the used ARM Cortex-A15 with 28nm Metal Gate CMOS technology was compared to a TTA implemented on Altera Cyclone IV FPGA with 60nm Silicon Gate CMOS technology. Both processor implementations cannot be directly compared.

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

These benefits come with an expense, VLIW has bigger instruction word, therefore a bigger program size. TTA especially suffers from this due to the redundant instructions. Some proposed solutions are variable length instructions and instruction templates, which reduce program size by between 30% and 44% [17, 18]; a compression based on arithmetic coding [19]; and a method to remove redundant instructions [20]. Software is another difficulty as the compiler needs to take additional steps for the data transportation optimisations. TTA software can be easily exploited however, to embed a software pipelining and parallelism without need of the extra hardware [21].

With the proposed `MOVE` framework, hardware utilisation was shown to be improved by reducing transition activity [22], reducing interconnects was shown to save 13% of energy [23] on a small scale. A novel architecture named SynZEN also showed a further improvement by using an adaptable processing unit and a simple control logic [24].

2.3 Project contents

Section 3 will provide more detail on the motivation and project decisions based on Related Work. Section 4 explains theory and result predictions. Section 5 explains both processor design choices and how each processor part is implemented on OISC and RISC processors. It also includes assembler design and system setup. In section 6, results will be discussed, including benchmark methods and future work. Summary and conclusion of design and results can be found in section 7. Appendix in section 8 includes any other information, such as both processors' instruction sets.

3 Goals and Objectives

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

1. Study and explore computer architectures, SystemVerilog and the assembly

language.

2. Compare how well an OISC `MOVE` architecture would perform in a 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 a SIMO or a SIMT structure.

3.1 RISC Processor

The RISC architecture will be mainly based on MIPS architecture explained in [25], except that this RISC processor would have 8bit data bus, four general purpose registers and would have multiple optimisations related to 8bit limits. Some of minimalistic design ideas were also from [5].

3.2 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 a short instruction word and a SISO configuration. The main theory for building OISC architecture will be based on [5].

3.3 Design Criteria

In order to make a fair comparison between both architectures, common design criteria are set:

- 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 the consideration.

3.4 Benchmark

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

4 Theory and Analytical Bases

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

4.1 RISC Processor

In this project, the proposed RISC is mainly based on the MIPS microarchitecture [25]. Figure 4.1.1 represents a simplified diagram of a proposed RISC processor. In this architecture, program data travels from a program memory to the control block where the instruction is decoded. Then, the control 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 the instruction, control block sets ALU, register file, memory operations and how data flows from one to other. Therefore, if none of the blocks are bypassed, data can flow though every single one of these blocks, creating a long chain of combinational logic and increasing the critical path. However, this enables great flexibility allowing multiple operations to be carried out during a single step, for example load value from register to memory, while address value is immediate offset by another register value using the ALU. In

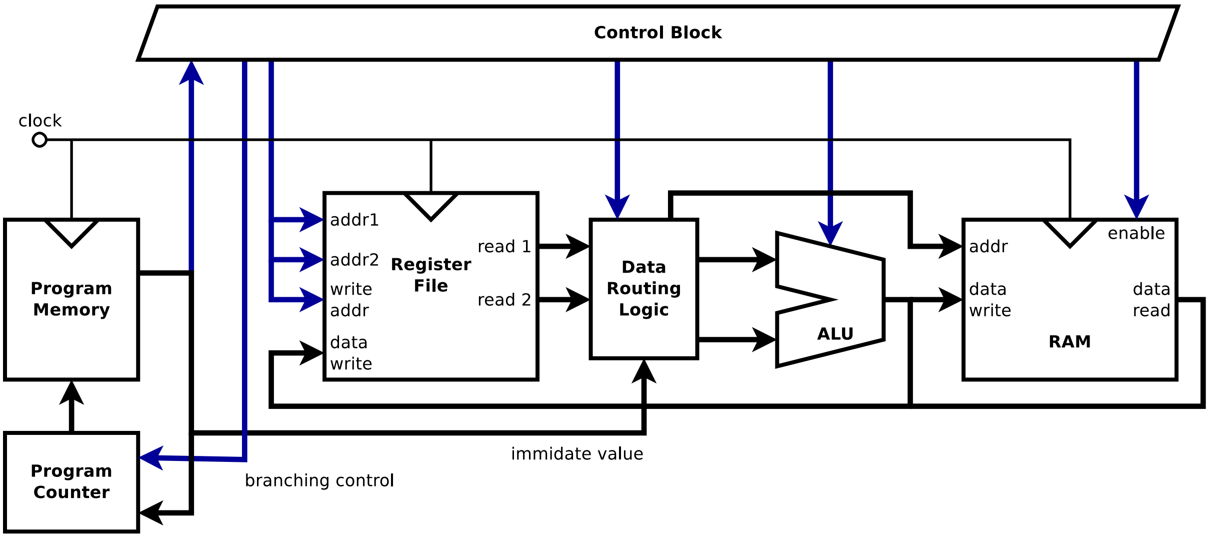

Figure 4.1.1: *Abstract diagram of proposed RISC structure*

order to increase performance of such processor, pipelining or multiple cores may be used.

4.1.1 Pipelining

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

Equation 1 shows the maximum processor cycle period T_c which depends on combinational 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's datapath block with a flip-flop. This changes critical path therefore reducing cycle period. A pipelined processor cycle period T_{cp} is represented in the equation 2. Such modification could theoretically increase clock frequency by 2 or 3 times.

Pipelining, however, introduces other design complications. Instructions that depend on each other, for example an operation $R = A + B + C$ needs to be executed in two steps, $t = A + B$ and $R = t + C$. The second step depends upon previous step result. Therefore, additional logic is required to detect such dependencies and bypass datapath stages, or stall pipelining. Furthermore, branching 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 timing hazards prevention logic which increases processor complexity and required resources.

4.1.2 Multiple cores

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

A multicore processor requires software adjustments as each processor's 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, programs most often cannot be perfectly divided into parallel tasks due to some result dependencies between each subtask. Therefore, doubling

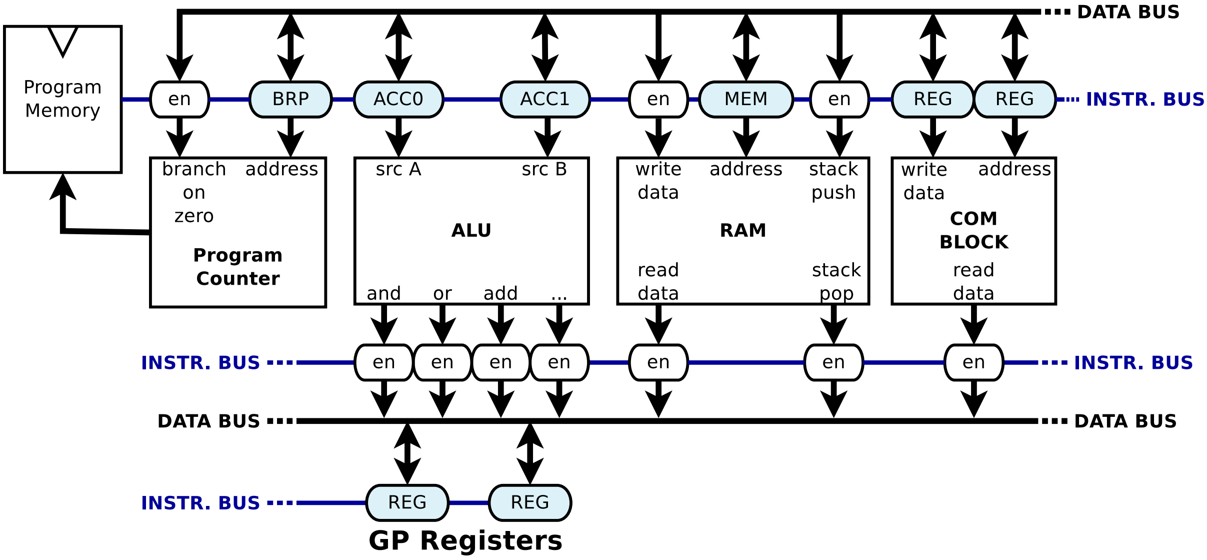

Figure 4.2.1: *Abstract diagram of proposed OISC structure*

processor core count would not likely result in doubling the performance.

4.2 OISC Processor

Figure 4.2.1 represents simplified structure of an OISC MOVE architecture. In the simplest case, the processor has a pair of buses data and instruction. An instruction bus has a source and destination address that connects two parts of processor via a data bus. This mechanism allows for the data to flow around processor. Computation is accomplished by setting accumulators at destination addresses and taking computed values from the source address. Other actions can be performed by destination node, for instance checking values for branching or sending data to memory.

4.2.1 OISC Pipelining

The maximum cycle period of such processor microarchitecture can be found in Equation 3.

$$\begin{gathered} t_{CL} = max\begin{pmatrix} t_{register}, \\ t_{ALU}, \\ t_{RAM} \end{pmatrix} \\ T_{cp} = max\begin{pmatrix} t_{en} + t_{buf}, \\ t_{pcq1} \end{pmatrix} + \\ + t_{pcq2} + t_{CL} + t_{setup} \end{gathered} \tag{3}$$

Where t_{en} is the period to check if instruction bus address match, t_{buf} is period for source buffer to output value into the data bus, t_{pcq2} is the propagation period for program memory, t_{CL} represents the longest propagation period though a logic block, t_{setup} is the setup time inside the logic block. t_{pcq1} and t_{pcq2} are clock to output delay for the sequential logic connecting source buffer and memory connecting instruction bus, respectively.

4.3 Predictions

Comparing RISC and OISC, the maximum processor cycle period of OISC is almost equivalent to the pipelined RISC, with addition of enable, buffer and additional ROM delays: $max\left(t_{en} + t_{buf}, t_{pcq1}\right)$.

Furthermore, due to the nature of the processor no additional timing hazard prevention logic is needed, making this a much simpler design. OISC t_{CL} pipelining can be also introduced to components that has high propagation delay. For instance, multiplication in an ALU could be pipelined into two stages. When setting ALU accumulators, software could be designed to retrieve multiplied result only after two cycles. This can further reduce required resources.

4.3.1 Execution time

OISC requires taking extra steps to perform basic functions. ALU, branch or memory operations need accumulator values to be set first to compute an output. A single data-instruction bus OISC therefore is expected to be slower executing the same task as RISC.

4.3.2 Instruction Space

RISC has compact instructions, as a single instruction can carry a small opcode, register addresses and optionality a multiple word immediate value. OISC has a bigger instruction overhead as it can only carry a source and destination address, meaning it can operate on only one register or immediate value in a single instruction. Therefore, it is expected the OISC will require more instruction space to perform the same function as RISC.

4.3.3 Resources

OISC does not have a control block which contains how data travels in the datapath. It also does not have multi-address register file and further routing logic within a datapath. This indicates that the OISC should require fewer logic elements to implement. This also should result in lower power consumption.

5 Technical Method

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

5.1 Machine Code

Machine code subsection talks about instructions and how they are encoded.

5.1.1 RISC Machine Code

One of the aim is to ensure instruction size is to be as minimal. An 8bit instruction width was chosen with an optional additional immediate value from one to three bytes. Immediate value operation is expanded upon in section 5.7.

The decision was made to have an instruction to composed of operation code and two operands first source & destination and second only source. This is more similar to x86 architecture rather than to MIPS. Three possible combinations of register address sizes are possible, from one to three bits in order to fit them in a single instruction. Two bits was the chosen option as it allows the addressing of four general purpose registers which is sufficient for most applications, and allowed four bits for operation code allowing up to 16 instructions.

Due to a small amount of possible operation codes and not all instructions requiring operate with two operands (for example, `JUMP` instruction does not need any operands, set immediate value only needs one operand), other two type instructions are added to the design with one and zero operands. See figure 5.1.1. This enables the 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 is listed in Table 8.1.1 in an Appendix section.

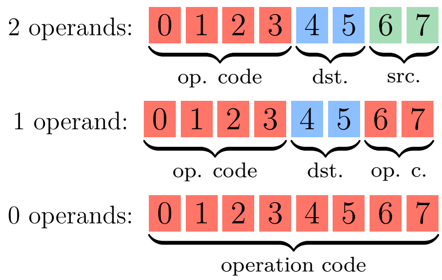

Figure 5.1.1: *RISC instructions composition. Number inside box represents bit index. Destination (dst.) bits represents of source and destination register address.*

5.1.2 OISC Machine Code

As OISC operaten on a single instruction, the composition of each instruction mainly consists of two parts source and destination. In order to allow higher instruction flexibility, an immediate flag has been added which sets source address to represent an immediate value. The composition of finalised machine code is shown in figure 5.1.2.

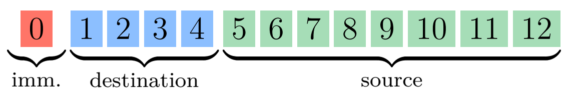

Figure 5.1.2: *OISC instruction composition. Number inside box represents bit index.*

The decision was made for source address to be 8bits, to match an immediate value and data bus width. The destination address was chosen to be as minimal as possible, leaving only four bits and 16 executable destinations. The final design has **15** destination and **41** source addresses. This is not the most space efficient design as 41 source addresses could be implemented with only six bits, not using two bits every time a non-immediate source is used.

A comprehensive list of OISC sources and destinations are given in Table 8.1.2 in an Appendix section.

5.2 Data flow

5.2.1 RISC Datapath

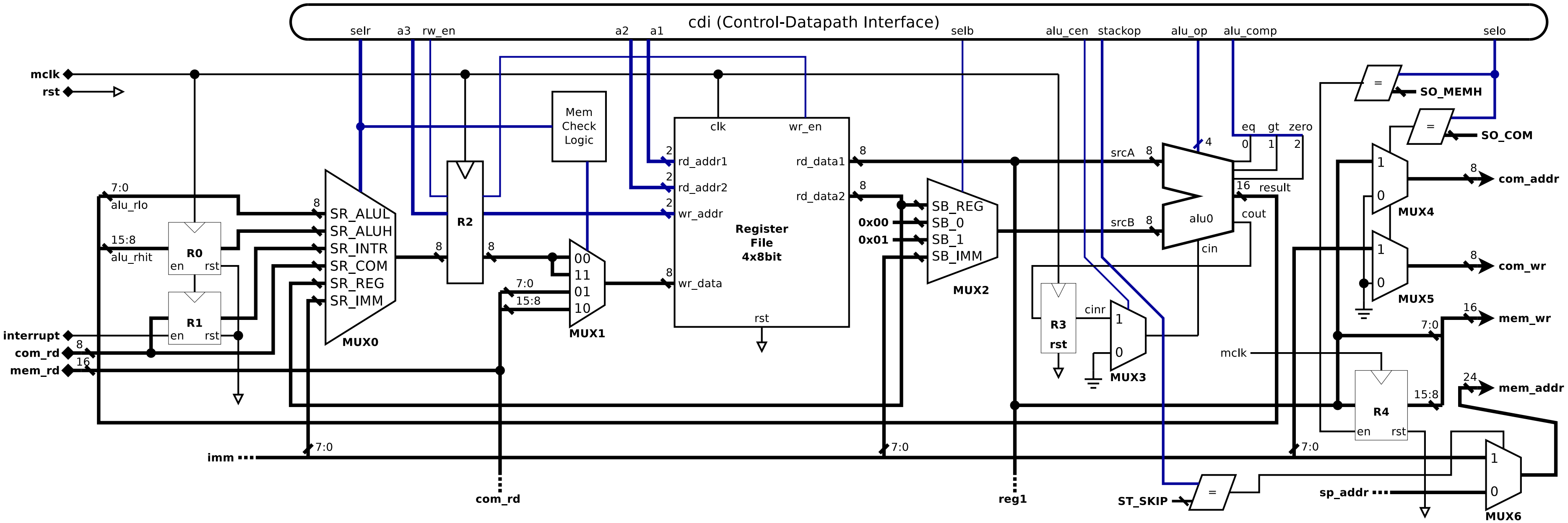

Figure 5.2.1: *Digital diagram of RISC datapath*

Figure 5.2.1 above represents a partial RISC datapath. This diagram can be extended to Program counter, Stack pointer and Immediate Override logics are shown in figures 5.4.1, 5.3.1 and 5.7.1 respectively. CDI (Control-Data Interface) is a HDL (Hardware Description Language) concept that connect datapath and control unit together. The immediate value is provided to datapath by IMO block described in section 5.7.1.

Data to register file is selected and saved with *MUX0*. This data is delayed by one cycle with *R2* to match timing that of data taken from the memory. If `LWLO` or `LWHI` instructions are executed, *MUX1* select high or low byte from memory to read. In order to compensate for timing, as value written to register file is delayed by one 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*, making it act more like latch.

MUX2 allows to override ALU source B, *R3* and *MUX3* enables control unit to enable ALU carry in bit, allowing multi-word number addition/subtraction. *MUX4* and *MUX5* allows sending data to the COM block with `COM` instruction. If any other instruction performed, then *0x00* byte for COM address and data is sent, indicating no action. Data can be stored to memory only with a `SWLO` instruction. It writes high byte value whatever is stored in *R4* register. This buffer can be written to using a `SWHI` instruction. Therefore, to change only a single byte in a particular memory location, other byte has to be fetched in advanced and used in a `SWLO` or `SWHI` instruction. *MUX6* selects memory address value from the *imm* or stack pointer.

5.2.2 OISC Datapath

OISC datapath only consists of instruction-data bus and a small circuit that connect them to logic blocks that computes the data. These logic blocks can represent ALU operation combinational logic, or any other part of a processor as shown in Figure 4.2.1.

Figure 5.2.2 represents a common destination circuit. It checks if a particular logic block destination address matches one in instruction bus, then enables latch and also sets flag that destination is used to the further logic.

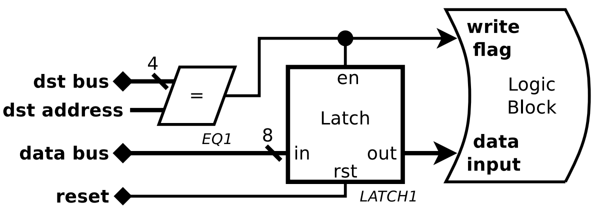

Figure 5.2.2: *OISC processor data bus to destination connection logic*

Similarly, Figure 5.2.3 represents a source circuit connecting output of a logic block. Logic block can be assumed to only contain combinational logic, therefore a register is placed at the output of it. A buffer *BUF1* is used to connect data in a register *REG1* to the data bus. This ensures that only one bus driver is present, ensuring no data collision.

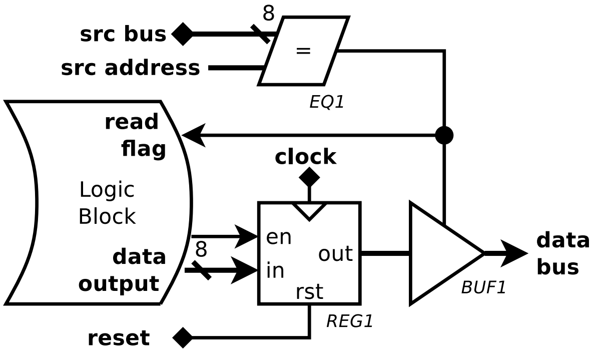

Figure 5.2.3: *OISC processor data bus to source connection logic*

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

5.2.3 OISC Datapath Implementation Problems

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

As a quick solution, latches at the destination have been replaced with a clocked register that is triggered at negative clock edge, which is opposite to source register trigger. This solution has resolved issue, however it effectively reduces the period of time that data has to propagate though logic blocks between source and destination by two.

5.3 Stack

This section describes dedicated logic for stack pointer control at both processors. The stack pointer starts from the highest memory address value and "stacks" towards lower address values. Both designs were

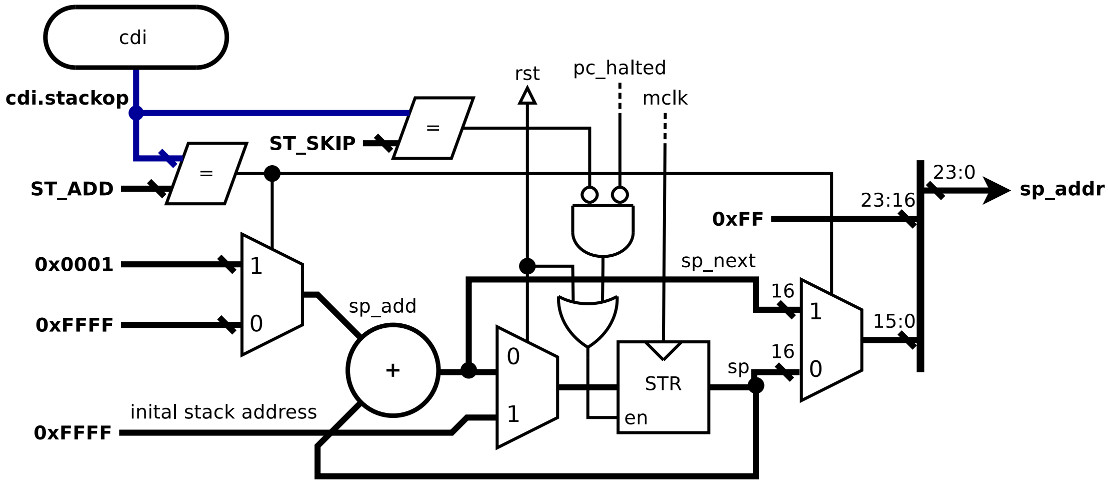

Figure 5.3.1: *Digital diagram of RISC stack pointer logic*

simplified to only operate on two byte addresses, meaning that stack pointer has a constant `FFh` value at the least significant byte.

5.3.1 RISC Stack

The RISC processor implements the stack pointer that is used in `PUSH`, `POP`, `CALL` and `RET` instructions. Figure 5.3.1 represents the logic diagram for stack pointer. This circuit also supports *pc_halted* signal from the program counter to prevent the stack pointer from being added by 1 twice during the `RET` instruction.

One of the problems with the current stack pointer implementation is 8bit data stored in 16bit memory address, wasting a byte, except when storing the program pointer with `CALL` instruction. This can be improved by adding a high byte register, however then it would cause complications 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 the current implementation this does not affect processor comparison, it only increases stack size in memory.

5.3.2 OISC Stack

The stack pointer circuit in OISC is very similar to RISC. 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 5.3.2.

Logic diagram of stack control unique to OISC processor is shown in Figure 5.3.3. Push and pop flags are taken from the 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 clock cycle. This ensures that once stack value is popped, lower stack value is written into the cache during next the clock cycle. Note that there is an issue with this design, stack source or destination instruction cannot be used together with other stack or memory operations as it creates a collision accessing system memory at the same time. This collision can be avoided with software however.

5.4 Program Counters

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

5.4.1 RISC Program Counter

Figure 5.4.1 represents the digital diagram for a 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 one byte to allow `BEQ`, `BGT`, `BGE` instructions as first immediate byte used as ALU source B); it can jump to an interrupt address; it produces a *pc_halted* signal when memory is read (`RET` instruction takes two cycles, because cycle one fetches the address from stack and second cycle fetches the instruction from the instruction memory).

5.4.2 OISC Program Counter

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

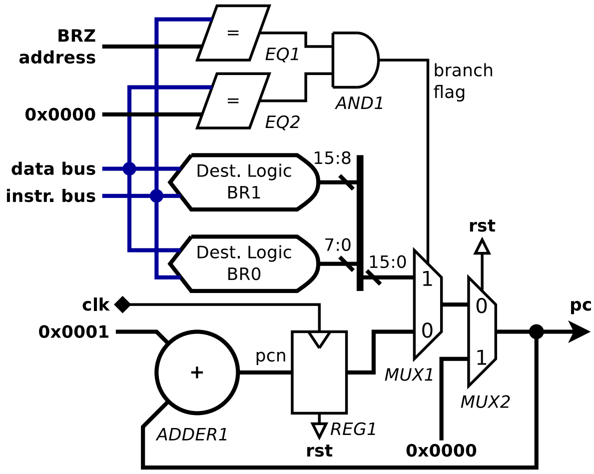

Figure 5.4.2: *Digital diagram of OISC program counter*

Looking at Figure 5.4.2 bottom, the basic 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 an instruction with the `BRZ` destination is executed, comparator *EQ2* checks if the data bus value is equal to zero. If this condi-

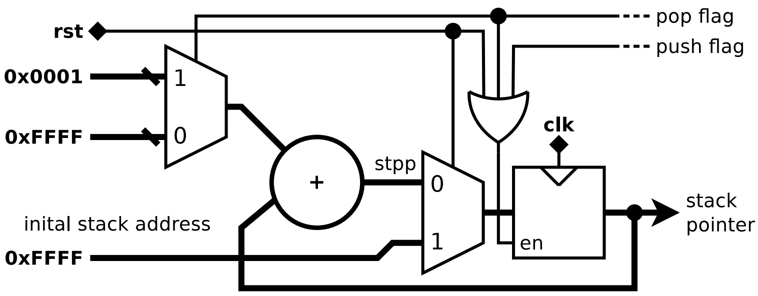

Figure 5.3.2: *Digital diagram of OISC stack pointer logic*

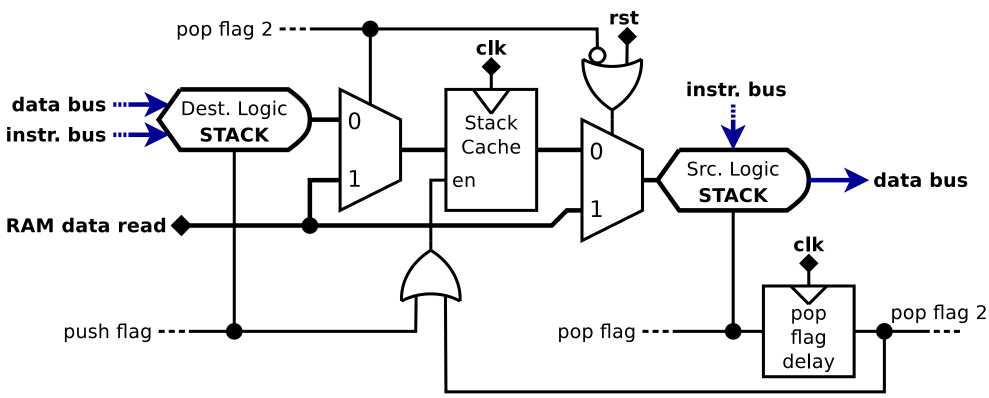

Figure 5.3.3: *Digital diagram of OISC stack control logic*

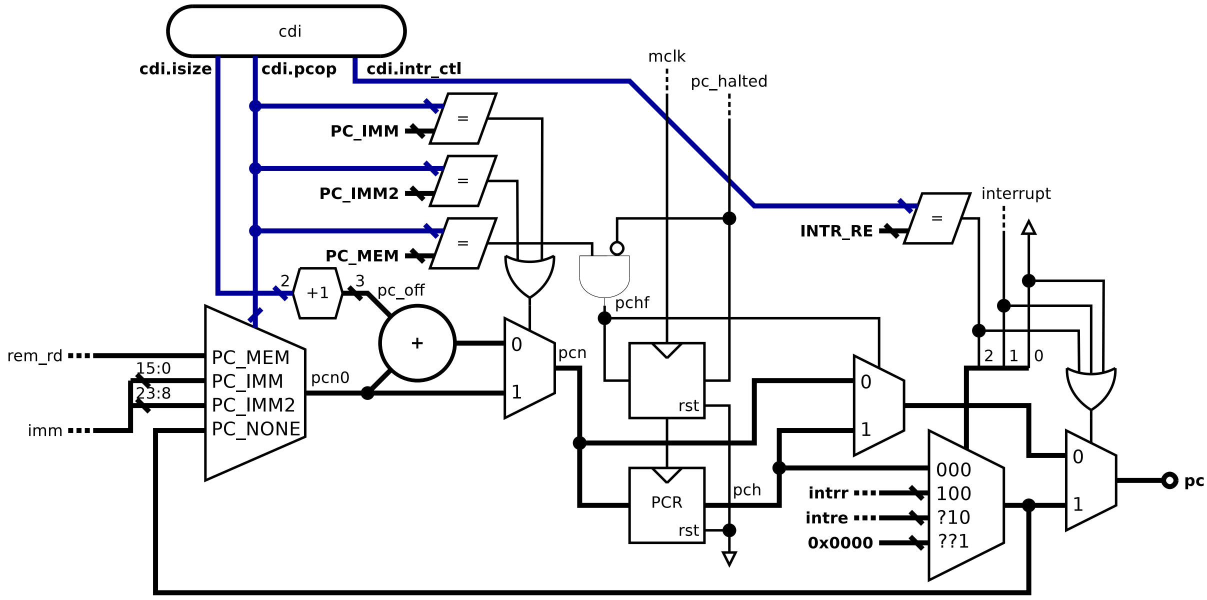

Figure 5.4.1: *Digital diagram of RISC program counter*

tion is met, it 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. 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
```

5.5 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 as described in Table 5.5.1.

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 modulo

Table 5.5.1: *Supported ALU commands for both processors*

5.5.1 OISC ALU

Due to the structure of the 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 a register so that it would be immediately available in a next clock cycle as a source data, as explained in OISC Datapath Section. Figure 5.5.1 represents a logic diagram of ALU with only an addition and multiplication operations present. Note that the output of *EQ3* is connected to enable of *REG3*, enabling output of carry to be only read after `ADD` source is requested. Similar configuration is also used for `SUB`, `ROL` and `ROR` operations.

5.5.2 RISC ALU

The RISC processor has very similar structure to OISC, however with two exceptions. Inputs to ALU comes from datapath data router logic. Output buffers are replaced by one multiplexer that selects a single output from all ALU operations. Another 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.

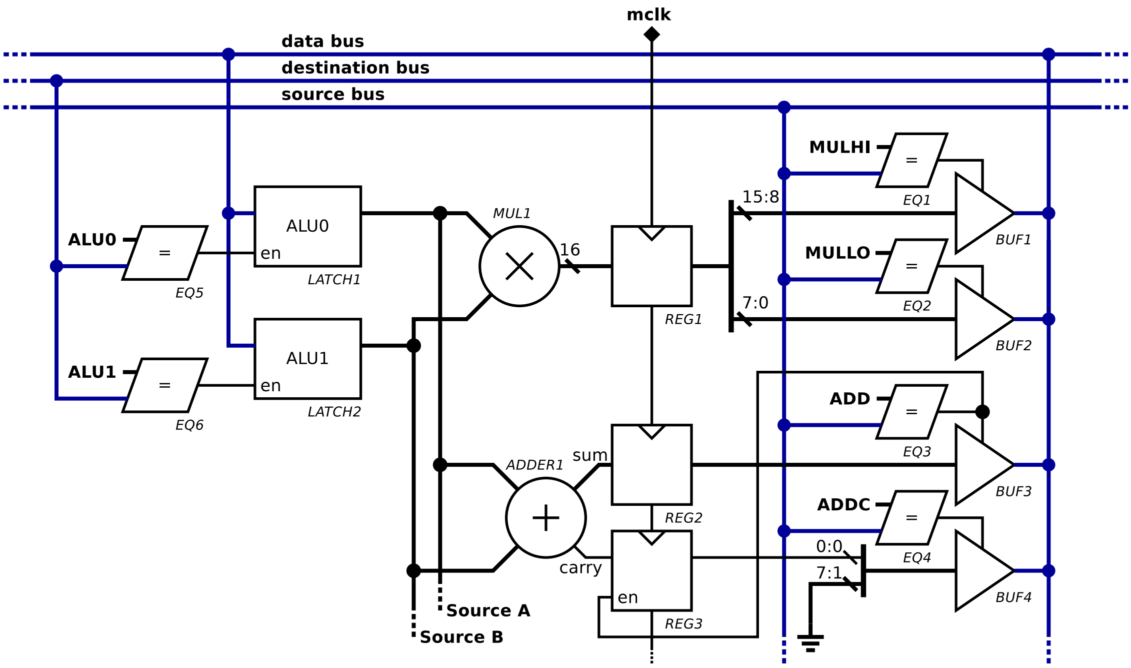

Figure 5.5.1: *Digital diagram of OISC partial ALU logic*

5.6 Program Memory

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

5.6.1 RISC Program Memory

In order to allow a dynamic instruction size from one to four bytes, a special memory arrangement is made. A system was required to access a word (8bits) from memory and next three words, meaning that memory cannot simply be packed to four word segments. To achieve desired functionality, 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 value four, which is done 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
```

5.6.2 OISC Program Memory

OISC instructions are fixed 13 bits, this non-standard memory word size causes some difficulties. To implement ROM in FPGA, Altera Cyclone IV M9K configurable memory blocks were used. Each blocks has 9kB of memory, each set as 1024x9bit configuration. Combining three these 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 5.6.2. Circuit extracting each instruction is fairly simple, shown in figure 5.6.3.

5.7 Instruction decoding

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

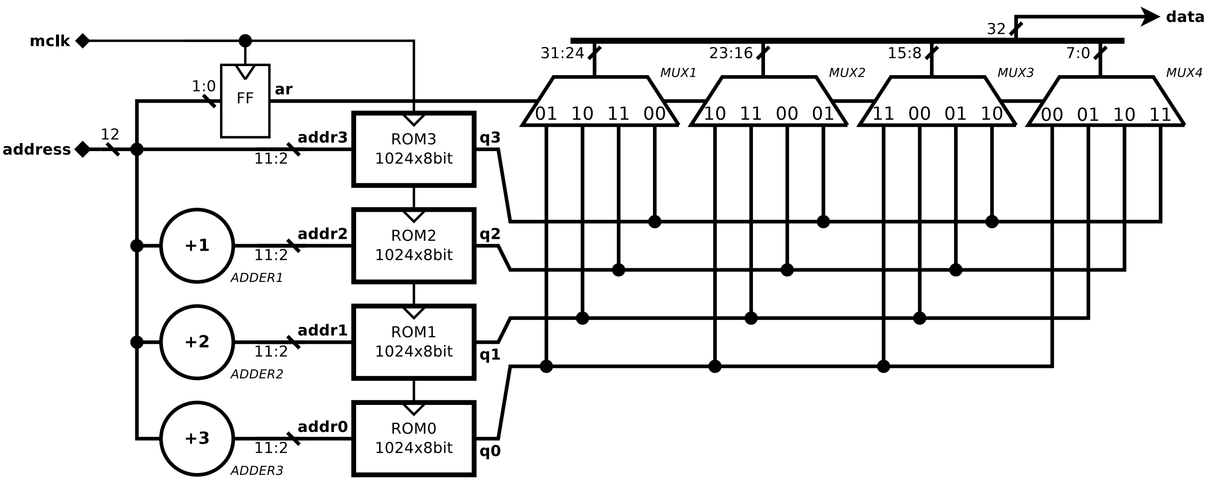

Figure 5.6.1: *Digital diagram of RISC sliced ROM memory logic*

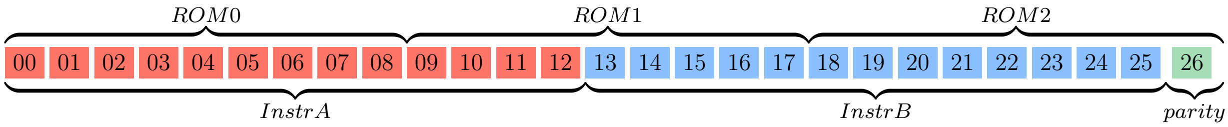

Figure 5.6.2: *OISC three memory words composition. Number inside box represents bit index.*

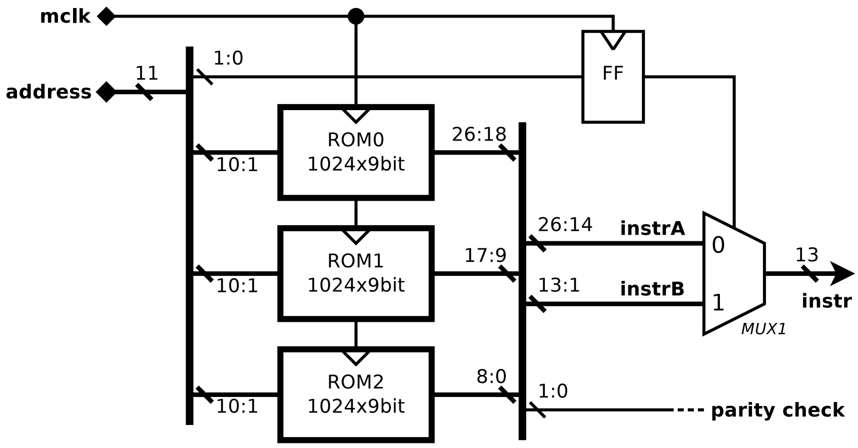

Figure 5.6.3: *Digital diagram of OISC instruction ROM logic*

5.7.1 RISC IMO

Already described in previous section 5.6, instruction from the memory comes as four bytes. The least significant byte is sent to control block, other three bytes are sent to the immediate override block (IMO) shown in figure 5.7.1. These three bytes are labelled as **immr**.

The IMO block is a solution to change the immediate sent further to the processor with a value from register. This enables dynamically calculated memory pointers, branches that are dependent on a 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 a signal is `0h`, this block is transparent connecting **immr** directly to **imm**. When any of `CI` instructions executed, one of IMO register is overwritten by *reg1* value from the 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 overwritten IMO bytes from *pc_off* signal (program counter offset that is dependent on i-size value) at the program counter, therefore effectively saving instruction memory space. This solution however, would introduce a complication with the assembler as additional checks would need to be done during assembly compilation to check if IMO instruction are used.

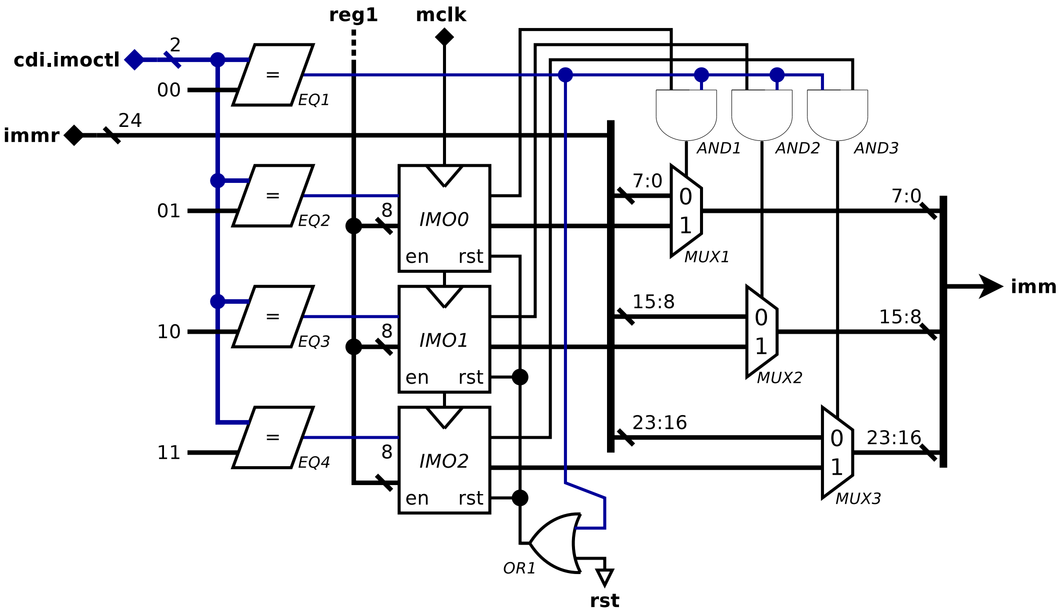

Figure 5.7.1: *Digital diagram of RISC immediate override system*

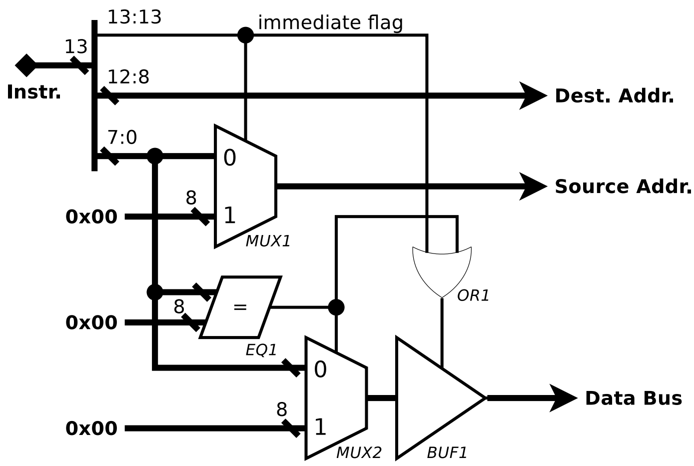

Figure 5.7.2: *Digital diagram of OISC instruction decoder*

5.7.2 OISC Instruction decoding

OISC immediate value is set in instruction decoder shown in figure 5.7.2. Decoder operation is simple - instruction machine code is split into three parts as described in 5.1.2. If instruction source address value is `00h`, it connects data bus with constant zero value via *MUX2*. If immediate flag is set, source address value is set to `00h` in order to make sure no other buffer source connects to data bus. Instruction source address then is connected to databus via *MUX2* and *BUF1*.

5.8 Assembly

There are two steps between the assembly code and its execution on a processor. First, it needs to be converted into a binary machine code. Secondly, binary data needs to be sliced to different parts described in section 5.6. These slices also need to be converted into appropriate formats, different for simulation, HDL synthesis and direct memory flashing.

A universal assembler was implemented using python for both processors. The flowchart in figure 5.8.1 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 a global label dictionary. Macros are a chunk of assembly code and are 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 is used to store values such as strings and uninitialised data structures. These values are accessed with labels which correspond to RAM memory location.

Section `.text` code is processed line by line. Each line may have label and an instruction or macro name following with argument values. If line contains a label, it is stored into global label dictionary with current line program address as a value. If line has a macro, line is replaces by macro code. Otherwise, instruction name is decoded and stored in an instruction list with original arguments.

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 a raw binary. Similarly, `.data` section labels also replaced and it is saved as binary data.

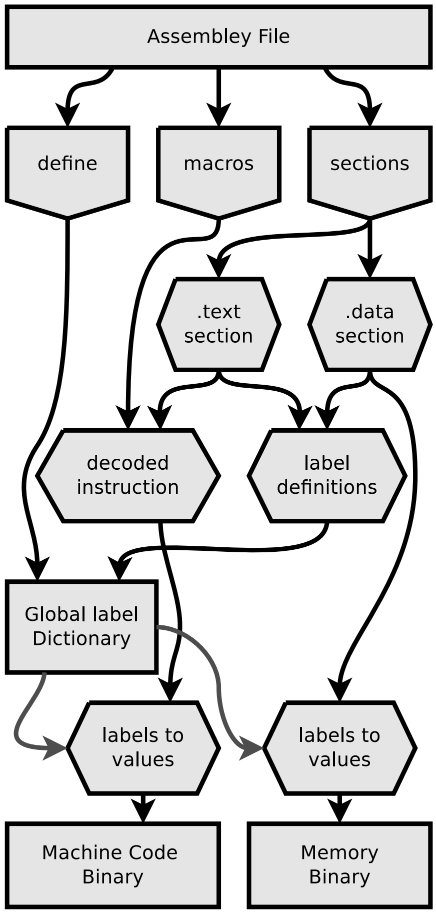

Figure 5.8.1: *Flow chart of assember converting assembly code into machine code and memory binary.*

5.9 System setup

This section will describe how the system is set up.

Processors are implemented on Terasic DE0-Nano board that use Altera Cyclone IV, EP4CE22P17C6 FPGA, which is manufactured using 60nm fabrication technology. The FPGA has embedded memory structure consisting of M9K memory blocks columns mentioned in Subsection 5.6.2. These memory structures were used to implement processors RAM and ROM memories. Board also has 32MB SDRAM chip, which initially was intended to be used. This set design criteria to have 24bit address space. However, M9K memory was used instead for flexibility and simplicity.

Thas FPGA has also an embedded phase-locked loop (PLL) stucture that is used to change 50MHz input that is generated by on-board crystal to other frequencies.

DE0-Nano board has an integrated JTAG port that is used to upload synthesised code and control additional debugging tools. Quartus has a "Signal Tap Logic Analyzer" tool that allow setup probes and sources within FPGA logic and control them via JTAG. Another "In-System Memory Content Editor" tool allows read and modify M9K memory which enables quick machine code uploading to the processor on FPGA, without need to resynthesise HDL code. This also allow reading RAM content enabling easier program debugging.

All Quartus functions can be accessed via TCL script. This lead to constructing Makefile which allow quick build operations. Quratus signal and memory tools were used to write a small program with Python and Curses library to read and change internal processor state which allowed easy debugging while writing the programs.

6 Results and Analysis

6.1 FPGA logic component composition

This subsection describes the testing and results which finds how much FPGA logic components each processor takes and what is composition of each part.

Testing was performed with Quartus synthesis tool by recording flow summary report data. This report includes synthesised design metrics including total logic elements, registers, memory bits and other FPGA resources. In this testing, only parameters that were recorded are logic elements and registers. Number of resources was found by synthesising full processor, then commenting relevant parts of code, resynthesising and viewing changes in the report. Such method may not be the most accurate, because during HDL synthesis, circuit is optimised as unused connections removed. This means that more of the logic than commented may be not synthesised.

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

1. **Common** - processor auxiliary logic that is used by both processors. It includes the communication block with UART, RAM and PLL (Phase-Locked Loop, for master clock generation).
2. **ALU** - as described in section 5.5, both processors have slightly different implementation of ALU.
3. **Memory** - the processors memory management, including stack.
4. **Other** - reminding processor logic that was not analysed.

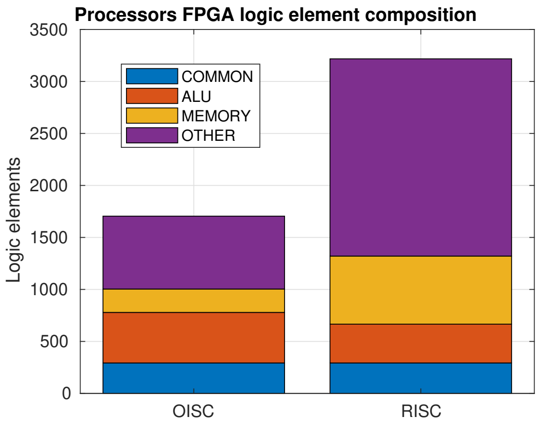

Figure 6.1.1: *Bar graph of FPGA logic components taken by each processor.*

The test results are shown in figures 6.1.1 and 6.1.2. The 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.

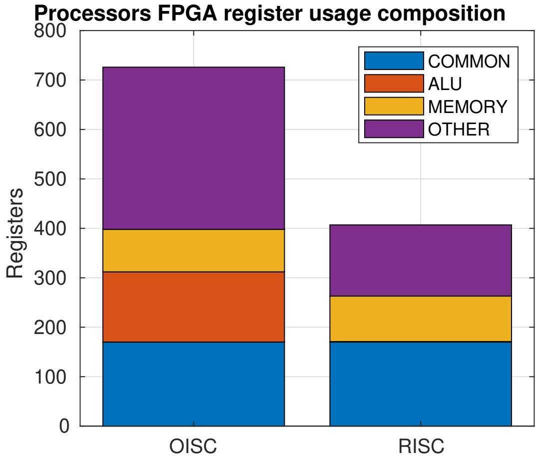

Figure 6.1.2: *Bar graph of FPGA register resources taken by each processor.*

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

Looking at the composition, OISC ALU takes 30.2% more logic gates. Figure 6.1.2 shows a high number of OISC ALU registers. This concludes that higher resource usage in OISC ALU code must be source and destination logic.

Memory logic element composition of OISC is only 34.4% of RISC's and 7% lower for register resources, comparing to RISC. This indicates 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 5.6, 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 source and destination logic requires more registers, whereas RISC uses combinational logic in the control block in order to control the same data in the datapath.

The much higher number of logic components in RISC can be also explained more complicated register file, ROM memory logic and program counter. All of these components have some additional logic for timing correction or other extra functionality required by these block integration into a datapath.

6.2 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 1.020Ω was connected in series to calculate current. Supply voltage and voltage across shut resistor were measured using an oscilloscope with a data sampling feature. Three tests have been performed with different processor configurations. Between each test a period of about 5 minutes was given for FPGA to reach steady state.

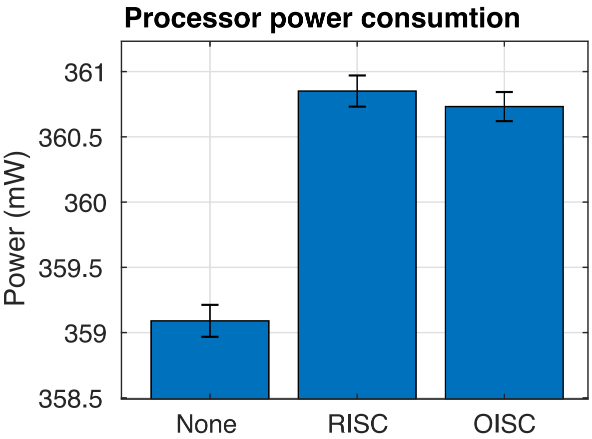

Figure 6.2.1: *Measured power of processors when implemented on FPGA, running 16bit multiplication function in loop. None indicates auxiliary-only power.*

Figure 6.2.1 represents power results. First configuration is "None" or auxiliary-only power, which includes the 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 processor implementations on FPGA, each running a multiplication program in a loop. These values also include auxiliary power plus processor power, which means that the processor itself takes relatively small amount comparing to auxiliary power, about 0.5%. Result shows that OISC require 0.4%, which including noise is almost insignificant result.

During this test clock frequency of 1MHz was used. Due to equipment unavailability, any further tests were not carried out to investigate power consumption at different frequencies. Due to constant noise, running at higher frequency may result in significant difference between processors.

6.2.1 Activity Factor

An activity factor could be also found using Equation 4 where P is power, C_{total} is total gate capacitance and V_{DD} is voltage supplied to the transistors.

$$\alpha = \frac{P}{C_{total} \cdot f \cdot V_{DD}^2} \tag{4}$$

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 [11, 15, 22, 23].

6.3 Benchmark Programs

A number of programs have been written to test both processors. These involve simple functions that could be commonly used in a 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 [26] to calculate primer number, operates on 16bit numbers and utilise 16bit multiplication and modulo functions.

6.3.1 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 6.3.1: *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 6.3.2: *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 6.3.3: *OISC processor instruction source groups used in instruction composition test*

Each function was executed on a simulated processor, program counter and instruction were recorded into file at every cycle. File recording was accomplished with SytemVerilog test bench. Start of a recording was triggered when program counter matched `.start` location and stopped when it matched `.done` location. Code shown in Listing 3 enabled both locations to be static and not depend on test function that was executed.

Listing 3: *Assembly frame for executring tests*

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

Each recorded file with function composition was then 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 the results in Figure 6.3.1, few key differences can be seen. Across every test, OISC has significantly more *BRANCH* destination and *MOVE* source groups. *BRANCH* group can be explained by emulated `CALL`, `RET` and `JUMP` instruction explained in section 5.4.2. High number of *MOVE* source group instructions may be explained by using the immediate values as a separate source, where RISC uses instructions that can integrate immediate as extra word, such as instruction `ADDI`. In most cases *ALU* group instructions are also higher than for OISC comparing to RISC. This shows a lower OISC ALU efficiency,

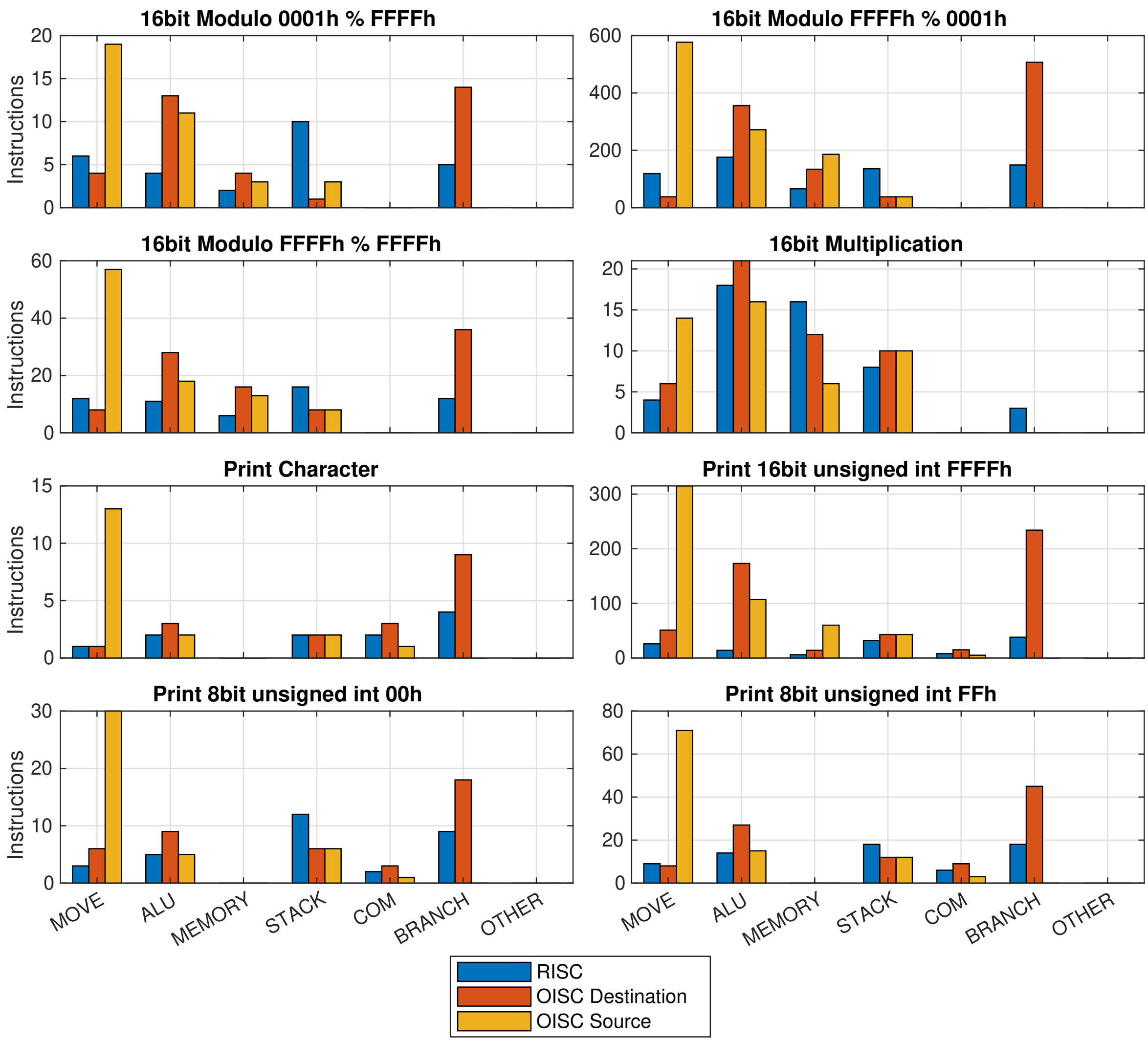

Figure 6.3.1: *Graph of instruction composition for every benchmark program.*

mostly due to a need to move data into the separate accumulators.

6.3.2 Performance

This subsection investigates time and clock cycles to run benchmark programs. The simulation was performed 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 input arguments. This allows to see the worst and the best case scenarios as algorithms length depend on inputs. This is not the case for 16bit multiplication as its implementation has no branching, therefore no execution time dependence on the inputs.

Results are shown in Figure 6.3.2. In most of the cases, OISC requires around 55-67% more instructions, with some exceptions.

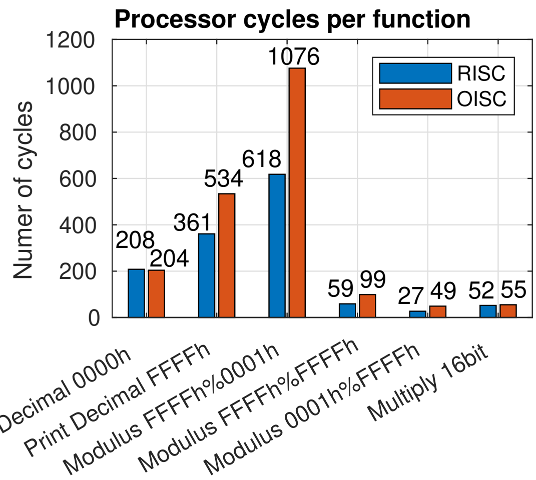

Figure 6.3.2: *Simulated results of cycles that taken to perform function.*

Another set of benchmarks have been performed and on both processors once they been implemented on the FPGA. Time taken to perform each set has been recorded. This has been done via UART connection, a single character was sent to indicate the start and the stop of a benchmark. In order to void a slight timing variation due low baud rate of UART or system kernel scheduler unpredictability to process UART input, each benchmark was performed with many iterations. Figure 6.3.3 represents results.

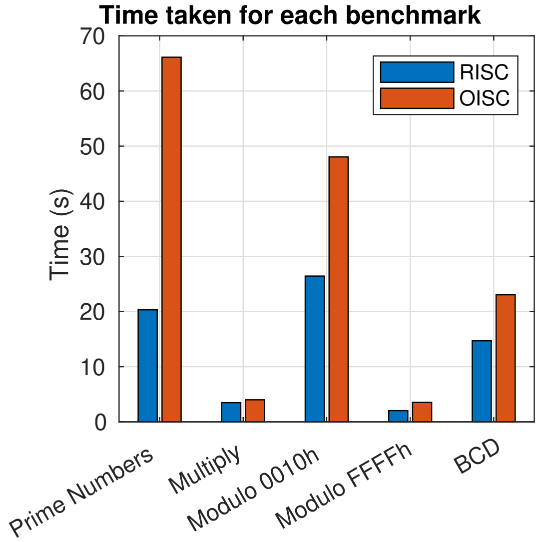

Figure 6.3.3: *Time taken to perform each benchmark on FPGA at 1MHz clock.*

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 65536.
- **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.

6.3.3 Program space

Data collected from previous instruction composition results were also used to find effective program size. Effective program size only includes instruction that been executed depending on argument, meaning that it does not fully represent complete function. A specific input to a function might cause branching and avoiding 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 irrelevant.

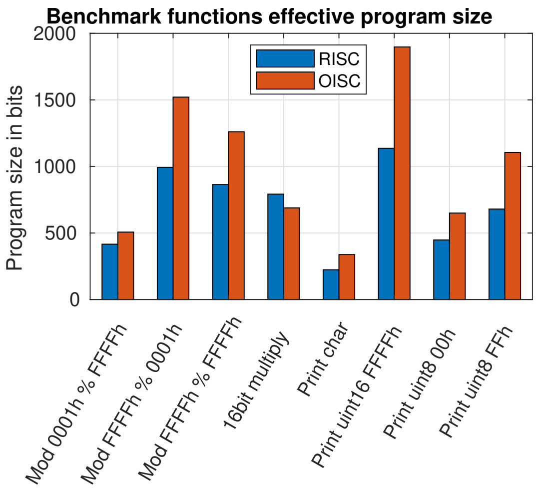

Figure 6.3.4: *Bar graph showing effective size in bits each benchmark function is taking in program memeory.*

Figure 6.3.4 represents an effective program size for each test function. On average, OISC instructions take 41.71% more space which is to be expected.

6.4 Maximum clock frequency

In order to find maximum clock frequency, processors were loaded with basic print string function and 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 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 $D = \left\lfloor \frac{f_{mclk}}{4 f_{baud}} \right\rfloor$. UART rate was set to 9600 baud. UART module itself has four times oversample.

Frequency was changed in 5MHz increments.

The 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 6.4.1: *Theoretical and actual maximum frequencies of both processors.*

Theoretical and actual results show unexpected results shown in Table 6.4.1, RISC operated at about 40% higher maximum frequency than OISC.

As explained in Subsection 5.2.3, OISC logic blocks takes approximately half the 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 twice as high, 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.

6.5 Future work

RISC has more sophisticated logic for various processor components. It is expected to see RISC having better results due to its higher optimisation. OISC should be implemented with multiple data & instruction buses. This could be performed with minimal corrections on hardware, however would require many changes in assembly programs. Instruction composition results show that OISC takes more instructions to store values in accumulators, which could benefit from multi-bus parallelisation. Adding a single additional bus should halve benchmark times, which would produce more comparable to RISC. In addition, multi-bus OISC can perform truly parallel programs assuming it has enough processor resources to perform operations (for example operate different ALU operations at the same time). This potentially would be dominant feature over RISC in time-sensitive programs, GPIO (General Purpose Input/Output) and interrupt handling.

Additional buses would not greatly increase processor logic element size, especially when using interconnect optimisation techniques [22, 23]. Matching processor complexity should also allow more fair and direct comparison specifically between two architectures.

A number of other improvements and future research are proposed:

1. Perform more tests on power analysis with different frequencies. Find the activity factor described in Subsection 6.2.1.
2. Further investigate maximum frequency. Try to resolve OISC timing issue and repeat maximum frequency test. This would allow to prove or disprove theorised higher frequency capabilities for OISC.
3. Design a higher level language compiler such as BASIC or C. This would allow performing more complicated programs which would more closely relate to microcontroller operations. However, OISC compiler would need extra optimisation layer to efficiently organise instructions.
4. Compare proposed processor designs with other commercially available 8-bit processors such as Atmel AVR microcontrollers, Motorola 6800 family and Microchip PIC.

7 Conclusion

In this paper, two novel RISC and OISC-MOVE architectures were designed and implemented on a FPGA. Logic element requirements, power consumption, maximum frequency were tested. Benchmark programs execution times were used to compare these two processors and investigate OISC-MOVE advantages. It was shown that power consumption differences are insignificant, RISC managed to reach 40% higher maximum frequency at 75-70MHz, however due to a timing design issue with OISC. OISC required 51.7% less logic elements to implement on FPGA. Benchmarks showed that OISC took 71% longer to execute on average while requiring 41.71% more instruction space.

This project has sucessfully covered its goals in studying architectures and investigating an alternative OISC implementation. Results show that proposed implementation of OISC-MOVE may be only suitable for microprocessor application with very strict logic element limit.

RISC processor has been shown to be superior in tests, however it has more optimised implementation. Further research is needed to investigate OISC-MOVE performance with multiple data and instruction buses to match RISC complexity.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] François Morain. “Atkin’s Test: News from the Front”. In: 1989, pp. 626–635. DOI: 10.1007/3-540-46885-4_59.

8 Appendix

8.1 Processor instruction set tables

Table 8.1.1: *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. modulo)	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 8.1.1: *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 8.1.2: *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 8.1.2: *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 modulo 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