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

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

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

A BEng Project Final Report

March 27, 2020

1 Abstract

2 Introduction

3 Goals and Objective

4 Theory and Analytical Bases

Decided design criteria:

- Minimal instruction size
- Minimalistic design

5 Technical Method

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

5.1 Machine Code

5.1.1 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 5.4.

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 5.1.1. 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 **??** in Appendix section.

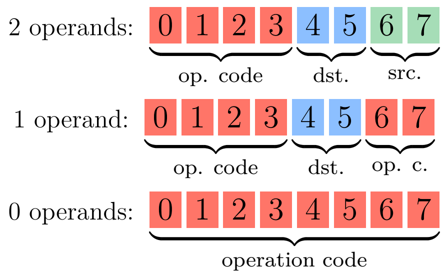

Figure 5.1.1: *RISC instructions composition. Number inside box represents bit index. Destination (dst.) bits represents of source and destination register address.*

5.1.2 OISC

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 5.1.2.

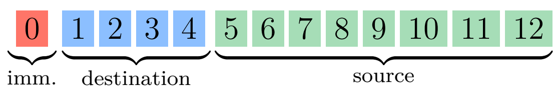

Figure 5.1.2: *OISC instruction composition. Number inside box represents bit index.*

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 2 in Appendix section.

5.2 Arithmetic Logic Unit

TO BE ADDED

5.3 Memory

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

5.3.1 RISC

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 1.

Listing 1: 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
```

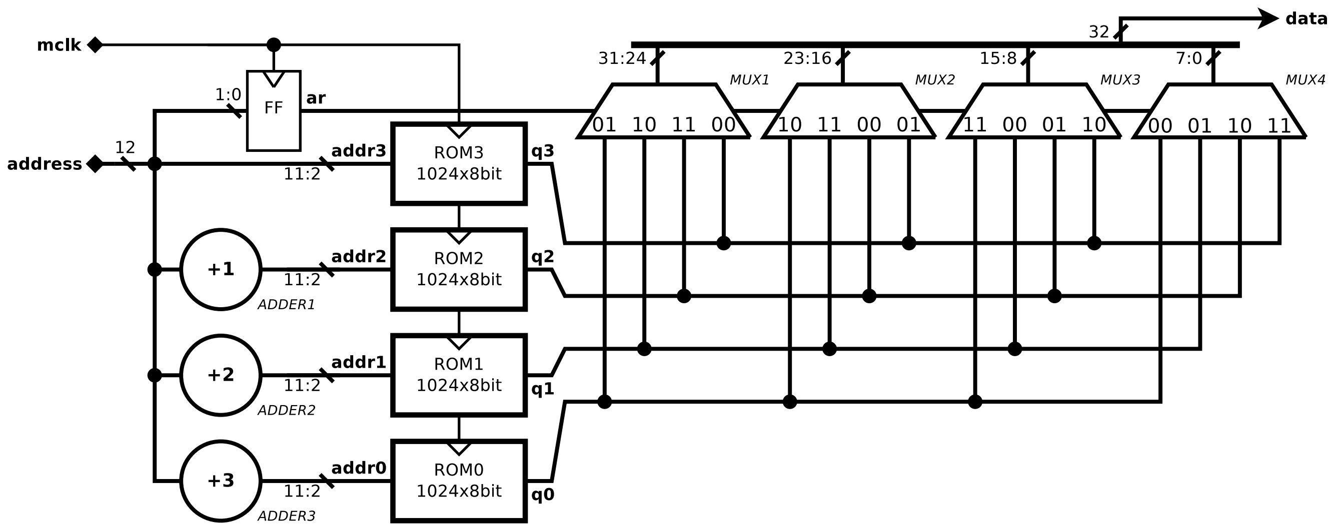

Figure 5.3.1: Digital diagram of RISC sliced ROM memory logic

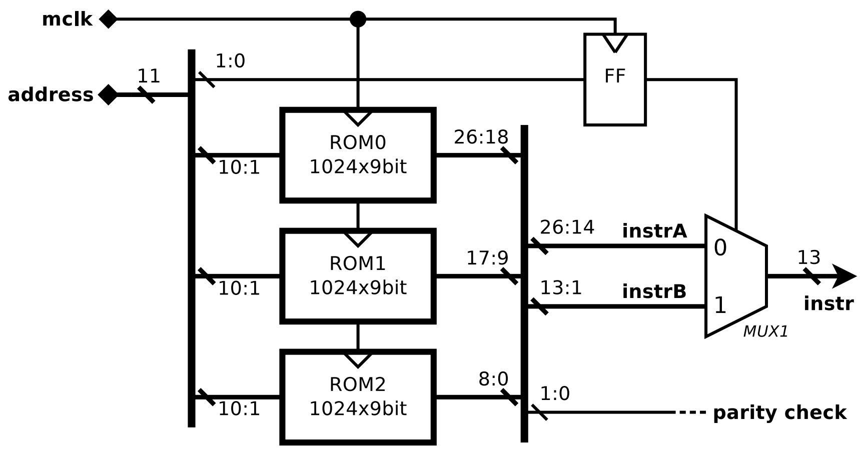

Figure 5.3.2: Digital diagram of OISC instruction ROM logic

5.3.2 OISC

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. Circuit extracting each instruction is fairly simple, shown in figure 5.3.2.

5.4 Instruction decoding

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

5.4.1 RISC

Already described in previous section 5.3, 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 5.4.1. 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.

5.4.2 OISC

OISC immediate value is set in instruction decoder shown in figure 5.4.2. Decoder operation is simple - instruction machine code is split into three parts as described in 5.1.2. 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*.

6 Results and Analysis

6.1 Benchmark Programs

6.1.1 Number of instructions

6.1.2 Instruction composition

Function composition was executed with following code:

Listing 2: RISC assembly frame for executring tests

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

Listing 3: 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
```

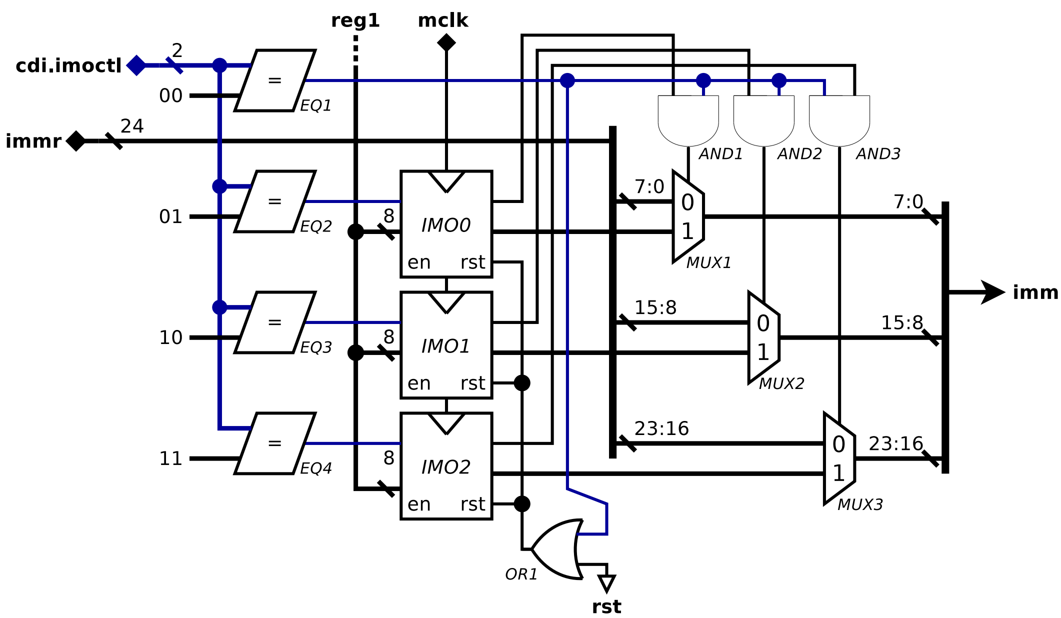

Figure 5.4.1: Digital diagram of RISC immediate override system

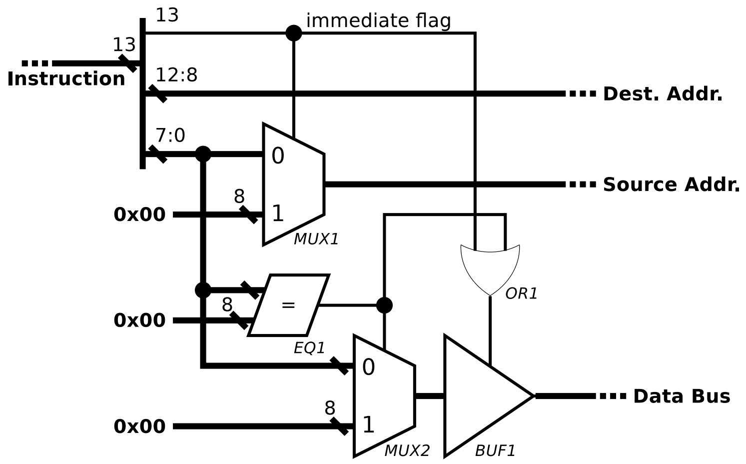

Figure 5.4.2: Digital diagram of OISC instruction decoder

6.2 Maximum clock frequency

6.3

7 Conclusion

8 References

9 Appendix

9.1 Processor instruction set tables

Table 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 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 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 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