University College London

Department of Electronic and Electrical Engineering

Project Progress Report No. 4

Author:
Minduagas Jarmolovičius
zceemja@ucl.ac.uk

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

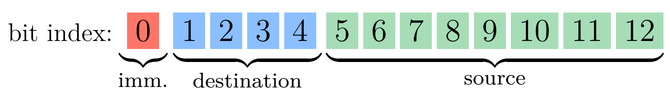
February 16, 2020

1 Progress

1.1 Completed OISC implementation

In this section brief OISC implementation will be described.

OISC machine code is stored in 13bits instructions that compose 1 bit that indicates if source is immediate value, 4 bits for destination address, 8 bits for source address or immediate value. This been shown in diagram below:



Such design been chosen to match RISC's small instructions design. Tables 1 and 2 describes currently implemented OISC source and destination addresses.

Name	Description
ALU0	Store value in ALU input A register
ALU1	Store value in ALU input B register
BRPT0	Store value in branch pointer lower byte register
BRPT1	Store value in branch pointer higher byte register
BRZ	Set program counter to branch pointer if value is 0x00
STACK	Push value to stack
MEMPT0	Store value in memory pointer lower byte register
MEMPT1	Store value in memory pointer middle byte register
MEMPT2	Store value in memory pointer higher byte register
MEMHI	Store higher byte in memory at address stored in memory pointer
MEMLO	Store lower byte in memory at address stored in memory pointer
COMA	Store value to communication block address register
COMD	Send instruction to communication block with address specified in COMA and data as a source
REG0	Store value in general purpose register 0
REG1	Store value in general purpose register 1

Table 1: Destination registers for OISC processor.

Name	Description
NULL	Always returns 0x00
ALU0	Returns value stored in ALU input A register
ALU1	Returns value stored in ALU input B register
ADD	Returns ALU input A added with input B
ADC	Returns ALU input A added with input B with carry from previous time ADD or ADC been used
ADDC	Returns carry bit from previous time ADD or ADC was used
SUB	Returns ALU input A subtracted input B
SBC	Returns ALU input A subtracted input B with carry from previous time SUB or SBC been used
SUBC	Returns carry bit from previous time SUB or SBC source was use

Name	Description
AND	Returns ALU input A AND gated with input B
OR	Returns ALU input A OR gated with input B
XOR	Returns ALU input A XOR gated with input B
SLL	Returns ALU input A shifted left by input B (only least significant 3bits)
SRL	Returns ALU input A shifted right by input B (only least significant 3bits)
ROL	Returns rolled off "reminder" from last time SLL was used
ROR	Returns rolled off "reminder" from last time SRL was used
EQ	Returns 0x01 if ALU input A is equals to input B, otherwise returns 0x00
NE	Returns 0x01 if ALU input A is not equals to input B, otherwise returns 0x00
LT	Returns 0x01 if ALU input A is less than input B, otherwise returns 0x00
GT	Returns 0x01 if ALU input A is greater than input B, otherwise returns 0x00
LE	Returns 0x01 if ALU input A is less or equal to input B, otherwise returns 0x00
GE	Returns 0x01 if ALU input A is greater or equal to input B, otherwise returns 0x00
MULLO	Returns ALU input A multiplied with input B, lower byte
MULHI	Returns ALU input A multiplied with input B, higher byte
DIV	Returns ALU input A divided by input B
MOD	Returns ALU input A modulus of input B
BRPT0	Returns value stored in branch pointer lower byte register
BRPT1	Returns value stored in branch pointer higher byte register
PC0	Returns program counter + 1, lower byte
PC1	Returns program counter + 1, higher byte
MEMPT0	Returns value stored in memory pointer lower byte register
MEMPT1	Returns value stored in memory pointer middle byte register
MEMPT2	Returns value stored in memory pointer higher byte register
MEMLO	Returns lower byte from memory at memory pointer address
MEMHI	Returns higher byte from memory at memory pointer address
STACK	Pop value from stack
STPT0	Returns stack pointer lower byte
STPT1	Returns stack pointer higher byte
COMA	Returns value stored in communication block address register
COMD	Requests and returns value from communication block
REG0	Returns value stored in general purpose register 0
REG1	Returns value stored in general purpose register 1

Table 2: Source registers for OISC processor.

1.2 OISC Benchmark

Following functions have written in assembly for OISC:

- `print_char`: writes byte to terminal.
- `read_char`: reads byte from terminal.
- `print_bin`: prints 8bit value as binary to terminal.
- `print_hex`: prints 8bit value as hexadecimal to terminal.

- `print_u8`: prints unsigned 8bit value as digit to terminal.
- `print_u16`: prints unsigned 16bit value as digit to terminal.
- `print_string`: prints string (until 0x00 is reached)from memory to terminal.
- `mul_u16`: Multiply 16bit numbers to produce 32bit result. Seems like easier to implement than in RISC due to possibility to use `MEMLO/MEMHI` instructions as general purpose registers.
- `mod_u16`: Uses Russian Peasant multiplication to calculate 16bit modulus
- `calc_sieve`: Uses Sieve of Atkin algorithm to populate memory and mark prime numbers from 5 to 255 (up to 8bits), 16bit calculation is already started. A more memory efficient code was written comparing to RISC, it packs every number as 1bit instead of 1 memory cell (16bits)

1.3 Improved assembler

Two new features have been added to assembler (which applies for both RISC and OISC)

1.3.1 Macros

Operators `%macro` and `%endmacro` allows to define code that can be reused multiple times.

1.3.2 Definitions

Operator `%def` allows to define a variable name under local scope (inside function) so assembly code would be easier to read. Example:

```
function_label:    ; function name
%def $x,REG0      ; define REG0 as $x
$x 10         ; set REG0 to decimal 10
```

2 Difficulties encountered

Multiple difficulties has been encountered:

2.1 Timing

There were multiple timing issues between source/destination latches/registers and common data bus. These issues were difficult to remove because initially ModelSim was running into cycling loop which did not allowed to proceed with simulation and locate issue. Secondly when this was resolved and simulation worked, synthesised code on FPGA did not work causing some issues when moving data from ALU calculated location back to ALU register. This been temporary resolved by adding negative edge sensitive flip-flop to ALU inputs which results is reducing time for combinational logic to settle by two.

2.2 Memory instructions

While writing benchmark programs another issue has been discovered - memory instructions (read to/write from memory and push to/pop from stack) cannot be used in single instruction e.g.:

```
MEMP 0x0000   ; macro to set memory pointer
MEMHI 0xFF   ; write immediate to memory high byte
STACK MEMHI   ; push memory high byte to stack
```

This will result in unexpected result as instruction tries to read and write from memory at the same time with two different addresses, therefore it can potentially store data to a unknown address. Current work-around is to avoid such instruction combination by storing value to a temporary register. This usually does not cause extra instructions as in most cases values from memory/stack are required and stored in temporary registers anyway.

2.3 ROM memory

Initial implementation of OISC stored instruction in 16bit, 2048 word configuration using M9K memory. As this was inefficient recently a decision was made to implement this using 3 M9K memory blocks with 2 instructions sliced into 3 - 9bits slices and stored in each memory block (extra bit is kept as parity check). This been successfully implemented in SystemVerilog and tested with ModelSim but still does not work with FPGA.

One of the problems is representation of non-integer byte sizes and multiple file formats used for each case - ModelSim uses `$readmemh` or `$readmemb` functions that reads file with raw binary or hexadecimal numbers; Quartus uses "mif" format which is quite well documented and already implemented; writing into ROM without resynthesising HDL requires reversed hexadecimal format, however it is not documented how it has to be represented when memory width is not an integer byte.

Furthermore probes might be implemented to record internal processor registers by using Quartus "In-System Sources and Probes" feature. This would allow quickly debug programs and find out any other problems related to processor.

3 Failure Risk Assessment

There are no updates on failure risk assessment.

4 Updated Safety Risk Assessment

There are no updates on safety risk assessment.

5 Help and Advice Needed

At this state no help is needed, and any issues and advices are sorted out and discussed in weekly supervisor meetings.

6 Updated Schedule

Table below includes project schedule. Note that a new objective has been added - **benchmarking**. This includes developing adequate methods to test and to performs these tests in order to evaluate performance of both processors on various tasks, such as time taking to execute a task, memory usage, power consumption etc.

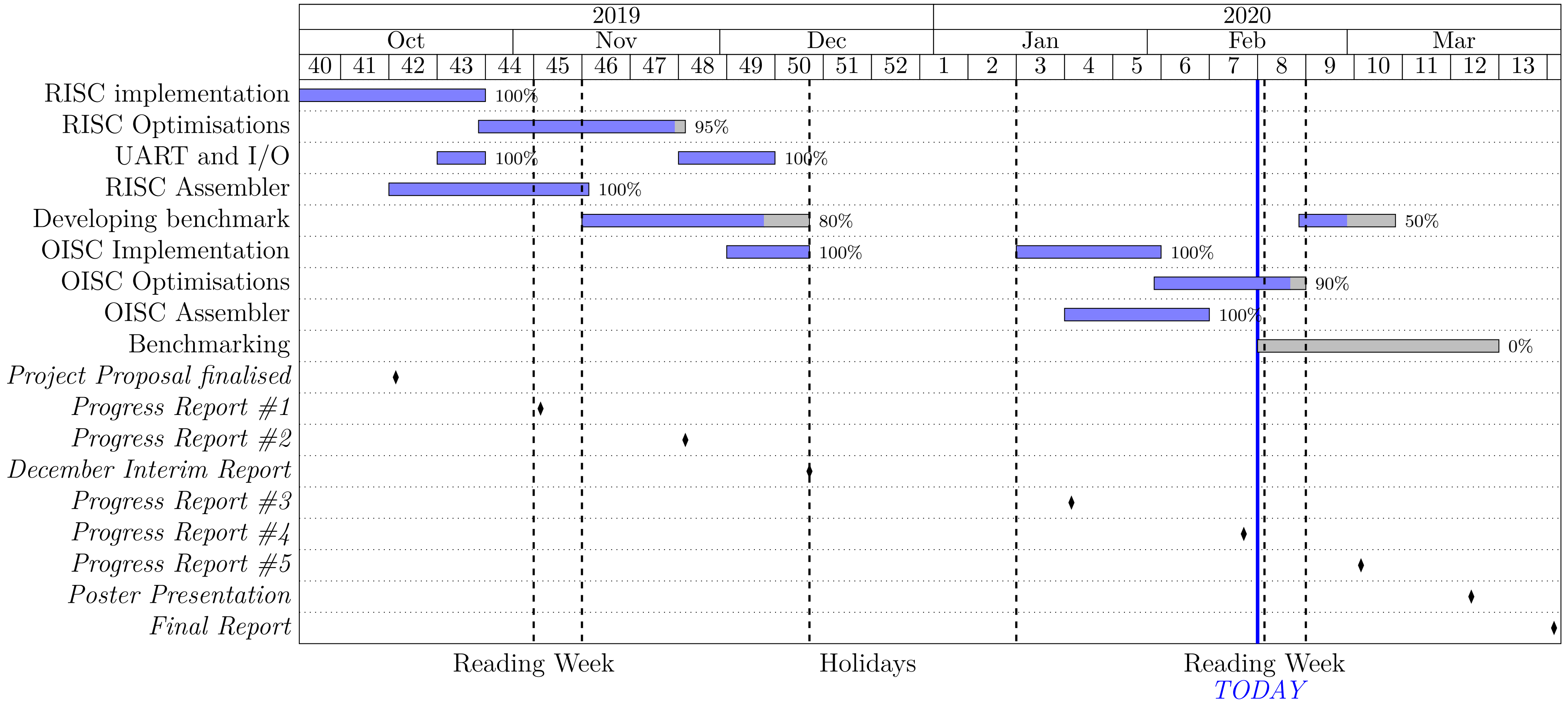


Table 3: Updated project schedule Grantt chart