Project Documentation

Jacob Priddy, Andrew Glencross, Charles Oroko, Seth Ballance

March 9, 2018

1 Introduction

Our processor implementation and instruction set are very lightweight meant to be flashed to and run an FPGA. This is an experimental project for a class and is not recommended for use in military or medical applications.

2 Processor

The processor does not have any floating point capabilities. There are 16 registers (including the program counter). It has 4KB of data memory and 64KB of program memory. It is pipelined, with 4 stages. All numbers are treated as two's compliment by the processor. Word size is 16 bits or 2 bytes stored big endian.

3 Instruction Set

All instructions are 1 word. Registers are addressed with 4 bits. 0000 would reference the PC while 0001 references R1. If there are 2 operands, the result gets put into the first operand. Opcode is 4 bits. All immediate values are interpreted to be decimal values.

Here is the complete instruction list.

- R - R type instruction
- I - I type instruction
- J - J type instruction
- P - Pseudo instruction

3.1 Instruction List

Instruction Description	Instruction	Type
add two registers	add	R
add an immediate to a register	addi	I
shift a register right arithmetic	sra	I
shift a register right logical	srl	I
shift a register left	sl	I
bitwise or two registers	or	R
bitwise or a register and immediate	ori	I
bitwise and two registers	and	R
bitwise and a register and immediate	andi	I
invert bits in a register	not	I
jump on register zero	jz	J
unconditional jump	j	J
load immediate into a register	loadi	I
move words from or to memory, or other registers	mov	R
No operation	nop	P

3.2 Register List

Register Name	Use	Binary Representation
R0	General purpose register	0000
R1	General purpose register	0001
R2	General purpose register	0010
R3	General purpose register	0011
R4	General purpose register	0100
R5	General purpose register	0101
R6	General purpose register	0110
R7	General purpose register	0111
R8	General purpose register	1000
R9	General purpose register	1001
R10	General purpose register	1010
R11	General purpose register	1011
R12	General purpose register	1100
R13	General purpose register	1101
R14	General purpose register	1110
R15	General purpose register	1111

4 Nomenclature

Bit key:

Bit	Definition
o	opcode bit
s	source/destination register bit
t	temporary register bit
i	immediate value
m	address mode bit
x	unused/reserved bit

5 Syntax

5.1 General Syntax

Instruction identifiers are to be followed by their arguments. If there are two arguments, they are to be separated by a comma. Only 1 instruction per line is allowed.

5.2 Comments

Comments start with a semicolon ';'. Anything after a semicolon to the end of the line is ignored by the assembler no matter where it is on a line.

5.3 Labels

Labels are to be followed by a colon ':', and may not contain spaces. Labels must come first in the line. They may be on a line by themselves or on a line with an instruction.

6 R-Type Instructions

6.1 Description

- 4 bit opcode
- 4 bits for operand 1
- 4 bits for operand 2

- 2 bits to select address mode (only used for mov instructions)
- 2 unused bits

6.2 Bit Field

o	o	o	o	s	s	s	s	t	t	t	t	m	m	x	x

6.3 List

Instruction	description	opcode	parameters
add	Add two registers	0000	2 reg
and	And two registers	0001	2 reg
or	Or two registers	0010	2 reg
mov	Moves first register to second register	0011	2 reg

6.4 Comments

6.5 add

Adds the first register and the second register, and stores the result into the first register. Treats numbers as two's compliment.

Example:

add r1, r2 r1 = r1 + r2

add PC, r15 PC = PC + r15

6.6 and

Does a bitwise AND operation on the bits in s and t, and store the results in s.

Example:

and r1, r2 r1 = r1 & r2

6.7 or

Does a bitwise OR operation on the bits in s and t, and store the results in s.

Example:

or r1, r2 r1 = r1 | r2

6.7.1 mov

The mov instruction is what is used to retrieve and write to memory. If you surround the register identifier in parentheses, the processor will treat the number in the register as a memory address, and read/write to/from said address.

Example:

mov r2, (r3) r2 = address(r3)
will copy the word at the address stored in r3 into r2

mov (r14), (r15) address(r14) = address(r15)
will copy the word at the address stored in r15 into the word pointed to by the address stored in r14.

7 I-Type Instructions

7.1 Description

- 4 bit opcode
- 4 bits for reg 1
- 8 bits for immediate value

7.2 Bit Field

o	o	o	o	s	s	s	s	i	i	i	i	i	i	i	i

7.3 List

Instruction	description	opcode	parameters
srl	shift right logical	0100	1 reg 1 immediate
sra	shift right arithmetic	0101	1 reg 1 immediate
sl	shift left arithmetic	0110	1 reg 1 immediate
not	flip bits in a register	0111	1 reg 1 immediate
andi	Does the and operation on a register and an immedate	1000	1 reg 1 immediate
addi	Add an immediate value to a register	1001	1 reg 1 immediate
ori	Does the or operation on a register and an immedate	1010	1 reg 1 immediate
loadi	Loads an immedate value into a register	1011	1 reg 1 immedate

7.4 Comments

7.4.1 srl

Shifts the number in the register to the right <immediate> number of times. Makes the incoming bit 0. Will not accept values > 16

Example:

srl r1, 10 r1 = r1 » 10

7.4.2 sra

Shifts the number in the register to the right <immediate> number of times. Keeps the sign of the number. Will not accept values > 16

Example:

sra r1, 11 r1 = r1 » 11

7.4.3 sl

Shifts the number in the register to the left <immediate> number of times. Makes the incoming bit 0. Will not accept values > 16

Example:

sl r1, 3 r1 = r1 « 3

7.4.4 not

Inverts the bits in a register up to <immediate> starting from the least significant bits. If no immediate value is provided, whole register will be flipped.

Example:

not r1 r1 = r̃1

not r1, 2 r1 = r1 except last 2 bits (least significant) will be inverted

7.4.5 andi

Ands an immediate with a register, and stores the result into the register. Can only and values from -127 to 127. If larger values are needed, see the R type instruction.

Example:

andi r1, 123 r1 = r1 & 0x7B

7.4.6 addi

Adds an immediate with a register, and stores the result into the register. Can only add an immediate value from -127 to 127. If larger values are needed, see the R type instruction.

Example:

addi r1, 123 r1 = r1 + 0x7B

7.4.7 ori

Bitwise or's an immediate with a register, and stores the result into the register. Can only or values from -127 to 127. If larger values are needed, see the R type instruction.

Example:

ori r1, 123 r1 = r1 | 0x7B

7.4.8 loadi

Stores the immediate value into the specified register. Can only load in values from -127 to 127.

Example:

loadi r5, 23 r1 = r1 & 0x17

8 J-Type

8.1 Description

- 4 bit opcode
- 4 bits for comparison register
- 8 bits for jump address

8.2 List

Instruction	description	opcode	parameters
jz	Jump to an offset if a register is zero	1100	1 reg 1 imm
j	Jump unconditionally	1101	immediate

8.3 Comments

8.3.1 General

Warning: it is possible to get the processor stuck by jumping a jump instruction.

8.3.2 jz

Jumps to a label. Stores the offset, so it can only be used to jump to an instruction ±127 instructions away. If you need to go further, use the j instruction.

Bit Field

o	o	o	o	s	s	s	s	i	i	i	i	i	i	i	i

Example:

jz r1, label jumps to label if the register is 0

8.3.3 j

Jumps to that position in memory via a label.

Bit Field

o	o	o	o	i	i	i	i	i	i	i	i	i	i	i	i

Example:

j label jumps to label

9 Other

Special instructions such as pseudo instructions.

Instruction	description	opcode	parameters
nop	No operation	1111	no parameters, all zeros

9.1 Comments

9.1.1 nop

Effectively adds 0 to R0.

10 Architecture

This is a description of the blocks on the CPU that have already been designed, and how they work.

Note For writing VHDL we kept to some conventions. Input ports have the prefix **i**, output ports have the prefix **q** and signals have the prefix **l**. Also, for ease of typing, we kept the naming in lowercase.

10.1 Top Level

Description The way the FPGA interacts with the outside world between user and CPU.

For inputs, the CPU has a 100MHz clock, **clk**, and a couple of buttons and switches, **switches**, **buttons**, so that different modes can be implemented, like a bootloading mode, or a debugging mode. **rx** is a UART input.

For outputs, the CPU has the seven segment display and 8 LEDs. **tx** is a UART output.

```
entity booths_architecture is
  port( i_clk_100   : in  std_logic; --100MHz clock
        i_switches  : in  std_logic_vector(  1 downto 0 ); --two switches
        i_buttons   : in  std_logic_vector(  1 downto 0 ); --two buttons
        i_rx        : in  std_logic; --serial input of UART

        q_7_segment : out std_logic_vector(  6 downto 0 ); --7 segment display for debugging
        q_leds      : out std_logic_vector(  7 downto 0 ); --8 lines for LEDs
        q_tx        : out std_logic); --serial output of UART
end booths_architecture;
```

Comments The operation of the other blocks with respect to the top level have not yet been implemented, as some are still being created and worked on. So far, there are two modes available: active mode and bootloading mode. a choice of input signal to choose the modes has yet to be decided, however most likely will be implemented as the position of the switches on power on. One button will be a reset signal that can be found in the code as **reset**.

10.2 rx Communication

Description UART input communication.

The rx communication protocol allows bootloading of the assembler to the FPGA. As can be seen from the entity below, it runs asynchronously. ie **clk** is not the same clock as the rest of the blocks. We get the **reset** and **baud 16** inputs from the switches on the FPGA board.

```
entity uart_rx is
  port( i_clk     : in  std_logic; --asynchronous clock
        i_reset   : in  std_logic; --reset signal
        i_baud_16 : in  std_logic; --16 times baud rate
        i_rx      : in  std_logic; --serial input line

        q_data    : out std_logic_vector( 7 downto 0 ); --byte to receive
        q_flag    : out std_logic); --toggle on byte receive
end uart_rx;
```

When **reset** and **baud 16** are set, the bootloading sequence begins. We double clock **clk** after the start bit is received in order to read the other bits as shown in the image below: