An 8-bit RISC Memory-scalable CPU Based on FPGA

Jialiang Zhao

February 26, 2017

Abstract

This paper describes a preliminary development of an 8-bit RISC CPU based on FPGA. This CPU runs a set of self-defined 8-bit instructions. A new method of memory addressing is adopted in order to enlarge or lessen the memory capacity. Unlike the memory segmentation method adopted in 8086/8088, the data address in this CPU is designed to be adjustable, which is composed of bits from asynchronous instruction sequences. The capacity of memory addressing can be adjusted from 2^4 bits to infinity in theory. A $640 * 480$ VGA module is designed for monitoring.

This project is based on Verilog HDL, complied by Xilinx ISE 14.7, fully tested on Xilinx XC6XLS45 (Spartan 6).

I Instruction Set Design

Instructions of this CPU are divided into three groups: register instructions, arithmetic instructions and branch instructions. Register instructions contains I/O commands between CPU registers and memory; arithmetic instructions include the register-level calculation, while branch instructions decide which instruction will be executed next depends on flag register.

I.1 Register Instruction

There are four registers inside this CPU, and each one comes with a unique address. Names and codes of these registers are listed in Table 1.

Table 1: Registers in this CPU

Name	Address
AX	00
BX	01
CX	10
DX	11

Register Instructions are listed in Table 2.

Table 2: Register instructions

Instruction	Description
0000 xx ADDR(2)	LOAD TMP, ADDRESS (first part)
0010 RD xx	MOV RD, TMP
0001 RD xx	STORE RD, ADDRESS (first part)

The loading process is achieved by two instructions. Firstly, users use LOAD instruction to load the memory of specific address to a temporary register TMP, and then use MOV instruction to move the data from TMP to destiny register (AX, BX, CX, or DX).

Take 10-bit data bus as an example. The memory address is divided into 2 parts, the first part is the last two bits of the LOAD or STORE instruction, which is also the last two bits of the memory address. The high 8 bits come from the next instruction, which means the LOAD or STORE is achieved by 2 instructions for a 10-bit-data-bus CPU. An example instruction series is as below:

```
00000001
11110111  // load the memory in 1111_0111_01 to TMP
00011000  // move data from TMP to register CX
```

```
/***** other operations *****/
00101001
11110111  // store data of register CV to
          // memory location 1111_0111_01
```

I.2 Arithmetic Instructions

Till now three groups of instructions are implemented in this CPU: addition, multiplication and division.

Instructions of addition are listed in Table 3. There are two kinds of addition: increment (decrement) and addition (subtraction).

Table 3: Addition

Instruction	Description
1000 RD xx	RD → RD + 1
1001 RD xx	RD → RD - 1
1010 RD RS	RD → RD + RS
1011 RD RS	RS → RD - RS

Instructions set of multiplication is listed in Table 4, which is divided into two groups, unsigned multiplication and signed multiplication.

Table 4: Multiplication

Instruction	Description
0100 RD RS	(unsigned) RD → RD[3:0] * RS[3:0]
0101 RD RS	(signed) RD → RD[3:0] * RS[3:0]

Instructions set of division is listed in Table 5, which is divided into two groups, unsigned division and signed division.

Table 5: Division

Instruction	Description
0110 RD RS	(unsigned) RD[3:0] → RD[7:0]/RS[3:0] MOD: RD[7:4]
0111 RD RS	(signed) RD[3:0] → RD[7:0]/RS[3:0] MOD: RD[7:4]

I.3 Branch Instructions

The flag register of this CPU is of 8-bit, where only the low 4 bits are used. The function of each bit is listed in Table 6.

Table 6: Flag Register

Position	Description
0000 0001	ZF: zero flag
0000 0010	CF: carry flag
0000 0100	OF: overflow flag
0000 1000	NF: negative flag

The branch instructions are listed in Table 7.

Table 7: Branch Instructions

Instruction	Description
1100 OFFSET	Branch Zero
1101 OFFSET	Branch Carry
1110 OFFSET	Branch Overflow
1111 OFFSET	Branch Negative
0011 1111	HALT

Take the *Branch Zero* instruction as an example. Assuming the sequence number of the current instruction is n, then after

$$11000101$$

the program counter will pop up the $(n + 5)_{th}$ instruction if $ZF = 1$, else the $(n + 1)_{th}$ instruction will be executed.

II Schematics Design

The overall RTL graph is Figure 1.

II.1 Datapath

The datapath of this CPU is demonstrated by Figure 3. Datapath is composed of a Register File, an Arithmetic Logic Unit (ALU) and two multiplexers. The RTL graph of datapath is Figure 2

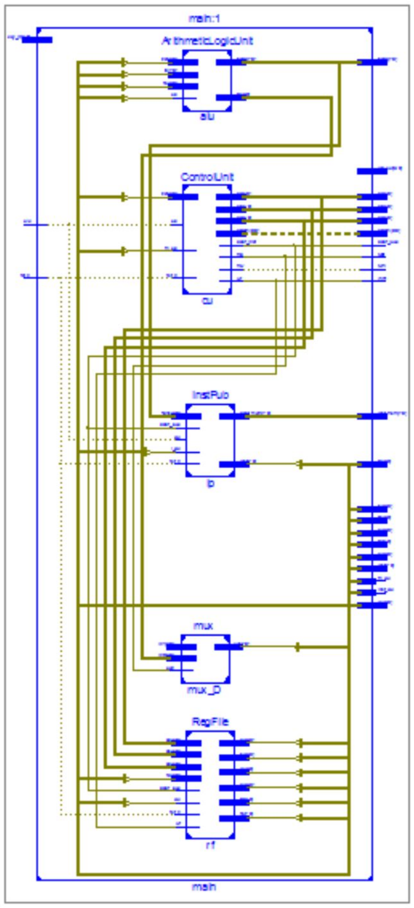


Figure 1: Overall RTL

Register File

There are three I/O registers in Register File, named RA, RB and RD. RA and RB serve as output, while RD serve as input. Each register has an 8-bit address pin, AA, AB, AD. Also, a write enable pin is adopted here. There are four inner registers in this Register File as stated earlier, named AX, BX, CX and DX. Because in this CPU the memory instructions are composed of two or more instructions, which means there may be some instructions that are pure address. In order to avoid mistaking these instructions as other operations, a signal named $addr_{l}ock$ is used as an identifier. When $addr_{l}ock$ is $HIGH$, it means the current instruction is a pure address, and it isn't when $addr_{l}ock$ is LOW. The port list is as below in *Verilog*.

```
module RegFile(
```

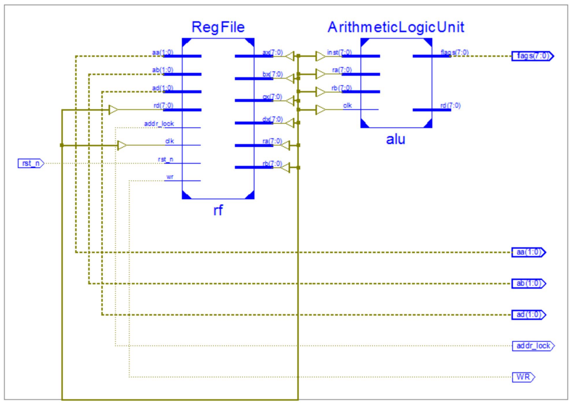


Figure 2: Datapath RTL

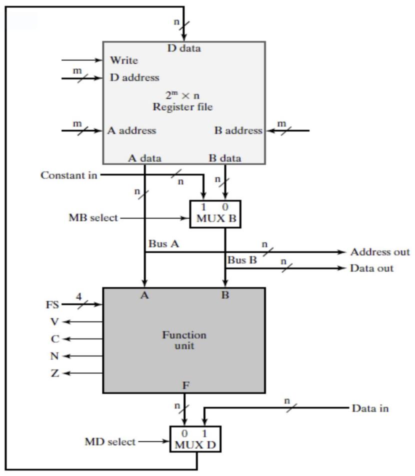


Figure 3: Datapath

```
ax, bx, cx, dx,
ra, rb,
wr, rd,
addr_lock,
aa, ab, ad, clk, rst_n
);

output reg [7:0] ra;
output reg [7:0] rb;
input wr, rst_n, clk;
input [7:0] rd;
input [1:0] aa;
input [1:0] ab;
input [1:0] ad;

input addr_lock;
//reg [7:0] ax, bx, cx, dx;
output reg [7:0] ax, bx, cx, dx;
```

Assignment of RA, RB and RD are asynchronous, value of them will change immediately after change of AA, AB and AD. Implementation are as below:

```
always @ (*) begin
    if(!addr_lock)
    case (aa)
        2'b00: ra <= ax;
        2'b01: ra <= bx;
        2'b10: ra <= cx;
        2'b11: ra <= dx;
        default: ra <= 8'b0;
    endcase
end

always @(negedge clk) begin
    if(!addr_lock) begin
    if(!rst_n) begin
        ax <= 8'd0;
        bx <= 8'd0;
        cx <= 8'd0;
        dx <= 8'd0;
    end
    else if({wr, ad} == 3'b100)
        ax <= rd;
    else if({wr, ad} == 3'b101)
        bx <= rd;
    else if({wr, ad} == 3'b110)
        cx <= rd;
    else if({wr, ad} == 3'b111)
        dx <= rd;
    end
end
endmodule
```

Arithmetic Logic Unit (ALU)

Two kinds of ALU operations are supported now: addition and multiplication. Flags are also generated in this module.

```
module ArithmeticLogicUnit(
rd, flags,
ra, rb, inst, clk
);

output reg [7:0] rd;
output reg [7:0] flags;
input [7:0] ra;
input [7:0] rb;
input [7:0] inst;
input clk;      // only modify flags when clk comes


/******* Adder part ****************/
```

```
wire [7:0] tmp_add, tmpf_add;
// tmp res and tmp flags
reg [7:0] add_oprand;

always @(*) begin   // adder: res = ra + add_oprand
case (inst[7:4])
4'b1000: add_oprand <= 8'd1;
4'b1001: add_oprand <= 8'b11111111;
4'b1010: add_oprand <= rb;
4'b1011: add_oprand <= ~rb + 8'b1;
default: add_oprand <= 8'b0;
endcase
end

FullAdder fa (.res(tmp_add),
.flags(tmpf_add),
.ra(ra),
.rb(add_oprand),
.cin(1'b0)
);
/***********************************/
/********* Multiplier ***************/
wire [7:0] tmp_mul, tmpf_mul;
Multiplier mul(
.res(tmp_mul),
.flags(tmpf_mul),
.ra(ra[3:0]),
.rb(rb[3:0])
);

/****** Signed Multiplication *******/
wire [7:0] tmp_mul_s, tmpf_mul_s;
MultiplierSigned mul_s(
.res(tmp_mul_s),
.flags(tmpf_mul_s),
.ra(ra[3:0]),
.rb(rb[3:0])
);

/**********************************/
always @(*) begin
case (inst[7:4])
4'b1000: rd <= tmp_add;
4'b1001: rd <= tmp_add;
4'b1010: rd <= tmp_add;
4'b1011: rd <= tmp_add;
```

```
4'b0100: rd <= tmp_mul;
4'b0101: rd <= tmp_mul_s;
default: rd <= 8'b0;
endcase
end

always @(posedge clk) begin
case (inst[7:4])
4'b1000: flags <= tmpf_add;
4'b1001: flags <= tmpf_add;
4'b1010: flags <= tmpf_add;
4'b1011: flags <= tmpf_add;
4'b0100: flags <= tmpf_mul;
4'b0101: flags <= tmpf_mul_s;
default: flags <= 8'b0;
endcase
end

endmodule
```

Because of the limited space, detailed code of addition and multiplication is not listed here.

ALU Control

The current instruction is stored in variable *inst*. When $addr_lock$ is low, which indicates the current instruction is not an address, the ALU will decode it, and if it's an ALU operation, the corresponding operation will be executed.

Flags

ALU operations cause changes of flags. Generation of flags is demonstrated in the previous code.

II.2 Control Unit

The Control Unit of this CPU decodes instructions, generates control signal and assigns addresses for Register File registers. Inputs of this module are instructions, clock, memory clock, and reset signal. Outputs are address for Register File (aa, ab, ad), address for memory, the two control signals for multiplexer (md and mw), and the control signal for Register File (wr). A graphical demonstration Figure 4.

II.3 Program Counter

This module instantiate a dual port 256-Byte RAM used for instruction storage. It is currently designed to be 1MB, or 2^{10} instructions in another word. The

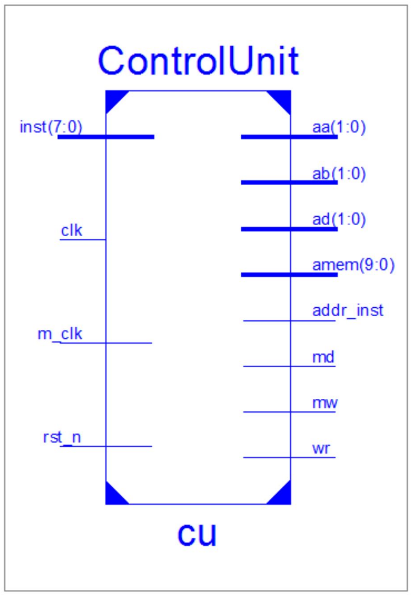


Figure 4: Control Unit

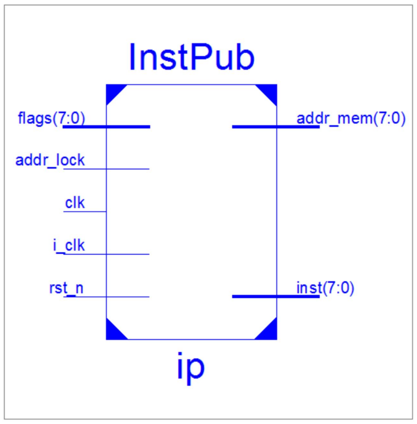


Figure 5: Program Counter

variable amem stores the address of next instruction. Variable amem increases by 1 after every period under normal conditions, or jumps to another address when brunching conditions are satisfied. Refer to Figure 5 for the RTL graph.

III Technical Innovations

Some new technologies are adopted in this CPU.

III.1 Scalable Data Bus

Although the control bus is fixed to be of 8-bit, the bandwidth of data bus is scalable. Unlike the segmented data bus adopted in 8086/8088 CPU, the data bus in this CPU is composed of bits from sequenced instructions, and users should specify the bandwidth before programming for it.

Take 10-bit implementation as an example. A full address is composed of 2 bit from the current instruction and 8 bits from next instruction, where the current instruction is called a *half-address instruction*, the second one is called a *full-adress instruction*. A variable *inst_lock* is used to determine in which kind the current instruction is. It is clear that the a *half-address instruction* also consists of commands (0000xxxx for LOAD, 0001xxxx for STORE, etc.), thus this kind of instructions shold also be decoded, while a *full-adress instruction* is pure-address and it shouldn't be decoded. In this case, other modules should stop working. For example, the assignment of register RB in the next program block is activated only when !*inst_lock*.

```
always @ (*) begin
  if(!addr_lock)
    case (ab)
      2'b00: rb <= ax;
      2'b01: rb <= bx;
      2'b10: rb <= cx;
      2'b11: rb <= dx;
      default: rb <= 8'b0;
    endcase
end
```

III.2 Phase-shifted Dual Clock

Two clocks are used in this CPU, and there is a 50% phase shift between them as illustrated in Figure 6. This design is used to avoid the situation where sequentially instructions are activated at the same time. Every ALU calculation is done with the positive edge or negative edge of the leading clock, while every memory I/O process is down with the positive edge or negative edge of the lagging clock.

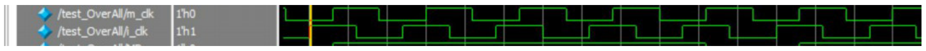


Figure 6: Dual Clock

IV Practical Example

Take the calculation of $|3-7|$ as an example.

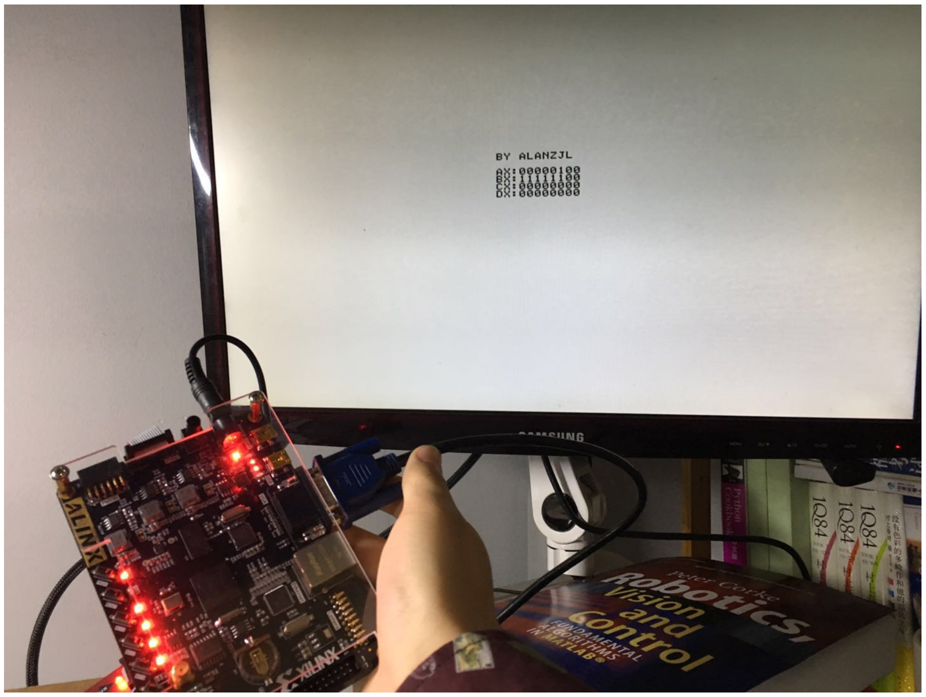
IV.1 Input/Output

A VGA module is designed for monitoring. Because of the lengthy and simplicity of the *Verilog* code, a *Python* script is designed to generate the displaying code. A part of this script is shown as follows, and the full script contains 8 parts, each with a goal to show one character.

```
#! /usr/bin/python2.7

mod_cnt = 19
row_cnt = 21
sub_cnt = 7

```

```
abcd_cnt = 7;

print "\n\n\n///////////////data␣1␣\n\n"
for i in range(1,7):
  print ("RAM_set␣u_ram_%d␣(.clk(clk),.rst(rst_n),\
␣␣.data({5'b0,%sx[%d:%d]}),.col0(p[%d][%d:%d]),\
␣␣.col1(p[%d][%d:%d]),.col2(p[%d][%d:%d]),\
␣␣.col3(p[%d][%d:%d]),.col4(p[%d][%d:%d]),\
␣␣.col5(p[%d][%d:%d]),.col6(p[%d][%d:%d]));"\
  %(mod_cnt,chr(94+i),abcd_cnt, abcd_cnt,row_cnt,\
  sub_cnt, sub_cnt-7, row_cnt+1,sub_cnt, sub_cnt-7, \
  row_cnt+2,sub_cnt, sub_cnt-7, row_cnt+3,sub_cnt,\
  sub_cnt-7,row_cnt+4,sub_cnt, sub_cnt-7, row_cnt+5,\
  sub_cnt, sub_cnt-7, row_cnt+6,sub_cnt, sub_cnt-7));
  mod_cnt = mod_cnt + 1;
  sub_cnt = sub_cnt + 8;
abcd_cnt = abcd_cnt - 1;
```

The results are shown in Figure 7.



```
BY ALANZJL
AX:00000100
BX:11111100
CX:00000000
DX:00000000
```

Figure 7: VGA Display

IV.2 Machine Code Program

```
00000011,
00000000,
// load 0000_0000_11 (3 in decimal) to TMP register
00100100,
// transfer TMP data to register 01 (BX)
00000011,
00000001,
// load 0000_0001_11 (7 in decimal) to TMP register
00100000,
// transfer TMP data to register 00 (AX)
10110100,
// do: BX = BX - AX (this instruction modifies flags)
11110010,
// if negative flag is 1 then pc=pc+2, else pc=pc+1
00111111,
// halt
00000000,
00000000,
// load 0000_0000_00 (0 in decimal) to TMP
00100000,
// transfer TMP data to register 00 (AX)
10110001,
// do: AX = AX - BX
00111111,
// halt
```

IV.3 Simulation Results

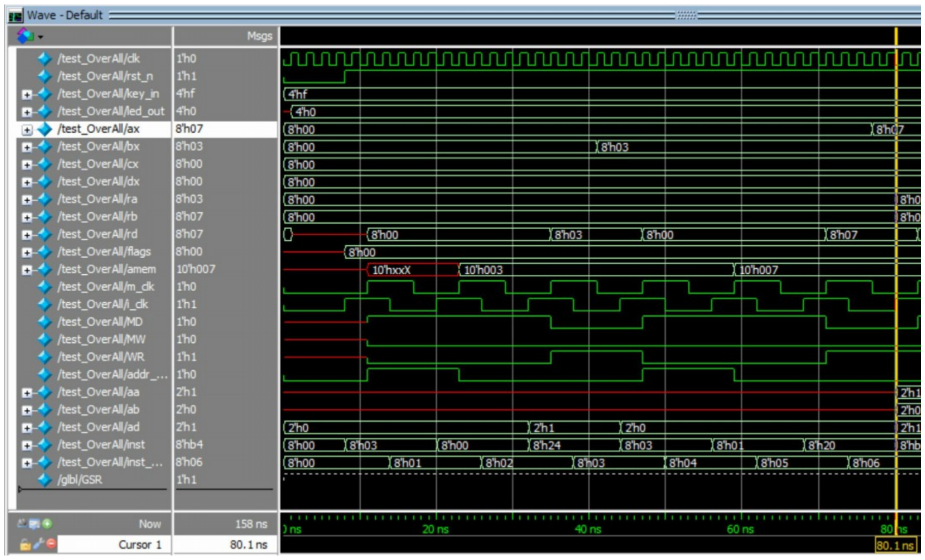
Simulation Results shown in Figure 8 are generated by *ModelSim*.

IV.4 Practical Results

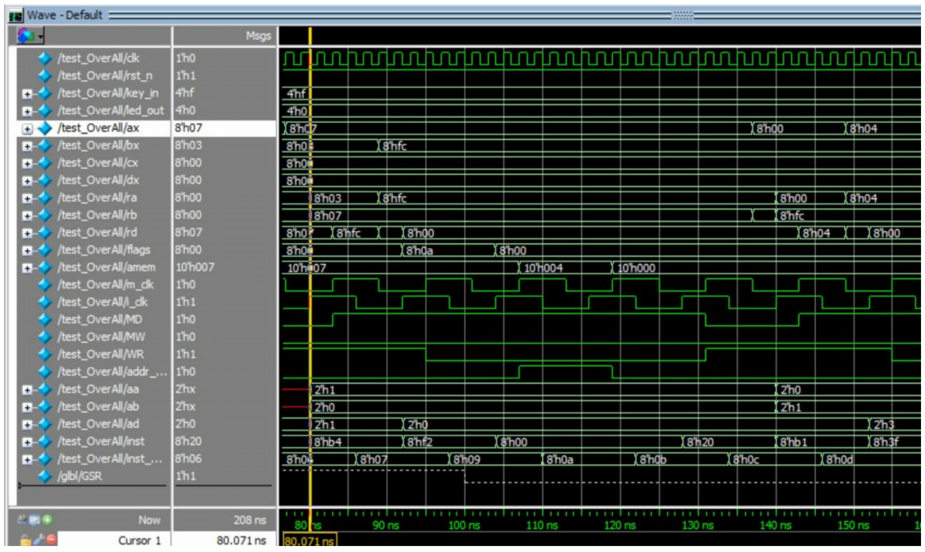
Practical Results shown in Figure 9 are observed by *ChipScope*, a virtual logic analyzer that reads outputs of Xilinx FPGA precisely and with high sampling frequency.

Variable *status* indicates the start of a program block. It also works as trigger condition of *ChipScope*. It is generated by:

```
reg status;          // activate chipscope
reg rst_tmp1, rst_tmp2;
always @ (posedge clk) rst_tmp1 <= rst_n;
always @ (posedge clk) rst_tmp2 <= rst_tmp1;
always @ (posedge clk) begin
```



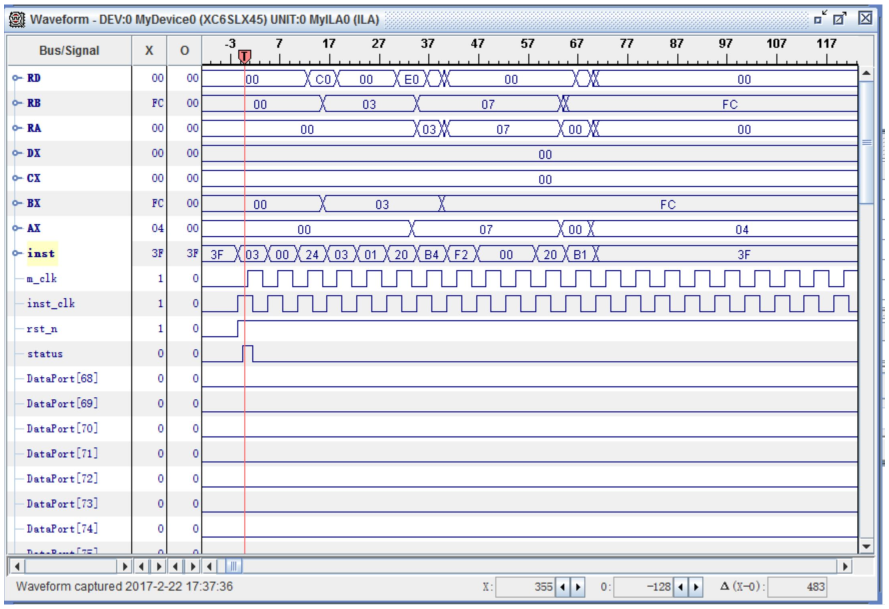
(a) First Part



(b) Second Part

Figure 8: Simulation Results

```
    if({rst_n , rst_tmp2} == 2'b10)
      status <= 1'b1;
    else
    status <= 1'b0;
  end
```



Bus/Signal	X	O
RD	00	00
RB	FC	00
RA	00	00
DX	00	00
CX	00	00
BX	FC	00
AX	04	00
inst	3F	3F
m_clk	1	0
inst_clk	1	0
rst_n	1	0
status	0	0
DataPort[68]	0	0
DataPort[69]	0	0
DataPort[70]	0	0
DataPort[71]	0	0
DataPort[72]	0	0
DataPort[73]	0	0
DataPort[74]	0	0

Waveform captured 2017-2-22 17:37:36

Figure 9: Practical Results