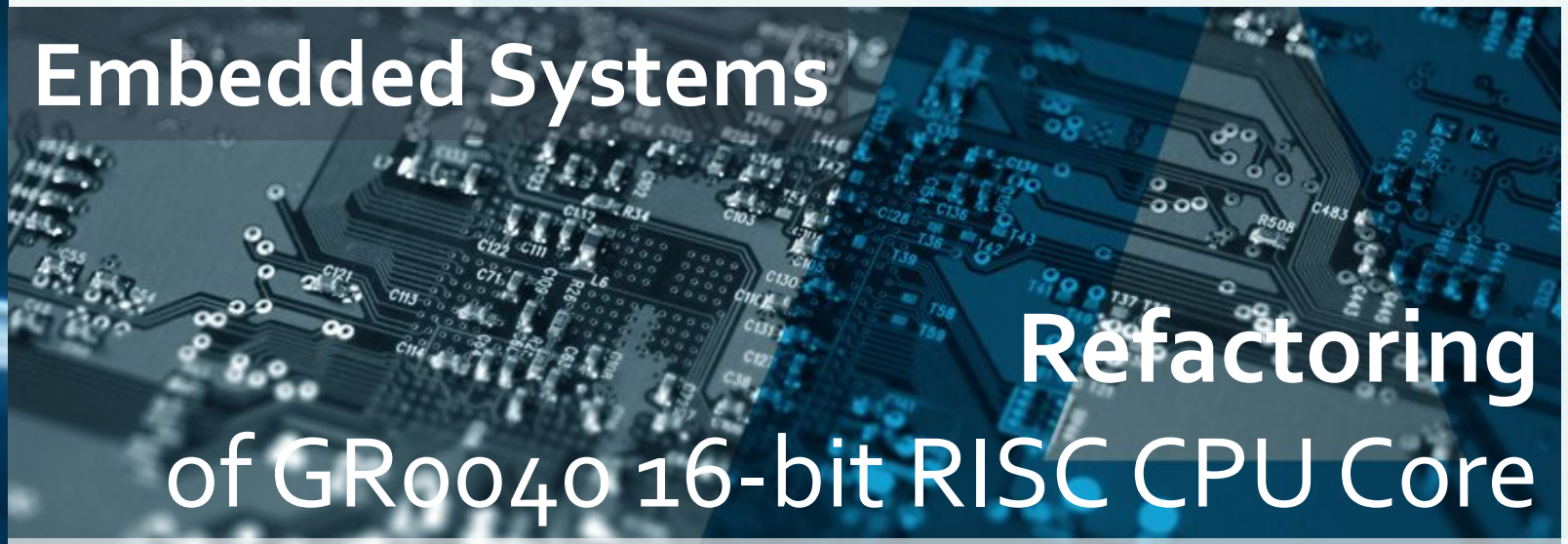
ESRG
Embedded Systems
Research Group

CENTROALGORITMI

Final Presentation
Processor GR0040

Embedded Systems



Refactoring
of GR0040 16-bit RISC CPU Core

André Martins– PG60192
Mariana Martins – PG60211

Universidade do Minho
Escola de Engenharia

COMPETE

QREN Quadro de Referência Estratégico Nacional

União Europeia
Fundo Europeu de Desenvolvimento Regional

FCT Fundação para a Ciência e a Tecnologia
Ministério da Ciência, Tecnologia e Ensino Superior

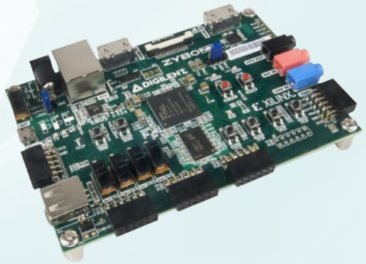
Agenda

- Problem statement;
- Zybo z7-10
- Original ISA
- Memory Organization
- GR0040 and GR0041
- SoC Overview
- Assembler
- Refactoring
- System Validation and Testing

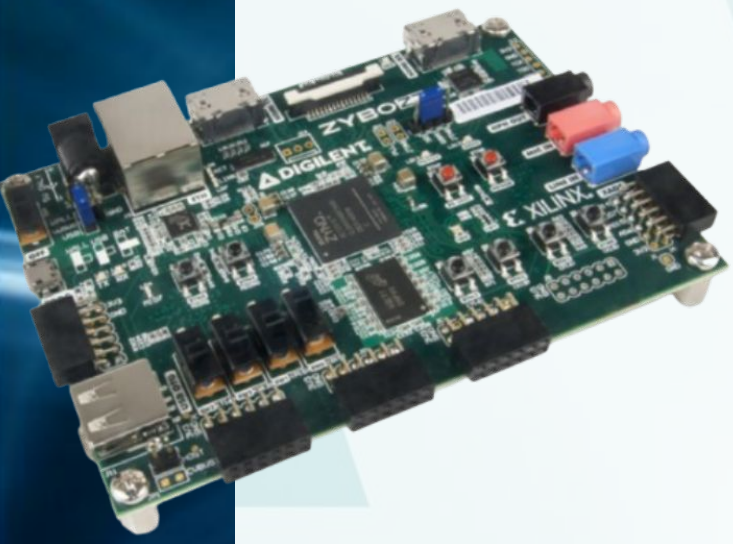
Problem Statement

This project **analyzes** the initial GR0040 SoC, **refactors** it into a clean, **modular** design, and preserves full functional equivalence. It maintains strict compliance with the GR0040 architecture and instruction set, retaining all existing behavior. The work **identifies** architectural weaknesses, **redesigns** the system structure, and **integrates** improvements and new peripherals such as UART and I2C. Finally, the resulting code should be implemented in FPGA.

The outcome delivers a robust, maintainable SoC that simplifies verification, enables extension, and supports long-term evolution.



Zybo Z7-10



Overview

Belongs to the 7 series family. Features the Xilinx Zynq-7010 SoC, which integrates a dual-core ARM Cortex-A9 processor running at 667 MHz with Artix-7 FPGA programmable logic. It is a Hybrid board since it already has a processor and implements also CLBs.
External 125MHz clock.

Resources

It provides 17,600 look-up tables (LUTs), 35,200 flip-flops, 270 KB block RAM.

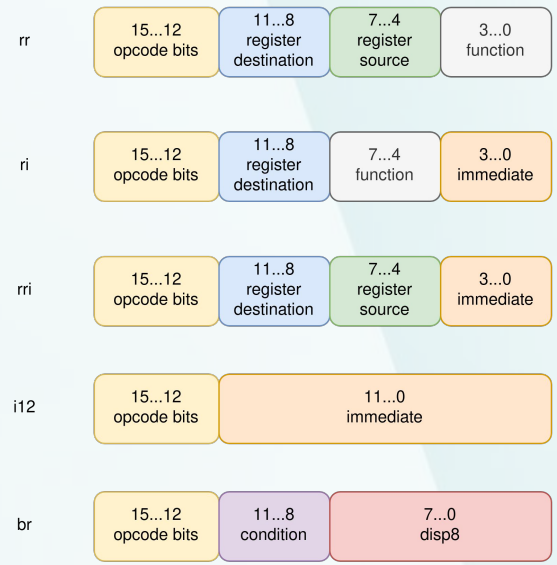
Memory and Storage

The board includes 1 GB DDR3L RAM on a 32-bit bus at 533 MHz (1066 MT/s), 16 MB Quad-SPI Flash for non-volatile storage, and a microSD slot.

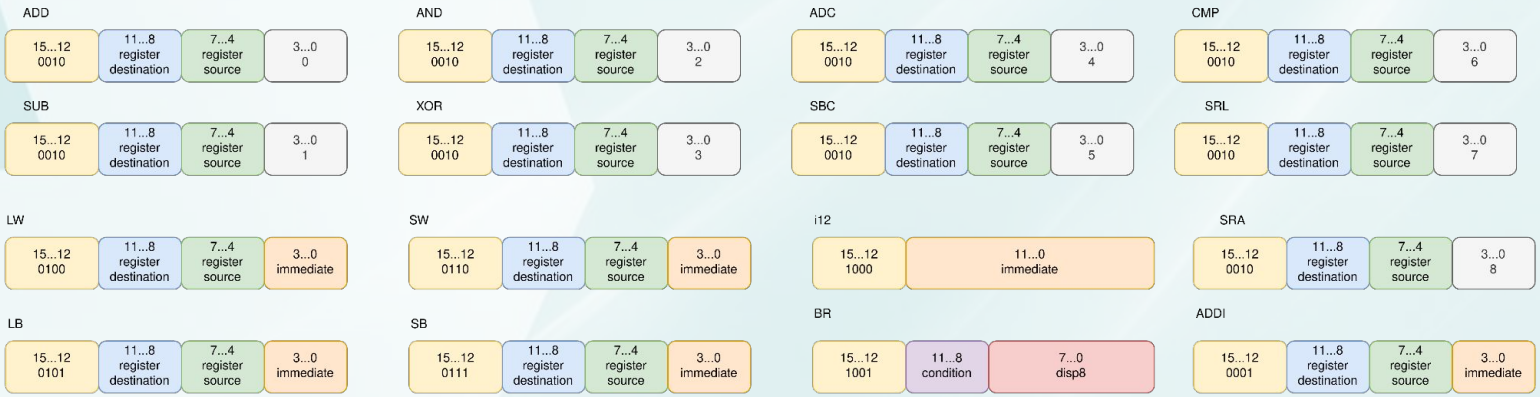
Connectivity

Key interfaces encompass Gigabit Ethernet, USB 2.0 OTG (host mode), HDMI input/output (no CEC on input for Z7-10), audio codec, 5 Pmod ports, USB-JTAG/UART, and low-speed controllers like SPI, I2C, UART. Power comes via USB or 5V external supply.

Original ISA

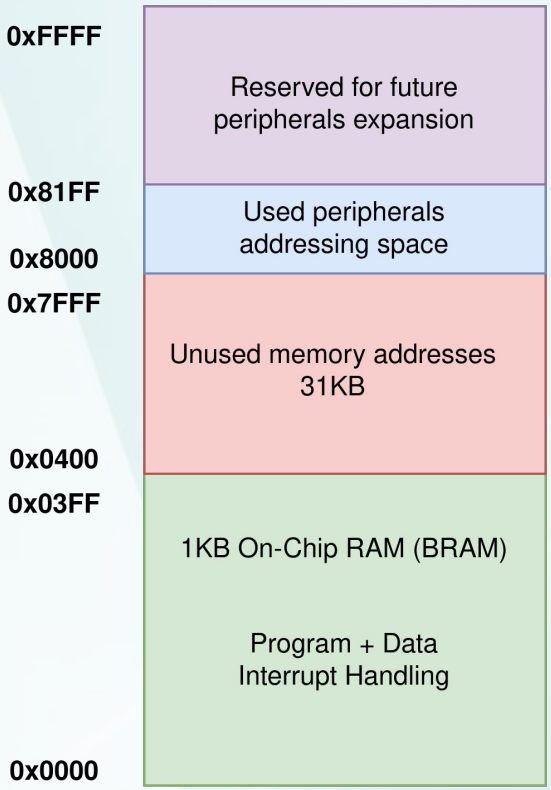


5 different formats of instructions



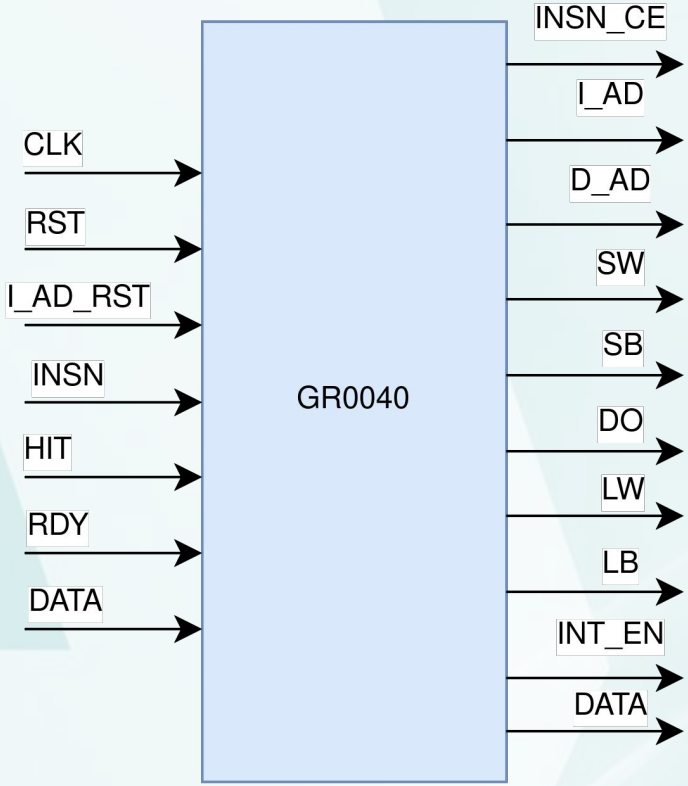
Memory Organization

Pre-Refactored SoC GR0040

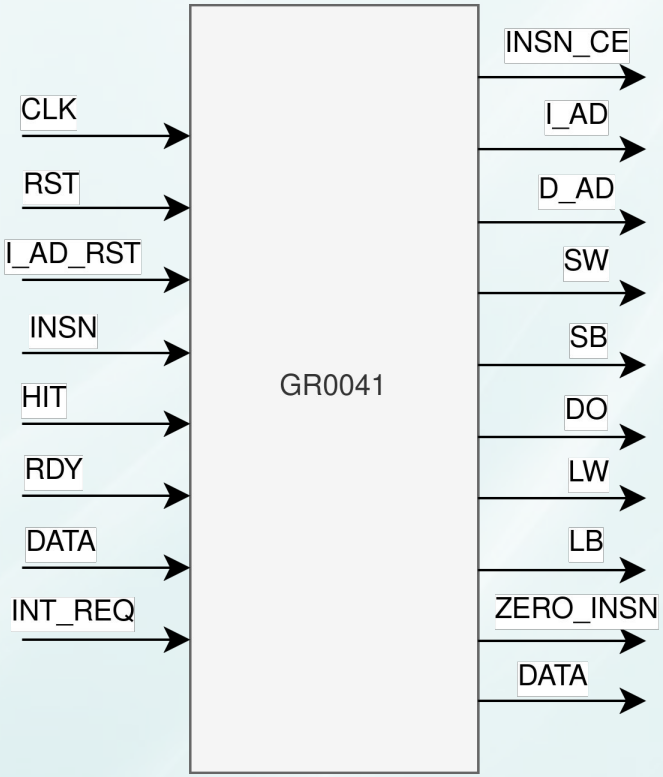


I/O Diagram

Pre-Refactored SoC GR0040



Module GR0040

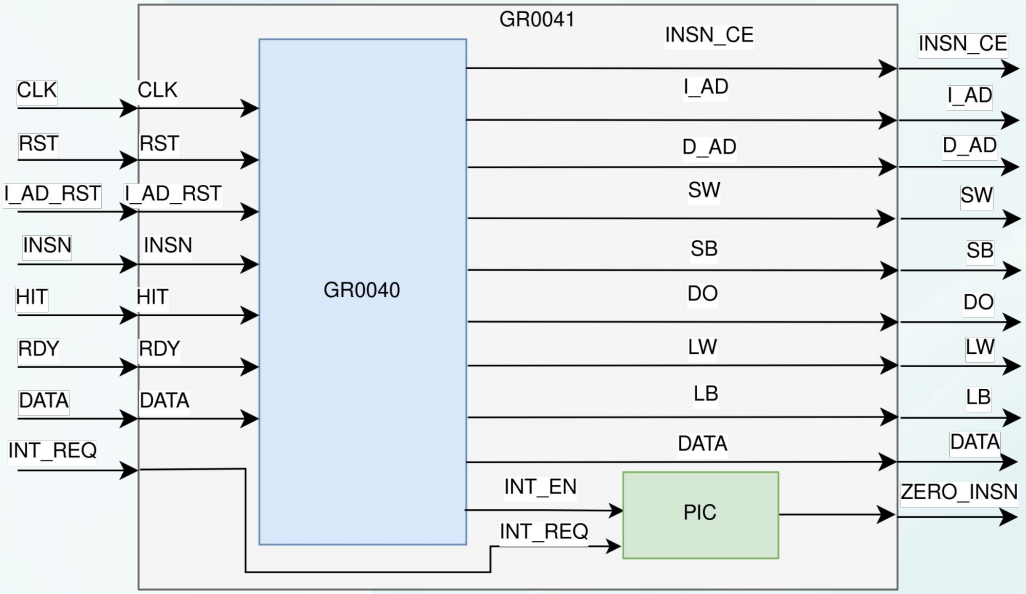


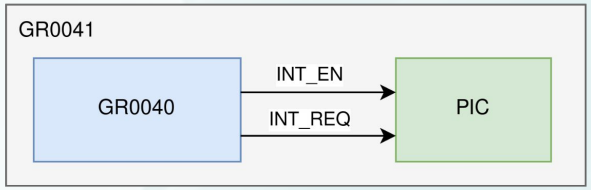
Module GR0041

I/O Diagram

CENTROALGORITMI

Pre-Refactored SoC GR0040





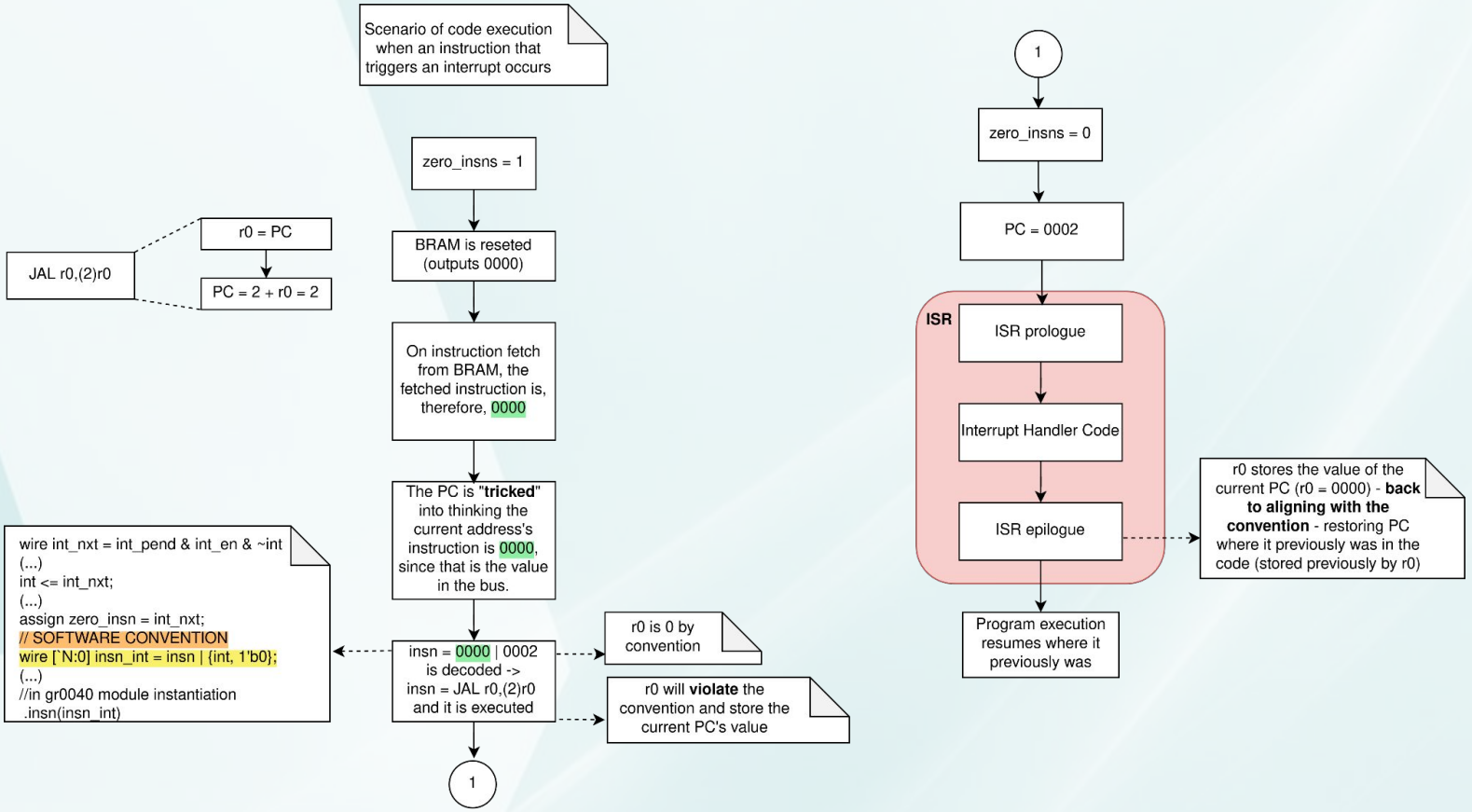
Module GR0041 (Specified)

Interrupt Logic

CENTROALGORITMI

Pre-Refactored SoC GR0040

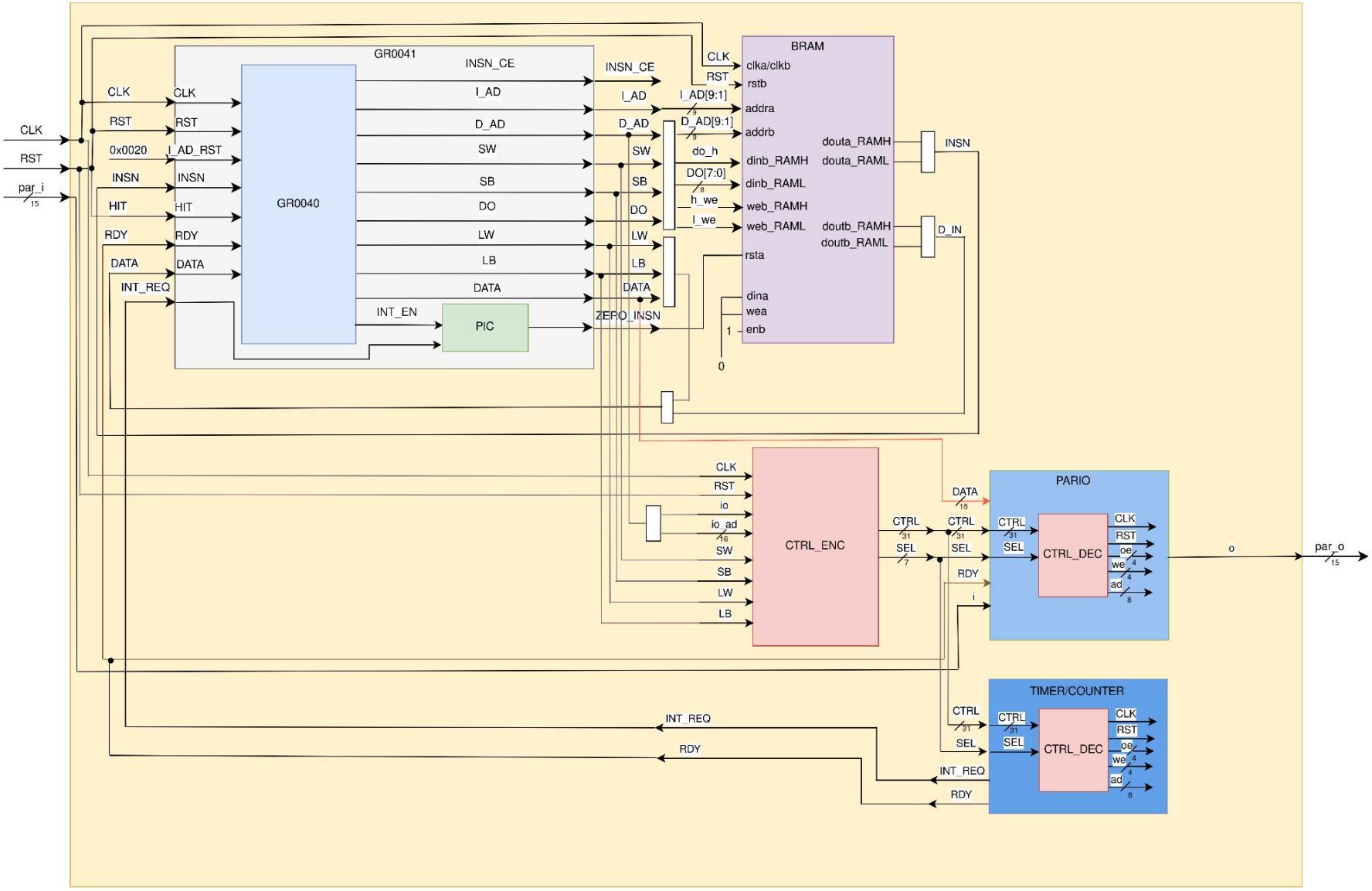
Only one interrupt existed and could be handled.



Architecture Overview

Pre-Refactored SoC GR0040

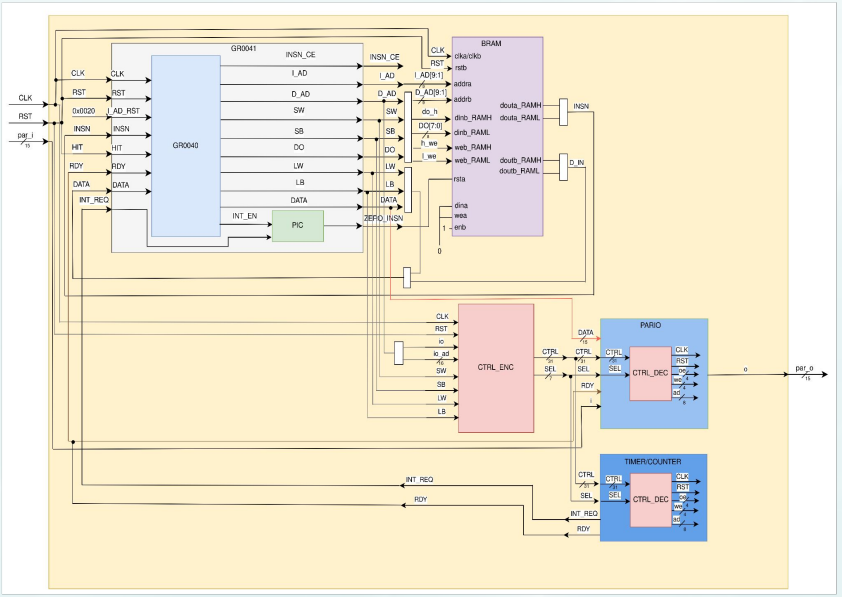
CENTROALGORITMI



Architecture Overview

Pre-Refactored SoC GR0040

- Datapath and Control unit mixed;
- Peripheral Timer;
- Peripheral Pario;
- Only one interrupt supported (timer interrupt);
- Instructions can be overwritten in runtime due to the poor memory organization.



Assembler

Assembler developed in C in order to simplify instruction decoding from Assembly to its equivalent hexadecimal value.

```
andre@macaco:~/Documents/Assembler$ make
gcc -Wall -o program main.c assembler.c
./program instructions.asm hexa.txt
Program Execution Terminated. Opening Output File
xdg-open hexa.txt
```

Separates instructions in High and Low memory .coe files

Open | instructions.asm ~/Documents/Assem... | Save

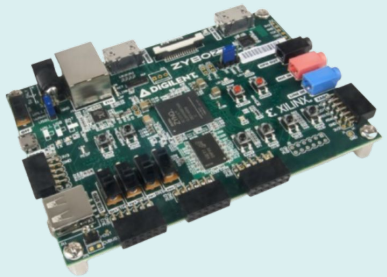
```
1 isr:
2 // Interrupt handler
3 JAL R0, R0, #0
4 ADDI R15, R15, #-1
5 SW R1, R15, #0
6 BR #-3
7
8 reset:
9 // Reset vector at 0x0020
10 IMM #128
11 ADDI R15, R0, #2
12
13 start:
14 // Main program
15 ADDI R1, R0, #10
16 ADDI R2, R0, #15
17 ADD R3, R1
18 CMP R1, R2
19 BEQ #5
```

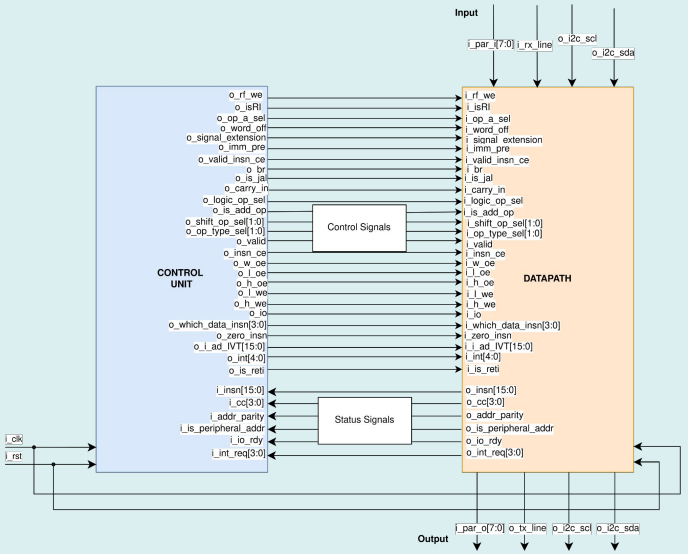
Open | hexa.txt ~/Downloads

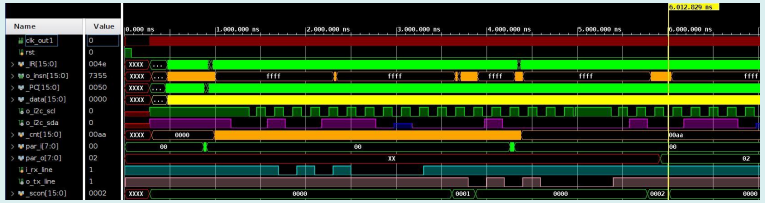
```
1 0000
2 1FFF
3 61F0
4 90FD
5 FFFF
6 FFFF
7 FFFF
8 FFFF
9 FFFF
10 FFFF
11 FFFF
12 FFFF
13 FFFF
14 FFFF
15 FFFF
16 FFFF
17 8080
18 1F02
19 110A
20 120F
21 2310
22 2126
23 9205
```

Refactoring of SoC GR0040

Making the good, greater







Relevant Issues

- Internal Tri-State;
- Latch Avoidance;

Peripherals

Removal of Tri-State Internal Variables

Before: The peripherals received data through an inout port, which put the bus in tri-state mode. This bidirectional approach used the same signal for both reading and writing.

```
timer timer(
.ctrl(ctrl), .data(data),
.sel(sel[0]), .rdy(per_rdy[0]), //0x8000-0x80FF --> 255 endereços
.int_req(int_req), .i(1'b1),
.cnt_init(16'hFFFB));

pario par(
.ctrl(ctrl), .data(data),
.sel(sel[1]), .rdy(per_rdy[1]), //0x8100-0x81FF --> 255 endereços
.i(par_i), .o(par_o));
```

After: We now use distinct variables: one that holds the content to be stored (written to the peripheral) and another that collects the data from load operations (read from the peripheral).

```
assign _io_data = _sel[0] ? {8'h00, _timer_data[`PARIO_MSB:0]} :
                  _sel[1] ? {8'h00, _pario_data[`PARIO_MSB:0]} :
                  _sel[2] ? {8'h00, _uart_data[`PARIO_MSB:0]}:
                  _sel[3] ? _i2c_data:
                  16'h0000;

  timer timer(
    .i_ctrl(_ctrl), .i_sel(_sel[0]),
    .i_i(1'b1), .i_data_to_store(_data_to_store),
    .i_cnt_init(16'h0000), .o_rdy(_per_rdy[0]), //0x1000-0x10FF --> 255 endereços
    .o_data(_timer_data), .o_int_req(o_int_req[0]));

  pario par(
    .i_rst(i_rst), .i_ctrl(_ctrl), .i_sel(_sel[1]),
    .i_i_8bits(i_par_i), .i_data_to_store(_data_to_store[`PARIO_MSB:0]),
    .o_data(_pario_data), .o_rdy(_per_rdy[1]), //0x1100-0x11FF --> 255 endereços
    .o_o_8bits(o_par_o), .o_int_req(o_int_req[2:1])); //o_int_req[2:1]
```

Internal-Tri State

The High Impedance State for internal variables is **not supported in recent Verilog**. Therefore it should only be used for I/O ports, as indicated on the UG901 (Vivado Design Suite User Guide: Synthesis) and confirmed by the AMD Community Support.

Solution:

To avoid this, all attributions are performed conferring known values. Where there was need for disconnection from the circuit (High-Z state) an additional variable was created (or properly reused) and the proper handling performed

Internal-Tri State Solution

Previous

```
assign data = sum_en       ? sum : 16'bz;
assign data = ('ALU&'LOG)  ? log : 16'bz;
assign data = ('ALU&'SR)   ? sr  : 16'bz;
assign data = 'JAL         ? pc  : 16'bz;

assign data[15:8] = w_oe ? di[15:8] : 8'bz;
assign data[7:0]  = l_oe ? di[7:0]  : 8'bz;
assign data[7:0]  = h_oe ? di[15:8] : 8'bz;
assign data[15:8] = lb   ? 8'b0     : 8'bz;

assign data[7:0]  = swsb ? do[7:0]  : 8'bz;
assign data[15:8] = sb   ? do[7:0]  : 8'bz;
assign data[15:8] = sw   ? do[15:8] : 8'bz;
```

Current

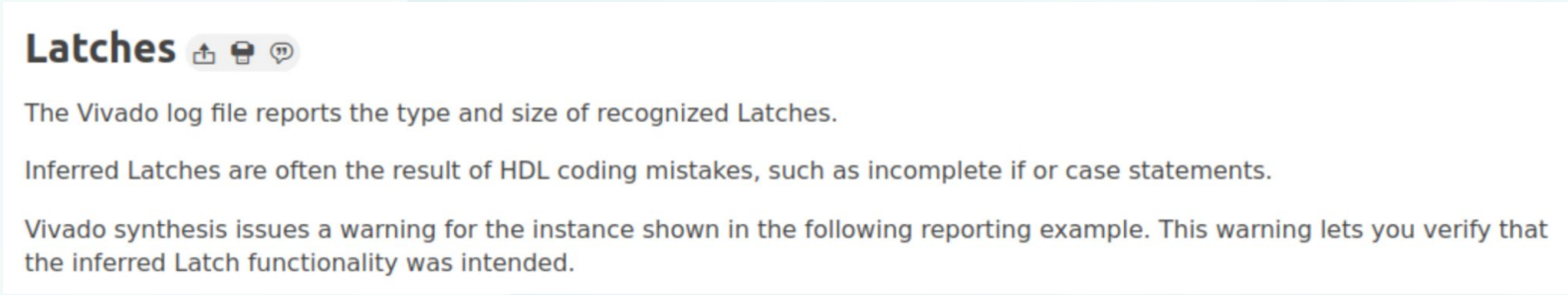
```
assign _data =
o_io_rdy ? _io_data :
i_w_oe ? {_di[15:8], _di[7:0]} :
i_l_oe ? {8'h00, _di[7:0]} :
i_h_oe ? {8'h00, _di[15:8]} :

(i_op_type_sel == 2'b00) ? _op_arithmetic_result :
(i_op_type_sel == 2'b01) ? _op_logic_result      :
(i_op_type_sel == 2'b10) ? _op_shift_result      :
/*(i_op_type_sel == 2'b11)  */ _pc;
// Default value is _pc, given the default value of
   o_op_type_sel in C.U.
```

In the Current Implementation, this is way of avoiding tri-state attribution and latch implementation.

```
assign _PC  =  (|i_int) ? i_i_ad_IVT :
                i_is_reti ? _return_addr :
               (i_is_jal) ? _op_arithmetic_result : _IRincd;
```

Latch Avoidance

Inferred latches must be avoided and it is warned by Vivado - they are a result of flawed combinational circuits which failed to address all conditions of input.



Latches

The Vivado log file reports the type and size of recognized Latches.

Inferred Latches are often the result of HDL coding mistakes, such as incomplete if or case statements.

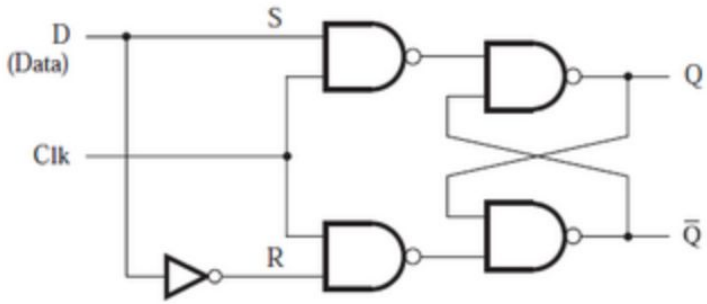
Vivado synthesis issues a warning for the instance shown in the following reporting example. This warning lets you verify that the inferred Latch functionality was intended.

Source: Vivado Design Suite User Guide: Synthesis (UG901)

Inferred latches should be avoided since they may introduce unexpected behavior, glitches and a significantly harder timing analysis and debugging.

Latches

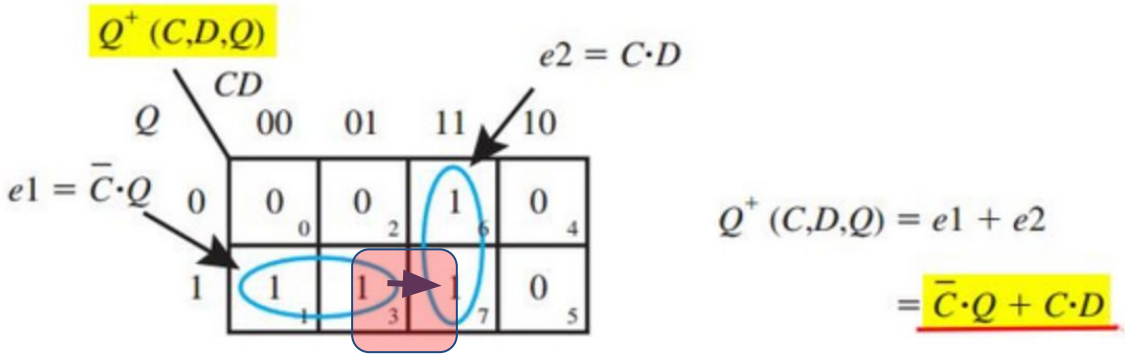
D-Latch



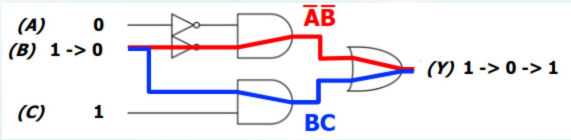
Boolean expression for $Q^+ = \overline{((\overline{D \cdot C}) + \overline{Q}) + (C \cdot \overline{D})}$

K-map reduced expression for $Q^+ = \overline{C} \cdot Q + C \cdot D$

Hazard analysed on Karnaugh Map - Glitch



Glitch Phenomenon



Adding a redundant loop solves the issue, but that does not qualify for inferred latches - the previous value of that variable is latched since there is no specification.

Refactoring

- Signal Naming Standardization;
- ISA;
- Register File;
- Memory Organization;
  - New memory implementation
- Datapath;
- Control Unit;
- Sources Structure;
  - Modules
- Peripherals;
- 3.
  - Timer/Counter;
  - Pario;
  - UART;
  - I2C master.
- Interrupts;

Signal Naming Standardization

Before

```
reg [‘AN:0] pc;           // program counter
output [‘AN:0] i_ad;      // next instruction address
output [‘AN:0] d_ad;      // data address for load/store
output [‘N:0] do;         // data to store in RAM
wire [‘N:0] dreg, sreg; //destination and source register
```

After

```
reg [‘ADDR_MSB:0] _IR;              // current instruction register
wire [‘ADDR_MSB:0] _PC;             // next instruction address
wire [‘ADDR_MSB:0] _data_address;   // data address for load/store
wire [‘DATA_MSB:0] _data_to_store; // data to write to RAM
wire [‘DATA_MSB:0] _destinationRegister;
wire [‘DATA_MSB:0] _2ndOp;

input i_clk;                        // system clock
output [‘DATA_MSB:0] o_insn;        // current instruction output
```

i_	Input Variable
o_	Output Variable
_	Internal Variable

ISA

CENTROALGORITMI

Before

SRL

15...12	11...8	7...4	3...0
0010	register destination	register source	7

SRA

15...12	11...8	7...4	3...0
0010	register destination	register source	8

0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0

1- shift right
0- shift left

1- Logic operation
0- Arithmetic operation

After

Shift Right Logic

15...12	11...8	7...4	3...0
0010	register destination	register source	7

Shift Left Arithmetic

15...12	11...8	7...4	3...0
0010	register destination	register source	8

Shift Right Arithmetic

15...12	11...8	7...4	3...0
0010	register destination	register source	9

Shift Left Logic

15...12	11...8	7...4	3...0
0010	register destination	register source	10

```
assign o_shift_result =
    i_shift_op_sel[0] ?                                  // RIGHT shift
        (i_shift_op_sel[1] ?
            {1'b0, i_a[`DATA_MSB:1]} :                   // LOGICAL
               right
            {i_a[`DATA_MSB], i_a[`DATA_MSB:1]}) :        // ARITHMETIC
               right
        {i_a[`DATA_MSB-1:0], 1'b0};                      // LEFT shift
```

ISA

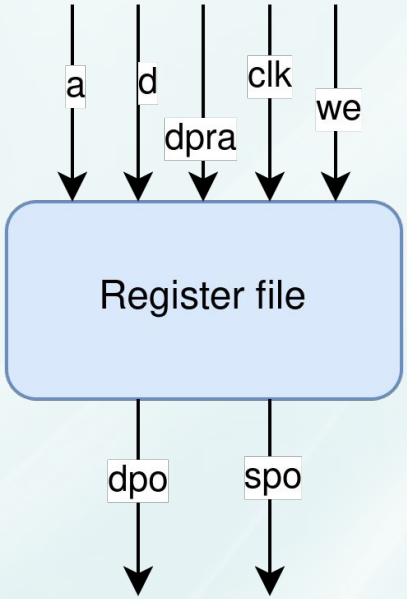
return_addr stores the instruction register value at the moment an interrupt occurs


```
// In control unit:
`define RETI    (_op==10)
assign o_is_reti = o_valid & `RETI;

// In datapath:
reg [`ADDR_MSB:0] _return_addr;
always @(posedge i_clk) begin
    if (i_rst)
        _return_addr <= 16'h0000;
    else if (|i_int)
        _return_addr <= _IR;
end

assign _PC   = (|i_int) ? i_i_ad_IVT :
               i_is_reti ? _return_addr :
               (i_is_jal) ? _op_arithmetic_result : _IRincd;
```

Register File

Implemented using the Distributed Memory Generator IP.

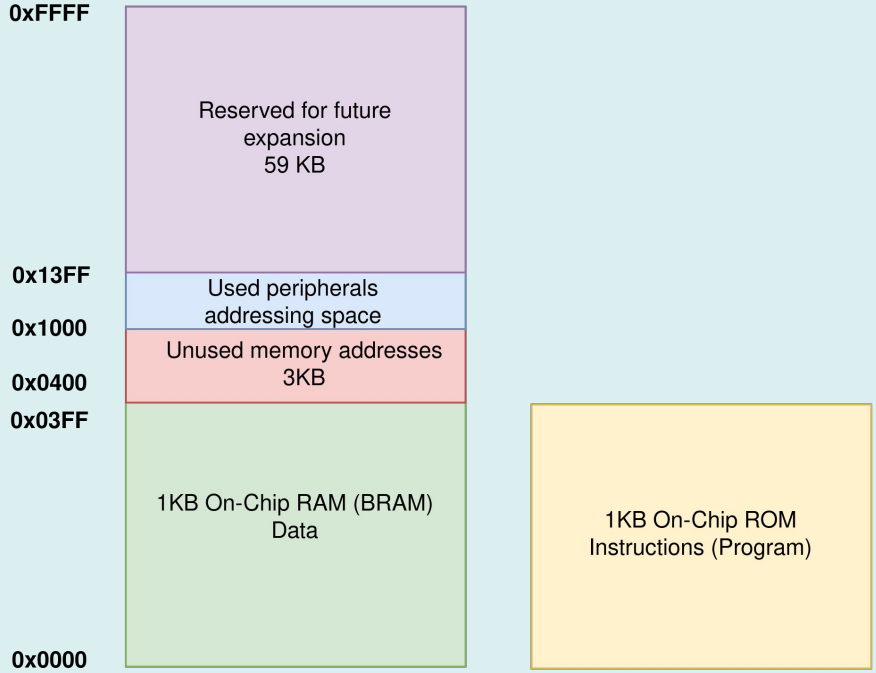
Distributed memory maps directly onto lookup tables (LUTs), allowing for low-latency read and write operations.




```
regfile16x16 regfile(
    .a(_rd),
    .d(_data),
    .dpra(i_isRI ? _rd : _rs),
    .clk(i_clk),
    .we(i_rf_we),
    .spo(_destinationRegister),
    .dpo(_2ndOp)
  );
```

Memory

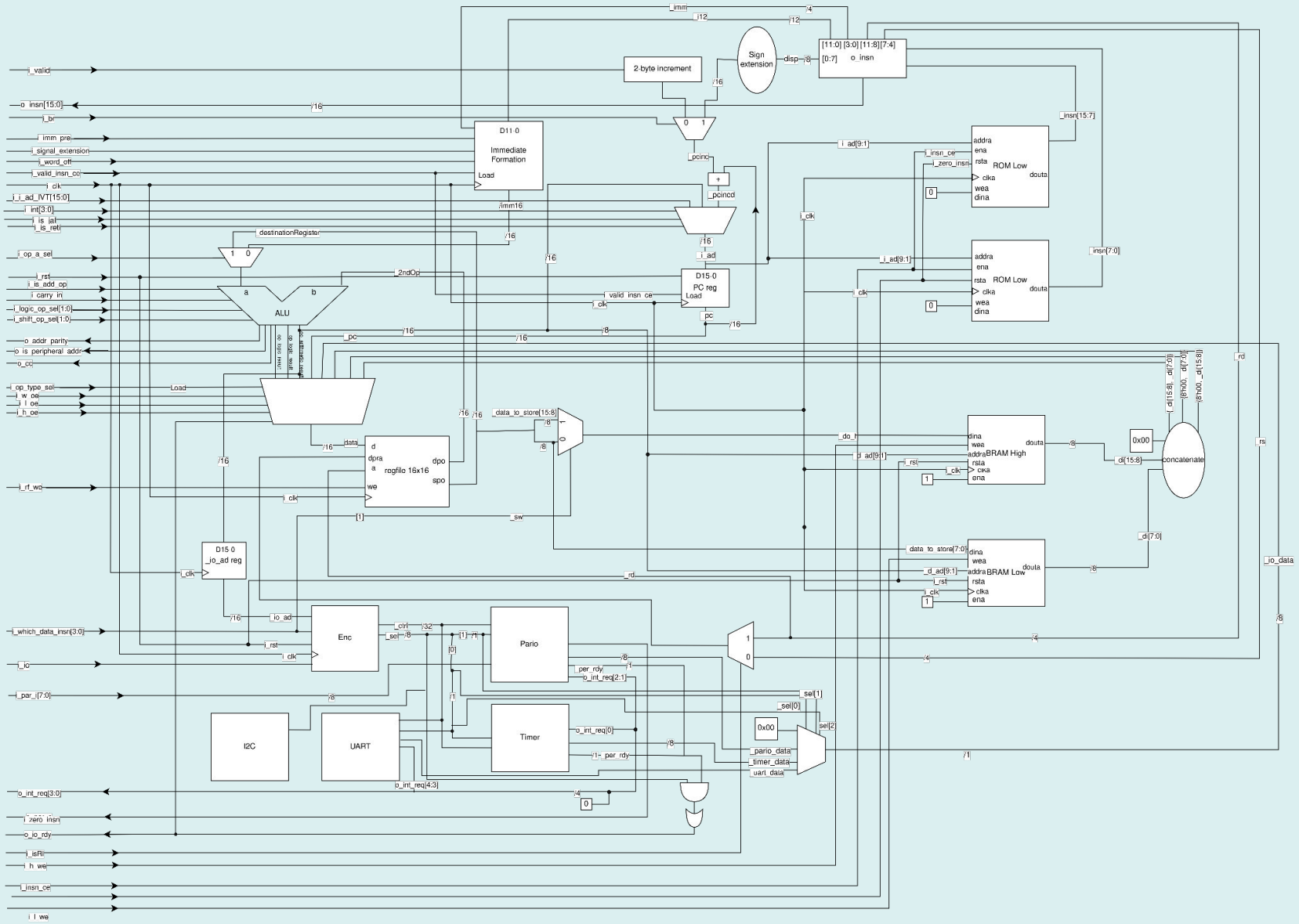
- **Pure Harvard**-style memory organization, in which instruction and data memories are **physically separated**.
  Instructions stored in **ROM**, therefore cannot be altered in runtime.
- **Data** is stored in **RAM**.
- Simple and Efficient implementation consider the processor at hand: no Von-Neumann bottleneck and no virtually separated memory.
- Peripheral addresses have been optimized to lower addresses that before for simpler memory organization.



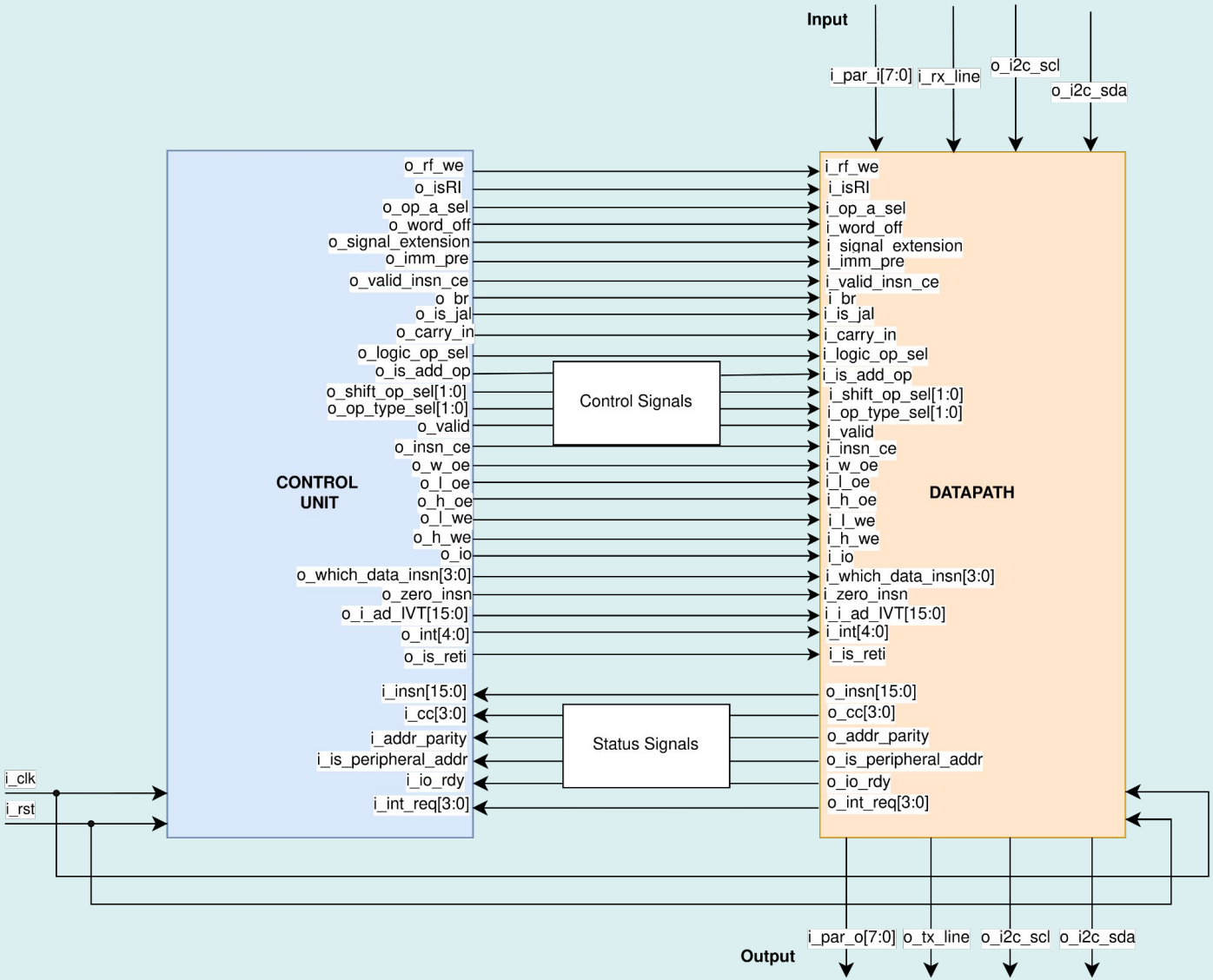

```
ROM_8_512_H romh_insn(
    .rsta(i_zero_insn),
    .ena(i_insn_ce),
    .clka(i_clk),
    .addra(_PC[9:1]),
    .douta(o_insn[15:8]));

ROM_8_512_L roml_insn(
    .rsta(i_zero_insn),
    .ena(i_insn_ce),
    .clka(i_clk),
    .addra(_PC[9:1]),
    .douta(o_insn[7:0]));

BRAM_8_512_H ramh_data(
    .rsta(i_rst),
    .wea(i_h_we),
    .ena(1'b1),
    .clka(i_clk),
    .addra(_data_address[9:1]),
    .dina(_do_h),
    .douta(_di[15:8]));

BRAM_8_512_L raml_data(
    .rsta(i_rst),
    .wea(i_l_we),
    .ena(1'b1),
    .clka(i_clk),
    .addra(_data_address[9:1]),
    .dina(_data_to_store[7:0]),
    .douta(_di[7:0]));
```

Datapath

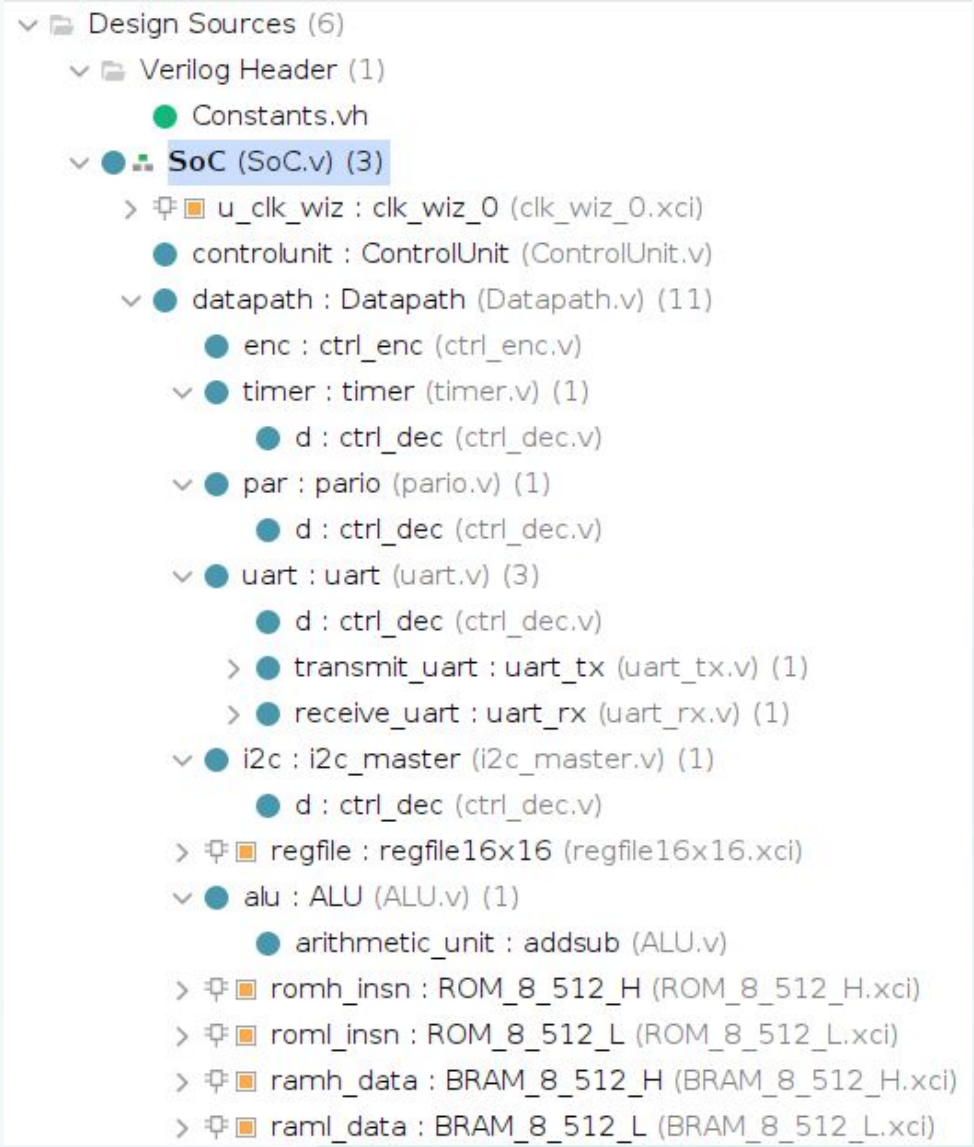
CENTROALGORITMI



Control Unit



Sources Structures



Modules


```
module SoC(
    input i_clk,
    input i_rst,
    input [`PARIO_MSB:0] i_par_i,
    input i_rx_line,
    output [`PARIO_MSB:0] o_par_o,
    output o_tx_line,
    inout o_i2c_scl,
    inout o_i2c_sda);
```

```
Datapath datapath (
 .i_clk(_clk50),
 .i_rst(_rst_sys),
 .i_rf_we(_rf_we),
 .i_isRI(_isRI),
 .i_op_a_sel(_op_a_sel),
 .i_word_off(_word_off),
 .i_signal_extension(_signal_
 extension),
 .i_imm_pre(_imm_pre),
 .i_valid_insn_ce(_valid_insn_ce)
 .i_br(_br),
 .i_is_jal(_is_jal),
 .i_carry_in(_carry_in),
 .i_logic_op_sel(_logic_op_sel),
 .i_is_add_op(_is_add_op),
 .i_shift_op_sel(_shift_op_sel),
 .i_op_type_sel(_op_type_sel),
 .i_valid(_valid),
 .i_insn_ce(_insn_ce),
 .i_w_oe(_w_oe),
 .i_l_oe(_l_oe),
 .i_h_oe(_h_oe),
 .i_l_we(_l_we),
 .i_h_we(_h_we),
 .i_io(_io),
 .i_which_data_insn(_which
 _data_insn),
 .i_par_i(w_par_i),
 .i_zero_insn(_zero_insn),
 .i_i_ad_IVT(_i_ad_IVT),
 .i_int(_int),
 .i_rx_line(w_rx_line),
 .i_is_reti(_is_reti),
 .o_insn(_insn),
 .o_cc(_cc),
 .o_addr_parity(_addr_parity),
 .o_is_peripheral_addr(_is_
 peripheral_addr),
 .o_io_rdy(_io_rdy),
 .o_par_o(w_par_o),
 .o_int_req(_int_req),
 .o_tx_line(_tx_line)
);
```

```
ControlUnit controlunit (
 .i_clk(_clk50),
 .i_rst(_rst_sys),
 .i_insn(_insn),
 .i_cc(_cc),
 .i_addr_parity(_addr_parity),
 .i_is_peripheral_addr(_is_
 peripheral_addr),
 .i_io_rdy(_io_rdy),
 .i_int_req(_int_req),
 .o_valid_insn_ce(_valid_insn_ce)
 .o_isRI(_isRI),
 .o_op_a_sel(_op_a_sel),
 .o_rf_we(_rf_we),
 .o_word_off(_word_off),
 .o_signal_extension(_signal_
 extension),
 .o_imm_pre(_imm_pre),
 .o_br(_br),
 .o_is_jal(_is_jal),
 .o_carry_in(_carry_in),
 .o_logic_op_sel(_logic_op_sel),
 .o_shift_op_sel(_shift_op_sel),
 .o_op_type_sel(_op_type_sel),
 .o_valid(_valid),
 .o_insn_ce(_insn_ce),
 .o_is_add_op(_is_add_op),
 .o_w_oe(_w_oe),
 .o_l_oe(_l_oe),
 .o_h_oe(_h_oe),
 .o_l_we(_l_we),
 .o_h_we(_h_we),
 .o_io(_io),
 .o_which_data_insn(_which_
 data_insn),
 .o_zero_insn(_zero_insn),
 .o_i_ad_IVT(_i_ad_IVT),
 .o_int(_int),
 .o_is_reti(_is_reti)
);
```

Peripherals

To simplify peripheral interaction a **control bus encoder** and **decoder** modules were used. They allow for simple encoding and decoding of all relevant control signals.

The **implemented peripherals** were:

- Timer/Counter;
- Pario;
- UART (Tx/Rx);
- I2C Master.

Peripherals **addresses** start at 0x1000 and go until 0x13FF, where each peripheral is given 0xFF memory spaces (256 bytes).

For each peripheral, **SFRs** were specified for clear handling.

Peripherals
General Implementation

Any peripheral has, at least, this implementation.

- Inputs/Outputs
- SFRs definition
- Register Instantiation

```
/* Peripheral Module Instantiation */
module peripheral (
    input  [`CONTROL_MSB:0] i_ctrl,          // Peripheral Control Bus
    input i_sel,                             // Peripheral Select
    output o_rdy,                            // Feedback Signal
    input  [`DATA_MSB:0] i_data_to_store,    // Data to Store in the Peripheral at a specified address
    output reg [`DATA_MSB:0] o_data_out,     // Data to Load from the Peripheral at a specified address
    );

    /* *PERIPHERAL* SFRs */
    parameter REG_1 = 16'h1200;      // RO
    parameter REG_2 = 16'h1202;      // RW
    parameter REG_3 = 16'h1204;      // WO

    /* Actual Registers */
    reg [`DATA_MSB:0] _reg_1;
    reg [`DATA_MSB:0] _reg_2;
    reg [`DATA_MSB:0] _reg_3;
```

Peripherals
Encoding/Decoding Control Bus

The **encoding** process is performed in the datapath module.

```
ctrl_enc enc(
   .i_clk(i_clk), .i_rst(i_rst), .i_io(i_io),
   .i_io_ad(_io_ad), .i_lw(_lw), .i_lb(_lb), .i_sw(_sw),
   .i_sb(_sb), .o_ctrl(_ctrl), .o_sel(_sel));
```

The **decoding** process happens in all peripherals and complies with the following standard structure.

```
/****************************************************************************
 * CONTROL BUS DECODING
 ****************************************************************************/

  /* Decode Module Required Signals */
   wire _clk, _rst;
   wire [3:0] _oe, _we;
   wire [`ADDR_MSB_P:0] _ad;

   /* Decoding the control bus */
   ctrl_dec d(
       .i_ctrl(i_ctrl),
       .i_sel(i_sel),      // Used for _oe and _we activation
       .o_clk(_clk),       // Sys  tem Clock
       .o_rst(_rst),       // System Reset
       .o_oe(_oe),         // Peripheral Load Enable:  bit 0 -> (i_lw | i_lb) | bit 1 -> i_lw
       .o_we(_we),         // Peripheral Store Enable: bit 0 -> (i_sw | i_sb) | bit 1 -> i_sw
       .o_ad(_ad)          // 8-bit Peripheral Address
       );
```

All peripherals must receive **i_ctrl** as a module's input.

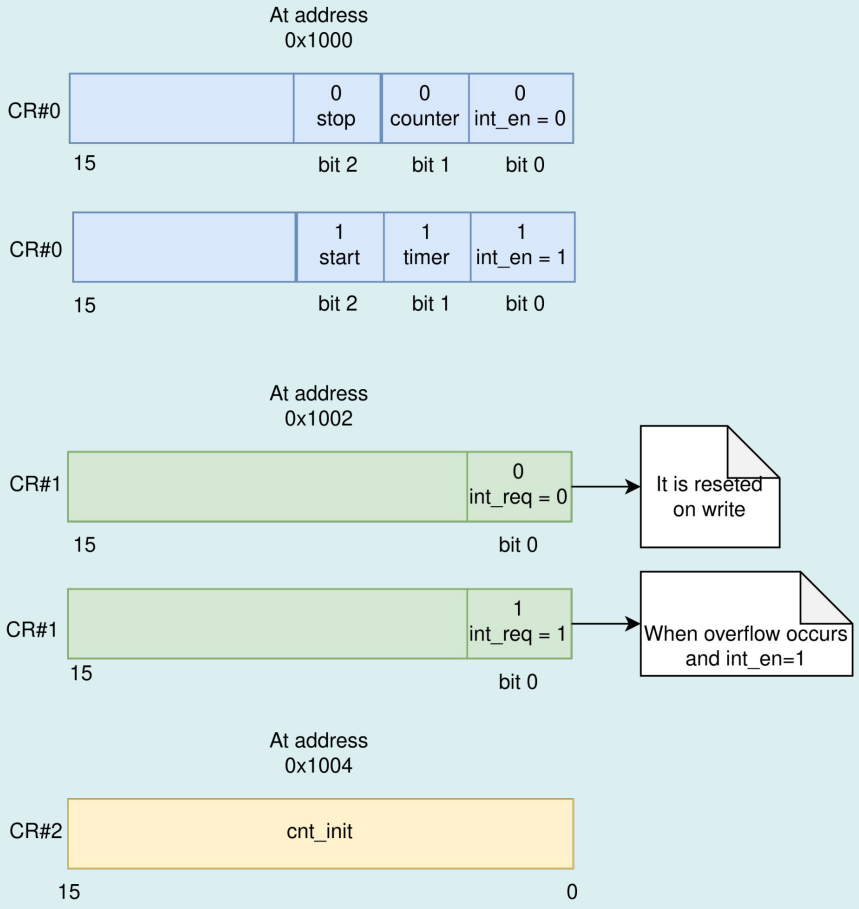
Peripherals
General Implementation

- Read/Write operations

```
/*****************************************************************************
 * Accessing the SFRs
 *****************************************************************************/

 // _reg_1 is internally changed by some variable

// WRITE (STORE) LOGIC
always @(posedge _clk) begin
    if (_rst) begin
        _reg_2 <= `DATA_BUS'b0;
        _reg_3 <= `DATA_BUS'b0;
    end else if (_we[0]) begin
        case (_ad [`ADDR_MSB_P:0])
            REG_2[7:0]: _reg_2 <= i_data_to_store;
            REG_3[7:0]: _reg_3 <= i_data_to_store;
            default: ;
        endcase
    end
end

// READ (LOAD) LOGIC
always @(*) begin
   always @(*) begin
        if (_oe[0]) begin
            case (_ad [`ADDR_MSB_P:0])
                REG_1[7:0]: o_data_out = _reg_1;
                REG_2[7:0]: o_data_out = _reg_2;
                default:    o_data_out = 16'h0;
            endcase
        end else begin
            o_data_out = 16'h0;
        end
    end
end
```

Timer

SFRs



- 16-bit Timer/Counter
- Interrupt on Timer overflow
- 1 interrupt line

```
/* TIMER SFRs */
parameter CR_0 = 16'h1000;      // WO
parameter CR_1 = 16'h1002;      // WO
parameter CR_2 = 16'h1004;      // WO

/* Actual Registers */
reg [`DATA_MSB:0] _cr_0;
reg [`DATA_MSB:0] _cr_1;
reg [`DATA_MSB:0] _cr_2;
```

Timer


```
/*****************************************************************************
 * REGISTERS CONFIGURATION
 *****************************************************************************/

    // Update register values based on current control signals
    always @ (*) begin
        _cr_0 = {13'b0, _start, _timer, _int_en};
        _cr_1 = {15'b0, o_int_req};
        _cr_2 = _cnt_init_reg;
    end

    always @(posedge _clk) begin
        if (_rst) begin
            {_start, _timer, _int_en} <= 3'b000;
            o_int_req <= 1'b0;
            _cnt_init_reg <= i_cnt_init;
        end
        else begin
            //CR#1 interrupt request
            if (_tick && _v && _int_en)
            o_int_req <= 1'b1;

            // Handle writes based on address
            if (_we[0]) begin
                case (_ad)
                    // 0x1000 - Counter Control Register
                    CR_0[7:0]:{_start, _timer, _int_en} <= i_data_to_store[2:0];
                    // 0x1002 - Interrupt Request Register
                    CR_1[7:0]: o_int_req <= 1'b0;  // Clear on write
                    // 0x1004 - Counter Init Register
                    CR_2[7:0]:  _cnt_init_reg <= i_data_to_store;
                    default: begin
                    end
                endcase
            end
        end
    end
```

```
/* TIMER SFRs */
parameter CR_0 = 16'h1000;      // WO
parameter CR_1 = 16'h1002;      // WO
parameter CR_2 = 16'h1004;      // WO

/* Actual Registers */
reg [`DATA_MSB:0] _cr_0;
reg [`DATA_MSB:0] _cr_1;
reg [`DATA_MSB:0] _cr_2;
```

Pario

- Pario module interfaces external pins/LEDs/buttons.
- Its implementation did not change much from the already existing: only **interrupts on falling edge were added**.
- 2 interrupt lines

```
/*****************************************************************************
 * INTERRUPT REQUEST DETECTION - 2 interrupt Lines
 *****************************************************************************/

   if (i_i_8bits[0] == 1 && ~last_bit[0])
       o_int_req[0] <=  1;
   else
       o_int_req[0] <= 0;

   if (i_i_8bits[1] == 1 && ~last_bit[1])
       o_int_req[1] <= 1;
   else
       o_int_req[1] <= 0;

   last_bit <= i_i_8bits[1:0];

  end
```

UART Peripheral

- Implements UART protocol (full-duplex).
- Implements 2 interrupt lines.

SFRs

Register	Address	Access	Description
SBUF_RX	0x1200	Read Only	Receive buffer register
SBUF_TX	0x1201	Write Only	Transmit buffer register
BAUDRATE_DIV	0x1203	Write Only	Baud rate divider
SCON	0x1205	Write Only	Serial control flags (TI, RI)

Bit	Name	Access	Reset	Description
7:2	–	–	0	Reserved. Read as zero.
1	TI	HW set	0	Transmit Interrupt flag. Set by hardware when a transmission completes (after the stop bit). This flag can be used to trigger an interrupt or indicate that the transmitter is ready for a new byte.
0	RI	HW set	0	Receive Interrupt flag. Set by hardware when a full byte has been successfully received and stored in the receive buffer. This flag indicates that valid data is available for software to read.

```
/* *UART* SFRs */
parameter SBUF_RX =          16'h1200;
parameter SBUF_TX =          16'h1201;
parameter BAUDRATE_DIV =     16'h1203;
parameter SCON =             16'h1205;

parameter RI_BIT = 0;
parameter TI_BIT = 1;
```

UART Peripheral

Based on three modules:

- Top module uart
  - Module uart_tx
  - Module uart_rx

Top Module UART

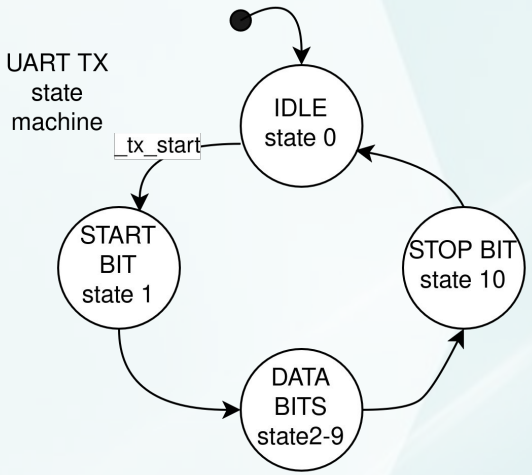
- Register store/load;
- Store data in SBUF_Rx;
- Generate interrupts requests for both UART Tx and UART Rx.

```
/*****************************************************
 * INTERNAL MODULES
 *****************************************************

    uart_tx #(.SBUF_TX(SBUF_TX)) transmit_uart(
        .i_ctrl(i_ctrl), .i_sel(i_sel),
        .i_baudrate_div(_baudrate_div),
        .i_data(/*_rx_data*/ i_data_to_store),
        .o_rdy(per_tx_rdy),
        .o_tx_out(o_tx_line),
        .o_tx_done(_tx_done));

    uart_rx receive_uart(
        .i_ctrl(i_ctrl), .i_sel(i_sel),
        .i_baudrate_div(_baudrate_div),
        .i_rx_in(i_rx_line),
        .o_rdy(per_rx_rdy),
        .o_data(_rx_data),
        .o_data_valid(_rx_valid_data));

    assign o_rdy = per_tx_rdy | per_rx_rdy;
```

UART Peripheral

TX State Machine Description:

1. Idle state with TX line held high
2. Start bit transmission
3. Sequential transmission of data bits
4. Stop bit transmission
5. Return to idle and interrupt generation

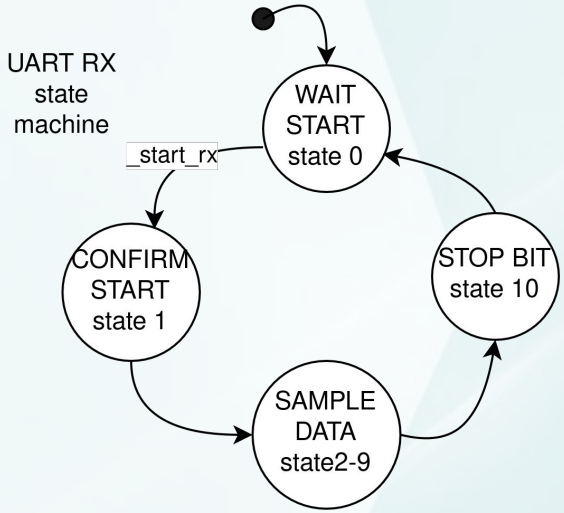



```
/**************************************************************************
 * RUNTIME CONFIGURATIONS
 **************************************************************************/
  always @(posedge _clk) begin
      if (~_rst) begin
          case (_state_tx)
              0: begin  // Idle
                  o_tx_out <= 1;  // High when idle
                  if (_tx_start) begin
                      _shift_reg <= _sbuf_tx;
                      _state_tx <= 1;
                      _counter <= 0;
                  end
              end
              1: begin  // Start bit (0)
                  o_tx_out <= 0;
                  if (_counter == i_baudrate_div - 1) begin
                      _counter <= 0;
                      _state_tx <= 2;
                  end else _counter <= _counter + 1;
              end
              2, 3, 4, 5, 6, 7, 8, 9: begin  // Data bits (LSB first)
                  o_tx_out <= _shift_reg[_state_tx - 2];
                  if (_counter == i_baudrate_div - 1) begin
                      _counter <= 0;
                      _state_tx <= _state_tx + 1;
                  end else _counter <= _counter + 1;
              end
              10: begin  // Stop bit (1)
                  o_tx_out <= 1;
                  if (_counter == i_baudrate_div - 1) begin
                      _counter <= 0;
                      _state_tx <= 0;
                  end else _counter <= _counter + 1;
              end
          endcase
      end
      else
          o_tx_out <= 1;
  end

  assign o_tx_done = (_state_tx == 10 && _counter == i_baudrate_div - 1);
```

UART Peripheral

RX State Machine Description:

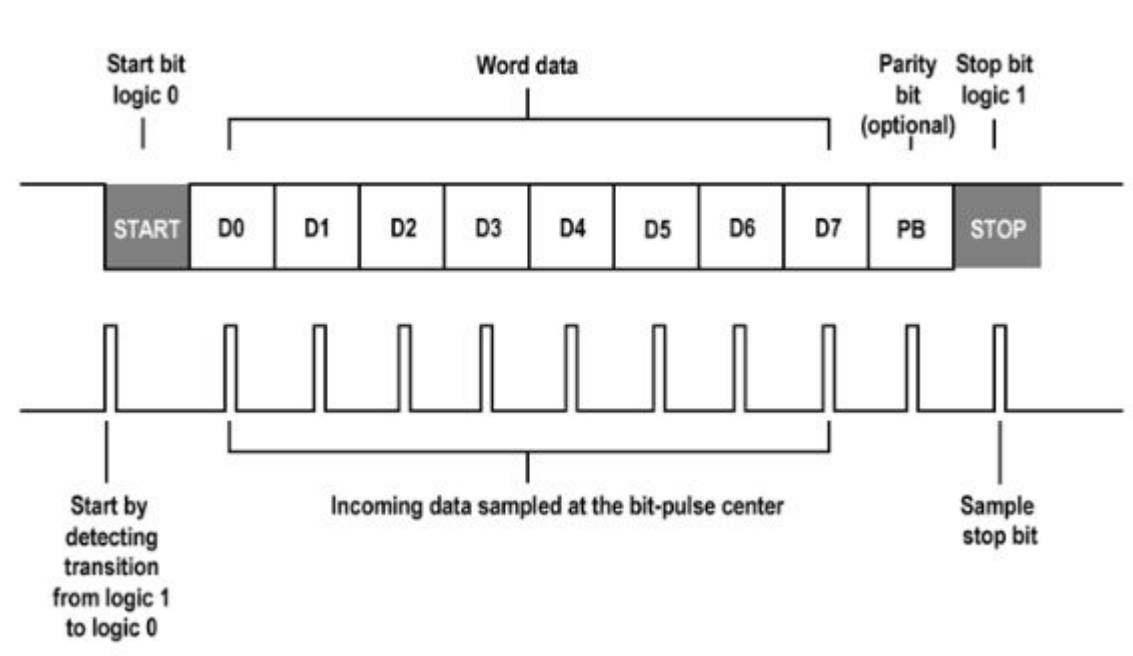
1. Idle state monitoring for a start bit
2. Start bit validation using half-baud delay
3. Sequential sampling of data bits
4. Stop bit verification
5. Data ready signaling and interrupt generation




```
/**************************************************************************************
 * REGISTERS AND FLAG CONFIGURATION
 **************************************************************************************
    always @(posedge _clk) begin
        case (_state_rx )
            0: begin  // Wait for start bit (falling edge)
                if (!i_rx_in) begin
                    _counter <= 0;
                    _state_rx  <= 1;
                end
            end
            1: begin  // Sample mid-start bit
                if (_counter == _half_baudrate_div - 1) begin
                    if (!i_rx_in) begin  // Confirm start bit
                        _counter <= 0;
                        _state_rx  <= 2;
                    end else _state_rx  <= 0;  // False start
                end else _counter <= _counter + 1;
            end
            2, 3, 4, 5, 6, 7, 8, 9: begin  // Sample data bits
                if (_counter == i_baudrate_div - 1) begin
                    _shift_reg[_state_rx  - 2] <= i_rx_in;  // Store bit
                    _counter <= 0;
                    _state_rx  <= _state_rx  + 1;
                end else _counter <= _counter + 1;
            end
            10: begin  // Stop bit
                if (_counter == i_baudrate_div - 1) begin
                    _counter <= 0;
                    _state_rx  <= 0;
                end else _counter <= _counter + 1;
            end
        endcase
    end

    assign o_data = _shift_reg;
    assign o_data_valid = (_state_rx  == 10 && _counter == i_baudrate_div - 1);
```

UART Peripheral

- Data Sampling
- Start/Stop conditions
- Sent Data



I2C Master

- Implements I2C Protocol, for the I2C Master.

SFRs

Register	Address	Access	Description
I2C_DIVIDER	0x1300	Write Only	Clock divider for SCL generation
I2C_DATA	0x1302	Write Only	Address and data payload register
I2C_CONTROL	0x1304	Write Only	Control bits (Enable, Start)

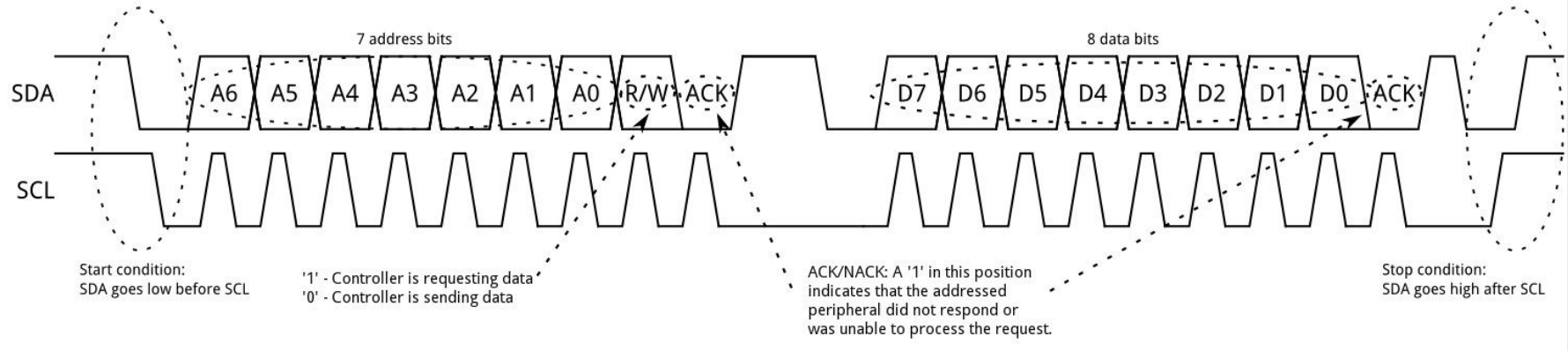
```
/* I2C SFRs */
parameter I2C_DIVIDER = 16'h1300;
parameter I2C_DATA = 16'h1302;

parameter FIRST_FRAME = 7;

parameter I2C_CONTROL = 16'h1304;

    // I2C_CONTROL bits
    parameter bit_EN = 0;
    parameter bit_START = 1;
```

I2C_DATA Register Specification

Bit	Name	Access	Reset	Description
15:8	DATA	WO	0	Data byte to be transmitted or buffer for received data during read operations.
7:0	ADDR	WO	0	7-bit slave address with the least significant bit indicating read/write operation.

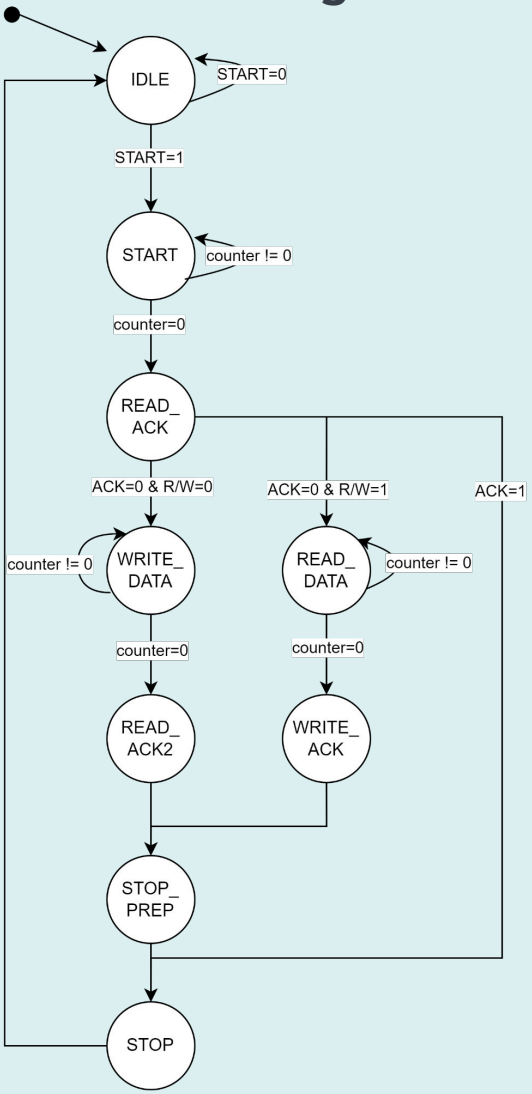
I2C Master

- 7-bit Address + R/!W bit
- 8-bit Data send
- Start/Stop Conditions
- ACK/NACK reception



I2C Master

CLK Positive Edge FSM



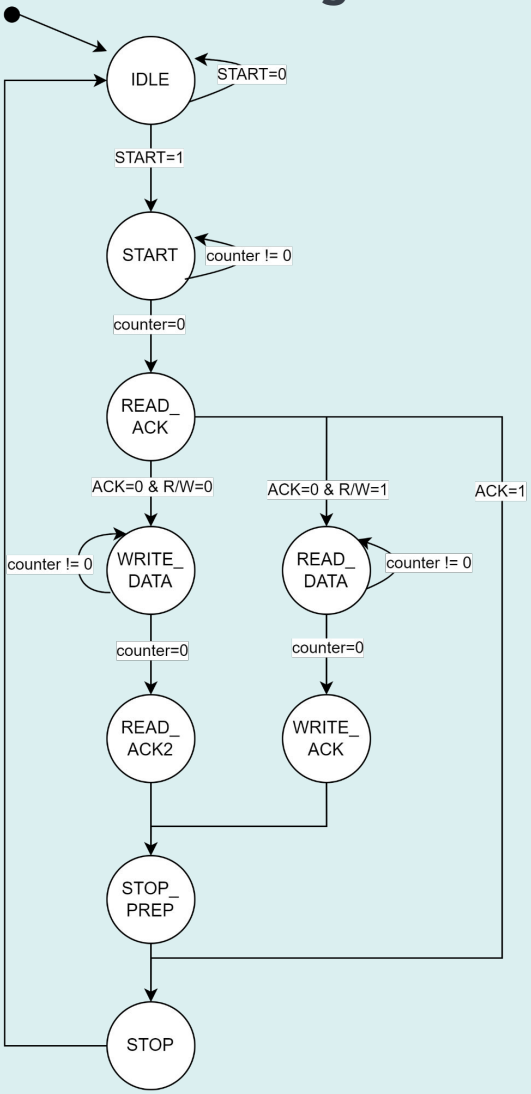
On CLK Negative Edge, data on SDA is updated, based on the stored value introduced in the instructions.

The I2C clock is derived from the system clock using a programmable divider. The divider value determines the SCL frequency and allows selection of multiple I2C speed modes at compile time or runtime

```
// Communication Frequency
`ifdef FAST_MODE     // 400kps
    `define DIVIDER_RST 16'h007D
`elsif FAST_MODE_PLUS // 1Mbps
    `define DIVIDER_RST 16'h0032
`elsif HIGH_SPEED_MODE // 3.4Mbps
    `define DIVIDER_RST 16'h000F
`elsif ULTRA_FAST_MODE // 5Mbps
    `define DIVIDER_RST 16'h000A
`else
   `define DIVIDER_RST 16'h01F4      // Default is STANDART MODE - 100kbps
`endif
```

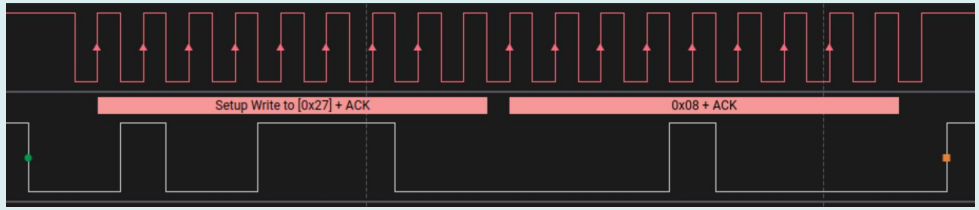
I2C Master

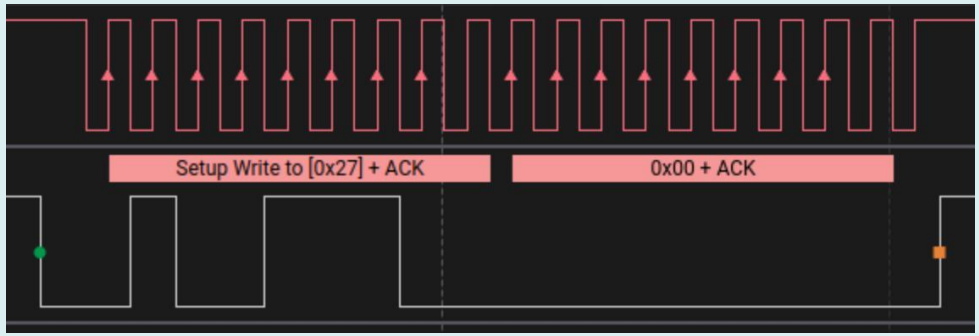
CLK Positive Edge FSM



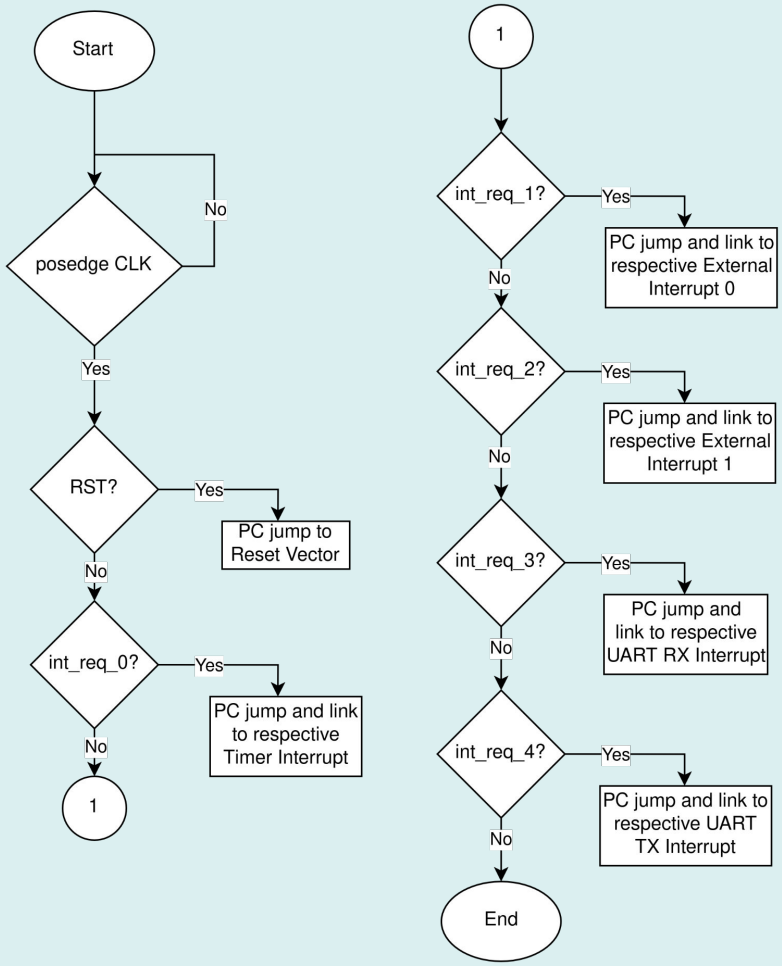
The current implementation supports 7 bits address + R/!W bit + data byte.

No multiple data bytes in the same frame are supported.





Interrupt Controller

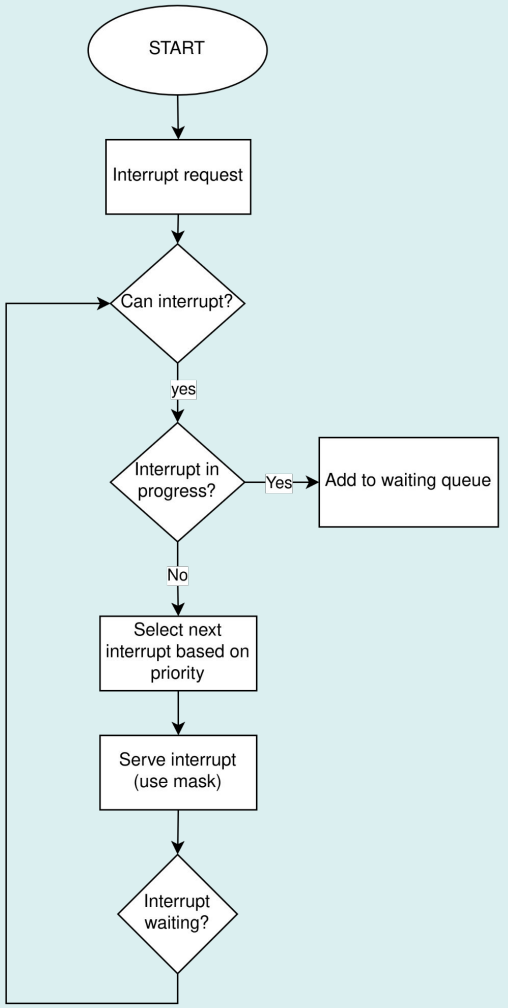


- **Priority Handling:** Interrupts are processed based on a hardwired priority. If multiple requests occur simultaneously, the ***"Priority If-Else"*** logic determines which source is serviced first.
- **Jump-and-Link:** The jump to the ISR and the linking process (saving the return address on *R0*) are entirely hardwired.
- **Address Space:** Interrupt handling and program data are mapped within the first 1 kB of on-chip RAM (BRAM), specifically in the range 0x0000 to 0x03FF.

Interrupt Controller

Approach for the **IVT** based on the 8051 uC:

- **Priority-based servicing:** Like the 8051, interrupts are assigned fixed priorities based on their source, with lower-indexed interrupts having higher priority.
- **Interrupt queuing:** Pending interrupts are queued while an interrupt service routine is executing, similar to the 8051's interrupt pending flags.
- **Non-maskable execution:** Once an interrupt begins servicing, it runs to completion (until RETI) before another interrupt can be serviced, preventing interrupt nesting.
- **Atomic instruction protection**: Certain instruction sequences (the interrupts) cannot be interrupted, maintaining data integrity during critical operation.



Priority Encoding Flowchart

Interrupt Controller

IVT - Interrupt Vector Table

Interrupt Source	ISR Address (Hex)	Description
Reset	0x0002	Reset Vector (Jumps to 0x0054)
Timer	0x0004	Timer/Counter Interrupt
Extern 0	0x0014	External Interrupt Line 0
Extern 1	0x0024	External Interrupt Line 1
UART_Rx	0x0034	Signals Reception on UART
UART_Tx	0x0044	Signals Transmission on UART

Each interrupt can have 16 bytes at most (8 instructions).

Interrupt Stages

1. Interruption Prologue: A hardwired sequence initiated upon an interrupt trigger.

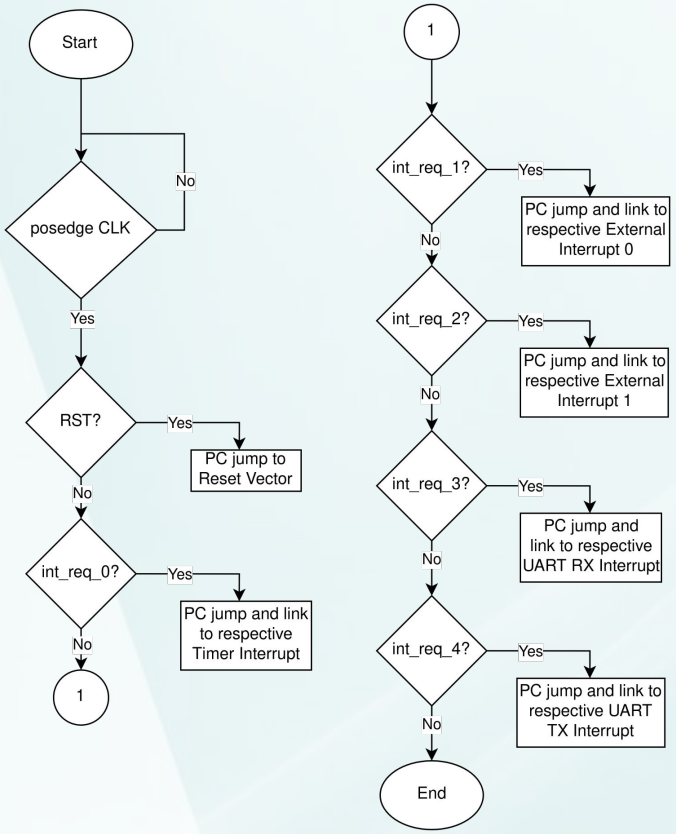
2. Vector Jump: The Program Counter (PC) automatically jumps to a fixed address corresponding to the specific interrupt source in the Interrupt Vector Table (IVT).

3. ISR Execution: The processor executes the Interrupt Service Routine (ISR) handler.

4. Interruption Epilogue: A hardwired sequence that restores the PC to its preinterruption state, allowing normal program execution to resume.

Interrupt Controller

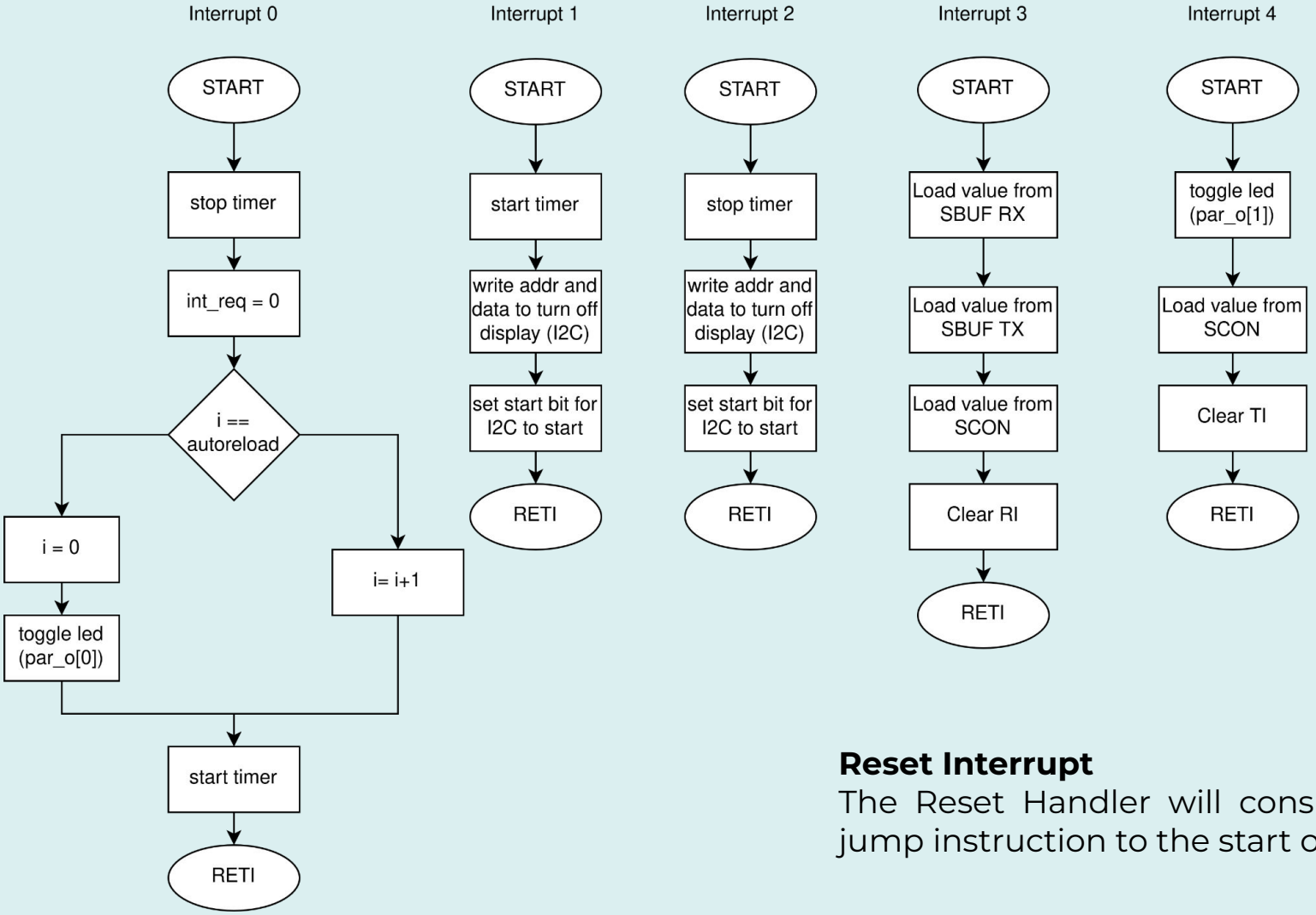
Priority Encoding




```
assign _int_index_next =   _int_next[0] ? 4'b0000:
                         (_int_next[1] ? 4'b0001 :
                         (_int_next[2] ? 4'b0010 :
                         (_int_next[3] ? 4'b0011 :
                         (_int_next[4] ? 4'b0100 : 4'b0000))));
```

Interrupt Controller

Algorithms



Reset Interrupt

The Reset Handler will consists only on a jump instruction to the start of the code.

Project's Steps

RTL Code - Verilog	
Behavioral Simulation	Verify Code Functionality
Synthesis: Convert the RTL into board's family primitives (abstract from the board)	
Post-Synthesis Simulation	Verify Code Functionality bearing the characteristics of the board family's primitives.
Implementation: Convert the code into the actual FPGA/board primitives	
Post-Implementation Simulation	Verify Code Functionality bearing the characteristics of the specific board primitives - mapped onto the board: connection delays are taken into account. Might not change from Post-Synthesis.

System Validation and Testing

Test Program

- The simulation test case will follow this steps:
  - Send 0x4E (addr) and 0x08(data) through I2C
  - Start the timer
  - Receive 0x85 in uart RX and echo it through interrupt using uart TX
  - Stop timer

Preloaded Register Usage

Register	Initial Value	Purpose
$r15$	769	Auto-reload value for the timer (2-second period)
$r14$	0	Counter used to track timer overflows
$r13$	1	Constant value used for increments
$r12$	1000	Base address of the timer peripheral
$r11$	1100	Base address of the parallel I/O peripheral
$r10$	0	–
$r9$	–	Value written to `par_o` to toggle or control outputs
$r7$	7	Control value used to start the timer
$r6$	3	Control value used to stop the timer
$r5$	1200	Base address of the UART peripheral
$r4$	1300	Base address of the I2C peripheral

- To simplify program development and interrupt handling
- Reduces number of instructions

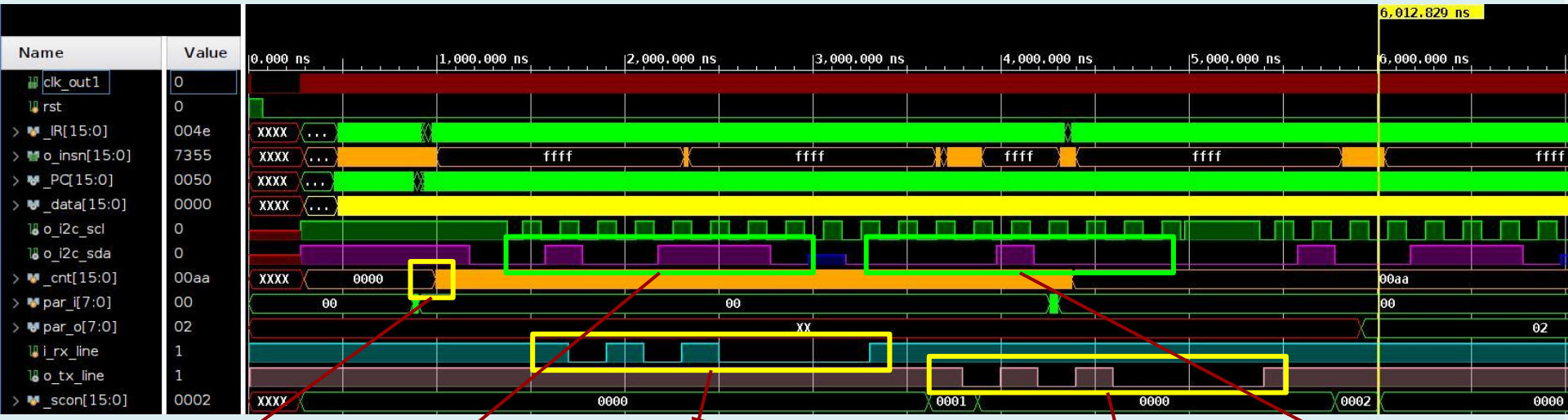
Test Bench


```
`timescale 1ns / 1ps

module tb_soc();

 reg clk;
 reg rst;
 reg [7:0] par_i;
 wire [7:0] par_o;
 reg i_rx_line;
 reg [7:0] rx_message;
 wire o_tx_line;
 wire o_i2c_scl;
 wire o_i2c_sda;

 SoC uut(
    .i_clk(clk), .i_rst(rst), .i_par_i(par_i),
    .i_rx_line(i_rx_line), .o_par_o(par_o),
    .o_tx_line(o_tx_line), .o_i2c_scl(o_i2c_scl),
    .o_i2c_sda(o_i2c_sda));

 initial clk = 0;
 always #4 clk = ~clk;

 initial begin
 rst = 1;
 par_i = 0;

 rx_message = 8'h85;
 i_rx_line = 1'b1;

 repeat (10) @(posedge clk);
 rst = 0;
```

```
//start timer
repeat (100) @(posedge clk);
par_i = 8'b01; // par_i[0]
repeat (2.5) @(posedge clk);
par_i = 8'b00;

repeat(100) @(posedge clk);
 i_rx_line = 0;
repeat (25) @(posedge clk);
i_rx_line = rx_message[0];
repeat (25) @(posedge clk);
i_rx_line = rx_message[1];
repeat (25) @(posedge clk);
i_rx_line = rx_message[2];
repeat (25) @(posedge clk);
i_rx_line = rx_message[3];
repeat (25) @(posedge clk);
i_rx_line = rx_message[4];
repeat (25) @(posedge clk);
i_rx_line = rx_message[5];
repeat (25) @(posedge clk);
i_rx_line = rx_message[6];
repeat (25) @(posedge clk);
i_rx_line = rx_message[7];
repeat (25) @(posedge clk);
i_rx_line = 1;

 //stop timer
repeat (95) @(posedge clk);
par_i = 8'b10; // par_i[1]
repeat (5) @(posedge clk);
par_i = 8'b0;
```

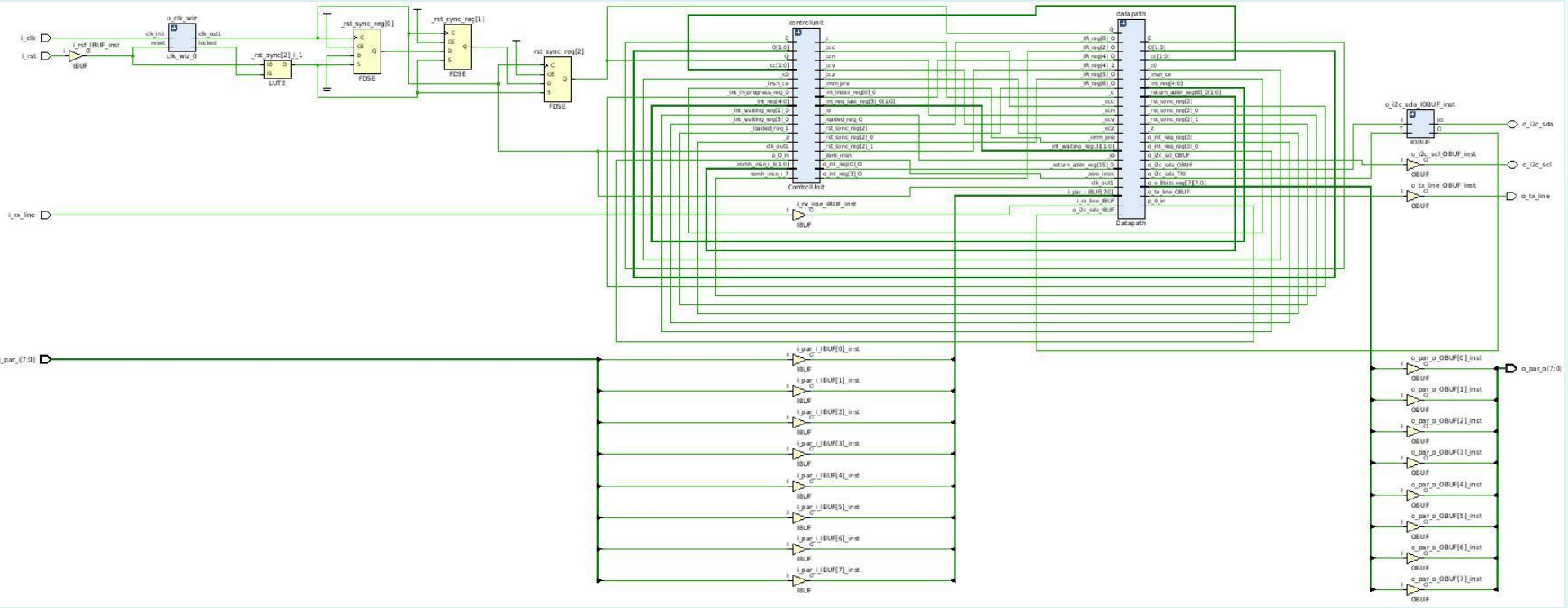
Behavioral Simulation

- Verifies the logical functionality of the design without considering hardware-specific implementation details such as gate delays, routing, or timing constraints.
- It focuses solely on the correctness of the RTL (Register Transfer Level) code, ensuring that the design produces the expected outputs for given inputs.



Synthesized Design

CENTROALGORITMI

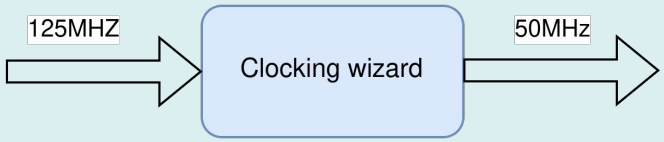


Post-Synthesis Timing Simulation

This synthesis attribute instructs the tool to preserve the register _cnt and prevents it from being optimized away

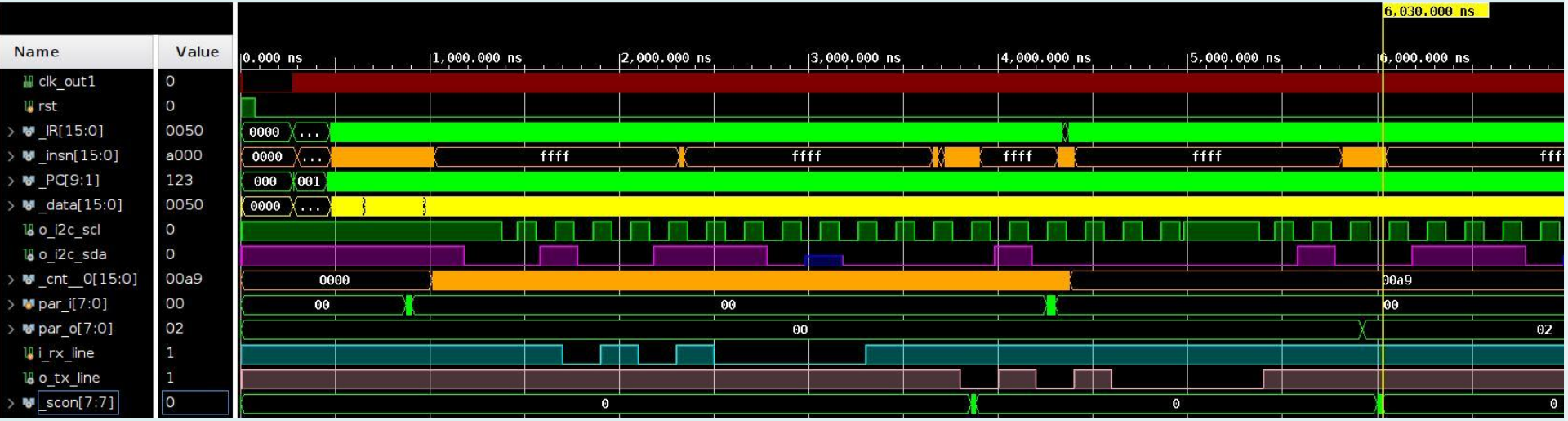
```
(* keep = "true" *) reg [15:0] _cnt;
```

```
wire _clk50;
wire _locked;

// Clock wizard instantiation
clk_wiz_0 u_clk_wiz (
  .clk_in1 (i_clk),
  .clk_out1(_clk50),
  .reset   (i_rst),
  .locked  (_locked)
);

reg [2:0] _rst_sync;

always @(posedge _clk50) begin
  if (i_rst || ~_locked)
    _rst_sync <= 3'b111;
  else
    _rst_sync <= {_rst_sync[1:0], 1'b0};
end

wire _rst_sys = _rst_sync[2];
```



Post-Synthesis Timing Simulation

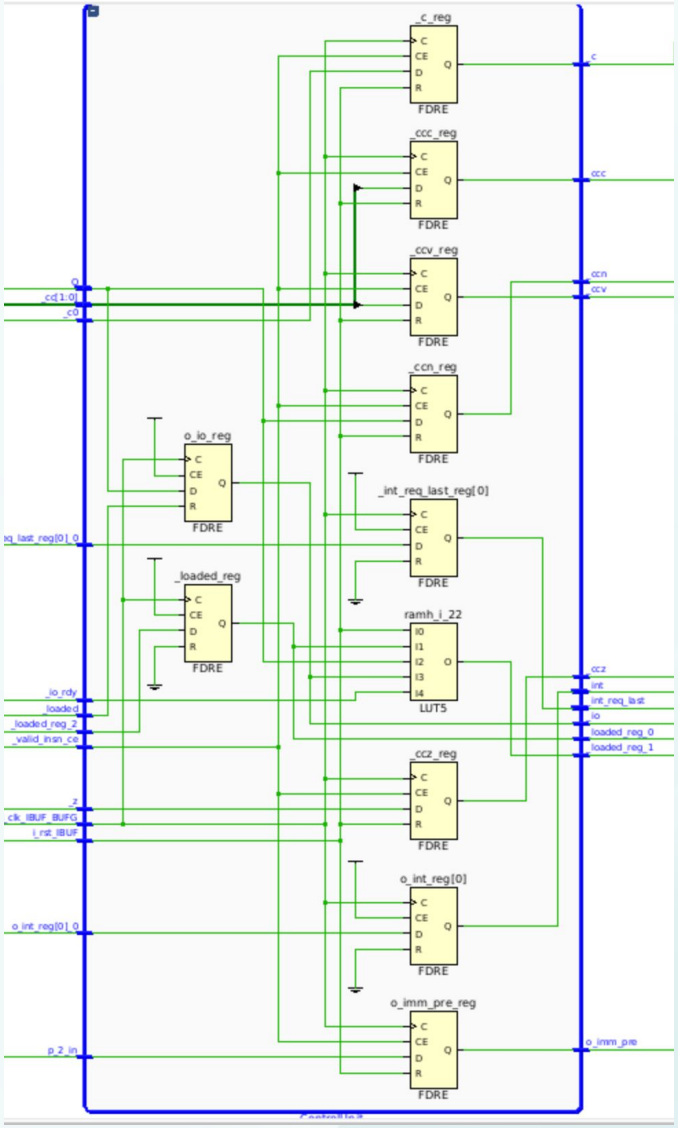
Propagation delay of approximately 18 ns



Implementation Schematic and Primitives

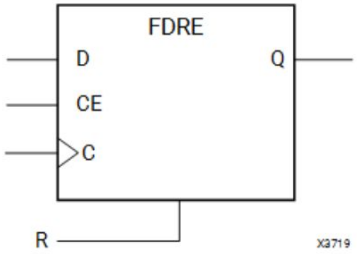
CENTROALGORITMI

Control Unit



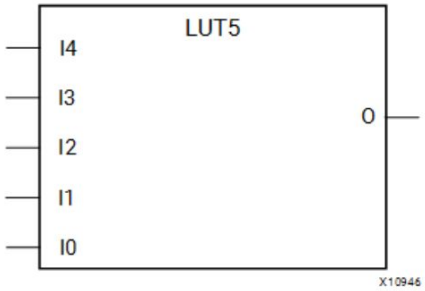
FDRE

Primitive: D Flip-Flop with Clock Enable and Synchronous Reset



LUT5

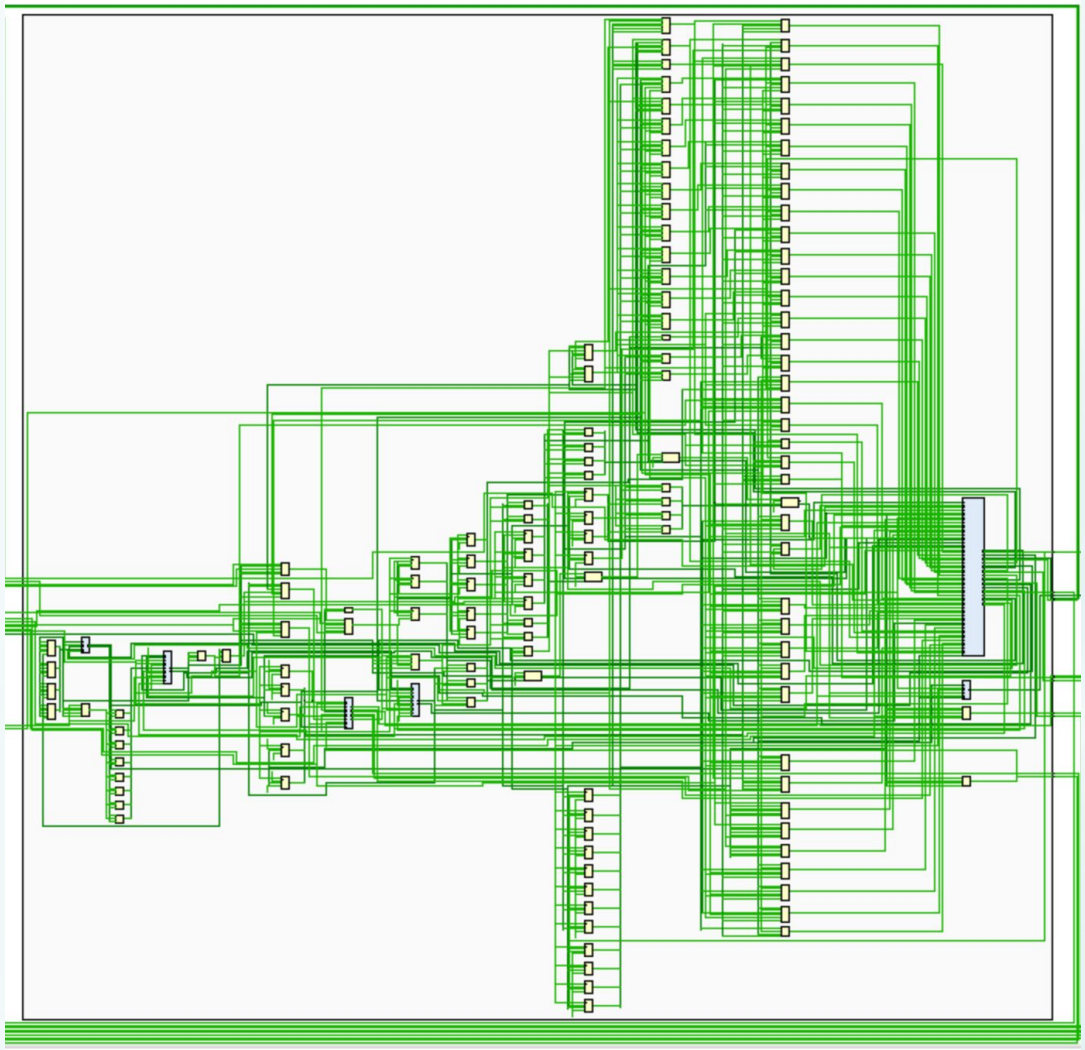
Primitive: 5-Input look-up Table with General Output



Source: Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)

Implementation Schematic

Datapath



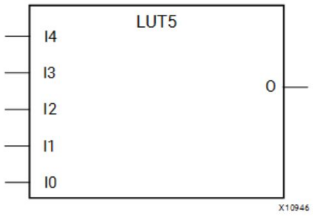
Implementation Primitives

CENTROALGORITMI

Datapath

LUT5

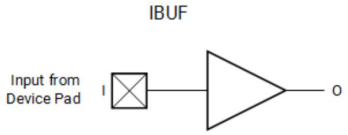
Primitive: 5-Input look-up Table with General Output



LUT2, LUT3, LUT4 and LUT6 are also used

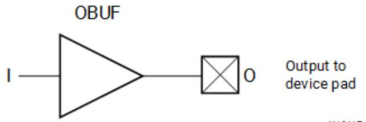
IBUF

Primitive: Input Buffer



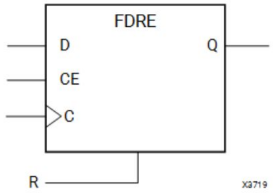
OBUF

Primitive: Output Buffer



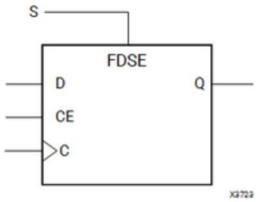
FDRE

Primitive: D Flip-Flop with Clock Enable and Synchronous Reset



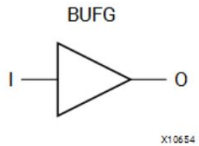
FDSE

Primitive: D Flip-Flop with Clock Enable and Synchronous Set



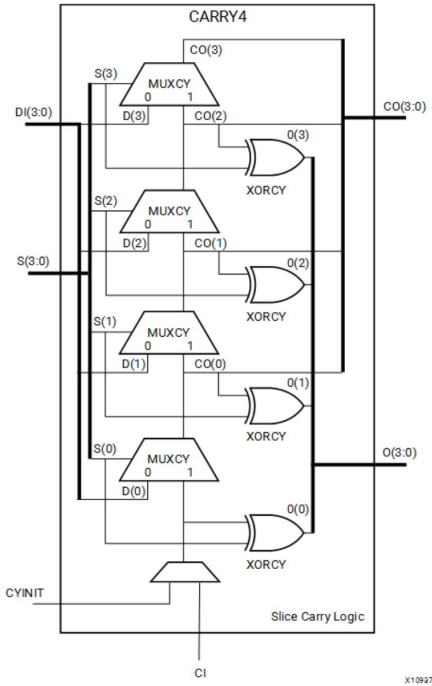
BUFG

Primitive: Global Clock Simple Buffer



CARRY4

Primitive: Fast Carry Logic with Look Ahead



Used in ALU

Distributed Memory IP is used
Block Memory IP is used - Primitive RAMB18E1

Source: Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)

Constraints

Physical constraints

Name	Direction	Board Part Pin	Board Part Interface	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength
i_clk	IN					K17	☑	35	LVCMOS33*	3.300		
i_rst	IN					T16	☑	34	LVCMOS33*	3.300		
i_rx_line	IN					Y17	☑	34	LVCMOS33*	3.300		
o_tx_line	OUT					T17	☑	34	LVCMOS33*	3.300		12
i_par_i[7]	IN					U12	☑	34	LVCMOS33*	3.300		
i_par_i[6]	IN					T12	☑	34	LVCMOS33*	3.300		
i_par_i[5]	IN					Y14	☑	34	LVCMOS33*	3.300		
i_par_i[4]	IN					W14	☑	34	LVCMOS33*	3.300		
i_par_i[3]	IN					T10	☑	34	LVCMOS33*	3.300		
i_par_i[2]	IN					T11	☑	34	LVCMOS33*	3.300		
i_par_i[1]	IN					K19	☑	35	LVCMOS33*	3.300		
i_par_i[0]	IN					Y16	☑	34	LVCMOS33*	3.300		
o_par_o[7]	OUT					V18	☑	34	LVCMOS33*	3.300		12
o_par_o[6]	OUT					V17	☑	34	LVCMOS33*	3.300		12
o_par_o[5]	OUT					U15	☑	34	LVCMOS33*	3.300		12
o_par_o[4]	OUT					U14	☑	34	LVCMOS33*	3.300		12
o_par_o[3]	OUT					R14	☑	34	LVCMOS33*	3.300		12
o_par_o[2]	OUT					P14	☑	34	LVCMOS33*	3.300		12
o_par_o[1]	OUT					M15	☑	35	LVCMOS33*	3.300		12
o_par_o[0]	OUT					M14	☑	35	LVCMOS33*	3.300		12

Utilization Report

report_utilization

CENTROALGORITMI

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	469	0	0	17600	2.66
LUT as Logic	453	0	0	17600	2.57
LUT as Memory	16	0	0	6000	0.27
LUT as Distributed RAM	16	0			
LUT as Shift Register	0	0			
Slice Registers	331	0	0	35200	0.94
Register as Flip Flop	331	0	0	35200	0.94
Register as Latch	0	0	0	35200	0.00
F7 Muxes	2	0	0	8800	0.02
F8 Muxes	1	0	0	4400	0.02

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	2	0	0	60	3.33
RAMB36/FIFO*	0	0	0	60	0.00
RAMB18	4	0	0	120	3.33
RAMB18E1 only	4				

5. IO and GT Specific

Site Type	Used	Fixed	Prohibited	Available	Util%
Bonded IOB	22	22	0	100	22.00
IOB Master Pads	12				
IOB Slave Pads	10				
Bonded IPADs	0	0	0	2	0.00
Bonded IOPADs	0	0	0	130	0.00
PHY_CONTROL	0	0	0	2	0.00
PHASER_REF	0	0	0	2	0.00

Summarizes how the design consumes FPGA resources

Timing Report

TPWS: The sum of the violations for each clock pin in the entire design

Design Timing Summary

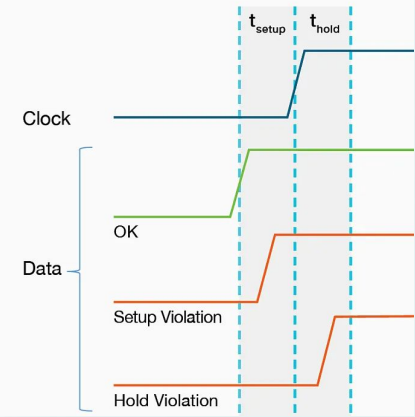
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.412 ns	Worst Hold Slack (WHS):	0.154 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	418	Total Number of Endpoints:	418	Total Number of Endpoints:	130

All user specified timing constraints are met.

the **worst timing path** has **6.412 ns of margin**

If this was negative it would mean: at least one path in the design is **x ns too slow**

TNS: The sum of the setup/recovery violations for each endpoint (where a timing path ends) in the entire design or for a particular clock domain. The worst setup/recovery slack is the worst negative slack (WNS).



Setup time

The time before which the new stable data must be available before the next active clock edge to be safely captured.

Hold requirement

The amount of time the data must remain stable after an active clock edge to avoid capturing an undesired value.

THS: The sum of the hold/removal violations for each endpoint in the entire design or for a particular clock domain. The worst hold/removal slack is the worst hold slack (WHS).

Timing Report

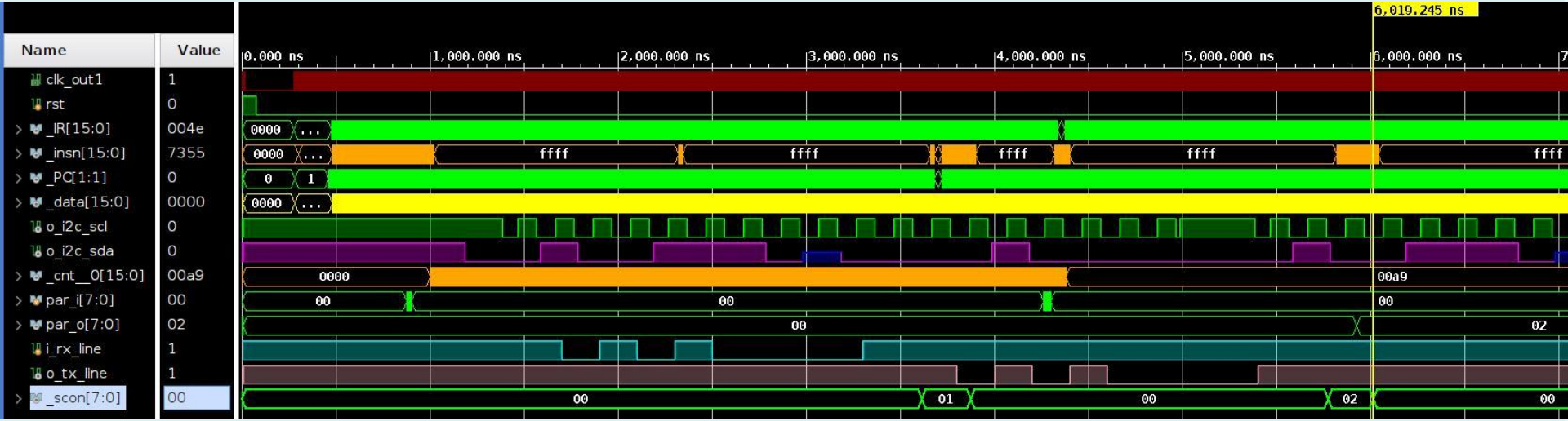

```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (MET) :             6.412ns  (required time - arrival time)
  Source:                 <hidden>
                            (rising edge-triggered cell RAMB18E1 clocked by clk  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            datapath/_pc_reg[12]/CE
                            (rising edge-triggered cell FDRE clocked by clk  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clk rise@20.000ns - clk rise@0.000ns)
  Data Path Delay:        13.048ns  (logic 5.161ns (39.553%)  route 7.887ns (60.447%))
  Logic Levels:           10  (CARRY4=3 LUT2=1 LUT3=1 LUT4=1 LUT6=3 RAMD32=1)
  Clock Path Skew:        -0.097ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.091ns = ( 24.091 - 20.000 )
    Source Clock Delay      (SCD):    5.380ns
    Clock Pessimism Removal (CPR):    0.391ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock clk rise edge)        0.000     0.000 r
    K17                                               0.000     0.000 r  i_clk (IN)
                         net (fo=0)                   0.000     0.000    i_clk
    K17                                                         r  i_clk_IBUF_inst/I
    K17                  IBUF (Prop_ibuf_I_O)         1.492     1.492 r  i_clk_IBUF_inst/O
                         net (fo=1, routed)           2.076     3.568    i_clk_IBUF
    BUFGCTRL_X0Y16                                              r  i_clk_IBUF_BUFG_inst/I
    BUFGCTRL_X0Y16       BUFG (Prop_bufg_I_O)         0.101     3.669 r  i_clk_IBUF_BUFG_inst/O
                         net (fo=129, routed)         1.711     5.380    <hidden>
    RAMB18_X2Y12         RAMB18E1                                    r  <hidden>
  -------------------------------------------------------------------    -------------------
6. checking no_output_delay (0)
--------------------------------
 There are 0 ports with no output delay specified.

 There are 0 ports with no output delay but user has a false path constraint

 There are 0 ports with no output delay but with a timing clock defined on it or propagating through it
```

Clock skew = The difference in the arrival time of a clock signal at two different registers

Verifies what was put in pin constraints

Post-Implementation Timing Simulation

- Compared with post-synthesis timing simulation, the post-implementation simulation shows a slight recovery of approximately 11 ns due to optimization during placement and routing



Hardware Validation

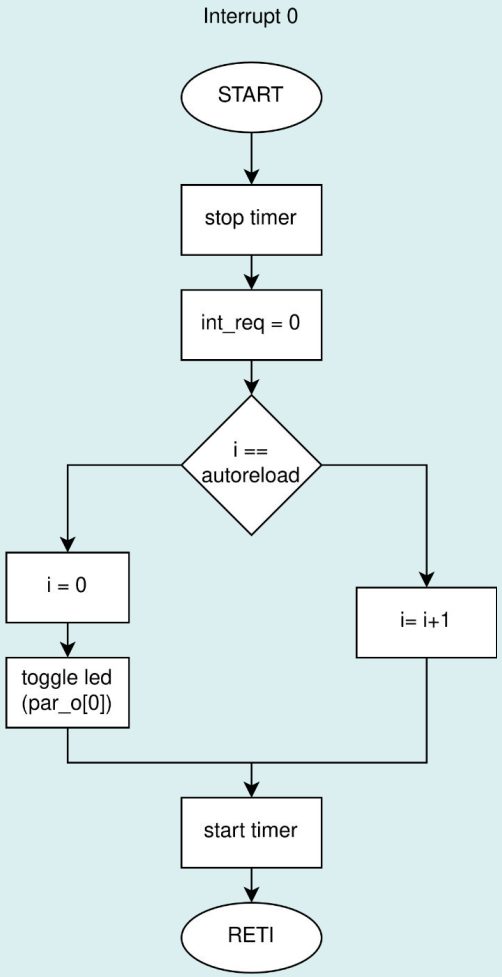
- **External Interrupts:** Button1 (pin Y13) is connected to external interrupt 0 and is used to start the timer. Button2 (pin K19) is connected to external interrupt 1 and is used to stop the timer.

- **LED Indicators:** LED0 (pin M14) is configured as the blinking LED, driven by the timer with a 2-second period. LED1 (pin M15) toggles whenever a UART transmission completes, providing a visual indication of UART activity.

- **UART and I2C Interfaces:**
  - The FPGA UART is connected to an FTDI module. The FTDI RX pin is connected to the FPGA TX, and the FTDI TX is connected to the FPGA RX. Terminal communication is established via a serial terminal (e.g., using the command sudo screen /dev/ttyUSB0 115200) to send characters and observe the echo response.
  - Additionally, the I2C lines (SDA and SCL) are connected to the corresponding pins on the peripheral board for initialization and data transfer.

Hardware Validation

Instructions in ROM being executed

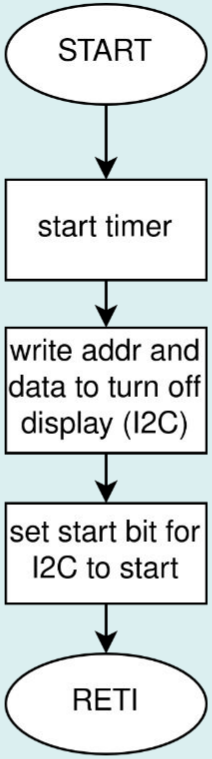



```
//ISR TIMER
SW r6, r12, #0
SW r10, r12, #2
CMP r15, r14
BEQ #124
ADD r14, r13
SW r7, r12, #0
RETI
FFFF
```

Hardware Validation

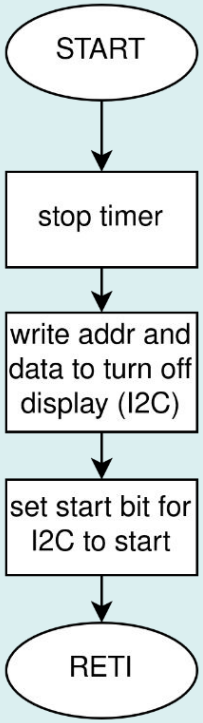
Instructions in ROM being executed

Interrupt 1




```
// ISR EXTERN 0
SW r7, r12, #0
IMM #084
ADDI r2, r0, #E
SW r2, r4, #2
ADDI r2, r0, #3
SW r2, r4, #4
RETI
FFFF
```

Interrupt 2

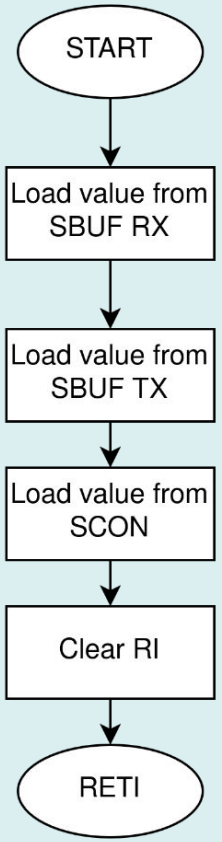



```
// ISR EXTERN 1
SW r6, r12, #0
IMM #004
ADDI r2, r0, #E
SW r2, r4, #2
ADDI r2, r0, #3
SW r2, r4, #4
RETI
FFFF
```

Hardware Validation

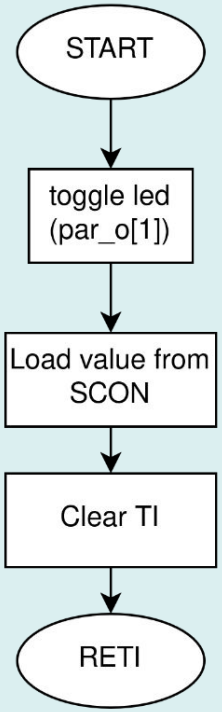
Instructions in ROM being executed

Interrupt 3




```
//ISR UART TX
ADDI r8, r10, #2
XOR r9, r8
SB r9, r11, #2
LB r3, r5, #5
AND r3, r13
SB r3, r5, #5
RETI
FFFF
```

Interrupt 4



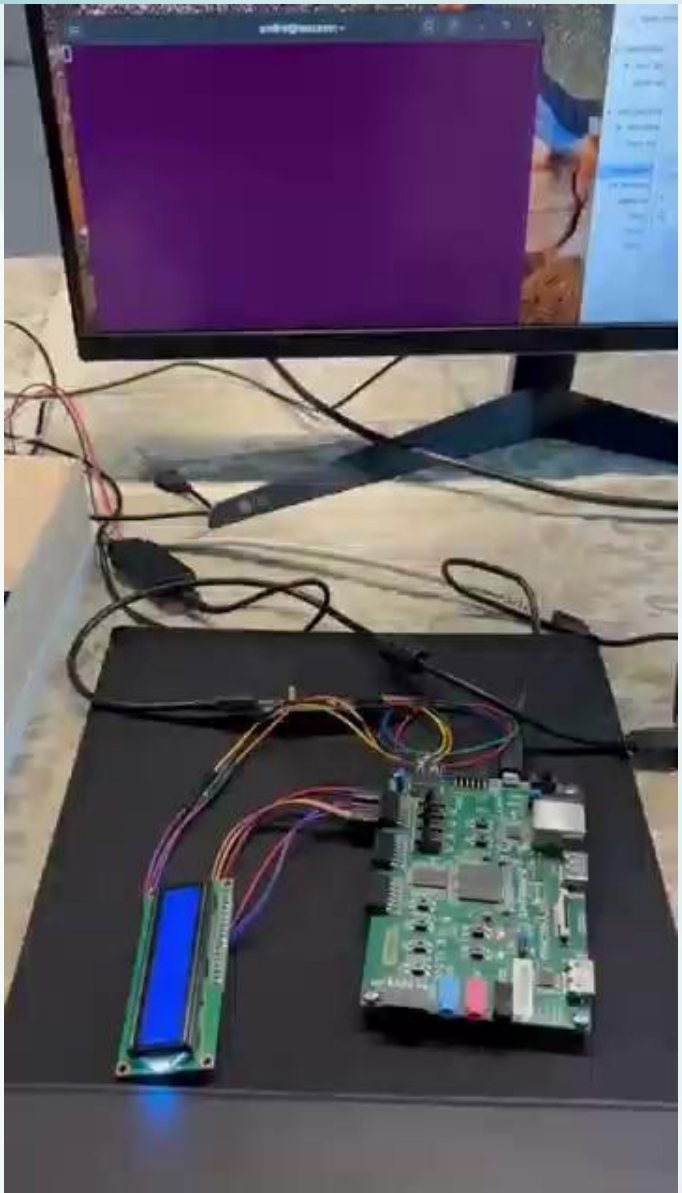

```
//ISR UART RX
LB r3, r5, #0
SB r3, r5, #1
LB r8, r5, #5
ADDI r3, r10, #2
AND r8, r3
SB r8, r5, #5
RETI
FFFF
```

Hardware Validation

Instructions in ROM being executed

```
//MAIN PROGRAM
IMM #076
ADDI r15, r0, #9
ADDI r14, r0, #0
ADDI r13, r0, #1
IMM #100
ADDI r12, r0, #0
IMM #110
ADDI r11, r0, #0
ADDI r10, r0, #0
ADDI r7, r0, #7
ADDI r6, r0, #3
IMM #120
ADDI r5, r0, #0
IMM #130
ADDI r4, r0, #0
IMM #004
ADDI r2, r0, #E
SW r2, r4, #2
ADDI r2, r0, #3
SW r2, r4, #4
FFFF
```

Practical Demonstration



Thank you!

Any questions?