FPGA for software developers

Miodrag Milanović

code::dive

Introduction

- Engineer
- Hardware and emulation enthusiast

- Software developer since year 2000
- Software architect at “Levi Nine” Serbia
- Worked in C,C++,C# and Java, lately in C++ only

- Working on MAME emulation project since 2006
- Was leading MAME project since 2012 till 2016

Why we are here?

- Understanding hardware
- To be better developer you need to know and understand how computer works.
- Best way to understand how something work is to make it.
- “Well I am just software developer and I don’t understand electronics…”
- We will write hardware as source code and then “compile” hardware.
- “Impossible” and “too hard” are just lame excuses ☺
- Overall, we are here to have some fun.

Let’s first remind ourselves…

and learn something new

Boolean logic

1 0 X Z

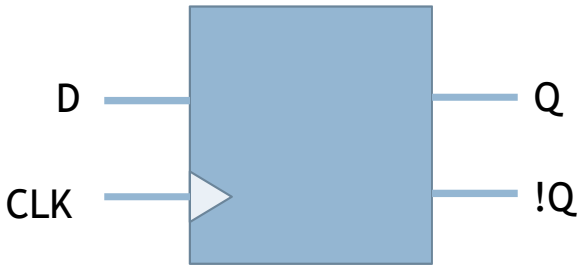
Boolean operations

- AND
- OR
- XOR
- NOT
- NAND
- NOR
- XNOR

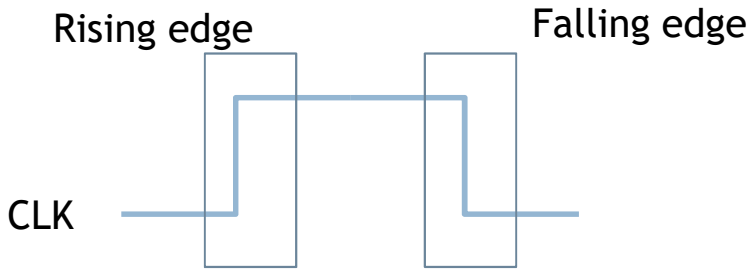
INPUT		OUTPUT					
A	B	A AND B	A NAND B	A OR B	A NOR B	A NOR B	A XNOR B
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

code::dive

D Flip-Flop



CLK	D	R	S	Q	!Q
rising edge	0	0	0	0	1
rising edge	1	0	0	1	0
falling edge	x	0	0	Q	!Q
x	x	1	0	0	1
x	x	0	1	1	0
x	x	1	1	1	1

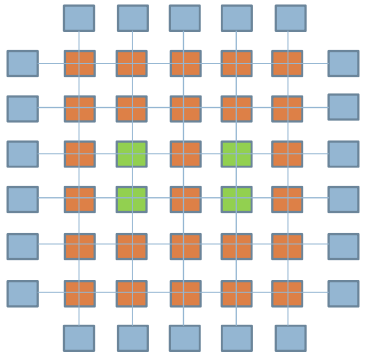


code::dive

What is FPGA ?

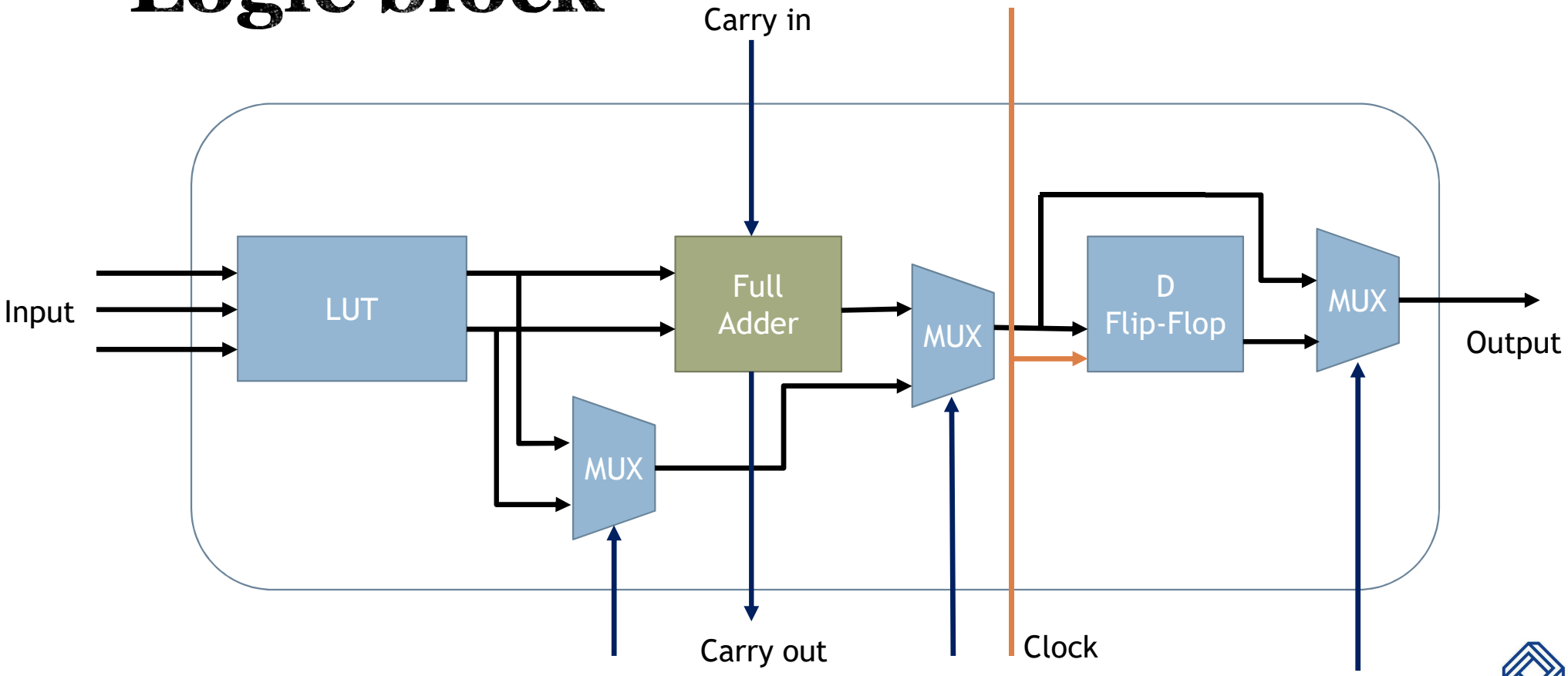
- Field-programmable gate array
- Logic block – LUT + D Flip-Flop + full adder
- I/O block – one per each pin
- Interconnections, clock tree
- Hard block – Block RAM, multipliers, DSP, CPU,…

- Vendors: Xilinx, Intel/Altera, Lattice, Microsemi …



code::dive

Logic block



code::dive

How do we program FPGA ?

- HDL - Hardware description language
- VHDL and Verilog

- Analysis: parsing and validation of HDL
- Synthesis: HDL -> netlist
- Place-and-route: netlist -> specific FPGA technology
- Assembler: specific FPGA technology -> bitstream
- Programming: deploying bitstream on device (serial flash memory or directly)
- Timing analysis
- Simulation

Open source tools

- Yosys - Verilog synthesis tool by Clifford Wolf
- Arachne PnR - Place and route tool by Cotton Seed
- Project IceStorm – Assembler, time analysis and FPGA programming tool
  by Clifford Wolf and Mathias Lasser

- Lattice iCE40 FPGA only

- Icarus Verilog by Stephen Williams
- Verilator by Wilson Snyder with Duane Galbi and Paul Wasson

And more...

- APIO – micro-ecosystem for open FPGAs by Jesús Arroyo Torrens and Juan González (Obijuan)
- FuseSoC - package manager and a set of build tools for FPGA/ASIC development by Olof Kindgren
- Icestudio - graphic editor for open FPGAs by Jesús Arroyo Torrens

Let’s get started

What does computer consists of ?

- CPU
  - Registers
  - ALU - Arithmetic logic unit
  - Control unit
- Memory
- I/O devices

- How do we name our project ?

code::dive

Grom-8

- Simple
- Not lightning fast
- Thunder is always slower
- Polish word is GROM
- And we will put 8 in name, since it is 8 bit

Architecture

- 8 bit CPU
- 4 general purpose registers
- 12 bit address bus and PC (program counter)
- Code and data segment registers (CS,DS) (4 bit)
- Stack pointer (SP) (12 bit)
- 8-bit ALU with C-carry, Z-zero and S-sign flags

Instruction set 1/2

IR			2nd	Instruction	Info
0000	dst	src		MOV dst, src	
0001	00	reg		ADD reg	r0 = r0 + reg
0001	01	reg		SUB reg	r0 = r0 - reg
0001	10	reg		ADC reg	r0 = r0 + reg + C
0001	11	reg		SBC reg	r0 = r0 - reg - C
0010	00	reg		AND reg	r0 = r0 and reg
0010	01	reg		OR reg	r0 = r0 or reg
0010	10	reg		NOT reg	r0 = not reg
0010	11	reg		XOR reg	r0 = r0 xor reg
0011	00	reg		INC reg	reg = reg + 1
0011	01	reg		DEC reg	reg = reg - 1
0011	10	reg		CMP reg	flags of r0 - reg
0011	11	reg		TST reg	flags of r0 and reg
0100	00	00		SHL	
0100	00	01		SHR	
0100	00	10		SAL	
0100	00	11		SAR	
0100	01	00		ROL	
0100	01	01		ROR	
0100	01	10		RCL	rotate with carry
0100	01	11		RCR	rotate with carry
0100	10	reg		PUSH reg	
0100	11	reg		POP reg	
0101	dst	src		LOAD dst, [src]	
0110	dst	src		STORE [dst], src	
0111	00	reg		MOV CS, reg	
0111	01	reg		MOV DS, reg	
0111	10	00		PUSH CS	
0111	10	01		PUSH DS	
0111	10	10		???	
0111	10	11		???	
0111	11	00		???	
0111	11	01		???	
0111	11	10		RET	
0111	11	11		HLT	

code::dive

Instruction set 2/2

1000	00	00	val	JMP val	(unconditionally jump)
1000	00	01	val	JC val	(carry=1 jump)
1000	00	10	val	JNC val	(carry=0 jump)
1000	00	11	val	JM val	(sign=1 jump)
1000	01	00	val	JP val	(sign=0 jump)
1000	01	01	val	JZ val	(zero=1 jump)
1000	01	10	val	JNZ val	(zero=0 jump)
1000	01	11	val	???	
1000	10	00	val	JR val	(unconditionally jump)
1000	10	01	val	JRC val	(carry=1 jump)
1000	10	10	val	JRNC val	(carry=0 jump)
1000	10	11	val	JRM val	(sign=1 jump)
1000	11	00	val	JRP val	(sign=0 jump)
1000	11	01	val	JRZ val	(zero=1 jump)
1000	11	10	val	JRNZ val	(zero=0 jump)
1000	11	11	val	???	

1001	high		low	JUMP addr	
1010	high		low	CALL addr	
1011	high		low	MOV SP,addr	
1100	xx	reg	val	IN reg,[val]	
1101	xx	reg	val	OUT [val],reg	
1110	xx	00	val	MOV CS,val	
1110	xx	01	val	MOV DS,val	
1110	xx	10	val	???	
1110	xx	11	val	???	
1111	00	reg	val	MOV reg, val	
1111	01	reg	val	LOAD reg, [val]	
1111	10	reg	val	STORE [val], reg	
1111	11	xx		???	

code::dive

Verilog

Memory

Modules

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095]

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

Inputs and outputs

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095]

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

Registers

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095];

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

Initial block

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095];

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

Always block

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095];

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

code::dive

Condition check

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095];

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

Non-blocking assignment

```
module ram_memory(
  input clk,
  input [11:0] addr,
  input [7:0] data_in,
  input we,
  output reg [7:0] data_out
);

  reg [7:0] store[0:4095];

  initial
  begin
    $readmemh("boot.mem", store);
  end

  always @(posedge clk)
    if (we)
      store[addr] <= data_in;
    else
      data_out <= store[addr];
endmodule
```

Parallel execution

```
addr  <= PC;
PC    <= PC + 1;
```

SAME

```
PC    <= PC + 1;
addr  <= PC;
```

```
addr  = PC;
PC    = PC + 1;
```

NOT SAME

```
PC    = PC + 1;
addr  = PC;
```

code::dive

I/O devices

Begin – end block

```
module hex_to_7seg
  (
   input [3:0] i_Value,
   // ...
   output      o_Segment_G
   );

  reg [6:0]    out = 7'b0000000;

  always @(posedge i_Clk)
    begin
      case (i_Value)
        4'b0000 : out <= 7'b0000001;
        4'b0001 : out <= 7'b1001111;
        // ...
        4'b1110 : out <= 7'b0110000;
        4'b1111 : out <= 7'b0111000;
      endcase
    end

  assign o_Segment_A = out[6];
  // ...
endmodule
```

code::dive

Case

```
module hex_to_7seg
  (
   input [3:0] i_Value,
   // ...
   output      o_Segment_G
   );

  reg [6:0]    out = 7'b0000000;

  always @(posedge i_Clk)
    begin
      case (i_Value)
        4'b0000 : out <= 7'b0000001;
        4'b0001 : out <= 7'b1001111;
        // ...
        4'b1110 : out <= 7'b0110000;
        4'b1111 : out <= 7'b0111000;
      endcase
    end

  assign o_Segment_A = out[6];
  // ...
endmodule
```

code::dive

Continuous assignment

```
module hex_to_7seg
  (
   input [3:0] i_Value,
   // ...
   output      o_Segment_G
   );

  reg [6:0]    out = 7'b0000000;

  always @(posedge i_Clk)
    begin
      case (i_Value)
        4'b0000 : out <= 7'b0000001;
        4'b0001 : out <= 7'b1001111;
        // ...
        4'b1110 : out <= 7'b0110000;
        4'b1111 : out <= 7'b0111000;
      endcase
    end

  assign o_Segment_A = out[6];
  // ...
endmodule
```

code::dive

ALU

Local parameters / Constants

```
module alu(
    input clk,
    input [7:0] A,
    input [7:0] B,
    input [3:0] operation,
    output reg [7:0] result,
    output reg CF,
    output reg ZF,
    output reg SF
);

    localparam ALU_OP_ADD  = 4'b0000;
    localparam ALU_OP_SUB  = 4'b0001;
    //...
    localparam ALU_OP_RCR  = 4'b1111;

    reg [8:0] tmp;

    always @(posedge clk)
    begin
        case (operation)
            ALU_OP_ADD :
                tmp = A + B;
```

code::dive

Blocking assignment

```
reg [8:0] tmp;

case (operation)
    ALU_OP_ADD :
        tmp = A + B;
    ALU_OP_SUB :
        tmp = A - B;
    ALU_OP_ADC :
        tmp = A + B + { 7'b0000000, CF };
    ALU_OP_SBC :
        tmp = A - B - { 7'b0000000, CF };
    // ...
endcase

CF <= tmp[8];
ZF <= tmp[7:0] == 0;
SF <= tmp[7];

result <= tmp[7:0];
```

code::dive

ALU - Arithmetic operations

```
reg [8:0] tmp;

case (operation)
    ALU_OP_ADD :
        tmp = A + B;
    ALU_OP_SUB :
        tmp = A - B;
    ALU_OP_ADC :
        tmp = A + B + { 7'b0000000, CF };
    ALU_OP_SBC :
        tmp = A - B - { 7'b0000000, CF };
    // ...
endcase

CF <= tmp[8];
ZF <= tmp[7:0] == 0;
SF <= tmp[7];

result <= tmp[7:0];
```

code::dive

ALU – Bitwise logic operations

```
reg [8:0] tmp;

case (operation)
    // ...
    ALU_OP_AND :
        tmp = {1'b0, A & B };
    ALU_OP_OR :
        tmp = {1'b0, A | B };
    ALU_OP_NOT :
        tmp = {1'b0, ~B };
    ALU_OP_XOR :
        tmp = {1'b0, A ^ B};
    // ...
endcase

CF <= tmp[8];
ZF <= tmp[7:0] == 0;
SF <= tmp[7];

result <= tmp[7:0];
```

code::dive

ALU - Shift operations

```
reg [8:0] tmp;

case (operation)
    // ...
    ALU_OP_SHL :
        tmp = { A[7], A[6:0], 1'b0};
    ALU_OP_SHR :
        tmp = { A[0], 1'b0, A[7:1]};
    ALU_OP_SAL :
        tmp = { A[7], A[6:0], 1'b0};       // Same as SHL
    ALU_OP_SAR :
        tmp = { A[0], A[7], A[7:1]};
    // ...
endcase

CF <= tmp[8];
ZF <= tmp[7:0] == 0;
SF <= tmp[7];

result <= tmp[7:0];
```

code::dive

ALU - Rotate operations

```
reg [8:0] tmp;

case (operation)
    // ...
    ALU_OP_ROL :
        tmp = { A[7], A[6:0], A[7]};
    ALU_OP_ROR :
        tmp = { A[0], A[0], A[7:1]};
    ALU_OP_RCL :
        tmp = { A[7], A[6:0], CF};
    ALU_OP_RCR :
        tmp = { A[0], CF, A[7:1]};
    // ...
endcase

CF <= tmp[8];
ZF <= tmp[7:0] == 0;
SF <= tmp[7];

result <= tmp[7:0];
```

code::dive

ALU – Flags and result

```
reg [8:0] tmp;

case (operation)
    // ...
    ALU_OP_ROL :
        tmp = { A[7], A[6:0], A[7]};
    ALU_OP_ROR :
        tmp = { A[0], A[0], A[7:1]};
    ALU_OP_RCL :
        tmp = { A[7], A[6:0], CF};
    ALU_OP_RCR :
        tmp = { A[0], CF, A[7:1]};
    // ...
endcase

CF <= tmp[8];
ZF <= tmp[7:0] == 0;
SF <= tmp[7];

result <= tmp[7:0];
```

code::dive

Other operations

Multiply

```
{R[1],R[0]} <= R[0] * R[1];
```

Divide

```
R[0] <= R[0] / R[1];
```

Modulus

```
R[0] <= R[0] % R[1];
```

Shift left by 1 (multiply by 2)

```
R[0] <= R[0] << 1;
```

Shift right by 1 (divide by 2)

```
R[0] <= R[0] >> 1;
```

CPU

Wires

```
module grom_cpu(
    input clk,input reset,
    output reg [11:0] addr,
    input [7:0] data_in,output reg [7:0] data_out,
    output reg we,output reg ioreq, output reg hlt
);
    reg[11:0] PC;          // Program counter
    reg[7:0] IR;           // Instruction register
    reg[11:0] SP;          // Stack pointer regiser
    reg[7:0] R[0:3];       // General purpose registers
    // ...
    reg [7:0]  alu_a;
    reg [7:0]  alu_b;
    reg [3:0]  alu_op;
    wire [7:0] alu_res;
    wire alu_CF;
    wire alu_ZF;
    wire alu_SF;
    alu alu(.clk(clk),.A(alu_a),.B(alu_b),.operation(alu_op),
            .result(alu_res),.CF(alu_CF),.ZF(alu_ZF),.SF(alu_SF));
    // ...
```

code::dive

Using modules

```
module grom_cpu(
    input clk,input reset,
    output reg [11:0] addr,
    input [7:0] data_in,output reg [7:0] data_out,
    output reg we,output reg ioreq, output reg hlt
);
    reg[11:0] PC;          // Program counter
    reg[7:0] IR;           // Instruction register
    reg[11:0] SP;          // Stack pointer regiser
    reg[7:0] R[0:3];       // General purpose registers
    // ...
    reg [7:0]  alu_a;
    reg [7:0]  alu_b;
    reg [3:0]  alu_op;
    wire [7:0] alu_res;
    wire alu_CF;
    wire alu_ZF;
    wire alu_SF;
    alu alu(.clk(clk),.A(alu_a),.B(alu_b),.operation(alu_op),
            .result(alu_res),.CF(alu_CF),.ZF(alu_ZF),.SF(alu_SF));
    // ...
```

code::dive

Main state machine

```
reg [4:0] state = STATE_RESET;

always @(posedge clk)
begin
    if (reset)
    begin
        state <= STATE_RESET;
        hlt   <= 0;
    end
    else
    begin
        case (state)
            STATE_RESET :
                begin
                    PC    <= 12'h000;
                    R[0]  <= 8'h00;
                    // ...
                    SP    <= 12'hfff;
                    state <= STATE_FETCH_PREP;
                end
```

Instruction fetch

```
STATE_FETCH_PREP :
    begin
        addr  <= PC;
        we    <= 0;
        ioreq <= 0;

        state <= STATE_FETCH_WAIT;
    end

STATE_FETCH_WAIT :
    begin
        // Sync with memory due to CLK
        state <= (hlt) ? STATE_FETCH_PREP : STATE_FETCH;
    end

STATE_FETCH :
    begin
        IR    <= data_in;
        PC    <= PC + 1;

        state <= STATE_EXECUTE;
    end
```

code::dive

Instruction decoding

```
STATE_EXECUTE :
    begin
        if (IR[7])
        begin
            addr  <= PC;
            state <= STATE_FETCH_VALUE_PREP;
            PC    <= PC + 1;
        end
        else
        begin
            case(IR[6:4])
                3'b000 :
                    begin
                        `ifdef DISASSEMBLY
                        $display("MOV R%d,R%d",IR[3:2],IR[1:0]);
                        `endif
                        R[IR[3:2]] <= R[IR[1:0]];
                        state <= STATE_FETCH_PREP;
                    end
```

MOV instruction

```
STATE_EXECUTE :
    begin
        if (IR[7])
        begin
            addr  <= PC;
            state <= STATE_FETCH_VALUE_PREP;
            PC    <= PC + 1;
        end
        else
        begin
            case(IR[6:4])
                3'b000 :
                    begin
                        `ifdef DISASSEMBLY
                        $display("MOV R%d,R%d",IR[3:2],IR[1:0]);
                        `endif
                        R[IR[3:2]] <= R[IR[1:0]];
                        state <= STATE_FETCH_PREP;
                    end
```

Conditional compiling

```
STATE_EXECUTE :
    begin
        if (IR[7])
        begin
            addr  <= PC;
            state <= STATE_FETCH_VALUE_PREP;
            PC    <= PC + 1;
        end
        else
        begin
            case(IR[6:4])
                3'b000 :
                    begin
                        `ifdef DISASSEMBLY
                        $display("MOV R%d,R%d",IR[3:2],IR[1:0]);
                        `endif
                        R[IR[3:2]] <= R[IR[1:0]];
                        state <= STATE_FETCH_PREP;
                    end
```

code::dive

Using ALU module 1/2

```
3'b001 :
    begin
        alu_a   <= R[0];      // first input R0
        alu_b   <= R[IR[1:0]];
        RESULT_REG <= 0;      // result in R0
        alu_op  <= { 2'b00, IR[3:2] };

        state   <= STATE_ALU_RESULT_WAIT;
    end
3'b010 :
    begin
        alu_a   <= R[0];      // first input R0
        alu_b   <= R[IR[1:0]];
        RESULT_REG <= 0;      // result in R0
        alu_op  <= { 2'b01, IR[3:2] };
        state   <= STATE_ALU_RESULT_WAIT;
    end
```

code::dive

Using ALU module 2/2

```
STATE_ALU_RESULT_WAIT :
    begin
        state <= STATE_ALU_RESULT;
    end
STATE_ALU_RESULT :
    begin
        R[RESULT_REG] <= alu_res;
        state <= STATE_FETCH_PREP;
    end
```

code::dive

Memory operations 1/2

```
3'b101 :
    begin
        `ifdef DISASSEMBLY
        $display("LOAD R%d,[R%d]", IR[3:2], IR[1:0]);
        `endif
        addr  <= { DS, R[IR[1:0]] };
        we    <= 0;
        ioreq <= 0;
        RESULT_REG <= IR[3:2];

        state <= STATE_LOAD_VALUE_WAIT;
    end
3'b110 :
    begin
        `ifdef DISASSEMBLY
        $display("STORE [R%d],R%d", IR[3:2], IR[1:0]);
        `endif
        addr     <= { DS, R[IR[3:2]] };
        we       <= 1;
        ioreq    <= 0;
        data_out <= R[IR[1:0]];
        state    <= STATE_FETCH_PREP;
    end
```

code::dive

Memory operations 2/2

```
STATE_LOAD_VALUE_WAIT :
    begin
        // Sync with memory due to CLK
        state <= STATE_LOAD_VALUE;
    end
STATE_LOAD_VALUE :
    begin
        R[RESULT_REG] <= data_in;
        we    <= 0;
        state <= STATE_FETCH_PREP;
    end
```

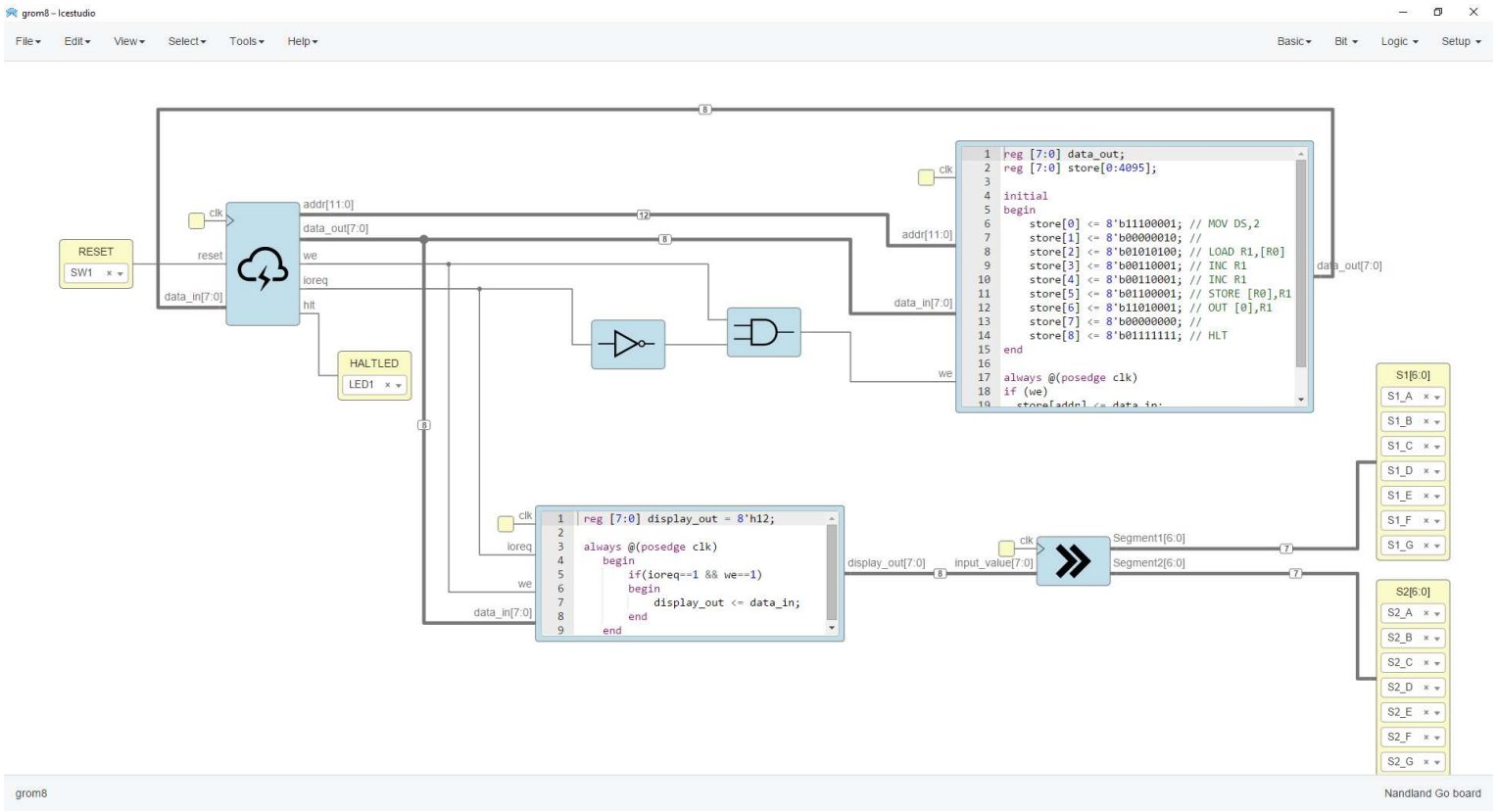
Computer

Main implementation

```
module grom_computer(input clk,input reset,output hlt,output reg[7:0] display_out );
 wire [11:0] addr;
 wire [7:0] memory_out;
 wire [7:0] memory_in;
 wire mem_enable;
 wire we;
 wire ioreq;

 grom_cpu cpu(.clk(clk),.reset(reset),.addr(addr),.data_in(memory_out),
              .data_out(memory_in),.we(we),.ioreq(ioreq),.hlt(hlt));

 assign mem_enable = we & ~ioreq;

 ram_memory memory(.clk(clk),.addr(addr),.data_in(memory_in),
                   .we(mem_enable),.data_out(memory_out));

 always @(posedge clk)
    begin
        if(ioreq==1 && we==1)
        begin
            display_out <= memory_in;
        end
    end
endmodule
```

code::dive

Icestudio



code::dive

Simulation

Test program

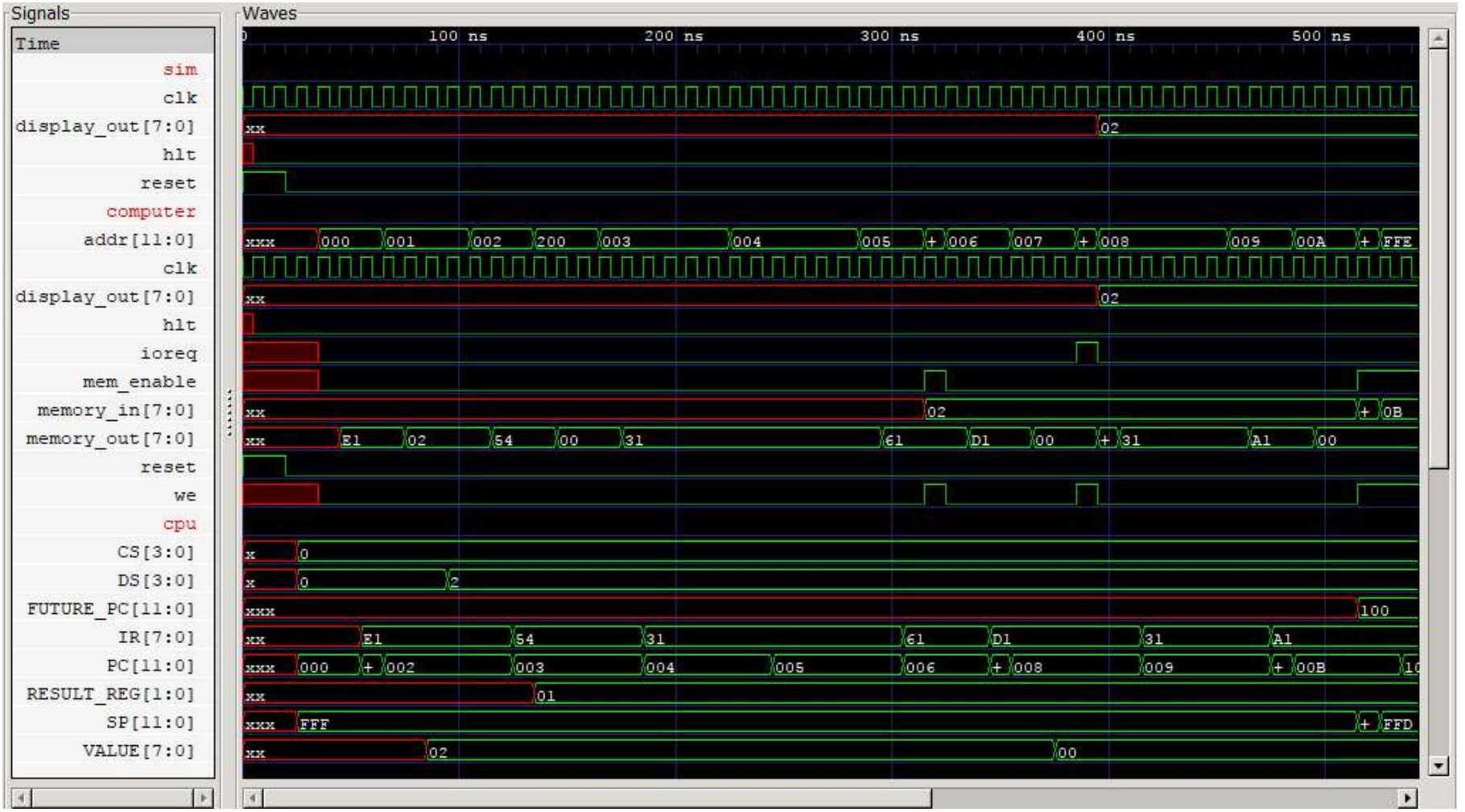
```
store[0] <= 8'b11100001; // MOV DS,2
store[1] <= 8'b00000010; //
store[2] <= 8'b01010100; // LOAD R1,[R0]
store[3] <= 8'b00110001; // INC R1
store[4] <= 8'b00110001; // INC R1
store[5] <= 8'b01100001; // STORE [R0],R1
store[6] <= 8'b11010001; // OUT [0],R1
store[7] <= 8'b00000000; //
store[8] <= 8'b00110001; // INC R1
store[9] <= 8'b10100001; // CALL 0x100
store[10] <= 8'b00000000; //
store[11] <= 8'b01111111; // HLT


store[256] <= 8'b11010001; // OUT [0],R1
store[257] <= 8'b00000000; //
store[258] <= 8'b01111110; // RET
```

code::dive

Test bench

```
`timescale 1ns/1ps
module grom_computer_tb();
    reg clk = 0;
    reg reset;
    wire [7:0] display_out;
    wire hlt;

    grom_computer computer(.clk(clk),.reset(reset),.hlt(hlt),.display_out(display_out));

    always
        #(5) clk <= !clk;

    initial
    begin
        $dumpfile("grom_computer_tb.vcd");
        $dumpvars(0,grom_computer_tb);
        reset = 1;
        #20
        reset = 0;
        #900
        $finish;
    end
endmodule
```

code::dive


Signals
Waves
Time
sim
clk
display_out[7:0]
hlt
reset
computer
addr[11:0]
clk
display_out[7:0]
hlt
ioreq
mem_enable
memory_in[7:0]
memory_out[7:0]
reset
we
cpu
CS[3:0]
DS[3:0]
FUTURE_PC[11:0]
IR[7:0]
PC[11:0]
RESULT_REG[1:0]
SP[11:0]
VALUE[7:0]
100 ns
200 ns
300 ns
400 ns
500 ns

code::dive

Text output

```
iverilog -D DISASSEMBLY -o grom.out grom_computer_tb.v ram_memory.v alu.v grom_cpu.v grom_computer.v
vvp grom.out
VCD info: dumpfile grom_computer_tb.vcd opened for output.
    PC 001 R0 00 R1 00 R2 00 R3 00 CS 0 DS 0 SP fff ALU [x x x]
MOV DS,0x02
    PC 003 R0 00 R1 00 R2 00 R3 00 CS 0 DS 2 SP fff ALU [x x x]
LOAD R1,[R0]
    PC 004 R0 00 R1 00 R2 00 R3 00 CS 0 DS 2 SP fff ALU [x x x]
INC R1
    PC 005 R0 00 R1 01 R2 00 R3 00 CS 0 DS 2 SP fff ALU [0 0 0]
INC R1
    PC 006 R0 00 R1 02 R2 00 R3 00 CS 0 DS 2 SP fff ALU [0 0 0]
STORE [R0],R1
    PC 007 R0 00 R1 02 R2 00 R3 00 CS 0 DS 2 SP fff ALU [0 0 0]
OUT [0x00],R1
Display output : 02
    PC 009 R0 00 R1 02 R2 00 R3 00 CS 0 DS 2 SP fff ALU [0 0 0]
INC R1
    PC 00a R0 00 R1 03 R2 00 R3 00 CS 0 DS 2 SP fff ALU [0 0 0]
CALL 100
Jumping to 100
    PC 101 R0 00 R1 03 R2 00 R3 00 CS 0 DS 2 SP ffd ALU [0 0 0]
OUT [0x00],R1
Display output : 03
    PC 103 R0 00 R1 03 R2 00 R3 00 CS 0 DS 2 SP ffd ALU [0 0 0]
RET
Jumping to 00b
    PC 00c R0 00 R1 03 R2 00 R3 00 CS 0 DS 2 SP fff ALU [0 0 0]
HALT
```

code::dive

Unit testing

C++ for testing 1/2

```cpp
#include <iostream>
#include <memory>
#include "Valu.h"
#include "verilated.h"

#define CATCH_CONFIG_RUNNER
#include "catch.hpp"

int main( int argc, char* argv[] )
{
  Verilated::commandArgs(argc, argv);
  int result = Catch::Session().run( argc, argv );
  return ( result < 0xff ? result : 0xff );
}
```

C++ for testing 2/2

```
TEST_CASE("Test ALU_OP_ADD", "ALU")
{
    std::unique_ptr<Valu> top = std::make_unique<Valu>();
    top->clk = 0;
    for(int A=0;A<0x100;A++)
    {
        for(int B=0;B<0x100;B++)
        {
            top->A = A;
            top->B = B;
            top->operation = top->alu__DOT__ALU_OP_ADD;
            top->clk ^= 1; top->eval();
            top->clk ^= 1; top->eval();
            uint8_t res = A + B;
            REQUIRE(res==top->result);
            REQUIRE(((res==0) ? 1:0)==top->ZF);
            REQUIRE((((A + B)>0xff) ? 1 : 0 )==top->CF);
            REQUIRE(((res & 0x80) ? 1 : 0) == top->SF);
        }
    }
}
```

code::dive

Real hardware

Connecting with hardware

```
module grom_top
  (input  i_Clk,         // Main Clock
   input  i_Switch_1,    // SW1 button
   output o_Segment1_A,
   // ...
   output o_Segment2_G
   );

 wire [7:0] display_out;
 wire hlt;

 grom_computer computer(.clk(i_Clk),.reset(i_Switch_1),.hlt(hlt),.display_out(display_out));

 hex_to_7seg upper_digit
  (.i_Clk(i_Clk),.i_Value(display_out[7:4]),
   .o_Segment_A(o_Segment1_A),.o_Segment_B(o_Segment1_B),.o_Segment_C(o_Segment1_C),
   .o_Segment_D(o_Segment1_D),.o_Segment_E(o_Segment1_E),.o_Segment_F(o_Segment1_F),
   .o_Segment_G(o_Segment1_G));

  hex_to_7seg lower_digit
  (.i_Clk(i_Clk),.i_Value(display_out[3:0]),
   .o_Segment_A(o_Segment2_A),.o_Segment_B(o_Segment2_B),.o_Segment_C(o_Segment2_C),
   .o_Segment_D(o_Segment2_D),.o_Segment_E(o_Segment2_E),.o_Segment_F(o_Segment2_F),
   .o_Segment_G(o_Segment2_G));
endmodule
```

code::dive

Board definition file

```
### Main FPGA Clock
set_io i_Clk 15

## Push-Button Switches
set_io i_Switch_1 53

### LED Pins:
set_io o_LED_1 56
set_io o_LED_2 57
set_io o_LED_3 59
set_io o_LED_4 60

### 7 Segment Outputs
set_io o_Segment1_A 3
set_io o_Segment1_B 4
...
set_io o_Segment2_C 97
set_io o_Segment2_D 95
set_io o_Segment2_E 94
set_io o_Segment2_F 8
set_io o_Segment2_G 96
```

code::dive

Go Board

code::dive

Live demo

The End

More info

- Full source code at https://github.com/mmicko/grom8
- Nandland YouTube video channel
- http://www.fpga4student.com/ for beginner tutorials
- http://www.clifford.at/yosys/ for those interested in open source tools

Q & A