ESRG
Embedded Systems
Research Group

CENTROALGORITMI

Instruction Set Architecture

Embedded Systems

VeSPA

João Araújo – PG53922
Tiago Sousa – PG54261

Universidade do Minho
Escola de Engenharia
COMPETE
QREN
União Europeia
FCT Fundação para a Ciência e a Tecnologia

Agenda

- Instruction Set
- Instruction Format
- Branch Conditions
- Datapath Design
- State Machine
- Test Benches
- RTL Schematic

Instruction Set

Mnemonic	Opcode	Description
NOP	00000	No operation
ADD	00001	Addition
SUB	00010	Subtraction
OR	00011	Logical OR
AND	00100	Logical AND
NOT	00101	Logical Complement
XOR	00110	Exclusive OR
CMP	00111	Arithmetic Comparison

Instruction Set

Mnemonic	Opcode	Description
Bxx	01000	Conditional Branch
JMP	01001	Jump
JMPL	01001	Jump and Link
LD	01010	Load direct from memory
LDI	01011	Load an immediate value
LDX	01100	Load indirect through reg + offset
ST	01101	Store direct to memory
STX	01110	Store indirect through reg + offset
HLT	11111	Halt
RETI	10000	Return from Interrupt

Instruction Format

	31…27	26…22	21…17	16	15…11	10…0
NOP	opcode	0…0				
ADD, SUB, OR, AND, X OR, CMP	opcode	rdst	rs1	0	rs2	0…0
ADD, SUB, OR, AND, X OR, CMP	opcode	rdst	rs1	1	immed16	
NOT	opcode	rdst	rs1	0…0		
MOV	opcode	rdst	rs1	1	0…0	

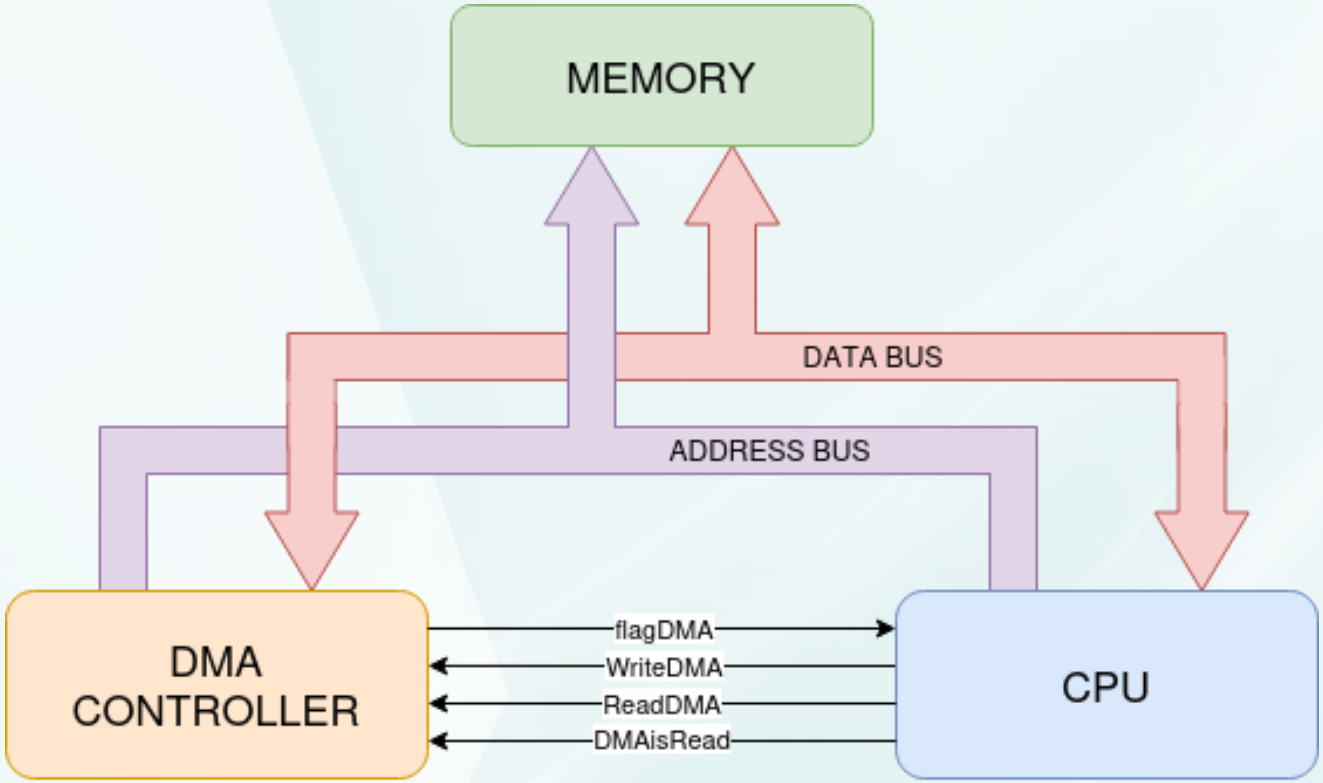
Instruction Format

	31…27	26…23	22	21…17	16	15…11
Bxx	opcode	Condition bits	immed23			
JMP	opcode	00000		rs1	0	immed16
JMPL	opcode	rdst		rs1	1	immed16
LD	opcode	rdst		immed22		
LDI	opcode	rdst		immed22		
LDX	opcode	rdst		rs1	immed17	
ST	opcode	rst		immed22		
STX	opcode	rst		rs1	immed17	
HLT	opcode	0…0				
RETI	opcode	00000		11111	0…0	

Branch Conditions

Cond	Mnemonic	Description
0000	BRA	Branch always
1000	BNV	Branch never
0001	BCC	Branch on carry clear
1001	BCS	Branch on carry set
0010	BVC	Branch on overflow clear
1010	BVS	Branch on overflow set
0011	BEQ	Branch on equal
1011	BNE	Branch on not equal
0100	BGE	Branch on greater than/equal to
1100	BLT	Branch on less than
0101	BGT	Branch on greater than
1101	BLE	Branch on less than/equal to
0110	BPL	Branch on plus
1110	BMI	Branch on minus

Top Design

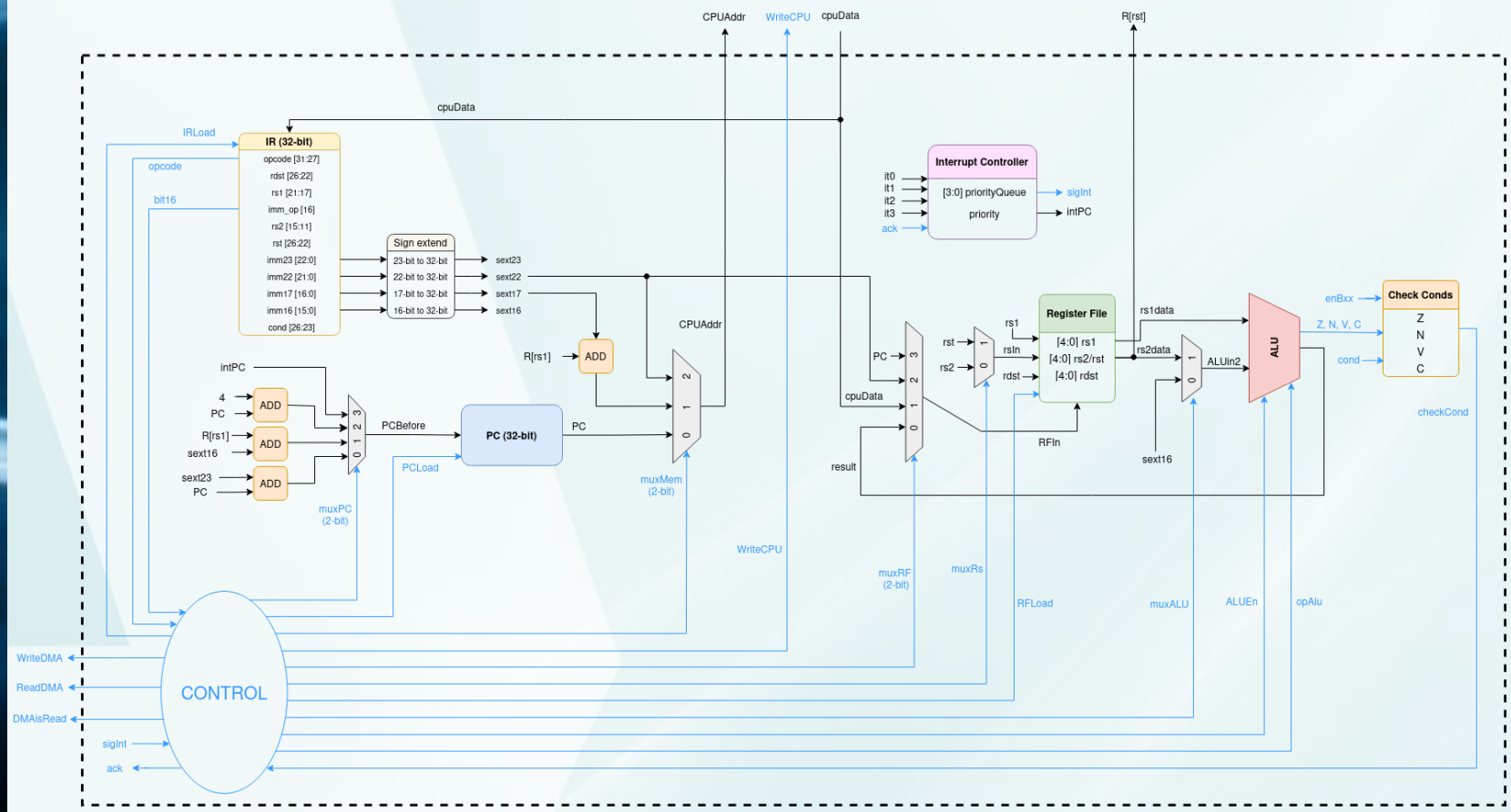

MEMORY
DATA BUS
ADDRESS BUS
DMA
CONTROLLER
flagDMA
WriteDMA
ReadDMA
DMAisRead
CPU

ESRG

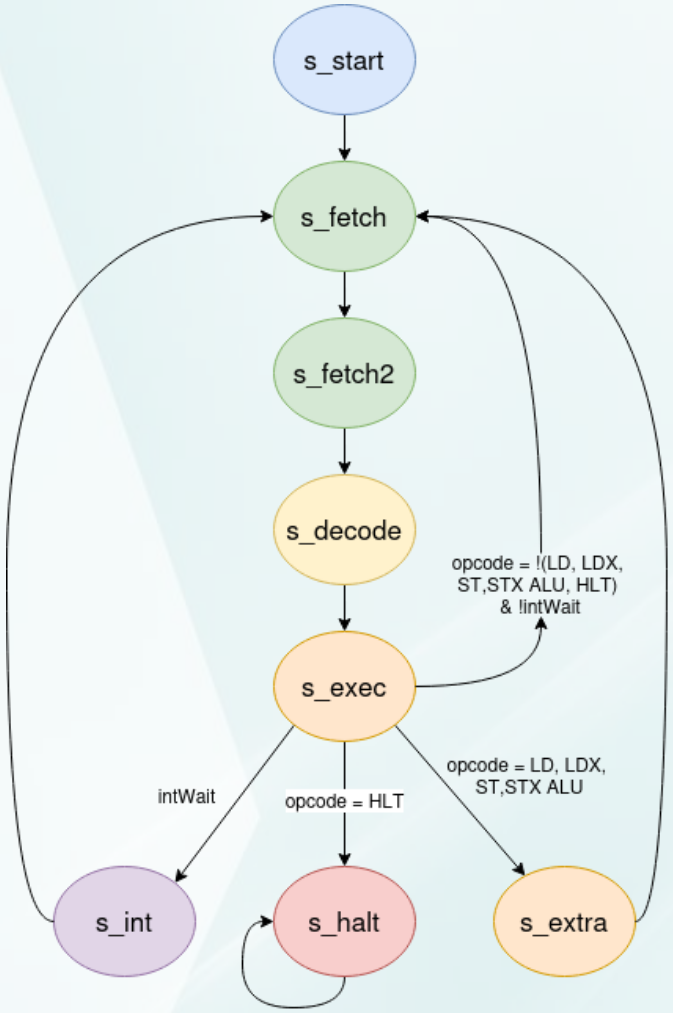
Embedded Systems Research Group

CPU Design

CENTROALGORITMI



State Machine

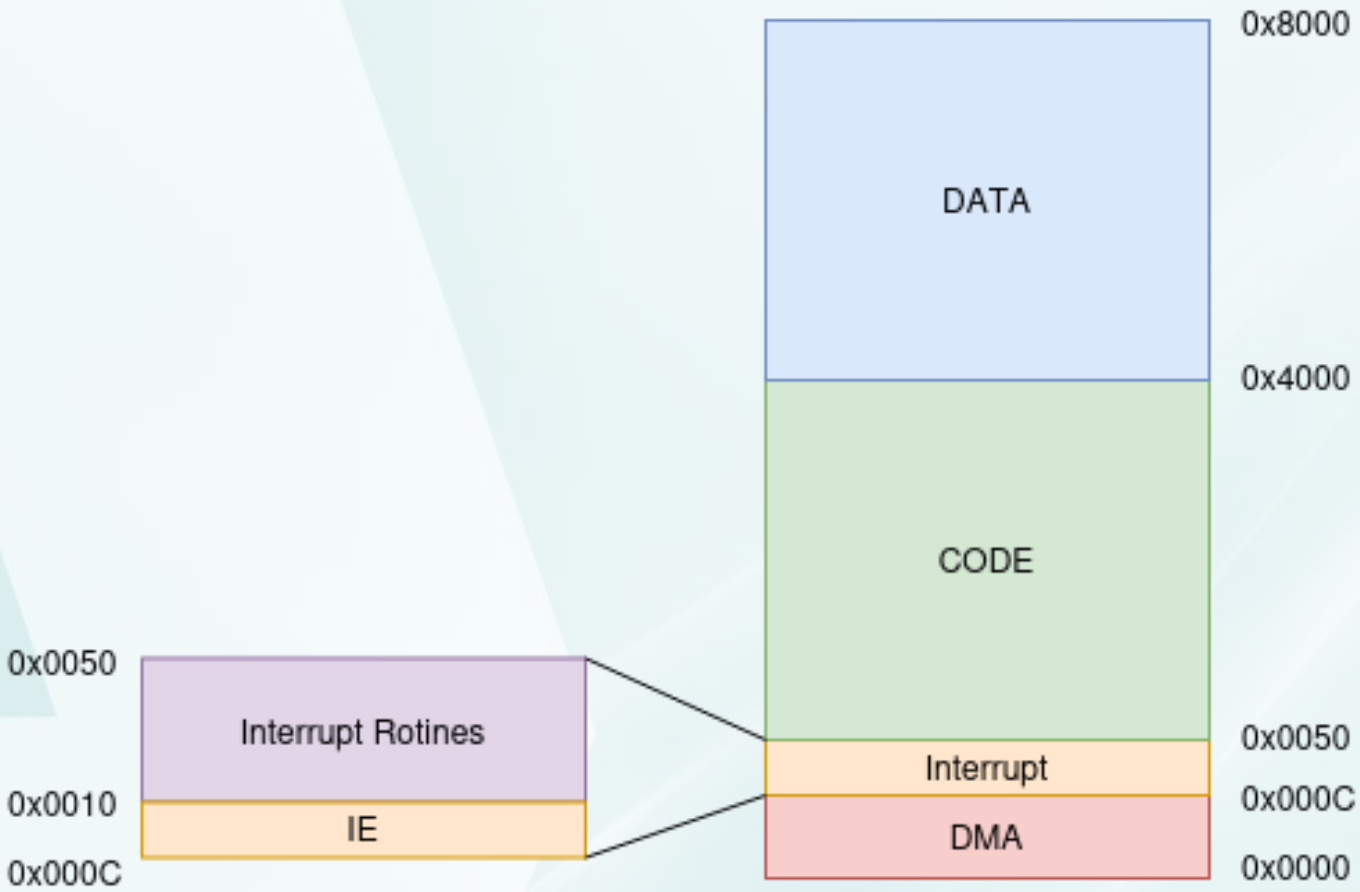


Control Signals

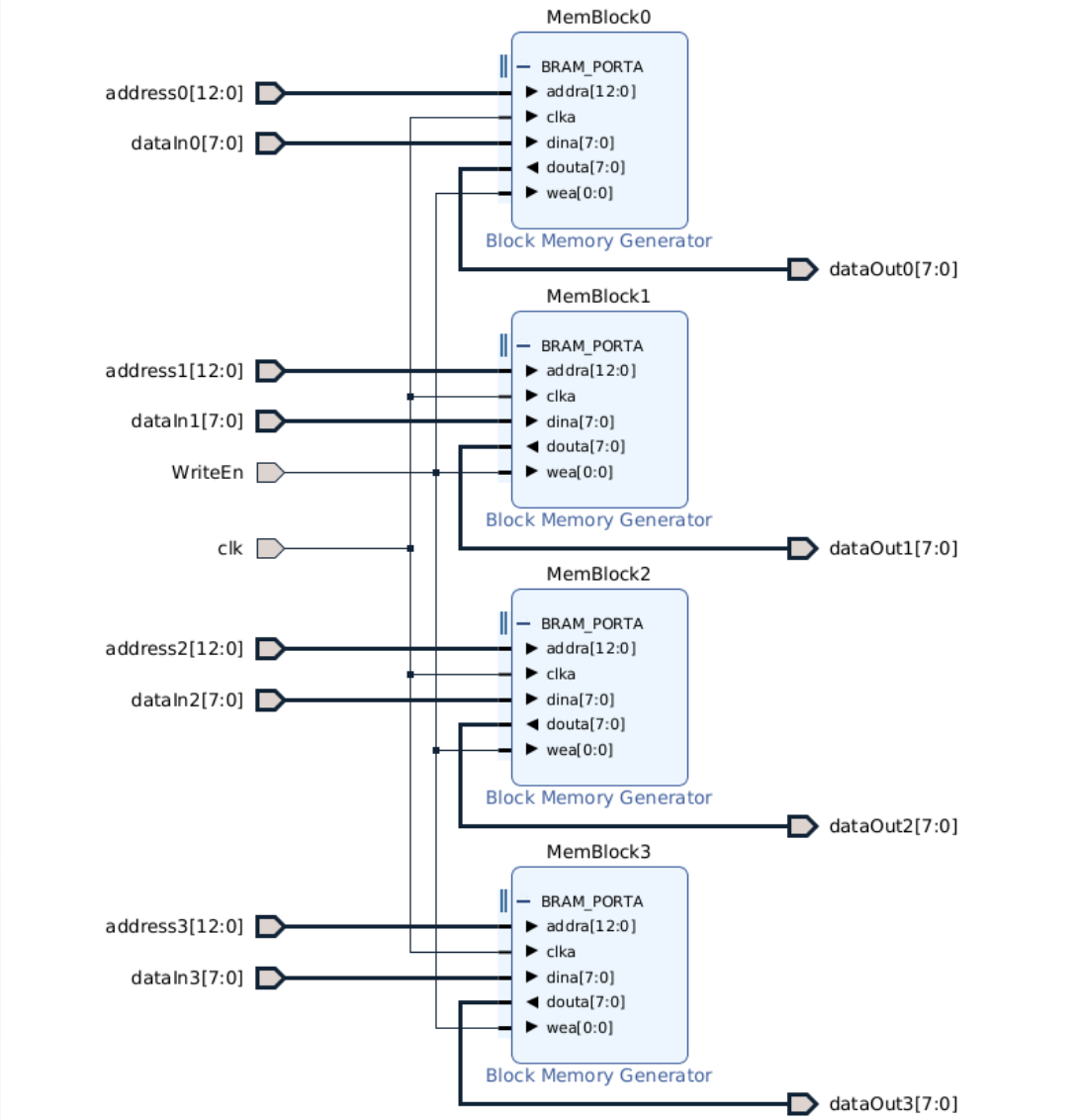
	Mux ALU	Mux Rs	Mux RF	Mux Mem	Mux PC	PC Load	RF Load	ALU En	Write CPU
ADD	0/1		0				1	1	
SUB	0/1		0				1	1	
OR	0/1		0				1	1	
AND	0/1		0				1	1	
NOT			0				1	1	
XOR	0/1		0				1	1	
CMP	0/1							1	
Bxx					0	1			
JMP					1	1	1		
JMPL			3		1	1			
LD			1	2			1		
LDI			2				1		
LDX			1	1			1		
ST		1		2					1
STX		1		1					1
NOP									
HLT									
RETI					1	1			

	mux RF	mux Mem	mux PC	PC Load	RF Load	IR Load	Read DMA	Write DMA
Fetch		1						
Fetch2			2	1		1		
Decode (dmaSwitch=0)							1	
Decode (dmaSwitch=1)								1
Int	3		3	1	1			

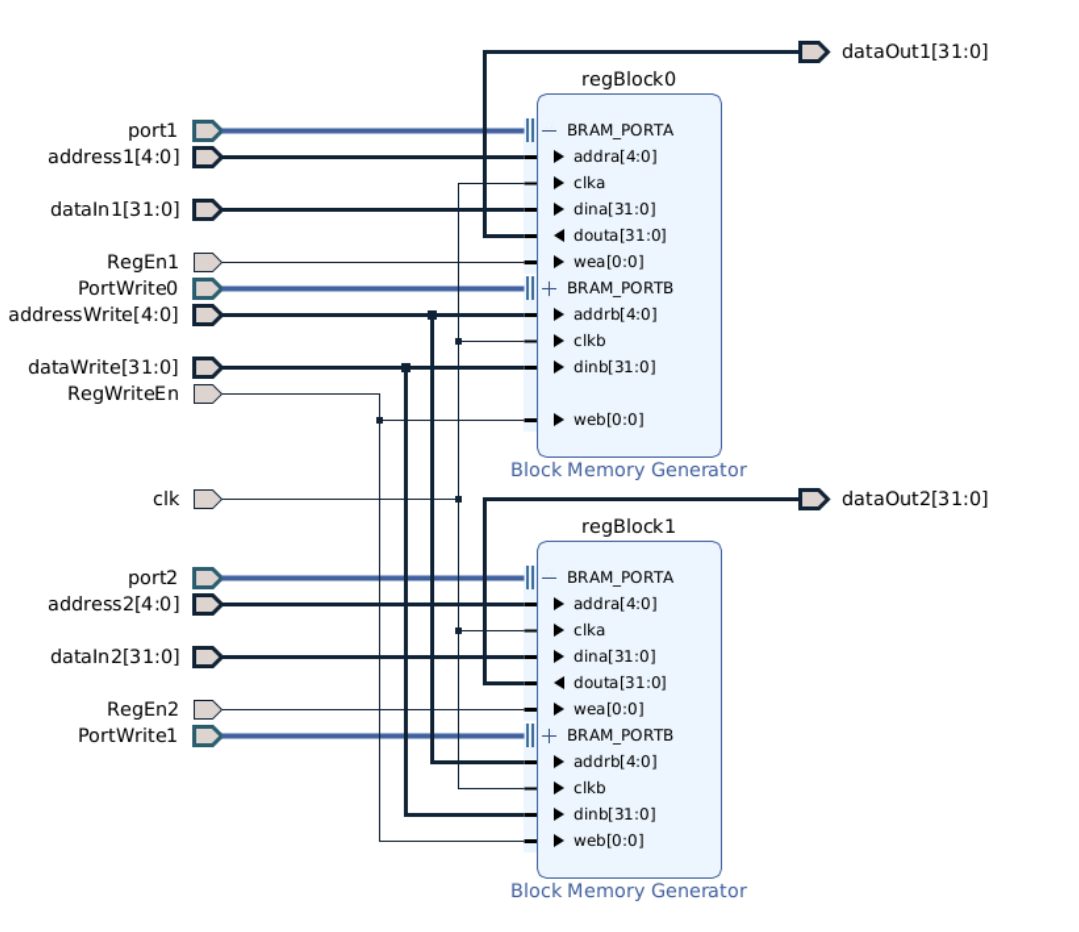
Memory Map


0x8000
DATA
0x4000
CODE
0x0050
Interrupt
0x000C
DMA
0x0000
0x0050
Interrupt Rotines
0x0010
IE
0x000C

Memory IP Design



Register File IP Design

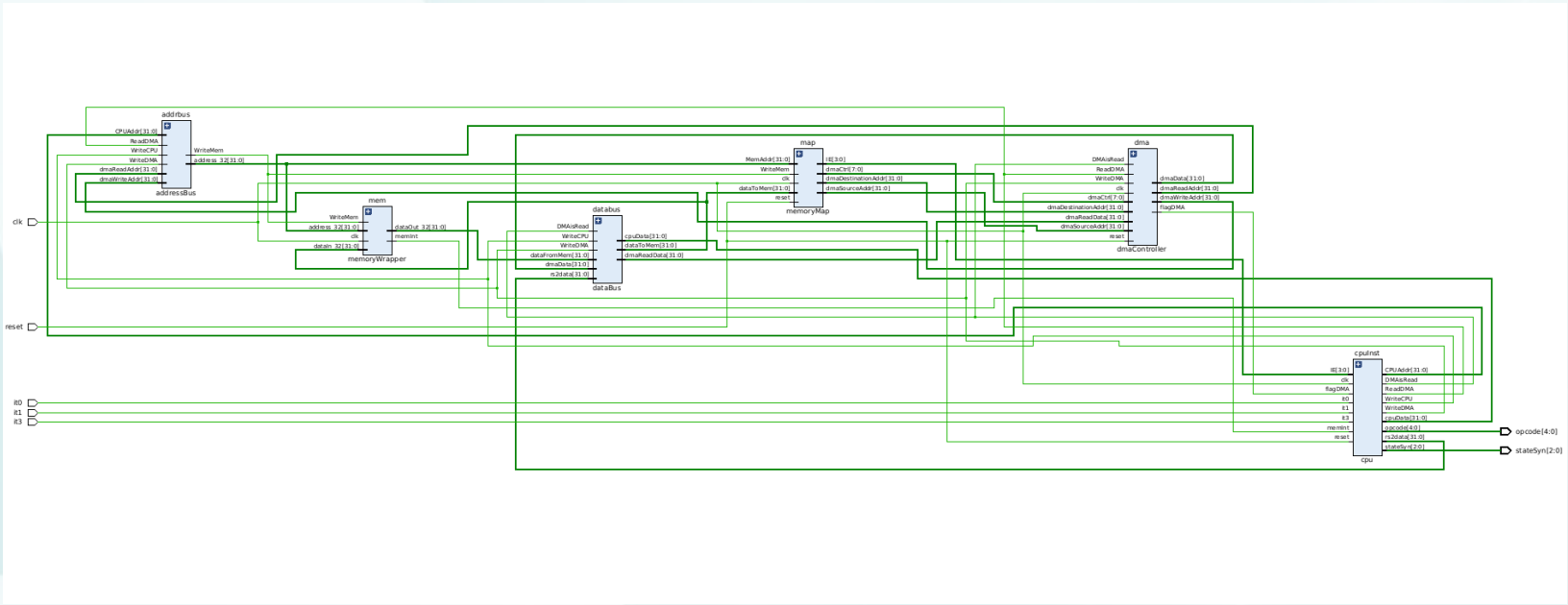


ESRG

Embedded Systems Research Group

RTL Schematic

CENTROALGORITMI



Top


```
module top(
    input clk,
    input reset,
    input it0,
    input it1,
    input it3,
    output [2:0] stateSyn,
    output [4:0] opcode
    );

    wire IRLoad, PCLoad, RFLoad, WriteCPU, bit16, ALUEn;
    wire [1:0] muxMem, muxRF, muxPC;
    wire [2:0] opAlu;
    wire [3:0] condBxxCu, IE;
    wire ack, sigInt;
    wire Z,N,C,V;
    wire clk_1hz;
    wire [7:0] dmaCtrl;
    wire [`MEM_DEPTH-1:0] intPC, address_32, CPUAddr;
    wire [`DATA_SIZE-1:0] rs2data, cpuData, dataIn_32, dataOut_32, dmaData, dmaRData;
    wire [`MEM_DEPTH-1:0] dmaWriteAddr, destinationAddr, sourceAddr, dmaRAddr;
    wire flagDMA, WriteMem, ReadDMA, WriteDMA, dmaSwitch, DMAisRead;

cpu cpuInst (
    .clk(clk),
    .reset(reset),
    .IE(IE),
    .it0(it0),
    .it1(it1),
    .it3(it3),
    .flagDMA(flagDMA),
    .memInt(memInt),
    .WriteCPU(WriteCPU),
    .WriteDMA(WriteDMA),
    .ReadDMA(ReadDMA),
    .DMAisRead(DMAisRead),
    .stateSyn(stateSyn),
    .opcode(opcode),
    .rs2data(rs2data),
    .CPUAddr(CPUAddr),
    .cpuData(cpuData)
    );
```

```
memoryWrapper mem (
    .clk(clk),
    .WriteMem(WriteMem),
    .dataIn_32(dataIn_32),
    .address_32(address_32),
    .dataOut_32(dataOut_32),
    .memInt(memInt)
    );

addressBus addrbus (
    .WriteCPU(WriteCPU),
    .WriteDMA(WriteDMA),
    .ReadDMA(ReadDMA),
    .CPUAddr(CPUAddr),
    .dmaWriteAddr(dmaWriteAddr),
    .dmaReadAddr(dmaRAddr),
    .WriteMem(WriteMem),
    .address_32(address_32)
    );

dataBus databus(
    .WriteCPU(WriteCPU),
    .WriteDMA(WriteDMA),
    .DMAisRead(DMAisRead),
    .rs2data(rs2data),
    .dmaData(dmaData),
    .dataFromMem(dataOut_32),
    .cpuData(cpuData),
    .dmaReadData(dmaRData),
    .dataToMem(dataIn_32)
    );
```

```
dmaController dma (
    .clk(clk),
    .reset(reset),
    .ReadDMA(ReadDMA),
    .DMAisRead(DMAisRead),
    .WriteDMA(WriteDMA),
    .dmaCtrl(dmaCtrl),
    .dmaDestinationAddr(destinationAddr),
    .dmaSourceAddr(sourceAddr),
    .dmaReadData(dmaRData),
    .dmaWriteAddr(dmaWriteAddr),
    .dmaReadAddr(dmaRAddr),
    .dmaData(dmaData),
    .flagDMA(flagDMA)
    );

memoryMap map (
    .clk(clk),
    .reset(reset),
    .WriteMem(WriteMem),
    .dataToMem(dataIn_32),
    .MemAddr(address_32),
    .dmaSourceAddr(sourceAddr),
    .dmaDestinationAddr(destinationAddr),
    .dmaCtrl(dmaCtrl),
    .IE(IE)
    );
endmodule
```

CPU


```
module cpu(
    input clk,
    input reset,
    input [3:0] IE,
    input it0,
    input it1,
    input it3,
    input flagDMA,
    input memInt,
    output WriteCPU,
    output WriteDMA,
    output ReadDMA,
    output DMAisRead,
    output [2:0] stateSyn,
    output [4:0] opcode,
    output [`DATA_SIZE-1:0] rs2data,
    output [`MEM_DEPTH-1:0] CPUAddr,
    output [`DATA_SIZE-1:0] cpuData
    );

    wire [`MEM_DEPTH-1:0] intPC;
    wire [3:0] condBxxCu;
    wire [2:0] opAlu;
    wire [1:0] muxPC, muxMem, muxRF;
    wire muxALU, muxRs;
    wire ALUEn, IRLoad, PCLoad, RFLoad, bit16;
    wire Z, N, C, V;
    wire sigInt, ack;

    datapath dp(
    .clk(clk),
    .reset(reset),
    .intPC(intPC),
    .opAlu(opAlu),
    .muxPC(muxPC),
    .muxMem(muxMem),
    .muxRF(muxRF),
    .muxALU(muxALU),
    .muxRs(muxRs),
    .ALUEn(ALUEn),
    .IRLoad(IRLoad),
    .PCLoad(PCLoad),
    .RFLoad(RFLoad),
    .bit16(bit16),
    .opcode(opcode),
    .condBxxCu(condBxxCu),
    .Z(Z),
    .N(N),
    .C(C),
    .V(V),
    .rs2data(rs2data),
    .CPUAddr(CPUAddr),
    .cpuData(cpuData)
    );
```

```
controlunit cu(
    .clk(clk),
    .reset(reset),
    .opcode(opcode),
    .bit16(bit16),
    .sigInt(sigInt),
    .condBxxCu(condBxxCu),
    .Z(Z),
    .N(N),
    .C(C),
    .V(V),
    .flagDMA(flagDMA),
    .muxPC(muxPC),
    .muxMem(muxMem),
    .muxRF(muxRF),
    .muxALU(muxALU),
    .muxRs(muxRs),
    .ALUEn(ALUEn),
    .IRLoad(IRLoad),
    .PCLoad(PCLoad),
    .RFLoad(RFLoad),
    .WriteCPU(WriteCPU),
    .WriteDMA(WriteDMA),
    .ReadDMA(ReadDMA),
    .DMAisRead(DMAisRead),
    .opAlu(opAlu),
    .stateSyn(stateSyn),
    .ack(ack)
    );

interruptController isr (
    .clk(clk),
    .reset(reset),
    .IE(IE),
    .it0(it0),
    .it1(it1),
    .it2(memInt),
    .it3(it3),
    .ack(ack),
    .sigInt(sigInt),
    .intPC(intPC)
    );

endmodule
```

Datapath

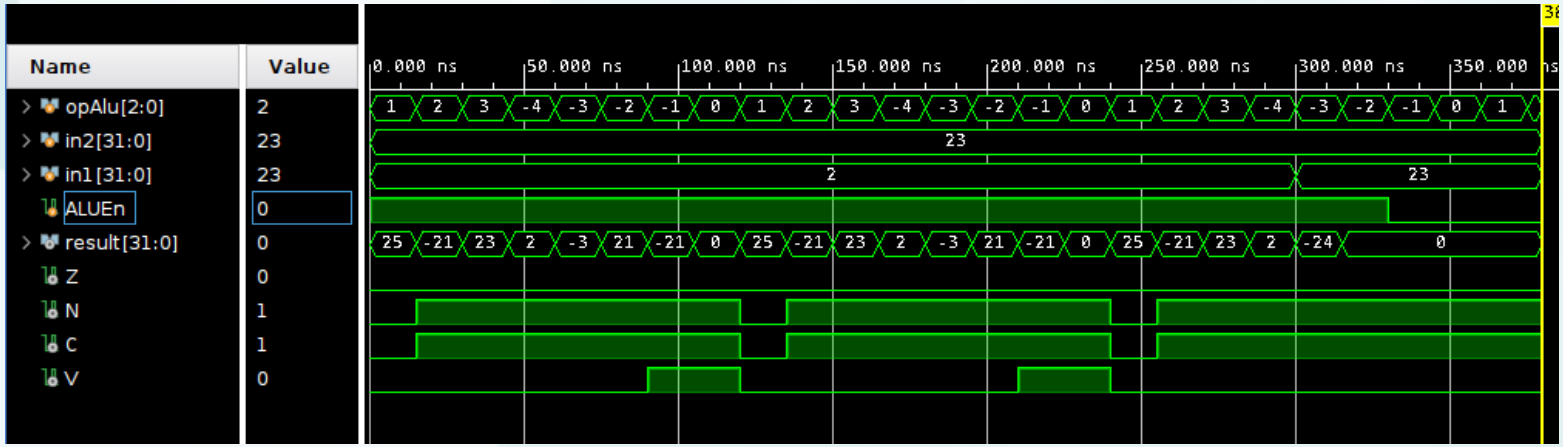

```
module datapath(
    input clk,
    input reset,
    input [`MEM_DEPTH-1:0] intPC,
    input [2:0] opAlu,
    input [1:0] muxPC,
    input [1:0] muxMem,
    input [1:0] muxRF,
    input muxALU,
    input muxRs,
    input ALUEn,
    input IRLoad,
    input PCLoad,
    input RFLoad,

    output bit16,
    output [4:0] opcode,
    output [3:0] condBxxCu,
    output Z,
    output N,
    output C,
    output V,
    output [`DATA_SIZE-1:0] rs2data,
    output [`MEM_DEPTH-1:0] CPUAddr,
    output [`DATA_SIZE-1:0] cpuData
    );

    reg [`DATA_SIZE-1:0] IR  ;
    reg [`MEM_DEPTH-1:0] PC  ;
    wire [`DATA_SIZE-1:0] result;

    wire [4:0] rdst;                        // destination register
    wire [4:0] rst =        IR[26:22];      // store register
    wire [4:0] rs1 =        IR[21:17];      // first operand register
    wire [4:0] rs2 =        IR[15:11];      // second operand register
    wire [3:0] cond =       IR[26:23];      // condition
    wire [15:0] immed16 =   IR[15:0];       // 16 bit immediate
    wire [16:0] immed17 =   IR[16:0];       // 17 bit immediate
    wire [21:0] immed22 =   IR[21:0];       // 22 bit immediate
    wire [22:0] immed23 =   IR[22:0];       // 23 bit immediate

    wire [`DATA_SIZE-1:0] sext16, sext17, sext22, sext23;
    wire [`MEM_DEPTH-1:0] nextPC;
    wire [`DATA_SIZE-1:0] RFIn;
    wire [`DATA_SIZE:0] rs1data;
    wire [4:0] rsIn;
    wire [`DATA_SIZE-1:0] ALUin2;

    assign rdst = (muxPC != 2'b11) ? IR[26:22] : 5'b11111;

    assign sext16[`DATA_SIZE-1:0] = { {(`DATA_SIZE-16){immed16[15]}} , immed16};
    assign sext17[`DATA_SIZE-1:0] = { {(`DATA_SIZE-17){immed17[16]}} , immed17};
    assign sext22[`DATA_SIZE-1:0] = { {(`DATA_SIZE-22){immed22[21]}} , immed22};
    assign sext23[`DATA_SIZE-1:0] = { {(`DATA_SIZE-23){immed23[22]}} , immed23};

      always@(posedge clk) begin
            if (reset) begin
                IR <= 32'd0;
            end if (IRLoad) begin
                IR <= cpuData;
            end
    end
```

```
    assign nextPC = (muxPC == 2'd0 ) ?  PC  + sext23 :
                    (muxPC == 2'd1 ) ?  rs1data + sext16 :
                    (muxPC == 2'd2 ) ?  PC + 4 :
                    (muxPC == 2'd3 ) ?  intPC : 2'bx;

    always@(posedge clk) begin
        if (reset) begin
            PC <= 32'h00000050;
        end if (PCLoad) begin
            PC <= nextPC;
        end
    end

    assign CPUAddr =    (muxMem == 2'b00 ) ?  PC :
                        (muxMem == 2'b01 ) ?  rs1data + sext17 :
                        (muxMem == 2'b10 ) ?  sext22  : 2'bx   ;

    assign RFIn =   (muxRF == 2'b00 ) ?  result :
                    (muxRF == 2'b01 ) ?  cpuData :
                    (muxRF == 2'b10 ) ?  sext22 :
                    (muxRF == 2'b11 ) ? PC : 2'bx;

    assign rsIn =   (muxRs == 1'b0 ) ? rs2 :
                    (muxRs == 1'b1 ) ? rst : 1'bx;

    RegFileHandler RF(
        .clk(clk),
        .WriteRF(RFLoad),
        .readAddr1(rs1),
        .readAddr2(rsIn),
        .writeAddr(rdst),
        .writeData(RFIn),
        .readData1(rs1data),
        .readData2(rs2data)
        );

    assign ALUin2 =    (muxALU == 1'b0 ) ?  sext16 :
                       (muxALU == 1'b1 ) ?  rs2data : 1'bx;
    alu al (
        .ALUEn(ALUEn),
        .opAlu(opAlu),
        .IN1(rs1data),
        .IN2(ALUin2),
        .Z(Z),
        .N(N),
        .C(C),
        .V(V),
        .result(result)
    );

    assign condBxxCu = cond;
    assign opcode = IR[31:27];
    assign bit16 = IR[16];
```

ALU


```
`define ADD_  (opAlu == 1)
`define SUB_  (opAlu == 2)
`define OR_   (opAlu == 3)
`define AND_  (opAlu == 4)
`define NOT_  (opAlu == 5)
`define XOR_  (opAlu == 6)
`define CMP_  (opAlu == 7)
`define ALUflags (`ADD_||`SUB_||`CMP_)
`define WIDTH 32
```

```
assign {carryAux, resultAux} =  (`ADD_) ? IN1 + IN2 :
                                (`SUB_) ? IN1 - IN2 :
                                (`OR_ ) ? IN1 | IN2 :
                                (`AND_) ? IN1 & IN2 :
                                (`NOT_) ? ~IN1 :
                                (`XOR_) ? IN1 ^ IN2 :
                                (`CMP_) ? IN1 - IN2 : 0;

assign {carry, result} = (ALUEn) ? {carryAux, resultAux} : 0;

assign sigSubb = (`SUB_) ? 1'b1 : 1'b0;

always @(*) begin
    if (ALUEn && ALUflags) begin
        Z <= ~(|result[`WIDTH-1:0]);
        C <= carry;
        N <= result[`WIDTH-1];
        V <= (result[`WIDTH-1] & ~IN1[`WIDTH-1] & ~(sigSubb ^ IN2[`WIDTH-1])) |
                        (~result[`WIDTH-1] & IN1[`WIDTH-1] & (sigSubb ^ IN2[`WIDTH-1]));
    end
end
```



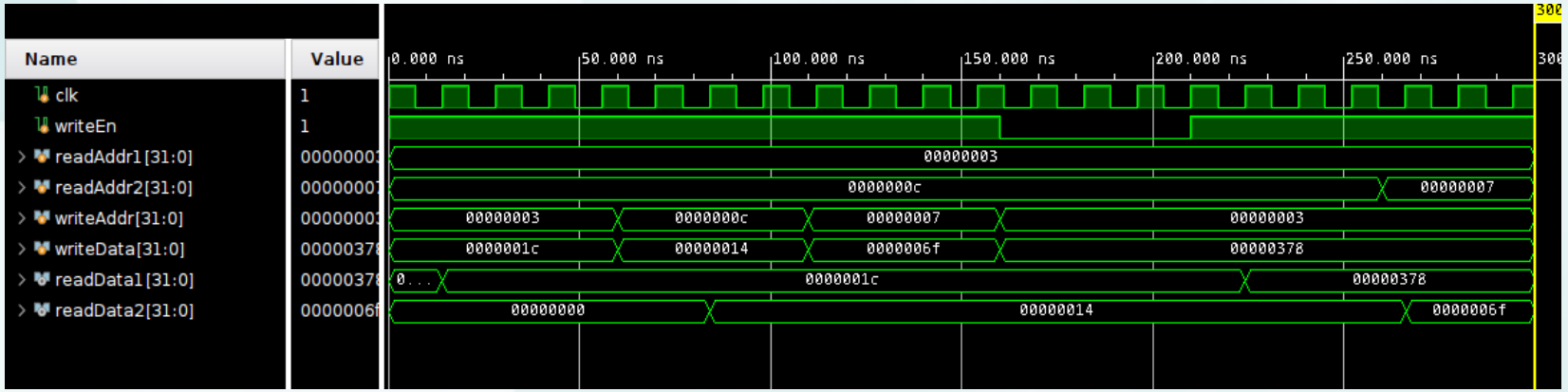
ESRG

Embedded Systems Research Group

Test Bench Register File

CENTROALGORITMI

```
`define REG_WIDTH 32
`define REG_DEPTH 5

module RegFileHandler(
    input   clk,
    input   WriteRF,
    input   [`REG_DEPTH-1:0] readAddr1,
    input   [`REG_DEPTH-1:0] readAddr2,
    input   [`REG_DEPTH-1:0] writeAddr,
    input   [`REG_WIDTH-1:0] writeData,
    output  [`REG_WIDTH-1:0] readData1,
    output  [`REG_WIDTH-1:0] readData2
    );

    RegFileIP regInst(
        .clk(clk),
        .address1(readAddr1),
        .address2(readAddr2),
        .RegWriteEn(WriteRF),
        .addressWrite(writeAddr),
        .dataWrite(writeData),
        .dataOut1(readData1),
        .dataOut2(readData2)
        );

endmodule
```



ESRG

Embedded Systems Research Group

Control Unit

CENTROALGORITMI

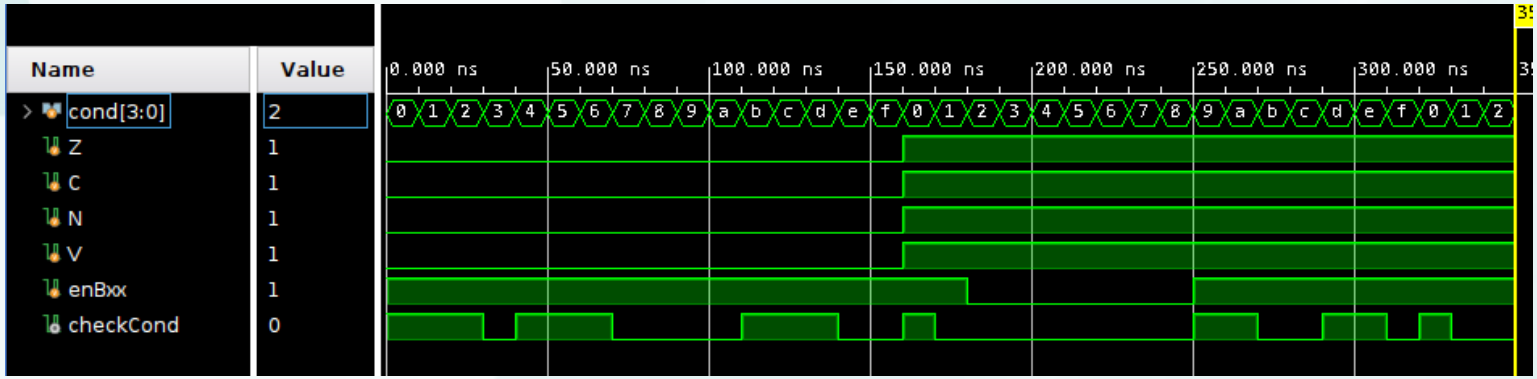
```
module controlunit(
    input clk,
    input reset,
    input [4:0] opcode,
    input bit16,
    input sigInt,
    input [3:0] condBxxCu,
    input Z,
    input N,
    input C,
    input V,
    input flagDMA,

    output [1:0] muxPC,
    output [1:0] muxMem,
    output [1:0] muxRF,
    output muxALU,
    output muxRs,
    output ALUEn,
    output IRLoad,
    output PCLoad,
    output RFLoad,
    output WriteCPU,
    output WriteDMA,
    output ReadDMA,
    output DMAisRead,
    output [2:0] opAlu,
    output [2:0] stateSyn,
    output ack
);

parameter   s_start = 5'd0, s_fetch = 5'd1, s_fetch2 = 5'd2, s_decode = 5'd3,
 s_exec = 5'd4, s_extra = 5'd5, s_halt = 5'd6, s_int = 5'd7;

reg [3:0] state;
wire enBxx;
reg dmaSwitch;
reg intProc;
reg intWait;

always @(posedge clk) begin
    if (reset == 1'b1) begin
        state <= s_start;
        intProc <= 0;
        intWait <= 0;
        dmaSwitch <= 1;
    end
    else begin
        case (state)
            s_start:
                state <= s_fetch;
            s_fetch:
                if (flagDMA) begin
                    dmaSwitch <= dmaSwitch + 1;
                    state <= s_fetch2;
                end else begin
                    state <= s_fetch2;
                end
            s_fetch2:
                state <= s_decode;
            s_decode:
                state <= s_exec;
            s_exec:
                if(`HLT) begin
                    state <= s_halt;
                end else if (`LD||`LDX||`ALU||`ST||`STX) begin
                    state <= s_extra;
                end else if(intWait) begin
                    intWait <= 0;
                    intProc <= 1;
                    state <= s_int;
                    end
                    else begin
                        state <= s_fetch;
                    end
                end
```

```
            s_extra:
                if(intWait) begin
                    intWait <= 0;
                    intProc <= 1;
                    state <= s_int;
                end else begin
                    state <= s_fetch;
                end
            s_halt:
                state <= s_halt;
            s_int:
                state <= s_fetch;
            default:
                state <= s_start;
        endcase
        if (sigInt) begin
            if(intProc == 0 && intWait == 0) begin
                intWait <= 1;
            end
        end
    end
end

assign muxRs = ((`ST || `STX) && (state == s_exec || state == s_extra)) ? 1'd1 : 1'd0;

assign muxALU = (bit16 && (`ALU) && (state == s_exec)) ? 1'b0 : 1'b1;

assign muxPC = (`Bxx && (state == s_exec)) ? 2'd0 :
               ((`JMP||`JMPL||`RETI) && (state == s_exec)) ? 2'd1 :
               (state == s_fetch2) ? 2'd2 :
               (state == s_int) ? 2'd3 : 2'b10;

assign muxMem = ((`LD || `ST) && (state == s_exec) || (`ST && state == s_extra)) ? 2'd2 :
                ((`LDX || `STX) && (state == s_exec) || (`STX && state == s_extra)) ? 2'd1 :
                (state == s_fetch) ? 2'd0 : 2'bx;

assign muxRF = ((`ALU) && (state == s_exec)) ? 2'd0 :
               ((`LDX || `LD) && ((state == s_exec) || state == s_extra )) ? 2'd1 :
               ((`LDI) && (state == s_exec)) ? 2'd2 :
               ((`JMPL) && (state == s_exec)) || (state == s_int) ? 2'd3 : 2'bx;

assign ALUEn =   (`ALU && (state == s_exec))  ? 1'd1 : 1'd0;
assign IRLoad =  (state == s_fetch2) ? 1'd1 : 1'd0;
assign PCLoad =  ((((`Bxx && checkCond) ||`JMP ||`JMPL||`RETI) && (state == s_exec)) ||
                 state == s_fetch2 || (state == s_int)) ? 1'd1 : 1'd0;
assign RFLoad =  ((`ADD||`SUB||`OR||`AND||`NOT||`XOR||`JMPL||`LD||`LDI||`LDX) && state == s_exec) ||
                 ((`LD ||`LDX) && (state == s_extra)) || (state == s_int) ? 1'd1 : 1'd0;
assign WriteCPU = ((`ST || `STX) && (state == s_exec || state == s_extra)) ? 1'd1 : 1'd0;

assign ReadDMA = (flagDMA && (state == s_decode) && dmaSwitch == 0) ? 1'd1 : 1'd0;
assign DMAisRead = (flagDMA && state == s_exec && dmaSwitch == 0) ? 1'd1 : 1'd0;
assign WriteDMA = (flagDMA && state == s_decode && dmaSwitch == 1) ? 1'd1 : 1'd0;

assign enBxx = (opcode == 8) ? 1'b1 : 1'b0;
assign opAlu = (`ALU && (state == s_exec)) ? opcode[2:0] : 0;
assign stateSyn = state;

assign ack = (intProc && `RETI) ? 1'd1 : 1'd0;

checkCondition ccond(
    .enBxx(enBxx),
    .cond(condBxxCu),
    .Z(Z),
    .N(N),
    .C(C),
    .V(V),
    .checkCond(checkCond)
    );
```

Test Bench CheckCond


```
`define BRA 4'b0000  // 0: always
`define BNV 4'b1000  // 8: never
`define BCC 4'b0001  // 1: carry clear
`define BCS 4'b1001  // 9: carry set
`define BVC 4'b0010  // 2: overflow clear
`define BVS 4'b1010  // a: overflow set
`define BEQ 4'b0011  // 3: equal
`define BNE 4'b1011  // b: not equal
`define BGE 4'b0100  // 4: greater than/ equal to
`define BLT 4'b1100  // c: less than
`define BGT 4'b0101  // 5: greater than
`define BLE 4'b1101  // d: less than/ equal to
`define BPL 4'b0110  // 6: on plus
`define BMI 4'b1110  // e: on minus
```

```
assign checkCondAux =   (cond == `BRA) ? 1'd1 :
                        (cond == `BNV) ? 1'd0 :
                        (cond == `BCC) ? ~C   :
                        (cond == `BCS) ? C    :
                        (cond == `BVC) ? ~V   :
                        (cond == `BVS) ? V    :
                        (cond == `BEQ) ? Z    :
                        (cond == `BNE) ? ~Z   :
                        (cond == `BGE) ?  (N & V) | (~N & ~V)    :
                        (cond == `BLT) ? ~(N & V) | (~N & ~V)    :
                        (cond == `BGT) ? ~Z & ((N & V) | (~N & ~V))    :
                        (cond == `BLE) ?  Z | ((N & V) | (~N & ~V))    :
                        (cond == `BPL) ? ~N   :
                        (cond == `BMI) ?  N   :  1'b0;

assign checkCond = enBxx ? checkCondAux : 1'b0;
```



Interrupt Controller


```
module interruptController(
    input clk,
    input reset,
    input [3:0] IE,
    input it0,
    input it1,
    input it2,
    input it3,
    input ack,

    output reg sigInt,
    output reg [`MEM_DEPTH-1:0] intPC
);
    reg [3:0] priorityQueue;
    reg [1:0] priority;

    always @(posedge clk) begin
        if (reset) begin
            priorityQueue <= 4'b0000;
        end else if (ack) begin
            priorityQueue[priority] <= 1'b0;
        end
    end

    always @* begin
        if (priorityQueue != 4'b0000) begin
            if (priorityQueue[3]) begin
                intPC <=  32'h00000010;
                priority <= 3;
            end else if (priorityQueue[2]) begin
                intPC <=  32'h00000020;
                priority <= 2;
            end else if (priorityQueue[1]) begin
                intPC <=  32'h00000030;
                priority <= 1;
            end else if (priorityQueue[0]) begin
                intPC <=  32'h00000040;
                priority <= 0;
            end else begin
                intPC <=  32'bx;
            end
            sigInt <= 1'b1;
        end else begin
            sigInt <= 1'b0;
            intPC <= 32'bx;
            priority <= 1'bx;
        end
        priorityQueue <= priorityQueue | {it3 & IE[3], it2 & IE[2], it1 & IE[1], it0 & IE[0]};
    end
endmodule
```

Interrupt	Priority	Interrupt Enable	intPC
it0	0	IE[0]	0x10
it1	1	IE[1]	0x20
it2	2	IE[2]	0x30
it3	3	IE[3]	0x40

ESRG

Embedded Systems Research Group

Test Bench InterruptController

CENTROALGORITMI

Memory


```
module memoryWrapper(
    input clk,
    input WriteMem,
    input [`MEM_WIDTH-1:0]dataIn_32,
    input [`MEM_DEPTH-1:0]address_32,
    output [`MEM_WIDTH-1:0]dataOut_32,
    output memInt
    );

    wire [`REAL_MEM_SIZE-1:0] address_15 = address_32[`REAL_MEM_SIZE-1:0];
    wire [`REAL_MEM_DEPTH-1:0] address0;
    wire [`REAL_MEM_DEPTH-1:0] address1;
    wire [`REAL_MEM_DEPTH-1:0] address2;
    wire [`REAL_MEM_DEPTH-1:0] address3;

    wire [`MEM_WIDTH-1:0] dataIn0;
    wire [`MEM_WIDTH-1:0] dataIn1;
    wire [`MEM_WIDTH-1:0] dataIn2;
    wire [`MEM_WIDTH-1:0] dataIn3;

    wire [`IP_WIDTH-1:0] dataOut0;
    wire [`IP_WIDTH-1:0] dataOut1;
    wire [`IP_WIDTH-1:0] dataOut2;
    wire [`IP_WIDTH-1:0] dataOut3;

    reg bit0;
    reg bit1;
    wire [`MEM_WIDTH-1:0] dataInput;
    wire WriteMemCond;

    assign WriteMemCond = ((address_32[`REAL_MEM_SIZE-1] & WriteMem &
                            ~|address_32[`MEM_DEPTH-1:`REAL_MEM_SIZE])|
                           (~|address_32[`MEM_WIDTH-1:4] & WriteMem));
    assign memInt = (~WriteMemCond & WriteMem);

    always @(*) begin
        if (address_15[0] == 1'b1 || address_15[0] == 1'b0) begin
            bit0 <= address_15[0];
            bit1 <= address_15[1];
        end
    end
```

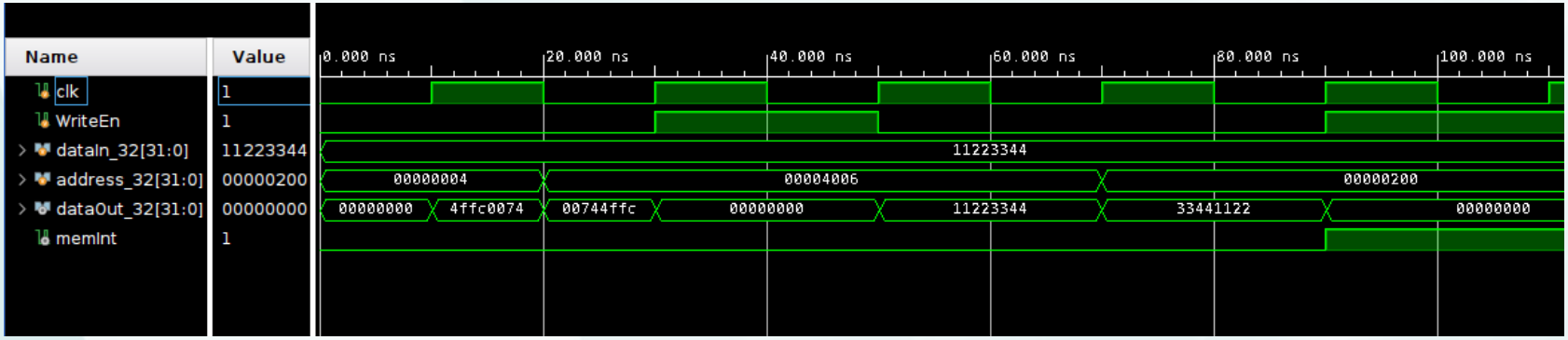
```
assign address0 = (bit1 | bit0) ? address_15[`REAL_MEM_SIZE-1:2]+1 : address_15[`REAL_MEM_SIZE-1:2];
assign address1 = (bit1)        ? address_15[`REAL_MEM_SIZE-1:2]+1 : address_15[`REAL_MEM_SIZE-1:2];
assign address2 = (bit1 & bit0) ? address_15[`REAL_MEM_SIZE-1:2]+1 : address_15[`REAL_MEM_SIZE-1:2];
assign address3 = address_15[`REAL_MEM_SIZE-1:2];

assign dataInput = (~bit1 & ~bit0)  ? {dataIn_32[31:24], dataIn_32[23:16], dataIn_32[15:8], dataIn_32[7:0]} :
                   (~bit1 & bit0)   ? {dataIn_32[23:16], dataIn_32[15:8], dataIn_32[7:0], dataIn_32[31:24]} :
                   (bit1 & ~bit0)   ? {dataIn_32[15:8], dataIn_32[7:0], dataIn_32[31:24], dataIn_32[23:16]} :
                   (bit1 & bit0)    ? {dataIn_32[7:0], dataIn_32[31:24], dataIn_32[23:16], dataIn_32[15:8]} :

assign dataIn0 [`MEM_WIDTH-1:0] = dataInput[31:24];
assign dataIn1 [`MEM_WIDTH-1:0] = dataInput[23:16];
assign dataIn2 [`MEM_WIDTH-1:0] = dataInput[15:8];
assign dataIn3 [`MEM_WIDTH-1:0] = dataInput[7:0];

assign dataOut_32 = (~bit1 & ~bit0)? {dataOut0, dataOut1, dataOut2, dataOut3} :
                    (~bit1 & bit0) ? {dataOut1, dataOut2, dataOut3, dataOut0} :
                    (bit1 & ~bit0) ? {dataOut2, dataOut3, dataOut0, dataOut1} :
                    (bit1 & bit0)  ? {dataOut3, dataOut0, dataOut1, dataOut0} : 32'bx;

memory_ip memIP (
.WriteEn(WriteMemCond),
.address0(address0),
.address1(address1),
.address2(address2),
.address3(address3),
.clk(clk),
.dataIn0(dataIn0),
.dataIn1(dataIn1),
.dataIn2(dataIn2),
.dataIn3(dataIn3),
.dataOut0(dataOut0),
.dataOut1(dataOut1),
.dataOut2(dataOut2),
.dataOut3(dataOut3)
);
```

ESRG

Embedded Systems Research Group

Test Bench Memory

CENTROALGORITMI



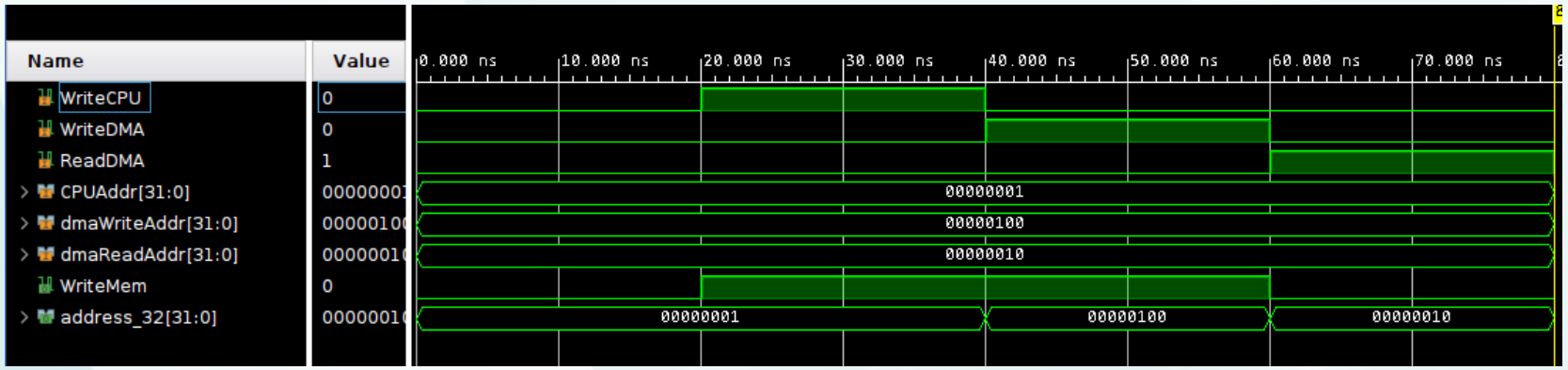
Address Bus


```
`define MEM_DEPTH 32
`define DATA_SIZE 32

// the Address Bus is responsible for selecting the appropriate memory address based on the control signals
module addressBus(
    input WriteCPU,                         // cpu memory write enable signal
    input WriteDMA,                         // dma memory write enable signal
    input ReadDMA,                          // dma memory read enable signal
    input [`MEM_DEPTH-1:0] CPUAddr,         // cpu memory address to read or write
    input [`MEM_DEPTH-1:0] dmaWriteAddr,    // dma memory address to write
    input [`MEM_DEPTH-1:0] dmaReadAddr,     // dma memory address to read

    output WriteMem,                        // memory write enable signal
    output [`MEM_DEPTH-1:0] address_32      // memory address
    );

    assign address_32 = (WriteCPU == 1) ? CPUAddr       :
                        (WriteDMA == 1) ? dmaWriteAddr  :
                        (ReadDMA == 1)  ? dmaReadAddr   : CPUAddr;

    assign WriteMem = ((WriteCPU != WriteDMA) && ReadDMA == 0) ? 1 : 0;
endmodule
```

Data Bus


```
`define MEM_DEPTH 32
`define DATA_SIZE 32

// the Data Bus is responsible for selecting the appropriate data to write in the memory
// and attributing the data read from the memory to the appropriate variable (dmaRead or cpuData)
module dataBus(
    input WriteCPU,                         // cpu memory write enable signal
    input WriteDMA,                         // dma memory write enable signal
    input DMAisRead,                        // dma read signal (ReadDMA cant be enabled on s_exec)
    input [`DATA_SIZE-1:0] rs2data,         // cpu data to write in memory
    input [`DATA_SIZE-1:0] dmaData,         // dma data to write in memory
    input [`DATA_SIZE-1:0] dataFromMem,     // memory data to be read by cpu or dma

    output [`DATA_SIZE-1:0] cpuData,        // cpu data to be read from memory
    output [`DATA_SIZE-1:0] dmaReadData,    // dma data to be read from memory
    output [`DATA_SIZE-1:0] dataToMem       // data to write in memory
    );

    assign dataToMem =  (WriteCPU == 1)  ? rs2data   :
                        (WriteDMA == 1) ? dmaData   : 32'bx;

    assign dmaReadData = (DMAisRead == 1) ? dataFromMem : 32'bx;
    assign cpuData = (DMAisRead == 0)  ? dataFromMem :32'bx;
endmodule
```

Test Bench AddressBus

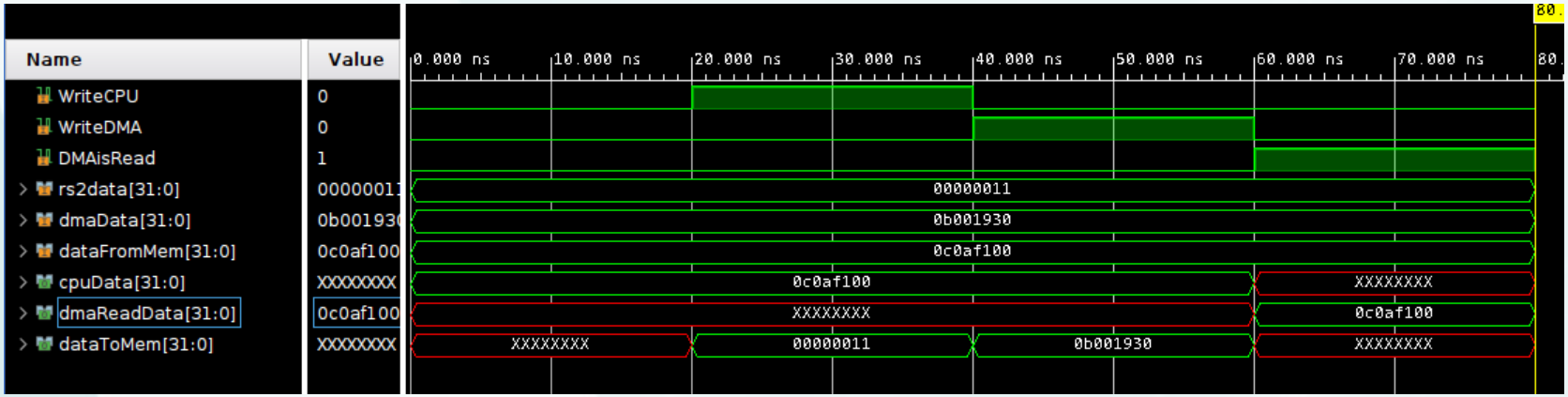


ESRG

Embedded Systems Research Group

Test Bench DataBus

CENTROALGORITMI



ESRG

Embedded Systems Research Group

DMA Controller

CENTROALGORITMI

```
`define DATA_SIZE 32
`define MEM_DEPTH 32
`define S_IDLE 0
`define S_TRANSFER 1
`define S_END 2

module dmaController(
    input clk,
    input reset,
    input ReadDMA,
    input DMAisRead,
    input WriteDMA,
    input [7:0] dmaCtrl,
    input [`MEM_DEPTH-1:0] dmaDestinationAddr,
    input [`MEM_DEPTH-1:0] dmaSourceAddr,
    input [`DATA_SIZE-1:0] dmaReadData,
    output reg [`MEM_DEPTH-1:0] dmaWriteAddr,
    output reg [`MEM_DEPTH-1:0] dmaReadAddr,
    output reg [`DATA_SIZE-1:0] dmaData,
    output reg flagDMA
    );

    reg [7:0] counter;
    reg [1:0] state;
    wire start = dmaCtrl[7];
    wire [6:0]transferLength = dmaCtrl[6:0];

    always @(*) begin
        if (flagDMA && DMAisRead) begin
            dmaData <= dmaReadData;
        end
    end
```

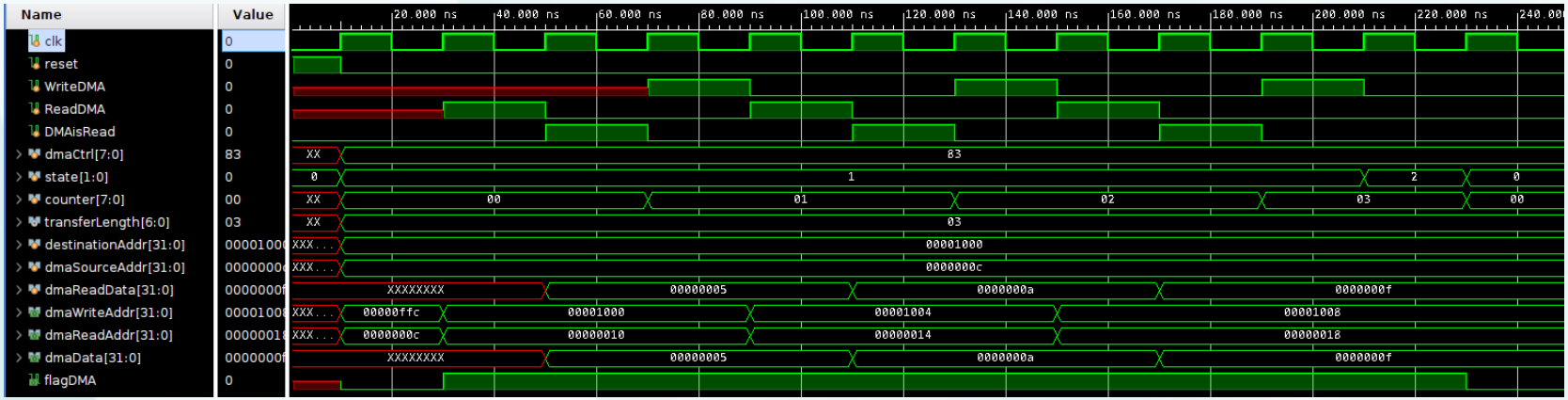
```
    always @(posedge clk) begin
        if (reset) begin
            state <= `S_IDLE;
        end else begin
            case(state)
                `S_IDLE: begin
                    flagDMA <=  2'b0;
                    if (start == 1) begin
                        state <= `S_TRANSFER;
                        counter <= 8'h00;
                        dmaWriteAddr <= dmaDestinationAddr - 4;
                        dmaReadAddr <= dmaSourceAddr;
                    end
                end
                `S_TRANSFER: begin
                    if (counter < transferLength) begin
                        flagDMA <= 1;
                        if (ReadDMA) begin
                            dmaWriteAddr = dmaWriteAddr + 4;
                            dmaReadAddr = dmaReadAddr + 4;
                        end
                        if (WriteDMA) begin
                            counter <= counter + 1;
                        end
                    end else begin
                        state <= `S_END;
                    end
                end
                `S_END: begin
                    state <= `S_IDLE;
                    flagDMA <= 0;
                    counter <= 0;
                end
            endcase
        end
    end
endmodule
```

ESRG

Embedded Systems Research Group

Test Bench DMA Controller

CENTROALGORITMI



Memory Mapping

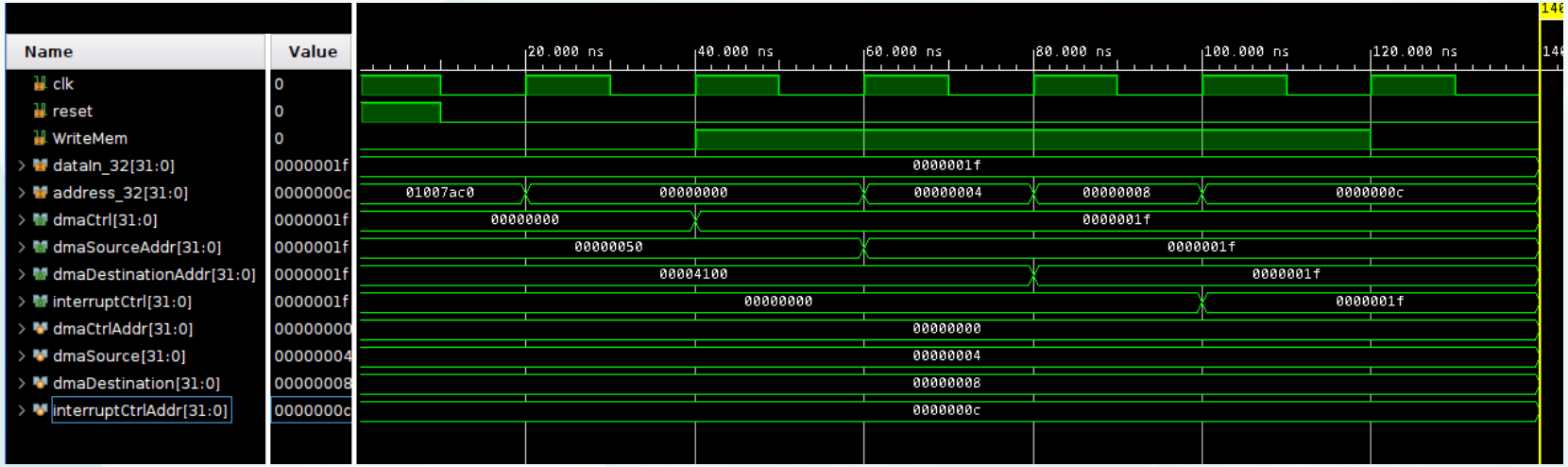

```
module memoryMap (
    input clk,
    input reset,
    input WriteMem,                                 // memory write enable signal
    input [`DATA_SIZE-1:0]dataToMem,                // data to write in memory
    input [`MEM_DEPTH-1:0]MemAddr,                  // memory address
    output reg [`MEM_DEPTH-1:0]dmaSourceAddr,       // dma source address
    output reg [`MEM_DEPTH-1:0]dmaDestinationAddr,  // dma destination address
    output [7:0]dmaCtrl,                            // dma control "register"
    output [3:0]IE                                  // interrupt enable
    );

    reg [`MEM_DEPTH-1:0]dmaCtrlAddr;
    reg [`MEM_DEPTH-1:0]dmaSource;
    reg [`MEM_DEPTH-1:0]dmaDestination;
    reg [`DATA_SIZE-1:0]dmaReg;
    reg [`MEM_DEPTH-1:0]interruptCtrlAddr;
    reg [`DATA_SIZE-1:0]interruptCtrl;

    assign IE = interruptCtrl [3:0];
    assign dmaCtrl = dmaReg[7:0];

    always @(posedge clk) begin
        if (reset) begin
            dmaCtrlAddr <= 32'h00000000;
            dmaSource <= 32'h0000004;
            dmaDestination <= 32'h00000008;
            interruptCtrlAddr <= 32'h0000000c;

            dmaDestinationAddr <= 32'h00004100;
            dmaSourceAddr <= 32'h00000050;
            dmaReg <= 32'h00000000;
            interruptCtrl <= 32'h00000000;
        end
    end

    always @(*) begin
        if (WriteMem) begin
            case (MemAddr)
                dmaCtrlAddr:
                    dmaReg <= dataToMem;
                dmaSource:
                    dmaSourceAddr <= dataToMem;
                dmaDestination:
                    dmaDestinationAddr <= dataToMem;
                interruptCtrlAddr:
                    interruptCtrl <= dataToMem;
                default:
                    ;
            endcase
        end
    end
endmodule
```

ESRG

Embedded Systems Research Group

Test Bench MemoryMap

CENTROALGORITMI

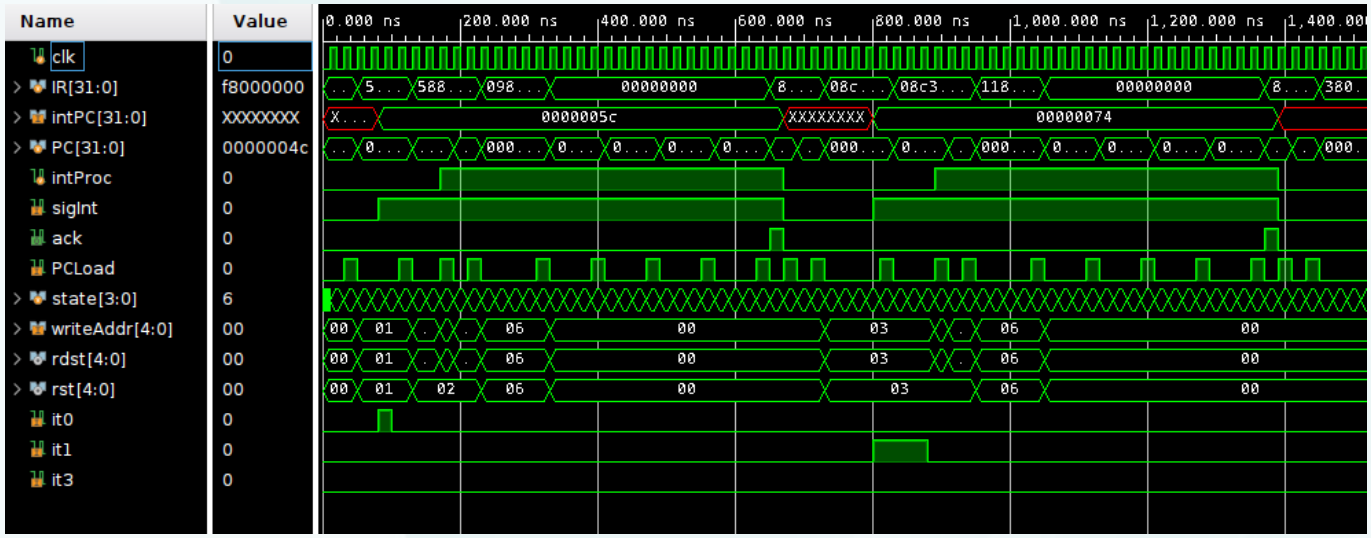


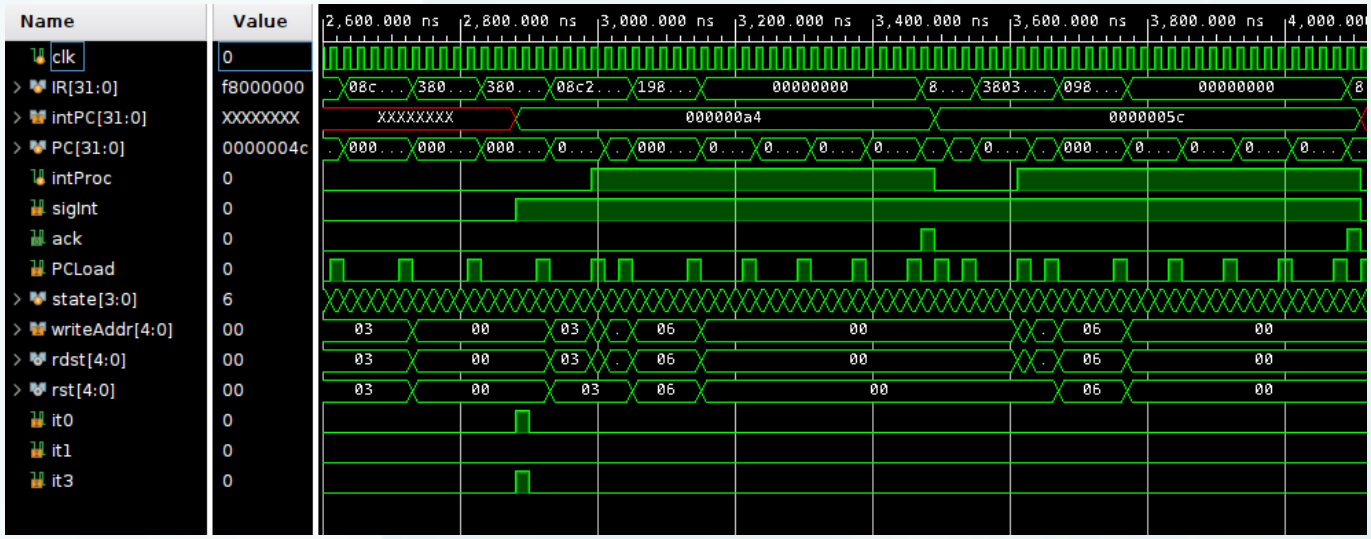
ESRG

Embedded Systems Research Group

Test Bench Interrupt

CENTROALGORITMI



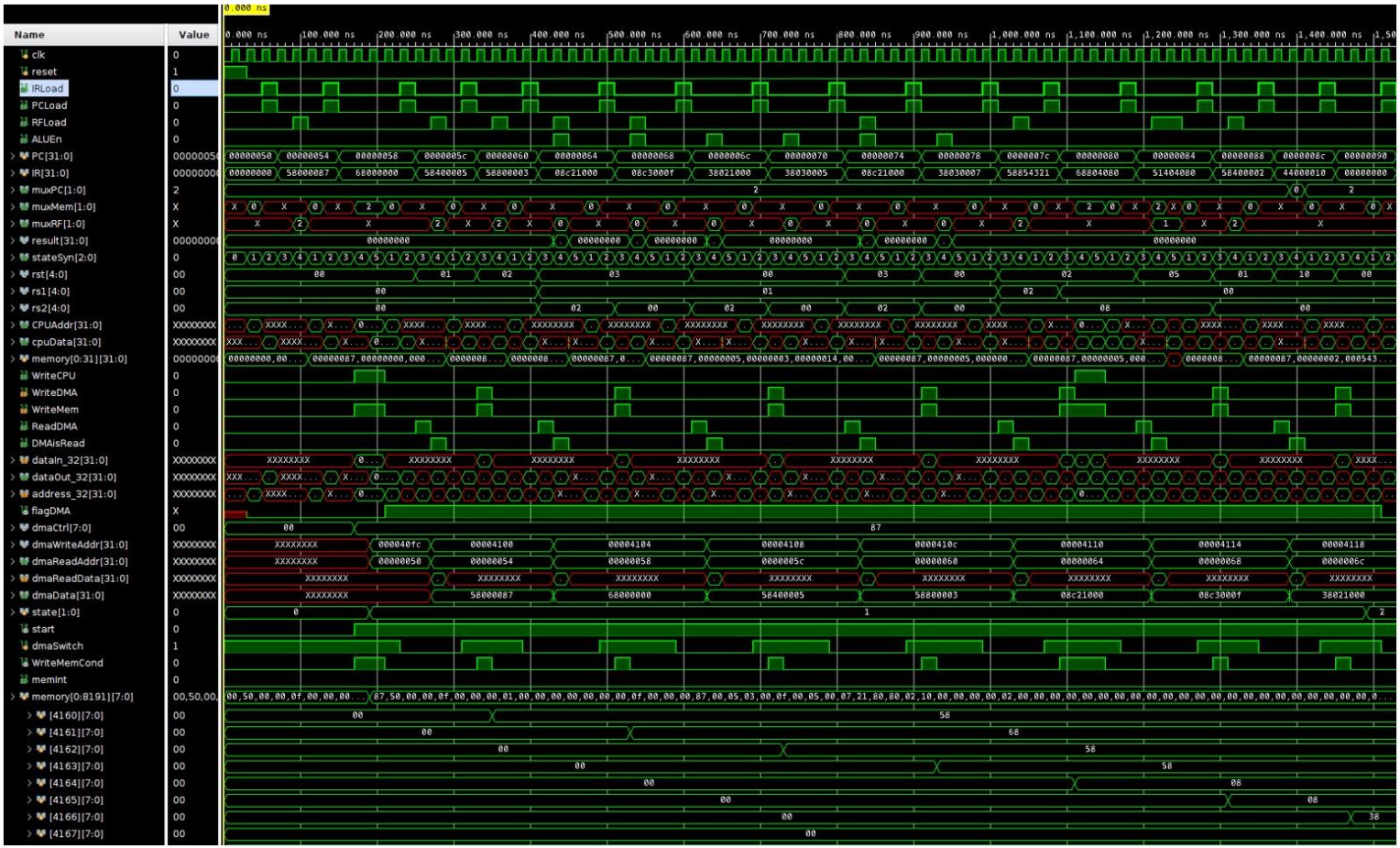


ESRG

Embedded Systems Research Group

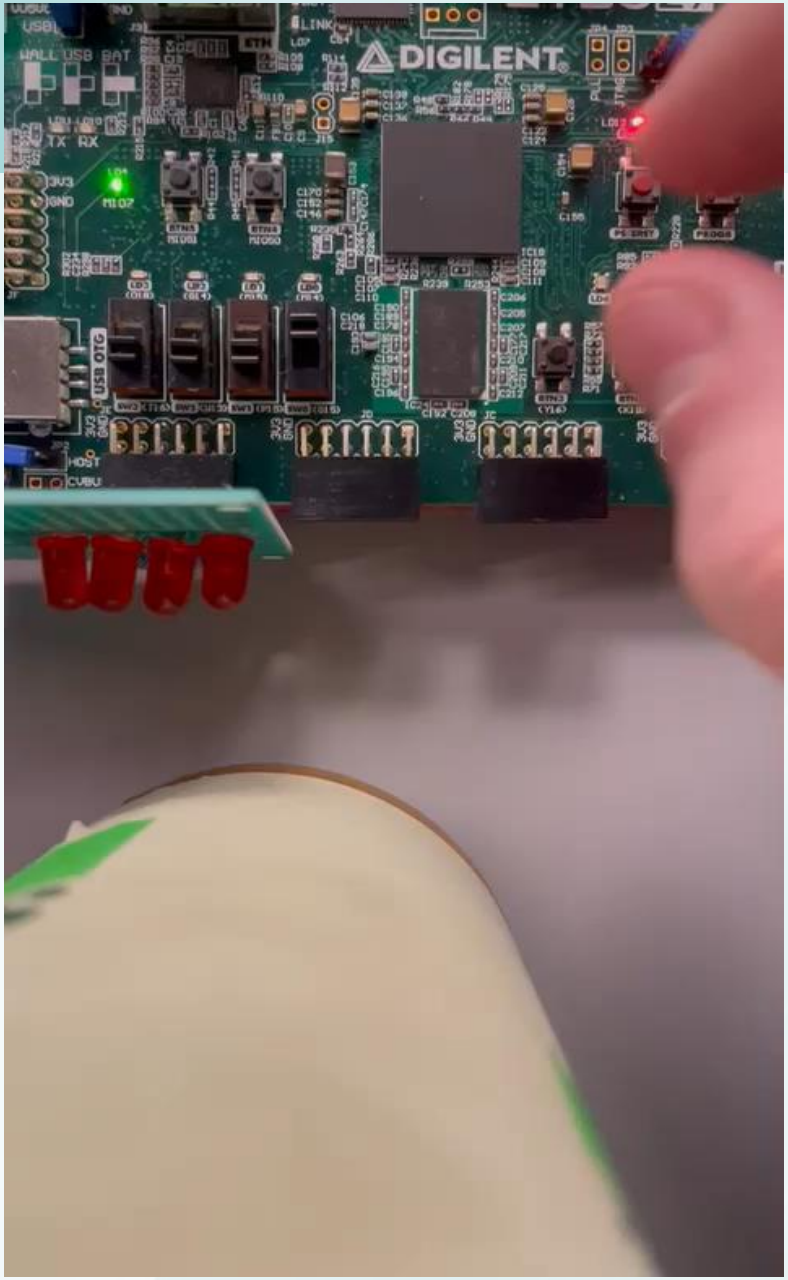
Test Bench DMA

CENTROALGORITMI



ESRG

Embedded Systems Research Group



CENTROALGORITMI