University of Minho

Master's degree in Industrial Electronic Engineering and Computers

Embedded Systems

VeSPA

Authors
João Araújo PG53922
Tiago Sousa PG54261

Professor:
Adriano Tavares

January 2024

Agenda

List of Acronyms

ALU	Arithmetic Logic Unit
BRAM	Block Random Access Memory
CPU	Central Processing Unit
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IE	Interrupt Enable
IR	Instruction Register
ISA	Instruction Set Architecture
LSB	Least Significant Byte
MSB	Most Significant Byte
PC	Program Counter
RISC	Reduced Instruction Set Computing
RTL	Register-Transfer Level
VESPA	Vector Extended Soft Processor Architecture

List of Figures

1 | Problem statement

The VESPA is a microprocessor developed by the University of Toronto in 2008 with the main purpose of encouraging simpler software development over difficult hardware design.[1]

It is based on the RISC-V architecture, which is an ISA with a reduced number of instructions so it's easier to understand and implement. The microprocessor has 32-bit registers including a register file with 32 registers. It is also a 3-address machine, which means it supports instructions with 3 addresses, normally 2 operands and one destination; this allows complex operations to be performed in a single instruction. Its architecture is also little endian, which means the LSB of a 32-bit register is stored at the lowest memory address and the MSB is stored at the highest memory address.

The assignment consists of developing and implementing the ISA of the VESPA, the memory of the microprocessor, an interrupt controller and a DMA in a FPGA.

2 | Analysis

There are some important requirements and constraints that must be fullfilled so the project turns well developed and within the standards.

2.1 Requirements

System requirements are crucial to define clearly which functionalities should be implemented and how each functionality must perform.

2.1.1 Functional Requirements

- The microprocessor must be able to execute whole instruction cycles (fetch, decode, execute)
- The microprocessor must be able to access the memory via register and via immediate
- The microprocessor's memory and register file must be implemented with Vivado IPs
- The microprocessor's memory must be byte addressable and aligned
- The microprocessor must be able to handle interrupt calls
- The microprocessor must include a DMA method

2.1.2 Non Functional Requirements

- The microprocessor must execute each instruction efficiently regarding execution time
- The microprocessor must be scalable to allow future additions or upgrades

2.2 Constraints

System constraints are also crucial to identify the limitations of the hardware to implement and reduce the risk of failure or unexpected behavior.

2.2.1 Technical Constraints

- The microprocessor must be implemented in the FPGA Zybo Z7-10
- The microprocessor must be programmed in Verilog

2.2.2 Non Technical Constraints

- Two member team
- Project deadline at the end of the semester

3 | Design

3.1 System Design

Having the previous problem statement and respective analysis in account, all instructions from the ISA will be implemented, the memory and the register file will be implemented using memory IP, the interrupt controller will have 4 interrupts and the DMA will be implememted with just one mode of operation. The microprocessor to develop will also be byte addressable and follow the Von Neumann architecture.

In addition to the original 16 instructions of the microprocessor's instruction set, one more instruction will be added, the RETI. This modification became necessary since initially VESPA did not have an interrupt controller, however this was incorporated so that the processor could support interrupts (Appendice A).

The CPU will contain the datapath (that incorporates the register file, the ALU and the registers PC and IR), the control unit and the interrupt controller (Appendice C).

Since the memory will be implemented exernally, an overview of the system architecture includes the CPU, the address bus, the data bus, the DMA and the memory (Fig. 3.1).

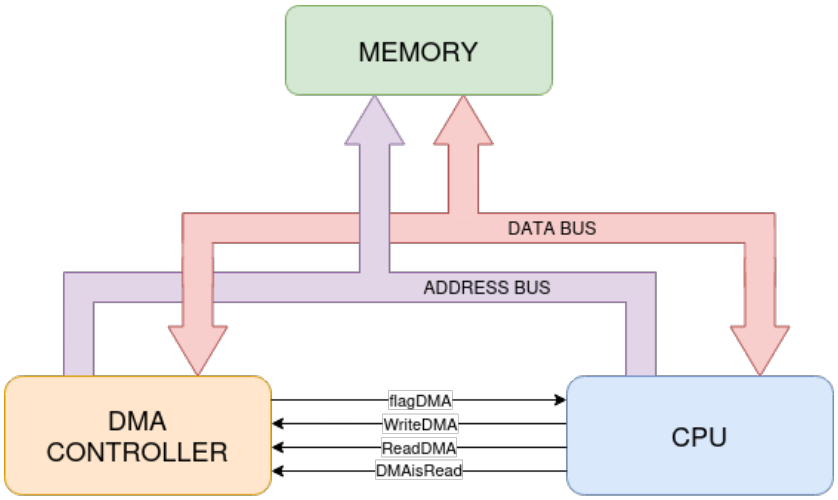


Figure 3.1: System architecture overview.

3.2 Design Software

The Design Software selected to develop and implement the microprocessor in a FPGA is Vivado. Developed by Xilinx, Vivado is a software suite for synthesis and analysis of HDL designs.

Vivado supports RTL design and eletronic system-level synthesis, provides debugging and simulation tools and includes various pre-designed IP cores, such as memory controllers and interfaces.

With all tools provided, design, develop and implement a digital system is easier and optimized.

3.3 Datapath Design

3.3.1 Program Counter

The program counter is a crucial component in the operation of a computer processor. It is a special 32-bit register that stores the address of the current instruction being executed.

As each instruction is executed, the PC is automatically incremented to point to the next instruction in the program sequence. However, during the execution of conditional branches, jumps or interrupts the program counter's value may be modified to redirect execution to a different part of the program. This functionality is essential for implementing control flow structures, such as loops and conditions, within the program code.

The *muxPC* is a 2-bit multiplexer with four inputs that determines the output based on the type of instruction executed, with the result being the current value of the program counter temporarily stored in *nextPC* (Fig. 3.2). The program counter value only changes if the *PCLoad* bit is active, which causes the *nextPC* value to be passed to the program counter register.

Control Signal 0: The output of *muxPC* is determined by the sum of *sext23* and the current value of programm counter, with the control facilitated by selector 0.

Control Signal 1: Configures the *muxPC* output based on the sum of values *sext16* and the *rs1* register , as directed by selector 1.

Control Signal 2: The *muxPC* output corresponds to the current value of *PC* plus 4, guided by selector 2.

Control Signal 3: Directs the *muxPC* output to be the value stored in the interruption program counter register, known as *intPC*, managed by selector 3.

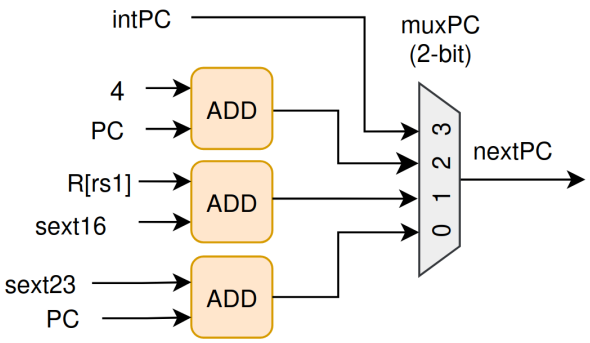


Figure 3.2: muxPC multiplexer design.

3.3.2 Instruction Register

The IR is a essential 32-bit register in the microprocessor's architecture that holds the current instruction being executed. As the PC is incremented each instruction, the IR takes the value of the data in the address stored in the PC using the *IRLoad* flag.

The IR value is a complex register who is divided into smaller values depending on the first 5 bits, the opcode.

3.3.3 Register File

The VESPA register file contains 32 32-bit registers that should be properly stored, and for that reason, the register file will be implemented using the memory IP Block Memory Generator from Vivado.

Since it is a 3-address machine, some instructions will use 3 registers and arithmetic and logical operations are performed asynchronously by the ALU, the register file required at least 3 different ports to be accessed at the same time. Considering that the previously mentioned memory IP is dual port, two blocks will be used. Each block will possess a separate read port and a connected write port so that both blocks contain the same data but 2 registers can be read simultaneously. Both blocks contain a clock input signal and a Write Enable flag; a Read Enable flag won't be necessary because the reading operation will be automatically always enabled (Fig. 3.3).

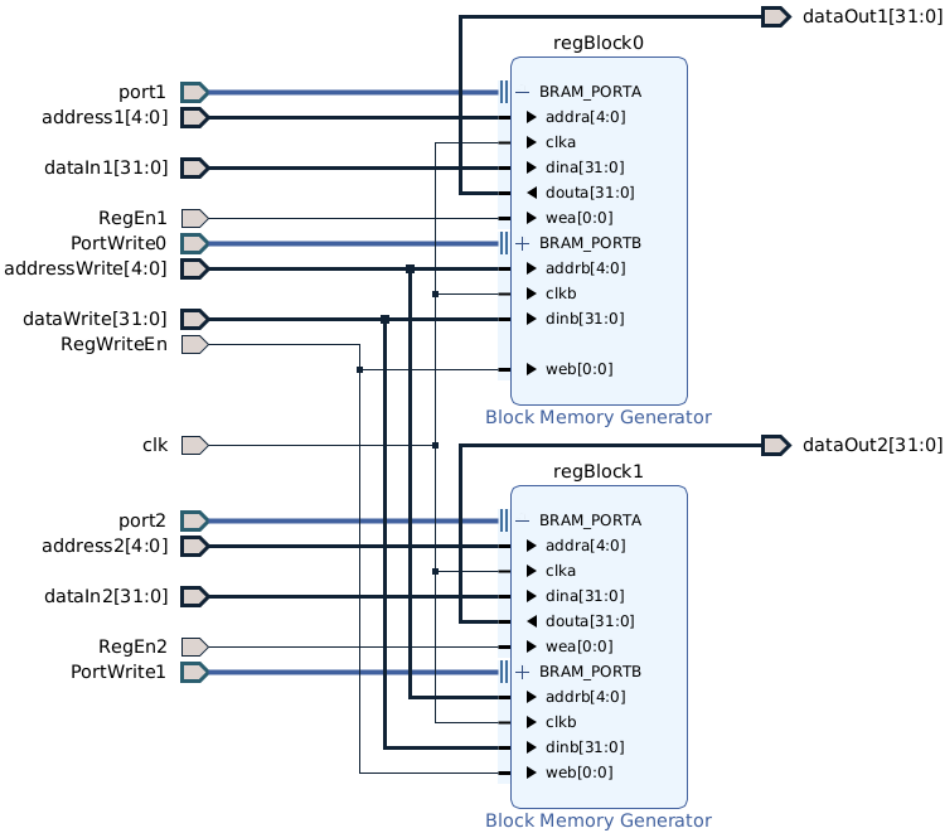


Figure 3.3: Register file design.

The *muxRF* multiplexer is used to select the writing content for the registers in the register file (Fig. 3.4).

Control Signal 0: Directs the output of *muxRF* to assume the value of *result*, capturing the result of an ALU operation.

Control Signal 1: Routes the output to embrace the value of *cpuData*, channeling data derived from memory operations.

Control Signal 2: Shifts the output of the multiplexer to embody the value of *sext22*.

Control Signal 3: Guides the output to take on the value of *PC*.

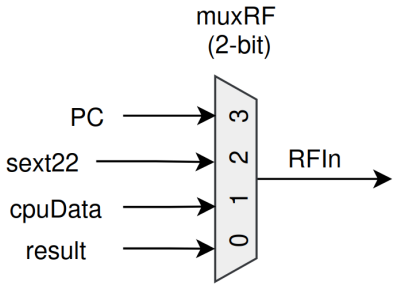


Figure 3.4: muxRF multiplexer.

There is a need to create a multiplexer to select one of the module inputs from the register file, so that data can be read from the *rst* register and written to memory, so the *muxRs* multiplexer will be implemented (Fig. 3.5).

Control Signal 0: In this scenario, the multiplexer output is set to the *rst* value.

Control Signal 1: The multiplexer directs the output to the *rs2* input.

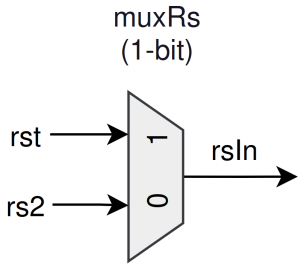


Figure 3.5: muxRs multiplexer.

3.3.4 ALU

The *alu* module is an implementation of an asynchronous ALU, a module responsible for performing arithmetic and logic operations.

The ALU is responsible for carrying out 7 different operations, with instructions being received in the module through the 3-bit ALU specific operation code *opALU* (Fig. 3.6). *IN1* and *IN2* are 32-bit inputs coming from registers or immediate value and both are used for the result of the operation with the exception of the NOT instruction which only needs *IN1*. Also among these instructions, the arithmetic instructions ADD, SUB and CMP can change the values of the flags: zero, carry, negative and overflow.

The control signal *ALUEn* acts as a mechanism for enabling or disabling the ALU's operation as necessary. Upon activation, the *result* assumes the value of the temporary variable *resultAux*, and the *carry* adopts the value of *carryAux*. In this active state, the module produces outputs, including the result of the operation, along with flags.

Inputs:

- ALUEn: Control signal determining whether the ALU should perform operations.
- opAlu: 3-bit input specifying the operation to be executed.
- IN1 and IN2: 32-bit inputs representing the operands for the arithmetic or logic operation.

Outputs:

- Z: Zero flag indicating whether the result is zero.
- N: Sign flag indicating the sign of the result.
- C: Carry flag resulting from addition or subtraction operations.
- V: Overflow flag indicating whether an overflow occurred during the operation.
- result: 32-bit output containing the result of the operation.

OpALU	Operation
001	ADD
010	SUB
011	OR
100	AND
101	NOT
110	XOR
111	CMP

Figure 3.6: opALU decoding.

The *muxALU* was implemented to opt whether the ALU performs calculations using the value from register *rs2* or the immediate value from the instruction (Fig. 3.7).

Control Signal 0: When the control signal at *muxALU* is set to 1, the ALU performs calculations using the values from both registers *rs1* and *rs2*.

Control Signal 1: If the control signal at *muxALU* is set to 0, the ALU conducts operations using the value from register *rs1* and the immediate value *sext16*.

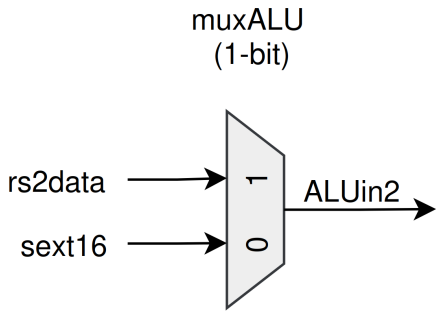


Figure 3.7: muxALU multiplexer.

3.4 Control Unit Design

The Control Unit is a crucial component in the architecture of a microprocessor. It is responsible for receiving instructions and generating control signals based on them, ensuring the proper functioning of the processor.

3.4.1 State Machine

The state machine of this microprocessor has 8 states (Fig. 3.8). The initial state, named *s_start*, is only reached in the first cycle of the state machine after a reset. From this state, the processor advances to the *fetch* state.

In the *s_fetch* state, the processor retrieves the next instruction from memory, where the program counter points during this cycle. However, due to the memory's latency cycle, an additional state (*s_fetch2*), is necessary. In the *s_fetch* state, the processor retrieves the next instruction from memory, where the program counter points during this cycle. If the DMA flag is active in this state, there is a transition in the DMA mode between read or write, depending on the operation previously performed. The *s_fetch* state is used to switch the memory read address, ensuring the instruction's value becomes available in *s_fetch2* for loading. In the *s_fetch2* state, is where the instruction actually loads to the instruction register.

In the *s_decode* state, the microprocessor ensures its proper functioning by identifying the opcode of the instruction obtained during the fetch state. Additionally, it is in the decode state that the necessary flags for DMA operation are activated, enabling the DMA to utilize this state for data transfers.

In the execution state of the processor *s_exec*, previously decoded instructions are effectively executed. In this state, various control signals, such as multiplexers's signals

or flags, are activated based on the instruction being executed to perform the instruction. Unlike the previously mentioned states, at this stage, the processor moves to the *s_halt*, in the case of a HALT instruction. In the case of instructions intended for ALU execution or involving load or store operations, the introduction of an additional state, *s_extra*, becomes imperative. Furthermore, if a pending interrupt is detected, the processor advances to the *s_int* state. If none of the previously mentioned conditions are met, the processor will perform the fetch cycle again, thus repeating the cycle.

In the case of the *s_extra* state, it is required for instructions that involve memory access during the execute phase. Due to the latency cycle of the memory, the extra state becomes necessary to ensure the correct value is loaded in the next fetch.

Regarding the halt state, its next state is itself, creating a loop where the processor remains without executing any actions.

The *s_int* state is reached when an interruption is requested. This state is responsible for updating the PC to the memory address containing the code for the requested interruption. This way, *s_int* serves as the transition point for the processor to handle interruptions, redirecting the program flow to the specified memory location where the corresponding interruption code is located.

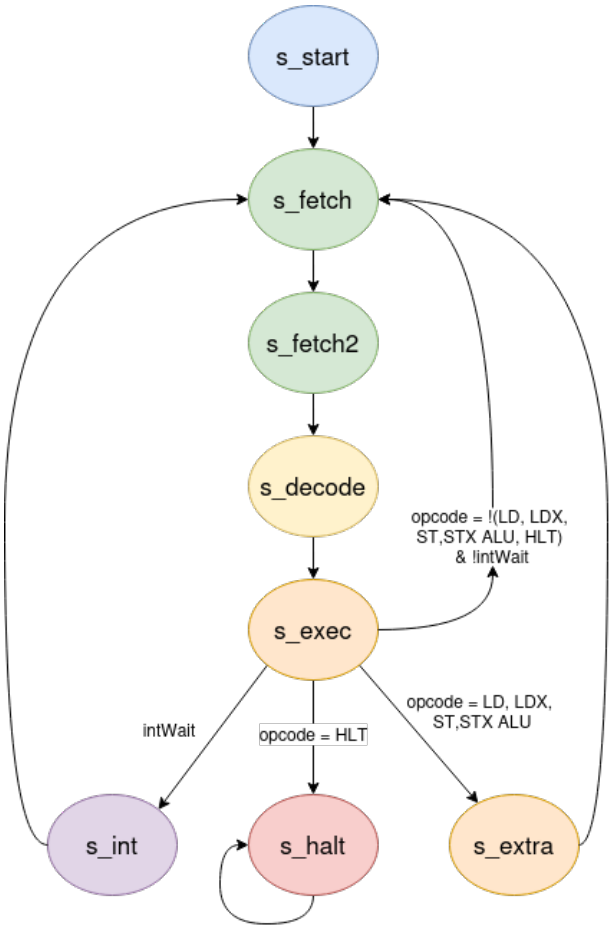


Figure 3.8: State Machine.

3.4.2 Control Signals

Given that the control unit is responsible for maintaining the proper functioning of the processor, it needs to send signals based on the current state and instruction. Thus, there are five control signals for multiplexers, signals for register loading, and various flags.

The following figure 3.9 illustrates the required control signals for each instruction during the execute cycle or extra cycle in the case of memory usage. The figure 3.10 presents the values of the signals generated by the control unit but according to the state, and the execute and extra states do not come into play here as they depend on the opcode.

It is important to note that the *muxAlu* signal is 0 if *bit16* is active and 1 otherwise. Blank spaces in the table indicate cases where the values are not significant (don't cares).

	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			

Figure 3.9: Control Signals (Opcode).

	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			

Figure 3.10: Control Signals (States).

3.4.3 Check Condition

The *checkCondition* module has five inputs and only one output and depending on the condition received in the "*cond*" input, will address whether the "*checkCond*" flag will be active or not. In the "always" condition it will always be 1 and in the "never" condition it will always be 0, in the remaining conditions the value is determined using the Z, C, N and V flags resulting from the last addition, subtraction or comparison instruction. This value from the condition is only valid if the *enBxx* flag is active, as this represents the execution of a branch instruction.

Inputs:

- *enBxx*: Control signal representing the execution of a branch instruction. When active, the module evaluates the condition and produces the corresponding output.
- *cond*: 4-bit input specifying the branching condition based on predefined macro definitions, such as BRA, BNV, BCC and more.
- Z, C, N, V: Flags resulting from the last arithmetic or logic operation, providing information about the outcome.

Output:

- *checkCond*: Output indicating whether the specified condition is met. This signal is active when the condition is satisfied and the *enBxx* control signal is active.

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

Figure 3.11: Branch Conditions.

3.5 Address Bus and Data Bus

Due to the fact that a DMA will be implemented, the memory will be accessed by the CPU and the DMA. In order to coordinate the access to the memory between the CPU and the DMA, an address bus and a data bus will be developed and implemented, so there is no simultaneous attempts to access the memory from both parts.

The address bus is an unidirectional component responsible for selecting the appropriate memory address based on the control signals. The data bus is a bidirectional component responsible for selecting the appropriate data to write in the memory and attributing the data read from the memory to the appropriate output variable (*dmaReadData* or *cpuData*). Generally, alongside these two buses, a control bus is also implemented, but in this specific case the control bus functions will be integrated in the address bus, so the Write flag to the memory will be an output of the address bus aswell.

3.6 Interrupt Controller Design

In computer systems, interrupt controllers are used to manage and prioritize diverse interrupt signals. Interrupts temporarily interrupt the normal flow of the CPU to carry out specific events intended for the interrupt. These events can be input/output operations, errors, or requests from peripheral devices.

The interrupt controller for this microprocessor can manage up to 4 interrupts, each assigned a priority level from 0 (lowest) to 3 (highest).

Depending on the interrupts present in the priority queue, the controller updates the *intPC* register, to later transfer this value to the program counter. At the same time, a signal is sent to the control unit, changing the normal flow to execute the interrupt code.

After it is finished, an acknowledge signal is received to clear the interruption from the queue. Importantly, the interrupt queue is filled with the interrupt only if the corresponding interrupt enable (IE) is active.

In this case, there will be an interrupt generated in case of improper memory access, this being priority 2 and the remaining 3 interrupts are external and buttons will be used to test them.

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

Figure 3.12: Interrupt controller.

3.7 Memory Design

The microprocessor is based on the Von Neumann architecture, so it's not going to be implemented 2 different memories for data and instructions (Fig. 3.13).

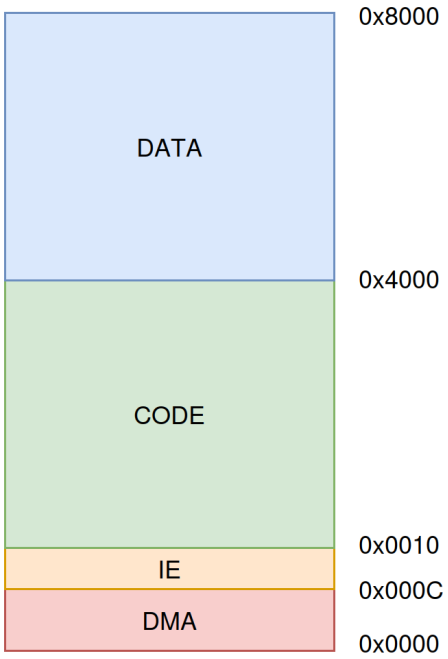


Figure 3.13: Memory design.

All registers and words are 32-bit, but the machine must be byte addressable, so there will be needed 4 input/output ports to access each byte individually. The memory IP Block Memory Generator from Vivado will be used, but since Vivado only provides dual port BRAMs, the memory to develop will be composed of 4 of those blocks 1 byte wide, each one with 1 read port and 1 write port, so they all together make a full 32 bit register (Fig. 3.14. Each block also contains a clock input signal and a Write Enable flag; a Read Enable flag won't be necessary because the reading operation will be automatically always enabled.

So the microprocessor can be byte addressable with 32-bit words, will be implemented a module to wrap the 4 BRAMs and turn them into, as far as the CPU is concerned, a single unified memory.

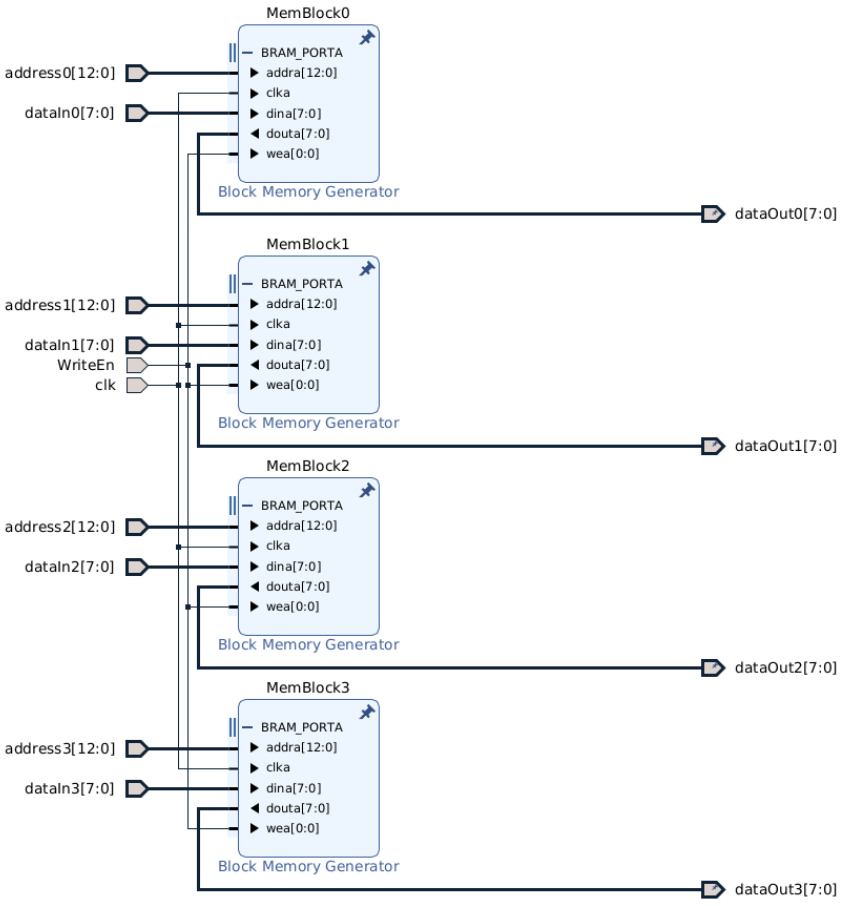


Figure 3.14: Memory design.

The memory size is not a constraint, but the addresses should be 32-bit as specified above, so although the addresses are 32-bit, the memory size will be 32768 bytes (2^{15}) or 8192 32-bit words.

The *muxMem* multiplexer works as a switch with three options, and uses a 2-bit signal to decide which option to choose (Fig. 3.15. This signal controls what comes out as the *CPUAddr*.

Control Signal 0: If the control signal is set to 0, the output will be the PC value.

Control Signal 1: When the control signal is set to 1, the output off the multiplexer will assume the value present of *sext22*.

Control Signal 2: In contrast in case of control signal equals to 1, the output reflects the sum of *sext17* and the content of register *rs1*. par

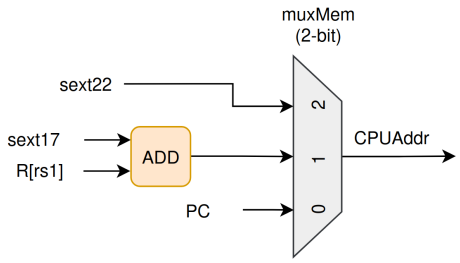


Figure 3.15: muxMem multiplexer.

3.8 DMA Design

A DMA is an important feature of the microprocessor system that allows the CPU and peripherals to transfer data to and from the memory without involving the CPU. Executing "background" data transfers improves the system performance allowing the CPU to focus on other processing tasks while the data is transferred.

Since there are no peripherals implemented, the DMA will just read data from the memory and write it in the same memory.

The DMA to implement will only have one mode of operation, this mode being a "cycle-stealing" mode: the DMA takes control of the address and data bus to access the memory during the CPU cycles that no data transfers occur. Since the whole process of executing an instruction takes at least 5 states/clock cycles, the *s_fetch* and *s_fetch2* states always access the memory (including the latency cycle from the IP) and the *s_exec* state may access the memory, the DMA memory access will always take place in the *s_decode* state, alternating between read and write operations.

The memory address to be read by the DMA, the memory address where the read data will be written, the number of transfers to be executed by the DMA and the DMA enable flag will be mapped in the first 12 bytes of memory so it can be customized using STORE instructions.

There are 4 control signals that will be used by the DMA and the control unit when using the DMA: the *flagDMA* flag will alternate between 1 and 0 each executed instruction so the operation to take place in the *s_decode* state is reading and writting alternately; the *WriteDMA* flag is enabled to request a write operation to the DMA destination address; the *ReadDMA* flag is enabled to request a read operation from the DMA source address; the *DMAisRead* flag is enabled one clock cycle after the *ReadDMA* flag to save the data read from the DMA source address in the register *dmaData*, since the *ReadData* flag can't be enabled in the *s_exec* state without disturb the well functioning off the microprocessor.

4 | Implementation

4.1 Datapath

4.1.1 ALU

```
1  module alu(
2      input ALUEn,
3      input [2:0] opAlu,
4      input [31:0] IN1,
5      input [31:0] IN2,
6      output reg Z,
7      output reg N,
8      output reg C,
9      output reg V,
10     output [31:0] result
11     );
12
13     wire carryAux, carry;
14     wire [31:0] resultAux;
15
16     assign {carryAux, resultAux} =  (`ADD_) ? IN1 + IN2 :
17                                     (`SUB_) ? IN1 - IN2 :
18                                     (`OR_ ) ? IN1 | IN2 :
19                                     (`AND_) ? IN1 & IN2 :
20                                     (`NOT_) ? ~IN1 :
21                                     (`XOR_) ? IN1 ^ IN2 :
22                                     (`CMP_) ? IN1 - IN2 : 0;
23
24     assign {carry, result} = (ALUEn) ? {carryAux, resultAux} : 0;
25     assign sigSubb = (`SUB_) ? 1'b1 : 1'b0;
26
27     always @(*) begin
28         if (ALUEn && (`ADD_||`SUB_||`CMP_)) begin
29             Z <= ~(|result[`WIDTH-1:0]);
30             C <= carry;
31             N <= result[`WIDTH-1];
32             V <= (result[`WIDTH-1] & ~IN1[`WIDTH-1] & ~(sigSubb ^
   ↪ IN2[`WIDTH-1])) |
33                  (~result[`WIDTH-1] & IN1[`WIDTH-1] & (sigSubb ^
   ↪ IN2[`WIDTH-1]));
34         end
35     end
36 endmodule
```

4.2 Control Unit

4.2.1 State Machine

```
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 begin
                    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;
```

```
51              default:
52                  state <= s_start;
53          endcase
54
55          if (sigInt) begin
56              if(intProc == 0 && intWait == 0) begin
57                  intWait <= 1;
58              end
59          end
60      end
61  end
```

4.2.2 Control Signals

```
1   assign muxRs = ((`ST || `STX) && (state == s_exec || state == s_extra)) ?
    ↪  1'd1 : 1'd0;
2
3   assign muxALU = (bit16 && (`ALU) && (state == s_exec)) ? 1'b0 : 1'b1;
4
5   assign muxPC = (`Bxx && (state == s_exec)) ? 2'd0 :
6                  ((`JMP||`JMPL||`RETI) && (state == s_exec)) ? 2'd1 :
7                  (state == s_fetch2) ? 2'd2 :
8                  (state == s_int) ? 2'd3 : 2'b10;
9
10  assign muxMem = ((`LD || `ST) && (state == s_exec) || (`ST && state ==
    ↪  s_extra)) ? 2'd2 :
11                  ((`LDX || `STX) && (state == s_exec) || (`STX && state ==
    ↪  s_extra)) ? 2'd1 :
12                  (state == s_fetch) ? 2'd0 : 2'bx;
13
14  assign muxRF = ((`ALU) && (state == s_exec)) ? 2'd0 :
15                 ((`LDX || `LD) && ((state == s_exec) || state == s_extra
    ↪  )) ? 2'd1 :
16                 ((`LDI) && (state == s_exec)) ? 2'd2 :
17                 ((`JMPL) && (state == s_exec)) || (state == s_int) ? 2'd3
    ↪  : 2'bx;
18
19  assign ALUEn  =   (`ALU && (state == s_exec))  ? 1'd1 : 1'd0;
20  assign IRLoad =  (state == s_fetch2) ? 1'd1 : 1'd0;
21  assign PCLoad =  (((`Bxx && checkCond) ||`JMP ||`JMPL||`RETI) && (state
    ↪  == s_exec)) || state == s_fetch2 || (state == s_int)) ? 1'd1 : 1'd0;
22  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;
23  assign WriteCPU = ((`ST || `STX) && (state == s_exec || state ==
    ↪  s_extra)) ? 1'd1 : 1'd0;
24
25  assign ReadDMA = (flagDMA && (state == s_decode) && dmaSwitch == 0) ?
    ↪  1'd1 : 1'd0;
26  assign DMAisRead = (flagDMA && state == s_exec && dmaSwitch == 0) ? 1'd1
    ↪  : 1'd0;
27  assign WriteDMA = (flagDMA && state == s_decode && dmaSwitch == 1) ? 1'd1
    ↪  : 1'd0;
```

```
28
29 assign enBxx = (opcode == 8) ? 1'b1 : 1'b0;
30 assign opAlu = (`ALU && (state == s_exec)) ? opcode[2:0] : 0;
31 assign stateSyn = state;
32
33 assign ack = (intProc && `RETI) ? 1'd1 : 1'd0;
```

4.2.3 Check Condition

```
1  module checkCondition (
2          input enBxx,
3          input [3:0] cond,
4          input Z,
5          input C,
6          input N,
7          input V,
8          output checkCond
9          );
10
11     assign checkCondAux =  (cond == `BRA) ? 1'd1 :
12                            (cond == `BNV) ? 1'd0 :
13                            (cond == `BCC) ? ~C   :
14                            (cond == `BCS) ? C    :
15                            (cond == `BVC) ? ~V   :
16                            (cond == `BVS) ? V    :
17                            (cond == `BEQ) ? Z    :
18                            (cond == `BNE) ? ~Z   :
19                            (cond == `BGE) ?  (N & V) | (~N & ~V)  :
20                            (cond == `BLT) ? ~(N & V) | (~N & ~V)  :
21                            (cond == `BGT) ? ~Z & ((N & V) | (~N & ~V)) :
22                            (cond == `BLE) ?  Z | ((N & V) | (~N & ~V)) :
23                            (cond == `BPL) ? ~N   :
24                            (cond == `BMI) ?  N   :  1'b0;
25     assign checkCond = enBxx ? checkCondAux : 1'b0;
26 endmodule
```

4.3 Address Bus

```
1  `define MEM_DEPTH 32
2  `define DATA_SIZE 32
3
4  module addressBus(
5      input WriteCPU,
6      input WriteDMA,
7      input ReadDMA,
8      input [`MEM_DEPTH-1:0] CPUAddr,
9      input [`MEM_DEPTH-1:0] dmaWriteAddr,
10     input [`MEM_DEPTH-1:0] dmaReadAddr,
11     output WriteMem,
12     output [`MEM_DEPTH-1:0] address_32
13     );
```

```
14
15      assign address_32 = (WriteCPU == 1) ? CPUAddr       :
16                          (WriteDMA == 1) ? dmaWriteAddr  :
17                          (ReadDMA == 1)  ? dmaReadAddr   : CPUAddr;
18      assign WriteMem = ((WriteCPU != WriteDMA) && ReadDMA == 0) ? 1 : 0;
19  endmodule
```

4.4 Data Bus

```
1   `define MEM_DEPTH 32
2   `define DATA_SIZE 32
3
4   module dataBus(
5       input WriteCPU,
6       input WriteDMA,
7       input DMAisRead,
8       input [`DATA_SIZE-1:0] rs2data,
9       input [`DATA_SIZE-1:0] dmaData,
10      input [`DATA_SIZE-1:0] dataFromMem,
11
12      output [`DATA_SIZE-1:0] cpuData,
13      output [`DATA_SIZE-1:0] dmaReadData,
14      output [`DATA_SIZE-1:0] dataToMem
15      );
16
17      assign dataToMem =  (WriteCPU == 1)  ? rs2data   :
18                          (WriteDMA == 1) ? dmaData   : 32'bx;
19      assign dmaReadData = (DMAisRead == 1) ? dataFromMem : 32'bx;
20      assign cpuData = (DMAisRead == 0)  ? dataFromMem :32'bx;
21  endmodule
```

4.5 Memory

```
1   `define MEM_WIDTH 32
2   `define MEM_DEPTH 32
3   `define IP_WIDTH 8
4   `define REAL_MEM_SIZE 15
5   `define REAL_MEM_DEPTH 13
6
7   module memoryWrapper(
8       input clk,
9       input WriteMem,
10      input [`MEM_WIDTH-1:0]dataIn_32,
11      input [`MEM_DEPTH-1:0]address_32,
12
13      output [`MEM_WIDTH-1:0]dataOut_32,
14      output memInt
15      );
16
17      wire [`REAL_MEM_SIZE-1:0]address_15 = address_32[`REAL_MEM_SIZE-1:0];
18      wire [`REAL_MEM_DEPTH-1:0] address0;
```

```
19      wire [`REAL_MEM_DEPTH-1:0] address1;
20      wire [`REAL_MEM_DEPTH-1:0] address2;
21      wire [`REAL_MEM_DEPTH-1:0] address3;
22
23      wire [`MEM_WIDTH-1:0] dataIn0;
24      wire [`MEM_WIDTH-1:0] dataIn1;
25      wire [`MEM_WIDTH-1:0] dataIn2;
26      wire [`MEM_WIDTH-1:0] dataIn3;
27      wire [`IP_WIDTH-1:0] dataOut0;
28      wire [`IP_WIDTH-1:0] dataOut1;
29      wire [`IP_WIDTH-1:0] dataOut2;
30      wire [`IP_WIDTH-1:0] dataOut3;
31
32      wire [`MEM_WIDTH-1:0] dataInput;
33      wire WriteMemCond;
34      reg bit0;
35      reg bit1;
36
37      assign WriteMemCond = ((address_32[`REAL_MEM_SIZE-1] & WriteMem &
    ↪  ~|address_32[`MEM_DEPTH-1:`REAL_MEM_SIZE]) |
    ↪  (~|address_32[`MEM_WIDTH-1:4] & WriteMem));
38      assign memInt = (~WriteMemCond & WriteMem);
39
40      always @(*) begin
41          if (address_15[0] == 1'b1 || address_15[0] == 1'b0) begin
42              bit0 <= address_15[0];
43              bit1 <= address_15[1];
44          end
45      end
46
47      assign address0 = (bit1 | bit0) ? address_15[`REAL_MEM_SIZE-1:2]+1 :
    ↪  address_15[`REAL_MEM_SIZE-1:2];
48      assign address1 = (bit1)        ? address_15[`REAL_MEM_SIZE-1:2]+1 :
    ↪  address_15[`REAL_MEM_SIZE-1:2];
49      assign address2 = (bit1 & bit0) ? address_15[`REAL_MEM_SIZE-1:2]+1 :
    ↪  address_15[`REAL_MEM_SIZE-1:2];
50      assign address3 = address_15[`REAL_MEM_SIZE-1:2];
51
52      assign dataInput = (~bit1 & ~bit0)  ? {dataIn_32[31:24],
    ↪  dataIn_32[23:16], dataIn_32[15:8], dataIn_32[7:0]} :
53                         (~bit1 & bit0)   ? {dataIn_32[23:16],
    ↪  dataIn_32[15:8], dataIn_32[7:0], dataIn_32[31:24]} :
54                         (bit1 & ~bit0)   ? {dataIn_32[15:8],
    ↪  dataIn_32[7:0], dataIn_32[31:24], dataIn_32[23:16]} :
55                         (bit1 & bit0)    ? {dataIn_32[7:0],
    ↪  dataIn_32[31:24], dataIn_32[23:16], dataIn_32[15:8]} : 32'bx;
56
57      assign dataIn0 [`MEM_WIDTH-1:0] = dataInput[31:24];
58      assign dataIn1 [`MEM_WIDTH-1:0] = dataInput[23:16];
59      assign dataIn2 [`MEM_WIDTH-1:0] = dataInput[15:8];
60      assign dataIn3 [`MEM_WIDTH-1:0] = dataInput[7:0];
61
62      assign dataOut_32 = (~bit1 & ~bit0)? {dataOut0, dataOut1, dataOut2,
    ↪  dataOut3} :
63                          (~bit1 & bit0) ? {dataOut1, dataOut2, dataOut3,
    ↪  dataOut0} :
```

```
64                          (bit1 & ~bit0) ? {dataOut2, dataOut3, dataOut0,
    ↪  dataOut1} :
65                          (bit1 & bit0)  ? {dataOut3, dataOut0, dataOut1,
    ↪  dataOut0} : 32'bx;
66
67      memory_ip memIP (
68      .WriteEn(WriteMemCond),
69      .address0(address0),
70      .address1(address1),
71      .address2(address2),
72      .address3(address3),
73      .clk(clk),
74      .dataIn0(dataIn0),
75      .dataIn1(dataIn1),
76      .dataIn2(dataIn2),
77      .dataIn3(dataIn3),
78      .dataOut0(dataOut0),
79      .dataOut1(dataOut1),
80      .dataOut2(dataOut2),
81      .dataOut3(dataOut3)
82      );
83  endmodule
```

4.6 Interrupt Controller

```
1   `define DATA_SIZE 32
2   `define MEM_DEPTH 32
3
4   module interruptController(
5       input clk,
6       input reset,
7       input [3:0] IE,
8       input it0,
9       input it1,
10      input it2,
11      input it3,
12      input ack,
13
14      output reg sigInt,
15      output reg [`MEM_DEPTH-1:0] intPC
16  );
17      reg [3:0] priorityQueue;
18      reg [1:0] priority;
19
20      always @(posedge clk) begin
21          if (reset) begin
22              priorityQueue <= 4'b0000;
23          end else if (ack) begin
24              priorityQueue[priority] <= 1'b0;
25          end
26      end
27
28      always @* begin
```

```
29          if (priorityQueue != 4'b0000) begin
30              if (priorityQueue[3]) begin
31                  intPC <=  32'h00000010;
32                  priority <= 3;
33              end else if (priorityQueue[2]) begin
34                  intPC <=  32'h00000020;
35                  priority <= 2;
36              end else if (priorityQueue[1]) begin
37                  intPC <=  32'h00000030;
38                  priority <= 1;
39              end else if (priorityQueue[0]) begin
40                  intPC <=  32'h00000040;
41                  priority <= 0;
42              end else begin
43                  intPC <=  32'bx;
44              end
45              sigInt <= 1'b1;
46          end else begin
47              sigInt <= 1'b0;
48              intPC <= 32'bx;
49              priority <= 1'bx;
50          end
51          priorityQueue <= priorityQueue | {it3 & IE[3], it2 & IE[2], it1 &
    ↪  IE[1], it0 & IE[0]};
52      end
53  endmodule
```

4.7 DMA

```
1   `define DATA_SIZE 32
2   `define MEM_DEPTH 32
3   `define S_IDLE 0
4   `define S_TRANSFER 1
5   `define S_END 2
6
7   module dmaController(
8       input clk,
9       input reset,
10      input ReadDMA,
11      input DMAisRead,
12      input WriteDMA,
13      input [7:0] dmaCtrl,
14      input [`MEM_DEPTH-1:0] dmaDestinationAddr,
15      input [`MEM_DEPTH-1:0] dmaSourceAddr,
16      input [`DATA_SIZE-1:0] dmaReadData,
17      output reg [`MEM_DEPTH-1:0] dmaWriteAddr,
18      output reg [`MEM_DEPTH-1:0] dmaReadAddr,
19      output reg [`DATA_SIZE-1:0] dmaData,
20      output reg flagDMA
21      );
22
23      reg [7:0] counter;
24      reg [1:0] state;
```

```
25      wire start = dmaCtrl[7];
26      wire [6:0]transferLength = dmaCtrl[6:0];
27
28      always @(*) begin
29          if (flagDMA && DMAisRead) begin
30              dmaData <= dmaReadData;
31          end
32      end
33
34      always @(posedge clk) begin
35          if (reset) begin
36              state <= `S_IDLE;
37          end else begin
38              case(state)
39                  `S_IDLE: begin
40                      flagDMA <=  2'b0;
41                      if (start == 1) begin
42                          state <= `S_TRANSFER;
43                          counter <= 8'h00;
44                          dmaWriteAddr <= dmaDestinationAddr - 4;
45                          dmaReadAddr <= dmaSourceAddr;
46                      end
47                  end
48                  `S_TRANSFER: begin
49                      if (counter < transferLength) begin
50                          flagDMA <= 1;
51                          if (ReadDMA) begin
52                              dmaWriteAddr = dmaWriteAddr + 4;
53                              dmaReadAddr = dmaReadAddr + 4;
54                          end
55                          if (WriteDMA) begin
56                              counter <= counter + 1;
57                          end
58                      end else begin
59                          state <= `S_END;
60                      end
61                  end
62                  `S_END: begin
63                      state <= `S_IDLE;
64                      flagDMA <= 0;
65                      counter <= 0;
66                  end
67              endcase
68          end
69      end
70  endmodule
```

5 | Validation

5.1 Datapath

5.1.1 ALU

As the ALU is responsible for arithmetic and logical operations, there was a need to perform tests on it, as can be seen below in the figure 5.1. The tests involved verifying the result and the operation flags, according to the instruction received in *opALU* and according to the *ALUEn* flag.

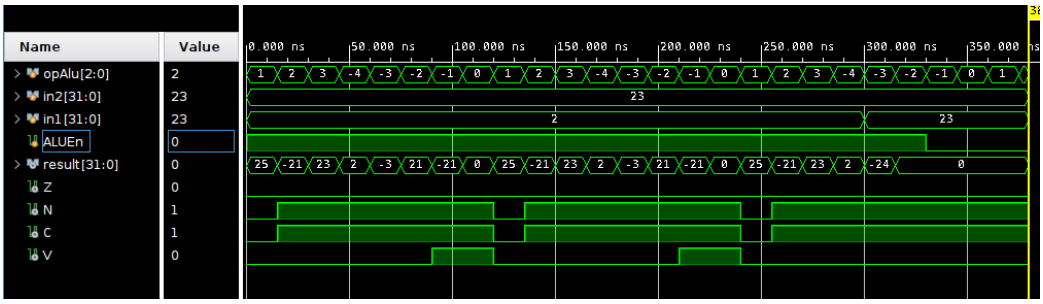


Figure 5.1: ALU testbench.

5.1.2 Register File

In the figure 5.2 are the tests for the register file that consisted of checking whether the two reading outputs corresponded to the values of the input addresses,

Writing to a register was also tested and it is possible to observe that it is only written when the WriteEn signal is active and that, as expected, its value takes one cycle to be changed due to the latency of the IP used.

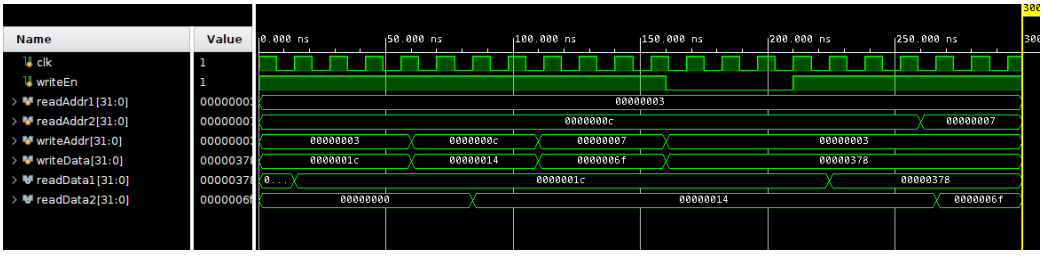


Figure 5.2: Register File testbench.

5.2 Control Unit

5.2.1 Check Condition

To test the *checkCondition* module, several conditions and flags were introduced into the simulation in order to verify the value of the checkCond flag (Fig. 5.3). This flag is only activated if *enBxx* is active and determines whether or not a branch will occur.

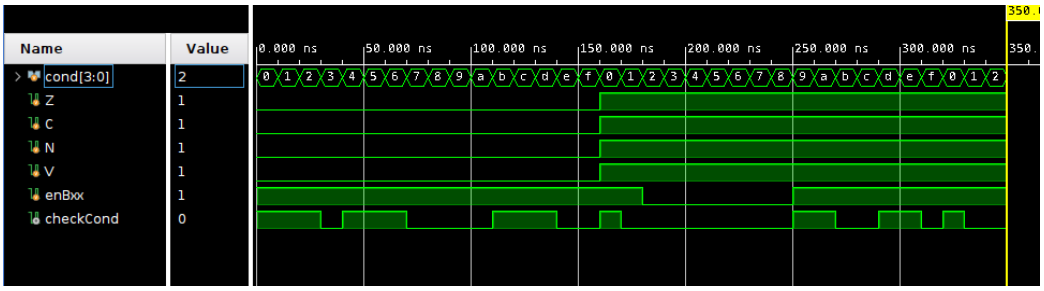


Figure 5.3: CheckCond testbench.

5.3 Memory

In order to test memory, memory reading and memory writing tests were performed (Fig. 5.4). It can be observed that the memory has one latency cycle, and when the write operation to the address 0x4006 occurs, only one cycle later the value is read correctly. Also, if there is an attempt to write to an address that does not belong to the data memory, this will enable the *memInt* flag and launch an interrupt routine.

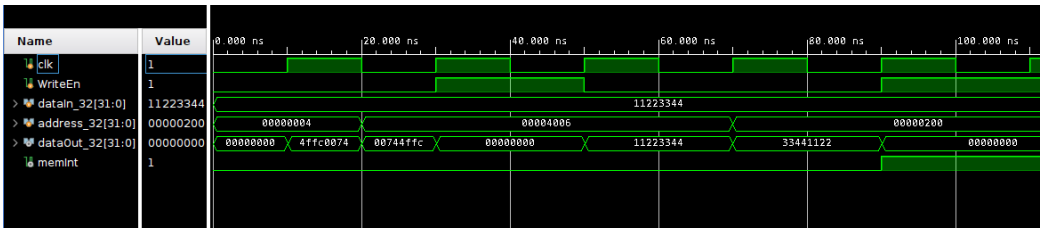


Figure 5.4: Memory testbench.

5.4 Memory Map

To make sure the DMA settings can be customized and the interrupts can be enabled by the user, some registers must be mapped in the memory. A simple way to map them is to verify the write operations to the memory and inform the respective module if the current memory address corresponds to one of a mapped register. If it does, the value to be written in the memory is also assigned to a specific variable.

As can be seen in the figure 5.5, when the *WriteMem* is enabled and the *address_32* address is equal to one of the last 4 variables in the figure, the value of *dataIn_32* is assigned to the respective register.

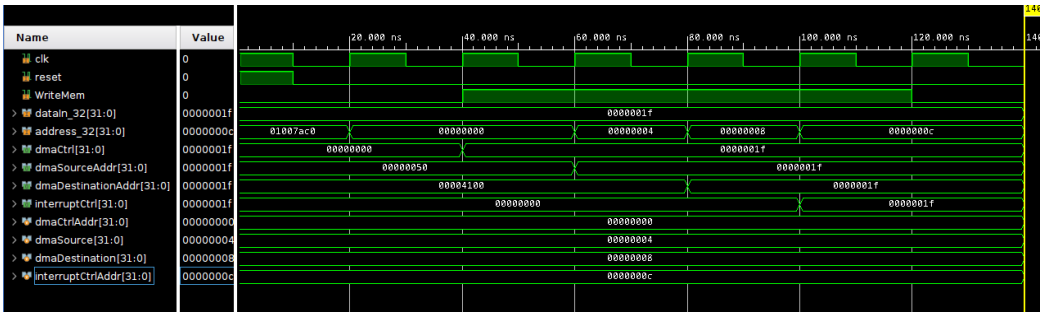


Figure 5.5: Memory map testbench.

5.5 Interrupt Controller

In the interrupt controller testbench, it is possible to observe that the results are in line with expectations (Fig. 5.6).

Tests were carried out with 4 interrupts, receiving interrupts at the same time, priority levels, only activation with IE, *intPC* register according to the interrupt, the correct use of the *sigInt* signal in the event of an interrupt and reception of an *ack* signal in order to clear the row.

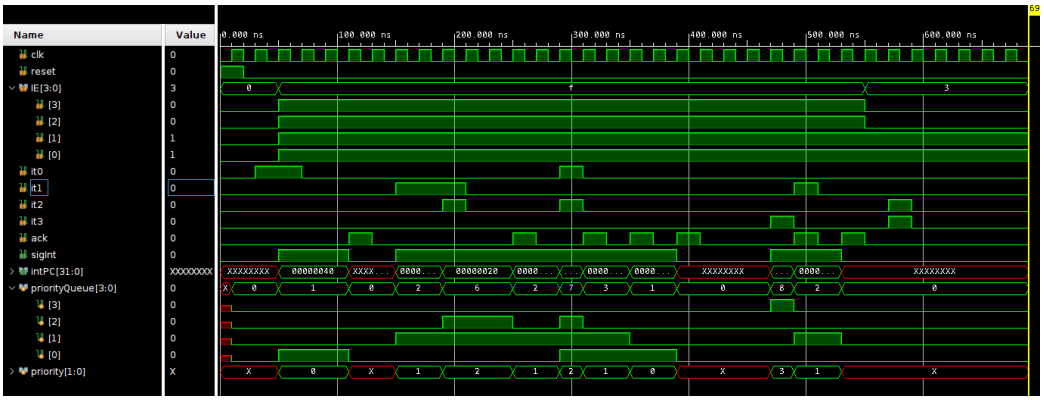


Figure 5.6: Interrupt controller testbench.

5.6 Address Bus

In order to test the address bus, a simulation was carried out with 3 input registers and 3 control flags (Fig. 5.7). Cosidering that the *WriteCPU*, *WriteDMA* and *ReadDMA* control flags can never be active at the same time, the simulation confirms the correct value in the *address_32* and *WriteMem* outputs.

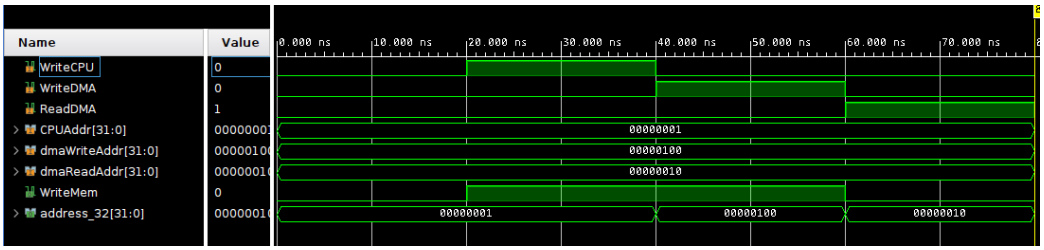


Figure 5.7: Address bus testbench.

5.7 Data Bus

To test the data bus, a simple simulation was executed to test if the correct data was assigned to the correct output. Considering that none of the control flags can be active simultaneously, the simulation results were positive: when the *WriteCPU* flag is set, the data from *rs2data* is assigned to *dataToMem* and when the *WriteDMA* flag is set, the data from *dmaData* is assigned to *dataToMem*; when the *DMAisRead* flag is set, the data from *dataFromMem* is assigned to *dmaReadData* and when the same flag is disabled, the data from *dataFromMem* is assigned to *cpuData* since the CPU read operation is automatic.

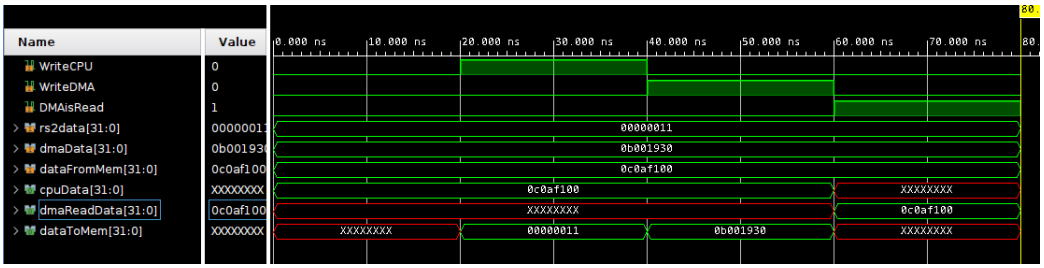


Figure 5.8: Data bus testbench.

5.8 DMA

For the purpose of testing the DMA, a testbench was done simulating the control signals, the addresses and the read data so the output could be analyzed and verified (Fig. 5.9). A full cycle of one read and one write operations from the DMA implies that the *ReadDMA*, *DMAisRead* and *WriteDMA* flags are enabled one after the other, respectively, one clock cycle each. After setting the transferlength equal to 3 and start the DMA, with 3 of the previously mentioned cycles, *dmaData* received 3 different values and after that the *state* went from *S_TRANSFER* to *S_END*, successfully terminating the DMA operations once the 3 data values were transmitted.

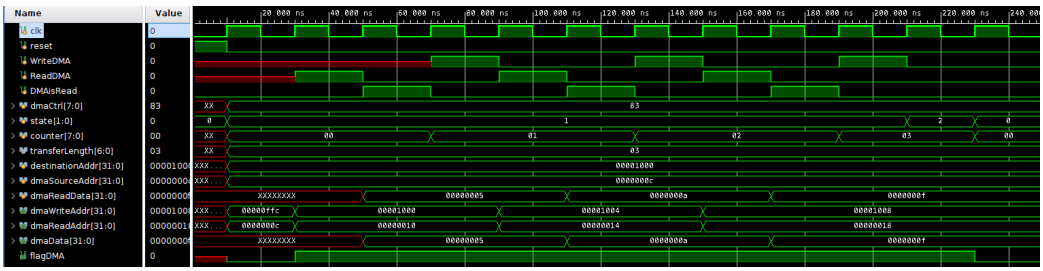


Figure 5.9: DMA controller testbench.

5.9 Final Simulations

After putting together all the above tested modules, it was necessary to test them together and see if everything was operational. Therefore, the following simulation, Fig. 5.10, presents a test composed of several instructions in which they are interrupted by two interrupt signals, so that correct operation can be seen by executing the interrupt code.

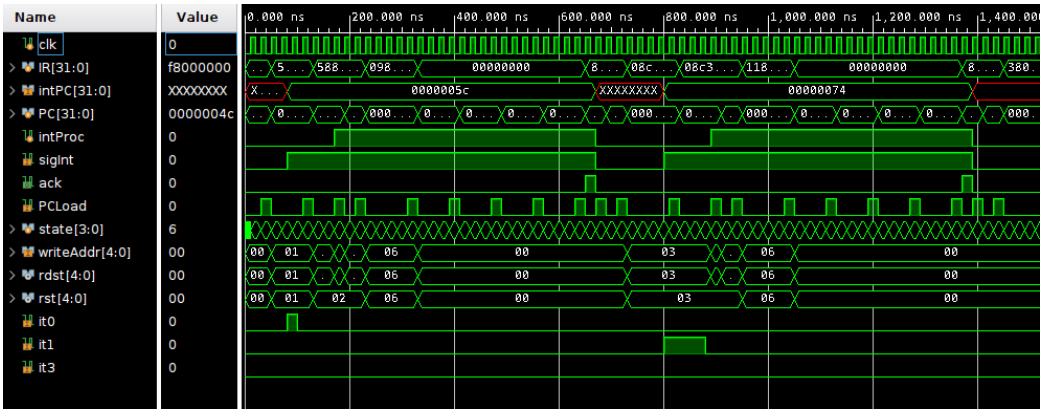


Figure 5.10: Interrupt testbench.

In Figure 5.11, two interrupt signals were simultaneously dispatched to test priority levels. Consequently, the interrupt with the highest priority is executed first, followed by the execution of the pending code for the other interrupt.

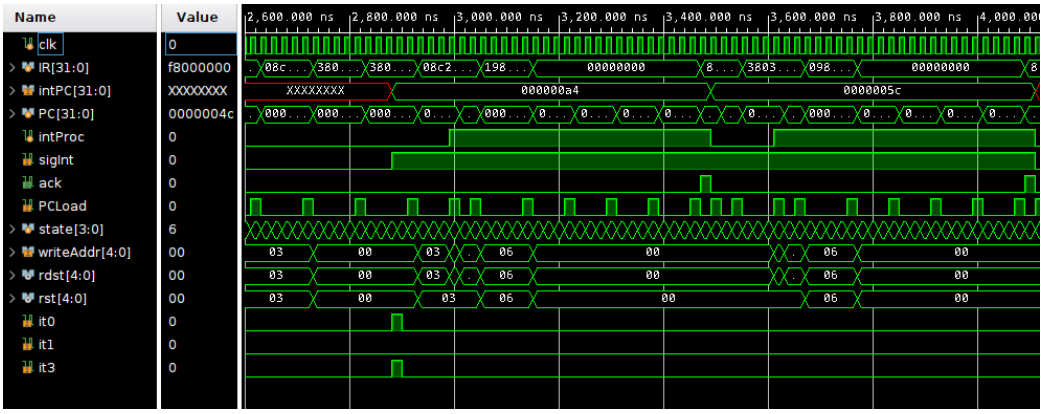


Figure 5.11: Interrupt testbench.

In order to test the correct functioning of the DMA during the execution of other instructions by the CPU, simulations were carried out with the DMA active and making transfers between different memory positions using the source and destination addresses destined for the DMA.

The simulation in Figure 5.12 allows us to see the memory positions destined for writing by the DMA being filled with the values previously read by the DMA and this without interrupting the processor instructions.

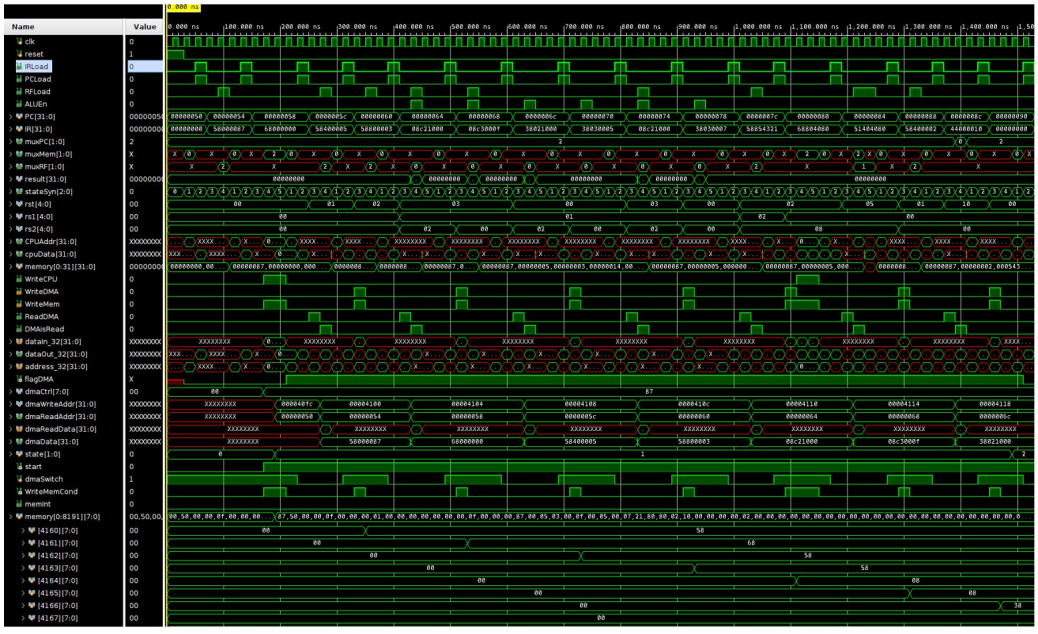


Figure 5.12: DMA testbench.

Bibliography

[1] https://www.eecg.toronto.edu/VESPA/

[2] https://en.wikipedia.org/wiki/Vivado

[3] https://www.geeksforgeeks.org/direct-memory-access-dma-controller-in-computer

[4] https://zipcpu.com/zipcpu/2019/04/02/icontrol.html

[5] Lilja, D. & Sapatnekar, S. (2005). Designing Digital Computer Systems with Verilog. Cambridge University Press.

[6] Vincent P. & Harry F. (1997). Computer Systems Design and Architecture . Pearson.

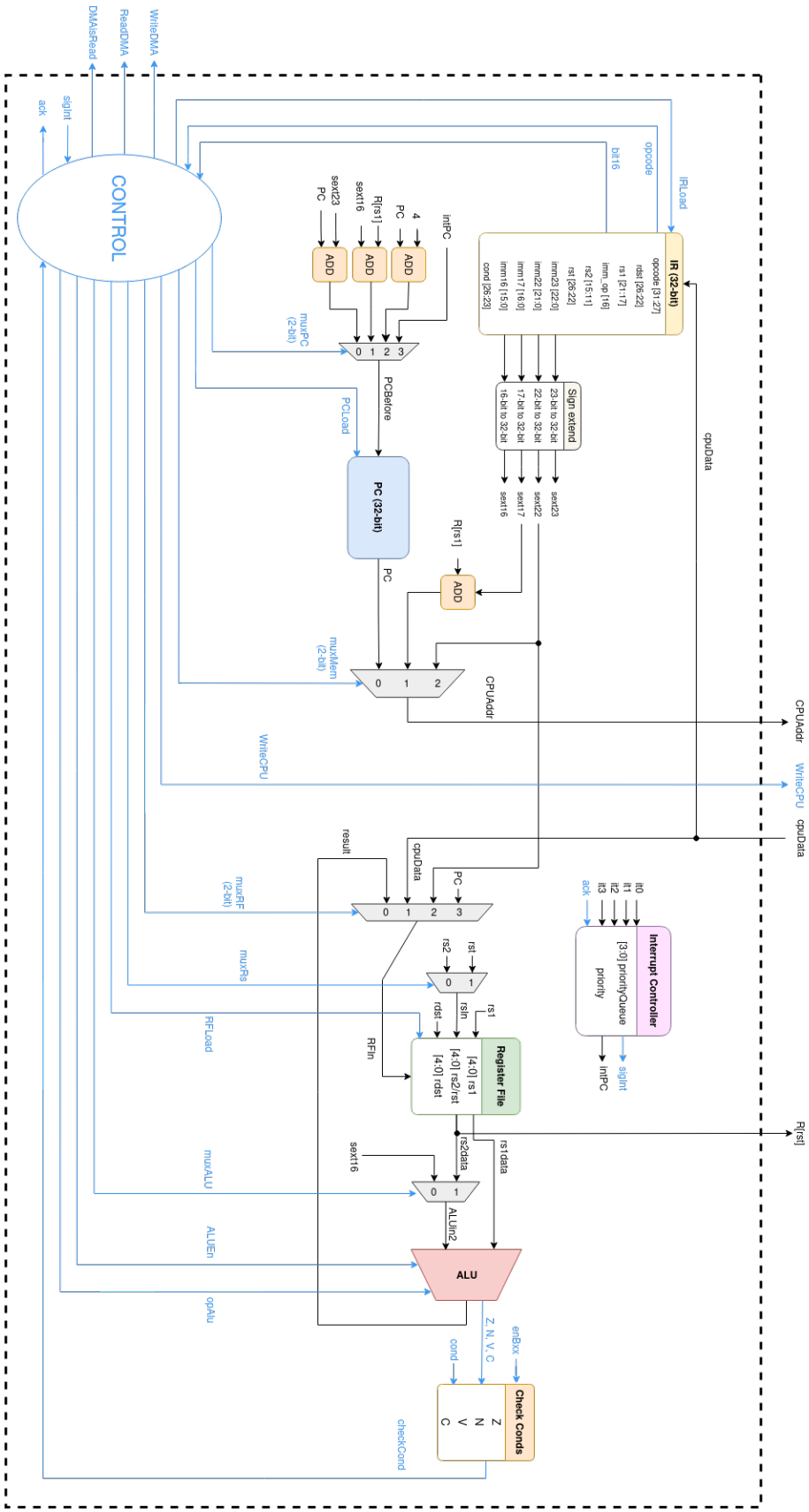
A | Instruction Set

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
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

B | Instruction Format

	31…27	26…23	22	21…17	16	15…11	10…0
NOP	opcode	0…0					
ADD, SUB, OR, AND, XOR, CMP	opcode	rdst		rs1	rs2		0…0
ADD, SUB, OR, AND, XOR, CMP	opcode	rdst		rs1	immed16		
NOT	opcode	rdst		rs1	0…0		
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	rdst		immed22			
STX	opcode	rst		rs1	immed17		
HLT	opcode	0…0					
RETI	opcode	0…0		11111	0…0		

C | CPU Design


DMAisRead
ReadDMA
WriteDMA
sigInt
ack
CONTROL
opcode
bit16
IRLoad
IR (32-bit)
opcode [31:27]
rdst [26:22]
rs1 [21:17]
imm_op [16]
rs2 [15:11]
rst [26:22]
imm23 [22:0]
imm22 [21:0]
imm17 [16:0]
imm16 [15:0]
cond [26:23]
cpuData
sext23
PC
sext16
R[rs1]
PC
4
ADD
ADD
ADD
intPC
muxPC (2-bit)
0 1 2 3
PCBefore
PCLoad
PC (32-bit)
Sign extend
23-bit to 32-bit
22-bit to 32-bit
17-bit to 32-bit
16-bit to 32-bit
sext23
sext22
sext17
sext16
R[rs1]
ADD
PC
muxMem (2-bit)
0 1 2
CPUAddr
WriteCPU
CPUAddr
WriteCPU
cpuData
result
cpuData
PC
muxRF (2-bit)
0 1 2 3
ack
int0
int1
int2
int3
Interrupt Controller
[3:0] priorityQueue
priority
intPC
sigInt
muxRs
rs2
rst
0 1
RFLoad
rdst
rsIn
rs1
RFIn
[4:0] rs2/rst
[4:0] rdst
[4:0] rs1
Register File
R[rs]
rs2data
rs1data
sext16
muxALU
0 1
ALUIn2
ALUEn
ALU
opAlu
Z, N, V, C
enBxx
cond
Z
N
V
C
Check Conds
checkCond

D | RTL Design