University of Minho

Master's of Industrial Electronics and Computers Engineering

Processadores Dedicados

RISC-V Processor

Group 3

Diogo Fernandes PG47150
Duarte Rodrigues PG47158
João Miranda PG47332
José Abreu PG47386

Supervised by:
Prof. Dr. Tiago Gomes
Prof. Dr. Rui Machado

June 2022

Contents

Glossary

ISA	Instruction Set Architecture
FPGA	Field-Programmable Gate Array
BRAM	Block Random Access Memory
PC	Program Counter
ALU	Arithmetic/Logical Unit
RISC	Reduced Instruction Set Computing
RAW	Read After Write
DMEM	Data Memory
IMEM	Instruction Memory
LED	Light Emiting Diode
HDL	Hardware Description Language

List of Figures

List of Tables

Listings

1 | Introduction

In 2010, there was the development of a new Instruction Set Architecture, ISA[1], called RISC[2]-V by the the University of California, Berkeley. The purpose of RISC-V is to create an extensible and open source instruction set, not only for academic use but also for commercial products. In the future, the goal is to support multiple implementations, giving companies, universities or research centers the freedom to develop their own hardware solutions, knowing that this software layer exists, and can be used without restrictions.

This work was developed under the premise of creating the means for developing a processor first in single-cycle and then using the pipeline, according to the RISC-V instruction set. Other objectives of this work is the development of a RISC-V processor to be implemented in an FPGA[3] and and add peripherals that interact with the processor, namely BRAM[4].

Analyzing the RISC-V document [1] [2] , it is possible to verify that the RV32 uses 32-bit wide registers (x0-x31) to store integer values, where x0 is a read-only register holding the value zero. In addition, it still contains the source registers `rs1` and `rs2`, as well as the destination register, `rd`. In the base ISA, there are 6 formats of 32-bit long instructions: `R`, `I`, `S`, `B`, `U` and `J`. The figure 1.1 summarises the 6 instruction formats and 37 instructions that are implementation goals:

- **21 Integer Computational Instructions:** Arithmetic (addition, subtraction, and bitwise shifts), logical (bitwise Boolean operations) and comparison (arithmetic magnitude comparisons) instructions.
- **8 Load and Store Instructions:** Load byte (signed and unsigned), Load half-word (signed and unsigned), Load Word, Store byte, Store half-word and Store word.
- **8 Control Transfer Instructions:** Branch instructions compare the values in two registers and assume the conditional branch range, given by the 12-bit signed immediate. There are still two more JUMP statements: `JAL` and `JALR`. `JAL` changes the Program Counter, PC[5] to anywhere in a range of instructions. `JALR` uses `rs1` and a 12-bit signed immediate to do an indirect jump to the address in `rs1` plus the immediate.

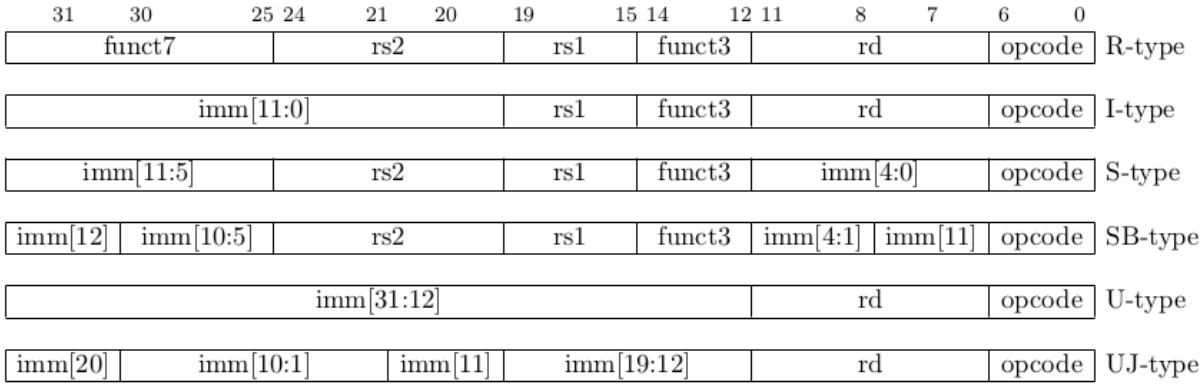

Figure 1.1: The RISC-V base instruction formats.

[1] Instruction Set Architecture
[2] Reduced Instruction Set Computing
[3] Field-Programmable Gate Array
[4] Block Random Access Memory
[5] Program Counter

2 | Single Cycle

The single-cycle processor executes instructions in a single cycle. It was already given to us a single-cycle processor which executed some instructions such as addi, or, and, beq, slt, sub, lw, sw and jal instructions. The next step was to modify the datapath and control unit for the remaining RV32I instructions [3].

2.1 Datapath

Based on single-cycle processor developed on the book [3], the logic necessary for the execution of instructions that were not yet supported was added.The single cycle diagram processor is shown in the figure 2.3.

2.1.1 JAL and JALR

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. JAL stores the address of the instruction following the jump (pc+4) into register `rd`.

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register `rs1`, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register `rd`.

As the `rd` you have to save the sum of each of the instructions, the solution was to add a MUX that, if the selection is 0, selects PC + SignExt($imm_{20:1}$, 1'b0)[JAL] and if it is 1, rs1 + SignExt (imm)[JARL]. The result the MUX enters in the selection of PC MUX. This requires a 1-bit control signal called PCResultSrc.

2.1.2 LUI

LUI (load upper immediate) places the 20-bit U-immediate into bits 31–12 of register `rd` and places zero in the lowest 12 bits.

The EXTEND block extends depending on the instruction type. Since this is a U-type instruction, the extend is done with the immediate bits placed in the most significant bits and the least ones are set to 0. To save in the register `rd`, ImmExt was connected to the output MUX at input 011.

2.1.3 AUIPC

AUIPC (add upper immediate to pc) use the U-type format. AUIPC appends 12 low-order zero bits to the 20-bit U-immediate, adds it to the PC and then places the result in register `rd`.

For JAL and JALR instructions, PCResult, if PCResultSrc was 1, would take the value of rs1 + SignExt (imm). Therefore, by connecting PCResult to the output MUX and selecting input 100, it is possible to save it in the register.

2.1.4 Load Instructions

The LW instruction loads a 32-bit value from memory into `rd`. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in `rd`. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in `rd`. LB and LBU are defined analogously for 8-bit values.In the case of LB and LH operations, misaligned access must be taken into account. For this purpose, the LoadDec block was created.The figure 2.1 summarizes the strategy used for the LOAD instructions. The 32-bit MemData comes out of the memory block. To make the difference between the word, half or byte instructions, two MUX were used in which the selection is made by the least significant 2-bits of the address, a. For example, if the address is 01, in the case of LB it means taking the second byte only, but in the case of LH it takes

the second and third bytes.These instructions require the EXTEND block. So sign or zero extend are chosen based on funct3. The unsigned_w output takes the value with the selection of bit 0 of funct3. In the other MUX, the two most significant bits allow you to decide between the byte, half or word. Finally, it is possible to extract the desired ReadData based on the most significant bit of funct3.

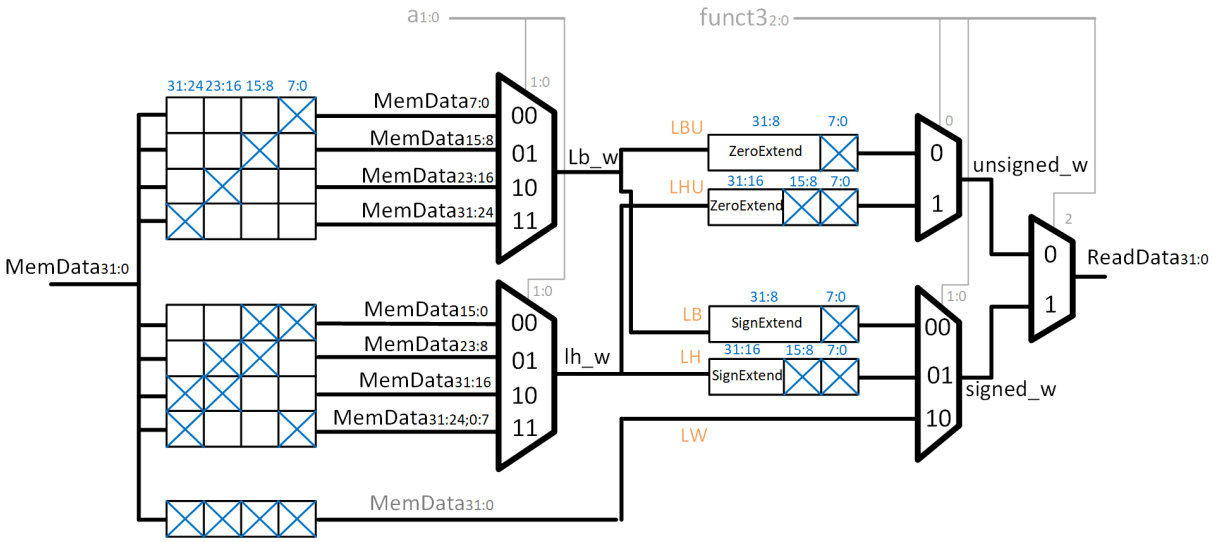

Figure 2.1: LoadDec Block.

2.1.5 Store Instructions

The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register `rs2` to memory. It is important to mention that in the SH and SB operations, the bytes that were in the positions where they are not to be written must be kept. The figure 2.2 summarizes the strategy used for the STORE instructions. In case the write enable is set to 1, the writing will depend on the type of operation, dictated by funct3 and the position, address (a).

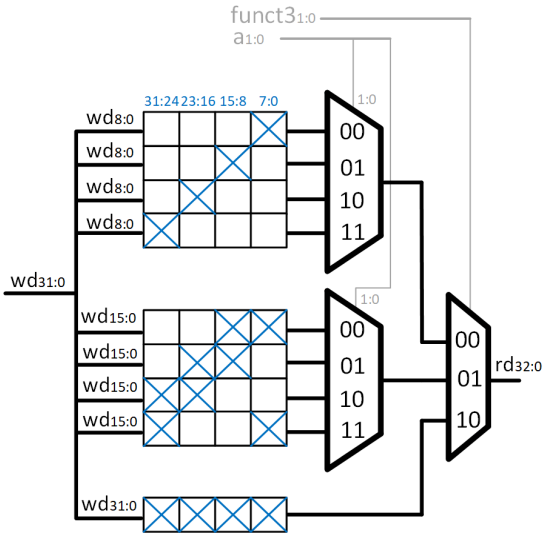

Figure 2.2: Data Memory Block.

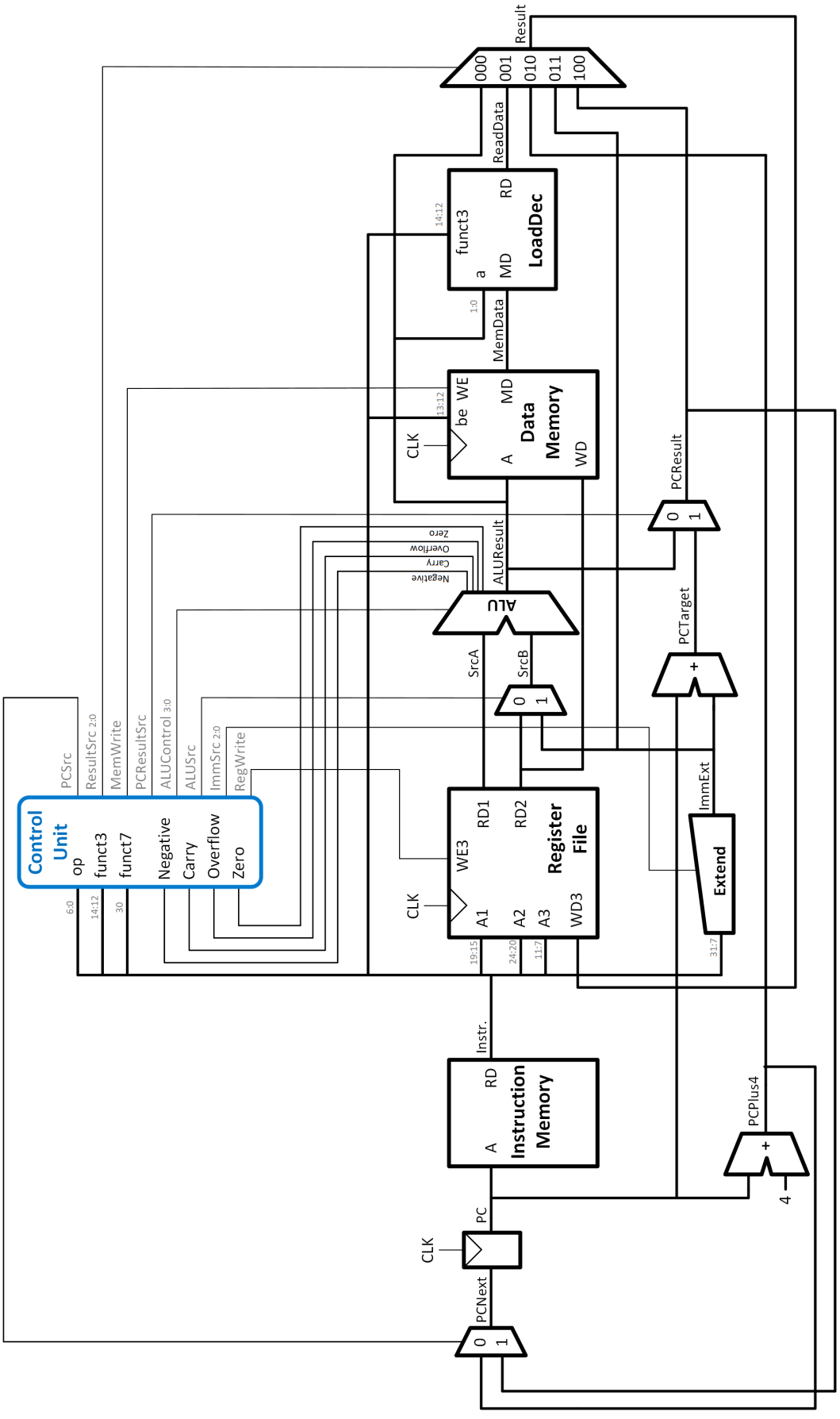

Figure 2.3: Single Cycle Processor.

2.1.6 ALU

An Arithmetic/Logical Unit, ALU[1] combines a variety of mathematical and logical operations into a single unit.

The ALU, figure 2.4 receives a 4-bit control signal `ALUControl` that specifies which function to perform. In addition, it has `A` and `B` inputs with 32-bit each.

The ALU contains an N-bit adder and N 2-input AND, OR and XOR gates. It also contains inverters and a multiplexer to invert input B when $ALUControl_0$ is asserted. `SR`, `SRA` and `SL` operations are performed using the operators >>, >>> and <<, respectively. A 10:1 multiplexer chooses the desired function based on `ALUControl`.

To perform `SLT` and `SLTU` ALU must produce extra outputs, called flags. As shown in the schematic, the output flags is composed of the **N**, **Z**, **C**, and **V** flags that indicate, respectively, that the ALU output, ALUResult, is negative or zero or that the adder produced a carry out or overflowed. the **N** (Negative) flag is connected to the most significant bit of the ALU output, $ALUResult_{31}$. The **Z** (Zero) flag is asserted when all of the bits of ALUResult are 0, as detected by the N-bit NOR gate in. The **C** (Carry out) flag is asserted when the adder produces a carry out and the ALU is performing addition or subtraction (indicated by $ALUControl_1$ = 0). **V**, (oVerflow) is asserted when the ALU is performing addition or subtraction, `A` and `Sum` have opposite signs and when the addition of two same signed numbers produces a result with the opposite sign. With this, for the operation SLT, when `A` < `B`, `ALUResult` = 1, otherwise, `ALUResult` = 0. Therefore, When Sum_{N1} = 1, the result of `A` − `B` is negative, and `A` is less than `B`, but when overflow occurs, `Sum` will have the incorrect sign. So, we XOR the sign bit of Sum with **V**, the overflow signal, to correctly indicate a negative `Sum`. To determine the SLTU operation, we compute A − B and check whether the result is negative. As the numbers are unsigned the result is negative if there is no carry out. For both previous instructions, the rest of the bits are extended to 0.

More specifically, if ALUControl = 0000, the output multiplexer chooses A + B. If ALUControl = 0001, the ALU computes A − B. If ALUControl = 0010, the ALU computes A AND B. If ALUControl = 0011, the ALU performs A OR B. If ALUControl = 0100 or 1101, the output multiplexer chooses the operation SLT or SLTU, respectively. If ALUControl = 0110, the ALU performs A XOR B. If ALUControl = 0111, 1000 or 1001, the ALU performs the operation SL, SR or SRA, respectively.

[1] Arithmetic/Logical Unit

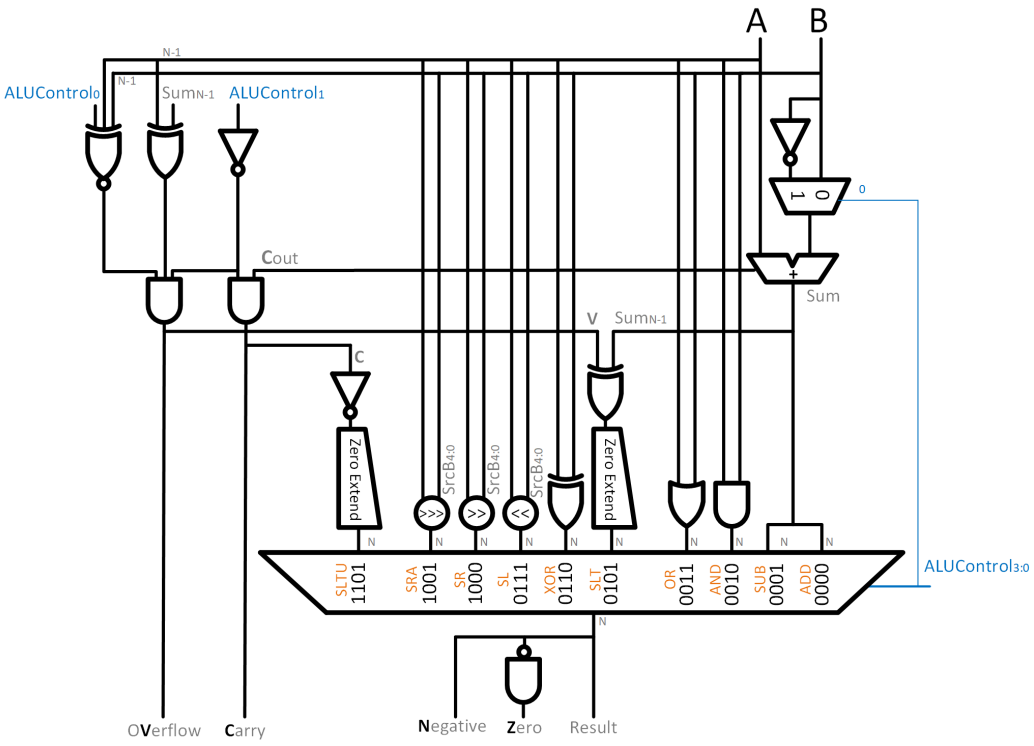

Figure 2.4: ALU.

2.2 Control Unit

There are 6 instructions that require jumping to another instruction. Therefore, it was necessary to create a new block, called JumpDec, to make the decision and assign the value to the control signal PCSrc.

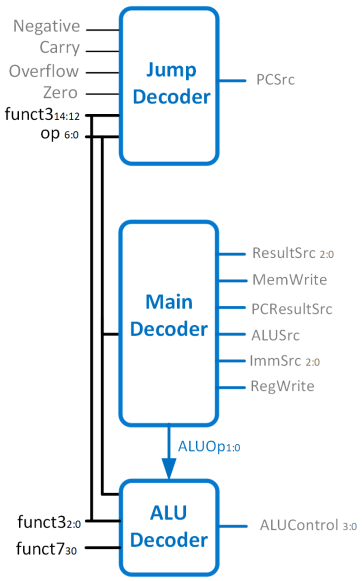

Figure 2.5: Control Unit.

2.2.1 Jump Decoder

The management and decision in this block to branch instructions is based on the flags generated by the ALU (**N**,**C**,**V** and **Z**), the instruction opcode and funct3. The table 2.1 presents the combination of flags for each case, whether signed or unsigned.

BEQ take the branch if registers rs1 and rs2 are equal. Therefore, subtracting the two registers and it is zero, it jumps. It is then possible to use the **Z** flag.

BNE take the branch if registers rs1 and rs2 are unequal. Therefore, subtracting the two registers and it is different from zero, it jumps. It is then possible to use the **Z** flag, but denied.

BLT take the branch if rs1 is less than rs2, using signed comparison. Therefore, making use of the table 2.1 and the signed column, it is possible to see that in this instruction, the comparison can be made by XOR between **N** and **V**.

BLTU take the branch if rs1 is less than rs2, using and unsigned comparison. Therefore, making use of the table 2.1 and the unsigned column, it is possible to see that in this instruction, the comparison can be made by **C** denied.

BGE take the branch if rs1 is greater than or equal to rs2, using signed. Therefore, making use of the table 2.1 and the signed column, it is possible to see that in this instruction, the comparison can be made by XOR between **N** and **V** denied.

BGEU take the branch if rs1 is greater than or equal to rs2, using unsigned comparison. Therefore, making use of the table 2.1 and the unsigned column, it is possible to see that in this instruction, the comparison can be made by **C**.

Table 2.1: Signed and unsigned comparisons.

Comparison	Signed	Unsigned
$=$	Z	Z
$\neq$	$\overline{Z}$	$\overline{Z}$
$<$	$N \oplus V$	$\overline{C}$
$\leq$	$Z + (N \oplus V)$	$Z + \overline{C}$
$>$	$\overline{Z} \bullet \overline{(N \oplus V)}$	$\overline{Z} \bullet C$
$\geq$	$\overline{(N \oplus V)}$	C

2.3 Simulations

2.3.1 Load Instructions

The listing 2.1 present the code corresponding to the waveforms in figures 2.6, 2.7, 2.8 and 2.9. As explained in section 2.1.4 it is possible to load a single byte from each position of the word and to load two bytes from an initial position of the word. In case the initial position is the last byte, the processor loads from the memory that byte and the first byte of the word.

```
1 lui x1, 0xaa0bc
2 addi x1, x1, 0xdd
3 sw x1, 96(x0)    # x1 = 0xaa0bc0dd
4
5 lb x2, 96(x0)    # x2 = 0xFFFFFFdd
6 sw x2, 160(x0)
7 lb x2, 97(x0)    # x2 = 0xFFFFFFc0
```

```
8  sw x2, 164(x0)
9  lb x2, 98(x0)    # x2 = 0x0000000b
10 sw x2, 168(x0)
11 lb x2, 99(x0)    # x2 = 0xFFFFFFaa
12 sw x2, 172(x0)
13
14 lh x2, 96(x0)    # x2 = 0xFFFFc0dd
15 sw x2, 176(x0)
16 lh x2, 97(x0)    # x2 = 0x00000bc0
17 sw x2, 180(x0)
18 lh x2, 98(x0)    # x2 = 0xFFFFaa0b
19 sw x2, 184(x0)
20 lh x2, 99(x0)    # x2 = 0xFFFFddaa
21 sw x2, 188(x0)
22
23 lbu x2, 96(x0)   # x2 = 0x000000dd
24 sw x2, 100(x0)
25 lbu x2, 97(x0)   # x2 = 0x000000c0
26 sw x2, 104(x0)
27 lbu x2, 98(x0)   # x2 = 0x0000000b
28 sw x2, 108(x0)
29 lbu x2, 99(x0)   # x2 = 0x000000aa
30 sw x2, 112(x0)
31
32 lhu x2, 96(x0)   # x2 = 0x0000c0dd
33 sw x2, 116(x0)
34 lhu x2, 97(x0)   # x2 = 0x00000bc0
35 sw x2, 120(x0)
36 lhu x2, 98(x0)   # x2 = 0x0000aa0b
37 sw x2, 124(x0)
38 lhu x2, 99(x0)   # x2 = 0x0000ddaa
39 sw x2, 128(x0)
```

Listing 2.1: Simulation: load instructions.

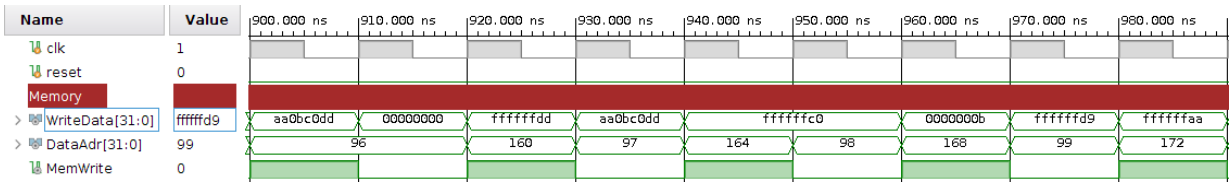

Figure 2.6: Simulation: lb instruction.

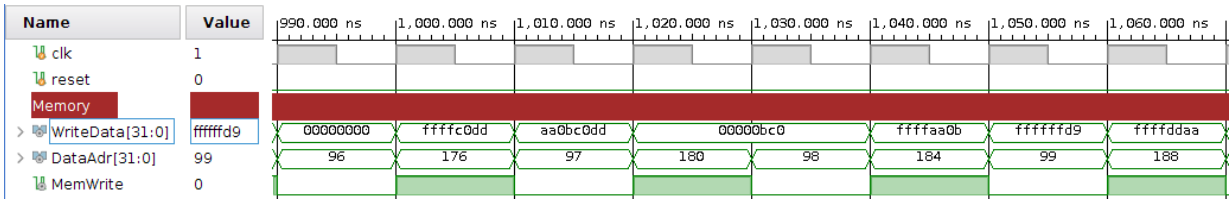

Figure 2.7: Simulation: lh instruction.

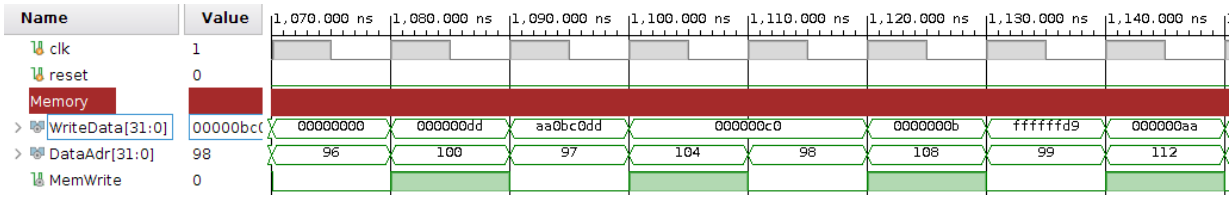

Figure 2.8: Simulation: lbu instruction.

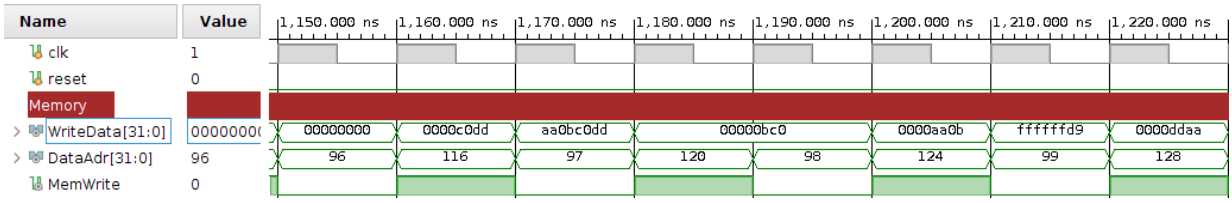

Figure 2.9: Simulation: lhu instruction.

2.3.2 Store Instructions

The listing 2.2 and 2.3 presents the code corresponding to the waveforms in figures 2.10 and 2.11, respectively. As explained in section 2.1.5 it is possible to store a single byte to each position of the word and to store two bytes from an initial position of the word. In case the initial position is the last byte, the processor stores to the memory that byte and to the first byte of the word.

```
1  lui x1, 0xaa0bc
2  addi x1, x1, 0xdd    # x1 = 0xaa0bc0dd
3  sw x1, 96(x0)
4
5  addi x2, x0, 0x77
6  sb x2, 99(x0)        # [96] = 0x770bc0dd
7  lw x3, 96(x0)
8  sw x3, 100(x0)
9
10 addi x2, x0, 0x11
11 sb x2, 98(x0)        # [96] = 0x7711c0dd
12 lw x3, 96(x0)
13 sw x3, 104(x0)
14
15 addi x2, x0, 0x22
16 sb x2, 97(x0)        # [96] = 0x771122dd
```

```
lw x3, 96(x0)
sw x3, 108(x0)

addi x2, x0, 0x33
sb x2, 96(x0)       # [96] = 0x77112233
lw x3, 96(x0)
sw x3, 112(x0)
```

Listing 2.2: Simulation: sb instruction.

```
lui x1, 0xaa0bc
addi x1, x1, 0xdd   # x1 = 0xaa0bc0dd
sw x1, 96(x0)

addi x2, x0, 0xaa
slli x2, x2, 8
addi x2, x2, 0xbb
sh x2, 99(x0)       # [96] = 0xbb0bc0aa
lw x3, 96(x0)
sw x3, 116(x0)

addi x2, x0, 0xcc
slli x2, x2, 8
addi x2, x2, 0xdd
sh x2, 98(x0)       # [96] = 0xccddc0aa
lw x3, 96(x0)
sw x3, 120(x0)

addi x2, x0, 0x33
slli x2, x2, 8
addi x2, x2, 0x44
sh x2, 97(x0)       # [96] = 0xcc3344aa
lw x3, 96(x0)
sw x3, 124(x0)

addi x2, x0, 0x55
slli x2, x2, 8
addi x2, x2, 0x66
sh x2, 96(x0)       # [96] = 0xcc335566
lw x3, 96(x0)
sw x3, 128(x0)
```

Listing 2.3: Simulation: sh instruction.

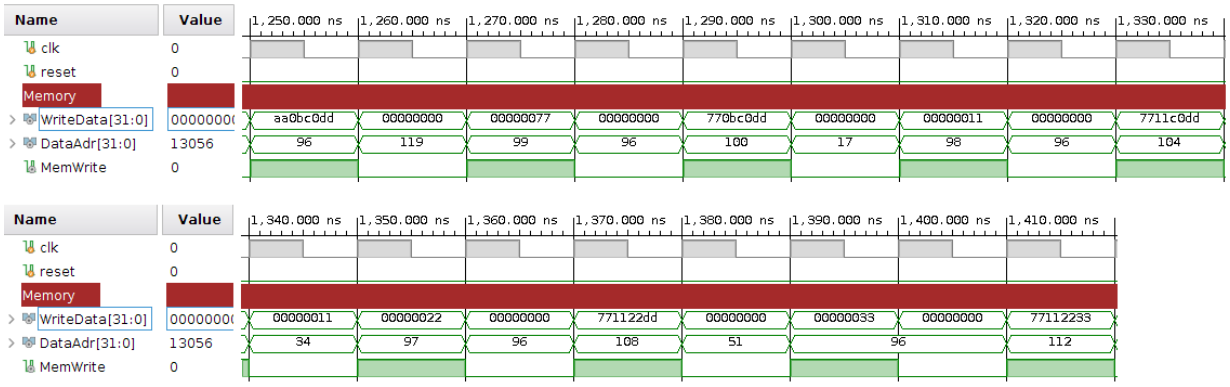

Figure 2.10: Simulation: sb instruction.

Figure 2.11: Simulation: sh instruction.

2.4 Block Design

In order to run the single cycle processor on ZYBO, we used the block diagram in figure 2.12. This is composed by riscvsingle, dmem, imem and even ledcontroler. The ledcontroler is responsible for lighting LED1 if the processor executes all instructions successfully. LEDs 2, 3 and 4 are debug LEDs, as they light up when the code of some instructions is read.

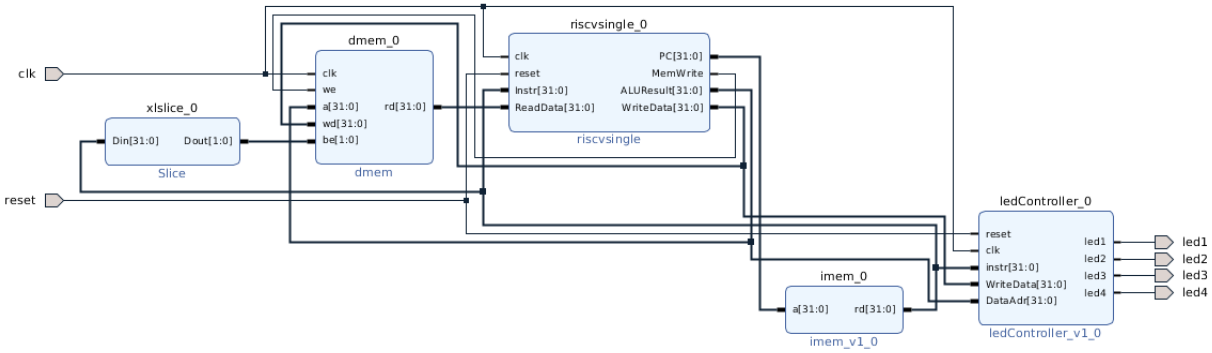

Figure 2.12: Single Cycle: Block Design.

3 | Pipeline

3.1 Introduction

The single cycle processor Pipelining subdivide the single-cycle processor into five pipeline stages, Fetch, Decode, Execute, Memory, and Writeback. The instruction are read from memory on Fetch stage. In the Decode stage, the processor reads the source operands from the register file and decodes the instruction . In the Execute stage, the processor performs a computation with the ALU. In the Memory stage, the processor reads or writes data memory. In the Writeback stage, the processor writes the result to the register file. By doing this the processor throughput is improved because is possible to execute five instructions simultaneously, one in each stage. Because the logic is distributed by the stages it allows higher frequency clocks. Besides the overhead introduced, pipelining gives greate advantages for a reduced cost.

3.2 Datapath

The pipelined datapath is very similar to single-cycle datapath. In fact, it is possible to develop a pipelined datapath by inserting four pipeline registers to the single-cycle one to separate the it into five stages. Another modifiction to perform regards to the register file. The register are read in the Decode stage and are written in the Writeback stage. To make it possible, the register file is written on the falling edge of the CLK so that it can be write a result in the first half of a cycle and read that result in the second half of the cycle. The pipelined datapath is illustrated in figure 3.1.

3.3 Control Unit

The pipelined processor uses the same control signals as the single-cycle processor i.e. the same control unit. All the signals associated to a particular instruction must also advance through the pipeline so that they remain synchronized with the instruction.

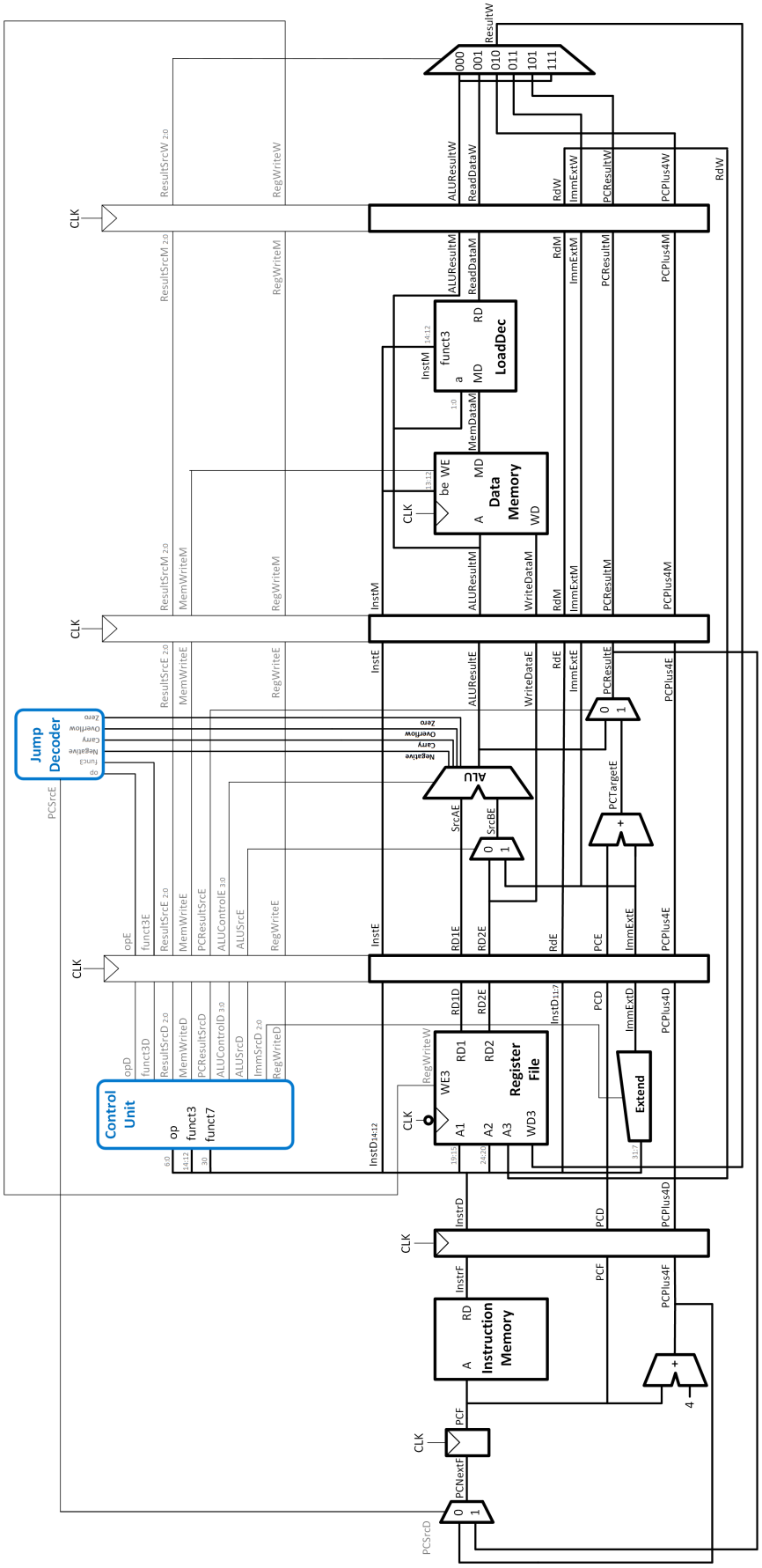

Figure 3.1: Pipeline: Datapath and Control Unit.

3.3.1 Hazard Unit

In a pipelined system, where multiple instructions are handled concurrently, when one instruction depends on the results of another that has not yet completed, a hazard occurs. Hazards are classified as data hazards or control hazards. A **data hazard** occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A **control hazard** occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. Therefore, the hazard unit is responsible for detecting hazards and handling them so that the processor executes the program correctly. In figure 3.2 one can see the pipeline processor with the hazard unit.

When one instruction writes a register and subsequent instructions read this register, this is called a **RAW**[1] **Hazard**. This hazard affects the two subsequent instructions that may need to read that register. A software solution would be to require the programmer or compiler to add nop instructions between the instruction that writes the register and the subsequent instructions that reads the register. However, this is not ideal since it degrades performance. Some data hazards can be solved with **forwarding**, where a result from the memory or writeback stage is forwarded to the execute stage of a dependent instruction. This requires adding MUXs in front of the ALU to select its operands from the register file or memory or writeback stage. This technique is needed when an instruction in execute stage has a source register matching the destination register of an instruction in the memory or writeback stage.

In listing 3.1 one can see the algorithm used to generate the forwarding signal for SrcAE, **ForwardAE**. The forwarding logic for SrcBE, **ForwardBE**, is identical, except that it checks Rs2E instead of Rs1E.

```
1  // Forward from Memory stage
2  if(((Rs1E == RdM) & RegWriteM) & (Rs1E != 0))
3      ForwardAE = 2'b10;
4
5  // Forward from Writeback stage
6  else if(((Rs1E == RdW) & RegWriteW) & (Rs1E != 0))
7      ForwardAE = 2'b01;
8
9  // No forwarding
10 else
11     ForwardAE = 2'b00;
```

Listing 3.1: Hazard Unit: Data Forwarding.

Unfortunately, load instructions does not finish reading data until the end of the memory stage, and therefore, its result cannot be forwarded to the next instruction. A solution is to **stall** the pipeline, holding the operation until the data is available. In order to stall a stage, one must disable the pipeline register, so that the stage's inputs remain unchanged. Along with that, all previous stages must also be stalled so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared (**flushed**), preventing wrong information from propagating forward. As seen in listing 3.2, a stall is executed when a load instruction is in the execute stage, indicated by the LSB of ResultSrcE, and if the load's destination register, RdE, matches Rs1D or Rs2D, the source operands of the instruction in the decode stage. Stalls are suported by adding enable inputs to the fetch and decode pipeline registers, as well as a synchronous reset input to the execute pipeline register.

In order to solve control hazards, one can stall the pipeline until the branch decision is made, i.e, PCSrcE is computed. The processor natively predicts that branches are not taken and simply continues

[1] Read After Write

executing the program in order until PCSrcE is asserted to select the next PC from PCTargetE instead. If the branch should have been taken, then the two instructions following the branch must be flushed by clearing the pipeline registers for those instructions. As seen in listing 3.2, branches stall's require a flush whenever PCSrcE is asserted. These are supported by adding synchronous clear input to the decode pipeline register.

```
// load word stalls
lwStall = ResultSrcb0E & ((Rs1D == RdE) | (Rs2D == RdE));
StallF = lwStall;
StallD = lwStall;

// branch control hazards
FlushD = PCSrcE;
FlushE = lwStall | PCSrcE;
```

Listing 3.2: Hazard Unit: Stalls.

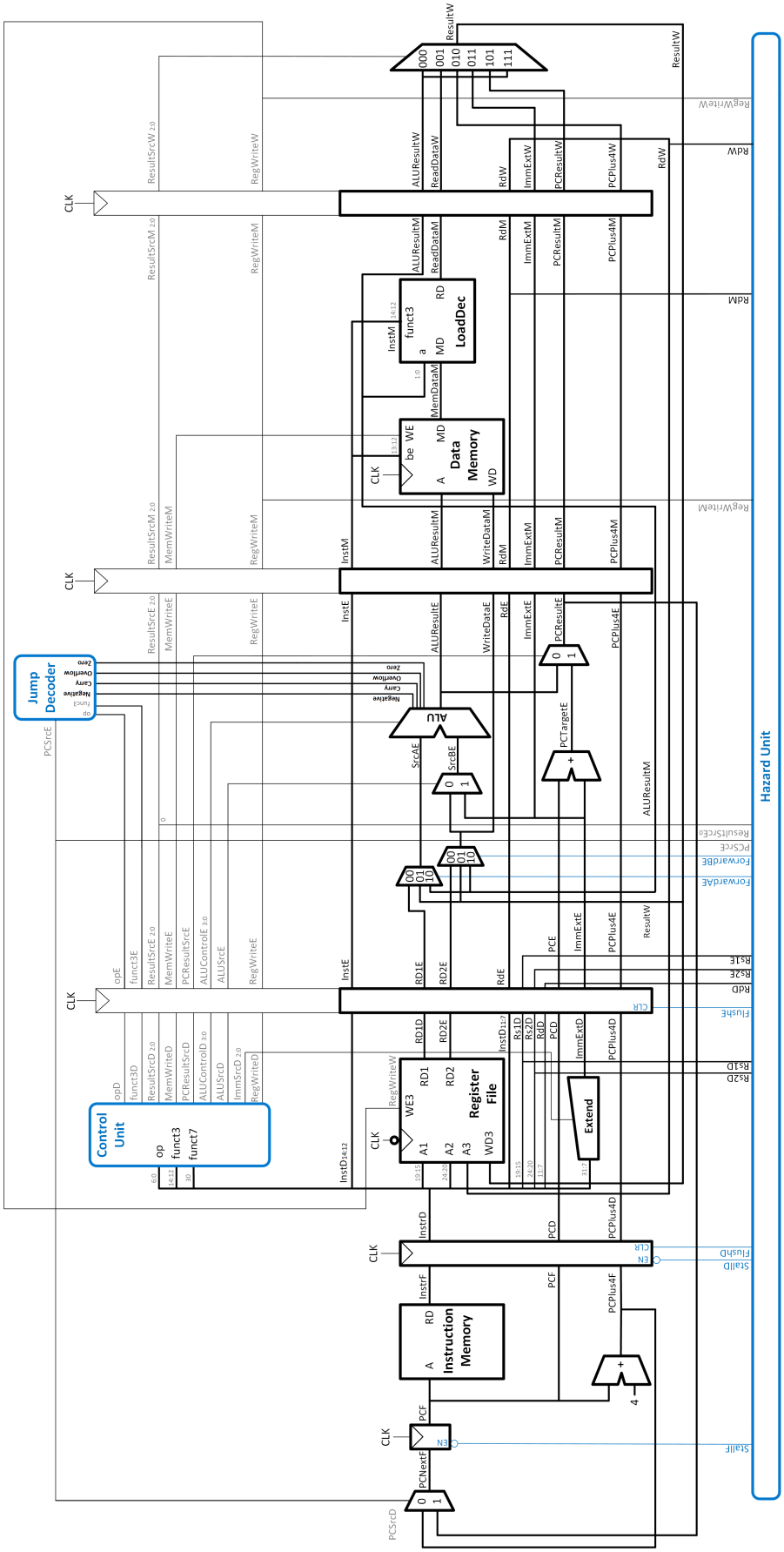

Figure 3.2: Pipelined processor.

3.4 Memory

The instruction memory has a single read port. It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

The data memory has a single read/write port. If its write enable, WE, is asserted, then it writes data WD into address A on the rising edge of the clock. If its write enable is 0, then it reads from address A onto the read data bus, RD.

In this project one used two BRAM memories for the DMEM and IMEM. In both BRAMs were defined 255 bytes of write depth, and enable port were defined as always enabled. One used the clock negative edge to write/read from the memory, in order to reduce its latency, since BRAM inserts one cycle of latency when accessing memory, compared to the previous memory type used in single cycle processor. These BRAMs are declared as well in WRITE FIRST mode, so when executing a write operation, its output presents the data written to memory.

3.4.1 Data Memory - DMEM

In the data memory BRAM, figure 3.3, one generated a 32 bit address interface BRAM, with a 4 bit write byte enable (`.wea`). This allows one to control on which byte, or set of bytes, the write is done. When `.wea` is 0, write operation is disabled. In `.addra` one must pass the address word aligned, i.e, with the last 2 bits equal to zero.

```
bram_32bit bram_dmem (
    .clka(~clk),                // input wire clka
    .rsta(rst),                 // input wire rsta
    .wea(we_bram),              // input wire [3 : 0] wea
    .addra(addr_bram),          // input wire [31 : 0] addra
    .dina(wd_bram),             // input wire [31 : 0] dina
    .douta(rd),                 // output wire [31 : 0] douta
    .rsta_busy(rsta_busy)       // output wire rsta_busy
);
```

Figure 3.3: Pipeline: DMEM BRAM instantiation.

When executing a store byte, `wea` signal must indicate which byte to store, and therefore, it should have only one of the four bits asserted, for example `wea = 4'b0100` in order to write to the third word byte. Along with that, write data signal, `din`, must have the byte that one wants to write. In listing 3.3, one can see the implementation of that, be using the shift operator with the LSBs of the write address, `a`, which index a byte inside a word.

```
1 reg [3:0] we_bram;
2 reg [31:0] wd_bram;
3
4 we_bram = (1 << a[1:0]);
5 wd_bram = (wd << 8*a[1:0]);
```

Listing 3.3: Pipeline: DMEM operation in store byte instruction.

In a store half word, `wea` must have two of the four bits asserted, since half word is two bytes. One can use identical logic as the one used in store byte, but with special attention on the writes of the third and first bytes, i.e, when `wea = 4'b1001`. Using the same logic as above with carry signal, one gets the expected outcome, as seen in listing 3.4.

```
reg Cout_we;
reg [7:0] Cout_wd;

{Cout_we, we_bram} = (3 << a[1:0]);
we_bram = we_bram + Cout_we;

{Cout_wd, wd_bram} = (wd << 8*a[1:0]);
wd_bram = wd_bram + Cout_wd;
```

Listing 3.4: Pipeline: DMEM operation in store half word instruction.

At last, in a store word instruction, all bits of `wea` must be asserted, and the write data goes straight to `din` input port, as shown in listing 3.5.

```
we_bram = 4'b1111;
wd_bram = wd;
```

Listing 3.5: Pipeline: DMEM operation in store word instruction.

3.4.2 Instruction Memory - IMEM

The instruction memory is a read only memory, as shown in figure 3.4 b), where write enable and data in inputs are grounded. This BRAM is loaded with instructions through the use of a .COE file. This file extension is associated with specialized applications and development environments created by the Xilinx. Since PC is always incremented in 4 bytes, to the 32 bit wide instructions, one needs to address IMEM from the lower 2 bits of PC, i.e, BRAM address should be `.addra(PC[31:2])`. Since BRAMs were created with 255 bytes write depth, one must limit address range as shown in the figure. The next instruction is loaded into `rd`.

```
bram imem_bram (
    .clka(~clk),    // input wire clka
    .wea(1'b0),     // input wire [0 : 0] wea
    .addra(a[9:2]), // input wire [7 : 0] addra
    .dina(32'b0),   // input wire [31 : 0] dina
    .douta(rd)      // output wire [31 : 0] douta
);
```

Figure 3.4: Pipeline: IMEM BRAM instantiation.

3.5 Simulations

3.5.1 Data Forwarding

The figure 3.5 present the waveform to the instructions of listing 3.6. It is possible to verify that the second instruction depends on the value calculated in the previous instruction. For the result to be as expected, the value on x3 must be forward.

```
addi x3, x0, 12          # x3 = 12                    4
addi x7, x3, -9          # x7 = (12 - 9) = 3          8
```

Listing 3.6: Simulation: Data Forwarding.

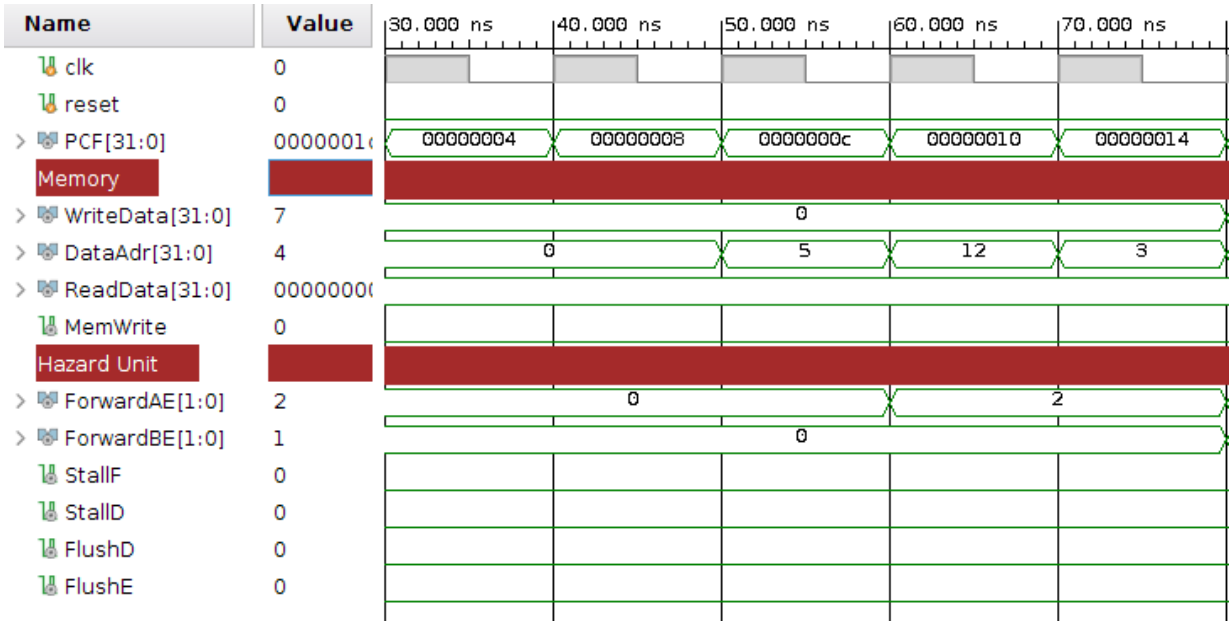

Figure 3.5: Pipeline Simulation: Data Forwarding.

3.5.2 Stall

The figure 3.6 present the waveform to the instructions of listing 3.7. In that case, data forward will not solve the problem alone. As lui stores the value in `x1` and the sw stores `x1` in memory, it will be necessary to stop the execution of the program. For this, a stall is performed, thus ensuring that the data forward that goes to Execute is propagated to Memory and the Write has the value in time to write it.

```
1 lui  x1, 0x01            # x1 = 4096                    50
2 sw   x1, 104(x0)                                        54
```

Listing 3.7: Simulation: Stall.

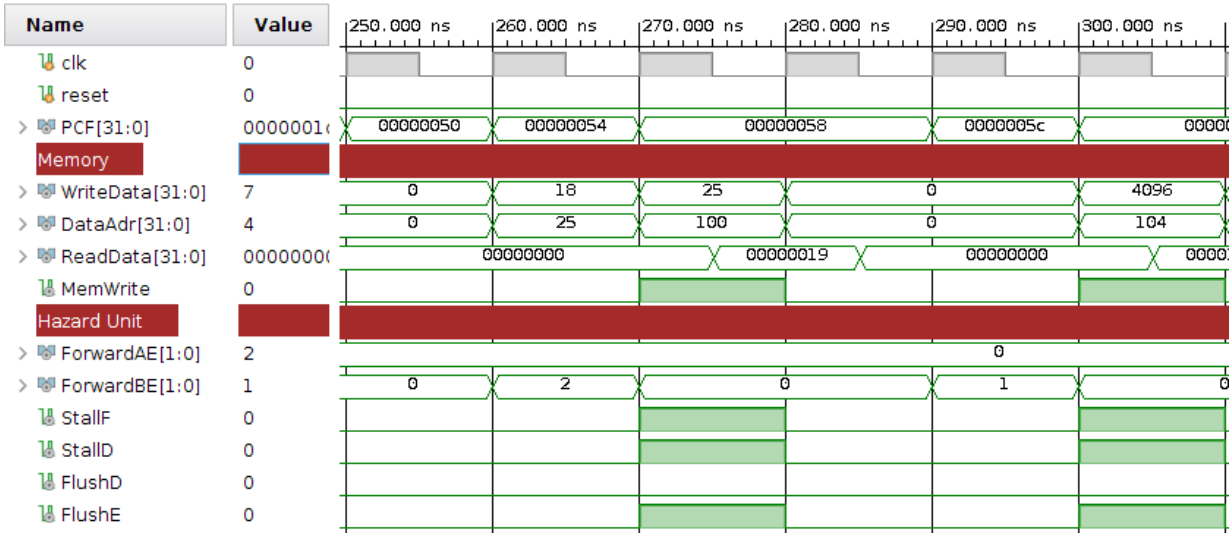

Figure 3.6: Pipeline Simulation: Stall.

3.5.3 Flush

The figures 3.7 and 3.8 present the waveform to the instructions of listings 3.8 and 3.9, respectively. Flush can happen for two reasons: there is a branch or it is a jump instruction. In the case of the jump, as the instruction of beq is verified, that is, `x1 = x2` the PScr is set to 1 and in turn the Flush as well. For the figure 3.8 and the instructions of listing 3.9 the flush happens again as it remains a jump operation.

```
addi x1, x0, 9                                  94
addi x2, x0, 9                                  98
beq x1, x2, beq_label                           9c
addi x1, x0, 8          # jump over             100
beq_label: sw x1, 124(x0)                       104
```

Listing 3.8: Pipeline Simulation: Branch Flush.

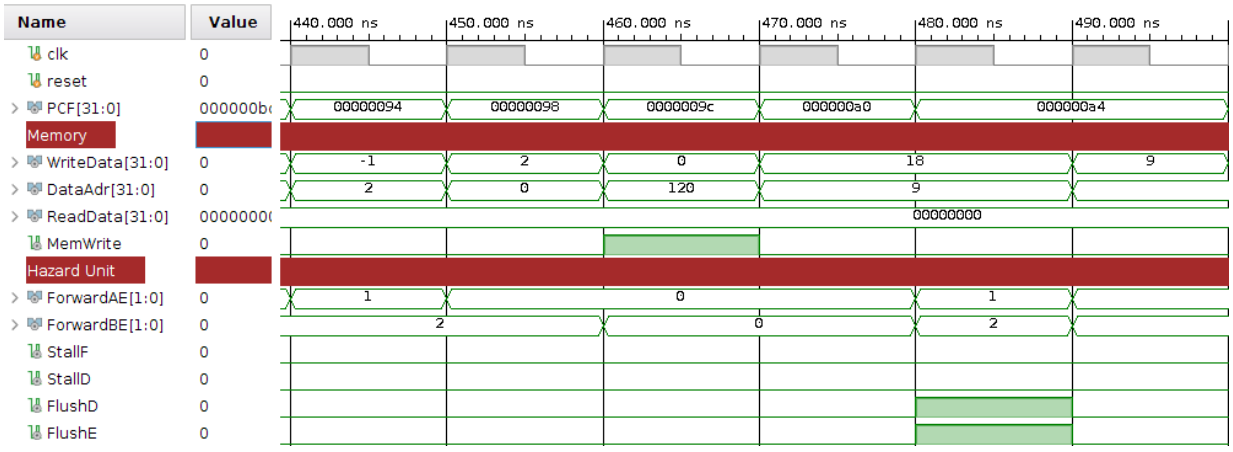

Figure 3.7: Pipeline Simulation: Branch Flush.

```
addi x3, x0, 97      # PC = 96                                    60
jalr x1, x3, 0xF     # PC = 100 ; x1 = 104, PC = x3 + 15 = 70     64
addi x1, x0, 1       # PC = 104 ; dummy - jump over               68
addi x1, x0, 2       # PC = 108 ; dummy - jump over               6c
sw  x1, 112(x0)      # PC = 112                                   70
```

Listing 3.9: Simulation: Jump Flush.

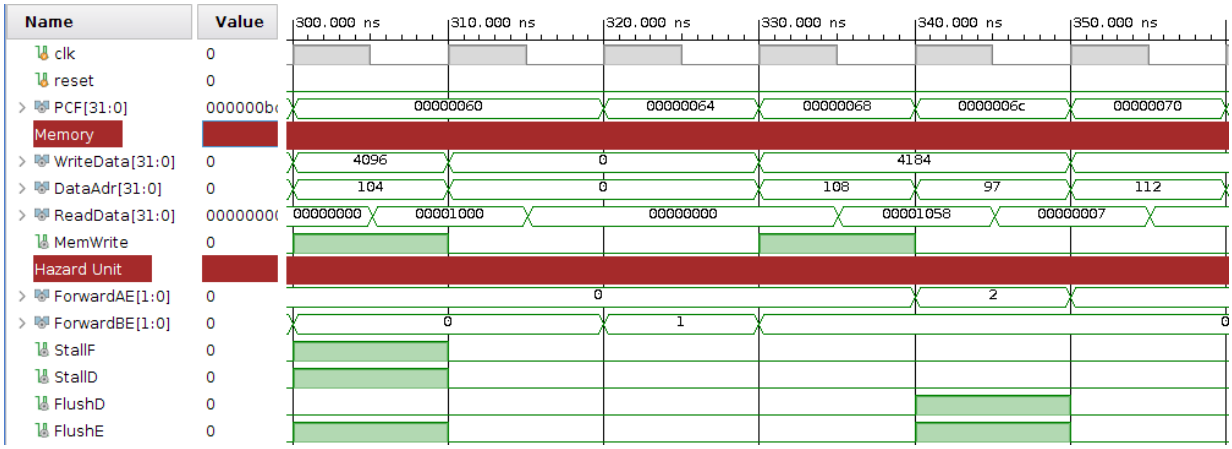

Figure 3.8: Pipeline Simulation: Jump Flush.

3.6 Block Design

In order to run the pipeline processor on ZYBO, we used the block diagram in figure 3.3. This is composed by riscvpipeline, dmem, imem and even ledcontroler. The ledcontroler, as in the case of single cycle, is responsible for lighting LED1 if the processor executes all instructions successfully. LED[2]s 2, 3 and 4 are debug LEDs, as they light up when the code of some instructions is read. For DMEM[3] and IMEM[4], BRAM was used. Therefore, both memories were instantiated, as you can see from the figures 3.3 and 3.4, getting the new block design of the figure 3.9.

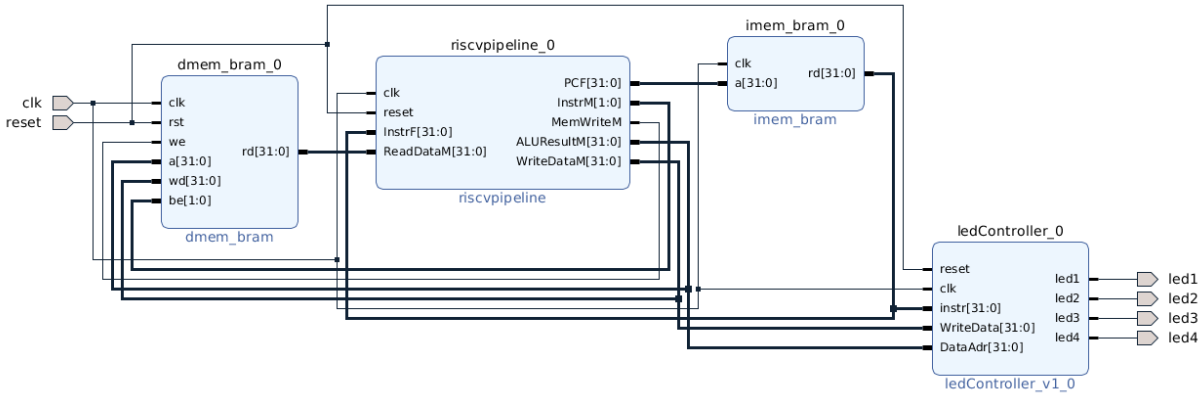

Figure 3.9: Pipeline: Block Design with BRAM.

[2]Light Emiting Diode
[3]Data Memory
[4]Instruction Memory

4 | Tools

4.1 RISC-V gnu-Toolchain

RISC-V offers an official GNU toolchain with support for the GCC compiler, GDB, newlib, and glibc. [4] The listing 4.1 presents the required setup to install this toolchain in a local machine.

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
$ sudo apt-get install autoconf automake autotools-dev curl python3
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex
texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
$ ./configure --prefix=/opt/riscv
$ make linux
```

Listing 4.1: Setup RICV-V compiler toolchain.

In listing 4.2 is presented the script written to generate the hexadecimal code from a given source file. Using RISC-V assembler one can assemble one source program from an assembly written file. Then one can use the `objcopy` tool to copy and translate the created object file, generating a raw binary file. One can then use `od` command to dump the binary file in hexadecimal.

```
#!/bin/bash

infile=../sim/test_all.s
outfile=../sim/riscvtest.txt

riscv64-unknown-linux-gnu-as -o a.elf $infile
riscv64-unknown-linux-gnu-objcopy -O binary -j .text a.elf
od -An -t x4 a.elf > $outfile
sed -i 's/ /\n/g' $outfile
sed -i '/^$/d' $outfile
rm a.elf
```

Listing 4.2: Script to generate Hexadecimal code from source file.

4.2 GitHub

Git is a version control system used for tracking changes in computer files, making it a top-rated utility for programmers world-wide. Git can handle projects of any size. In our case the GIT was used to coordinate the workflow among project team members and track the progress over time. It also allows multiple users to work together without disrupting each other's work. It was then possible to distribute the tasks and work at different times without resorting to .zip or .rar or even pen drives. It is possible access the group's github via https://github.com/TomasLAbreu/riscv-processor. The README explains how someone else can make changes to the Vivado project in a simple way:

- Clone the repo:

```
$ git clone git@github.com:TomasLAbreu/riscv-processor.git
$ cd riscv-processor/
```

- Create the Vivado project:

```
$ cd single-cycle-risc-v/ or pipeline-risc-v/
$ vivado &
```

- When Vivado opens, go to Tools > Run Tcl Script... and run create_proj.tcl.

4.3 RISC-V Interpreter

Another tool used was the RISC-V Interpreter found. Despite not having support for all instructions, it helped us to create the test bench, as well as to understand what could happen with the proposed sequence of instructions. It is possible access the RISC-V Interpreter through the reference [5]. Supported Instructions:

- **Arithmetics**: ADD, ADDI, SUB;
- **Logical**: AND,ANDI,OR,ORI, XOR,XORI;
- **Sets**: SLT, SLTI, SLTU, SLTIU;
- **Shifts**: SRA, SRAI, SRL, SRLISLL, SLLI;
- **Memory**: LW, SW, LB, SB;
- **PC**: LUI, AUIPC;
- **Jumps**: JAL, JALR;
- **Branches**: BEQ, BNE,BLT, BGE, BLTU, BGEU.

4.4 Vivado

Vivado [6] was used in this project. This is software produced by Xilinx for synthesis and analysis of hardware description language, HDL[1] designs. It is a highly complex integrated development environment (IDE) tool for the entire FPGA design and implementation process. It includes a text editor that you can use to create new Verilog files and it does a certain amount of on-the-fly syntax verification.

The basic process in Vivado is to start a new project and then start adding source files to the project. Test bench code files are added separately as simulation-only source. With the files added to the project, you can either go straight to behavioral simulation which will launch the built-in simulator, or you can synthesize the design and then do post-synthesis functional or timing simulations. You can also view a schematic of the “elaborated design” inferred from your source code. Then continue after synthesis to implementation timing constraints and pin assignments, place & route the design, (iterate if the touted design does not meet all timing constraints), and finally generate a bitstream for programming the real part.

[1] Hardware Description Language

5 | Experimental Results

5.1 Test Bench

In listing 5.1 is shown a snippet of the written test bench. This will validate if the program presented in appendix Appendix A runs correctly in the designed micro architecture. If memory write is asserted one checks if `WriteData` written in `DataAdr` match the expected result. If so, one uses the `printOK` macro to print to the console the name of the instruction which passed on the test, as well as to keep track of the number of instructions that passed, by using a counter variable `count`.

In order to complete the test bench one executes a store instruction, writing 30 into address 40. If there is a write with a different value than expected or to a different address, the simulation fails. One used addresses 96 to 99 to store data that wasn't supposed to be checked by the test bench.

```
always @(negedge clk) begin
    if (MemWrite) begin
        if ((DataAdr == 104) & (WriteData == 4096))
            `printOK("lui",count)
        else if ((DataAdr == 108) & (WriteData == 4184))
            `printOK("auipc",count)
        // ...
        // more instructions checks
        // ...
        else if ((DataAdr == 40) && (WriteData == 30)) begin
                $display("\nSimulation completed");
                $display("  %2d/37 instructions PASSED\n", count);
                $stop;
        end
        else if ((DataAdr < 96) && (DataAdr > 99))
                $display("\nSimulation failed");
                $display("  dataAddr  = %d", DataAdr);
                $display("  writeData = %d\n", WriteData);
                $stop;
        end
    end
end
```

Listing 5.1: Test Bench Snippet.

In listing 5.2 is shown the test bench output when running it with single-cycle or pipeline processor.

```
Time resolution is 1 ps
        lw OK
      addi OK
        sw OK
       add OK
       sub OK
        or OK
       and OK
       jal OK
       lui OK
     auipc OK
      jalr OK
       slt OK
      sltu OK
```

```
15         beq OK
16         bne OK
17         blt OK
18         bge OK
19        bltu OK
20        bgeu OK
21         xor OK
22        xori OK
23         ori OK
24        andi OK
25        slti OK
26       sltiu OK
27        slli OK
28        srli OK
29        srai OK
30         sll OK
31         srl OK
32         sra OK
33          lb OK
34          lh OK
35         lbu OK
36         lhu OK
37          sb OK
38          sh OK
39
40 Simulation completed
41   37/37 instructions PASSED
```

Listing 5.2: Test Bench Results.

5.2 Timing Reports

The timing reports presented in this section are based in the timing report summaries generated using the default settings defined by Vivado, not being defined any other user time constraints. This time report is divided into two types: the setup times report and the hold times report. Failing in this timing reports may lead to a failure of the design when programming the hardware. The setup violations can be resolved by reducing the clock speed, adding pipeline stages and optimize critical paths. On the other hand, to solve hold time violations it can be added extra logic to the failing paths, but, in general, this are very hard to solve. Therefore, it is expected to the pipeline processor fulfill the setup constraints more easily than the single cycle processor, because of the pipeline stages, still the critical path has increased. So, it is predictable that the clock source may have a higher frequency for the pipeline processor, than the single cycle processor.

Next, it will be shown the timing reports summary for the single cycle, pipeline processor without the BRAMs instruction and data memories and for the pipeline processor with the BRAMs instruction and data memories.

5.2.1 Single Cycle Processor

In the figure 5.1 is shown the report timing summary for the single cycle processor, fed by a clock source with 7 ns period, approximately 142.857 MHz. It can be seen that the setup times report didn't passed, as the *Worst Negative Slack* (WNS) is negative, meaning that the register's inputs of the critical path are only stable 0.497 ns after the next clock cycle.

Setup		Hold	
Worst Negative Slack (WNS):	-0.497 ns	Worst Hold Slack (WHS):	0.264 ns
Total Negative Slack (TNS):	-17.508 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	80	Number of Failing Endpoints:	0
Total Number of Endpoints:	934	Total Number of Endpoints:	934

Figure 5.1: Performance Analysis: Single cycle timing report for a clock frequency of 142,857 MHz (7 ns).

In order to make this processor pass the timing constraints, it was increased the clock period and subsequently decrease the clock frequency. The clock period was increased by 1 ns, being 8 ns, as shown in the figure 5.2. With this clock frequency, the design was able to pass the timing constraints, being the register's inputs stable 0.433 ns before the next clock cycle.

Setup		Hold	
Worst Negative Slack (WNS):	0.433 ns	Worst Hold Slack (WHS):	0.292 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	931	Total Number of Endpoints:	931

Figure 5.2: Performance Analysis: Single cycle timing report for a clock frequency of 125 MHz (8 ns).

5.2.2 Pipeline Processor

For the pipeline processor, the clock frequency that brings timing faults is presented in the figure 5.3. With the clock frequency of 166.667 MHz (6 ns), the register's inputs of the critical path are only stable 0.264 ns after the next clock cycle.

Setup		Hold	
Worst Negative Slack (WNS):	-0.264 ns	Worst Hold Slack (WHS):	0.056 ns
Total Negative Slack (TNS):	-2.137 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	20	Number of Failing Endpoints:	0
Total Number of Endpoints:	2876	Total Number of Endpoints:	2876

Figure 5.3: Performance Analysis: Pipeline timing report for a clock frequency of 166.667 MHz (6 ns).

Increasing the clock period by 1 ns, it is obtained a clock frequency of 142.847 MHz, where the the register's inputs of the critical path are stable 0.173 ns before the next clock cycle, as presented in the figure 5.4.

Setup		Hold	
Worst Negative Slack (WNS):	0.173 ns	Worst Hold Slack (WHS):	0.107 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2876	Total Number of Endpoints:	2876

Figure 5.4: Performance Analysis: Pipeline timing report for a clock frequency of 142.857 MHz (7 ns).

As one can see, with the pipeline processor, it is possible to decrease the clock frequency by 1 ns relatively to the single cycle processor, so the performance for the pipeline processor is bigger than the single cycle.

5.2.3 Pipeline Processor with BRAM

After adding the BRAM memories to the processor, it is expected that the processor performance decreases, as the critical path gets longer, being the times to access the register bigger.

The figure 5.5, shows that for a clock period of 9 ns, the register's inputs are not stable 0.255 ns before the next clock cycle, meaning that the setup times were not met.

Setup		Hold	
Worst Negative Slack (WNS):	-0.255 ns	Worst Hold Slack (WHS):	0.119 ns
Total Negative Slack (TNS):	-2.922 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	23	Number of Failing Endpoints:	0
Total Number of Endpoints:	1637	Total Number of Endpoints:	1637

Figure 5.5: Performance Analysis: Pipeline processor with BRAM memory timing report for a clock frequency of 111.111 MHz (9 ns).

After increasing the clock period by 1 ns, the clock frequency is 100 MHz (10 ns period). With this clock frequency, the register's inputs are stable 0.157 ns before the next clock frequency.

Setup		Hold	
Worst Negative Slack (WNS):	0.157 ns	Worst Hold Slack (WHS):	0.122 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1637	Total Number of Endpoints:	1637

Figure 5.6: Performance Analysis: Pipeline timing report for a clock frequency of 100 MHz (10 ns).

6 | Conclusions

This project allowed to consolidate and put into practice the knowledge acquired during this semester in the specialization of Embedded Systems, namely on micro architecture, hardware design and virtual memories.

Initially a single-cycle processor micro architecture was designed, using the datapath and control unit studied in Embedded Systems classes. This presented some difficulties when implementing some instructions like store, load and branch. This required the addition of several MUXs and logical units in the datapath in order to provide support for those instructions, requiring as well changes on the initial control unit.

With all instructions implemented and tested on the developed single-cycle architecture, one upgraded to a pipeline processor. Pipelining, is a powerful way to improve the throughput of a digital system, and therefore made total sense to be approached in this project. One can design a pipeline processor by subdividing the single-cycle processor into five pipeline stages. With this, five instructions can execute simultaneously, one in each stage. As a result of that the clock frequency is faster. The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds many 32-bit pipeline registers, along with multiplexers, smaller pipeline registers, and control logic to resolve hazards. So, ideally, the latency of each instruction is unchanged, but the throughput is five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be as high as one might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

After having both micro architectures design running on Zybo's board one explored another type of memory to use in DMEM and IMEM. Eventually, BRAM memories were used, being instantiated inside DMEM and IMEM blocks, communicating through a custom bus.

In short, this was a great opportunity to enlarge our knowledge in RISC-V architecture, as well as to develop our hardware design skills.

Bibliography

[1] K. A. Andrew Waterman, *RISC-V Instruction Set SummaryRISC-V Instruction Set Summary*, 2nd ed., CS Division, EECS Department, University of California, Berkeley, https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, may 2017.

[2] ——, *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, 20191213th ed., CS Division, EECS Department, University of California, Berkeley, https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf, December 2019.

[3] S. L. Harris and D. M. Harris, *Digital Design and computer architecture: RISC-V edition*. Morgan Kaufmann Publishers, 2022.

[4] Riscv-Collab, "Riscv-collab/riscv-gnu-toolchain: Gnu toolchain for risc-v, including gcc." [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[5] D. Qiu, "Risc-v interpreter." [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/

[6] Xilinx, "Vivado." [Online]. Available: https://www.xilinx.com/support/download.html

Appendices

Appendix A

```
main:   addi x2, x0, 5          # x2 = 5                    0
        addi x3, x0, 12         # x3 = 12                   4
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10
        add  x5, x5, x4         # x5 = 4 + 7 = 11           14
        beq  x5, x7, end        # shouldnt be taken         18
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C
        beq  x4, x0, around     # should be taken           20
        addi x5, x0, 0          # shouldnt execute          24
around: slt x4, x7, x2          # x4 = (3 < 5) = 1          28
        add x7, x4, x5          # x7 = (1 + 11) = 12        2C
        sub x7, x7, x2          # x7 = (12 - 5) = 7         30
        sw  x7, 84(x3)          # [96] = 7                  34
        lw  x2, 96(x0)          # x2 = [96] = 7             38
        add x9, x2, x5          # x9 = (7 + 11) = 18        3C
        jal x3, end             # jump to end, x3 = 0x44    40
        addi x2, x0, 1          # shouldnt execute          44
end:    add x2, x2, x9          # x2 = (7 + 18) = 25        48
        sw  x2, 0x20(x3)        # [100] = 25                4C
lui x1, 0x01            # x1 = 4096
sw  x1, 104(x0)

auipc x1, 1            # x1 = 4096 + PC(88) = 4184
sw  x1, 108(x0)

addi x3, x0, 97          # PC = 96
jalr x1, x3, 0xF        # PC = 100 ; x1 = 104, PC = x3 + 15 = 16
addi x3, x0, 1          # PC = 104 ; dummy - jump over
addi x3, x0, 2          # PC = 108 ; dummy - jump over
sw  x1, 112(x0)         # PC = 112

addi x2, x0, -1
addi x3, x0, 2
slt x4, x2, x3
sw x4, 116(x0)

addi x2, x0, -1
addi x3, x0, 2
sltu x4, x2, x3
sw x4, 120(x0)

addi x1, x0, 9
addi x2, x0, 9
beq x1, x2, beq_label
addi x1, x0, 8 # jump over
beq_label: sw x1, 124(x0)
```

```
48
49 addi x1, x0, 9
50 addi x2, x0, 10
51 bne x1, x2, bne_label
52 addi x1, x0, 8 # jump over
53 bne_label: sw x1, 128(x0)
54
55 addi x1, x0, -1
56 addi x2, x0, 1
57 blt x1, x2, blt_label
58 addi x1, x0, 8 # jump over
59 blt_label: sw x1, 132(x0)
60
61 addi x1, x0, 1
62 addi x2, x0, -1
63 bge x1, x2, bge_label
64 addi x1, x0, 8 # jump over
65 bge_label: sw x1, 136(x0)
66
67 addi x1, x0, 1
68 addi x2, x0, -1
69 bltu x1, x2, bltu_label
70 addi x1, x0, 8 # jump over
71 bltu_label: sw x1, 140(x0)
72
73 addi x1, x0, -1
74 addi x2, x0, 1
75 bgeu x1, x2, bgeu_label
76 addi x1, x0, 8 # jump over
77 bgeu_label: sw x1, 144(x0)
78
79 addi x1, x0, 0x54
80 addi x2, x0, 0xaa
81 xor x3, x1, x2 # 0x54 ^ 0xaa = 254
82 sw x3, 148(x0)
83
84 addi x1, x0, 0x98
85 xori x3, x1, 0x26 # 0x98 ^ 0x26 = 190
86 sw x3, 152(x0)
87
88 ori x3, x1, 0x72  # 0x98 | 0x26 = 250
89 sw x3, 156(x0)
90
91 andi x3, x1, 0x6a # 0x98 & 0x6a = 8
92 sw x3, 160(x0)
93
94 slti x3, x1, -1 # 0x98 < 0xfff = 0
95 sw x3, 164(x0)
96
```

```
97  # slti x3, x1, 0x9f # 0x98 < 0x9f = 1
98  # sw x3, 168(x0)
99
100 sltiu x3, x1, -1 # 0x98 < 0xfff = 1
101 sw x3, 172(x0)
102
103 # sltiu x3, x1, 0x92 # 0x98 < 0x92 = 0
104 # sw x3, 176(x0)
105
106
107 addi x1, x0, -77
108 slli x3, x1, 0x1
109 sw x3, 100(x0)
110
111 srli x3, x1, 0x1
112 sw x3, 104(x0)
113
114 srai x3, x1, 0x1
115 sw x3, 108(x0)
116
117
118 addi x2, x0, 0x1
119 sll x3, x1, x2
120 sw x3, 112(x0)
121
122 srl x3, x1, x2
123 sw x3, 116(x0)
124
125 sra x3, x1, x2
126 sw x3, 120(x0)
127
128 lui x1, 0xaa0bc
129 addi x1, x1, 0xdd
130 sw x1, 96(x0) # x1 = 0xaa0bc0dd
131
132 lb x2, 96(x0) # x2 = 0xFFFFFFdd = -35
133 sw x2, 160(x0)
134
135 lb x2, 97(x0) # x2 = 0xFFFFFFc0 = -64
136 sw x2, 164(x0)
137
138 lb x2, 98(x0) # x2 = 0x0000000b = 11
139 sw x2, 168(x0)
140
141 lb x2, 99(x0) # x2 = 0xFFFFFFaa = -86
142 sw x2, 172(x0)
143
144
145 lh x2, 96(x0) # x2 = 0xFFFFc0dd = -16163
```

```
146 sw x2, 176(x0)
147
148 lh x2, 97(x0) # x2 = 0x00000bc0 = 3008
149 sw x2, 180(x0)
150
151 lh x2, 98(x0) # x2 = 0xFFFFaa0b = -22005
152 sw x2, 184(x0)
153
154 lh x2, 99(x0) # x2 = 0xFFFFddaa = -8790
155 sw x2, 188(x0)
156
157
158 lbu x2, 96(x0) # x2 = 0x000000dd = 221
159 sw x2, 100(x0)
160
161 lbu x2, 97(x0) # x2 = 0x000000c0 = 192
162 sw x2, 104(x0)
163
164 lbu x2, 98(x0) # x2 = 0x0000000b = 11
165 sw x2, 108(x0)
166
167 lbu x2, 99(x0) # x2 = 0x000000aa = 170
168 sw x2, 112(x0)
169
170
171 lhu x2, 96(x0) # x2 = 0x0000c0dd = 49373
172 sw x2, 116(x0)
173
174 lhu x2, 97(x0) # x2 = 0x00000bc0 = 3008
175 sw x2, 120(x0)
176
177 lhu x2, 98(x0) # x2 = 0x0000aa0b = 43531
178 sw x2, 124(x0)
179
180 lhu x2, 99(x0) # x2 = 0x0000ddaa = 56746
181 sw x2, 128(x0)
182
183
184
185 lui x1, 0xaa0bc
186 addi x1, x1, 0xdd # x1 = 0xaa0bc0dd
187 sw x1, 96(x0)
188
189 addi x2, x0, 0x77
190 sb x2, 99(x0) # [96] = 0x770bc0dd
191 lw x3, 96(x0)
192 sw x3, 100(x0)
193
194 addi x2, x0, 0x11
```

```
sb x2, 98(x0) # [96] = 0x7711c0dd
lw x3, 96(x0)
sw x3, 104(x0)

addi x2, x0, 0x22
sb x2, 97(x0) # [96] = 0x771122dd
lw x3, 96(x0)
sw x3, 108(x0)

addi x2, x0, 0x33
sb x2, 96(x0) # [96] = 0x77112233
lw x3, 96(x0)
sw x3, 112(x0)


lui x1, 0xaa0bc
addi x1, x1, 0xdd # x1 = 0xaa0bc0dd
sw x1, 96(x0)

addi x2, x0, 0xaa
slli x2, x2, 8
addi x2, x2, 0xbb
sh x2, 99(x0) # [96] = 0xbb0bc0aa
lw x3, 96(x0)
sw x3, 116(x0)

addi x2, x0, 0xcc
slli x2, x2, 8
addi x2, x2, 0xdd
sh x2, 98(x0) # [96] = 0xccddc0aa
lw x3, 96(x0)
sw x3, 120(x0)

addi x2, x0, 0x33
slli x2, x2, 8
addi x2, x2, 0x44
sh x2, 97(x0) # [96] = 0xcc3344aa
lw x3, 96(x0)
sw x3, 124(x0)

addi x2, x0, 0x55
slli x2, x2, 8
addi x2, x2, 0x66
sh x2, 96(x0) # [96] = 0xcc335566
lw x3, 96(x0)
sw x3, 128(x0)


```

```
244 # ------- end simulation
245 addi x2, x0, 30
246 sw  x2, 40(x0)
```