VHDL MIPS pipeline processor implementation

Auteurs :
Gabriel FREITAS OLIVEIRA

Last Update
June 28, 2020

Chapter 1

General Architecture

This document aims to describe a simple MIPS pipeline architecture processor that was implemented in VHDL using Quartus II. First, let's take a look on the requirements of the project.

- The processor must be a pipeline variant
- It must use a 16 bit architecture
- It must have 8 registers on cache, where register 0 is always 0 (also referenced as $ZERO)
- It must interact with the user with a 16-bit input and a 16- bit output
- It must execute these instructions
  - **ADD $DestReg, $Reg1, $Reg2** → $DestReg = $Reg1 + $Reg2
  - **ADDI $DestReg, $Reg1, Constant** → $DestReg = $Reg1 + Const
  - **NOR $DestReg, $Reg1, $Reg2** → $DestReg = $Reg1 NOR $Reg2
  - **AND $DestReg, $Reg1, $Reg2** → $DestReg = $Reg1 AND $Reg2
  - **BEQ $Reg1, $Reg2, Jump_to_line** → if($Reg1 == $Reg2) PC = LINE
  - **JUMP Jump_to_line** → PC = Line
  - **LOAD $DestReg, $RAM_ADDR** → $DestReg = $RAM_ADDR
  - **STORE $SrcReg, $RAM_ADDR** → $RAM_ADDR = $SrcReg
  - **IN $DestReg** → $DestReg = IN
  - **OUT $SourceReg** → OUT = $SourceReg
  - **MOVE $DestReg, $SourceReg** → $DestReg = $SourceReg
  - **BGT $Reg1, $Reg2, Jump_to_line** → if($Reg1 > $Reg2) PC = LINE

Knowing that, I decided to separate the instruction in 4 parts: 4-bit OP, 3-bit First Segment, 3-bit second segment, and 6-bit last segment. With that in mind, here is the proposed architecture:

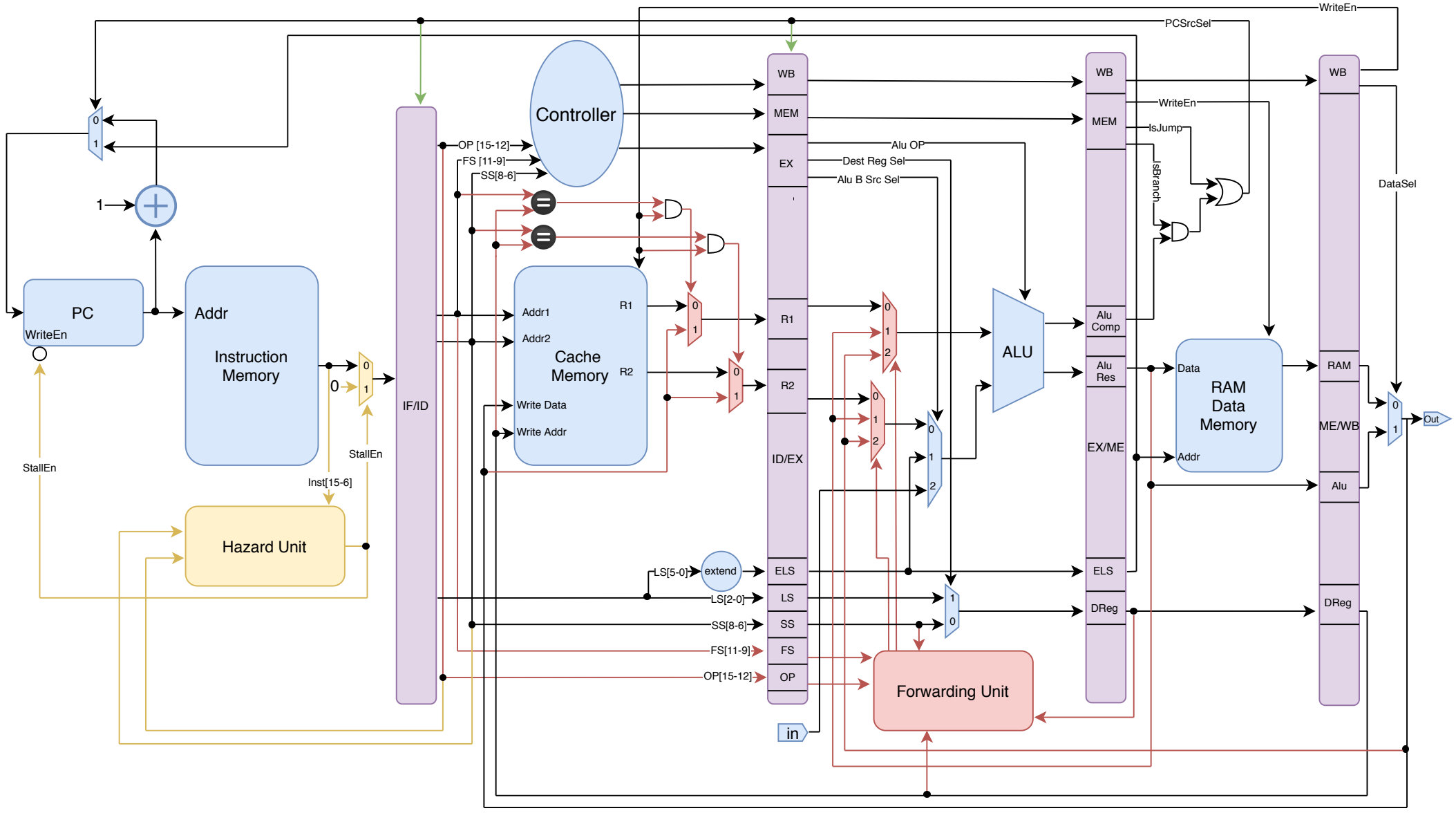

Figure 1.1: Processor Architecture

The processor has 5 different steps, Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). To guarantee that the next instruction will not interfere with the previous ones, 4 interface registers were used (the purple ones). The processor has 3 control units : the Hazard Unit, the Forwarding Unit, and the Main Controller. The Hazard Unit is responsible for enabling the StallEn signal, blocking the PC's WriteEn, and stoping the IF step, and send "0" to IF/ID, this way a NOP is sent to the other parts. The forwaring unit is responsible for avoiding data hazards, where an instruction depends on the result of a loaded instruction that has not finished yet. The main controller will set every control signal so that the instruction takes the right path though the processor's components. All these components will be explained later

This architecture implements branch prediction, where we assume that the branch will never be taken, and continue loading the next instructions. Once the branch is taken (hence PCSrcSel = 1), this signal is used to reset IF/ID and ID/EX registers, erasing the loaded instructions. The interface processors will also skip the next instruction, since at the rising edge PCSrcSel is still equals to 1, what is really helpful since the correct instruction is still not fetched (effectively sending 2 NOPs to the processor). The last feature present in the architecture is in the ID step, where we select between the Write Data and the output of the Cache memory. This logic is used to avoid a structural hazard when the instruction loaded in ID is trying to read from an address at the same time the instruction loaded in WB is trying to write it (it this happens, EX will be fed with the outdated values), thus we check if the Write Addr equals any of the read address and the Cache WriteEn is set. If so, we redirect the data sent to ID/EX to feed the Write Data instead. This is done automatically by the data path, and none of the controller units have to interact with it.

The following table depicts the instruction set of this processor, and their assembly code ("X" means don't care):

Instruction Table				
Instruction	ID (binary)	Assembly Example	Arguments	Machine code
ADD	0001	ADD $R3, $R1, $R2	R3 → Destiny Register R1 → Source register 1 R2 → Source register 2	0001 001 010 000|011
ADDI	0010	ADDI $R3, $R1, 10	R3 → Destiny Register R1 → Source register 1 10 → 6 bit constant	0010 001 011 001010
NOR	0011	NOR $R3, $R1, $R2	R3 → Destiny Register R1 → Source register 1 R2 → Source register 2	0011 001 010 000|011
AND	0100	AND $R3, $R1, $R2	R3 → Destiny Register R1 → Source register 1 R2 → Source register 2	0100 001 010 000|011
BEQ	0101	BEQ $R1, $R2, 10	R1 → Equals operator 1 R2 → Equals operator 2 10 → Line to jump to	0101 001 010 001010
JUMP	0110	JUMP 10	10 → Line to jump to	0110 XXX XXX 001010
LOAD	0111	LOAD $R1, 10	R1 → Destiny Register 10 → RAM address	0111 XXX 001 001010
STORE	1000	STORE $R1, 10	R1 → Source Register 10 → RAM address	1000 000 001 001010
IN	1001	IN $R1	R1 → Destiny Register	1010 000 001 XXXXXX
OUT	1010	OUT $R1	R1 → Data Source	1011 000 001 XXXXXX
MOVE	1011	MOVE $R1, $R2	R1 → Destiny Register R2 → Source Register	1100 000 010 000|001
BGT	1100	BGT $R1, $R2, 10	R1 → Greater operator 1 R2 → Greater operator 2 10 → Line to jump to	1101 001 010 001010

NOTE : BGT will always consider signed inputs.

The limitations of this kind of architecture are:

- Only holds a maximum of 16 different instructions
- Register bank can only have a maximum of 8 registers
- ADDI instruction can only assign values between 0 and 63, or -32 and 31
- Jump instructions can only jump to lines between 0 and 63
- LOAD and STORE can only get data in addresses 0 to 63

To test the processor, this simple code has been made

```
int main(void)
{
    int i, j, k;
    int input, output;

    i = 22;
    j = -2;
    k = 16;
    input = 0;
    output = 0;

    while (input != 2) {
        if (input > k)
            output += j;
        else
            output += i;
        scanf("%d", input);
    }
    printf("%d", output);
}
```

```
    ADDI $R1, $ZERO, 22 //i = 22
    ADDI $R2, $ZERO, -2 //j = -2
    ADDI $R3, $ZERO, 16 //k = 16
    ADDI $R4, $ZERO, 0 //input = 0
    ADDI $R5, $ZERO, 0 //output = 0
    ADDI $R6, $ZERO, 2 //temp = 2
LOOP:
    BEQ $R4, $R6, END //END if ==
    BGT $R4, $R3, IF //IF if >
    ADD $R5, $R5, $R1 //output+=i
    JUMP ENDIF //goto ENDIF
IF:
    ADD $R5, $R5, $R2 //output+=j
ENDIF:
    IN $R4 //scanf
    JUMP LOOP //While
END:
    OUT $R5 //printf
```

This code was chosen to use various different instructions, while only using 6 registers from the cache. Using Quartus simulation (Modelsim does not work in my computer, so I had to work with Quartus), I ran a timing simulation that takes 3us with a 50 MHz clock. The results can be verified in figure 1.2

Since input will always begin as 0, the output will be 22 -2*n where n is the times the IF part is executed. By counting the ALU compare high states, we can verify how many times the If part was ran, in our case 4 times (the last one is the BEQ to end the loop), thus output = 14. Using the simulation, I theorize a maximum clock of 100 MHz, but since this code was not tested on a real board, this number can vary a lot. The compilation was made using a Cyclone II EP2C5T144C8 containing 89 pins, while the debugging used Cyclone IV E EP4CE40F29I8L containing 533 pins (every signal was outputted on the pins, thus needing a large amount of them (NOTE: modelSim software crashes on my PC)). If you want more details, the next section will cover the component details.

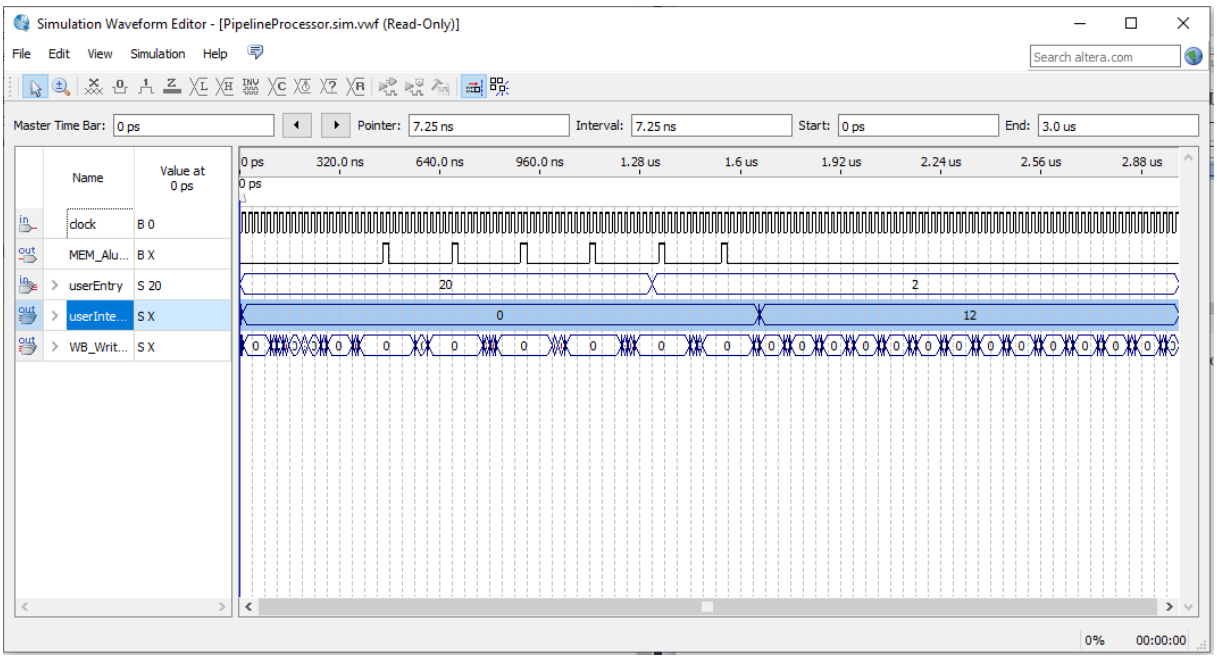

Figure 1.2: Processor Results

1.1 Comparison with multi-cycle architecture

By doing the same simulation (using the same clock and the same input signals), we can compare this architecture with the Multi-cycle variant made previously. The first Alu Compare pulse in the multi-cycle happens at 880 us, while the pipeline triggers it at 460 us, meaning a considerable 48% increase in throughput. This happens because the first 8 instructions need 26 cycles to execute (24 to raise alu compare) on the multi-cycle, while only 12 cycles in the pipeline. The period between alu compare raises also tells us the improved efficiency of pipeline, the multi-cycle has a 320 us while the pipeline 220 us, a 31% increase. This happens because this code has a lot of jumping instructions which erase the loaded instructions, inserting 2 NOP every time a jump is made. In an ideal scenario, the pipeline would only process R-type instructions, and considering permanent period each one can be executed in 1 cycle, while consuming 3 for the multi-cycle, a 66% increase. However, in a bad scenario, where jump instructions are abundant, the pipeline requires 5 cycles, while the Multi-cycle only 4, causing a 1 cycle loss.

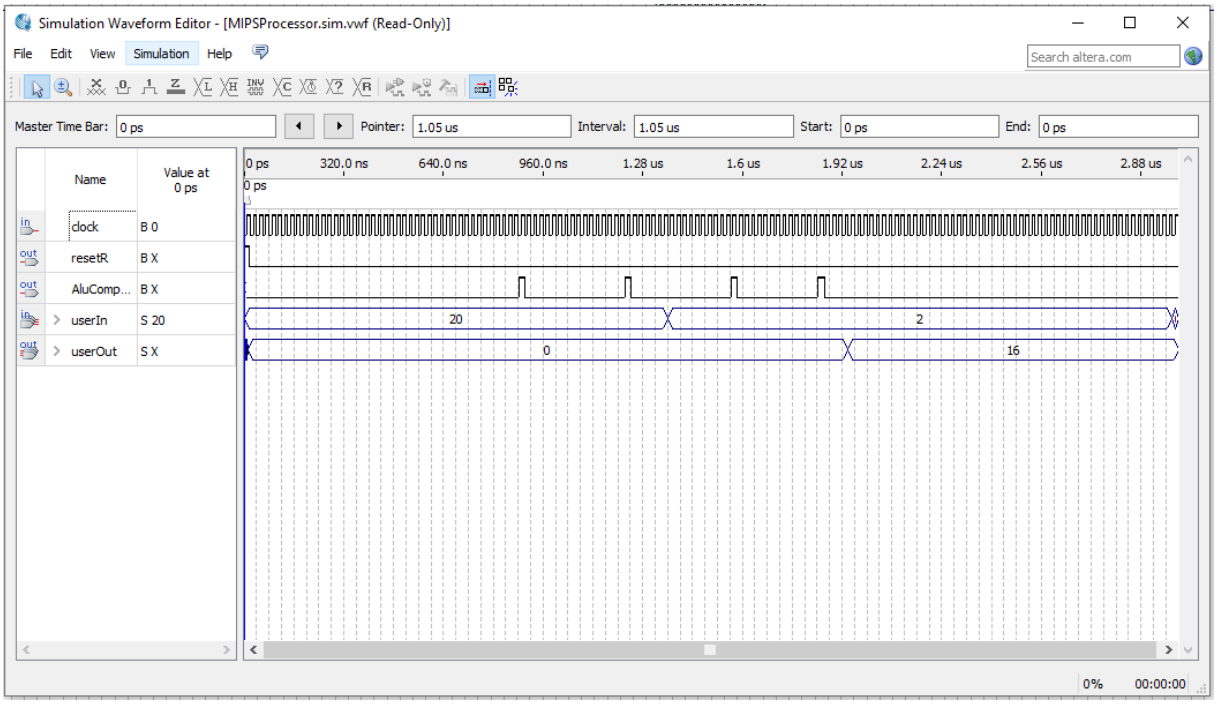

Figure 1.3: MIPS Processor Results

Chapter 2

Component details

This section will give some details about the components that make the processor

2.1 Registers

All registers in this project (Interface Registers, PC register, Out register, ...) have an asynchronous reset, a Write Enable, a clock, an input, and an output. All of them are sensible to the rising edge of the clock, when they get updated if write control bit is enabled

2.2 Instruction Memory

Like registers, it has an asynchronous reset, is sensible to the rising edge of the clock, when the register indicated by the address input get updated if the write enable control bit is enabled. Differently from other register banks, this one reads from the address sent in Address In immediately, this was made to be able to execute the IF step in one cycle. In this project, the instruction memory is read only and cannot be written, thus the clock, the Data In, and the Write Enable are forced to GND.

2.3 Cache

Like Instruction Memory, it has an asynchronous reset, is sensible to the rising edge of the clock, when the register indicated by the address input get updated if the write enable control bit is enabled, and the output is updated when the Address in changes. All registers are initialized with 0, and the register in the 0 address (aka $ZERO) is write-protected by hardware.

2.4 Signal extend

The signal extend is an asynchronous component that extends its input from 6 bits, to 16 bits. This component has a mode input, that decides whether it will make a zero extend (filling the MSB as 0 if mode is '0'), or a sign extend (filling as '1' if the 6th bit is 1, keeping the sign, if mode is '1').

2.5 ALU

The ALU is another asynchronous component that receives 2 numbers and 1 mode, and outputs AluRes as result, and AluComp as compare (if applicable). The following table describes every operation possible in the ALU.

ALU operation Table			
Operation	ID (binary)	Result	Compare
ADD	001	A + B	0
NOR	010	A NOR B	0
AND	011	A AND B	0
Equals	100	0	1 if A == B, 0 otherwise
Greater than	101	0	1 if A > B, 0 otherwise
NOP	others	0	0

2.6 RAM

Like Instruction Memory, it has an asynchronous reset, is sensible to the rising edge of the clock, when the register indicated by the address input get updated if the write enable control bit is enabled, and the output is updated when the Address in changes. To not use all the board's resources only the 2 LSBs are kept from the Extended Last Segment, creating 2 registers (easy to modify in code)

2.7 Controller Unit

The pipeline processor has 3 control units : the Hazard Unit, the Forwarding Unit, and the Main Controller

2.7.1 Hazard Detection Unit

The hazard unit detects if a stall is necessary or not. Since the processor has forwarding and other components to avoid hazards, the only problem that can occur is then ID has a LOAD instruction, and IF instruction wants to read from the register that LOAD will write. The problem happens because the LOAD data will only by available in WB,

thus 1 cycle of stalling is required to avoid data hazards. Any other combination of instructions can be solved with branch prediction or forwarding.

2.7.2 Forwarding Unit

The forwarding Unit is very simple, it checks if the EX instruction reads from R1 or R2 (or both), and if the MEM instruction wants to write on that same address (Cache WriteEn control signal passed by the controller must be set (This connection was omitted to avoid too much signal crossing)), then the mux selector will be set to "1", and ALU will receive the updated value (even though the value in memory is outdated). This logic is then applied to WB, but since MEM has the most updated value, MEM has priority over WB.

2.8 Main Controller

Now the last and most important component, the controller. It's job is to set every control signal in the processor so that the processor does the correct instruction. The first state is the Reset, that resets every memory element to 0, and the case of Instruction Memory, inserting the instructions. After resetting everything, it goes to Run state, where it behaves as a traditional combinational circuit, receiving its inputs, and updating the control signals. Since the pipeline controller has to decide everything in only one state (because it will receive another instruction the next cycle), and the control in pipeline is made in a distributed manner, the FSM in this case is way simpler than the multi-cycle variant

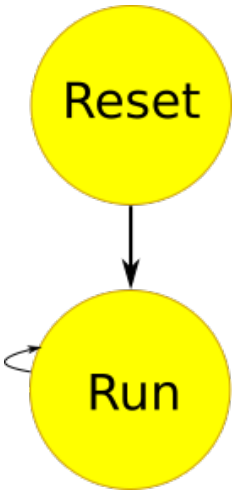

Figure 2.1: Controller