Icon

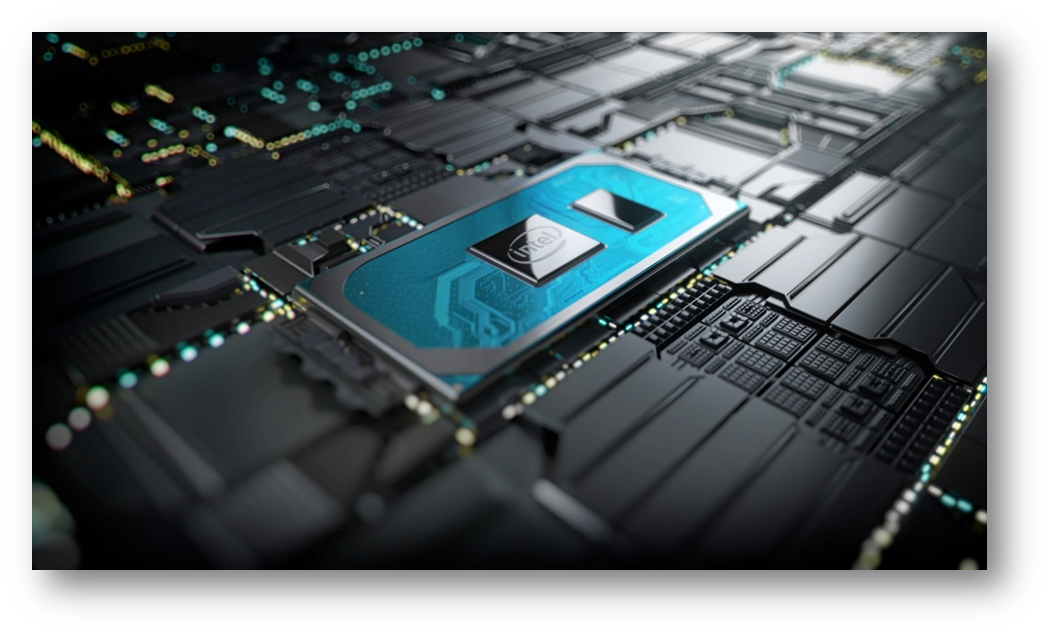
Description automatically generated

Technical University of Cluj Napoca

Beata Keresztes

Structure of Computer Systems Project

2021



MIPS Pipeline

Pipelined CPU

Table of contents:

1. [**Introduction**](#_Introduction) **2**
   1. [Context](#_Context) 2

* 1. [Specifications](#_Specifications) 2
  2. [Objectives](#_Objectives) 2

1. [**Bibliographic study**](#_Bibliographic_study) **3**
2. **Analysis 12**
3. **Design**
4. **Implementation**
5. **Testing and Validation**
6. **Conclusions**
7. **Bibliography**

# Introduction

## Context

The aim of this project is to implement a Central Processing Unit based on a pipelined architecture. Pipelining exploits the Instruction Level Parallelism, by allowing the execution of multiple instructions in parallel, thus reducing the average execution time per instruction (CPI: Clocks per Instruction ~ 1).

As a result, pipelining decreases the clock cycle time:

***CPU time*** = = \* \*

This RISC type processor can execute in a fast and efficient way a total set of 26 instructions, and it can be used to perform some simple computations or execute a shorter code section written based on the available instruction set. It can be then integrated as the control unit in a device allowing it to process and store data, produce control signals, communicate with other peripherals, etc.

## Specifications

The simulation of the working mode of the CPU will be done in Vivado, and it will be programmed and tested on a Basys3 board.

The 32 bits RISC processor’s instructions have a fixed length, and each instruction is represented on exactly 32 bits (4 bytes).

The processor has to support at least 20 instructions, and as the R-type instructions have an opcode zero (0), they are differentiated only by the function bits, we need at least 4 bits for the opcode and function bits. The 32 bits RISC processor has 6 bits for the opcode, meaning that it can support maximum (I and J type instr) + (R-type instr) = (2^6– 1) + 2^6 = 127 instructions.

This processor has a 3 address (load & store) ISA, so the operands of the operations will be registers. All operations are performed on integer values.

The registers contain values on 32 bits. The register file contains 2^5 = 32 registers, numbered R0-R31.

Register R0 always contains value 0, while register R31 is used to save the return address after a jump.

The address of the data memory is also represented on 32 bits, and each slot can store a value on 32 bits.

The instruction format for the 3 types of instructions is the following:

1. R-type:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31-26 | 25-21 | 20-16 | 15-11 | 10-6 | 5-0 |
| 00000 | rs | rt | rd | sh | function |

1. I-type:

|  |  |  |  |
| --- | --- | --- | --- |
| 31-26 | 25-21 | 20-16 | 15-0 |
| opcode | rs | rt | imm |

1. J-type:

|  |  |
| --- | --- |
| 31-26 | 25-0 |
| opcode | target address |

This processor will be able to execute the basic set of instructions plus some more complex instructions (26 in total):

Arithmetic & Logical instructions:

* NOP
* NAND
* ORI
* SUB
* ADDI
* SLLV
* SRLV
* ROL
* ROR
* SLTI

Data memory and register file access instructions:

* LWR
* LWRD
* LB
* SB
* LWPI
* SWPI
* LUI
* MOV
* ADDM
* SWAPM

Branch (conditional jump) instructions:

* BLTZAL
* BEZR
* BGT

Unconditional jump instructions:

* JAL
* JR
* JALR

The supported instructions are:

|  |  |  |
| --- | --- | --- |
| Type | Instruction | Description |
| R type | NOP | No operation. It will not modify any data in the registers or memory. It is represented as OR between 0 and 0.  RTL abstract: RF[0] <- RF[0] | RF[0] ; PC <- PC + 1  ALU op: |  Syntax: nop  Instruction encoding: b”000000\_00000\_00000\_00000\_00000 \_000000”  Example:  Nop  RF[0] <- RF[0] | RF[0]  b”000000\_00000\_00000\_00000\_00000\_000000” |
| NAND | Nand between two registers, the result is stored in the third register.  RTL abstract: RFrd] <- RF[rs] nand RF[rt] ; PC <- PC + 1  ALU op: nand  Syntax: nand rd, rs, rt  Instruction encoding: b”000000\_ sssss \_tttttt \_ddddd\_00000\_000001”  Example:  Nand r3,r1,r2  RF[r3] <- RF[r1] nand RF[r2]  b”000000\_00001\_00010\_00011\_00000\_000001” |
| SUB | Subtract the content of two registers, the result is stored in the third register.  RTL abstract: RFrd] <- RF[rs] - RF[rt] ; PC <- PC + 1  ALU op: -  Syntax: or rd, rs, rt  Instruction encoding: b”000000\_ sssss \_tttttt \_ddddd\_00000\_000010”  Example:  Sub r3,r1,r2  RF[r3] <- RF[r1] - RF[r2]  b”000000\_00001\_00010\_00011\_00000\_000010” |
| LWR | Load word from memory whose address is given by the sum of 2 registers.  RTL abstract: RF[rd] <- M[RF[rs] + RF[rt]]; PC <- PC + 1  ALU op: +  Syntax: lwr rd, rs, rt  Instruction encoding: b”000000\_sssss \_ttttt\_ddddd\_00000 \_000011”  Example:  lwr r1, r2, r3  RF[r1] <- M[RF[r2] + RF[r3]]  b”000000\_00001\_00000\_00001\_00000\_000011” |
| LWRD | Load word register difference: Load word from memory whose address is given by the difference of 2 registers.  RTL abstract: RF[rd] <- M[RF[rs] - RF[rt]]; PC <- PC + 1  ALU op: -  Syntax: lwrd rd, rs, rt  Instruction encoding: b”000000\_ sssss\_ttttt\_ddddd\_00000\_000101”  Example:  lwrd r1, r2, r3  RF[r1] <- M[RF[r2] - RF[r3]]  b”000000\_00001\_00000\_00001\_00000\_000101” |
| MOV | Move the content of one register into another. The value of the register rt is transferred through the ALU by an addition with 0. Rs register has value 0.  RTL abstract: RF[rd] <- RF[rt]; PC <- PC + 1  ALU op: pass B  Syntax: mov rd, rt  Instruction encoding: b”000000\_00000 \_ttttt\_ddddd\_00000 \_001001”  Example:  Mov r1,r2,r3  RF[r1] <- RF[r2]  b”000000\_00000\_00010\_00011\_00000\_001001” |
| SLLV | Shift left logical the content of a register, with a variable amount, indicated by the least significant 5 bits of the second register, and save the result in a third register. Zeros are shifted in.  RTL abstract: RF[rd] <- RF[rt] << RF[rs]; PC <- PC + 1  ALU op: <<  Syntax: sllv rd, rt, rs  Instruction encoding: b”000000\_sssss\_ttttt\_ddddd\_00000 \_001000”  Example:  Sllv r3,r2,r1  RF[r3] <- RF[r2] << RF[r1]  b”000000\_00001\_00010\_00011\_00000\_001000” |
| SRLV | Shift right logical the content of a register, with a variable amount, indicated by the least significant 5 bits of the second register, and save the result in a third register. Zeros are shifted in.  RTL abstract: RF[rd] <- RF[rt] >> RF[rs]; PC <- PC + 1  ALU op: >>  Syntax: srlv rd,rt,rs  Instruction encoding: b”000000\_sssss\_ttttt\_ddddd\_00000 \_001001”  Example:  Srlv r3,r2,r1  RF[r3] <- RF[r2] >> RF[r1]  b”000000\_00001\_00010\_00011\_00000\_001001” |
| BEZR | Execute the branch when the content of the register rs is equal to 0. The address of the jump is given in the register rt.  RTL abstract: if RF[rs] = 0 then PC <- PC + RF[rt];  Else PC <- PC + 1  ALU op: - (Subtract the value of the register rs from 0 and check if the result is negative)  Syntax: bezr rs, rt  Instruction encoding: b”000000\_sssss\_ttttt\_ddddd\_00000\_001010”  Example:  Bezr r1, r2  if RF[r1] < 0 then PC <- PC+RF[r2];  Else PC <- PC + 1  b”000000\_00001\_00010\_00000\_00000\_001010” |
| JR | Jump to the address stored in register rs.  RTL abstract:  PC <- PC[31:26] | RF[rs];  Instruction encoding: b”000000\_ sssss\_00000\_00000\_00000\_001011”  Jr rs  Example:  Jr r3  PC <- PC[31:26] | RF[r3]  b”000000\_00001\_00000\_00000\_00000\_001011” |
| JALR | The target address is given by the source register. Store the next address in register R31, before executing the jump.  RTL abstract:  RF[31] <- PC + 2 (Jump is evaluated in the ID stage, so there is already another instr that has entered the pipeline)  PC <- PC[31:26] | RF[rs]  Syntax: jalr rs  Instruction encoding: b”000000\_ssssss\_00000\_00000\_00000\_001100”  Jal target  Example:  Jal r1  RF[31] <- PC + 2  PC <- PC[31:26] | RF[rs]  b”000000\_00001\_00000\_00000\_00000\_001100” |
| I type | ORI | Or between a register and an immediate value, the result is stored in the third register. Zero extend the value of the immediate.  RTL abstract: RFrd] <- RF[rs] | Zero\_ext(imm); PC <- PC +1  ALU op: |  Syntax: ori rt, rs, imm  Instruction encoding: b”000001\_sssss \_ttttt \_iiii\_iiii\_iiii\_iiii”  Example:  Ori r2,r1, 3  RF[r2] <- RF[r1] | Zero\_ext(3)  b”000001\_00001\_00010\_0000\_0000\_0000\_0011” |
| ADDI | Add a register and an immediate value, the result is stored in the third register. Sign extend the value of the immediate.  RTL abstract: RF[rt] <- RF[rs] + Sign\_ext(imm); PC <- PC +1  ALU op: +  Syntax: addi rt, rs, imm  Instruction encoding: b”000010\_ sssss \_tttttt \_iiii\_iiii\_iiii\_iiii”  Example:  Addi r2,r1, 3  RF[r2] <- RF[r1] + Sign\_ext(3)  b”000010\_00001\_00010\_0000\_0000\_0000\_0011” |
| LB | Load byte: load the least significant byte from memory location.  RTL abstract: RF[rd] <- M[RF[rs] +Sign\_Ext(imm)]&0x000000FF; PC <- PC + 1  ALU op: +  Syntax: lb rt, rs, imm  Instruction encoding: b”000011\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  lb r2,r1,3  RF[r2] <- M[RF[r1] + Sign\_Ext(3)] &0x000000FF;  b”000011\_00001\_00010\_0000\_0000\_0000\_0011” |
| SB | Store byte: store the least significant byte from register to memory location.  RTL abstract: M[RF[rs] +Sign\_Ext(imm)]<- RF[rt] & 0x000000FF; PC <- PC + 1  ALU op: +  Syntax: sb rt, rs, imm  Instruction encoding: b”000100\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  sb r2,r1,3  M[RF[r1] + Sign\_Ext(3)] <- RF[r2] & 0x000000FF;  b”000100\_00001\_00010\_0000\_0000\_0000\_0011” |
| BGT | Branch on greater than: Execute branch if the value of the first register is larger than the value of the second.  RTL abstract: if RF[rs] > RF[rt] then PC <- PC + Sign\_ext(imm)  Else PC <- PC + 1  ALU op: - (Subtract the value of the 2 registers and check if the result is positive)  Syntax: bgt rs, rt, offset  Instruction encoding: b”000101\_sssss\_ttttt\_ iiii\_iiii\_iiii\_iiii”  Example:  Bgtz r1, r2, 3  if RF[r1] > RF[r2] then PC <- PC + Sign\_ext(3)  else PC <- PC + 1  b”000101\_00001\_00010\_0000000000000011” |
| BLTZAL | Branch on less than 0 and link. Store the next address of PC to register R31, before evaluating the condition and executing the jump.  RTL abstract:  RF[r31] <- PC + 8 (Branch is evaluated in the ID stage, so there is already another instr that has entered the pipeline)  if RF[rs] < 0 then PC <- PC + Sign\_ext(imm);  else PC <- PC + 1;  ALU op: - (check the sign bit: if MSB[RF[rs]] == 1 then it is negative nr)  Syntax: bltzal rs, offset  Instruction encoding: b”000110\_sssss\_00000\_iiii\_iiii\_iiii\_iiii”  Example:  Bltzal r1, 3  RF[r31] <- PC + 1  if RF[r1] >= 0 then PC <- PC + Sign\_Ext(3)  else PC <- PC + 1  b”000110\_00001\_00000\_0000\_0000\_0000\_0011” |
| LUI | Load upper immediate: the immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.  RTL abstract: RF[rt] <- Sign\_ext(imm) << 16; PC <- PC + 1  ALU op : <<  Syntax: lui rt, imm  Instruction encoding: b”000111\_00000\_ttttt\_\_iiii\_iiii\_iiii\_iiii”  Example:  Lui r1,2  RF[r1] <- Sign\_ext(2) << 16  b”000111\_00000\_00001\_0000\_0000\_0000\_0010” |
| ROL | Rotate left: rotate to the left the value in register rs by the amount specified in the immediate, and save the result in the second register, rt. The value of the immediate is zero extended as we cannot rotate by a negative amount.  RTL abstract: RF[rt] <- RF[rs] rol Sign\_ext(imm); PC <- PC + 1  ALU op : rol  Syntax: rol rt, rs, imm  Instruction encoding: b”001000\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  Rol r2,r1,3  RF[r2] <- RF[r1] rol Zero\_ext(3);  b”001000\_00001\_00010\_0000\_0000\_0000\_0011” |
| ROR | Rotate right: rotate to the right the value in register rs by the amount specified in the immediate, and save the result in the second register, rt. The value of the immediate is zero extended as we cannot rotate by a negative amount.  RTL abstract: RF[rt] <- RF[rs] ror Sign\_ext(imm); PC <- PC + 1  ALU op : ror  Syntax: ror rt, rs, imm  Instruction encoding: b”001001\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  Ror r2,r1,3  RF[r2] <- RF[r1] ror Zero\_ext(3);  b”001001\_00001\_00010\_0000\_0000\_0000\_0011” |
| LWPI | Load word with post increment: load word from memory to the destination register, then post increment the value in the register.  RTL abstract: RF[rt] <-Mem[RF[rs] + Sign\_ext(imm)] + 1; PC <- PC + 1  ALU op : +  Syntax: lwpi rt, rs, imm  Instruction encoding: b”001010\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  Lwpi r2,r1,3  RF[r2] <-Mem[RF[r1] + Sign\_ext(3)] + 1  b”001010\_00001\_00010\_0000\_0000\_0000\_0011” |
| SWPI | Store word with post increment: store word from register to memory, then post increment the value in the source register.  RTL abstract:  Mem[RF[rs] + Sign\_ext(imm)] <- RF[rt];  RF[rt] <- RF[rt] + 1;  PC <- PC + 4  ALU op : +  Syntax: lwpi rt, rs, imm  Instruction encoding: b”001011\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  Lwpi r2,r1,3  Mem[RF[r1] + Sign\_ext(3)] <- RF[r2];  RF[r2] <- RF[r2] + 1;  b”001011\_00001\_00010\_0000\_0000\_0000\_0011” |
| SLTI | Set on less than immediate: compare the value of a register and an immediate value and set the destination register to 1, if it is smaller than the immediate, 0 otherwise.  if RF[rs] < imm then RF[rt] <- 1  else RF[r31] <- 0;  PC <- PC + 1  ALU op : - (Subtract the 2 values and compare the result to 0)  Syntax: slti rs, rt, imm  Instruction encoding: b”001100\_sssss\_ttttt\_ iiii\_iiii\_iiii\_iiii”  Example:  Slti r1,r2,3  if RF[r1] < 3 then RF[r2] <- 1  else RF[r2] <- 0;  b”001100\_00001\_00010\_0000\_0000\_0000\_0011” |
| SWAPM | Swap the content of a register and a value in memory.  RTL abstract: M[RF[rs] + Sign\_ext(imm)] <-> RF[rt]; PC <- PC + 1  ALU op: + (For the address calculation)  Syntax: swapm rt, rs, imm  Instruction encoding: b”001101\_sssss\_ttttt\_iiii\_iiii\_iiii\_iiii”  Example:  Swapm r2,r1,3  M[RF[r1 + Sign\_ext(3)<<2] <-> RF[r2]  b”001101\_00001\_00010\_0000\_0000\_0000\_0011” |
| ADDM | Add the content of a register to a value in memory and store the result in the register.  RTL abstract: RF[rt] <- RF[rt] + M[RF[rs] + Sign\_ext(imm)]; PC <- PC + 1  ALU op: + (For the address calculation)  Syntax: addm rt, rs, imm  Instruction encoding: b”001110\_sssss\_ttttt\_”  Example:  Addm r2,r1,3  RF[r2] <- RF[r2] + M[RF[r1] + Sign\_ext(3)]  b”001110\_00001\_00010\_0000\_0000\_0000\_0011” |
| J type | JAL | Jump to the given address and store the return address in register R31.  RTL abstract:  RF[31] <- PC + 2 (there is already another instr in the pipeline, jump is executed in the ID stage).  PC <- PC[31:26] | target  Syntax: jal target  Instruction encoding: b”001111\_ iiii\_iiii\_iiii\_iiii\_iiii\_iiii\_ii”  Example:  Jal 3  RF[31] <- PC + 2  PC <- PC[31:26] | 3  b”001111\_0000\_0000\_0000\_0000\_0000\_0000\_11” |

[Back to the list of instruction](#instructions_list)

The corresponding codes for the ALU operations:

A picture containing table

Description automatically generated

For each instruction the following opcode, function bits (for R-type instructions) and the corresponding ALU operation is associated:

Table

Description automatically generated with medium confidence

## Objectives

Design and implement a CPU that can execute multiple instructions using a pipeline mechanism.

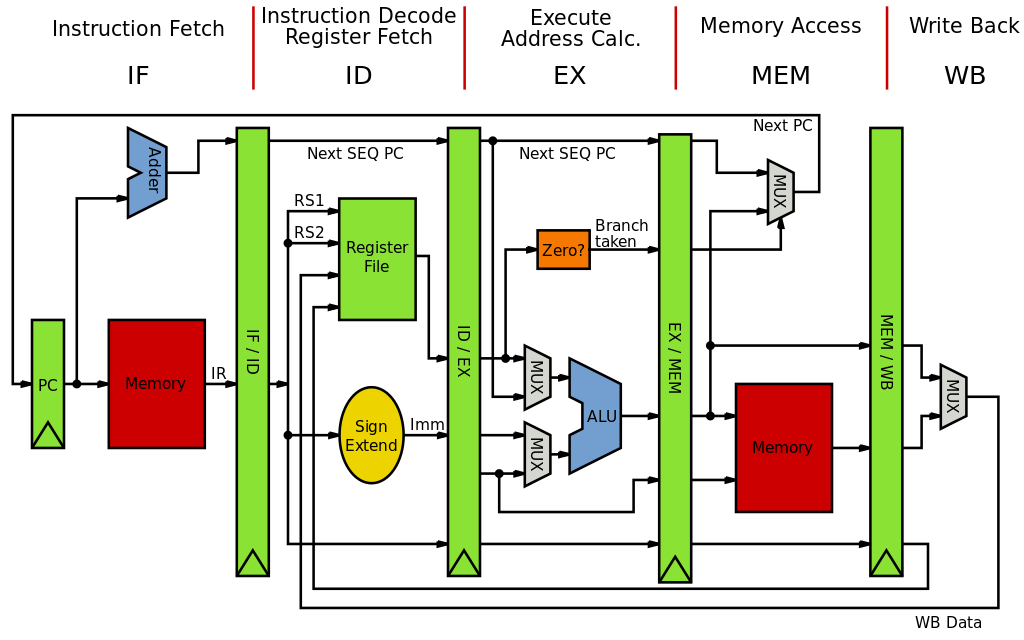
Create a short code using the implemented instructions to demonstrate the working mode of the processor.

For ex. A sorting algorithm, such as bubble sort.

Design an interface for the user to be able to test the instructions. Allow the user to introduce the value of the operands, in a BCD format, using the switches available on the development board. Display the result of the computations on the 7 segment display of the board, also in BCD format, the content of the registers/data memory, or the current instruction in each step of the pipeline (The user selects the data to be viewed based on the first 3 switches). Show either the control signals or the instruction bits on the leds. By pressing one of the buttons from the board the user can control the execution of the instructions, move one step in the pipeline, then check the content of the registers/control signals.

# Bibliographic study

In a pipelined execution the instruction is broken up into smaller steps, called “stages”, which take only a small fraction of the time needed to execute the whole instruction. These stages are connected between each other such that the output of one stage is the input to the next one. The communication between the stages is achieved using intermediate registers. We insert a register between each pair of pipe stages. The registers transfer data values and control signals from one stage to the next.



For integer instructions the optimal number of pipeline stages is 5. This means that 5 instructions will be executed in each clock cycle simultaneously.

A picture containing text, clock

Description automatically generated

Table

Description automatically generated

The five stages are:

* **Instruction Fetch (IF):** fetches the current instruction from memory, increment PC.
* **Instruction Decode and Operand Fetch (ID&OF):** translate the opcode into control signals and read registers.
* **Execute (EX):** execute ALU operations, compute memory address, branch address.
* **Memory (MEM):** access data memory (load/store instruction), read or write to the mem address computed in the previous stage.
* **Write Back (WB):** update register file (write back the result to the destination register, for R-type instructions or Load).

Unlike in case of the multi-cycle MIPS, where each instruction can have variable clock cycles, in the Pipeline architecture each instruction takes exactly 5 clock cycles until it can exit the pipeline.

The more stages we have the more the complexity of the CPU increases, and it becomes more difficult to treat the pipeline hazards that may occur.

There are 3 types of pipeline hazards:

* **Structural hazard** (resource dependency): when 2 instructions attempt to use the same resource simultaneously;

*Solution:* We use separate memories for instruction and data to avoid structural hazards when accessing the memory (tries to read data in the Mem stage of a previous instruction and at the same time it tries to fetch the next instruction, using different addresses).

Also, we use separate arithmetic units for computing the branch address and the other ALU operations.

* **Data hazard** (data dependency): an instruction depends on a previous instruction; When the result of a previous instruction has to be used to execute the next one.

*Solution:* In order to avoid data hazards we need to wait until the previous instruction reaches the write back stage then resume with the next instructions. For this we stall the pipeline, by inserting “bubbles” (= No ops), which freeze the pipeline, clearing all the control signals (at least the write signals) in the following stages.

In order to avoid RAW hazard, we write the new value from the WB stage in the registers in the first half of the clock cycle and read the value contained in those registers in the second half of the clock cycle.

* **Control hazards** (flow control): attempting to make a decision about a program control flow before the condition has been evaluated and the new PC address calculated. Relevant for instructions which change the PC, such as branches or jumps.

*Solution:* To eliminate control hazards for the branch instructions, which compute the branch address in the ID stage, we need to stall the pipeline for 1 clock cycle. Otherwise in the next clock cycle, the next instruction after the branch would enter the pipeline, but it should not be executed in case the branch is taken. Other more efficient methods use different branch prediction techniques, but for the sake of simplicity, we only have to insert a “bubble” (= No op) after each branch instruction.

# Analysis

The MIPS pipeline processor has the following basic components:

1. **PC (Program counter):** stores the instruction pointer, in other words the address of the current instruction that is being executed.
2. **Adder:** incrementing the program counter by 1, to generate the next address. The instructions in memory are aligned on 4 bytes each, therefore the pointer to the next instruction can be obtained by adding 1 to the pc.
3. **ALU unit** for arithmetic/logic operations**:** performs ALU operations.
4. **ALU Branch** for address computation: computes the branch address for conditional jumps.
5. **IM (Instruction memory):** stores all the instructions defined for the processor, which can be accessed through the PC.
6. **RF (Register file):** contains the 32 general purpose registers each of 8 bits.
7. **DM (Data memory):** stores the data related to the specific program. Separate from the instruction register, which only stores the instructions.
8. **CU (Control unit):** generates the control signals for each instruction.

The pipeline architecture is achieved through the use of **pipeline registers**, which hold the partial results and realize the communication and data transfer between the stages. We need 4 such intermediate registers to transfer the data between the 5 pipeline stages.

Diagram, schematic

Description automatically generated

In each pipeline register we store the results of the current computation so that it can be used in the next stage. The following signals (values) must be stored in each of the pipeline registers:

Register IF\_ID (instruction fetch & instruction decode):

* + IF\_ID.PC <- PC + 1
  + IF\_ID.IR <- IM[PC]

Register ID\_EX (instruction decode & execution):

* + ID\_EX.PC <- IF\_ID.PC
  + ID\_EX.A <- RF[rs]
  + ID\_EX.B <- RF[rt]
  + ID\_EX.Imm <- Ext(imm)
  + ID\_EX.sa <- sa
  + ID\_EX.RT <- rt
  + ID\_EX.RD <- rd

Register EX\_MEM (execution & memory):

* + EX\_MEM.AluResult <- AluResult
  + EX\_MEM.Zero <- Zero
  + EX\_MEM.Sign <- Sign
  + EX\_MEM.B <- ID\_EX.B
  + EX\_MEM.RegWriteAddr <- ID\_EX.rt when RegDest=’00’ else when RegDest=’01’ ID\_EX.rd else 31.

Register MEM\_WB (memory & write back):

* + MEM\_WB.PC <- EX\_MEM.PC
  + MEM\_WB.MemData <- M[EX\_MEM.AluResult]
  + MEM\_WB.MemDataInc <- M[EX\_MEM.AluResult] + 1
  + MEM\_WB.AddedMemData <- M[EX\_MEM.AluResult] + EX\_MEM.B
  + MEM\_WB.Binc <- EX\_MEM.B + 1
  + MEM\_WB.AluResult <- EX\_MEM.AluResult
  + MEM\_WB.RegWriteAddr <- EX\_MEM.RegWriteAddr

The control signals are also transmitted to each stage until they are used:

Diagram

Description automatically generated

For transmitting the control signals we use separate registers:

* C\_ID\_EX:
  + C\_ID\_EX.EX <- (AluOp, AluSrc, RegDest, ShiftVar)
  + C\_ID\_EX.MEM <- (MemRead, MemWrite, SB, LB)
  + C\_ID\_EX.WB <- ( MemToReg, RegWrite, LinkRetAddr)
* C\_EX\_MEM:
  + C\_EX\_MEM.MEM <- C\_ID\_EX.MEM
  + C\_EX\_MEM.WB <- C\_ID\_EX.WB
* C\_MEM\_WB:
  + C\_MEM\_WB.WB <- C\_EX\_MEM.WB

# Design

## Instruction Fetch stage

The instruction fetch unit is responsible for setting the program counter and reading the next instruction.

It consists of the following components:

* **Program counter:** iterates through the set of instructions.
* **Multiplexer**: for selecting the next address for the program counter in case the program flow is altered by a (un)conditional jump.
* **Instruction memory:** stores the instructions. For each entry there are 32 bits allocated, which means it can hold the whole instruction (it is not split into 4 consecutive addresses, each holding 1 byte). Therefore the instructions are on consecutive addresses, and it is enough to increment the program counter by 1 in order to access the next instruction.
* **Pipeline register IF\_ID:** stores the partial results:
  + Next address of the program counter
  + Current instruction to be executed

Diagram

Description automatically generated

## Instruction Decode stage

The instruction decode stage incorporates the logic for generating the control signals based on the instruction’s opcode and reading the necessary operands from the Register File.

It also is contains the branching unit, which evaluates the branch condition and computes the branch address.

The main components of the ID stage are:

* **Register File:** contains 32x32 general purpose registers; it has 2 read and 1 write port. The read is asynchronous, while the write is synchronous and it happens in the first half cycle, meaning that on the falling edge of the clock.

The RegWrite signal for updating the value of a register comes from a later pipe stage(WB).

* **Adder for PC:** increment the PC by 1 before passing it forward in the pipeline, so that it can be used by the jump instructions which need to link the next return address to register $31 before executing the jump. This is necessary because by the time, the branch condition is evaluated or the jump address is computed, which happens in the ID stage, there has already entered another instruction in the pipeline, so the return address would be that of the following instruction (which is not yet present in the pipeline).
* Adder for computing the branch address.
* **Branching unit:** consists of a comparator, which compares the values read from the register file and evaluates the branch condition.
* **Sign or Zero Extender:** for computing the immediate value. Sign extend is used for arithmetic operations and zero extend for logic operations.
* **Control unit:** based on the opcode and the func bits, generates and sets the control signals specific for each stage.

Diagram, schematic

Description automatically generated

The branch evaluation and the branch address computation are realized in the ID stage, when the jump address is also calculated. This means that 1 instruction would enter the pipeline until the jump is realized/branch is taken, therefore when we need to save the return address of the next instruction (Jal, Jalr, Bltzal), instead of storing PC+1 we would store PC+2, to point to the next instruction which has not yet entered the pipeline.

The jump address is computed by appending it to the first 6 bits of the PC. It is not needed to shift the target address by 2, because the instructions in memory are allocated on 4 bytes each, and they can be accessed on consecutive addresses. That means, instead of incrementing the address by 4, to jump to the next address, we only have to increment it by 1, therefore the last 2 bits are not necessarily 0.

The branching unit is responsible for the branch evaluation, it takes the 3 branch signals, and the values of the 2 registers: rs and rt, and compares them (by subtraction), in order to determine if the branch should be taken or not. This computation is separate from the other arithmetic-logic operations performed by the ALU.

The branching logic is the following:

(We have defined a separate signal for each branch instruction: Bgt, Bezr and Bltzal)

Diagram

Description automatically generated

The following control signals are derived from the opcode of each instruction:

Diagram

Description automatically generated

Control signals for each instruction:

A picture containing text, crossword puzzle

Description automatically generated

Remark: Instead of the ‘X’ (don’t care) the actual value of the signals will be ‘0’.

## Execution stage

The execution stage contains the **ALU unit**, and it is responsible for performing the arithmetic-logic operations specific to each instruction.

The ALU unit receives the AluOp opcode which determines the type of operation to be performed. This signal is generated in the Control Unit along with the other control signals, there is no separate control unit defined for the Alu.

The destination register, where to write the result, is also determined at this stage, and the final address is forwarded to the next pipeline stage.

Diagram

Description automatically generated

## Memory stage

The main component which belongs to this stage is the Data memory, reading and writing to the memory unit is done at this point. The value read from memory, along with the previous AluResult and the write address of the register file in case of R type instructions are forwarded to the next pipe stage.

Diagram, schematic

Description automatically generated

## Write Back stage

In this stage, the result is written back to the register file in case of R and I type instructions or when the return address must be saved in register $31 before in case of jumps.

It mainly consists of 2 multiplexors which select the data to be written in the register file to the correct address.

Diagram

Description automatically generated

The final RTL schematic, containing the 5 pipeline registers, has the following structure (the rest of the schematic was trimmed, mainly the part related to the selection of which data to be displayed on the SSD):

Diagram, schematic

Description automatically generated

* Black box with I/O signals of the IF stage unit:

A picture containing graphical user interface

Description automatically generated

Internal schematic of the IF stage generated by Vivado:

Diagram, schematic

Description automatically generated

* Black box with I/O signals of the ID stage unit:

Table

Description automatically generated

The internal schematic of the ID stage generated by Vivado is too large to be captured on an image, because it contains many registers for all the control signals and content of the register file that is transmitted to the next pipeline stages.

* Black box with I/O signals of the EX stage unit:

Table

Description automatically generated with low confidence

Internal schematic of the ID stage generated by Vivado:

Diagram, schematic

Description automatically generated

* Black box with I/O signals of the MEM stage unit:

A picture containing table

Description automatically generated

In case of the MEM stage as well, the internal schematic generated by Vivado is too large to be captured on an image, because it contains many registers for all the storing and sending forward in the pipeline the content of the data memory at the requested address or the previous signals that it received, such as the result of the alu operation, the content of the register file at a given address, or the write address of the register file.

* Black box with I/O signals of the MEM stage unit:

A picture containing table

Description automatically generated

Internal schematic of the WB stage generated by Vivado:

Diagram, schematic

Description automatically generated

# Testing

The Pipelined MIPS processor is tested on the previously defined set of instructions. (See Specifications)

Because in this initial architecture the hazard that could occur in a pipelined execution are not treated, the operands of each instruction were chosen in such a way as to avoid hazardous situations during the execution.

The first 16 registers in the register file are initialized with values: (X"0000\_0000", X"0000\_0001", X"0000\_0002", X"0000\_0003", X"0000\_0004", X"0000\_0005", X"0000\_0006", X"0000\_0007", X"0000\_0008", X"0000\_0119", X"0000\_000A", X"0000\_000B", X"0000\_000C", X"0000\_000D", X"0000\_000E", X"0000\_000F", X"0000\_0000", X"0000\_0005", X"0000\_0026", X"0000\_002A”), The rest of the registers contain zero, as they will not be used by the instructions.

The first 16 addresses in the data memory contain the following values initially: (X"0000\_0010", X"0000\_0011", X"0000\_0012", X"0000\_0013", X"0000\_0014", X"0000\_0015", X"0000\_0016", X"0000\_0017", X"0000\_0018", X"0000\_0019", X"0000\_001A”, X"0000\_001B", X"0000\_001C", X"0000\_001D", X"0000\_001E", X"0000\_001F"). The rest of the memory addresses contain values 0, because they will not be accessed during the execution of the instructions.

The following table presents the instructions with their operands and the expected result after their execution:

Table

Description automatically generated

Remark: Between the conditional and unconditional jump instructions I added NOP instructions, as in all test cases the branch was taken, and those instructions were never executed.

The simulation was performed in Vivado, using a Testbench for verifying the correct results, and that each instruction committed their results “in-order” (in the order they were issued).

The simulation waveforms for the instructions are the following:

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated