

# MIPS Datasheet

## Description:

This is a working synthesizable 32-bit instructions MIPS multicycle processor designed for general use for clients to integrate in a variety of products. It is implemented with a memory-mapped bus interface and can perform 48 instructions.

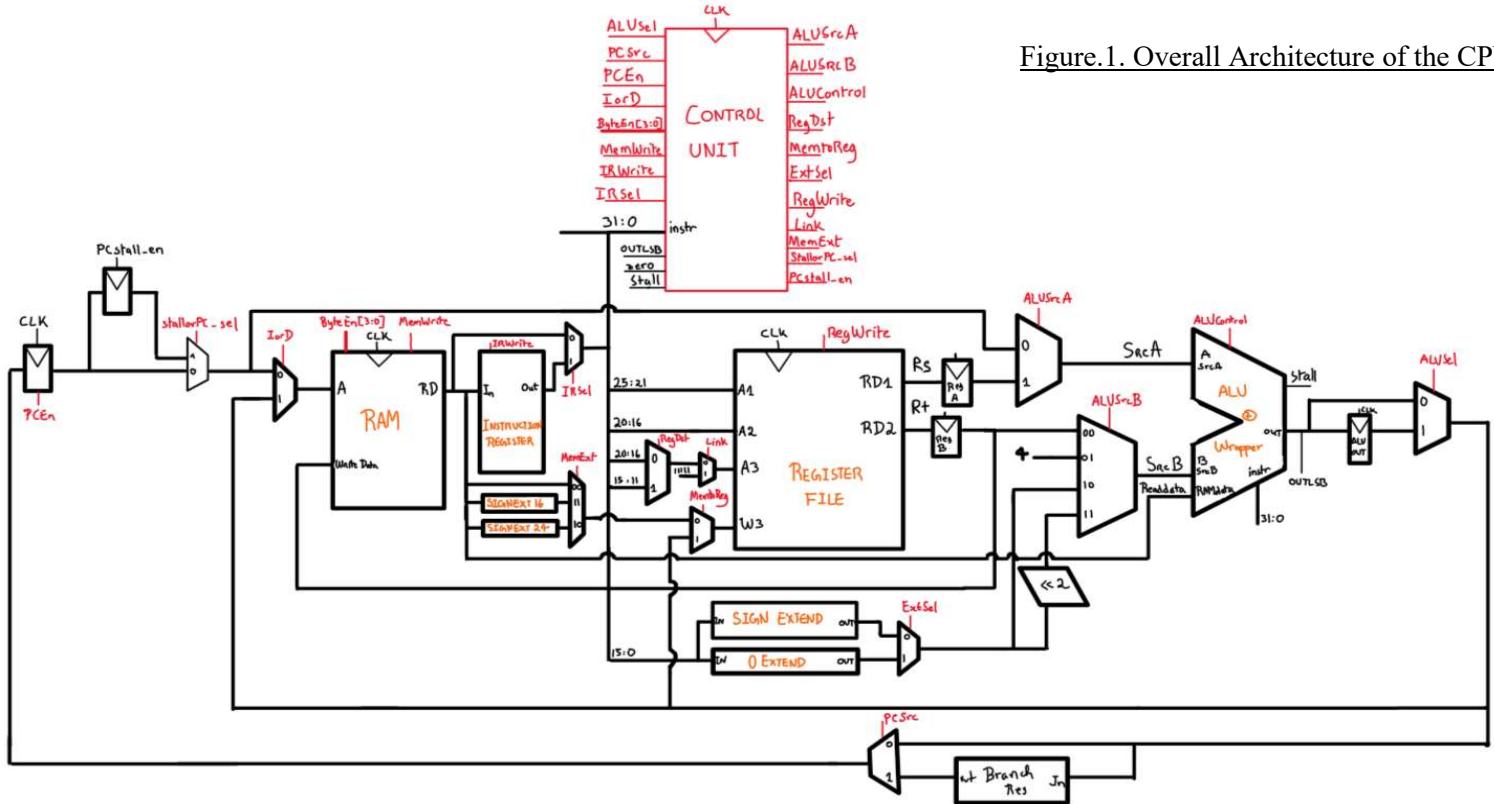


Figure.1. Overall Architecture of the CPU

## Specifications:

- ❖ CPU interface: bus [requires instructions and data to be fetched over the same interface, allows memory to have variable latency]
- ❖ Avalon supported RAM
- ❖ Must successfully execute the 48 wanted instruction
- ❖ All signals are synchronous to clock
- ❖ Includes a reset behavior
  - Must be set high for at least 1 cycle
- ❖ Includes halt behavior
  - Active signal should be held high
- ❖ Must have two special memory locations
  - 0x00000000 : Attempt to execute address 0 causes CPU to halt
  - 0xBFC00000: where execution should start after reset
- ❖ Focus is stressed on functionality rather than performance.

## Featured Elements:

Datapath:

- 1) ALU + wrapper**
  - Execute arithmetic operations and deals with R-type instructions.
- 2) Register File**
  - Holds on to data, reducing data access time by avoiding the RAM
- 3) Instruction Register**
  - Holds onto instructions
- 4) Sign Extend**
  - When an I-type instruction is called, the 16-bit signed offset stored in the immediate field needs to be signed extended for it to interact accordingly with the 32-bit values from the registers

## 5) ALUSrcA Multiplexer

- Selects between Program Counter [PC] instruction and output RD1 of register file.

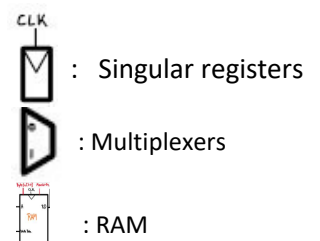
## 6) ALUSrcB Multiplexer

- Selects between the output RD2 of the register file, the value 4, the immediate offset or the immediate offset shifted twice to the left.

## 7) Branch Register

- Holds onto branched address due to delay slot.

Other :



## Control Unit

### Inputs

- ❖ **Instruction Word [31:0]**
- ❖ **OUTLSB:** Least significant bit coming from the output of ALU. This is used when Set On Less instructions are called.
- ❖ **Stall:** Delays the execution by one cycle for multiplying and dividing.

### Outputs:

- ❖ **ALUSrcA/ALUSrcB:** Multiplexer selects SrcA/SrcB.
- ❖ **ALU\_Control:** Tells ALU which arithmetic operation to execute as well as which R-type instruction to execute.
- ❖ **ALUSel:** Skip ALUOut Register or use register.
- ❖ **ByteEn [3:0]:** Selects specific byte lanes: each bit corresponds to a byte in Read Data1 (RD1) /RD2 and Write3 (W3). When writing in RAM, the number of byte indicate the length of the word being written. Uncalled bytes are ignored. Similar process for reading.
- ❖ **ExtSel:** Selects the immediate offset sign extended or zero extended.
- ❖ **lorD:** Selects between the next instruction or the data from the output of the ALU.

- ❖ **IRWrite:** Enables writing in Instruction Register
- ❖ **IRSel:** Selects between output of RAM or output of the instruction register block.
- ❖ **Link:** Can select to bypass the option of Register Destination for the Return Address Register, \$ra. Useful for instruction such as JAL and Branch/link.
- ❖ **MemExt:** Selects between data outputted from RAM, data extended by 16 bits when half loads are called or data extend by 24 bits when a byte is loaded.
- ❖ **MemtoReg:** Selects between data from RAM or output from the ALU as an input to W3.
- ❖ **MemWrite:** Enables Writing in RAM.
- ❖ **PCSrc:** Selects between output of the ALU or the Branch register when branch instructions are called.
- ❖ **PCEn:** Enables register to hold on to the next instruction.
- ❖ **PCorStall\_en:** Selects path with stalled PC or regular path
- ❖ **RegDst:** Selects either [20:16] as input or [15:11]. Depending if it's an R-type or I-type instruction the Destination Register is specified in different parts of the instruction.
- ❖ **RegWrite:** Enables writing in Registers

### Design decisions taken when implementing the CPU:

- The team decided to implement a **multicycle processor**.
  - It is much easier to debug. Furthermore, its control path is faster and more easily pipelined if more speed is desired.
- Implement a **state machine decoder**
  - Decodes each instruction individually apart from R-type which are approached in one category.
- Implement a **full ALU** instead of using specific Verilog arithmetic operations for each instruction
  - A **wrapper** processing specifically R-type instructions was also added around the main ALU to keep better track of the different type of operations.
- Have an **assembler** to convert assembly language to hexadecimal code
- When errors were encountered during testing, minimized changes to the original circuit were favoured.
  - Thus, resulting in **addition of multiplexers** to fix paths that encountered errors and find solutions to repair instructions

### Clever Features:

- Implement state machine in decoder
  - The path for each implemented instruction is straightforward. Having a standard format consisting of 5-cycles makes additional external professional cooperation easier. It facilitates their understanding of the processor and their implementation for further contribution.
  - If the client encounters any bugs or want to include a new instruction the decoder facilitates future debugging processes and the addition of new instructions.
- Sequential multiplier/divider and stall
  - Since it is sequential, the operations will be done faster and the hardware will take less area when manufacturing it.
  - Does take multiple cycles but can work on high frequency clock.
- ALU + wrapper
  - All arithmetic operations are basically performed by the ALU and its wrapper. Bugs revolving arithmetic operations are more rapidly pin pointed.
- Script and C++ file which generates testcases
  - The file intakes a string and generates 7 testcases with proper naming and reference which are automatically place in wanted directories. This allows a more thorough testing on a wide range of random and edge cases.

### Approach taken to testing CPU:

First, the team began by testing the major blocks of the datapath individually. This approach was taken to minimize the probability of failure when testing the whole CPU. It is making sure that each block does what it is supposed to do.

#### 1) Register File testbench approach:

##### **Components tasks (what it needs to do)**

- Need to temporarily hold on to data in wanted register and output when asked from wanted register

##### **Observable outputs to consider the testbench successful**

- RD1 output data read from register Address1 (A1), RD2 output data read from register A2
- A1 and A2 must be ranged between 0 and 31
- Data inputted in W3 must be written in the register Destination A3
- The Testbench incorporated a couple of specific cases to verify its global functionality. If the register file performed as asserted at each clock the test was deemed correct.

#### 2) ALU and its wrapper testbench approach:

##### **Component tasks**

- Combinatorially complete an operation depending on the opcode

##### **Note**

- Multiplier and divider have their own testbench due to complexity
  - Manually verify multiple different specific test cases to verify its overall functionality.

##### **Observable outputs to consider the testbench successful**

- To test the ALU, 3 testcases were written per ALU arithmetic operations.
- Two edge cases and a random test case.
- All the test cases are manually written and if the ALU outputs as the wanted referenced value than the testcase is considered a pass.

#### 3) Testing of the CPU as a whole

##### **Main approach**

As mentioned above, a script and C++ file were written to generate multiple specific and random testcases.

- The C++ file takes an input string and determines whether the instruction is signed or unsigned
  - It will adjust the limits used in testcases and variables inputted in the function depending on the sign.
  - 7 sets of arguments are then generated :
    - 2 sets for the lower limit of the 32-bit signed/unsigned values, 3 randomly generated sets between the upper and lower limit, and, 2 sets for the upper limit.

- A function takes the input instruction and arguments and writes the skeleton for the test-cases and produces the result in the same scope. These are outputted to stdout.
- The script file compiles and echoes the input parameter into the executable and goes through the output file and separates the test-case skeletons and results into their own files with the correct name and directories.
- Instructions that could not conform to the script's structure were manually written by team members.
- The testing is broken down in 5 directories:
  - **0- instructions assembly** : contains all the testcases in assembly language.
  - **1- binary**: contains the assembled testcases in hex generated which are to be inputted into the RAM.
  - **2- simulator**: has the output executable file from compiling all the files together.
  - **3- output**: has .stdout file which is the raw output from the executable cpu but also contains .out file which is the extracted output lines from .stdout that are going to be compared to the reference file
  - **4- reference**: has the expected output for the testcases.

The .out file is compared to the reference file to verify the functionality of the CPU regarding each instruction such as correct read/writing in RAM. Having a testing approach broken down in 5 directories facilitates testing / debugging.

### Area and Timing Summary for the "Cyclone IV E 'Auto' variant in Quartus

#### Area

**Fitter Status:** Successful – Sun Dec 20 15:01:31 2020

**Quartus Prime Version:** 16.0.0 Build 211 04/27/2016 SJ Lite Edition

<b>Revision Name:</b>	mips
<b>Top-level Entity Name:</b>	mips_cpu_bus
<b>Family:</b>	Cyclone IV E
<b>Device:</b>	EP4CE10F17C6
<b>Timing Models:</b>	Final
<b>Total logic elements:</b>	4,113/10,320 (40%)
<b>Total combinational functions:</b>	4,106/10,320 ( <b>40%</b> )
<b>Dedicated logic registers</b>	1,225/10,320 ( <b>12%</b> )
<b>Total Registers:</b>	1225
<b>Total pins:</b>	138/180 (77%)
<b>Total virtual pins:</b>	0
<b>Total memory bits:</b>	0/423,936 (0%)
<b>Embedded Multiplier 9-bit elements:</b>	0/46 (0%)
<b>Total PLLs:</b>	0/2 (0%)

#### Timing

**Fmax:** 37.6MHz

**Restricted Fmax:** 37.6Mhz

**Clock Name:** clk

# TESTING PROCESS DIAGRAM

