**Team Blue CPU**

Table of Contents

[Credits 3](#_Toc512377421)

[MIPS-CPU 4](#_Toc512377422)

[System Requirements 4](#_Toc512377423)

[Getting the Cosimulation to Run on Windows 4](#_Toc512377424)

[Installing Python myhdl 5](#_Toc512377425)

[Building and installing myhdl.vpi 5](#_Toc512377426)

[Generating an input file 5](#_Toc512377427)

[Random Generation 5](#_Toc512377428)

[Using the assembler 6](#_Toc512377429)

[Running the cosimulation 6](#_Toc512377430)

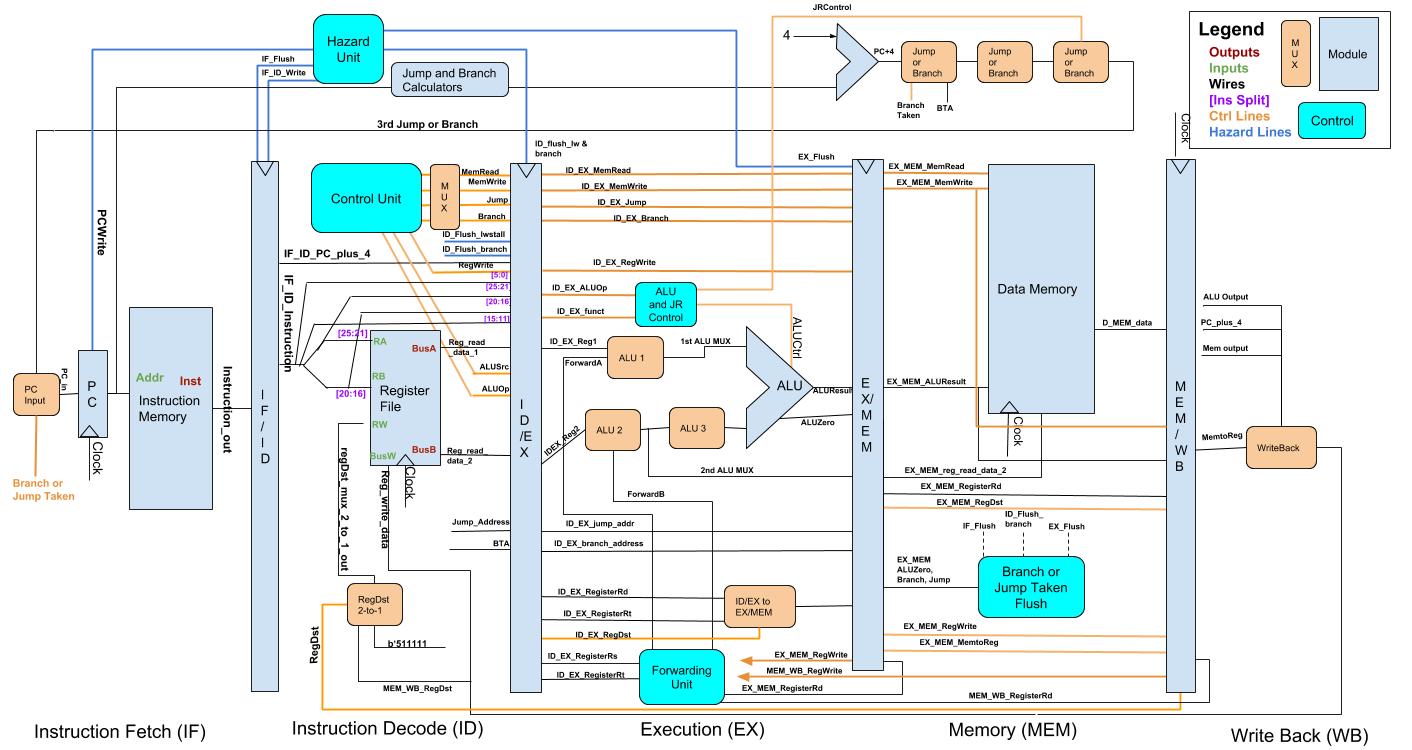
[Project Structure 7](#_Toc512377431)

[Folder Conventions 7](#_Toc512377432)

[File Conventions 7](#_Toc512377433)

[Design Overview 8](#_Toc512377434)

[Instruction Opcode Dissection 9](#_Toc512377435)

[ 10](#_Toc512377436)

[Stages 11](#_Toc512377437)

[Pipeline Stages 12](#_Toc512377438)

[Multiplexers for routing data and control signals 12](#_Toc512377439)

[Hazard and Forwarding 14](#_Toc512377440)

[5    Jump and Branch Address Calculators 15](#_Toc512377441)

## Credits

This program was built by the Texas State PC Architecture Blue Team.

**Team Lead:** Michael Volling

|  |  |
| --- | --- |
| **Verilog Team**   * Boris Sabotinov **Lead** * Daniel Le * Cody Sears | **Python Team**   * Kevin Maxey **Lead** * Matthew Maluschka * Erich McLaurin |

# MIPS-CPU

A basic implementation of a simplified MIPS CPU with pipelining. Built using verilog and python. Our project simulates the working of a processor using both Verilog and Python implementations. We are using signals to simulate the functions of wires. We are using Python to generate a clock for both the Python and Verilog simulation and synchronizing them together while verifying their outputs are correct. Components have a block for defining the expected inputs and an @always statement which tells the component when it should execute its code. Each component also has a testbench which uses a clock and random signal generators to simulate values it might receive and verify that it is working correctly. We are usually running a minimum of 1000 simulations for each component.

## System Requirements

In order to build and run the project the computer needs to be running Linux with the following programs installed:

* iverilog
* Python3 versions 3.4 through 3.6 (myhdl doesn't work with 3.7 and above)
* Python library myhdl version 10
* A built myhdl.vpi (See below for building instructions)

## Getting the Cosimulation to Run on Windows

As the cosimulation depends on the behavior of Linux, it is impossible to run the cosimulation directly on Windows. Luckily, a workaround is available in the "[Windows Subsystem For Linux](https://docs.microsoft.com/en-us/windows/wsl/install-win10)". The link will walk you through the installation. Note that this only works for Windows 10.

Trying to use the cygwin/mingw toolkit will not work as the OS features are not emulated.

## Installing Python myhdl

Run the following commands to build the myhdl library:

sudo apt-get install python3-pip

sudo pip3 install myhdl

## Building and installing myhdl.vpi

myhdl is the python library we are using to connect the verilog code to the python code. It requires a compiled component to connect to the verilog model. Running ./buildmyhdl.sh will automatically download, build, and copy myhdl.vpi to the bin directory.

## Generating an input file

### Random Generation

Input files can be generated with random instructions via the tools/Generator.py program. This generation is configured through editing the tools/gen.conf file. These are the changable parameters:

* "i\_count: n" sets the number of generated instructions to be n.
* "pf\_p: f" sets the percentage of program flow assignments, f must be in [0:1]
* "mem\_p: f" sets the percentage of memory access functions, f must be in [0:1]

ALU operations are supplied to fill in all operations that are not program flow or memory operations.

The generator can be called with:

cd tools

python3 Generator output.hex

### Using the assembler

The tools/Assembler.py python programs works if you already have a given program's source code. Note that there is only .text generation and no .data.

The assembler can be called with:

cd tools

python3 Assembler.py [-o out\_file] infile.S

## Running the cosimulation

The cosimulation is compiled and run through a single command "python3 CPU\_Cosim.py <instruction\_file.hex>". This will launch the cosimulation with a command line prompt that accepts the following commands:

* run: Runs the simulation for n ticks. There are 20 ticks per clock cycle.
* show: Prints out both register files and tests for equality.
* quit: Exits the program, terminating the simulation.

## Project Structure

### Folder Conventions

Files are stored in this tree structure as follows:

.

├── bin Stores compiled files

├── lib Stores external libraries

│   └── myhdl myhdl libary, used to compile myhdl.vpi

├── python Stores python simulation files

│   └── helpers Stores python test bench helping functions

├── samples Stores sample code for simulation testing

├── tools Stores assembler and generator files

└── verilog Stores verilog simulation files

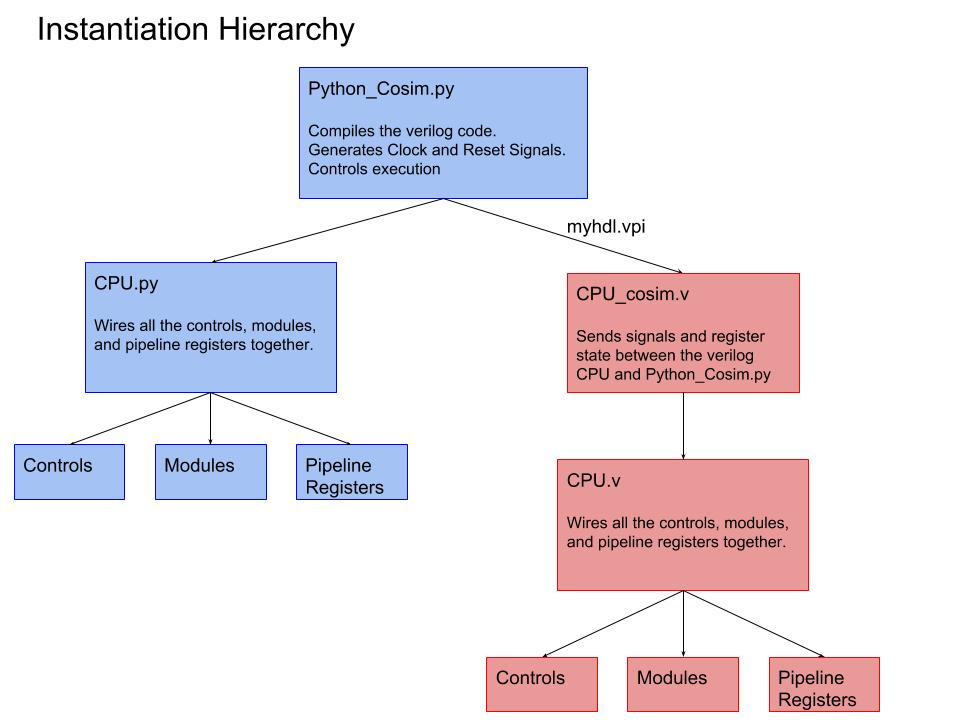
### File Conventions

All verilog files end with .v and all python files end with .py

If a file end with \_tb.[py|v], then the file is used to test the module.

If a file end with \_cosim.[py|v], then the file is used to control cosimulation. Each \_cosim.[py|v] pair is used to control one build target. I.E. the Register\_File\_cosim files are not used in the CPU Cosimulation. Run the python files to initiate cosimulation.

## Design Overview



## Instruction Opcode Dissection

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Opcode (HEX)** | **Format** | **Reg-write** | **Mem-write** | **Funct** | **RS** | **RT** | **RD** | **Target Address** | **New PC** | **Register Write #** |
| lw | 0x23 | i-format | Yes | No | N/A | [6:10] | N/A | N/A | MEM[RS] | PC + 4 | RS |
| sw | 0x2b | i-format | No | Yes | N/A | [6:10] | N/A | N/A | MEM[RS] | PC + 4 | N/A |
| add | 0x00 | r-format | Yes | No | 0x20 | [6:10] | [11:15] | [16:20] | N/A | PC + 4 | RD |
| sub | 0x00 | r-format | Yes | No | 0x22 | [6:10] | [11:15] | [16:20] | N/A | PC + 4 | RD |
| xor | 0x00 | r-format | Yes | No | 0x26 | [6:10] | [11:15] | [16:20] | N/A | PC + 4 | RD |
| addi | 0x08 | i-format | Yes | No | N/A | [6:10] | [11:15] | N/A | N/A | PC + 4 | RD |
| subi | 0x09 | i-format | Yes | No | N/A | [6:10] | [11:15] | N/A | N/A | PC + 4 | RD |
| beq | 0x04 | i-format | No | No | N/A | [6:10] | [11:15] | N/A | PC + (imm) | PC + 4 OR TA | N/A |
| j label | 0x02 | j-format | No | No | N/A | N/A | N/A | N/A | [6:31] | jump target | N/A |
| jal | 0x03 | j-format | Yes | No | N/A | N/A | N/A | N/A | [6:31] | jump target | 31 |
| jr $ra | 0x00 | r-format | No | No | 0x08 | N/A | N/A | N/A | $ra | $ra | N/A |

# C:\Users\boris\Downloads\Pipeline Diagram.jpg

## Stages

1. IF

Send PC to IM and fetch current instruction from memory. Update the PC to next in sequence: PC = PC+4

1. ID

Decode the instruction and read the registers in register file. Sign extend the offset field if it is needed. Compute the possible branch target and jump address. ‘Fixed field decoding’ - Decoding can be done in parallel with reading the registers since the register specifiers at a fixed location.

Hazard Detection Unit: This unit receives signal from various locations to check if there is going to be a hazard. If it detects a problem, it causes a stall, if not, the system continues normally. The Verilog and Python are implemented almost identically.

Other modules: Register File, RegDst Mux, Control, branch address and jump address calculators

1. EX

ALU: The ALU receives two 32-bit input variables and performs operations on them according to what the ALU Control line tells it is needed. It stores the result in a 33-bit register to prevent overflow but only returns a 32-bit value. It also has a zero line to signal a zero value and a reset line to reset the values if necessary. Unlike the Python implementation, the Verilog implementation uses a 32-bit variable for the output.

ALU Control: The ALU Control tells the ALU which operation to perform. Based on the operations signal it receives it tells the ALU which operation is needed, including JR.

Forwarding Unit: This unit checks different criteria and checks if it needs to forward data to avoid a stall. If it does, it sends a signal to the appropriate location telling it what to do. The Python implementation of this code was much simpler than Verilog since Verilog had to send out several signals instead of one.

Other modules: JRControl, three ALU multiplexers, IDEX\_to\_EXMEM mux

1. MEM

Load and store are performed. If load - read an address; store – write data into memory.

Data Memory: Data Memory fetches and stores data from main memory. It creates 256 32-bit memory locations and receives signals giving it data and a control signal telling it what to do with it.

Other module: Branch or Jump Taken Flush, three multiplexers to determine PC address (branch or jump)

1. WB

Last stage, register ALU or LW instruction to write result into register file in ID stage.

Modules: writeback\_source\_mux\_3\_to\_1

## Pipeline Stages

IF ID: The Fetch/Decode pipeline. This one has far fewer inputs since it mainly has to check for a branch and data hasn't been retrieved at this point.

ID EX: This is another pipeline register between the Decode and Execute stage. Similar to the EX/MEM pipeline, it checks for a reset signal and acts accordingly as well as checking for a stall or branch, making a flush necessary.

EX: A pipeline register used to flush or reset our entire system if necessary. It takes a signal and determines if it needs to reset or flush the pipeline. It takes a tremendous number of inputs because all of our data needs to pass through this pipeline stage.

MEM WB: The Memory/Writeback Pipeline register. Checks for a need to flush, if not, passes data appropriately.

## Multiplexers for routing data and control signals

Multiplexors: Contains each of our multiplexer's. They are all defines separately to help with identify and errors occuring during testing. Each of them takes a number of inputs and selects the appropriate one, from the control signal, and sends out the correct data.

Multiplexers.v: this file contains modules for all the necessary multiplexors. While a general purpose multiplexor can be used, an an instance created as needed,

decision is to explicitly define each multiplexor to aid in readability, understanding, and reduced risk of error. Once one of the choice inputs presented to the multiplexor is selected based on the provided control line, it is sent out as an output.

|  |  |  |
| --- | --- | --- |
| first\_alu\_mux\_3\_to\_1  //1st ALU source rs:  //Control is forwarding unit - Forward A  //if 0, selected input is 1st register file's output: rs  //if 1, selected input is forwarded ex to ex  //if 2, selected input is forwarded mem to ex | second\_alu\_mux\_3\_to\_1  //2nd ALU source rt:  //Control is from Forwarding Unit - Forward B  //if 0, selected input is 2nd register output rt  //if 1, selected input is forwarded ex to ex  //if 2, selected input is forwarded mem to ex | third\_alu\_mux\_2\_to\_1  //3rd ALU for source: 2nd mux output feeds into this one  // Control is ALUSrc  // if 0, take input from 2nd mux (can be either 2nd register rt, forwarded value mem to ex,  // or forwarded value ex to ex  // if 1, take input from 16bit immediate after it's sign extended |
| idEx\_to\_exMem\_mux\_2\_to\_1  //Mux to determine which destination address will be used and send to ex/mem pipeline stage  //Control: RegDst  //Inputs: rd, rt; Outputs: the chosen destination register  //if 0: rt (immediate type)  //if 1: rd (R type) | writeback\_source\_mux\_3\_to\_1  //mux to determine writeback source (32 bit value)  //Control is MemToReg  //if 0, R type and take ALU result  //if 1, lw and take mem stage output  //if 2, jal and take hardcoded PC+4 because we need to save in $ra for returning | regDst\_mux\_2\_to\_1  // multiplexer reg\_dst to determine write address  //assign reg\_write\_dest = (reg\_dst==2'b10) ? 3'b111: ((reg\_dst==2'b01) ? instr[6:4] :instr[9:7]);  // Control line is RegDst  // if 0, if some immediate type (e.g., lw) address comes from 2nd read register, bits 20:16  // if 1, R type, address comes from rd bits 15:11  // if 2, jal and address is hardcoded $31 for $ra slot |
| first\_PC4\_or\_branch\_mux\_2\_to\_1  //Jump and branch: can't use the simple 1 mux with PCSrc strategy in the book  //Need 3 muxes, outlined below. If result of last two muxes is 0, then what  //gets passed to the PC is simply PC+4.  // 1st mux: PC or branch 2:1  // Control line comes from branch decision AND gate  // if control is 0, select PC+4  // if control is 1, select sign extended label added to PC for BTA | second\_jump\_or\_first\_mux\_2\_to\_1  // 2nd mux: jump or first mux output  // 1st mux feeds it's output value into 2nd as one of the inputs.  // 2nd input comes from jump\_address\_calculator.  // Control line comes from Control Unit jump line. If 1, then it's a jump. Take 2nd input.  // If 0, it is the 1st input (can be either PC+4 or BTA depending on 1st mux result)  // If 1, calculated jump address (shift two, concat top 4 of PC) | third\_jr\_or\_second\_mux\_2\_to\_1  //3rd mux: jr  //2nd mux sends its output as one of the inputs. 2nd input comes from 1st read data  //in register (rs) at all times.  //Control line comes from the JRControl module in ALU Control  //If control is 0 we take 1st input as determined by 2nd mux  //If control is 1 we take register value which contains jr address |
| hazard\_stall\_mux\_2\_to\_1  //Hazard detection mux in ID stage  //2:1 mux that stalls if lw conflict detected  //Control line: Mux\_Select\_Stall (from Hazard Unit)  //Inputs: Control Unit signal, hardcoded zero  //If Control is 0, output is whatever is sent by Control Unit  //If Control is 1, output is 0 and sent to ID/EX wb, m, and ex control lines | PC\_input\_mux\_2\_to\_1  //Mux to select either output of third\_jr\_or\_second\_mux output  //or next available instruction  //Control: Ctrl\_branch\_or\_jump\_taken  //If control is high, take output of 3rd mux that contains branch or jump address  //If zero, proceed to PC+4 |  |

## Hazard and Forwarding

Forwarding unit selects the correct ALU inputs for the EX stage. If no hazard, ALU operands come from reg file as normal. Data hazard - operand come from either EX\_MEM or MEM\_WB pipeline reg. MEM/WB hazard may occur between an instruction in the EX stage and the instruction from two cycles ago.

Flush IF/ID, ID/EX and EX/MEM if branch OR jump is determined viable at MEM stage.

Stalls: (ID\_EX\_MemRead == 1 && ((ID\_EX\_RegRt == IF\_ID\_RegRs) || (ID\_EX\_RegRt == IF\_ID\_RegRt)))

Mux\_Select\_Stall <= 1; PCWrite <= 0; IF\_ID\_Write <= 0;

Flushes: (((EX\_MEM\_ALU\_Zero\_out\_in == 1) && (EX\_MEM\_branch\_out\_in == 1)) || (EX\_MEM\_jump\_out\_in == 1))

IF\_Flush <=1 ; ID\_Flush\_Branch <=1 ; EX\_Flush <=1 ; branch\_or\_jump\_taken <= 1;

Forwarding:

// Mux\_ForwardA equations

if (EX\_MEM\_RegWrite == 1 && EX\_MEM\_RegRd == ID\_EX\_RegRs) begin

Mux\_ForwardA <= 2;

end

else if(MEM\_WB\_RegWrite == 1 && MEM\_WB\_RegRd == ID\_EX\_RegRs &&

(EX\_MEM\_RegRd != ID\_EX\_RegRs || EX\_MEM\_RegWrite == 0)) begin

Mux\_ForwardA <= 1;

end

else begin

Mux\_ForwardA <= 0;

end

// Mux\_ForwardB equations

if (EX\_MEM\_RegWrite == 1 && EX\_MEM\_RegRd == ID\_EX\_RegRt) begin

Mux\_ForwardB <= 2;

end

else if (MEM\_WB\_RegWrite == 1 && MEM\_WB\_RegRd == ID\_EX\_RegRt &&

(EX\_MEM\_RegRd != ID\_EX\_RegRt || EX\_MEM\_RegWrite == 0)) begin

Mux\_ForwardB <= 1;

end

else begin

Mux\_ForwardB <= 0;

end

# 5    Jump and Branch Address Calculators

branch\_calculator:

Sign extend 16 bit offset

Shift by 2 / multiply by 4. PC is already word aligned, immediate value must be aligned

Add resulting 32 bit address to PC + 4 to obtain BTA

Inputs: immediate (or offset) value already sign extended to 32bits

Outputs: BTA (32bits)

jump\_calculator:

Shift by 2 / multiply 26 bit value by 4

Jumping is PC-relative, concatenate first 4 bits of PC to left of jump address.

Resulting 32 bit address is the jump value.

Inputs: Instr[25:0] - last 26 bits of instruction

Outputs: Jump Address 32bits