# MIPS in VHDL

Booker A Robinson

January 2025

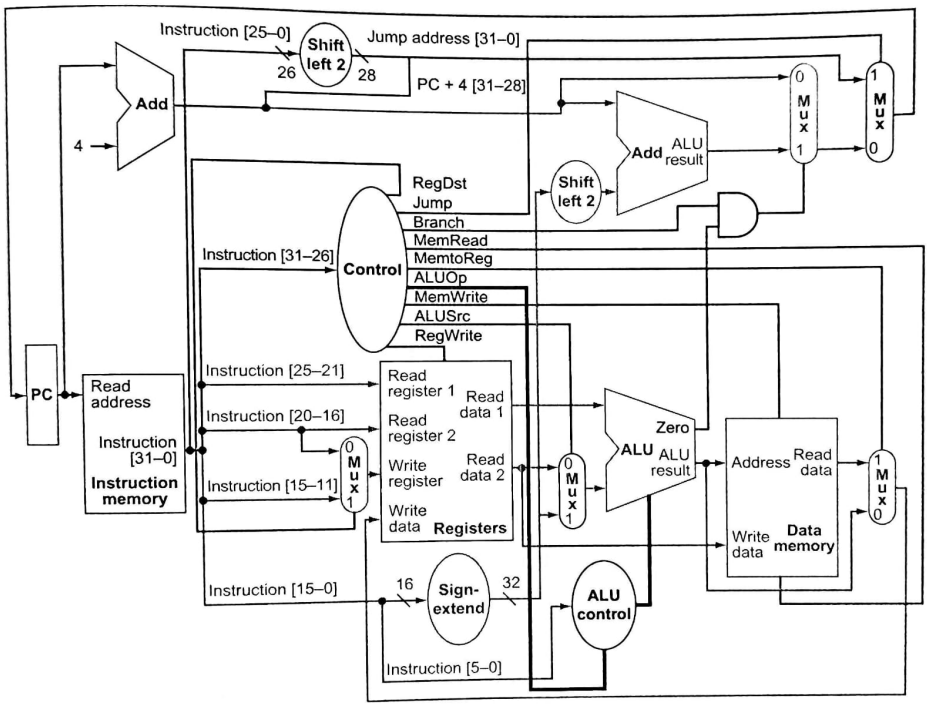
## Contents

# 1 Introduction

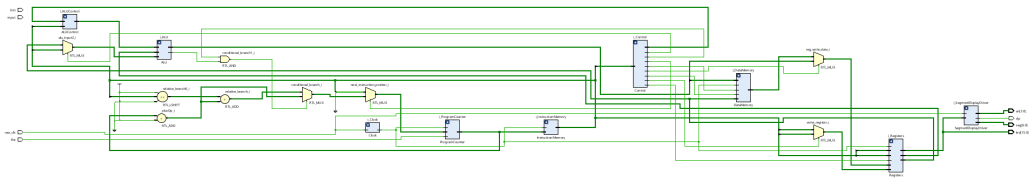
This document will attempt to distill the implementation of a simple Single-Cycle CPU architecture in VHDL. The architecture that we will be implementing is called MIPS.

# 2 Dissecting MIPS

Here we can see a schematic of the MIPS architecture that we will be implementing.



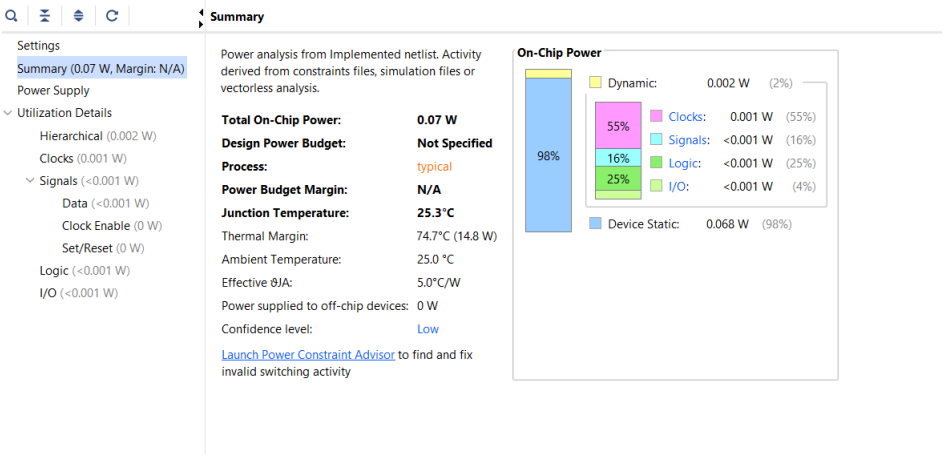

MIPS architecture schematic [4]



Xilinx

| Name | Slice LUTs (20800) | Slice Registers (41600) | F7 Muxes (16300) | F8 Muxes (8150) | Slice (8150) | LUT as Logic (20800) | Bonded IOB (106) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|---|---|
| N MIPS | 611 | 822 | 64 | 31 | 368 | 611 | 30 | 2 |
| i_ALU (ALU) | 0 | 0 | 0 | 0 | 16 | 0 | 0 | 0 |
| i_ALUControl (ALUControl) | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| i_Clock (Clock) | 6 | 24 | 0 | 0 | 10 | 6 | 0 | 0 |
| i_ProgramCounter (ProgramCounter) | 206 | 30 | 0 | 0 | 94 | 206 | 0 | 0 |
| i_Registers (Registers) | 390 | 737 | 64 | 31 | 336 | 390 | 0 | 0 |
| i_SegmentDisplayDriver (SegmentDisplayDriver) | 8 | 31 | 0 | 0 | 17 | 8 | 0 | 0 |

Utilization




Timing

## 2.1 Registers

| Instruction | Index | Binary |
|---|---|---|
| $zero | 0 | 00000 |
| $at | 1 | 00001 |
| $v0 | 2 | 00010 |
| $v1 | 3 | 00011 |
| $a0 | 4 | 00100 |
| $a1 | 5 | 00101 |
| $a2 | 6 | 00110 |
| $a3 | 7 | 00111 |
| $t0 | 8 | 01000 |
| $t1 | 9 | 01001 |
| $t2 | 10 | 01010 |
| $t3 | 11 | 01011 |
| $t4 | 12 | 01100 |
| $t5 | 13 | 01101 |
| $t6 | 14 | 01110 |
| $t7 | 15 | 01111 |

| Instruction | Index | Binary |
|---|---|---|
| $s0 | 16 | 10000 |
| $s1 | 17 | 10001 |
| $s2 | 18 | 10010 |
| $s3 | 19 | 10011 |
| $s4 | 20 | 10100 |
| $s5 | 21 | 10101 |
| $s6 | 22 | 10110 |
| $s7 | 23 | 10111 |
| $t8 | 24 | 11000 |
| $t9 | 25 | 11001 |
| $k0 | 26 | 11010 |
| $k1 | 27 | 11011 |
| $gp | 28 | 11100 |
| $sp | 29 | 11101 |
| $fp | 30 | 11110 |
| $ra | 31 | 11111 |

## 2.2 Control

The controller takes in the op code of an instruction and outputs control flags for each component in the CPU. Our 12 bit control output will be as follows.

| reg_dst | jump | link | branch | mem_read | mem_to_reg | alu_op | mem_write | alu_src | reg_write |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 00 | 0 | 0 | 000 | 0 | 0 | 0 |

### 2.2.1 Conditional Branching

| branch | Action |
|---|---|
| 0 0 | No branch |
| 0 1 | Branch if equal |
| 1 0 | Branch if not equal |
| 1 1 | Uncondition relative branch |

*Note: The unconditional relative branch is not used*

## 2.3 Decoding R-type Instructions

| Instruction | op | rs | rt | rd | shamt | funct | Control |
|---|---|---|---|---|---|---|---|
| sll $\$t2, \$t0, 2$ | 000000 | 00000 | 01001 | 01010 | 00010 | 000000 | 1000000 010 001 |
| srl $\$t2, \$t0, 2$ | 000000 | 00000 | 01001 | 01010 | 00010 | 000010 | 1000000 010 001 |
| add $\$t2, \$t0, \$t1$ | 000000 | 01000 | 01001 | 01010 | 00000 | 100000 | 1000000 010 001 |
| sub $\$t2, \$t0, \$t1$ | 000000 | 01000 | 01001 | 01010 | 00000 | 100010 | 1000000 010 001 |
| and $\$t2, \$t0, \$t1$ | 000000 | 01000 | 01001 | 01010 | 00000 | 100100 | 1000000 010 001 |
| or $\$t2, \$t0, \$t1$ | 000000 | 01000 | 01001 | 01010 | 00000 | 100101 | 1000000 010 001 |
| nor $\$t2, \$t0, \$t1$ | 000000 | 01000 | 01001 | 01010 | 00000 | 100111 | 1000000 010 001 |
| slt $\$t2, \$t0, \$t1$ | 000000 | 01000 | 01001 | 01010 | 00000 | 101010 | 1000000 010 001 |
| jr $\$ra$ | 000000 | 01101 | 00000 | 00000 | 00000 | 001000 | 1000000 010 001 |

## 2.4 Decoding I-type Instructions

| Instruction | op | rs | rt | constant/address | Control |
|---|---|---|---|---|---|
| beq $\$t2, \$t0, 2$ | 000100 | 01000 | 01010 | 0000000000000010 | 0000100 001 000 |
| bne $\$t2, \$t0, 2$ | 000101 | 01000 | 01010 | 0000000000000010 | 0001000 001 000 |
| addi $\$t2, \$t0, 2$ | 001000 | 01000 | 01010 | 0000000000000010 | 0000000 000 011 |
| slti $\$t2, \$t0, 2$ | 001010 | 01000 | 01010 | 0000000000000010 | 0000000 110 000 |
| andi $\$t2, \$t0, 2$ | 001100 | 01000 | 01010 | 0000000000000010 | 0000000 011 000 |
| ori $\$t2, \$t0, 2$ | 001101 | 01000 | 01010 | 0000000000000010 | 0000000 100 000 |
| lw $\$t2, 2(\$t0)$ | 100011 | 01000 | 01010 | 0000000000000010 | 0000011 000 011 |
| sw $\$t2, 2(\$t0)$ | 101011 | 01000 | 01010 | 0000000000000010 | 0000000 000 011 |

## 2.5 Decoding J-type Instructions

| Instruction | op | constant/address | Control |
|---|---|---|---|
| j label | 000010 | 00000000000000000000000000 | 0100000 000 000 |
| jal label | 000011 | 00000000000000000000000000 | 0110000 000 000 |

## 2.6 Understaning ALU OP

Most resources discussing MIPS implement an extremely limited subset of instructions (MIPS-lite)[5]. Namely; add, sub, and, or, slt, lw, sw, beq, j. This subset allows us to in general to use only two bits to encode aluop. The truth table is usually shown as such:

| alu_op1 | alu_op0 | ALU Instruction |
|---|---|---|
| 0 | 0 | $add\ [lw, sw]$ |
| 0 | 1 | $sub\ [beq]$ |
| 1 | 0 | $r - type$ |
| 1 | 1 | $n/a$ |

This resolution is not sufficient for us because we want to implement datapaths for I-type instructions other than lw, sw, and beq. To accomplish this we will ad an extra bit to extend the supported I-type instructions according to the following table.

| alu_op2 | alu_op1 | alu_op0 | ALU Instruction |
|---|---|---|---|
| 0 | 0 | 0 | $add$ |
| 0 | 0 | 1 | $sub$ |
| 0 | 1 | 0 | $r - type$ |
| 0 | 1 | 1 | $and$ |
| 1 | 0 | 0 | $or$ |
| 1 | 0 | 1 | $nor$ |
| 1 | 1 | 0 | $slt$ |
| 1 | 1 | 1 | $n/a$ |

### 2.7 ALU Control

ALU Control takes as input the ALU OP and the funct. It outputs the control lines directly to the ALU. With our modified ALU Op signals we will design the ALU control as such.

| alu_op | funct | ALU Control | ALU Instruction | shift | shift right |
|---|---|---|---|---|---|
| 000 | xxxxxx | 0010 | *add* | 0 | 0 |
| 001 | xxxxxx | 0110 | *sub* | 0 | 0 |
| 010 | 000000 | 0001 | *sll* | 1 | 0 |
| 010 | 000010 | 0001 | *srl* | 1 | 1 |
| 010 | 100000 | 0010 | *add* | 1 | 0 |
| 010 | 100010 | 0110 | *sub* | 1 | 0 |
| 010 | 100100 | 0000 | *and* | 1 | 0 |
| 010 | 100101 | 0001 | *or* | 1 | 0 |
| 010 | 100111 | 1100 | *nor* | 1 | 0 |
| 010 | 101010 | 0111 | *slt* | 1 | 0 |
| 011 | xxxxxx | 0000 | *and* | 0 | 0 |
| 100 | xxxxxx | 0001 | *or* | 0 | 0 |
| 101 | xxxxxx | 1100 | *nor* | 0 | 0 |
| 110 | xxxxxx | 0111 | *slt* | 0 | 0 |
| 111 | xxxxxx | _ | *n/a* | 0 | 0 |

# 3 Designing the CPU

## 3.1 Designing the Instruction Decoder (Controller)

## 3.2 Designing the Instruction Memory

### 3.2.1 Reading Program from a File

```
impure function init_ram_hex return array_instructions is
    file text_file: text open read_mode is program_file;



    variable text_line: line;
    variable ram_content: array_instructions;
    variable word: std_logic_vector(31 downto 0);
begin
    for i in 0 to 15 loop
        readline(text_file, text_line);
        bread(text_line, word);
        ram_content(i*4) := word(31 downto 24);
        ram_content(i*4+1) := word(23 downto 16);
        ram_content(i*4+2) := word(15 downto 8);
        ram_content(i*4+3) := word(7 downto 0);
    end loop;
    return ram_content;
end function;
```

```
signal instruction_memory: array_instructions := init_ram_hex;
```

### 3.2.2 Multiplexing the Instruction Memory

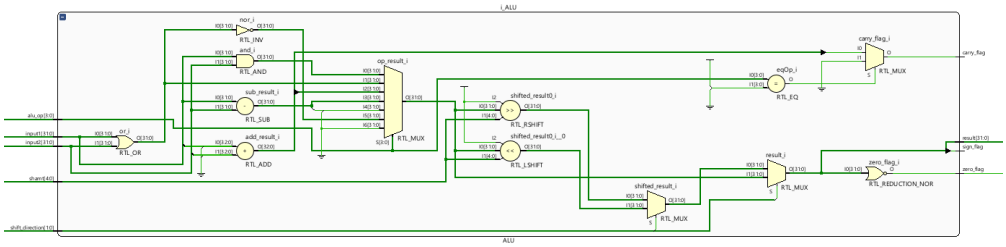
```
instruction <=
    instruction_memory(to_integer(instruction_pointer)) &
    instruction_memory(to_integer(instruction_pointer+1)) &
    instruction_memory(to_integer(instruction_pointer+2)) &
    instruction_memory(to_integer(instruction_pointer+3))
    when (instruction_pointer <= instruction_count*4-4) else x"
00000000";
```

## 3.3 Designing the ALU

| Function | ALU control lines |
|---|---|
| and | 0000 |
| or | 0001 |
| add | 0010 |
| subtract | 0110 |
| set on less than | 0111 |
| nor | 1100 |

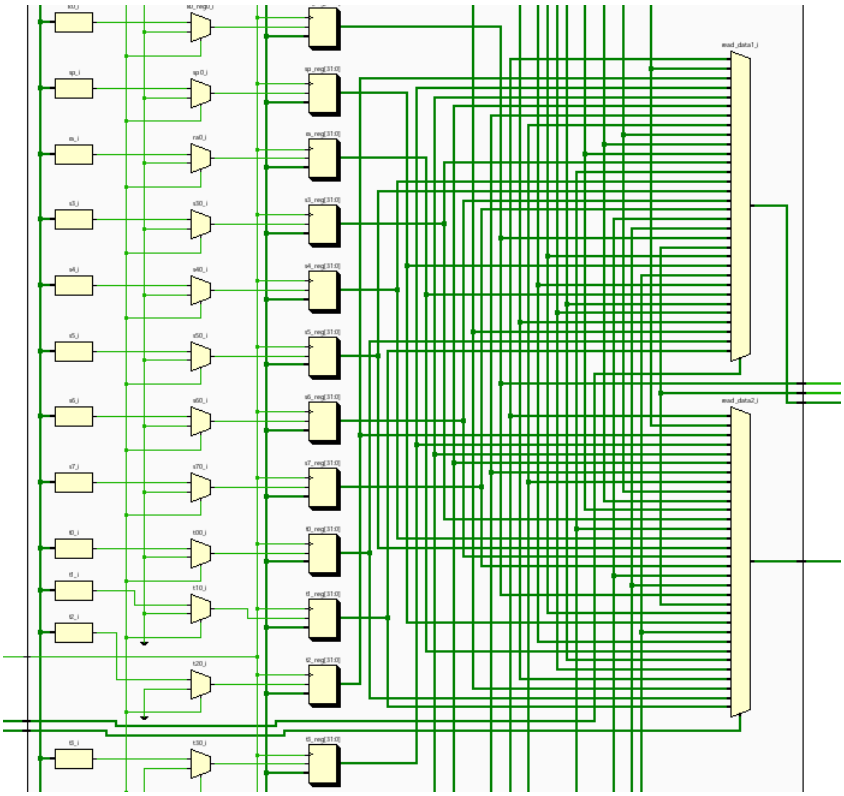
```
zero_result <= nor result;

with alu_op select
    result <=
        std_logic_vector(unsigned(input1) and unsigned(input2))
 when b"0000",
        std_logic_vector(unsigned(input1) or unsigned(input2))
when b"0001",
        std_logic_vector(signed(input1) + signed(input2)) when
b"0010",
        std_logic_vector(signed(input1) - signed(input2)) when
b"0110",
        set_less_than(input1, input2) when b"0111",
        std_logic_vector(unsigned(input1) nor unsigned(input2))
 when b"1100",
        X"0000" when others;
```



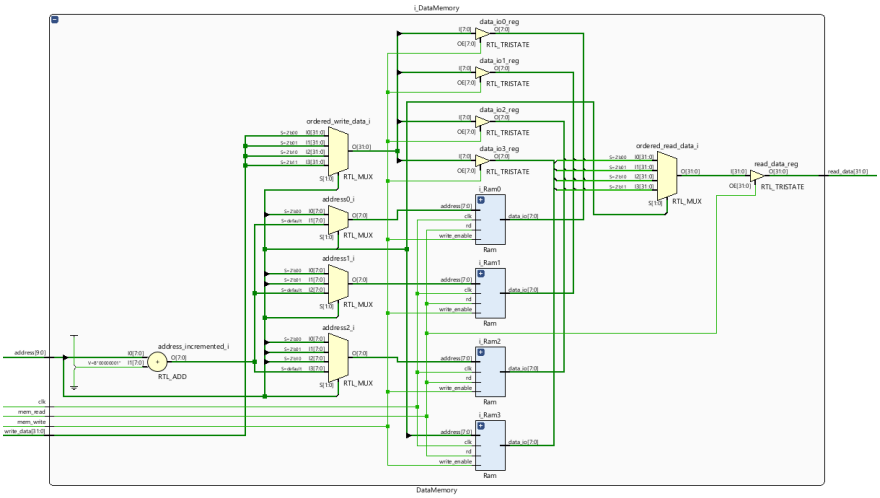
ALU

## 3.4 Designing the Registers



Registers

## 3.5 Designing the Data Memory




Data Memory

### 3.5.1 Defining the Memory mapped IO

## 3.6 IO

## 3.7 Segment Display Driver

## 3.8 Connecting all the parts

# 4 Programming with Assembly and Machine Code

## 4.1 Examples

### 4.1.1 Fibonacci

```
reset:
addi $t0, $zero, 1
addi $k0, $zero, 1
loop: add $t1, $k0, $t0
  add $t0, $k0, $zero
  add $k0, $t1, $zero
  addi $t4, $zero, 28657 # largest 16 bit fibonacci number
  beq $k0, $t4, reset
  j loop
```

```
00100000000010000000000000000001
00100000000011010000000000000001
00000011010010000100100000100000
00000011010000000100000000100000
00000001001000001101000000100000
00100000000011000110111111110001
```

```
00010001100110101111111111111001
00001000000000000000000000000010
```

### 4.1.2 Interactive program

```
# Interactive Program.
#   IO:
#     First 16 bits of k0 is the 7 segment display.
#     Second 16 bits of k0 is the led array.
#     First 16 bits of k1 is the switches.
#     Bits 16-20 of k1 are the buttons in the following order:
#     C, U, D, L, R

# Functionality.
  # Runs a loop selection.



addi $k0, $0, 1 # Set Segment display 1.
loop_selection:
  srl $t0, $k1, 17 # index Up button.
  andi $t0, $t0, 1 # Bitmask button.
  bne $t0, $zero, run_lights # Check button press.
  addi $k0, $0, 2 # Set Segment display 2.
  srl $t0, $k1, 18 # index Down button.
  andi $t0, $t0, 1 # Bitmask button.
  bne $t0, $zero, increment_segment_display # Check button press.
  addi $k0, $0, 3 # Set Segment display 3.
  srl $t0, $k1, 16 # index Center button.
  andi $t0, $t0, 1 # Bitmask button.
  bne $t0, $zero, fibonacci # Check button press.
  addi $k0, $0, 1 # Set Segment display 1.
  j loop_selection




run_lights:
  addi $t0, $0,16384 # 100000000000000
  sll $t0, $t0, 1
  addi $t3, $0, 15
  addi $t1, $0, 0
  run_lights_loop:
    sll $k0, $t0, 1 # Light up each led one at a time.
    sll $t0, $t0, 1
    addi $t1, $t1, 1
    bne $t1, $t3, run_lights_loop
  j loop_selection

increment_segment_display:
  addi $t3, $0, 16
  addi $k0, $0, 0
  increment_segment_display_loop:
    addi $k0, $k0, 1 # Increment the segment display 16 times.
```

```
    bne $k0, $t3, increment_segment_display_loop
  j loop_selection


fibonacci:
  reset:
  addi $t0, $zero, 1
  addi $k0, $zero, 1
  loop: add $t1, $k0, $t0
    add $t0, $k0, $zero
    add $k0, $t1, $zero
    addi $t4, $zero, 28657 # largest 16 bit fibonacci number
    beq $k0, $t4, reset
    j loop
  j loop_selection
```

# 5 TODO

1. Add jump link and jump reg instructions.
2. Add MMIO
3. Fix lw and sw in assembler
4. Add .data file
5. Mul
6. Implement convolution

Future work

1. vector registers / Instructions (ie. SIMD)

# 6 Conclusions and Future Work

# 7 References

[1] https://edg.uchicago.edu/~tang/VHDLref.pdf
[2] https://staff.fysik.su.se/~silver/digsyst/vhdl_ref.pdf
[3] https://digilent.com/reference/programmable-logic/basys-3/start
[4] Computer Organization and Design, Patterson & Hennessy [5] https://www.cs.umd.edu/~meesh/cmsc311/clin-cmsc311/Lectures/lecture32/single_control.pdf