# Very Naive MIPS CPU using Clash

ZHU Yifan (118010469) <i@zhuyi.fan>

April 27, 2020

In memory of Carl Quinn, for his great contributions to the field of Programming Languages.

# Introduction

In CUHK(SZ) and many other universities, writing an MIPS CPU with pipelines is a required work for the architecture courses. However, most teaching materials just provide students with some basic concepts of CPUs and do not give essential introductions on languages or the potential difficulties of the implementation. Writing this book, we want to achieve the following goals:

- Give a detailed description on each part of the MIPS CPU.
- Clarify how we can write a sequential logic circuit, avoiding oscillations and other problems.
- Introduce Clash, a higher level HDL that can generate synthesizable Verilog files and reduce the complexity of development.

We are **NOT** going to implement a fully functional MIPS CPU. Instead, we will only structure some skeletons that can help us understand the concepts and principles. We hope the readers can gain some basic knowledge of hardware design and the Clash language from this book.

# Contents

# Chapter 1

# Preparation

RIGHT before our journey of implementing the MIPS CPU using Clash language, we need to get out equipment ready.

## 1.1 Prerequisites of this Book

Reading this book, you are expected to have some basic knowledge of Verilog HDL and the Haskell language. However, if you happen to have little experience on these two languages, do not worry too much; they are just the language tools that we are going to use to express the logic and thoughts. The expressions should be easy to understand and we are going to provide some detailed descriptions on those critical lines.

It is also a good idea to acquire some basic knowledge about Digital Logic Circuits. You'd better grab the concepts of clock, combinatorial logic and sequential logic.

## 1.2 Install iVerilog

iVerilog is a tool to synthesis Verilog sources and generate simulation executables. We are going to use it as our default Verilog compiler. It is available to GNU/Linux, Mac OS and Windows.

Windows users can follow this link (`http://bleyer.org/icarus/`) to download it.

Mac users can follow this tutorial

(`https://blog.csdn.net/zach_z/article/details/78787509`) to download it.

As for GNU/Linux users, I believe you have already found a way to get it work.

## 1.3 Prepare Haskell Environment

We are using stack for the projects. It should be easy to install, just go through this document (`https://docs.haskellstack.org`) to get all the requirements settled.

As for Clash, there are several ways to install it. It is ready for Nix build system, Snapcraft and it is also doable to compile from the source. You are recommended to visit its website (`https://clash-lang.org`) before you start installing it.

After all things are settled, you should be able to play with the template project at GitHub, under dramforever/clash-with-stack (Great thanks for **dramforever**).

Its clash version is a little bit old, but it is enough for this book. feel free to upgrade the version to the latest ones (tested until 1.2.0).

This template does not use `mtl` library that we needs (for the state monad), you may need to add it on you own at the `package.yaml`.

# Chapter 2

# Essential Verilog

We would like to introduce some basic Verilog knowledge that will be used in this book. We will not go into details here, just showing some most common use cases. Let us first take a look at the final outcome of out project.

## 2.1 Module Interface

```
1  `timescale 100fs/100fs
2  module CPU
3      ( input  CLOCK // clock
4      , input  RESET // reset
5      , input  ENABLE
6      );
7      wire [32:0] BRANCH;
8      wire [30:0] PC_INSTRUCTION;
9      wire [31:0] PC_VALUE;
10     wire [37:0] WRITE_PAIR;
11     wire        STALL;
12     wire [5:0]  DM_WRITE;
13     wire [1:0]  DM_MEM;
14     //...
15
```

```
16      InstructionModule IM
17      ( // Inputs
18      .CLOCK(CLOCK), // clock
19      .RESET(RESET), // reset
20      .ENABLE(ENABLE),
21      .BRANCH(BRANCH),
22      .STALL(STALL),
23
24      // Outputs
25      .PC_INSTRUCTION(PC_INSTRUCTION),
26      .PC_VALUE(PC_VALUE)
27      );
28      // ...
29
30      assign AM_FW_0 = MMO_WRITE_PAIR;
31
32      assign AM_FW_1 = WB_WRITE_PAIR;
33
34      assign WRITE_PAIR = WB_WRITE_PAIR;
35
36      assign BRANCH  = WB_BRANCH;
37
38  endmodule
```

These lines are extracted from the CPU source code.

1. The first line is a compiler derivative that defines the precision of timing;
2. Line 2 to line 6 defines the module interface of a CPU, with three input ports. As for outputs, you can add something like `output wire OUTPUT;`
3. Line 7 to line 13 declare some wire variables. As you can see, you can point out the number of bits in the declaration. Wires are used to connect different components of the circuits, you can treat them as a renaming of the original port because the value of a wire is refresh as soon as the input side changes.

4. Apart from wires, another commonly used thing is `reg [31:0] REGISTER`; you can treat this as the variables in the common sense, which has its own state.

5. `InstructionModule IM (...)` declares a component named `IM` whose definition is in another module called `InstructionModule`;

6. There are mainly two ways to interact with another module, one is to use it as what is listed in the code: use a syntax like `.PORT(SOME_WIRE)` to connect the inputs and outputs; the other way is commonly used in debugging: you can get the value of the components in another module via something like `IM.CLOCK`;

7. Those `assign` statements are used to connect the wires.

## 2.2 Sequential Structures

The previous example is just about connecting wires, however, many circuits also to need to handle sequential events.

### 2.2.1 Assigning Sequential Values

There are several ways to assigning values in a sequential logic environment:

```
module example();
    reg A, B, C;
    initial begin
        A = 1;
        B = A;
        C = B;
    end
endmodule
```

This example shows a way to assign the initial values of some registers within a `initial` block; Notice that = stands for the blocking assignment, which means the assignments will happen one by one.

What if we want to handle the assignment at some specific time? We can then use a statement in the form of `always @( ... sensitivity list ... ) begin`,

the following example shows the assignments happening on each rising edge of the clock:

```
always @(posedge CLOCK) begin
    B <= A;
    C <= B;
    D <= C;
end
```

If you want to describe a combinatorial logic, you should use `always@( * )` (only blocking assignment should be used within the scope, otherwise it is likely to generate unexpected oscillations – the circuit will never reach a stable state), the event within the block will be triggered as long as any of the inputs changes.

### 2.2.2 Test Bench

```
module TEST();
    reg clk, reset, enable;
    initial
        begin
        clk = 0;
        reset = 0;
        enable = 1;
    end

    always
       #1000 clk = !clk;


    CPU cpu(clk, reset, enable);


    initial begin
```

```
19 $monitor(
20 "========================================\n",
21 "TIME:                %-d\n",             $time,
22 "STALL:               %b\n",              cpu.STALL,
23 "----------------- Instruction ------------\n",
24 "PC/4 + 1:            %-d\n",             cpu.IM.PC_VALUE,
25 "INSTRUCTION:         %b\n",              cpu.IM.result_1[31:0],
26 "INSTRUCTION:         %b [inner form]\n", cpu.IM.PC_INSTRUCTION,
27 "-------------------- Decode --------------\n",
28 "RS:                  %-d\n",    cpu.DM_RS,
29 "RS VALUE:            %b\n",     cpu.DM_RSV,
30 "RT:                  %-d\n",    cpu.DM_RT,
31 "RT VALUE:            %b\n",     cpu.DM_RTV,
32 "MEM_OP:              %b\n",     cpu.DM_MEM,
33 "REG_WRITE:           %b\n",     cpu.DM_WRITE,
34 "ALU_CTL:             %b\n",     cpu.DM_ALU,
35 "IMMEDIATE:           %b\n",     cpu.DM_IMM,
36 "STAGE_PC/4 + 1:      %-d\n",    cpu.DM_COUNTER,
37 "----------------- Arithmetic --------------\n",
38 "REG_WRITE:           %b\n",     cpu.AM_WRITE_REG,
39 "MEM_OP:              %b\n",     cpu.AM_MEM_OP,
40 "ALU_RESULT:          %b\n",     cpu.AM_RESULT,
41 "BRANCH_TARGET:       %b\n",     cpu.AM_BRANCH_TARGET,
42 "--------------------Memory -----------------\n",
43 "BRANCH_TARGET:       %b\n",     cpu.MMO_BRANCH,
44 "WRITE_BACK:          %b\n",     cpu.MMO_WRITE_PAIR,
45 "NEXT_FETCH_ADDRESS:  %-x\n",
   ↪  cpu.MM.MainMemory_res.FETCH_ADDRESS,
46 "FETCH_RESULT:        %-x\n",    cpu.MM.MainMemory_res.DATA,
47 "WRITE_SERIAL:        %b\n",
   ↪  cpu.MM.MainMemory_res.EDIT_SERIAL,
48 "----------------- Write Back ---------------\n",
```

```
49 "BRANCH_TARGET:       %b\n",     cpu.WB_BRANCH,
50 "WRITE_BACK:          %b\n",     cpu.WB_WRITE_PAIR,
51 "=========================================\n"
52 );
53     #100000 $finish();
54     end
55
56 endmodule
```

Here is the test bench that we are going to use. Those #XXXX statements mean delaying the given amount nanoseconds before the event happening. Hence,

```
always
    #1000 clk = !clk;
```

actually defines a clock with period 2000. There are several special functions we are going to use,

- `$finish()` terminates the simulation
- `$stop()` pauses the simulation
- `$display("format string", a, "format string", b)` displays the instant value of the variables; basic formats are:
  - %d: digits
  - %-d: digits (left aligned)
  - %b: binary
  - %x: hexdecimal
- `$monitor("format string", a, "format string", b)` used the same as display, but it will be triggered everytime a monitored variable updates
- `$time` gets the current time
- `$readmemb("file.bin",BLOCK);` initializes a large memory block with a file

# Chapter 3

# Journey to the Clash Language

Clash will be our main language to write CPU. Clash supports most of Haskell syntax, but yet it cannot support some advanced features like GADT pattern matching. To see the full list of limitations, please check its official tutorial (http://hackage.haskell.org/package/clash-prelude-1.2.0/docs/Clash-Tutorial.html). It will also be a great idea to go through the troubleshooting part if you face some difficulties later.

## 3.1 Define Circuits

You can simply write a circuit in the way of writing a Haskell function:

```
module Example where
orGate :: Bool -> Bool -> Bool
orGate = (||)
```

How to generate a verilog module from the code? If your function is named as `topEntity`, just load the clash.clashi on your own or using stack and then input `:verilog Example` in the REPL, then the outputs are ready at the `verilog` subdirectory under your working directory.

However, in most cases, you need to write a special annotation for the function:

```
{-# ANN orGate
    (Synthesize{
```

```
        t_name = "OrGate",
        t_inputs = [PortName "X", PortName "Y"],
        t_output = PortName "RESULT" })#-}
```

As you can see, you can customize the name for the ports and the whole module. There is another cool thing that you can also set a test bench for your circuits. As we are not going to use clash to generate test benches, it is up to you to investigate it on your own. Here is the link:

`http://hackage.haskell.org/package/clash-prelude-1.2.1/docs/Clash-Annotations-TopEntity.html#v:TestBench`

Here is another example to demonstrate how to handle product ports.

```
{-# ANN someGates
  (Synthesize
     {  t_name = "SomeGates"
     ,  t_inputs =
        [  PortName "X"
        ,  PortProduct "IN"
             [  PortName "Y"
             ,  PortName "Z"
             ]
         ]
     ,  t_output = PortProduct "OUT"
        [  PortName "0"
        ,  PortName "1"
        ,  PortName "2"
        ]
     }
  )
#-}
```

This annotation can be used to handle functions in the form of

```
someGate :: x -> (y, z) -> (o0, o1, o2)
```

## 3.2 Useful Types

### 3.2.1 Bit and BitVector