Very Naive MIPS CPU using Clash

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

April 28, 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

- `$dumpfile("file.vcd")` dumps IEEE standard vcd files. These files are be visualized by a someware like GtkWave to provide handy debugging information.
- `$dumpvars(0, cpu)` sets the value and module to dump; level 0 will automatically dumps the variables in the module recursively while level 1 will only dumps those manually listed variables.

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

Bit is just bit and BitVector is just a statically sized vector of bits. As a single Bit and Bool are quite the same, `Clash.Prelude` provides some handy functions for us to convert them from each other.

```
boolToBit :: Bool -> Bit
bitToBool :: Bit -> Bool
boolToBV :: KnownNat n => Bool -> BitVector (n + 1)
```

BitVector can be sliced and indexed, the following code shows some examples:

```
(!) :: (BitPack a, Enum i) => a -> i -> Bit
-- ^ BitVector is within the class of BitPack, so we can use (!)
↪  operator to fetch the value

vec ! 0
-- ^ fetch the lowest bit

slice :: (BitPack a, BitSize a ~ ((m + 1) + i))
    => SNat m -> SNat n -> a -> BitVector ((m + 1) - n)
-- ^ fetch a slice from the the bit vector
-- ^ because the vector is defined as a dependent type
-- ^ this interface is a bit terrifying, but it is handy to use

slice d31 d20 vec
-- ^ slice from index 31 to index 20
```

As you can see, `d0` to `d1024` are predefined literals for static natural numbers, you can use them for the vector index.

How about the bitwise operations? There is a whole set of operators and functions.

```
(.|.) :: Bits a => a -> a -> a          -- ^ bitwise or
(.&.) :: Bits a => a -> a -> a          -- ^ bitwise and
```

```
xor :: Bits a => a -> a -> a             -- ^ bitwise xor
complement :: Bits a => a -> a           -- ^ bitwise not
shiftR :: Bits a => a -> Int -> a        -- ^ bitwise shift
shiftL :: Bits a => a -> Int -> a        -- ^ bitwise shift
unsafeShiftR :: Bits a => a -> Int -> a -- ^ bitwise shift
```

3.2.2 Sized Integers

There are mainly two types of sized integers: `Unsigned (n :: Nat)` and `Signed (n :: Nat)`. Most bitwise operations can also be applied to sized integers, but please notice that for right shifting, the normal version takes care of the sign bit and the unsafe version just do the logical shifting.

It is also possible to extend sized integers and BitVector,

```
extend :: (Resize f, KnownNat a, KnownNat b) => f a -> f (b + a)
```

It is very handy that extensions will handle the changing of the sign bits automatically.

Although BitVector and sized integers are very similar, you cannot treat them as the same thing. However, if you need to convert the types, you can use the following functions:

```
pack :: BitPack a => a -> BitVector (BitSize a)
unpack :: BitPack a => BitVector (BitSize a) -> a
```

3.2.3 Remarks on Haskell Types

Other Haskell types like `Maybe` can also be used without problem. For example, `Maybe Bool` will be represented by 2 bits in Verilog.

```
Just True  = 11
Just False = 10
Nothong    = 0x
```

Clash will also find a way to represent your own defined product types, for example,

```
data MyEnum = A | B | C | D
```

will be represented by something like

```
A -> 00
B -> 01
C -> 10
D -> 11
```

3.3 Step into the Sequential Logic World

The previous parts are mainly talking about combinatorial logic; how about the sequential one?

In Clash, sequential logic things are wrapped into a type `Signal (dom :: Symbol) a`. The `dom` stands for the signal domain, which provides some basic configurations such as clock, reset, enable, frequency and etc. The default domain is `System`, which is the standard global domain. It is also possible to define domains on your own and setup some multiple clock domains; these advanced features are not used in this book.

Signal is not Monad, but it provides the interfaces of Functor and Applicative.

To check the content of the signal, you can use a function called `sampleN`, to sample several signals.

To test the signals, you can also use `enableGen` to generate enable signals, `clockGen` to generate clock signals and `resetGen` to generate reset signals. What's more, these is also a `simulate` function, which allows you to use a syntax like

```
simulate @System myInterface
```

to generate the result.

Usually, enable, clock and reset signals are required everywhere within a synchronous circuit. Image writing these three signals repeatedly at every function, it will definitely become tedious. Hence, Clash provides a special way to define a generalized signal domain which hides some global signals, you can simply expose them at those interfaces to synthesize. The following example illustrates how to use this feature

```
1 example :: HiddenClockResetEnable dom
2     => Signal dom Bool
3     -> Signal dom Bool
```

```
4  example = fmap complement
5
6  example'
7      :: Clock System
8      -> Reset System
9      -> Enable System
10     -> Signal System Bool
11     -> Signal System Bool
12 example' = exposeClockResetEnable example
```

3.4 Function Utilities

3.4.1 ROM and RAM

Clash provides some predefined functions for us to define large block of memory.

Asynchronous Memory

Let us first have a look at the asynchronous ROM and RAM,

```
asyncRom :: (KnownNat n, Enum addr)
  => Vec n a  -- ^ initial vector
  -> addr     -- ^ read address
  -> a        -- ^ read result

asyncRam
  :: (  Enum addr, KnownDomain dom
     ,  GHC.Classes.IP (AppendSymbol dom "_clk") (Clock dom)
     ,  GHC.Classes.IP (AppendSymbol dom "_en") (Enable dom)) => SNat n
     -> Signal dom addr                -- ^ read address
     -> Signal dom (Maybe (addr, a))   -- ^ write data
     -> Signal dom a                   -- ^ read result
```

The asynchronous version will output the content in the read address at the same clock cycle. If the read address and the write address conflicts, the default strategy is write-

after-read; however, you can use `readNew . asyncRam` to apply the read-after-write strategy.

Synchronous Memory

Asynchronous memory is handy enough, but it is not the optimal structure: asynchronous memory may require a lot of LUTs in FPGA and the cost must be considered if the memory size if relatively large. Fortunately, there is a synchronous version of RAM, it corresponds to the BRAM structure in FPGA. However, there is a big difference that the read and write operation issued in the current clock cycle will generate outcome in the next cycle; we must take care of this feature when designing circuits.

```
blockRam
  :: (  KnownDomain dom
     ,  GHC.Classes.IP (AppendSymbol dom "_clk") (Clock dom)
     ,  GHC.Classes.IP (AppendSymbol dom "_en") (Enable dom), NFDataX a
     ,  Enum addr )
     => Vec n a
     -> Signal dom addr
     -> Signal dom (Maybe (addr, a))
     -> Signal dom a
```

Similarly, you can change the read-write conflict resolution.

As for asynchronous ROM and synchronous RAM, Clash also provide some functions like `blockRamFile`, which will be translated into some Verilog code using `readmemb` function; which is handy for us to initialize the memory field using external files.

3.4.2 State Machine

There are several ways to handle to stateful procedures.

Register

Register is the basic state machine; it takes a input as the new state and outputs the previous state.

```
register
  :: (  KnownDomain dom
     ,  GHC.Classes.IP (AppendSymbol dom "_clk") (Clock dom)
     ,  GHC.Classes.IP (AppendSymbol dom "_rst") (Reset dom)
     ,  GHC.Classes.IP (AppendSymbol dom "_en") (Enable dom)
     ,  NFDataX a )
     => a
     -> Signal dom a
     -> Signal dom a

register 1 -- ^ declare a register with initial value 1
```

Mealy

Mealy Machine is a sort of state machine whose output is determined by the current input and state.

```
mealy
  :: (  KnownDomain dom
     ,  GHC.Classes.IP (AppendSymbol dom "_clk") (Clock dom)
     ,  GHC.Classes.IP (AppendSymbol dom "_rst") (Reset dom)
     ,  GHC.Classes.IP (AppendSymbol dom "_en") (Enable dom), NFDataX s )
     => (s -> i -> (s, o))
     -> s
     -> Signal dom i
     -> Signal dom o
```

Let us write a special counter using Mealy Machine: if the outside provides a input, it will set the counting value for the next state, otherwise, it just increase the counter and output the current value.

It seems that we can describe the state transformation with the following function:

```
counterT
  :: Unsigned 32
  -> Maybe (Unsigned 32)
```

```
  -> (Unsigned 32, Unsigned 32)
counterT state Nothing  = (state + 1, state)
counterT state (Just x) = (x    , state)
```

To transform `counterT` into a state machine, just apply the `mealy` function together with anz initial value to it

```
counter
  :: HiddenClockResetEnable dom
  => Signal dom (Maybe (Unsigned 32))
  => Signal dom (Unsigned 32)
counter = mealy counterT 0
```

Moore

Another kind of commonly used finite state machine is the Moore Machine, whose output is determined only by the current state.

```
moore
  :: ( HiddenClockResetEnable dom
     , NFDataX s )
     => (s -> i -> s)
     -> (s -> o)
     -> s
     -> (Signal dom i -> Signal dom o)
```

The counter example can also be transformed into a Moore Machine:

```
counterT
  :: Unsigned 32
  -> Maybe (Unsigned 32)
  -> Unsigned 32
counterT state Nothing  = state + 1
counterT _     (Just x) = x + 1

counter
```

```
  :: HiddenClockResetEnable dom
  => Signal dom (Maybe (Unsigned 32))
  -> Signal dom (Unsigned 32)
counter = moore counterT id 0
```

State Monad

It is also possible to

3.4.3 Bundle and Unbundle

Consider we have the following functions

```
foo
  :: HiddenClockResetEnable dom
  => Signal dom Input
  -> Signal dom (TypeA, TypeB)

bar
  :: HiddenClockResetEnable dom
  => Signal dom TypeA
  -> Signal dom TypeB
  -> Signal dom Result
```

The problem emerges when we want to combine this too functions: it is hard to take out the value of each part of the tuple wrapped in the signal environment.

Hence, we can use the `unbundle` function.

```
let (a, b) = unbundle $ foo input
in bar a b
```

The `bundle` function just does the inverse of `unbundle`:

```
a :: Signal System A
b :: Signal System B
func :: Signal System (A, B) -> Signal System C
```

```
res  :: Signal System C
res  =  func $ bundle (a, b)
```

Chapter 4

Writing the CPU

Now, we are fully armed with our equipment. It is the time to get our hands dirty and start implementing a very naive MIPS CPU.

4.1 What is CPU

CPU, the Central Process Unit, is the most important part of computers. It is in charge of the memory loading and storing, arithmetic operations and lots of other important functions. Most CPUs consist of a register called program counter, which record the current instruction position in the memory; in each cycle, the CPU fetches an instruction according to the program counter and start to handling a series of events encoded in the instruction.

The CPU we are going to implement consists of five parts:

1. Instruction Module: Fetch the instruction and maintain the value of the program counter.
2. Decode Module: Decode the instruction, determine the operations to be executed and get the value of the operands from the register file.
3. Arithmetic Unit: Execute the arithmetic operations, determine the branch targets and memory operations.
4. Memory Module: Load and store data from and into memory.

5. Write Back Module: Act as a transition module before register writing and branching.

4.2 Pipeline: Why and How

Although the functions of the CPU can be achieved within a single cycle: on each rising edge of the clock, we just fetch a new instruction and wait until all the required operations are finished, it is apparent that the cycle may become too long and inefficient.

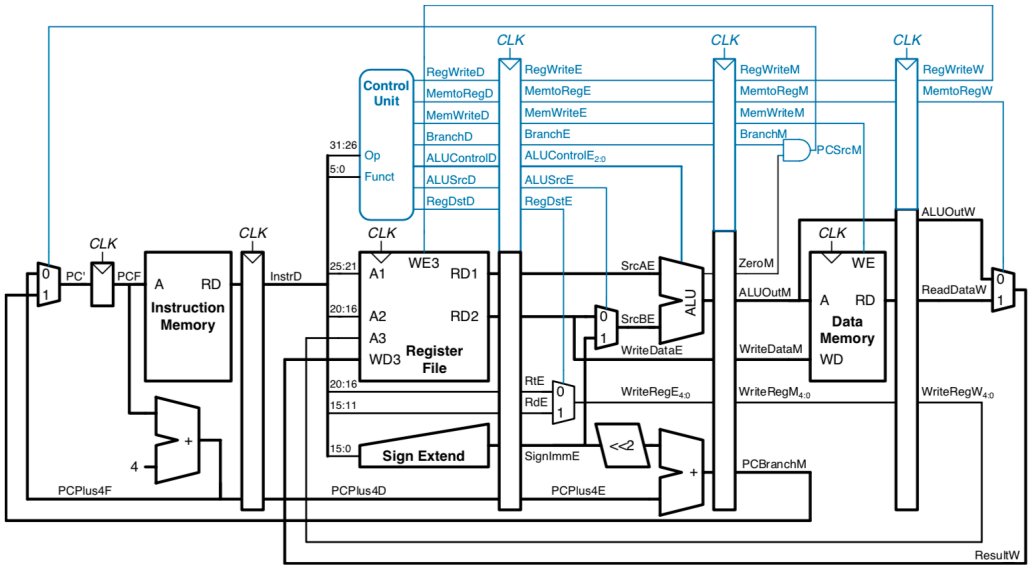

Figure 4.1: Classical MIPS pipelines (without any hazard handling)

Hence, many CPUs use a strategy called pipeline which split the execution into multiple stages. Each stage stores the previous input (the output from the predecessor stage) in its register, and when the clock rises, the CPU is handling multiple instructions at different stages. The efficiency is thus largely improved.

However, lots of new problems emerge because of the pipeline:

4.2.1 Control Hazards: Branch and Jump

Just consider the CPU shown in Figure 4.1. Without pipeline, a branch instruction triggers the branching in the single cycle and setup the new PC value within the same

cycle; the next instruction to execute will always be correct. However, in the pipeline model, if the branch condition fails, everything works fine; nevertheless, when the branch condition checks and the instruction reaches the Write-Back stage, there will be three invalid instructions already accumulated in the pipeline. We must find a way to flush those invalid ones.

This is actually easy, we just check the branch signal; if the branch target is set, we send a signal to all four modules before Write-Back and ask them to clear their instructions. **Notice that the memory writing should be stopped immediately while the register writing should be allowed**, because there are jumping instructions like `jal` that will store the current PC value into a register and this operation is always valid as it arrives at the Write-Back part at the same time of the branch target.

In most CPUs, the branch cost can be further reduced by apply branch prediction and moving branching checking into an earlier stage. Here, we are not going to care about these strategies and just handle the control hazards with stalling.

4.2.2 Data Hazards

There are several kinds of data hazards:

1. Consecutive arithmetic operations with data dependency:

```
addi $a0, $a0, 1
addi $a0, $a0, 1
addi $a0, $a0, 1
```

   Notice that when the second/third instruction reaches the Arithmetic Module, the first/second instruction is at the Memory Part/Write-Back Part; which means the write-back operation is not applied. Hence, we must figure out a way to forward the result stored in Memory Part/Write-Back Part to Arithmetic Module in advance.

2. Arithmetic Operation followed by SW:

```
addi $a0, $a0, 1
sw   $a0, -4($sp)
```

In the SW case, the value of register a0 is not ready. We can simplify the handling by moving determination of write value into Arithmetic Part. Hence, using the strategy of forwarding, we can also handle this problem.

3. LW followed by Arithmetic Operation

```
lw   $t0, 0($sp)
addi $t0, $t0, 1
```

In the pipeline shown in Figure 4.1, the load value will not be ready even with forwarding. In this case, we must stall the pipeline: keep the PC value, the state of Decode Module and the Arithmetic Module unchanged (but accepting register write-back) and insert a NOP state into memory module for the next cycle while handling the current operation of loading. Therefore, in the next cycle, the second instruction is still at the Arithmetic Module while the load result reaches the Write-Back Part which makes it possible to be forwarded to ALU.

4.2.3 Special Changes

We made several special changes to make life easier:

1. Because we are using the BRAM structure, which means that there is one cycle delay before we can get the real data, we need to forward the ALU results to the Memory Module in the same cycle so that in the next cycle the Memory Output is exactly what the instruction in the Memory Module requires.

```
j    SOME_PLACE
sw   $zero, 4($sp)
sw   $zero, 4($sp)
```

Previously, when the branch target reaches Write-Back Part, the Memory Operation hasn't taken place; however, in our case, the stall caused by branching will only prevent the second Memory Operation after branching; we need to also check the branching before we want to write something into the memory. Fortunately, this is always available since, when the instruction right after branching arrives ALU, the branching instruction is exactly at the Memory Part, we can simply check whether the branching target is set or not.

2. There is no need to care about the Load Data Hazards as we will also forward the data together with the memory fetching result. (Thanks to BRAM delay, the extra cost is little)

4.3 Implementation

4.3.1 Instruction Module

Instruction Set

We will only implement very a small set of MIPS instructions. Let's first write some functions to help us decode the instruction:

```
-- | The Format of MIPS Instructions
data Format
  = NoType              -- ^ NoType (Specialized for NOP Instruction)
  | RType               -- ^ R-Type Instruction Format
      (BitVector 6)     -- ^ Operation Code
      (BitVector 5)     -- ^ Register S
      (BitVector 5)     -- ^ Register T
      (BitVector 5)     -- ^ Register D
      (BitVector 5)     -- ^ Extra Infomation for Shifting Amount
      (BitVector 6)     -- ^ Function Code
  | IType               -- ^ I-Type Instruction Format
      (BitVector 6)     -- ^ Operation Code
      (BitVector 5)     -- ^ Register S
      (BitVector 5)     -- ^ Register T
      (BitVector 16)    -- ^ Immediate Value
  | JType               -- ^ J-Type Instruction Format
      (BitVector 6)     -- ^ Operation Code
      (BitVector 26)    -- ^ Jump Target
  deriving Show
```

We will first recognize the instruction format and then transform it into each recognized instruction.

```
decodeFormat :: BitVector 32 -> Format
decodeFormat 0 = NoType
decodeFormat vec =
    let opcode = slice d31 d26 vec
    in case opcode of
       0 ->
            pure RType
                <*> (slice d31 d26)
                <*> (slice d25 d21)
                <*> (slice d20 d16)
                <*> (slice d15 d11)
                <*> (slice d10 d6)
                <*> (slice d5 d0)
                 $  vec
      code | code == 0b000010 || code == 0b000011 ->
            pure JType
                <*> (slice d31 d26)
                <*> (slice d25 d0)
                 $ vec
      _ ->
            pure IType
                <*> (slice d31 d26)
                <*> (slice d25 d21)
                <*> (slice d20 d16)
                <*> (slice d15 d0)
                 $  vec
```

The format decode is trivial:

- An all-zero instruction is a NOP;
- Otherwise, instructions with 0 opcode will be dispatched into R-Format;
- Instructions with special jumping opcode will be dispatched into J-Format;

- Other instructions are in the I-Format

After format is determined, we can then transform instructions into our inner forms:

```
type Register = Unsigned 5
data Instruction
    = NOP
    | ADD Register Register Register
    | ADDI Register Register (Signed 16)
    | ADDU Register Register Register
    | ADDIU Register Register (Unsigned 16)
    | SUB Register Register Register
    | SUBU Register Register Register
    | AND Register Register Register
    | ANDI Register Register (BitVector 16)
    | NOR Register Register Register
    | OR Register Register Register
    | ORI Register Register (BitVector 16)
    | XOR Register Register Register
    | XORI Register Register (BitVector 16)
    | BEQ Register Register (Signed 16)
    | BNE Register Register (Signed 16)
    | SLT Register Register Register
    | SLTI Register Register (Signed 16)
    | SLTU Register Register Register
    | SLTIU Register Register (Unsigned 16)
    | LW Register Register (Signed 16)
    | SW Register Register (Signed 16)
    | SLL Register Register (Unsigned 5)
    | SRL Register Register (Unsigned 5)
    | SRA Register Register (Unsigned 5)
    | SLLV Register Register Register
    | SRLV Register Register Register
    | SRAV Register Register Register
```

```
    | J (Unsigned 26)
    | JAL (Unsigned 26)
    | JR (Unsigned 5)
    deriving Show
    deriving Generic
    deriving NFDataX
```

We do not provide the function

```
decodeTyped :: Format -> Instruction
```

here as it should be easy to write and just require some repeated works to recognize the instruction based on the opcode and the function code.

Just mention a small trick: to reduce the repeated code, you can use the Applicative property of Readers (or, more precisely, the isomorphic function structures of Reader):

```
t1 (x, _, _) = x
t2 (_, y, _) = y
t3 (_, _, z) = z
makeType func = pure func <*> unpack . t1 <*> unpack . t2 <*> unpack . t3
-- then you can just use something like:
case fn of
    0b100000 -> (makeType ADD) (rs, rt, rd)
    0b100001 -> (makeType ADDU) (rs, rt, rd)
    0b100100 -> (makeType AND) (rs, rt, rd)
    -- more cases ...
```

The Instruction RAM

We will use BRAM to implement the instruction memory space.