ercesiMIPS Lab Manual

A guide to Multi Cyclic CPU with 7-9(11) MIPS Instructions

[Contents]

click on it

Authors: Jianfeng An, Meng Zhang & Danghui Wang

CS 11007 Computer Organization and Architecture
(Spring, 2017)

Northwestern Polytechnical University, China
Faculty of Computer Science
ERCESI

27 April 2017

"SWEET JEZUS WHY!?"

If you're reading this, chances are that either your arms are suffering from anemia in waiting for the TA's to finish helping the 20 people in the room, or you're trying to design in Chisel without any TA's whatsoever, and you just wanna end it all. Fret not! This document serves as a specification for basic explanations of single cyclic MIPS CPU with 7-11 instructions supporting, also a helpful links and forbidden deigning secrets whose names none dare speak.

Contents

1   Overview

The structure of Multi cyclic MIPS CPU has been introduced in CS 11007 class lecture.

- **Supported Instructions** includes *sub, add, ori, lw, sw, beq, j*, or adding *addi, and, andi, or, slt* for better programming experience in assembly. All these instructions can be supported without exception detecting (overflow detecting)

- **These instructions work in different multi cycles.** A very basic concept adopted here is that the multi cycles are controlled by FSM (Finite State Machine).

- **Consisted of Data Path, Control Unit and Memory Unit.** To illustrate the typical systematic idea of computer, we recommend you design your first CPU with two separated modules, CPath and DPath, in such coding style, both blocks can also be easily verified separately. Additionally, if more complement MIPS ISA is chosen, this structure will be high efficient to be extended.

- **Chisel3 is also recommended.**

2 Specification

A typical multi cyclic CPU core can also be consisted of Control Unit and Data Path unit. Although there is no big picture for a whole multi cyclic CPU, in our lecture 12, Fig. 1 still can be figured out by composing several slides.
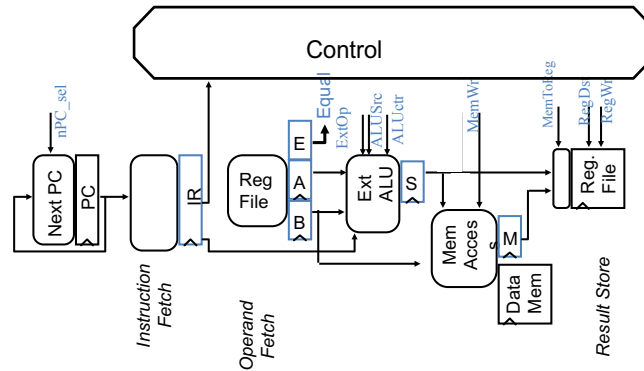

Figure 1: Multi Cyclic CPU Block Diagram

Two Module classes are still recommended, Control Unit and Data Path Unit respectively, as shown in following Fig. 2. To simply the design, Harvard structure is still adopted in multi cyclic CPU lab, a MIPS adaptive memory model (Princeton Structure) is mandatory in further pipeline structure. In Fig. 2, a **red** marked signal bus, Dec_Inst[31:0], is recommend as a IR
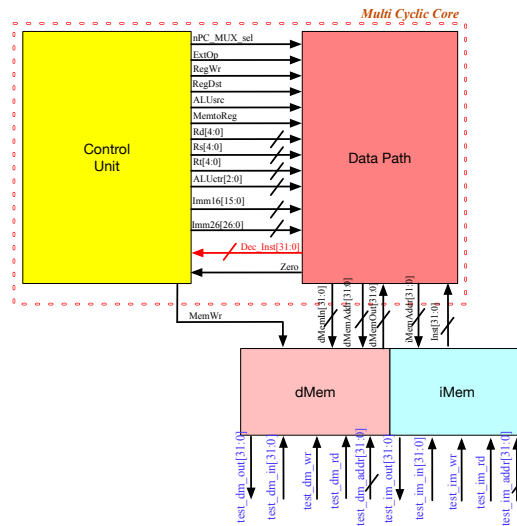

Figure 2: Multi Cyclic CPU Block Diagram

output for FSM Decode state. Almost all signals can be renamed by your self except those evaluation required singles specified in Section 4.2.

4

3 Chisel3 101

3.1 FSM in Chisel

TBD

4   Evaluation Requirement

We (or TAs) will check and evaluate your design with a pre-coded test program, in which a pseudo-random instruction queue will be feed into you instruction memory and data memory. To ensure this test works correctly, the test signal interfaces name **MUST** meets the requirements shown in Table 1.

4.1   Test Procedure

As Operation System is still not available for this stage, the program and initial data are feed into Instruction Memory and Data Memory from the Tester program respectively. By utilizing `ori` instruction, the Tester program provides an alternative data memory initialization, `$t = $zero | imm16` (`ori, rt, $zero Imm16`). In which, `$t` stores the result of *or* operation (immediate "or" zero register), i.e. the immediate is loaded into register file by `ori` instruction.
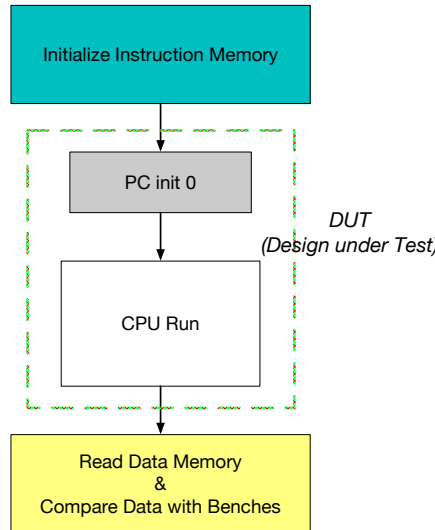

Figure 3: Test Procedure

Fig. 3 shows the whole test procedure. In which, the designed CPU is denoted by DUT (Design under Test), the instruction is feed into Imem at the initialization stage, and the data about how program running is collected at the end.

4.2   Signals for Test mode

The recommended Top block structure is shown in Fig. 4, which the blue marked interfaces are connected with testbench. So please define these interfaces in your top module as exactly the same as Table 1 shown.

Additionally, a `valid` signal is highly recommended. It can be monitored to find whether CPU works correctly. As shown in Table 1, `valid` will be set to 0, if the fetched instruction is not defined in our 7-Inst ISA (e.g. `0xFFFFFFFF`), which is to tell test the CPU is working in exception.

Table 1: Signals Definition for Test Mode

| Signal Name	Direction	Width	Function
boot	Input	1-bit	Trigger the boot test mode, set to 0 in CPU regular process mode
test_im_wr	Input	1-bit	Instruction memory write enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if writing instructions to imem, otherwise it is set to 0.
test_im_re	Input	1-bit	Instruction memory read enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if reading instructions out, otherwise it is set to 0.
test_im_addr	Input	32-bit	Instruction memory address
test_im_in	Input	32-bit	Instruction memory data input for test mode.
test_im_out	Output	32-bit	Instruction memory data output for test mode.
test_dm_wr	Input	1-bit	Data memory write enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if writing data to dmem, otherwise it is set to 0.
test_dm_re	Input	1-bit	Data memory read enable in test mode,set to 0 in CPU regular process mode. In test mode, it will be set to 1 when if reading data out, otherwise it is set to 0.
test_dm_addr	Input	32-bit	Data memory address
test_dm_in	Input	32-bit	Data memory input for test mode.
test_dm_out	Output	32-bit	Data memory output for test mode.
valid	Output	1-bit	If CPU stopped or any exception happens, valid signal is set to 0.
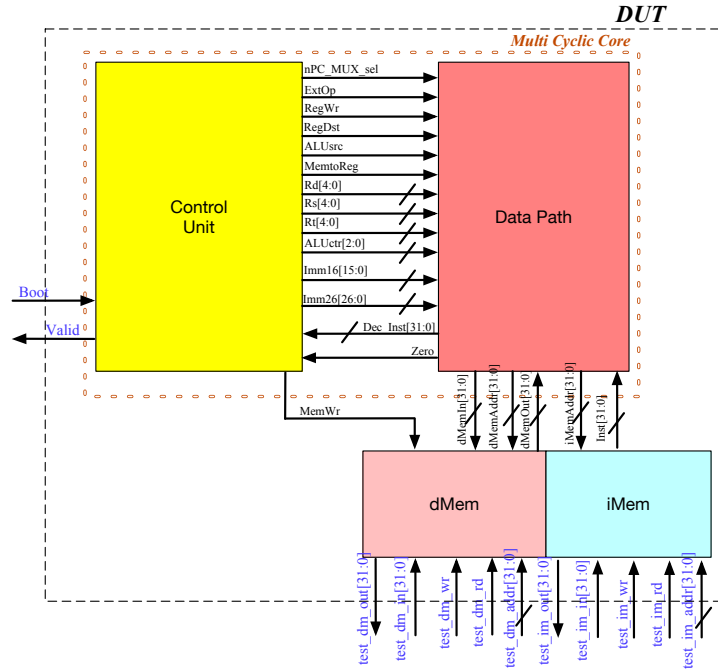
Figure 4: Top Block Diagram for Test Mode

4.3 Basic Logic for Test

Please add the test logic into your top module as shown in Table 2.
The `Top` Module is presented below, please **DO NOT** change any code of Top module:

```
package MultiCycle

import chisel3._
import chisel3.iotesters.Driver
//import utils.ercesiMIPSRunner
class TopIO extends Bundle() {
  val boot = Input(Bool())
// imem and dmem interface for Tests
  val test_im_wr   = Input(Bool())
  val test_im_rd   = Input(Bool())
  val test_im_addr = Input(UInt(32.W))
  val test_im_in   = Input(UInt(32.W))
  val test_im_out  = Output(UInt(32.W))

  val test_dm_wr   = Input(Bool())
  val test_dm_rd   = Input(Bool())
  val test_dm_addr = Input(UInt(32.W))
  val test_dm_in   = Input(UInt(32.W))
  val test_dm_out  = Output(UInt(32.W))

  val valid      = Output(Bool())
}

class Top extends Module() {
```

```scala
val io    = IO(new TopIO())//in chisel3, io must be wrapped in IO(...)
val cpath = Module(new CtlPath())
val dpath = Module(new DatPath())

cpath.io.ctl <> dpath.io.ctl
cpath.io.dat <> dpath.io.dat
io.valid := cpath.io.valid
cpath.io.boot := io.boot

val imm = Mem(256, UInt(32.W))
val dmm = Mem(1024, UInt(32.W))
io.test_dm_out := 0.U
io.test_im_out := 0.U
cpath.io.Inst := 0.U
when (io.boot && io.test_im_wr){
  imm(io.test_im_addr >> 2) := io.test_im_in
  cpath.io.Inst := 0.U
 }
when (io.boot && io.test_dm_wr){
  dmm(io.test_dm_addr >> 2) := io.test_dm_in
  cpath.io.Inst := 0.U
}
when (io.boot && io.test_im_rd){
  io.test_im_out := imm(io.test_im_addr >> 2)
  cpath.io.Inst := 0.U
}
when (io.boot && io.test_dm_rd){
  io.test_dm_out := dmm(io.test_dm_addr >> 2)
  cpath.io.Inst := 0.U
}
when (!io.boot){
  cpath.io.Inst := Mux(io.boot, 0.U, imm(dpath.io.imem_addr >> 2))
  dpath.io.dmem_datOut := dmm(dpath.io.dmem_addr >> 2)
  when (cpath.io.MemWr) {
    dmm(dpath.io.dmem_addr >> 2) := dpath.io.dmem_datIn
  }
}
}
```

Table 2: Logic Operation in Test Mode

Operation	Signal	Boolean Logic
Imem Write	boot	1
	test_im_wr	1
	test_im_rd	0
	test_dm_wr	0
	test_dm_rd	0
Dmem Write	boot	1
	test_im_wr	0
	test_im_rd	0
	test_dm_wr	1
	test_dm_rd	0
Imem Read	boot	1
	test_im_wr	0
	test_im_rd	1
	test_dm_wr	0
	test_dm_rd	0
Dmem Read	boot	1
	test_im_wr	0
	test_im_rd	0
	test_dm_wr	0
	test_dm_rd	1