ercesiMIPS Lab Manual

A guide to Single 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 Single cyclic MIPS CPU has been introduced in CS 11007 class lecture.

- **Supported Instructions** includes *sub, add, or, ori, lw, sw, lst, beq, j*, or adding *addi, and, andi* for better programming experience in assembly. All these instructions can be supported without exception detecting (overflow detecting)
- **All instructions work in one cycle.** For the very beginning stage, Single Cyclic CPU model is a great example to explain how CPU works.
- **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.** Chisel is a powerful structural hardware description language, with more efficient expression for block, operation, and IO bundles compared with Verilog. However, the most significant feature of Chisel is that it can express the structure of system without detailed circuits coding. Further more, we prefer Chisel3 instead of Chisel2, which relies on verilator for verilog simulations instead of Synopsys vcs. The difference between these tow versions can be referenced here: `https://github.com/ucb-bar/chisel3/wiki/Chisel3-vs-Chisel2`.

2 Specification

A typical single cyclic CPU core is consisted of Control Unit and Data Path unit. In lecture 10 11, the whole system is illustrated in Fig. 1.

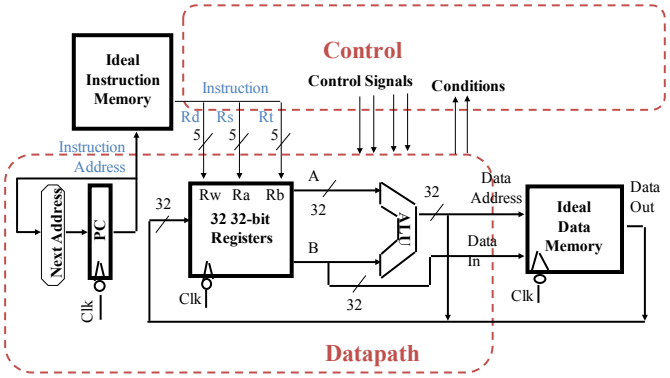

Figure 1: Single Cyclic CPU Block Diagram

Two Module classes are recommended for Control Unit and Data Path Unit respectively, as shown in following Fig. 2. Although the Harvard structure is adopted now to simplify the memory control and initialization, a MIPS adaptive memory model (Princeton Structure) is mandatory in further MIPS program test (for Multi-cyclic CPU Lab). In Fig. 2, almost all signals

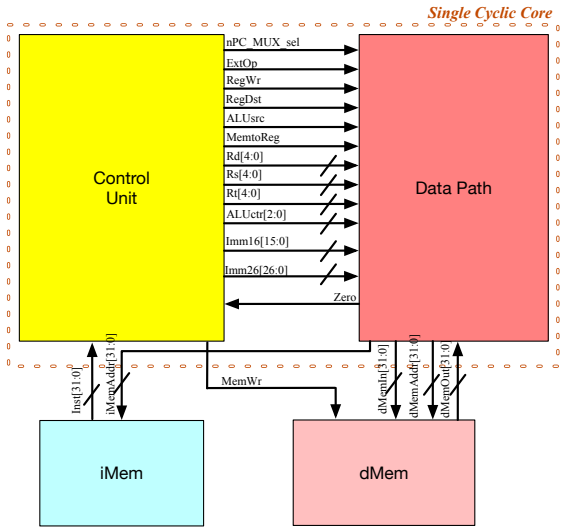

Figure 2: Single Cyclic CPU Block Diagram

can be renamed by your self except those evaluation required singles specified in Section 4.2.

3 Chisel3 101

3.1 Files, Directory and `package`

All designed files should contain `package` definition at the very beginning position for `sbt` interpreting.

```
package SingleCycle
```

If the package name is re-defined by you own, please check all files to revise the package name. Theoretically, the files with correct package definition could be placed anywhere of `src` directory. We preferred `src/main/scala/SingleCycle` for the design files and `src/test/scala/SingleCycle` for the tests files.

3.2 Define Ports

In single cycle ercesiMIPS project, bunches of singles connect Control Unit and Data Path Unit, which are only with different direction in different **Module**. Simply wrap the object in an `Input()` or `Output()` function. A example of `CtltoDatIo` is defined below.

```
class CtltoDatIo extends Bundle()
{
  val nPC_MUX_sel = Output(Bool())
  val RegWr   = Output(Bool())
  val RegDst    = Output(Bool())
  val ExtOp   = Output(Bool())
  val ALUctr    = Output(UInt(2.W))
  val ALUsrc    = Output(Bool())
  val MemtoReg  = Output(Bool())
  val Rd      = Output(UInt(5.W))
  val Rt      = Output(UInt(5.W))
  val Rs      = Output(UInt(5.W))
  val Imm16     = Output(UInt(16.W))
  val Imm26   = Output(UInt(26.W))
}
```

To make the definition efficiently, the signals can be instanced in both modules, as shown below:

```
class CPathIo extends Bundle()
{
  val Inst    = Input(UInt(32.W))
  val resetSignle = Input(Bool())
  val MemWr   = Output(Bool())
  val valid   = Output(Bool())
  val ctl     = new CtltoDatIo()
  val dat     = new DatToCtlIo().flip()
}

class CtlPath extends Module()
{
  val io      = IO(new CPathIo ())
  ...
}
```

```
class DatToCtlIo extends Bundle()
{
  val cmp_out = Output(Bool())
}
class DPathIo extends Bundle()
{
  //val host    = new HTIFIO()
  val imem_addr = Output(UInt(32.W))
  val dmem_addr = Output(UInt(32.W))
  val dmem_datIn  = Output(UInt(32.W))
  val dmem_datOut = Input(UInt(32.W))
  val ctl     = new CtltoDatIo().flip()
  val dat     = new DatToCtlIo()
}

class DatPath extends Module {
  val io = IO(new DPathIo ())
  ...
```

```
}
```

In these examples, the connected IOs in different modules are instanced with `flip()` function for reversing its direction.

3.3 Define a Module

To object the hierarchical mechanism, modules defined in Chisel are described as a type of new **Module** class, shown as below.

```
class DatPath extends Module {
  val io = IO(new DPathIo ())
  // Internal Signal
  val BusA  = Wire(UInt(32.W))
  val BusB  = Wire(UInt(32.W))
  ......
  ......
}
```

In which, a user-defined Data Path module, with module name `DatPath`, is defined as a *class* which:

- inherits from Module,
- contains an interface wrapped in an `IO()` function and stored in a port field named io, and
- wires together subcircuits in its constructor.

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, and the logic operation of these interfaces **MUST** be design as

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.

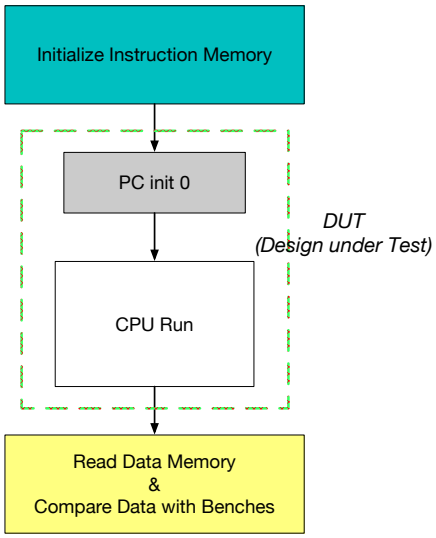

Figure 3: Test Procedure

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.

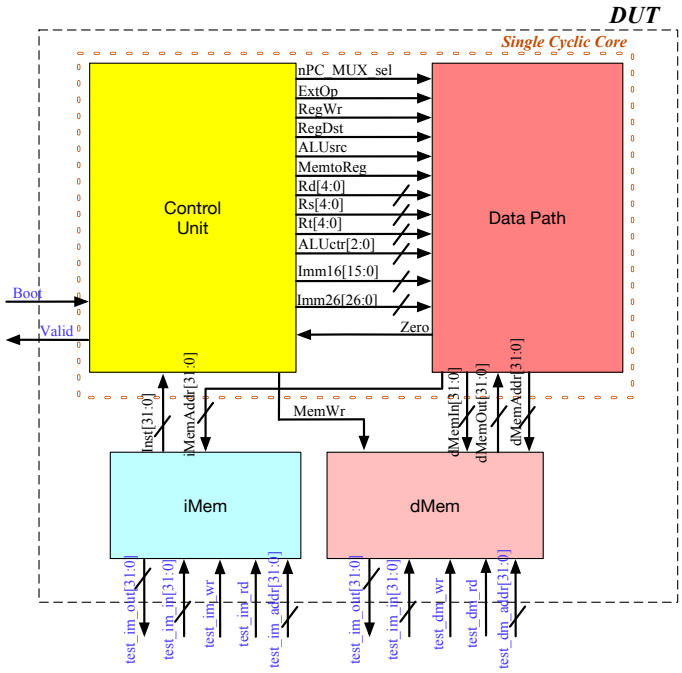

Figure 4: Top Block Diagram for Test Mode

4.3 Basic Logic for Test

Please add the test logic into your top module as shown in Table 2. An example `Top` Module is presented below:

```
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() {
  val io    = IO(new TopIO())//in chisel3, io must be wrapped in IO(...)
  //...
  when (io.boot & io.test_im_wr){
    imm(io.test_im_addr) := io.test_im_in
    } .elsewhen (io.boot & io.test_dm_wr){
    // please finish it
    } //...
}
```

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