在本课程班级名单中排名序号 84

**南京航空航天大学**

**计算机组成原理实验报告**

|  |  |
| --- | --- |
| 题 目 | 单周期MIPS CPU设计与仿真验证 |

|  |  |
| --- | --- |
| 学生姓名 | 谢骏鑫 |
| 学 号 | 162110132 |
| 学 院 | 16院 |
| 专 业 | 计算机科学与技术 |
| 班 级 | 1621101 |
| 课程教师 | 施慧彬 |

二〇二三年 六月

**南京航空航天大学**

**实验报告诚信承诺书**

本人郑重声明：所呈交的报告是本人独立进行研究所取得的成果。尽本人所知，除了特别加以标注引用的内容外，本报告不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名：谢骏鑫 2023年 7 月 1 日

（学号）：162110132

单周期MIPS CPU的设计与仿真验证

摘 要

通过对计算机组成原理课程的学习，融会贯通所学的知识，独立设计出可以运行实验指导书中要求的36条MIPS指令的单周期CPU，不支持溢出和异常处理，提升自己的代码能力，加强自己对计算机组成原理和CPU架构和设计原理的深刻理解。

关键词**：**计算机组成原理，MIPS，单周期CPU

Design and simulation verification of single-cycle MIPS CPU

Abstract

Through the study of the Computer Composition Principle course, I can integrate the knowledge I have learned and independently design a single-cycle CPU that can run the 36 MIPS instructions required in the experiment guide without overflow and exception handling, so as to improve my code ability and strengthen my deep understanding of the computer composition principle and CPU architecture and design principle.

KEYWORDS**：**Principles of computer composition，MIPS，Single-cycle CPU

目录

[第一章 引言 1](#_Toc139119848)

[1.1 单周期CPU总电路设计图 1](#_Toc139119849)

[1.2 单周期CPU各模块定义 1](#_Toc139119850)

[1.2.1 PC模块 1](#_Toc139119851)

[1.2.2 pc2addr模块 2](#_Toc139119852)

[1.2.3 npc模块 2](#_Toc139119853)

[1.2.4 im\_4k模块 4](#_Toc139119854)

[1.2.5 decode模块 4](#_Toc139119855)

[1.2.6 extend模块 5](#_Toc139119856)

[1.2.7 registers模块 6](#_Toc139119857)

[1.2.8 ALU模块 8](#_Toc139119858)

[1.2.9 mux\_alusrc模块 9](#_Toc139119859)

[1.2.10 addr2dm模块 10](#_Toc139119860)

[1.2.11 dm\_4k模块 10](#_Toc139119861)

[1.2.12 mux\_memtoreg模块 11](#_Toc139119862)

[1.2.13 Control模块 11](#_Toc139119863)

[1.2.14 mips模块 13](#_Toc139119864)

[1.3 单周期CPU各个指令控制信号取值表图 14](#_Toc139119865)

[1.4 单周期CPU各个指令的设计逻辑 14](#_Toc139119866)

[1.4.1 R型指令 15](#_Toc139119867)

[1.4.2 I型指令 17](#_Toc139119868)

[1.4.3 J型指令 20](#_Toc139119869)

[第二章 代码实现和运行结果验证 20](#_Toc139119870)

[2.1 具体Verilog代码实现 20](#_Toc139119871)

[2.1.1 PC模块 20](#_Toc139119872)

[2.1.2 pc2addr模块 21](#_Toc139119873)

[2.1.3 npc模块 22](#_Toc139119874)

[2.1.4 im\_4K模块 24](#_Toc139119875)

[2.1.5 decode模块 25](#_Toc139119876)

[2.1.6 extend模块 26](#_Toc139119877)

[2.1.7 registers模块 27](#_Toc139119878)

[2.1.8 ALU模块 29](#_Toc139119879)

[2.1.9 mux\_alusrc模块 33](#_Toc139119880)

[2.1.10 addr2dm模块 34](#_Toc139119881)

[2.1.11 dm\_4k模块 34](#_Toc139119882)

[2.1.12 mux\_memtoreg模块 35](#_Toc139119883)

[2.1.13 Control模块 36](#_Toc139119884)

[2.1.14 mips模块 45](#_Toc139119885)

[2.1.15 tmips模块（testbench） 48](#_Toc139119886)

[2.2 ModelSim上模拟仿真测试指令的信号结果 48](#_Toc139119887)

[第三章 总结与展望 52](#_Toc139119888)

[参考文献 55](#_Toc139119889)

[附录 56](#_Toc139119890)

[附录1 单周期CPU测试所用的指令程序 56](#_Toc139119891)

[附录2 实现的36条MIPS指令解析表图 60](#_Toc139119892)

# 第一章 引言

## 1.1 单周期CPU总电路设计图

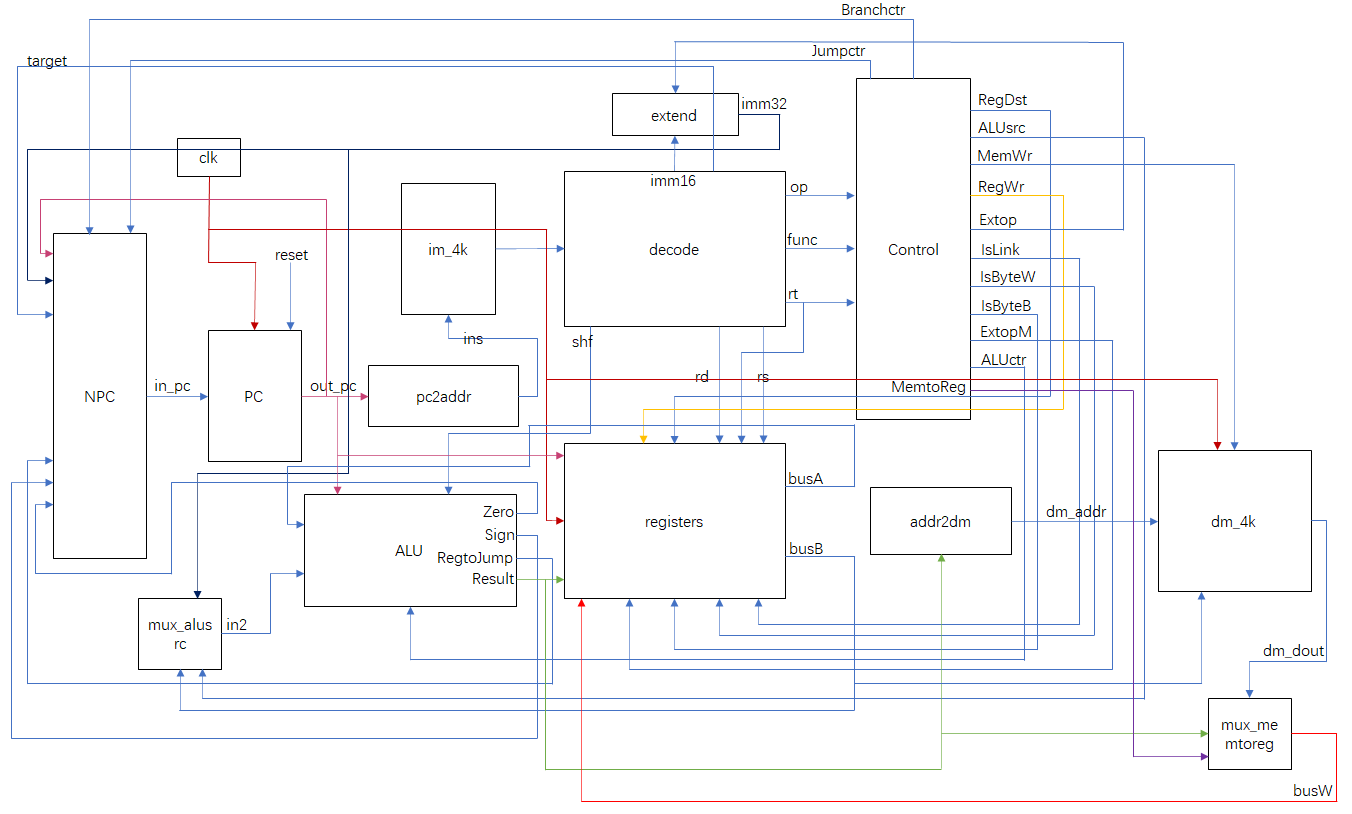


图1.1 单周期CPU总电路设计图

图1.1展示了我这次计算机组成原理实验大作业设计的单周期CPU总的电路逻辑设计图，可以看到，各个模块之间的联系以及信号之间的传递在图中均非常清楚地体现出来了（具体浏览时可以放大更清楚地观察到具体连线细节）。

## 1.2 单周期CPU各模块定义

### 1.2.1 PC模块

（1）基本描述

在我的单周期CPU设计逻辑中，PC模块的作用是来选择下一条需要取指令的PC值，从功能接口上来看，它的作用就是传入in\_pc,在时钟clk上升沿更新pc作为out\_pc输出进行下一条指令的取指并且运行。而in\_pc的计算逻辑主要在npc模块中进行。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| in\_pc[31:0] | I | 传入的PC值用来更新计算下一条的PC值 |
| clk | I | 时钟信号 |
| reset | I | 复位信号，用来将PC值复位更新为0（使用扩展要求，其实是更新为0x00003000） |
| out\_pc[31:0] | O | PC的输出，下一条指令的取指地址 |

### 1.2.2 pc2addr模块

（1）基本描述

正如模块名字一样，将32位PC值转换为可以从指令存储器读取对应指令的10位地址值，这是由于指令存储器im\_4k的接口要求的，指令存储器是以一个字为一个单位存放，而PC值是按照字节进行读取，因此在取指令时需要先对PC值做一个转换才可以。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| out\_pc[31:0] | I | 传入的32位PC值 |
| addr[11:2] | O | 输出的10位地址值，能够从指令存储器的接口中正确读取传入PC对应的指令 |

### 1.2.3 npc模块

（1）基本描述

npc模块，就是所说的“下地址”逻辑模块，在指令的执行过程中，下一条指令的不可能只是通过PC+4计算得到的，还存在着分支跳转和J型的跳转指令，这些都需要重新计算下一条PC的地址值，具体就是在npc模块里面实现，这里传入了控制分支和跳转的控制信号以及相关判断的标志位，并根据实验测试要求的方式进行PC的重新拼接和赋值。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| out\_pc[31:0] | I | 传入的当前PC值，依赖这个计算下一条指令的PC |
| imm32[31:0] | I | I型指令解析出来的立即数字段并且扩展为32位的结果，用来进行分支跳转PC的计算 |
| target[25:0] | I | J型指令解析出来的target字段，用来进行直接跳转PC的计算 |
| RegtoJump[31:0] | I | 跳转寄存器指令的需要，传递的某个需要跳转的寄存器的值，用来更新PC值 |
| Jumpctr[1:0] | I | 跳转指令的控制信号，00控制不跳转，01控制类j指令的跳转，10控制类jr指令的跳转 |
| Branchctr[2:0] | I | 分支指令的控制信号，根据要实现的指令实现，主要控制等于、不等、大于等于0、大于0、小于等于0和小于0的分支跳转 |
| Zero | I | ALU计算得到的标志位信息，用来表明要跳转判断的两个数是否相等 |
| Sign | I | ALU计算得到的标志位信息，用来表明要跳转判断的两个数的大小关系 |
| in\_pc | O | 根据上述控制逻辑最终得到的下一条指令的PC地址，用来传递给PC模块得到取指PC |

### 1.2.4 im\_4k模块

（1）基本描述

指令存储器模块，按照大实验要求的指令存储器接口模块进行设计，具体的存储器是1024个字大小的，故只有10位的取指地址，这是经过pc2addr模块得到的结果，以及输出32位的取得的具体指令编码。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[11:2] | I | 10位取指地址 |
| dout[31:0] | O | 取得的具体的指令的编码 |

### 1.2.5 decode模块

（1）基本描述

译码模块，由于指令存储器的硬性接口的要求，特地安排了一个译码模块用来对取出来的指令按位数进行译码，对于具体的指令，依次按照规则解析出它的op、rs、rt、rd、shamt、func、imm16和target字段，这些字段将用于生成相对应的控制信号。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ins[31:0] | I | 传入的具体的32位指令编码 |
| op[5:0] | O | 解析出来的op字段 |
| rs[4:0] | O | 解析出来的rs字段 |
| rt[4:0] | O | 解析出来的rt字段 |
| rd[4:0] | O | 解析出来的rd字段 |
| shamt[4:0] | O | 解析出来的shamt字段 |
| func[5:0] | O | 解析出来的func字段 |
| imm16[15:0] | O | 解析出来的imm16字段 |
| target[25:0] | O | 解析出来的target字段 |

### 1.2.6 extend模块

（1）基本描述

扩展器模块，就是对立即数进行扩展的，I型指令中解析得到的立即数是16位的，参与运算的元素都是32位的，并且需要根据相关指令的需求决定是零扩展还是符号扩展，这是由控制信号Extop决定的。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| imm16[15:0] | I | 传入的16位立即数 |
| Extop | I | 控制是进行零扩展还是符号扩展 |
| imm32[31:0] | O | 输出的扩展完成的32位立即数 |

### 1.2.7 registers模块

（1）基本描述

非常核心和重要的一个模块——寄存器堆模块，因为所有指令归根结底无外乎就是读写寄存器，所以控制着整个CPU的32个寄存器的寄存器堆的设计显得很重要，同时又有较大的自主设计的灵活性。我在设计单周期CPU的时候，将所有相关的不同的读写寄存器类型都集中到了这个模块进行处理，并且通过传入相关的控制信号，来控制相关的读写逻辑，比如控制写哪个寄存器、是否需要更新链接需要的31号ra寄存器、是按字读还是按字节读、是按字写还是按字节写、是否需要对要写的字节进行符号扩展等。具体对应的逻辑可以见如下表格和控制单元的解析。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号，在时钟上升沿进行写寄存器的操作 |
| rs[4:0] | I | rs寄存器编号，读这个寄存器送出对应的值到busA |
| rt[4:0] | I | rt寄存器编号，读这个寄存器送出对应的值到busB,同时作为写目的寄存器的一个可选寄存器 |
| rd[4:0] | I | rd寄存器编号，写目的寄存器的一个可选寄存器 |
| RegDst | I | 控制信号，决定是写rt寄存器还是rd寄存器，为1时写rd寄存器，反之写rt寄存器 |
| RegWr | I | 寄存器写使能信号，控制是否可以写寄存器 |
| Is\_Link | I | 是否需要链接控制信号，为真时需要将当前的PC+4送入31号ra寄存器 |
| out\_pc[31:0] | I | 传入的当前的PC值用来实现链接更新ra寄存器的需要 |
| IsByteW | I | 控制信号，控制是否是按字节将内容写入寄存器，为真时表示按字节写入 |
| IsByteB | I | 控制信号，sb指令设计的信号，控制是否按字节读寄存器，为真时表示按字节读入 |
| ExtopM | I | 控制在按字节写寄存器的时候是否需要对写入的字节进行符号扩展，为真时表示需要进行符号扩展 |
| busW[31:0] | I | 要写入的寄存器的值 |
| dm\_addr[31:0] | I | sb指令设计的需要，sb指令需要将指定地址处的1位字节写入数据存储器，并且按小端方式且不覆盖，设计逻辑采取先将数据存储器对应地址的那一个字读出来并且根据具体的要存储的地址计算修改具体哪一位字节再写入存储器，这个接口就是负责计算出具体要修改的哪一位字节的 |
| busA[31:0] | O | 寄存器堆读出来的busA端口的值 |
| busB[31:0] | O | 寄存器堆读出来的busB端口的值 |

### 1.2.8 ALU模块

（1）基本描述

单周期CPU进行算术逻辑运算的核心单元，所有指令需要的各种运算都在这个单元中进行，支持对传入的数值进行无符号加法、无符号减法、高位拼接、小于比较置位、按位与、按位或非、按位或、按位异或、逻辑左移、逻辑右移、算术右移等运算。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ALUctr[3:0] | I | 控制信号，4位ALU运算控制信号，控制当前的ALU单元对传入的值进行何种运算 |
| in1[31:0] | I | ALU单元的输入值的1号端口 |
| in2[31:0] | I | ALU单元的输入值的2号端口 |
| shf[4:0] | I | R型指令中非变量的移位运算需要的shamt字段信息 |
| out\_pc[31:0] | I | 链接指令的需要将PC的值写入31号寄存器，因此传入这个值用来特殊的运算 |
| Zero | O | ALU单元运算生成的标志位信息，Zero为真表示传入的两个计算值相等，反之则不等 |
| Sign | O | ALU单元运算生成的标志位信息，Sign为真表示in1大于in2，反之则小于 |
| Result[31:0] | O | ALU单元对传入的in1和in2运算生成的结果 |
| RegtoJump[31:0] | O | 寄存器跳转的需要，需要读取寄存器的值，作为ALU的输出统一处理 |

### 1.2.9 mux\_alusrc模块

（1）基本描述

正如模块名字一样，这是一个二选一多路选择器模块，用来选择ALU算术逻辑运算单元第二个输入端口的值来源，即控制从寄存器堆读出来的busB参与运算，还是选择立即数参与运算。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| regs\_busB[31:0] | I | 从寄存器堆读取的busB的值 |
| ext\_imm32[31:0] | I | 经过扩展单元的立即数的值 |
| ALUsrc | I | 控制信号，用来控制选择哪个值参与ALU的运算 |
| ALU\_Bin[31:0] | O | 输出，最终选择的用于参与ALU运算的值 |

### 1.2.10 addr2dm模块

（1）基本描述

正如模块名字命名一样，这是将32位的地址转换为10位的可以用来读写数据存储器的地址，在指令执行逻辑中计算得到的地址都是32位的，并且是按照字节进行编址的，因此在按字存储的数据存储器中，需要取32位地址中的[11:2]位进行转换。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| in\_addr[31:0] | I | 传入的32位地址 |
| addr[11:2] | O | 输出可以读写数据存储器的10位地址 |

### 1.2.11 dm\_4k模块

（1）基本描述

数据存储器模块，接口的设计和大实验要求的一致，作为单周期CPU的数据存储单元，存储单元按照一个字（4个字节）为单位进行设计，大小一共1024个字，所以地址只需要10位，支持读存储单元的内容和在写使能的控制下将内容写进存储单元中去。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[11:2] | I | 传入的控制读写内容的地址 |
| din[31:0] | I | 要写入的一个字的内容 |
| we | I | 存储器写使能信号，控制是否可以将内容写进存储器 |
| clk | I | 时钟控制信号，在时钟上升沿检查是否可写 |
| dout[31:0] | O | 从数据存储器指定地址读出来的内容 |

### 1.2.12 mux\_memtoreg模块

（1）基本描述

正如模块名字所述，本模块是一个二选一多路选择器，来选择将哪一个值写入寄存器堆，传入信号MemtoReg控制是选择ALU单元运算的结果写入寄存器堆还是选择从数据存储器读出来的值写入寄存器堆。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ALU\_out[31:0] | I | 传入的ALU单元运算得到的结果 |
| datamem\_out[31:0] | I | 传入的从数据存储器dm读出来的值 |
| MemtoReg | I | 控制信号，为1时选择datamem\_out,否则选择ALU\_out |
| memtoreg\_out[31:0] | O | 输出，最终将被写入寄存器堆的值 |

### 1.2.13 Control模块

（1）基本描述

上述12个模块一同构成了单周期CPU的数据通路模块，本模块则是构成了单周期CPU的控制单元模块，此模块主要通过从取得的指令解析得到的结果作为传入参数，控制单元通过这些信息来生成对应指令所需要的相关控制信号，至于具体指令所对应的控制信号可见后续的内容。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| op[5:0] | I | 传入的指令op字段信息 |
| func[5:0] | I | 传入的指令func段信息，一般为R型指令所使用 |
| rt[4:0] | I | 传入的指令rt段信息，bgez和bltz指令区分的需要 |
| RegDst | O | 控制信号，控制写哪个寄存器 |
| ALUsrc | O | 控制信号，决定ALU单元使用哪个值进行运算 |
| MemtoReg | O | 控制信号，控制从数据存储器读出的值是否需要写寄存器 |
| RegWr | O | 控制信号，寄存器写使能，控制是否可以写寄存器堆 |
| MemWr | O | 控制信号，存储器写使能，控制是否可以写数据存储器 |
| Extop | O | 控制信号，控制是否需要对16位立即数作符号扩展 |
| ExtopM | O | 控制信号，实现lb指令的需要，控制对取出来的字节是否作符号扩展 |
| IsLink | O | 控制信号，链接需要，控制是否需要将pc+4的值写入31号寄存器（即ra寄存器） |
| IsByteW | O | 控制信号，控制写寄存器时是写入字（4个字节）还是写入字节 |
| IsByteB | O | 控制信号，sb指令需要，控制读寄存器时是按字节读还是按字读从而写入寄存器 |
| ALUctr[3:0] | O | 控制信号，4位ALUctr，控制ALU单元执行何种运算 |
| Jumpctr[1:0] | O | 控制信号，控制需要跳转的方式，是直接跳转还是寄存器跳转 |
| Branchctr[2:0] | O | 控制信号，控制分支跳转的类型，根据不同类型判断相对于的分支跳转条件 |

### 1.2.14 mips模块

（1）基本描述

最后一个模块是设计的CPU的顶层mips模块，这个模块只有时钟信号和PC复位信号两个输入的接口，在这个模块中，就是将整个五级流水线CPU设计中的所有模块进行例化，统一组织并且将它们联系起来，CPU功能的测试就是经过这个模块进行的。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| reset | I | PC复位信号 |

## 1.3 单周期CPU各个指令控制信号取值表图



图1.3 单周期CPU各个指令控制信号取值表

图1.3展示了本次实验设计的单周期CPU各个指令的控制信号的取值，对于每条指令，在本单周期CPU的设计中所有的控制信号对应的取值在表中均非常清楚地罗列出来了，可以清楚看到指令对应的相关控制信号的取值。（具体浏览时可以放大更清楚地观察到具体的指令和相对应的控制取值）。

## 1.4 单周期CPU各个指令的设计逻辑

### 1.4.1 R型指令

（1）无符号加法：addu rd,rs,rt

将rs寄存器和rt寄存器的值相加并且写到rd寄存器中，无符号加法因此不作溢出判断，实现逻辑很简单，从寄存器堆读出相应的值送到ALU单元进行运算，并将相应的结果再写回寄存器堆即可。

（2）无符号减法：subu rd,rs,rt

将rs寄存器的值减去rt寄存器的值相减并写到rd寄存器中，同样不作溢出判断，实现逻辑同无符号加法，ALU单元中作无符号减法运算即可。

（3）有符号比较，小于置位：slt rd,rs,rt

比较rs寄存器和rt寄存器中的值，若小于则将rd寄存器置为1，否则置0，实现逻辑与前面一样。

（4）按位与：and rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位与运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位与运算即可。

（5）按位或非：nor rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位或非运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位或非运算即可。

（6）按位或：or rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位或非运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位或非运算即可。

（7）按位异或：xor rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位或非运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位异或运算即可。

（8）逻辑左移：sll rd,rt,shf

将rt寄存器中的值逻辑左移shf位并将结果写回rd寄存器，shf字段信息已经提前被解析出来并且送到ALU单元，根据控制信号进行相关运算即可。

（9）逻辑右移：srl rd,rt,shf

将rt寄存器中的值逻辑右移shf位并将结果写回rd寄存器，逻辑是平凡的，ALU单元中的运算采取逻辑右移即可。

（10）无符号小于置位：sltu rd,rs,rt

将rs寄存器和rt寄存器中的值进行比较，若小于则将1写回rd寄存器，这里的有符号和无符号比较在代码中不需要特意加以区分，可以共用一个ALU计算方式控制信号。

（11）跳转寄存器并链接：jalr rs

这一条指令特殊安排一个ALUctr的控制信号用来计算，当是jalr运算时，ALU单元将rs寄存器中的值写到RegtoJump端口中并送到npc模块单元，同时这条指令生成的控制信号会更新相关的跳转信号Jumpctr，同时本条指令解析格式中的rd寄存器为31号寄存器，所以将ALU运算的结果表示为当前的PC值+4之后的值，通路逻辑会将这个结果自动写回31号ra寄存器。

（12）跳转寄存器：jr rs

这一条指令和上一条指令少了链接的要求，因此实现更为简单，不需要考虑链接功能，将从rs读出来的值送到RegtoJump中去即可，npc单元中会正确计算跳转之后的下一条指令的PC。

（13）变量逻辑左移：sllv rd,rt,rs

和之前逻辑左移shf位指令的区别是，变量逻辑左移是将rt寄存器中的值逻辑左移rs寄存器中的值对应的位数，因此可以通过增加新的ALUctr计算控制信号控制运算方式，选择用rs寄存器的值进行移位运算，并将结果写回rd寄存器即可。

（14）算术右移：sra rd,rt,shf  
 算术右移和逻辑左移有一定的区别，在Verilog代码模拟中，可以通过系统函数$signed并将原来的逻辑右移符号>>改为>>>即可，其他的逻辑是一致的且平凡的。

（15）变量算术右移：srav rd,rt,rs

结合上述指令中对变量移位和逻辑移位的逻辑，这一条指令可以使用类似的逻辑进行设计，不再赘述。

（16）变量逻辑右移：srlv rd,rt,rs

逻辑基本和变量逻辑左移一致，ALUctr控制信号来控制相对于的运算方式即可，不再赘述。

### 1.4.2 I型指令

（17）立即数、无符号加法：addiu rt,rs,imm

将rs寄存器中的值加上符号扩展后的imm立即数并将结果写入rt寄存器，与前面介绍过的R型指令不一样的是目的寄存器不是rd寄存器而是rs寄存器，ALU另一个端口选择的操作数不是rt寄存器而是经扩展器扩展的立即数，这些控制单元生成的控制信号相应地都会去进行控制。ALU单元中的计算方式仍然是无符号加法。

（18）相等跳转：beq rs,rt,offset

分支跳转指令，控制单元会生成相应的对应类型的分支跳转的控制信号Branchctr，相等逻辑的判断在ALU单元中其实是执行的无符号减法的运算，根据结果是否为0生成对应的标志位Zero，npc模块会根据这个标志位和对应的分支跳转控制信号判断是否进行跳转，同时，经过扩展的offset立即数也会送到npc单元中进行地址的计算。当Branchctr控制信号取001并且Zero标志位为1时发生跳转。

（19）不等跳转：bne rs,rt,offset

与上一条相等跳转指令逻辑几乎一致，判断逻辑根据不等稍微改变一下即可，即相等判断Zero是否等于1，不等即判断Zero是否等于0，其余的执行逻辑如出一辙。当Branchctr控制信号取010并且Zero标志位为0时发送跳转。

（20）装载字：lw rt,offset(base)

这条指令base给出的是基址寄存器，加上符号扩展的立即数offset后得到在数据存储器中需要取数的地址，并经过从存储器取数后将取得的一个字的数据写回rt寄存器，计算地址在ALU中使用的无符号加法运算方式，从数据存储器取数的相关逻辑在前一部分模块定义中已经说明清楚，相关控制信号会控制正确取数并将写回寄存器的值改为从数据存储器中读出来的值。

（21）存储字：sw rt,offset(base)

与前面一条lw指令不同的是，这个是将从rt寄存器中读出来的值写到数据存储器对应的地址中去，更新相关需要的控制信号即可完成，计算目的地址在ALU单元中使用的是无符号加法的运算。

（22）立即数装载高位：lui rt,imm

这是将16位的立即数放到rt寄存器对应值的高16位，并在低16位添加0即可，这里需要注意的是控制信号中的Extop需要置为0，即不需要进行符号扩展，ALU运算单独设计一个lui的运算方式特殊处理。

（23）立即数有符号比较，小于置位：slti rt,rs,imm

将rs寄存器的值和经符号扩展的imm立即数进行比较，如果小于则将rt寄存器置1，否则置0，没有什么特殊的逻辑，控制信号保证可以选择正确的值进行比较，ALU单元中的运算方式采用小于比较置位。

（24）立即数无符号比较，小于置位：sltiu rt,rs,imm

与立即数有符号比较并没有区别，这里不在赘述。

（25）大于或等于零跳转：bgez rs,offset

分支跳转指令，比较rs寄存器中的值和0的大小关系，接下来的包括这条指令在内的和0比较的分支跳转指令，解析出来的rt寄存器的值都是0，因此在ALU运算单元中使用无符号减法，得到的结果转化为无符号数与0比较，如果小于0则说明rs的值小于0，Sign置1，否则Sign标志位置0。控制信号会生成对应的分支信号，offset经符号扩展后会送到npc单元，根据Zero（表示两个值是否相等）和Sign标志位的取值情况判断是否需要进行跳转并计算得到正确的下一条指令的PC具体值。当Branchctr控制信号取011并且Zero标志位取0或者Sign标志位取1时发生跳转。

（26）大于零跳转：bgtz rs,offset

逻辑与bgez指令一致，不再赘述。当Branchctr控制信号取101并且Zero标志位取0并且Sign标志位取1时发生跳转。

（27）小于或等于零跳转：blez rs,offset

逻辑与bgez指令一致，不再赘述。当Branchctr控制信号取110并且Zero标志位取1或者Sign标志位取0时发生跳转。

（28）小于零跳转：bltz rs,offset

逻辑与bgez指令一致，不再赘述。当Branchctr控制信号取100并且Zero标志位取0并且Sign标志位取0时发生跳转。

（29）装载字节并作符号扩展：lb rt,offset(base)

与前面分析的lw装载字指令的区别是本条指令写入寄存器的值是字节，这个字节就是取原来读出来的字的低8位，并且这个字节的高24位需要作符号扩展，这些是通过在原有lw指令执行的逻辑上新增了IsByteW和ExtopM两个控制信号来实现的，前者为真时表示写寄存器时写入字节，后者为真时表示写入的字节需要进行符号扩展，其余的逻辑与lw指令无异。

（30）装载字节并作无符号扩展：lbu rt,offset(base)  
 与lb指令不同的是，ExtopM控制信号控制为0对要写入的字节作零扩展即可。

（31）存储字节：sb rt,offset(base)

与sw指令不同的是，sb指令需要将字节写入具体地址的数据存储器中，但是数据存储器的接口要求是按字为单位进行存储的，因此对于计算得到的地址在数据存储器中读出来的那一个32位的字信息，一共4个字节，需要根据ALU计算得到的地址找到是哪一个字节需要被写入，并且保证不覆盖其他3个字节的信息，要注意的是这里的存储方式是按照小端进行的，因此需要注意找对应字节的顺序。在实现逻辑上这一条指令会先允许从数据存储器中读出相应地址的字，并且送到寄存器堆中，不过寄存器写使能控制信号是关闭的，然后生成IsByteB为真的控制信号，表示读寄存器的busB端口的值要从读数据存储器中的busW进行获取，同时根据传入的具体地址计算应该修改busW哪一个字节，将rt寄存器的低8位写入对应的那个字节中。最后得到正确修改完成的busB信息，接下来和sw指令一样，将busB写入数据存储器即可。

（32）立即数按位与：andi rt,rs,imm

ALU运算单元中采取按位与运算，且第二个源操作数选择经过零扩展的立即数进行运算即可，并将结果写回rt寄存器。

（33）立即数按位或：ori rt,rs,imm

ALU运算单元中采取按位或运算，且第二个源操作数选择经过零扩展的立即数进行运算即可，结果写回rt寄存器。

（34）立即数按位异或：xori rt,rs,imm

ALU运算单元中采取按位异或运算，且第二个源操作数选择经过零扩展的立即数，运算的结果写回rt寄存器即可。

### 1.4.3 J型指令

（35）直接跳转：j target

控制单元会生成直接跳转指令相关的控制信号，并将26位的target字段信息送到npc单元中，npc模块会根据对应的跳转信号和target来计算下一条PC的值，由于PC的起始地址为0x00003000，故在计算j指令的下一条指令的地址时偏移量需要从0x00003000进行计算。

（36）跳转和链接：jal target

与上一条直接跳转指令的区别是本条指令需要链接，这可以通过生成IsLink控制信号来让寄存器堆将当前PC+4的值写入31号ra寄存器中去。其余的跳转逻辑是一致的。

# 第二章 代码实现和运行结果验证

## 2.1 具体Verilog代码实现

### 2.1.1 PC模块

// PC模块

module pc (

in\_pc,clk,reset,out\_pc

);

input[31:0] in\_pc;

input clk,reset;

output reg[31:0] out\_pc;

initial begin

out\_pc=32'h00003000; // 0x00003000

end

always @(posedge clk) begin

out\_pc=in\_pc;

if(reset)

out\_pc=32'h00003000; // 0x00003000

end

endmodule

### 2.1.2 pc2addr模块

// 将PC地址转为访问指令存储器的地址

module pc2addr (

out\_pc,addr

);

input[31:0] out\_pc;

output[11:2] addr;

wire[31:0] temp\_pc;

assign temp\_pc=out\_pc-32'h00003000;

assign addr=temp\_pc[11:2];

endmodule

### 2.1.3 npc模块

// 计算下地址模块

module npc (

out\_pc,imm32,target,RegtoJump,Jumpctr,Branchctr,Zero,Sign,in\_pc

);

input[31:0] out\_pc;

input[31:0] imm32;

input[25:0] target;

input[31:0] RegtoJump;

input[1:0] Jumpctr;

input[2:0] Branchctr;

input Zero,Sign;

output reg[31:0] in\_pc;

always @(\*) begin

in\_pc<=out\_pc+4;

if(Jumpctr!=0)begin

// 是跳转

if(Jumpctr==2'b01)begin

// in\_pc<={out\_pc[31:28],target<<2};

in\_pc<=32'h00003000+(target<<2);

end

else if(Jumpctr==2'b10)begin

// in\_pc<=RegtoJump;

in\_pc<=32'h00003000+RegtoJump;

end

end

else if(Branchctr!=0)begin

if(Branchctr==3'b001)begin

// 相等跳转

if(Zero==1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b010)begin

// 不等跳转

if(Zero!=1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b011)begin

// 大于等于0跳转

if(Zero==1 || Sign==1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b100)begin

// 小于0跳转

if(Zero!=1 && Sign==0)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b101)begin

// 大于0跳转

if(Zero!=1 && Sign==1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b110)begin

// 小于等于0跳转

if(Zero==1 || Sign==0)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

end

end

endmodule

### 2.1.4 im\_4K模块

// 指令存储器模块

module im\_4k (

addr,dout

);

input[11:2] addr;

output[31:0] dout;

reg[31:0] im[0:1023];

initial begin

$readmemh("code.txt",im);

// $display("%h",im[0]);

end

assign dout=im[addr];

endmodule

### 2.1.5 decode模块

// 对一条指令译码

module decode (

ins,

op,rs,rt,rd,shamt,func,imm16,target

);

input[31:0] ins;

output[5:0] op;

output[4:0] rs,rt,rd;

output[4:0] shamt;

output[5:0] func;

output[15:0] imm16;

output[25:0] target;

assign op=ins[31:26];

assign rs=ins[25:21];

assign rt=ins[20:16];

assign rd=ins[15:11];

assign shamt=ins[10:6];

assign func=ins[5:0];

assign imm16=ins[15:0];

assign target=ins[25:0];

endmodule

### 2.1.6 extend模块

// 扩展器模块

module extend (

imm16,imm32,Extop

);

input[15:0] imm16;

input Extop;

output[31:0] imm32;

reg[31:0] temp;

always @(\*) begin

if(Extop==0)

temp={16'b0,imm16[15:0]}; // 零扩展

else

temp={{16{imm16[15]}},imm16[15:0]}; // 符号扩展

end

assign imm32=temp;

endmodule

### 2.1.7 registers模块

// 寄存器组模块

module registers (

clk,rs,rt,rd,

RegDst,RegWr,IsLink,out\_pc,IsByteW,IsByteB,ExtopM,

busA,busB,busW,dm\_addr

);

input clk,RegDst,RegWr,IsLink,IsByteW,IsByteB,ExtopM;

input[4:0] rs,rt,rd;

input[31:0] out\_pc;

input[31:0] busW;

input[31:0] dm\_addr;

output reg[31:0] busA,busB;

reg[31:0] regs[0:31];

integer i,j;

initial begin

for(i=0;i<32;i=i+1)begin

regs[i]<=0;

end

end

always @(posedge clk) begin

if(RegWr)begin

if(IsLink)begin

regs[31]=out\_pc+4;

end

if(RegDst)begin

if(IsByteW)begin

if(ExtopM)begin

regs[rd]={{24{busW[7]}},busW[7:0]};

end

else begin

regs[rd]={24'b0,busW[7:0]};

end

end

else begin

regs[rd]=busW;

end

end

else begin

if(IsByteW)begin

regs[rt]={24'b0,busW[7:0]};

end

else begin

regs[rt]=busW;

end

end

end

end

always @(\*) begin

busA<=regs[rs];

if(IsByteB)begin

// 注意是小端

j=dm\_addr%4;

if(j==0)begin

busB<={busW[31:8],regs[rt][7:0]};

end

else if(j==1)begin

busB<={busW[31:16],regs[rt][7:0],busW[7:0]};

end

else if(j==2)begin

busB<={busW[31:24],regs[rt][7:0],busW[15:0]};

end

else begin

busB<={regs[rt][7:0],busW[23:0]};

end

end

else begin

busB<=regs[rt];

end

end

endmodule

### 2.1.8 ALU模块

// 算术逻辑运算单元模块

module alu (

ALUctr,in1,in2,shf,out\_pc,Zero,Sign,Result,RegtoJump

);

input[3:0] ALUctr;

input[31:0] in1,in2;

input[4:0] shf;

input[31:0] out\_pc;

output reg Zero,Sign;

output reg[31:0] Result,RegtoJump;

always @(\*) begin

case(ALUctr)

4'b0000:begin

// 无符号加

Result=in1+in2;

Zero=0;

Sign=0;

end

4'b0001:begin

// 无符号减

Result=in1-in2;

if(Result==0)begin

Zero=1;

end

else begin

Zero=0;

if($signed(Result)<0)begin

Sign=0;

end

else begin

Sign=1;

end

end

end

4'b0010:begin

// 小于置位

if(in1<in2)begin

Result=1;

end

else begin

Result=0;

end

end

4'b0011:begin

// 按位与

Result=in1 & in2;

end

4'b0100:begin

// 按位或非

Result=~(in1 | in2);

end

4'b0101:begin

// 按位或

Result=in1 | in2;

end

4'b0110:begin

// 按位异或

Result=in1 ^ in2;

end

4'b0111:begin

// 逻辑左移

Result=in2 << shf;

end

4'b1000:begin

// 逻辑右移

Result=in2 >> shf;

end

4'b1001:begin

// jalr

RegtoJump=in1;

Result=out\_pc+4;

end

4'b1010:begin

// jr

RegtoJump=in1;

Result=0;

end

4'b1011:begin

// 变量逻辑左移

Result=in2 << in1;

end

4'b1100:begin

// 算术右移

Result=$signed(in2) >>> shf;

end

4'b1101:begin

// 变量算术右移

Result=$signed(in2) >>> in1;

end

4'b1110:begin

// 变量的逻辑右移

Result=in2 >> in1;

end

4'b1111:begin

// lui特殊运算

Result={in2,16'd0};

end

endcase

end

endmodule

### 2.1.9 mux\_alusrc模块

// 确定ALU源操作数二选一模块

module mux\_alusrc (

regs\_busB,ext\_imm32,ALUsrc,ALU\_Bin

);

input[31:0] regs\_busB;

input[31:0] ext\_imm32;

input ALUsrc;

output reg[31:0] ALU\_Bin;

always @(\*) begin

if(ALUsrc==1)

ALU\_Bin<=ext\_imm32;

else

ALU\_Bin<=regs\_busB;

end

endmodule

### 2.1.10 addr2dm模块

// 将计算得到的地址转为访问数据存储器的地址

module addr2dm (

in\_addr,addr

);

input[31:0] in\_addr;

output[11:2] addr;

assign addr=in\_addr[11:2];

endmodule

### 2.1.11 dm\_4k模块

// 数据存储器模块

module dm\_4k (

addr,din,we,clk,dout

);

input[11:2] addr;

input[31:0] din;

input we,clk;

output reg[31:0] dout;

reg[31:0] dm[0:1023];

integer i;

initial begin

for(i=0;i<1024;i=i+1)begin

dm[i]<=0;

end

end

always @(posedge clk) begin

if(we==1)begin

dm[addr]<=din;

end

end

always @(\*) begin

dout<=dm[addr];

end

endmodule

### 2.1.12 mux\_memtoreg模块

// 写寄存器二选一模块

module mux\_memtoreg (

ALU\_out,datamem\_out,MemtoReg,memtoreg\_out

);

input[31:0] ALU\_out;

input[31:0] datamem\_out;

input MemtoReg;

output reg[31:0] memtoreg\_out;

always @(\*) begin

if(MemtoReg==1)

memtoreg\_out<=datamem\_out;

else

memtoreg\_out<=ALU\_out;

end

endmodule

### 2.1.13 Control模块

// 单周期CPU控制单元模块

// 不进行统一设计管理 单独分析设置

module control (

op,func,rt,

RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB,

ALUctr,Jumpctr,Branchctr

);

input[5:0] op,func;

input[4:0] rt;

output reg RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB;

output reg[3:0] ALUctr;

output reg[1:0] Jumpctr;

output reg[2:0] Branchctr;

// 使用朴素的判别方式

always @(\*) begin

if(op==6'b000000)begin

// R型

// init

Branchctr=3'b000;

Jumpctr=2'b00;

ALUctr=4'b0000;

RegDst=1;

ALUsrc=0;

MemtoReg=0;

RegWr=1;

MemWr=0;

Extop=0;

ExtopM=0;

IsLink=0;

IsByteW=0;

IsByteB=0;

if(func==6'b100001)begin

// addu

ALUctr=4'b0000; // 无符号加

end

else if(func==6'b100011)begin

// subu

ALUctr=4'b0001; // 无符号减

end

else if(func==6'b101010)begin

// slt

ALUctr=4'b0010; // 小于置位

end

else if(func==6'b100100)begin

// and

ALUctr=4'b0011; // 按位与

end

else if(func==6'b100111)begin

// nor

ALUctr=4'b0100; // 按位或非

end

else if(func==6'b100101)begin

// or

ALUctr=4'b0101; // 按位或

end

else if(func==6'b100110)begin

// xor

ALUctr=4'b0110; // 按位异或

end

else if(func==6'b000000)begin

// sll

ALUctr=4'b0111; // 逻辑左移

end

else if(func==6'b000010)begin

// srl

ALUctr=4'b1000; // 逻辑右移

end

else if(func==6'b101011)begin

// sltu

ALUctr=4'b0010; // 小于置位

end

else if(func==6'b001001)begin

// jalr

ALUctr=4'b1001; // jalr特殊1

Jumpctr=2'b10;

end

else if(func==6'b001000)begin

// jr

ALUctr=4'b1010; // jr特殊2

Jumpctr=2'b10;

end

else if(func==6'b000100)begin

// sllv

ALUctr=4'b1011; // 变量的左移

end

else if(func==6'b000011)begin

// sra

ALUctr=4'b1100; // 算术右移

end

else if(func==6'b000111)begin

// srav

ALUctr=4'b1101; // 变量的算术右移

end

else if(func==6'b000110)begin

// srlv

ALUctr=4'b1110; // 变量的逻辑右移

end

end

else begin

// 非R型

Branchctr=3'b000;

Jumpctr=2'b00;

ALUctr=4'b0000;

RegDst=0;

ALUsrc=1;

MemtoReg=0;

RegWr=1;

MemWr=0;

Extop=0;

ExtopM=0;

IsLink=0;

IsByteW=0;

IsByteB=0;

if(op==6'b001001)begin

// addiu

ALUctr=4'b0000;

Extop=1;

end

else if(op==6'b000100)begin

// beq

ALUctr=4'b0001;

Branchctr=3'b001;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b000101)begin

// bne

ALUctr=4'b0001;

Branchctr=3'b010;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b100011)begin

// lw

ALUctr=4'b0000;

Extop=1;

MemtoReg=1;

end

else if(op==6'b101011)begin

// sw

ALUctr=4'b0000;

Extop=1;

RegWr=0;

MemWr=1;

end

else if(op==6'b001111)begin

// lui

ALUctr=4'b1111;

end

else if(op==6'b001010)begin

// slti

ALUctr=4'b0010;

Extop=1;

end

else if(op==6'b001011)begin

// sltiu

ALUctr=4'b0010;

Extop=1;

end

else if(op==6'b000001)begin

if(rt==5'b00001)begin

// bgez

ALUctr=4'b0001;

Branchctr=3'b011;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(rt==5'b00000)begin

// bltz

ALUctr=4'b0001;

Branchctr=3'b100;

Extop=1;

ALUsrc=0;

RegWr=0;

end

end

else if(op==6'b000111)begin

// bgtz

ALUctr=4'b0001;

Branchctr=3'b101;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b000110)begin

// blez

ALUctr=4'b0001;

Branchctr=3'b110;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b100000)begin

// lb

ALUctr=4'b0000;

Extop=1;

ExtopM=1;

MemtoReg=1;

IsByteW=1;

end

else if(op==6'b100100)begin

// lbu

ALUctr=4'b0000;

Extop=1;

MemtoReg=1;

IsByteW=1;

end

else if(op==6'b101000)begin

// sb

ALUctr=4'b0000;

Extop=1;

RegWr=0;

MemWr=1;

IsByteB=1;

MemtoReg=1;

end

else if(op==6'b001100)begin

// andi

ALUctr=4'b0011;

end

else if(op==6'b001101)begin

// ori

ALUctr=4'b0101;

end

else if(op==6'b001110)begin

// xori

ALUctr=4'b0110;

end

else if(op==6'b000010)begin

// j

Jumpctr=2'b01;

RegWr=0;

end

else if(op==6'b000011)begin

// jal

Jumpctr=2'b01;

RegWr=0;

IsLink=1;

end

end

end

endmodule

### 2.1.14 mips模块

`timescale 1ns/1ns

`include "datapath/pc.v"

`include "datapath/pc2addr.v"

`include "datapath/im.v"

`include "datapath/decode.v"

`include "datapath/addr2dm.v"

`include "datapath/alu.v"

`include "datapath/dm.v"

`include "datapath/extend.v"

`include "datapath/mux.v"

`include "datapath/npc.v"

`include "datapath/registers.v"

`include "control/ctrl.v"

module mips (

clk,reset

);

input clk,reset;

wire[31:0] in\_pc,out\_pc;

pc my\_pc(in\_pc,clk,reset,out\_pc);

wire[11:2] im\_addr;

pc2addr my\_pc2addr(out\_pc,im\_addr);

wire[31:0] ins;

im\_4k my\_im\_4k(im\_addr,ins);

wire[5:0] op;

wire[4:0] rs,rt,rd;

wire[4:0] shf;

wire[5:0] func;

wire[15:0] imm16;

wire[25:0] target;

wire[31:0] imm32;

decode my\_decode(ins,op,rs,rt,rd,shf,func,imm16,target);

wire RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB;

wire[3:0] ALUctr;

wire[1:0] Jumpctr;

wire[2:0] Branchctr;

extend my\_extend(imm16,imm32,Extop);

wire[31:0] busA,busB,busW;

wire[31:0] Result,RegtoJump;

registers my\_registers(clk,rs,rt,rd,RegDst,RegWr,IsLink,out\_pc,IsByteW,IsByteB,ExtopM,busA,busB,busW,Result);

wire[31:0] in2;

mux\_alusrc my\_mux\_alusrc(busB,imm32,ALUsrc,in2);

wire Zero,Sign;

alu my\_alu(ALUctr,busA,in2,shf,out\_pc,Zero,Sign,Result,RegtoJump);

wire[11:2] dm\_addr;

wire[31:0] dm\_dout;

addr2dm my\_addr2dm(Result,dm\_addr);

dm\_4k my\_dm\_4k(dm\_addr,busB,MemWr,clk,dm\_dout);

mux\_memtoreg my\_mux\_memtoreg(Result,dm\_dout,MemtoReg,busW);

npc my\_npc(out\_pc,imm32,target,RegtoJump,Jumpctr,Branchctr,Zero,Sign,in\_pc);

control my\_control(op,func,rt,RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB,ALUctr,Jumpctr,Branchctr);

endmodule

### 2.1.15 tmips模块（testbench）

`timescale 1ns/1ns

`include "mips.v"

module testmips;

reg clk,reset;

mips example(clk,reset);

initial begin

$dumpfile("wave.vcd");

$dumpvars;

reset=0;

end

always begin

clk=0;#5;

clk=1;#5;

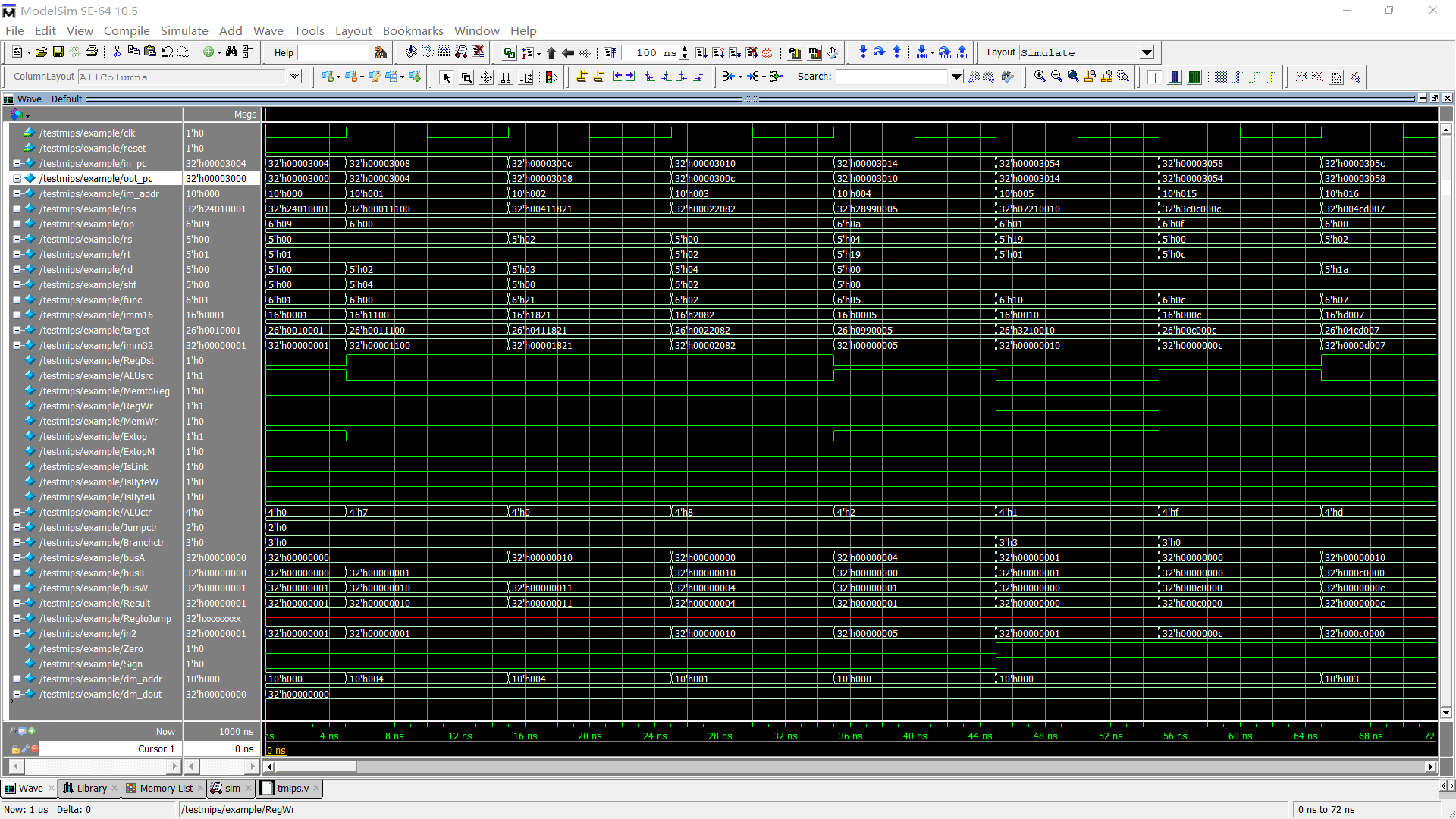
if($time>=1000)$finish;

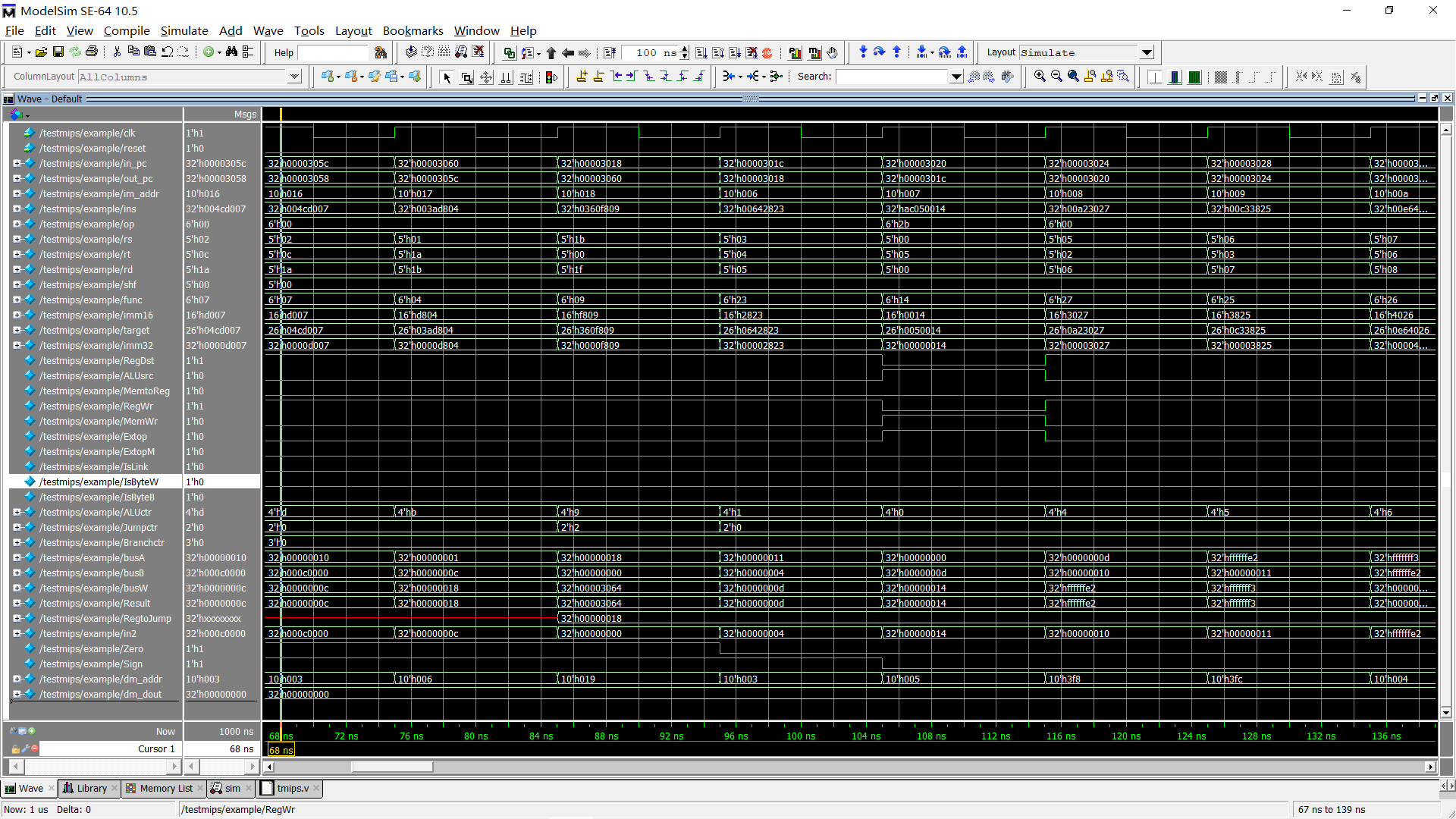
end

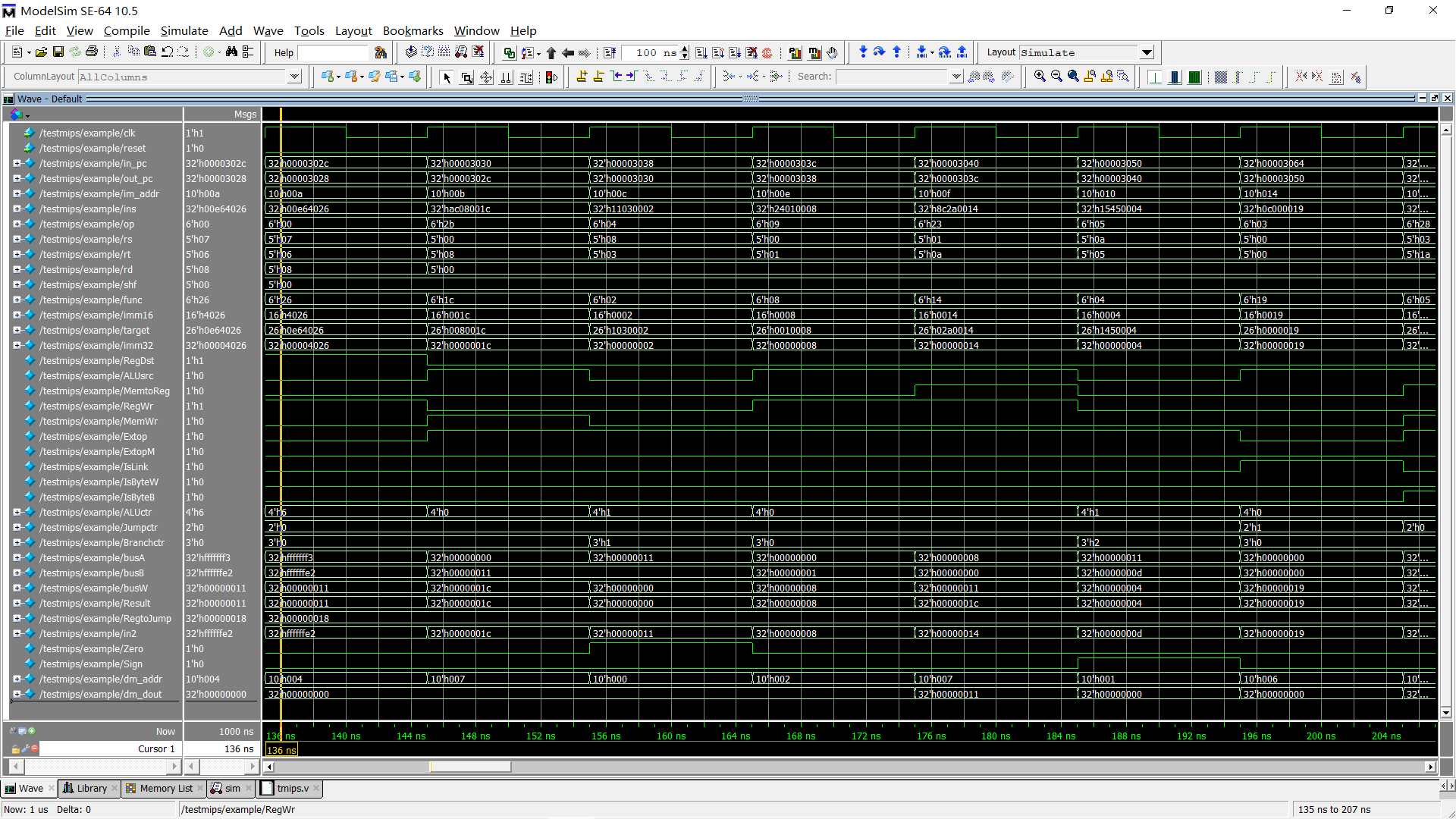
endmodule

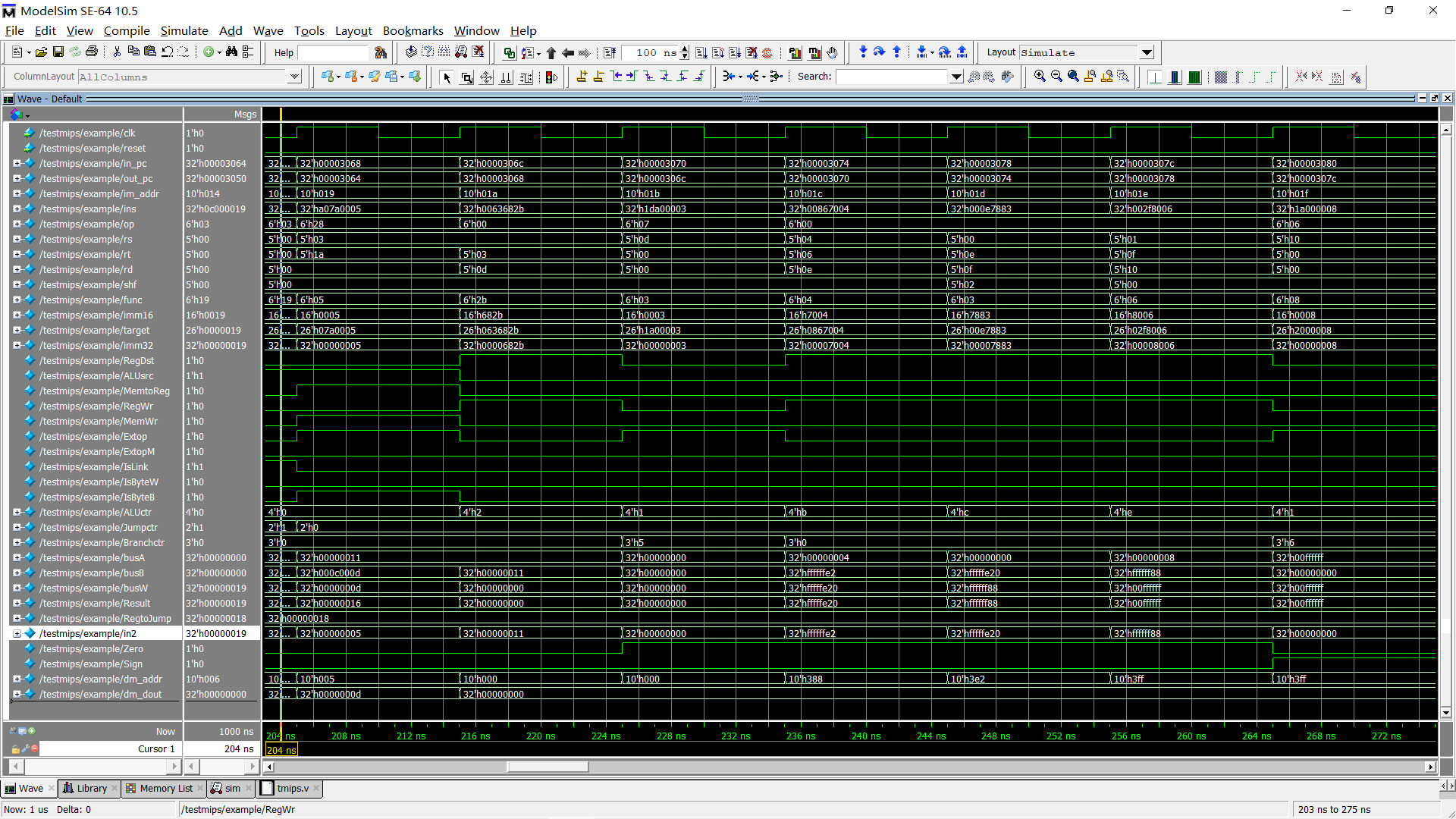
## 2.2 ModelSim上模拟仿真测试指令的信号结果

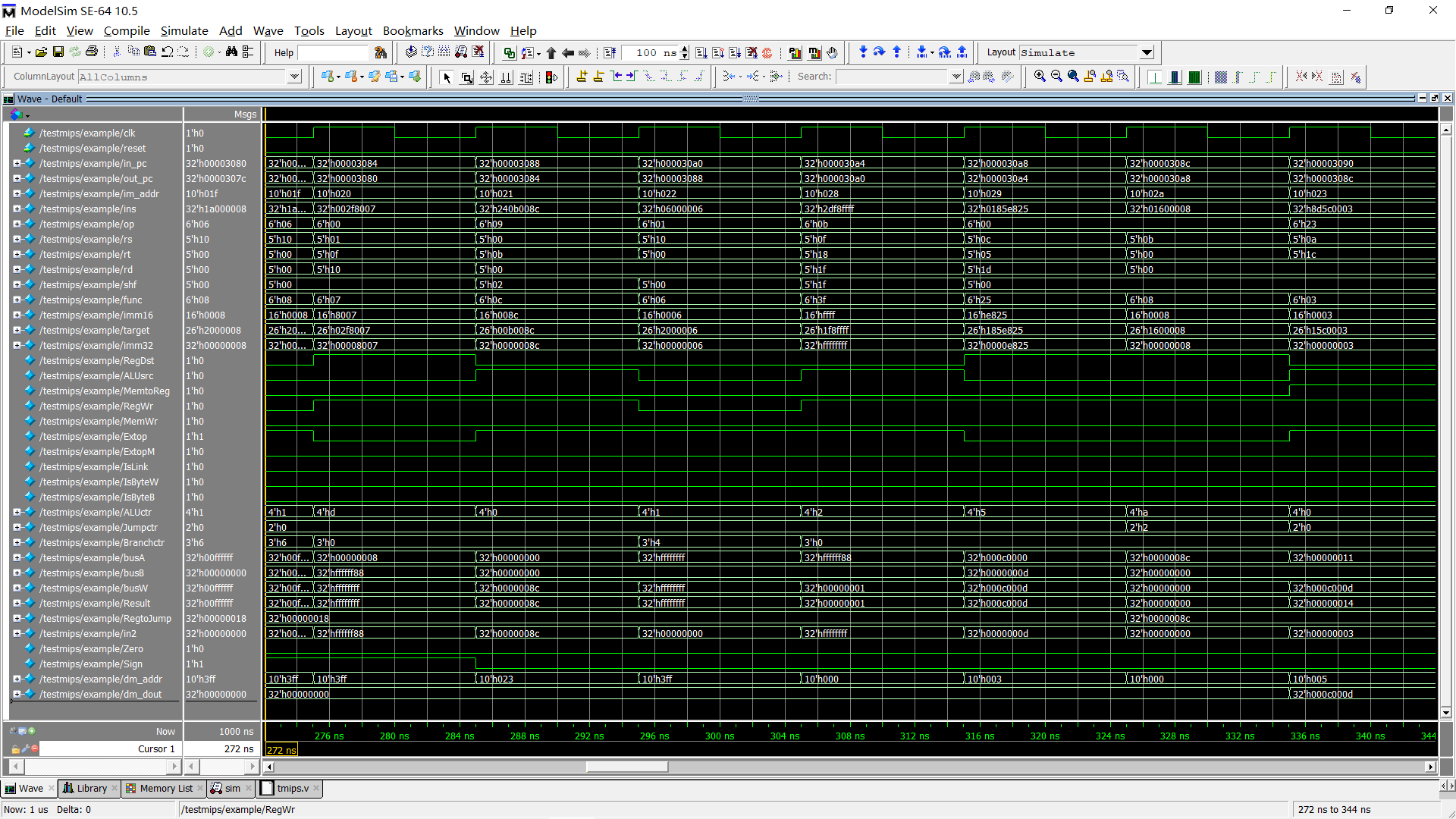
如下的截图是自己设计的单周期CPU在ModelSim仿真上所有信号的结果。（由于是在ModelSim上截取的所有信号的变化取值，故在图片上可能会表现地较为密集，在具体浏览时可以将所呈现的文档放大一定倍数即可清楚地浏览相关信号的取值变化）

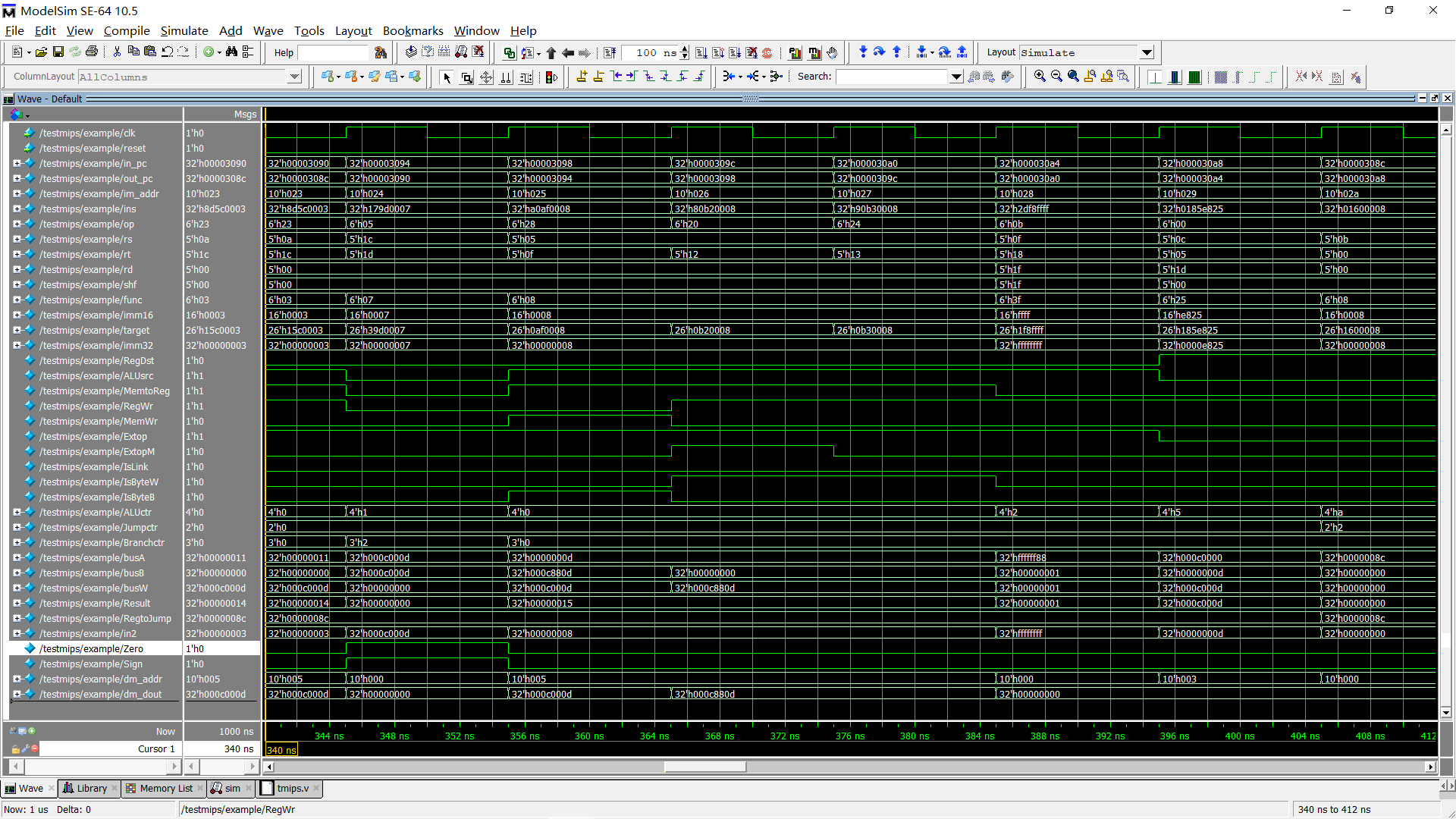


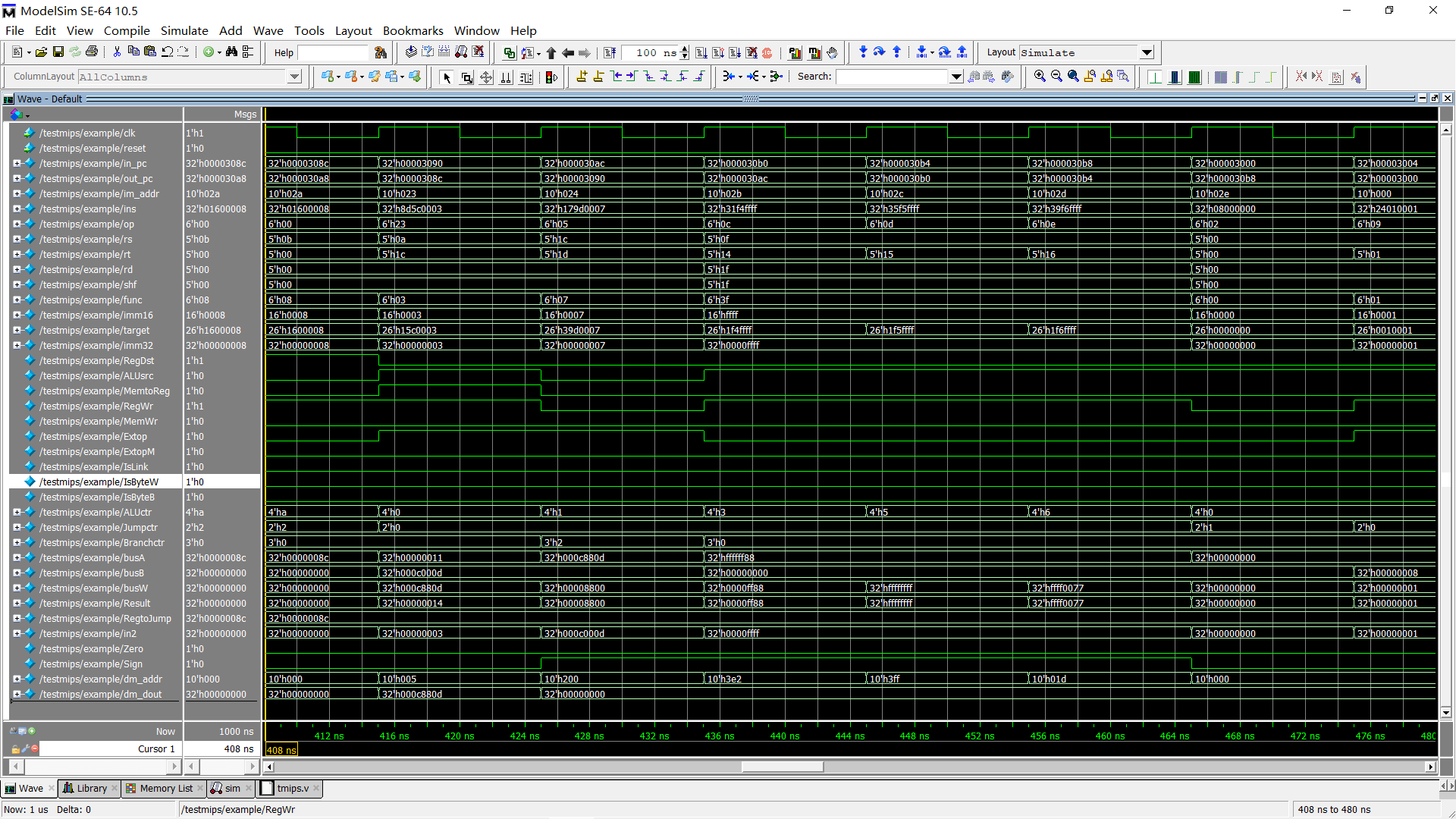












从直观地体现指令程序执行的正确性来看，可以重点观察这一段完整的运行过程中PC值信号（主要体现为上面所有信号中的sim:/testmips/example/out\_pc）的变化，以及寄存器堆写目的寄存器的寄存器编号（sim:/testmips/example/rt，sim:/testmips/example/rd）和写目的寄存器的具体值（sim:/testmips/example/busW），基本根据这些信息就可以充分验证自己设计的单周期CPU在验证指令测试集上的正确性。

可以看到，在时间单位轴上第480ns时，当前的PC值又变回了初始的PC值，即0x00003000，即顺利正确地执行了测试指令的一轮过程，并且相关的PC值均按照给出的测试指令的指令逻辑依次正确地进行了跳转，每条指令最终写入的目的寄存器和写入目的寄存器的值都和测试指令预计的一模一样，可以认为自己所编写的单周期CPU通过了36条基础指令的测试。

红线的说明：对于上面的信号图中，sim:/testmips/example/RegtoJump这一信号（这是在ALU运算单元输出的寄存器跳转的地址值）由于未进行初始化，在初始的指令中并没有相关的指令会去更新这个信号值，故呈现为未定态的红线特征。

# 第三章 总结与展望

提笔至此，也可以说是计算机组成原理这门课程的大实验也算是告一段落了，回顾起来，这次实验是让我们自己设计一个基于MIIPS36条基础指令集的单周期CPU，用Verilog代码进行代码的编写和仿真测试，在之前的实验基础上，我们在老师的带领和要求下，循序渐进地自己尝试使用Verilog代码编写了一些基础简单的模块，同时在进行最终的大实验之前也尝试对着教材完成了仅仅支持7-11条基础MIPS指令的单周期CPU的简单设计，所以在扩展成36条指令的时候这些之前打好的基础、做好的铺垫都让我开发的过程更加轻松。

这次单周期CPU的设计我在参考教材上对于11条指令的设计的基础上，通过对于新增加的指令进行扩展和设计，完善了教材上的数据通路和控制器部分，最终顺利实现了36条MIPS基础指令的单周期CPU设计，同时支持了实验要求的扩展内容，即让PC的起始地址位于0x00003000处。总的来讲，过程中进行得尚为顺利，可能主要原因是单周期CPU的逻辑结构还是非常清晰的，一个一个模块之间的关系以及每个模块承担的使命和责任都是非常分明的。印象中遇到的比较深刻的阻碍是对于sb这一条指令的设计，由于大实验要求对于存储器的接口是固定不变的，因此导致在实现sb指令时思考了一些手段和实现方式，最终采取了现在呈现出来的先将寄存器的值读取出来，并且在相关控制信号的控制下，正确修改相应的字节再将这个字写回数据存储器这样的方式，同时sb指令是按照小端的方式进行的，这个问题一开始最先由其他的小组成员发现的，当时没有考虑到这一点从而困扰了我们一段的时间。

通过这次单周期CPU的设计，虽然并没有实现很多扩展的功能，仅仅实现了在要求的36条MIPS基础指令上的设计和测试，但是从一开始模块的设计、数据通路的布局、控制器及控制信号的设计到最后的测试和调试，这一个过程中自己对CPU的工作方式有了非常深刻的理解，虽然我们的实验并没有用到开发板进行真正的上板烧写实验，而是仅仅通过Verilog代码testbench的方式进行模拟仿真，但是给我带来的收获是非常巨大的。在这个过程中，我对单周期CPU的工作方式和逻辑以及计算机系统结构的层次关系有了更加深刻和清楚的认知，也进一步加强了自己的代码编写能力和系统能力，一次一次的调试过程中也磨练了自己的意志，在成立的小组中与小组成员一起探讨问题、解决问题也增强了自己的团队协作能力，总之，受益匪浅。

当然，我想也是非常有必要，在这里感谢自己的计算机组成原理课程的任课老师施慧彬老师！施老师在整个教学和实验过程中，一丝不苟，兢兢业业，努力将自己的学识传授给我们，勤奋认真地批改我们的作业，积极关心并且能够及时解决学生的问题，施老师在整个过程的辛勤付出，是我这次实验可以顺利设计出来的关键因素！

也希望自己能力借助这次CPU设计的过程，将提升的能力内化为自己的过硬本领，在日后的学习和求索过程中，遇到困难能够一步一步努力克服，遇到问题能够一步一步尽力解决，同时，也将以更高的标准去要求自己，努力成为拥有过硬本领、能投身祖国建设的高水平人才！

# 参考文献

[1] 袁春风, 唐杰, 杨若瑜, 李俊等. 计算机组成与系统结构[M]. 第3版, 北京: 清华大学出版社, 2022.

# 附录

## 附录1 单周期CPU测试所用的指令程序

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令地址 | 汇编指令 | 结果描述 |  | 机器指令的机器码 |
| **16** 进制 | 二进制 |
| 00H | addiu $1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_0001 |
| 04H | sll $2, $1,#4 | [$2] = 0000\_0010H | 00011100 | 0000\_0000\_0000\_0001\_0001\_0001\_0000\_0000 |
| 08H | addu $3, $2,$1 | [$3] = 0000\_0011H | 00411821 | 0000\_0000\_0100\_0001\_0001\_1000\_0010\_0001 |
| 0CH | srl $4, $2,#2 | [$4] = 0000\_0004H | 00022082 | 0000\_0000\_0000\_0010\_0010\_0000\_1000\_0010 |
| 10H | slti $25,$4,#5 | [$25]=0000\_0001H | 28990005 | 0010\_1000\_1001\_1001\_0000\_0000\_0000\_0101 |
| 14H | bgez $25,#16 | 跳转到 54H | 07210010 | 0000\_0111\_0010\_0001\_0000\_0000\_0001\_0000 |
| 18H | subu $5, $3,$4 | [$5] = 0000\_000DH | 00642823 | 0000\_0000\_0110\_0100\_0010\_1000\_0010\_0011 |
| 1CH | sw $5, #20($0) | Mem[0000\_0014H] =  0000\_000DH | AC050014 | 1010\_1100\_0000\_0101\_0000\_0000\_0001\_0100 |
| 20H | nor $6, $5,$2 | [$6] = FFFF\_FFE2H | 00A23027 | 0000\_0000\_1010\_0010\_0011\_0000\_0010\_0111 |
| 24H | or $7, $6,$3 | [$7] = FFFF\_FFF3H | 00C33825 | 0000\_0000\_1100\_0011\_0011\_1000\_0010\_0101 |
| 28H | xor $8, $7,$6 | [$8] = 0000\_0011H | 00E64026 | 0000\_0000\_1110\_0110\_0100\_0000\_0010\_0110 |
| 2CH | sw $8, #28($0) | Mem[0000\_001CH] =  0000\_0011H | AC08001C | 1010\_1100\_0000\_1000\_0000\_0000\_0001\_1100 |
| 30H | beq $8, $3,#2 | 跳转到 38H | 11030002 | 0001\_0001\_0000\_0011\_0000\_0000\_0000\_0010 |
| 34H | slt $9, $6,$7 | 不执行 | 00C7482A | 0000\_0000\_1100\_0111\_0100\_1000\_0010\_1010 |
| 38H | addiu $1, $0,#8 | [$1] = 0000\_0008H | 24010008 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_1000 |
| 3CH | lw $10,#20($1) | [$10]=0000\_0011H | 8C2A0014 | 1000\_1100\_0010\_1010\_0000\_0000\_0001\_0100 |
| 40H | bne $10,$5,#4 | 跳转到 50H | 15450004 | 0001\_0101\_0100\_0101\_0000\_0000\_0000\_0100 |
| 44H | and $11,$2,$1 | 不执行 | 00415824 | 0000\_0000\_0100\_0001\_0101\_1000\_0010\_0100 |
| 48H | sw $11,#28($1) | 不执行 | AC2B001C | 1010\_1100\_0010\_1011\_0000\_0000\_0001\_1100 |
| 4CH | sw $4, #16($1) | 不执行 | AC240010 | 1010\_1100\_0010\_0100\_0000\_0000\_0001\_0000 |
| 50H | jal #25 | 跳转到 64H,  [$31] =0000\_0054H | 0C000019 | 0000\_1100\_0000\_0000\_0000\_0000\_0001\_1001 |
| 54H | lui $12,#12 | [$12] = 000C\_0000H | 3C0C000C | 0011\_1100\_0000\_1100\_0000\_0000\_0000\_1100 |
| 58H | srav $26,$12,$2 | [$26] = 0000\_000CH | 004CD007 | 0000\_0000\_0100\_1100\_1101\_0000\_0000\_0111 |
| 5CH | sllv $27,$26,$1 | [$27]=0000\_0018H | 003AD804 | 0000\_0000\_0011\_1010\_1101\_1000\_0000\_0100 |
| 60H | jalr $27 | 跳转到 18H , | 0360F809 | 0000\_0011\_0110\_0000\_1111\_1000\_0000\_1001 |
|  |  | [$31]=0000\_0064H |  |  |
| 64H | sb $26,#5($3) | 取出$26的最低字节处的0x0C, 将0x0C存入MEM[0000\_0016H] 字节单元处。表示一个字时，  Mem[0000\_0014H]  000C\_000DH | A07A0005 | 1010\_0000\_0111\_1010\_0000\_0000\_0000\_0101 |
| 68H | sltu $13,$3,$3 | [$13]= 0000\_0000H | 0063682B | 0000\_0000\_0110\_0011\_0110\_1000\_0010\_1011 |
| 6CH | bgtz $13,#3 | 不跳转 | 1DA00003 | 0001\_1101\_1010\_0000\_0000\_0000\_0000\_0011 |
| 70H | sllv $14,$6,$4 | [$14] =FFFF\_FE20H | 00867004 | 0000\_0000\_1000\_0110\_0111\_0000\_0000\_0100 |
| 74H | sra $15,$14,#2 | [$15] =FFFF\_FF88H | 000E7883 | 0000\_0000\_0000\_1110\_0111\_1000\_1000\_0011 |
| 78H | srlv $16,$15,$1 | [$16] =00FF\_FFFFH | 002F8006 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0110 |
| 7CH | blez $16,#8 | 不跳转 | 1A000008 | 0001\_1010\_0000\_0000\_0000\_0000\_0000\_1000 |
| 80H | srav $16,$15,$1 | [$16] =FFFF\_FFFFH | 002F8007 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0111 |
| 84H | addiu $11,$0,#140 | [$11] = 0000\_008CH | 240B008C | 0010\_0100\_0000\_1011\_0000\_0000\_1000\_1100 |
| 88H | bltz $16, #6 | 跳转到 A0H | 06000006 | 0000\_0110\_0000\_0000\_0000\_0000\_0000\_0110 |
| 8CH | lw $28,#3($10) | [$28] = 第一次执行时赋值000C\_000DH，  /第二次执行时赋值  000C\_880DH  测试45条指令的汇编代码执行情况说明中的‘/’类似地表示多次依次执行的结果。 | 8D5C0003 | 1000\_1101\_0101\_1100\_0000\_0000\_0000\_0011 |
| 90H | bne $28,$29,#7 | 第一次执行时,不跳转  /第二次执行时跳转到指令地址ACH处执行 | 179D0007 | 0001\_0111\_1001\_1101\_0000\_0000\_0000\_0111 |
| 94H | sb $15,#8($5) | Mem[0000\_0014H] =  000C\_880DH | A0AF0008 | 1010\_0000\_1010\_1111\_0000\_0000\_0000\_1000 |
| 98H | lb $18,#8($5) | [$18] =FFFF\_FF88H | 80B20008 | 1000\_0000\_1011\_0010\_0000\_0000\_0000\_1000 |
| 9CH | lbu $19,#8($5) | [$19] = 0000\_0088H | 90B30008 | 1001\_0000\_1011\_0011\_0000\_0000\_0000\_1000 |
| A0H | sltiu $24,$15,#0xFFFF | [$24] = 0000\_0001H | 2DF8FFFF | 0010\_1101\_1111\_1000\_1111\_1111\_1111\_1111 |
| A4H | or $29,$12,$5 | [$29] = 000C000DH | 0185E825 | 0000\_0001\_1000\_0101\_1110\_1000\_0010\_0101 |
| A8H | jr $11 | 跳转到8CH | 01600008 | 0000\_0001\_0110\_0000\_0000\_0000\_0000\_1000 |
| ACH | andi $20,$15,#0xFFFF | [$20] = 0000\_FF88H | 31F4FFFF | 0011\_0001\_1111\_0100\_1111\_1111\_1111\_1111 |
| B0H | ori  $21,$15,#0xFFFF | [$21] =FFFF\_FFFFH | 35F5FFFF | 0011\_0101\_1111\_0101\_1111\_1111\_1111\_1111 |
| B4H | xori $22,$15,#0xFFFF | [$22] = FFFF\_0077H | 39F6FFFF | 0011\_1001\_1111\_0110\_1111\_1111\_1111\_1111 |
| B8H | j #00H | 跳转指令，到 00H | 08000000 | 0000\_1000\_0000\_0000\_0000\_0000\_0000\_0000 |

## 附录2 实现的36条MIPS指令解析表图



图3 36条MIPS指令解析

图3是自己对照实验指导书在实验开始对36条MIPS指令实现的解析表图，其中解析了每条指令的类型和对应的字段信息，以及各自的功能和预期的计算方式，方便在代码设计中对于控制单元的编写和ALU控制信号的设计和生成。（具体浏览时可以将word文档放大更清楚查看具体内容和细节）