编号 051

南京航空航天大学

实验报告

题　目　　单周期处理器设计

学生姓名　　刘凌云

学　　号　　161710237

学　　院　　计算机科学与技术学院

专　　业　　计算机科学与技术

班　　级　　1617102

指导教师　　施慧彬　副教授

二〇一九年七月

南京航空航天大学
实验报告诚信承诺书

本人郑重声明：所呈交的毕业设计（论文）（题目：单周期处理器设计）是本人在导师的指导下独立进行研究所取得的成果。尽本人所知，除了毕业设计（论文）中特别加以标注引用的内容外，本毕业设计（论文）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：刘凌云　　2019 年 7 月 4 日

（学号）：161710237

南京航空航天大学

单周期处理器设计

摘要

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。 单周期 CPU 的功能：能够实现一些指令功能操作。单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。

批注 [F1]: 300 字左右，小四宋体，首行缩进 2 个字符，摘要正文 1.5 倍行距

关键词：五级流水线处理器 指令实现

Single-cycle processor design

Abstract

Single cycle refers to the execution of an instruction CPU completed in one clock cycle, and then start the next instruction execution, namely an instruction in a single cycle, the CPU clock cycles to complete function: to achieve some CPU instruction function single cycle refers to the execution of an instruction completed in one clock cycle, and then start the next instruction execution, namely an instruction with a clock cycles to complete level changes from low to high moment called clock rise along, the time interval between two adjacent clock up along the called one clock cycle

Key Words：**Single cycle processor**

批注 [x2]: 小三新罗马字体，居行首。

目录

1．课程设计说明

1.1 指令实现

11 条指令部分对于 11 条指令全部实现。为：add,sub,subu,slt,sltu,ori,addiu,lw,sw,beq,j.

36 条指令部分期盼实现指令如下：addu、subu、stl、and、nor、or、xor、addiu、lui、stlu、sllv、srav、srlv、slti、sltiu、andi、ori、xori、sll、srl、sra、beq、bne、begez、bgtz、blez、bltz、lw、sw、lb、lbu、sb、j、jr、jalr、jal.

1.2 完成情况

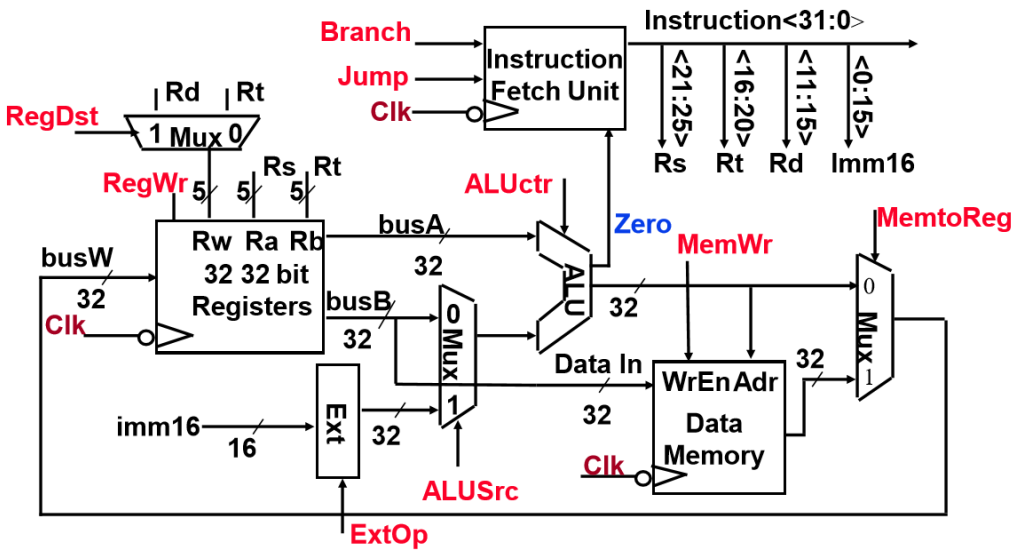
自主完成了 36 条指令并且通过了测试程序的测试。

2.模块化和层次化设计说明

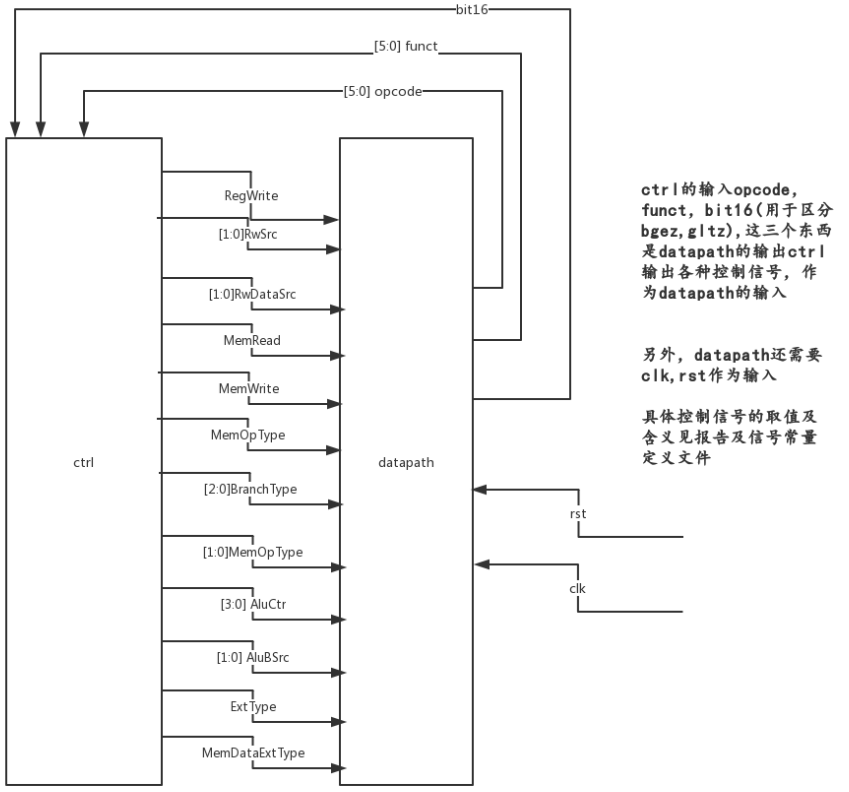
2.1 设计电路图

2.1.1 总体设计图

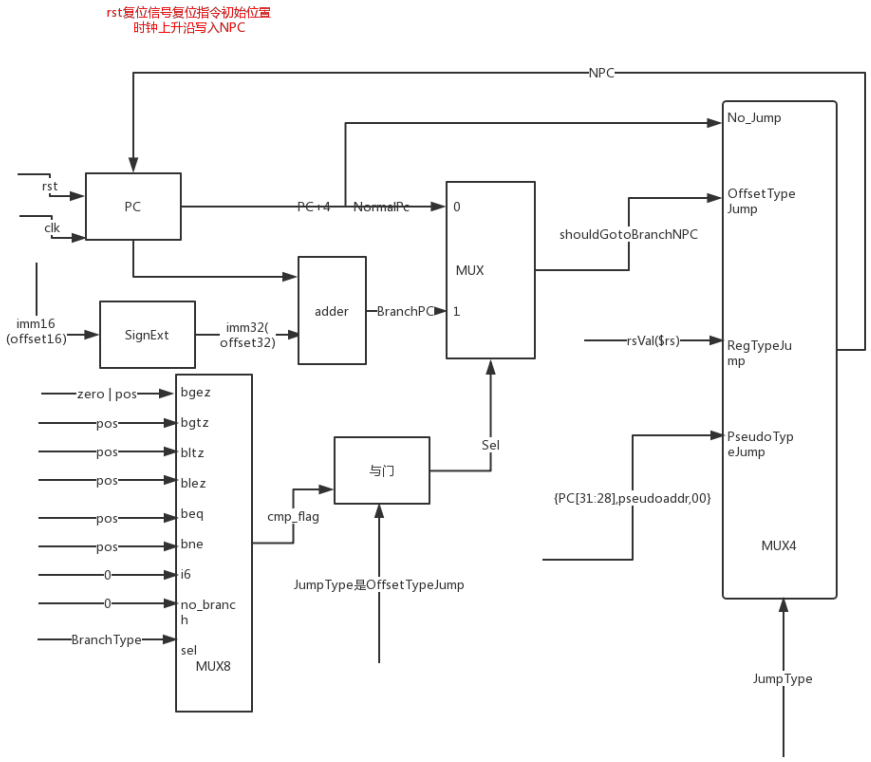
这张图时 ppt 的 11 条的。36 条的图有点大。总体上 36 条和 11 条的总体图差不多，只是控制信号线的位数多了一些。后面分开来画了 36 条的图。


Branch
Jump
Clk
Instruction Fetch Unit
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rs
Rt
Rd
Imm16
RegDst
Rd
Rt
1 Mux 0
RegWr
5
5
5
Rs
Rt
ALUctr
Zero
MemWr
MemtoReg
busW
32
Clk
Rw Ra Rb
32 32 bit
Registers
busA
32
busB
32
0 Mux 1
ALU
32
0 Mux 1
Data In
32
WrEn Adr
Data
Memory
32
Clk
imm16
16
Ext
32
ALUSrc
ExtOp

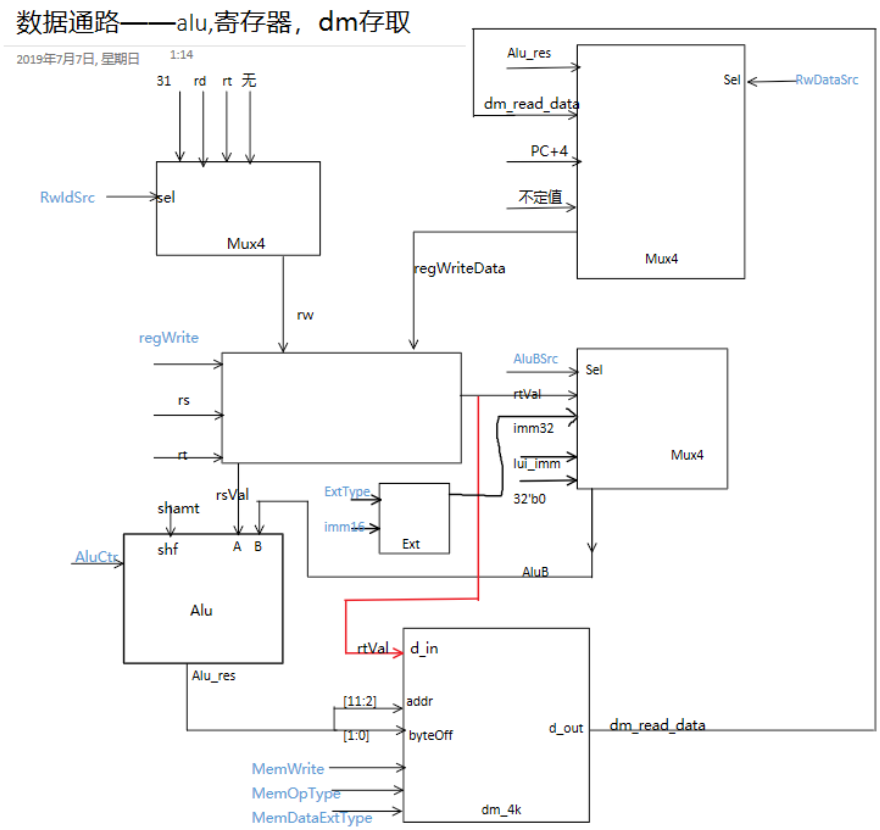
2.1.2 ctrl 与 datapath



2.1.3 指令顺序执行、各种分支与跳转的 NPC 与 PC 部分的数据通路



2.1.4 数据通路中的 alu，寄存器，dm 存取等



数据通路——alu、寄存器组、dm 存取

2.2 设计原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔

称为一个时钟周期。时钟周期一般也称振荡周期（如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍）

3.具体模块化定义

3.1.1 ctrl 模块化定义

(1) 基本描述

ctrl 模块以从 datapath 得到的 opcode,funct,bit16 为输入，产生各种控制信号，以让各部件协同工作。其输出是各类控制信号。

(2) 模块接口

信号名	方向	描述
op[5:0]	I	指令 opcode，用于区分指令
Funct[5:0]	I	用于区分 opcode=0x00 的 R 型指令
bit16	I	是指令的位序号 16 的数据，用于区分 bgez 和 bltz 指令
RegWrite	O	控制寄存器是否能够写入
RwIdSrc [1:0]	O	控制寄存器写入地址来源
RwDataSrc[1:0]	O	控制写入寄存器的数据的来源
BranchType[2:0]	O	控制分支类型
JumpType[1:0]	O	控制跳转类型
AluCtr[4:0]	O	控制 Alu 的具体行为
AluBSrc[1:0]	O	标记 Alu 的第二运算数 B 的来源
ExtType	O	标记是立即数的拓展是符号拓展还是零拓展

MemDataExtType	0	控制 Alu 的具体运算行为

表中涉及到的类型对应取值及其含义如下

(3) 输出的各种控制信号常量

```
// RwIdSrc 写入寄存器的编号是由什么字段决定的
`define RwIdSrc_rt 2'b00  // rt 字段决定
`define RwIdSrc_rd 2'b01  // rd 字段决定
`define RwIdSrc_ra 2'b10  // 固定的ra(31)号寄存器

// RwDataSrc
`define RwDataSrc_alu 2'b00
`define RwDataSrc_dm 2'b01
`define RwDataSrc_pc1 2'b10 // 计算出来的pc+4

// MemOpType
`define MemByWord 1'b1
`define MemByByte 1'b0

// Branch Types 分支类型
`define branch_beq 3'd4
`define branch_bne 3'd5
`define branch_bgez 3'd0
`define branch_bgtz 3'd1
`define branch_blez 3'd2
`define branch_bltz 3'd3
`define no_branch 3'd7
// emmm,this is wrong
//`define branch_jr 3'd6
```

```
//`define branch_jalr 3'd6
//`define branch_jal 3'd7


// Jump Types 跳转类型
`define No_Jump 2'b00  // 不跳转
`define OffsetTypeJump 2'b01 // 使用offset 计算的地址
`define RegTypeJump 2'b10 // 直接从寄存器读取的数据
`define PseudoTypeJump 2'b11 // 26 位直接地址

// ExtType 常量
`define SignExt 1'b1
`define ZeroExt 1'b0

// AluCtr 信号常量及意义
`define alu_addu 5'd0
`define alu_subu 5'd1
`define alu_slt 5'd2
`define alu_and 5'd3
`define alu_nor 5'd4
`define alu_or 5'd5
`define alu_xor 5'd6
`define alu_sltu 5'd7
`define alu_sllv 5'd8
`define alu_srav 5'd9
`define alu_srlv 5'd10
`define alu_sll 5'd11
`define alu_srl 5'd12
`define alu_sra 5'd13
```

```
`define alu_lui 5'd14
//`define alu_add 5'd15 // 虽然指令中没有直接的实现 add 指令，但是地址的计算需要有符号加
// 不需要，在不考虑溢出抛出异常的情况下，alu 的 add 和 addu 无差异
`define alu_nop 5'b11111

// AluBSrc Alu 运算器 B 的来源 常量表
`define aluBsrc_rt 2'd0  //  来源于$rt
`define aluBsrc_imm 2'd1  // 立即数经过拓展之后的 imm32，具体拓展受 ExtOp 控制
`define aluBsrc_zero 2'd2  // 指令固定的 0
`define aluBsrc_imm00 2'd3 // 立即数后面添 16 个 0 的立即数拓展,for lui

`define True 1'b1
`define False 1'b0
`define ra 5'b11111
```

（4）功能定义及实现

寄存器写入控制

信号名	取值	解释
RegWrite	1/0	控制是否需要写入寄存器；仅当 RegWrite=1 时下列寄存器写方面的信号有效
[1:0] RwIdSrc	RwIdSrc/ RwIdSrc_rd/ RwIdSrc_ra	需要写入的寄存器的情况下，R 型写入 rd，I 型写入 rt，jalr 与 jal 写入固定 ra(31)寄存器。 对于 R 型中的 jr 和 jalr 无需特别置。因为，对于 jalr 而言，rd 是 11111，恰好是 ra 寄存器，是 jalr

		需要写入的目标寄存器编号；对于 jr 的 rd 是 0 号寄存器，始终为 0，写入操作被忽略
[1:0] RwDataSrc	RwDataSrc_alu/RwDataSrc_dm/ RwDataSrc_pc1	需要写入时，R 型和大多 I 型写入寄存器的数据来自 alu 运算结果；oad 类型的来自于 dm 中读取的数据；jalr 和 jal 写入的寄存器来自 pc+4

访存控制

信号名	取值	解释
MemRead	1/0	控制是否读 dm，更准确地说，控制 dm 中流出的数据是否要被使用。当且仅当 lw/lb/lbu 指令取 1.
MemWrite	1/0	控制能否向 dm 写入数据。当且仅当 sw/sb 指令取 1.
MemOpType	MemByByte/MemByWord	只有 MemRead 或者 MemWrite 有一个有效时，MemOpType 有效。它控制访存指令是按照字节还是 Word 操作。lw/sw,MemOpType=` MemByWord;lb/lbu/sb 时,MemOpType= MemByByte
MemDataExtType	SignExt/ZeroExt	Lb 时，取 SignExt; lbu 时取 ZeroExt.

分支跳转控制

信号名	取值	解释

[2:0] BranchType	branch_beq/branch_bne/ branch_bgez/branch_bgtz/ branch_blez/branch_bltz/no_branch	6 条分支指令每条对应一个值，另一个额外的值表示 wu 分支顺序执行，非分支指令 no_branch.
[1:0] JumpType	No_Jump/OffsetTypeJump/ RegTypeJump/PseudoTypeJump	J 型指令即 jal 和 j 设 PseudoTypeJump; jr,jalr 设 RegTypeJump，从寄存器读取地址；分支指令全部需要通过 offset 计算分支地址，设 OffsetTypeJump；其余指令顺序执行不跳转，设 No_Jump。

3.1.2 datapath 模块化定义

(1) 基本描述

以 clk,rst 信号及控制器给出的各种输出的控制信号为输入，输出 opcode，funct，bit16 作为 ctrl 模块的输入。同时随着指令的执行，数据在数据通路内部流动。数据通路 datapath 模块内部包含 im,dm,pc,npc,reg_files,alu 等模块。

(2) 模块接口

信号名	方向	描述
op[5:0]	0	指令 opcode，用于区分指令
Funct[5:0]	0	用于区分 opcode=0x00 的 R 型指令
bit16	0	是指令的位序号 16 的数据，用于进一步区分 bgez 和 bltz 指令

RegWrite	I	控制寄存器是否能够写入
RwSrc[1:0]	I	控制寄存器写入地址来源
RwDataSrc[1:0]	I	控制写入寄存器的数据的来源
BranchType[2:0]	I	控制分支类型
JumpType[1:0]	I	控制跳转类型
AluCtr[4:0]	I	控制 Alu 的具体行为
AluBSrc[1:0]	I	标记 Alu 的第二运算数 B 的来源，具体来源标记已在控制器部分给出
ExtType	I	标记是立即数的拓展是符号拓展还是零拓展
MemDataExtType	I	控制对于 lb,lbu 的取出来的字节做不同的拓展
clk	I	输入信号
rst	I	复位信号

(3) 功能定义

序号	功能名称	功能描述
1	控制程序按照正确顺序执行	NPC 计算下一条要执行的指令地址，PC 每次时钟上升沿更新为 NPC.
2	取值译码	根据 PC 地址取指令并分离各字段译码。im_4k 和 Instruction 模块。
3	根据指令执行不同的运算	Alu 模块。
4	寄存器组正确读写	regfiles 模块
5	正确读写主存	dm_4k 模块
6	各模块配合正常	不应该出现错误的数据

		通路。

3.1.3 instruction 模块化定义

(1) 基本描述

Instruction 主要功能是根据当前的指令，进行各字段的分离，进行译码。

(2) 模块接口

信号名	方向	描述
[31:0] Instruction	I	输入的指令码
[5:0] op	0	指令 op 字段
[4:0] rs	0	rs 寄存器编号
[4:0] rt	0	rt 寄存器编号
[4:0] rd	0	rd 寄存器编号
[4:0] shamt	0	移位操作的位移数
[5:0] funct	0	指令的 funct 字段
[15:0] imm16_or_offset	0	指令的立即数字段
[25:0] PseudoAddr	0	指令的 PseudoAddr 字段，J 类型指令需要
bit16	0	指令的第 16 位

(3) 功能定义

各字段截取指令 Instuction 的位置如表所示。

序号	功能名称	功能描述
1	op	Instruction[31:26];

2	rs	Instruction[25:21];
3	rt	Instruction[20:16];
4	rd	Instruction[15:11];
5	shamt	Instruction[10:6];
6	funct	Instruction[5:0];
7	imm16_or_offset	Instruction[15:0];
8	PseudoAddr	Instruction[25:0];
9	bit16	Instruction[16];

3.1.4 PC 模块化定义

(1) 基本描述

pc 主要功能是完成输出当前指令地址。复位后，pc 指向 0x0000_3000，此处为第一条指令的地址。

(2) 模块接口

信号名	方向	描述
[31:0]npc	I	输入的指令地址
clk	I	时钟信号
rst	I	复位信号
[31:0]pc	0	输出的指令地址

(3) 功能定义

序号	功能名称	功能描述

1	复位	rst=1 时，将 pc 置为 0X0000_3000
2	输出指令地址	时钟信号到来时，将 npc 赋给 pc

3.1.5 NPC 模块化定义

(1) 基本描述

npc 主要功能是通过计算下一条要执行的指令的地址 NPC，并送入 pc，以实现指令执行顺序的控制。其输出是下一条指令。

其从四类大情况的“NPC”中用 mux4 模块选择 NPC。

四类情况为——正常执行的 PC+4、分支指令条件应当前往分支的地址、寄存器 rs 直接读出的地址、J 型指令 PseudoAddr 直接特殊拓展的地址。

选择由 JumpType 控制信号控制。分支指令条件应当前往分支的地址受 JumpType 和 BranchType 共同控制。

(2) 模块接口

信号名	方向	描述
[31:0] PC	I	当前指令地址
[25:0] PseudoAddr	I	J 型指令直接给出的截断地址
[15:0] offset	I	分支指令给出的地址偏移量
[31:0] RegTypeNPC	I	NPC 是寄存器 rs 存储的地址

[2:0] BranchType	I	分支指令类型
pc		
[1:0] JumpType	I	跳转类型
zero	I	标记 alu 运算结果（分支条件通过 alu 做减法）是否为 0，用于判断分支条件成立与否
pos	I	标记 alu 运算结果（分支条件通过 alu 做减法）是否为正，用于判断分支条件成立与否
neg	I	标记 alu 运算结果（分支条件通过 alu 做减法）是否为负，用于判断分支条件成立与否
[31:0] NPC	O	下条应该执行的指令
[31:0] NormalNPC	O	PC+4，为了跳转链接类型的指令准备写入寄存器的数据

（3）功能定义

序号	功能名称	功能描述
1	输出下一指令地址	根据 zero，pos,neg,branchType,JumpType,PC,offset,rsVal 的值输出下一条指令的地址。
2	输出跳转回调地址	计算 pc+4，方便$ra 保存回调地址

3.1.6 alu 模块化定义

(1) 基本描述

实现 addu, subu, slt, and, nor, or, xor, sltu, sllv, srav, srlv, sll, srl, sra, lui 十五种计算。

并且通过设置 AluCtr，AluBSrc 等控制信号的设置，使得 load/store 类型之类的内存地址被 alu 计算并通过 result 输出;而分支类型的指令的比较通过 subu 操作转换为判断 result 的正负零性，并输出了 pos,neg,zero 三个标志。

(2) 模块接口

信号名	方向	描述
[4:0] ALUctr	I	ALU 控制信号
[31:0]A	I	Alu 运算的操作数 A，固定来源 rs 寄存器
[31:0]B	I	Alu 运算的操作数 A，由 datapath 中控制信号 AluBSrc 决定来源
shamt	I	由 shamt 字段决定的位移数
zero	0	若 A!=B,zero=0,否则，zero=1
pos	0	为 A-B 数学上是否为正的标志
neg	0	为 A-B 数学上是否为负的标志
[31:0]result	0	Alu 计算输出的结果。

(3) 功能定义

序号	功能名称	功能描述
1	输出计算结果	根据 alu 控制信号，输出 A 与 B 的计算结果
2	输出 zero，neg，pos	分别标记 result 是否为 0,正数，负数。

3.1.7 mux,mux4,mux8 模块化定义

(1) 基本描述

实现 32 位二选一数据选择器

(2) 2 选 1mux 模块接口

信号名	方向	描述
Sel	I	mux 控制信号
[WIDTH-1:0]i0	I	mux 输入
[WIDTH-1:0]i1	I	mux 输入
[WID	O	mux 输出

TH - 1:0]out		

4 选 1 模块 mux4

信号名	方向	描述
[1:0]Sel	I	mux 控制信号
[WIDTH - 1:0]i00	I	mux 输入
[WIDTH - 1:0]i01	I	mux 输入
[WIDTH - 1:0]i10	I	mux 输入
[WIDTH - 1:0]i11	I	mux 输入
[31:0]out	O	mux 输出

8 选 1 模块 mux8

信号名	方向	描述
[2:0]Sel	I	mux 控制信号
[WIDTH -	I	mux 输入

1:0]i000		
[WIDTH-1:0]i001	I	mux 输入
[WIDTH-1:0]i010	I	mux 输入
[WIDTH-1:0]i011	I	mux 输入
[1:0]Sel	I	mux 控制信号
[WIDTH-1:0]i100	I	mux 输入
[WIDTH-1:0]i101	I	mux 输入
[WIDTH-1:0]i110	I	mux 输入
[WIDTH-1:0]i111	I	mux 输入
[31:0]out	O	mux 输出

(3) 功能定义

根据 sel 的值，选择相应的输入引脚的值作为。WIDTH 是 parameter 参数，默认为 32.由于功能比较简单不在制表。

3.1.8 ext 模块化定义：

(1) 基本描述

将输入的 WIDTH(默认 16)位地址按 ExtType 指定的拓展方式扩展为 BIGWIDTH(默认 32)位。

(2) 模块接口

信号名	方向	描述
ExtType	I	控制拓展类型,1 为符号拓展 0 为零拓展
[WIDTH -1:0]in	I	输入的 16 位地址
[BIGWIDTH -1:0]out	O	输出的默认 32 位的拓展数

(3) 功能定义

序号	功能名称	功能描述
1	输出扩展的数.	根据拓展类型将 a 扩展为默认 32 位的 out.

3.1.9 regfiles 模块化定义

(1) 基本描述

根据输入的两个寄存器地址，读取相应寄存器的值，根据寄存器写信号和寄存器地址，将输入的数据选择写入寄存器。

(2) 模块接口

信号名	方向	描述
clk	I	时钟信号
[4:0]ra	I	Rs 寄存器地址
[4:0]rb	I	Ra 寄存器编号
[31:0]busA	O	Rb 寄存器编号
[31:0]busB	O	Rt 寄存器值
writeEn	I	写寄存器信号
[4:0]rw	I	Rw 写寄存器地址
[31:0]busW	I	写入寄存器的数据流

(3) 功能定义

序号	功能名称	功能描述
1	读寄存器数据	读 ra、rb 寄存器的数据
2	向寄存器写入数据	根据写信号向 rw 寄存器选择写入数据

3.1.10 im_4k 模块化定义

(1) 基本描述

指令内存大小为 4K，初始化从 code.txt 载入指令。根据输入的指令地址，输出当前位置存储的指令。

(2) 模块接口

信号名	方向	描述
[11:2] addr	I	指令地址
[31:0] dout	O	指令

(3) 功能定义

序号	功能名称	功能描述
1	载入指令	初始化载入

		code.txt 中的指令
2	输出指令	根据输入的指令地址，输出当前指令

3.1.10 dm_4k 模块化定义

(1) 基本描述

“数据内存”大小为 4K，根据输入的地址读出“数据内存”中的数据，并根据数据写信号，将输入的数据选择写入“数据内存”中。

(2) 模块接口

信号名	方向	描述
clk	I	时钟信号
[11：2]addr	I	数据地址
[1:0]ByteOff	I	数据地址后两位，即字节偏移量
MemOpType	I	控制存/取操作是按字还是字节模式，有`MemByWord 和`MemByByte 两种模式
MemDataExtType	I	当为 lb/lbu 按字节

		模式读时，MemDataExtType 控制此字节是符号扩展还是零扩展以写入目标寄存器
[31:0]d_in	I	写入的数据
wr_en	I	数据内存写信号
[31:0]d_out	O	读出的数据

(3) 功能定义

序号	功能名称	功能描述
1	Lw 指令读整个 word 内存数据	根据输入的数据地址，读出数据内存的数据，读出的数据不一定被使用， lw 指令会使用。
2	Lb/lbu 指令读指定内存中的字节的数据并符号/零拓展为 32 位	根据输入的数据地址，读出数据内存的一个字节的数据并符号/零拓展为 32 位， lb/lbu 指令才会使用。
2	向数据内存写入数据	在时钟信号到来时，根据写数据信号，sw 指令将输入的数据选择写入数据内存中的一个 word，

		而 sb 写入指定的一个字节，写入的那个字节的值位$rt&0xff。

4 测试代码

代码使用 visual studio code 编写，文件编码为 gbk，编译运行使用 modelsim。

4.1Verilog 代码实现

adder.v

```
module adder(a, b, ci,
  sum, co, cf, zf, sf, of
);
  parameter WIDTH = 32;

  input [WIDTH-1:0] a,b;
  input ci;

  output [WIDTH-1:0] sum;
  output co;
  output cf,zf,sf,of;

  assign {co,sum}=a+b+ci;
  assign sf = sum[WIDTH-1];
  assign of = (sf^a[WIDTH-1])&(sf^b[WIDTH-1]);
  assign zf = sum==0 ? 1 : 0;
  assign cf = co^ci;
```

```
endmodule
```

alu.v

```
`include "..\\control\\ctrl_signals_def.v"
module alu (
  A, B, ALUctr, result,
  zero, pos, neg,
  shamt
);
  parameter WIDTH = 32;

  input [4:0] ALUctr;
  input [WIDTH-1:0] A, B;
  input [4:0] shamt;
  output reg [WIDTH-1:0] result;
  output zero, pos, neg;

  always@(A or B or ALUctr or shamt)
  begin
    case(ALUctr)
      `alu_addu: result <= A + B;
      `alu_subu: result <= (A - B);
      `alu_slt:  result <= ($signed(A) < $signed(B));
      `alu_and:  result <= (A & B);
      `alu_nor:  result <= (~(A | B));
      `alu_or:   result <= (A | B);
      `alu_xor:  result <= (A ^ B);
      `alu_sltu: result <=  (A < B);
```

```
    `alu_sllv: result <= (B << A);
    `alu_srav: result <= (($signed(B)) >>> A);
    `alu_srlv: result <= (B >> A);
    `alu_sll:  result <= (B << shamt);
    `alu_srl:  result <= (B >> shamt);
    `alu_sra:  result <= (($signed(B)) >>> shamt);
    `alu_lui:  result <= B;
    `alu_nop:  result <= 32'h0000_0000;
    default:   result <= 32'h0000_0000;
   endcase
  end
  assign zero = (result == 0);
  assign pos = ($signed(result) > 0);
  assign neg = ($signed(result) < 0);
endmodule
```

ctrl_signals_def.v

```
// opcode
`define Most_R_Type_op 6'b000000 // need differ  by funct
/*contains follow
sllv
srlv
srav
sllv
srlv
srav
jr
```

```
jalr
addu
subu
and
or
xor
nor
slt
sltu
*/
`define bgez_bltz_op 6'b000001 // need differ by bit16
// can differ by just opcode
`define opcode_addiu 6'b001001
`define opcode_lui 6'b001111
`define opcode_slti 6'b001010
`define opcode_sltiu 6'b001011
`define opcode_andi 6'b001100
`define opcode_ori 6'b001101
`define opcode_xori 6'b001110
`define opcode_beq 6'b000100
`define opcode_bne 6'b000101
`define opcode_bgtz 6'b000111
`define opcode_blez 6'b000110
`define opcode_lw 6'b100011
`define opcode_sw 6'b101011
`define opcode_lb 6'b100000
`define opcode_lbu 6'b100100
`define opcode_sb 6'b101000
`define opcode_j 6'b000010
```

```
`define opcode_jal 6'b000011


// bit16
`define bit16_bgez 1
`define bit16_bltz 0


// funct
`define funct_addu 6'b100001
`define funct_subu 6'b100011
`define funct_slt 6'b101010
`define funct_and 6'b100100
`define funct_nor 6'b100111
`define funct_or 6'b100101
`define funct_xor 6'b100110
`define funct_sltu 6'b101011
`define funct_sllv 6'b000100
`define funct_srav 6'b000111
`define funct_srlv 6'b000110
`define funct_sll 6'b000000
`define funct_srl 6'b000010
`define funct_sra 6'b000011
`define funct_jr 6'b001000
`define funct_jalr 6'b001001
// 16 条
// RwIdSrc 写入寄存器的编号是由什么字段决定的
`define RwIdSrc_rt 2'b00  // rt 字段决定
`define RwIdSrc_rd 2'b01  // rd 字段决定
`define RwIdSrc_ra 2'b10  // 固定的ra(31)号寄存器
```

```
// RwDataSrc
`define RwDataSrc_alu 2'b00
`define RwDataSrc_dm 2'b01
`define RwDataSrc_pc1 2'b10 // 计算出来的pc+4

// MemOpType
`define MemByWord 1'b1
`define MemByByte 1'b0

// Branch Types 分支类型
`define branch_beq 3'd4
`define branch_bne 3'd5
`define branch_bgez 3'd0
`define branch_bgtz 3'd1
`define branch_blez 3'd2
`define branch_bltz 3'd3
`define no_branch 3'd7
// emmm,this is wrong
//`define branch_jr 3'd6
//`define branch_jalr 3'd6
//`define branch_jal 3'd7


// Jump Types 跳转类型
`define No_Jump 2'b00  // 不跳转
`define OffsetTypeJump 2'b01 // 使用offset计算的地址
`define RegTypeJump 2'b10 // 直接从寄存器读取的数据
`define PseudoTypeJump 2'b11 // 26位直接地址
```

```
// ExtType 常量
`define SignExt 1'b1
`define ZeroExt 1'b0

// AluCtr 信号常量及意义
`define alu_addu 5'd0
`define alu_subu 5'd1
`define alu_slt 5'd2
`define alu_and 5'd3
`define alu_nor 5'd4
`define alu_or 5'd5
`define alu_xor 5'd6
`define alu_sltu 5'd7
`define alu_sllv 5'd8
`define alu_srav 5'd9
`define alu_srlv 5'd10
`define alu_sll 5'd11
`define alu_srl 5'd12
`define alu_sra 5'd13
`define alu_lui 5'd14
//`define alu_add 5'd15 // 虽然指令中没有直接的实现 add 指令，但是地址的计算需要有符号加
// 不需要，在不考虑溢出抛出异常的情况下，alu 的 add 和 addu 无差异
`define alu_nop 5'b11111

// AluBSrc ALu 运算器 B 的来源 常量表
`define aluBsrc_rt 2'd0  //  来源于$rt
`define aluBsrc_imm 2'd1  // 立即数经过拓展之后的 imm32，具体拓展受 ExtOp 控制
`define aluBsrc_zero 2'd2  // 指令固定的 0
`define aluBsrc_imm00 2'd3 // 立即数后面添 16 个 0 的立即数拓展,for lui
```

```
`define True 1'b1
`define False 1'b0
`define ra 5'b11111

`define clock_T 30
`define alu_temp_to_alu_res_T 1
```

ctrl.v

```
`include "ctrl_signals_def.v"
module ctrl(
  input [5:0] opcode, funct,
  input bit16,

  output reg RegWrite,
  output reg [1:0] RwIdSrc, RwDataSrc,

  output reg MemRead, MemWrite, MemOpType,

  output reg [2:0] BranchType,
  output reg [1:0] JumpType,

  output reg [4:0] AluCtr,
  output reg [1:0] AluBSrc,

  output reg ExtType, MemDataExtType
);
```

```
// MemDataExtType 仅仅当MemRead 或者MemWrite 有效时方有效

always@(opcode or funct or bit16)
begin
  if (opcode == `Most_R_Type_op) begin
    // differ by funct
    RegWrite = `True;
     // 对于jalr 和jr 而言，下面的设置也是正确的。
     // 对于jr 的rd 是0 号寄存器，始终为0，写入操作被忽略
     // 对于jalr 而言，rd 是11111，恰好是ra 寄存器，因此设为rd 是正确的。
    RwIdSrc = `RwIdSrc_rd;
    //RwDataSrc = (funct == `funct_jalr) ? `RwDataSrc_pc1 : `RwDataSrc_alu;
    MemRead = `False; MemWrite = `False; //MemOpType = `MemByWord; MemOpType no need to
set
    // branch and jump
    BranchType = `no_branch;
    // 因为只有jr，jalr JumpType 才有效
    if (funct == `funct_jr || funct == `funct_jalr) begin
      JumpType = `RegTypeJump; RwDataSrc = `RwDataSrc_pc1;
    end else begin
      JumpType = `No_Jump; RwDataSrc =`RwDataSrc_alu;
    end

    // for alu
    AluBSrc = `aluBsrc_rt;
    case (funct)
      `funct_addu : AluCtr = `alu_addu;
      `funct_subu : AluCtr = `alu_subu;
```

```
    `funct_slt : AluCtr = `alu_slt;
    `funct_and : AluCtr = `alu_and;
    `funct_nor : AluCtr = `alu_nor;
    `funct_or : AluCtr = `alu_or;
    `funct_xor : AluCtr = `alu_xor;
    `funct_sltu : AluCtr = `alu_sltu;
    `funct_sllv : AluCtr = `alu_sllv;
    `funct_srav : AluCtr = `alu_srav;
    `funct_srlv : AluCtr = `alu_srlv;
    `funct_sll : AluCtr = `alu_sll;
    `funct_srl : AluCtr = `alu_srl;
    `funct_sra : AluCtr = `alu_sra;
    `funct_jr : AluCtr = `alu_nop;
    `funct_jalr : AluCtr = `alu_nop;
    default: AluCtr = `alu_nop;
    // 16 条
  endcase

  ExtType = `ZeroExt; // not important,0 and 1 is both ok.

end else if (opcode == `bgez_bltz_op) begin
  // differ by bit16:
  // bgez,bltz
  RegWrite = `False;
  MemRead = `False; MemWrite = `False;
  BranchType = (bit16 == `bit16_bgez) ? `branch_bgez : `branch_bltz;
  JumpType = `OffsetTypeJump;
  AluBSrc = `aluBsrc_zero; AluCtr = `alu_subu; // 通过减法作比较
  ExtType = `SignExt;
```

```
end else begin
  // differ by opcodes
  case (opcode)
    `opcode_addiu:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_alu;
        MemWrite = `False; MemRead = `False;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_addu;
        AluBSrc = `aluBsrc_imm;
        ExtType = `SignExt;
      end
    `opcode_lui:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_alu;
        MemWrite = `False; MemRead = `False;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_lui;
        AluBSrc = `aluBsrc_imm00;
        ExtType = `ZeroExt;
      end
    `opcode_slti:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_rt;  RwDataSrc = `RwDataSrc_alu;
        MemWrite = `False; MemRead = `False;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_slt;
        AluBSrc = `aluBsrc_imm;
        ExtType = `SignExt;
```

```
    end
  `opcode_sltiu:
    begin
      RegWrite = `True; RwIdSrc = `RwIdSrc_rt;  RwDataSrc = `RwDataSrc_alu;
      MemWrite = `False; MemRead = `False;
      JumpType = `No_Jump; BranchType = `no_branch;
      AluCtr = `alu_sltu;
      AluBSrc = `aluBsrc_imm;
      ExtType = `SignExt;
    end
  `opcode_andi:
    begin
      RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_alu;
      MemWrite = `False; MemRead = `False;
      JumpType = `No_Jump; BranchType = `no_branch;
      AluCtr = `alu_and;
      AluBSrc = `aluBsrc_imm;
      ExtType = `ZeroExt;
    end
  `opcode_ori:
    begin
      RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_alu;
      MemWrite = `False; MemRead = `False;
      JumpType = `No_Jump; BranchType = `no_branch;
      AluCtr = `alu_or;
      AluBSrc = `aluBsrc_imm;
      ExtType = `ZeroExt;
    end
  `opcode_xori:
```

```
    begin
      RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_alu;
      MemWrite = `False; MemRead = `False;
      JumpType = `No_Jump; BranchType = `no_branch;
      AluCtr = `alu_xor;
      AluBSrc = `aluBsrc_imm;
      ExtType = `ZeroExt;
    end
  `opcode_beq:
    begin
      RegWrite = `False; // no need to set RwIdSrc,RwDataSrc
      MemWrite = `False; MemRead = `False;
      JumpType = `OffsetTypeJump;
      BranchType = `branch_beq;
      AluCtr = `alu_subu; AluBSrc = `aluBsrc_rt;
      ExtType = `SignExt;
    end
  `opcode_bne:
    begin
      RegWrite = `False; // no need to set RwIdSrc,RwDataSrc
      MemWrite = `False; MemRead = `False;
      JumpType = `OffsetTypeJump;
      BranchType = `branch_bne;
      AluCtr = `alu_subu; AluBSrc = `aluBsrc_rt;
      ExtType = `SignExt;
    end
  `opcode_bgtz:
    begin
      RegWrite = `False; // no need to set RwIdSrc,RwDataSrc
```

```
        MemWrite = `False; MemRead = `False;
        JumpType = `OffsetTypeJump;
        BranchType = `branch_bgtz;
        AluCtr = `alu_subu; AluBSrc = `aluBsrc_zero;
        ExtType = `SignExt;
      end
    `opcode_blez:
      begin
        RegWrite = `False; // no need to set RwIdSrc,RwDataSrc
        MemWrite = `False; MemRead = `False;
        JumpType = `OffsetTypeJump;
        BranchType = `branch_blez;
        AluCtr = `alu_subu; AluBSrc = `aluBsrc_zero;
        ExtType = `SignExt;
      end
    `opcode_lw:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_dm;
        MemRead = `True; MemWrite = `False; MemOpType = `MemByWord;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_addu; AluBSrc = `aluBsrc_imm;
        ExtType = `SignExt; // no need to set MemDataExtType
      end
    `opcode_lb:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_dm;
        MemRead = `True; MemWrite = `False; MemOpType = `MemByByte;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_addu; AluBSrc = `aluBsrc_imm;
```

```
        ExtType = `SignExt; MemDataExtType = `SignExt;
      end
    `opcode_lbu:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_rt; RwDataSrc = `RwDataSrc_dm;
        MemRead = `True; MemWrite = `False; MemOpType = `MemByByte;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_addu; AluBSrc = `aluBsrc_imm;
        ExtType = `SignExt; MemDataExtType = `ZeroExt;
      end
    `opcode_sw:
      begin
        RegWrite = `False; // no need to set RwIdSrc RwDataSrc
        MemRead = `False; MemWrite = `True; MemOpType = `MemByWord;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_addu; AluBSrc = `aluBsrc_imm;
        ExtType = `SignExt; // no need to set MemDataExtType
      end
    `opcode_sb:
      begin
        RegWrite = `False; // no need to set RwIdSrc RwDataSrc
        MemRead = `False; MemWrite = `True; MemOpType = `MemByByte;
        JumpType = `No_Jump; BranchType = `no_branch;
        AluCtr = `alu_addu; AluBSrc = `aluBsrc_imm;
        ExtType = `SignExt; // no need to set MemDataExtType
        // emmm, how to only write a byte in dm? maybe need change dm,
        // add a MemOpType signal to dm
        // yes, that's ok!
      end
```

```
    `opcode_j:
      begin
        RegWrite = `False; // no need to set RwIdSrc RwDataSrc
        MemRead = `False; MemWrite = `False; // no need to set MemOpType
        JumpType = `PseudoTypeJump; BranchType = `no_branch;
        AluCtr = `alu_nop; // no need to set AluBSrc
        //use another special ext-module, no need to set ExtType signal
      end
    `opcode_jal:
      begin
        RegWrite = `True; RwIdSrc = `RwIdSrc_ra; RwDataSrc = `RwDataSrc_pc1;
        MemRead = `False; MemWrite = `False; // no need to set MemOpType
        JumpType = `PseudoTypeJump; BranchType = `no_branch;
        AluCtr = `alu_nop; // no need to set AluBSrc
        //use another special ext-module, no need to set ExtType signal
      end
  endcase
 end
end
endmodule
```

datapath. v

```
`include "..\\control\\ctrl_signals_def.v"
module datapath(
 input clk, rst,

 input RegWrite,
```

```
  input [1:0] RwIdSrc, RwDataSrc,

  input MemRead, MemWrite, MemOpType,

  input [2:0] BranchType,
  input [1:0] JumpType,

  input [4:0] AluCtr,
  input [1:0] AluBSrc,

  input ExtType, MemDataExtType,

  output [5:0] opcode, funct,
  output bit16
);
  wire [31:0] PC, NPC, NormalNPC;
  // 根据npc 和clk,rst 设置pc
  // 无需变动，直接复用11 条的
  pc pc(.NPC(NPC), .PC(PC), .rst(rst), .clk(clk));

  wire [31:0] Instruction;
  // 根据pc 从im 中读取指令，取指令，直接复用
  im_4k im(.addr(PC[11:2]), .dout(Instruction));

  wire [4:0] rs, rt, rd;
  wire [4:0] shamt;
  wire [15:0] imm16_or_offset;
  wire [25:0] PseudoAddr;
  // 译码部分，分离指令各个字段
```

```
// 直接复用
instruction instruction(
  Instruction,
  opcode,
  rs, rt, rd,
  shamt,
  funct,
  imm16_or_offset,
  PseudoAddr,
  bit16
);

// 数据通路寄存器存取部分
wire [4:0]rw;
wire [31:0]rsVal, rtVal, regWriteData;
assign rw = (RwIdSrc == `RwIdSrc_ra) ? `ra :
           ((RwIdSrc == `RwIdSrc_rd) ? rd : rt);
regfiles regfiles(
  .writeEn(RegWrite), .rw(rw), .busW(regWriteData),
  .ra(rs), .rb(rt), .busA(rsVal), .busB(rtVal),
  .clk(clk)
);

wire [31:0] imm32_or_offset, lui_imm;
wire [31:0] AluB; // AluA is always rsVal

ext imm_ext(
  .in(imm16_or_offset),
  .out(imm32_or_offset),
```

```
  .ExtType(ExtType)
 ); // prepare extend im16_or_offset to imm32_or_offset_or_offset
assign lui_imm = { imm16_or_offset, 16'b0 }; // prepare lui_imm

// 与AluBSrc 的信号常量对应书写代码
mux4 alub_mux(
  .i00(rtVal),
  .i01(imm32_or_offset), // Done: to prepare imm32_or_offset by ext module
  .i10(32'd0),
  .i11(lui_imm), // Done: to prepare lui_imm by special extend
  .sel(AluBSrc),
  .out(AluB)
);

//wire co, cf, zf, sf, of;
wire zero, pos, neg;
wire [31:0] alu_res;
alu alu(
  .A(rsVal), .B(AluB), // DONE:  to prepare AluB
  .ALUctr(AluCtr),
  .result(alu_res),
  //.co(co), .cf(cf), .zf(zf), .sf(sf), .of(of),
  .zero(zero), .pos(pos), .neg(neg),
  .shamt(shamt)
);

wire [31:0] dm_read_data;
dm_4k dm(
  .addr(alu_res[11:2]), .ByteOff(alu_res[1:0]),
```

```
  .wr_en(MemWrite), .d_in(rtVal),
  .d_out(dm_read_data),
  .MemOpType(MemOpType), .MemDataExtType(MemDataExtType),
  .clk(clk)
);

// set regWriteData
mux4 sel_regWriteData(
  .i00(alu_res),
  .i01(dm_read_data), // dm
  .i10(NormalNPC), // calulate pc
  .i11(32'hxxxx_xxxx),
  .sel(RwDataSrc),
  .out(regWriteData)
);

// not imm32, imm32 is for load/store type instructions calculate memory addr
npc npc(
  .PC(PC),
  .PseudoAddr(PseudoAddr),
  .offset(imm16_or_offset),
  .RegTypeNPC(rsVal),
  .BranchType(BranchType),
  .JumpType(JumpType),
  //.zf(zf), .sf(sf), .of(of),
  .zero(zero), .pos(pos), .neg(neg),
  .NPC(NPC),
  .NormalNPC(NormalNPC)
);
```

```
endmodule
```

dm.v：

```
`include "..\\control\\ctrl_signals_def.v"
module dm_4k(
  addr, d_in,
  wr_en, d_out,
  MemOpType, MemDataExtType,
  ByteOff,
  clk
);
  input [11:2] addr;
  input MemOpType, MemDataExtType;
  input [1:0] ByteOff;
  input [31:0] d_in;
  input wr_en;
  input clk;
  output [31:0]d_out;
  reg [31:0] dm[1023:0];
  wire [7:0] byte, be_to_wr_byte;
  wire [31:0] word_buffer, byte_ext_word;
  integer i;
  initial
  begin
    for(i = 0;i < 1024;i = i + 1)
```

```
  dm[i] = 32'b0; // 初始清0
end
assign word_buffer = dm[addr[11:2]];
//先去取出word，根据byteOff 4选1 选byte
// byte 扩展(类型根据控制信号决定)成byte_ext_word
// 2选1 word or byte_ext_word 作为dm 的输出（即从dm 是按照字节还是字读）
mux4 #(.WIDTH(8)) byte_mux(
  .i00(word_buffer[7:0]),
  .i01(word_buffer[15:8]),
  .i10(word_buffer[23:16]),
  .i11(word_buffer[31:24]),
  .sel(ByteOff),
  .out(byte)
);

ext #(.WIDTH(8), .BIGWIDTH(32)) ext_byte_ext_word(
  .in(byte),
  .ExtType(MemDataExtType),
  .out(byte_ext_word)
);

mux sel_memOpType(
  .i0(byte_ext_word), // byte type
  .i1(word_buffer), // word type
  .sel(MemOpType),
  .out(d_out)
);

assign be_to_wr_byte = d_in[7:0];
```

```
  always@(posedge clk)
  begin
    if (wr_en)
      if (MemOpType == `MemByByte)
      begin
        // big endian
        case (ByteOff)
         2'b00: dm[addr[11:2]][7:0] <= be_to_wr_byte;
         2'b01: dm[addr[11:2]][15:8] <= be_to_wr_byte;
         2'b10: dm[addr[11:2]][23:16] <= be_to_wr_byte;
         2'b11: dm[addr[11:2]][31:24] <= be_to_wr_byte;
        endcase
      end else
      begin
        dm[addr[11:2]][31:0] <= d_in;
      end
  end
endmodule
```

ext. v

```
module ext #(parameter WIDTH = 16,BIGWIDTH = 32)(in, out, ExtType);
  input [WIDTH - 1:0]in;
  input ExtType;
  output [BIGWIDTH - 1:0]out;
  assign out = ExtType ?
```

```
                {{BIGWIDTH-WIDTH{in[WIDTH - 1]}},in} :
                  {{BIGWIDTH-WIDTH{1'b0}},in};
endmodule
```

im.v

```
module im_4k(
  input [11:2]  addr,
  output [31:0] dout
);
  reg [31:0] im[1023:0];
  initial
  begin
    $readmemh("code.txt",im);
  end
  assign dout = im[addr[11:2]];
endmodule
```

instructions.v

```
module instruction(
  input [31:0] Instruction,
  output [5:0]  op,
  output [4:0]  rs,rt,rd,
  output [4:0]  shamt,
  output [5:0]  funct,
  output [15:0] imm16_or_offset,
```

```
  output [25:0] PseudoAddr,
  output bit16
);

  assign op = Instruction[31:26];
  assign rs = Instruction[25:21];
  assign rt = Instruction[20:16];
  assign rd = Instruction[15:11];
  assign shamt = Instruction[10:6];
  assign funct = Instruction[5:0];
  assign imm16_or_offset = Instruction[15:0];
  assign PseudoAddr = Instruction[25:0];
  assign bit16 = Instruction[16];
endmodule
```

mux.v

```
module mux #(parameter WIDTH = 32) (i0, i1, sel,out);
  input [WIDTH - 1:0] i0,i1;
  input sel;
  output [WIDTH - 1:0] out;
  assign out = sel ? i1 : i0;
endmodule

// 本来是4选1复用mux的，然而发生了奇怪的错误

module mux4 #(parameter WIDTH = 32) (
  i00, i01,
  i10, i11,
```

```
  sel,
  out
);
  input [WIDTH - 1:0] i00, i01, i10, i11;
  output [WIDTH - 1:0] out;
  input [1:0] sel;
  assign out = (sel[1]) ?
         ( (sel[0]) ? i11 : i10 ) :
         ( ((sel[0]) ? i01 : i00 ) );
endmodule

module mux8 #(parameter WIDTH = 32) (
  i000, i001, i010, i011,
  i100, i101, i110, i111,
  sel,
  out
);
  input [WIDTH - 1:0] i000, i001, i010, i011, i100, i101, i110, i111;
  output [WIDTH - 1:0] out;
  input [2:0] sel;
  assign out = (sel[2]) ?
         ( (sel[1]) ?
         ( (sel[0]) ? i111 : i110 ) :
         ( ((sel[0]) ? i101 : i100 ) ) )
             :
           ( (sel[1]) ?
         ( (sel[0]) ? i011 : i010 ) :
         ( ((sel[0]) ? i001 : i000 ) ) );
endmodule
```

```
/*module mux4 #(parameter WIDTH = 32) (
  i00, i01,
  i10, i11,
  sel,
  out
);
  input [WIDTH - 1:0] i00, i01, i10, i11;
  output [WIDTH - 1:0] out;
  input [1:0] sel;
  wire [WIDTH - 1:0] out0, out1;
  mux #(.WIDTH(WIDTH)) mux_lowpart0(
    .i0(i00), .i1(i01), .sel(sel[0]), .out(out0)
  );
  mux #(.WIDTH(WIDTH)) mux_lowpart1(
    .i0(i10), .i1(i11), .sel(sel[0]), .out(out1)
  );
  mux #(.WIDTH(WIDTH)) mux(
    .i0(out0), .i1(out1), .sel(sel[1]), .out(out)
  );
endmodule

module mux8 #(parameter WIDTH = 32) (
  i000, i001, i010, i011,
  i100, i101, i110, i111,
  sel, out
);
  input [WIDTH - 1:0] i000, i001, i010, i011, i100, i101, i110, i111;
```

```
  output [WIDTH - 1:0] out;
  input [2:0] sel;
  wire [WIDTH - 1:0] out0, out1;
  mux4 #(.WIDTH(WIDTH)) mux_lowpart0(
    .i00(i000), .i01(i001),
    .i10(i010), .i11(i011),
    .sel(sel[1:0]), .out(out0)
  );
  mux4 #(.WIDTH(WIDTH)) mux_lowpart1(
    .i00(i100), .i01(i101),
    .i10(i110), .i11(i111),
    .sel(sel[1:0]), .out(out0)
  );
  mux #(.WIDTH(WIDTH)) mux(
    .i0(out0), .i1(out1), .sel(sel[2]), .out(out)
  );
endmodule*/
```

npc. v

```
`include "..\\control\\ctrl_signals_def.v"
module npc(
  PC,
  PseudoAddr,
  offset,
  RegTypeNPC,
  BranchType,
  JumpType,
```

```
  //zf, sf, of,
  zero, pos, neg,
  NPC, NormalNPC
);

  input [31:0] PC;
  input [25:0] PseudoAddr;
  input [15:0] offset;
  input [31:0] RegTypeNPC;
  input [2:0] BranchType;
  input [1:0] JumpType;
  //input zf, sf, of;
  input zero, pos, neg;
  output [31:0] NPC;
  output [31:0] NormalNPC;

  wire cmp_flag;
  wire should_go_to_branch;

  mux8 #(.WIDTH(1)) mux_jump(
    .i000(zero | pos), // bgez
    .i001(pos), // bgtz
    .i010(neg | zero), // blez
    .i011(neg), // bltz

    .i100(zero), // beq
    .i101(zero^1'b1), // bne
    .i110(`False), //
    .i111(`False), // no_branch
```

```
    .sel(BranchType),
    .out(cmp_flag)
  );
  assign should_go_to_branch = (JumpType == `OffsetTypeJump) ? cmp_flag : `False;
  wire [29:0]offset_30;
  wire [31:0]offset_32;
  ext #(.WIDTH(16),.BIGWIDTH(30)) pc_offset_ext(
    .in(offset),
    .ExtType(`SignExt),
    .out(offset_30)
  );
  assign offset_32 = {offset_30, 2'b00};

  wire [31:0] PseudoTypeNPC = {PC[31:28], PseudoAddr, 2'b00};
  assign NormalNPC = PC + 4;

  wire [31:0] BranchOffsetNPC = PC + offset_32;
  wire [31:0] shouldGotoBranchNPC = (should_go_to_branch) ? BranchOffsetNPC :
NormalNPC;
  mux4 mux_jump_npc(
    .i00(NormalNPC), // No_Jump
    .i01(shouldGotoBranchNPC), // OffsetTypeJum  for branch

    .i10(RegTypeNPC), // RegTypeJump
    .i11(PseudoTypeNPC), // PseudoTypeJump

    .sel(JumpType),
```

```
    .out(NPC)
  );
endmodule
```

pc.v

```
module pc(
  input rst,clk,
  input [31:0] NPC,
  output reg [31:0] PC
);
  initial
  begin
     PC = 32'h0000_3000;
  end
  always@(posedge clk)
  begin
    PC <= rst ? 32'h0000_3000 : NPC;
  end
endmodule
// 无需更改，36 条直接复用11 条的
```

register.v

```
module regfiles(writeEn, rw, busW, ra, rb, busA, busB, clk);
  input clk, writeEn;
  input [4:0] rw, ra, rb;
  input [31:0] busW;
  output [31:0] busA, busB;
```

```
  reg [31:0] reg_file[31:0];
  initial
  begin
     reg_file[0] = 0; // set $zero to 0. keep $zero=0
  end
  always@(posedge clk)
  begin
     if(writeEn && (rw != 0)) // keep $zero=0 确保零寄存器始终为0
      reg_file[rw] = busW;
  end
  assign busA = (ra != 0) ? reg_file[ra] : 0;
  assign busB = (rb != 0) ? reg_file[rb] : 0;
endmodule
```

测试指令

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 , [$31] = 0000_0064H	0360F809	0000_0011_0110_0000_1111_1000_0000_1001
64H	sb $26,#5($3)	MEM[0000_0016H] = 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] = 000C_000DH /000C_880DH	8D5C0003	1000_1101_0101_1100_0000_0000_0000_0011
90H	bne $28,$29,#7	不跳转/跳转 ACH	179D0007	0001_0111_1001_1101_0000_0000_0000_0111
94H	sb $15,#8($5)	Mem[0000_0015H] = 0000_0088H	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	跳转指令 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 00H	08000000	0000_1000_0000_0000_0000_0000_0000_0000

表 1　CPU 测试所用 36 条汇编指令程序(续)

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
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] = 000C_000DH /000C_880DH	8D5C0003	1000_1101_0101_1100_0000_0000_0000_0011
90H	bne $28,$29,#7	不跳转/跳转 ACH	179D0007	0001_0111_1001_1101_0000_0000_0000_0111

94H	sb $15,#8($5)	Mem[0000_0015H] = 0000_0088H	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	跳转指令 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 00H	08000000	0000_1000_0000_0000_0000_0000_0000_0000

code.txt

24010001

00011100

00411821

00022082

28990005

07210010

00642823

AC050014

00A23027

00C33825

00E64026

AC08001C

11030002

00C7482A

24010008

8C2A0014

15450004

00415824

AC2B001C

AC240010

0C000019

3C0C000C

004CD007

003AD804

0360F809

A07A0005

0063682B

1DA00003

00867004

000E7883

002F8006

1A000008

002F8007

240B008C

06000006

8D5C0003

179D0007

A0AF0008

80B20008

90B30008

2DF8FFFF

0185E825

01600008

31F4FFFF

35F5FFFF

39F6FFFF

08000000

240B008C

06000006

8D5C0003

179D0007

A0AF0008

80B20008

90B30008

2DF8FFFF

0185E825

01600008

31F4FFFF

35F5FFFF

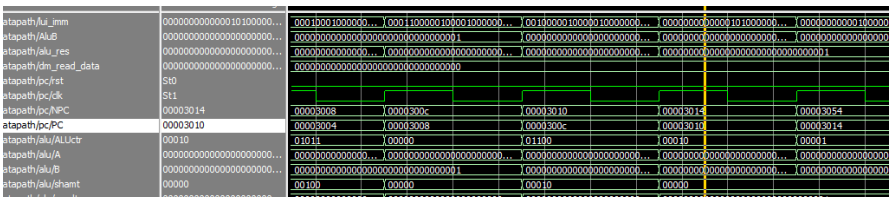
39F6FFFF

08000000

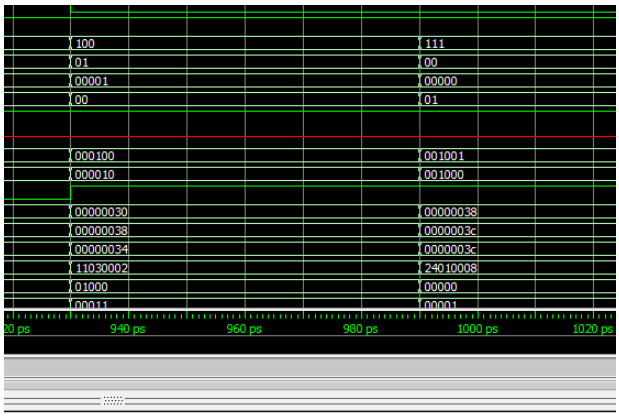
5 测试截图

应跳转与应不执行

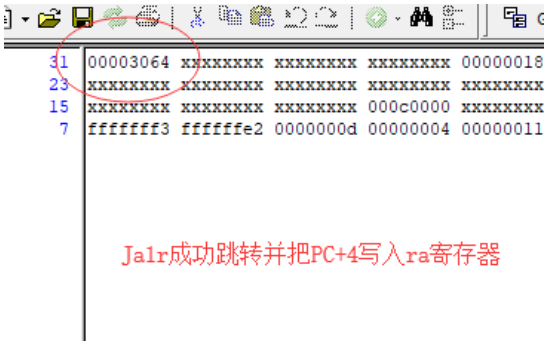
14H 跳转到 54H，bgez 测试成功

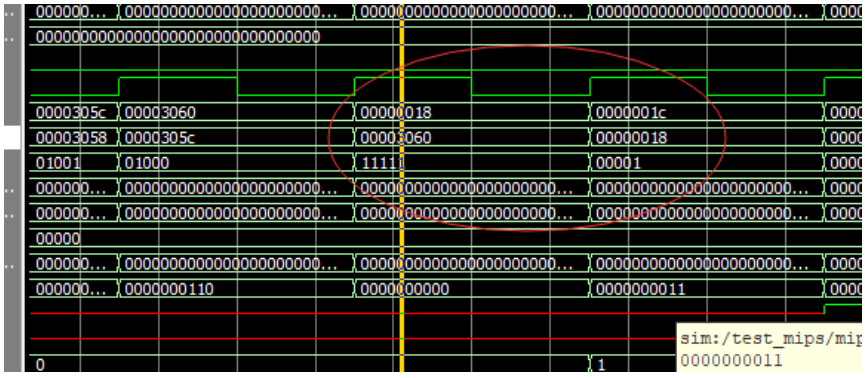


beq 跳转到 38H，见下页图。

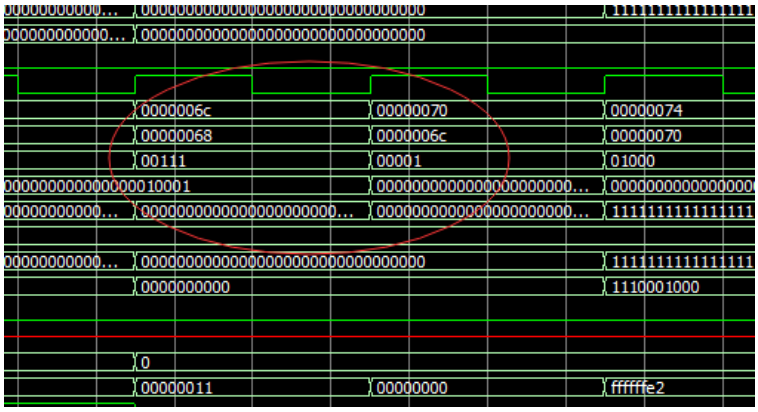


jalr60 跳转到 18H，并且检验寄存器值

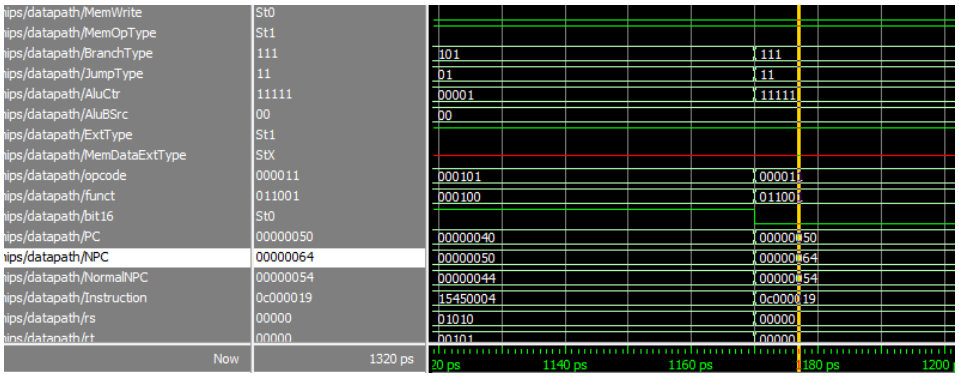




bgtz 6ch 不跳转通过测试

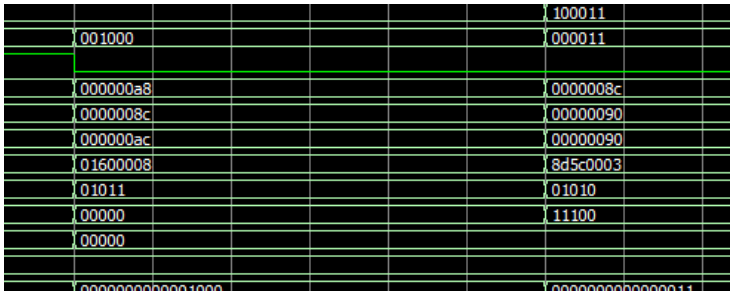


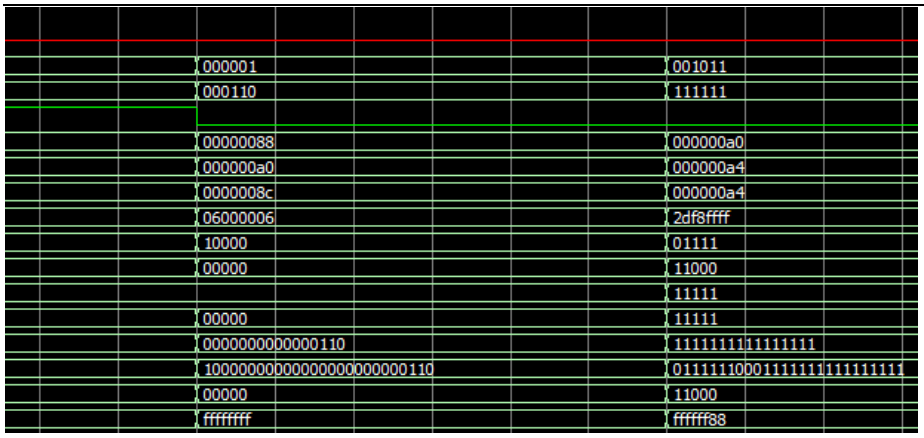
其余分支指令，跳转指令也测试检验过，通过测试。

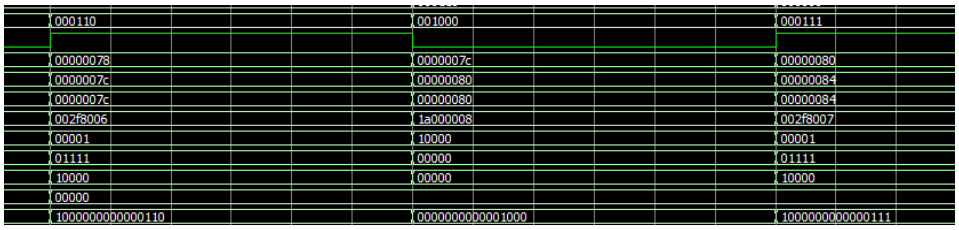


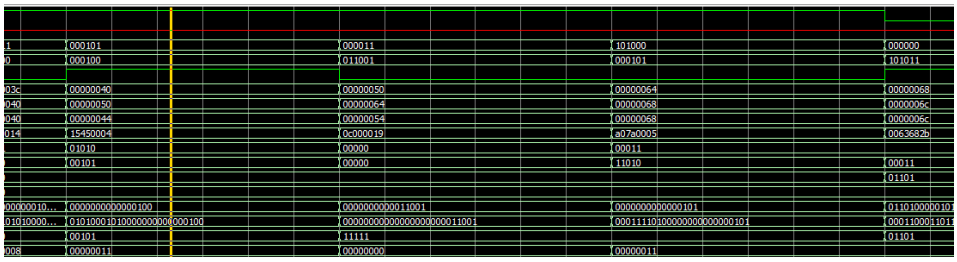

000010
000000
000000b8
00000000
000000bc
08000000
00000
00000
00000
00000
0000000000000000
00000000000000000000000000000000
00000
00000000
00000000
00000004
00000004
24010001
00001
0000000000000001
0000000010000000000000001
00001

000101
000111
00000090
000000ac
00000094
179d0007
11100
11101
0000000000000111
11100111010000000000000111
11101
0000000c
000c000d
00110
11111
00000
00000
11111
11111
11111
11111
01111
10100
11111
11111
11111111111111111
10100
ffffff8

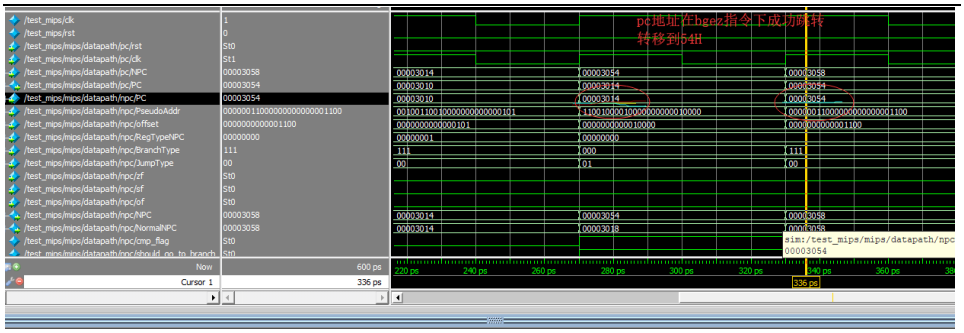

001000
000000a8
0000008c
000000ac
01600008
01011
00000
00000
100011
000011
0000008c
00000090
00000090
8d5c0003
01010
11100


000001
000110
00000088
000000a0
0000008c
06000006
10000
00000
00000
0000000000000110
00000
ffffffff
001011
111111
000000a0
000000a4
000000a4
2df8ffff
01111
11000
11111
11111
1111111111111111
11000
ffffff88


000110
00000078
0000007c
0000007c
002f8006
00001
01111
10000
00000
1000000000000110
001000
0000007c
00000080
00000080
1a000008
10000
00000
00000
0000000000001000
000111
00000080
00000084
00000084
002f8007
00001
01111
10000
1000000000000111

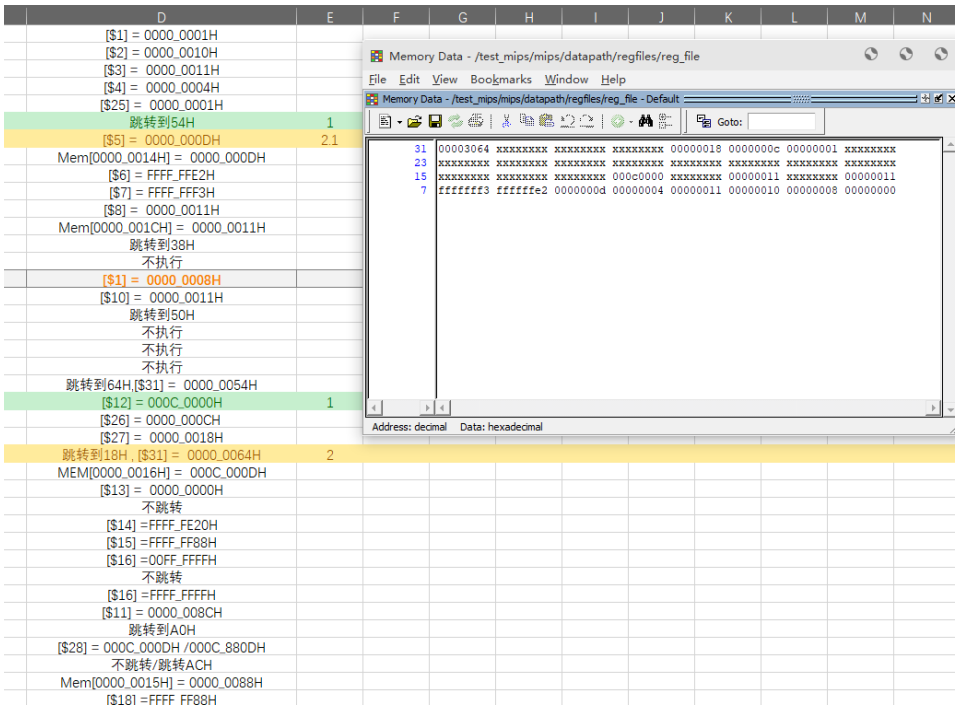

000101
000100
00000040
00000050
00000044
15450004
01010
00101
000011
011001
00000050
00000064
00000054
0c000019
00000
00000
101000
000101
00000064
00000068
00000068
a07a0005
00011
11010
000000
101011
00000068
0000006c
0000006c
0063682b
00011
01101
00101
00000011
11111
00000000
00000011
01101

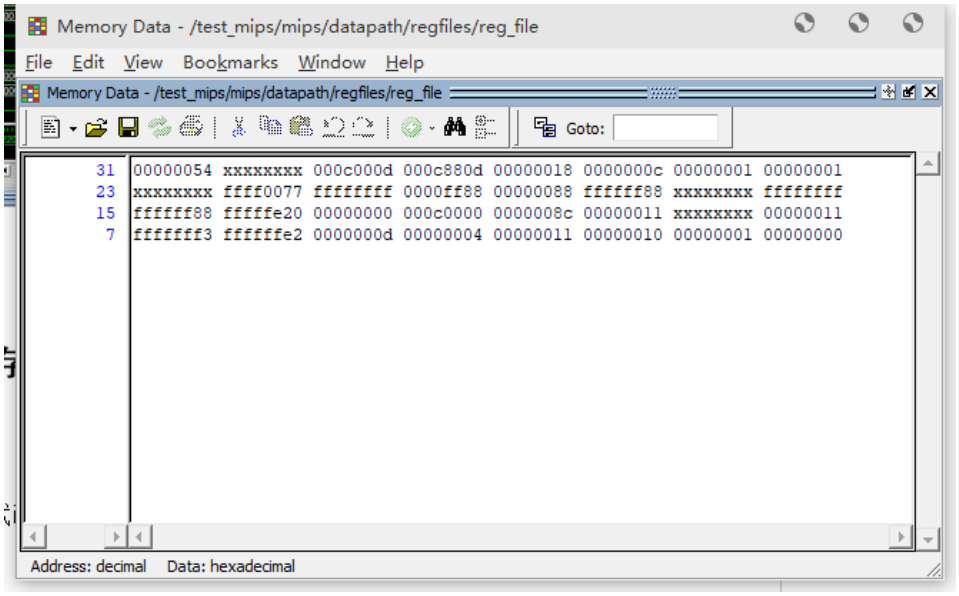
主　　存　　读　　取



寄存器、alu 运算结果、主存存取测试

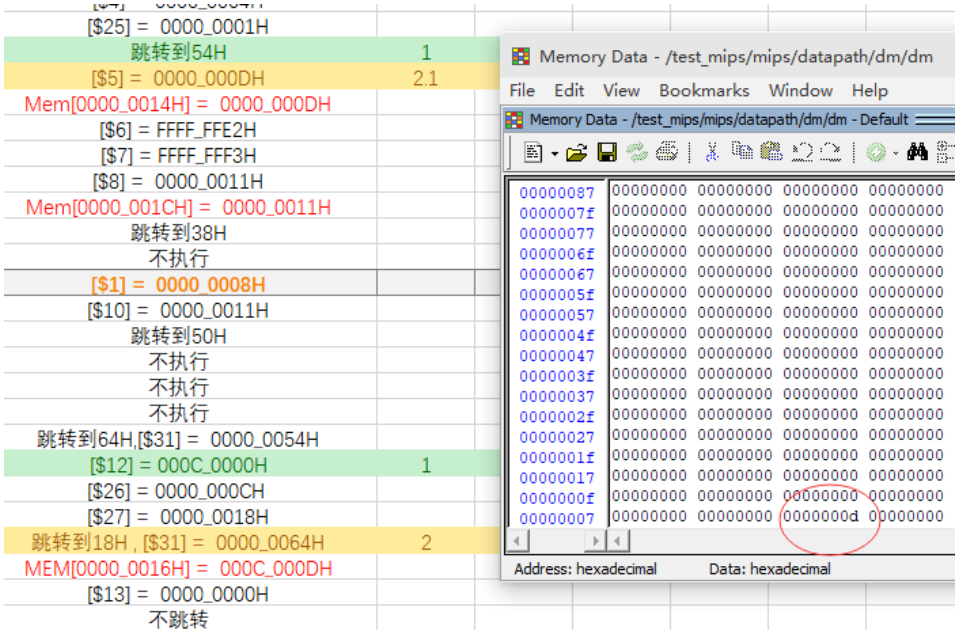
Alu 运算结果及寄存器内容



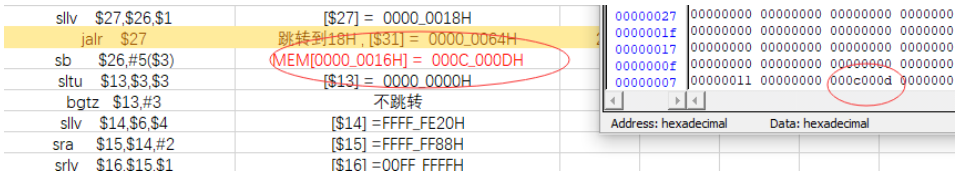


一开始 dm 的 sb 实现有点问题。后经过调试已更正。

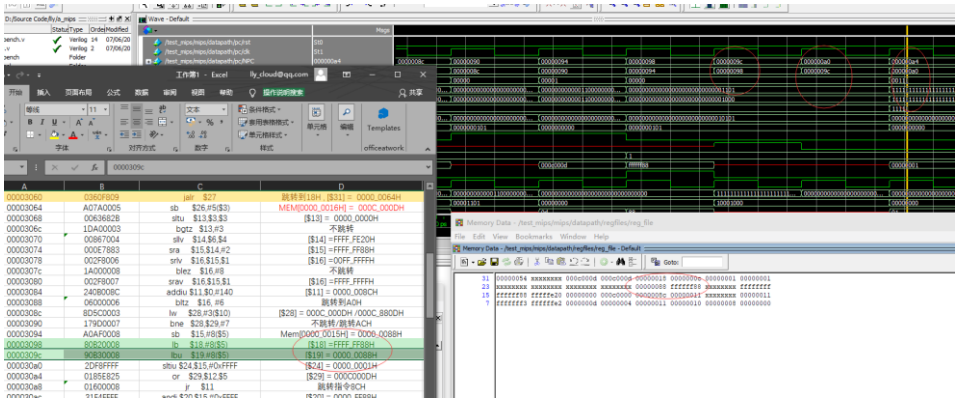
先是错误例子——



更正后——



lw/lb/lbi 等指令也通过了测试



6 心得体会

这一个学期学 cpu，说句实在话，感觉比较吃力，在单周期的任务表出来的时候，尝试过很多次去写去设计，却屡屡一晚上过去进度没有任何进展，于是有一个星期彻底的不想写计组试验了。可是想想，东西总归是要学的。于是，每周都会有硬着头皮看计组的时间。

前期做实验的时候感觉很多重要的东西都不明白。走了许多弯路，很多代码写的完全不知道什么意思，前期实验课的很多任务我感觉很多都只是在抄老师给的文档上的例子，当在让我们完成加减法运算器的时候，尚算比较艰难的完成了，当进入 11 条指令的设计的时候，真的尝试了不少次，也看了不少次书，但依旧很不理解，好在许多晚上的努力加上书与 ppt 的对于 11 条都有，所以最后算是比较顺利的完成了。

当我完成 11 条的时候，我感觉我不在是一头雾水了，我有了大概的框架。我有自信能写出 36 条来。但是，由于我是非常晚的时间才完成了最简单的 11 条。在计组的单周期的截至时间之前提交对于我看起来几乎是不可能的事情。加上 7 月 2 日的拉稀，我果然最终没能按时完成。

我身边有不少人早早放弃。但是，即使知道肯定无法截至时间之前完成，我也不想放弃。我在过了截至时间之后依然默默的写着自己未完成的任务。我做好了重修的准备，写只是为了让自己能真正的取学一些东西。

在实现的 36 条的控制器部分的时候，与 11 条所不同的是，指令的分析等工作，不再是书上比较容易找到，而是要真正的理解书上的内容并且加上自己的设计。虽然在代码的实现过程中，我发现了不少纰漏，但是，但一点点的改完这些错误之后，开始总结的时候，我发现我对很多东西理解的更为透彻了。

目前回过头来，发现自己有不少的问题是出在自己硬件思维的缺乏。

对于流水线，我扪心自问，我不会。因此，我也不愿拿着他人代码替换替换变量名混过去。同时，这些实验的历程走来，我发现，有些东西，只要肯硬着头皮干，总会会有收获的。

最终还是谢谢老师，虽然这门课任务重，题目难，但是老师真的很考虑我们的条件，在该验收指令的时候，因考试拖延，我们真的真的很感激老师！