计算机联合实验报告

计35 高胜寒 2013011400

计35 杨明 2013011412

计35 陈立康 2012011304

目录

[计算机联合实验报告（学号填一下） 1](#_Toc458959512)

[1. 大实验概述 2](#_Toc458959513)

[1.1 实验目标 2](#_Toc458959514)

[1.2 实验硬件环境（下面都完善一下） 2](#_Toc458959515)

[1.3 实验软件环境（完善） 2](#_Toc458959516)

[1.4 实验指令集（完善） 2](#_Toc458959517)

[2. 数据通路及各模块设计 2](#_Toc458959518)

[2.1 phymem模块 4](#_Toc458959519)

[2.1.1 模块功能概述 4](#_Toc458959520)

[2.1.2 mem访存信号 4](#_Toc458959521)

[2.1.3 模块数据接口 5](#_Toc458959522)

[2.2 decode模块 5](#_Toc458959523)

[2.2.1 模块功能概述 5](#_Toc458959524)

[2.2.2 模块数据接口 6](#_Toc458959525)

[2.3 alu模块 6](#_Toc458959526)

[2.3.1 模块功能概述 6](#_Toc458959527)

[2.3.2 ALU运算器信号 6](#_Toc458959528)

[2.3.3 模块数据接口 7](#_Toc458959529)

[2.4 cmp模块 7](#_Toc458959530)

[2.4.1 模块功能概述 7](#_Toc458959531)

[2.4.2 cmp跳转信号 8](#_Toc458959532)

[2.4.3 模块数据接口 8](#_Toc458959533)

[2.5 WriteBack模块 8](#_Toc458959534)

[2.5.1 wb写回信号 8](#_Toc458959535)

[3. 实验内容 9](#_Toc458959536)

[3.1 监控程序（完善一下） 9](#_Toc458959537)

[3.2 CPU 9](#_Toc458959538)

[4. 实验中遇到的困难 9](#_Toc458959539)

[4.1 环境配置问题（写一下） 9](#_Toc458959540)

[4.2 内存读写问题 10](#_Toc458959541)

[4.3 读写串口问题 10](#_Toc458959542)

[4.4 指令sb 10](#_Toc458959543)

# 大实验概述

## 实验目标

1. 设计并实现多周期CPU，支持MIPS32位的部分指令
2. 支持运行监控程序（部分完成）
3. 支持Flash自启动
4. 支持开关改变状态对CPU的中断

## 实验硬件环境

1. FPGA 芯片：XILINX Spartan-6E XC6SLX100TM FGG676BIV1337 D4615061A 2C
2. CPLD 芯片：XILINX XC95144XLTM TQG100AWN1349 D4658551A 10C
3. BASE SRAM：ISSI IS61WV102416BLL-10TLI D5A202S1 1348
4. EXTRA SRAM: ISSI IS61WV102416BLL-10TLI D2Q061S1 1213
5. Flash：JS28F640 J3D75 A1090243

## 实验软件环境

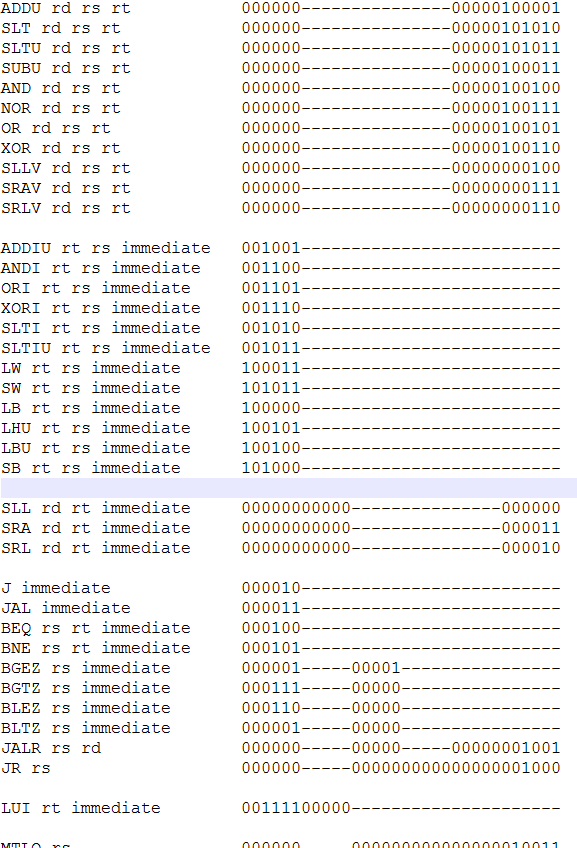
1. FPGA开发工具软件Xilinx ISE Design Suite 14.7
2. 16位CPU中用到的FlashAndRam.exe支持对flash的读写
3. 去年魏鑫鼎写的term.py以及32bitkernel.s, util.s，分别为运行监控程序的软件和监控程序的汇编码
4. mips-sde-elf-g++/mips-sde-elf-gcc，Mipsel-gnu-linux-g++/Mipsel-gnu-linux-gcc，完成从C++到二进制文件的转换
5. python，Qemu，Mars，分别用来编译终端term.py，模拟CPU以及辅助调试。
6. flashandram，用来烧录测试样例以及监控程序。

## 实验指令集

ISE枚举如下：

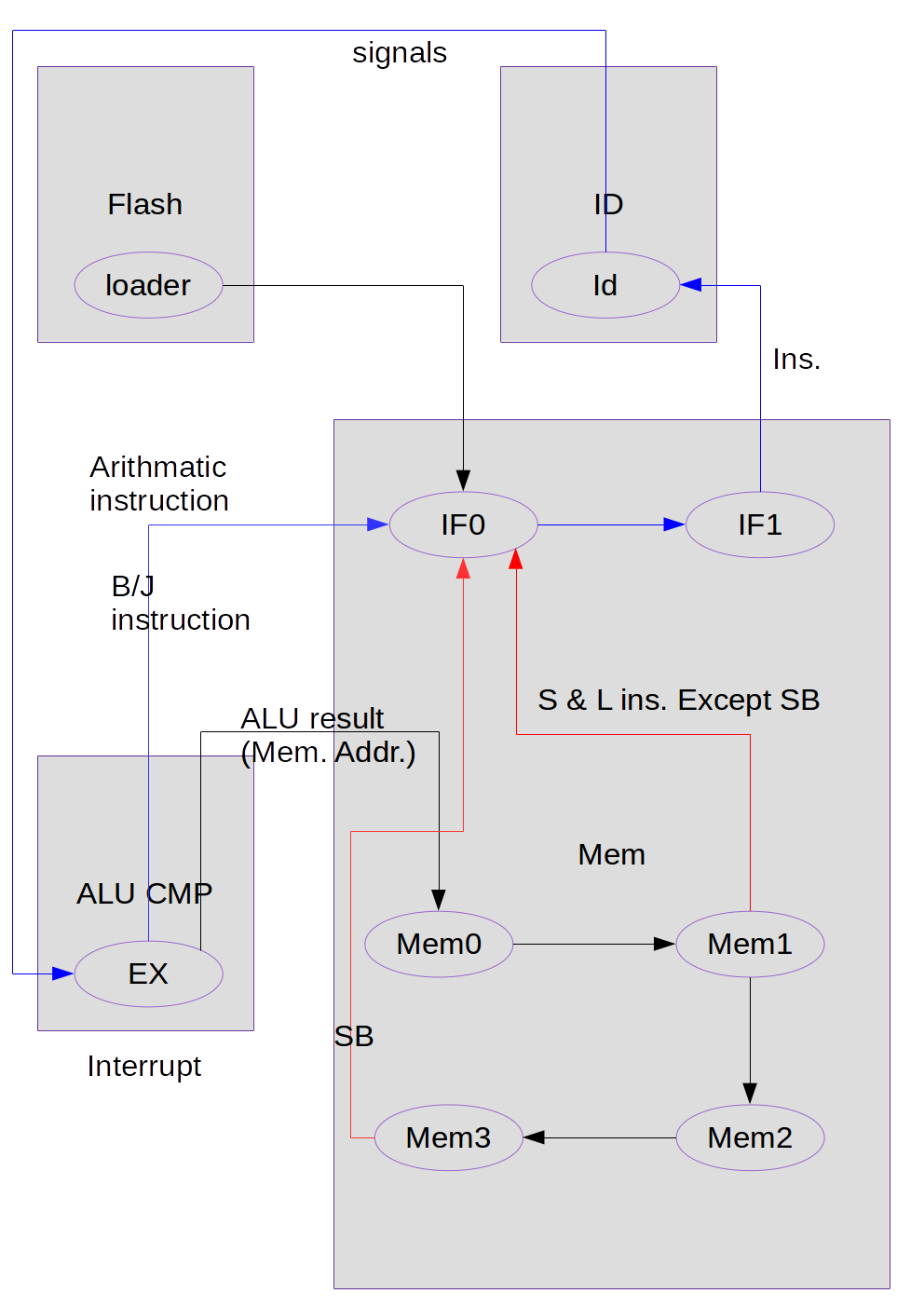
ADDIU ADDU SLT SLTI SLTIU SLTU SUBU MULT MFLO MFHI MTLO MTHI BEQ BGEZ BGTZ BLEZ BLTZ BNE J JAL JALR JR LW SW LB LBU LHU SB AND ANDI OR ORI NOR LUI XOR XORI SLL SLLV SRA SRAV SRL SRLV SYSCALL ERET MFC0 MTC0 TLBWI

以上指令共47条，编码方案参考下图：



# 数据通路及各模块设计

数据通路图状态机



寄存器标号

|  |  |
| --- | --- |
| 寄存器编号 | 对应寄存器 |
| 000000 | $zero |
| 000001 | $at |
| 000010, 000011 | $v0, $v1 |
| 000100 - 000111 | $a0 - $a3 |
| 001000 - 001111 | $t0 - $t7 |
| 010000 - 010111 | $s0 - $s7 |
| 011000, 011001 | $t8, $t9 |
| 011100 | $gp |
| 011101 | $sp |
| 011110 | $fp |
| 011111 | $ra |
| 100000 – 101101 | CP0 |
| HILO | HI, LO（乘法用到的寄存器） |

CPU多周期状态

|  |  |
| --- | --- |
| CPU状态state，类型status | 内容 |
| S0-S6 | 用于flashloader的读取部分 |
| IF0 | 在sram地址线中准备好访存地址PC，开启读使能，准备读取指令，上一条指令寄存器写回 |
| IF1 | 读出指令，将结果保存，关闭读使能 |
| ID | 指令译码，生成所有以后要用到的信号，读出源寄存器和目标寄存器的数值 |
| EX | alu根据信号进行计算，cmp根据信号计算新的PC值 |
| MA0 | 访问存储模块，准备好访存地址，若为写则向数据线写入源寄存器的值，开启相应使能 |
| MA1 | 访问存储模块，若为读则读取数据线上的数值，关闭相应使能 |
| MA2 | 同MA0，仅在sb指令会执行到，此外还需要计算出需要写入的数值 |
| MA3 | 同MA1，仅在sb指令会执行到 |
| ST | 停止 |

CPU全局寄存器

|  |  |
| --- | --- |
| 寄存器名 | 内容 |
| clk | 由50MHz时钟和clk按键复合而成的时钟，作为CPU时钟 |
| pc(31 to 0) | 当前pc |
| reg(63 to 0)(31 to 0) | 通用寄存器、CP0寄存器 |
| rs\_addr(5 to 0) | 源寄存器s地址 |
| rt\_addr(5 to 0) | 源寄存器t地址 |
| rd\_addr(5 to 0) | 目标寄存器d地址 |
| alu\_signal(5 to 0) | 运算器信号 |
| cmp\_signal(3 to 0) | 跳转信号 |
| mem\_signal(2 to 0) | 访存信号 |
| ram\_signal(1 to 0) | 访存信号 |
| wb\_signal(1 to 0) | 写回信号 |
| rs\_value(31 to 0) | 源寄存器s数值 |
| rt\_value(31 to 0) | 源寄存器t数值 |
| rd\_value(31 to 0) | 目标寄存器d数值 |
| alu\_result(31 to 0) | 运算器结果 |
| npc(31 to 0) | 译码模块给出的跳转成功的地址 |
| imme(31 to 0) | 立即数 |
| addr\_in(19 to 0) | 准备的访存地址 |
| data\_in(31 to 0) | 准备的访存数据 |
| ram\_out(31 to 0) | 读取ram的数据 |
| flash\_out(31 to 0) | 读取flash的数据 |
| mem\_data(31 to 0) | 当前等同于rt\_value |
| state | CPU状态 |
| spc | 保存的中断时pc的地址 |
| lsw | 保存中断开关的状态 |

## phymem模块

该模块一次运作占用2个时钟周期，在一条指令中将使用1至3次，对应与一条指令的取指、访存1、访存2部分，对应的CPU的状态state为IF0，IF1，MA0，MA1，MA2，MA3。

在初始程序的flashloader阶段时也用到了该模块，对应的CPU的状态state为S5和S6。

### 模块功能概述

1. 从寄存器PC中读出指令地址
2. 访问寄存器RAM0，取指部分从中读出32位指令，访存部分读出或写入32位数据
3. 返回串口访问状态
4. 访问串口，支持对串口的读写

### mem访存信号

由alu生成的访存信号共3位，而phymem输入的有2位

|  |  |
| --- | --- |
| alu生成的信号操作号 | 操作 |
| 000 | 不访存 |
| 001 | sw |
| 010 | sb |
| 100 | lw |
| 101 | lhu |
| 110 | lb |
| 111 | lbu |

|  |  |
| --- | --- |
| phymem需要的信号操作号 | 操作 |
| 0x | 不访存 |
| 10 | 写操作 |
| 11 | 读操作 |

### 模块数据接口

|  |  |  |
| --- | --- | --- |
| 接口名称 | 类型 | 说明 |
| clk | In | CPU时钟 |
| rst | In | reset按键输入 |
| state | In | CPU的状态，status类型 |
| addr(19 to 0) | In | 访存地址（包含访问串口） |
| data(31 to 0) | In | 准备写入的数据（取指段一定不会使用） |
| ram\_signal(1 to 0) | In | 表示访存控制信号 |
| ram\_addr(19 to 0) | Out | sram地址线 |
| ram\_data(31 to 0) | Inout | sram数据线 |
| data\_out(31 to 0) | Out | 读操作的数据读取结果 |
| ram\_ce | Out | sram的使能 |
| ram\_oe | Out | sram的读取使能 |
| ram\_we | Out | sram的写入使能 |
| data\_ready | In | 串口控制信号 |
| write\_ready | In | 串口控制信号 |
| rdn | Out | 串口读取使能 |
| wrn | Out | 串口写入使能 |

## decode模块

该模块一次运作占用1个时钟周期，在1条指令中执行1次，对应CPU状态state为ID

### 模块功能概述

1. 从指令操作码中译出ALU，PC，MEM，WB的控制信号、源寄存器目标寄存器编号、立即数（对立即数进行符号扩展或0扩展）、跳转地址、NPC，存入全局的寄存器。
2. 从2个源寄存器以及目标寄存器中读出3个操作数，作为ALU模块的输入。

### 模块数据接口

|  |  |  |
| --- | --- | --- |
| 接口名称 | 类型 | 说明 |
| clk | In | CPU时钟 |
| rst | In | reset按键输入 |
| pc(31 to 0) | In | PC寄存器 |
| ins(31 to 0) | In | 32位指令 |
| state | In | CPU状态，status类型 |
| npc(31 to 0) | Out | npc跳转地址 |
| rs\_addr(5 to 0) | Out | 源寄存器s地址 |
| rt\_addr(5 to 0) | Out | 源寄存器t地址 |
| rd\_addr(5 to 0) | Out | 目标寄存器d地址 |
| imme(31 to 0) | Out | 立即数 |
| mem\_signal(2 to 0) | Out | 访存信号 |
| alu\_signal(5 to 0) | Out | 运算器信号 |
| cmp\_signal(3 to 0) | Out | 跳转信号 |
| wb\_signal(1 to 0) | Out | 写回信号 |

## alu模块

该模块一次运作占用1个时钟周期，在1条指令中执行一次，对应CPU状态state为EX

### 模块功能概述

1. 根据运算器信号得到alu的计算结果，存入全局寄存器
2. 将源寄存器t的值存入全局寄存器（作为写指令的数据）
3. 保存HILO寄存器，以及完成相关指令

### ALU运算器信号

alu信号共有6位，最高位表示是否使用立即数代替一个源寄存器，1表示使用立即数，0表示寄存器。

|  |  |
| --- | --- |
| 操作号 | 操作 |
| 10000 | s\*t结果保存至HILO |
| 10001 | HI <= s |
| 10010 | LO <= s |
| 10011 | s <= HI |
| 10100 | s <= LO |
| 00000 | s+t |
| 00001 | s-t |
| 00010 | s and t |
| 00011 | s or t |
| 00100 | s xor t |
| 00101 | s nor t |
| 00110 | t sll s |
| 00111 | t sra s |
| 01000 | t srl s |
| 01001 | s < t (signed) |
| 01010 | d |
| 01011 | t |
| 01100 | s < t (unsigned) |

### 模块数据接口

|  |  |  |
| --- | --- | --- |
| 接口名称 | 类型 | 说明 |
| clk | In | CPU时钟 |
| rst | In | reset按键输入 |
| rs\_value(31 to 0) | In | 源寄存器s的值 |
| rt\_value(31 to 0) | In | 源寄存器t的值 |
| rd\_value(31 to 0) | In | 目标寄存器d的值 |
| imme(31 to 0) | In | 立即数 |
| alu\_signal(5 to 0) | In | 运算器信号 |
| state | In | CPU状态，类型status |
| alu\_result(31 to 0) | Out | 运算器计算结果 |
| data\_out(31 to 0) | Out | 源寄存器t的值 |

## cmp模块

该模块一次运作占用1个时钟周期，在1条指令中执行一次，对应CPU状态state为EX

### 模块功能概述

1. 根据跳转信号，由寄存器和立即数得到新的跳转PC值
2. 响应来自开关状态变化（与上一条指令状态不同则视为一次中断），跳转到中断处（0x60000000）执行

### cmp跳转信号

跳转信号共4位，用于计算下一条指令的

|  |  |
| --- | --- |
| 操作号 | 操作 |
| 0000 | 不跳转pc <= pc + lpc |
| 0001 | s = t |
| 0010 | s /= t |
| 0011 | s > 0 |
| 0100 | s >= 0 |
| 0101 | s < 0 |
| 0110 | s <= 0 |
| 0111 | 跳转 |
| 1111 | ps <= s |
| 1000 | eret，pc <= spc |
| 中断 | spc <= pc，pc <= ipc（中断程序地址） |

### 模块数据接口

该模块是主模块cpu中的一个process，可以将其分离为单独模块。

若分离，则该模块将需要rst（reset按键）, clk（cpu时钟）, pc（原pc值）, npc（跳转地址）, cmp\_signal（跳转信号）, rs\_addr（源寄存器s地址）, rt\_addr（源寄存器t地址）, sw（中断信号）作为输入，pc（新pc值）作为输出。

## WriteBack模块

若指令为写回操作，则将数值写入目标寄存器。

### wb写回信号

跳转信号共4位，用于计算下一条指令的

|  |  |
| --- | --- |
| 操作号 | 操作 |
| 00 | 不写回 |
| 01 | 写回到t地址 |
| 10 | 写回到d地址 |
| 11 | 写回到$ra |

# 实验内容

## 监控程序

kernel.ld文件：

ENTRY(\_\_start)

SECTIONS

{

ident : {\*(.text)}

. = 0x80100000;

.text : { \*(.text) }

.ktext : { \*(.ktext) }

.data : { \*(.data) }

.bss : { \*(.bss) }

. = . + 0x1000; /\* 4kB of stack memory \*/

stack\_top = .;

}

32bitkernel.s文件：

.global \_\_start

.global \_\_ident

.text

.include "util.s"

\_\_start:

.ifeq (SerialMode - UARTMode)

InitUART

.endif

SendByte\_Imm 0x004F

SendByte\_Imm 0x004B

SendByte\_Imm 0x000A

SendByte\_Imm 0x000D

# init users' registers

lui $t0, RegDumpHAddr

addiu $t0, $t0, RegDumpLAddr

li $t1, 0

li $t2, 32

li $t3, 0

RegInitLoop:

addu $t4, $t0, $t3

sw $zero, ($t0)

addiu $t1, $t1, 1

addiu $t3, $t3, 4

bne $t1, $t2, RegInitLoop

BEGIN:

# read an operation

ReadByte

addiu $t3, $v0, 0

# check R operation

li $t0, 0x52

beq $t3, $t0, SHOWREGS

nop

# check D operation

li $t0, 0x44

beq $t3, $t0, SHOWMEM

nop

# check A operation

li $t0, 0x41

beq $t3, $t0, EDITMEM

nop

# check U operation

li $t0, 0x55

#beq $t3, $t0, SHOWREGS

nop

# check G operation

li $t0, 0x47

beq $t3, $t0, GOTOCOMPILE

nop

b BEGIN

nop

# print all registers's value

SHOWREGS:

lui $t5, RegDumpHAddr

addiu $t5, $t5, RegDumpLAddr + 0x4

addiu $t6, $t5, 0x74

jal SENDMEM

nop

SendByte\_Imm Finish

b BEGIN

nop

SHOWMEM:

ReadWord

addiu $t5, $v0, -4

ReadWord

addiu $t6, $v0, -4

jal SENDMEM

nop

SendByte\_Imm Finish

b BEGIN

nop

EDITMEM:

ReadWord

addiu $t6, $v0, 0

ReadWord

addiu $t5, $v0, 0

beq $t5, $t6, MEMLOAD\_END

nop

MEMLOAD\_LOOP:

ReadByte

sb $v0, ($t6)

addiu $t6, $t6, 1

bne $t5, $t6, MEMLOAD\_LOOP

nop

MEMLOAD\_END:

# the procedure may take too long time, need sync

SendByte\_Imm Finish

b BEGIN

nop

GOTOCOMPILE:

ReadWord

addiu $ra, $v0, 0

# recover register

recover\_reg

jal $ra

nop

# dump register

restore\_reg

SendByte\_Imm Finish

b BEGIN

nop

util.s文件：

# TODO: it is MMIO address of MARS simulator,

# so there need be changed when the code runs on THICO board.

.equ SerialMode, 0

.equ MMIOMode, 0

.equ UARTMode, 1

.equ MMIOWHAddr, 0x1fd0

.equ MMIOWLSAddr, 0x03fc

.equ MMIOWLCAddr, 0x03f8

.equ MMIORHAddr, 0x1fd0

.equ MMIORLSAddr, 0x03fc

.equ MMIORLCAddr, 0x03f8

# register dump address, takes 32 \* 4 = 128 Bytes

# Reg2($v0) will store in 0xBF00008, Reg3($v1) will store in 0xBF00000C

# $zero, $at, $ra will not be stored

.equ RegDumpHAddr, 0x8000

.equ RegDumpLAddr, 0x0000

# UART address offset

.equ UARTHAddr, 0xbfd0

.equ UARTLAddr, 0x03f8

# magic char of Term protocol

.equ Finish, 0x0000

# macro function define

# macro reg restore operation loop

# used $at register

.macro restore\_reg

.set noat

lui $at, RegDumpHAddr

addiu $at, $at, RegDumpLAddr

sw $v0, 0x08($at)

sw $v1, 0x0C($at)

sw $a0, 0x10($at)

sw $a1, 0x14($at)

sw $a2, 0x18($at)

sw $a3, 0x1C($at)

sw $t0, 0x20($at)

sw $t1, 0x24($at)

sw $t2, 0x28($at)

sw $t3, 0x2C($at)

sw $t4, 0x30($at)

sw $t5, 0x34($at)

sw $t6, 0x38($at)

sw $t7, 0x3C($at)

sw $s0, 0x40($at)

sw $s1, 0x44($at)

sw $s2, 0x48($at)

sw $s3, 0x4C($at)

sw $s4, 0x50($at)

sw $s5, 0x54($at)

sw $s6, 0x58($at)

sw $s7, 0x5C($at)

sw $t8, 0x60($at)

sw $t9, 0x64($at)

sw $k0, 0x68($at)

sw $k1, 0x6C($at)

sw $gp, 0x70($at)

sw $sp, 0x74($at)

sw $fp, 0x78($at)

.set at

.endm

# macro reg recover operation loop

# used $at register

.macro recover\_reg

.set noat

lui $at, RegDumpHAddr

addiu $at, $at, RegDumpLAddr

lw $v0, 0x08($at)

lw $v1, 0x0C($at)

lw $a0, 0x10($at)

lw $a1, 0x14($at)

lw $a2, 0x18($at)

lw $a3, 0x1C($at)

lw $t0, 0x20($at)

lw $t1, 0x24($at)

lw $t2, 0x28($at)

lw $t3, 0x2C($at)

lw $t4, 0x30($at)

lw $t5, 0x34($at)

lw $t6, 0x38($at)

lw $t7, 0x3C($at)

lw $s0, 0x40($at)

lw $s1, 0x44($at)

lw $s2, 0x48($at)

lw $s3, 0x4C($at)

lw $s4, 0x50($at)

lw $s5, 0x54($at)

lw $s6, 0x58($at)

lw $s7, 0x5C($at)

lw $t8, 0x60($at)

lw $t9, 0x64($at)

lw $k0, 0x68($at)

lw $k1, 0x6C($at)

lw $gp, 0x70($at)

lw $sp, 0x74($at)

lw $fp, 0x78($at)

.set at

.endm

.macro WriteMMIO\_I byte\_data

jal TESTW

nop

lui $t0, MMIOWHAddr

addiu $t0, $t0, MMIOWLCAddr

li $t1, \byte\_data

sw $t1, ($t0)

.endm

.macro WriteMMIO\_R register\_name

jal TESTW

nop

lui $t0, MMIOWHAddr

addiu $t0, $t0, MMIOWLCAddr

andi \register\_name, \register\_name, 0xff

sw \register\_name, ($t0)

.endm

.macro ReadMMIO

jal TESTR

nop

lui $t0, MMIORHAddr

addiu $t0, $t0, MMIORLCAddr

lw $v0, ($t0)

li $t0, 0xff

and $v0, $v0, $t0

.endm

.macro InitUART

# init uart

# magic, don't modify it

lui $t0, UARTHAddr

addiu $t0, $t0, UARTLAddr

li $t1, 0x00

sb $t1, 0x2($t0)

li $t1, 0x80

sb $t1, 0x3($t0)

li $t1, 0x0C

sb $t1, 0($t0)

li $t1, 0x00

sb $t1, 0x1($t0)

li $t1, 0x03

sb $t1, 0x3($t0)

li $t1, 0x00

sb $t1, 0x4($t0)

li $t1, 0x01

sb $t1, 0x1($t0)

.endm

.macro WriteUART\_I byte\_data

jal TESTW\_UART

nop

li $t1, \byte\_data

sb $t1, ($t0)

.endm

.macro WriteUART\_R register\_name

jal TESTW\_UART

nop

andi \register\_name, \register\_name, 0xff

sb \register\_name, ($t0)

.endm

.macro ReadUART

jal TESTR\_UART

nop

lb $v0, ($t0)

li $t0, 0xff

and $v0, $v0, $t0

.endm

# Send byte data in register

.macro SendByte\_Reg register\_name

.ifeq (SerialMode - UARTMode)

WriteUART\_R \register\_name

.endif

.ifeq (SerialMode - MMIOMode)

WriteMMIO\_R \register\_name

.endif

.endm

# Send Immediate byte data

.macro SendByte\_Imm byte\_data

.ifeq (SerialMode - UARTMode)

WriteUART\_I \byte\_data

.endif

.ifeq (SerialMode - MMIOMode)

WriteMMIO\_I \byte\_data

.endif

.endm

# Send register data through $at, send by little endian order

.macro SendRegister register\_name

.set noat

SRL $at, \register\_name, 0

SendByte\_Reg $at

SRL $at, \register\_name, 8

SendByte\_Reg $at

SRL $at, \register\_name, 16

SendByte\_Reg $at

SRL $at, \register\_name, 24

SendByte\_Reg $at

.set at

.endm

# Read byte, return to $v0

.macro ReadByte

.ifeq (SerialMode - UARTMode)

ReadUART

.endif

.ifeq (SerialMode - MMIOMode)

ReadMMIO

.endif

.endm

# Read a word, return to $v0

.macro ReadWord

li $t4, 0

ReadByte

addiu $t4, $v0, 0

ReadByte

sll $v0, $v0, 8

addu $t4, $t4, $v0

ReadByte

sll $v0, $v0, 16

addu $t4, $t4, $v0

ReadByte

sll $v0, $v0, 24

addu $v0, $t4, $v0

.endm

# Code function

TESTW:

lui $t0, MMIOWHAddr

addiu $t0, $t0, MMIOWLSAddr

lw $t1, ($t0)

andi $t1, $t1, 1

beqz $t1, TESTW

nop

jr $ra

nop

TESTR:

lui $t0, MMIORHAddr

addiu $t0, $t0, MMIORLSAddr

lw $t1, ($t0)

andi $t1, $t1, 2

beqz $t1, TESTR

nop

jr $ra

nop

TESTW\_UART:

lui $t0, UARTHAddr

addiu $t0, $t0, UARTLAddr

lb $t1, 0x5($t0)

andi $t1, $t1, 0x20

beqz $t1, TESTW\_UART

nop

jr $ra

nop

TESTR\_UART:

lui $t0, UARTHAddr

addiu $t0, $t0, UARTLAddr

lb $t1, 0x5($t0)

andi $t1, $t1, 0x01

beqz $t1, TESTR\_UART

nop

jr $ra

nop

# send $t5 + 4 to $t6, return to $ra

SENDMEM:

addiu $t4, $ra, 0

SENDMEM\_Loop:

addiu $t5, $t5, 0x4

lw $t7, ($t5)

SendRegister $t7

bne $t5, $t6, SENDMEM\_Loop

nop

addiu $ra, $t4, 0

jr $ra

nop

运行脚本：

mips-sde-elf-gcc -mips32 -EL -c 32bitkernel.s -o kernel.o

mips-sde-elf-ld -mips32 -EL -T kernel.ld kernel.o -o kernel

mipsel-linux-gnu-objcopy -O binary kernel kernel.bin

python trans.py kernel.bin k\_modify.bin

rm kernel.s

mipsel-linux-gnu-objdump -b binary -mmips:isa32 -EL -D kernel.bin > kernel.s

rm k\_modify.s

mipsel-linux-gnu-objdump -b binary -mmips:isa32 -EL -D k\_modify.bin > k\_modify.s

前两条命令是从监控程序的汇编文件爱你编译为elf格式的文件，之后一句命令是将elf文件转换为二进制文件，紧接着的命令是调用我们写的一个python脚本调整二进制文件的执行顺序，最后的几句命令是生成反汇编文件，调试用。

## CPU

最终的CPU实现了基本的指令集，支持访问sram和串口（访问串口存在问题），支持一个简易的中断，支持flash的自启动。

开始时将FlashAndRam的bit文件下载到电路板中，将写好的程序的bin文件写入到电路板的flash中，然后重新将我们的CPU的bit文件下载到电路板中，运行即可。

使用简单的测例：

（需要修改CPU的flashloader部分的程序）将一个无限循环（J 0x00）作为主程序，将斐波那契数列的程序写入flash，在flashloader时将其写到中断程序（0x60000000）处（原来期望使用监控程序完成），CPU开始执行时改变中断开关，程序进入中断，执行完毕后使用eret回到主程序，程序再次进入无限循环，使用信号灯可以得知斐波那契的第20项的结果为1A6D。这样我们同时测试了中断的正确性。

使用监控程序进行测试的结果：

将之前修改的监控程序写入flash中，其余操作如之前所述，运行程序，可以得到OK，此后可以输入各条指令，如’R’，但由于串口接收存在一定的问题，所以并不能得到想要的结果。

# 实验中遇到的困难

由于之前有16位CPU的经验，我们认为在一些模块中直接使用之前的程序（或者仅做简单的修改）就可以达到预期的效果。所以我们在一开始并没有单独重新写工程详细测试每一个的模块（尤其是访存、串口等），事后我们才发现模块单独测试的重要性。事实上由于一些微妙的变化，之前在16位CPU中的代码在新的电路板中并不都能正常运行，给了后期调试不小的困难，我们在内存读写、串口读写上花费了很多不必要的时间。

## 环境配置问题

windows下硬件设计开发环境搭建

硬件设计平台的搭建：ISE

Design Suite 14.7

下载网址：

<http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_4---14_7.html>

配置说明：选择 Full Installer for Windows(TAR/GZIP - 6.18 GB) 一项（需注册XILINX账户，才能下载试用版）

Linux平台下，Design Suite 14.7在与THINPAD进行JTAG连接时会出问题，这个暂时没有解决方案，除非下载ISE Design Suit之前的版本。

在windows平台下，这个问题的解决方案见下面的描述。

假设ise安装在D盘根目录下，那么可以运行

D:\Xilinx\14.7\ISE\_DS\common\bin\nt64\install\_drivers.ex

即可进行驱动安装，然后就可以正常进行JTAG连接了。

win10下ISE Design suite遇到了一些新问题，就是软件在访问文件系统时，如果目标文件的文件路径中含有空格等不可见字符，系统就会崩溃，这个解决方案如下：

还是假设ise安装在D盘根目录下，那么找到以下两个文件夹

D:\Xilinx\14.7\ISE\_DS\ISE\lib\nt64

D:\Xilinx\14.7\ISE\_DS\common\lib\nt64

在第一个文件中，重命名libPortability.dll为libPortability.dll.orig，然后复制libPortabilityNOSH.dll

的一个副本并重命名为libPortability.dll，在第二个文件夹，将之前得到的新的libPortability.dll覆盖到这个文件夹中。

这样，文件系统就可以正常访问了。

编译环境搭建与使用

编译分为3步，首先是从C/C++到MIPS汇编，其次是从MIPS汇编到elf格式文件，最后是从elf文件到可烧入FLASH的bin文件。

我们的编译环境分为三个部分，第一部分完成编译的前两步，第二部分完成编译的最后一步，并支持反汇编以检查二进制文件，第三部分辅助调试。

编译环境Part 1：

mips-sde-elf-g++/mips-sde-elf-gcc：

可从https://sourcery.mentor.com/GNUToolchain/release2009下载，并将所有工具加入环境变量。该工具主要用来编译监控程序kernel，使用方法：

mips-sde-elf-gcc-mips32 -EL -c 32bitkernel.s -o kernel.o

mips-sde-elf-ld-mips32 -EL -T kernel.ld kernel.o -o kernel

其中-EL参数代表小端序，需要明确并注意。

编译环境Part 2：

mipsel-gnu-linux-g++/Mipsel-gnu-linux-gcc：

如果是Ubuntu debian，可以使用apt-get工具下载。该工具主要用来实现从elf文件转为二进制文件，以及对二进制文件进行反汇编。

从elf转换为二进制文件，参考http://stackoverflow.com/questions/19944441/make-executable-binary-file-from-elf-using-gnu-objcopy，设需转换的elf文件名为myfile.elf，则命令为：

objcopy -O binary myfile.elf myfile.bin

将二进制文件反汇编，参考http://stackoverflow.com/questions/9916130/how-do-i-disassemble-raw-mips-code

命令为：

mipsel-linux-gnu-objdump -b binary -mmips:isa32 -EL -D xxx

其中xxx为需查看的二进制文件，其中编译参数-EL为小端序，需要明确与注意。

编译环境Part 3：

Python：

用来编译终端term.py文件，使用2.7版本，需pip安装pyserial包。

Qemu模拟器：

可通过http://wiki.qemu.org/Download下载，Ubuntu Debian下可以通过apt-get工具安装。

Mar:

一个java写的MIPS汇编模拟器，支持反汇编、模拟执行MIPS代码，我们附录了其jar文件Mars4\_5.jar，用java

-jar Mars4\_5.jar运行。

Qemu模拟器与term.py终端的连接：

Linux下，如果正常连入设备，可以用

ls

-l /dev/ttyS\*

命令查看连入设备的串口名，如果使用Qemu模拟器，先运行kernel目录下的runQemu.sh脚本，命令行会显示模拟器的串口文件路径。在得到串口文件路径后（设其为xxx）,在term文件夹下执行命令

python

term.py xxx

即可连接term，对硬件或模拟器进行监控。

windows平台下类似，只需将串口文件路径xxx替换为串口号(假设为COM1)

python

term.py COM1

FLAH烧录程序的搭建：

使用实验室开发的flashandram即可。

## 内存读写问题

在开始在电路板上面调试时，一开始我们遇到了内存的读写问题。

由于启动时需要从flash中读取数据，我们一开始的时候我们并没有尝试过内存的读写，而FlashAndRam工程中使用的是16位的Ram，也无法正常的访问内存。

后来通过观察ram专用的32位信号灯，发现数据线的最低8位不能正常访问，根据在做16位CPU时的经验，我们意识到是串口的问题，于是修复了这一问题。

## 读写串口问题

在开始监控程序的调试时，会遇到与计算机通信的问题，于是我们移植了16位读写串口的程序，但在绑定管脚的时候发现实验指导书中并没有给出管脚的具体编号。由于我们发现FlashAndRam也用到串口，所以就参考了FlashAndRam的管脚，此外我们还找到了另一个版本的串口程序，但两者都不能正常读写（我们并没有得知问题所在）。

## 指令sb

对于所有其它的指令，最多都可以在取指、译码、运算、访存4个阶段内完成，但是电路板并不能支持对1个字节的修改，所以需要先读取相应的4字节32位，此后修改相应字节后存储。开始的时候并没有考虑到有这种情况，所以在最初的数据通路中遇到了不小的麻烦，甚至进行了比较大的改动，最终演变成了目前的数据通路（状态机）。