



中国科学技术大学
University of Science and Technology of China

计算机组成原理

Lab3 汇编程序设计

计算机实验教学中心

2023-4-10

实验目标

- 理解RISC-V常用32位整数指令功能
- 熟悉RISC-V汇编仿真软件RARS，掌握程序调试的基本方法
- 掌握RISC-V简单汇编程序设计，以及存储器初始化文件(COE)的生成方法
- 理解CPU调试模块PDU的使用方法

实验原理

1.RV32I寄存器：PC和32个通用寄存器

Register	ABI Name	Description
x0	zero	Hard-wired zero 硬编码 0
x1	ra	Return address 返回地址
x2	sp	Stack pointer 栈指针
x3	gp	Global pointer 全局指针
x4	tp	Thread pointer 线程指针
x5	t0	Temporary/alternate link register
x6–7	t1–2	Temporaries 临时寄存器
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register 保存寄存器
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments 函数参数
x18–27	s2–11	Saved registers 保存寄存器
x28–31	t3–6	Temporaries 临时寄存器

RV32I寄存器汇编助记符

实验原理

2.RV32I指令类型

□ 运算类

✓ **算术**: add, sub, addi,

auipc, lui

✓ **逻辑**: and, or, xor, andi,

ori, xori

✓ **移位(shift)**: sll, srl, sra,

slli, srli, srai

✓ **比较(set if less than)**: slt,

sltu, slti, sltiu

Category	Name	Fmt	RV32I Base	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm

实验原理

2.RV32I指令类型

□ 访存类

✓ 加载(load): **lw**, lb, lbu, lh, lhu

✓ 存储(store): **sw**, sb, sh

□ 转移类

✓ 分支(branch): **beq**, **blt**, bltu, bne, bge, bgeu

✓ 跳转(jump): **jal**, **jalr**

<i>Category</i>	<i>Name</i>	<i>Fmt</i>	<i>RV32I Base</i>	
Branches	Branch =	B	BEQ	rs1,rs2,imm
	Branch ≠	B	BNE	rs1,rs2,imm
	Branch <	B	BLT	rs1,rs2,imm
	Branch ≥	B	BGE	rs1,rs2,imm
	Branch < Unsigned	B	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm
Jump & Link	J&L	J	JAL	rd,imm
	Jump & Link Register	I	JALR	rd,rs1,imm
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm

实验原理

3.RV32I指令格式及功能

□ 运算指令

✓ **add rd, rs1, rs2**

$x[rd] = x[rs1] + x[rs2]$

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

✓ **addi rd, rs1, imm**

$x[rd] = x[rs1] + \text{sext}(\text{imm})$

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM	
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM	

实验原理

3.RV32I指令格式及功能

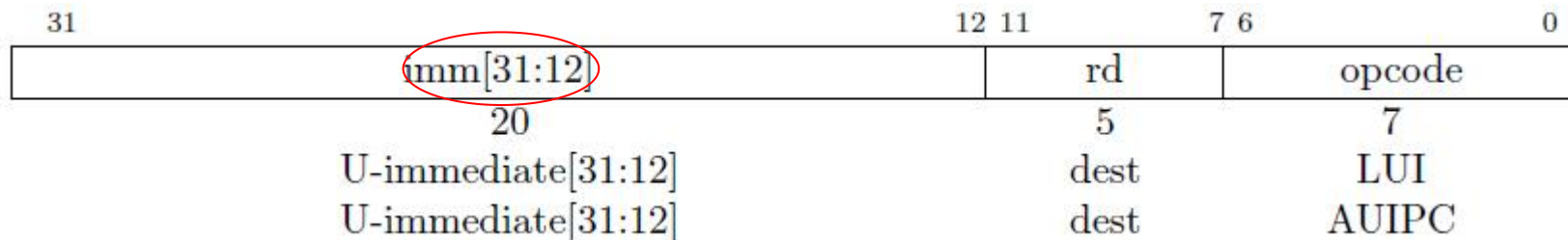
□ 运算指令

✓ lui rd, imm

$x[rd] = \text{sext}(\text{imm}[31:12] \ll 12)$

✓ auipc rd, imm

$x[rd] = \text{pc} + \text{sext}(\text{imm}[31:12] \ll 12)$



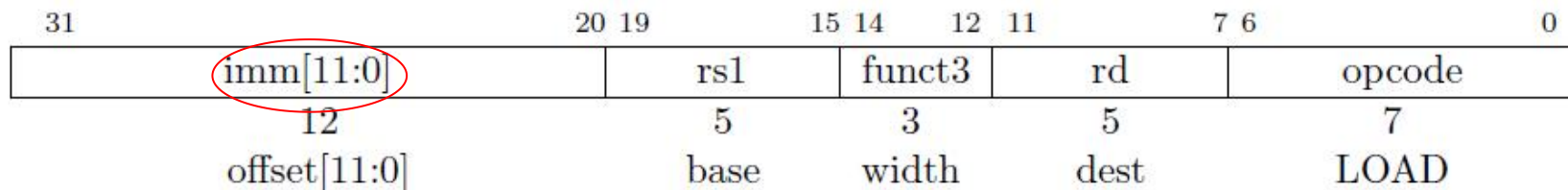
实验原理

3.RV32I指令格式及功能

□ 访存指令

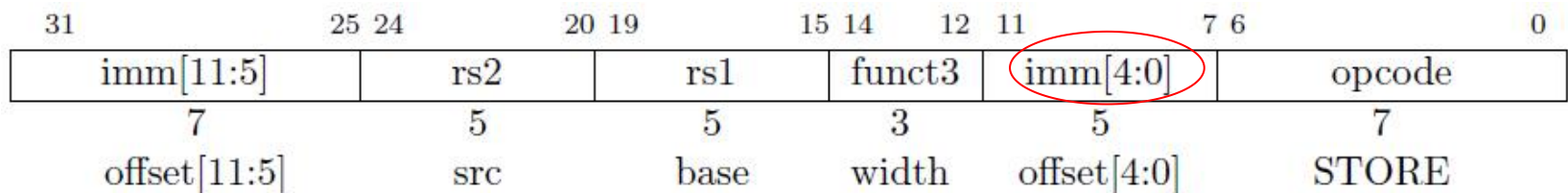
✓ lw rd, offset(rs1)

$x[rd] = M[x[rs1] + sext(offset)]$



✓ sw rs2, offset(rs1)

$M[x[rs1] + sext(offset)] = x[rs2]$



实验原理

3.RV32I指令格式及功能

□ 分支指令

✓ beq rs1, rs2, offset

if (rs1 == rs2) pc += sext(offset)

✓ blt rs1, rs2, offset

if (rs1 < rs2) pc += sext(offset)

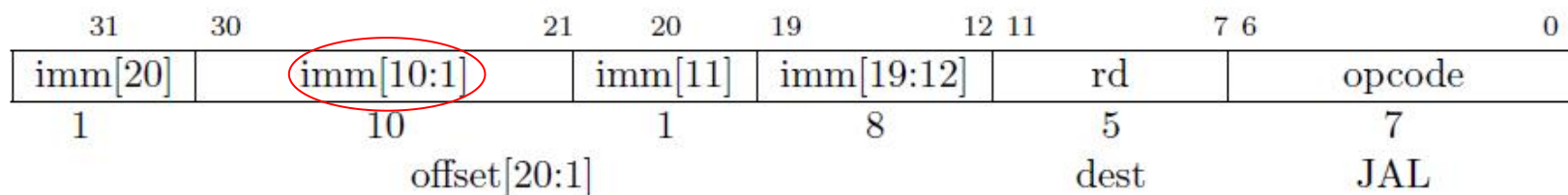
31	30	25	24	20	19	15	14	12	11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode						
1	6	5	5	3	4	1	7						
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]	BRANCH							
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]	BRANCH							
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]	BRANCH							

实验原理

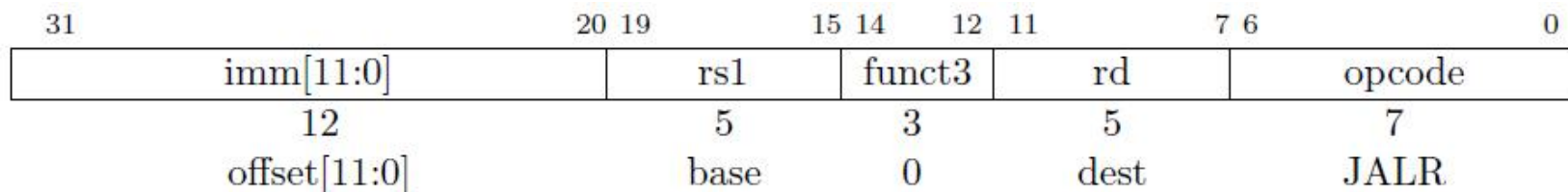
3.RV32I指令格式及功能

□ 跳转指令

✓ **jal rd, offset** **# x[rd] = pc+4; pc += sext(offset)**



✓ **jalr rd, offset(rs1)** **# t = pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t**



实验原理

4. 汇编指示符和伪指令

□ 汇编指示符 (Assembly Directives)

.data, .text

.word, .half, .byte, .string

.align

□ 伪指令 (Pseudo Instructions)

li, la, mv

nop, not, neg

j, jr, call, ret

Example:

```
.eqv CONSTANT, 0xdeadbeef
```

```
.data
```

```
    myarray: .word 1 2
```

```
.text
```

```
li a0, CONSTANT
```

```
# lui a0,0xdead
```

```
# addi a0,a0,0xfffffeef
```

□ 参考资料: RISC-V Assembly Programmer's Manual

<https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md#risc-v-assembly-programmers-manual>

实验原理

5.RARS:RISC-V Assembler & Runtime Simulator

□ 界面介绍

File Edit **Run** Settings Tools Help

Run speed at max (no interaction)

Registers Floating Point Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00002ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00003004

```
1 # This example shows an implementation of the mathematical
2 # factorial function (! function) to find the factorial value of !7 = 5040.
3
4 .data
5 argument: .word 7
6 str1: .string "Factorial value of "
7 str2: .string " is "
8
9 .text
10 main:
11     lw a0, argument # Load argument from static data
12     jal ra, fact    # Jump-and-link to the 'fact' label
13
14     # Print the result to console
15     mv a1, a0
16     lw a0, argument
17     jal ra, printResult
18
19     # Exit program
20     li a7, 10
21     ecall
22
23 fact:
24     addi sp, sp, -16
25     sw ra, 8(sp)
```

Line: 13 Column: 1 Show Line Numbers

Messages Run I/O

实验原理

5.RARS:RISC-V Assembler & Runtime Simulator

□ 界面介绍

G:\software\ASM 汇编\RISC\array generation.asm - RARS 1.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Control and Status

Floating Point

Registers

Name	Nu...	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00002ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00003000

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00003000	0xffffd117	auipc x2,0xffffd117	9: la sp, myArray # 数组的地址
	0x00003004	0x00010113	addi x2,x2,0	
	0x00003008	0x00000313	addi x6,x0,0	10: addi t1, zero, 0 #存放产生的随机数
	0x0000300c	0x00010393	addi x7,x2,0	11: addi t2, sp, 0 # 比较数据时, 数组下标
	0x00003010	0x00000e13	addi x28,x0,0	12: addi t3, zero, 0 #当前比较的两个数之一
	0x00003014	0x00000e93	addi x29,x0,0	13: addi t4, zero, 0 #当前比较的两个数之一
	0x00003018	0x00000f13	addi x30,x0,0	14: addi t5, zero, 0 #比较轮数
	0x0000301c	0x10000f93	addi x31,x0,0x00000100	15: addi t6, zero, 256 #数据个数
	0x00003020	0x00000413	addi x8,x0,0	16: addi s0, zero, 0 #每轮比较的次数
	0x00003024	0x40010493	addi x9,x2,0x00000400	17: addi s1, sp, 1024 #数据空间大小
	0x00003028	0x00010293	addi x5,x2,0	18: addi t0, sp, 0 #数组产生时数据地址
	0x0000302c	0x00010913	addi x18,x2,0	19: addi s2, sp, 0 #排序完成后数据地址
	0x00003030	0x02928e63	beq x5,x9,0x0000003c	25: beq t0, s1, gen_over_print #每个数值4bytes(32bits), 12...
	0x00003034	0x06400593	addi x11,x0,0x00000064	27: li a1, 100
	0x00003038	0x02a00893	addi x17,x0,0x0000002a	28: li a7, 42
	0x0000303c	0x00000073	ecall	29: ecall
	0x00003040	0x00a00333	add x6,x0,x10	30: mv t1, a0

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Val
0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	

0x00001000 (.extern)
0x00000000 (.data)
0x00002000 (heap)
current gp
current sp
0x00003000 (.text)
0x00007f00 (MMIO)
0x00000000 (.data)

Hexadecimal Addresses

实验原理

5.RARS:RISC-V Assembler & Runtime Simulator

□ 存储器配置

✓ Setting >> Memory Configuration

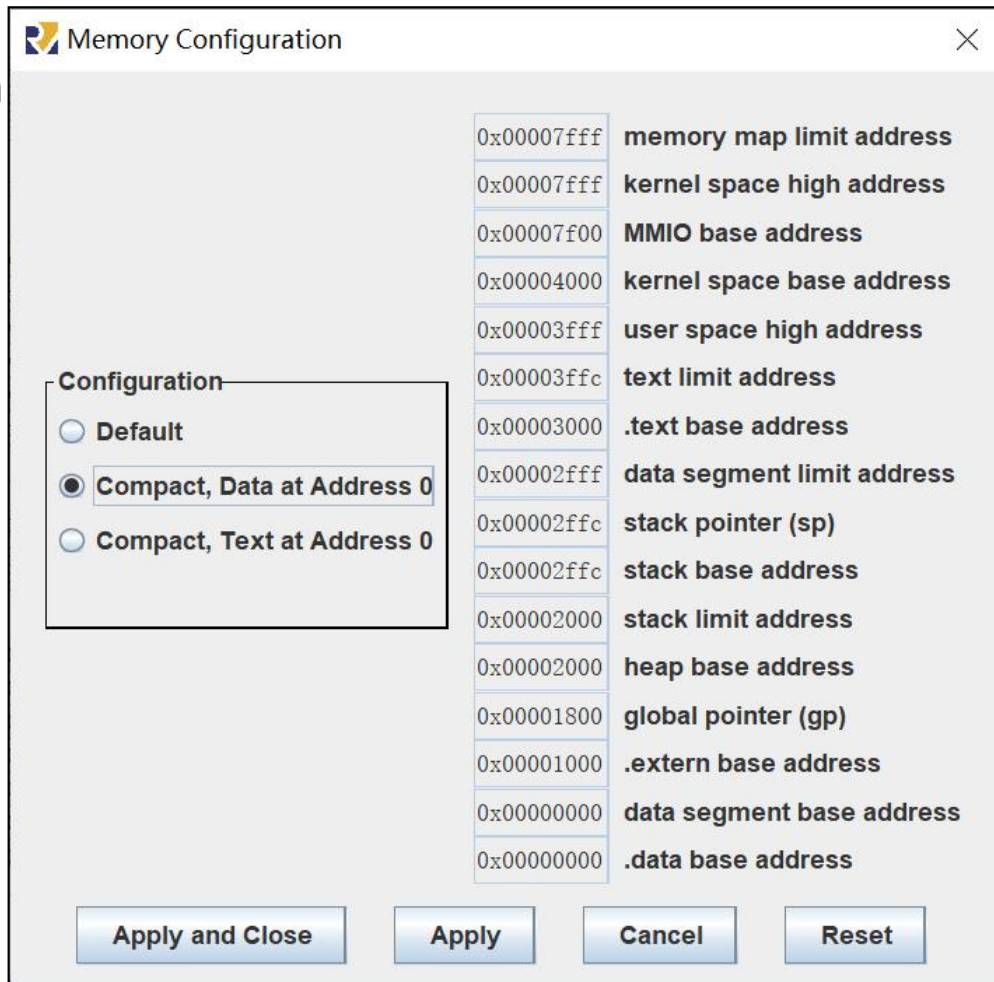
✓ 假定配置为紧凑型

数据地址:

– 0x0000 ~ 0x2fff

代码地址:

– 0x3000 ~ 0x3fff



The image shows a 'Memory Configuration' dialog box with a title bar and a close button. It contains a 'Configuration' section with three radio buttons: 'Default', 'Compact, Data at Address 0' (which is selected), and 'Compact, Text at Address 0'. To the right of the configuration section is a list of memory addresses and their corresponding labels. At the bottom of the dialog are four buttons: 'Apply and Close', 'Apply', 'Cancel', and 'Reset'.

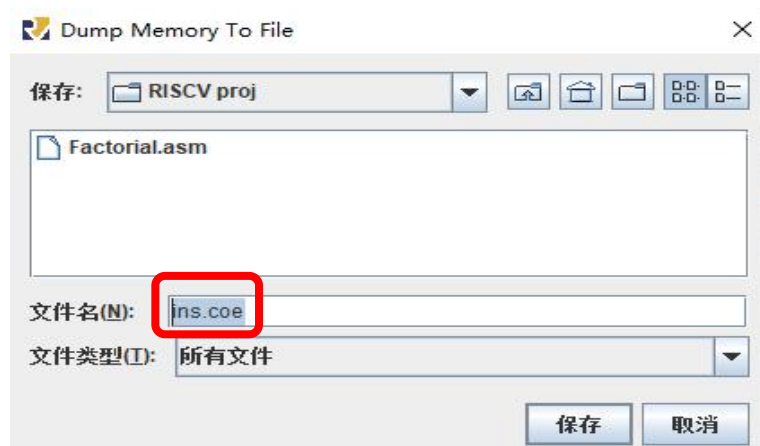
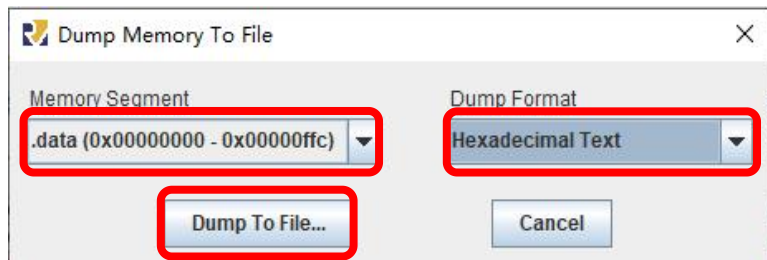
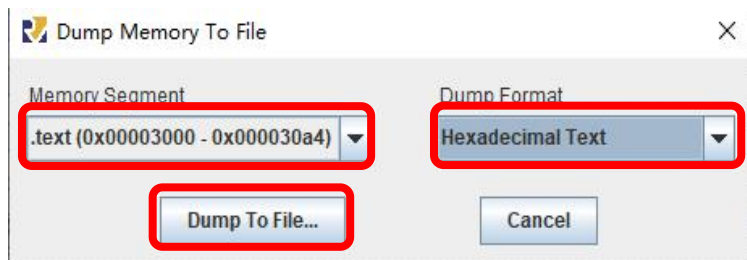
Address	Label
0x00007fff	memory map limit address
0x00007fff	kernel space high address
0x00007f00	MMIO base address
0x00004000	kernel space base address
0x00003fff	user space high address
0x00003ffc	text limit address
0x00003000	.text base address
0x00002fff	data segment limit address
0x00002ffc	stack pointer (sp)
0x00002ffc	stack base address
0x00002000	stack limit address
0x00002000	heap base address
0x00001800	global pointer (gp)
0x00001000	.extern base address
0x00000000	data segment base address
0x00000000	.data base address

实验原理

5.RARS:RISC-V Assembler & Runtime Simulator

□ 汇编程序转COE文件

- ✓ 配置存储器: Setting >> Memory Configuration...
- ✓ 汇编程序: Run >> Assemble
- ✓ 导出代码和数据: File >> Dump Memory...



实验原理

5.RARS:RISC-V Assembler & Runtime Simulator

□ 汇编程序转COE文件

✓ 生成COE文件:

采用记事本分别打开生成的ins.coe和data.coe，在文档的开头添加以下两行语句后保存：

```
memory_initialization_radix = 16;
```

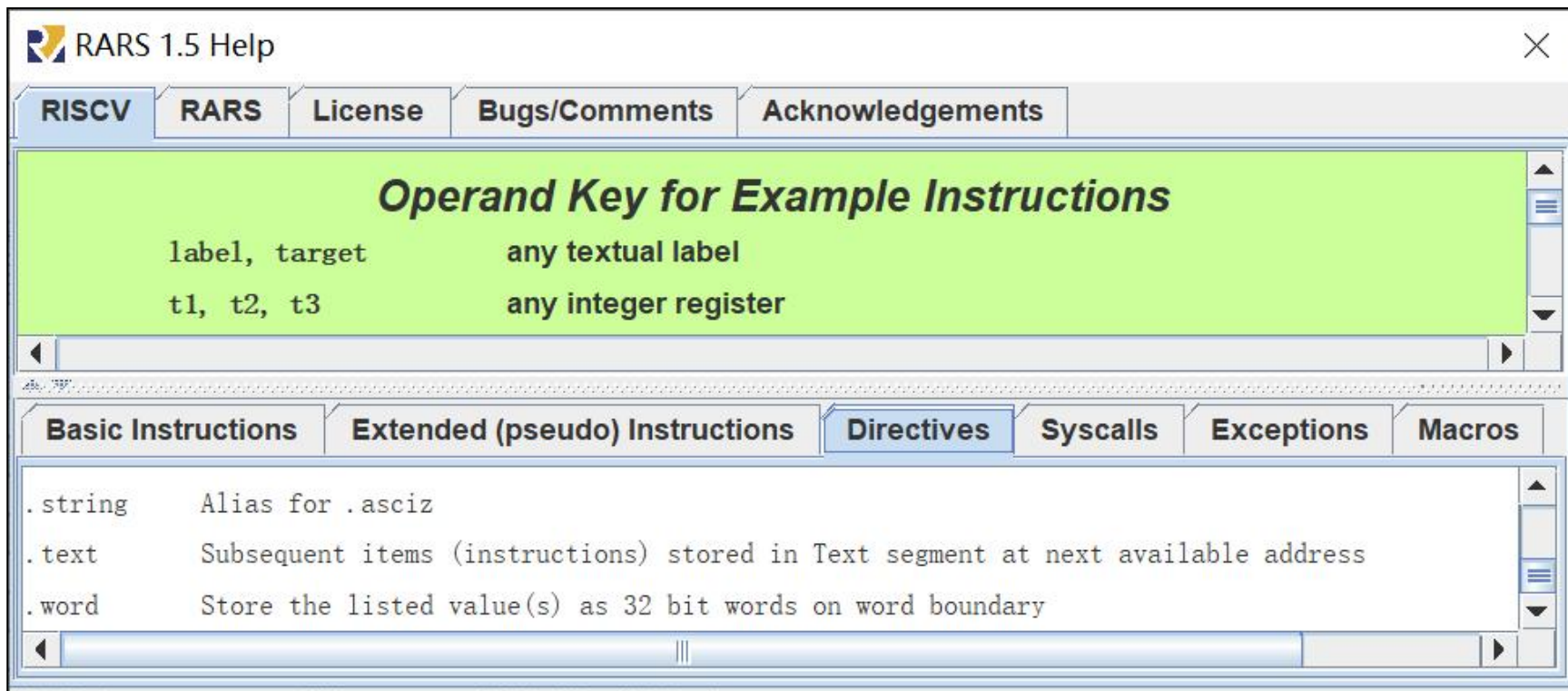
```
memory_initialization_vector =
```

实验原理

5.RARS:RISC-V Assembler & Runtime Simulator

□ help

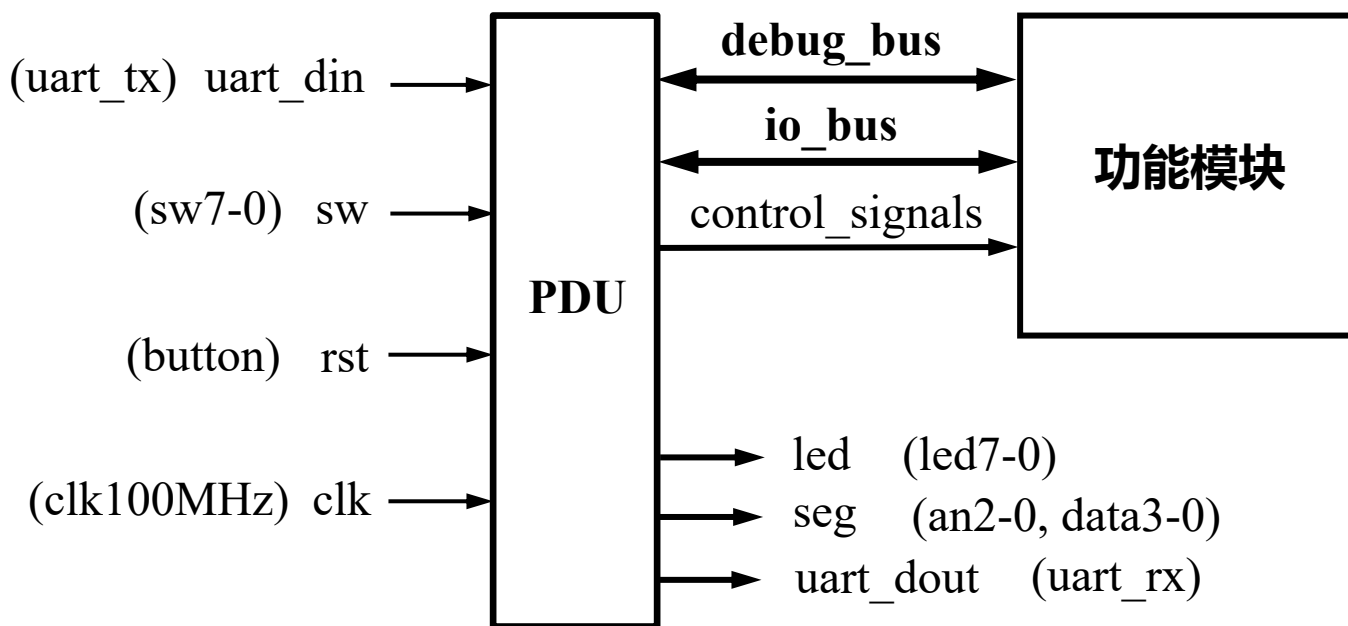
- ✓ RISC-V: 指令、伪指令、指示符、系统调用.....
- ✓ RARS: IDE、调试、工具.....



实验原理

6.PDU:外设与调试单元

□ PDU 逻辑结构:



*功能模块是可更换的，只需要约定好与 PDU 的相关接口即可

实验原理

6.PDU:外设与调试单元

□ PDU 功能

✓ 管理 FPGAOL 平台外设，主要包括:

- 开关：共计 8 个，每个开关的输入相互独立；
- 按钮：共计 1 个，按下后输出高电平，松开后输出低电平；
- 七段数码管：共计 8 个，采用分时复用的方式进行扫描显示；
- UART 串口：支持 FPGA 与用户之间的数据传输。

✓ 支持用户与功能模块之间的信号（调试和I/O）交互

- 功能模块通过PDU将数据输出至七段数码管或LED上进行显示，实现数据输出；
- 用户通过开关输入数据，经PDU传递给功能模块，实现数据输入；
- 用户通过串口输入命令，经PDU传递给功能模块，实现信息交互。

实验内容

1. 设计汇编程序：计算斐波那契-卢卡斯数列，生成 COE 文件

□ 设计要求

- ✓ 数列项数为 n ，存储在寄存器 $t0$ (即 $x5$) 处 ($3 \leq n \leq 40$);
- ✓ 数列的前两项为 1, 1, 将 n 项数列存储在地址为 $0x0000$ 开始的数据段 Data Segment;
- ✓ 汇编程序指令只能从以下10条指令中选用，或选用基于该10条指令的伪指令，eg:la, li;
- add addi lui auipc lw sw beq blt jal jalr

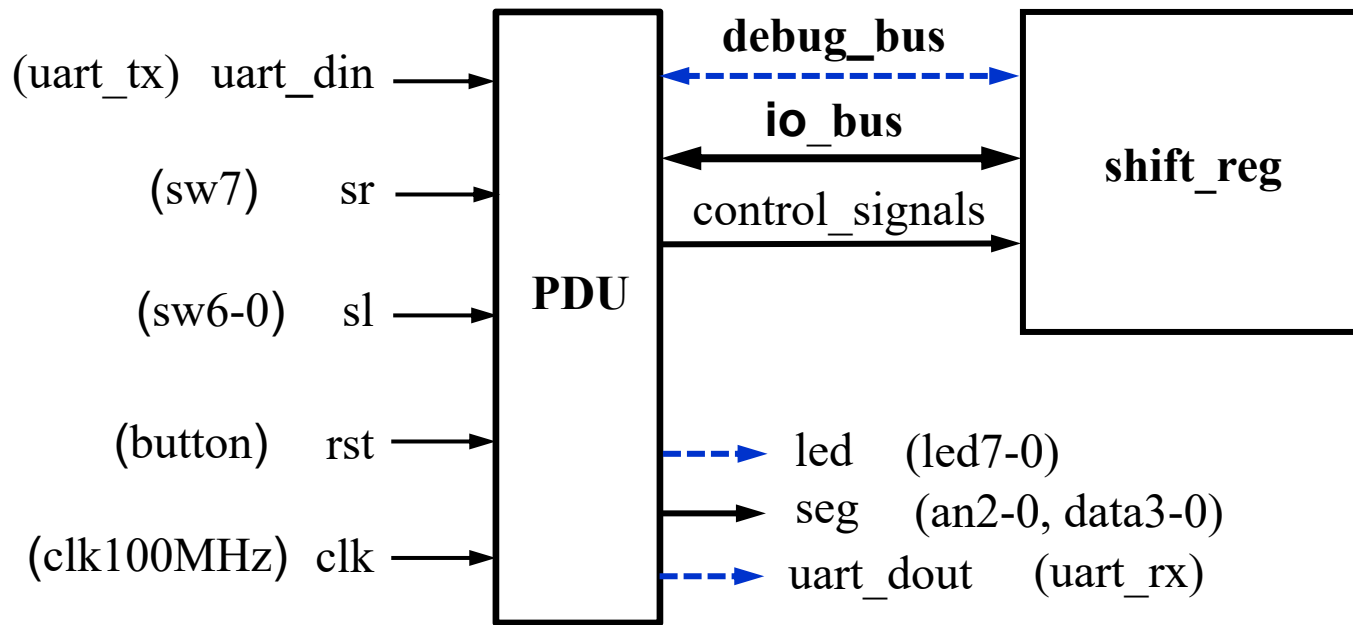
□ COE文件

- ✓ 采用rars软件生成coe文件，用于后续CPU功能测试。

实验内容

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

□ 逻辑结构



实验内容

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

□ 功能说明

✓ 数码管以十六进制形式实时显示当前移位寄存器中的 32bits 数据；

✓ 拨动 **sl[4: 0]** 某一位开关，双边沿有效

移位寄存器执行一次左移4位操作，低4位数据置为被拨动的开关编号，且上下拨动各为一次有效数据输入；

✓ 拨动 **sr**，双边沿有效

移位寄存器执行一次右移4位操作，删除低4位数据，高4位数据置0；

✓ 串口中输入指令：**set + [空格] + [32bit 数据(八位十六进制数)] + [;]**

移位寄存器执行一次置数操作。不足32位的在前面补 0，超出32位的高位截断。例如：set 1a34; 会将移位寄存器中的内容设置为 0x00001a34，寄存器内的原始数据会被丢弃。

实验内容

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

□ 功能说明

- ✓ 串口中输入指令：add + [空格] + [4bit 数据(一位十六进制数)] + [;]

移位寄存器执行一次左移4位操作，例如：add 1; 移位寄存器左移4位，同时低4位插入一个十六进制数 1。

- ✓ 串口中输入指令：del + [;]

移位寄存器执行一次右移4位操作，删除低4位数据，高4位数据置0。

实验内容

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

□ 开关输入规则说明

sw[6:5]	sw[4]	sw[3]	sw[2]	sw[1]	sw[0]
00	4	3	2	1	0
01	9	8	7	6	5
10	e	d	c	b	a
11	0	0	0	0	f

实验内容

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

□ 模块接口

```
module Shift_reg(  
    input rst,          //button  
    input clk,          // 100MHz clk  
    //control_signals  
    input add,          // 左移使能  
    input del,          // 右移使能  
    input set,          // 寄存器初值设定  
    //io_bus  
    input [31:0] din,   // 移位寄存器设定数据  
    input [3:0] hex,    // 开关移位数据  
    output reg [31:0] dout );// 寄存器当前数据
```

实验内容

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

□ 模块接口

```
module PDU(  
    input clk,rst,                // 100MHz clk,button  
    input [7:0] sw,               // sw7-0  
    output [7:0] led,             // led7-0  
    output [2:0] hexplay_an,      // hexplay_an  
    output [3:0] hexplay_data,    // hexplay_data  
    input uart_din,               // uart_tx  
    output uart_dout              // uart_rx, Unused!  
    //io_bus  
    input [31:0] dout             // 移位寄存器当前数据  
    output [31:0] din,            // 移位寄存器设定数据  
    output [3:0] hex,             // 开关移位数据  
    //control_signals  
    output add,                  // 左移使能  
    output del,                  // 右移使能  
    output set );                // 寄存器初值设定 使能
```

实验要求[必做]

1. 设计汇编程序：计算斐波那契-卢卡斯数列，生成 COE 文件

- 根据实验设计要求编写汇编程序；
- 生成COE文件，作为后续CPU测试数据。

2. 设计32bits移位寄存器，通过PDU实现对移位寄存器数据的实时操作

- 设计32bits移位寄存器；
- 将移位寄存器模块添加至工程中综合、实现，生成bit文件；
- 将电路下载至FPGAOL中测试。

实验要求[选做]

1. 设计汇编程序，计算斐波那契-卢卡斯数列

□ 采用外设输入（选做1）

- ✓ 数列的项数 n 由用户通过 Rars 的 Keyboard MMIO Simulator 输入 ($3 \leq n \leq 40$);
 - 使用轮询的方式进行输入查询;
 - n 采用十进制，可能是一位数，也可能是两位数，且为一次性输入。
- ✓ 数列的前两项为 1, 1, 将 n 项数列存储在地址为 0x0000 开始的数据段 Data Segment;
- ✓ 汇编程序采用 RV32I 中的指令实现。

□ 采用外设输出（选做2）

- ✓ 在选做1的基础上，将数列的前 n 项通过 Rars 的 Display MMIO Simulator 输出;
 - 输出时每行输出一个 32bit 整数后换行，数据进制不做要求（建议十六进制）；
 - 需要输出从第 1 项到第 n 项的全部数据；
 - 数列数据可以输出前导零，eg.0x00123456，不输出前导零的实现可以得到额外的分数 eg.0x123456。
- ✓ 汇编程序采用 RV32I 中的指令实现。

实验要求[选做]

1. 设计汇编程序，计算斐波那契-卢卡斯数列

□ 支持大整数存储（选做3）

- ✓ 数列的前两项为 1, 1, 存储在 0x0000 开始的连续地址处;
- ✓ 数列项数 n 的输入方式、结果的输出方式不限;
- ✓ 数列的项数 n 的范围为 $40 < n \leq 80$;
 - 此时数列的结果有可能超出了 32bit 整数的范围，因此需要保证输出结果的正确性（不会溢出）：可以使用 2 个 32bit 寄存器/地址单元保存 64 bit 的数据。
- ✓ 只能采用 RV32I 中的指令实现。

*回车：ASCII 码 13, “\r”

*换行：ASCII 码 10, “\n”

*空格：ASCII 码 32

*相关外设使用规则可查看 Tools >> Keyboard And Display MMIO Simulator 的 help 文档。



中国科学技术大学
University of Science and Technology of China

The End