Computer Organization

Lab5 RISC-V vs MIPS

RISC-V vs MIPS;
RARS(a RISC-V assembler and simulator)

RISC-V vs MIPS（summary)

- Samilarities：

1. RISC
2. Memory accessed only by load/store instructions
3. 32-bit instructions(RISC-V basic instruction)
4. 32 general purpose registers, register 0 is always 0
5. almost same directives (.data, .text, .align, .byte, .ascii,...)

...

- Differences:

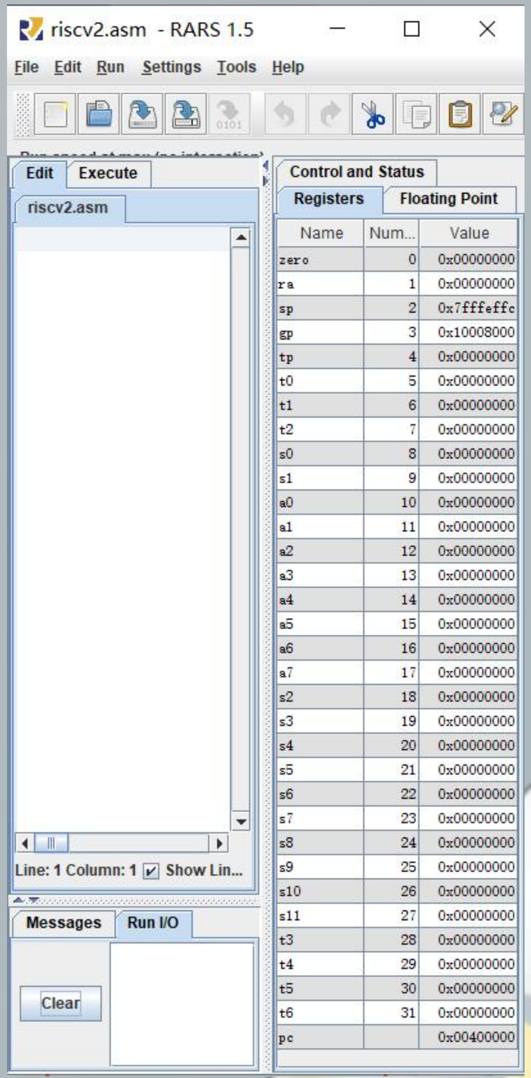
1. Basic instruction set size (MIPS > RISC-V （I）)
2. RISC-V: Modular instruction structure, more flexible expansion
3. Instruction encoding format （location of register id in machine code is fixed in different types of RISC-V)
4. The representation about registers in instructions(prefix “$” , “x”)
5. Specific instructions: lui， addtion, mul, div, nor, the usage of “x0” register in un-conditional jump instrucion ...
6. system service （syscall in MIPS vs ecall in RISC-V)

...

RARS-1 (introduction)

RARS - RISC-V Assembler and Runtime Simulator (Release 1.0)

- **RARS**, **the RISC-V Assembler**, **Simulator**, **and Runtime**, will **assemble** and **simulate the execution of RISC-V assembly language programs**.
- **RARS** is written in Java and **requires at least Release 1.8 of the Java SE Java Runtime Environment (JRE) to work**. It is distributed as an executable JAR file.
- **RARS** supports most of **RV32IMFN** (**base 32 bit instruction set** + **multiplication, floating point**, **and user-level interrupts**).
  - The guiding reference for implementing the instruction set has been **version 2.2 of the official specification**.



Tips: The basic operation of RARS is the same as that of Mars.
Download: https://github.com/TheThirdOne/rars/releases

RARS-2(system service)

A part of avaliable sevices on RARS

Table of Available Services

Name	Number	Description	Inputs	Ouputs
PrintInt	1	Prints an integer	a0 = integer to print	N/A
PrintFloat	2	Prints a floating point number	fa0 = float to print	N/A
PrintDouble	3	Prints a double precision floating point number	fa0 = double to print	N/A
PrintString	4	Prints a null-terminated string to the console	a0 = the address of the string	N/A
ReadInt	5	Reads an int from input console	N/A	a0 = the int
ReadFloat	6	Reads a float from input console	N/A	fa0 = the float
ReadDouble	7	Reads a double from input console	N/A	fa0 = the double
ReadString	8	Reads a string from the console	a0 = address of input buffer a1 = maximum number of characters to read	N/A
Sbrk	9	Allocate heap memory	a0 = amount of memory in bytes	a0 = address to the allocated block
Exit	10	Exits the program with code 0	N/A	N/A
PrintChar	11	Prints an ascii character	a0 = character to print (only lowest byte is considered)	N/A
ReadChar	12	Reads a character from input console	N/A	a0 = the character

```
.data              #RISC-V
str: .asciz "Hello,RISC-V"

.text
li a7,4
la a0,str
ecall

li a7,10
ecall
```

```
.data              #MIPS
str: .asciiz "Hello,MIPS"

.text
li $v0,4
la $a0,str
syscall

li $v0,10
syscall
```

RISC-V(1) data storage(1)

```
.data              #RISC-V
str: .asciz "Hello,RISC-V"

.text
li a7,4
la a0,str
ecall

li a7,10
ecall
```

```
//in Java，  C，  Phython
a = b + 1

# in RISC-V
lw x5, b        #from memory to register

addi x6, x5, 1

sw x6, a        #from register to memory
```

- Data Storage: **instruction**, **register**, **memory**
  - **register**
    - using "name" of the register without "$" as prefix, or using "number" of the register with "x" as prefix
    - regiseter **zero(x0)** is non-writable, the data in it is a 32bit all-zero
    - register **ra(x1)** is used to save the return address
    - register **sp(x2)** is used as the stack pointer
    - register **a7(x17)** is used to save the "ecall" service number of ecall
      - register **a0(x10)** is used as the parameter or return value of the "ecall" service
      - register **a1(x11)** is used as the parameter of the "ecall" service if necessary
    - register **pc** is used as program counter
    - ....

Registers

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
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		0x00400000

RISC-V(1) data storage(2)

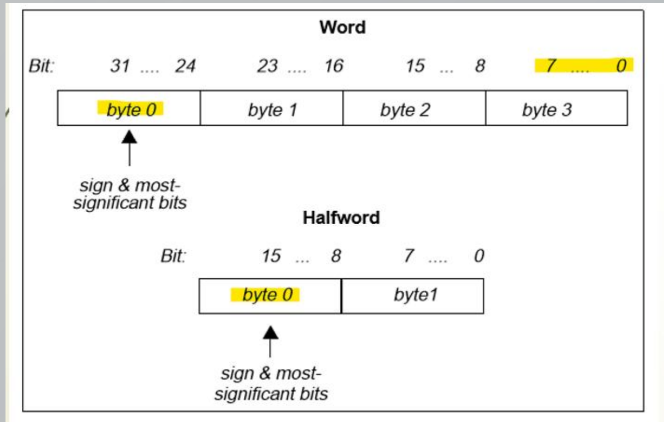


Figure 1-1: Big-endian Byte Ordering

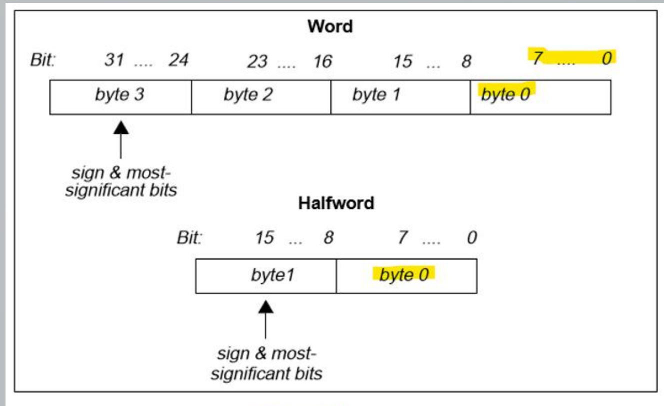


Figure 1-2: Little-endian Byte Ordering

The CPU's **byte ordering scheme** (or **endian issues**) affects memory organization and defines the relationship between address and byte position of data in memory.

- a **Big-endian** system means byte 0 is always the most-significant (leftmost) byte.
- a **Little-endian** system means byte 0 is always **the least-significant (rightmost) byte**.

RISC-V is little-endian!

```
.data                #RISC-V
td: .word 0x12345678
td12: .byte 0x12
td34: .byte 0x34
td56: .byte 0x56
td78: .byte 0x78
```

Labels

Label	Address ▲
big_little_endian.asm	
td	0x10010000
td12	0x10010004
td34	0x10010005
td56	0x10010006
td78	0x10010007

Data Segment

0x10010003 0x10010000

Address	Value (+0)	Value (+4)
0x10010000	0x12345678	0x78563412

Practice1-1

Use **RISC-V** to program and realize the following functions on **RARS**:

- Using 1 syscall to get the sid which has 8 numbers from input
- Using 1 syscall to print out the string:

Welcome XXXXXXXX to RISC-V World

(XXXXXXXX is an 8-digit number)

A demo on MIPS for reference is on the right hand--->

```
.data                                           #MIPS
        s1:     .ascii   "Welcome "
        sid:    .space  9
        e1:     .asciz   "to RISC-V World"
.text
        li $v0,8        #get a string
        la $a0,sid
        li $a1,9
        syscall

        li $t0,32       #replace '\0' with space
        sb $t0,sid+8

        li $v0,4        #print a string
        la $a0,s1
        syscall

        li $v0,10       #exit
        syscall
```

RISC-V(2) data-details(signed vs unsigned)

Tips: While the instruction ends with "**u**", it means the data are treated as **unsigned** integer, else the data are treated as **signed by defalut**. The rule in RISC-V is same as in MIPS.

```
.data                       #RISC-V
    tdata: .byte 0x80
.text
    lb a0,tdata
    li a7,1
    ecall

    lb a0,tdata
    li a7,36
    ecall

    li a7,10
    ecall
```

```
.data                       #RISC-V
    tdata: .byte 0x80
.text
    lbu a0,tdata
    li a7,1
    ecall

    lbu a0,tdata
    li a7,36
    ecall

    li a7,10
    ecall
```

Tips: syscall

1) code in **a7** : **1**
Display data in **a0** as **signed** decimal value

2)code in **a7** : **36**
Display data in **a0** as **unsigned** decimal value

Q1: Run the two demos, what's the value stored in the register a0 after the operation of 'lb' and 'lbu'

Q2: using "-1" as initial value of tdata instead of "0x80", answer Q1 again.

Practice 1-2

Use RISC-V to program and realize the following functions on RARS:

1) The data in a word is 0x12345678, print it in hexdecimal, then exchange the bytes of this word to get the new value 0x78563412 and print the updated data in hexdecimal.

2) If the data in the word is 0x8192a3b4, to exchange the bytes of this word to get the new value 0xb4a39281 and print, whether the instructions of text segment needs to be modified ?

The demo on MIPS for reference is on the right hand--->

```
.data                                    #MIPS
        di:     .word   0x12345678

.text
main:
        la $t0,di
        lw $a0,($t0)
        li $v0,34
        syscall

        lb $t1,($t0)
        lb $t2,1($t0)
        lb $t3,2($t0)
        lb $t4,3($t0)

        sb $t1,3($t0)
        sb $t2,2($t0)
        sb $t3,1($t0)
        sb $t4,($t0)

    add $a0,$t0,$0
    syscall

     li $v0,10    #to exit
     syscall
```

RISC-V(3) calculation(1)

```
#MIPS
.data
        tdata: .word 0x71111111
.text
main:
        lw $t0,tdata
        addu $a0,$t0,$t0
        li $v0,1
        syscall

        add $a0,$t0,$t0
        li $v0,1
        syscall

        li $v0,10
        syscall
```

```
#RISC-V
.data
        tdata: .word 0x71111111
.text
main:
        lw t0,tdata
        addu a0,t0,t0
        li a7, 1
        ecall

        add a0,t0,t0
        li a7,1
        ecall

        li a7,10
        ecall
```

Compare the difference on **addition** between **MIPS** and **RISC-V**:

Q1: Which demo(s) would invoke assemly fail ? *the demo in MIPS or the demo in RISC-V?*

Q2: Which demo(s) would invoke an exception *(arithmetic overflow)* , *the demo in MIPS or the demo in RISC-V?*

Q3: Which instruction would invoke the exception? *lw, add or addu?*

RISC-V(3) calculation(2)

The multiply and divide instruction is an extension instruction of RISC-V.

- There are no 'hi' and 'lo' register, no move instrction between hi/lo and normal register in RISC-V.
- 'mulh' could be used to get the higer 32bit of product, 'rem' could be used to get he remiander of the division.

```
.data        #MIPS
da: .word 0xF000F0F0
db: .word 0xF0F0FF00

.text
lw $t0, da
lw $t1,db
mul $s0,$t0,$t1

mflo $s0  #optional
mfhi $s1

li $v0,10
syscall
```

```
.data            #RISC-V
da: .word 0xF000F0F0
db: .word 0xF0F0FF00

.text
lw t0, da
lw t1,db

mul s0,t0,t1
mulh s1,t0,t1

li a7,10
ecall
```

```
.data            #MIPS
da: .word 16
db: .word 5

.text
lw $t0, da
lw $t1,db
div  $t0,$t1

mflo $s0
mfhi $s1

li $v0,10
syscall
```

```
.data            #RISC-V
da: .word 16
db: .word 5

.text
lw t0, da
lw t1,db

div s0,t0,t1
rem s1,t0,t1

li a7,10
ecall
```

RISC-V(4) Conditional Jump

In **RISC-V**,
all the conditional jump instructions are basic type.

Basic	Source
beq x6, x5, 0x00000000	10: beq t1, t0, main
bne x6, x5, 0xfffffffc	11: bne t1, t0, main
blt x5, x6, 0xfffffff8	12: bgt t1, t0, main
bltu x5, x6, 0xfffffff4	13: bgtu t1, t0, main
bge x6, x5, 0xfffffff0	14: bge t1, t0, main
bgeu x6, x5, 0xffffffec	15: bgeu t1, t0, main
blt x6, x5, 0xffffffe8	16: blt t1, t0, main
bltu x6, x5, 0xffffffe4	17: bltu t1, t0, main
bge x5, x6, 0xffffffe0	18: ble t1, t0, main
bgeu x5, x6, 0xffffffdc	19: bleu t1, t0, main

vs

In **MIPS, ONLY “beq” and “bne” are basic conditional jump instruction,** other branch instrucions are implemented by the “set” and the basic branch instruction.

Basic	Source
beq $9, $8, 0xffffffff	3: beq $t1, $t0, main
bne $9, $8, 0xfffffffe	4: bne $t1, $t0, main
slt $1, $8, $9	5: bgt $t1, $t0, main
bne $1, $0, 0xfffffffc	
sltu $1, $8, $9	6: bgtu $t1, $t0, main
bne $1, $0, 0xfffffffa	
slt $1, $9, $8	7: bge $t1, $t0, main
beq $1, $0, 0xfffffff8	
sltu $1, $9, $8	8: bgeu $t1, $t0, main
beq $1, $0, 0xfffffff6	
slt $1, $9, $8	9: blt $t1, $t0, main
bne $1, $0, 0xfffffff4	
sltu $1, $9, $8	10: bltu $t1, $t0, main
bne $1, $0, 0xfffffff2	
slt $1, $8, $9	11: ble $t1, $t0, main
beq $1, $0, 0xfffffff0	
sltu $1, $8, $9	12: bleu $t1, $t0, main
beq $1, $0, 0xffffffee	

RISC-V(5) UnConditional Jump(1)

In RISC-V, the basic unconditional jump instructions are : jal, jalr

jal t1, target	Jump and link : Set t1 to Program Counter (return address) then jump to statement at target address
jal label	Jump And Link: Jump to statement at label and set the return address to ra

jalr t1, t2, -100	Jump and link register: Set t1 to Program Counter (return address) then jump to statement at t2 + immediate
jalr t0	Jump And Link Register: Jump to address in t0 and set the return address to ra
jalr t0, -100	Jump And Link Register: Jump to address in t0 and set the return address to ra
jalr t0,-100(t1)	Jump And Link Register: Jump to address in t1 and set the return address to t0

Basic	Source	
jal x0,0xfffffff4	36:	j print_string
jal x0,0xfffffff0	37:	jal x0,print_string
jalr x0,x1,0	39:	jr x1
jalr x0,x1,0	40:	jalr x0,x1,0
jalr x0,x1,0	41:	jalr x0,0(x1)

RARS implement two pesudo instructions : j and jr

- **j** is implement by basic instrutcion: **jal**
  - **x0(zero)** is non-writable register, the data in it is 32bit all-zero.
- **jr** is implement by basic instruction: **jalr**
  - **x1(ra)** is used as return address by default

RISC-V(5) UnConditional Jump(2)

```
.include "macro_print_str.asm"      #MIPS
.text
        print_string("please input your score (0~100):")
        li $v0,5
        syscall
        move $t0,$v0
        bge $t0,60,passLable
        j failLable

passLable:
        print_string("\nPASS (exceed or equal 60) ")
        j caseEnd
failLable:
        print_string("\nFaild(less than 60)")
caseEnd:
        end
```

```
 .data     #RISC-V  part1
        dpass: .byte 60
        str1: .asciz "please input your score (0~100):"
        strp: .asciz "\nPASS (exceed or equal 60) "
        strf: .asciz "\nFaild(less than 60)"
.text
        la a0,str1
        ____  print_string
        li a7,5
        ecall
        mv t0,a0

        lb a1,dpass
        bge t0,a1,passLable
        ___   failLable
```

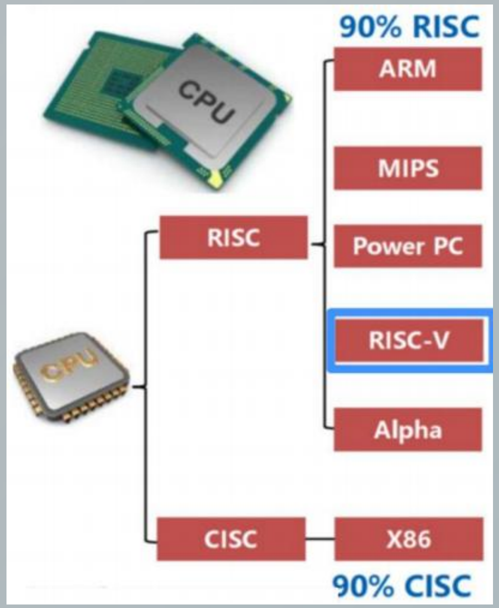
```
passLable:        #RISC-V  part2
        la a0,strp
        ___  print_string
        ___  caseEnd
failLable:
        la a0,strf
        ___  print_string
caseEnd:
        li a7,10
        ecall

print_string:
        li a7,4
        ecall
        __________
```

Practice 1-3: Please supplement the underlined code in RISC-V part1 and part2 **with basic unconditional instruction in RISC-V** to make the demo in RISC-V working same **as** the demo in **MIPS**.

Practice2

- 2-1 Implement the following function in RISC-V:
  - 1) get a string which is 1 character or 2 characters from input
  - 2) print out the string based on the input string:
    - 2-1) if the input string is 1 character and the character is ‘C’, print out string: “CISC-X86”
    - 2-2) if the input string is 2 characters, and the 1st character is ‘R’, then go to a ‘switch-case’ based on the 2nd character
      - if the 2nd character is 0, print out string: “RISC ARM”
      - if the 2nd character is 1, print out string: “RISC MIPS”
      - if the 2nd character is 2, print out string: “RISC Power PC”
      - if the 2nd character is 3, print out string: “RISC RISCV”
      - if the 2nd character is 4, print out string: “RISC Alpha”
    - 2-3) in other situation, print out string “invalid input,exit”



- 2-2 Find more details about RISC-V :
  - How many bits are load to the destination register by lui in RISC-V? is it same as in MIPS ?
  - Is there “nor” instruction in RISC-V, if not, how to use basic instruction to implement the “nor” function ?
  - How to use stack and heap space in RISC-V ?

MIPS & RISC-V Instructions

Register-register

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 11	10 6	5 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)

Load

	31 20	19 15	14 12	11 7	6 0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Store

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Branch

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)	Const(16)

Tips： macro_print_str.asm

```
.macro print_string(%str)   #MIPS
	.data
		pstr:  .asciiz  %str
	.text
		la $a0,pstr
		li $v0,4
		syscall
.end_macro

.macro end
	li $v0,10
	syscall
.end_macro
```

Get help of defination and usage about macro from Mars‘ help page.

While using the macro, put this file to the same directory as the file which use the macro.