Report

☑ to do ☐

1. Memory and Register Simulation

1.1. Memory Simulation

💡 Big-Endian VS Little-Endian

- Big-Endian (for .ascii, .asciiz)

  Eg. str = '00100000000000100000000000000101'

  - -> str[24:32] str[16:24] str[8:16] str[0:8] -->.memory[idx] memory[idx+1] memory[idx+2] memory[idx+3]

  Eg2. str = 0x1234abcd

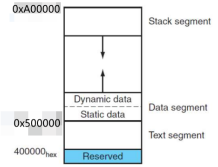
  —> memory 0xcd 0xab 0x34 0x12

- Little-Endian(for .word .half,.byte)

  Eg1. str = 0x1234abcd

  - —> memory 0x12 0x34 0xab 0xcd

1.1.1. Initialize data



In general, this part will put the data in .data segment of MIPS file piece by piece in the static data segment. To be more precise, data will be obtained after reading .asm file, intercepting .data part, deleting comment, etc. To simplify the problem, only .ascii and .asciiz use Big-Endian, and .word .half .byte use the Little-endian for storage in memory. And the *start index of data* is 500000hex (real memory address), with simulated address 100000hex.

1.1.2. Initialize text

In general, this part will put the assembled machine code in the text segment of your simulated memory. To be more precise, assembled machine code can be obtained by the result in project 1 (also provided in ".txt" file). The assembled machine code is 32 bits, which is 4 bytes. Thus, each byte will be translate to a decimal number and stored as an integer. Each 32-bit code will be store in 4byte memory using Little-Endian. And the *start index of text* is 400000hex (real memory address), with simulated address 000000hex.

```
##IMPLEMENTATION:
data_parse.py
  set_data
  """
  func  = set_data
  usage = read .asm file, intercept .data part, delete comment
  sample_input: 'a-plus-b.asm'
  sample_output: ['num: .word 13']
  """
MIPS_simulator.py
  load_data
  """
  func = load_data
  usage = load .data, store in memory(static data segment)
  sample_input: ['num: .word 13']
  saple_output: memory[0x100000] = 13
  """
  load_text
  """
  func = load_text
  usage = load .text, store in memory(text segment)
  sample_input: ['00111100000000010000000001010000']
  saple_output: memory[0x000000] = 80
                memory[0x000001] = 0
                memory[0x000002] = 1
                memory[0x000003] = 60

  """
```

1.2. Register Simulation

```
##IMPLEMENTATION:
## In total: 32 general registers + pc register + hi register + lo register
self.registers = [0]*32   ## all registers store 32-bit value
self.pc = 0x400000
self.hi = 0
self.lo = 0
self.registers[28] = 0x508000  ## $gp = 0x508000
self.registers[29] = 0xA00000  ## $sp = 0xA00000
self.registers[30] = 0xA00000  ## $fp = 0xA00000
```

2. Execution

2.1. Analysis Instructions

imm(16 bit)

zero_extended(imm) (32 bit)

If immediate value ≥ 0 :

zero_extended(imm) will not change the value

else if immediate value < 0:

zero_extended(imm) will change from negative to positive, and the absolute value will also change. zero_extended(imm) = imm + (2**32)

0x82	1000 0010	130	−126
0x81	1000 0001	129	−127
0x80	1000 0000	128	−128

2.2. Instructions function definition

2.2.1. Logic Instructions (7)

R-type:

and	rd ← AND(rs,rt)
nor	rd <- NOR(rs,rt)
or	rd <- OR(rs,rt)
xor	rd <- XOR(rs,rt)

I-type:

andi	rt ← AND(rs, zero-extended(imm))
ori	rt ← OR(rs, zero-extended(imm))
xori	rt ← XOR(rs, zero-extended(imm))

2.2.2. Arithmetic Instructions (10)

R-type:

		overflow check
add	rd ← rs+rt	☑
addu	rd ← rs+rt	
sub	rd ← rs-rt	☑
subu	rd ← rs-rt	

		operand
mult	{hi，lo} ← rs×rt	signed
multu	{hi，lo} ← rs×rt	unsigned
div	{hi，lo} ← rs/rt	signed
divu	{hi，lo} ← rs/rt	unsigned

I-type

		overflow check
addi	rt ← rs+（sign_extended）immediate	☑
addiu	rt ← rs+（sign_extended）immediate	

- overflow checking:

temp_value = rs (+-) rt

Since add/addu/sub/subu all has signed operand, the temp_value only have 31bit to store its actual value. That is, the temp_value should be in

$$[-2^{31},\ 2^{31}-1]$$

If temp_value is not in the above range, the error of overflow is called, and rd will not store the temp_value.

- operand: signed VS unsigned

  Note that rs, rt store the integer number. In the calculation is exactly the same as signed value.

  However, if we are regaoverflow checkrding the operand as unsigned, which should be positive all the time, things changed. In this case, we should changed to unsigned value first.

  The process from signed value to unsigned value is exactly similar to zero extension of immediate value.

  Eg. if you look at 1111 as unsigned, then the value should be 15, which can be calculated as (2**4 + (-1))

2.2.3. Comparison Instruction (4)

R-type		operand
slt	rd ← （rs<rt）	signed
sltu	rd ← （rs<rt）	unsigned

I-type		operand
slti	rt ← （rs <（sign_extended）immediate）	signed
sltiu	rt ← （rs <（sign_extended）immediate）	unsigned

2.2.4. Jump Instruction (4)

R-type	
jalr	rd(default $31) ← address in rs + 8 pc ← rs
jr	pc ← rs

J-type	
j	pc ← （pc+4）[31，28]|| target || ‘00'
jal	pc ← （pc+4）[31，28]|| target ||‘00’ $31 ← address in rs + 8

2.2.5. Branch Instruction (6)

Immediate value is the offset in brach instruction.

Branch address = （signed_extend）(offset ||“00”）+（pc+4）

I-type	
beq	if rs=rt then branch
bgez	if rs≥0 then branch
bltz	if rs<0 then branch
bgtz	if rs>0 then branch
blez	if rs≤0 then branch
bne	if rs≠rt then branch

2.2.6. Movement Instructions (4)

R-type	
mfhi	rd ← hi
mflo	rd ← lo
mthi	hi ← rs
mtlo	lo ← rs

2.2.7. Load Instructions (8)

In load instructions, immediate value represents offset, rs represents base.

load_address = signed_extended（offset）+ register[base]

n = load_address % 4

I-type		address align requirement
lb	rt ← (signed_extended) byte(8-bit) from load_address	
lbu	rt ← (unsigned_extended) byte(8-bit) from load_address	
lh	rt ← (signed_extended) halfword (16-bit) from load_address	load_address %2 = 0
lhu	rt ← (unsigned_extended) halfword (16-bit) from load_address	load_address %2 = 0
lw	rt ← word (32-bit) from load_address	load_address %4 = 0

lwl	rt ← byte(l_a) || byte(l_a +1) || .. || # byte = 4-n	not required
lwr	rt ← || .. || byte(l_a) || byte(l_a-1) || # byte = n+1	not required
lui	rt ← imm(16) || 0..0 (16)	

图 9-7 lwl 指令作用举例

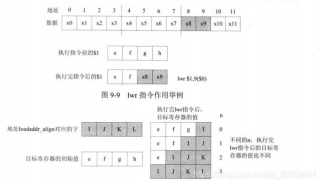


图 9-9 lwr 指令作用举例

2.2.8. Store Instructions (5)

In store instructions, immediate value represents offset, rs represents base.

store_address = signed_extended（offset) + registers[base]

I-type		address align requirement
sb	store_address ← lowest byte rt	
sh	store_address ← lowest 2 bytes rt	load_address %2 = 0
sw	store_address ← rt	load_address %4 = 0
swl	store_address ←	
swr	rt ← byte(l_a) || byte(l_a +1) || .. || # byte = 4-n	

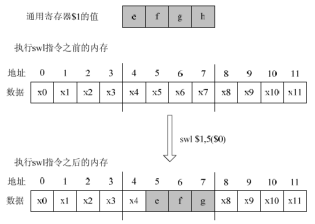


图 9-13 swl 指令作用举例

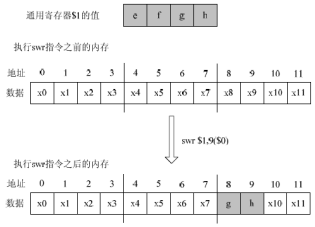


图 9-15 swr 指令作用举例

2.2.9. Shift Instructions (6)

R-type	
sll	rd ← rt << sa（logic） logic: extend with 0
sllv	rd ← rt << rs[4：0]（logic）
sra	rd ← rt >> sa（arithmetic) arithmetic: extend with $31
srav	rd ← rt >> rs[4：0]（arithmetic）
srl	rd ← rt >> sa（logic）
srlv	rd ← rt >> rs[4：0]（logic）

2.2.10. Syscall Instructions

syscall_code is stored in register $v0.

1. For sys calls 1,4,11: print the argument in the .out file one line at a time

		arguments
1	print_int	$a0 = int
4	print_string	$a0 = str
11	print_char	$a0 = char

2. For sys calls 5,8,12: read .in file one line at a time

		arguments	result
5	read_int		$v0 ← int(in)
8	read_string	$a0 = buffer $a1 = length	
12	read_char		$v0 ← char(in)

3. For sys calls 10,13,14,15.16,17: Simulate by directly invoking os

		arguments	result
10	exit		
13	open	$a0 = filename, $a1 = flags, $a2 = mode	$a0 ← file descriptor
14	read	$a0 = file descriptor, $a1 = buffer, $a2 = length	$a0 ← num chars read
15	write	$a0 = file descriptor, $a1 = buffer, $a2 = length	$a0 ← num chars written
16	close	$a0 = file descriptor	
17	exit2	$a0 = result	

3. For sys call 9: simulate the program break in your allocated 6MB memory, and return a pointer to the location in dynamic data so that we can put stuff in.

		arguments	result
9	sbrk	$a0 = amount	$v0 ← address

3. Testing

```
## Test1. a-plus-b
.data
num: .word 13             ## mem[0x1000000] = 13

.text
addi $v0, $zero, 5        ## reg[$v0] = reg[0] + 5
syscall                   ## syscall 5(read_int())
                          ##    in_1 = int(input), reg[$v0] = in_1
add $t0, $zero, $v0       ## reg[$t0] = reg[0] + in_1
addi $v0, $zero, 5        ## reg[$v0] = reg[0] + 5
syscall                   ## syscall 5(read_int())
                          ##    in_2 = int(input), reg[$v0] = in_2
add $t1, $zero, $v0       ## reg[$t1] = reg[0] + in_2
add $a0, $t1, $t0         ## reg[$a0] = reg[$t0]+reg[$t1]
                          ##            = (in_1)+ (in_2)
addi $v0, $zero, 1        ## reg[$v0] = reg[0] + 1
syscall                   ## syscall 1 (print_int)
                          ##    print reg[$a0]
addi $v0, $zero, 10       ## reg[$v0] = reg[0] + 10
syscall                   ## syscall 10 (exit)
```

```
## Test2. fib
.data
# .align 2
FIB_START: .asciiz "fib(" ## mem[0x500000] = ...
# .align 2
FIB_MID: .asciiz ") = "
# .align 2
LINE_END: .asciiz "\n"

.text
addi $v0, $zero, 5         ## reg[$v0] = reg[0] + 5
syscall                    ## syscall 5: read_int reg[$v0] = int1(10)
add $s1, $zero, $v0        ## reg[$s1] = reg[0] + int1(10)
                           ##   reg[$s1]= 10
lui $at, 80                ## reg[$at] = 80 || 0..0(16) = 5242880 = 0x500000
ori $a0, $at, 0            ## reg[$a0] = 80 || 0..0(16)
addi $v0, $zero, 4         ## reg[$v0] = reg[0]+4
syscall                    ## syscall 4 printstring:  "fib("
addu $a0, $s1, $zero       ## reg[$a0] = (10)
addi $v0, $zero, 1         ## reg[$v0] = 1
```



```
syscall                    ## syscall 1 printint: "10"

lui $at, 80                ## reg[$at] = 80 || 0..0(16) = 5242880 = 0x500000
ori $a0, $at, 8            ## reg[$a0] = 80 || 8 = 0x500008
addi $v0, $zero, 4.        ## reg[$v0] = reg[0] + 4
syscall                    ## syscall 4 printstring: ") ="

add $a0, $zero, $s1.       ## reg[$a0] = reg[0] + reg[$s1] = 10
jal fibonacci.             ## fibonacci
add $a0, $zero, $v0
addi $v0, $zero, 1
syscall

lui $at, 80
ori $a0, $at, 16
addi $v0, $zero, 4
syscall

addi $v0, $zero, 10
syscall



fibonacci:
addi $sp, $sp, -12 # 26     ## reg[$sp] = reg[$sp] - 12 = 0xA00000 -12
sw $ra, 8($sp)              ## store word: mem[0xA0000 -4] = reg[$ra]
sw $s0, 4($sp)              ## store word: mem[0xA0000 -8] = reg[$s0]
sw $s1, 0($sp)              ## store word: mem[0xA0000 -12] = reg[$s1] = 10
add $s0, $a0, $zero.        ## reg[$s0] = reg[$a0] + reg[0] = 10
addi $v0, $zero, 1.         ## reg[$v0] = reg[0] + 1 = 1
slti $t7, $s0, 3.           ## reg[$t7] 0/1
bne $t7, $zero, fibonacciExit # 33  ##reg[$t7] = 0: exit
addi $a0, $s0, -1           ## reg[$a0] -= 1
jal fibonacci # 35
add $s1, $zero, $v0         ## reg[s1] = reg[0] + reg[$v0]
addi $a0, $s0, -2           ## reg[$a1] = reg[$s0] -2 = 8
jal fibonacci
add $v0, $s1, $v0
fibonacciExit:
lw $ra, 8($sp)
lw $s0, 4($sp)
lw $s1, 0($sp)
addi $sp, $sp, 12
jr $ra
```