Lab Report4

1. Program1

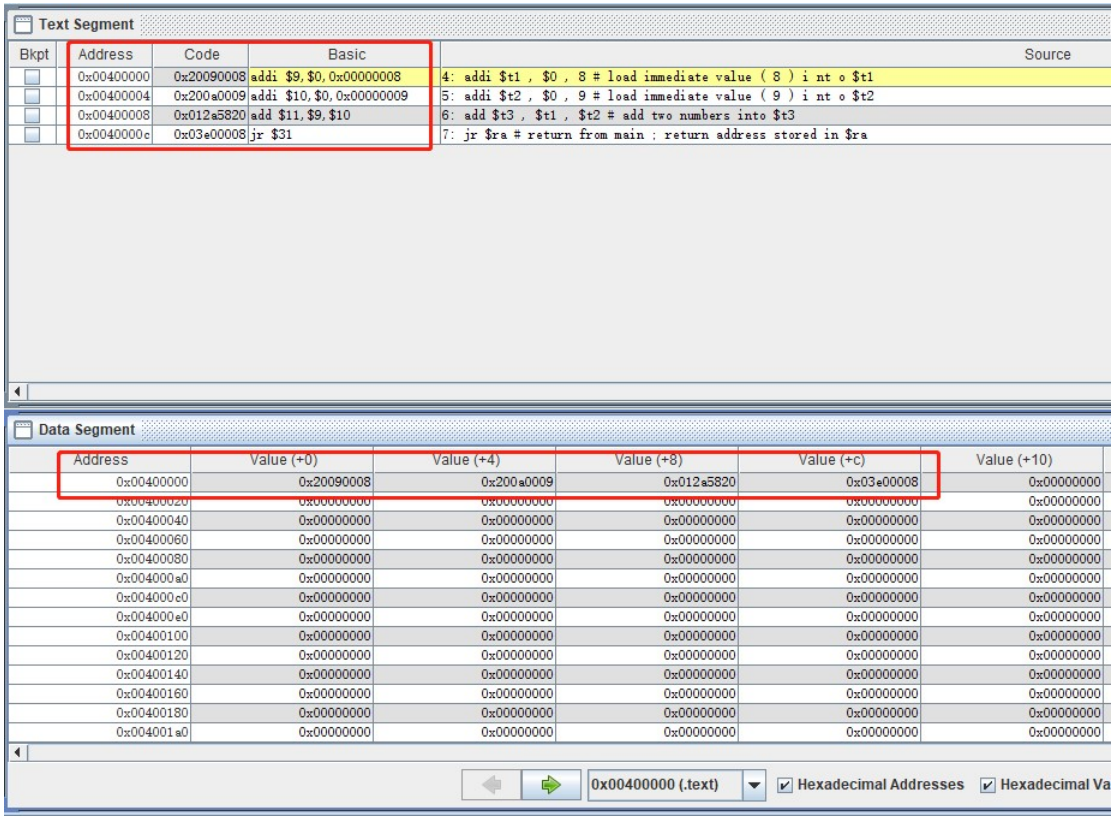

Fig 1.1 program1 MIPS instructions and machine codes.

What this program does is that, store 8 in register t1, store 9 in register t2. Add them up and store the result in register t3 then return. We can see from the figure above that each instruction is translated into a basic MIPS instruction first then translated into machine codes. Codes are stored in text segments whose address starts from 0x00400000 in this case.

2. Program2

What this program does is that, receive a input integer by system call. And do 2bit left shift and 2bit right shift. Then output these 2 integers.

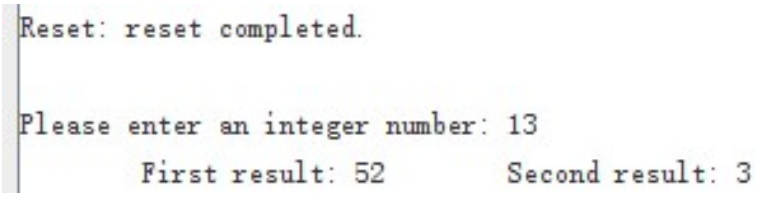

Fig 2.1 RUN I/O output

About text segment and data segment: translated machine codes are stored in text segments, and static variables are stored in data segments. In this case, the three strings are stored in data segments in ASCII codes.

Fig 2.2 string and ASCII codes

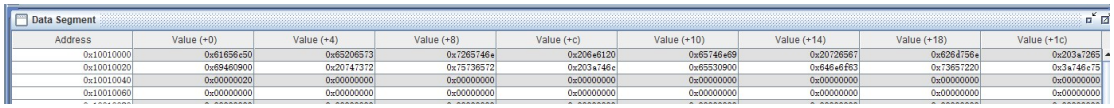

Fig 2.3 data segments

String ASCII codes corresponds with data segments after big endian and little endian transform.

About what does each register do specifically:
v0: used for system call command index(4 for output string, 5 for input int, 1 for output int).
a0: used for system call value.
t0: used for store input number;
t1: used for store integer after left shift
t2: used for store integer after right shift

3. Program3

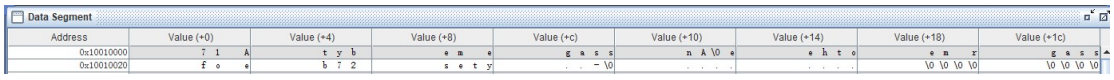

Fig 3.1 data segment(ASCII)

In data segment, ASCII codes of 2 strings are stored. The starting addresses are:
0x10010000(msg1) and 0x10010012(msg2)

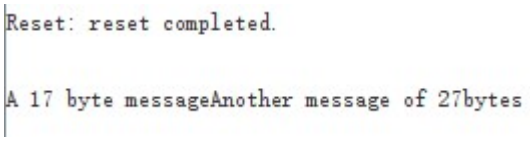

Fig 3.2 RUN I/O outputs

4. Program4

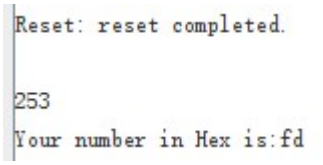

Fig 4.1 RUN I/O outputs

What this program does is that, by separating input integer into high 4 bits and low 4 bits. And adding this index to starting address of hextable, then return the corresponding char. And combining them to the final hex number.

5. Program5

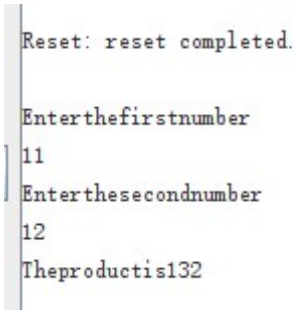

Fig 5.1 RUN I/O outputs

After setting Run speed bar to 2 inst/sec, the running speed is slowed so we can see how registers change in time.

This program implements a simple but inefficient multiplier. The way is by adding repeatedly one input integer onto another. Repeating time is the second input integer. What can be optimized is that by using shift and add method, the time complexity will be much better from O(N) to O(logN).

New code:

```
.data
msg1:      .asciiz "Enter the first number \n"
msg2:      .asciiz"Enter the second number \n"
msg:.asciiz"The productis"
.text
.globl main
.globl my_mul
main:
addi $sp, $sp, -8 #make room for $raand $fp on the stack
sw $ra, 4($sp)#push $ra
sw $fp, 0($sp)#push $fp
la $a0, msg1    #load address of msg1 into $a0
li $v0, 4
syscall#print msg1
```

```
li $v0, 5
syscall#read int
add $t0, $v0, $0 #putin $t0
la $a0, msg2 #load address of msg2 into $a0
li $v0, 4
syscall#print msg2
li $v0, 5
syscall#read int
add $a1, $v0, $0#put in $a1
add $a0, $t0, $0#put first number in $a0
add $fp, $sp, $0#set fp to top of stack prior
# to function call
jal my_mul #do mul,result is in $v0
add $t0, $v0, $0#save the result in $t0
la $a0, msg
li $v0, 4
syscall#print msg
add $a0, $t0, $0#put computation result in $a0
li $v0, 1
syscall#print result number
lw $fp, 0($sp)#restore (pop) $fp
lw $ra, 4($sp)#restore (pop) $ra
addi $sp, $sp, 8#adjust $sp
jr $ra#return
my_mul:
#push s0, s1, s2
addi $sp, $sp, -12
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
#set s0 and s1 as 0
add $s0, $a0, $0
add $s1, $a1, $0
#set result v0 as zero
add $v0, $0, $0
mult_loop:
#exit condition: a == 0
beqz $s0, mult_eol
#if(s0 & 1 == 1) v0 += s1
#else continue
andi $s2, $s0, 1
beqz $s2, continue
add $v0, $v0, $s1
continue:
```

```
#each time left shift s1 and right shift s0 1 bit
sll $s1, $s1, 1
srl $s0, $s0, 1
j mult_loop

mult_eol:
#when exit, restore s0, s1, s2 from sp
lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
#push original sp from stack
addi $sp, $sp, 12
#return
jr $ra
```

Result:

Fig 5.2 new code result

Difference between two methods:
If the input number is X with n bits. The original codes needs to run mult_loop for X times; while the new method only needs to run n times. Which is much more efficient.

6. Questions

1. In cond branches instructions, the next address is calculated by the distance from next instruction to the target instruction. While for jump and jump-link instruction, the address is calculated from absolute address. Because there is no need to check registers.
2. Before calling a function, the stack pointer(sp) needs to be saved(or pushed into stack). Besides, the S temp registers(s0 - s9) needs to be saved also. We can store them into long word pointer $sp. And restore them before returning.
3. First we set stack pointer(sp) to frame point(fp). Then move sp to make space for tmp registers. Then push temp registers in stack. Then save return address to register ra. Then call jump instructions. When returning, restore registers from stack. Move stack pointer back. Then jump to return address ra.