VE370 RC 3

VE370 TA Group

Content

Lecture Slides

Exercise 2.21

Exercise 2.21

The following three problems in this Exercise refer to this function, written in MIPS assembly following the calling conventions from Figure 2.14:

a.

```
f: add   $v0,$a1,$a0
   bnez  $a2,L
   sub   $v0,$a0,$a1
L: jr    $v0
```

b.

```
f: add   $a2,$a3,$a2
   slt   $a2,$a2,$a0
   move  $v0,$a1
   beqz  $a2, L
   jr    $ra
L: move  $a0,$a1
   jal   g        ; Tail call
```

2.21.4 [10] <2.8> This code contains a mistake that violates the MIPS calling convention. What is this mistake and how should it be fixed?

2.21.5 [10] <2.8> What is the C equivalent of this code? Assume that the function's arguments are named `a`, `b`, `c`, etc. in the C version of the function.

2.21.6 [10] <2.8> At the point where this function is called register `$a0`, `$a1`, `$a2`, and `$a3` have values 1, 100, 1000, and 30, respectively. What is the value returned by this function? If another function `g` is called from `f`, assume that the value returned from `g` is always 500.

Exercise 2.21

Figure 2.14 summarizes the register conventions for the MIPS assembly language.

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called `$at`, is reserved for the assembler (see Section 2.12), and registers 26–27, called `$k0–$k1`, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

2.21.4

Mistakes in the code:

- a: `v0` is not preserved on call. And for `jr`, it should followed by `$ra` if we wish to use a return address.
- b: `v0` is not preserved on call, regristers start with `$a` should not be used as temporaries.

2.21.5

a.

```
/*Let a represet $a0, b represent $a1, c represent $a2 */
void function_a(int a, int b, int c){
    /*Let the address of a+b corresponds to subfunc_a(),
     that of a-b corresponds to subfunc_b()*/
    if (c == 0) subfunc_b();
    else subfunc_a();
}
```

b.

```
/*Let a represet $a0, b represent $a1, c represent $a2, d represent $a3
    void function_b(int a, int b, int c, int d){
    c = c + d;
    if (c < a) c = 1;
    else c = 0;
    int ret_val = b;
      if (c == 0){
        a = b;
        g();
      }
      else return ret_val;
 }
```

2.21.6

a. 101, if we take the final return result as $ra
b. 500

2.31

Exercise 2.31

The table below contains the link-level details of two different procedures. In this exercise, you will be taking the place of the linker.

a.	Procedure A				Procedure B			
	Text Segment	Address	Instruction		Text Segment	Address	Instruction	
		0	`lbu $a0, 0($gp)`			0	`sw $a1, 0($gp)`	
		4	`jal 0`			4	`jal 0`	
	Data Segment	0	`(X)`		Data Segment	0	`(Y)`	
		…	…			…	…	
	Relocation Info	Address	Instruction Type	Dependency	Relocation Info	Address	Instruction Type	Dependency
		0	`lbu`	`X`		0	`sw`	`Y`
		4	`jal`	`B`		4	`jal`	`A`
	Symbol Table	Address	Symbol		Symbol Table	Address	Symbol	
		–	`X`			–	`Y`	
		–	`B`			–	`A`	

b.	Procedure A				Procedure B			
	Text Segment	Address	Instruction		Text Segment	Address	Instruction	
		0	`lui $at, 0`			0	`sw $a0, 0($gp)`	
		4	`ori $a0, $at, 0`			4	`jmp 0`	
		…	…			…	…	
		0x84	`jr $ra`			0x180	`jal 0`	
		…	…			…	…	
	Data Segment	0	`(X)`		Data Segment	0	`(Y)`	
		…	…			…	…	
	Relocation Info	Address	Instruction Type	Dependency	Relocation Info	Address	Instruction Type	Dependency
		0	`lui`	`X`		0	`sw`	`Y`
		4	`ori`	`X`		4	`jmp`	`FOO`
						0x180	`jal`	`A`
	Symbol Table	Address	Symbol		Symbol Table	Address	Symbol	
		–	`X`			–	`Y`	
						0x180	`FOO`	
						–	`A`	

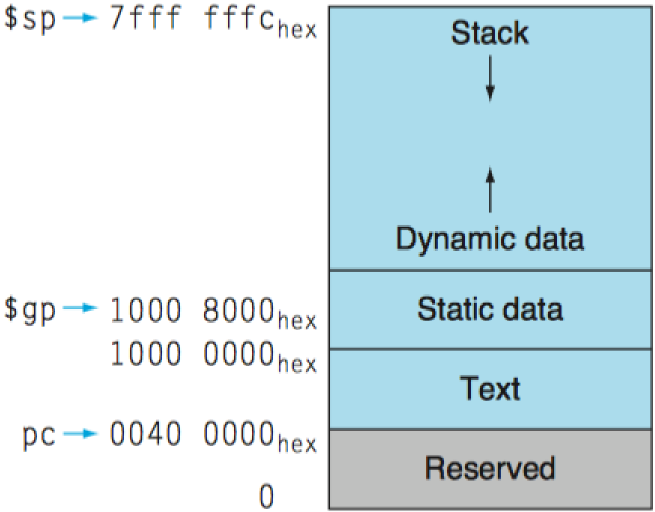
2.31

2.31.1 [5] <2.12> Link the object files above to form the executable file header. Assume that Procedure A has a text size of 0x140 and data size of 0x40 and Procedure B has a text size of 0x300 and data size of 0x50. Also assume the memory allocation strategy as shown in Figure 2.13.

2.31.2 [5] <2.12> What limitations, if any, are there on the size of an executable?

2.31.3 [5] <2.12> Given your understanding of the limitations of branch and jump instructions, why might an assembler have problems directly implementing branch and jump instructions an object file?

2.31.1


$sp → 7fff fffc hex
$gp → 1000 8000 hex
1000 0000 hex
pc → 0040 0000 hex
0
Stack
Dynamic data
Static data
Text
Reserved

2.31.1(2)

To solve this kind of problem, please follow the following steps.

- First: Identify the size in the corresponding data segment (begin from `0x1000 0000`) and text segment (begin from `0x0040 0000`)
- Second: Identify the corresponding dependency.
- Third: Judge the correct address.

2.31.1(3)

a.

```
A: lbu $a0, 8000($gp)
   jal 0x0040 0140
B: sw  $a1, 8040($gp)
   jal 0x0040 0000
```

b.

```
A: lui $at, 0x1000 0000
   ori $a0, $at, 0x1000 0000
   jr 0x0040 02C4
B: sw $a0, 8040($gp)
   j 0x0040 02C0
   (0x0040 02C0:) jal 0x40 0000
```

2.31.2

The limitation shall be the size of data segment/ text segment in memory. Namely, the size of all data shall not exceed `0x1 0000`, and the size of all instructions shall not exceed `0x0FC0 0000`.

2.31.3

The assembler can direct to any place with the object file **without considering the limitation**. However, the beq/bne instruction shall not exceed 2^{16}, and j instruction shall not exceed 2^{28}

Problem 7, 32 bit MUX

```
module 32_bit_mux (A_in, B_in, s,  mux_out);
input [31:0] A_in, B_in;
input s;
output reg [31:0] mux_out;
always @ (*) begin
  if (s) begin
    mux_out = A_in;
  end else begin
    mux_out = B_in;
  end
end
endmodule
```

Problem 8, 1-bit ALU

```
module HW3_8 (a, b, Less, A_invert, B_invert,
 Operation, Carry_in, Result, Set, Overflow)
    input [1:0] Operation;
    input a, b;
    wire a_selected, b_selected;
    output reg Result, Set, Overflow;
    assign a_selected = A_invert ? ~a : a;
    assign b_selected = B_invert ? ~b : b;
    always @(*) begin
    /* Predefined Logics: 00 and, 01 or, 10 plus, 11 less */
            if (Operation == 0) begin
                Result = a_selected & b_selected;
            end else if(Operation == 1) begin
                Result = a_selected | b_selected;
            end else if(Operation == 2) begin
                Result = a_selected + b_selected;
            end else begin
                Result = Less;
            end
            Set = a_selected + b_selected;
        end
endmodule
```

Problem 9, 32*32 Reg File

```
module RegFile
(Regwrite, rs, rt, rd, Write_data, Read_data1, Read_data2);
  input Regwrite;
  input [4 : 0] rs, rt, rd;
  input [31 : 0] Write_data;
  output wire [31 : 0] Read_data1, Read_data2;
  reg [31 : 0] Data_register[0:31] ;

  assign  Read_data1 = Data_register[rs];
  assign  Read_data2 = Data_register[rt];

  always @ (*) begin
    if (Regwrite)  Data_register[rd] = Write_data;
  end

endmodule// RegFile
```