VE370 RC3

HW3 SOLUTION

2.21.4(b)

```
f: addi  $sp,$sp,8
   sw    $ra,4($sp)
   sw    $s0,0($sp)
   move  $s0,$a2
   jal   g
   add   $v0,$v0,$s0
   lw    $ra,4($sp)
   lw    $s0,0($sp)
   addi  $sp,$sp,-8
   jr    $ra
```

```
f:  addi  $sp,  $sp,  -8
    sw    $ra,  4($sp)
    sw    $s0,  0($sp)
    move  $s0,  $a2
    jal   g
    add   $v0,  $v0,  $s0
    lw    $ra,  4($sp)
    lw    $s0,  0($sp)
    addi  $sp,  $sp,  8
    jr    $ra
```

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(b)

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.

```
f:  addi  $sp,  $sp,  -8
    sw    $ra,  4($sp)
    sw    $s0,  0($sp)
    move  $s0,  $a2
    jal   g
    add   $v0,  $v0,  $s0
    lw    $ra,  4($sp)
    lw    $s0,  0($sp)
    addi  $sp,  $sp,  8
    jr    $ra
```

```
int f(int a, int b, int c){
    return g(a,b) + c;
}
```

\$v0–\$v1	2–3	Values for results and expression evaluation
\$a0–\$a3	4–7	Arguments

2.21.6(b)

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.

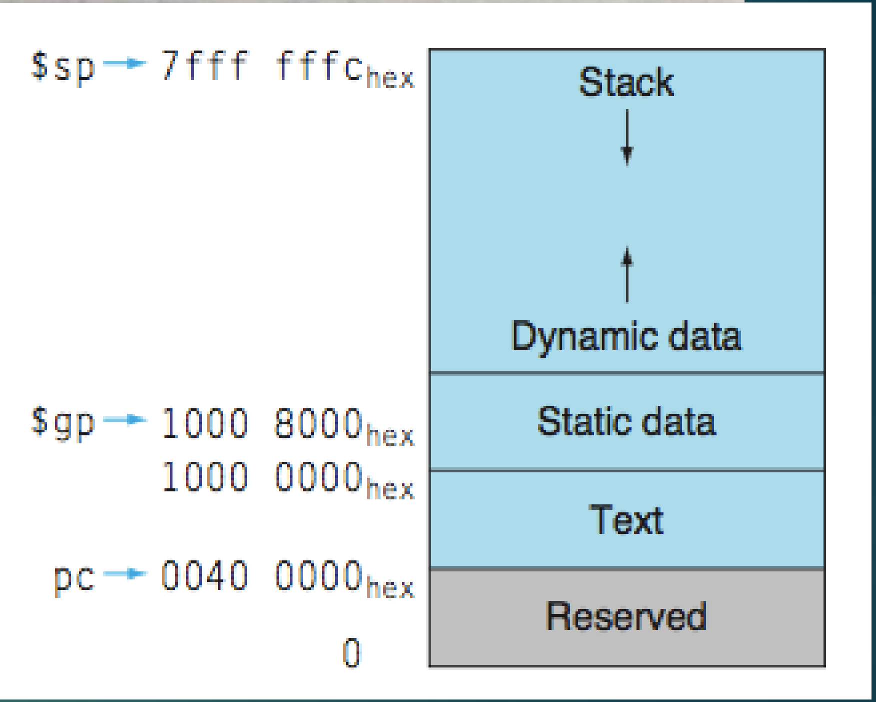
```
int f(int a, int b, int c){
    return g(a,b) + c;
}
```

ANSWER: 500 + 1000 = 1500

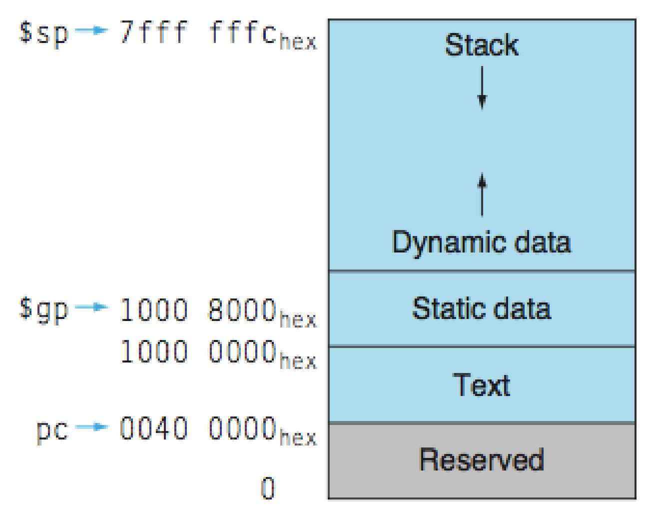
2.31.1(b)

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, 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.

	Procedure A	Procedure B
Text	0x140	0x300
data	0x40	0x50



	Procedure A	Procedure B
Text	0x140	0x300
data	0x40	0x50



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	
		8	jal 0			…	…	
		…	…			0x180	jr $ra	
						…	…	
	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
		8	jal	B				
	Symbol Table	Address	Symbol		Symbol Table	Address	Symbol	
		–	X			–	Y	
		–	B			0x180	FOO	

Executable File		
	Text size	0x440
	Data size	0x90
Text Segment	**Address**	**Instruction**
	0x0040,0000	lui $at, 0x1000
	0x0040,0004	ori $a0, $at, 0
	0x0040,0008	jal 0x400140
	…	
	0x0040,0140	sw $a0, 8040($gp)
	0x0040,0144	j 0x4002c0
	…	
	0x0040,02c0	jr $ra
	…	
Data Segment	**Address**	
	0x1000,0000	X
	…	
	0x1000,0040	Y

2.31.2~3(b)

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 in an object file?

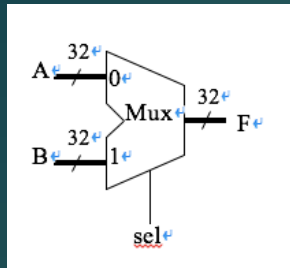
2.31.2: data size: 0x10010000– 0x10000000=0x10000
text: 0x10000000 – 0x400000 = 0xFC00000

	31–26	25–21	20–16	15–0
I	opcode	rs	rt	immediate
J	opcode	immediate		

2.31.3: limited immediate length, can not jump too far.

Branch : 16bits
Jump: 26bits

7.



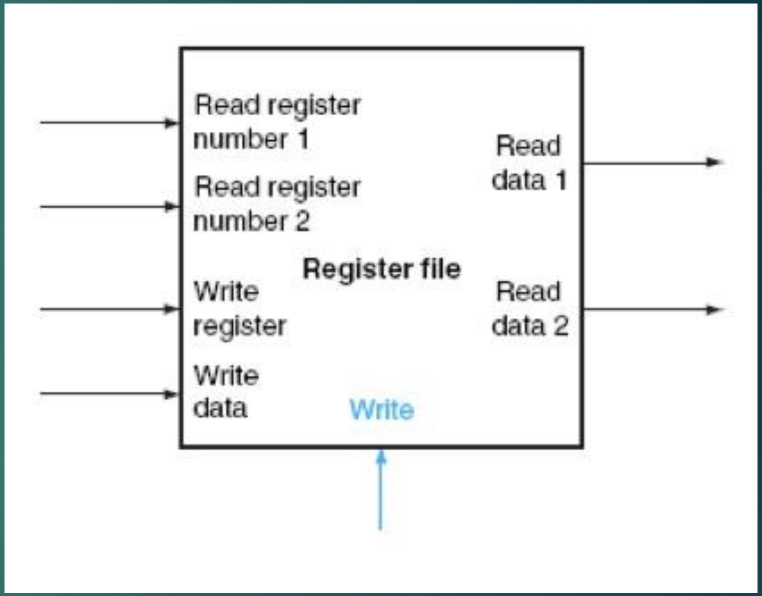

```
Module mux(A,B,sel,F);
input [31:0] A;
Input [31:0] B;
input sel;
output [31:0] F;
reg[31:0] F;
always @(sel or A or B)
begin
      if (sel == 0) F = A;
      else F = B;
end
endmodule
```

```
module testalucontrol;
    reg [5:0] funct;
    reg [1:0] ALUOp;
    wire [3:0] ALUControl;
    ALUcontrol uut (
        .funct(funct),
        .ALUOp(ALUOp),
        .ALUControl(ALUControl)
    );
    initial begin
        $display("*******************************");
        $display("The textual results:");
        $monitor($time,"ALUOP=%b, funct=%b, ALUcontrol=%b",ALUOp,funct,ALUControl);
    end
    initial begin
        funct = 6'bxxxxxx;
        ALUOp = 2'b00;
        #100 ALUOp=2'b01;
        #100 ALUOp=2'b10; funct=6'b100000;
        #100 funct=6'b100010;
        #100 funct=6'b100100;
        #100 funct=6'b100101;
        #100 funct=6'b101010;
        #100 ALUOp=2'b11; funct=6'bxxxxxx;
    end
endmodule
```

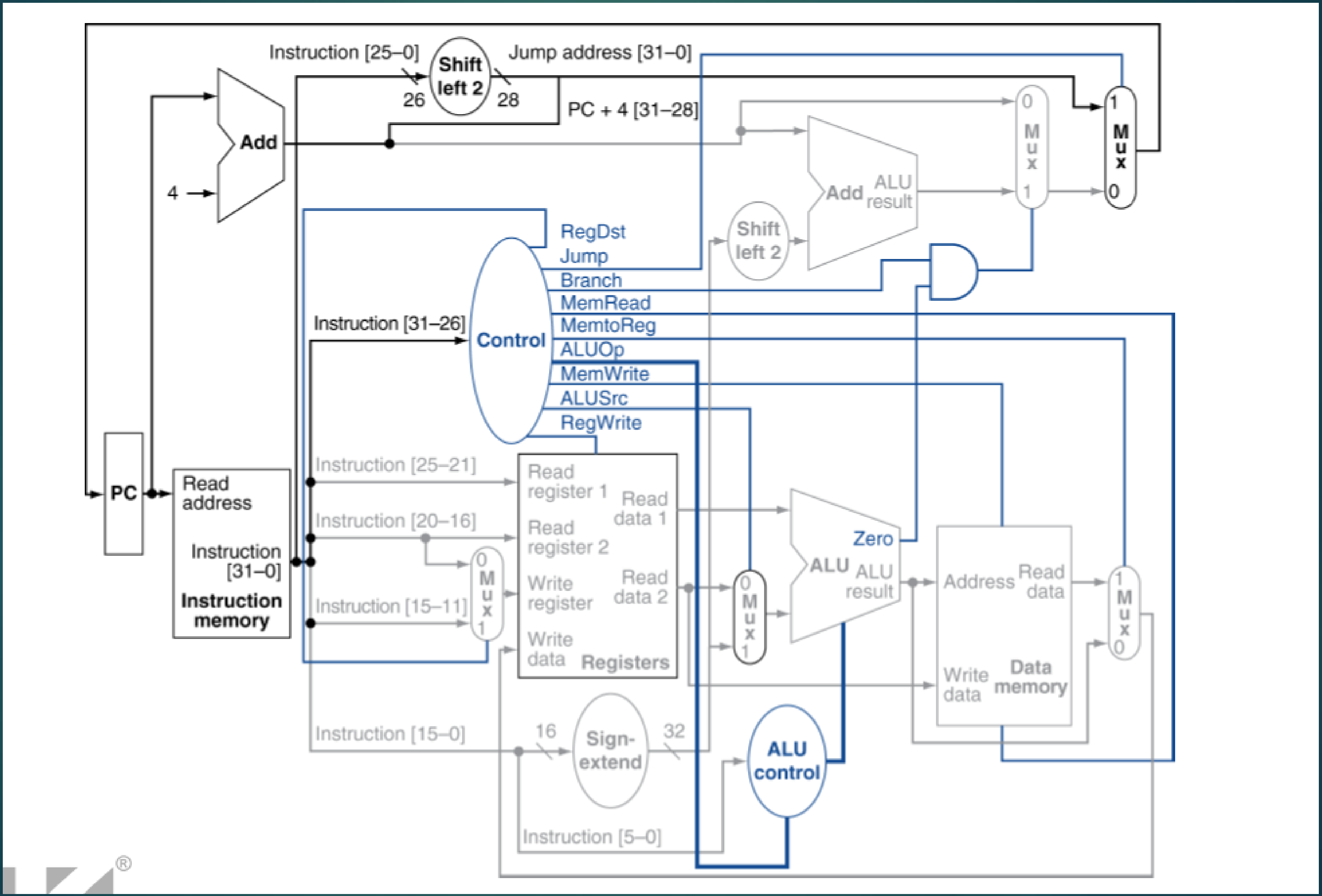
```
module
register(rdreg1,rdreg2,wrreg,wrdata,write,rddata1,rddata2);
input [4:0]rdreg1;
input [4:0]rdreg2;
input [4:0]wrreg;
input [31:0]wrdata;
input write;
output[31:0]rddata1;
output[31:0]rddata2;
reg [31:0]data[0:31];
reg [31:0]rddata1;
reg [31:0]rddata2;
always @(posedge write) begin
      data[wrreg]=wrdata;
end

always @(rdreg1) begin
      assign rddata1=data[rdreg1];
end

always @(rdreg2) begin
      assign rddata2=data[rdreg2];
end
endmodule
```



Single Cycle



Single Cycle

R-format

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

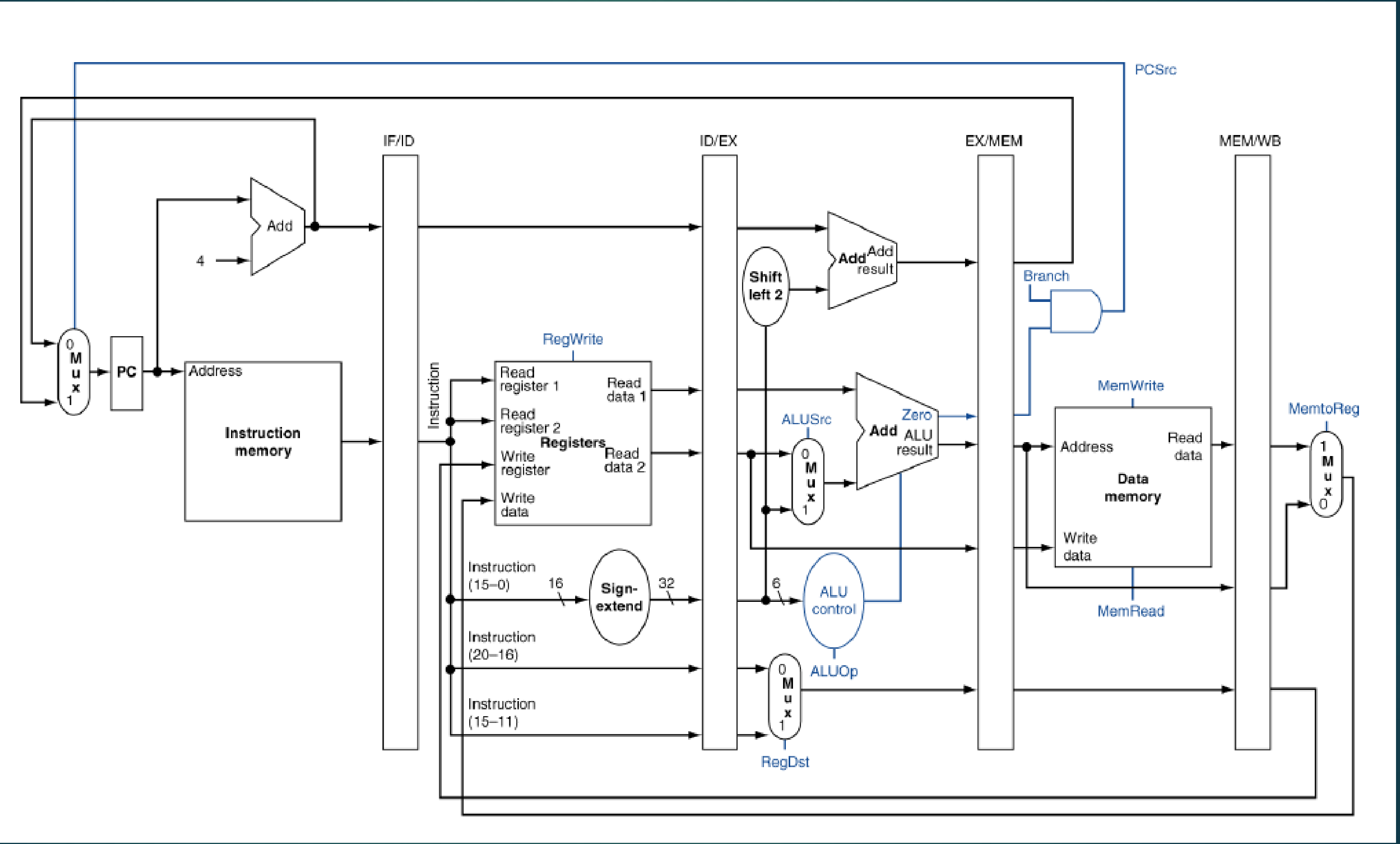
I-format

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

J-format

op	address
6 bits	26 bits

Pipeline


PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add
Add result
Branch
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
Instruction (15–0)
16
Sign-extend
32
6
Instruction (20–16)
Instruction (15–11)
RegDst