1. Assume we have an array in memory that contains **int\* arr = {1,2,3,4,5,6,0}**. Let the values of arr be a multiple of 4 and stored in register s0. What do the snippets片段 of RISC-V code do? Assume that all the instructions are run one after the other in the same context. (15 points)

**a) lw t0, 12(s0) --> 读s0[12] 到t0,也就是t0 =4**

**b) slli t1, t0, 2 t1 = t0 <<2 t1 = 16;**

**add t2, s0, t1 t2 = s0+ t1,**

**lw t3, 0(t2) --> t3 = t2[0] t3= 5;**

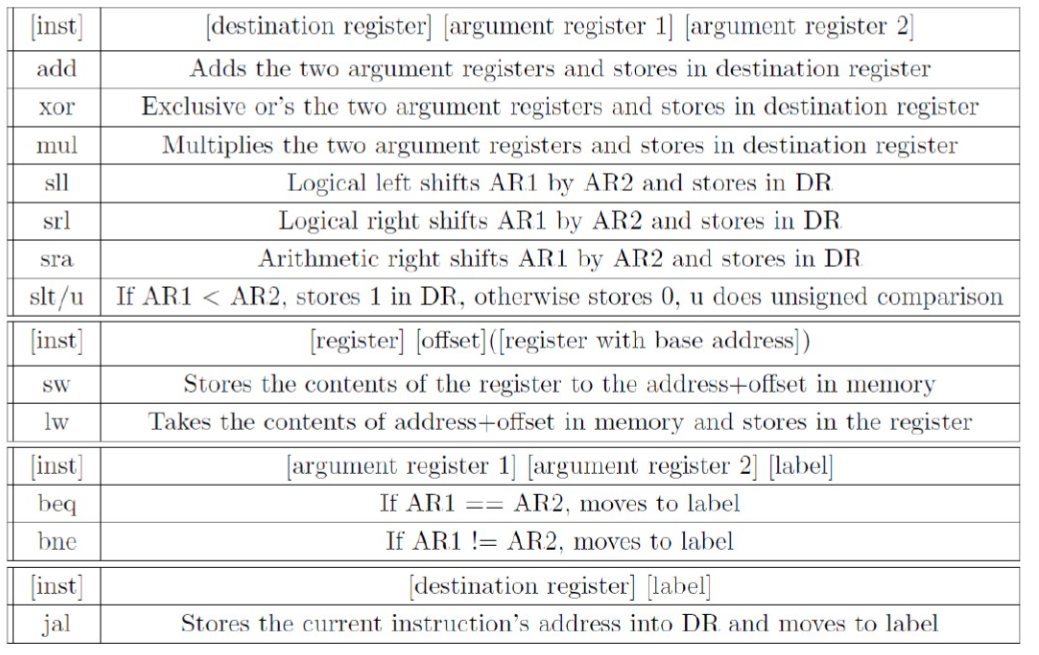
**addi t3, t3, 1 t3++;**

**sw t3, 0(t2) t2[0] = t3; 也就是让数组中5变为6**

**c) lw t0, 0(s0) t0 = 1**

**xori t0, t0, 0xFFF --> t0 = 0000 1111 1111 1110**

**addi t0, t0, 1 #t0 = 0000 1111 1111 1111**

**d)** 

You may also see that there is an "i" at the end of certain instructions, such as

addi, slli, etc. This means that AR2 becomes an "immediate" or an integer instead

of using a register.

While only using the instructions (and their "i" forms就是加i) given above, how can we branch on the following conditions:

**s0 < s1,**

**slt t0,s0,s1**

**beq t0,x0**

**s0 ≥ s1**

**slt t0,s1,s0**

**beq t0,x0**

**s0 > 1**

**slti t0,s0,1 # s0>1 存0**

**beq t0,x0 #如果是 0就br**

1. Write a function quadruple四倍 in RISC-V that, when given an integer x, returns 4x. (6 points)

Quadruple

**slli t1, t0, 2**

1. Write a function power in RISC-V that takes in two numbers x and n, and returns xn. You may assume that n≥0 and that multiplication will always result in a 32-bit number. (8 points)

N ==1 , 返回x, n ==2 return x^2

T =1;// x 存在t中

While (n >= 1) {

t = t \* x;

n--;

}

Return t;

pow: addi sp,sp,-8 # adjust stack for 2 items

      sw s1, 4(sp)  # save s1 for use afterwards

      sw s0, 0(sp)  # save s0 for use afterwards

      addi s0,x0,1  # t = 1;

      add s1,a1,0 # s1= n

      sub s1,s1,1

Loop:

      mul s0,s0,a0; # t= t \*x

      sub s1,s1,1

      bge s1,x0,Loop; # if n >= 0 ,because n >= 0 so we could do  t= t \*x at lease times.

      add a0,s0,x0  # return value s0

      lw s0, 0(sp)  # restore register s0 for caller

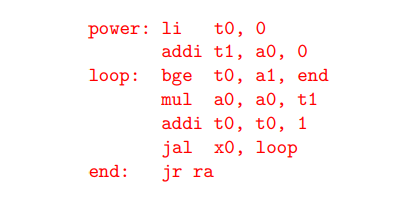
      lw s1, 4(sp)  # restore register s1 for caller

      addi sp,sp,8  # adjust stack to delete 2 items

      jr ra         # jump back to calling routine

答案如下:

power: li t0, 0 addi t1, a0, 0 loop: bge t0, a1, end mul a0, a0, t1 addi t0, t0, 1 jal x0, loop end: jr r



1. Translate between the C and RISC-V verbatim (15 points)

|  |  |
| --- | --- |
| C | RISC-V |
| **// computes s1 = 2ˆ30**  **s1 = 1;**  **for(s0=0;s0<30;s++) {**  **s1 \*= 2;**  **}** | **Add x10,x0,x0 # s1 = 0**  **addi x10, x0, 1 # s1 = 1;**  **Add x9,x0,x0 # s0 = 0**  **Loop:**  **slli x10,x10,2 # s1<<1**  **addi x9,x9,1 # i++**  **addi x13,x0,30 # x13=30**  **blt x9,x13,Loop** |
| **S0[0] = 0;**  **S1 = 2;**  **s0[1] = s1;**  **s1= 8;**  **t0 = 8;**  **t0 = s1;** | **sw x0, 0(s0) # s 0[0] = 0**  **addi s1, x0, 2 # s1 = 2**  **sw s1, 4(s0)# s0[1] = 2**  **slli t0, s1, 2 # s1= 8**  **add t0, t0, s0 # t0 = 8**  **sw s1, 0(t0) # s1 = 8** |
| S0  =5;  S1 = 10;  T0 = 10;  If( t0 != s1){      s1 = s0 -1;  }  else{      s0 = 0^0;// s0 = 0  }  exit;  答案  // s0 -> a, s1 -> b int a = 5, b = 10; if(a + a == b) { a = 0; } else { b = a - 1; } | **addi s0, x0, 5**  **addi s1, x0, 10**  **add t0, s0, s0**  **bne t0, s1, else**  **xor s0, x0, x0**  **jal x0, exit**  **else:**  **addi s1, s0, -1**  **exit:** |
| s0 = 4;  s1 = 5;  s2 = 6;  s3 = s0+s1; // s3 = 9  s3 = s3+s2;  s3 = s3+10; | **addi s0, x0, 4**  **addi s1, x0, 5**  **addi s2, x0, 6**  **add s3, s0, s1**  **add s3, s3, s2**  **addi s3, s3, 10** |
| s1 = s1;  while(s0 != 0){      s1 = s1+ s0;      s0  = s0 -1;  } | **addi s1, s1, 0**  **loop:**  **beq s0, x0, exit**  **add s1, s1, s0**  **add s0, s0, -1**  **jal x0, loop**  **exit:** |

1. Computing a Fibonacci Number. The Fibonacci number Fn is recursively defined as

**F(n) = F(n - 1) + F(n - 2);**

where **F(1) = 1 and F(2) = 1. So, F(3) = F(2) + F(1) = 1 + 1 = 2**, and so on. Write the RISC-V assembly for the fib(n) function, which computes the Fibonacci number F(n): (12 points)

**int fib(int n){**

**int a = 0;**

**int b = 1;**

**int c = a + b;**

**while (n > 1) {**

**c = a + b;**

**a = b;**

**b = c;**

**n--;**

**}// n == 1, c =1 ; n ==2 ; 循环一次**

**return c;**

**}**

fib: addi sp,sp,-8 # adjust stack for 2 items

      sw s1, 4(sp)  # save s1 for use afterwards

      sw s0, 0(sp)  # save s0 for use afterwards

      addi s0,x0,x0 # int a = 0;

      addi s1,x0,1  #int b = 1;

      addi t1,x0,1  # int c = a + b;

      sub a0,a0,1  # n--

Loop:

      ble s1,x0,end     # n == 1, 不循环

      add t1,s1,s0      # c= a+ b

      add s0,s1,x0      # a= b

      add s1,t1,x0      # b  =c

      sub a0,a0,1       #n--

      bgt a0,x0,Loop; # if n >= 0 ,because n >= 0 so we could do  t= t \*x at lease times.

end:

      add a0,t1,x0 # return

      lw s0, 0(sp)  # restore register s0 for caller

      lw s1, 4(sp)  # restore register s1 for caller

      addi sp,sp,8  # adjust stack to delete 2 items

      jr ra         # jump back to calling routine

1. We have several addressing modes to access memory (immediate not listed):
   1. Base displacement addressing: Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
   2. PC-relative addressing: Uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions)
   3. Register Addressing: Uses the value in a register as a memory address (jr)

(1) What is range of 32-bit instructions that can be reached from the current PC using a branch instruction? (4 points)

Each bits target 2bytes.

32bit = 4bytes

The immediate field of the branch instruction is 12bits. Thus, branch instruction can reach [-2^11, 2^11-1]. 32-bit = 4bytes. We can reach [-2^10, 2^10-1] instructions.

每个位以 2 个字节为目标。32bit = 4bytes 分支指令的立即数字段为 12bits。 因此，分支指令可以达到 [-2^11, 2^11-1]。 32 位 = 4 字节。 我们可以达到 [-2^10, 2^10-1] 指令。

(2) What is the range of 32-bit instructions that can be reached from the current PC using a jump instruction? (4 points)

2^20 bits = > [-2^19 , 2^19-1]

Each bits target 2bytes.

32bit = 4bytes

The current PC use a jump instruction can reach [-2^18, 2^18-1] instructions

(3) Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you’ll need your RISC-V green card!). (4 points)

0x002cff00: loop: sub t1, t2, t0 |\_\_\_01000\_\_\_\_\_|\_\_00\_\_\_\_\_\_|\_\_0x05(x5)\_\_\_\_ \_\_|\_\_\_0x07\_\_\_|\_\_0x06\_ t1= x6, \_\_\_\_\_|\_\_0x33\_\_|

0x002cff04: jal ra, foo|\_\_\_\_\_\_\_\_\_\_\_0x14(不是28因为省略了一个2)\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_rd= ra= 0x01 \_\_\_\_\_\_\_\_\_\_\_\_|\_\_0x6F\_\_|

0x002cff08: bge t1, zero, loop |\_ 1 [12] 111111[10:5]\_\_\_\_|\_\_\_0x00\_\_\_\_\_|\_\_0x06\_\_\_\_\_\_|\_\_101\_\_\_\_\_\_|\_1100\_[4:1]\_ 1[11]\_\_\_\_\_|\_\_0x63\_\_|

... imm = -8 1111 1111 1000,

0x002cff2c: foo: jr ra ra=\_\_ pc+4 =\_\_0x002cff08\_\_\_\_\_\_\_\_\_\_\_\_

1. Which step in CALL resolves relative addressing? Absolute addressing? (5 points)

汇编器解析相对寻址，链接器解析绝对寻址。

(1) Assembler 转换的结果是.o 文件。 .o 文件包含目标文件头，用于描述目标文件的其他文件的大小和位置。 文本段包含机器代码，静态数据段包含程序中使用的程序。 重定位信息是在程序载入内存时写入指令和数据的地址，符号表包含一些外部参数的未定义标签，调试信息用于描述C与机器码之间的编译。

(2)通过Linker实现一个可执行文件。 这种可执行文件的主要工作是为代码和数据分配实内存地址。 将多个目标文件链接在一起。

Assembler resolves relative addressing,

Linker resolves absolute addressing.

(1) The result converted by Assembler is an .o file. The .o file includes the object file header, which is used to describe the size and location of other files of the object file. The text segment contains the machine code, and the static data segment contains the programs used in the program. relocation information is written in the address of instructions and data when the program is loaded into memory, symbol table contains undefined labels of some external parameters, and debugging information is used to describe the compilation between C and machine code.

(2) An executable file is realized through Linker. The main job of such an executable file is to allocate real memory addresses for code and data. Link multiple object files together.

1. What does RISC stand for? How is this related to pseudo instructions? (5 points)

Reduced Instruction Set Computer.

Reduced Instruction needs more lines of assemble codes, pseudo instructions could help programmer to accelerate their coding process without lengthy code writing.

精简指令集计算机。减少指令需要更多的汇编代码行，伪指令可以帮助程序员加速他们的编码过程，而无需冗长的代码编写。