**2019 Computer Architecture Problem Set #2 Solution**

1. Solution)

**addi x30, x10, 8 // x30 will be the address of A[1]**

**addi x31, x10, 0 // x31 will be the base address of array A**

**sd x31, 0(x30) // store the x31 (base address) into the A[1]**

**ld x30, 0(x30) // load the A[0] into the x30**

**add x5,x30, x31 // f = the base address of array A + the base address of array A**

**→ Corresponding C statement:**

**A[1] = &A[0];**

**f = &A[0] + A[1];**

1. Solution)

**slli x30, x5, 3 // f \* 8**

**add x30, x10, x30 // x30 = the address of A[f]**

**slli x31, x6, 3 // g \* 8**

**add x31, x11, x31 // x31 = the address of B[g]**

**ld x5, 0(x30) // x5 = A[f]. Here, register x5 holds the value of f.**

**addi x12, x30, 8 // x12 = the address of A[f] + 8 = the address of A[f+1]**

**ld x30, 0(x12) // x30 = A[f+1]**

**add x30, x30, x5 // x30 = A[f+1] + f**

**sd x30, 0(x31) // B[g] = A[f+1] + A[f]**

**→ Corresponding C statement:**

**B[g] = A[f+1] + A[f];**

**f=A[f];**

**Though f=A[f] is performed before B[g] = A[f+1] + A[f], in the C code, we should do B[g] = A[f+1] + A[f] first. If we do f=A[f] first, f value will be overwritten to the value of A[f], affecting the result of B[g] = A[f+1] + A[f].**

1. Solution)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x6 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| x5 | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |

When x6 is 0, the procedure will jump to DONE.

Therefore, the final value of x5 is 20.

1. Solution)

**addi x6, x0, 0**

**addi x29, x0, 100**

**// Initialize the variable of “result” to 0.**

**// And then, initialize the value of register x29 to 100.**

**LOOP: ld x7, 0(x10)**

**// load the value of base address what is saved in x10 into register x7.**

**add x5, x5, x7**

**// execution of the addition that “i = i + the value of MemArray”**

**addi x10, x10, 8**

**// increase the address of MemArray by 8. It means that x10 will be the address of the next element of //MemArray.**

**addi x6, x6, 1**

**// Increase by 1 for the variable of “result” for counting.**

**blt x6, x29, LOOP**

**// Go to start of the LOOP if the x6(result) is smaller than x29(100).**

**→ C statement:**

**result = 0;**

**for(result=0; result<100; result++){**

**i=i + \*MemArray;**

**MemArray = MemArray+8;**

**}**

1. Solution)

**B[8] = A[i] + A[j];**

**→ slli x28, x5, 3**

**slli x29, x6, 3**

**add x30, x10, x28 // save the address of A[i] into x30**

**add x31, x10, x29 // save the address of A[j] into x31**

**ld x30, 0(x30) // load the A[i] into x30**

**ld x31, 0(x31) // load the A[j] into x31**

**add x7, x30, x31 // x7 = A[i] + A[j]**

**sd x7, 64(x11) // store the x7 (A[i] + A[j]) into B[8].**

1. Solution)

**B[8] = A[i−j];**

**→ sub x7, x5, x6**

**slli x7, x7, 3**

**add x28, x10, x7 // find the address of the A[i-j] and save it into x28**

**ld x28, 0(x28) // load A[i-j] into x28**

**sd x28, 64(x11) // store the value of A[i-j] into B[8]**

1. Solution)

**for(i=0; i<a; i++)**

**for(j=0; j<b; j++)**

**D[4\*j] = i + j;**

**→ RISC-V assembly code**

**addi x7, x0, 0**

**LOOP1:**

**bge x7, x5, Exit**

**addi x29, x0, 0**

**LOOP2:**

**bge x29, x6, LOOP3**

**slli x30, x29, 5 // x30 = 4\*j\*8**

**add x30, x10, x30 // x30 = the address of D[j\*4]**

**add x31, x7, x29 // x31 = i+j**

**sd x31, 0(x30) // store “i+j” into the D[j\*4]**

**addi x29, x29, 1 // j ++**

**jal x0, LOOP2**

**LOOP3:**

**addi x7, x7, 1 // i ++**

**jal x0, LOOP1**

**Exit:**

1. Solution)

**→ The number of immediate filed is 20 and 12 for jal and beq instruction, respectively, and PC[0] (LSB of PC) is always filled with 0. It means that the jal instruction can jump to (PC-) ~ (PC-1) and the beq instruction can branch to (PC-) ~ (PC+-1). Therefore, the range of each instruction for the current PC (0x20000000) is:**

**jal instruction range: 0x1FF00000 ~ 0x200FFFFF**

**branch instruction range: 0x1FFFF000 ~ 0x20000FFF**

1. Solution)

**→ slli x7, x5, 4 -> x7 = 0x0000000AAAAAAAA0**

**or x7, x7, x6 -> x7 = x7 | x6 = 0x0000000AAAAAAAA0 | 0x1234567812345678**

**Therefore, x7 will be 0x1234567ABABEFEF8**

1. Solution)

**→ temp= v[k];**

**v[k]= v[k+1];**

**v[k+1] = temp;**

**→ Assume that x5 holds the variable called “temp”, x6 holds the variable called “k”, and the base address of array “v” is x7.**

**slli x28, x6,3 // k\*8**

**addi x29, x28, 8 // k\*8 + 8**

**add x30, x7, x28 // x30 = Address of v[k]**

**add x31, x7, x29 // x31 = Address of v[k+1]**

**ld x10, 0(x30) // x10 = v[k]**

**ld x11, 0(x31) // x11 = v[k+1]**

**addi x5, x10, 0 // temp = v[k]**

**addi x10, x11, 0 // v[k] = v[k+1]**

**addi x11, x5, 0 // v[k+1] = temp**

**sd x11, 0(x30) // save the value of v[k+1] into v[k]**

**sd x10, 0(x31) // save the value of v[k] into v[k+1]**

1. Solution)

**→ Let’s assume that register x18 (a), x19 (b), x20 (c), and x21 (d) are used in function “f”, and after procedure “g” is called, procedure “g” uses register x10 and x11 for passing the arguments for int a and int b, respectively. Also, we assume that procedure “g” uses register x12 for storing the return value. As the register x18~x21 are callee save registers, so in the caller, we don’t need to care about them. So RISC-V assembly code is:**

**f: addi sp, sp, -8 // allocate the stack for register x1**

**sd x1, 0(sp) // save the return address x1 to the stack**

**addi x10, x18, 0 // place an argument for int a**

**addi x11, x19, 0 // place an argument for int b**

**jal x1, g // jump to procedure g**

**addi x10, x12, 0 // place an argument for the result of g(a,b)**

**add x11, x20, x21 // c+d is saved in register x11**

**jal x1, g // jump to procedure g**

**ld x1, 0(sp) // restore the return address x1 from the stack**

**addi sp, sp, 8 // increase the stack pointer by 8**

**jalr x0, 0(x1) // The end of procedure**