|  |
| --- |
| **Assignment # 2**  **SYSC 5704 – Elements of Computer Systems** |
| Fall 2014  Submitted To  Dr. R. Gregory Franks  By  **Ferhan Jamal (100 953 487)**  Carleton University |

**2.8 [5] <§2.4> Translate 0xabcdef12 into decimal.**

**Answer:**

10\*16E7 + 11\*16E6 + 12\*16E5 + 13\*16E4 + 14\*16E3 + 15\*16E2 + 1\*16E1 + 2\*16E0 = 2882400018

**2.7 [5] <§2.3> Show how the value 0xabcdef12 would be arranged in memory of a little-endian and big-endian machine. Assume the data is stored starting at address 0.**

**Answer:**

Big-endian and little-endian are the terms which describe the symmetry in which a sequence of bytes are stored in computer memory.

**Little-endian**: In little endian, the least significant byte is stored at the smallest address.

**Big-endian**: In big endian, the most significant byte is stored at the smallest address.

The value 0xabcdef12 is a hexadecimal number. Let us suppose that it is a 32 bits value with hex digit having a value of 4 bits. Therefore the ab, cd, ef, and 12 will individually have a size of 1 byte with the total of 4 bytes. It is given in the question that the data is stored starting at address 0 so the value 0xabcdef12 would take the little endian and big endian form in the following way:

**Little endian:** In little endian, the least significant byte is stored at the smallest address so 0x12 will come 1st in the memory followed by 0xef, 0xcd and 0xab from the value 0xabcdef12 computer memory in little endian case.

**Big endian:** In big endian, the most significant byte is stored at the smallest address so 0xab will come 1st in the memory followed by 0xcd, 0xef and 0x12 from the value 0xabcdef12 in computer memory in big endian case.

|  |  |  |  |
| --- | --- | --- | --- |
| Little-Endian | | Big-Endian | |
| Address | Data | Address | Data |
| 0 | 0x12 | 0 | 0xab |
| 1 | 0xef | 1 | 0xcd |
| 2 | 0xcd | 2 | 0xef |
| 3 | 0xab | 3 | 0x12 |

**2.12 Assume that registers $s0 and $s1 hold the values 0x80000000 and 0xD0000000, respectively.**

**1. [5] <§2.4> What is the value of $t0 for the following assembly code?**

**add $t0, $s0, $sl**

**Answer:**

$s0= 0x80000000 which can also be written as: (80000000)16

$s1= 0xD0000000 which can also be written as: (D0000000)16

So, $s0+$s1 = (80000000)16 + (D0000000)16 =(150000000)16 and $t0 will be = (50000000)16 not (150000000)16 as (150000000)16 can be written in binary form as 0001 0101 0000 0000 0000 0000 0000 0000 0000 which is 36 bits i.e. more than 32 bits which will results in an overflow.

**2. [5] <§2.4 Is the result in $t0 the desired result, or has there been overflow?**

**Answer:**

No, $t0 is not the desired output obtained above by doing the addition as there is an overflow.

The desired output of $t0 will be (50000000)16

**3. [5] <§2.4> For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?**

**sub $t0, $s0, $sl**

**Answer:**

$s0= 0x80000000 which can also be written as: (80000000)16

$s1= 0xD0000000 which can also be written as: (D0000000)16

So, $t0 = $s0 - $s1 = (80000000)16 - (D0000000)16 =-(50000000)16

The 2’s complement for -(50000000)16 is 1011 0000 0000 0000 0000 0000 0000 0000 so in this case no overflow occurs.

**4. [5] <§2.4> Is the result in $t0 the desired result, or has there been overflow?**

As explained above, the value of $t0 is -(50000000)16. The 2’s complement of it is 1101 0000 0000 0000 0000 0000 0000 0000 which is 32 bits so in this case, no overflow occurs.

**5. [5] <§2.4> For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?**

**add $t0, $s0, $sl**

**add $t0, $t0, $s0**

After the addition of the above 2 instructions, $t0 will be: 0xD0000000. The explanation is given below:

As explained above,

add $t0, $s0, $sl implies $t0 = $s0 + $sl

$s0= 0x80000000 which can also be written as: (80000000)16

$s1= 0xD0000000 which can also be written as: (D0000000)16

So, $t0 ($s0+ $sl) = (50000000)16

In the 2nd part of the question, it is given that add $t0, $t0, $s0 which implies $t0 = $t0 + $s0. The value of $t0 is (50000000)16 and $s0 is (80000000)16. Therefore,

$t0 = $t0 + $s0 = (50000000)16 + (80000000)16

$t0 = $t0 + $s0 = (50000000)16 + (80000000)16

$t0= $t0 + $s0 = D0000000

**6. [5] <§2.4> Is the result in $t0 the desired result, or has there been overflow?**

**Answer:**

In the 1st instruction, there is an overflow as the correct result cannot be expressed in 32 bits but there is no overflow in the 2nd instruction.

**2.13 Assume that $s0 holds the value 128ten.**

**1. [5] <§2.4> For the instruction add $t0, $s0, $s1, what is the range(s) of values for $s1 that would result in overflow?**

It is given in the question that $s0 = (128)10 which can be written in the binary form as (10000000)2 .  The line add $t0, $s0, $s1 implies $t0=$s0+$s1 where value of $s0 is (128)10 or (10000000)2 Overflow will occur only when size of the value represented in binary form is more than 32 bits (>32 bits). The values greater than 0x7FFFFF7F in $s1 will result in an overflow

**2. [5] <§2.4> For the instruction sub $t0, $s0, $s1, what is the range(s) of values for $s1 that would result in overflow?**

The line sub $t0, $s0, $s1 implies $t0=$s0-$s1 where value of $s0 is (128)10 or (10000000)2 Here overflow will occur only when size of the value ($t0) represented in binary form is more than 32 bits. The values smaller than 0x80000081 in $s1 will result in an overflow in the instruction.

**3. [5] <§2.4> For the instruction sub $t0, $s1, $s0, what is the range(s) of values for $s1 that would result in overflow?**

The line sub $t0, $s0, $s1 implies $t0=$s1-$s0 where value of $s0 is (128)10 or (10000000)2 Hereoverflow will occur when size of the value ($t0) represented in binary form is more than 32 bits. The values smaller than 0x80000080 in $s1 will cause an overflow in the instruction.

**2.18 Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.**

**1. [5] <§2.5> How this would this affect the size of each of the bit fields in the R-type instructions?**

In case of r-type instructions, the size of rs, rt and rd bit should be increased to 7 bits with the increase of the register file to 128 registers and the size of opcode or func fields or both of them should also be increased as the ISA is expanded to 4 times. Now, the new ISA can have any one the following values:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| op | rs | rt | rd | shamt | func |
| 8 | 7 | 7 | 7 | 7 | 6 |
| 6 | 7 | 7 | 7 | 7 | 8 |
| 7 | 7 | 7 | 7 | 7 | 7 |

**2. [5] <§2.5> How this would this affect the size of each of the bit fields in the I-type instructions?**

In I-type instructions, the same of the register fields rs and rt same in the above case should be increased to 7 bits from 5 bits and with the increase in the value of instruction count, the opcode must be increased to 8 bits from 6 bits and the constant or address should be increased from 16 bits to 18 bits. The new MIPS I-format instructions would look like:

Opcode- 8 bits

rs- 7 bits

rt- 7 bits

constant or address- 18 bits

**3. [5] <§§2.5, 2.10> How could each of the two proposed changes decrease the size of an MIPS assembly program? On the other hand, how could the proposed change increase the size of an MIPS assembly program?**

The size of the MIPS assembly program will increase for sure if there is an increase in instructions in the ISA

In the opposite case, if the new included instructions have more complex instructions then the programs using these new instructions may decrease the program size. We now have 128 registers which can also decrease the load/store instructions; etc.

**2.24 [5] <§2.7> Suppose the program counter (PC) is set to Ox2000 0000. Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as Ox4000 0000? Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?**

PC (Initial Value in Hex):- 0x2000 0000

PC (Initial Value in Binary):- 0010 0000 0000 0000 0000 0000 0000 0000

PC (To be required in binary):- 0100 0000 0000 0000 0000 0000 0000 0000

PC (To be required in Hex):- 0x4000 0000

No, it is not possible to use the jump(j) MIPS assembly instruction to set the PC value to 0x40000 000 from 0x2000 000. This is because jump instruction can only jump to the addresses which have the same 1st 4 bits of the current value of the PC.

If we see above, the 1st four bits of the program counter i.e. the initial value is 0010 (binary) and the 1st four bits which is to be required is 0100 (binary) which is totally impossible to set [0100 (binary 1st 4 bits)] by using jump instruction.

PC (Initial Value in Hex):- 0x2000 0000

PC (Initial Value in Binary):- 0010 0000 0000 0000 0000 0000 0000 0000

PC (To be required in binary):- 0100 0000 0000 0000 0000 0000 0000 0000

PC (To be required in Hex):- 0x4000 0000

In the case of branch-on-equal instruction, it is also not possible to set the PC address to 0x4000 0000 from 0x2000 000. The constant required to achieve this value is 0x80000000 and this number will not come in 16 bits and it will take 28 bits.

**2.27 [5] <§2.7> Translate the following C code to MIPS assembly code. Use a minimum number of instructions.**

**Assume that the values of a, b, i, and j are in registers $s0, $s1, $t0, and $t1, respectively. Also, assume that register $s2 holds the base address of the array D.**

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

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

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

**}**

**}**

**Answer:**

addi $t0, $0, 0

beq $0, $0, TEST 1

LOOP1: addi $t1, $0, 0

beq $0, $0, TEST 2

LOOP2: add, $t5, $t0, $t1

s11 $t4, $t1, 4

add $t4, $t4, $s2

sw $t5, 0($t4)

addi $t1, $t1, 1

TEST2: slt $t4, $t1, $s1

bne $t4, $0, LOOP2

addi $t0, $t0, 1

TEST 1: slt $t4, $t0, $s0

bne $0, $t4, LOOP1

**2.33 [5] <§2.8> For each function call, show the contents of the stack after the function call is maked. Assume the stack pointer is opriginally at address 0x7ffffffc, and follow the register conventions as specified below:**

|  |  |
| --- | --- |
| **Preserved** | **No Preserved** |
| **Saved registers $s0-$s7** | **Temporary registers $t0-$t9** |
| **Stack pointer register $sp** | **Argument registers $ar0-$ar3** |
| **Return address registers $ra** | **Return value registers $v0-$v1** |
| **Stack above the stack pointer** | **Stack above the stack pointer** |

**2.34 [5] <§2.8> Translate function f into MIPS assembly language. If you need to use registers $t0 through $t7, use the lower-numbered registers first. Assume the function declaration for func is “int func(int a, int b);”.**

**The code for function f is as follows:**

**int f(int a, int b, int c, int d ) {**

**return func( func( a, b ), c + d );**

**}**

**Answer:**

f:    add $t0, $a2, $a3       # $t0 = c+d

      addi $sp, $sp, -8       # allocate frame 8 bytes

      sw $ra, 0($sp)          # save return address

      sw $t0, 4($sp)          # save c+d

      jal func                # call func(a,b)

      addi $a0, $v0, 0        # save result of func(a,b) into $a0

      lw $a1, 4($sp)          # $a1 = c+d

      jal func                # call func(func(a,b),c+d)

      lw $ra, 0($sp)          # restore return address

      addi $sp, $sp, 8        # free stack frame

      jr $ra                  # return to caller

**2.35 [5] <§2.8> Can we use the tail-call optimization in this function? If no, explain why not. If yes, what is the difference in the number of executed instructions in f with and without the optimization.**

**Answer:**

Yes, the tail-call optimization is possible in the function by setting the value of $ra register and making the following changes:

“Changing the jump and link to a jump for the 2nd call made in the func”

f: add $t0, $a2, $a3 # $t0 = c+d

addi $sp, $sp, -8 # allocate frame 8 bytes

sw $ra, 0($sp) # save return address

sw $t0, 4($sp) # save c+d

jal func # call func(a,b)

addi $a0, $v0, 0 # save result of func(a,b) into $a0

lw $a1, 4($sp) # $a1 = c+d

lw $ra, 0($sp) # restore return address

addi $sp, $sp, 8 # free stack frame

j func # jump to func. It will run this process (func(a,b),c+d)

and then return # to caller of f.

Instructions by 1 count is reduced with the optimization process as jump return process is not used with the two times execution of the func process.

**2.38 [5] <§2.9> Consider the following code:**

**lbu $t0, 0($t1)**

**sw $t0, 0($t2)**

**Assume that the register $t1 contains the address 0x1000 0000 and the register $t2 contains the address 0x1000 0010. Note the MIPS architecture utilizes big-endian addressing. Assume that the data (in hexadecimal) at address 0x1000 0000 is: 0x1122 3344. What value is stored at the address pointed to by register $t2 ?**

**Answer:**

According to the question,

$t1 (Address)    - 0x10000000

$t2 (Address)     - 0x10000010

Data in hexadecimal at address 0x1000 0000 is: 0x1122 3344. Now, we have to calculate data stored in hexadecimal pointed register by register $t2.

In the code:

 lbu $t0, 0($t1)

sw  $t0, 0($t2)

The value which will be stored at the address pointed by register $t2: 0x00000011 in hexadecimal.