CS2100: Computer Organisation

Tutorial #2: C and MIPS

Answers

Discussion questions D1 and D2 are meant for your own exploration and discussion on LumiNUS Forum. They will not be discussed in class.

D1. **Exploration: C to MIPS**

Go to this website http://reliant.colab.duke.edu/c2mips/ and copy the C code below into the box. Check that the 'Optimization' option is "-O0 – No optimization (default)" and click the 'Submit' button. Scroll down and study the Assembly output.

```
int main(void) {
   int a, b, c;

   a = 3;
   b = 5;
   c = a + b;
   return 0;
}
```

The following is extracted from the Assembly output.

```
    addiu   $sp,$sp,-32
    sw      $fp,28($sp)
    move    $fp,$sp
    li      $2,3                    # 0x3
    sw      $2,8($fp)
    movz    $31,$31,$0
    li      $2,5                    # 0x5
    sw      $2,12($fp)
    lw      $3,8($fp)
    lw      $2,12($fp)
    nop
    addu    $2,$3,$2
    sw      $2,16($fp)
    move    $2,$0
    move    $sp,$fp
    lw      $fp,28($sp)
    addiu   $sp,$sp,32
    j       $31
    nop
```

We covered **sw**, **lw** and **j** in lecture, but not the rest. Find out what they are. (Note: **li** will be used in the labs later and **nop** will be mentioned in the topic on Pipelining. **move**, like **li**, is a pseudo-instruction. **$sp**, **$fp**, **movz**, **addiu**, and **addu** are not in the syllabus.)

D2. For each of the following instructions, indicate if it is valid or not. If not, explain why and suggest a correction. Note that the | in (d) is the bitwise OR operation.

a. `add  $t1, $t2, $t3` # $t3 = $t1 + $t2
b. `addi $t1, $0, 0x25` # $t1 = 0x25
c. `subi $t2, $t1, 3` # $t2 = $t1 - 3
d. `ori  $t3, $t4, 0xAC120000` # $t3 = $t4 | 0xAC120000
e. `sll  $t5, $t2, 0x21` # shift left $t2 33 bits and store in $t5

1. **Bitwise operations**

Find out about the following bitwise operations in C and explain and illustrate each of them with an example.

- | (bitwise OR)
- & (bitwise AND)
- ^ (bitwise XOR)
- ~ (one's complement)
- << (left shift)
- >> (right shift)

You may use the following code template for your illustration. Variables of the data type **`char`** take up 8 bits of memory.

```
#include <stdio.h>

typedef unsigned char byte_t;

void printByte(byte_t);

int main(void) {
   byte_t a, b;

   a = 5;
   b = 22;
   printf("a    = "); printByte(a);    printf("\n");
   printf("b    = "); printByte(b);    printf("\n");
   printf("a|b  = "); printByte(a|b);  printf("\n");
   return 0;
}

void printByte(byte_t x) {
   printf("%c%c%c%c%c%c%c%c",
          (x & 0x80 ? '1': '0'),
          (x & 0x40 ? '1': '0'),
          (x & 0x20 ? '1': '0'),
          (x & 0x10 ? '1': '0'),
          (x & 0x08 ? '1': '0'),
          (x & 0x04 ? '1': '0'),
          (x & 0x02 ? '1': '0'),
          (x & 0x01 ? '1': '0'));
}
```

2. **Bitwise operations**

Your friend is challenging you with a magic trick. He has a deck of 17 cards. All cards are either ♣ or ♠ suit with numbers from 2 to 10. Since there are only 17 cards, it means one of the cards is missing. He says that he can find the missing card by only looping through the deck exactly once! To prove his point, he produces the following code.

```
int findCard(int deck[]) {
  int res, i;
  int count[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

  for (i=0; i<17; i++) {
    count[deck[i]-2]++;
  }

  for (i=2; i<=10; i++) {
    if (count[i-2] == 1) {
      res = i;
    }
  }

  return res;
}
```

As a CS2100 student who has learned about bitwise operator, you say that you can produce a better code! In particular, your friend is using a separate array to count the number of occurrences and you believe you can do without it. To do that, you will need to use one of the bitwise operators from Question 1.

Improve the function findCard(int deck[]) such that it can find the missing card without using a separate array to store the number of occurrences.

Hint: You will need a bitwise operator ⊗ such that $x \otimes x = 0$ and $0 \otimes x = x$.

```
int findCard(int deck[]) {
  int res = 0, i;
  for (i=0; i<17; i++) {
    res = res ^ deck[i];
  }
  return res;
}
```

3. For this question, show the minimal sequence of MIPS instructions needed to translate the given C statement. Use only MIPS instructions covered in MIPS Part 1: Introduction. Assume that variables **a** and **b** are mapped to registers **$s0** and **$s1** respectively.

   (a) `b = a % 16;` // **b** is the remainder of **a** divides by 16

   (b) `b = (a / 8) * 8;` // **b** is the nearest multiple of 8 that is smaller than **a**.
   // Note that integer division is used.

Answers:

(a)

```
andi $s1, $s0, 0xF    # 0xF = 0000 0000 0000 0000 0000 0000 0000 1111
```

By getting the last 4 bits of **a**, it is equivalent to "**a** modulo 16".

(b) A direct translation will result in:

```
srl $t0, $s0, 3    # divide by 8 == right shift by 3 bits
sll $s1, $t0, 3    # multiply by 8 == left shift by 3 bits
```

Alternatively, by setting the last 3 bits of **a** to zeroes, we have the equivalent operation. The trick here is to use XOR operator to set the last 3 bits to zeroes. We first need to get the last 3 bits of **a** while the upper 29 bits are 0s. Using this last 3 bits, we can then use XOR to turn of the last 3 bits of **a** since a ⊗ a = 0 and 0 ⊗ a = a:

```
  0000 0000 0000 0000 0000 0000 0000 0aaa
  aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
----------------------------------------- XOR
  aaaa aaaa aaaa aaaa aaaa aaaa aaaa a000
```

```
andi $t0, $s0, 0x0007 # $t0: 0000 0000 0000 0000 0000 0000 0000 0aaa
xor  $s1, $s0, $t0    # $s1: aaaa aaaa aaaa aaaa aaaa aaaa aaaa a000
```

Another way is to observe that the effect is similar to AND operation with the mask

0xFFFF FFF8 = 1 … 1 1111 1111 1000

which "turns off" the last 3 bits of a value.

However, as this is a 32-bit constant, we are unable to directly specify it as an immediate value as it is restricted to 16-bit signed value. So, one way is to do the two-step loading of 32-bit constant as follows:

```
lui  $t0, 0xFFFF        # load 0xFFFF into the upper 16-bit of $t0
ori  $t0, $t0, 0xFFF8   # now put in the lower 16-bit
and  $s1, $s0, $t0      # AND operation with the mask 0xFFFF FFF8
```

The above results in a 3-line code which is longer than the 2-line solution.

4. The mysterious MIPS code below assumes that **$s0 is a 16-bit binary sequence**, i.e. the upper 16-bits of **$s0** and **$s1** are assumed to be zero at the start of the code.

```
    add  $t0, $s0, $zero
    add  $t1, $s1, $zero
    add  $t3, $zero, $zero
lp: beq  $t1, $zero, e
    andi $t2, $t1, 1
    beq  $t2, $zero, s
    add  $t3, $t3, $t0
s:  sll  $t0, $t0, 1
    srl  $t1, $t1, 1
    j    lp
e:
```

a) Given the initial value of **$s0 = 3** and **$s1 = 5**. What is the value of **$t3** in hexadecimal at the end of execution?

b) Given the initial value of **$s0 = 5** and **$s1 = 13**. How many times do statement "**beq $t2, $zero, s**" results in a branch to the label **s**?

c) Explain the purpose of the code in one sentence.

Answers:

a. **$s0 = 0x0000 000F**

b. 1 (*this occurs when the binary value is 0, since $13_{10} = 1101_2$, it occurs 1 time*)

c. The code multiplies the value of **$s0** with the value of **$s1**.

5. [AY2012/13 Semesters 2 Term Test #2]
Take a look at the following MIPS program and the memory contents. For simplicity, all values (memory addresses and contents) are in decimal.

```
# s1 is initialized to 0
# t0 is initialized to 112

loop: beq  $t0, $zero, exit
      lw   $t1, 0($t0)
      add  $s1, $s1, $t1
      lw   $t0, 4($t0)
      j    loop
exit:
```

Address	Content
100	120
104	132
108	128
112	108
116	124
120	116
124	104
128	100
132	136
136	112

Suppose we execute the code for **3 iterations**, answer the following:

(a) Give the value of register **`$s1`** at the end of the third iteration.

(b) If we want to terminate the loop at the fourth check of **`beq`**, some memory contents need to be modified. Give the address(es) and the modified content(s) to achieve this. You should find the minimum changes required.

(c) What do you think the code does? Give a high-level code fragment that works in a similar way.

Answers:

a) 108 + 104 + 120 = 332

b) Change the content of location 104 from 132 to 0.

c) The code is similar to a "linked list" hopping in a high-level code:

```
while ( ptr != NULL ) {
    sum += ptr->item;
    ptr = ptr->next;
}
```