LAB C
Translating Utility Classes

Perform the following groups of tasks:

LabC1.s

1. Create a directory to hold the files for this lab.

2. Create and run the following two Java classes:

```
public class IntegerMath
{
     public static final int MAX_VALUE = 2147483647;
     public static final byte SIZE = 32;
     private static int count = 0;
}
```

```
public class IntegerMathClient
{
     public static void main(String[] args)
     {
          System.out.print(IntegerMath.MAX_VALUE);
          System.out.print(IntegerMath.SIZE);
     }
}
```

   The first is a utility (i.e. all features are static) which serves as a library. The second is an app that uses the library. We will expand both as we progress through these tasks. We seek to explore how such classes are translated.

3. Launch your favourite editor and create LabC1.s as follows:

```
          .globl   MAX
          .globl   SIZE
          #-----------------------------
          .data
MAX:      .word    2147483647;
SIZE:     .byte    32;
count:    .word    0
          #-----------------------------
          .text
```

   This is a translation of `IntegerMath`: the top section specifies which names are `public`; the data segment declares the static attributes and their initial values; and the text segment (where the methods will reside) is currently empty.

4. Open LabC1.s in SPIM and observe the contents of the data pane. It displays a *hex dump* of memory in *word format* beginning at address at 0x10000000 (i.e. 256 MB):

```
[0x10010000]   0x7fffffff  0x00000020  0x00000000  0x00000000
```

   The bracketed numbers are addresses whereas the remaining numbers are contents of the addresses. Hence, the above dump indicates that if we look at DRAM starting at address 0x10010000 we will see the four shown words.

5. The first word contains `0x7fffffff`, i.e. 32 bits all of which are 1 except the MSb. This is the largest 32-bit signed integer, and we recognize it as the value of the `MAX` field of our utility. We therefore conclude that `MAX` is stored in memory in a block that begins at address 0x10010000; i.e. its four bytes are stored at the four consecutive addresses: 0x10010000, 0x10010001, 0x10010002, and 0x10010003.

6. The second word contains `0x00000020`. Note that SPIM (and most other debugging tools) dumps the content as words *even though* the data may not be. Indeed, the 2nd field, `SIZE`, is a byte. Assuming a little-endian machine (i.e. the LSB is at the lowest address), we see that the byte at address 0x10010004 contains `0x20`, or 32, which is indeed the value of `SIZE`.

7. The next attribute in our class is `count` and it is declared as an `int` (4 bytes or one word). Hence, based on alignment rules, it can only be stored at an address that is divisible by 4. The next free address (after `SIZE`) is 0x10010005 but it is clearly not a multiple of 4. Hence, we must skip the three bytes at 0x10010005, 0x10010006, and 0x10010007 and store `count` at 0x10010008.

8. Explore changing the values of the three attributes; re-ordering their declarations; and adding a few more attributes. Open the modified program in SPIM after each such modification and observe the data segment. Make sure you are comfortable with the hex dump, the word format, and the alignment issue. Report your observations.

9. Explore adding an attribute of type `half` using the **`.half`** directive. This type is 2B wide (and, hence, must align to even addresses) and corresponds to Java's `short` integers (if signed) and `char` ones (if unsigned).

10. We mention, for completeness, two other possible directives: **`.float`** and **`.double`**. They correspond to Java's `float` and `double` real numbers.

LabC1Client.s

11. Create the program LabC1Client.s that translates the `IntegerMathClient` app. It is intended to use the services of LabC1.s created above. Note that even though the two programs are written (and compiled) independently, they will be linked / loaded when the app is launched.

12. Our program needs to access data in memory. To do that, it must use the load/store family of the MIPS instruction set. The family contains instructions to load and store words (**`lw`** and **`sw`**), half words (**`lh`** and **`sh`**), and bytes (**`lb`** and **`sb`**).

13. Write the client as shown below:

```
            .text
main:       lw      $a0, MAX($0)
            addi    $v0, $0, 1
            syscall

            lbu     $a0, SIZE($0)
            addi    $v0, $0, 1
            syscall

            jr      $ra
```

    The load / store instructions use an array-like syntax to address memory operands: you provide an immediate (a label or a hard-coded constant) followed by a register between two parentheses. The immediate acts as a *base* address and the register plays the role of an *index*. The CPU adds the immediate to the content of the register and interprets the sum as the address of the data.

14. Launch SPIM and open LabC1.s and then LabC1Client.s. When you open two (or more) programs without clearing the contents in between the open's, SPIM consolidates the programs by merging their text and data segments. As a result, you will see only one text and one data segment in the end.

15. Run the program. Note that regardless of how many programs are loaded into SPIM, only one of them has a **`main`** label, and it is that one that starts when we run.

16. Report the output and verify it is what you expect.

17. To improve readability, modify LabC1Client so it prints a new line in between its two outputs; i.e. as if we replaced the first `print` in the Java source with `println`.

LabC2 and its Client

18. Save LabC1.s as LabC2.s but replace `32` with `-32` as the value of **`SIZE`**. Also save LabC1Client.s as LabC2Client.s.

19. Load LabC2 and its client in SPIM and run. The output is not what one would expect from the Java perspective. Explain.

20. Unlike all the other instructions in the load / store family, **`lb`** and **`lh`** store a piece of data represented in less than 32 bits in a register that is 32-bit wide. Because of this, the issue of zero versus sign extension comes into play. Fix the bug in LabC2Client based on your understanding of this issue and the *u* suffix. Explain how you fix it.

21. Examine the text segment of LabC2Client, and in particular, the first **`lw`** at (or near) address [0x0040002c]. The text pane shows that SPIM did not handle this as a single instruction but rather as two:

```
lui     $1, 4097
lw      $4, 0($1)
```

Why is that? We found earlier that `MAX` is stored at address 0x10010000. We therefore expect SPIM to simply replace the symbolic reference to MAX with 0x10010000; i.e. replace:

```
lw      $a0, MAX($0)
```

with (given that register **`$a0`** is the nickname of **`$4`**):

```
lw      $4, 0x10010000($0)
```

Explain why the above did not occur and why two instructions were used instead.

22. Recall that the immediate in any (non-jump) instruction can only be 16-bit wide. If a larger immediate must be used then we must break into two pieces and arrange to assemble them back together. Does this explain SPIM's action?

23. Note that SPIM used register **`$1`** for the replacement. Could it have used any other register? Why?

LabC3 and its Client

24. Save LabC2 and its client as LabC3 and its client.

25. We want to modify the client so that it functions the same as before (and produces the same output) yet it does not refer to the `SIZE` symbol. This sounds impossible (because the program would end up outputting the value of a variable without ever referring to that variable) but it is doable because we know where the attributes are stored in memory. Specifically, remove the `SIZE` reference in:

```
lb      $a0, SIZE($0)
```

26. You know that `SIZE` is stored 4 bytes away from `MAX`. We also know that the CPU computes addresses by adding the immediate and the register. These observations lead us to explore replacing the above statement with something like this:

```
addi    $t0, $0, 4
lb      $a0, MAX($t0)
```

27. Make the needed changes, load the new program pair, and run. Did you get the expected output?

28. Notice that the above technique (of doing arithmetic on addresses) allows us not only to access `SIZE` through `MAX` but also to access `count`. This works despite the fact that `count` was declared as `private`. But since Java does not support any kind of address arithmetic (such as adding or subtracting 4), privacy of `count` is assured.

29. Explore loading a word from an address one byte before `SIZE`. Based on our earlier findings, do you expect this word to contain `0x0000ff20`? Carry out the suggested change and run. Explain what you observe.

LabC4 and its Client

30. We now seek to translate the following pair of classes:

```
public class IntegerMath
{
     public static final int MAX_VALUE = 2147483647;
     public static final byte SIZE = 32;
     private static int count = 0;

     // Accessor for the count attribute
     public static int getCount()
     {
          int result = IntegerMath.count;
          return result;
     }
}
```

```
public class IntegerMathClient
{
     public static void main(String[] args)
     {
          System.out.print(IntegerMath.MAX_VALUE);
          System.out.print(IntegerMath.SIZE);
          System.out.println(IntegerMath.getCount());
     }
}
```

31. Save LabC1.s as LabC4.s. Keep the data segment and its associated **globl** directives as before and then append with the following:

```
            .globl getCount
            #-----------------------------
            .text
getCount:   #-----------------------------
            lw      $v0, count($0)
            jr      $ra
```

32. Implementing a method is not much different from implementing a **main**. The last statement returns to the caller by jumping to the address stored in register **$ra**. This implies that the caller must have stored (in **$ra**) the address of the instruction that immediately follows the method invocation.

33. Methods use the **a** registers to receive parameters passed to them and **v** registers to return data to the caller. This is why the above accessor uses **$v0** for its return

34. Save LabC1Client.s as LabC4Client.s and add to it:

```
jal      getCount
add      $a0, $0, $v0
addi     $v0, $0, 1
syscall
```

35. The **jal** (**J**ump **A**nd **L**ink) instruction is similar to **j** except it stores **PC** (the program counter register) in **$ra** prior to jumping. Since **PC** holds the address of the instruction that follows the currently executing one, the method invoked by **jal** will return to the statement that follows the call by using **jr $ra**.

36. Note that even the main **app** ends with a jump to the content of **$ra**. This is because an **app** is in reality a method invoked by the O/S (which is SPIM in our case).

37. Load the LabC4 pair and run. The system will hang!

38. To understand what triggered the infinite loop, restart SPIM and re-open the LabC4 pair but, this time, step through the program one instruction at a time. As you do, you will see why an infinite loop was created.

39. Since all methods rely on **$ra** to remember the return address, we must obviously store the content of this register in some safe place before we invoke any method. This storage is typically done at the beginning of each method. The safe place is known as the *stack*. You can think of it as a system-provided array in memory with base address stored in **$sp**, the **S**tack **P**ointer register. The only peculiar aspect of the stack is that it grows *backward*, i.e. toward smaller addresses.

40. Add the following instructions to the beginning of the client:

```
main:   sw      $ra, 0($sp)
        addi    $sp, $sp, -4
```

    The first instruction saves **$ra** at the location pointed at by **$sp** while the second decrements **$sp** so that any further storage to it would not overwrite **$ra**. We refer to this pair of instructions as *pushing **$ra** into the stack*.

41. Add the following instructions to the end of the client:

```
        addi    $sp, $sp, 4
        lw      $ra, 0($sp)
        jr      $ra
```

    The first two instructions serve to *pop the stack into **$ra*** while the third is the usual return instruction.

42. Now that **$ra** is properly saved and restored, you should be able to run the LabC4 pair without problems.

43. Note that you could also implement the push / pop mechanism within the **getCount** method; indeed, some compilers implement it in each and every method. But when a method does not itself invoke any other method (aka a *leaf* method), there is no need to save **$ra** since no instruction changes it.

44. The stack is used not only to save **$ra** but also to save other registers that the caller may need. We shall see an example of this shortly.

LabC5 and its Client

45. We now seek to translate the addition of a mutator to the utility:

```
// Mutator for the count attribute
public static void setCount(int count)
{
     IntegerMath.count = count;
}
```

    And we also need a corresponding test in the app:

```
System.out.println(IntegerMath.getCount());
int a = new Scanner(System.in).nextInt();
IntegerMath.setCount(a);
System.out.println(IntegerMath.getCount());
```

46. Save LabC4 as LabC5 and add the new method. Since the mutator has receives one parameter, it should expect it to be stored in **`$a0`**.

47. Save LabC4Client as LabC5Client and add a translation of the above sequence.

48. Run the LabC5 pair and report your result.

49. In order to make your client easier to read, you may want to add a method named **`println`** to LabC5. This way, whenever the client needs to print a new line, it would simply use:

```
jal      println
```

LabC6 and its Client

50. We now seek to add to our utility a method similar to one in Java's `Integer` class. Moreover we like to see the **`count`** attribute acting as a hit counter for the method; i.e. the method should increment **`count`** whenever it is invoked:

```
public static int signum(int i)
{
     int result;
     if (i < 0)
     {
               result = -1;
     } else if (i == 0)
     {
               result = 0;
     } else
     {
               result = 1;
     }
     IntegerMath.count++;
     return result;
}
```

51. Save LabC5 as LabC6 and implement **signum**. Note that your method may only use the **t** registers for its local variables; the **a** registers for parameters, and the **v** registers for returns. If you must make a change to any other register, you must preserve its value. See the notes about caller/callee saving convention at the end of this Lab.

52. Save LabC5Client as LabC6Client and add to it a test that reads an integer from the user and then print its **signum** and the new value of **count**.

53. Run your pair of programs. If the output is not what you expect, set a breakpoint or step through your code until you identify the problem and correct it. Report the results.

LabC7 and its Client

54. We now seek to add the following to our utility:

```
public static boolean isPrime(int i)
{
    boolean result = true;
    for (int candidate = 2; result && candidate < i; candidate++)
    {
        result = !IntegerMath.isFactor(i, candidate);
    }
    return result;
}

private static boolean isFactor(int n, int factor)
{
    boolean result = (n % factor) == 0;
    return result;
}
```

55. Save LabC6 as LabC7 and implement **isPrime**. Note that you must treat `boolean` as an integer with preset values for `true` and `false`. The private **isFactor**, for example, would return zero in **$v0** for `false` and the negation of that for `true`.

56. Note that since **isPrime** invokes **isFactor**, it cannot assume that the **t** registers will not be changed by **isFactor**, and hence, must push whatever it needs before the call pop after. In other words, when you write **isPrime** do not assume that you know the internals of **isFactor**; instead, treat it as a black box.

57. Save LabC6Client as LabC7Client and add to it a test that reads an integer from the user and determines if it is prime by invoking **isPrime**.

58. Run your pair of programs. If the output is not what you expect, set a breakpoint or step through your code until you identify the problem and correct it. Report the results.

LabC8 and its Client

59. We now seek to add to our utility a method that prints an integer in decimal vertically (one digit per line) starting with its most significant digit. Specifically, we need to translate the following recursive method:

```
public static void printVertical(int n)
{
     if (n > 0)
     {
         printVertical(n / 10);
         System.out.println(n % 10);
     }
}
```

60. Save the LabC7 pair as LabC8 and implement this method along with a test. Report the results.

LAB C
Notes

- As a caller, you can safely assume that the contents of all your registers will not be changed by the methods you call except for: **`$a0-$a3`** (used to pass parameters and can conceivably be changed by the method), **`$v0-$v1`** (used for the return(s) of the method and will obviously change if the method is not `void`), and **`$t0-$t9`** (used as scratch registers by the method and may thus change).

- As a callee (a method), you must ensure that the above caller's contract will be met. Hence, if you plan to change any register other than the **`a`**'s, **`v`**'s, and **`t`**'s, then save their contents at your prologue and restore them prior to returning.

- Since a non-leaf method is both a caller and a callee, it must be careful in its usage of the **`t`**'s. It does not need to save them for the caller's sake, but it may need to do so for its own sake. Data stored in them must be saved and restored before and after every invocation the method makes.

- The very act of invoking a method automatically changes **`$ra`**. Hence, this register must be saved by any non-leaf method, i.e. one that invokes another.

- The saving of registers, by caller or callee, is always done on the stack, a last-in-first out data structure that grows toward smaller addresses and whose "last" location, or top, is pointed at by **`$sp`**.

- Don't "play it safe" by pushing all registers (or all used registers) on the stack before each call and popping them after. Such a conservative approach may have a severe impact on performance. For example, a leaf method should only push the **`s`** registers it uses. No other register needs to be preserved.

- Since we assign a register to each variable declared in a method, all local variables declared in a Java method end up being effectively allocated on the stack. And since we pop all these values off the stack at the end of the method, the variables cease to exist. This means the lifetime of the method's variables is as long as the method.