**Massachusetts Institute of Technology**
**Department of Electrical Engineering and Computer Science**

6.035, Fall 2000 Handout 20 – Codegen Practice Quiz Solutions Monday, October 16

---

1. Convert the following Decaf into MIPS assembly code, including an appropriate prologue and epilogue:

```
int triangle(int num)
{
        int cur;

        while (num > 0) {
               cur = cur + num;
               num = num - 1;
        }

        return cur;
}
```

**Solution:**

```
_L3triangle:
        sw $fp,-4($sp)
        move $fp,$sp
        sw $ra,-8($sp)
        sw $gp,-12($sp)
        sub $sp,$sp,16
_L3triangle_start:
_L8_w_eval:
        bgt $a1,0,_L2Lblock
        b _L7_w_lpad
_L2Lblock:
        add $t9,$t9,$a1
        sub $a1,$a1,1
        b _L8_w_eval
_L7_w_lpad:
        move $v0,$t9
        b _L3triangle_epilogue
_L3triangle_epilogue:
        move $sp,$fp
        lw $fp,-4($sp)
        lw $ra,-8($sp)
        lw $gp,-12($sp)
        jr $ra
_L3triangle_end:
```

2. The code in the previous question is a *leaf* procedure, because it doesn't call any other procedures. In the space below, write an version of the procedure that uses an optimized calling convention. Note that you shouldn't change the body at all; we're looking for just the specific optimizations that take advantage of it being a leaf procedure. (We've reproduced the code here for your convenience.)

```
myproc:
        sw $fp,-4($sp)
        move $fp,$sp
        sw $ra,-8($sp)
        sw $gp,-12($sp)
        sub $sp,$sp,16
        sw $a0,0($fp)
        sw $a1,4($fp)
        sw $a2,8($fp)
        sw $a3,12($fp)
        lw $v0,4($fp)
        lw $v1,4($fp)
        mulo $v0,$v0,$v1
        la $v1,-16($fp)
        sw $v0,0($v1)
        lw $v0,-16($fp)
        move $sp,$fp
        lw $fp,-4($sp)
        lw $ra,-8($sp)
        lw $gp,-12($sp)
        jr $ra
```

**Solution:**

```
myproc:
        sw $fp,-4($sp)
        move $fp,$sp
        sub $sp,$sp,16
        sw $a1,4($fp)
        lw $v0,4($fp)
        lw $v1,4($fp)
        mulo $v0,$v0,$v1
        la $v1,-16($fp)
        sw $v0,0($v1)
        lw $v0,-16($fp)
        move $sp,$fp
        lw $fp,-4($sp)
        jr $ra
```

3. In Decaf simple arguments are passed *by value*, meaning that if a called procedure changes the values of its arguments, those changes are not visible to a calling procedure. Sometimes the programmer really wants to be able to change the value. C gets around this by allowing the programmer to pass a *pointer*, or address, of the variable and having the called procedure use that address. This is klunky though, since the programmer has to be explicit about using the address, and also exposes all sorts of possible programming errors.

   Instead, we want to extend Decaf to pass arguments *by reference*, meaning changes are visible to calling procedures. We will do this by appending an argument with `&` in the procedure declaration. Below is an example program that makes use of this:

```
void sort_in_place(int &a, int &b)
{
        int tmp;

        if (a > b) {
                tmp = a;
                a = b;
                b = tmp;
        }
}
```

   Without pass by reference, this procedure wouldn't do anything, but with it, it will swap `a` and `b` if `a` is larger.

   It turns out we can do this in a Decaf compiler with very few changes, all inside the code generator. Suppose that every argument parameter for each function is annotated with a flag saying whether or not it is passed by value or reference. On this page, explain in words what changes you need to make to the compiler to get call-by-reference semantics. Your changes do not need to be comprehense. On the next page is the compiler's output if `a` and `b` weren't flagged as call-by-reference. Indicate in the listing what additional code the compiler would have to output to get call-by-reference behavior.

   **Solution:**

   We need to add something to the calling convention. When a called procedure returns, for each parameter that is passed by reference, we add code to copy the value in the correct argument slot (either `$a`*n* or a stack slot) into that variable's location in the calling procedure's stack frame. If the argument was an immediate, do nothing.

```
    # method: sort_in_place
_L4sort_in_place:
        sw $fp,-4($sp)
        move $fp,$sp
        sw $ra,-8($sp)
        sw $gp,-12($sp)
        sub $sp,$sp,16
_L4sort_in_place_start:
        bgt $a1,$a2,_L2Lblock
        b _L3Lblock
_L2Lblock:
        move $t9,$a1
        move $a1,$a2
        move $a2,$t9
_L3Lblock:
_L8_if_out:
_L4sort_in_place_epilogue:
        move $sp,$fp
        lw $fp,-4($sp)
        lw $ra,-8($sp)
        lw $gp,-12($sp)
        jr $ra
    # method: main
_L5main:
        sw $fp,-4($sp)
        move $fp,$sp
        sw $ra,-8($sp)
        sw $gp,-12($sp)
        sub $sp,$sp,12
_L5main_start:
        sw $a0,0($fp)
        li $a2,1
        li $a1,2
        sub $sp,$sp,16
        lw $v0,0($a0)
        lw $v0,0($v0)
        jal $v0
    # here, copy the values back
    # except the values passed in are immediates, so nothing to do!
        add $sp,$sp,16
        lw $a0,0($fp)
_L5main_epilogue:
        move $sp,$fp
        lw $fp,-4($sp)
        lw $ra,-8($sp)
        lw $gp,-12($sp)
        jr $ra
```

4. Losing the frame pointer.

   Here's an Espresso program,

```
class Program {
    void bar()
    {
        return;
    }

    void baz()
    {
        bar();
    }

    void main()
    {
        baz();
    }
}
```

   and here is the code generated for `baz()`:

```
_baz:
        sw $fp,-4($sp)
        move $fp,$sp
        sw $ra,-8($sp)
        sw $gp,-12($sp)
        sub $sp,$sp,12
    # code for baz
_baz_start:
        sw $a0,0($fp)
        sub $sp,$sp,16
        lw $v0,0($a0)
        lw $v0,0($v0)
    # call bar
        jal $v0
        add $sp,$sp,16
        lw $a0,0($fp)
    # epilogue for baz
_baz_epilogue:
        move $sp,$fp
        lw $fp,-4($sp)
        lw $ra,-8($sp)
        lw $gp,-12($sp)
        jr $ra
_baz_end:
```

As we've described in class, `$fp` is used to keep track of where on the stack our data is saved, so that we don't use `$sp`. It turns out this isn't necessary, though.

(a) Short answer: why don't we need `$fp`. Why do we use it anyway?

(b) Edit the listing above to eliminate the use of `$fp`, using `$sp` to refer to items on the stack instead.

(c) List two advantages of not using `$fp`.

**Solution:**

(a) Short answer: why don't we need `$fp`. Why do we use it anyway?

We don't need `$fp` because we know the value of `$sp` at compile time, and can just reference off that. We use it because it's a bit simpler to use `$fp`, and because in the case where you can change the value of `$sp` at runtime, you need it.

(b) Edit the listing above to eliminate the use of `$fp`, using `$sp` to refer to items on the stack instead.

```
_baz:
        # sw $fp,-4($sp)
        # move $fp,$sp
        sw $ra,-8($sp)
        sw $gp,-12($sp)
        sub $sp,$sp,12
    # code for baz
_baz_start:
        sw $a0,0+12($sp)
        sub $sp,$sp,16
        lw $v0,0($a0)
        lw $v0,0($v0)
    # call bar
        jal $v0
        add $sp,$sp,16
        lw $a0,0+12($sp)
    # epilogue for baz
_baz_epilogue:
        # move $sp,$fp
        # lw $fp,-4($sp)
        lw $ra,-8($sp)
        lw $gp,-12($sp)
        jr $ra
_baz_end:
```

(c) List two advantages of not using `$fp`.

i. It frees up `$fp` for use as another register (`$s8`).

ii. We save the instructions to save `$fp` and update it with a new value.