CS 352 – Compilers: Principles and Practice
Final Examination, 12/17/05

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (your grade will be the percentage of your answers that are correct).

This exam is **closed book, closed notes**. You may *not* refer to any book or other materials.

You have **two hours** to complete all nine (9) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature:

Assume we are compiling to a target architecture similar to the PowerPC, as we did in the project, but having only the following general-purpose registers:

- `$t0`: a special-purpose caller-save register
- `$sp`: the stack-pointer
- `$a0`: a caller-saved argument/result register
- `$a1`: a caller-saved argument register
- `$s0`: a callee-save register

Both `$sp` and `$t0` are special-purpose registers that cannot be allocated to temporaries by the register allocator.

```
class Integer {
    int value;
    int mul (int i) {
        int result = 0;
        if (i != 0) result = this.value + this.mul(i-1);
        return result;
    }
    public static void main (String[] args) {
        Integer x = new Integer();
        x.value = 9;
        int y = x.mul(3);
    }
}
```

1. (Semantics; 5%) What is the value of the variable `y` computed by the `main` method?

   Answer:

   27

2. (Translation; 15%) MJ translates the `mul` method into the following canonicalized trees:

```
MOVE(TEMP this, TEMP $a0)
MOVE(TEMP i, TEMP $a1)
MOVE(TEMP result, CONST 0)
BNE(TEMP i, CONST 0, L3, L2)
LABEL L3
MOVE(TEMP result,
     ADD(MEM(TEMP this, 0),
         CALL(MEM(ADD(MEM(ADD(TEMP this,
                              CONST -4)),
                      CONST 0)),
              TEMP this,
              SUB(TEMP i, CONST 1))))
JUMP(NAME L2)
LABEL L2
MOVE(TEMP $a0, TEMP result)
```

For these intermediate trees, MJ generates the following PowerPC instructions:

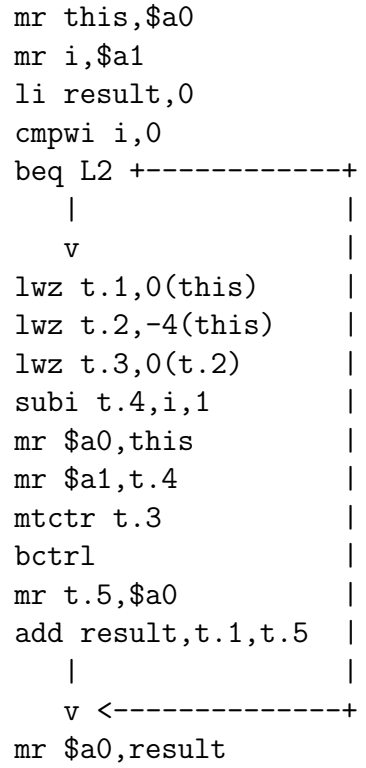
```
    mr this,$a0
    mr i,$a1
    li result,0
    cmpwi i,0
    beq L2
L3:
    lwz t.1,0(this)
    lwz t.2,-4(this)
    lwz t.3,0(t.2)
    subi t.4,i,1
    mr $a0,this
    mr $a1,t.4
    mtctr t.3
    bctrl
    mr t.5,$a0
    add result,t.1,t.5
L2:
    mr $a0,result
```

Given that these instructions have been generated using "maximal munch", circle the tiles (in the trees above) and number the tiles in the order that they are "munched" (*ie*, the order that instructions are emitted for the tiles). Then circle the instructions (above) corresponding to each tile and mark them with the number of that tile.

3. (Control flow analysis; 5%) Identify the *basic blocks* in this program (given again below), and draw its control flow graph (CFG), having nodes which are the basic blocks and edges representing flow of control between the basic blocks. Remember that a *call* is not treated as a branch in the CFG.

```
    mr this,$a0
    mr i,$a1
    li result,0
    cmpwi i,0
    beq L2
L3:
    lwz t.1,0(this)
    lwz t.2,-4(this)
    lwz t.3,0(t.2)
    subi t.4,i,1
    mr $a0,this
    mr $a1,t.4
    mtctr t.3
    bctrl
    mr t.5,$a0
    add result,t.1,t.5
L2:
    mr $a0,result
```

Answer:



4. (Liveness analysis; 15%) Compute *liveness* information for each instruction in the program as described in class, by tracing uses back to definitions, being careful to propagate along both edges at a merge point in the CFG. Remember that a call instruction uses its argument registers and defines all argument/result registers. I have already labeled the variables/registers *used* and *defined* by each instruction. Moves are marked as "`def <= use`", while other instructions are marked as "`def <- use`".

   To get you started I have already traced uses back to definitions for the first four instructions for you, so liveness information on edges leading to/from those instructions has been computed for those variables only on those instructions (liveness information at those instructions for other variables is incomplete)! You should complete the rest of the analysis.

```
                        DEF <- USE            LIVE
                                              $a0 $a1
    mr this,$a0         this <= $a0
                                              $a1
    mr i,$a1            i <= $a1
                                              i
    li result,0         result <-
                                              i
    cmpwi i,0           <- i

    beq L2              <- : goto L2 L3
L3:
    lwz t.1,0(this)     t.1 <- this

    lwz t.2,-4(this)    t.2 <- this

    lwz t.3,0(t.2)      t.3 <- t.2

    subi t.4,i,1        t.4 <- i

    mr $a0,this         $a0 <= this

    mr $a1,t.4          $a1 <= t.4

    mtctr t.3           <- t.3

    bctrl               $a0 $a1 <- $a0 $a1

    mr t.5,$a0          t.5 <= $a0

    add result,t.1,t.5  result <- t.1 t.5
L2:
    mr $a0,result       $a0 <= result
                                              $a0
```

Answer:

```
                          DEF <- USE            LIVE
                                                $a0 $a1
    mr this,$a0           this <= $a0
                                                $a1 this
    mr i,$a1              i <= $a1
                                                i this
    li result,0           result <-
                                                i this result
    cmpwi i,0             <- i
                                                i this result
    beq L2                <- : goto L2 L3
L3:                                             i this
    lwz t.1,0(this)       t.1 <- this
                                                i this t.1
    lwz t.2,-4(this)      t.2 <- this
                                                t.2 i this t.1
    lwz t.3,0(t.2)        t.3 <- t.2
                                                i this t.3 t.1
    subi t.4,i,1          t.4 <- i
                                                this t.4 t.3 t.1
    mr $a0,this           $a0 <= this
                                                t.4 t.3 $a0 t.1
    mr $a1,t.4            $a1 <= t.4
                                                t.3 $a0 $a1 t.1
    mtctr t.3             <- t.3
                                                $a0 $a1 t.1
    bctrl                 $a0 $a1 <- $a0 $a1
                                                $a0 t.1
    mr t.5,$a0            $a0 $a1 <- $a0 $a1
                                                t.1 t.5
    add result,t.1,t.5    result <- t.1 t.5
L2:                                             result
    mr $a0,result         $a0 <= result
                                                $a0
```

5. (Interference; 15%) Fill in the following adjacency table representing the interference graph for the program; an entry in the table should contain an $\times$ if the variable in the left column interferes with the corresponding variable/register in the top row. Since machine registers are pre-colored, we choose to omit adjacency information for them. Naturally, you must still record if a non-precolored node (variable) interferes with a pre-colored node (register); the columns for pre-colored nodes are there for that purpose.

   Also, record the *unconstrained* move-related nodes in the table by placing an $\circ$ in any empty entry where the variable in the left column is the source or target of any move involving the variable/register in the top row. **Nodes that are move-related should not interfere if their live ranges overlap only starting at the move and neither is subsequently redefined**.

	`$a0`	`$a1`	`$s0`	`this`	`i`	`result`	`t.1`	`t.2`	`t.3`	`t.4`	`t.5`
`this`											
`i`											
`result`											
`t.1`											
`t.2`											
`t.3`											
`t.4`											
`t.5`											

   Answer:

	`$a0`	`$a1`	`$s0`	`this`	`i`	`result`	`t.1`	`t.2`	`t.3`	`t.4`	`t.5`
`this`	$\circ$	$\times$			$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	
`i`		$\circ$		$\times$		$\times$	$\times$	$\times$	$\times$		
`result`	$\circ$			$\times$	$\times$						
`t.1`	$\times$	$\times$		$\times$	$\times$			$\times$	$\times$	$\times$	$\times$
`t.2`				$\times$	$\times$		$\times$				
`t.3`	$\times$	$\times$		$\times$	$\times$		$\times$			$\times$	
`t.4`	$\times$	$\circ$		$\times$			$\times$		$\times$		
`t.5`	$\circ$						$\times$				

6. (Register allocation; 5%) The graph coloring register allocator decides it must spill `t.1` in favor of keeping `t.3` in a register for the call. Why can `t.1` *not* receive a register if `t.3` does?

   Answer:

   Because the only register for `t.3` is `$s0`, `t.1` conflicts with `t.3`, and also conflicts with the only other registers `$a0` and `$a1`.

7. (Spilling; 10%) Given that `t.1` must spill, rewrite the program so that the value of `t.1` resides at byte offset 20 from the stack pointer (`$sp`). You should use new temporaries `t.6` in place of `t.1` where `t.1` is *defined*, and `t.7` in place of `t.1` where `t.1` is *used*, storing/loading to/from memory as necessary. Your program should now contain no uses or definitions of `t.1`.

   Answer:

```
        mr this,$a0
        mr i,$a1
        li result,0
        cmpwi i,0
        beq L2
    L3:
        lwz t.6,0(this)
        stw t.6,20($sp)
        lwz t.2,-4(this)
        lwz t.3,0(t.2)
        subi t.4,i,1
        mr $a0,this
        mr $a1,t.4
        mtctr t.3
        bctrl
        mr t.5,$a0
        lwz t.7,20($sp)
        add result,t.7,t.5
    L2:
        mr $a0,result
```

8. (Code emission; 15%) The register allocator now determines the following register assignments:

```
this->$a0
i->$a1
result->$s0
t.6->$s0
t.2->$s0
t.3->$s0
t.4->$a1
t.5->$a0
t.7->$a1
```

Write the resulting assembly code program, eliminating any redundant move instructions, in between the method prologue/epilogue below:

```
mflr $t0                      # load return address from link register
stw $t0,8($sp)                # store it at $sp+8
stmw $s0,-4($sp)              # save callee-save $s0, since we use it
stwu $sp,-framesize($sp)      # allocate stack frame
```

Answer:

```
#   mr $a0,$a0
#   mr $a1,$a1
    li $s0,0
    cmpwi $a1,0
    beq L2
L3:
    lwz $s0,0($a0)
    stw $s0,20($sp)
    lwz $s0,-4($a0)
    lwz $s0,0($s0)
    subi $a1,$a1,1
#   mr $a0,$a0
#   mr $a1,$a1
    mtctr $s0
    bctrl
#   mr $a0,$a0
    lwz $a1,20($sp)
    add $s0,$a1,$a0
L2:
    mr $a0,$s0

addi $sp,$sp,framesize      # deallocate stack frame
```

```
lmw $s0,-4($sp)          # restore callee-save $s0, since we use it
lwz $t0,8($sp)           # load return address
mtlr $t0                 # move it to the link register
blr                      # return (branch to link register)
```

9. (Runtime management; 15%) Given what you now know about the code for method `mul`, draw a diagram of the stack for the program *at the point* where `mul` is about to return the value 0. Show *all* memory-resident local and heap variables (*ie*, that are not allocated to registers), including both integer values and references to the heap (as well as the object to which each reference variable refers).

   The original source code is repeated again here, for your convenience:

```
class Integer {
    int value;
    int mul (int i) {
        int result = 0;
        if (i != 0) result = this.value + this.mul(i-1);
        return result;
    }
    public static void main (String[] args) {
        Integer x = new Integer();
        x.value = 9;
        int y = x.mul(3);
    }
}
```

   Answer:

   In the following we show `args`, `x` and `y` in memory though they probably are in registers. You also need not show the value of `$s0` since it is a register.