Quiz 4

EECS665 - Compiler Construction
Spring 2023

Name: ____________________ Student ID: ______________

Do Not Open Until Instructed!

Before the Quiz starts:

- Read all of the instructions on this page
- Write your name and student ID on this page
- Retrieve your page of notes and writing materials
- Put all other materials away and silence your devices

After the Quiz starts:

- Write your student ID (**not** your name) on all subsequent pages
- If you feel a question is wrong or impossible, notify course staff.
- Announcements / corrections will appear on the projector
- Turn in all your related paper when finished, including:
  - the provided quiz itself
  - provided reference pages
  - provided scratch paper
- You may leave when done (no new material will be presented).
- Work quickly, move on if you are stuck.

Score: ☐/50

Question 1: ☐/10

Question 2: ☐/10

Question 3: ☐/10

Question 4: ☐/10

Question 5: ☐/10

Feel free to draw something summery in the box below

Question 1 (10 Points)

Write a valid JEFF program that, when compiled with the basic compilation method described in class, would benefit from an instruction strength reduction and dead code elimination. Point out what the instruction strength reduction is and point out what dead code would be eliminated.

Question 2 (10 Points)

Student ID: ____________________

Complete All Parts

Part I

Some compilers write specially-optimized target code for *leaf functions*, i.e. functions that have no callees. What optimizations might be possible with such functions?

Part II

Imagine the System V ABI was changed such that ALL registers were caller-saved (i.e. "volatile" registers). What changes would the compiler have to make when compiling code?

Question 3 (10 Points)

Student ID: ______________________

Complete All Parts

Part I

Describe the purpose of the assembler, linker, and loader.

Part II

Draw a single basic block where there are three variables, but the interference-graph is 2-colorable. Also draw the interference graph.

Question 4 (10 Points)

Student ID: ____________________

Draw the control-flow graph of the following 3AC snippet

```
main:       enter main
            RECEIVE [a]
            RECEIVE [b]
            [tmp0] := [a] LT64 [b]
            IFZ [tmp0] GOTO lbl_1
            [b] := [b] ADD64 1
lbl_2:      nop
            [tmp1] := [a] LT64 [b]
            IFZ [tmp1] GOTO lbl_3
            [a] := [a] ADD64 1
            goto lbl_2
lbl_3:      nop
            REPORT [a]
lbl_1:      nop
            REPORT [b]
lbl_0:      leave main
```

QUESTION 5 (10 POINTS)

Student ID: ____________________

Complete All Parts

PART I:

Between mark-and-sweep and reference counting heap management, which version uses more memory (assume no memory leaks)? Explain your answer and illustrate with an example.

PART II:

Consider a JEFF program has 1 global variable and 2 functions, foo and main. Assume foo has two arguments, declares two local variables, and requires one temporary variable. Draw the activation record for foo upon entry (i.e. after the function prologue). Label each memory slot and justify why each slot is necesary and placed in the AR.

X64 Reference

- movq $\langle \text{opd}_1 \rangle$ $\langle \text{opd}_2 \rangle$ - *Copy the 8-byte value of opd_1 into opd_2*
- addq $\langle \text{opd}_1 \rangle$ $\langle \text{opd}_2 \rangle$ - *Put the result of $opd_2 + opd_1$ into opd_2*
- subq $\langle \text{opd}_1 \rangle$ $\langle \text{opd}_2 \rangle$ - *Put the result of $opd_2 - opd_1$ into opd_2*
- imulq $\langle \text{opd}_1 \rangle$ - *Put the result of %rax * opd_1 into the octoword %rdx:%rax*
- callq $\langle \text{lbl} \rangle$ - *Stack (push) the next instruction address, move %rip to the address $\langle lbl \rangle$*
- retq - *Unstack (pop) into %rip*
- xorq $\langle \text{opd}_1 \rangle$ $\langle \text{opd}_2 \rangle$ - *Put the result of $\langle opd_2 \rangle$ XOR $\langle opd_1 \rangle$ into opd_2*
- negq $\langle \text{opd} \rangle$ - *Put the 2's complement negation of $\langle opd \rangle$ into $\langle opd \rangle$*
- notq $\langle \text{opd} \rangle$ - *Flip all bits of $\langle opd \rangle$*
- jmp $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$*
- cmpq $\langle \text{opd}_1 \rangle$ $\langle \text{opd}_2 \rangle$ - *Set rflags according to $\langle opd_2 \rangle$ - $\langle opd_1 \rangle$*
- je $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$ if rflags indicates a $=$ relation on prior operands*
- jne $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$ if rflags indicates a $\neq$ relation on prior operands*
- jge $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$ if rflags indicates a $\geq$ relation on prior operands*
- jl $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$ if rflags indicates a $<$ relation on prior operands*
- jg $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$ if rflags indicates a $>$ relation on prior operands*
- jle $\langle \text{lbl} \rangle$ - *jump to $\langle lbl \rangle$ if rflags indicates a $\leq$ relation on prior operands*
- sete $\langle \text{opd} \rangle$ - *Set opd to be 1 if rflags indicates that the last compare operation had equal operands, 0 otherwise. $\langle opd \rangle$ must be a 1-byte register.*
- setg $\langle \text{opd} \rangle$ - *Set opd to be 0 if rflags indicates that the last compare operation had an opd_2 less than or equal to its opd_1, 1 otherwise. $\langle opd \rangle$ must be a 1-byte register.*
- setle $\langle \text{opd} \rangle$ - *Set opd to be 0 if rflags indicates that the last compare operation had an opd_2 greater than its opd_1, 0 otherwise. $\langle opd \rangle$ must be a 1-byte register.*

Registers

General-purpose registers

- `%rax - %rdx` (lowest 1 byte is %al - %dl)
- `%r8 - %r15` (lowest 1 byte is %r8b - %r15b)
- `%rsi` (lowest 1 byte is %sil)
- `%rdi` (lowest 1 byte is %dil)
- `%rsp` - stack pointer
- `%rbp` - base pointer

Select special-purpose registers

- `%rflags` status flags, stores comparison results
- `%rip` instruction pointer, next address to execute

3AC Reference

List of pseudoinstructions operating over pseudovariables. It's ok to fudge this a little bit, as long as you don't nest expressions or instructions.

`x := y op z`
Perform a logical, relational, or mathematical operation on y and z, then assign the result to x. You may assume relational and logical operators represent true as 1, false as 0.

`x := y`
Assign the value of pseudovariable y to pseudovariable x

`ifz x goto L`
If value x has the value 0, jump to the program location with label L.

`goto L`
Jump to the program location with label L.

`call p`
Transfer control to the body of function p with any arguments set via the set_arg pseudoinstruction.

`setarg k, x`
Set the kth argument value in caller to x.

`setret x`
Set the return value to x.

`getarg k, x`
Set the kth argument value in callee to x.

`getret x`
Set x to the return value from the last call.

enter ⟨proc⟩
Begin procedure ⟨proc⟩.

leave ⟨proc⟩
End procedure ⟨proc⟩.

`label L` Mark the next instruction as being at label L.

`WRITE x, y` Output the value of x to filesystem handle x.

`READ x, y` Get the value of x from filesystem handle y.

Syscall Reference

- sys_read: Syscall 0
  - `%rdi`: file descriptor (unsigned int) to read from
  - `%rsi`: address of memory buffer to place read characters
  - `%rdx`: maximum number of characters (unsigned int) to read

  If the file has fewer characters than the maximum number requested, all remaining characters will be read to the buffer. The number of characters actually read will be placed in `%rax`.

- sys_write: Syscall 1
  - `%rdi`: file descriptor (unsigned int) to write
  - `%rsi`: address of string to write to the file
  - `%rdx`: maximum number of characters (unsigned int) to write
- sys_open: Syscall 2
  - `%rdi`: filename
  - `%rsi`: address of string to write to the file
  - `%rdx`: maximum number of characters (unsigned int) to write
- sys_exit: Syscall 60
  - `%rdi`: program exit code

AST Node Reference

```
AstNode
        - Position (source code location)

ProgramNode
        - list of DeclNodes (globals)

VarDeclNode
        - TypeNode (variable type declared)
        - IDNode (variable name declared)

FnDeclNode
        - TypeNode (return type)
        - IDNode (name of declared function)
        - FormalDeclNode list (formal parameters)
        - StmtNode list (function body)

AssignStmtNode
        - LocNode (the destination location)
        - ExpNode (the source expression)

ReadStmtNode
        - LocNode (file variable to read from)
        - LocNode (variable to receive file data)

WriteStmtNode
        - LocNode (file variable to write)
        - ExpNode (expression being written)

OpenNode
        - LocNode (file to open)
        - std::string (path to file)

CloseNode
        - LocNode (the file to close)

PostDecStmtNode / PostIncStmtNode
        - LocNode (location being changed)

ReturnStmtNode
        - ExpNode (return value - maybe null)

CallStmtNode
        - CallExpNode (underlying function call)

CallExpNode
        - IDNode (name of the callee function)
        - list of ExpNode (arguments)

TernaryExpNode
        - ExpNode (condition being evaluated)
        - ExpNode (result on true condition)
        - ExpNode (result on false condition)

IfStmtNode
        - ExpNode (condition being evaluated)
        - StmtNode list (body of the if stmt)

IfElseStmtNode
        - ExpNode (condition being evaluated)
        - StmtNode list (true branch body)
        - StmtNode list (false branch body)

WhileStmtNode
        - ExpNode (condition being evaluated)
        - StmtNode list (body of the loop)
IDNode
        - std::string (the name of the identifier)

BinaryExpNode
        - ExpNode (the lhs operand)
        - ExpNode (the rhs operand)

PlusNode / MinusNode / etc
        (no extra fields needed beyond superclass)

UnaryExpNode
        - ExpNode (the underlying expression)

NotNode / NegNode
        (no extra fields needed beyond superclass)

IntTypeNode / other primitive type nodes
        (no fields needed beyond superclass)

FileTypeNode
        (no fields needed beyond superclass)

IndexNode
        - IDNode (array being indexed)
        - ExpNode (index expression)

Literal nodes
        - the underlying literal value
```