

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

Feel free to draw something summery
in the box below

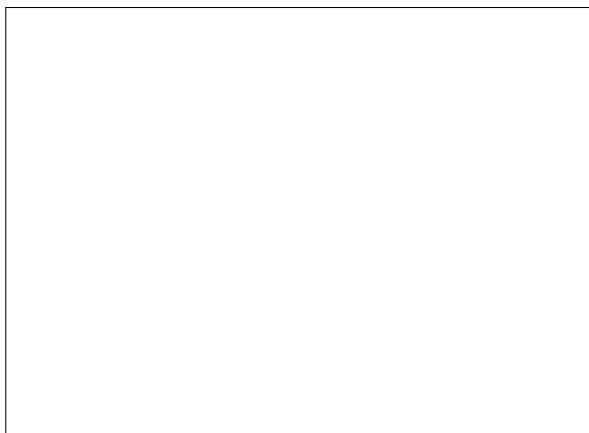
Question 1: /10

Question 2: /10

Question 3: /10

Question 4: /10

Question 5: /10



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.

```
fn : () int main {  
    int a;  
    a = 2;   
    a = 1;  
    a = a + 1;   
    return a;  
}
```

dead code (def is immediately killed)

instruction strength reduction
(addq replaceable by incq)

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?

Simplify the AB to use
space below rsp. simplify the
prologue and epilogue.

PART II

Treat all registers as callee-saved

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?

All registers in use would need to
be saved before a call and
restored after a call. The compiler
would also need to set up side
table information for restoring
the base pointer (rbp)
and stack pointer (rsp)

boxes ↙

QUESTION 3 (10 POINTS)

Student ID: _____

Complete All Parts

PART I

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

Assembler: converts a text format assembly code to ~~an~~ a binary format object code file with abstract addresses

Linker: replace abstract addresses with virtual addresses, stitch together multiple object files

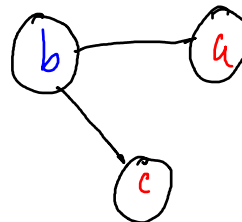
Loader: Map the program in disk into process memory,

PART II

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

[a] := 1
[b] := 2
output [a]
[c] := 3

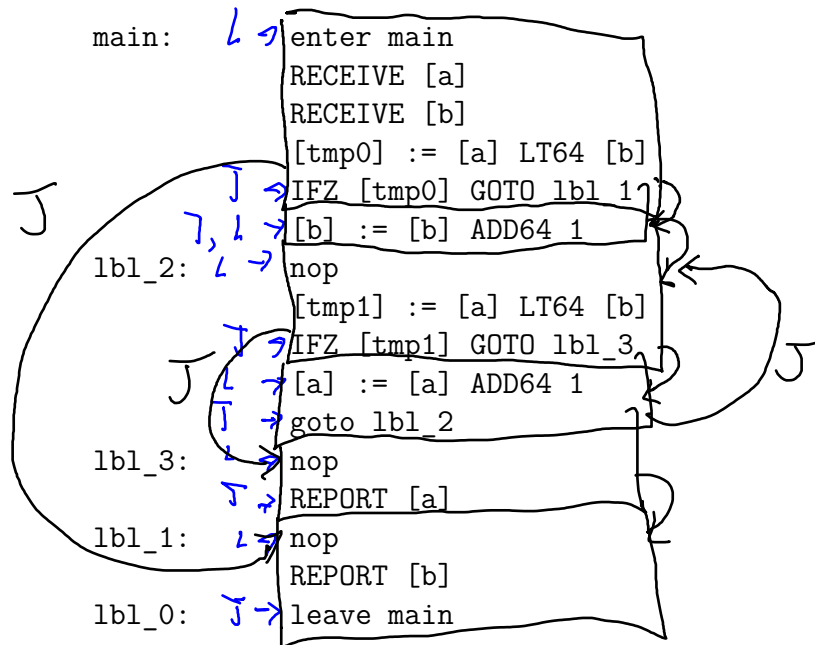
output [b]
output [c]



QUESTION 4 (10 POINTS)

Student ID: _____

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



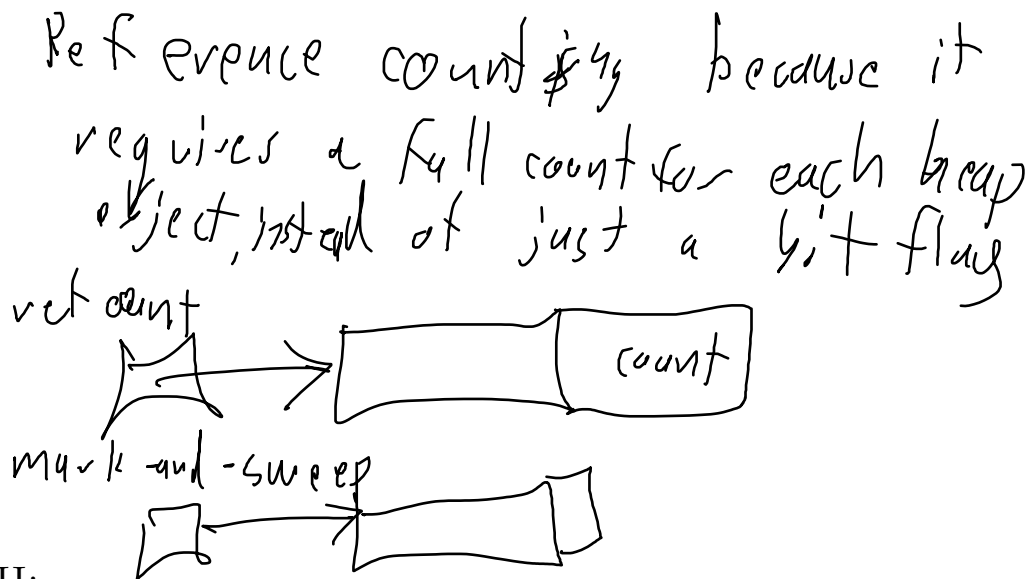
QUESTION 5 (10 POINTS)

Student ID: _____

Complete All Parts

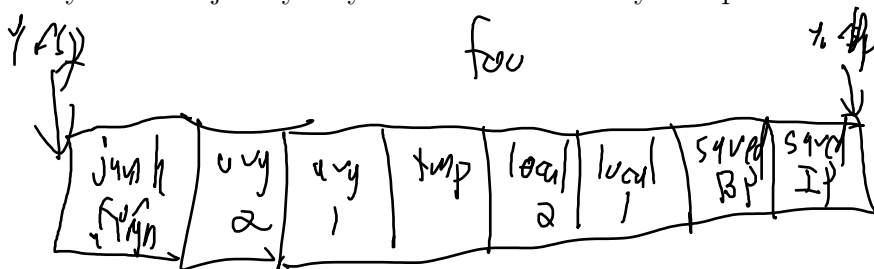
PART I:

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 necessary and placed in the AR.



X64 REFERENCE

- `movq <opd1> <opd2>` - Copy the 8-byte value of `opd1` into `opd2`
- `addq <opd1> <opd2>` - Put the result of `opd2 + opd1` into `opd2`
- `subq <opd1> <opd2>` - Put the result of `opd2 - opd1` into `opd2`
- `imulq <opd1>` - Put the result of `%rax * opd1` into the octoword `%rdx:%rax`
- `callq <lbl>` - Stack (push) the next instruction address, move `%rip` to the address `<lbl>`
- `retq` - Unstack (pop) into `%rip`
- `xorq <opd1> <opd2>` - Put the result of `<opd2> XOR <opd1>` into `opd2`
- `negq <opd>` - Put the 2's complement negation of `<opd>` into `<opd>`
- `notq <opd>` - Flip all bits of `<opd>`
- `jmp <lbl>` - jump to `<lbl>`
- `cmpq <opd1> <opd2>` - Set `rflags` according to `<opd2> - <opd1>`
- `je <lbl>` - jump to `<lbl>` if `rflags` indicates a `=` relation on prior operands
- `jne <lbl>` - jump to `<lbl>` if `rflags` indicates a `≠` relation on prior operands
- `jge <lbl>` - jump to `<lbl>` if `rflags` indicates a `≥` relation on prior operands
- `jl <lbl>` - jump to `<lbl>` if `rflags` indicates a `<` relation on prior operands
- `jg <lbl>` - jump to `<lbl>` if `rflags` indicates a `>` relation on prior operands
- `jle <lbl>` - jump to `<lbl>` if `rflags` indicates a `≤` relation on prior operands
- `sete <opd>` - Set `opd` to be 1 if `rflags` indicates that the last compare operation had equal operands, 0 otherwise. `<opd>` must be a 1-byte register.
- `setg <opd>` - Set `opd` to be 0 if `rflags` indicates that the last compare operation had an `opd2` less than or equal to its `opd1`, 1 otherwise. `<opd>` must be a 1-byte register.
- `setle <opd>` - Set `opd` to be 0 if `rflags` indicates that the last compare operation had an `opd2` greater than its `opd1`, 0 otherwise. `<opd>` must be a 1-byte register.

REGISTERS

General-purpose registers

- `%rax - %rdx` (lowest 1 byte is `%al - %dl`) Select special-purpose registers
- `%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
- `%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 pseudovvariable y to pseudovvariable 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)	TernaryExpNode - ExpNode (condition being evaluated) - ExpNode (result on true condition) - ExpNode (result on false condition)
ProgramNode - list of DeclNodes (globals)	IfStmtNode - ExpNode (condition being evaluated) - StmtNode list (body of the if stmt)
VarDeclNode - TypeNode (variable type declared) - IDNode (variable name declared)	IfElseStmtNode - ExpNode (condition being evaluated) - StmtNode list (true branch body) - StmtNode list (false branch body)
FnDeclNode - TypeNode (return type) - IDNode (name of declared function) - FormalDeclNode list (formal parameters) - StmtNode list (function body)	WhileStmtNode - ExpNode (condition being evaluated) - StmtNode list (body of the loop)
AssignStmtNode - LocNode (the destination location) - ExpNode (the source expression)	IDNode - std::string (the name of the identifier)
ReadStmtNode - LocNode (file variable to read from) - LocNode (variable to receive file data)	BinaryExpNode - ExpNode (the lhs operand) - ExpNode (the rhs operand)
WriteStmtNode - LocNode (file variable to write) - ExpNode (expression being written)	PlusNode / MinusNode / etc (no extra fields needed beyond superclass)
OpenNode - LocNode (file to open) - std::string (path to file)	UnaryExpNode - ExpNode (the underlying expression)
CloseNode - LocNode (the file to close)	NotNode / NegNode (no extra fields needed beyond superclass)
PostDecStmtNode / PostIncStmtNode - LocNode (location being changed)	IntTypeNode / other primitive type nodes (no fields needed beyond superclass)
ReturnStmtNode - ExpNode (return value - maybe null)	FileTypeNode (no fields needed beyond superclass)
CallStmtNode - CallExpNode (underlying function call)	IndexNode - IDNode (array being indexed) - ExpNode (index expression)
CallExpNode - IDNode (name of the callee function) - list of ExpNode (arguments)	Literal nodes - the underlying literal value