Exam 3

601.428/628 Compilers and Interpreters

December 16, 2022

Complete all questions.

Time: 120 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: ____________________

Print name: ____________________

Date: ____________________

Reference

Reference for high-level instructions (operand size suffixes denoted x, y are `b`=8 bits, `w`=16 bits, `l`=32 bits, `q`=64 bits, comparisons denoted T are `lt`=less than, `lte`=less than or equal, `gt`=greater than, `gte`=greater than or equal, `eq`=equality, `neq`=inequality):

Instruction		Meaning
`mov_`*x*	*Vrd*, *Op*	Copy value of *Op* into *Vrd*
`mov_`*x*	(*Vrs*), *Op*	Copy value of *Op* into memory location pointed to by *Vrs*
`add_`*x*	*Vrd*, *Op*, *Op*	Store sum of operands in *Vrd*
`sub_`*x*	*Vrd*, *Op*, *Op*	Store difference of operands (left - right) in *Vrd*
`mul_`*x*	*Vrd*, *Op*, *Op*	Store product of operands in *Vrd*
`div_`*x*	*Vrd*, *Op*, *Op*	Store result of dividing operands (left / right) in *Vrd*
`sconv_`*xy*	*Vrd*, *Vrs*	Sign-extend value of *Vrs*, store in *Vrd* (size conversion from x to y)
`localaddr`	*Vrd*, \$N	Store pointer to local memory at offset \$N in *Vrd*
`cmp`*T*`_`*x*	*Vrd*, *Op*, *Op*	Compare left and right operands, place boolean result in *Vrd*
`cjmp_t`	*Vrs*, *label*	Conditional jump to label if *Vrs* contains a true value
`cjmp_f`	*Vrs*, *label*	Conditional jump to label if *Vrs* contains a false value
`call`	*label*	Call function named by label
`ret`		Return from instruction
`enter`	\$N	Create stack frame with specified amount of local storage
`leave`	\$N	Tear down stack frame with specified amount of local storage
`spill`	*Vrs*/Mr, \$N	Spill value of *Vrs* to spill location \$N
`restore`	*Vrd*/Mr, \$N	Restore *Vrd* from spill location \$N

- Virtual registers are `vr0`, `vr1`, etc.
- `vr0` is return value, `vr1` through `vr6` are arguments
- *Vrd* means a destination virtual register (modified by the instruction)
- *Vrs* means a source virtual register (not modified by the instruction)
- Parentheses surrounding a virtual register means a memory reference using the virtual register as a pointer (e.g., `(vr12)`)
- \$N means an integer constant (e.g, `$42`)
- *Op* means a source operand (source virtual register not modified, integer constant, or memory reference)
- *label* means a target label
- Mr means a machine register (assigned as part of local register allocation)

Question 1. [25 points] Consider the following basic block of high-level instructions:

```
add_l   vr20, vr12, vr13

sub_l   vr21, vr12, vr13

add_l   vr22, vr13, vr12

sub_l   vr23, vr13, vr12

mov_l   vr12, $42

add_l   vr24, vr12, vr13

mov_l   vr25, (vr17)

mov_l   vr26, (vr18)

add_l   vr28, vr25, vr26

mov_l   (vr19), vr28

mov_l   vr29, (vr17)

add_l   vr30, vr29, vr26
```

Annotate each reference to a virtual register with a value number representing its runtime value immediately *after* the instruction executes. If two values are guaranteed to be the same at runtime, they should be assigned the same value number.

Note that in the case of a memory reference operand (e.g., `(vr17)`), you are specifying the value number of the virtual register, not the memory operand. However, in the case where a value is loaded from memory into a virtual register, the destination virtual register should be annotated with a value number representing the value loaded from memory.

Question 2. [25 points] Consider the following basic block of high-level instructions:

```
localaddr vr16, $1600

mul_l     vr17, vr11, vr10

add_l     vr18, vr17, vr12

sconv_lq  vr19, vr18

mul_q     vr20, vr19, $8

add_q     vr21, vr16, vr20

mov_q     vr22, (vr21)

localaddr vr23, $800

mul_l     vr24, vr13, vr10

add_l     vr25, vr24, vr12

sconv_lq  vr26, vr25

mul_q     vr27, vr26, $8

add_q     vr28, vr23, vr27

mov_q     vr30, (vr28)

mul_q     vr29, vr14, vr30

add_q     vr31, vr22, vr29

mov_q     (vr21), vr31

add_l     vr39, vr12, $1

mov_l     vr12, vr39
```

[Question continues on next page.]

[Continuation of Question 2.]

(a) Which virtual registers are definitely live at the beginning of the basic block? Ignore the possibility that there are virtual registers live at the end of the block that are still live at the beginning, so focus only on "upward exposed" virtual registers. (Note that Question 4 has a description of liveness.)

(b) Assume that `vr10` through `vr14` are live at the end of the basic block. Assume that machine registers called A, B, and C are available for local register allocation. Annotate the code on the previous page to indicate, for each instruction, an assignment of a machine register for each virtual register that is eligible for register allocation. (Hint: virtual registers live at the beginning and/or end of the basic block are not eligible.) You should use bottom-up register allocation. If any spills or restores are required, indicate where they occur as well as the virtual register, machine register, and spill location. For example,

```
spill vr27/A, $1
```

would spill `vr27` to spill location 1, reclaiming machine register A, and

```
restore vr27/B, $1
```

would restore the previously spilled value of `vr27` from spill location 1, loading the value into machine register B.

Suggestion: it will be helpful to make a note of which machine registers are available and in-use at each point in the sequence. Also, don't forget that when a virtual register becomes dead (has no subsequent uses), its assigned machine register can be reclaimed.

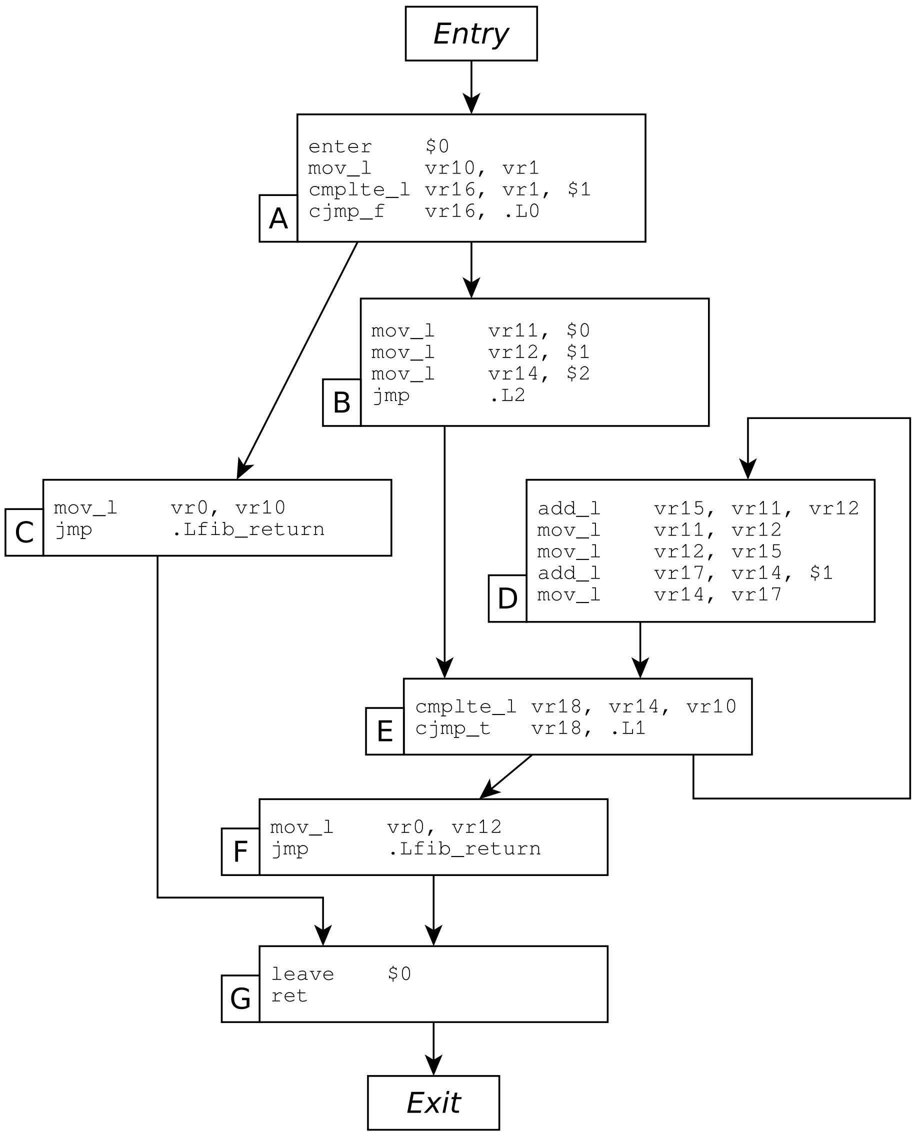
Question 3. [20 points] Consider the high-level code for a function called `bubble`:

```
        .globl bubble
bubble:
        enter    $0
        mov_q    vr10, vr1
        mov_l    vr11, vr2
        mov_l    vr12, $1
        jmp      .L1
.L0:
        sconv_lq vr14, vr12
        mul_q    vr15, vr14, $4
        add_q    vr16, vr10, vr15
        sub_l    vr18, vr12, $1
        sconv_lq vr19, vr18
        mul_q    vr20, vr19, $4
        add_q    vr21, vr10, vr20
        mov_l    vr23, (vr16)
        mov_l    vr24, (vr21)
        cmplt_l  vr22, vr23, vr24
        cjmp_f   vr22, .L2
        mov_q    vr1, vr10
        sub_l    vr15, vr12, $1
        mov_l    vr2, vr15
        mov_l    vr3, vr12
        call     swap
.L2:
        add_l    vr17, vr12, $1
        mov_l    vr12, vr17
.L1:
        cmplt_l  vr18, vr12, vr11
        cjmp_t   vr18, .L0
        leave    $0
        ret
```

On the next page, draw a control flow graph of the `bubble` function. Be sure to designate which blocks are the entry and exit. Make sure all control-flow edges are clearly indicated as arrows. Make sure the instructions in each block are clearly indicated. (You can label sequences of instructions on this page and refer to those labels in the control-flow graph, to avoid the need to copy all of the instructions.)

[Draw your control-flow graph for Question 3 on this page.]

Question 4. [20 points] Consider the following control-flow graph:



Recall that the dataflow equations for liveness are

$$\text{LiveOut}(n) = \emptyset \qquad \textit{(initially)}$$
$$\text{LiveOut}(n) = \cup_{m \in \text{Succ}(n)}(\text{UEVar}(m) \cup (\text{LiveOut}(m) - \text{VarKill}(m)))$$

Succ(n) is the set of control successors of block n. LiveOut(n) is the set of virtual registers which are live at the end of block n. UEVar(n) is the set of "upward exposed" virtual registers (those where there is a use not preceded by an assignment) in block n. VarKill(n) is the set of virtual registers assigned in block n.

[Continued on next page.]

[Continuation of Question 4.]

(a) Specify the UEVar and VarKill sets for each basic block A–G. (You may annotate this on the control-flow graph if you wish.)

(b) Specify the LiveOut set for each basic blocks A–G and also the *Entry* block, as determined by iterating the dataflow equations until there are no changes. (You may annotate this on the control-flow graph if you wish.)

(c) Were multiple iterations needed for any basic block(s)? If so, explain briefly.

Question 5. [10 points] Consider the following sequence of x86-64 instructions:

```
movq     %rdx, %rsi
imulq    $8, %rsi
movq     %r9, %r8
addq     %rsi, %r8
movq     (%r8), %rcx
```

This sequence could potrentially be replaced by the following instruction:

```
movq     (%r9,%rdx,8), %rcx
```

Note that the x86-64 indexed/scaled addressing mode, specified as an operand of the form (B,I,S), accesses a value in memory at the address $B + I \times S$.

Under what circumstances would this replacement preserve the semantics of the program? Explain briefly. Hint: think about the original sequence as being part of a larger function. Also: keep in mind that in x86-64, the destination operand is the *last* operand.

[You can use this page for scratch work and/or answers.]

[You can use this page for scratch work and/or answers.]