Exam 3

601.428/628 Compilers and Interpreters

December 21, 2021

Complete all questions.

Closed book, no use of electronic resources is permitted.

You may use two sheets of hand-written notes.

Time: 90 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:

Instruction		Meaning
`ldci`	*Vrd*, $N	Store integer constant $N to *Vrd*
`addi`	*Vrd*, *Op*, *Op*	Store sum of operands to *Vrd*
`subi`	*Vrd*, *Op*, *Op*	Store difference of operands (left - right) to *Vrd*
`muli`	*Vrd*, *Op*, *Op*	Store product of operands to *Vrd*
`divi`	*Vrd*, *Op*, *Op*	Store result of dividing operands (left / right) to *Vrd*
`modi`	*Vrd*, *Op*, *Op*	Store remainder after dividing operands (left / right) to *Vrd*
`negi`	*Vrd*, *Op*	Store negation of operand to *Vrd*
`localaddr`	*Vrd*, $N	Store address of memory location at offset of $N bytes into local memory storage area to *Vrd*
`ldi`	*Vrd*, (*Vra*)	Store value loaded from memory location (*Vra*) to *Vrd*
`sti`	(*Vra*), *Vrb*	Store value in *Vrb* to memory location (*Vra*)
`readi`	*Vrd*	Read integer input value and store it to *Vrd*
`writei`	*Vra*	Write integer value in *Vra* to output
`jmp`	*label*	Unconditional jump to *label*
`cmpi`	*Op*, *Op*	Compare operands
`je`	*label*	Conditional jump to *label* if in previous comparison, left = right
`jne`	*label*	Conditional jump to *label* if in previous comparison, left ≠ right
`jlt`	*label*	Conditional jump to *label* if in previous comparison, left < right
`jlte`	*label*	Conditional jump to *label* if in previous comparison, left ≤ right
`jgt`	*label*	Conditional jump to *label* if in previous comparison, left > right
`jgte`	*label*	Conditional jump to *label* if in previous comparison, left ≥ right
`mov`	*Vrd*, *Vra*	Store value in *Vra* to *Vrd*

Notes:

- Virtual registers are `vr0`, `vr1`, etc.
- *Vrd* means a destination virtual register (modified by the instruction)
- *Vra* and *Vrb* are source virtual registers (not modified by the instruction)
- (*Vra*) and (*Vrb*) mean a memory reference using a virtual register as a pointer, e.g., `(vr0)`
- $N means an integer constant (e.g, `$42`)
- *Op* means a source operand (either source virtual register or integer constant)
- *label* means a target label
- All values are 64-bit signed integers

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

```
                     0,2
 1:    localaddr vr13, $0
                     0,2, 13
 2:    ldci      vr14, $10
                     0,2,13,14
 3:    muli      vr15, vr0, vr14
                     0,2,13,15
 4:    addi      vr16, vr15, vr2
                     0,2,13,16
 5:    muli      vr17, vr16, $8
                     0,2,13,17
 6:    addi      vr18, vr13, vr17
                     0,2,18
 7:    ldi       vr3, (vr18)
                     0,2,3
 8:    ldci      vr19, $0
                     0,2,3, 19
 9:    mov       vr1, vr19
                     0, 1, 2,3
10:    jmp       .L16
                     0,1,2,3
```

Assume that `vr0`, `vr1`, `vr2`, and `vr3` are live at the end of the basic block.

Show:

1. The virtual registers that are live at the beginning of the block
2. For each instruction in the block, which virtual registers are live at the point *just after* the instruction

Recall that a virtual register is live it will be used at a later point, but there is not an intervening def (assignment) of the virtual register.

Question 2. [25 points] Assume that the following code is a basic block:

```
ldci      vr4, $3
ldci      vr5, $2
muli      vr6, vr4, vr5
addi      vr7, vr0, vr6
writei    vr7
```

Assume that vr4, vr5, vr6, or vr7 are temporaries, so they are all dead at the end of the block.

(a) Rewrite this code so that all uses of virtual registers with known constant values are replaced with the appropriate constant. For example, in an instruction `mov vr8, vr9`, if vr9 is known to have the constant value 42, then you would rewrite the instruction as `mov vr8, $42`. Note that if a value is computed from constant operands, the computed value should also be treated as a constant (constant folding.)

```
X ldci vr4, $3
X ldci vr5, $2
X muli vr6, $3, $2
  addi vr7, vr0, $6
  writei vr7
```

(b) Which instructions in the transformed code can be eliminated? Explain briefly.

Instructions marked with an X above can be eliminated because they are stores to dead virtual registers.

Question 3. [20 points] Assume that a local register allocator has three machine registers available, called A, B, and C. For each the following instructions, we want the local register allocator to assign a machine register for each virtual register. The first instruction is annotated to show that `vr0` has been assigned machine register A.

□ last use

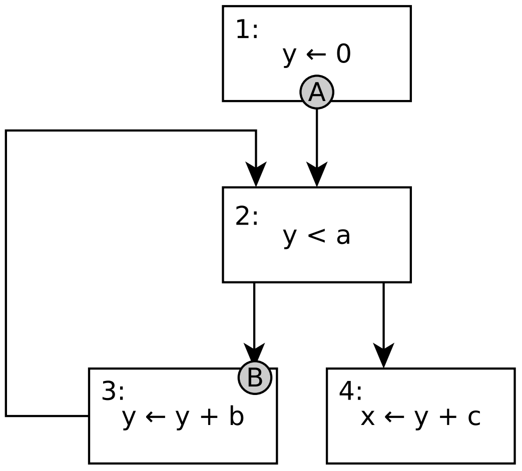
Instruction	Register assignments	Spill and/or restore?
`ldci vr0, $1`	vr0→A	
`ldci vr1, $2`	vr0→A, vr1→B	
`ldci vr2, $2`	vr0→A, vr1→B, vr2→C	spill vr2, C
`addi vr3,` $\boxed{\texttt{vr0}}$, $\boxed{\texttt{vr1}}$	vr0→A, vr1→B, vr3→C	
`muli vr4,` $\boxed{\texttt{vr3}}$, `$8`	vr4→A, vr3→C	restore vr2, B
`addi vr5,` $\boxed{\texttt{vr4}}$, $\boxed{\texttt{vr2}}$	vr4→A, vr2→B, vr5→C	

Complete the table above by performing local register allocation. For each instruction in the block, show a possible assignment of machine registers to the referenced virtual registers. Use bottom-up allocation, so that when a spill is necessary, the live virtual register whose next use is furthest in the future is selected as a victim.

Indicate where spills and restores are necessary. You can use the notation `spill VR,MR` and `restore VR,MR`, where `VR` is the virtual register being spilled or restored, and `MR` is the machine register assignment being removed by the spill or established by the restore.

Don't forget to reclaim machine registers after the last use of the virtual register to which the machine register is assigned.

Question 4. [20 points] Consider the control-flow graph on the right, where block $\boxed{1}$ is the entry block and block $\boxed{4}$ is the exit block, and the variables are a, b, c, x, and y:



(a) At the point labeled (A) (end of block $\boxed{1}$), which variables are guaranteed to be used on some forward path? (I.e., which variables are guaranteed to be used at some point in the future?)

{ a, c, y }

(b) At the point labeled (B) (beginning of block $\boxed{3}$), which variables are guaranteed to be used on some forward path?

{ a, b, c, y }

(c) Would a dataflow analysis to find guaranteed uses of variables be a forward analysis or a backward analysis? Explain briefly.

Backwards: we want to know what uses will occur in the future, so we need to work backwards

(d) Would a dataflow analysis to find guaranteed uses of variables be a "may" analysis or a "must" analysis? Explain briefly.

Must: we want uses that are guaranteed on all forward paths

Question 5. [10 points] Local value numbering (LVN) is useful for detecting redundant computations and replacing them with a use of a previously-computed value.

One potential obstacle to using LVN to eliminate redundant computations is that a storage location (i.e., virtual register) containing a computed value might be overwritten. For example, consider the following code:

```
1:    addi      vr4, vr5, $42
2:    writei    vr4
3:    ldci      vr4, $17
4:    addi      vr6, vr5, $42
5:    writei    vr6
```

At line 4, there is a recomputation of the sum `vr5` + 42, which is a value that was previously stored in `vr4`. However, because `vr4` was modified at line 3, it is not correct to replace the instruction at line 4 with

```
mov       vr6, vr4
```

or to replace the instruction at line 5 with

```
writei    vr4
```

State whether you think this is a significant problem in practice. I.e., is it likely that computed values will become unavailable by being overwritten, or could the compiler guarantee that this will not happen? Hint: think about how code generation for expression evaluation and address computation works. Briefly justify your answer.

No, it shouldn't be a problem. If, when generating code to evaluate an expression or compute an address, the code generator consistently allocates a new virtual register for each computed value, and never re-uses a virtual register, computed values will never be destroyed.

[Extra page for answers and/or scratch work.]