CSE 401 15wi Final Exam Sample Solution

Question 1. (10 points) Compiler phases. For each of the following tasks, indicate where the task would normally be done in a production compiler. Assume that the compiler is a conventional one that generates native code for a single target machine (say, x86-64), and assume that the source language is standard Java (if it matters). Use the following abbreviations for the stages:

scan – scanner
parse – parser
sem – semantics/type check
opt – optimization
instr – instruction selection & scheduling
reg – register allocation
run – runtime (i.e., when the compiled code is executed)
can't – can't always be done during compilation or execution

___**opt**____ Remove an assignment to a variable because the variable is never referenced later in the program.

___**scan**___ Report that '#' is not a legal character in a source program.

___**instr**___ Rearrange the order of instructions in the final code to reduce delays caused by the long execution times of LOAD and STORE instructions.

__**parse**___ Report that a closing '}' at the end of a method definition is missing.

___**sem**___ In full Java, report that a reference to field `x.a` is illegal because the class (type) of variable `x` does not contain a field named `a`.

___**can't**__ Report that a loop will never terminate once it has begun execution (i.e., produce an error message that the program contains an "infinite" loop).

___**opt**___ Replace a multiplication by 2 with an add operation. **(we also allowed instr for this, since it can be done there, but the optimizer would be the more usual location)**

___**sem**___ Report that a variable has not been declared.

___**opt**___ Report that a variable might not be initialized when it is used in a statement. **(In a moment of weakness we decided to allow sem as another answer for this. While it is true that Java requires detecting this as part of the language semantics, it requires dataflow analysis to do so, and that is normally part of the infrastructure in the optimization passes.)**

___**run**___ In a method call `x.f(…)`, select the method `f` to be executed based on the actual class (type) of the object referenced by `x`.

Question 2. (16 points) Compiler hacking: a question of several parts. One of our customers wants to use MiniJava to implement an operating system, but insists that this *cannot be done* unless MiniJava includes a `do-while` loop in the language to go with the existing while loop. We don't know why this is so essential, but we've decided to go ahead and add this new kind of loop to the language to make the customer happy.

To do this we need to add one new production to the MiniJava Grammar:

Statement ::= "do" Statement "while" "(" Expression ")" ";"

The meaning of a `do-while` loop is the customary one: the Statement in the loop body is executed, then the Expression (condition) following the keyword "while" is evaluated. If the Expression evaluates to `true`, execution of the `do-while` loop repeats. If the Expression evaluates to `false`, execution of the `do-while` loop terminates.

(a) (2 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add this new `do-while` loop to the original MiniJava language? Just describe any necessary changes; you don't need to give JFlex or CUP specifications or code.

Add a new token for the "do" terminal symbol

(b) (3 points) What changes are needed to the MiniJava abstract syntax tree (AST) data structures to add this new `do-while` loop to MiniJava? Again, you do not need to give any Java or CUP code, just describe the changes (what kinds of new or changed nodes, what children would they have, etc.).

Add a new "dowhile" node with two children: the loop body Statement and the loop condition Expression.

Note: it would not be a good solution to use the existing while-loop node for this because the new loop has different execution semantics.

(c) (3 points) What checks need to be performed to verify that there are no type compatibility or other semantics errors for this new `do-while` loop?

Verify that the Expression following "while" has type Boolean.

(continued next page)

Question 2. (cont.) (d) (8 points) Describe the x86-64 code shape for this new `do-while` loop that would be generated by a MiniJava compiler. Your answer should be similar in format to the descriptions we used in class for other language constructs. Be sure to show where the code to evaluate the loop condition expression and the code for the statement in the loop body would appear in the generated code for the loop. If needed, you should assume that the code generated for an expression will leave the value of that expression in `%rax`, as we did in the MiniJava project. Also, assume that the stack is aligned on a 16-byte boundary at the beginning of the code sequence, and, if you change the size of the stack, you need to be sure this alignment is preserved if the loop body statement or the loop condition expression could contain a method call.

Use the Linux/gcc x86-64 assembler syntax for your code. If you need to make any assumptions about code generated by the rest of the compiler you should state them.

The basic idea is pretty straightforward:

```
do_label:
            <code for Statement>
            <code to evaluate Expression into %rax>
            testq   %rax,%rax           # test expression and jump if not 0 (i.e., not false)
            jnz     do_label
```

We gave a fair amount of latitude in how the conditional test was written (i.e., evaluate the expression and jump if true to the top of the loop would be another reasonable answer).

Question 3. (16 points) A bit of coding. This question concerns these classes from a MiniJava program:

```
class Base {
  int a;
  int b;

  public int f(int n) { b = n+1; return n+2; }
  public int g(int n) { return a + n; }
  public int setA(int v) { a = v; return a; }
  public int setB(int v) { b = v; return b; }
}
class Sub extends Base  {
  int c;
  public int setC(int v) { c = v; return c; }
  public int g(int n) {
    c = this.f(b);
    return b + n;
  }
}
```

Answer questions about this code on the next pages.

Ground rules for x86-64 code, needed in part of the question (same as for the MiniJava project and other x86-64 code, with the addition that `%rbp` must be used as the stack frame base register – it is not optional for this question):

- You must use the Linux/gcc assembly language, and must follow the x86-64 function call, register, and stack frame conventions.
  - Argument registers: `%rdi, %rsi, %rdx, %rcx, %r8, %r9` in that order
  - Called function must save and restore `%rbx, %rbp,` and `%r12-%r15` if these are used in the function
  - Function result returned in `%rax`
  - `%rsp` must be aligned on a 16-byte boundary when a `call` instruction is executed
  - `%rbp` must be used as the base pointer (frame pointer) register for this question, even though this is not strictly required by the x86-64 ABI.
- Pointers and `int`s are 64 bits (8 bytes) each, as in MiniJava.
- Your x86-64 code must implement all of the statements in the original method. You may *not* rewrite the method into a different form that produces equivalent results (i.e., restructuring or reordering the code). Other than that, you can use any reasonable x86-64 code that follows the standard function call and register conventions – you do not need to mimic the code produced by your MiniJava compiler.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended.)

(You may detach this page from the exam if that is convenient.)

Question 3. (cont.) (a) (6 points) When class `Base` was compiled, the compiler picked the following layout for objects of type `Base`, and generated the following vtable for that class:

Object Layout

offset	field
+0	vtable pointer
+8	a
+16	b

Vtable layout

```
Base$$:  .quad 0            # no superclass
         .quad Base$f       # +8
         .quad Base$g       # +16
         .quad Base$setA    # +24
         .quad Base$setB    # +32
```

Below, show the object and vtable layouts for class `Sub`, in the same format used above for class `Base`. Be sure to properly account for the fields and methods inherited from class `Base` in the object and vtable layouts for class `Sub`.

Object Layout

offset	field
+0	vtable pointer
+8	a
+16	b
+24	**c**

Vtable layout

```
Sub$$:   .quad Base$$       # superclass
         .quad Base$f       # +8
         .quad Sub$g        # +16
         .quad Base$setA    # +24
         .quad Base$setB    # +32
         .quad Sub$setC     # +40
```

Note: this is the only possible solution. The object layout needs to match the superclass for the portions that are inherited, and method ordering in the vtable also needs to match the superclass, with pointers to overriding methods in the proper places and with pointers to new subclass methods added at the end.

(continued next page)

Question 3. (cont.) (b) (10 points) Now translate method `g(n)` in class `Sub` into x86-64 assembly language. You should use the standard runtime conventions for parameter passing (including the `this` pointer), register usage, and so forth that we used in the MiniJava project, including using `%rbp` as a stack frame pointer. Your translation should be consistent with the object and vtable layouts from part (a) of the question on the previous page. The method source code is repeated below for convenience.

`call` instruction hints: Recall that if `%rax` contains a pointer to (i.e., the memory address of) the first instruction in a method, then you can call the method by executing `call *%rax`. If `%rax` contains the address of a vtable, we can call a method whose pointer is at offset *d* in that vtable by executing `call *d(%rax)`.

```
public int g(int n) {
  c = this.f(b);
  return b + n;
}
```

```
        # on entry: %rdi = "this" pointer,  %rsi = n
Sub$g:                                  # prologue
        pushq   %rbp                    # save old frame pointer (also aligns %rsp on 16-byte bndry)
        movq    %rsp,%rbp               # set up new frame pointer
        subq    $16,%rsp                # allocate space for 2 temporaries
        movq    %rdi,0(%rsp)            # save "this" at %rsp (equiv. %rbp-16)
        movq    %rsi,8(%rsp)            # save n at %rsp+8    (equiv. %rbp-8)
                                        # call this.f(b) - "this" already in %rdi
        movq    16(%rdi),%rsi           # this.b is second argument
        movq    (%rdi),%rax             # copy vtable ptr into %rax
        call    *8(%rax)                # call f through vtable

        movq    0(%rsp),%rdi            # restore "this" (may have been clobbered by call)
        movq    %rax,24(%rdi)           # store this.f(b) result in this.c

        movq    16(%rdi),%rax           # load this.b
        addq    8(%rsp),%rax            # add n

        movq    %rbp,%rsp               # return - result already in %rax
        popq    %rbp
        ret
```

In retrospect we should have allocated more points for this question and a few less for the previous one. One big issue in the above code is that calling another method can potentially change all of the argument and other transient registers, so we need to save and restore `this` and `n`. Also, the call of `this.f` must go through the vtable since it could potentially be overridden in a subclass of `Sub`. We can't make any assumptions about method is called or what variables it might or might not change.

Question 4. (16 points) Register allocation. Considering the following code:

```
a = read();
b = read();
c = a + b;
if (a < b) {
   d = b - 1;
   e = d * 2;
} else {
   e = c - a;
}
print(e);
```

On the next page write your answer to the following questions. You can remove this page from the exam for convenience if you wish.

(a) (9 points) Draw the interference graph for the variables in this code. You are not required to draw the control flow graph, but it might be useful to sketch that to help find the solution and to leave clues in case we need to assign partial credit.

(b) (7 points) Give an assignment of variables to registers using the minimum number of registers possible, based on the information in the interference graph. You do not need to go through the steps of the graph coloring algorithm explicitly, although that may be helpful as a guide to assigning registers. If there is more than one possible answer that uses the minimum number of registers, any of them will be fine. Use R1, R2, R3, … for the register names.

write

your

answer

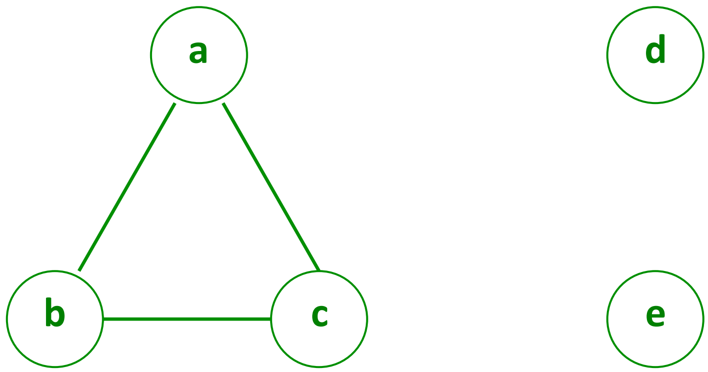
on

the

next

page ->

Question 4. (cont.) Draw the interference graph and write your register assignments below the graph.



Three registers are needed. a, b, and c each need to be in a separate register. d and e can occupy any of the registers since they don't interfere with any other variables.

So, one possibility is R1 = a, d, e; R2 = b; R3 = c.

Question 5. (16 points) First things first. We'd like to use forward list scheduling to pick a good order for executing a sequence of instructions. For this problem, assume that we're using the same hypothetical machine that was presented in lecture and in the textbook examples. Instructions are assumed to take the following number of cycles:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2

Given the assignment statement `x=(a+b)+(c*d);`, our compiler's instruction selection phase initially emits the following sequence of instructions:

a. LOAD r1 <- a
b. LOAD r2 <- b
c. ADD r1 <- r1, r2
d. LOAD r3 <- c
e. LOAD r4 <- d
f. MULT r3 <- r3, r4
g. ADD r1 <- r1, r3
h. STORE x <- r1

Answer the following questions on the next page. You can remove this page for convenience if you like.

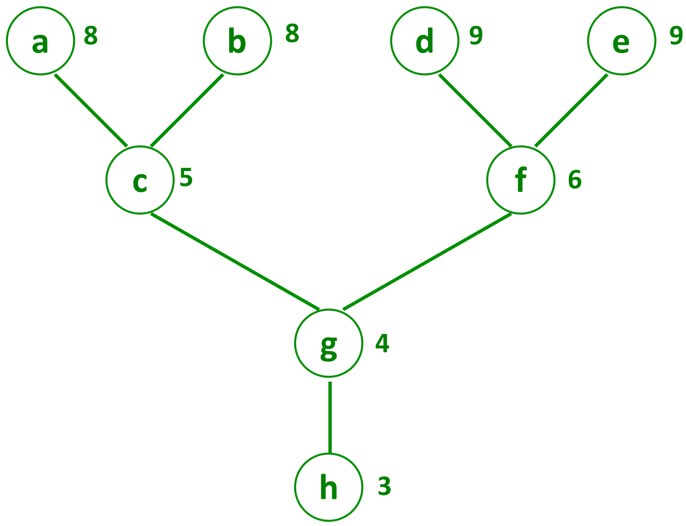
(a) (7 points) Draw the precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with the letter identifying the instruction (a-h) and it's latency – the number of cycles between the beginning of that instruction and the end of the graph on the shortest possible path that respects the dependencies.

(b) (7 points) Rewrite the instructions in the order they would be chosen by forward list scheduling (i.e., choosing on each cycle an instruction that is not dependent on any other instruction that has not yet been issued or is still executing). If there is a tie at any step when picking the best instruction to schedule next, pick one of them arbitrarily. Label each instruction with its letter from the original sequence above and the cycle number on which it begins execution. The first instruction begins on cycle 1.

You do not need to show your bookkeeping or trace the algorithm as done in class, although if you leave these clues about what you did, it could be helpful if we need to figure out how to assign partial credit.

(c) (2 points) At the bottom of the next page, write down the number of cycles needed to execute the instructions in the original order and the number needed by the new schedule.

Question 5. (cont.) (a) and (b) Draw the precedence diagram and write the new instruction schedule (sequence) below. Then fill in part (c) at the bottom of the page.



1. d LOAD r3 <- c
2. e LOAD r4 <- d
3. a LOAD r1 <- a
4. b LOAD r2 <- b
5. f MULT r3 <- r3, r4
6. (can't start anything this cycle)
7. c ADD r1 <- r1, r2
8. g ADD r1 <- r1, r3
9. h STORE x <- r1

Instructions d and e can be issued in either order, and a and b can be in either order. But d and e need to go before the other loads so the MULT can be started before the first ADD.

(c) Fill in: Number of cycles needed to completely execute all instructions in the original schedule _15_

Number of cycles needed to completely execute all instructions in the new schedule _11_

Question 6. (16 points) SSA. This Java function computes and returns the n^{th} Fibonacci number.

```
public static int fib(int n) {
  int fp, fk, k, temp;
  if (n < 2)
    return n;
  fp = 0;  // fib(0)
  fk = 1;  // fib(1)
  k = 1;   // fk = fib(k)
  while (k < n) {
    temp = fp;
    fp = fk;
    fk = fk + temp;
    k = k + 1;
  }
  return fk;
}
```

On the next page, draw a control flow graph for this function (nodes = basic blocks, edges = possible control flow), using Static Single Assignment (SSA) form as described in section. You should include ϕ-functions where needed to merge different versions of the same variable. For full credit you should only include necessary ϕ-functions and not have extraneous ones scattered everywhere, but we will give generous partial credit if all of the necessary ϕ-functions are included and there are a minimal number of extra ones.

Hint: it might be easiest to sketch the flow graph first without converting it into SSA, and then add the needed variable version numbers and ϕ-functions to get the final answer.

Write

your

answer

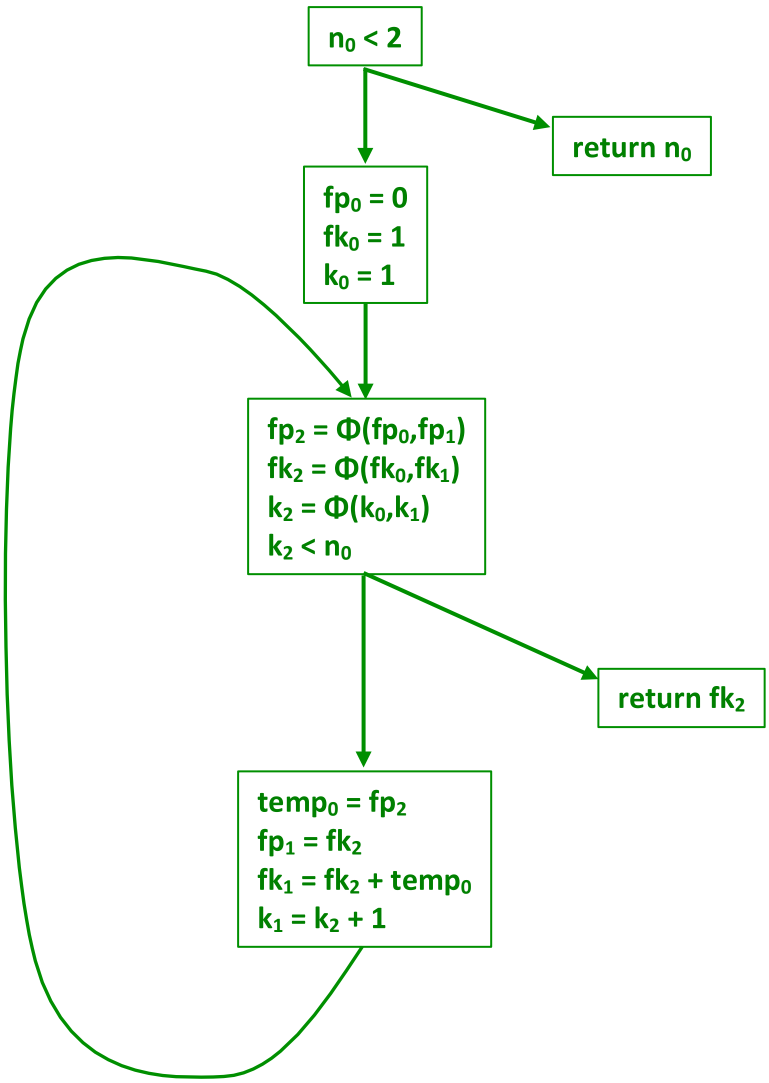
on the

next

page.

You can remove this page from the exam if you wish.

Question 6. (cont.) Write your answer here.



Question 7. (10 points, 5 each) A few short questions about memory management.

(a) In an automatic garbage collector for Java (or for most other languages), figuring out what storage is reachable, and therefore must be live, starts by examining the contents of the *root set*. What exactly *is* the root set?

The root set is the collection of all directly known variables in the program that could potentially refer to heap objects. These include all global variables, all static variables in classes or other scopes, and all variables in active functions, including local variables in stack frames and temporary variables stored in registers or other storage locations.

(b) A garbage collector for Java can be *precise* or *accurate*, meaning that it can identify all previously allocated memory that is no longer in use (garbage) and automatically reclaim it. Supposedly this cannot be done for languages like C or C++, so the best that can be done for those languages is to implement a *conservative* or approximate garbage collector. What is it about C/C++ that prevents us from implementing a precise garbage collector, and what does it mean to have a conservative collector? (Briefly please)

A collector for Java has precise information about types and locations of all heap objects because it is impossible to allocate or create a pointer to a heap object without going through the memory manager.

A C/C++ program on the other hand can create pointers to arbitrary locations in memory, including pointers into the heap that don't refer to locations of objects allocated by the memory manager. Also, given a pointer, we do not have guarantees about the type of the object that it references. Any attempt to automatically collect storage in this environment has to treat any bit pattern that potentially could be used as a pointer into heap storage as if it were one.

Have a great spring break!
The CSE 401 staff