CS314 Spring 2017
March 3

Due Friday, February 24, 11:59pm
submission: pdf file through sakai.rutgers.edu

1 Virtual and Physical Registers

In our first project, we use the following RISC instructions as the target language of our compiler. The compiler uses the convention (code shape) that every computed or loaded value is copied into a "fresh", virtual register. In the backend of a compiler, these virtual registers have to be "mapped" to physical registers, since an executable machine code can only contain the limited set of physical register names of the target machine architecture.

instr. format	**description**	**semantics**
	memory instructions	
`loadI` $c \Rightarrow r_x$	load constant value $c$into register r_x	$r_x \leftarrow c$
`loadAI` r_x, $c \Rightarrow r_y$	load value of MEM(r_x+ c) into r_y	$r_y \leftarrow$ MEM(r_x+ c)
`storeAI` $r_x \Rightarrow r_y$, c	store value in $r_x$into MEM(r_y+ c)	MEM(r_y+ c) $\leftarrow r_x$
	arithmetic instructions	
`add` r_x, $r_y \Rightarrow r_z$	add contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x + r_y$
`sub` r_x, $r_y \Rightarrow r_z$	subtract contents of register $r_x$from register r_y, and store result into register r_z	$r_z \leftarrow r_x - r_y$
`mult` r_x, $r_y \Rightarrow r_z$	multiply contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x * r_y$
`div` r_x, $r_y \Rightarrow r_z$	divide contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x / r_y$
	I/O instruction	
`outputAI` r_x, c	write value of MEM(r_x+ c) to standard output	print(MEM(r_x+ c)

1. Our project compiler generates code using virtual registers. What is the minimal number of physical registers needed to map any compiler-generated RISC code (see listed instructions) with virtual registers into a program that uses only physical registers?

   Describe a basic strategy that maps the virtual registers of a program to a program with the minimal number of physical registers.

   Your strategy has to replace virtual registers in the RISC code written as `r0` , `r1` , `r2` etc by physical registers written as `rA` , `rB` , `rC` etc. By definition, `r0` is mapped to register `rFP`, the register that contains the frame pointer (base address). Your strategy can only replace virtual by physical register names in the instructions, and cannot change the instructions in any other way. However, your strategy may introduce additional instructions to save values in virtual registers into memory in order to reuse a physical register. This additional code is called "spill code". The idea of "spilling" virtual registers into memory is similar to the idea of paging in virtual memory where virtual pages are mapped to physical pages based on the locality in the program (working set). Virtual registers can "live" in memory locations at a compiler-determined address, and can be loaded into physical registers when their value needs to be accessed. In contrast to virtual memory and paging, this swapping-in and swapping-out of virtual registers is managed by the compiler, not the hardware and operating system.

2. The following sequence of instructions is generated by the provided sample compiler for the tinyL program in file `test2`.

```
loadI 1024 => r0
loadI 5 => r1
storeAI r1 => r0, 0
loadI 1 => r2
loadI 2 => r3
loadAI r0, 0 => r4
add r3, r4 => r5
add r2, r5 => r6
loadI 5 => r7
mult r6, r7 => r8
loadI 8 => r9
sub r8, r9 => r10
storeAI r10 => r0, 4
outputAI r0, 4
```

   Give a version of the above code that uses as few physical registers as possible **without** introducing any spill code. So you are only allowed to replace virtural register names by physical register names. Remember: Virtual register `r0` has to be mapped to physical register `rFP`.

2 Problem — Pointers

Given the following correct program in C,

1. give the correct type definitions for pointer variables `ra, rb, rc, raa, rbb, and rcc`.
2. draw a picture that shows all of the variables and their contents similar to the picture as shown in lecture 11. Your picture should show the variables and their values just before the first print statement (*).
3. show the output from this program.
4. write a statement involving a pointer expression using the variables in this program which is ILLEGAL given your declared types.

```
main() {int a, b, c;
        ??? ra; ??? rb; ??? rc; ??? raa; ??? rbb; ??? rcc;
        a = 3; b = 2; c = 1;
        ra = &a;
        rb = &b;
        rc = &c;
        ra = rb;
        raa = &rb;
        rc = *raa;
        rcc = raa;
        rc = &a;
        rbb = &rc;
        rb = &c;
        *ra = 4;
        *rb = *ra + 4;
        /* (*) */
printf ("%d %d %d\n",a,b,c);
printf("%d %d\n",*ra,*rb);
printf("%d %d %d\n",**raa,**rbb,**rcc);
}
```

3 Problem — Freeing Memory

Here is a code fragment from our singly-linked list example from class.

```
/* DEALLOCATE LIST */
for (current_cell = head;
     current_cell != NULL;
     current_cell = current_cell->next)
  free(current_cell);
```

1. Is there a safety issue with this code? Explain.
2. How can you rewrite this code to make it safe? You can introduce new variables, if needed.

4 Compiler Optimization and Aliasing

Assume the following program fragement without any control flow branches (straight line code). Your job is it to implement a compiler optimization called "constant folding" for straight line code. This optimization identifies program variables with values that are known at compile time. Expressions that consist of only such variables can be evaluated at compile time.

```
begin
  int a, b, c;
  ... /* some other declarations */
  a = 5;
  b = 7;
  ... /* no statements that mention ‘‘a’’ or ‘‘b’’ */
  c = a + b; /* c == 12 ? */
  print c;
end.
```

Would it always be safe for the compiler optimization of constant folding to replace the assignment "`c = a + b`" by "`c = 12`" ? Note that there are no assignments to variables `a` or `b` between "`b = 7`" and "`c = a + b`". The control flow is linear, so there are no branches. Give an example where constant propagation would be not be safe (incorrect) in this situation, without violating any of the above assumptions about the code fragment. Note: You can add declarations of other variables and other statements that do not mention `a` or `b`.