Computation Structures

Compilation Worksheet

compile_expr(expr) ⇒ Rx

- Constants: `1234` ⇒ Rx
  - `CMOVE(1234,Rx)`
  - `LD(c1,Rx)`
    …
    `c1: LONG(123456)`
- Variables: `a` ⇒ Rx
  - `LD(a,Rx)`
    `...`
    `a: LONG(0)`
- Variables: `b[expr]` ⇒ Rx
  - `compile_expr(expr)⇒Rx`
    `MULC(Rx,bsize,Rx)`
    `LD(Rx,b,Rx)`
    …
    `// reserve array space`
    `b: . = . + bsize*blen`
- Operations: $expr_1$ + $expr_2$ ⇒ Rx
  - `compile_expr(`$expr_1$`)⇒Rx`
    `compile_expr(`$expr_2$`)⇒Ry`
    `ADD(Rx,Ry,Rx)`
- Procedure call: `f(`e_1, e_2, …`)` ⇒ Rx
  next lecture!
- Assignment: `a=expr` ⇒ Rx
  - `compile_expr(expr)⇒Rx`
    `ST(Rx,a)`

compile_statement(…)

- Unconditional: *expr*;
  - `compile_expr(expr)`
- Compound: { *s1; s2;* … }
  - `compile_statement(s1)`
    `compile_statement(s2)`
    …
- Conditional: `if (expr) s1;`
  - `compile_expr(expr)⇒Rx`
    `BF(Rx,Lendif)`
    `compile_statement(s1)`
    `Lendif:`
- Conditional: `if (expr) s1; else s2;`
  - `compile_expr(expr)⇒Rx`
    `BF(Rx,Lelse)`
    `compile_statement(s1)`
    `BR(Lendif)`
    `Lelse:`
    `compile_statement(s1)`
    `Lendif:`
- Iteration: while (*expr*) *s1*;
  - `BR(Ltest)`
    `Lwhile:`
    `compile_statement(s1)`
    `Ltest:`
    `compile_expr(expr)⇒Rx`
    `BT(Rx, Lwhile)`
- Iteration: `for (init; test; incr) s1;`
  init;
  while (*test*) { *s1; incr;* }

Problem 1.

Please hand-compile the following snippets of C code into equivalent Beta assembly language statements. Assume that memory locations have been allocated for the all C variables with labels that corresponds to the variable names. So to load the value of the C variable `a` into register R3, the appropriate assembly language statement would be `LD(R31,a,R3)`. And to store the value in R17 to the C variable `b`, the appropriate assembly language statement would be `ST(R17,b,R31)`. Similarly, assume that memory locations have been allocated for each C array, with a label defined whose value is the address of the 0th element of the array.

(A)
```
a = 42;
```

```
CMOVE(42,R0)
ST(R0,a,R31)
```

(B)
```
c = 5*x - 13;
```

```
CMOVE(5,R0)
LD(x,R1)
MUL(R0,R1,R0)
CMOVE(13,R1)
SUB(R0,R1,R0)
ST(R0,c)
```

Optimized:

```
LD(x,R1)
MULC(R0,5,R0)
SUBC(R0,13,R0)
ST(R0,c)
```

(C)
```
y = (x - 3)*(y + 123456);
```

```
LD(x,R0)
SUBC(R0,3,R0)
LD(y,R1)
LD(const,R2)
ADD(R1,R2,R1)
MUL(R0,R1,R0)
ST(R0,y)
```

```
// mem locn to hold
// large constant
const: LONG(123456)
```

(D)
```
if (a == 3) b = b + 1;
```

```
LD(R31,a,R0)
CMPEQC(R0,3,R0S)
BF(R0,L1)
LD(R31,b,R0)
ADDC(R0,1,R0)
ST(R0,b,R31)
L1:
```

(E)
```
a[i] = a[i-1];
```

```
LD(i,R0)
MULC(R0,4,R0) // 4 bytes/element
LD(R0,a-4,R1) // sub 4 at assy time!
ST(R1,a,R0)
```

(F)
```
x = y[3] + y[12];
```

```
LD(y+4*3,R0)
LD(y+4*12,R1)
ADD(R0,R1,R0)
ST(R0,x)
```

(G)
```
if (b == 0 || b < min) {
    min = b;
} else {
    too_big += 1;
}
```

```
LD(B,R0)
BEQ(R0,L2)
LD(min,R1)
CMPLT(R0,R1,R1)
BF(R1,L3)
L2:
ST(R0,min)
BR(L4)
L3:
LD(too_big,R0)
ADDC(R0,1,R0)
ST(R0,too_big)
L4:
```

(H)
```
sum = 0;
i = 0;
while (i < 10) {
    sum = sum + i
    i = i + 1;
}
```

Unoptimized:

```
ST(R31,sum)
ST(R31,i)
L5:
LD(sum,R0)
LD(i,R1)
ADD(R0,R1,R0)
ST(R0,sum)
LD(i,R0)
ADDC(R0,1,R0)
ST(R0,i)
L6:
LD(i,R0)
CMPLTC(R0,10,R1)
BT(R2,L5)
```

Problem 2.

In block-structured languages such as C or Java, the scope of a variable declared locally within a block extends only over that block, i.e., the value of the local variable cannot be accessed outside the block. Conceptually, storage is allocated for the variable when the block is entered and deallocated when the block is exited. In many cases, this means the compiler if free to use a register to hold the value of the local variable instead of a memory location.

Consider the following C fragment:

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+1) sum += i;
}
```

A. Hand-compile this loop into assembly language, using registers to hold the values of the local variables "i" and "sum".

```
  MOVE(R31,R2)  // sum
  ST(R2,sum)
  MOVE(R31,R1)  // i
L7:
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  CMPLTC(R1,10,R0)
  BT(R0,L7)
  ST(R2,sum)
```

B. Define a *memory access* as any access to memory, i.e., instruction fetch, data read (LD), or data write (ST). Compare the number of total number of memory accesses generated by executing the optimized loop with the total number of memory access for the unoptimized loop (part H of the preceding problem).

```
Part (H): each iteration
  10 instruction fetches
  6 data memory accesses
10 iterations => 160 accesses
+ 4 from first two insts => 164
```

```
Part (A): each iteration
  4 instruction fetches
  0 data memory accesses
10 iterations => 40 accesses
+ 6 from before/after loop => 46
```

C. Some optimizing compilers "unroll" small loops to amortize the overhead of each loop iteration over more instructions in the body of the loop. For example, one unrolling of the loop above would be equivalent to rewriting the program as

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+2) {
      sum += i; sum += i+1;
  }
}
```

Hand-compile this loop into Beta assembly language and compare the total number of memory accesses generated when it executes to the total number of memory accesses from part (1).

```
  MOVE(R31,R2)  // sum
  ST(R2,sum)
  MOVE(R31,R1)  // i
L7:
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  CMPLTC(R1,10,R0)
  BT(R0,L7)
  ST(R2,sum)
```

```
Part (C): each iteration
  6 instruction fetches
  0 data memory accesses
5 iterations => 30 accesses
+ 6 from before/after loop => 36
```

Problem 3.

Which of the following Beta instruction sequences might have resulted from compiling the following C statement? For each sequence describe the value that does end up as the value of y.

```
int x[20], y;
y = x[1] + 4;
```

A.
```
LD (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Not this one. If x[0] is stored a location x, x[1] is stored at location x+4 since array element takes one word (4 bytes). Program exception trying to read from a location whose address is not a multiple of 4.

B.
```
CMOVE (4, R0)
ADDC (R0, x + 4, R0)
ST (R0, y, R31)
```

Not this one. The second instruction adds the *address* of x[1] to R0, not the contents of x[1].

C.
```
LD (R31, x + 4, R0)
ST (R0, y + 4, R31)
```

Not this one. This stores x[1] in the location *following* the one word of storage allocated for y.

D.
```
CMOVE (4, R0)
LD (R0, x, R1)
ST (R1, y, R0)
```

Not this one. It's equivalent to (C).

E.
```
LD (R31, x + 4, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Yes!!! This one…

F.
```
ADDC (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Not this one. The ADDC instruction loads the address of x plus 1 into R0, then increments it by 4. So y gets the value "x+5" where x is address of the 0^{th} element of the array.

6.004 Computation Structures
Spring 2017