Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Software Development
Assignment 4 — Machine Code Generation

Due: Monday 30th May @ Mid-night

Overview

This project is concerned with *x86 machine code generation*. The x86 machine architecture presents a number of interesting challenges, such as limited registers with non-uniform, overlapping representations. Furthermore, generating machine code itself is more challenging than for e.g. JVM bytecode. To help with this, the `jx86` library is provided for writing x86 assembly language (which e.g. can then be compiled with `gcc`). You can download this library from the SWEN430 homepage, and you can view the library source at: `http://github.com/DavePearce/jx86`.

You can download the latest version of the compiler from the SWEN430 homepage, along with the supplementary code associated with this assignment. *We recommend that you start from a fresh version of the compiler, rather than any extensions from previous assignments.* Initially, you should find a large number of all tests currently fail and, furthermore, that a number are marked as *ignored*. For the purposes of this assignment, those which are ignored are not being considered. However, the remainder should all pass when the assignment is complete.

NOTE: you should not attempt to modify the `jx86` library in anyway, and your program should compile against the stock version provided on the course homepage.

NOTE: the supplementary code includes a *runtime library*, `whilelang/runtime/runtime.c`, which is written in the C language. You are not permitted to modify this library.

1 While Loops (20%)

The translation of `while` loops is not currently implemented in the compiler. The following illustrates a simple example which counts up to a given number `n`:

```
int count(int n) {
  int i = 0;
  while(i < n) {
    i = i + 1;
  }
  return i;
}
```

Your first objective is to implement While loops in the language, whilst ignoring `break` and `continue` for now. You should find the existing implementation of `for` loops to be quite helpful here. The above program should compile into machine code which resembles the following:

```
wl_count:
    pushq %rbp
    movq %rsp, %rbp
    subq 16, %rsp
    movq 0, %rbx         //
    movq %rbx, -8(%rbp)  // i = 0
label1:
    movq -8(%rbp), %rbx  //
    movq 24(%rbp), %rcx  //
    cmpq %rcx, %rbx      //
    jge label2           // i >= n
    movq -8(%rbp), %rbx  //
    movq 1, %rcx         //
    addq %rcx, %rbx      //
    movq %rbx, -8(%rbp)  // i = i + 1
    jmp label1
label2:
    movq -8(%rbp), %rbx  //
    movq %rbx, 16(%rbp)  //
    jmp label0           // return i
label0:
    movq %rbp, %rsp
    popq %rbp
    ret
```

In the above, we have annotated the x86 assembly language to indicate which statements in the While program each instruction corresponds with.

2 Break and Continue (20%)

The translation of `break` and `continue` statements is not currently implemented in the compiler. The following illustrates a simple example:

```
int f(int n) {
  switch(n) {
    case 0:
      break;
    default:
      n = n + 2;
  }
  return n;
}
```

The next objective is to update the compiler to support these statements. To do this, break and continue statements should be implemented as *unconditional branching instructions* (i.e. `jmp`). You will need to modify the `Context` to include the necessary targets for `break` and `continue`.

3 Record Assignment (20%)

The next step of the assignment is to implement *record assignment*. The following illustrates a simple example:

```
void f({int x, int y} record) {
  record.y = 3;
}
```

This shows a single assignment to the field `y` of `record`. The above program should compile into machine code which resembles the following:

```
wl_f:
    pushq %rbp
    movq %rsp, %rbp
    movq %rbp, %rax    //
    addq 16, %rax     //
    addq 8, %rax      //
    movq 3, %rbx      //
    movq %rbx, 0(%rax) // record.y = 3
label0:
    movq %rbp, %rsp
    popq %rbp
    ret
```

They key here is the use of an instruction writing a register (`rbx` above) into an *indirect location* determined by another register (`rcx` above) with an immediate offset *locating the field*.

4 Integer Arrays (40%)

The implementation of arrays in While on the x86 architecture is challenging. In particular, supporting arrays or arbitrary type requires considerable effort. To simplify things, we will restrict ourselves to `int[]` arrays which have a uniform structure. Since arrays correspond to *dynamically sized* structures, they cannot be allocated on the stack. Instead, we must allocate them on the heap and the method `allocateSpaceOnHeap()` is provided for doing this. To implement arrays, we recommend the following steps:

1. **Array Initialisers**. These are necessary in order to perform any operations on arrays. An appropriate amount of memory needs to be allocated, and then every element in the initialiser evaluated and assigned into the array. One important consideration is the *layout* of your array. In particular, arrays will need to contain length information. We recommend that the first word of the array is reserved for holding the length.

2. **Array Generators**. At this point, implementing array generators should be very straightforward. To help with this, a method `intnfill()` is provided in the runtime library.

3. **Array Length**. This is a simple operation which should read out the length of an array from its location in the underlying data structure.

4. **Array Equality**. In order for many assertions to pass, you will need to implement array equality. To help with this a method `intncmp()` is provided in the runtime library.

5. **Array Access**. The ability to read an element from an array using access notation (e.g. `xs[i]`) is relatively straightforward to implement. The key difference from reading a field from a record is that the location of the array element is not known statically. Therefore, you will need to use an instruction involving an operand of the form `(rdi,rax,w)` where `rdi` points to the base of the array, `rax` gives the index into that array and `w` gives the width (in bytes) of an array element.

6. **Array Assignment**. Assigning an element into an integer array is made challenging because the semantics of WHILE dictate that arrays have *value semantics*. Therefore, when assigning into an array, you should *clone the entire array*. To help with this a method `intncpy()` is provided in the runtime library.

NOTE: Although arrays are allocated on the heap, we will not attempt to deallocate them. This is challenging in a language like WHILE which has no explicit deallocation operator. Instead, one could use *reference counting* — but this is beyond the scope of this assignment!

NOTE: You can debug arrays by printing out their contents with the `print` statement. Internally, this calls the method `prnintn` in the runtime library.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.*

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be awarded based on the number of passing tests, as determined by the *automated marking system*. **NOTE:** the test cases used for marking will differ from those in the supplementary code provided for this assignment. In particular, they may constitute a more comprehensive set of test cases than given in the supplementary code.