Third Weekly IPS Assignment

This is the text of the third weekly assignment for the DIKU course "Implementation of Programming Languages", 2023.

Hand in your solution in the form of a short report in text or PDF format.

Task 1

This task refers to intermediate and machine code generation. Following the translation rules in the textbook (**not** the simplified variant for assignment statements in the lecture slides) as closely as possible, translate the following statement first to intermediate code and then to MIPS (**not** RISC-V) code (still with symbolic register names), assuming `vtable = [a ↦ v0, b ↦ v1]`:

```
while !(b = 0) && (1 < a/b) do {
    if b < a then { a := a - b }
             else { b := b - a }
}
```

For the first part, do not worry too much about the exact order in which fresh IL variables and labels are generated, i.e., their numbers, as long as the translation is correct. For the second, assume that the IL→MIPS pattern/replacement pairs for subtraction and division are analogous to those for addition, but using the instructions `sub` and `div`, respectively.) Again: don't try to "hand-optimize" the generated code in either pass; show exactly what is produced by the formal translation rules.

Note: The answer for both parts should be just the actual sequence of generated instructions (IL and MIPS, respectively). You do not have to show or explain in detail how you obtained the result by applying the translation functions or code-pattern rules, though you are welcome to include brief comments or annotations to elucidate any subtle points about how translation output relates to the input.

Task 2

This task refers to machine code generation, and specifically to constructing the code-generation rules for a new architecture.

Make **RISC-V** pattern/replacement pairs as in the "5 - Machine Code Generation" lecture for each of the following two IL instructions.

- z := x ≥ y
- w := ! z

where `x`, `y`, `z` and `w` are assumed to be variables/registers of integer type, and

- `≥` is an operator that returns 1 if its first argument is greater or equal than the second, and 0 if it is not, and
- `!` is an operator that returns 1 if its argument is 0, and 0 if its argument is different from 0.

Also, make rules for the combined sequence of the two operations that leads to a more efficient translation than using the individual rules for each, both for the case where `z` is subsequently used, and where its not, i.e., for the IL patterns:

```
z := x ≥ y
w := ! z
```

and

$z := x \geq y$
$w := !\, z^{last}$

The solution of this task should be a table in which you put all patterns and translations, as in Fig. 7.1 in the textbook.

Use only the subset of RISC-V supported by `RiscV.fs` in the Fasto compiler. Try to use as few instructions and registers as possible in the translation, and avoid jumps if you can.

Task 3 (Mainly to help you with the project)

This task refers to RISC-V code generation for second-order array combinators (SOACs), and is intended to help you with the project — a very similar example has been presented in lecture notes 5 - Machine Code Generation, slide `26`, which solves `map(f,a)`.

In group project you will have to implement `scan`, which has type `(a*a→a) * a * [a] → [a]`, with the semantics as specified in the project document. Assume that a function `myop : bool*bool → bool` is available (known), and that variables `ne : bool` and `x : [bool]` are visible in the current scope. Assume also that a word has 4 bytes, that each `bool`-array element is represented as one byte in memory, and that heap memory can only be allocated in chunks whose sizes are (integer) multiples of 4 bytes.

- a.) Assuming that the length of array `x` is `n`, translate the call `y = scan(myop, ne, x)` into a simple C-like language with while loops. Try to keep this language, morally, as close as possible to our IL in the book and slides. To do so, you may use `malloc(m)` to allocate a chunk of `m` bytes for the result array `y`.

Hint: For example the C-like code may have a structure that resembles:

```
char* y     = (char*)malloc( ... n ... );
int   i     = 0;
...
while (i < n) { ... }
```

- b.) Assuming `myop` to be a function name, write the RISC-V code for `scan(myop, ne, x)` using the most efficient translation you can find, and using the array layout from the Group Project, i.e. the first word of the array representation (for both the input and the output array) holds the length of the array, followed by the content of the array (the array's elements) in individual bytes. Place the scan result (a pointer) in symbolic register `regy` and assume that:
  - vtable is `[x ↦ regx, ne ↦ regne]`
  - your code may use additional symbolic registers, such as `regX` to represent each C variable `X`, as well as temporaries `regt1`, `regt2`, etc. for holding intermediate calculation results that have no associated names in the C code.
  - (physical) register `Rhp` is the heap pointer, holding the first address of non-allocated memory, and which must always stay word-aligned
  - you have a RISC-V (pseudo)instruction that calls a function, e.g., `res = CALL fname(`arg_1`,…,`arg_n`)` calls the function named `fname` on the (list of) actual-argument registers `arg`$_1$`,…,arg`$_n$ and places the result in register `res`.

(Hint: traverse the flat array representation while maintaining an accumulator for the last-computed element. For the RISC-V code that allocates the heap memory space for an array with `n` elements, you may take a look at function `dynalloc` in `CodeGen.fs`. The solution should be short; `map` took only one slide (number 26) in 5 - Machine Code Generation lecture notes, but you should mention at least which part of the machine code corresponds to the C pseudocode — we have used colors in the slides, you can just provide comments.)

(In the Group Project, you will also need to handle scans over arrays with element types other than `bool`, including word-sized ones, and employ the existing machinery to invoke an arbitrary combining operation, whether provided to `Scan` as a function name, or as an anonymous function.)