CSC488
ASSIGNMENT 5
ASSEMBLER

DANIEL BLOEMENDAL

Contents

1. Overview

The first goal, before beginning work on code generation, was to implement an assembler for our IR instruction set. This included instructions with label operands, instead of static addresses, and each macro, or IR instruction, outlined in the code generation templates. The entire IR instruction set will be covered again in detail in this document. There are also a few notable bugs that were fixed since the code generation templates in A4.

The assembler was designed to be decoupled from the code generator. It is standalone and can even be executed from command line on IR assembly programs. A run script is provided along with an "`ant`" command to build the assembler. To build the assembler, run "`ant assembler`". To execute the assembler from command line, run "`./RUNASSEMBLER.sh <source.ir>`"

2. Grammar

The following is the grammar for assembly programs in EBNF.
Please refer to http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form for more information.

⟨*program*⟩ ::= { ⟨*section*⟩ }

⟨*section*⟩ ::= '`SECTION`', '`.`', ⟨*ident*⟩, '`\n`',
{ [⟨*label*⟩], [⟨*instruction*⟩], '`\n`' }

⟨*label*⟩ ::= ⟨*ident*⟩, '`:`'

⟨*instruction*⟩ ::= (⟨*machine_operation*⟩ | ⟨*ir_operation*⟩), { ⟨*operand*⟩ }

⟨*machine_operation*⟩ ::= '`HALT`' | '`ADDR`' | '`LOAD`' | '`STORE`' | '`PUSH`' | '`PUSHMT`' | '`SETD`' | '`POP`' | '`POPN`' | '`DUP`' | '`DUPN`' | '`BR`' | '`BF`' | '`NEG`' | '`ADD`' | '`SUB`' | '`MUL`' | '`DIV`' | '`EQ`' | '`LT`' | '`OR`' | '`SWAP`' | '`READC`' | '`PRINTC`' | '`READI`' | '`PRINTI`' | '`TRON`' | '`TROFF`' | '`ILIMIT`'

⟨*ir_operation*⟩ ::= '`PUSHSTR`' | '`SETUPCALL`' | '`JMP`' | '`BFALSE`' | '`NOT`' | '`SAVECTX`' | '`RESTORECTX`' | '`RESERVE`'

⟨*operand*⟩ ::= ⟨*value_operand*⟩
| ⟨*boolean_operand*⟩
| ⟨*string_operand*⟩

⟨*value_operand*⟩ ::= ⟨*integer_operand*⟩
| ⟨*label_operand*⟩

⟨*integer_operand*⟩ ::= ['`-`'], ⟨*digit*⟩, {⟨*digit*⟩}

⟨*boolean_operand*⟩ ::= '`$true`' | '`$false`'

⟨*label_operand*⟩ ::= ⟨*ident*⟩

⟨*string_operand*⟩ ::= '`"`', { ? all characters ? }, '`"`'

⟨*ident*⟩ ::= ⟨*letter*⟩, {⟨*letter*⟩ | ⟨*digit*⟩}

⟨*letter*⟩ ::= '`a`'..'`z`' | '`A`'..'`Z`' | '`_`'

⟨*digit*⟩ ::= '`0`'..'`9`'

3. Labels

The biggest difference between the IR assembly language and the assembly language built into `Machine` is the support for labels. Labels can be thought of as instructions of with zero size and no operation. The addresses of labels are stored as instructions are assembled. Afterwards, any label operand pointing to a valid label is set to resolve to the computed address of the label it points to.

4. Instructions

In addition to support for labels, the assembler also provides an extended instruction set. The following instructions outlined below are not available in `Machine`.

4.1. **`PUSHSTR(`**⟨`string_operand`⟩ `string`**`).`** This instruction adds "`string`"to the text constant pool, if it has not already been added, and obtains the address of the string in the pool. The address is then pushed on to the stack.

4.2. **`SETUPCALL(`**⟨`value_operand`⟩ `return_address`**`).`** `SETUPCALL` sets up a call, according to calling convention designed for the compiler, by pushing a place holder for the return value, and by pushing `return_address`, the return address for the call. The placeholder for the return value is set to be `Machine.UNDEFINED`, to support returning from a function or procedure without an explicit "`return`" or "`result`" statement. For a more detailed discussion on the calling convention, see the code generation documentation.

```
PUSH UNDEFINED
PUSH return_address
```

4.3. **`JMP(`**⟨`value_operand`⟩ `address`**`).`** `JMP` pushes `address` on to the stack and then performs a branch.

```
PUSH address
BR
```

4.4. **`BFALSE(`**⟨`value_operand`⟩ `address`**`).`** Like `JMP`, `BFALSE` is also a short hand for pushing an address and then branching. This time, the branch is the conditional `BF`.

```
PUSH address
BF
```

4.5. **`NOT.`** Unfortunately the machine has no built in method of negating boolean values. `NOT` provides this much needed functionality. It does so by comparing the top of the stack with `$false`. To see this, suppose *top* = `$true`, then (`$true` = `$false`) = `$false` on the other hand if *top* = `$false` then (`$false` = `$false`) = `$true`

```
PUSH $false
EQ
```

4.6. `SAVECTX(`⟨`value_operand`⟩ `lexical_level). SAVECTX` is used at the beginning of a major scope to preserve any previous display set by a major scope at the same lexical level, a sibling scope. After preserving the display, it sets the display to the current address stored in the stack pointer. This is part of the calling convention and is explained in more detail in the code generation documentation.

```
ADDR lexical_level 0
PUSHMT
SETD lexical_level
```

4.7. `RESTORECTX(`⟨`value_operand`⟩ `lexical_level,` ⟨`value_operand`⟩ `argument_count). RESTORECTX` is used at the end of major scope restore the display saved by `SAVECTX`. It also pops `argument_count` arguments off of the stack. Again, for more details on the calling convention, refer to the code generation documentation.

```
SETD lexical_level
PUSH argument_count
POPN
```

4.8. `RESERVE(`⟨`value_operand`⟩ `words). RESERVE` is used to reserve `words` of memory on the stack. It is current used in the prolog of a major scope to reserve memory for locals.

```
PUSH 0
PUSH words
DUPN
```

5. Example

The example below is the currently used runtime library for the code generator. It contains only one function right now, `print`. It serves as a good demonstration of the IR assembly, using a number of the different IR instructions.

```
1  ; ------------------------------------
2  ; Start of runtime library
3  ; ------------------------------------
4      SECTION .library
5
6  print:
7      SAVECTX 0
8      PUSH 0
9  __print_start:
10     DUP
11     ADDR 0 -2
12     LOAD
13     LOAD
14     LT
15     PUSH __print_end
16     BF
17     DUP
18     ADDR 0 -2
```

```
19      LOAD
20      PUSH 1
21      ADD
22      ADD
23      LOAD
24      PRINTC
25      PUSH 1
26      ADD
27      PUSH __print_start
28      BR
29  __print_end: POP
30      RESTORECTX 0 1
31      BR
32  ; -----------------------------------
33  ; End of runtime library
34  ; -----------------------------------
```

6. Error handling

The machine has a number of nontrivial limitations that are easy to run into. The limited memory and 16 bit operands ensure that it is easy to hit the limits. The assembler is careful to perform bounds checking on integer operands, enforcing the `Machine.MIN_INTEGER` and `Machine.MAX_INTEGER` bounds. In addition to integer bounds checking, the assembler also will gracefully handle programs that exceed the available machine memory `Machine.maxMemory`. An exception will be thrown when these errors are encountered.