BRB Specification
Big Red Bytecode

Susan Garry, Charles Sherk

October 14, 2023

1 Overview

We provide a description of a bytecode for bril. We support the standard extensions, and make it relatively easy to extend the bytecode to support new extensions as well. We also provide a reference for the opcodes that we have chosen in appendix A. The general encoding is that each instruction is 64 bits, but some instructions require more information. These instructions encode in some multiple of 64 bits which can be determined by reading the first 64 bits.

2 Base Instructions

Temporaries (stack variables) are represented by a number in the range $[0, 2^{16})$. This does technically impose a limit over the original bril implementation, but we don't believe that it will be relevant for any programs. Since all data types take up 8 bytes in our implementation, this comes out to half a megabyte of space maximum for each stack frame. In java, the maximum stack size is 1 megabyte, so this seems like a reasonable limitation.

We also limit the number of instructions to 2^{16} per function so that labels can be represented in 16 bits as instruction indices. This allows us to use the following general instruction format:

	`labelled`	`opcode`	`dest`	`arg1`	`arg2`
Size (bits):	1	15	16	16	16

The `labelled` bit marks if an instruction has a label immediately preceding it, that is, it can be a jump target. This is important for faithful execution of the ϕ instruction, as it needs to keep track of the most recent label.

The `opcode` is 15 bits, which is far bigger than we need, but it makes sense to keep everything in 16 bit increments. There is space for another couple bits of tagging extensions after the label bit since we don't use many opcode bits, but there is no need for them yet.

Next is the `dest` temporary. This is where the result of this instruction will be put. `arg1` and `arg2` are the two arguments. This works for all instructions of the form

$$\texttt{dest} := \texttt{arg1}\ \textit{op}\ \texttt{arg2}$$

There are however other formats.[1]

[1] Not EVERY instruction uses the second argument, such as the `not` instruction, so in this case the second argument is undefined.

3 Types

We can represent the bril types `int`, `float`, `bool`, `void`, and arbitrarily nested pointers to these. The 2 bit encodings for base types are in appendix B.

	`ptr depth`	`base type`
Size(bits)	14	2

This means something like `ptr<ptr<ptr<bool>>>` would be represented as `0b1101`. The right two bits are the base type, and the rest to the left is how many `ptr` there are.

4 ID instruction

The `ID` instruction only has one argument and a destination, leaving us 16 bits unused. However, in order to facilitate translation back to the original bril code, we need to be able to write down types, and `ID` is a polymorphic instruction, so we hold the type in this last slot, giving this layout:

	`labelled`	`opcode`	`dest`	`arg`	`type`
Size (bits):	1	15	16	16	16

5 Branch Instructions

Branches are the simplest instructions that don't fit into the "base" format, and are laid out as follows:

	`labelled`	`opcode`	`test`	`ltrue`	`lfalse`
Size (bits):	1	15	16	16	16

The `labelled` and `opcode` sections work the same way as base instructions (the opcode will always be the branch opcode, but we still need it to be able to decode easily). `test` is a temporary which if it is 1 will transfer control flow to `ltrue`, and if it is 0 will transfer to `lfalse`. Other values result in undefined behavior, as we only allow booleans to be 0 or 1.

6 Constant Instruction

Constants present a bit of an issue: bril supports constants up to 64 bits, but that would be the whole instruction. We also observe that usually constants are much smaller, so we provide two encodings. First of all is the "standard" encoding:

	`labelled`	`opcode`	`dest`	`value`
Size (bits):	1	15	16	32

This works for integers which can be represented in 32 bits as well as booleans, but for larger integers and floats, there is a loss of precision. We also provide a "long constant" format as follows:

	`labelled`	`opcode`	`dest`	`type`	`unused`	`value`
Size (bits):	1	15	16	16	48	64

This is 128 bits instead of 64, as we use `labelled`, `opcode`, and `dest` the same way. We also include the type since we have the space, and this is necessary to translate back to bril. The next 64 bits are the constant. This opcode is distinct from the opcode for the "standard" constant instruction encoding, so it allows distinction.

7 Function Call Instruction

Bril supports function calls in a single instruction with an unlimited number of arguments, which is obviously a problem for a 64 bit encoding. We provide an initial encoding as follows:

	labelled	opcode	dest	num_args	target
Size (bits):	1	15	16	16	16

`labelled`, `opcode`, and `dest` work as in the standard encoding. `num_args` tells us the number of arguments to the function, so we know how many of the following 64 bit chunks contain them, and `target` is the function to call. Functions are represented as their indices in order they appear in the source, which makes linking multiple sources together impossible at the bytecode stage. For this reason, the outputted file includes the name of the function, so that it is possible to reparse and link multiple files.

We then have a sequence of 64 bit words that are split into 4 arguments, not all of which are used. These are represented as temps.

8 Print Instruction

Bril also includes a fairly high level `print` instruction, which deals with it's arguments differently depending on their types. In order to encode this, we need to have an encoding for the types supported by bril, which we provide in section 3. The first word of a `print` is encoded as follows:

	labelled	opcode	num_prints	type1	arg1
Size (bits):	1	15	16	16	16

As usual, `labelled` and `opcode` work as above. `num_prints` is the number of arguments to the `print` instruction, as bril supports an arbitrary amount. `type1` is the type of the first argument, and `arg1` is the encoding of the temporary. Subsequent arguments come in a sequence of 64 bit chunks which hold up to two arguments each, alternating the type and then the temp.

9 Φ Instruction

In order to properly mimic the behavior of the reference interpreter, we need to be able to faithfully execute the Φ instruction on SSA programs. This is the reason we have the bit to keep track of which instructions were labelled in the source. The Φ instruction also supports an arbitrary number of arguments, so we use multiple 64 bit words to encode it. The first is as follows:

	labelled	opcode	dest	num_choices	unused
Size (bits):	1	15	16	16	16

`labelled`, `opcode`, and `dest` work the same as a standard instruction, and `num_choices` is the number of places we might come from into the Φ instruction. Following this instruction are a sufficient number of 64 bit words to hold all the choices, which are represented as the encoding of a label followed by the value, which is a temp. Remember labels are indices into the instruction list.

Appendices

A Opcodes

base Instructions	
CONST	1
ADD	2
MUL	3
SUB	4
DIV	5
EQ	6
LT	7
GT	8
LE	9
GE	10
NOT	11
AND	12
OR	13
JMP	14
BR	15
CALL	16
RET	17
PRINT	18
LCONST	19
NOP	20
ID	21

ssa Instructions	
PHI	22

mem Instructions	
ALLOC	23
FREE	24
STORE	25
LOAD	26
PTRADD	27

float Instructions	
FADD	28
FMUL	29
FSUB	30
FDIV	31
FEQ	32
FLT	33
FLE	34
FGT	35
FGE	36

B Types

Types	
BRILINT	0
BRILBOOL	1
BRILFLOAT	2
BRILVOID	3