BASIL Architecture Specification Version 1.0.0

Introduction

BASIL (**B**ass **A**ckwards **S**tack **I**SA **L**anguage) is a "simple" RISC instruction set architecture. BASIL is a stack-based architecture; it forgoes registers and memory, leaving the stack as the only vehicle for data storage.

BASIL is designed to be easy to implement; no consideration is given to ease of *programming*.

Features

BASIL uses four-bit (henceforth referred to as a *nibble*) fixed length[1] instructions, which reduces executable size significantly. As instructions implicitly pull their arguments from the stack, the entire nibble can be used for the opcode.

The maximum length of the stack is implementation specific. BASIL assembly does not expose the stack pointer to the programmer, so it must be managed by the processor. The bit length of words on the stack is implementation specific. Undefined behavior occurs in the event of a stack underflow or stack overflow.

The program counter always points to the next instruction and is always incremented after the execution of an instruction.

BASIL has no exit instruction. For this reason, programmers are recommended end their program by infinitely branching to the same address. Allowing control to advance past the end of program data results in undefined behavior.

Hardware design is relatively undefined by this specification; a correct implementation must at minimum implement a stack, a program counter, and a storage unit for the program. Endianness is implementation specific.

BASIL is believed to be Turing complete.

Acknowledgements

BASIL was designed by Justin Lardinois and Daniel Bittman.

This specification was written by Justin Lardinois.

[1] The PUSH instruction arguably makes BASIL instructions variable length; see instruction reference for details.

Opcodes

```
0000 PUSH
0001 AND
0010 NOT
0011 OR
0100 MUL
0101 DIV
0110 ADD
0111 CMP
1000 POP
1001 SWP
1010 DUP
1011 PPC
1100 GET
1101 PUT
1110 BR
1111 reserved for future expansion
```

Alphabetical Instruction Reference

`ADD 0110`
Pops two words, treating them as signed integers, and pushes their sum.

`AND 0001`
Pops two words and pushes their bitwise AND.

`BR 1110`
Pops two words. If the first is nonzero, the program counter is set to the second.

`CMP 0111`
Pops two words, pushing a nonzero value if they are equal and 0 if they are not equal.

`DIV 0101`
Pops two words, treating them as signed integers; the first is considered the dividend and the second the divisor. Integer division is performed and the quotient is pushed.

`DUP 1010`
Pushes a word equal to the word on top of the stack.

`GET 1100`
Reads a word from an implementation specific input device and pushes the least significant byte.

`MUL 0110`
Pops two words, treating them as signed integers, and pushes their product.

`NOT 0010`
Pops a word and pushes its bitwise NOT.

`OR 0011`
Pops two words and pushes their bitwise OR.

`POP 1000`
Pops one word.

`PPC 1011`
Pushes the program counter.

`PUSH <imm4> 0000 XXXX`
Pushes the subsequent nibble and increments the program counter.

`PUT 1101`
Pops a word and writes it to an implementation specific output device. Only the least significant byte should be considered data; the rest of the word is reserved for future expansion.

`SWP 1001`
Pops a word *n*. The word at the top of the stack and the word *n* below it are swapped.