Z v2.0

An interpreted programming language with RISC bytecode, and c-style syntax, written in c++

Changes from v1.0

The main reason for starting a new version was to clean up some of the code for the previous one. v1.0 has a working vm, but that vm was created as I was learning how to design it, and as such has some flaws. The new version intends to take that knowledge to streamline the code.

Running It

Compiles (on my machine) in Microsoft Visual Studio Community 2019 Version 16.11.2

Currently my command line arguments are:
`z2_0.exe -debug -profile -asmandexec "file.azm" "file.eze"` for assembly files, and
`z2_0.exe -debug -profile -compile "file.z" "file.eze" -exec "file.exe"` for compiling Z files

Command	Meaning
debug	Turns on debug mode. Only affects "assemble," "exec," and "asmandexec" commands after this command.
nodebug	Turns off debug mode. Only affects "assemble," "exec," and "asmandexec" commands after this command.
profile	Turns on profile mode. Only affects "exec" and "asmandexec" commands after this command.
noprofile	Turns off profile mode. Only affects "exec" and "asmandexec" commands after this command.
assemble	Assembles the first file argument (text, .azm) into the second file argument (binary, .eze)
exec	Executes the first file argument (binary, .eze)
asmandexec	Assembles the first file argument (text, .azm) into the second file argument (binary, .eze), which is then executed
stacksize	The first argument sets the stack size for execution. Only affects "exec" and "asmandexec" commands after this command.

The Language

(TODO)

VM

Bytecode

A register-based RISC bytecode

AZM File Format

Anything in parentheses or following a semicolon until a new line is a comment. Ex `(comment...)`, `; comment...` Everything else is instructions and their arguments, with tokens separated by whitespace or commas. Anything in double-quotes is treated as a single object (whitespace and commas are not terminators/separators, some escape codes, such as `\"` for ", work). Numbers use typical formatting: ### for decimal, 0x### for hex, 0b### for binary, -### for negative numbers. Be careful of overflow. See below section **Examples** for examples.

EZE File Format

First 4 bytes: Address of first instruction
Next (?) bytes: global memory
Next (?) bytes: program

Registers

The bytecode has several general-purpose registers that can hold any word, byte, or short value (including memory addresses). There are also several special-purpose registers.

ID	Register	Purpose
N/A	IP	Instruction pointer (not accessible by program)
0	PP	Program memory pointer (points to the base of the binary file loaded into memory)
1	BP	Stack base pointer
2	FZ	Zero flag register (zero flag is stored in the boolean part)
3 ... 17	W0 .. 14	General purpose word
18 .. 31	B0 .. 13	General purpose byte

Instructions

TODO: put in all of the number codes once finalized \

Possible Arguments:
`[reg]` is a 1-byte register ID
`[off]` is a 4-byte offset used for branching
`[byte]` is a byte
`[word]` is a 4-byte word. A `[label]` or `[var]` can go in the place of any word, and the program/memory address will be inserted there. Numerical values will be formatted as integers unless they contain a decimal point, then they will be formatted as floats. (Ex: INTEGERS: 3, 4, -17; FLOATS: 3., 4.17, -19.21904)
`[label]` is a label, starting with "@", that can go in the place of any word and will be converted to a program address
`[var]` is the address of a global, starting with "%", that can go in the place of any word and will be converted to a memory address relative to PP
`[string]` is a null-terminated string

(F) means it sets the flags based on the result
{addr} means the value at address addr

Code	Instruction	Arguments	Equation	Action
0x00	nop	N/A	N/A	Nothing
0x01	halt	N/A	N/A	Halts the program
0x02	break	N/A	N/A	Triggers a breakpoint in Visual Studio for when I'm debugging
0x03	alloc	[reg1], [reg2]	N/A	Allocates memory on the heap using C++ "new". The # of bytes is the integer in [reg2], and the resulting address is stored in [reg1]
0x04	free	[reg1]	N/A	Frees the heap memory at the address in [reg1] using C++ "delete"
0x05	rmovw	[reg1], [reg2]	[reg1] = [reg2]	Copies the word from [reg2] to [reg1]
0x06	rmovb	[reg1], [reg2]	[reg1] = [reg2]	Copies the byte from [reg2] to [reg1]
0x07	movw	[reg1], [word]	[reg1] = [word]	Puts the word value [word] into [reg1]
0x08	movb	[reg1], [byte]	[reg1] = [byte]	Puts the byte value [byte] into [reg1]
0x09	loadw	[reg1], [reg2], [off]	[reg1] = {[reg2] + [off]}	Load the word at address [reg2] + [off] into [reg1]
0x0a	storew	[reg1], [off], [reg2]	{[reg1] + [off]} = [reg2]	Store the word from [reg2] at address [reg1] + [off]
0x0b	loadb	[reg1], [reg2], [off]	[reg1] = {[reg2] + [off]}	Load the byte at address [reg2] + [off] into [reg1]
0x0c	storeb	[reg1], [off], [reg2]	{[reg1] + [off]} = [reg2]	Store the byte from [reg2] at address [reg1] + [off]
0x0d	jmp	[label]	N/A	Jump to [label]
0x0e	jmpz	[label]	N/A	Jump to [label] if the zero flag is set
0x0f	jmpnz	[label]	N/A	Jump to [label] if the zero flag is not set
0x10	rjmp	[reg1]	N/A	Jump to the address in [reg1]

Code	Instruction	Arguments	Equation	Action
0x11	rjmpz	[reg1]	N/A	Jump to the address in [reg1] if the zero flag is set
0x12	rjmpnz	[reg1]	N/A	Jump to the address in [reg1] if the zero flag is not set
0x13	iflag (F)	[reg1]	N/A	Sets the flags based on the integer value in [reg1]
0x14	icmpeq (F)	[reg1], [reg2]	[reg2] == [reg3]	Checks if the integers in [reg2] and [reg3] are equal, sets the flags according to the boolean result
0x15	icmpne (F)	[reg1], [reg2]	[reg2] != [reg3]	Checks if the integers in [reg2] and [reg3] are not equal, sets the flags according to the boolean result
0x16	icmpgt (F)	[reg1], [reg2]	[reg2] > [reg3]	Checks if the integer in [reg2] is greater than that in [reg3], sets the flags according to the boolean result
0x17	icmplt (F)	[reg1], [reg2]	[reg2] < [reg3]	Checks if the integer in [reg2] is less than that in [reg3], sets the flags according to the boolean result
0x18	icmpge (F)	[reg1], [reg2]	[reg2] >= [reg3]	Checks if the integer in [reg2] is greater than or equal to that in [reg3], sets the flags according to the boolean result
0x19	icmple (F)	[reg1], [reg2]	[reg2] <= [reg3]	Checks if the integer in [reg2] is less than or equal to that in [reg3], sets the flags according to the boolean result
0x1a	iinc (F)	[reg1]	[reg1] = [reg1] + 1	Increments the integer value in [reg1] by 1
0x1b	idec (F)	[reg1]	[reg1] = [reg1] - 1	Decrements the integer value in [reg1] by 1
0x1c	iadd (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Adds the values from [reg2] and [reg3] into [reg1] as integers
0x1d	isub (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Subtracts the value in [reg3] from the value in [reg2] into [reg1] as integers
0x1e	imul (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Multiplies the values from [reg2] and [reg3] into [reg1] as integers
0x1f	idiv (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Divides the value in [reg2] by the value in [reg3] into [reg1] as integers. Throws divide by zero error if the value in [reg3] is zero.

Code	Instruction	Arguments	Equation	Action
0x20	imod (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Puts value from [reg2] modulo the value in [reg3] into [reg1] as integers. Throws divide by zero error if the value in [reg3] is zero.
0x21	itoc	[reg1], [reg2]	[reg1] = [reg2]	Casts the integer value from [reg1] to a char and puts it in [reg2]
0x22	itof	[reg1], [reg2]	[reg1] = [reg2]	Casts the integer value from [reg1] to a float and puts it in [reg2]
0x23	cflag (F)	[reg1]	N/A	Sets the flags based on the char value in [reg1]
0x24	ccmpeq (F)	[reg1], [reg2]	[reg2] == [reg3]	Checks if the chars in [reg2] and [reg3] are equal, sets the flags according to the boolean result
0x25	ccmpne (F)	[reg1], [reg2]	[reg2] != [reg3]	Checks if the chars in [reg2] and [reg3] are not equal, sets the flags according to the boolean result
0x26	ccmpgt (F)	[reg1], [reg2]	[reg2] > [reg3]	Checks if the char in [reg2] is greater than that in [reg3], sets the flags according to the boolean result
0x27	ccmplt (F)	[reg1], [reg2]	[reg2] < [reg3]	Checks if the char in [reg2] is less than that in [reg3], sets the flags according to the boolean result
0x28	ccmpge (F)	[reg1], [reg2]	[reg2] >= [reg3]	Checks if the char in [reg2] is greater than or equal to that in [reg3], sets the flags according to the boolean result
0x29	ccmple (F)	[reg1], [reg2]	[reg2] <= [reg3]	Checks if the char in [reg2] is less than or equal to that in [reg3], sets the flags according to the boolean result
0x2a	cinc (F)	[reg1]	[reg1] = [reg1] + 1	Increments the char value in [reg1] by 1
0x2b	cdec (F)	[reg1]	[reg1] = [reg1] - 1	Decrements the char value in [reg1] by 1
0x2c	cadd (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Adds the values from [reg2] and [reg3] into [reg1] as chars
0x2d	csub (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Subtracts the value in [reg3] from the value in [reg2] into [reg1] as chars
0x2e	cmul (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Multiplies the values from [reg2] and [reg3] into [reg1] as chars

Code	Instruction	Arguments	Equation	Action
0x2f	cdiv (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Divides the value in [reg2] by the value in [reg3] into [reg1] as chars. Throws divide by zero error if the value in [reg3] is zero.
0x30	cmod (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Puts value from [reg2] modulo the value in [reg3] into [reg1] as chars. Throws divide by zero error if the value in [reg3] is zero.
0x31	ctoi	[reg1], [reg2]	[reg1] = [reg2]	Casts the char value from [reg1] to an integer and puts it in [reg2]
0x32	ctof	[reg1], [reg2]	[reg1] = [reg2]	Casts the char value from [reg1] to a float and puts it in [reg2]
0x33	fflag (F)	[reg1]	N/A	Sets the flags based on the float value in [reg1]
0x34	fcmpeq (F)	[reg1], [reg2]	[reg2] == [reg3]	Checks if the floats in [reg2] and [reg3] are equal, sets the flags according to the boolean result
0x35	fcmpne (F)	[reg1], [reg2]	[reg2] != [reg3]	Checks if the floats in [reg2] and [reg3] are not equal, sets the flags according to the boolean result
0x36	fcmpgt (F)	[reg1], [reg2]	[reg2] > [reg3]	Checks if the float in [reg2] is greater than that in [reg3], sets the flags according to the boolean result
0x37	fcmplt (F)	[reg1], [reg2]	[reg2] < [reg3]	Checks if the float in [reg2] is less than that in [reg3], sets the flags according to the boolean result
0x38	fcmpge (F)	[reg1], [reg2]	[reg2] >= [reg3]	Checks if the float in [reg2] is greater than or equal to that in [reg3], sets the flags according to the boolean result
0x39	fcmple (F)	[reg1], [reg2]	[reg2] <= [reg3]	Checks if the float in [reg2] is less than or equal to that in [reg3], sets the flags according to the boolean result
0x3a	fadd (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Adds the values from [reg2] and [reg3] into [reg1] as floats
0x3b	fsub (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Subtracts the value in [reg3] from the value in [reg2] into [reg1] as floats
0x3c	fmul (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Multiplies the values from [reg2] and [reg3] into [reg1] as floats

Code	Instruction	Arguments	Equation	Action
0x3d	fdiv (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Divides the value in [reg2] by the value in [reg3] into [reg1] as floats. Throws divide by zero error if the value in [reg3] is zero.
0x3e	fmod (F)	[reg1], [reg2], [reg3]	[reg1] = [reg2] + [reg3]	Puts value from [reg2] modulo the value in [reg3] into [reg1] as floats. Throws divide by zero error if the value in [reg3] is zero.
0x3f	ftoi	[reg1], [reg2]	[reg1] = [reg2]	Casts the float value from [reg1] to an integer and puts it in [reg2]
0x40	ftoc	[reg1], [reg2]	[reg1] = [reg2]	Casts the float value from [reg1] to a char and puts it in [reg2]
0x41	prntc	[reg1]	N/A	Prints the char value in [reg1] to the console
0x42	prntstr	[reg1], [off]	N/A	Prints the null-terminated string at address [reg1] + [off] to the console
0x43	readc	[reg1]	[reg1] = Input	Stores a char value entered from the console into [reg1]
0x44	readstr	[reg1], [off]	{[reg1] + [off]} = Input	Stores a null-terminated string from the console at address [reg1] + [off]
0x45	rprnti	[reg1]	N/A	Prints the integer value in [reg1] to the console
0x46	rprntf	[reg1]	N/A	Prints the float value in [reg1] to the console
N/A	N/A	N/A	N/A	Separates global and non-global opcodes. The below opcodes must come before all others in a program, and are used to define globals
N/A	globalw	[var], [word]	[var] = [word]	Sets global [var] to [word]
N/A	globalb	[var], [byte]	[var] = [byte]	Sets global [var] to [byte]
N/A	globalstr	[var], [string]	[var] = [string]	Sets global [var] to [string]

Stack

The program provides a stack base pointer, BP, and that's it. See example `fibonacci_recursive.azm` for an example of how the stack is used

Examples

Note: `.azm` files should be up-to-date with the bytecode, but `.eze` files might require regeneration.
File path for examples: "Z\Z (Attempt 2)\AssemblyExamples\"

Recursive Fibonacci: `fibonacci_recursive.azm / .eze`

Fast Fibonacci: `fibonacci_fast.azm / .eze`

Babylonian Square Root: `babylonian_sqrt_.azm / .eze`

Number Guessing Game (With user input): `guessing_game_.azm / .eze`

The Compiler

The compilation happens in several key steps:

- **Tokenization** (`compiler::tokenize`): The input text is turned into tokens (numbers, identifiers, operators, etc.)
- **Abstract Syntax Tree (AST) Construction** (`compiler::constructAST`): A syntax tree is formed from the tokens. This happens by calling `compiler::condenseAST`, which first finds groups of parentheses and brackets and calls istelf recursively within those groups. It then combines individual nodes into the tree. For example a number, a plus token, and a number would become an addition expression with the two numbers as children. This bottom-up recursion is currrently only implemented for basic arithmetic and number casts, but the basic idea will be used for all syntactic structures.
- **Bytecode Construction** (`compiler::makeBytecode`): This takes the AST as input and makes a bytecode program that can be run on the VM. The compiler is currently capable of creating bytecode from the AST for an arithmetic expression.