Fungus: the Funge Machine

Alexios Chouchoulas
alexios@vennea.demon.co.uk

Abstract

The Funge family of programming languages consists of a group of n-dimensional, stack-based programming languages. The most prominent and original member of the family, Befunge (a two-dimensional language), was invented in 1993 by Chris Pressey. The Funge family is Turing-Complete, yet was designed to be 'a nightmare to compile' [?]. Considering the author is aware of two compilers for Befunge, it would be reasonable to claim that Funge programmers are at home with nightmares. This paper describes Fungus, an architecture designed and optimised for Funge. It is hoped that this will give rise to further nightmares, possibly involving Cthulhu in a bikini teaching INTERCAL to first-year law students.

Fungus is a microcoded, 16-bit, two-dimensional extreme RISC machine extremely suited to the interpretation of Funge at the hardware level. The author visualises the implementation of Funge compilers to generate Fungus-native code. The reader to whom the concept of Cthulhu in a bikini sounds acceptable may additionally visualise *optimising* Funge compilers for Fungus (in which case, the image of a lecture theatre full of future lawyers should be considered).

1 Introduction

The Funge family of programming languages consists of a group of n-dimensional, stack-based programming languages. The most prominent and original member of the family, Befunge (a two-dimensional language), was invented in 1993 by Chris Pressey. Befunge, like its n-dimensional[1] siblings, is Turing-complete, yet was designed to be 'a nightmare to compile' [?]. It can safely be said that Funges are 'unusual' languages. For an example, the following is the archetypal 'Hello World' programme in Unefunge (one-dimensional Funge):

```
052*"dlroW olleH">:#,_@
```

This programme already demonstrates quite a few of the features of a Funge (the syntax used here is the common denominator, Befunge'93): the existence of a stack; changing the direction of the PC; "0gnirts"-type strings et cetera. In the discussions to follow, it is assumed that the reader already knows at least Befunge '93. Otherwise, apart from boredom, insanity of click-happiness, the reader has little reason to be reading this paper.

Compiling Funges is problematic because of its self-modifying tendencies and multi-directional PC. Befunge compilers are not impossible to write, but they *are* nightmarish. Considering there at least two compilers for Befunge available, it would be reasonable to claim that Funge programmers are at home with nightmares.

[1]where $n \in \mathbb{N}, n > 0$ — the existence of fractal and zero-dimensional Funges is left as an excercise to the suicidal reader. The author purposefully avoids contemplating the existence of negative-dimension Funges in a last bid to retain sanity.

Following the example of the (in)famous Lisp machines, would it not be possible to accelerate and facilitate the creation of a Funge system using dedicated hardware? Not only is this possible, it also an idea perverted enough to fit in with Funge itself.

This paper describes Fungus, an architecture designed and optimised for the execution of Funge software. It is hoped that this will give rise to further nightmares, quite likely involving Cthulhu in a bikini teaching INTERCAL to first-year law students.

Fungus is a microcoded, 16-bit, two-dimensional extreme RISC machine extremely suited to the interpretation of Funge at the hardware level. The author visualises the implementation of Funge compilers to generate Fungus-native code. The reader to whom the concept of Cthulhu in a bikini sounds acceptable may additionally visualise *optimising* Funge compilers for Fungus (in which case, the image of a lecture theatre full of future lawyers should be considered).

The entire concept is theoretical, but a working emulator of Fungus can be built. Funge machines (like Lisp machines before them) can be utilised in the exploration of hack value, and as a means of punishing cocky undergraduates who think programming is an activity best done using a mouse. It is this author's belief that emerging programmers should be made painfully aware of the nightmares lurking in these Black Arts. The thorny path of Fear eventually leads to the green meadows of Knowledge.

2 Design Aims

Certainly, any Turing-Complete architecture can run Funge, in the same was as any architecture can run Lisp. Fungus therefore aims to be a minimal microprocessor capable of supporting the execution of Funge at a low level. The following features are therefore desired:

- Microcoded design. The processor is aware of a very small set of basic micro-instructions that help implement other, more complex macro-instructions. This allows Fungus to interpret various dialects of Funge.
- Two-dimensional memory model. Since Befunge is by far the most common language of the family, this is also the dimensionality of the Fungus architecture. Befunge subsumes Unefunge, and higher dimensions could, potentially, be introduced to the architecture through hacking microcode. Memory is seen as two-dimensional, which is entirely acceptable, especially since certain types of DRAMs use a row/column scheme for address selection.
- Vector registers. To support Funge at the lowest possible level, the architecture's PC is an $\mathbb{R}^2$ vector. An additional ΔPC register (also in $\mathbb{R}^2$ is employed to provide the direction vector. For higher dimensionalities, the reader is urged to look at the works of Cray Research.
- Hardware queue. Although Funges are stack-based languages, recent dialects have introduced the ability to push values to either the bottom or top of the stack, and pull values from either the bottom or top of the stack. These preferences are user-selectable, leading to the so-called stack actually behaving more like two different queues or stacks. The author believes this to be an extremely perverse, counter-intuitive, bug-prone, paradigm-breaking design and applauds it wholeheartedly. Fungus embraces this ingenious bit of design and implements a hardware stack/queue using two stack pointer registers dealt with by microcode.
- Hardware contexts. In an effort to allow complex operating systems to run on Fungus (an additional form of punishment for rapidly despairing students), Funge implements hardware contexts, somewhat similar and at the same time completely different from those of traditional memory managers. Hardware context registers allow delimiting a rectangular area of memory and allowing a program to run in it without having access to memory outside its own. The PC wraps around the edges of this region, thus forming a sub-torus of the super-torus that is Fungus' main memory. Lahey space is not supported by the hardware, but the masochistic topology enthusiast can still extract hours of pleasure attempting to visualise this sub-/super-torus relation.

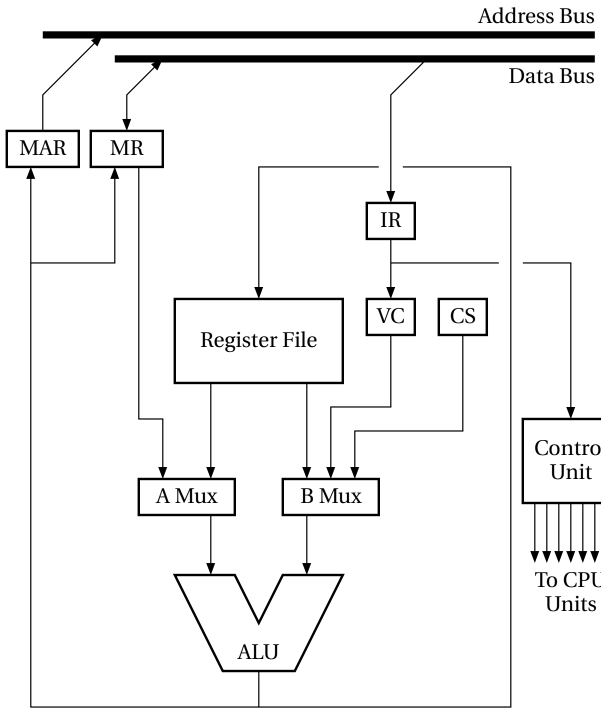

Figure 1: Block diagram of the processor.

- Sharp blades. It is said UNIX gives one enough rope to shoot oneself in the foot. Fungus is required to conform to this time-honoured programming tradition, but the rope tricks are becoming dated. Hence, Fungus aims at providing one with enough sharp blades to shoot oneself in the foot. Various early-Eighties-style design flaws are introduced in an effort to make the user's life even more miserable.
- Ease of implementation. Fungus is designed as a project that can be implemented using simple solid-state components (of the $74xxx$ family, for instance). This allows Funge to be inflicted on students taking Electrical Engineering, in addition to those taking Computer Science. Also, this increases hack value by allowing the reader to toy with the idea of physically building the processor, thereby making such a reader appear (to others) a guru of esoteric hardware[2].

3 Hardware

The design of Fungus intentionally resembles a simplified version of the MIPS R$\times$000 architecture [**?**]. The data flow is built around a register file and an arithmetic/logic unit (ALU). Like the MIPS, all Fungus instructions involve the ALU and almost all involve the register file. Here is an explanation of the constituent parts, as outlined in Fig. 1:

[2]not to mention somewhat obsessive-compulsive.

Register File. A 8×18-bit RAM containing the values of the eight, 18-bit registers of the CPU. The register file (RF) has two read (output) 18-bit ports A and B and one write (input) 18-bit port C. Each port can address independently any of the eight registers by means of three sets of three address lines each. An additional latch line clocks data from the input port into the register addressed by the C address lines.

ALU. This has two input ports (A and B) and one output port (C), all 18 bits wide. It also has three control lines to select the operation to be performed. The result of performing the selected operation on the two input ports appears on the output port after a certain stabilisation delay. An additional pair of control lines selects the current mask mode.

Memory Address Register (MAR). This register can only be written by the CPU. It buffers and outputs an 18-bit address to the system's address bus. A control line latches data from the ALU's C (output) port into this register when this is required.

Memory Register (MR). This register buffers and makes available data read from or written to system memory. This register has three ports. One is tri-state (bi-directional) and directly connected to the system's data bus; the other allows values from the ALU's C (output) port to be written to the register; and the third allows values to be read from the register.

Instruction Register (IR). This register is connected to the system's data bus and is latched during the fetch cycle. It contains the instruction word currently being executed. This is connected to the control unit and, indirectly, to the ALU.

A MUX. The A multiplexer selects one of two data sources for the ALU's port A. The two choices are the RF's A port (using a register's value as the left-hand operand); and the contents of the MR register (to access data read from memory). A single control line chooses among the two.

B MUX. Like the A multiplexer, this unit selects among different data sources for the ALU's right-hand operand. There are three choices here: the value of the RF's B output port (to access a register's value); a value drawn from the current Instruction Register (IR), as processed by the Vector Control (VC) unit (to access literals embedded in the current instruction); or a value from the Constant Store (CS) ROM unit, to use a hardwired constant value. Two control lines choose among the three sources.

Vector Control (VC). This unit takes the literal 9-bit field the IR and either outputs it as a rd literal to the ALU, or copies the nine bits to both wo and rd and outputs the entire word to the ALU. This allows a 9-bit literal L to be used either as a scalar literal or as the vector (L, L). A single control line selects the behaviour of this unit.

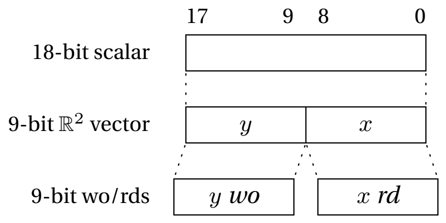

Figure 2: The Fungus data types.

Constant Store (CS). This small 18-bit ROM contains a number of constants used in processing pico-code and other things. Scalars like like 1, -1 and vectors $(1, 1)$ and $(-1, -1)$ are stored permanently in this ROM. Combined with masking modes in the ALU, this implements useful features and simplifies the CPU pico-code.

Control Unit (CU). This unit is driven by the contents of the IR. It contains a ROM containing VLIW pico-instructions. Each bit of a pico-instruction directly drives one of the control signals controlling the various units of Fungus. A pico-PC steps through the ROM executing pico-instructions.

4 Programming Model

4.1 Word Length

Fungus is an 18-bit word machine. The author strongly believes in machines with word lengths that are not a multiple of four. The multiple-of-three approach is a time-honoured one, with support from such giants as IBM and Digital. Besides, forcing programmers to start thinking in octal after well nigh twenty-five years of thinking in hexadecimal works in accordance with Fungus philosophy[3].

The system deals with 18-bit words and pairs of 9-bit *wos* and *rds*. These are known in mainstream computer science as most signifncant and least significant half-words, respectively.

An 18-bit quantity can have one of three interpretations, as illustrated in Fig. 2:

1. An 18-bit scalar value, in the range 0–262,143.
2. An $\mathbb{R}^2$ vector, where the wo and rd represents the y and x ordinates respectively. The wo and rd are in the range 0–511.
3. The two ordinates of the vector representation may be accessed individually using *instruction masking*. This allows Fungus to access individual wos and rds in memory.

[3]obfuscation and baroque design.

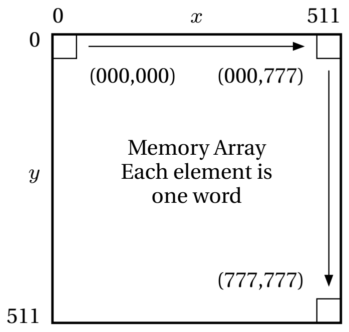

Figure 3: The Fungus address space.

	17 — 9	8 — 0	
\$0	$\emptyset$	$\emptyset$	$\emptyset$
\$1	PC_y	PC_x	PC
\$2	ΔPC_y	ΔPC_x	ΔPC
\$3	A_y	A_x	A
\$4	B_y	B_x	B
\$5	C_y	C_x	C
\$6	D_y	D_x	D
\$7	E_y	E_x	E

Figure 4: The Fungus register file.

To the programmer, it is most convenient to write Fungus numbers in base eight (octal), with three octal digits to a wo or rd and six octal digits to a word.

4.2 Byte order

Fungus does not have bytes, hence no byte order. However, this specification *does* require that the wo (y ordinate) is most significant. Hence, Fungus is wo-endian or y-endian. Note, however, that symbolic vector notation gves the x ordinate first as (x, y).

4.3 Address Space

For convenience, the address space is identical to the word length: 18 bits wide. The programmer is free to use vectors or scalars to address memory, but Fungus internally uses vectors. Of the 18 address lines provided by the microprocessor, the most significant 9 (A_9–A_{17}) correspond to the wo and the y ordinate. The least significant 9 lines (A_0–A_8) correspond to the rd and the x ordinate. Thus, the maximum amount of memory addressable by Fungus is 256 kwords[4].

Unlike conventional architectures, this memory is organised as a two-dimensional array, with 512×512 elements. Hence the use of vectors to address memory.

Interestingly, this vector view of memory is neither alien nor inconsistent with existing RAM technology. Most DRAM chips distinguish between 'row' and 'column' addresses and use external signals like $\overline{\text{CAS}}$ to change the semantics of their address pins. It would appear that computer technology has been building up to Fungus, doubtlessly the peak of CPU design for the discerning sadomasochist.

The topology of the address space is toroidal. This is a side effect of the use of vector registers, as outlined in Section 4.5. An intentional lack of overflow detection in the ALU allows wrapping around of vector ordinates to simulate this popular yet simple Funge topology.

[4]that is 2^{18} words, with one kword being 1024 words, as the Elder Gods intended.

4.4 Memory

Memory also consists of 18-bit words. Fungus only reads and writes 18-bit quantities. Thus, the 256 kwords of accessible memory is 18-bits wide.

For the byte-ophiliac reader, a word is 2.25 bytes. A single 512-word row or column of memory is 1,152 bytes (1.125 kbytes). The entire address space corresponds to 589,824 bytes, or 576 kbytes. However, this is irrelevant as values cannot be accessed in byte-sized chunks but only in word-sized quanta.

The address space may be expanded using memory mapping, swapping and paging techniques with external, kludgy hardware. Again, this is consistent with Fungus design.

4.5 Registers

As seen in Fig. 4, Fungus has eight word-wide registers. Registers may be treated as 18-bit scalar values and two-dimensional vectors with 9-bit wo and rd ordinates. The registers are referred to by number as \$0–\$7 (pronounced like 'big-money-zero'), or by name. All registers can be used as general purpose registers by the programmer who points loaded guns at her feet, but in reality, all but three registers have special uses:

\$0 or 0: a source of zeroes. In compliance with Fungus design philosophy, this register *is* writable. Changing it, however, will massively disrupt CPU operation as \$0 is used internally by CPU picocode.

\$1 or PC: the program counter. Like all CPUs, this register points to the next instruction to be fetched from memory. Unlike all CPUs, the PC is a vector.

\$2 or ΔPC: the program counter delta. This vector value is added to the PC vector immediately after an instruction fetch. The ordinate values are arbitrary, though values of -1 (left or up), 0 (no change) and 1 (right, down) are typical. Here are a few examples of useful ΔPC values:

- **(000,000)**: halts the CPU (PC stops moving).
- **(777,000)**: PC moves to the north.
- **(001,000)**: PC moves to the south.

- **(000,777)**: PC moves to the east.
- **(000,001)**: PC moves to the west.

Diagonal movement is, of course possible, as are *flying* PCs, though these should not be attempted by the faint of heart.

$\$3$ or A: the first general purpose register.

$\$4$ or B: the second general purpose register.

$\$5$ or C: the third and last general purpose register.

$\$6$ or D: a general purpose register. This one is used to store temporary copies of the ΔPC register by the `TRP` command. It can still be used outside system traps/micro-instructions.

$\$7$ or E: likewise, this register may hold temporary copies of the PC register during a `TRP`.

5 CPU Architecture

Many CPUs of the past have been microcoded. Machine code instructions are internally interpreted as short programmes in microcode. Fungus makes an extra step towards the cliff of insanity by introducing pico-instructions and pico-code. The CPU is built on top of a simple[5] RISC core, as a multi-layer architecture.

Pico-code. Executed inside the CPU, Fungus pico-code is a simple Very Long Instruction Word (VLIW) machine language. Pico-code is immutable and resides in a ROM inside the CPU. It translates instructions to control signals for the CPU's component units. Pico-code is not accessible by the programmer.

Microcode. Is the lowest possible level of machine code executed by the CPU. This is a RISC language, with one instruction per word. Microcode is not mutable in itself, but it is expandable using user defined *traps.*

Befunge code. This can be implemented as a set of traps in microcode. Each Befunge instruction is the least significant 8 bits of an instruction word. The instructions are interpreted and executed in Microcode. Other versions of Befunge can be implemented by redefining the traps; a feature that allows for diverse lower extremity injuries via chemically propelled metal projectiles.

6 Instruction Set

Fungus is a Reduced Instruction Set Computer (RISC). The instruction set is as small as possible. There are 26 instructions and they are all one word wide. Instructions are not executed in a single clock tick, however: they range from three to eight cycles, with most instructions needing three.

[5] by INTERCAL standards

6.1 Addressing Modes

Fungus does not have the usual large family (or small country) of addressing modes. In fact, the mention of addressing modes with respect to this architecture is officially deprecated. However, in the interest of providing an explanation to users hopelessly lodged in this paradigm, here is a list of 'addressing modes':

Immediate. An instruction operates on a literal, storing the address in a register.

Indexed. An instruction accesses memory by applying an arithmetic or logic operation on two register values, and using the result as the memory address. The result of the instruction is stored in the target register.

Register. An instruction operates on one or two registers, storing the result in a third, target register.

6.2 Masking Modes

Since Fungus is an $\mathbb{R}^2$ machine, it needs to deal with vector values, but also with their ordinates in an independent fashion. To provide facilities, it also needs to access words as scalar values. This duplication of functionality would increase unacceptably the size of the instruction set. This, in the interest of additional obfuscation, masking modes were introduced. Not to be mistaken with addressing modes, masking modes modify the semantics of instructions as follows:

Vector mode. This is the default. The wo and rd parts of a word are treated independently. Literals are written like $(123, 456)$ (although this is not necessary; the same literal could still be written 123456). The result of $777000 + 001001$ would be 000001. In vector notation: $(777, 0) + (1, 1) = (0, 1)$.

X mode. This mode masks the wo (y ordinate) part of words. In this way, all instructions affect *only* the rd (x ordinate) part of data. In this context, the addition $(777, 0) + (1, 1)$ yields $(777, 1)$.

Y mode. As above, but the rd (x ordinate) part of data is masked, making it immutable.

Scalar mode. Treats words as scalars. In scalar mode, the addition $115333 + 225511$ would yield 343044 (note how the carry crosses the wo-rd boundary).

6.3 Instruction Format

There are two groups of instructions: group 0 involves a target register and 9-bit literal; group 1 involves a target

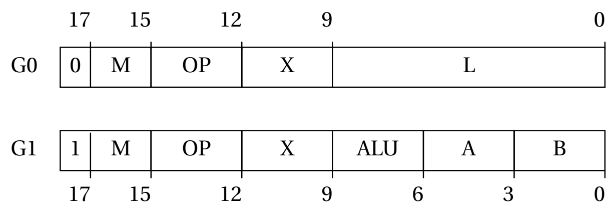

Figure 5: Instruction Format.

register and one or two source registers. The two groups are illustrated in Fig. 5. The formats themselves are as follows.

Group 0. Comprises of the following fields (in order of increasing significance):

- **L:** 9 bits. A 9-bit literal argument.
- **X:** 3 bits. The target register.
- **OP:** 3 bits. The instruction opcode.
- **M:** 2 bits. Masking mode.
- **0:** 1 bit. The instruction group (always 0).

Group 1. Comprises of the following fields (in order of increasing significance):

- **B:** 3 bits. For binary instructions, the register used as right-hand operand. For unary instructions, this field acts as an extension of the ALU field.
- **A:** 3 bits. The register used as left-hand operand.
- **ALU:** 3 bits. The ALU op code.
- **X:** 3 bits. The target register.
- **OP:** 3 bits. The instruction opcode.
- **M:** 2 bits. Masking mode.
- **1:** 1 bit. The instruction group (always 1).

7 Instruction Reference

7.1 Arithmetic and Logic Binary Operations

Overview All instructions in this category are G1 instructions. In fact, they are the *same* instruction: `ALU`, engaging the ALU in different modes. For the programmer's convenience, the `ALU` instructions is assembled and disassembled as different sub-instructions, depending on the contents of the `ALU`. Since binary arithmetic and logic instructions are effectively one instruction, semantics are exactly the same throughout. Only the arithmetic or logic operation differs.

Operation These instructions apply an arithmetic or logic operation on the contents of registers A (denoted by bits `aaa`) and B (denoted by bits `bbb`) and store the result in register X (denoted by bits `xxx`).

Masking Modes Use of MMs modifies the way arithmetic/logic is performed and masks the result. Vector mode (e.g. `ADD`) adds vector operands (wo and rd ordinates added separately). Scalar mode (e.g. `XOR.s`) treats register contents as 18-bit words. X and Y modes (e.g. `SUB.x` and `AND.y` respectively) add only the rd and wo parts of a word respectively, leaving the rest *untouched*.

Examples The easiest and most illustrative instruction is, of course, addition. Using initial register values $\$1 = 123456_8$, $\$2 = 654321_8$, $\$3 = 555555_8$, $\$4 = \$5 = \$6 = \$7 = 222222_8$, the following instructions can be executed:

```
ADD     $4,$1,$2    ; $4 is now 777777
ADD.x   $5,$1,$2    ; $5 is now 222777
ADD     $6,$1,$3    ; $6 is now 700233
ADD.s   $7,$1,$3    ; $7 is now 701233
```

Final register values: $\$4 = 777777_8$, $\$5 = 222777_8$, $\$6 = 700233_8$, $\$7 = 701233_8$. Note the difference between the last two instructions. The Scalar mode instruction (`.s` prefix) propagates the carry past the wo/rd boundary, whereas the default, vector mode does not.

7.1.1 ADD — Add registers

Instruction	**ADD x,a,b**
Format	`1mm 000 xxx 000 aaa bbb`
Semantics	$X \leftarrow A + B$
Cycles	4

7.1.2 SUB — Subtract registers

Instruction	**SUB x,a,b**
Format	`1mm 000 xxx 001 aaa bbb`
Semantics	$X \leftarrow A - B$
Cycles	4

7.1.3 AND — Bitwise And

Instruction	**AND x,a,b**
Format	`1mm 000 xxx 010 aaa bbb`
Semantics	$X \leftarrow A \wedge B$
Cycles	4

Note Since there is no carry, this instruction works in exactly the same way in both Vector and Scalar modes.

7.1.4 OR — Bitwise Or

Instruction	**OR x,a,b**
Format	`1mm 000 xxx 011 aaa bbb`
Semantics	$X \leftarrow A \vee B$
Cycles	4

Note Since there is no carry, this instruction works in exactly the same way in both Vector and Scalar modes.

Group 0

GM	OP	X	L	Cycles	Instruction	Semantics
0M	000	X	L	8	`TRP L`	TPC $\leftarrow$ PC; TΔPC $\leftarrow$ ΔPC; PC $\leftarrow (L, 0)$; ΔPC $\leftarrow (-1, 0)$
0M	001	X	L	4	`LI X,L`	$\underline{X \leftarrow L}$
0M	010	X	L	4	`LV X,L`	$\underline{X \leftarrow (L, L)}$
0M	011	X	L	5/6	`SZ X,L`	$\underline{X = 0} \Rightarrow \text{PC} \leftarrow \text{PC} + \Delta\text{PC}$
0M	100	X	L	5/6	`SNZ X,L`	$\underline{X \neq 0} \Rightarrow \text{PC} \leftarrow \text{PC} + \Delta\text{PC}$
0M	101	X	L	7/8	`DZ X,L`	$\underline{\Delta\text{PC} \leftarrow (-1, -1)}$; $X = 0 \Rightarrow \underline{\Delta\text{PC} \leftarrow (1, 1)}$
0M	110	X	L	7/8	`DNZ X,L`	$\underline{\Delta\text{PC} \leftarrow (-1, -1)}$; $X \neq 0 \Rightarrow \underline{\Delta\text{PC} \leftarrow (1, 1)}$
0M	111	X	L	5	`RET`	$PC \leftarrow$ TPC; ΔPC $\leftarrow$ TΔPC

Group 1

GM	OP	X	ALU	A	B	Cycles	Instruction	Semantics
1M	000	X	000	A	B	4	`ADD X,A,B`	$X \leftarrow A + B$
1M	000	X	001	A	B	4	`SUB X,A,B`	$X \leftarrow A - B$
1M	000	X	010	A	B	4	`AND X,A,B`	$X \leftarrow A \wedge B$
1M	000	X	011	A	B	4	`OR X,A,B`	$X \leftarrow A \vee B$
1M	000	X	100	A	B	4	`XOR X,A,B`	$X \leftarrow A \otimes B$
1M	000	X	101	A	B	4	(reserved)	
1M	000	X	110	A	B	4	(reserved)	
1M	000	X	111	A	000	4	`NOT X,A`	$X \leftarrow \neg A$
1M	000	X	111	A	001	4	`SHR X,A`	$X \leftarrow \lfloor A/2 \rfloor$
1M	000	X	111	A	010	4	`INV X,A`	$X \leftarrow A + (1, 1)$
1M	000	X	111	A	011	4	`DEV X,A`	$X \leftarrow A - (1, 1)$
1M	000	X	111	A	100	4	`INC X,A`	$X \leftarrow A + 1$
1M	000	X	111	A	101	4	`DEC X,A`	$X \leftarrow A - 1$
1M	000	X	111	A	110	4	(reserved)	
1M	000	X	111	A	111	4	(reserved)	
1M	001	X	OP^1	A	B	5	`LW X,A,B`	$X \leftarrow [\underline{A \circ B}]$
1M	010	X	OP^1	A	B	5	`LX X,A,B`	$X_x \leftarrow [\underline{A \circ B}]_x$
1M	011	X	OP^1	A	B	5	`LY X,A,B`	$X_y \leftarrow [\underline{A \circ B}]_y$
1M	100	X	OP^1	A	B	5	`SW X,A,B`	$[\underline{A \circ B}] \leftarrow X$
1M	101	X	OP^1	A	B	5	`SX X,A,B`	$[\underline{A \circ B}]_x \leftarrow X_x$
1M	110	X	OP^1	A	B	5	`SY X,A,B`	$[\underline{A \circ B}]_y \leftarrow X_y$
1M	111	X	000	A	B	4	(reserved)	

Table 1: Fungus instruction set.

7.1.5 XOR — Bitwise Exclusive Or

Instruction	**XOR x,a,b**
Format	`1mm 000 xxx 100 aaa bbb`
Semantics	$X \leftarrow A \otimes B$
Cycles	4

Note Since there is no carry, this instruction works in exactly the same way in both Vector and Scalar modes.

7.2 Arithmetic and Logic Unary Operations

Overview Instructions in this category are cascaded extensions of the `ALU` instruction, where the ALU operation is 111_2 and the B field is to select a unary ALU operation. These are, of course, G1 instructions.

Operation These instructions apply an arithmetic or logic operation on the contents of register A (denoted by bits `aaa`), storing the result in register X (denoted by bits `xxx`).

Masking Modes As always, use of MMs modifies the way arithmetic/logic is performed and masks the result. Vector mode (e.g. `DEV`) increases vector operands (wo and rd ordinates increased separately). Because of the logic or kludgy nature of most of these instructions, masking modes do not work as expected. Please read along for more details following each instruction.

7.2.1 NOT — Bitwise Negation

Instruction	**NOT x,a,b**
Format	`1mm 000 xxx 111 aaa 000`
Semantics	$X \leftarrow A\neg$
Cycles	4

Masking Modes Since there is no carry, this instruction works in exactly the same way in both Vector and Scalar modes. In X and Y modes, only the rd and wo of the target register are modified respectively.

7.2.2 SHR — Shift Right

Instruction	**SHR x,a**
Format	`1mm 000 xxx 111 aaa 001`
Semantics	$X \leftarrow \lfloor A/2 \rfloor$
Cycles	4

Notes This instruction halves the source register, rounding down. The most significant bit (or bits, in vector mode — see below) are zero-padded. There is no corresponding `SHL` instruction. This can be simulated at the assembly level using the `ADD` instruction. Hence `SHL x,a` is equivalent to `ADD x,a,a` (a left shift effectively doubles the operand). In the interest of simplicity, only single bit shifts are available.

Masking Modes In Vector mode, the wo and rd parts of a word are shifted separately. In scalar mode, the entire word is shifted in unison. In X and Y modes, the rd and wo are respectively shifted without disturbing the other half of the target register.

Examples Using $\$1 = 123456_8$ and $\$4 = \$5 = \$6 = \$7 = 333333_8$:

```
SHR.s   $4,$1        ; $4 is now 051627
SHR.x   $5,$1        ; $5 is now 333227
SHR.y   $6,$1        ; $6 is now 051333
SHR     $7,$1        ; $7 is now 051227
```

Note the difference between the scalar and vector instructions. Shifted bit values do not cross the wo/rd boundary in vector mode. Also of note is the third instruction which shifts $123_8 = 83_{10}$ to the right yielding $51_8 = 41_{10}$ (rounding down).

7.2.3 INV — Increment Vector

Instruction	**INV x,a**
Format	`1mm 000 xxx 111 aaa 010`
Semantics	$X \leftarrow A + (1,1)$
Cycles	4

Notes This unusual instruction increments a vector by adding $(1,1)$ to it. At first it sounds inane, but masking modes turn this instruction to a powerful tool. Grasping the use of `INV` (and a few other similar instructions) is an important part of understanding fully the mentality of the Fungus architecture.

Masking Modes In Vector mode, the vector value of register a is incremented along the main diagonal by adding the vector $(1,1)$ to it. In X mode, only the rd is incremented, effectively moving the vector register a along the X axis. In Y mode, only the wo is incremented, effectively moving the vector register a along the Y axis. In scalar mode, the constant 001001_8 is added to register a.

Examples Using $\$1 = \$6 = \$7 = 123456_8$:

```
INV     $5,$1        ; $7 is now 124457
INV.x   $6,$1        ; $5 is now 123457
INV.y   $7,$1        ; $6 is now 124456
```

This instruction is useful in incrementing vector pointers.

7.2.4 DEV — Decrement Vector

Instruction	**DEV x,a**
Format	`1mm 000 xxx 111 aaa 011`
Semantics	$X \leftarrow A - (1,1)$
Cycles	4

Notes Works like `INV`, but decrements vectors.

Masking Modes Works like `INV`, but the constant $(1,1)$ (or the scalar 001001_8) is substracted from register a and stored in register x.

7.2.5 INC — Increment Scalar

Instruction	**INC x,a**
Format	`1mm 000 xxx 111 aaa 100`
Semantics	$X \leftarrow A + 1$
Cycles	4

Notes This mundane, scalar counterpart of `INV` increments a scalar value. This is a particularly kludgy instruction, but needed for many uni-dimensional tasks. As such, it lacks the elegance of most other Funge instructions[6].

Masking Modes In Vector mode, the vector $(0,1)$ is added to register a. In X mode, this instruction behaves exactly like `INV.x`. In Y mode, this is a `NOP`. In scalar mode, register x receives the scalar value $a + 0000001_8$.

Examples Using $\$1 = 123456_8$, $\$4 = \$5 = \$6 = \$7 = 000000_8$:

```
INC     $4,$1       ; $7 is now 123457
INC.s   $5,$1       ; $7 is now 123457
INC.x   $6,$1       ; $5 is now 123457
INC.y   $7,$1       ; $6 is now 123456
```

7.2.6 DEC — Decrement Scalar

Instruction	**DEC x,a**
Format	`1mm 000 xxx 111 aaa 101`
Semantics	$X \leftarrow A - (1,1)$
Cycles	4

Notes `DEC`[7] works like `INC`, but decrements scalars.

Masking Modes Works like `INC`, but the constant $(0,1)$ (or the scalar 000001_8) is substracted from register a and stored in register x.

7.3 Literals

Loading registers with literal values is needed by any self respecting architecture. Like the MIPS R×000, Fungus has a fixed single-word instruction length, which precludes loading an entire literal word into a register. A pair of instructions are therefore provided, but, in true Fungus fashion, they are not what the MIPS programmer would expect.

[6]although it *does* offer a novel way to do a `NOP`.

[7]requests to change the name of `DEC` to `COMPAQ` will be diverted to `/dev/null`, as soon as a Fungus-based ⋆nix kernel is developed.

7.3.1 LI — Load Literal

Instruction	**LI x,L**
Format	`0mm 001 xxx LLLLLLLLL`
Semantics	$X \leftarrow L$
Cycles	4

Notes This instruction loads a literal value into a register's rd. Since the literal L is only 9 bits wide, only the rd can be affected. The wo is zero-padded. To set a wo to a literal value, please use `LV.y`.

Masking Modes In Vector and Scalar modes, `LI` and `LI.s` simply copy the zero-padded literal L to register X. In X mode, the literal overwrites the rd of register X. The wo is not modified. In Y mode, this instruction zeroes the wo of the target register without modifying the rd.

Examples Here are a few examples of the use of `LI`, where $\$6 = \$7 = 555555_8$:

```
LI      $4,145      ; $4 is now 000145
LI.s    $5,145      ; $7 is now 000145
LI.x    $6,777      ; $5 is now 555777
LI.y    $7,666      ; $6 is now 000555
```

Note how Scalar and Vector modes work in an identical fashion. Also noteworthy is the unusual `LI.y` construct that zeroes an rd.

7.3.2 LV — Load Vector

Instruction	**LV x,L**
Format	`0mm 010 xxx LLLLLLLLL`
Semantics	$X \leftarrow (L, L)$
Cycles	4

Notes Yet another unusual instruction. This one duplicates L to form the vector (L, L), which it then stores in register X.

Masking Modes In Vector and Scalar modes, `LV` and `LV.s` simply copy the literal vector (L, L) or the scalar $(2^9 + 1)L$ (of doubtful usefulness) to the target register. In X mode, this instruction behaves exactly like an `LI.x`. In Y mode, the most useful invocation, L is stored in the wo of the target register.

Examples Here are a few examples of the use of `LV`, where $\$6 = \$7 = 555555_8$:

```
LV      $4,145      ; $4 is now 145145
LV.s    $5,707      ; $7 is now 707707
LV.x    $6,777      ; $5 is now 555777
LV.y    $7,666      ; $6 is now 666555
```

Note how Scalar and Vector modes work in an identical fashion and the use of `LV.y` to load a literal into a wo.

Table 2: ALU operations and their corresponding *nazg* expressions.

Op	**ALU**	**B**	**Nazg Expression**
`ADD`	000	B	$a + b$
`SUB`	001	B	$a - b$
`AND`	010	B	$a \,\&\, b$
`OR`	011	B	$a \mid b$
`XOR`	100	B	$a\,\hat{}\,b$
`NOT`	111	000	$\sim a$
`SHR`	111	001	$>> a$
`INV`	111	010	$+ + a$
`DEV`	111	011	$- - a$
`INC`	111	100	$+a$
`DEC`	111	101	$-a$

7.4 Memory Input/Output

Overview Memory input and output is accomplished through the abuse of six G1 instructions (dealing with indexed register memory access). The G1 instructions incorporate and subsume the `ALU` instruction. As such they *could* be seen as 66 different instructions, but this author will not because it's not convenient enough[8].

Operation These instructions apply an arbitrary arithmetic or logic binary or unary operation on the contents of registers A (denoted by bits `aaa`) and B (denoted by bits `bbb`, naturally only used on binary operations).

The result is then used to address memory. Load instructions transfer a word, wo or rd from that location in memory to the X register. Store instructions transfer a word from register X to the resultant location in memory. This last case is the only exception in the use of register X as a target register: Store instructions use memory as a target and register X as a source.

Obviously, not all arithmetic and logic operations will be useful in addressing memory. However, the elegance of Fungus is such that using obscure operations is not forbidden. It is, in fact, encouraged.

In the instruction descriptions below, the symbol $\circ$[9] denotes an arithmetic or logic operation, either binary or unary. Where the operation is unary, the nazg is written in prefix fashion. Table 2 lists ALU operations and their nazg symbols.

Masking Modes MMs in the context of load and store instructions work slightly less intuitively than with other instructions. Masking is only applied to the memory address *calculation*, not the entire operation. All of `LW`, `LW.x`, `LW.y` and `LW.s` load entire words from memory. The difference is in the way the memory addresses are calculated.

In vector mode, registers A and B are treated as vector values, and behave as specified in the corresponding ALU operation. In scalar mode, registers A and B behave like scalars, with carry crossing the wo/rd boundary, et cetera. Modes X and Y are not particularly useful. They respectively apply the operation on the rd and wo, but the other half of the word is filled with zeroes.

Examples Here are a few examples of the flexibility afforded by this scheme, where $\$4 = 123456_8$, $\$5 = 111111_8$ and $\$6 = 555555_8$:

```
LW      $4,$5+$6    ; $4 is mem[666666]
LW      $4,$5|$6    ; $4 is mem[555555]
LX      $4,$5^$6    ; $4.x is mem[444444].x
LW.x    $4,$5+$6    ; $4 is mem[000666]
SY.x    $4,$5&$6    ; mem[000111].y is (123456).y
SW      $4,+$5      ; mem[111112] is 123456
```

Note how the `.x` mode in the fifth example applies the address calculation to the rd only, but a *wo* is written to memory.

7.4.1 LW — Load Word

Instruction	**LW x,a $\circ$ b** or **SW x,$\circ$ b**
Format	`1mm 001 xxx alu aaa bbb`
Semantics	$X \leftarrow [A \circ B]$
Cycles	5

Notes Evaluates A nazg B (or nazg A for unary operations) and addresses memory with the operation result to retrieve a whole word. The word is stored in register X.

7.4.2 LX — Load Rd

Instruction	**LX x,a $\circ$ b** or **SX x,$\circ$ b**
Format	`1mm 010 xxx alu aaa bbb`
Semantics	$X_x \leftarrow [A \circ B]_x$
Cycles	5

Notes Evaluates A nazg B (or nazg A for unary operations) and addresses memory with the operation result to retrieve an rd only. The rd is stored in register X's rd.

7.4.3 LY — Load Wo

Instruction	**LY x,a $\circ$ b** or **SY x,$\circ$ b**
Format	`1mm 011 xxx alu aaa bbb`
Semantics	$X_y \leftarrow [A \circ B]_y$
Cycles	5

Notes Evaluates A nazg B (or nazg A for unary operations) and addresses memory with the operation result to retrieve a wo only. The wo is stored in register X's wo.

[8]and to add even more confusion and obfuscation to the architecture.

[9]the author humbly submits the name *nazg* for this most useful meta-symbol. One nazg to denote them all.

7.4.4 SW — Store Word

Instruction	**SW x,a** ∘ **b** or **SW x,**∘ **b**
Format	`1mm 100 xxx alu aaa bbb`
Semantics	$[A \circ B] \leftarrow X$
Cycles	5

Notes Evaluates A nazg B (or nazg A for unary operations) and addresses memory with the operation result. The word contained in register X is stored at that address.

7.4.5 SX — Store Rd

Instruction	**SX x,a** ∘ **b** or **SX x,**∘ **b**
Format	`1mm 101 xxx alu aaa bbb`
Semantics	$[A \circ B]_x \leftarrow X_x$
Cycles	5

Notes Evaluates A nazg B (or nazg A for unary operations) and addresses memory with the operation result. The rd contained in register X is stored at that address' rd.

7.4.6 SY — Store Wo

Instruction	**SY x,a** ∘ **b** or **SY x,**∘ **b**
Format	`1mm 110 xxx alu aaa bbb`
Semantics	$[A \circ B]_y \leftarrow X_y$
Cycles	5

Notes Evaluates A nazg B (or nazg A for unary operations) and addresses memory with the operation result. The wo contained in register X is stored at that address' wo.

7.5 Flow Control

Flow control is implemented by means of three pairs of instructions.

1. The trap mechanism inclues `TRP` and `RET` is a cross between the subroutine calling of most architectures, ×86 software interrupts and Motorola 68k traps. Traps can be used to build up to 512 macro-instructions or system services, or to implement Befuge on top of Fungus.
2. The skip mechanism includes the `SZ` and `SNZ` instructions. These skip the next instruction depending on the value of the specified register.
3. The divert mechanism includes the `DZ` and `DNZ` instructions. These modify (divert) the ΔPC register and hence the direction of the PC based on the value of the specified register.

7.5.1 TRP — Trap

Instruction	**TRP L**
Format	`0XX 000 XXX LLLLLLLLL`
Semantics	TPC ← PC; TΔPC ← ΔPC; PC ← $(L, 0)$; ΔPC ← $(-1, 0)$
Cycles	8

Notes Bits marked '`X`' in the instruction format above are Don't-Care values. The target register is ignored and assembly notation of `TRP` omits it altogether.

This is the most complex Fungus instruction. It works as follows: the PC and ΔPC registers are saved in the TPC ($7) and TΔPC ($6) registers respectively; then ΔPC is made to point 'north' and PC is assigned the vector $(0, L)$. This effectively jumps to a specified subroutine on the first row of memory. The subroutine performs any processing necessary and issues the `RET` instruction to return to the caller.

These traps can be used for interrupt handling, system service vectors, and to implement Befunge as a set of macro-instructions built on top of Fungus.

Traps 32–127 correspond to ASCII characters. These can be issued on most modern keyboards. The observant reader will no doubt have noticed that the instruction format for `TRP` uses don't-care values for the ALU and MM fields. Thus, the wo of a trap instruction is typically 0 and the rd denotes the trap to jump to.

In this way, plain text files comprising printable ASCII characters are seen as traps by Fungus. Each trap performs one Funge instruction and returns to the caller. In this manner, complex Funges can be implemented cleanly and elegantly on top of the lower-level Fungus instruction set. At the same time, Fungus machine code can still be mixed in with high-level Befunge instructions for added hack value.

Trap vectors are arranged on the row 0 of the address space for two reasons:

- RAM is expected to be mapped starting with row 0.
- Fungus boot ROM can modify row 0 to displace the PC either 'north' or 'south'. North wraps around to row 777, where ROM is expected to be mapped. Thus the default, ROM handler for a trap may be set. South allows user-supplied traps to be implemented.

Masking Modes MMs do not apply to the `TRP` command and are ignored.

Example The Unifunge programme `52*.` that evaluates and prints out '10' looks as follows in 18-bit octal words: `000065 000062 000052 000056`. This disassembles into the following Fungus code, for a hypothetical Befunge programming environment.

```
TRP      065       ; '5': Push 5
TRP      062       ; '2': Push 2
TRP       052       ; '*': Multiply
TRP       056       ; '.': Print number
```

7.5.2 RET — Return

Instruction	**RET**
Format	`0XX 001 XXX XXXXXXXXX`
Semantics	$PC \leftarrow$ TPC; ΔPC $\leftarrow$ TΔPC
Cycles	5

Notes Bits marked 'X' in the instruction format above are don't-care values. Both target register and literal value are ignored for this instruction. No arguments need to be passed to it in assembly.

This instruction marks the end of a trap handler. It simply restores the values of the PC and ΔPC registers using the values stored by the TRP instruction.

Masking Modes MMs do not apply to the RET command and are ignored.

7.6 Proposed Macro-Instructions

7.6.1 SZ — Skip If Zero

Instruction	**SZ A**
Format	`0mm 011 XXX XXX aaa XXX`
Semantics	$X = 0 \Rightarrow$ PC $\leftarrow$ PC $+ \Delta$PC
Cycles	6–7 (see below)

Notes Bits marked 'X' in the instruction format above are don't-care values. The 9-bit literal is ignored for this instruction. This instruction needs an extra clock cycle when the skip is taken.

The SZ instruction tests register A. If the register is zero (depending on the MM used), the next instruction is skipped. Skipping is performed by adding ΔPC to PC, hence are skipped along the current direction of the PC.

Masking Modes MMs apply to the comparison. Vector and scalar mode yield identical effects, testing the entire word. X and Y mode only test the rd and wo's bits respectively.

7.6.2 SNZ — Skip Unless Zero

Instruction	**SNZ A**
Format	`0mm 100 XXX XXX aaa XXX`
Semantics	$X \neq 0 \Rightarrow$ PC $\leftarrow$ PC $+ \Delta$PC
Cycles	6–7 (see below)

Notes This instruction is almost identical to SZ above.

Bits marked 'X' in the instruction format above are don't-care values. The 9-bit literal is ignored for this instruction. This instruction needs an extra clock cycle when the skip is taken.

Table 3: Values of ΔPC immediately after a DZ instruction.

	`DZ A`	`DZ.x A`	`DZ.y A`
$A = 0$	$(-1,-1)$	$(-1,0)$	$(0,-1)$
$A \neq 0$	$(1,1)$	$(1,0)$	$(0,1)$

The SNZ instruction tests register A. If the register is non-zero (depending on the MM used), the next instruction is skipped. Skipping is performed by adding ΔPC to PC, hence instructions are skipped along the current direction of the PC.

Masking Modes MMs apply to the comparison. Vector and scalar mode yield identical effects, testing the entire word. X and Y mode only test the rd and wo's bits respectively.

7.6.3 DZ — Divert If Zero

Instruction	**DZ A**
Format	`0mm 101 XXX XXX aaa XXX`
Semantics	ΔPC $\leftarrow (0,0)$; ΔPC $\leftarrow (-1,-1)$;
	$X = 0 \Rightarrow \Delta$PC $\leftarrow (1,1)$
Cycles	8–9 (see below)

Notes Bits marked 'X' in the instruction format above are don't-care values. The 9-bit literal is ignored for this instruction. This instruction needs an extra clock cycle if the register *is* zero.

The DZ instruction tests register A. If the register is zero (depending on the MM used), the PC moves southwest (ΔPC $= (1,1)$). Otherwise, the PC moves northeast (ΔPC $= (-1,-1)$).

The ΔPC register is updated immediately. The next instruction to be fetched will be at address PC $+ \Delta$PC.

Masking Modes MMs apply to the diversion (ΔPC assignment). Vector and scalar modes are identical, assigning southwest/northeast directions to ΔPC. X mode will assign west/east directions; Y mode will asign south/north directions. Table 3 illustrates this.

7.6.4 DNZ — Divert Unless Zero

Instruction	**DNZ A**
Format	`0mm 110 XXX XXX aaa XXX`
Semantics	ΔPC $\leftarrow (0,0)$; ΔPC $\leftarrow (-1,-1)$;
	$X = 0 \Rightarrow \Delta$PC $\leftarrow (1,1)$
Cycles	8–9 (see below)

Notes Bits marked 'X' in the instruction format above are don't-care values. The 9-bit literal is ignored for this instruction. This instruction needs an extra clock cycle if the register *is* non-zero.

Table 4: Values of ΔPC immediately after a `DNZ` instruction.

	`DNZ A`	`DNZ.x A`	`DNZ.y A`
$A = 0$	$(1, 1)$	$(1, 0)$	$(0, 1)$
$A \neq 0$	$(-1, -1)$	$(-1, 0)$	$(0, -1)$

The `DNZ` instruction tests register A. If the register is non-zero (depending on the MM used), the PC moves southwest (ΔPC $= (1, 1)$). Otherwise, the PC moves northeast (ΔPC $= (-1, -1)$).

The ΔPC register is updated immediately. The next instruction to be fetched will be at address PC $+$ ΔPC.

Masking Modes MMs apply to the diversion (ΔPC assignment). Vector and scalar modes are identical, assigning southwest/northeast directions to ΔPC. X mode will assign west/east directions; Y mode will asign south/north directions. Table 4 illustrates this.

7.7 Proposed Assembly Aliases

It is evident from the above discussion that the Fungus instruction set is as minimalistic as possible. It would help, however, to define a set of Assembly micro-instructions to simplify commonly used tasks and to fill in the missing Fungus instructions.

However, a macro-assembler for Fungus is a highly non-trivial task. Expanding single Assembly commands to multiple machine instructions disrupts the topology of the programme. This *could* be detected by the assembler, but correcting it in an elegant, space-efficient manner is quite difficult.

Thus, Fungus assemblers should instead use instruction *aliases*: commands that expand to *single* machine instructions. This simplifies disassembly too.

```
ADD x,a,a ; SHL x,a
DZ.x $0 ; GOE
DNZ.x $0 ; GOW
DZ.y $0 ; GON
DNZ.y $0 ; GOS
ADD $1,$0,R ; JR R
```