The Puck Virtual Machine Specification

Tristan Brandt, Lukas Gail, Thomas Letschert, Björn Pfarr

August 2017

Contents

1 Introduction **2**

2 The Puck Virtual Machine **2**

2.1 The Instruction Set of the Puck Virtual Machine . . . 2

2.1.1 (Unsigned) Integer Arithmetic Operators . . . 2

2.1.2 Bitwise Operators . . . 3

2.1.3 (Unsigned) Integer Comparison Operators . . . 3

2.1.4 Flow Control Instructions . . . 4

2.1.5 Memory Instructions . . . 4

2.1.6 I\O Instructions . . . 4

2.1.7 Register Data Instructions . . . 5

2.1.8 Floating Point Arithmetic Operators . . . 5

2.1.9 Floating Point Comparison Operators . . . 6

2.1.10 Operators with immediate operands . . . 6

2.2 The File Format of Executable Images . . . 6

3 The Puck Linker **6**

3.1 File Format of Object Files . . . 7

3.2 The Process of Linking . . . 8

3.3 Symbols defined by the linker . . . 8

3.4 Conventions defined by the linker . . . 8

4 The Puck Assembly Language **8**

4.1 Comments . . . 9

4.2 Instructions . . . 9

4.3 Identifiers . . . 10

4.3.1 Register Arguments . . . 10

4.3.2 Address Arguments . . . 11

4.3.3 Number Arguments . . . 11

4.4 Labels . . . 11

4.5 Directives . . . 11

4.5.1 Name of the Object . . . 11

4.5.2 Import and Export . . . 11

4.5.3 Special Addresses . . . 11

4.5.4 Static Data . . . 12

5 The Puck standard object convention **12**
5.1 The Stack 12
5.2 The Calling Convention 12

1 Introduction

This document describes the Puck virtual machine, the Puck linker, and the Puck assembler.

2 The Puck Virtual Machine

The Puck Virtual Machine is a 32 bit big endian machine. It is equipped with 32 MiB of memory, in which the code and data reside. Instructions are of variable size. Instruction sizes range from just a single byte to up to 6 bytes. The Virtual machine is equipped with 32 registers, each of which can contain four bytes (a word). The registers are named $0 to $31. $0 always has the value 0 and can not be written to. A word contains four bytes.

The Puck Virtual Machine has in contrast to modern machines no hardware supported stack. The programmer needs to manage he stack by keeping track of the stack pointer and manually allocating and freeing space on the stack by incrementing or decrementing the stack pointer.

2.1 The Instruction Set of the Puck Virtual Machine

Instructions for the Puck Virtual Machine always start with an Op-Code, followed by a variable amount of bytes, the immediate arguments. Immediate arguments may refer to specific registers or contain a specific value. All immediate values are big endian encoded.

Operators ending with 'f' interpret their operands as 32 bit floating point values. Operators ending with 'u' are specific to 32 bit unsigned integers. Operators ending with 'i' are used with 32 bit signed integers in two's complement. Operators without such a postfix may be used with unsigned and signed integers.

2.1.1 (Unsigned) Integer Arithmetic Operators

Integer Arithmetic instructions take up the Op-Codes from 0x00 to 0x07 and are operators.

Operators are four bytes large and each one has three one byte immediate values, each of which refer to a register. Those registers are called (in order): dst, op1, and op2.
All operators work in the same way: They take the values of op1 and op2, apply some operation to it, and write the result in dst.

Table 1: Integer Arithmetic Instructions

Op-Code	Instruction	Description
0x00	ADD	dst ← op1 + op2, sign agnostic
0x01	SUB	dst ← op1 - op2, sign agnostic
0x02	MULU	dst ← op1 * op2, unsigned
0x03	DIVU	dst ← op1 / op2, unsigned
0x04	MODU	dst ← op1 % op2, unsigned
0x05	MULI	dst ← op1 * op2, signed
0x06	DIVI	dst ← op1 / op2, signed
0x07	MODI	dst ← op1 % op2, signed

2.1.2 Bitwise Operators

Bitwise operators apply an operation to the bit pattern of their operands and are not specific to the type of their operands.

Table 2: Bitwise Operators

Op-Code	Instruction	Description
0x10	AND	dst ← op1 & op2, bitwise AND
0x11	OR	dst ← op1 | op2, bitwise OR
0x12	XOR	dst ← op1 ^ op2, bitwise XOR
0x13	SL	dst ← op1 << op2, left shift (towards to MSB)
0x14	SR	dst ← op1 >> op2, right shift

2.1.3 (Unsigned) Integer Comparison Operators

The result of the logical operators is 0 if the expression is false and 1 if it is true.

Table 3: Integer Comparison Instructions

Op-Code	Instruction	Description
0x20	EQ	dst ← op1 = op2, sign agnostic
0x21	NE	dst ← op1 ≠ op2, sign agnostic
0x22	LTI	dst ← op1 < op2, signed
0x23	LEI	dst ← op1 ≤ op2, signed
0x24	GTI	dst ← op1 > op2, signed
0x25	GEI	dst ← op1 ≥ op2, signed
0x26	LTU	dst ← op1 < op2, unsigned
0x27	LEU	dst ← op1 ≤ op2, unsigned
0x28	GTU	dst ← op1 > op2, unsigned
0x29	GEU	dst ← op1 ≥ op2, unsigned

2.1.4 Flow Control Instructions

To control the flow of the program's execution the Virtual Machine has the instructions with the Op-Codes from 0x30 to 0x36. They consist of conditional jumps to immediate addresses, unconditional jumps to immediate addresses or addresses from registers, and unconditional jumps that place the return address into a register and take their destination from the immediate value or a register. There also is an instruction that stops execution.

Table 4: Flow Control Instructions

Op-Code	Instruction	Description
0x30	BRT	Has two arguments: 1 byte register number (condition) and 4 byte address (destination). if (least significant byte of condition != 0) program counter ← destination
0x31	BRF	Same argument layout as BRT if (least significant byte of condition == 0) program counter ← destination
0x32	JMP	Has a single 4 byte immediate value (destination). program counter ← destination
0x33	JMPR	Has a 1 byte immediate value (destination register). program counter ← destination
0x34	CALL	Has a 1 byte argument (return register) and a 4 byte immediate value (destination). return register ← program counter + 1, program counter ← destination
0x35	CALLR	Has two 1 byte arguments (return register and destination register). return register ← program counter + 1, program counter ← destination
0x36	HALT	Has no immediate arguments. Ends execution.

2.1.5 Memory Instructions

The memory instructions from 0x40 to 0x45 are used to load data from memory into registers and store data from registers into memory.
Each memory instruction has two one byte immediate values. The first one refers to the register that the value is loaded into or read from. The second one refers to a register where the address of the memory operation is stored.

Table 5: Memory Instructions

Op-Code	Instruction	Description
0x40	LDB	Reads the byte at the memory location into the register.
0x41	LDHW	Reads the halfword at the memory location into the register.
0x42	LDW	Reads the word at the memory location into the register.
0x43	STB	Stores the byte from the register at the memory location
0x44	STHW	Stores the halfword from the register at the memory location
0x45	STW	Stores the word from the register at the memory location

2.1.6 I\O Instructions

I\O instructions were added to introduce the concept of standard in- and output. Currently there are eight instructions in this category, divided into four input instructions and four output instructions. They each have a single one byte immediate argument which refers to a register. The output instructions read the contents of that register, interpret them as a specific type, and output them formatted into standard output. The input instructions parse a formatted value of the appropriate type and place it into the specified register.

Table 6: I\O Instructions

Op-Code	Instruction	Description
0x50	INC	Reads a single byte from standard input and places it into the register.
0x51	OUTC	Prints the least significant byte of the register into standard output
0x52	INU	Parses a 32 bit unsigned integer and places it into the register
0x53	OUTU	Prints the contents of the register as a 32 bit unsigned integer to standard output
0x54	INI	Parses a 32 bit signer integer and places it into the register
0x55	OUTI	Prints the contents of the register as a 32 bit signed integer to standard output
0x56	INF	Parses a 32 bit floating point number from standard input and places it into the register
0x57	OUTF	Prints the contents of the register formatted as a 32 bit floating point number

2.1.7 Register Data Instructions

The instructions from 0x60 to 0x63 are used to directly work with data in registers. Three of those instructions are used to place static data into registers. These have two immediate arguments. The first on refers to the target register, the second one is of varying size and contains the data.

The value of the immediate argument is placed starting from the least significant byte. The rest of the register will be set to 0.

Table 7: Register Data Instructions

Op-Code	Instruction	Description
0x60	SETB	Immediate value is a single byte large. The least significant byte is overwritten with that value.
0x61	SETHW	Immediate value is two bytes large. The two least significant bytes are filled with that data.
0x62	SETW	Immediate value is four bytes large. Register is completely filled with that data.
0x63	CP	Has two register arguments. Copies the complete contents of the second register into the first one.

2.1.8 Floating Point Arithmetic Operators

Floating point operators work in the same way as integer operators.

Table 8: Floating Point Arithmetic Operators

Op-Code	Instruction	Description
0x70	ADDF	dst ← op1 + op2
0x71	SUBF	dst ← op1 - op2
0x72	MULF	dst ← op1 * op2
0x73	DIVF	dst ← op1 / op2

2.1.9 Floating Point Comparison Operators

This instructions are used to compare floating point values. Their result is an integral value. The result of this logical operators is 0 if the expression is false and 1 if it is true.

Table 9: Floating Point Comparison Instructions

Op-Code	Instruction	Description
0x80	LTF	dst $\leftarrow$ op1 $<$ op2
0x81	LEF	dst $\leftarrow$ op1 $\leq$ op2
0x82	GTF	dst $\leftarrow$ op1 $>$ op2
0x83	GEF	dst $\leftarrow$ op1 $\geq$ op2

2.1.10 Operators with immediate operands

Those operators function a little differently than other operators. They are seven bytes large. The first immediate byte refers to the register the result will be stored in while the second immediate byte refers to the register of the first operand. The next four bytes are the value of the second operand. They were defined to simplify working with a stack.

Table 10: Operators with immediate arguments

Op-Code	Instruction	Description
0x90	ADDC	dst $\leftarrow$ op1 + op2
0x91	SUBC	dst $\leftarrow$ op1 - op2

2.2 The File Format of Executable Images

A Puck executable contains an 8 byte header, followed by the executable code. Table 11 shows the exact format of the file.

Table 11: Executable File Format

Size	Description
4 bytes	Magic number. 0x5055434B ("PUCK" in ASCII encoding)
4 bytes	Number of bytes in the code segment N_c
N_c bytes	Code segment

3 The Puck Linker

The Puck Linker is used to combine multiple files of object code into a single executable. It is also used to determine the procedure that will be used as the entry point of the program. When invoking the Puck Linker it is possible to specify an object path. This is done by providing the command line argument `-op` followed by the path.

The object path is used by the linker to find the specified object files. Object names are case sensitive.

To specify the procedure that will be used as the entry point use the command line argument `-proc` followed by the procedure name. If this is not specified the linker will look for a procedure named `main` in the linked object.

Only one command line argument without a prior switch may be used, which is the name of the object file that should be linked. This file will be searched in the object path. All other object files that may be needed for linking will be selected automatically based on imported symbols and will be searched in the object path.

3.1 File Format of Object Files

Object Files contain answers to the following questions about modules:

- What symbols does this module export? Where are those symbols located in the code of this module? Are they eligible as entry points?
- What symbols does this module import? Where in the code of this module are those symbols used?
- Where in the code of this module are internal addresses used?
- Which symbols are used as static initialization?

Table 12 describes the complete structure of an object file. All numbers and addresses used in the header are big endian encoded. Internal addresses used in the header are offsets into the code segment of the file.

Table 12: Object File Structure

Size	**Description**	
Null-terminated string	name of the object this file defines. Has to match filename without extension	
4 bytes	number of exported symbols N_e	
N_e	**Size**	**Description**
	Null-terminated string	name of exported label
	4 bytes	internal address
4 bytes	number of imported symbols N_i	
N_i	**Size**	**Description**
	Null-terminated string	name of object the label is imported from
	Null-terminated string	name of imported label
	4 bytes	number of occurrences N_o
	N_o * 4 bytes	internal addresses of usages
4 bytes	number of executable addresses N_x	
N_x	**Size**	**Description**
	Null-terminated string	name of the executable address
	4 bytes	internal address
4 bytes	number of static initialization addresses N_s	
N_s	**Size**	**Description**
	Null-terminated string	name of the static initialization address
	4 bytes	internal address
4 bytes	number of internal jumps N_j	
N_j * 4 bytes	internal addresses of jump address	
4 bytes	number of bytes in the code segment N_C	
N_C bytes	code segment	

3.2 The Process of Linking

When linking the Linker will append the code segments of all object files. It will also prepend code that will call all necessary static initialization procedures as well as call the entry point of the program. It will also initialize the stack. The linker will resolve all symbolic and internal addresses of modules.

Internal addresses are resolved by adding the offset of this module's code in the executable to the internal address that is already located in the object code.

External addresses are resolved by searching the export table of the object the label is imported from. The internal address of the other object that is found there is added to the offset of this object in the executable. The sum will be placed at each place this external address should be used.

The linker will also prepend code that will call all necessary static initialization procedures as well as call the entry point of the program. To achieve this it needs to set up a stack and specify a register where the return address will be placed. The Linker will place the stack pointer (pointing to the next free byte) in register $31. When calling static initialization procedures or the entry point it will place the return address in $30 and is is recommended that user code follows this convention.

3.3 Symbols defined by the linker

The linker will define multiple symbolic addresses that may be used by user code. They will always be placed in a virtual object named `LinkerSymbols`, which means this name may not be used by user defined objects.

Table 13 shows the symbols that will be defined.

Table 13: Symbols defined by the linker

Symbol	Description
firstmemory	The first free address after the code and static data segment
lastmemory	The highest address in the address space of the target machine

3.4 Conventions defined by the linker

As the linker will call procedures it needs to define a stack and a calling convention for procedures without parameters. Please refer to section 5 to learn more about the conventions the linker abides to.

4 The Puck Assembly Language

As the binary code of the Puck Virtual Machine is not human-readable, an Assembly Language is defined to simplify working with it. The following sections will describe the syntax and semantics of the Assembly Language.

The Puck Assembly Language is a line oriented language. Each line consists of one or more tokens, which are separated by one or more whitespace characters. There are three kinds of assembly lines:

- Instructions: Those directly correspond to instructions in machine code.
- Labels: Those are used to define locations in the code that other instructions may refer to, for example jump statements and memory operations.

- Directives: Those are special instructions for the assembler and include import/export as well as static data directives.

4.1 Comments

Comments may be inserted by placing a semicolon (';'). The rest of the line will be ignored.

4.2 Instructions

The first token of an instruction line is the name of the instruction[1]. The following tokens are the arguments to that instruction.
Table 14 shows the argument count and type of the available instructions

Instruction	Argument 1	Argument 2	Argument 3
ADD	destination (register)	first operand (register)	second operand (register)
SUB	destination (register)	first operand (register)	second operand (register)
MULU	destination (register)	first operand (register)	second operand (register)
DIVU	destination (register)	first operand (register)	second operand (register)
MODU	destination (register)	first operand (register)	second operand (register)
MULI	destination (register)	first operand (register)	second operand (register)
DIVI	destination (register)	first operand (register)	second operand (register)
MODI	destination (register)	first operand (register)	second operand (register)
AND	destination (register)	first operand (register)	second operand (register)
OR	destination (register)	first operand (register)	second operand (register)
XOR	destination (register)	first operand (register)	second operand (register)
SL	destination (register)	first operand (register)	second operand (register)
SR	destination (register)	first operand (register)	second operand (register)
EQ	destination (register)	first operand (register)	second operand (register)
NE	destination (register)	first operand (register)	second operand (register)
LTI	destination (register)	first operand (register)	second operand (register)
LEI	destination (register)	first operand (register)	second operand (register)
GTI	destination (register)	first operand (register)	second operand (register)
GEI	destination (register)	first operand (register)	second operand (register)
LTU	destination (register)	first operand (register)	second operand (register)
LEU	destination (register)	first operand (register)	second operand (register)
GTU	destination (register)	first operand (register)	second operand (register)
GEU	destination (register)	first operand (register)	second operand (register)
JMP	destination (address)		
JMPR	destination (register)		
CALL	return (register)	destination (address)	
CALLR	return (register)	destination (register)	
HALT			
BRF	condition (register)	destination (address)	
BRT	condition (register)	destination (address)	
LDB	destination (register)	address (register)	
LDHW	destination (register)	address (register)	
LDW	destination (register)	address (register)	

[1] Please refer to 2.1 for more information about available instructions

STB	source (register)	address (register)	
STHW	source (register)	address (register)	
STW	source (register)	address (register)	
INC	destination (register)		
OUTC	source (register)		
INU	destination (register)		
OUTU	source (register)		
INI	destination (register)		
OUTI	source (register)		
INF	destination (register)		
OUTF	source (register)		
SETB	destination (register)	value (number)	
SETHW	destination (register)	value (number)	
SETW	destination (register)	value (number or address)	
CP	destination (register)	source (register)	
ADDF	destination (register)	first operand (register)	second operand (register)
SUBF	destination (register)	first operand (register)	second operand (register)
MULF	destination (register)	first operand (register)	second operand (register)
DIVF	destination (register)	first operand (register)	second operand (register)
LTF	destination (register)	first operand (register)	second operand (register)
LEF	destination (register)	first operand (register)	second operand (register)
GTF	destination (register)	first operand (register)	second operand (register)
GEF	destination (register)	first operand (register)	second operand (register)
ADDC	destination (register)	first operand (register)	second operand (number)
SUBC	destination (register)	first operand (register)	second operand (number)

Table 14: Argument layout of instructions

4.3 Identifiers

Identifiers are used as labels or object names. Identifiers may only contain the characters 'a' to 'z', their uppercase counterparts, as well as underscores and digits. Identifiers may not start with a digit.

4.3.1 Register Arguments

Register arguments are either the name of a register ($0 to $31) or one of the defined aliases. Aliases are defined for convenience. They do not define any semantics and the implied semantics do not have to be adhered to. Table 15 contains an overview of existing aliases.

Alias	Corresponding Register
NULL	$0
RETURN	$30
STACK	$31

Table 15: Register Aliases

4.3.2 Address Arguments

Addresses are the name of a label. It is not allowed to use numbers as addresses, only symbolic addresses are allowed.

4.3.3 Number Arguments

Unsigned numbers may be hexadecimal values formatted by an "0x"-prefix or decimal values. Signed numbers may be an unsigned number or a "-" followed by an unsigned number.

Important: Note that negative numbers are simply translated to their twos complement representation. Depending on the instruction it may be interpreted as an unsigned integer.

4.4 Labels

To define a label that can be used as a symbolic address a line must contain exactly one token. This token must start with an identifier, which will be the name of the label, and end with a colon (':').

4.5 Directives

Directives are special instructions to the assembler, which are not directly translated into code. A directive line consists of one ore more tokens. The first token starts with a period ('.'), followed by the name of the directive. The other tokens contain the arguments to the directive.

4.5.1 Name of the Object

The `object` directive may be used to declare the name of the object this file defines. It takes a single identifier as its argument which is the name of the object. It must be defined if using separate translation and it is recommended that is matches the file name of the source file. The defined name will be used as the name of the object file by the assembler. When it is not defined a generic name will be chosen. When multiple `object` directives are used in a single source file the last one will be used.

4.5.2 Import and Export

The `import` and `export` directives are used to implement separate translation of multiple modules.

`import` takes two or three arguments, all of which are identifiers. The first is the name of the object a label is imported from. The second is the name of the label that shall be imported. The third is the name of the address that will be used internally. If the third argument is omitted the original name will be used. The `import` directive declares an external label and tells the assembler that it should fill the header of the generated object file with information that is used to resolve that address by the linker.

`export` has a single argument, which is the name of an internal label. It declares that this label may be `imported` by other objects and tells the assembler to fill the header of the generated object file with information necessary to resolve this address.

4.5.3 Special Addresses

There are two kinds of special addresses.

Executable Addresses are the addresses of code that may be used as entry points to the program. They are declared using the `executable` directive. It takes a single argument, which has to be the name of a label defined in this module When encountering a `executable` directive the assembler will place an entry into the table of executable addresses in the object file's header.

Static Initialization Addresses are the addresses of code that has to be called to perform static initialization. They are declared using the `initialization` directive. It also takes the name of a label defined in this module as its only argument. The assembler will place an entry into the table of static initialization addresses in the object file's header.

Important: The code at those special addresses will be executed using the CALL instruction, placing the return address in register $30 (RETURN). This means the programmer has to make sure the code returns to the caller using a JMPR instruction.

4.5.4 Static Data

Static data in the Puck Virtual Machine lives in between code. This is a major difference compared to most modern languages, where static data has a dedicated data segment in the memory space of a process.
Interleaving code with static data has the advantage that address resolving of symbolic addresses becomes much easier.

To declare static data there are multiple directives: `byte`, `halfword`, and `word`. Each of these work in the same way: They insert a data segment of the appropriate size in the code right where they are placed. They may be used without or with exactly one argument. The argument has to be a signed number[2], which will be the initial value of the data segment. If the argument is omitted zero will be used as the initial value.

Important: Those directives can be placed *anywhere* inside the code. The programmer has to make sure execution may never get there or the static data may be interpreted as code. The recommended way to achieve this is placing all static data above any actual code or between a return jump and the address of the next procedure.

5 The Puck standard object convention

This section describes the conventions the linker and the standard library abide to.

5.1 The Stack

The linker will define a stack. The stack will grow *downwards*. The first stack slot will be at the highest address of the address space. Each subsequent stack slot is placed at the lower addresses. The linker will place the address of the next free stack byte in register $31 (alias STACK).

5.2 The Calling Convention

When the linker generates code to call procedures it will place the return address in register $30 (alias RETURN). All library code follows the same convention.

When calling procedures with parameters it gets a little more complicated. The arguments will be pushed to the stack in reverse order. The last argument will be pushed first, the second to last argument will be pushed

[2] Please refer to section 4.3.2 for more information about the grammar of numbers

second and so on. Because the stack grows downwards this means the first argument is placed at the lowest address and the last address is placed at the highest address.

Local variables are completely in the domain of the procedure that defines them. The procedure is responsible for allocating stack space for them and freeing it before returning. Remember that the offset of arguments in relation to the stack pointer changes when local variables are used this way, so accesses to arguments have to be adjusted appropriately.