

AssEmby Reference Manual

Applies to versions: 3.0.0

Last revised: 2023-12-10

Introduction

AssEmby is a custom processor architecture and assembly language implemented in .NET. It is designed to simplify the process of learning and writing in assembly language, while still following the same basic concepts and constraints seen in mainstream architectures such as x86.

AssEmby was designed and implemented in its entirety by [Tolly Hill](#).

Table of Contents

- [AssEmby Reference Manual](#)
 - [Introduction](#)
 - [Table of Contents](#)
 - [Technical Information](#)
 - [Basic Syntax](#)
 - [Mnemonics and Operands](#)
 - [Comments](#)
 - [Labels](#)
 - [Operand Types](#)
 - [Register](#)
 - [Literal](#)
 - [Character Literal](#)
 - [Address](#)
 - [Pointer](#)
 - [Registers](#)
 - [Register Table](#)
 - [rpo - Program Offset](#)
 - [rsf - Status Flags](#)
 - [rrv - Return Value](#)
 - [rfp - Fast Pass Parameter](#)
 - [rso - Stack Offset](#)
 - [rsb - Stack Base](#)
 - [rg0 - rg9 - General Purpose](#)
 - [Moving Data](#)
 - [Moving with Literals](#)
 - [Moving with Registers](#)

- Moving with Memory
- Maths and Bitwise Operations
 - Addition and Multiplication
 - Subtraction
 - Division
 - Shifting
 - Bitwise
 - Random Number Generation
- Negative Numbers
 - Arithmetic Right Shifting
 - Extending Smaller Signed Values
 - The Overflow Flag vs. the Carry Flag
- Floating Point Numbers
 - Floating Point Math
 - Converting Between Integers and Floats
 - Converting Between Floating Point Sizes
- Jumping
- Comparing, Testing, and Branching
 - Comparing Unsigned Numbers
 - Comparing Signed Numbers
 - Comparing Floating Point Numbers
 - Testing Bits
 - Checking the Carry, Overflow, Zero, and Sign Flags
- Assembler Directives
 - PAD - Byte Padding
 - DAT - Byte Insertion
 - NUM - Number Insertion
 - MAC - Macro Definition
 - IMP - File Importing
 - ANALYZER - Toggling Assembler Warnings
- Console Input and Output
- File Handling
 - Opening and Closing
 - Reading and Writing
 - Other Operations
- The Stack
 - Using the Stack to Preserve Registers
- Subroutines
 - Fast Calling
 - Return Values

- Subroutines and the Stack
- Passing Multiple Parameters
- Text Encoding
- Escape Sequences
- Instruction Data Type Acceptance
- Status Flag Behaviour
- Full Instruction Reference
 - Base Instruction Set
 - Signed Extension Set
 - Floating Point Extension Set
 - Extended Base Set
 - External Assembly Extension Set
 - Memory Allocation Extension Set
- ASCII Table

Technical Information

Bits	64 (registers, operands & addresses)
Word Size	8 bytes (64-bits – called a Quad Word for consistency with x86)
Minimum Addressable Unit	Byte (8-bits)
Register Count	16 (10 general purpose)
Architecture Type	Register-memory
Endianness	Little
Signed Number Representation	Two's Complement
Branching	Condition code (status register)
Opcode Size	1 byte (base instruction set) / 3 bytes (extension sets)
Operand Size	1 byte (registers, pointers) / 8 bytes (literals, addresses/labels)
Instruction Size	1 byte – 17 bytes (current) / unlimited (theoretical)
Instruction Count	329 opcodes (114 unique operations)
Text Encoding	UTF-8

Basic Syntax

Mnemonics and Operands

All AssEmbly instructions are written on a separate line, starting with a **mnemonic** — a human-readable code that tells the **assembler** exactly what operation needs to be performed — followed by any **operands** for the instruction. The assembler is the program

that takes human-readable assembly programs and turns them into raw numbers — bytes — that can be read by the processor. This process is called **assembly** or **assembling**. An operand can be thought of as a parameter to a function in a high-level language — data that is given to the processor to read and/or operate on. Mnemonics are separated from operands with spaces, and operands are separated with commas.

A simple example:

```
MVQ rg0, 10
```

MVQ	rg0,	10
↑	↑	↑
Mnemonic	Operand	Operand

|-----Instruction-----|

You can have as many spaces as you like between commas and mnemonics/operands. There do not need to be any around commas, but there must be at least one between mnemonics and operands. Mnemonics and operands **cannot** be separated with commas.

Some instructions, like CFL, don't need any operands. In these cases, simply have the mnemonic alone on the line.

A line may end in a trailing comma as long as there is at least one operand on the line. Mnemonics taking no operands cannot be followed by a trailing comma.

Mnemonics correspond to and are assembled down to **opcodes**, numbers (in the case of AssEmbly either 1 or 3 bytes) that the processor reads to know what instruction to perform and what types of operands it needs to read. If an opcode starts with a 0xFF byte, the opcode will be 3 bytes long, with the second byte corresponding to an *extension set* number, and the third byte corresponding to an *instruction code*. If an opcode starts with any other byte, that single byte will be the entire opcode, with the byte corresponding to an *instruction code* in the base instruction set (extension set number 0x00). This means that opcodes in the form 0xFF, 0x00, 0x?? and opcodes in the form 0x?? refer to the same instruction, though this **only** works when the extension set is 0x00. A full list of extension sets and instruction codes can be found toward the end of the document.

The processor will begin executing from the **first line** in the file downwards, unless a label with the name ENTRY is defined, in which case the processor will start there (more in the following section on labels). Programs should *always* end in a HLT instruction (with no operands) to stop the processor.

For the most part, if an instruction modifies or stores a value somewhere, the **first** operand will be used as the **destination**.

Comments

If you wish to insert text into a program without it being considered by the assembler as part of the program, you can use a semicolon (;). Any character after a semicolon will be ignored by the assembler until the end of the line. You can have a line be entirely a comment without any instruction if you wish.

For example:

```
MVQ rg0, 10 ; This text will be ignored
; As will this text
DCR rg0 ; "DCR rg0" will assemble as normal
; Another Comment ; HLT - This is still a comment and will not insert an HLT
instruction!
```

Labels

Labels mark a position in the file for the program to move (**jump**) to or reference from elsewhere. They can be given any name you like (names are **case-sensitive**), but they must be unique per program and can only contain letters, numbers, and underscores. Label names **may not** begin with a number, however. A definition for a label is marked by beginning a line with a colon — the entire rest of the line will then be read as the new label name (excluding comments).

For example:

```
:AREA_1 ; This comment is valid and will not be read as part of the label
MVQ rg0, 10 ; :AREA_1 now points here
```

```
:Area2
DCR rg0 ; :Area2 now points here
HLT
```

Labels will point to whatever is directly below them, **unless that is a comment**. Comments are not assembled and so cannot be pointed to.

For example:

```
:NOT_COMMENT ; Comment 1
; Comment 2
; Comment 3
WCC 10
```

Here :NOT_COMMENT will point to WCC, as it is the first thing that will be assembled after the definition was written.

Labels can also be placed at the very end of a file to point to the first byte in memory that is not part of the program.

For example, in the small file:

```
MVQ rg0, 5
MVQ rg1, 10
:END
```

:END here will have a value of 20 when referenced, as each instruction prior will take up 10 bytes (more on this later).

The label name :ENTRY (case insensitive) has a special meaning. If it is present in a file, execution will start from wherever the entry label points to. If it is not present, execution will start from the first line.

For example, in this small file:

```
MVQ rg0, 5
:ENTRY
MVQ rg1, 10
HLT
```

When this program is executed, only the MVQ rg1, 10 line will run. MVQ rg0, 5 will never be executed.

Operand Types

There are four different types of operand that an instruction may be able to take. If an instruction supports multiple different possible combinations of operands, the assembler will automatically determine their types, you do not need to change the mnemonic at all.

Register

Registers are named, single-number stores separate from the processor's main memory. Most operations must be performed on them, instead of in locations in memory. They are referenced by using their name (currently always 3 letters — the first one being r, for example rg0). They always occupy a single byte of memory after being assembled.

The first operand in this instruction is a register:

```
MVQ rg0, 10
```

Literal

Literals are numeric values that are directly written in an assembly file and **do not change**. Their value is read literally instead of being subject to special consideration, hence the name. They always occupy 8 bytes (64-bits) of memory after assembly and can be written in base 10 (denary/decimal), base 2 (binary), or base 16 (hexadecimal). To write in binary, place the characters 0b before the number, or to write in hexadecimal, place 0x before the number.

The second operand in each of these instructions is a literal that will each represent the same number (ten) after assembly:

```
MVQ rg0, 10 ; Base 10
MVQ rg0, 0b1010 ; Base 2
MVQ rg0, 0xA ; Base 16
```

When writing literals, you can place an underscore anywhere within the number value to separate the digits. Underscores cannot be the first character of the number.

For example:

MVQ rg0, 1_000_000 ; This is valid, will be assembled as 1000000 (0xF4240)
MVQ rg0, 0x_10_0__000_0 ; This is still valid, underscores don't have to be uniform

MVQ rg0, _1_000_000 ; This is not valid
MVQ rg0, 0_x1_000_000 ; This is also not valid
MVQ rg0, _0x1_000_000 ; Nor is this

Literals can be made negative by putting a - sign directly before them (e.g. -42), or be made floating point by putting a . anywhere in them (e.g. 2.3). Floating point literals can also be made negative (e.g. -2.3). This is explained in more detail in the relevant sections on negative and floating point values.

Character Literal

In addition to numeric literals, literal values can also be written in the form of **character literals**. A character literal is a single character, surrounded by single quotes ('), that is assembled into the numeric representation of the contained character in UTF-8.

For example:

MVQ rg0, 'a' ; Move the value 97 to rg0
MVQ rg0, '*' ; Move the value 42 to rg0
MVQ rg0, '卜' ; Move the value 8946659 to rg0
; 8946659 is the numeric value of the UTF-8 bytes 0xE3, 0x83, 0x88 that represent '卜' when interpreted as little endian

MVQ rg0, 'aa' ; Results in an error (character literals can only contain a single character)
MVQ rg0, '' ; Results in an error (character literals cannot be empty)

Character literals can also contain escape sequences, assuming the escape sequence is the only thing in the literal and there is only one.

For example:

MVQ rg0, '\\' ; Move the value 39 to rg0
MVQ rg0, '\\\\' ; Move the value 92 to rg0
MVQ rg0, '\\n' ; Move the value 10 to rg0
MVQ rg0, '\\uABCD' ; Move the value 9285610 to rg0
; 9285610 is the numeric value of the UTF-8 bytes 0xEA, 0xAF, 0x8D that represent the unicode codepoint U+ABCD when interpreted as little endian

MVQ rg0, '\\r\\n' ; Results in an error (character literals can only contain a single character)
MVQ rg0, '\\ ' ; Results in an error (the only closing quote of the literal has been escaped)

Escape sequences are explained in more detail and listed in full in a dedicated section toward the end of the document.

Address

An address is a value that is interpreted as a location to be read from, written to, or jumped to in a processor's main memory. In AssEmbly, an address is always specified by using a **label**. Once a label has been defined as seen earlier, they can be referenced by prefixing their name with a colon (:), similarly to how they are defined — only now it will be in the place of an operand. Like literals, they always occupy 8 bytes (64-bits) of memory after assembly.

Consider the following example:

```
:AREA_1
WCC 10
MVQ rg0, :AREA_1 ; Move whatever is stored at :AREA_1 in memory to rg0
```

Here :AREA_1 will point to the **first byte** (i.e. the start of the **opcode**) of the **directly subsequent assemble-able line** — in this case WCC. The second operand to MVQ will become the address that WCC is stored at in memory, 0 if it is the first instruction in the file. As MVQ is the instruction to move to a destination from a source, rg0 will contain 0xCD after the instruction executes (0xCD being the opcode for WCC <Literal>).

Another example, assuming these are the very first lines in a file:

```
WCC 10
:AREA_1
WCX :AREA_1 ; Will write "CA" to the console
```

:AREA_1 will have a value of 9, as WCC 10 occupies 9 bytes. Note that CA (the opcode for WCX <Address>) will be written to the console, *not* 9, as the processor is accessing the byte in memory *at* the address — *not* the address itself.

If, when writing an instruction, you want to utilise the address *itself*, rather than the value in memory at that address, insert an ampersand (&) after the colon, before the label name.

For example:

```
:AREA_1
WCC 10
MVQ rg0, :&AREA_1 ; Move 0 (the address itself) to rg0
WCX :&AREA_1 ; Will write "0" to the console
```

Pointer

So what if you've copied an address to a register? You now want to treat the value of a register as if it were an address in memory, not a number. This can be achieved with a **pointer**. Simply prefix a register name with an asterisk (*) to treat the register contents as a location to store to, read from, or jump to — instead of a number to operate on. Just like registers, they will occupy a single byte in memory after assembly.

For example:


```

:AREA_1
WCC 10
MVQ rg0, :&AREA_1 ; Move 0 (the address itself) to rg0
MVQ rg1, *rg0 ; Move the item in memory (0xCD) at the address (0) in rg0 to
rg1

```

rg1 will contain 0xCD after the third instruction finishes.

Registers

As with most modern architectures, operations in AssEmbly are almost always performed on **registers**. Each register contains a 64-bit number and has a unique, pre-assigned name. They are stored separately from the processor's memory, therefore cannot be referenced by an address, only by name. There are 16 of them in AssEmbly, 10 of which are *general purpose*, meaning they are free to be used for whatever you wish. All general purpose registers start with a value of 0. The remaining six have special purposes within the architecture, so should be used with care.

Please be aware that to understand the full operation and purpose for some registers, knowledge explained later on in the manual may be required.

Register Table

Byte	Symbol	Writable	Full Name	Purpose
0x00	rpo	No	Program Offset	Stores the memory address of the current location in memory being executed
0x01	rso	Yes	Stack Offset	Stores the memory address of the highest non-popped item on the stack
0x02	rsb	Yes	Stack Base	Stores the memory address of the bottom of the current stack frame
0x03	rsf	Yes	Status Flags	Stores bits representing the status of certain instructions
0x04	rrv	Yes	Return Value	Stores the return value of the last executed subroutine
0x05	rfp	Yes	Fast Pass Parameter	Stores a single parameter passed to a subroutine
0x00	rg0	Yes	General 0	<i>General purpose</i>

Byte	Symbol	Writable	Full Name	Purpose
6				
0x07	rg1	Yes	General 1	<i>General purpose</i>
0x08	rg2	Yes	General 2	<i>General purpose</i>
0x09	rg3	Yes	General 3	<i>General purpose</i>
0x0A	rg4	Yes	General 4	<i>General purpose</i>
0x0B	rg5	Yes	General 5	<i>General purpose</i>
0x0C	rg6	Yes	General 6	<i>General purpose</i>
0x0D	rg7	Yes	General 7	<i>General purpose</i>
0x0E	rg8	Yes	General 8	<i>General purpose</i>
0x0F	rg9	Yes	General 9	<i>General purpose</i>

rpo - Program Offset

Stores the memory address of the current location in memory being executed. For safety, it cannot be directly written to. To change where you are in a program, use a **jump instruction** (explained later on).

For example, in the short program (assuming the first instruction is the first in a file):

```
MVQ rg0, 10
DCR rg0
```

When the program starts, rpo will have a value of 0 — the address of the first item in memory. After the first instruction has finished executing, rpo will have a value of 10: its

previous value 0, plus 1 byte for the mnemonic's opcode, 1 byte for the register operand, and 8 bytes for the literal operand. rpo is now pointing to the opcode of the next instruction (DCR).

Note: rpo is incremented by 1 *before* an instruction begins execution, therefore when used as an operand in an instruction, it will point to the address of the **first operand, not to the address of the opcode**. It will not be incremented again until *after* the instruction has completed.

For example, in the instruction:

```
MVQ rg0, rpo
```

Before execution of the instruction begins, rpo will point to the opcode corresponding to MVQ with a register and literal. Once the processor reads this, it increments rpo by 1. rpo now points to the first operand: rg0. This value will be retained until after the instruction has completed, when rpo will be increased by 2 (1 for each register operand). This means there was an increase of 3 overall when including the initial increment by 1 for the opcode.

rsf - Status Flags

The status flags register is used to mark some information about previously executed instructions. While it stores a 64-bit number just like every other register, its value should instead be treated bit-by-bit rather than as one number.

Currently, the **lowest 5** bits of the 64-bit value have a special use — the remaining 59 will not be automatically modified as of current, though it is recommended that you do not use them for anything else in case this changes in the future.

The 5 bits currently in use are:

```
0b00...000000SFCZ
```

... = 52 omitted bits

Z = Zero flag

C = Carry flag

F = File end flag

S = Sign Flag

O = Overflow Flag

Each bit of this number can be considered as a true (1) or false (0) value as to whether the flag is “set” or not.

More information on using these flags can be found in the section on comparison and testing.

A full table of how each instruction modifies the status flag register can be found toward the end of the document.

rrv - Return Value

Stores the return value of the last executed subroutine. Note that if a subroutine doesn't return a value, rrv will remain unaffected.

For example:

```
:SUBROUTINE_ONE
...
...
...
RET 4 ; Return, setting rrv to the literal 4

:SUBROUTINE_TWO
...
...
...
RET ; Return, leaving rrv unaffected

CAL :SUBROUTINE_ONE
; rrv is now 4
CAL :SUBROUTINE_TWO
; rrv is still 4
```

More information can be found in the section on subroutines.

rfp - Fast Pass Parameter

Stores a single parameter passed to a subroutine. If such a parameter is not provided, rfp remains unaffected.

For example:

```
:SUBROUTINE_ONE
ADD rfp, 1
RET rfp

:SUBROUTINE_TWO
ADD rfp, 2
RET rfp

CAL :SUBROUTINE_ONE, 4 ; This will implicitly set rfp to 4
; rrv is now 5
CAL :SUBROUTINE_TWO, 6 ; This will implicitly set rfp to 6
; rrv is now 8
CAL :SUBROUTINE_TWO ; rfp will remain 6 here
; rrv is now 10
```

Implicitly setting rfp like this with the CAL instruction is called **fast passing** or **fast calling**, hence the name fast pass parameter.

Note that in practice, if a subroutine is designed to take a fast pass parameter, you should **always** explicitly provide it, even if you think `rfp` will already have the value you want. Similarly, you should not use `rfp` in a subroutine if it has not been explicitly set in its calls.

More information can be found in the section on subroutines.

`rso` - Stack Offset

Stores the memory address of the highest non-popped item on the stack (note that the stack fills from the end of memory backwards). If nothing is left on the stack in the current subroutine, it will be equal to `rsb`, and if nothing is left on the stack at all, it will still be equal to `rsb`, with both being equal to one over the highest possible address in memory (so will result in an error if that address is read from).

More information can be found in the dedicated sections on the stack and subroutines.

A simple example, assuming memory is 8192 bytes in size (making 8191 the highest address):

```
WCN rso ; Outputs "8192"
PSH 5   ; Push the literal 5 to the stack
WCN rso ; Outputs "8184" (stack values are 8 bytes)
POP rg0  ; Pop the just-pushed 5 into rg0
WCN rso  ; Outputs "8192"
```

`rsb` - Stack Base

Stores the memory address of the bottom of the current stack frame. `rsb` will only ever change when subroutines are being utilised — see the dedicated sections on the stack and subroutines for more info.

Note that `rsb` does not contain the address of the first item pushed to the stack, rather the address that all pushed items will be on top of.

`rg0` - `rg9` - General Purpose

These 10 registers have no special purpose. They will never be changed unless you explicitly change them with either a move operation, or another operation that stores to registers. These will be used most of the time to store and operate on values, as using memory or the stack to do so is inefficient (and in many cases impossible without copying to a register first), so should only be done when you run out of free registers.

Moving Data

There are four different instructions that are used to move data around without altering it in AssEmbly, each one moving a different number of bytes. `MVB` moves a single byte, `MVW` moves two (a.k.a. a word, 16-bits), `MVD` moves four (a.k.a. a double word, 32-bits), and `MVQ` moves eight (a.k.a. a quad word, 64-bits, a full number in AssEmbly).

Data can either be moved between two registers, from a register to a memory location, or from a memory location to a register. You cannot move data between two memory locations, you must use a register as a midpoint instead. To move data to or from a memory location, you can use either a label or a pointer.

The move instructions are also how the value of a register or memory location is set to a literal value. In a sense, they can be considered the equivalent of the = assignment operator in higher-level languages.

When using move instructions, the destination always comes first. The destination cannot be a literal.

Moving with Literals

An example of setting registers to the maximum literal values for each instruction:

```
MVQ rg0, 18446744073709551615 ; 64-bit integer limit
MVD rg1, 4294967295 ; 32-bit integer limit
MVW rg2, 65535 ; 16-bit integer limit
MVB rg3, 255 ; 8-bit integer limit
```

Or labels and pointers:

```
MVQ *rg0, 18446744073709551615 ; 64-bit integer limit
MVD *rg1, 4294967295 ; 32-bit integer limit
MVW :AREA_1, 65535 ; 16-bit integer limit
MVB :AREA_2, 255 ; 8-bit integer limit
```

Note that providing a literal over the limit for a given instruction will not result in an error. Instead, the **upper** bits that do not fit in the specified size will be truncated. All 64-bits will still be assembled into the binary (literals are **always** assembled to 8 bytes).

For example:

```
MVB rg0, 9874
```

MVB can only take a single byte, or 8 bits, but in binary 9874 is 10011010010010, requiring 14 bits at minimum to store. The lower 8 bits will be kept: 10010010 — the remaining 6 (100110) will be discarded. After this instruction has been executed, rg0 will have a value of 146.

Moving with Registers

When moving to and from a register, MVQ will update or read all of its bits (remember that registers are 64-bit). If any of the smaller move instructions are used, the **lower** bits of the register will be used, with the remaining upper bits of a destination register all being set to 0.

For example, assume that before the MVD instruction, rg1 has a value of 14,879,176,506,051,693,048:

MVW rg1, 65535

14,879,176,506,051,693,048 in binary is

1100111001111101011101000011001011110001100011001000100111111000, a full 64-bits, and 65535 is 11111111111111, requiring only 16 bits. MVW will only consider these 16 bits (if there were more they would have been truncated, see above section). Instead of altering only the lowest 16 bits of rg1, MVW will instead set all the remaining 48 bits to 0, resulting in a final value of 001111111111111111 — 65535 perfectly.

Similarly to literals, if a source register contains a number greater than what a move instruction can handle, the upper bits will be disregarded.

Moving with Memory

Unlike with registers, using different sizes of move instruction *will* affect how any bytes are read from memory. Bytes are read from or written to **starting** at the address in the given label or pointer, and only the required number for the given instruction are read or written (1 for MVB, 2 for MVW, 4 for MVD, 8 for MVQ). The instructions will *always* write these numbers of bytes, if a number to be moved takes up less, it will be padded with 0s.

Numbers are stored in memory in little endian encoding, meaning that the smallest byte is stored first, up to the largest. For example, the 32-bit number 2,356,895,874 is represented in hexadecimal as 0x8C7B6082, which can be broken down into 4 bytes: 8C, 7B, 60, and 82. When stored in memory, this order will be *reversed*, as follows:

Address	00	01	02	03	
Value	82	60	7B	8C	

This allows you to read a number with a smaller move instruction than what it was written with, whilst maintaining the same upper-bit truncating behaviour seen with literals and registers.

An example with a 64-bit number, 35,312,134,238,538,232 (0x007D7432F18C89F8):

Address	00	01	02	03	04	05	06	07	
Value	F8	89	8C	F1	32	74	7D	00	

Be aware that moving directly between two memory locations is not allowed. To move from one location in memory to another, use a register as a midpoint, like so:

```
MVQ rg0, :MEMORY_SOURCE
MVQ :MEMORY_DESTINATION, rg0
```

This also applies to pointers as well as labels (rg1 contains the source address, rg2 the destination):

```
MVQ rg0, *rg1
MVQ *rg2, rg0
```

When using any move instruction larger than MVB, be careful to ensure that not only the starting point is within the bounds of available memory, but also all of the subsequent bytes. For example, if you have 8192 bytes of available memory (making 8191 the maximum address), you cannot use MVQ on the starting address 8189, as that requires at least 8 bytes.

Maths and Bitwise Operations

Math and bitwise instructions operate **in-place** in AssEmbly, meaning the first operand for the operation is also used as the destination for the resulting value to be stored to. Destinations, and thus the first operand, must always be a **register**.

Mathematical and bitwise operations are always done with 64-bits, therefore if an address (i.e. a label or pointer) is used as the second operand, 8 bytes will be read starting at that address for the operation in little endian encoding (see the “moving with memory” section above for more info on little endian).

Addition and Multiplication

Examples of addition and multiplication:

```
MVQ rg0, 55 ; Set the value of rg0 to 55
ADD rg0, 45 ; Add 45 to the value of rg0, storing in rg0
; rg0 is now 100
MUL rg0, 3 ; Multiply the value of rg0 by 3, storing in rg0
; rg0 is now 300
MVQ rg1, rg0
MUL rg1, rg0 ; Multiply the value of rg1 by the value of rg0, storing in rg1
; rg1 is now 90000
```

Be aware that because there is a limit of 64-bits for mathematical operations, if an addition or multiplication operation results in this limit (18446744073709551615) being exceeded, the carry status flag will be set to 1, and the result will be wrapped around back to 0, plus however much the limit was exceeded by.

For example:

```
MVQ rg0, 18446744073709551615 ; Set rg0 to the 64-bit limit
ADD rg0, 10 ; Add 10 to rg0
; rg0 is now 10
```

```
MVQ rg0, 18446744073709551590 ; Set rg0 to the 64-bit limit take 25
ADD rg0, 50 ; Add 50 to rg0
; rg0 is now 24
```

In the specific case of adding 1 to a register, the ICR (increment) operation can be used instead.

```
MVQ rg0, 5
ICR rg0
; rg0 is now 6
```


Subtraction

An example of subtraction:

```
MVQ rg0, 55 ; Set the value of rg0 to 55
SUB rg0, 45 ; Subtract 45 from the value of rg0, storing in rg0
; rg0 is now 10
MVQ rg1, rg0
SUB rg1, rg0 ; Subtract the value of rg0 from rg1, storing in rg1
; rg1 is now 0
```

If a subtraction causes the result to go below 0, the carry status flag will be set to 1, and the result will be wrapped around up to the upper limit 18446744073709551615, minus however much the limit was exceeded by.

For example:

```
MVQ rg0, 0 ; Set rg0 to 0
SUB rg0, 1 ; Subtract 1 from rg0
; rg0 is now 18446744073709551615 (-1)

MVQ rg0, 25 ; Set rg0 to 25
SUB rg0, 50 ; Subtract 50 from rg0
; rg0 is now 18446744073709551591 (-25)
```

This overflowed value can also be interpreted as a negative number using two's complement if desired, which is explained further in the section on negative numbers.

In the specific case of subtracting 1 from a register, the DCR (decrement) operation can be used instead.

```
MVQ rg0, 5
DCR rg0
; rg0 is now 4
```

Division

There are three types of division in AssEmby: integer division (DIV), division with remainder (DVR), and remainder only (REM).

Integer division divides the first operand by the second, discards the remainder, then stores the result in the first operand. For example:

```
MVQ rg0, 12 ; Set rg0 to 12
DIV rg0, 4 ; Divide the value in rg0 by 4, storing the result in rg0
; rg0 is now 3

MVQ rg1, 23 ; Set rg1 to 23
DIV rg1, 3 ; Divide the value in rg1 by 3, storing the result in rg1
; rg1 is now 7 (the remainder of 2 is discarded)
```

Division with remainder, unlike most other operations, takes three operands, the first two being destination registers, and the third being the divisor. Like with the other operations, the first operand is used as the dividend and the result for the integer part of the division. The value of the second operand is not considered, the second operand simply being the register to store the remainder of the division.

For example:

```
MVQ rg0, 12 ; Set rg0 to 12
DVR rg0, rg1, 4 ; Divide the value in rg0 by 4, storing the integer result
in rg0, and remainder in rg1
; rg0 is now 3, rg1 is now 0
```

```
MVQ rg2, 23 ; Set rg2 to 23
DVR rg2, rg3, 3 ; Divide the value in rg2 by 3, storing the integer result
in rg2, and remainder in rg3
; rg2 is now 7, rg3 is now 2
```

Remainder only division is similar to integer division in that it only keeps one of the results, but this time the dividend (first operand) is overwritten by the remainder, and the integer result is discarded:

```
MVQ rg0, 12 ; Set rg0 to 12
REM rg0, 4 ; Divide the value in rg0 by 4, storing the remainder in rg0
; rg0 is now 0
```

```
MVQ rg1, 23 ; Set rg1 to 23
REM rg1, 3 ; Divide the value in rg1 by 3, storing the remainder in rg1
; rg1 is now 2 (the integer result of 7 is discarded)
```

Shifting

Shifting is the process of moving the bits in a binary number either up (left — SHL) or down (right — SHR) a certain number of places.

For example:

```
MVQ rg0, 0b11010
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
```

```
SHL rg0, 2
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
```

The bits were shifted 2 places to the left, and new bits on the right were set to 0.

Here's one for shifting right:

```
MVQ rg0, 0b11010
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
```

```
SHR rg0, 2
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
```

The bits were shifted 2 places to the right, and new bits on the left were set to 0.

If, like with the right shift example above, a shift causes at least one 1 bit to go off the edge (either below the first bit or above the 64th), the carry flag will be set to 1, otherwise it will be set to 0.

Bitwise

Bitwise operations consider each bit of the operands individually instead of as a whole number. There are three operations that take two operands (AND, ORR, and XOR), and one that takes only one (NOT).

Here are tables of how each two-operand operation will affect each bit

Bitwise And (AND):

```
+---+---+
| 0 | 1 |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
```

The AND operation will only set a bit to 1 if the bit in both operands is 1. For example:

```
MVQ rg0, 0b00101
AND rg0, 0b10100
; rg0 now has a value of 0b00100
```

Bitwise Or (ORR):

```
+---+---+
| 0 | 1 |
+---+---+---+
| 0 | 0 | 1 |
+---+---+---+
| 1 | 1 | 1 |
+---+---+---+
```

The ORR operation will set a bit to 1 if the bit in either operand is 1. For example:

```
MVQ r0, 0b00101
ORR r0, 0b10100
; r0 now has a value of 0b10101
```

Bitwise Exclusive Or (XOR):

	+	-	-	+	-	-	+
		\emptyset		1			
+	-	-	+	-	-	+	+
	\emptyset		\emptyset		1		
+	-	-	+	-	-	+	+
	1		1		\emptyset		
+	-	-	+	-	-	+	+

The XOR operation will set a bit to 1 if the bit in one, but not both, operands is 1. For example:

```
MVQ r0, 0b00101
XOR r0, 0b10100
; r0 now has a value of 0b10001
```

The NOT operation only takes a single operand, which must be a register. It simply “flips” the value of each bit (i.e. 1 becomes 0, 0 becomes 1).

For example:

```
MVQ r0, 0b00101
NOT r0
; r0 now has a value of 0b11010
```

Random Number Generation

The random number instruction (RNG) takes a single operand: the register to store the result in. The instruction always randomises all 64-bits of a register, meaning the result could be anywhere between 0 and 18446744073709551615.

Remainder only division (REM) by a value one higher than the desired maximum can be used to limit the random number to a maximum value, like so:

```
RNG rg0 ; rg0 could now be any value between 0 and 18446744073709551615
REM rg0, 5 ; rg0 is now constrained between 0 and 4 depending on its initial
value
```

To set a minimum value also, simply add a constant value to the result of the REM operation:

```
RNG rg0 ; rg0 could now be any value between 0 and 18446744073709551615
REM rg0, 5 ; rg0 is now constrained between 0 and 4 depending on its initial
value
ADD rg0, 5 ; rg0 is now constrained between 5 and 9
```

Negative Numbers

Negative numbers are stored using two's complement in AssEmbly, which means that negative values are stored as their positive counterpart with a bitwise NOT performed, then incremented by 1.

For example:

[illegible]

To switch between the positive and negative form of a number, use the `SIGN_NEG` instruction:

MVQ rg0, 9547
SIGN_NEG rg0 ; Performs the equivalent of "NOT rg0" then "ICR rg0" in one instruction
; rg0 is now -9547 (or 18446744073709542069 when interpreted as unsigned)

Stored values can be interpreted as either **unsigned** or **signed**. Unsigned values are always positive and use all 64 bits to store their value, giving a range of 0 to 18,446,744,073,709,551,615. Signed values can be either positive *or* negative and, while still stored using 64-bits, the highest bit is instead to store the sign. This gives a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 for signed operations. The number of distinct values is the same as unsigned values, but now half of the values are negative.

To check if the limits of a signed number have been exceeded after an operation instead of the limits of an unsigned number, the **overflow flag** should be used instead of the carry flag. This is explained in detail in the dedicated section on the overflow flag vs. the carry flag.

Numeric literals can be made negative by prepending the - sign onto them. Much of the base instruction set can take negative numbers as operands and work exactly as expected, though there are some exceptions. A full table of which instructions work as expected with negative values and which ones do not can be found toward the end of the document, though as a general rule, if an instruction has an equivalent that begins with SIGN_, you should use the signed one instead if negative values are expected.

Some instructions that work normally with negative values include ADD, SUB, and MUL. Some that do not include DIV and WCN, where the distinction between unsigned and signed values becomes important, as it will affect the result. The SIGN_DIV and SIGN_WCN instructions for example should be used instead when negative numbers are possible and desired. It is worth noting that instructions in the base instruction set (instructions not beginning with

an extension like SIGN_) always interpret numbers as unsigned; the reason some operations do not need a signed counterpart to counteract this is that the usage of two's complement allows overflowed unsigned results and signed results to have the same bit representation with these compatible operations.

For example:

```
MVQ rg0, 12
ADD rg0, -5
; rg0 is now 7, ADD works as expected with negative values
```

```
MVQ rg0, 12
SUB rg0, -5
; rg0 is now 17, SUB works as expected with negative values
```

```
MVQ rg0, 12
DIV rg0, -6
; rg0 is NOT -2, the SIGN_DIV instruction needs to be used instead
```

```
MVQ rg0, 12
SIGN_DIV rg0, -6
; rg0 is now -2, as expected
```

```
WCN rg0
; 18446744073709551614 has been printed to the console, as WCN always assumes
that the value is unsigned
SIGN_WCN rg0
; -2 has now been printed to the console, as expected
```

There are other instructions that have signed equivalents, these are simply used as an example. The signed operations also work on positive values, so the signed equivalent of relevant instructions should always be used wherever negative values are *possible* and desired, not just where they are guaranteed.

Arithmetic Right Shifting

When shifting bits to the right, there are two options: logical shifting (as explained in the previous shifting section), or arithmetic shifting. Arithmetic shifting should be used when you wish to shift a value whilst retaining its sign.

Arithmetic right shifts can be performed with the SIGN_SHR, which takes the same operands as SHR, but behaves slightly differently when the sign bit of the initial value is set.

For example:

```
MVQ rg0, 0b11010
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
; All omitted bits are 0
```

```

SIGN_SHR rg0, 2
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
; All omitted bits are 0

```

This behaviour is identical to SHR, as the value is not signed.

Here's an example with a negative value:

```

MVQ rg0, -26
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
; All omitted bits are 1

```

```

SIGN_SHR rg0, 2
; rg0:
; | Bit | ... | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
; | Value | ... | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
; All omitted bits are 1

```

Because the sign bit was set in the original value, all new bits shifted into the most significant bit were set to 1 instead of 0, keeping the sign of the result the same as the initial value.

The behaviour of the carry flag is also altered when performing an arithmetic shift. Where SHR sets the carry flag if any 1 bit is shifted past the least significant bit and discarded, SIGN_SHR instead sets the carry flag if any bits **not equal to the sign bit** are discarded. This means that for negative initial values, any 0 bit being discarded will set the carry bit, and for positive initial values, any 1 bit being discarded will set the carry bit.

Using an 8-bit number for demonstration, the behaviour of a **logical shift** (SHR) looks like this:

```

-26 >> 1
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | -> discarded bit not 1,
UNSET carry flag
  \  \  \  \  \  \  \
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
= 115

```

Whereas the behaviour of an **arithmetic shift** (SIGN_SHR) looks like this:

```

-26 >> 1
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | -> discard not equal to
sign, SET carry flag
  \  \  \  \  \  \  \
|  |  |  |  |  |  |  |

```

```
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
= -13
```

Extending Smaller Signed Values

Operations on signed numbers will always expect them to be 64-bits in size, with the 64th bit as the sign bit. If you have a signed value stored in a smaller format, using the 8th (byte), 16th (word), or 32nd (double word) bits as the sign bit, you can use one of the extension instructions (SIGN_EXB, SIGN_EXW, and SIGN_EXD respectively) to convert the number to its equivalent value in 64 bits.

For example:

```
MVW rg0, 0b1111111101011011
; rg0 is 0b000000000000000000000000000000000000000000001111111101011011
in binary
; This is -165 when considering only the lower 16 bits as a signed number,
; however we need the value to occupy all 64-bits to be interpreted properly.
; As of current, even the signed instructions will read rg0 as 65371
```

```
SIGN_EXW rg0 ; SIGN_EXW is for extending 16->64, use SIGN_EXB for 8->64 or
SIGN_EXD for 32->64
; rg0 is now
0b111111111111111111111111111111111111111111111111111111101011011 in binary
; This occupies all 64-bits, so rg0 will now work correctly as -165
```

Using the extending instructions with a positive value will not affect the value of the register up to the specified size of bits, though any bits higher than the number supported by the used extend instruction will be set to 0 instead of 1.

For example:

```
MVB rg0, 12
; rg0 is 0b000000000000000000000000000000000000000000000000000000000000001100
in binary
```

```
SIGN_EXB rg0
; rg0 is unchanged
```

```
MVW rg0, 569
; rg0 is 0b000000000000000000000000000000000000000000000000000000001000111001
in binary
; rg0 doesn't fit in a single byte!
```

```
SIGN_EXB rg0
; rg0 is now
0b00000000000000000000000000000000000000000000000000000000111001
; Any bits higher than the 8th bit have been unset, making rg0 equal to only
57
```


The second example here caused part of the number to be lost as `SIGN_EXB` was used when the value was larger than 8-bits. A similar scenario will occur if a negative value requires more bits than the used extend instruction can handle, though the upper bits will all be set to 1 instead of 0 in this case.

Converting from a larger size of signed integer to a smaller one is as simple as taking only the desired number of lower bits. Assuming the value can fit within the target signed integer size's limits, no specific operation needs to be used.

The Overflow Flag vs. the Carry Flag

As explained earlier, during most mathematical operations the carry flag is set whenever a subtraction goes below 0, or an addition goes above 18446744073709551615. This is useful in unsigned arithmetic, as it will inform you when the result of an operation is not mathematically correct, however in signed arithmetic, it cannot be used for this purpose. To overcome this, the status flag register also contains an **overflow flag**. This flag is set specifically when the result of an operation is incorrect when interpreted as a *signed* value. It has no useful meaning during unsigned arithmetic.

Some examples:

```
MVQ rg0, 10
SUB rg0, 5
; As unsigned, rg0 is now 5. As signed it is also 5.
; Carry flag has been UNSET, answer is correct as unsigned.
; Overflow flag has been UNSET, answer is correct as signed.
```

```
MVQ rg0, 0
SUB rg0, 5
; As unsigned, rg0 is now 18446744073709551611. As signed it is -5.
; Carry flag has been SET, answer is incorrect as unsigned.
; Overflow flag has been UNSET, answer is correct as signed.
```

```
MVQ rg0, 0x7FFFFFFFFFFFFFFF ; (hexadecimal for 9223372036854775807 as both
signed and unsigned)
ADD rg0, 5
; As unsigned, rg0 is now 9223372036854775812. As signed it is -
9223372036854775804.
; Carry flag has been UNSET, answer is correct as unsigned.
; Overflow flag has been SET, answer is incorrect as signed.
```

```
MVQ rg0, 0x7FFFFFFFFFFFFFFF
SUB rg0, 0xFFFFFFFFFFFFFFF
; As unsigned, rg0 is now 9223372036854775808. As signed it is -
9223372036854775808.
; Carry flag has been SET, answer is incorrect as unsigned.
; Overflow flag has been SET, answer is incorrect as signed.
```

Floating Point Numbers

AssEmbly has instructions to perform operations on floating point values. These instructions work with the IEEE 754 double-precision floating point format (also known as float64 or double). In this format, values, including whole numbers, are stored using an entirely different format from regular integer values, which means that, unlike with signed values, very little of the base instruction set can work with floating point values. Instead, instructions in the floating point instruction set (mnemonics starting with FLPT_) must be used. There is a full table towards the end of the document that details which instructions accept which formats of data.

To make an integer literal into a floating point literal, it must contain a decimal point (.). Any numeric literal containing a decimal point will be assembled into a 64-bit float.

For example:

```
MVQ rg0, 5
; rg0 is 0x0000000000000005, which cannot be used in floating point
operations
```

```
MVQ rg0, 5.0 ; The trailing 0 can be omitted to just have "5." if desired
; rg0 is 0x4014000000000000, or 5.0 in double floating point format,
; and can now be used in floating point operations
```

Floating Point Math

There are floating point equivalents of all the math operations in the base instruction set, as well as some additional mathematical operations exclusive to floating point values. Integers and floating point values *cannot* be mixed when performing floating point operations; any integer values must be converted to a float first, as explained in the following section.

Some examples of basic floating point math:

```
MVQ rg0, 5.7
FLPT_ADD rg0, 3.2
FLPT_WCN rg0
; "8.9" is printed to the console
```

```
MVQ rg1, -12.3
FLPT_MUL rg0, rg1
FLPT_WCN rg0
; "-109.47000000000001" is printed to the console (note the floating point
inaccuracy)
```

```
MVQ rg0, 1.0
FLPT_DIV rg0, 3.0
FLPT_WCN rg0
; "0.3333333333333333" is printed to the console
```

As can be seen with the second operation, floating point values cannot always represent decimal numbers with 100% accuracy, and may sometimes be off by a tiny fractional amount when converted to and from base 10.

Operations exclusive to floating point include trigonometric functions (i.e. Sine, Cosine, and Tangent and their inverses), single-instruction exponentiation, and logarithms. The trigonometric functions all operate on **radians** (a full circle is $2 * \text{PI}$ radians). You can convert degrees to radians by multiplying the degrees by 0.017453292519943295 ($\text{PI} / 180$), and you can convert radians to degrees by multiplying the radians by 57.295779513082323 ($180 / \text{PI}$).

Some examples:

```
MVQ rg0, 5.0
FLPT_POW rg0, 2.0
FLPT_WCN rg0
; "25" is printed to the console
```

```
FLPT_LOG rg0, 5.0
FLPT_WCN rg0
; "2" is printed to the console
```

```
FLPT_SIN rg0
FLPT_WCN rg0
; "0.9092974268256817" is printed to the console
```

Converting Between Integers and Floats

Because integers and floating point values are stored in separate formats and are not implicitly compatible, you must explicitly convert between them to have data in the format expected by each instruction being used.

There are two instructions for converting integers to floats: `FLPT_UTF` and `FLPT_STF`. These interpret the integer value of a register as either unsigned or signed respectively, and convert it to its closest equivalent in floating point format. Be aware that integers that require more than 53 bits to represent as an integer may not be converted to an identical value as a float, due to precision limitations with large numbers in the double-precision floating point format.

Examples of integer to float conversion:

```
MVQ rg0, 5
; rg0 is 0x0000000000000005, which cannot be used in floating point
operations
```

```
FLPT_UTF rg0 ; FLPT_STF would produce the same result in this case
; rg0 is 0x4014000000000000, or 5.0 in double floating point format,
; and can now be used in floating point operations
```

```
MVQ rg0, -8
```

```
; rg0 is 0xFFFFFFFFFFFFFFF8
FLPT_STF rg0
; rg0 is 0xC020000000000000 (-8.0)
```

```
MVQ rg0, -8
; rg0 is 0xFFFFFFFFFFFFFFF8
FLPT_UTF rg0
; rg0 is 0x43F0000000000000 (18446744073709552000.0)
```

There are four instructions for converting floats to integers: FLPT_FTS, FLPT_FCS, FLPT_FFS, and FLPT_FNS. These convert a floating point value to an integer which can be interpreted as signed, using one of four rounding methods respectively: truncation (rounding toward zero), ceiling (rounding to the greater adjacent integer), floor (rounding to the lesser adjacent integer), and nearest (rounding to the closest integer, with exact midpoints being rounded to the adjacent integer that is even).

Examples of float to integer conversion:

```
MVQ rg0, 5.7
FLPT_FTS rg0
SIGN_WCN rg0
; "5" is printed to console
```

```
MVQ rg0, 5.7
FLPT_FCS rg0
SIGN_WCN rg0
; "6" is printed to console
```

```
MVQ rg0, 5.7
FLPT_FFS rg0
SIGN_WCN rg0
; "5" is printed to console
```

```
MVQ rg0, 5.7
FLPT_FNS rg0
SIGN_WCN rg0
; "6" is printed to console
```

```
MVQ rg0, -5.7
FLPT_FTS rg0
SIGN_WCN rg0
; "-5" is printed to console
```

```
MVQ rg0, -5.7
FLPT_FCS rg0
SIGN_WCN rg0
; "-5" is printed to console
```

```
MVQ rg0, -5.7
```

```
FLPT_FFS rg0
SIGN_WCN rg0
; "-6" is printed to console
```

```
MVQ rg0, -5.7
FLPT_FNS rg0
SIGN_WCN rg0
; "-6" is printed to console
```

Some further examples of FLPT_FNS with midpoint and lower values:

```
MVQ rg0, 5.5
FLPT_FNS rg0
SIGN_WCN rg0
; "6" is printed to console
```

```
MVQ rg0, 6.5
FLPT_FNS rg0
SIGN_WCN rg0
; "6" is printed to console
```

```
MVQ rg0, 2.5
FLPT_FNS rg0
SIGN_WCN rg0
; "2" is printed to console
```

```
MVQ rg0, 3.5
FLPT_FNS rg0
SIGN_WCN rg0
; "4" is printed to console
```

```
MVQ rg0, 12.4
FLPT_FNS rg0
SIGN_WCN rg0
; "12" is printed to console
```

```
MVQ rg0, 3.2
FLPT_FNS rg0
SIGN_WCN rg0
; "3" is printed to console
```

Converting Between Floating Point Sizes

Floating point operations work solely on 64-bit floating point values, however there are other common sizes of floating point value which you may wish to convert between. There are instructions to convert to and from the half-precision (16-bit) and single-precision (32-bit) IEEE 754 floating point formats. To convert **to** a double-precision float, the FLPT_EXH and FLPT_EXS instructions are used to convert from half-precision and single-precision floats respectively. To convert **from** a double-precision float, the FLPT_SHH and FLPT_SHS

instructions are used to convert to half-precision and single-precision floats respectively. You cannot convert directly between half- and single-precision floats without converting to a double-precision float first.

Here are some examples of direct conversion:

```
MVQ rg0, 0x4248 ; 3.141 as a half-precision float
; rg0 cannot currently be used with floating point operations
FLPT_EXH rg0
; rg0 is now 0x4009200000000000 (3.140625) and can be used in floating point
operations
```

```
MVQ rg0, 0x40490FDB ; 3.1415927 as a single-precision float
; rg0 cannot currently be used with floating point operations
FLPT_EXS rg0
; rg0 is now 0x400921FB60000000 (3.14159274101257) and can be used in
floating point operations
```

```
MVQ rg0, 3.141592653589793
; rg0 is 0x400921FB54442D18
FLPT_SHH rg0
; rg0 is now 0x4248 (3.141 as a half-precision float)
```

```
MVQ rg0, 3.141592653589793
; rg0 is 0x400921FB54442D18
FLPT_SHS rg0
; rg0 is now 0x40490FDB (3.1415927 as a single-precision float)
```

And one for converting a single-precision to a half-precision float:

```
MVQ rg0, 0x40490FDB ; 3.1415927 as a single-precision float
FLPT_EXS rg0
; rg0 is now 0x400921FB60000000 (3.14159274101257)
FLPT_SHH rg0
; rg0 is now 0x4248 (3.141 as a half-precision float)
```

Jumping

Jumping is the processes of changing where the processor is currently executing in a program (represented with the rpo register). Jumps can be used to make loops, execute code if only a certain condition is met, or to reuse code, such as with subroutines. After a jump, the processor will continue to execute instructions from the new location, it will not automatically return to where it was before.

Jumps are usually made to labels, like so:

```
MVQ rg0, 0 ; Set rg0 to 0
:ADD_LOOP ; Create a label to the following instruction (ADD)
ADD rg0, 5 ; Add 5 to the current value of rg0
JMP :ADD_LOOP ; Go back to ADD_LOOP and continue executing from there
```

This program will set rg0 to 0, then infinitely keep adding 5 to the register by jumping back to the ADD_LOOP label. To only jump some of the time, for example to create a conditional loop, see the following section on branching.

Here is another example of a jump:

```
MVQ rg0, 0
ADD rg0, 5
JMP :SKIP
ADD rg0, 5 ; This won't be executed
ADD rg0, 5 ; This won't be executed
:SKIP
; rg0 is 5 here
```

rg0 only ends up being 5 at the end of this example, as jumping to the SKIP label prevented the two other ADD instructions from being reached.

Jumps can also be made to pointers, though you must be sure that the pointer will contain the address of a valid opcode before jumping there.

For example:

```
MVQ rg0, :&MY_CODE ; Move the literal address of MY_CODE to rg0
JMP *rg0 ; Jump to that address
MVQ rg0, 5 ; This won't be executed
:MY_CODE
MVQ rg0, 17
; rg0 will be 17, not 5
```

Comparing, Testing, and Branching

Branching is similar to jumping in that it changes where in the program execution is currently taking place, however, when branching, a condition is checked first before performing the jump. If the condition is not met, the program will continue execution as normal without jumping anywhere.

The conditional jump instructions are as follows:

Mnemonic	Meaning
JEQ	Jump if Equal
JNE	Jump if not Equal
JLT	Jump if Less Than
JLE	Jump if Less Than or Equal To
JGT	Jump if Greater Than
JGE	Jump if Greater Than or Equal To
JZO	Jump if Zero (=JEQ)
JNZ	Jump if not Zero (=JNE)
JCA	Jump if Carry (=JLT)

JNC	Jump if no Carry (=JGE)	
+-----+	+-----+	+-----+
SIGN_JLT	Jump if Less Than	
SIGN_JLE	Jump if Less Than or Equal To	
SIGN_JGT	Jump if Greater Than	
SIGN_JGE	Jump if Greater Than or Equal To	
+-----+	+-----+	+-----+
SIGN_JSI	Jump if Sign	
SIGN_JNS	Jump if not Sign	
SIGN_JOV	Jump if Overflow	
SIGN_JNO	Jump if not Overflow	
+-----+	+-----+	+-----+

The top section of instructions should be performed following a `CMP` operation on unsigned values, or a `FLPT_CMP` operation on floating point values. The instructions in the second section are aliases of four of the mnemonics in the top section (i.e. they share the same opcode) designed for use after mathematical operations or for bit testing (explained more in the relevant sections).

The bottom two sections are part of the signed extension set, with the higher of the two being designed for use following a `CMP` instruction on signed values, and the bottom section being for use specifically to branch based on the state of the sign or overflow flags.

Comparing Unsigned Numbers

To branch based on how two unsigned (always positive) numbers relate to each other, the `CMP` instruction can be utilised. It takes two operands (the first of which must be a register — it won't be modified), and compares them for use with a conditional jump instruction immediately afterwards.

For example:

```
RNG rg0 ; Set rg0 to a random number
CMP rg0, 1000 ; Compare rg0 to 1000
JGT :GREATER ; Jump straight to GREATER if rg0 is greater than 1000
ADD rg0, 1000 ; This will execute only if rg0 is less than or equal to 1000
:GREATER
SUB rg0, 1000 ; This will execute in either situation
```

Be aware that the `GREATER` label will still be reached if `rg0` is less than or equal to `1000` here, the `ADD` instruction will just be executed first.

To have the contents of the `GREATER` label execute **only** if `rg0` is greater than `1000`, include an unconditional jump like so:

```
RNG rg0 ; Set rg0 to a random number
CMP rg0, 1000 ; Compare rg0 to 1000
JGT :GREATER ; Jump straight to GREATER if rg0 is greater than 1000
ADD rg0, 1000 ; This will execute only if rg0 is less than or equal to 1000
JMP :END ; Jump straight to END to prevent GREATER section being executed
:GREATER
```



```
SUB rg0, 1000 ; This will execute only if rg0 is greater than 1000
:END
```

The CMP instruction works by subtracting the second operand from the first, but not storing the result anywhere. This operation still updates the status flags (rsf) however, and these can be used to check how the numbers relate. For example, if the second operand is greater than the first, you can guarantee that the operation will set the carry flag, as it would cause the result to be negative. This means to check if the first is greater than or equal to the second, you can simply check if the carry flag was unset. To check if the values were equal, the zero flag can be checked, as if the two operands of a subtraction are equal, the result will always be zero.

A full list of what each conditional jump instruction is checking for in terms of the status flags can be found in the full instruction reference.

Comparing Signed Numbers

The CMP instruction can also be used to compare signed (negative and positive) values, with its usage and behaviour remaining unchanged. After using the CMP instruction, however, you should use the signed version of the base conditional jump instructions, e.g. SIGN_JLT instead of JLT. The only exception to this is JEQ and JNE, which do not have signed versions, as they work with both signed and unsigned values.

For example:

```
MVQ rg0, 25
MVQ rg1, -6
CMP rg0, rg1
SIGN_JGT :GREATER
WCN 10 ; This will not execute, 25 is greater than -6
:GREATER
WCN 20 ; This will execute
; Only "20" is output to the console
```

And what would happen if the regular JGT instruction was used:

```
MVQ rg0, 25
MVQ rg1, -6
CMP rg0, rg1
JGT :GREATER
WCN 10 ; This will execute, even though 25 is greater than -6
:GREATER
WCN 20 ; This will execute
; "1020" is output to the console, -6 was interpreted instead as
18446744073709551610
```

Here the comparison doesn't work as expected because the conditional jump used (JGT) only works assuming the comparison was intended to be unsigned. The signed versions of these instructions (like SIGN_JGT) use the state of the sign, overflow, and zero status flags so that they work as expected when used after signed comparisons. A full list of what each

conditional jump instruction is checking for in terms of the status flags can be found in the full instruction reference.

Comparing Floating Point Numbers

To compare two floating point values, the FLPT_CMP instruction needs to be used instead of the CMP instruction. After using FLPT_CMP, the **unsigned** version of the desired conditional jump should be used, **even if one or both of the floating point values were negative**. There are no dedicated conditional jump instructions for floating point values.

For example:

```
MVQ rg0, 25.4
MVQ rg1, -6.3
FLPT_CMP rg0, rg1
JGT :GREATER
WCN 10 ; This will not execute, 25.4 is greater than -6.3
:GREATER
WCN 20 ; This will execute
; Only "20" is output to the console
```

FLPT_CMP updates the status flags with the unsigned conditional jumps in mind. If the first operand is less than the second, the carry flag is set. If they are equal, the zero flag is set. The overflow flag is always 0 after using FLPT_CMP.

Testing Bits

To test if a single bit of a number is set or not, the TST instruction can be used. Just like CMP, it takes two operands, the first of which being a register. The second should usually be a binary literal with only a single bit (the one to check) set as 1. It should then be followed by either JZO (jump if zero), or JNZ (jump if not zero). An example of where this may be used is checking if the third bit of rsf is set (the file end flag), as there isn't a built-in conditional jump that checks this flag.

This would be done like so:

```
:READ
RFC rg0 ; Read the next byte from the open file to rg0
TST rsf, 0b100 ; Check if the third bit is set
JZO :READ ; If it isn't set (i.e. it is equal to 0), jump back to READ
```

This program will keep looping until the third bit of rsf becomes 1. meaning that the end of the file has been reached.

Similarly to CMP, TST works by performing a bitwise and on the two operands, discarding the result, but still updating the status flags. A bitwise and will ensure that only the bit you want to check remains as 1, but only if it started as 1. If a bit is not one that you are checking, or it wasn't 1 to start with, it will end up as 0. If the resulting number isn't zero, leaving the zero flag unset, the bit must've been 1, and vice versa.

Checking the Carry, Overflow, Zero, and Sign Flags

The carry, overflow, zero, and sign flags also have specific jump operations that can check if they are currently set or unset.

For example:

```
MVQ rg0, 5
SUB rg0, 10
JCA :CARRY ; Jump to label if carry flag is set
WCN 10 ; This will not execute, as 5 SUB 10 will cause the carry flag to be
set
:CARRY
WCN 20
; Only "20" will be written to the console
```

JCA here is checking if the carry flag is set or not following the subtraction. The jump will only occur if the carry flag is 1 (set), otherwise, as with the other jump types, execution will continue as normal. JNC can be used to perform the inverse, jump only if the carry flag is unset.

The zero flag checks can also be used following a mathematical operation like so:

```
SUB rg0, 7 ; Subtract 7 from rg0
JNZ :NOT_ZERO ; Jump straight to NOT_ZERO if rg0 didn't become 0
ADD rg0, 1 ; Only execute this if rg0 became 0 because of the SUB operation
:NOT_ZERO
```

The ADD instruction here will only execute if the subtraction by 7 caused rg0 to become exactly equal to 0.

The SIGN_JOV, SIGN_JNO, SIGN_JSI, and SIGN_JNS instructions can be used to check if the overflow and sign flags are set and unset respectively in the same way:

```
SUB rg0, 7 ; Subtract 7 from rg0
SIGN_JNS :NOT_NEGATIVE ; Jump straight to NOT_NEGATIVE if rg0 didn't become
negative
SIGN_NEG rg0 ; Only execute this if rg0 became negative because of the SUB
operation
:NOT_NEGATIVE
; rg0 is now the absolute result
```

An equivalent of the first example, but for the overflow flag instead of the carry flag, as should be used for signed operations:

```
MVQ rg0, 5
SUB rg0, 10
JOV :OVERFLOW ; Jump to label if overflow flag is set
WCN 10 ; This will execute, as 5 SUB 10 will not cause the overflow flag to
be set
:OVERFLOW
```

```
WCN 20
; "1020" will be written to the console
```

Assembler Directives

Assembler directives follow the same format as standard instructions, however, instead of being assembled to an opcode for the processor to execute, they tell the assembler itself to do something to modify either the final binary file or the lines of the source file as its being assembled.

PAD - Byte Padding

The PAD directive tells the assembler to insert a certain number of 0 bytes wherever the directive is placed in the file. This is most often used just after a label definition to allocate a certain amount of guaranteed free and available memory to store data.

For example, consider the following program:

```
MVQ rg0, :&PADDING ; Store the address of the padding in rg0
JMP :PROGRAM      ; Jump to the next part of the program, skipping over the
padding
```

```
:PADDING
PAD 16 ; Insert 16 empty bytes
```

```
:PROGRAM
MVQ *rg0, 765 ; Set the first 8 bytes of the padding to represent 765
ADD rg0, 8   ; Add 8 to rg0, it now points to the next number
```

This program would assemble to the following bytes:

```
99 06 13 00 00 00 00 00 00 00 02 23 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 9F 06 FD 02 00 00 00 00 00 11 06 08 00 00 00
00 00 00
```

Which can be broken down to:

Address	Bytes
0x00	99 06 13 00 00 00 00 00 00 00 MVQ (reg, lit) rg0 :PADDING (address 0x13)
0x0A	02 23 00 00 00 00 00 00 00 00 JMP :PROGRAM (address 0x23)
0x13	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 PAD 16
0x23	9F 06 FD 02 00 00 00 00 00 00 MVQ (ptr, lit) *rg0 765 (0x2FD)

0x2D	11	06	08 00 00 00 00 00 00 00
	ADD	rg0	8

Note that usually, to reduce the number of jumps required, PADs would be placed after all program instructions. It was put in the middle of the program here for demonstration purposes.

DAT - Byte Insertion

The DAT directive inserts either a single byte, or a string of UTF-8 character bytes, into a program wherever the directive is located. As with PAD, it can be directly preceded by a label definition to point to the byte or string of bytes. If not being used with a string, DAT can only insert single bytes at once, meaning the maximum value is 255. It is also not suitable for inserting numbers to be used in 64-bit expecting operations (such as maths and bitwise), see the following section on the NUM directive for inserting 64-bit numbers.

An example of single byte insertion:

```
MVB rg0, :BYTE ; MVB must be used, as DAT will not insert a full 64-bit
number
; rg0 is now 54
HLT ; Stop the program executing into the DAT insertion (important!)
```

```
:BYTE
DAT 54 ; Insert a single 54 byte (0x36)
```

This program assembles into the following bytes:

```
82 06 0B 00 00 00 00 00 00 00 00 36
```

Which can be broken down to:

Address	Bytes
0x00	82 06 0B 00 00 00 00 00 00 00 MVB (reg, adr) rg0 :BYTE (address 0x0B)
0x0A	00 HLT
0x0B	36 DAT 54

To insert a string using DAT, the desired characters must be surrounded by double quote marks (") and be given as the sole operand to the directive. For example:

```
MVQ rg0, :&STRING ; Move literal address of string to rg0
:STRING_LOOP
MVB rg1, *rg0 ; Move contents of address stored in rg0 to rg1
CMP rg1, 0 ; Check if rg1 is 0
JEQ :END ; If it is, stop program
```

```
ICR rg0 ; Otherwise, increment source address by 1
WCC rg1 ; Write the read character to the console
JMP :STRING_LOOP ; Loop back to print next character
```

```
:END
HLT ; End execution to stop processor running into string data
```

```
:STRING
DAT "Hello!\0" ; Store a string of character bytes after program data.
; Note that the string ends with '\0' (a 0 or "null" byte)
```

This program will loop through the string, placing the byte value of each character in `rg0` and writing it to the console, until it reaches the 0 byte, when it will then stop to avoid looping infinitely. While not a strict requirement, terminating a string with a 0 byte like this should always be done to give an easy way of knowing when the end of a string has been reached. Placing a `DAT 0` directive on the line after the string insertion will also achieve this 0 termination, and will result in the exact same bytes being assembled, however using the `\0` escape sequence is more compact. Escape sequences are explained toward the end of the document along with a table listing all of the possible sequences.

The example program assembles down to the following bytes:

```
99 06 2E 00 00 00 00 00 00 00 83 07 06 75 07 00 00 00 00 00 00 00 04 2D 00
00 00 00 00 00 00 14 06 CC 07 02 0A 00 00 00 00 00 00 00 48 65 6C 6C 6F 21
00
```

Which can be broken down to:

Address	Bytes
0x00	99 06 2E 00 00 00 00 00 00 00 MVQ (reg, lit) rg0 :STRING (address 0x2E)
0x0A	83 07 06 MVB (reg, ptr) rg1 *rg0
0x0D	75 07 00 00 00 00 00 00 00 00 CMP (reg, lit) rg1 0
0x17	04 2D 00 00 00 00 00 00 00 JEQ (adr) :END (address 0x2D)
0x20	14 06 ICR (reg) rg0
0x22	CC 07 WCC (reg) rg1
0x24	02 0A 00 00 00 00 00 00 00 JMP (adr) :STRING_LOOP (address 0x0A)

0x2D	00	
	HLT	
0x2E	48 65 6C 6C 6F 21 00	
	DAT "Hello!\0"	

NUM - Number Insertion

The NUM directive is similar to DAT, except it always inserts 8 bytes exactly, so can be used to represent 64-bit numbers for use in instructions which always work on 64-bit values, like maths and bitwise operations. NUM cannot be used to insert strings, only single 64-bit numerical values (including unsigned, signed, and floating point).

An example:

```
MVQ rg0, 115 ; Initialise rg0 to 15
ADD rg0, :NUMBER ; Add the number stored in memory to rg0
; rg0 is now 100130
HLT ; End execution to stop processor running into number data
```

```
:NUMBER
NUM 100_015 ; Insert the number 100015 with 8 bytes
```

Which will produce the following bytes:

```
99 06 73 00 00 00 00 00 00 00 12 06 15 00 00 00 00 00 00 00 AF 86 01 00 00
00 00 00
```

Breaking down into:

Address	Bytes
0x00	99 06 73 00 00 00 00 00 00 00 MVQ (reg, lit) rg0 115 (0x73)
0x0A	12 06 15 00 00 00 00 00 00 00 ADD (reg, adr) rg0 :NUMBER (address 0x15)
0x14	00 HLT
0x15	AF 86 01 00 00 00 00 00 NUM 100_015 (0x186AF)

As with other operations in AssEmbly, NUM stores numbers in memory using little endian encoding. See the section on moving with memory for more info on how this encoding works.

MAC - Macro Definition

The MAC directive defines a **macro**, a piece of text that the assembler will replace with another on every line where the text is present. The directive takes the text to replace as the first operand, then the text for it to be replaced with as the second. Macros only take effect on lines after the one where they are defined, and they can be overwritten to change the replacement text by defining a new macro with the same name as a previous one. Unlike other instructions, the operands to the MAC directive don't have to be a standard valid format of operand, both will automatically be interpreted as literal text.

For example:

```
MVQ rg0, Number ; Results in an error
```

```
MAC Number, 345
MVQ rg0, Number
; rg0 is now 345
```

```
MAC Number, 678
MVQ rg1, Number
; rg1 is now 678
```

```
MAC Inst, ICR rg1
Inst
; rg1 is now 679
```

```
MAC Inst, ADD rg1, 6
Inst
; rg1 is now 685
```

The first line here results in an error, as a macro with a name of Number hasn't been defined yet (macros don't apply retroactively). MVQ rg0, Number gets replaced with MVQ rg0, 345, setting rg0 to 345. MVQ rg1, Number gets replaced with MVQ rg1, 678, as the Number macro was redefined on the line before, setting rg1 to 678. Inst gets replaced with ICR rg1, incrementing rg1 by 1, therefore setting it to 679 (macros can contain spaces and can be used to give another name to mnemonics, or even entire instructions, as seen in the last example).

Note that macro definitions ignore many standard syntax rules due to each operand being interpreted as literal text. Both operands can contain whitespace, and the second operand may contain commas. They are case sensitive, and macros with the same name but different capitalisations can exist simultaneously. Be aware that aside from a **single** space character separating the MAC mnemonic from its operands, leading and trailing whitespace in either of the operands will not be removed. Macros can also contain quotation marks ("), which will not be immediately parsed as a string within the macro. If the quotation marks are placed into a line as replacement text, they will be parsed normally as a part of the line.

IMP - File Importing

The IMP directive inserts the contents of another file wherever the directive is placed. It allows a program to be split across multiple files, as well as allowing code to be reused across multiple source files without having to copy the code into each file. The directive takes a single string operand (which must be enclosed in quotes), which can either be a full path (i.e. Drive:/Folder/Folder/file.asm) or a path relative to the directory of the source file being assembled (i.e. file.asm, Folder/file.asm, or ../Folder/file.asm).

For example, suppose you had two files in the same folder, one called program.asm, and one called numbers.asm.

Contents of program.asm:

```
MVQ rg0, :NUMBER_ONE
MVQ rg1, :NUMBER_TWO
HLT ; Prevent program executing into number data
```

IMP "numbers.asm"

Contents of numbers.asm:

```
:NUMBER_ONE
NUM 123
```

```
:NUMBER_TWO
NUM 456
```

When program.asm is assembled, the assembler will open and include the lines in numbers.asm once it reaches the IMP directive, resulting in the file looking like so:

```
MVQ rg0, :NUMBER_ONE
MVQ rg1, :NUMBER_TWO
HLT ; Prevent program executing into number data
```

```
IMP "numbers.asm"
:NUMBER_ONE
NUM 123
```

```
:NUMBER_TWO
NUM 456
```

Meaning that rg0 will finish with a value of 123, and rg1 will finish with a value of 456.

The IMP directive simply inserts the text contents of a file into the current file for assembly. This means that any label names in files being imported will be usable in the main file, though imposes the added restriction that label names must be unique across the main file and all its imported files.

Files given to the IMP directive **must** be AssEmbly source files, not already assembled binaries. It is recommended, though not a strict requirement, that import statements are

placed at the end of a file, as that will make it easier to ensure that the imported contents of a file aren't executed by mistake as part of the main program.

Care should be taken to ensure that a file does not end up depending on itself, even if it is through other files, as this will result in an infinite loop of imports (also known as a circular dependency). The AssEmby assembler will detect these and throw an error should one occur.

An example of a circular dependency:

```
file_one.asm:
```

```
IMP "file_two.asm"
```

```
file_two.asm:
```

```
IMP "file_three.asm"
```

```
file_three.asm:
```

```
IMP "file_one.asm"
```

Attempting to assemble any of these three files would result in the assembler throwing an error, as each file ends up depending on itself as it resolves its import.

ANALYZER - Toggling Assembler Warnings

The AssEmby assembler checks for common issues with your source code when you assemble it in order to alert you of potential issues and improvements that can be made. There may be some situations, however, where you want to suppress these issues from being detected. This can be done within the source code using the ANALYZER directive. The directive takes three operands: the severity of the warning (either error, warning, or suggestion); the numerical code for the warning (this is a 4-digit number printed alongside the message); and whether to enable (1), disable (0) or restore the warning to its state as it was at the beginning of assembly (r).

After using the directive, its effect remains active until assembly ends, or the same warning is toggled again with the directive further on in the code.

For example:

```
CMP rg0, 0 ; generates suggestion 0005
```

```
ANALYZER suggestion, 0005, 0
```

```
CMP rg0, 0 ; generates no suggestion
```

```
CMP rg0, 0 ; still generates no suggestion
```

```
ANALYZER suggestion, 0005, 1 ; 'r' would also work if the suggestion isn't disabled via a CLI argument
```

```
CMP rg0, 0 ; generates suggestion 0005 again
```

Be aware that some analyzers do not run until the end of the assembly process and so cannot be re-enabled without inadvertently causing the warning to re-appear. This can be overcome by placing the disabling ANALYZER directive at the end of the base file for any analyzers where this behaviour is an issue, or by simply not re-enabling the analyzer.

Console Input and Output

AssEmbly has native support for reading and writing from the console. There are four types of write that can be performed: 64-bit number in decimal; byte in decimal; byte in hexadecimal; and a raw byte (character). There is only a single type of read: a single raw byte. There is no native support for reading numbers in any base, nor is there support for reading or writing multiple numbers/bytes at once.

Writing can be done from registers, literals, labels, and pointers; reading must be done to a register. As with the move instructions, if a byte write instruction is used on a register or literal, only the lowest byte will be considered. If one is used on a label or a pointer, only a single byte of memory will be read, as opposed to the 8 bytes that are read when writing a 64-bit number.

An example of each type of write:

```
MVQ rg0, 0xFF0062
```

```
WCN rg0 ; Write a 64-bit number to the console in decimal  
; "16711778" (0xFF0062) is written to the console
```

```
WCC 10 ; Write a newline character
```

```
WCB rg0 ; Write a single byte to the console in decimal  
; "98" (0x62) is written to the console
```

```
WCC 10 ; Write a newline character
```

```
WCX rg0 ; Write a single byte to the console in hexadecimal  
; "62" is written to the console
```

```
WCC 10 ; Write a newline character
```

```
WCC rg0 ; Write a single byte to the console as a character  
; "b" (0x62) is written to the console
```

```
WCC 10 ; Write a newline character
```

Keep in mind that newlines are not automatically written after each write instruction, you will need to manually write the raw byte 10 (a newline character) to start writing on a new line. See the ASCII table at the end of the document for other common character codes.

An example of reading a byte:

`RCC rg0 ; Read a byte from the console and save the byte code to rg0`

When an `RCC` instruction is reached, the program will pause execution and wait for the user to input a character to the console. Once a character has been inputted, the corresponding byte value of the character will be copied to the given register. In this example, if the user types a lowercase “b”, `0x62` would be copied to `rg0`.

Be aware that if the user types a character that requires multiple bytes to represent in UTF-8, `RCC` will still only retrieve a single byte. You will have to use `RCC` multiple times to get all of the bytes needed to represent the character. `WCC` will also only write a single byte at a time, though as long as the console has UTF-8 support, simply writing each UTF-8 byte one after the other will result in the correct character being displayed.

Note that the user does not need to press enter after inputting a character, execution will resume immediately after a single character is typed. If you wish to wait for the user to press enter, compare the inputted character to `10` (the code for a newline character). The example program `input.ext.asm` contains a subroutine which does this. The user pressing the enter key will always give a single `10` byte, regardless of platform.

File Handling

As well as interfacing with the console, `AssEmbly` also has native support for handling files.

Opening and Closing

Files must be explicitly opened with the `OFL` instruction before they can read or written to, and only one file can be open at a time. You should close the currently open file with the `CFL` instruction when you have finished operating on it.

Filepaths given to `OFL` to be opened should be strings of UTF-8 character bytes in memory, ending with at least one `0` byte. An example static filepath definition is as follows:

```
:FILE_PATH
DAT "file.txt\0"
```

This would normally be placed after all program code and a `HLT` instruction to prevent it accidentally being executed as if it were part of the program. The file can be opened with the following line anywhere in the program:

```
OFL :FILE_PATH
...
CFL
```

You could also use a pointer if you wish:

```
MVQ rg0, :&FILE_PATH
OFL *rg0
...
CFL
```

CFL will close whatever file is currently open, so does not require any operands. If a file at the specified path does not exist when it is opened, an empty one will be created.

Reading and Writing

Reading and writing from files is almost identical to how it is done from the console. Registers, literals, labels, and pointers can all be written, and reading must be done to a register. When using byte writing instructions, only the lower byte of registers and literals is considered, and only a single byte of memory is read for labels and pointers. An open file can be both read from and written to while it is open, though changes written to the file will not be reflected in either the current AssEmbly program or other applications until the file is closed. If a file already has data in it when it is written to, the new data will start overwriting from the first byte in the file. Any remaining data that does not get overwritten will remain unchanged, and the size of the file will not change unless more bytes are written than were originally in the file. To clear a file before writing it, use the DFL instruction to delete the file beforehand.

An example of writing to a file:

```
MVQ rg0, 0xFF0062
OFL :FILE_PATH ; Open file with the 0-terminated string at :FILE_PATH
```

```
WFN rg0 ; Write a 64-bit number to the file in decimal
; "16711778" (0xFF0062) is appended to the file
```

```
WFC 10 ; Write a newline character
```

```
WFB rg0 ; Write a single byte to the file in decimal
; "98" (0x62) is appended to the file
```

```
WFC 10 ; Write a newline character
```

```
WFX rg0 ; Write a single byte to the file in hexadecimal
; "62" is appended to the file
```

```
WFC 10 ; Write a newline character
```

```
WFC rg0 ; Write a single byte to the file as a character
; "b" (0x62) is appended to the file
```

```
WFC 10 ; Write a newline character
```

```
CFL ; Close the file, saving newly written contents
```

```
HLT ; Prevent executing into string data
```

```
:FILE_PATH
DAT "file.txt\0"
```

Executing this program will create a file called file.txt with the following contents:

16711778
98
62
b

File contents can be read with the RFC instruction, taking a single register as an operand. The next unread byte from the file will be stored in the specified register. Text files are not treated specially, RFC will simply retrieve the characters 1 byte at a time as they are encoded in the file. If the end of the file has been reached after reading, the file end flag will be set to 1. The only way to reset the current reading position in a file is to close and reopen the file.

To read all bytes until the end of a file, you will need to continually read single bytes from the file, testing the file end flag after every read, stopping as soon as it becomes set. The example program `read_file.asm` has an example of this, as well as this example from the bit testing section:

```
:READ  
RFC rg0 ; Read the next byte from the open file to rg0  
TST rsf, 0b100 ; Check if the third bit is set  
JZO :READ ; If it isn't set (i.e. it is equal to 0), jump back to READ
```

Other Operations

As well as reading and writing, there are also instructions for checking whether a file exists (FEX), getting the size of a file (FSZ), and deleting a file (DFL). They all take a path in the same way OFL does. DFL has no effect other than deleting the file. FEX and FSZ first take a register operand to store their result in, then the path to the file as the second operand. FEX stores 1 in the register if the file exists, 0 if not. FSZ stores the total size of the file in bytes.

The Stack

The stack is a section of memory most often used in conjunction with subroutines, explained in the subsequent section. It starts at the very end of available memory, and dynamically grows backwards as more items are added (**pushed**) to it. The stack contains exclusively 64-bit (8 byte) values. Registers, literals, labels, and pointers can all be given as operands to the push (PSH) instruction.

Once items have been pushed to the stack, they can be removed (**popped**), starting with the most recently pushed item. As with most other instructions with a destination, items from the stack must be popped into registers with the POP instruction. Once an item is removed from the stack, the effective size of the stack shrinks back down, and the popped item will no longer be considered part of the stack until and unless it is pushed again.

The rso register contains the address of the first byte of the top item in the stack. Its value will get **lower** as items are **pushed**, and **greater** as items are **popped**. More info on the rso register's behaviour can be found in the registers section.

Take this visual example, assuming memory is 8192 bytes in size (making 8191 the maximum address):

```
; rso = 8192
; | Addresses |      8168..8175      |      8176..8183      |      8184..8191      ||
; |   Value   |  ?????????????????? |  ?????????????????? |  ?????????????????? ||
```

PSH 0xDEADBEEF ; Push 0xDEADBEEF (3735928559) to the stack

```
; rso = 8184
; | Addresses |      8168..8175      |      8176..8183      ||      8184..8191      |
; |   Value   |  ?????????????????? |  ?????????????????? || 00000000EFBEADDE |
```

PSH 0xCAFE0BA ; Push 0xCAFE0BA (3405689018) to the stack

```
; rso = 8176
; | Addresses |      8168..8175      ||      8176..8183      |      8184..8191      |
; |   Value   |  ?????????????????? || 00000000BAB0FECA | 00000000EFBEADDE |
```

PSH 0xD00D2BAD ; Push 0xD00D2BAD (3490524077) to the stack

```
; rso = 8168
; | Addresses ||      8168..8175      |      8176..8183      |      8184..8191      |
; |   Value  || 00000000AD2B0DD0 | 00000000BAB0FECA | 00000000EFBEADDE |
```

POP rg0 ; Pop the most recent non-popped item from the stack into rg0

```
; rso = 8176
; | Addresses |      8168..8175      ||      8176..8183      |      8184..8191      |
; |   Value   |  ?????????????????? || 00000000BAB0FECA | 00000000EFBEADDE |
; rg0 = 0xD00D2BAD
```

POP rg0 ; Pop the most recent non-popped item from the stack into rg0

```
; rso = 8184
; | Addresses |      8168..8175      |      8176..8183      ||      8184..8191      |
; |   Value   |  ?????????????????? |  ?????????????????? || 00000000EFBEADDE |
; rg0 = 0xCAFE0BA
```

Using the Stack to Preserve Registers

A common use of the stack is to store the value of a register, use the register for a purpose that differs from its original one, then restore the register to the stored value. This is particularly useful in sections of reusable code (such as subroutines) where you cannot guarantee whether a register will be in use or not.

An example of this is as follows:

```
MVQ rg0, 45
ADD rg0, 20
```

```

; rg0 is 65

PSH rg0 ; Push the current value of rg0 to the stack
MVQ rg0, 200
MUL rg0, 10
; rg0 is 2000

POP rg0 ; Pop the old rg0 back into rg0
; rg0 is back to 65

```

Subroutines

A subroutine is a section of a program that can be specially jumped to (**called**) from multiple different points in a program. They differ from a standard jump in that the position in the program that a subroutine is called from is stored automatically, so can be **returned** to at any point with ease. This makes reusing the same section of code across different parts of a program, or even across different programs, much easier.

Subroutines are defined with a label as with any other form of jump destination — to call one, use the CAL instruction with either the label or a pointer to that label. Once you are within a subroutine, you can return to the calling location with the RET instruction, no operands required.

An example of a simple subroutine:

```

MVQ rg0, 5
CAL :ADD_TO_RG0
; rg0 is now 15

MVQ rg1, :&ADD_TO_RG0
MVQ rg0, 46
CAL *rg1
; rg0 is now 56

HLT

:ADD_TO_RG0
ADD rg0, 10
RET

```

Specifically, RET will cause rpo to be updated to the address storing the opcode directly after the CAL instruction that was used to call the subroutine. Unless they are halting the program, subroutines should always exit with a RET instruction and nothing else.

Fast Calling

The CAL instruction can also take an optional second operand: a value to pass to the subroutine. This is called **fast calling** or **fast passing**; the passed value gets stored in rfp and can be any one of a register, literal, label, or pointer. More info on the behaviour of the

register itself and how it should be used can be found in its part of the registers section. Parameters are always 64-bit values, so when passing a label or a register, 8 bytes of memory will always be read.

An example of subroutines utilising fast calling:

```
:SUBROUTINE_ONE
ADD rfp, 1
MVQ rg0, rfp
RET
```

```
:SUBROUTINE_TWO
ADD rfp, 2
MVQ rg0, rfp
RET
```

```
CAL :SUBROUTINE_ONE, 4 ; This will implicitly set rfp to 4
; rg0 is now 5
CAL :SUBROUTINE_TWO, 6 ; This will implicitly set rfp to 6
; rg0 is now 8
```

Return Values

The RET instruction can also take an optional operand to return a value. Return values can be registers, literals, labels, or pointers, and are stored in rrv. As with fast pass parameters, return values are always 64-bits/8 bytes. The exact behaviour and usage of the register can be found in its part of the registers section.

Here is the above example for fast calling adapted to use return values:

```
:SUBROUTINE_ONE
ADD rfp, 1
RET rfp ; Return, setting rrv to the value of rfp
```

```
:SUBROUTINE_TWO
ADD rfp, 2
RET rfp ; Return, setting rrv to the value of rfp
```

```
CAL :SUBROUTINE_ONE, 4
; rrv is now 5
CAL :SUBROUTINE_TWO, 6
; rrv is now 8
```

Subroutines and the Stack

In order to store the address to return to when using subroutines, the stack is utilised. Every time the CAL instruction is used, the address of the next opcode, and the current value of rsb, are pushed to the stack in that order. rsb and rso will then be updated to the new address of the top of the stack (the address where rsb was pushed to). rsb will continue to point here (the **base**) until another subroutine is called or the subroutine is

returned from. `rso` will continue to update as normal as items are popped to and pushed from the stack, always pointing to the top of it. The area from the current **base** (`rsb`) to the top of the stack (`rso`) is called the current **stack frame**. Multiple stack frames can be stacked on top of each other if a subroutine is called from another subroutine.

When returning from a subroutine, the opposite is performed. `rsb`, and `rpo` are popped off the top of the stack, thereby continuing execution as it was before the subroutine was called. All values apart from these two must be popped off the stack before using the `RET` instruction (you can ensure this by moving the value of `rsb` into `rso`). After returning `rso` will point to the same address as when the function was called.

If you utilise registers in a subroutine, you should use the stack to ensure that the value of each modified register is returned to its initial value before returning from the subroutine. See the above section on using the stack to preserve registers for info on how to do this.

Passing Multiple Parameters

The `CAL` instruction can only take a single data parameter, however, there may be situations where multiple values need to be passed to a subroutine; it is best to use the stack in situations such as these. Before calling the subroutine, push any values you want to act as parameters to the subroutine, to the stack. Once the subroutine has been called, you can use `rsb` to calculate the address that each parameter will be stored at. To access the first parameter (the last one pushed before calling), you need to account for the two automatically pushed values first. These, along with every other value in the stack, are all 8 bytes long, so adding 16 ($8 * 2$) to `rsb` will get you the address of this parameter (you should do this in another register, `rsb` should be left unmodified). To access any subsequent parameters, simply add another 8 on top of this.

For example:

```
PSH 4 ; Parameter D
PSH 3 ; Parameter C
PSH 2 ; Parameter B
CAL :SUBROUTINE, 1 ; Parameter A (rfp)
; rrv is now 10
```

```
:SUBROUTINE
PSH rg0 ; Preserve the value of rg0
```

```
MVQ rg0, rsb
ADD rg0, 16 ; Parameter B
ADD rfp, *rg0
; rfp is now 3
ADD rg0, 8 ; Parameter C
ADD rfp, *rg0
; rfp is now 6
ADD rg0, 8 ; Parameter D
ADD rfp, *rg0
; rfp is now 10
```

```
POP rg0 ; Restore rg0 to its original value
RET rfp
```

Text Encoding

All text in AssEmbly (input from/output to the console; strings inserted by DAT; strings given to OFL, DFL, FEX, etc.) is encoded in UTF-8. This means that all characters that are a part of the ASCII character set only take up a single byte, though some characters may take as many as 4 bytes to store fully.

Be aware that when working with characters that require multiple bytes, instructions like RCC, RFC, WCC, and WFC still only work on single bytes at a time. As long as you read/write all of the UTF-8 bytes in the correct order, they should be stored and displayed correctly.

Text bytes read from files **will not** be automatically converted to UTF-8 if the file was saved with another encoding.

Escape Sequences

There are some sequences of characters that have special meanings when found inside a string or character literal. Each of these begins with a backslash (\) character and are used to insert characters that couldn't be included normally. Every supported sequence is as follows:

Escape sequence	Character name	Notes
\"	Double quote	Used to insert a double quote into a string without causing the string to end. Not required in single character literals.
\'	Single quote	Used to insert a single quote into a single character literal without causing the literal to end. Not required in string literals.
\\	Backslash	For a string to contain a backslash, you must escape it so it isn't treated as the start of an escape sequence.
\0	Null	ASCII 0x00. Should be used to terminate every string.
\a	Alert	ASCII 0x07.
\b	Backspace	ASCII 0x08.
\f	Form feed	ASCII 0x0C.
\n	Newline	ASCII 0x0A. Will cause the string to move onto a new console/file line when printed. Should be preceded by \r on Windows.
\r	Carriage return	ASCII 0x0D.
\t	Horizontal tab	ASCII 0x09.

Escape sequence	Character name	Notes
	l tab	
\v	Vertical tab	ASCII 0x0B.
\u.. ..	Unicode codepoint (16-bit)	Inserts the unicode character with a codepoint represented by 4 hexadecimal digits in the range 0x0000 to 0xFFFF.
\U..	Unicode codepoint (32-bit)	Inserts the unicode character with a codepoint represented by 8 hexadecimal digits in the range 0x00000000 to 0x0010FFFF, excluding 0x0000d800 to 0x0000dfff.

Instruction Data Type Acceptance

The following is a table of which types of numeric data can be given to each instruction and have them function as expected. AssEmbyl **does not** keep track of data types, it is your responsibility to do so. If you use the wrong instruction for the type of data you have, it is unlikely you will receive an error — you will most likely simply get an unexpected answer, as the processor is interpreting the data as a valid, but different, numeric value in a different format.

If an instruction supports signed integers but not unsigned integers, the instruction *will* still accept positive values, but those positive values must be below the signed limit (9,223,372,036,854,775,807), or they will be erroneously interpreted as negative.

- 0 = Instruction accepts the data type
- x = Instruction does not accept the data type
- (...) = Instruction accepts the data type, but see the numbered footnote below the table for additional information to keep in mind

Instructions that don't take any data or are otherwise not applicable have been omitted.

Instruction	Unsigned Integer	Signed Integer	Floating Point
ADD	0	0	X
ICR	0	0	X
SUB	0	0	X
DCR	0	0	X
MUL	0	0	X
DIV	0	X	X
DVR	0	X	X
REM	0	X	X

Instruction	Unsigned Integer	Signed Integer	Floating Point
SHL	0	0	X
SHR	0	(1)	X
AND	0	(2)	X
ORR	0	(2)	X
XOR	0	(2)	X
NOT	0	(2)	X
TST	0	(2)	X
CMP	0	X	X
MVB	0	(3)	X
MVW	0	(3)	X
MVD	0	(3)	X
MVQ	0	0	0
PSH	0	0	0
CAL	0	0	0
RET	0	0	0
WCN	0	X	X
WCB	0	X	X
WCX	0	X	X
WCC	0	X	X
WFN	0	X	X
WFB	0	X	X
WFX	0	X	X
WFC	0	X	X
SIGN_DIV	X	0	X
SIGN_DVR	X	0	X
SIGN_REM	X	0	X
SIGN_SHR	X	0	X
SIGN_MVB	X	0	X
SIGN_MVW	X	0	X
SIGN_MVD	X	0	X
SIGN_WCN	X	0	X
SIGN_WCB	X	0	X
SIGN_WFN	X	0	X
SIGN_WFB	X	0	X
SIGN_EXB	X	0	X

Instruction	Unsigned Integer	Signed Integer	Floating Point
SIGN_EXW	X	0	X
SIGN_EXD	X	0	X
SIGN_NEG	X	0	X
FLPT_ADD	X	X	0
FLPT_SUB	X	X	0
FLPT_MUL	X	X	0
FLPT_DIV	X	X	0
FLPT_DVR	X	X	0
FLPT_REM	X	X	0
FLPT_SIN	X	X	0
FLPT_ASN	X	X	0
FLPT_COS	X	X	0
FLPT_ACS	X	X	0
FLPT_TAN	X	X	0
FLPT_ATN	X	X	0
FLPT_PTN	X	X	0
FLPT_POW	X	X	0
FLPT_LOG	X	X	0
FLPT_WCN	X	X	0
FLPT_WFN	X	X	0
FLPT_EXH	X	X	0
FLPT_EXS	X	X	0
FLPT_SHS	X	X	0
FLPT_SHH	X	X	0
FLPT_NEG	X	X	0
FLPT_UTF	0	X	X
FLPT_STF	X	0	X
FLPT_FTS	X	X	0
FLPT_FCS	X	X	0
FLPT_FFS	X	X	0
FLPT_FNS	X	X	0
FLPT_CMP	X	X	0
EXTD_BSW	(4)	(4)	(4)
HEAP_ALC	0	X	X
HEAP_TRY	0	X	X

Instruction	Unsigned Integer	Signed Integer	Floating Point
HEAP_REA	O	X	X
HEAP_TRE	O	X	X

1. Signed integers *can* still be used with SHR, though it will perform a logical shift, not an arithmetic one, which may or may not be what you desire. See the section on Arithmetic Right Shifting for the difference.
2. Bitwise operations on signed integers will treat the sign bit like any other, there is no special logic involving it.
3. Using smaller-than-64-bit move instructions on signed integers if the target is a label or pointer will work as expected, truncating the upper bits. If the target is a register, however, you may wish to use the signed versions to automatically extend the smaller integer to a signed 64-bit one so it is correctly interpreted by other instructions.
4. Reversing the byte order of a register can work on any data type, however, registers **must** be in little endian order *after* reversing to have their value correctly interpreted by other instructions (this does not apply to instructions where the format of the register's value is unimportant, such as with MVQ).

Status Flag Behaviour

- 0 = Instruction always unsets flag
- 1 = Instruction always sets flag
- (...) = Instruction sets flag if the given condition is satisfied, otherwise it unsets it
- [...] = Instruction sets flag if the given condition is satisfied, otherwise it maintains its current value
- {...} = Instruction unsets flag if the given condition is satisfied, otherwise it maintains its current value
- X = Instruction does not affect flag
- STD = Instruction uses standard behaviour for flag according to result, unaffected by operands. They are as follows:
 - For zero flag, set if the result is equal to 0, otherwise unset (for floating point operations, -0 is considered equal to 0 and will set the zero flag)
 - For sign flag, set if the most significant bit of the result is set, otherwise unset

Instruction	Zero Flag		File End	Sign Flag	
	Carry	Overflow		Carry	Overflow
HLT	X	X	X	X	X
NOP	X	X	X	X	X
JMP	X	X	X	X	X
JEQ / JZ0	X	X	X	X	X
JNE /	X	X	X	X	X

Instruction	Zero	Carry	File End	Sign	Overflow
JNZ					
JLT / JCA	X	X	X	X	X
JLE	X	X	X	X	X
JGT	X	X	X	X	X
JGE / JNC	X	X	X	X	X
ADD	S T D	(Result is unrepresentable as unsigned)	X	S T D	(Result is unrepresentable as signed)
ICR	S T D	(Result is unrepresentable as unsigned)	X	S T D	(Result is unrepresentable as signed)
SUB	S T D	(Result is unrepresentable as unsigned)	X	S T D	(Result is unrepresentable as signed)
DCR	S T D	(Result is unrepresentable as unsigned)	X	S T D	(Result is unrepresentable as signed)
MUL	S T D	(Result is unrepresentable as both unsigned and signed)	X	S T D	0
DIV	S T D	0	X	S T D	0
DVR	S T D	0	X	S T D	0
REM	S T D	0	X	S T D	0
SHL	S T D	(Any 1 bit was shifted past MSB)	X	S T D	0
SHR	S T D	(Any 1 bit was shifted past LSB)	X	S T D	0
AND	S	0	X	S	0

Instruction	Zero	Carry	File End	Sign	Overflow
	T			T	
	D			D	
ORR	S	0	X	S	0
	T			T	
	D			D	
XOR	S	0	X	S	0
	T			T	
	D			D	
NOT	S	0	X	S	0
	T			T	
	D			D	
RNG	S	0	X	S	0
	T			T	
	D			D	
TST	S	X	X	S	X
	T			T	
	D			D	
CMP	S	(Result is unrepresentable as unsigned)	X	S	(Result is unrepresentable as signed)
	T			T	
	D			D	
MVB	X	X	X	X	X
MVW	X	X	X	X	X
MVD	X	X	X	X	X
MVQ	X	X	X	X	X
PSH	X	X	X	X	X
POP	X	X	X	X	X
CAL	X	X	X	X	X
RET	X	X	X	X	X
WCN	X	X	X	X	X
WCB	X	X	X	X	X
WCX	X	X	X	X	X
WCC	X	X	X	X	X
WFN	X	X	X	X	X
WFB	X	X	X	X	X
WFX	X	X	X	X	X
WFC	X	X	X	X	X

Instru ction	Z e r o	Carry	File End	S i g n	Overflow
OFL	X	X	(File is empty)	X	X
CFL	X	X	X	X	X
DFL	X	X	X	X	X
FEX	X	X	X	X	X
FSZ	X	X	X	X	X
RCC	X	X	X	X	X
RFC	X	X	[No more unread bytes in file]	X	X
SIGN_ JLT	X	X	X	X	X
SIGN_ JLE	X	X	X	X	X
SIGN_ JGT	X	X	X	X	X
SIGN_ JGE	X	X	X	X	X
SIGN_ JSI	X	X	X	X	X
SIGN_ JNS	X	X	X	X	X
SIGN_ JOV	X	X	X	X	X
SIGN_ JNO	X	X	X	X	X
SIGN_ DIV	S T D	0	X	S T D	0
SIGN_ DVR	S T D	0	X	S T D	0
SIGN_ REM	S T D	0	X	S T D	0
SIGN_ SHR	S T D	(Any bit not equal to the sign bit was shifted past LSB)	X	S T D	0
SIGN_ 	X	X	X	X	X

Instru ction	Z e r o		Carry	File End	S i g n	Overflow
MVB						
SIGN_ MVW	X	X		X	X	X
SIGN_ MVD	X	X		X	X	X
SIGN_ WCN	X	X		X	X	X
SIGN_ WCB	X	X		X	X	X
SIGN_ WFB	X	X		X	X	X
SIGN_ EXB	S T D	0		X	S T D	0
SIGN_ EXW	S T D	0		X	S T D	0
SIGN_ EXD	S T D	0		X	S T D	0
SIGN_ NEG	S T D	0		X	S T D	0
FLPT_ ADD	S T D	(Result is less than the initial value)		X	S T D	0
FLPT_ SUB	S T D	(Result is greater than the initial value)		X	S T D	0
FLPT_ MUL	S T D	(Result is less than the initial value)		X	S T D	0
FLPT_ DIV	S T D	0		X	S T D	0
FLPT_ DVR	S T	0		X	S T	0

Instruction	Zero	Carry	File End	Sign	Overflow
	D			D	
FLPT_ REM	S T D	0	X	S T D	0
FLPT_ SIN	S T D	0	X	S T D	0
FLPT_ ASN	S T D	0	X	S T D	0
FLPT_ COS	S T D	0	X	S T D	0
FLPT_ ACS	S T D	0	X	S T D	0
FLPT_ TAN	S T D	0	X	S T D	0
FLPT_ ATN	S T D	0	X	S T D	0
FLPT_ PTN	S T D	0	X	S T D	0
FLPT_ POW	S T D	(Result is less than the initial value)	X	S T D	0
FLPT_ LOG	S T D	(Result is greater than the initial value)	X	S T D	0
FLPT_ WCN	X	X	X	X	X
FLPT_ WFN	X	X	X	X	X
FLPT_ EXH	S T D	0	X	S T D	0

Instru ction	Z e r o	Carry	File End	S i g n	Overflow
FLPT_ EXS	S T D	0	X	S T D	0
FLPT_ SHS	S T D	0	X	S T D	0
FLPT_ SHH	S T D	0	X	S T D	0
FLPT_ NEG	S T D	0	X	S T D	0
FLPT_ UTF	S T D	0	X	S T D	0
FLPT_ STF	S T D	0	X	S T D	0
FLPT_ FTS	S T D	0	X	S T D	0
FLPT_ FCS	S T D	0	X	S T D	0
FLPT_ FFS	S T D	0	X	S T D	0
FLPT_ FNS	S T D	0	X	S T D	0
FLPT_ CMP	S T D	(Value of first operand is less than second)	X	S T D	0
EXTD_ BSW	X	X	X	X	X
ASM_ LDA	X	X	X	X	X
ASM_ LDA	X	X	X	X	X

Instruction	Zero	Carry	File End	Sign	Overflow
LDF					
ASMX_ CLA	X	X	X	X	X
ASMX_ CLF	X	X	X	X	X
ASMX_ AEX	X	X	X	X	X
ASMX_ FEX	X	X	X	X	X
ASMX_ CAL	X	X	X	X	X
HEAP_ ALC	X	X	X	X	X
HEAP_ TRY	X	X	X	X	X
HEAP_ REA	X	X	X	X	X
HEAP_ TRE	X	X	X	X	X
HEAP_ FRE	X	X	X	X	X

Full Instruction Reference

Base Instruction Set

Extension set number `0x00`, opcodes start with `0xFF`, `0x00`. Contains the core features of the architecture, remaining mostly unchanged by updates.

Note that for the base instruction set (number `0x00`) *only*, the leading `0xFF`, `0x00` to specify the extension set can be omitted, as the processor will automatically treat opcodes not starting with `0xFF` as base instruction set opcodes.

Mnemonic	Full Name	Operands	Function	Instruction Code
Control				
HLT	Halt	-	Stops the processor from executing the program	<code>0x00</code>
NOP	No Operation	-	Do nothing	<code>0x01</code>

Mnemonic	Full Name	Operands	Function	Instruction Code
Jumping				
JMP	Jump	Address	Jump unconditionally to an address in a label	0x02
JMP	Jump	Pointer	Jump unconditionally to an address in a register	0x03
JEQ / JZ0	Jump if Equal / Jump if Zero	Address	Jump to an address in a label only if the zero status flag is set	0x04
JEQ / JZ0	Jump if Equal / Jump if Zero	Pointer	Jump to an address in a register only if the zero status flag is set	0x05
JNE / JNZ	Jump if not Equal / Jump if not Zero	Address	Jump to an address in a label only if the zero status flag is unset	0x06
JNE / JNZ	Jump if not Equal / Jump if not Zero	Pointer	Jump to an address in a register only if the zero status flag is unset	0x07
JLT / JCA	Jump if Less Than / Jump if Carry	Address	Jump to an address in a label only if the carry status flag is set	0x08
JLT / JCA	Jump if Less	Pointer	Jump to an	0x09

Mnemonic	Full Name	Operands	Function	Instruction Code
	Than / Jump if Carry		address in a register only if the carry status flag is set	
JLE	Jump if Less Than or Equal To	Address	Jump to an address in a label only if either the carry or zero flags are set	0x0A
JLE	Jump if Less Than or Equal To	Pointer	Jump to an address in a register only if either the carry or zero flags are set	0x0B
JGT	Jump if Greater Than	Address	Jump to an address in a label only if both the carry and zero flags are unset	0x0C
JGT	Jump if Greater Than	Pointer	Jump to an address in a register only if both the carry and zero flags are unset	0x0D
JGE / JNC	Jump if Greater Than or Equal To / Jump if no Carry	Address	Jump to an address in a label only if the carry status flag is unset	0x0E
JGE / JNC	Jump if Greater Than or Equal To / Jump if no Carry	Pointer	Jump to an address in a register only if the carry status flag is unset	0x0F

Mnemonic	Full Name	Operands	Function	Instruction Code
Math				
ADD	Add	Register, Register	Add the contents of one register to another	0x10
ADD	Add	Register, Literal	Add a literal value to the contents of a register	0x11
ADD	Add	Register, Address	Add the contents of memory at an address in a label to a register	0x12
ADD	Add	Register, Pointer	Add the contents of memory at an address in a register to a register	0x13
ICR	Increment	Register	Increment the contents of a register by 1	0x14
SUB	Subtract	Register, Register	Subtract the contents of one register from another	0x20
SUB	Subtract	Register, Literal	Subtract a literal value from the contents of a register	0x21
SUB	Subtract	Register, Address	Subtract the contents of memory at an address in a label from a register	0x22
SUB	Subtract	Register, Pointer	Subtract the contents of	0x23

Mnemonic	Full Name	Operands	Function	Instruction Code
			memory at an address in a register from a register	
DCR	Decrement	Register	Decrement the contents of a register by 1	0x24
MUL	Multiply	Register, Register	Multiply the contents of one register by another	0x30
MUL	Multiply	Register, Literal	Multiply the contents of a register by a literal value	0x31
MUL	Multiply	Register, Address	Multiply a register by the contents of memory at an address in a label	0x32
MUL	Multiply	Register, Pointer	Multiply a register by the contents of memory at an address in a register	0x33
DIV	Integer Divide	Register, Register	Divide the contents of one register by another, discarding the remainder	0x40
DIV	Integer Divide	Register, Literal	Divide the contents of a register by a literal value, discarding the	0x41

Mnemonic	Full Name	Operands	Function	Instruction Code
			remainder	
DIV	Integer Divide	Register, Address	Divide a register by the contents of memory at an address in a label, discarding the remainder	0x42
DIV	Integer Divide	Register, Pointer	Divide a register by the contents of memory at an address in a register, discarding the remainder	0x43
DVR	Divide With Remainder	Register, Register, Register	Divide the contents of one register by another, storing the remainder	0x44
DVR	Divide With Remainder	Register, Register, Literal	Divide the contents of a register by a literal value, storing the remainder	0x45
DVR	Divide With Remainder	Register, Register, Address	Divide a register by the contents of memory at an address in a label, storing the remainder	0x46
DVR	Divide With Remainder	Register, Register, Pointer	Divide a register by the contents of memory at	0x47

Mnemonic	Full Name	Operands	Function	Instruction Code
			an address in a register, storing the remainder	
REM	Remainder Only	Register, Register	Divide the contents of one register by another, storing only the remainder	0x48
REM	Remainder Only	Register, Literal	Divide the contents of a register by a literal value, storing only the remainder	0x49
REM	Remainder Only	Register, Address	Divide a register by the contents of memory at an address in a label, storing only the remainder	0x4A
REM	Remainder Only	Register, Pointer	Divide a register by the contents of memory at an address in a register, storing only the remainder	0x4B
SHL	Shift Left	Register, Register	Shift the bits of one register left by another register	0x50
SHL	Shift Left	Register, Literal	Shift the bits of a register	0x51

Mnemonic	Full Name	Operands	Function	Instruction Code
			left by a literal value	
SHL	Shift Left	Register, Address	Shift the bits of a register left by the contents of memory at an address in a label	0x52
SHL	Shift Left	Register, Pointer	Shift the bits of a register left by the contents of memory at an address in a register	0x53
SHR	Shift Right	Register, Register	Shift the bits of one register right by another register	0x54
SHR	Shift Right	Register, Literal	Shift the bits of a register right by a literal value	0x55
SHR	Shift Right	Register, Address	Shift the bits of a register right by the contents of memory at an address in a label	0x56
SHR	Shift Right	Register, Pointer	Shift the bits of a register right by the contents of memory at an address in a register	0x57
Bitwise				
AND	Bitwise And	Register, Register	Bitwise and one register	0x60

Mnemonic	Full Name	Operands	Function	Instruction Code
AND	Bitwise And	Register, Literal	by another Bitwise and a register by a literal value	0x61
AND	Bitwise And	Register, Address	Bitwise and a register by the contents of memory at an address in a label	0x62
AND	Bitwise And	Register, Pointer	Bitwise and a register by the contents of memory at an address in a register	0x63
ORR	Bitwise Or	Register, Register	Bitwise or one register by another	0x64
ORR	Bitwise Or	Register, Literal	Bitwise or a register by a literal value	0x65
ORR	Bitwise Or	Register, Address	Bitwise or a register by the contents of memory at an address in a label	0x66
ORR	Bitwise Or	Register, Pointer	Bitwise or a register by the contents of memory at an address in a register	0x67
XOR	Bitwise Exclusive Or	Register, Register	Bitwise exclusive or one register by another	0x68
XOR	Bitwise Exclusive Or	Register, Literal	Bitwise exclusive or a register by a literal value	0x69

Mnemonic	Full Name	Operands	Function	Instruction Code
XOR	Bitwise Exclusive Or	Register, Address	Bitwise exclusive or a register by the contents of memory at an address in a label	0x6A
XOR	Bitwise Exclusive Or	Register, Pointer	Bitwise exclusive or a register by the contents of memory at an address in a register	0x6B
NOT	Bitwise Not	Register	Invert each bit of a register	0x6C
RNG	Random Number Generator	Register	Randomise each bit of a register	0x6D
Comparison				
TST	Test	Register, Register	Bitwise and two registers, discarding the result whilst still updating status flags	0x70
TST	Test	Register, Literal	Bitwise and a register and a literal value, discarding the result whilst still updating status flags	0x71
TST	Test	Register, Address	Bitwise and a register and the contents of memory at an address in	0x72

Mnemonic	Full Name	Operands	Function	Instruction Code
			a label, discarding the result whilst still updating status flags	
TST	Test	Register, Pointer	Bitwise and a register and the contents of memory at an address in a register, discarding the result whilst still updating status flags	0x73
CMP	Compare	Register, Register	Subtract a register from another, discarding the result whilst still updating status flags	0x74
CMP	Compare	Register, Literal	Subtract a literal value from a register, discarding the result whilst still updating status flags	0x75
CMP	Compare	Register, Address	Subtract the contents of memory at an address in a label from a register, discarding the result whilst still	0x76

Mnemonic	Full Name	Operands	Function	Instruction Code
			updating status flags	
CMP	Compare	Register, Pointer	Subtract the contents of memory at an address in a register from a register, discarding the result whilst still updating status flags	0x77
Data Moving				
MVB	Move Byte	Register, Register	Move the lower 8-bits of one register to another	0x80
MVB	Move Byte	Register, Literal	Move the lower 8-bits of a literal value to a register	0x81
MVB	Move Byte	Register, Address	Move 8-bits of the contents of memory starting at an address in a label to a register	0x82
MVB	Move Byte	Register, Pointer	Move 8-bits of the contents of memory starting at an address in a register to a register	0x83
MVB	Move Byte	Address,	Move the lower 8-bits	0x84

Mnemonic	Full Name	Operands	Function	Instruction Code
		Register	of a register to the contents of memory at an address in a label	
MVB	Move Byte	Address, Literal	Move the lower 8-bits of a literal to the contents of memory at an address in a label	0x85
MVB	Move Byte	Pointer, Register	Move the lower 8-bits of a register to the contents of memory at an address in a register	0x86
MVB	Move Byte	Pointer, Literal	Move the lower 8-bits of a literal to the contents of memory at an address in a register	0x87
MVW	Move Word	Register, Register	Move the lower 16-bits (2 bytes) of one register to another	0x88
MVW	Move Word	Register, Literal	Move the lower 16-bits (2 bytes) of a literal value to a register	0x89
MVW	Move Word	Register, Address	Move 16-bits (2 bytes) of the contents of memory starting at an	0x8A

Mnemonic	Full Name	Operands	Function	Instruction Code
			address in a label to a register	
MVW	Move Word	Register, Pointer	Move 16-bits (2 bytes) of the contents of memory starting at an address in a register to a register	0x8B
MVW	Move Word	Address, Register	Move the lower 16-bits (2 bytes) of a register to the contents of memory at an address in a label	0x8C
MVW	Move Word	Address, Literal	Move the lower 16-bits (2 bytes) of a literal to the contents of memory at an address in a label	0x8D
MVW	Move Word	Pointer, Register	Move the lower 16-bits (2 bytes) of a register to the contents of memory at an address in a register	0x8E
MVW	Move Word	Pointer, Literal	Move the lower 16-bits (2 bytes) of a literal to the contents of memory at an address in a register	0x8F

Mnemonic	Full Name	Operands	Function	Instruction Code
MVD	Move Double Word	Register, Register	Move the lower 32-bits (4 bytes) of one register to another	0x90
MVD	Move Double Word	Register, Literal	Move the lower 32-bits (4 bytes) of a literal value to a register	0x91
MVD	Move Double Word	Register, Address	Move 32-bits (4 bytes) of the contents of memory starting at an address in a label to a register	0x92
MVD	Move Double Word	Register, Pointer	Move 32-bits (4 bytes) of the contents of memory starting at an address in a register to a register	0x93
MVD	Move Double Word	Address, Register	Move the lower 32-bits (4 bytes) of a register to the contents of memory at an address in a label	0x94
MVD	Move Double Word	Address, Literal	Move the lower 32-bits (4 bytes) of a literal to the contents of memory at an address in a label	0x95
MVD	Move Double	Pointer,	Move the	0x96

Mnemonic	Full Name	Operands	Function	Instruction Code
	Word	Register	lower 32-bits (4 bytes) of a register to the contents of memory at an address in a register	
MVD	Move Double Word	Pointer, Literal	Move the lower 32-bits (4 bytes) of a literal to the contents of memory at an address in a register	0x97
MVQ	Move Quad Word	Register, Register	Move all 64-bits (8 bytes) of one register to another	0x98
MVQ	Move Quad Word	Register, Literal	Move all 64-bits (8 bytes) of a literal value to a register	0x99
MVQ	Move Quad Word	Register, Address	Move 64-bits (8 bytes) of the contents of memory starting at an address in a label to a register	0x9A
MVQ	Move Quad Word	Register, Pointer	Move 64-bits (8 bytes) of the contents of memory starting at an address in a register to a register	0x9B
MVQ	Move Quad Word	Address, Register	Move all 64-bits (8 bytes)	0x9C

Mnemonic	Full Name	Operands	Function	Instruction Code
			of a register to the contents of memory at an address in a label	
MVQ	Move Quad Word	Address, Literal	Move all 64-bits (8 bytes) of a literal to the contents of memory at an address in a label	0x9D
MVQ	Move Quad Word	Pointer, Register	Move all 64-bits (8 bytes) of a register to the contents of memory at an address in a register	0x9E
MVQ	Move Quad Word	Pointer, Literal	Move all 64-bits (8 bytes) of a literal to the contents of memory at an address in a register	0x9F
Stack				
PSH	Push to Stack	Register	Insert the value in a register to the top of the stack	0xA0
PSH	Push to Stack	Literal	Insert a literal value to the top of the stack	0xA1
PSH	Push to Stack	Address	Insert the contents of memory at an address in a label to the	0xA2

Mnemonic	Full Name	Operands	Function	Instruction Code
PSH	Push to Stack	Pointer	top of the stack Insert the contents of memory at an address in a register to the top of the stack	0xA3
POP	Pop from Stack	Register	Remove the value from the top of the stack and store it in a register	0xA4
Subroutines				
CAL	Call Subroutine	Address	Call the subroutine at an address in a label, pushing <i>rpo</i> and <i>rsb</i> to the stack	0xB0
CAL	Call Subroutine	Pointer	Call the subroutine at an address in a register, pushing <i>rpo</i> and <i>rsb</i> to the stack	0xB1
CAL	Call Subroutine	Address, Register	Call the subroutine at an address in a label, moving the value in a register to <i>rfp</i>	0xB2
CAL	Call Subroutine	Address, Literal	Call the subroutine at an address in a label, moving a	0xB3

Mnemonic	Full Name	Operands	Function	Instruction Code
			literal value to rfp	
CAL	Call Subroutine	Address, Address	Call the subroutine at an address in a label, moving the contents of memory at an address in a label to rfp	0xB4
CAL	Call Subroutine	Address, Pointer	Call the subroutine at an address in a label, moving the contents of memory at an address in a register to rfp	0xB5
CAL	Call Subroutine	Pointer, Register	Call the subroutine at an address in a register, moving the value in a register to rfp	0xB6
CAL	Call Subroutine	Pointer, Literal	Call the subroutine at an address in a register, moving a literal value to rfp	0xB7
CAL	Call Subroutine	Pointer, Address	Call the subroutine at an address in a register, moving the contents of memory at	0xB8

Mnemonic	Full Name	Operands	Function	Instruction Code
CAL	Call Subroutine	Pointer, Pointer	an address in a label to rfp Call the subroutine at an address in a register, moving the contents of memory at an address in a register to rfp	0xB9
RET	Return from Subroutine	-	Pop the previous states of rsb and rpo off the stack	0xBA
RET	Return from Subroutine	Register	Pop the previous states of rsb and rpo off the stack, moving the value in a register to rrv	0xBB
RET	Return from Subroutine	Literal	Pop the previous states of rsb and rpo off the stack, moving a literal value to rrv	0xBC
RET	Return from Subroutine	Address	Pop the previous states off the stack, moving the contents of memory at an address in a label to rrv	0xBD
RET	Return from	Pointer	Pop the	0xBE

Mnemonic	Full Name	Operands	Function	Instruction Code
	Subroutine		previous states off the stack, moving the contents of memory at an address in a register to rrv	
Console Writing				
WCN	Write Number to Console	Register	Write a register value as a decimal number to the console	0xC0
WCN	Write Number to Console	Literal	Write a literal value as a decimal number to the console	0xC1
WCN	Write Number to Console	Address	Write 64-bits (4 bytes) of memory starting at the address in a label as a decimal number to the console	0xC2
WCN	Write Number to Console	Pointer	Write 64-bits (4 bytes) of memory starting at the address in a register as a decimal number to the console	0xC3
WCB	Write Numeric Byte to Console	Register	Write the lower 8-bits of a register value as a	0xC4

Mnemonic	Full Name	Operands	Function	Instruction Code
			decimal number to the console	
WCB	Write Numeric Byte to Console	Literal	Write the lower 8-bits of a literal value as a decimal number to the console	0xC5
WCB	Write Numeric Byte to Console	Address	Write contents of memory at the address in a label as a decimal number to the console	0xC6
WCB	Write Numeric Byte to Console	Pointer	Write contents of memory at the address in a register as a decimal number to the console	0xC7
WCX	Write Hexadecimal to Console	Register	Write the lower 8-bits of a register value as a hexadecimal number to the console	0xC8
WCX	Write Hexadecimal to Console	Literal	Write the lower 8-bits of a literal value as a hexadecimal number to the console	0xC9
WCX	Write Hexadecimal to Console	Address	Write contents of memory at	0xCA

Mnemonic	Full Name	Operands	Function	Instruction Code
			the address in a label as a hexadecimal number to the console	
WCX	Write Hexadecimal to Console	Pointer	Write contents of memory at the address in a register as a hexadecimal number to the console	0xCB
WCC	Write Raw Byte to Console	Register	Write the lower 8-bits of a register value as a raw byte to the console	0xCC
WCC	Write Raw Byte to Console	Literal	Write the lower 8-bits of a literal value as a raw byte to the console	0xCD
WCC	Write Raw Byte to Console	Address	Write contents of memory at the address in a label as a raw byte to the console	0xCE
WCC	Write Raw Byte to Console	Pointer	Write contents of memory at the address in a register as a raw byte to the console	0xCF

File Writing

Mnemonic	Full Name	Operands	Function	Instruction Code
WFN	Write Number to File	Register	Write a register value as a decimal number to the opened file	0xD0
WFN	Write Number to File	Literal	Write a literal value as a decimal number to the opened file	0xD1
WFN	Write Number to File	Address	Write 64-bits (4 bytes) of memory starting at the address in a label as a decimal number to the opened file	0xD2
WFN	Write Number to File	Pointer	Write 64-bits (4 bytes) of memory starting at the address in a register as a decimal number to the opened file	0xD3
WFB	Write Numeric Byte to File	Register	Write the lower 8-bits of a register value as a decimal number to the opened file	0xD4
WFB	Write Numeric Byte to File	Literal	Write the lower 8-bits	0xD5

Mnemonic	Full Name	Operands	Function	Instruction Code
			of a literal value as a decimal number to the opened file	
WFB	Write Numeric Byte to File	Address	Write contents of memory at the address in a label as a decimal number to the opened file	0xD6
WFB	Write Numeric Byte to File	Pointer	Write contents of memory at the address in a register as a decimal number to the opened file	0xD7
WFX	Write Hexadecimal to File	Register	Write the lower 8-bits of a register value as a hexadecimal number to the opened file	0xD8
WFX	Write Hexadecimal to File	Literal	Write the lower 8-bits of a literal value as a hexadecimal number to the opened file	0xD9
WFX	Write Hexadecimal to File	Address	Write contents of memory at	0xDA

Mnemonic	Full Name	Operands	Function	Instruction Code
			the address in a label as a hexadecimal number to the opened file	
WFX	Write Hexadecimal to File	Pointer	Write contents of memory at the address in a register as a hexadecimal number to the opened file	0xDB
WFC	Write Raw Byte to File	Register	Write the lower 8-bits of a register value as a raw byte to the opened file	0xDC
WFC	Write Raw Byte to File	Literal	Write the lower 8-bits of a literal value as a raw byte to the opened file	0xDD
WFC	Write Raw Byte to File	Address	Write contents of memory at the address in a label as a raw byte to the opened file	0xDE
WFC	Write Raw Byte to File	Pointer	Write contents of memory at the address in a register	0xDF

Mnemonic	Full Name	Operands	Function	Instruction Code
			as a raw byte to the opened file	
File Operations				
OFL	Open File	Address	Open the file at the path specified by a 0x00 terminated string in memory starting at an address in a label	0xE0
OFL	Open File	Pointer	Open the file at the path specified by a 0x00 terminated string in memory starting at an address in a register	0xE1
CFL	Close File	-	Close the currently open file	0xE2
DFL	Delete File	Address	Delete the file at the path specified by a 0x00 terminated string in memory starting at an address in a label	0xE3
DFL	Delete File	Pointer	Delete the file at the path specified by a	0xE4

Mnemonic	Full Name	Operands	Function	Instruction Code
			0x00 terminated string in memory starting at an address in a register	
FEX	File Exists	Register, Address	Store 1 in a register if the filepath specified in memory starting at an address in a label exists, else 0	0xE5
FEX	File Exists	Register, Pointer	Store 1 in a register if the filepath specified in memory starting at an address in a register exists, else 0	0xE6
FSZ	Get File Size	Register, Address	In a register, store the byte size of the file at the path specified in memory starting at an address in a label	0xE7
FSZ	Get File Size	Register, Pointer	In a register, store the byte size of the file at the path specified in memory starting at an	0xE8

Mnemonic	Full Name	Operands	Function	Instruction Code
			address in a register	
Reading				
RCC	Read Raw Byte from Console	Register	Read a raw byte from the console, storing it in a register	0xF0
RFC	Read Raw Byte from File	Register	Read the next byte from the currently open file, storing it in a register	0xF1

Signed Extension Set

Extension set number 0x01, opcodes start with 0xFF, 0x01. Contains instructions required for interacting with two's complement signed/negative values.

Mnemonic	Full Name	Operands	Function	Instruction Code
Signed Conditional Jumps				
SIGN_JLT	Jump if Less Than	Address	Jump to an address in a label only if the sign and overflow status flags are different	0x00
SIGN_JLT	Jump if Less Than	Pointer	Jump to an address in a register only if the sign and overflow status flags are different	0x01
SIGN_JLE	Jump if Less Than or Equal To	Address	Jump to an address in a label only if the sign and overflow	0x02

Mnemonic	Full Name	Operands	Function	Instruction Code
			status flags are different or the zero status flag is set	
SIGN_JLE	Jump if Less Than or Equal To	Pointer	Jump to an address in a register only if the sign and overflow status flags are different or the zero status flag is set	0x03
SIGN_JGT	Jump if Greater Than	Address	Jump to an address in a label only if the sign and overflow status flags are the same and the zero status flag is unset	0x04
SIGN_JGT	Jump if Greater Than	Pointer	Jump to an address in a register only if the sign and overflow status flags are the same and the zero status flag is unset	0x05
SIGN_JGE	Jump if Greater Than or Equal To	Address	Jump to an address in a label only if the sign and overflow status flags are the same	0x06
SIGN_JGE	Jump if Greater	Pointer	Jump to an	0x07

Mnemonic	Full Name	Operands	Function	Instruction Code
	Than or Equal To		address in a register only if the sign and overflow status flags are the same	
SIGN_JSI	Jump if Signed	Address	Jump to an address in a label only if the sign status flag is set	0x08
SIGN_JSI	Jump if Signed	Pointer	Jump to an address in a register only if the sign status flag is set	0x09
SIGN_JNS	Jump if not Sign	Address	Jump to an address in a label only if the sign status flag is unset	0x0A
SIGN_JNS	Jump if not Sign	Pointer	Jump to an address in a register only if the sign status flag is unset	0x0B
SIGN_JOV	Jump if Overflow	Address	Jump to an address in a label only if the overflow status flag is set	0x0C
SIGN_JOV	Jump if Overflow	Pointer	Jump to an address in a register only if the overflow status flag is set	0x0D

Mnemonic	Full Name	Operands	Function	Instruction Code
SIGN_JNO	Jump if not Overflow	Address	Jump to an address in a label only if the overflow status flag is unset	0x0E
SIGN_JNO	Jump if not Overflow	Pointer	Jump to an address in a register only if the overflow status flag is unset	0x0F
Math				
SIGN_DIV	Integer Divide	Register, Register	Divide the contents of one register by another, discarding the remainder	0x10
SIGN_DIV	Integer Divide	Register, Literal	Divide the contents of a register by a literal value, discarding the remainder	0x11
SIGN_DIV	Integer Divide	Register, Address	Divide a register by the contents of memory at an address in a label, discarding the remainder	0x12
SIGN_DIV	Integer Divide	Register, Pointer	Divide a register by the contents of memory at an address in a register,	0x13

Mnemonic	Full Name	Operands	Function	Instruction Code
			discarding the remainder	
SIGN_DVR	Divide With Remainder	Register, Register, Register	Divide the contents of one register by another, storing the remainder	0x14
SIGN_DVR	Divide With Remainder	Register, Register, Literal	Divide the contents of a register by a literal value, storing the remainder	0x15
SIGN_DVR	Divide With Remainder	Register, Register, Address	Divide a register by the contents of memory at an address in a label, storing the remainder	0x16
SIGN_DVR	Divide With Remainder	Register, Register, Pointer	Divide a register by the contents of memory at an address in a register, storing the remainder	0x17
SIGN_REM	Remainder Only	Register, Register	Divide the contents of one register by another, storing only the remainder	0x18
SIGN_REM	Remainder Only	Register, Literal	Divide the contents of a register by a literal value, storing only	0x19

Mnemonic	Full Name	Operands	Function	Instruction Code
SIGN_REM	Remainder Only	Register, Address	the remainder Divide a register by the contents of memory at an address in a label, storing only the remainder	0x1A
SIGN_REM	Remainder Only	Register, Pointer	Divide a register by the contents of memory at an address in a register, storing only the remainder	0x1B
SIGN_SHR	Arithmetic Shift Right	Register, Register	Shift the bits of one register right by another register, preserving the sign of the original value	0x20
SIGN_SHR	Arithmetic Shift Right	Register, Literal	Shift the bits of a register right by a literal value, preserving the sign of the original value	0x21
SIGN_SHR	Arithmetic Shift Right	Register, Address	Shift the bits of a register right by the contents of memory at an address in	0x22

Mnemonic	Full Name	Operands	Function	Instruction Code
			a label, preserving the sign of the original value	
SIGN_SHR	Arithmetic Shift Right	Register, Pointer	Shift the bits of a register right by the contents of memory at an address in a register, preserving the sign of the original value	0x23
Sign- Extending Data Moves				
SIGN_MVB	Move Byte, Extend to Quad Word	Register, Register	Move the lower 8-bits of one register to another, extending the resulting value to a signed 64-bit value	0x30
SIGN_MVB	Move Byte, Extend to Quad Word	Register, Literal	Move the lower 8-bits of a literal value to a register, extending the resulting value to a signed 64-bit value	0x31
SIGN_MVB	Move Byte, Extend to Quad Word	Register, Address	Move 8-bits of the contents of memory	0x32

Mnemonic	Full Name	Operands	Function	Instruction Code
			starting at an address in a label to a register, extending the resulting value to a signed 64-bit value	
SIGN_MVB	Move Byte, Extend to Quad Word	Register, Pointer	Move 8-bits of the contents of memory starting at an address in a register to a register, extending the resulting value to a signed 64-bit value	0x33
SIGN_MVW	Move Word, Extend to Quad Word	Register, Register	Move the lower 16-bits (2 bytes) of one register to another, extending the resulting value to a signed 64-bit value	0x34
SIGN_MVW	Move Word, Extend to Quad Word	Register, Literal	Move the lower 16-bits (2 bytes) of a literal value to a register, extending the resulting value to a signed 64-bit value	0x35
SIGN_MVW	Move Word,	Register,	Move 16-bits	0x36

Mnemonic	Full Name	Operands	Function	Instruction Code
	Extend to Quad Word	Address	(2 bytes) of the contents of memory starting at an address in a label to a register, extending the resulting value to a signed 64-bit value	
SIGN_MVW	Move Word, Extend to Quad Word	Register, Pointer	Move 16-bits (2 bytes) of the contents of memory starting at an address in a register to a register, extending the resulting value to a signed 64-bit value	0x37
SIGN_MVD	Move Double Word, Extend to Quad Word	Register, Register	Move the lower 32-bits (4 bytes) of one register to another, extending the resulting value to a signed 64-bit value	0x40
SIGN_MVD	Move Double Word, Extend to Quad Word	Register, Literal	Move the lower 32-bits (4 bytes) of a literal value to a register, extending the resulting value to a signed 64-bit	0x41

Mnemonic	Full Name	Operands	Function	Instruction Code
			value	
SIGN_MVD	Move Double Word, Extend to Quad Word	Register, Address	Move 32-bits (4 bytes) of the contents of memory starting at an address in a label to a register, extending the resulting value to a signed 64-bit value	0x42
SIGN_MVD	Move Double Word, Extend to Quad Word	Register, Pointer	Move 32-bits (4 bytes) of the contents of memory starting at an address in a register to a register, extending the resulting value to a signed 64-bit value	0x43
Console Writing				
SIGN_WCN	Write Number to Console	Register	Write a register value as a signed decimal number to the console	0x50
SIGN_WCN	Write Number to Console	Literal	Write a literal value as a signed decimal number to the console	0x51
SIGN_WCN	Write Number	Address	Write 64-bits	0x52

Mnemonic	Full Name to Console	Operands	Function (4 bytes) of memory starting at the address in a label as a signed decimal number to the console	Instruction Code
SIGN_WCN	Write Number to Console	Pointer	Write 64-bits (4 bytes) of memory starting at the address in a register as a signed decimal number to the console	0x53
SIGN_WCB	Write Numeric Byte to Console	Register	Write the lower 8-bits of a register value as a signed decimal number to the console	0x54
SIGN_WCB	Write Numeric Byte to Console	Literal	Write the lower 8-bits of a literal value as a signed decimal number to the console	0x55
SIGN_WCB	Write Numeric Byte to Console	Address	Write contents of memory at the address in a label as a signed decimal number to	0x56

Mnemonic	Full Name	Operands	Function	Instruction Code
SIGN_WCB	Write Numeric Byte to Console	Pointer	the console Write contents of memory at the address in a register as a signed decimal number to the console	0x57
File Writing				
SIGN_WFN	Write Number to File	Register	Write a register value as a signed decimal number to the opened file	0x60
SIGN_WFN	Write Number to File	Literal	Write a literal value as a signed decimal number to the opened file	0x61
SIGN_WFN	Write Number to File	Address	Write 64-bits (4 bytes) of memory starting at the address in a label as a signed decimal number to the opened file	0x62
SIGN_WFN	Write Number to File	Pointer	Write 64-bits (4 bytes) of memory starting at the address in a register	0x63

Mnemonic	Full Name	Operands	Function	Instruction Code
			as a signed decimal number to the opened file	
SIGN_WFB	Write Numeric Byte to File	Register	Write the lower 8-bits of a register value as a signed decimal number to the opened file	0x64
SIGN_WFB	Write Numeric Byte to File	Literal	Write the lower 8-bits of a literal value as a signed decimal number to the opened file	0x65
SIGN_WFB	Write Numeric Byte to File	Address	Write contents of memory at the address in a label as a signed decimal number to the opened file	0x66
SIGN_WFB	Write Numeric Byte to File	Pointer	Write contents of memory at the address in a register as a signed decimal number to the opened file	0x67

Mnemonic	Full Name	Operands	Function	Instruction Code
Sign Extension				
SIGN_EXB	Extend Signed Byte to Signed Quad Word	Register	Convert the signed value in the lower 8-bits of a register to its equivalent representation as a signed 64-bit number	0x70
SIGN_EXW	Extend Signed Word to Signed Quad Word	Register	Convert the signed value in the lower 16-bits of a register to its equivalent representation as a signed 64-bit number	0x71
SIGN_EXD	Extend Signed Double Word to Signed Quad Word	Register	Convert the signed value in the lower 32-bits of a register to its equivalent representation as a signed 64-bit number	0x72
Negation				
SIGN_NEG	Two's Complement Negation	Register	Replace the value in a register with its two's complement, thereby flipping the sign of the value.	0x80

Floating Point Extension Set

Extension set number 0x02, opcodes start with 0xFF, 0x02. Contains instructions required for interacting with IEEE 754 floating point values.

Mnemonic	Full Name	Operands	Function	Instruction Code
Math				
FLPT_ADD	Add	Register, Register	Add the contents of one register to another	0x00
FLPT_ADD	Add	Register, Literal	Add a literal value to the contents of a register	0x01
FLPT_ADD	Add	Register, Address	Add the contents of memory at an address in a label to a register	0x02
FLPT_ADD	Add	Register, Pointer	Add the contents of memory at an address in a register to a register	0x03
FLPT_SUB	Subtract	Register, Register	Subtract the contents of one register from another	0x10
FLPT_SUB	Subtract	Register, Literal	Subtract a literal value from the contents of a register	0x11
FLPT_SUB	Subtract	Register, Address	Subtract the contents of memory at an address in a label from a register	0x12
FLPT_SUB	Subtract	Register, Pointer	Subtract the contents of	0x13

Mnemonic	Full Name	Operands	Function	Instruction Code
			memory at an address in a register from a register	
FLPT_MUL	Multiply	Register, Register	Multiply the contents of one register by another	0x20
FLPT_MUL	Multiply	Register, Literal	Multiply the contents of a register by a literal value	0x21
FLPT_MUL	Multiply	Register, Address	Multiply a register by the contents of memory at an address in a label	0x22
FLPT_MUL	Multiply	Register, Pointer	Multiply a register by the contents of memory at an address in a register	0x23
FLPT_DIV	Integer Divide	Register, Register	Divide the contents of one register by another, discarding the remainder	0x30
FLPT_DIV	Integer Divide	Register, Literal	Divide the contents of a register by a literal value, discarding the remainder	0x31
FLPT_DIV	Integer Divide	Register, Address	Divide a register by the contents of memory at	0x32

Mnemonic	Full Name	Operands	Function	Instruction Code
FLPT_DIV	Integer Divide	Register, Pointer	an address in a label, discarding the remainder Divide a register by the contents of memory at an address in a register, discarding the remainder	0x33
FLPT_DVR	Divide With Remainder	Register, Register, Register	Divide the contents of one register by another, storing the remainder	0x34
FLPT_DVR	Divide With Remainder	Register, Register, Literal	Divide the contents of a register by a literal value, storing the remainder	0x35
FLPT_DVR	Divide With Remainder	Register, Register, Address	Divide a register by the contents of memory at an address in a label, storing the remainder	0x36
FLPT_DVR	Divide With Remainder	Register, Register, Pointer	Divide a register by the contents of memory at an address in a register, storing the remainder	0x37
FLPT_REM	Remainder	Register,	Divide the	0x38

Mnemonic	Full Name	Operands	Function	Instruction Code
	Only	Register	contents of one register by another, storing only the remainder	
FLPT_REM	Remainder Only	Register, Literal	Divide the contents of a register by a literal value, storing only the remainder	0x39
FLPT_REM	Remainder Only	Register, Address	Divide a register by the contents of memory at an address in a label, storing only the remainder	0x3A
FLPT_REM	Remainder Only	Register, Pointer	Divide a register by the contents of memory at an address in a register, storing only the remainder	0x3B
FLPT_SIN	Sine	Register	Calculate the sine of the value in a register in radians	0x40
FLPT_ASN	Inverse Sine	Register	Calculate the inverse sine of the value in a register in radians	0x41
FLPT_COS	Cosine	Register	Calculate the cosine of the	0x42

Mnemonic	Full Name	Operands	Function	Instruction Code
FLPT_ACS	Inverse Cosine	Register	value in a register in radians Calculate the inverse cosine of the value in a register in radians	0x43
FLPT_TAN	Tangent	Register	Calculate the tangent of the value in a register in radians	0x44
FLPT_ATN	Inverse Tangent	Register	Calculate the inverse tangent of the value in a register in radians	0x45
FLPT_PTN	2 Argument Inverse Tangent	Register, Register	Calculate the 2 argument inverse tangent between 2 registers in the order y, x	0x46
FLPT_PTN	2 Argument Inverse Tangent	Register, Literal	Calculate the 2 argument inverse tangent between a register and a literal in the order y, x	0x47
FLPT_PTN	2 Argument Inverse Tangent	Register, Address	Calculate the 2 argument inverse tangent between a register and the contents of memory at	0x48

Mnemonic	Full Name	Operands	Function	Instruction Code
FLPT_PTN	2 Argument Inverse Tangent	Register, Pointer	<p>an address in a label in the order y, x</p> <p>Calculate the 2 argument inverse tangent between a register and the contents of memory at an address in a register in the order y, x</p>	0x49
FLPT_POW	Exponentiation	Register, Register	Calculate the value of a register raised to the power of another register	0x50
FLPT_POW	Exponentiation	Register, Literal	Calculate the value of a register raised to the power of a literal	0x51
FLPT_POW	Exponentiation	Register, Address	Calculate the value of a register raised to the power of the contents of memory at an address in a label	0x52
FLPT_POW	Exponentiation	Register, Pointer	Calculate the value of a register raised to the power of the contents of memory at	0x53

Mnemonic	Full Name	Operands	Function	Instruction Code
			an address in a register	
FLPT_LOG	Logarithm	Register, Register	Calculate the logarithm of a register with the base from another register	0x60
FLPT_LOG	Logarithm	Register, Literal	Calculate the logarithm of a register with the base from a literal	0x61
FLPT_LOG	Logarithm	Register, Address	Calculate the logarithm of a register with the base from the contents of memory at an address in a label	0x62
FLPT_LOG	Logarithm	Register, Pointer	Calculate the logarithm of a register with the base from the contents of memory at an address in a register	0x63
Console Writing				
FLPT_WCN	Write Number to Console	Register	Write a register value as a signed decimal number to the console	0x70
FLPT_WCN	Write Number to Console	Literal	Write a literal value as a signed	0x71

Mnemonic	Full Name	Operands	Function	Instruction Code
			decimal number to the console	
FLPT_WCN	Write Number to Console	Address	Write 64-bits (4 bytes) of memory starting at the address in a label as a signed decimal number to the console	0x72
FLPT_WCN	Write Number to Console	Pointer	Write 64-bits (4 bytes) of memory starting at the address in a register as a signed decimal number to the console	0x73
File Writing				
FLPT_WFN	Write Number to File	Register	Write a register value as a floating point decimal number to the opened file	0x80
FLPT_WFN	Write Number to File	Literal	Write a literal value as a floating point decimal number to the opened file	0x81
FLPT_WFN	Write Number to File	Address	Write 64-bits (4 bytes) of memory starting at	0x82

Mnemonic	Full Name	Operands	Function	Instruction Code
			the address in a label as a floating point decimal number to the opened file	
FLPT_WFN	Write Number to File	Pointer	Write 64-bits (4 bytes) of memory starting at the address in a register as a floating point decimal number to the opened file	0x83
Conversions				
FLPT_EXH	Extend Half Precision Float to Double Precision Float	Register	Convert the value in a register from a half- precision float (16- bits) to a double- precision float (64- bits)	0x90
FLPT_EXS	Extend Single Precision Float to Double Precision Float	Register	Convert the value in a register from a single- precision float (32- bits) to a double- precision float (64- bits)	0x91
FLPT_SHS	Shrink Double Precision Float	Register	Convert the value in a	0x92

Mnemonic	Full Name	Operands	Function	Instruction Code
	to Single Precision Float		register from a double-precision float (64-bits) to a single-precision float (32-bits)	
FLPT_SHH	Shrink Double Precision Float to Half Precision Float	Register	Convert the value in a register from a double-precision float (64-bits) to a half-precision float (16-bits)	0x93
FLPT_NEG	Negation	Register	Reverse the sign of the floating point number in a register, equivalent to flipping the sign bit.	0xA0
FLPT_UTF	Convert Unsigned Quad Word to Double Precision Float	Register	Convert the unsigned value in a register to a double-precision float (64-bits)	0xB0
FLPT_STF	Convert Signed Quad Word to Double Precision Float	Register	Convert the signed value in a register to a double-precision float (64-bits)	0xB1

Mnemonic	Full Name	Operands	Function	Instruction Code
FLPT_FTS	Convert Double Precision Float to Signed Quad Word through Truncation	Register	Convert the double-precision float (64-bits) value in a register to a signed 64-bit integer by rounding toward 0	0xC0
FLPT_FCS	Convert Double Precision Float to Signed Quad Word through Ceiling Rounding	Register	Convert the double-precision float (64-bits) value in a register to a signed 64-bit integer by rounding to the greater integer	0xC1
FLPT_FFS	Convert Double Precision Float to Signed Quad Word through Floor Rounding	Register	Convert the double-precision float (64-bits) value in a register to a signed 64-bit integer by rounding to the lesser integer	0xC2
FLPT_FNS	Convert Double Precision Float to Signed Quad Word through Nearest Rounding	Register	Convert the double-precision float (64-bits) value in a register to the nearest signed 64-bit integer, rounding midpoints to	0xC3

Mnemonic	Full Name	Operands	Function	Instruction Code
			the nearest even number	
Comparison				
FLPT_CMP	Compare	Register, Register	Subtract a register from another, discarding the result whilst still updating status flags	0xD0
FLPT_CMP	Compare	Register, Literal	Subtract a literal value from a register, discarding the result whilst still updating status flags	0xD1
FLPT_CMP	Compare	Register, Address	Subtract the contents of memory at an address in a label from a register, discarding the result whilst still updating status flags	0xD2
FLPT_CMP	Compare	Register, Pointer	Subtract the contents of memory at an address in a register from a register, discarding the result whilst still updating status flags	0xD3

Extended Base Set

Extension set number 0x03, opcodes start with 0xFF, 0x03. Contains additional instructions that complement the base instruction set, but do not provide any major additional functionality.

Mnemonic	Full Name	Operands	Function	Instruction Code
Byte Operations				
EXTD_BSW	Reverse Byte Order	Register	Reverse the byte order of a register, thereby converting little endian to big endian and vice versa	0x00

External Assembly Extension Set

Extension set number 0x04, opcodes start with 0xFF, 0x04. Contains instructions that enable interoperation with external C#/.NET programs.

Mnemonic	Full Name	Operands	Function	Instruction Code
Loading				
ASMX_LDA	Load Assembly	Address	Open the .NET Assembly at the path specified by a 0x00 terminated string in memory starting at an address in a label	0x00
ASMX_LDA	Load Assembly	Pointer	Open the .NET Assembly at the path specified by a 0x00 terminated string in	0x01

Mnemonic	Full Name	Operands	Function	Instruction Code
			memory starting at an address in a register	
ASMX_LDF	Load Function	Address	Open the function in the open .NET assembly with the name specified by a 0x00 terminated string in memory starting at an address in a label	0x02
ASMX_LDF	Load Function	Pointer	Open the function in the open .NET assembly with the name specified by a 0x00 terminated string in memory starting at an address in a register	0x03
Closing				
ASMX_CLA	Close Assembly	-	Close the currently open .NET Assembly, as well as any open function	0x10
ASMX_CLF	Close Function	-	Close the	0x11

Mnemonic	Full Name	Operands	Function	Instruction Code
			currently open function, the assembly stays open	
Validity Check				
ASMX_AEX	Assembly Valid	Address	Store 1 in a register if the .NET Assembly at the path specified in memory starting at an address in a label exists and is valid, else 0	0x20
ASMX_AEX	Assembly Valid	Pointer	Store 1 in a register if the .NET Assembly at the path specified in memory starting at an address in a register exists and is valid, else 0	0x21
ASMX_FEX	Function Valid	Address	Store 1 in a register if the function with the name specified in memory starting at an address in a label exists in the open .NET Assembly	0x22

Mnemonic	Full Name	Operands	Function	Instruction Code
			and is valid, else 0	
ASMX_FEX	Function Valid	Pointer	Store 1 in a register if the function with the name specified in memory starting at an address in a register exists in the open .NET Assembly and is valid, else 0	0x23
Calling				
ASMX_CAL	Call External Function	-	Call the loaded external function, giving null as the passed value	0x30
ASMX_CAL	Call External Function	Register	Call the loaded external function, giving the value of a register as the passed value	0x31
ASMX_CAL	Call External Function	Literal	Call the loaded external function, giving a literal value as the passed value	0x32
ASMX_CAL	Call External Function	Address	Call the loaded	0x33

Mnemonic	Full Name	Operands	Function	Instruction Code
			external function, giving the contents of memory at an address in a label as the passed value	
ASMX_CAL	Call External Function	Pointer	Call the loaded external function, giving the contents of memory at an address in a register as the passed value	0x34

Memory Allocation Extension Set

Extension set number 0x05, opcodes start with 0xFF, 0x05. Contains instructions that provide runtime memory management, ensuring that memory regions are non-overlapping and that there is enough free memory available.

Mnemonic	Full Name	Operands	Function	Instruction Code
Allocation				
HEAP_ALC	Allocate Memory	Register, Register	Allocate a block of memory with the value of a register as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	0x00
HEAP_ALC	Allocate	Register,	Allocate a	0x01

Mnemonic	Full Name	Operands	Function	Instruction Code
	Memory	Literal	block of memory with a literal value as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	
HEAP_ALC	Allocate Memory	Register, Address	Allocate a block of memory with the contents of memory at an address in a label as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	0x02
HEAP_ALC	Allocate Memory	Register, Pointer	Allocate a block of memory with the contents of memory at an address in a register as its size, storing the first address of the allocated block in a	0x03

Mnemonic	Full Name	Operands	Function	Instruction Code
			register, throwing an error if the operation fails	
HEAP_TRY	Try Allocate Memory	Register, Register	Allocate a block of memory with the value of a register as its size, storing the first address of the allocated block in a register, or storing -1 if the operation fails	0x04
HEAP_TRY	Try Allocate Memory	Register, Literal	Allocate a block of memory with a literal value as its size, storing the first address of the allocated block in a register, or storing -1 if the operation fails	0x05
HEAP_TRY	Try Allocate Memory	Register, Address	Allocate a block of memory with the contents of memory at an address in a label as its size, storing the first address of the allocated	0x06

Mnemonic	Full Name	Operands	Function	Instruction Code
HEAP_TRY	Try Allocate Memory	Register, Pointer	Allocate a block of memory with the contents of memory at an address in a register as its size, storing the first address of the allocated block in a register, or storing -1 if the operation fails	0x07
Re-allocation				
HEAP_REA	Re-allocate Memory	Register, Register	Re-allocate a block of memory starting at the address in a register with the value of a register as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	0x10

Mnemonic	Full Name	Operands	Function	Instruction Code
HEAP_REA	Re-allocate Memory	Register, Literal	Re-allocate a block of memory starting at the address in a register with a literal value as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	0x11
HEAP_REA	Re-allocate Memory	Register, Address	Re-allocate a block of memory starting at the address in a register with the contents of memory at an address in a label as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	0x12
HEAP_REA	Re-allocate Memory	Register, Pointer	Re-allocate a block of memory starting at the address	0x13

Mnemonic	Full Name	Operands	Function	Instruction Code
			in a register with the contents of memory at an address in a register as its size, storing the first address of the allocated block in a register, throwing an error if the operation fails	
HEAP_TRE	Try Re-allocate Memory	Register, Register	Re-allocate a block of memory starting at the address in a register with the value of a register as its size, storing the first address of the allocated block in a register, or storing -1 if the operation fails	0x14
HEAP_TRE	Try Re-allocate Memory	Register, Literal	Re-allocate a block of memory starting at the address in a register with a literal value as its size, storing	0x15

Mnemonic	Full Name	Operands	Function	Instruction Code
			the first address of the allocated block in a register, or storing -1 if the operation fails	
HEAP_TRE	Try Re-allocate Memory	Register, Address	Re-allocate a block of memory starting at the address in a register with the contents of memory at an address in a label as its size, storing the first address of the allocated block in a register, or storing -1 if the operation fails	0x16
HEAP_TRE	Try Re-allocate Memory	Register, Pointer	Re-allocate a block of memory starting at the address in a register with the contents of memory at an address in a register as its size, storing the first address of the allocated	0x17

Mnemonic	Full Name	Operands	Function	Instruction Code
			block in a register, or storing -1 if the operation fails	
Freeing				
HEAP_FRE	Free Memory	Register	Free a block of memory starting at the address in a register	0x20

ASCII Table

The following is a list of common characters and their corresponding byte value in decimal.

Code (Dec)	Code (Hex)	Character
10	0A	LF (line feed, new line)
13	0D	CR (carriage return)
32	20	SPACE
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3

Code (Dec)	Code (Hex)	Character
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V

Code (Dec)	Code (Hex)	Character
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y

Code (Dec)	Code (Hex)	Character
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~

Copyright © 2022-2023 Ptolemy Hill