# AssEmbly Language Reference

Applies to versions: 2.0.0

Last revised: 2023-08-27

## Introduction

AssEmbly is a mock processor architecture and assembly language written in C# and running on .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.

AssEmbly was designed and implemented in its entirety by [Tolly Hill](https://github.com/TollyH).

## Table of Contents

* [AssEmbly Language Reference](#assembly-language-reference)
  + [Introduction](#introduction)
  + [Table of Contents](#table-of-contents)
  + [Technical Information](#technical-information)
  + [Basic Syntax](#basic-syntax)
    - [Mnemonics and Operands](#mnemonics-and-operands)
    - [Comments](#comments)
    - [Labels](#labels)
  + [Operand Types](#operand-types)
    - [Register](#register)
    - [Literal](#literal)
    - [Address](#address)
    - [Pointer](#pointer)
  + [Registers](#registers)
    - [Register Table](#register-table)
    - [rpo — Program Offset](#rpo--program-offset)
    - [rsf — Status Flags](#rsf--status-flags)
    - [rrv — Return Value](#rrv--return-value)
    - [rfp — Fast Pass Parameter](#rfp--fast-pass-parameter)
    - [rso — Stack Offset](#rso--stack-offset)
    - [rsb — Stack Base](#rsb--stack-base)
    - [rg0 - rg9 — General Purpose](#rg0---rg9--general-purpose)
  + [Moving Data](#moving-data)
    - [Moving with Literals](#moving-with-literals)
    - [Moving with Registers](#moving-with-registers)
    - [Moving with Memory](#moving-with-memory)
  + [Maths and Bitwise Operations](#maths-and-bitwise-operations)
    - [Addition and Multiplication](#addition-and-multiplication)
    - [Subtraction](#subtraction)
    - [Division](#division)
    - [Shifting](#shifting)
    - [Bitwise](#bitwise)
    - [Random Number Generation](#random-number-generation)
    - [Lack of Signed Numbers and Workarounds](#lack-of-signed-numbers-and-workarounds)
  + [Jumping](#jumping)
  + [Comparing, Testing, and Branching](#comparing-testing-and-branching)
    - [Comparing Numbers](#comparing-numbers)
    - [Testing Bits](#testing-bits)
    - [Checking For a Carry or For Zero](#checking-for-a-carry-or-for-zero)
  + [Assembler Directives](#assembler-directives)
    - [PAD — Byte Padding](#pad--byte-padding)
    - [DAT — Byte Insertion](#dat--byte-insertion)
      * [Escape Sequences](#escape-sequences)
    - [NUM — Number Insertion](#num--number-insertion)
    - [MAC — Macro Definition](#mac--macro-definition)
    - [IMP — File Importing](#imp--file-importing)
    - [ANALYZER — Toggling Assembler Warnings](#analyzer--toggling-assembler-warnings)
  + [Console Input and Output](#console-input-and-output)
  + [File Handling](#file-handling)
    - [Opening and Closing](#opening-and-closing)
    - [Reading and Writing](#reading-and-writing)
    - [Other Operations](#other-operations)
  + [The Stack](#the-stack)
    - [Using the Stack to Preserve Registers](#using-the-stack-to-preserve-registers)
  + [Subroutines](#subroutines)
    - [Fast Calling](#fast-calling)
    - [Return Values](#return-values)
    - [Subroutines and the Stack](#subroutines-and-the-stack)
    - [Passing Multiple Parameters](#passing-multiple-parameters)
  + [Text Encoding](#text-encoding)
  + [Full Instruction Reference](#full-instruction-reference)
    - [Base Instruction Set](#base-instruction-set)
  + [ASCII Table](#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 |
| 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 (practical) / unlimited (theoretical) |
| Instruction Count | 167 opcodes (49 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 and all **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 like 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.

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 a 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 34 when referenced, as each instruction prior will take up 17 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, 0x1\_000\_000 ; This is valid, will be assembled as 0x1000000 (16777216)  
MVQ rg0, 0x10\_0\_\_000\_0 ; This is still valid, underscores don't have to be uniform  
  
MVQ rg0, 0x\_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

### 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 | Writeable | 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 |
| 0x06 | rg0 | Yes | General 0 | *General purpose* |
| 0x07 | rg1 | Yes | General 1 | *General purpose* |
| 0x08 | rg2 | Yes | General 2 | *General purpose* |
| 0x09 | rg3 | Yes | General 3 | *General purpose* |
| 0x0A | rg4 | Yes | General 4 | *General purpose* |
| 0x0B | rg5 | Yes | General 5 | *General purpose* |
| 0x0C | rg6 | Yes | General 6 | *General purpose* |
| 0x0D | rg7 | Yes | General 7 | *General purpose* |
| 0x0E | rg8 | Yes | General 8 | *General purpose* |
| 0x0F | rg9 | Yes | General 9 | *General purpose* |

### 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 3** bits of the 64-bit value have a special use — the remaining 61 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 3 bits currently in use are:

0b00...0000000FCZ  
  
... = 52 omitted bits  
F = File end flag  
C = Carry flag  
Z = Zero 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.

The file end flag is set to 1 by the RFC operation if the byte that has just been read from the currently open file was the last byte in that file. It is reset to 0 only upon opening a file again.

The carry flag is set to 1 after a mathematical or bitwise operation if the result of that operation caused the destination register to go below 0 and wrap around to the upper limit, go above the limit of a 64-bit integer and wrap around to 0 (more info in the section on maths), otherwise it is proactively set to 0.

The zero flag is set to 1 after a mathematical or bitwise operation if the result of that operation caused the destination register to become equal to 0, otherwise it is proactively set to 0.

For example:

MVQ rg0, 0 ; Set rg0 to 0  
; The zero flag has NOT been set here, moving is not a mathematical or bitwise operation  
SUB rg0, 1 ; Subtract 1 from rg0  
; rg0 will now equal the 64-bit integer limit, as it went below 0  
; The second bit (carry flag) in rsf is now 1  
; The first bit (zero flag) has not changed from 0  
ADD rg0, 1 ; Add 1 back to rg0  
; rg0 will now once again equal 0, as it went over the 64-bit integer limit  
; The second bit (carry flag) in rsf is still 1  
; The first bit (zero flag) has now been changed to 1  
ADD rg0, 1  
; rg0 is now 1, no limits were exceeded  
; The second bit (carry flag) in rsf has been set back to 0  
; The first bit (zero flag) in rsf has been set back to 0

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

### 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 2046 bytes in size (making 2045 the highest address):

WCN rso ; Outputs "2046"  
PSH 5 ; Push the literal 5 to the stack  
WCN rso ; Outputs "2038" (stack values are 8 bytes)  
POP rg0 ; Pop the just-pushed 5 into rg0  
WCN rso ; Outputs "2046"

### 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 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 1111111111111111, 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 0000000000000000000000000000000000000000000000001111111111111111 — 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 2046 bytes of available memory (making 2045 the maximum address), you cannot use MVQ on the starting address 2043, as that requires at least 8 bytes.

## Maths and Bitwise Operations

AssEmbly supports ten different mathematical operations and five different bitwise operations (one being the random number generator). Each one operates **in-place**, 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

Numbers in AssEmbly are **unsigned**, meaning that negative numbers cannot be natively represented (more on how to work around this later). 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  
  
MVQ rg0, 25 ; Set rg0 to 25  
SUB rg0, 50 ; Subtract 50 from rg0  
; rg0 is now 18446744073709551591

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 AssEmbly: 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 rg0, 0b00101  
ORR rg0, 0b10100  
; rg0 now has a value of 0b10101

Bitwise Exclusive Or (XOR):

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

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

MVQ rg0, 0b00101  
XOR rg0, 0b10100  
; rg0 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 rg0, 0b00101  
NOT rg0  
; rg0 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

### Lack of Signed Numbers and Workarounds

AssEmbly can only store and operate on **unsigned** integers, meaning it has no native support for negative numbers.

In some scenarios, this won’t make a difference. For example, if a negative number is simply a midpoint for an addition/subtraction with a positive overall answer, you can continue to use the wrapped-around value without error:

MVQ rg0, 5  
SUB rg0, 10  
; Expected value is -5, rg0 is actually 18446744073709551611  
MVQ rg1, 25  
ADD rg1, rg0  
; rg1 is still 20 now as expected, even though rg0 doesn't have the value we expect

If a result of a subtraction has wrapped around (which can be checked using the carry flag), the absolute result (the non-negative result) can be found by performing a bitwise not on and incrementing the result by 1, like so:

MVQ rg0, 5  
SUB rg0, 10  
NOT rg0  
ICR rg0  
; rg0 is now 5, carry bit is set

Using the carry flag, this operation can be done conditionally (more in the section on branching), as it should not be done on positive results:

MVQ rg0, 5  
SUB rg0, rg1 ; Assume rg1 has a value that could cause rg0 to be negative or positive  
JNC :NOT\_NEGATIVE ; Jump to label if carry flag is unset  
NOT rg0 ; These will only run if rg0 wrapped (setting carry flag)  
ICR rg0  
:NOT\_NEGATIVE  
; rg0 is now the absolute result

## 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 it is important that you are 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 a condition is first checked 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) |  
+----------+----------------------------------+

The top section of instructions should be performed following a CMP instruction (explained in the section on comparing). The bottom 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).

### Comparing Numbers

One of the most common types of branch is checking how two numbers relate to each other. This can be achieved with the CMP instruction. 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.

### 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 For a Carry or For Zero

As shown in the section on working around unsigned numbers, the carry flags and zero flags can also be checked following a mathematical operation.

To repeat that example:

MVQ rg0, 5  
SUB rg0, rg1 ; Assume rg1 has a value that could cause rg0 to be negative or positive  
JNC :NOT\_NEGATIVE ; Jump to label if carry flag is unset  
NOT rg0 ; This will only run if rg0 wrapped (setting carry flag)  
ICR rg0  
:NOT\_NEGATIVE  
; rg0 is now the absolute result

JNC here is checking if the carry flag is set or not following the subtraction. The jump will only occur if the carry flag is 0 (unset), otherwise, as with the other jump types, execution will continue as normal. JCA can be used to perform the inverse, jump only if the carry flag is set.

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.

## 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 00 9F 06 FD 02 00 00 00 00 00 00 11 06 08 00 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  
 | 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

Or an example of using a string:

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 after this example.

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 00 04 2D 00 00 00 00 00 00 00 14 06 CC 07 02 0A 00 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"

#### Escape Sequences

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

| Escape sequence | Character name | Notes |
| --- | --- | --- |
| \" | Double quote | Used to insert a double quote into the string without causing the string to end. |
| \\ | Backslash | In order 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. |
| \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. |
| \' | Single quote | Included for future expansion. Not currently required - simply type a ' character instead. |

### 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 integers.

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 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

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).

Note that macros cannot contain commas or unclosed quotes ("), and surrounding whitespace will be ignored. They are case sensitive, and macros with the same name but different capitalisation can exist simultaneously.

### 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 AssEmbly 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 AssEmbly 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 an 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 at 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 2046 bytes in size (making 2045 the maximum address):

; rso = 2046  
; | Addresses | 2022..2029 | 2030..2037 | 2038..2045 ||  
; | Value | ???????????????? | ???????????????? | ???????????????? ||  
  
PSH 0xDEADBEEF ; Push 0xDEADBEEF (3735928559) to the stack  
  
; rso = 2038  
; | Addresses | 2022..2029 | 2030..2037 || 2038..2045 |  
; | Value | ???????????????? | ???????????????? || 00000000EFBEADDE |  
  
PSH 0xCAFEB0BA ; Push 0xCAFEB0BA (3405689018) to the stack  
  
; rso = 2030  
; | Addresses | 2022..2029 || 2030..2037 | 2038..2045 |  
; | Value | ???????????????? || 00000000BAB0FECA | 00000000EFBEADDE |  
  
PSH 0xD00D2BAD ; Push 0xD00D2BAD (3490524077) to the stack  
  
; rso = 2022  
; | Addresses || 2022..2029 | 2030..2037 | 2038..2045 |  
; | Value || 00000000AD2B0DD0 | 00000000BAB0FECA | 00000000EFBEADDE |  
  
POP rg0 ; Pop the most recent non-popped item from the stack into rg0  
  
; rso = 2030  
; | Addresses | 2022..2029 || 2030..2037 | 2038..2045 |  
; | Value | ???????????????? || 00000000BAB0FECA | 00000000EFBEADDE |  
; rg0 = 0xD00D2BAD  
  
POP rg0 ; Pop the most recent non-popped item from the stack into rg0  
  
; rso = 2038  
; | Addresses | 2022..2029 | 2030..2037 || 2038..2045 |  
; | Value | ???????????????? | ???????????????? || 00000000EFBEADDE |  
; rg0 = 0xCAFEB0BA

### 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. It is important that all values apart from these two are popped off the stack prior to 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 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.

## Full Instruction Reference

### Base Instruction Set

Extension set number 0x00, opcodes start with 0xFF, 0x00.

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 | 0x00 |
| NOP | No Operation | - | Do nothing | 0x01 |
| **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 / JZO | Jump if Equal / Jump if Zero | Address | Jump to an address in a label only if the zero status flag is set | 0x04 |
| JEQ / JZO | 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 Than / Jump if Carry | Pointer | Jump to an address in a register only if the carry status flag is set | 0x09 |
| 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 |
| **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 memory at an address in a register from a register | 0x23 |
| 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 remainder | 0x41 |
| 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 an address in a register, storing the remainder | 0x47 |
| 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 left by a literal value | 0x51 |
| 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 by another | 0x60 |
| AND | Bitwise And | Register, Literal | 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 |
| 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 a label, discarding the result | 0x72 |
| TST | Test | Register, Pointer | Bitwise and a register and the contents of memory at an address in a register, discarding the result | 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 | 0x76 |
| CMP | Compare | Register, Pointer | Subtract the contents of memory at an address in a register from a register, discarding the result | 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, Register | Move the lower 8-bits of a register to the contents of memory at an address in a label | 0x84 |
| 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 address in a label to a register | 0x8A |
| 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 |
| 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 Word | Pointer, Register | Move the lower 32-bits (4 bytes) of a register to the contents of memory at an address in a register | 0x96 |
| 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) of a register to the contents of memory at an address in a label | 0x9C |
| 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 top of the stack | 0xA2 |
| PSH | Push to Stack | Pointer | 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 rpo and rsb to the stack | 0xB0 |
| CAL | Call Subroutine | Pointer | Call the subroutine at an address in a register, pushing rpo and rsb 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 rfp | 0xB2 |
| CAL | Call Subroutine | Address, Literal | Call the subroutine at an address in a label, moving a literal value to rfp | 0xB3 |
| 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 an address in a label to rfp | 0xB8 |
| CAL | Call Subroutine | Pointer, Pointer | 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 Subroutine | Pointer | Pop the previous states off the stack, moving the contents of memory at an address in a register to rrv | 0xBE |
| **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 decimal number to the console | 0xC4 |
| 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 the address in a label as a hexadecimal number to the console | 0xCA |
| 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** |  |  |  |  |
| 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 of a literal value as a decimal number to the opened file | 0xD5 |
| 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 the address in a label as a hexadecimal number to the opened file | 0xDA |
| 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 as a raw byte to the opened file | 0xDF |
| **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 0x00 terminated string in memory starting at an address in a register | 0xE4 |
| 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 address in a register | 0xE8 |
| **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 |

## ASCII Table

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

| Code | Character |
| --- | --- |
| 10 | LF (line feed, new line) |
| 13 | CR (carriage return) |
| 32 | SPACE |
| 33 | ! |
| 34 | ” |
| 35 | # |
| 36 | $ |
| 37 | % |
| 38 | & |
| 39 | ’ |
| 40 | ( |
| 41 | ) |
| 42 | \* |
| 43 | + |
| 44 | , |
| 45 | - |
| 46 | . |
| 47 | / |
| 48 | 0 |
| 49 | 1 |
| 50 | 2 |
| 51 | 3 |
| 52 | 4 |
| 53 | 5 |
| 54 | 6 |
| 55 | 7 |
| 56 | 8 |
| 57 | 9 |
| 58 | : |
| 59 | ; |
| 60 | < |
| 61 | = |
| 62 | > |
| 63 | ? |
| 64 | @ |
| 65 | A |
| 66 | B |
| 67 | C |
| 68 | D |
| 69 | E |
| 70 | F |
| 71 | G |
| 72 | H |
| 73 | I |
| 74 | J |
| 75 | K |
| 76 | L |
| 77 | M |
| 78 | N |
| 79 | O |
| 80 | P |
| 81 | Q |
| 82 | R |
| 83 | S |
| 84 | T |
| 85 | U |
| 86 | V |
| 87 | W |
| 88 | X |
| 89 | Y |
| 90 | Z |
| 91 | [ |
| 92 | \ |
| 93 | ] |
| 94 | ^ |
| 95 | \_ |
| 96 | ` |
| 97 | a |
| 98 | b |
| 99 | c |
| 100 | d |
| 101 | e |
| 102 | f |
| 103 | g |
| 104 | h |
| 105 | i |
| 106 | j |
| 107 | k |
| 108 | l |
| 109 | m |
| 110 | n |
| 111 | o |
| 112 | p |
| 113 | q |
| 114 | r |
| 115 | s |
| 116 | t |
| 117 | u |
| 118 | v |
| 119 | w |
| 120 | x |
| 121 | y |
| 122 | z |
| 123 | { |
| 124 | | |
| 125 | } |
| 126 | ~ |

**Copyright © 2022–2023 Ptolemy Hill**