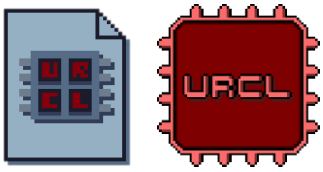
URCL UNNAMED BETA OFFICIAL DOCUMENTATION

Written by Ben Aitken (ModPunchtree) – 3rd January 2021



ABSTRACT

URCL == Universal Reduced Computer Language

URCL is a simple universal intermediate language. It is designed to be as similar as possible to common RISC assembly in order to make it as easy as possible to translate to any specific assembly language. This documentation will go into greater depth than the Google Sheet documentation.

CONTENTS

INTRODUCTION

URCL first started with Minecraft CPUs and has also been called Universal Redstone Computer Language. However, URCL is not limited to only Minecraft, as it can be applied to a wide range of CPUs with any ISA (Instruction Set Architecture).

CPUs which are compatible with URCL can make use of the tools for URCL. These tools include emulators and high-level language compilers. Programs which are written in URCL can also be shared between any other URCL compatible CPU regardless of the ISAs of the CPUs.

Links

URCL Official Documentation Repository:

https://github.com/ModPunchtree/URCL

> *Go here to find the most up to date version of the official URCL documentation.*

URCL Discord:

https://discord.gg/Nv8jzWg5j8

URCL Google Sheet Documentation:

https://docs.google.com/spreadsheets/d/1YUCj-J1KTTxho59_RsKWj9JZa96_mLqB-j_kK2pjqM8/edit?usp=sharing

URCL Ports Google Sheet Documentation:

https://docs.google.com/spreadsheets/d/1_ztRKWEm2LjHLb3Bxw0JOyZjJ9Drsj-hyOE-TFpwtS4/edit?usp=sharing

OVERVIEW

Source Files

URCL Source Files

All URCL code should be contained in ".urcl" files. These are plain text files. The name of the file can be any string of letters, numbers, and underscore.

> *To edit URCL code it is recommended that VSCode is used along with the "URCL & B Syntax Highlighter" extension by RedCMD on the VSCode marketplace.*

> *To directly emulate URCL code, several emulators are available in the URCL Discord server. Some emulators run in the browser and others run using Python or C++.*

General Syntax

General Layout

All URCL instructions contain an Identifier as well as Sources and a Destination. The Identifier is simply the name of the instruction. Sources specify where data should be fetched from, and the Destination specifies where the result of the instruction should be written to.

URCL instructions generally follow a three operand format. This means that there are three or fewer operands in each instruction.

In written form, the instructions take the format:

```
Identifier Destination Source1 Source2
```

For example:

```
ADD R1 R2 R3
```

> *Add the contents of register 2 to the contents of register 3. Then write result into register 1.*

All URCL instructions are atomic and are fully self-contained.

Atomic means that instructions are executed one at a time sequentially, and the next instruction does not start until the previous has finished.

Self-contained means that each instruction can be executed without any external information outside of the sources specified by the instruction itself. This means that the exact same instruction will always do the exact same thing regardless of the current state of the CPU.

URCL instructions are designed to be translated one at a time into the target CPU's assembly.

This means that any URCL program can be translated easily, provided each instruction has an equivalent translation on the target CPU.

> *URCL uses a load-store architecture.*
>
> *Which means that values must be loaded from the RAM into the registers in order to be used. Then the results must be stored back into the RAM.*

Prefixes

There are prefixes for general purpose registers, memory, labels, relative numbers, and ports.

Registers

Registers are prefixed with either `R` or `$`. For example:

`R1` or `$1` refer to general purpose register 1

Memory

Memory locations are prefixed with either `M` or `#`. For example:

`M0` or `#0` refer to memory location 0.

> *Note that "memory" here does not refer to the entire RAM space, it only refers to the Heap which is later described in the Memory Map section.*

> *If memory locations are used in an instruction which is not LOD, STR, LLOD or LSTR then it gets translated to an immediate value which points to that memory location.*

Labels

Labels are prefixed with `.`. For example:

`.test` refers to the label called "test".

Relative Numbers

Relative Numbers are prefixed with + or -. For example:

`+2` is a relative number that is positive 2.

Ports

Ports are prefixed with %. For example:

`%7SEG` refers to a port called "7SEG"

Comments

Comments in URCL are the same as comments in C. Line comments are denoted using `//`. Multi-line comments are denoted using `/*` and `*/`. For example:

`//comment` is a line comment.

```
/*
comment
*/
```

is a multi-line comment.

Macros

All macros are prepended with `@`. For example:

```
@define TEST 2
```

> *Individual macros are not defined in URCL as they are completely up to the particular URCL interpreter to define. This is to enable different interpreters to define macros to suit their own needs.*

Numbers

Numbers that have no prefix are in base 10, such as a number used as an immediate value. For example:

`IMM R1 5` in this example the `5` is being used as an immediate value and it has no prefix so it will be interpreted as being in base 10.

Base 16 and base 2 numbers can also be used but they must be prepended with 0x and 0b respectively. For example:

`IMM R1 0x5` the `0x5` will be interpreted as a base 16 value.

`IMM R1 0b101` the `0b101` will be interpreted as a base 2 value.

> *The example instruction in the previous 3 examples all load an immediate value of 5, into register 1.*

Base 8 numbers are prefixed with 0o. Numbers that are prefixed with 0 will be treated as base 10 numbers.

> *Base 8 numbers are rarely used.*

Relative Numbers

Relative numbers are used to specify the address of an instruction, relative to the current instruction. These are values are prefixed with a `~+` or a `~-`. For example:

`JMP ~+5` the ~+5 refers to the URCL instruction 5 ahead of the current instruction.

> *A relative value of ~+0 or ~-0 refers to the address of the current instruction.*

> *Relative values must be converted into labels before being translated.*

Defined Immediate Values

Defined immediate values are values which are directly translated into an immediate value before translating the URCL code into the target assembly. All defined immediate values are prepended with a `&`.

The following table contains all the defined immediate values:

Defined Immediate Value	Full Name	Value
`&BITS`	Bits	Equal to the value of the BITS header
`&MINREG`	Minimum registers	Equal to the value of the MINREG header
`&MINHEAP`	Minimum heap	Equal to the value of the MINHEAP header
`&MINSTACK`	Minimum stack	Equal to the value of the MINSTACK header
`&HEAP`	Heap	Equal to the maximum size of the heap (where the stack is empty, and the heap occupies all available space in the RAM) Note this is specific to the target CPU instead of the URCL program
`&MSB`	Most significant bit	Equal to a binary value with only the most significant bit active (128 in an 8 bit program)
`&SMSB`	Signed most significant bit	Equal to a binary value with only the second most significant bit active (64 in an 8 bit program)
`&MAX`	Maximum	Equal to a binary value with all bits active (255 in an 8 bit program)
`&SMAX`	Signed maximum	Equal to a binary value with all bits active except the most significant bit (127 in an 8 bit program)
`&UHALF`	Upper half	Equal to a binary value with all bits greater than or equal to $2^{\frac{BITS}{2}}$ active (240 in an 8 bit program)
`&LHALF`	Lower half	Equal to a binary value with all bits less than $2^{\frac{BITS}{2}}$ active (15 in an 8 bit program)

ASCII Characters

ASCII Characters In the code must enclosed using `'`. These characters are directly translated into an immediate value (based on 7 bit ASCII) before being translated from URCL code into the target assembly.

For example:

`'C'` would become an immediate value of 67.

`'5'` would become an immediate value of 53.

When translating characters into immediate values, refer to:

https://montcs.bloomu.edu/Information/Encodings/ascii-7.html

Whitespace

All spaces in URCL are ignored. This means that spaces can be put anywhere, and code can be indented however much the programmer wants while still being valid.

However, newlines are important as these mark where one instruction ends and the next begins. This means that multiple instructions cannot be put on the same line. Empty lines will be ignored though, meaning the programmer can have as many empty lines in between their instructions as they like.

> *The whitespace should be used to make the code as legible as possible.*

Zero Register

The zero register is a register that cannot be overwritten and always reads 0. The zero register is referred to in the same way as any other general purpose register. So, `R0` and `$0` both refer to register zero.

If the zero register is specified as a source operand to in an instruction, then it is the same as using an immediate value of zero.

If the zero register is specified as the destination operand in an instruction, then the output of the instruction is simply discarded.

> *In most circumstances it is advisable that the zero register is never used as a destination as this is the same as doing nothing in most instructions.*

Program Counter

The program counter is a register that points to the beginning of the current instruction. The program counter in URCL can be read from in the exact same way that any other general purpose register is read.

The program counter is referred to using `PC`. For example:

`PSH PC` this pushes the value currently in the program counter onto the stack.

> *Reading or writing to the program counter directly should be avoided, if possible, as some target CPUs may struggle to translate this code if the program counter cannot be accessed directly.*

Headers

Headers contain information which tells the URCL interpreter the required specific parameters for running a program. The headers can also allow you to see if a program is compatible with a target CPU.

CPU Word Length

URCL assumes that the target CPU uses the same word length for everything. This means that an 8 bit CPU can have a maximum of 256 memory locations, 256 general purpose registers and any value larger than 8 bits in the code would be truncated to make it 8 bits.

The word length is specified on a per-program basis, this means that every URCL program must specify the word length it runs at. This is done using the `BITS` header. For example:

`BITS == 8` this specifies that the word length must be exactly 8 bits for this program.

`BITS >= 8` this specifies that the word length can be 8 or more bits.

`BITS <= 8` this specifies that the word length can be 8 or fewer bits.

> *Most programs will only run at a single word length, so >= and <= are rarely used outside of libraries.*

> *If the BITS header is missing, then the program should be assumed to be 8 bit.*

> *If the BITS header is missing the "==" or ">=" or "<=" then it is assumed to be "==".*
>
> *So "BITS 8" is the same as "BITS == 8".*

Minimum Number of Registers

The number of registers that can be used in URCL is fixed and each program needs to specify the minimum number of general purpose registers it requires. This is done using the MINREG header. For example:

`MINREG 4` this specifies that this program requires a minimum of 4 general purpose registers which means that any CPU at least 4 general purpose registers can run the program (provided it meets all other requirements).

> *If the MINREG header is missing, then the assumed value is 8.*

Minimum Heap Space

The minimum number of words of heap space a program needs is specified using the `MINHEAP` header. For example:

`MINHEAP 16` this specifies that this program needs 16 words of heap space to run.

> *Note that the Heap does not refer to the entire RAM space. The Heap is described in more detail in the Memory Map section.*

> *If the MINHEAP header is missing, then the assumed value is 16.*

Instruction Storage Architecture

There are two ways that instructions can be stored on a target CPU. The instructions can be stored in the same RAM space that the program runs in (for example von Neumann architecture) or the instructions can be stored in a separate space which cannot be accessed while the program is running (for example Harvard architecture).

URCL programs which store data inside of the instructions will only work if the instructions are stored in the same space that the program is running in. So, it is important that programs specify which storage architecture they require. This is done using the `RUN` header. For example:

`RUN RAM` specifies that the instructions are stored in the same space the program runs in.

`RUN ROM` specifies that the instructions are not stored in the same space the program runs in.

> *If the RUN header is missing, then the assumed value is RUN ROM.*

Minimum Stack Size

Programs can specify the minimum number of words that the stack must be able to hold in order to run a program. This is done using the `MINSTACK` header. For example:

`MINSTACK 32` specifies that the stack must be able to hold at least 32 values to run this program.

> *If the MINSTACK header is missing, then the assumed value is 8.*

> *The stack is explained further in the Memory Map section.*

> *Headers can be located anywhere within a program, but they should be at the very top to make it clearer to anyone reading the code.*

Define Words

Define words are predefined values that exist inside of the RAM/ROM space where the URCL program is stored. In a RUN RAM program these values can freely be read and written to in the same way any other value in the heap can be. However, in a RUN ROM program, these values can be read but not written to, since the program is stored separately from the main RAM.

These are useful for storing predefined arrays, strings, or lookup tables within a URCL program.

Define Word Definition

Defined words are created by writing `DW` followed by a value. The value must be able to fit in a single word (so, the value must be between 0 and 255 in an 8 bit program). For example:

```
DW 0x45
```

This defines a value of “0x45” directly inside of the URCL program.

Arrays of values can also be defined by writing `DW` followed by an array of values enclosed with square braces `[` and `]`. For example:

```
DW [0 1 2 3]
```

This defines 4 sequential values. The first is 0, the second is 1, the third is 2 and the final value is 3.

The above example is identical to:

```
DW 0
DW 1
DW 2
DW 3
```

Where each value inside of the array is a single, separate defined value.

> *Note that the order the values in the array are defined, starts with the first item in the array. The order must not be changed.*

> *Defined values that are in series (such as an array) must be stored in adjacent RAM address values in the target CPU. This is so that any value in the array can be accessed by adding its index to the address of the first value.*

Define Word Usage

Defined words can be pointed to using relative values or labels. For example:

```
.test2
    DW 0x45
```

The label “test2” points to the defined value of “0x45”.

Values inside of an array can be accessed by adding the array index to the address value of the first item in the array. For example:

```
.test3
    DW 0
    DW 1
```

The second defined value (1) can be accessed by adding 1 the “test3” label, then reading/writing to that address. So:

```
LLOD R1 .test3 1
```

This will load the value located at the address value pointed to by “test3” plus one, into register one.

Labels

Labels point towards a particular memory or instruction location. Labels in URCL work similar to labels in most assembly languages.

Label Definition

Labels are defined by writing `.` followed by the name of the label on a line. That label then points to the instruction or data contained in the next line. The label name must be unique and can be made of string of letters, numbers and underscore. For example:

```
.test
    ADD R1 R2 R3
```

This defines the label “test”, and this label points to the instruction: `ADD R1 R2 R3`.

Labels can also point to data that is stored inside of the instructions as defined values. For example:

```
.test2
    DW 0x45
```

This defines the label “test2”, and this points to the defined value “0x45” which is located inside of the instructions.

> *DW means “Define Word” and it is used to put one word of data into the instructions.*
>
> *Since DW values are located in the instructions there is a risk of executing these as instructions. This should be avoided as this can cause undefined behaviour in the target CPU.*
>
> *This particular fault is defined as “Non-Instruction Execution”.*

Label Usage

Once defined, labels can be used in the code as source operands. They act the same as immediate values as a label is simply an immediate value which points to the address it was defined at. For example:

`JMP .test` which branches to the location of the label “test”.

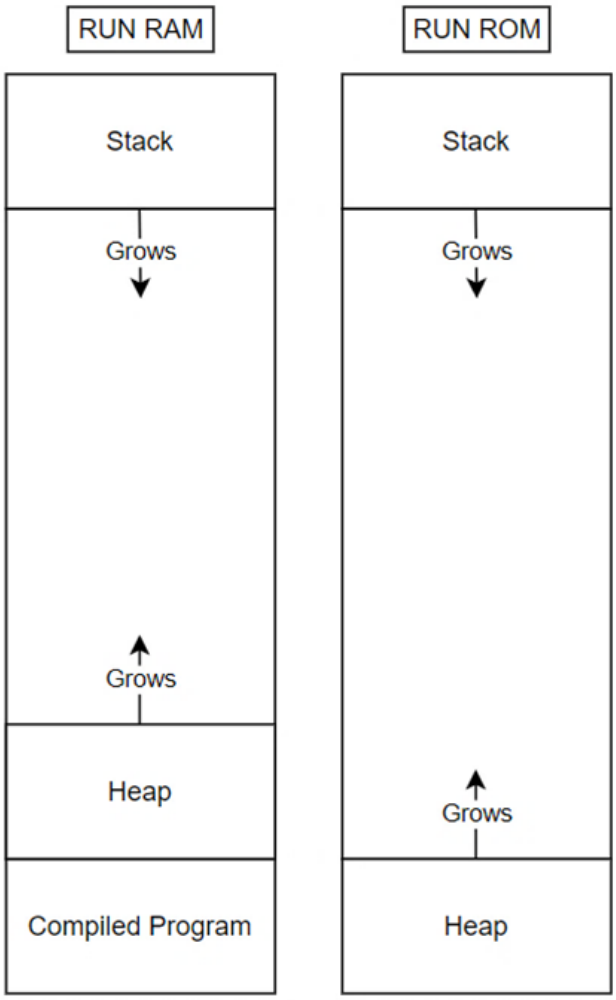
> *Labels are converted to immediate values after being translated to the target CPUs assembly code.*
>
> *This means that they can be translated in the exact same way as an immediate value would.*

`ADD R1 .test 1` which adds 1 to the location of the label “test”.

> *Since the size of the instructions on the target CPU can be bigger than one word, adding 1 to a label which points at an instruction does not make that label point to the next instruction.*
>
> *Labels can only be added to or subtracted from if that label points to DW values as these are guaranteed to occupy 1 word per value, regardless of the target CPU.*

Memory Map

- Normal memory values should be prepended with ‘M’ or ‘#’.
- Normal memory values do not use an absolute RAM address, instead they are relative to the start of the Heap.
- M0 points to the start of the Heap.
- If a memory address is specified without being prepended with ‘M’ or ‘#’ then it will be treated as an absolute address.
- The stack pointer must be stored in a dedicated register.



The RAM layout depends on the instruction storage architecture specified using the RUN header.

Heap

The heap either starts at location zero in a RUN ROM program or it starts at the first available location after the space that the program itself occupies in a RUN RAM program. Then the heap in both cases expands upwards.

The Heap is where M and # prepended values go.

> *There is no limit to the size of the Heap, other than the total size of the RAM.*

The literal RAM address of M or # prepended values in a RUN RAM program requires an offset to be added. The offset is equal to the location of M0. So, the location of MX is M0 + X which applies to any memory address.

> *Knowing the location of M0 is important when translating URCL to the target CPU’s assembly.*

Stack

The stack always starts at the top of the RAM (the highest address value) and expands downwards. This is a LIFO stack.

> *There is no limit to the size of the Stack, other than the total size of the RAM.*

> *While the Heap and Stack can be any size, they must **never** cross over each other. If they crossed over each other, they would overwrite each other.*
>
> *This particular fault is defined as "Stack Overflow".*

Stack Pointer

The stack pointer points to the final item on the stack, rather than the next available space. When an item is added to the stack the stack pointer is decremented by 1, and when an item is removed from the stack the stack pointer is incremented by 1.

The stack pointer in URCL must be stored in a dedicated general purpose register. This means it can be read and written to in the same way as any other register. To specify the stack pointer register, `SP` is used. For example:

`MOV R1 SP` this instruction reads from the stack pointer register.

> *Modifying the stack pointer directly is potentially dangerous as it can become out of sync to the stack. So, avoid doing this if it is not necessary.*

INSTRUCTIONS

> *This section will define all the instructions within URCL*

There are two main categories of instructions. These are "Basic" and "Complex". There are also "Core" instructions which are a specific subset of the Basic instructions which are the minimum instructions required for a CPU to be 100% compatible with URCL.

All Complex instructions can be translated to Basic instructions and all Basic instructions can be translated to Core instructions. This means that if a target CPU can translate the Core instructions it can translate all URCL instructions.

Note that all instructions are unsigned, unless otherwise stated.

Core Instructions

> *A CPU must be able to translate all these instructions to be 100% compatible with URCL*

There are 6 Core instructions.

ADD

Full Name

Add

Description

The ADD instruction adds two values together, then it stores the result in a register.

> *The input values can be either registers or immediate values.*

Operands

ADD requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`ADD R1 R2 R3`

Code Examples

```
ADD R1 R1 R2
```

This instruction adds the value in register 1 to the value in register 2, then it stores the result into register 1.

RSH

Full Name

Right shift

Description

The RSH instruction does a bitwise right shift of a value, then it stores the result in a register.

> *Note that this is unsigned.*
>
> *The lowest bit is shifted out and is lost in this instruction.*
>
> *So, if the lowest bit is important, then save it before right shifting.*
>
> *Note that this is non-cyclic.*

Operands

RSH requires 2 operands.

Destination	Source1	Example
Register	Register	`RSH R1 R2`

Code Examples

```
RSH R1 R1
```

This instruction right shifts the value in register 1, then it stores the result into register 1.

LOD

Full Name

Load

Description

The LOD instruction copies a value from the RAM at a specified address into a register.

Operands

LOD requires 2 operands.

Destination	Source1	Example
Register	Register (Pointer)	`LOD R1 R2`

Code Examples

```
LOD R1 R1
```

This instruction copies the RAM value addressed by the value in register 1, then it stores the result into register 1.

STR

Full Name

Store

Description

The STR instruction copies a value into the RAM at a specified address.

Operands

STR requires 2 operands.

Destination	Source1	Example
Register (Pointer)	Register	`STR R1 R2`

Code Examples

```
STR R1 R1
```

This instruction copies the value in register 1 into the RAM value addressed by the value in register 1.

BGE

Full Name

Branch if greater than or equal to

Description

The BGE instruction branches to a specified address if one value is greater than or equal to another value.

> *Note that this is unsigned.*

Operands

BGE requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`BGE R1 R2 R3`

Code Examples

```
BGE R1 R3 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 3 is greater than or equal to the value in register 2.

NOR

Full Name

Bitwise NOR

Description

The NOR instruction does a bitwise NOR of two values, then it stores the result in a register.

Operands

NOR requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`NOR R1 R2 R3`

Code Examples

```
NOR R1 R1 R2
```

This instruction does a bitwise NOR of the value in register 1 and the value in register 2, then it stores the result into register 1.

IMM

Full Name

Immediate

Description

The IMM instruction copies an immediate value into a register.

Operands

IMM requires 2 operands.

Destination	Source1	Example
Register	Immediate	`IMM R1 6`

Code Examples

```
IMM R3 5
```

This instruction copies the immediate value 5 and stores it into register 3.

Basic Instructions

These are relatively simple instructions that can be translated into core instructions if needed.

ADD

Full Name

Add

Description

The ADD instruction adds two values together, then it stores the result in a register.

> *The input values can be either registers or immediate values.*

Operands

ADD requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`ADD R1 R2 R3`
Register	Register	Immediate	`ADD R1 R2 1`
Register	Immediate	Register	`ADD R1 1 R3`
Register	Immediate	Immediate	`ADD R1 1 2`

Code Examples

```
ADD R3 3 5
```

This instruction adds the immediate value of 3 to the immediate value of 5 and stores the result (8) into register 3.

```
ADD R1 R1 R2
```

This instruction adds the value in register 1 to the value in register 2, then it stores the result into register 1.

RSH

Full Name

Right shift

Description

The RSH instruction does a bitwise right shift of a value, then it stores the result in a register.

Note that this is unsigned.

The lowest bit is shifted out and is lost in this instruction.

So, if the lowest bit is important, then save it before right shifting.

Note that this is non-cyclic.

Operands

RSH requires 2 operands.

Destination	Source1	Example
Register	Register	`RSH R1 R2`
Register	Immediate	`RSH R1 1`

Code Examples

```
RSH R3 3
```

This instruction right shifts the immediate value of 3 and stores the result (1) into register 3.

```
RSH R1 R1
```

This instruction right shifts the value in register 1, then it stores the result into register 1.

LOD

Full Name

Load

Description

The LOD instruction copies a value from the RAM at a specified address into a register.

Operands

LOD requires 2 operands.

Destination	Source1	Example
Register	RAM Address (Relative)	`LOD R1 M2`
Register	RAM Address (Literal)	`LOD R1 1`
Register	Register (Pointer)	`LOD R1 R2`
Program Counter	RAM Address (Relative)	`LOD PC M2`
Program Counter	RAM Address (Literal)	`LOD PC 1`
Program Counter	Register (Pointer)	`LOD PC R2`

Loading directly into the program counter should be avoided if possible. This is because it may be hard to translate to some target CPUs which cannot access their program counter directly.

Code Examples

```
LOD R3 3
```

This instruction copies the RAM value addressed by an immediate value of 3 and stores the result into register 3.

```
LOD R1 R1
```

This instruction copies the RAM value addressed by the value in register 1, then it stores the result into register 1.

STR

Full Name

Store

Description

The STR instruction copies a value into the RAM at a specified address.

Operands

STR requires 2 operands.

Destination	Source1	Example
RAM Address (Relative)	Register	`STR M2 R1`
RAM Address (Literal)	Register	`STR 1 R1`
Register	Register	`STR R1 R2`
RAM Address (Relative)	Immediate	`STR M2 5`
RAM Address (Literal)	Immediate	`STR 1 5`
Register	Immediate	`STR R1 5`

Code Examples

```
STR 3 R3
```

This instruction copies the value in register 3 into the RAM value addressed by an immediate value of 3.

```
STR R1 R1
```

This instruction copies the value in register 1 into the RAM value addressed by the value in register 1.

BGE

Full Name

Branch if greater than or equal to

Description

The BGE instruction branches to a specified address if one value is greater than or equal to another value.

> *Note that this is unsigned.*

Operands

BGE requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BGE .foo R2 R3`
Immediate	Register	Immediate	`BGE .foo R2 1`

Immediate	Immediate	Register	`BGE .foo 1 R3`
Register	Register	Register	`BGE R1 R2 R3`
Register	Register	Immediate	`BGE R1 R2 1`
Register	Immediate	Register	`BGE R1 1 R3`

Code Examples

```
BGE .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is greater than or equal to the immediate value of 5.

```
BGE R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 is greater than or equal to the value in register 2.

NOR

Full Name

Bitwise NOR

Description

The NOR instruction does a bitwise NOR of two values, then it stores the result in a register.

Operands

NOR requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`NOR R1 R2 R3`
Register	Register	Immediate	`NOR R1 R2 1`
Register	Immediate	Register	`NOR R1 1 R3`

Code Examples

```
NOR R3 3 R2
```

This instruction does a bitwise NOR of the immediate value of 3 and the value in register 2 and stores the result into register 3.

```
NOR R1 R1 R2
```

This instruction does a bitwise NOR of the value in register 1 and the value in register 2, then it stores the result into register 1.

SUB

Full Name

Subtract

Description

The SUB instruction subtracts one values from another, then it stores the result in a register.

Operands

SUB requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SUB R1 R2 R3`
Register	Register	Immediate	`SUB R1 R2 1`
Register	Immediate	Register	`SUB R1 1 R3`
Register	Immediate	Immediate	`SUB R1 1 2`

Code Examples

```
SUB R3 3 5
```

This instruction subtracts the immediate value of 3 from the immediate value of 5 and stores the result (-2 in 2's complement) into register 3.

> *Negative numbers will use 2's complement. So, -2 on an 8 bit CPU would be the equivalent of 254.*

```
SUB R1 R1 R2
```

This instruction subtracts the value in register 1 from the value in register 2, then it stores the result into register 1.

JMP

Full Name

Jump

Description

The JMP instruction branches to a specified value.

Operands

JMP requires 1 operand.

Destination	Example
Immediate	`JMP 5`
Register	`JMP R1`

Code Examples

```
JMP .test
```

This instruction jumps to the instruction addressed by the label "test".

```
JMP R1
```

This instruction jumps to the instruction addressed by the value in register 1.

MOV

Full Name

Move

Description

The MOV instruction copies a value into a register.

Operands

MOV requires 2 operands.

Destination	Source1	Example
Register	Register	`MOV R1 R2`
Register	Immediate	`MOV R1 .foo`

Code Examples

```
MOV R1 R2
```

This instruction copies the value in register 2, then it stores it into register 1.

```
MOV R3 M5
```

This instruction copies the address of memory location 5 (as an immediate value) and stores it into register 3.

> *Note that if a memory address is used in a location where an immediate would normally go, it is converted into an immediate value which points to the address of that memory location.*
>
> *So, the M5 here is converted to the literal RAM address of memory location 5 in the Heap.*

NOP

Full Name

No operation

Description

The NOP instruction does nothing.

> *NOP should never be used in the majority of URCL programs since there is no point to making the target CPU do nothing if every instruction is atomic.*
>
> *Note that there are no read before write hazards in URCL and branching occurs instantly.*

Operands

NOP requires 0 operands.

Code Examples

```
NOP
```

This instruction does nothing.

IMM

Full Name

Immediate

Description

The IMM instruction copies an immediate value into a register.

Operands

IMM requires 2 operands.

Destination	Source1	Example
Register	Immediate	`IMM R1 6`

Code Examples

```
IMM R3 5
```

This instruction copies the immediate value 5 and stores it into register 3.

> *Since MOV also accepts immediates, MOV can always be used in place of IMM.*
>
> *But using IMM when loading immediates is preferred as it makes the code clearer to the reader.*
>
> *MOV allows immediates because this makes compiling to URCL a little easier.*

LSH

Full Name

Left shift

Description

The LSH instruction does a bitwise left shift of a value, then it stores the result in a register.

> *The uppermost bit is shifted out and is lost in this instruction.*
>
> *So, if the uppermost bit is important then save it before left shifting.*

> *Note that this is non-cyclic.*

Operands

LSH requires 2 operands.

Destination	Source1	Example
Register	Register	`LSH R1 R2`
Register	Immediate	`LSH R1 1`

Code Examples

```
LSH R3 3
```

This instruction left shifts the immediate value of 3 and stores the result (6) into register 3.

```
LSH R1 R1
```

This instruction left shifts the value in register 1, then it stores the result into register 1.

INC

Full Name

Increment

Description

The INC instruction adds 1 to a value then stores the result into a register.

Operands

INC requires 2 operands.

Destination	Source1	Example
Register	Register	`INC R1 R2`
Register	Immediate	`INC R1 .foo`

Code Examples

```
INC R1 R2
```

This instruction adds 1 to the value in register 2, then it stores it into register 1.

```
INC R3 .foo
```

This instruction adds 1 to the address of the label “foo” and stores it into register 3.

> *Since the size of the instructions on the target CPU can be bigger than one word, adding 1 to a label which points at an instruction does not make that label point to the next instruction.*
>
> *Labels should only be added to or subtracted from if that label points to DW values as these are guaranteed to occupy 1 word per value regardless of the target CPU.*

DEC

Full Name

Decrement

Description

The DEC instruction subtracts 1 from a value then stores the result into a register.

Operands

DEC requires 2 operands.

Destination	Source1	Example
Register	Register	`DEC R1 R2`
Register	Immediate	`DEC R1 .foo`

Code Examples

```
DEC R1 R2
```

This instruction subtracts 1 from the value in register 2, then it stores it into register 1.

```
DEC R3 .foo
```

This instruction subtracts 1 from the address of the label “foo” and stores it into register 3.

NEG

Full Name

Negate

Description

The NEG instruction calculates the negation of the value, interpreted as 2's compliment, then stores the result into a register.

Operands

NEG requires 2 operands.

Destination	Source1	Example
Register	Register	`NEG R1 R2`
Register	Immediate	`NEG R1 5`

Code Examples

```
NEG R1 R2
```

This instruction calculates the 2’s complement of the value in register 2 and stores the result into register 1.

```
NEG R3 5
```

This instruction calculates the 2’s complement of the immediate value 5 and stores the result (-5) into register 3.

AND

Full Name

Bitwise AND

Description

The AND instruction does a bitwise AND of two values, then it stores the result in a register.

Operands

AND requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`AND R1 R2 R3`
Register	Register	Immediate	`AND R1 R2 1`
Register	Immediate	Register	`AND R1 1 R3`

Code Examples

```
AND R3 3 R2
```

This instruction does a bitwise AND of the immediate value of 3 and the value in register 2 and stores the result into register 3.

```
AND R1 R1 R2
```

This instruction does a bitwise AND of the value in register 1 and the value in register 2, then it stores the result into register 1.

OR

Full Name

Bitwise OR

Description

The OR instruction does a bitwise OR of two values, then it stores the result in a register.

Operands

OR requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`OR R1 R2 R3`
Register	Register	Immediate	`OR R1 R2 1`
Register	Immediate	Register	`OR R1 1 R3`

Code Examples

```
OR R3 3 R2
```

This instruction does a bitwise OR of the immediate value of 3 and the value in register 2 and stores the result into register 3.

```
OR R1 R1 R2
```

This instruction does a bitwise OR of the value in register 1 and the value in register 2, then it stores the result into register 1.

NOT

Full Name

Bitwise NOT

Description

The NOT instruction does a bitwise NOT of a value, then it stores the result in a register.

Operands

NOT requires 2 operands.

Destination	Source1	Example
Register	Register	`NOT R1 R2`
Register	Immediate	`NOT R1 1`

Code Examples

```
NOT R1 R1
```

This instruction does a bitwise NOT of the value in register 1, then it stores the result into register 1.

```
NOT R3 3
```

This instruction does a bitwise NOT of the immediate value of 3 and stores the result into register 3.

> *On an 8 bit CPU the result of NOT of 3 would be 252.*

XNOR

Full Name

Bitwise XNOR

Description

The XNOR instruction does a bitwise XNOR of two values, then it stores the result in a register.

Operands

XNOR requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`XNOR R1 R2 R3`
Register	Register	Immediate	`XNOR R1 R2 1`
Register	Immediate	Register	`XNOR R1 1 R3`

Code Examples

```
XNOR R3 3 R2
```

This instruction does a bitwise XNOR of the immediate value 3 and the value in register 2 and stores the result into register 3.

```
XNOR R1 R1 R2
```

This instruction does a bitwise XNOR of the value in register 1 and the value in register 2, then it stores the result into register 1.

XOR

Full Name

Bitwise XOR

Description

The XOR instruction does a bitwise XOR of two values, then it stores the result in a register.

Operands

XOR requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`XOR R1 R2 R3`
Register	Register	Immediate	`XOR R1 R2 1`
Register	Immediate	Register	`XOR R1 1 R3`

Code Examples

```
XOR R3 3 R2
```

This instruction does a bitwise XOR of the immediate value 3 and the value in register 2 and stores the result into register 3.

```
XOR R1 R1 R2
```

This instruction does a bitwise XOR of the value in register 1 and the value in register 2, then it stores the result into register 1.

NAND

Full Name

Bitwise NAND

Description

The NAND instruction does a bitwise NAND of two values, then it stores the result in a register.

Operands

NAND requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`NAND R1 R2 R3`
Register	Register	Immediate	`NAND R1 R2 1`
Register	Immediate	Register	`NAND R1 1 R3`

Code Examples

```
NAND R3 3 R2
```

This instruction does a bitwise NAND of the immediate value of 3 and the value in register 2 and stores the result into register 3.

```
NAND R1 R1 R2
```

This instruction does a bitwise NAND of the value in register 1 and the value in register 2, then it stores the result into register 1.

BRL

Full Name

Branch if less than

Description

The BRL instruction branches to a specified address if one value is less than another value.

> *Note that this is unsigned.*

Operands

BRL requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BRL .foo R2 R3`
Immediate	Register	Immediate	`BRL .foo R2 1`
Immediate	Immediate	Register	`BRL .foo 1 R3`
Register	Register	Register	`BRL R1 R2 R3`
Register	Register	Immediate	`BRL R1 R2 1`
Register	Immediate	Register	`BRL R1 1 R3`

Code Examples

```
BRL .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is less than the immediate value of 5.

```
BRL R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 is less than the value in register 2.

BRG

Full Name

Branch if greater than

Description

The BRG instruction branches to a specified address if one value is less than another value.

> *Note that this is unsigned.*

Operands

BRG requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BRG .foo R2 R3`
Immediate	Register	Immediate	`BRG .foo R2 1`
Immediate	Immediate	Register	`BRG .foo 1 R3`

Register	Register	Register	`BRG R1 R2 R3`
Register	Register	Immediate	`BRG R1 R2 1`
Register	Immediate	Register	`BRG R1 1 R3`

Code Examples

```
BRG .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is greater than the immediate value of 5.

```
BRG R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 is greater than the value in register 2.

BRE

Full Name

Branch if equal to

Description

The BRE instruction branches to a specified address if one value is equal to another value.

Operands

BRE requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BRE .foo R2 R3`
Immediate	Register	Immediate	`BRE .foo R2 1`
Immediate	Immediate	Register	`BRE .foo 1 R3`
Register	Register	Register	`BRE R1 R2 R3`
Register	Register	Immediate	`BRE R1 R2 1`
Register	Immediate	Register	`BRE R1 1 R3`

Code Examples

```
BRE .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is equal to the immediate value of 5.

```
BRE R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 is equal to the value in register 2.

BNE

Full Name

Branch if not equal to

Description

The BNE instruction branches to a specified address if one value is equal to another value.

Operands

BNE requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BNE .foo R2 R3`
Immediate	Register	Immediate	`BNE .foo R2 1`
Immediate	Immediate	Register	`BNE .foo 1 R3`
Register	Register	Register	`BNE R1 R2 R3`
Register	Register	Immediate	`BNE R1 R2 1`
Register	Immediate	Register	`BNE R1 1 R3`

Code Examples

```
BNE .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is not equal to the immediate value of 5.

```
BNE R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 is not equal to the value in register 2.

BOD

Full Name

Branch if odd

Description

The BOD instruction branches to a specified address if a value is odd.

> *A value is odd if the lowest bit is active.*

Operands

BOD requires 2 operands.

Destination	Source1	Example
Immediate	Register	`BOD .foo R2`
Register	Register	`BOD R1 R2`

Code Examples

```
BOD .foo R1
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is odd.

```
BOD R1 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 2 is odd.

BEV

Full Name

Branch if even

Description

The BEV instruction branches to a specified address if a value is even.

> *A value is even if the lowest bit is not active.*

Operands

BEV requires 2 operands.

Destination	Source1	Example
Immediate	Register	`BEV .foo R2`
Register	Register	`BEV R1 R2`

Code Examples

```
BEV .foo R1
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is even.

```
BEV R1 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 2 is even.

BLE

Full Name

Branch if less than or equal to

Description

The BLE instruction branches to a specified address if one value is less than or equal to another value.

> *Note that this is unsigned.*

Operands

BLE requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BLE .foo R2 R3`
Immediate	Register	Immediate	`BLE .foo R2 1`
Immediate	Immediate	Register	`BLE .foo 1 R3`
Register	Register	Register	`BLE R1 R2 R3`
Register	Register	Immediate	`BLE R1 R2 1`
Register	Immediate	Register	`BLE R1 1 R3`

Code Examples

```
BLE .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is less than or equal to the immediate value of 5.

```
BLE R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 is less than or equal to the value in register 2.

BRZ

Full Name

Branch if zero

Description

The BRZ instruction branches to a specified address if a value is equal to zero.

Operands

BRZ requires 2 operands.

Destination	Source1	Example
Immediate	Register	`BRZ .foo R2`
Register	Register	`BRZ R1 R2`

Code Examples

```
BRZ .foo R1
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is equal to zero.

```
BRZ R1 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 2 is equal to zero.

BNZ

Full Name

Branch if not zero

Description

The BNZ instruction branches to a specified address if a value is equal to zero.

Operands

BNZ requires 2 operands.

Destination	Source1	Example
Immediate	Register	`BNZ .foo R2`
Register	Register	`BNZ R1 R2`

Code Examples

```
BNZ .foo R1
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 is not equal to zero.

```
BNZ R1 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 2 is not equal to zero.

BRN

Full Name

Branch if negative

Description

The BRN instruction branches to a specified address if a value is negative.

> *This is signed.*

A value is negative if the highest bit is active. (2's complement)

Operands

BRN requires 2 operands.

Destination	Source1	Example
Immediate	Register	`BRN .foo R2`
Register	Register	`BRN R1 R2`

Code Examples

```
BRN .foo R1
```

This instruction branches to the instruction pointed to by the label "foo" if the value in register 1 is negative.

```
BRN R1 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 2 is negative.

BRP

Full Name

Branch if positive

Description

The BRP instruction branches to a specified address if a value is positive or zero.

This is signed.

A value is positive if the highest bit is not active. (2's complement)

Operands

BRP requires 2 operands.

Destination	Source1	Example
Immediate	Register	`BRP .foo R2`
Register	Register	`BRP R1 R2`

Code Examples

```
BRP .foo R1
```

This instruction branches to the instruction pointed to by the label "foo" if the value in register 1 is positive.

```
BRP R1 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the value in register 2 is positive.

PSH

Full Name

Push

Description

The PSH instruction pushes a value onto the stack.

> *Since the stack pointer points to the topmost filled value in the stack, the stack pointer is first decremented before writing to the location it points to. This happens automatically in the PSH instruction.*

Operands

PSH requires 1 operand.

Source1	Example
Register	`PSH R1`
Immediate	`PSH 5`

Code Examples

```
PSH R1
```

This instruction pushes the value in register 1 onto the stack.

```
PSH .test
```

This instruction pushes the address of the label “test” onto the stack.

POP

Full Name

Pop

Description

The POP instruction pops a value from the stack into a register.

> *Since the stack pointer points to the topmost filled value in the stack, the value at the location where the stack pointer points is first read before incrementing the stack pointer. This happens automatically in the POP instruction.*

Operands

POP requires 1 operand.

Destination	Example
Register	`POP R1`

Code Examples

```
POP R1
```

This instruction pops from the stack into register 1.

CAL

Full Name

Call

Description

The CAL instruction pushes the address of the next instruction onto the stack then it branches to a specific address.

> *This is used to branch to subroutines.*

> *The address pushed onto the stack is the return address.*

Operands

CAL requires 1 operand.

Source1	Example
Immediate	`CAL .test`
Register	`CAL R1`

Code Examples

```
CAL .test
```

This instruction pushes the address of the next instruction onto the stack then it branches to the instruction pointed to by the label “test”.

```
CAL R1
```

This instruction pushes the address of the next instruction onto the stack then it branches to the instruction pointed to by the value in register 1.

RET

Full Name

Return

Description

The RET instruction pops a value from the stack then it branches to that value.

> *The value at the top of the stack must be a valid address of an instruction for RET to work.*
>
> *Otherwise, a “Non-Instruction Execution” fault may occur.*

Operands

RET requires 0 operands.

Code Examples

```
RET
```

This instruction pops a value from the stack then it branches to that value.

HLT

Full Name

Halt

Description

The HLT instruction halts execution.

> *This marks the end of a program.*
>
> *Once halted, the target CPU will need to be manually reset to run again.*

Operands

HLT requires 0 operands.

Code Examples

```
HLT
```

This instruction halts the target CPU.

CPY

Full Name

Copy

Description

The CPY instruction copies a value from the RAM at a specified address into another RAM location at another specified address.

Operands

CPY requires 2 operands.

Destination	Source1	Example
RAM Address (Relative)	RAM Address (Relative)	`CPY M1 M2`
RAM Address (Relative)	RAM Address (Literal)	`CPY M1 1`
RAM Address (Relative)	Register (Pointer)	`CPY M1 R2`
RAM Address (Literal)	RAM Address (Relative)	`CPY 1 M2`
RAM Address (Literal)	RAM Address (Literal)	`CPY 1 1`
RAM Address (Literal)	Register (Pointer)	`CPY 1 R2`
Register	RAM Address (Relative)	`CPY R1 M2`
Register	RAM Address (Literal)	`CPY R1 1`
Register	Register (Pointer)	`CPY R1 R2`

This instruction should be used when moving values around in the RAM.

This instruction allows for potentially shorter or faster translations than that of the equivalent LOD followed by a STR instruction.

Code Examples

```
CPY M3 3
```

This instruction copies the RAM value addressed by an immediate value of 3 and stores the result into memory location 3.

```
CPY R2 R1
```

This instruction copies the RAM value addressed by the value in register 1, then stores it into the RAM value addressed by the value in register 2.

BRC

Full Name

Branch if carry

Description

The BRC instruction branches to a specified address if one value added to another value activates the carry flag.

Note that the results of the addition in this instruction are not kept.

Operands

BRC requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BRC .foo R2 R3`
Immediate	Register	Immediate	`BRC .foo R2 1`
Immediate	Immediate	Register	`BRC .foo 1 R3`
Register	Register	Register	`BRC R1 R2 R3`
Register	Register	Immediate	`BRC R1 R2 1`
Register	Immediate	Register	`BRC R1 1 R3`

Code Examples

```
BRC .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 added to the immediate value of 5 activates the carry flag.

```
BRC R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 added to the value in register 2 activates the carry flag.

BNC

Full Name

Branch if no carry

Description

The BNC instruction branches to a specified address if one value added to another value does not activate the carry flag.

> *Note that the results of the addition in this instruction are not kept.*

Operands

BNC requires 3 operands.

Destination	Source1	Source2	Example
Immediate	Register	Register	`BNC .foo R2 R3`
Immediate	Register	Immediate	`BNC .foo R2 1`
Immediate	Immediate	Register	`BNC .foo 1 R3`
Register	Register	Register	`BNC R1 R2 R3`
Register	Register	Immediate	`BNC R1 R2 1`
Register	Immediate	Register	`BNC R1 1 R3`

Code Examples

```
BNC .foo R1 5
```

This instruction branches to the instruction pointed to by the label “foo” if the value in register 1 added to the immediate value of 5 does not activate the carry flag.

```
BNC R1 5 R2
```

This instruction branches to the instruction pointed to by the value in register 1 if the immediate value of 5 added to the value in register 2 does not activate the carry flag.

Complex Instructions

These are instructions which are typically more difficult to translate directly to a target CPU's assembly. These instructions can be translated into equivalent Basic and Core instructions if they cannot be directly translated.

There are 17 complex instructions.

MLT

Full Name

Multiply

Description

The MLT instruction multiplies two values together, then it stores the result in a register.

Operands

MLT requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`MLT R1 R2 R3`
Register	Register	Immediate	`MLT R1 R2 1`
Register	Immediate	Register	`MLT R1 1 R3`

Code Examples

```
MLT R3 3 R2
```

This instruction multiplies the immediate value of 3 with the value in register 2 and stores the result into register 3.

```
MLT R1 R1 R2
```

This instruction multiplies the value in register 1 with the value in register 2, then it stores the result into register 1.

DIV

Full Name

Division

Description

The DIV instruction divides one value by another, then it stores the result in a register.

> *This is integer division. So, the result is rounded down (towards zero) to the nearest integer.*
>
> *Note that this is unsigned.*

Operands

DIV requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`DIV R1 R2 R3`
Register	Register	Immediate	`DIV R1 R2 1`
Register	Immediate	Register	`DIV R1 1 R3`

Code Examples

```
DIV R3 5 R2
```

This instruction divides the immediate value of 5 by the value in register 2 and stores the result into register 3.

```
DIV R1 R1 R2
```

This instruction divides the value in register 1 by the value in register 2, then it stores the result into register 1.

MOD

Full Name

Modulus

Description

The MOD instruction calculates the remainder left after one value is divided by another, then it stores the result in a register.

This uses integer division. So, the dividend is rounded down (towards zero) to the nearest integer, leaving the remainder as the result.

Note that this is unsigned.

Operands

MOD requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`MOD R1 R2 R3`
Register	Register	Immediate	`MOD R1 R2 1`
Register	Immediate	Register	`MOD R1 1 R3`

Code Examples

```
MOD R3 5 R2
```

This instruction calculates the remainder left after the immediate value of 5 is divided by the value in register 2 and stores the result into register 3.

```
MOD R1 R1 R2
```

This instruction calculates the remainder left after the value in register 1 is divided by the value in register 2, then it stores the result into register 1.

BSR

Full Name

Barrel shift right

Description

The BSR instruction does a specific number of bitwise right shifts of a value, then it stores the result in a register.

Note that this is unsigned.

The bits that are shifted out in this instruction are lost.

So, if those bits are important, save them before shifting.

Note that this is non-cyclic.

Operands

BSR requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`BSR R1 R2 R3`
Register	Register	Immediate	`BSR R1 R2 1`
Register	Immediate	Register	`BSR R1 1 R3`

Code Examples

```
BSR R3 3 R2
```

This instruction right shifts the immediate value of 3 a number of times, this number is the value in register 2. Then it stores the result into register 3.

```
BSR R1 R1 R2
```

This instruction right shifts the value in register 1 a number of times, this number is the value in register 2. Then it stores the result into register 1.

BSL

Full Name

Barrel shift left

Description

The BSL instruction does a specific number of bitwise left shifts of a value, then it stores the result in a register.

> *The bits that are shifted out in this instruction are lost.*
>
> *So, if those bits are important, save them before shifting.*
>
> *Note that this is non-cyclic.*

Operands

BSL requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`BSL R1 R2 R3`
Register	Register	Immediate	`BSL R1 R2 1`
Register	Immediate	Register	`BSL R1 1 R3`

Code Examples

```
BSL R3 3 R2
```

This instruction left shifts the immediate value of 3 a number of times, this number is the value in register 2. Then it stores the result into register 3.

```
BSL R1 R1 R2
```

This instruction left shifts the value in register 1 a number of times, this number is the value in register 2. Then it stores the result into register 1.

SRS

Full Name

Signed right shift

Description

The SRS instruction does a signed right shift of a value, then it stores the result in a register.

Note that this is signed.

The lowest bit is shifted out and is lost in this instruction.

So, if the lowest bit is important, then save it before right shifting.

The sign bit (uppermost bit) is extended in this instruction.

Note that this is non-cyclic.

Operands

SRS requires 2 operands.

Destination	Source1	Example
Register	Register	`SRS R1 R2`
Register	Immediate	`SRS R1 1`

Code Examples

```
SRS R1 R1
```

This instruction does a signed right shift of the value in register 1, then it stores the result into register 1.

```
SRS R3 3
```

This instruction does a signed right shift of the immediate value of 3 and stores the result into register 3.

BSS

Full Name

Barrel shift right signed

Description

The BSS instruction does a specific number of signed right shifts of a value, then it stores the result in a register.

Note that this is signed.

The bits that are shifted out in this instruction are lost.

So, if those bits are important, save them before shifting.

Note that this is non-cyclic.

Operands

BSS requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`BSS R1 R2 R3`

Register	Register	Immediate	`BSS R1 R2 1`
Register	Immediate	Register	`BSS R1 1 R3`

Code Examples

```
BSS R3 3 R2
```

This instruction does a signed right shift of the immediate value of 3 a number of times, this number is the value in register 2. Then it stores the result into register 3.

```
BSS R1 R1 R2
```

This instruction does a signed right shift of the value in register 1 a number of times, this number is the value in register 2. Then it stores the result into register 1.

SETE

Full Name

Set if equal to

Description

The SETE instruction sets a register to all 1's in binary if one value is equal to another value, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*

Operands

SETE requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETE R1 R2 R3`
Register	Register	Immediate	`SETE R1 R2 1`
Register	Immediate	Register	`SETE R1 1 R3`

Code Examples

```
SETE R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 is equal to the immediate value of 5, otherwise it will write 0 into register 2.

```
SETE R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 is equal to the value in register 2, otherwise it will write 0 into register 1.

SETNE

Full Name

Set if not equal to

Description

The SETNE instruction sets a register to all 1's in binary if one value is not equal to another value, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*

Operands

SETNE requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETNE R1 R2 R3`
Register	Register	Immediate	`SETNE R1 R2 1`
Register	Immediate	Register	`SETNE R1 1 R3`

Code Examples

```
SETNE R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 is not equal to the immediate value of 5, otherwise it will write 0 into register 2.

```
SETNE R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 is not equal to the value in register 2, otherwise it will write 0 into register 1.

SETG

Full Name

Set if greater than

Description

The SETG instruction sets a register to all 1's in binary if one value is greater than another value, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*
>
> *Note that this is unsigned.*

Operands

SETG requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETG R1 R2 R3`
Register	Register	Immediate	`SETG R1 R2 1`
Register	Immediate	Register	`SETG R1 1 R3`

Code Examples

```
SETG R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 is greater than the immediate value of 5, otherwise it will write 0 into register 2.

```
SETG R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 is greater than the value in register 2, otherwise it will write 0 into register 1.

SETL

Full Name

Set if less than

Description

The SETL instruction sets a register to all 1's in binary if one value is less than another value, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*
>
> *Note that this is unsigned.*

Operands

SETL requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETL R1 R2 R3`
Register	Register	Immediate	`SETL R1 R2 1`
Register	Immediate	Register	`SETL R1 1 R3`

Code Examples

```
SETL R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 is less than the immediate value of 5, otherwise it will write 0 into register 2.

```
SETL R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 is less than the value in register 2, otherwise it will write 0 into register 1.

SETGE

Full Name

Set if greater than or equal to

Description

The SETGE instruction sets a register to all 1's in binary if one value is greater than another value, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*
>
> *Note that this is unsigned.*

Operands

SETGE requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETGE R1 R2 R3`
Register	Register	Immediate	`SETGE R1 R2 1`
Register	Immediate	Register	`SETGE R1 1 R3`

Code Examples

```
SETGE R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 is greater than or equal to the immediate value of 5, otherwise it will write 0 into register 2.

```
SETGE R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 is greater than or equal to the value in register 2, otherwise it will write 0 into register 1.

SETLE

Full Name

Set if less than or equal to

Description

The SETLE instruction sets a register to all 1's in binary if one value is greater than another value, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*

> *Note that this is unsigned.*

Operands

SETLE requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETLE R1 R2 R3`
Register	Register	Immediate	`SETLE R1 R2 1`
Register	Immediate	Register	`SETLE R1 1 R3`

Code Examples

```
SETLE R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 is less than or equal to the immediate value of 5, otherwise it will write 0 into register 2.

```
SETLE R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 is less than or equal to the value in register 2, otherwise it will write 0 into register 1.

SETC

Full Name

Set if carry

Description

The SETC instruction sets a register to all 1's in binary if one value added to another value activates the carry flag, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*

> *Note that the result of the addition is not kept.*

Operands

SETC requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETC R1 R2 R3`

Register	Register	Immediate	`SETC R1 R2 1`
Register	Immediate	Register	`SETC R1 1 R3`

Code Examples

```
SETC R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 added to the immediate value of 5 activates the carry flag, otherwise it will write 0 into register 2.

```
SETC R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 added to the value in register 2 activates the carry flag, otherwise it will write 0 into register 1.

SETNC

Full Name

Set if no carry

Description

The SETNC instruction sets a register to all 1's in binary if one value added to another value does not activate the carry flag, otherwise it sets that register to 0.

> *All 1's in binary on an 8 bit CPU is 255.*

> *Note that the result of the addition is not kept.*

Operands

SETNC requires 3 operands.

Destination	Source1	Source2	Example
Register	Register	Register	`SETNC R1 R2 R3`
Register	Register	Immediate	`SETNC R1 R2 1`
Register	Immediate	Register	`SETNC R1 1 R3`

Code Examples

```
SETNC R2 R1 5
```

This instruction will write all 1's into register 2 if the value in register 1 added to the immediate value of 5 does not activate the carry flag, otherwise it will write 0 into register 2.

```
SETNC R1 R1 R2
```

This instruction will write all 1's into register 1 if the value in register 1 added to the value in register 2 does not activate the carry flag, otherwise it will write 0 into register 1.

LLOD

Full Name

List load

Description

The LLOD instruction copies a value from the RAM at a specified address + offset into a register.

Operands

LLOD requires 3 operands.

Destination	Source1 (Base)	Source2 (Offset)	Example
Register	Register	Register	`LLOD R1 R2 R3`
Register	Register	Immediate	`LLOD R1 R2 1`
Register	Immediate	Register	`LLOD R1 1 R3`
Register	Immediate	Immediate	`LLOD R1 .foo 2`

Code Examples

```
LLOD R3 .foo 5
```

This instruction copies a value from the RAM at a specific address. This address is the address of the label “foo” added to the offset of an immediate value of 5. Then it stores the result into register 3.

```
LLOD R1 R1 R2
```

This instruction copies a value from the RAM at a specific address. This address is the value in register 1 added to the value in register 2. Then it stores the result into register 1.

LSTR

Full Name

List store

Description

The LSTR instruction writes a value into the RAM at a specified address + offset.

Operands

LSTR requires 3 operands.

Destination (Base)	Source1 (Offset)	Source2	Example
Register	Register	Register	`LSTR R1 R2 R3`
Register	Register	Immediate	`LSTR R1 R2 1`
Register	Immediate	Register	`LSTR R1 1 R3`
Register	Immediate	Immediate	`LSTR R1 2 R3`
Immediate	Register	Register	`LSTR .foo R2 R3`
Immediate	Register	Immediate	`LSTR .foo R2 1`
Immediate	Immediate	Register	`LSTR .foo 2 R3`
Immediate	Immediate	Immediate	`LSTR .foo 2 1`

Code Examples

```
LSTR .foo 5 R3
```

This instruction writes the value in register 3 into the RAM at a specific address. This address is the address of the label “foo” added to an immediate value of 5.

```
LSTR R1 R2 R3
```

This instruction writes the value in register 3 into the RAM at a specific address. This address is the value in register 1 added to the value in register 3.

I/O Instructions

These instructions cannot be translated into other instructions and must be directly translated in order to be ran on the target CPU.

There are 2 I/O instructions.

IN

Full Name

In

Description

The IN instruction reads the value on a particular port and writes it into a register.

> *Specific ports are defined in the Ports section.*
>
> *Note that ports can also be made up and do not have to follow the official documentation.*
>
> *In this case the programmer should define what is meant by each port if it is not obvious. A simple comment in the code is usually fine.*

Operands

IN requires 2 operands.

Destination	Source1	Example
Register	Port	`IN R1 %RNG`

Code Examples

```
IN R1 %RNG
```

This instruction reads from the port “RNG” (which is defined in the port documentation as a random number generator) and the result is written into register 1.

```
IN R2 %7SEG
```

This instruction reads from the port “7SEG” (which is **not** defined in the port documentation so it should be defined somewhere by the programmer), and the result is written into register 2.

OUT

Full Name

Out

Description

The OUT instruction reads a value and outputs the result into a specific port.

> *Specific ports are defined in the Ports section.*
>
> *Note that ports can also be made up and do not have to follow the official documentation.*
>
> *In this case the programmer should define what is meant by each port if it is not obvious. A simple comment in the code is usually fine.*

Operands

OUT requires 2 operands.

Destination	Source1	Example
Port	Register	`OUT %RNG R1`
Port	Immediate	`OUT %RNG 5`

Code Examples

```
OUT %RNG R1
```

This instruction reads the value in register 1 and writes it into the port “RNG” (which is defined in the port documentation as a random number generator).

```
OUT %7SEG 5
```

This instruction takes the immediate value 5 and writes it into port “7SEG” (which is **not** defined in the port documentation so it should be defined somewhere by the programmer).

INSTRUCTION TRANSLATIONS

All Basic instructions can be translated into Core instructions and all Complex instructions can be translated into Basic and Core instructions.

This section covers the translations for each instruction.

In this section “<A>” refers to the first operand, “<B>” refers to the second operand and “<C>” refers to the third operand.

There are multiple translations for different operands. Each with specific conditions where that translation is valid.

So, of the translations where the conditions are met, the shortest translation should be used.

All relative values must be converted into labels before translating.

This is to prevent relative values from being broken as the translations are usually longer than one instruction.

Basic Instruction Translations

ADD

Operand 1 <A>	Operand 2 <B>	Operand 3 <C>	Condition	Translation
Register	Register	Register	Any	None
Register	Register	Immediate	A temporary register is required	`IMM tempREG <C>` `ADD <A> <B> tempREG`
Register	Immediate	Register	A temporary register is required	`IMM tempREG <B>` `ADD <A> tempREG <C>`
Register	Immediate	Immediate	Two temporary registers are required	`IMM tempREG1 <B>` `IMM tempREG2 <C>` `ADD <A> tempREG1 tempREG2`

RSH

Operand 1 <A>	Operand 2 <B>	Condition	Translation
Register	Register	Any	None
Register	Immediate	A temporary register is required	`IMM tempREG <B>` `RSH <A> tempREG`

LOD

Operand 1 <A>	Operand 2 <B>	Condition	Translation
Register	Register	Any	None
Register	Immediate	A temporary register is required	`IMM tempREG <B>` `LOD <A> tempREG`
PC	Register	A temporary register is required	`LOD tempREG <B>` `JMP tempREG`
PC	Immediate	A temporary register is required	`LOD tempREG <B>` `JMP tempREG`

STR

Operand 1 <A>	Operand 2 <B>	Condition	Translation
Register	Register	Any	None
Register	Immediate	A temporary register is required	`IMM tempREG <A>` `STR tempREG <B>`
Immediate	Register	A temporary register is required	`IMM tempREG <B>` `STR <A> tempREG`
Immediate	Immediate	Two temporary registers are required	`IMM tempREG1 <A>` `IMM tempREG2 <B>` `STR tempREG1 tempREG2`

BGE

Operand 1 <A>	Operand 2 <B>	Operand 3 <C>	Condition	Translation
Register	Register	Register	Any	None
Register	Register	Immediate	A temporary register is required	`IMM tempREG <C>` `BGE <A> <B> tempREG`
Register	Immediate	Register	A temporary register is required	`IMM tempREG <B>` `BGE <A> tempREG <C>`

Immediate	Register	Register	A temporary register is required	`IMM tempREG <A>` `BGE tempREG <B> <C>`
Immediate	Register	Immediate	Two temporary registers are required	`IMM tempREG1 <A>` `IMM tempREG2 <C>` `BGE tempREG1 <B> tempREG2`
Immediate	Immediate	Register	Two temporary registers are required	`IMM tempREG1 <A>` `IMM tempREG2 <B>` `BGE tempREG1 tempREG2 <C>`

NOR

Operand 1 <A>	Operand 2 <B>	Operand 3 <C>	Condition	Translation
Register	Register	Register	Any	None
Register	Register	Immediate	A temporary register is required	`IMM tempREG <C>` `NOR <A> <B> tempREG`
Register	Immediate	Register	A temporary register is required	`IMM tempREG <B>` `NOR <A> tempREG <C>`

SUB

If the Operand types are not specified, then the translation applies to all possible combinations of operand types.

Condition	Translation
<B> is the same as <C>	`MOV <A> R0`
<A> is different to <B>	`NOT <A> <C>` `ADD <A> <A> <B>` `INC <A> <A>`
<A> is different to <C> and <C> is a register	`NOT <C> <C>` `ADD <A> <B> <C>` `INC <A> <A>` `NOT <C> <C>`
<A> is not R1	`PSH R1` `NOT R1 <C>` `ADD <A> <B> R1` `INC <A> <A>` `POP R1`
<A> is not R2	`PSH R2` `NOT R1 <C>` `ADD <A> <B> R1`

	`INC <A> <A>` `POP R2`

MOV

Condition	Translation
Any	`ADD <A> <B> R0`

NOP

Condition	Translation
Any	`MOV R0 R0`

IMM

Condition	Translation
Any	`ADD <A> <B> R0`

LSH

Condition	Translation
Any	`ADD <A> <B> <B>`

INC

Condition	Translation
Any	`ADD <A> <B> 1`

DEC

Condition	Translation
X is equal to an immediate value which has all its bits active. (255 in 8 bit)	`ADD <A> <B> X`

Some of the translations include instructions that are in the same category as the original instruction.

If this is the case, then the code will need further translation if the goal is to lower the tier of instructions.

NEG

Condition	Translation
Any	`NOT <A> <B>` `INC <A> <A>`

AND

Condition	Translation
<B> is the same as <C>	`MOV <C> <A>`
<A> is different to <C> and <C> is a register	`NOT <A> <B>` `NOT <C> <C>` `NOR <A> <A> <C>` `NOT <C> <C>`
<A> is different to <B> and <B> is a register	`NOT <B> <B>` `NOT <A> <C>` `NOR <A> <A> <B>` `NOT <B> <B>`
<A> is different to R1 or R2	`PSH R1` `PSH R2` `NOT R1 <B>` `NOT R2 <C>` `NOR <A> R1 R2` `POP R2` `POP R1`
<A> is different to R3 or R4	`PSH R3` `PSH R4` `NOT R3 <B>` `NOT R4 <C>` `NOR <A> R3 R4` `POP R4` `POP R3`

OR

Condition	Translation
Any	`NOR <A> <B> <C>` `NOT <A> <A>`

NOT

Condition	Translation
Any	`NOR <A> <B> R0`

XNOR

Condition	Translation
<B> is the same as <C> and X is equal to an immediate value which has all its bits active. (255 in 8 bit)	`IMM <A> X`

<A> is different to R1	AND <A> <B> <C> PSH R1 NOR R1 <B> <C> NOR <A> <A> R1 POP R1 NOT <A> <A>
<A> is different to R2	AND <A> <B> <C> PSH R2 NOR R1 <B> <C> NOR <A> <A> R2 POP R2 NOT <A> <A>

XOR

Condition	Translation
<B> is the same as <C>	MOV <A> R0
<A> is different to R1	AND <A> <B> <C> PSH R1 NOR R1 <B> <C> NOR <A> <A> R1 POP R1
<A> is different to R2	AND <A> <B> <C> PSH R2 NOR R1 <B> <C> NOR <A> <A> R2 POP R2

NAND

Condition	Translation
<B> is the same as <C>	NOT <A> <B>
<A> is different to <C> and <C> is a register	NOT <A> <B> NOT <C> <C> NOR <A> <A> <C> NOT <C> <C> NOT <A> <A>
<A> is different to <B> and <B> is a register	NOT <B> <B> NOT <A> <C> NOR <A> <A> <B> NOT <B> <B> NOT <A> <A>
<A> is different to R1	PSH R1 NOT R1 <B>

	NOT <A> <C> NOR <A> <A> R1 POP R1 NOT <A> <A>
<A> is different to R2	PSH R2 NOT R1 <B> NOT <A> <C> NOR <A> <A> R2 POP R2 NOT <A> <A>

BRL

Condition	Translation
Any	BGE +2 <B> <C> JMP <A>

BRG

Condition	Translation
Any	BGE +2 <C> <B> JMP <A>

BRE

Condition	Translation
Any	BGE +2 <B> <C> JMP +4 BGE +2 <C> <B> JMP +2 JMP <A>

BNE

Condition	Translation
Any	BGE +2 <B> <C> JMP +2 BGE +2 <C> <B> JMP <A>

BOD

Condition	Translation
Any	PSH R1 AND R1 <B> 1 BGE +2 R1 1 JMP +3

	POP R1 JMP <A>

BEV

Condition	Translation
Any	PSH R1 AND R1 <B> 1 BGE +2 R1 1 JMP <A> POP R1

BLE

Condition	Translation
Any	BGE <A> <C> <B>

BRZ

Condition	Translation
Any	BGE +2 <B> 1 JMP <A>

BNZ

Condition	Translation
Any	BGE <A> <B> 1

BRN

Condition	Translation
X is equal to an immediate value which has only the uppermost bit active. (128 in 8 bit)	BGE <A> <B> X

BRP

Condition	Translation
X is equal to an immediate value which has only the uppermost bit active. (128 in 8 bit)	BGE +2 <B> X JMP <A>

PSH

Condition	Translation
Any	INC SP SP STR SP <A>

POP

Condition	Translation
Any	`LOD <A> SP` `DEC SP SP`

CAL

Condition	Translation
Any	`PSH +2` `JMP <A>`

RET

Condition	Translation
A temporary register is required	`POP tempREG` `JMP tempREG`
A temporary RAM location is required	`PSH R1` `INC SP SP` `LOD R1 SP` `STR tempRAM R1` `DEC SP SP` `POP R1` `INC SP SP` `LOD PC tempRAM`
Program must be RUN RAM	`PSH R1` `INC SP SP` `LOD R1 SP` `STR +5 R1` `DEC SP SP` `POP R1` `INC SP SP` `LOD PC +1` `DW 0`

HLT

Condition	Translation
Any	`JMP +0`

CPY

Condition	Translation
A temporary register is required	`LOD tempREG <B>` `STR <A> tempREG`

<A> is different to R1	PSH R1 LOD R1 <B> STR <A> R1 POP R1
<A> is different to R2	PSH R2 LOD R2 <B> STR <A> R2 POP R2

BRC

Condition	Translation
A temporary register is required	ADD tempREG <B> <C> BRL <A> tempREG <B> BRL <A> tempREG <C>
A temporary RAM location is required and <A> is different to R1 and <B> is different to R1 and <C> is different to R1	PSH R1 ADD R1 <B> <C> STR tempRAM <A> BRL +3 R1 <B> BRL +2 R1 <C> STR tempRAM +3 POP R1 LOD PC tempRAM
A temporary RAM location is required and <A> is different to R2 and <B> is different to R2 and <C> is different to R2	PSH R2 ADD R2 <B> <C> STR tempRAM <A> BRL +3 R2 <B> BRL +2 R2 <C> STR tempRAM +3 POP R2 LOD PC tempRAM
A temporary RAM location is required and <A> is different to R3 and <B> is different to R3 and <C> is different to R3	PSH R3 ADD R3 <B> <C> STR tempRAM <A> BRL +3 R3 <B> BRL +2 R3 <C> STR tempRAM +3 POP R3 LOD PC tempRAM
A temporary RAM location is required and <A> is different to R4 and <B> is different to R4 and <C> is different to R4	PSH R4 ADD R4 <B> <C> STR tempRAM <A> BRL +3 R4 <B> BRL +2 R4 <C> STR tempRAM +3

	POP R4 LOD PC tempRAM

BNC

Condition	Translation
A temporary register is required	LOD tempREG <B> STR <A> tempREG
A temporary RAM location is required and <A> is different to R1 and <B> is different to R1 and <C> is different to R1	PSH R1 ADD R1 <B> <C> STR tempRAM +6 BRL +3 R1 <B> BRL +2 R1 <C> STR tempRAM <A> POP R1 LOD PC tempRAM
A temporary RAM location is required and <A> is different to R2 and <B> is different to R2 and <C> is different to R2	PSH R2 ADD R2 <B> <C> STR tempRAM +6 BRL +3 R2 <B> BRL +2 R2 <C> STR tempRAM <A> POP R2 LOD PC tempRAM
A temporary RAM location is required and <A> is different to R3 and <B> is different to R3 and <C> is different to R3	PSH R3 ADD R3 <B> <C> STR tempRAM +6 BRL +3 R3 <B> BRL +2 R3 <C> STR tempRAM <A> POP R3 LOD PC tempRAM
A temporary RAM location is required and <A> is different to R4 and <B> is different to R4 and <C> is different to R4	PSH R4 ADD R4 <B> <C> STR tempRAM +6 BRL +3 R4 <B> BRL +2 R4 <C> STR tempRAM <A> POP R4 LOD PC tempRAM

Complex Instruction Translations

MLT

Condition	Extra Information	Translation
<A> is different to R1 and <A> is different to R2	Shift and Add	PSH R1 PSH R2 MOV R1 <B> MOV R2 <C> MOV <A> R0 BEV +2 R2 ADD <A> <A> R1 RSH R2 R2 LSH R1 R1 BNZ -4 R2 POP R2 POP R1
<A> is different to R3 and <A> is different to R4	Shift and Add	PSH R3 PSH R4 MOV R3 <B> MOV R4 <C> MOV <A> R0 BEV +2 R4 ADD <A> <A> R3 RSH R4 R4 LSH R3 R3 BNZ -4 R4 POP R4 POP R3
<A> is different to R1 and <A> is different to R2	Repeated Addition	PSH R1 PSH R2 MOV R1 <C> MOV R2 <B> MOV <A> R0 BRZ +4 <C> DEC R1 R1 ADD <A> <A> R2 BNZ -2 R1 POP R2 POP R1
<A> is different to R3 and <A> is different to R4	Repeated Addition	PSH R3 PSH R4 MOV R3 <C> MOV R4 <B> MOV <A> R0 BRZ +4 <C> DEC R3 R3

		`` ADD <A> <A> R4 BNZ -2 R3 POP R4 POP R3 ```

DIV

Condition	Extra Information	Translation
<A> is different to R1 and <A> is different to <C>	Repeated Subtraction	``` BRL +9 <B> <C> PSH R1 MOV R1 <B> MOV <A> R0 INC <A> <A> SUB R1 R1 <C> BGE -2 R1 <C> POP R1 JMP +2 MOV <A> R0 ```
<A> is different to R2 and <A> is different to <C>	Repeated Subtraction	``` BRL +9 <B> <C> PSH R2 MOV R2 <B> MOV <A> R0 INC <A> <A> SUB R2 R2 <C> BGE -2 R2 <C> POP R2 JMP +2 MOV <A> R0 ```
<A> is different to R1 and <A> is different to R2	Repeated Subtraction	``` BRL +13 <B> <C> PSH R1 PSH R2 MOV R1 <B> MOV R2 <C> MOV <A> R0 INC <A> <A> SUB R1 R1 R2 BGE -2 R1 R2 POP R2 POP R1 JMP +2 MOV <A> R0 ```
<A> is different to R3 and <A> is different to R4	Repeated Subtraction	``` BRL +13 <B> <C> PSH R3 PSH R4 MOV R3 <B> MOV R4 <C> MOV <A> R0 ```

		INC <A> <A> SUB R3 R3 R4 BGE -2 R3 R4 POP R4 POP R3 JMP +2 MOV <A> R0

MOD

Condition	Extra Information	Translation
<A> is different to <C>	Repeated Subtraction	MOV <A> <B> BRL +3 <A> <C> SUB <A> <A> <C> JMP -2
<A> is different to R1 and <B> is different to R1	Repeated Subtraction	PSH R1 MOV R1 <C> MOV <A> <B> BRL +3 <B> R1 SUB <A> <A> R1 JMP -2 POP R1
<A> is different to R2 and <B> is different to R2	Repeated Subtraction	PSH R2 MOV R2 <C> MOV <A> <B> BRL +3 <A> R2 SUB <A> <A> R2 JMP -2 POP R2
<A> is different to R3 and <B> is different to R3	Repeated Subtraction	PSH R3 MOV R3 <C> MOV <A> <B> BRL +3 <A> R3 SUB <A> <A> R3 JMP -2 POP R3

BSR

Condition	Extra Information	Translation
<A> is different to R1 and <B> is different to R1		PSH R1 MOV R1 <C> MOV <A> <B> BRZ +4 R1 RSH <A> <A> DEC R1 R1

		JMP -3 POP R1
<A> is different to R2 and <B> is different to R2		PSH R2 MOV R2 <C> MOV <A> <B> BRZ +4 R2 RSH <A> <A> DEC R2 R2 JMP -3 POP R2
<A> is different to R3 and <B> is different to R3		PSH R3 MOV R3 <C> MOV <A> <B> BRZ +4 R3 RSH <A> <A> DEC R3 R3 JMP -3 POP R3

BSL

Condition	Extra Information	Translation
<A> is different to R1 and <B> is different to R1		PSH R1 MOV R1 <C> MOV <A> <B> BRZ +4 R1 LSH <A> <A> DEC R1 R1 JMP -3 POP R1
<A> is different to R2 and <B> is different to R2		PSH R2 MOV R2 <C> MOV <A> <B> BRZ +4 R2 LSH <A> <A> DEC R2 R2 JMP -3 POP R2
<A> is different to R3 and <B> is different to R3		PSH R3 MOV R3 <C> MOV <A> <B> BRZ +4 R3 LSH <A> <A> DEC R3 R3 JMP -3 POP R3

SRS

Condition	Extra Information	Translation
X is equal to an immediate value which has only the uppermost bit active.		`BRN +3 <B>` `RSH <A> <B>` `JMP +3` `RSH <A> <B>` `ADD <A> <A> X`

BSS

Condition	Extra Information	Translation
<A> is different to R1 and <B> is different to R1		`PSH R1` `MOV R1 <C>` `MOV <A> <B>` `BRZ +4 R1` `SRS <A> <A>` `DEC R1 R1` `JMP -3` `POP R1`
<A> is different to R2 and <B> is different to R2		`PSH R2` `MOV R2 <C>` `MOV <A> <B>` `BRZ +4 R2` `SRS <A> <A>` `DEC R2 R2` `JMP -3` `POP R2`
<A> is different to R3 and <B> is different to R3		`PSH R3` `MOV R3 <C>` `MOV <A> <B>` `BRZ +4 R3` `SRS <A> <A>` `DEC R3 R3` `JMP -3` `POP R3`

SETE

Condition	Extra Information	Translation
Any		`BRE +3 <B> <C>` `MOV <A> R0` `JMP +2` `IMM <A> 1`

SETNE

Condition	Extra Information	Translation
Any		`BNE +3 <B> <C>` `MOV <A> R0` `JMP +2` `IMM <A> 1`

SETG

Condition	Extra Information	Translation
Any		`BRG +3 <B> <C>` `MOV <A> R0` `JMP +2` `IMM <A> 1`

SETL

Condition	Extra Information	Translation
Any		`BRL +3 <B> <C>` `MOV <A> R0` `JMP +2` `IMM <A> 1`

SETGE

Condition	Extra Information	Translation
Any		`BGE +3 <B> <C>` `MOV <A> R0` `JMP +2` `IMM <A> 1`

SETLE

Condition	Extra Information	Translation
Any		`BLE +3 <B> <C>` `MOV <A> R0` `JMP +2` `IMM <A> 1`

SETC

Condition	Extra Information	Translation
A temporary register is required		`MOV tempREG <B>` `BRG +2 <B> <C>` `MOV tempREG <C>`

		ADD <A> <B> <C> SETL <A> <A> tempREG
<A> is different to R1 and <B> is different to R1 and <C> is different to R1		PSH R1 MOV R1 <B> BRG +2 <B> <C> MOV R1 <C> ADD <A> <B> <C> SETL <A> <A> R1 POP R1
<A> is different to R2 and <B> is different to R2 and <C> is different to R2		PSH R2 MOV R2 <B> BRG +2 <B> <C> MOV R2 <C> ADD <A> <B> <C> SETL <A> <A> R2 POP R2
<A> is different to R3 and <B> is different to R3 and <C> is different to R3		PSH R3 MOV R3 <B> BRG +2 <B> <C> MOV R3 <C> ADD <A> <B> <C> SETL <A> <A> R3 POP R3
<A> is different to R4 and <B> is different to R4 and <C> is different to R4		PSH R4 MOV R4 <B> BRG +2 <B> <C> MOV R4 <C> ADD <A> <B> <C> SETL <A> <A> R4 POP R4

SETNC

Condition	Extra Information	Translation
A temporary register is required		MOV tempREG <B> BRG +2 <B> <C> MOV tempREG <C> ADD <A> <B> <C> SETGE <A> <A> tempREG
<A> is different to R1 and <B> is different to R1 and <C> is different to R1		PSH R1 MOV R1 <B> BRG +2 <B> <C> MOV R1 <C> ADD <A> <B> <C>

		SETGE <A> <A> R1 POP R1
<A> is different to R2 and <B> is different to R2 and <C> is different to R2		PSH R2 MOV R2 <B> BRG +2 <B> <C> MOV R2 <C> ADD <A> <B> <C> SETGE <A> <A> R2 POP R2
<A> is different to R3 and <B> is different to R3 and <C> is different to R3		PSH R3 MOV R3 <B> BRG +2 <B> <C> MOV R3 <C> ADD <A> <B> <C> SETGE <A> <A> R3 POP R3
<A> is different to R4 and <B> is different to R4 and <C> is different to R4		PSH R4 MOV R4 <B> BRG +2 <B> <C> MOV R4 <C> ADD <A> <B> <C> SETGE <A> <A> R4 POP R4

LLOD

Condition	Extra Information	Translation
Any		ADD <A> <B> <C> LOD <A> <A>

LSTR

Condition	Extra Information	Translation
A temporary register is required		ADD tempREG <A> <B> STR <C> tempREG
<C> is different to R1		PSH R1 ADD R1 <A> <B> STR <C> R1 POP R1
<C> is different to R2		PSH R2 ADD R1 <A> <B> STR <C> R2 POP R2

PORTS

There are 64 official ports.
The word length of the value of each port is equal to the word length of the CPU.
Ports can be written to or read from using the I/O instructions as appropriate.

Official ports can be used in URCL programs without having to be defined. They use the definition given here.

Note that the programmer can make up any ports and these do not have to follow the official documentation.

In this case the programmer should define what is meant by each port if it is not obvious. A simple comment in the code is usually fine if it is not too complex.

Type	Port Number	Alias (Port Name)	Full Name	Input Notes/Usage	Output Notes/Usage	Valid Inputs
General	0	%CPUBUS	CPU Bus			Number
	1	%TEXT	Text IO			Number, Character
	2	%NUMB	Numeric IO			Number
	3		Reserved			
	4		Reserved			
	5	%SUPPORTED	Port Supported	Returns 0 if the port does not exist	Sets value to return if it does exist, this can be handled by a compiler if the CPU does not support it	Number, Port Alias
	6	SPECIAL	Special	User Defined	User Defined	
	7	PROFILE	Profile	Tells current Profile	Sets Profile	Number
Graphics	8	X	Display X	Tells display width	Sets X Vertex	Number
	9	Y	Display Y	Tells display height	Sets Y Vertex	Number
	10	COLOR or COLOUR	Colour	Reads colour at x, y into a register	Sets Colour and draws a pixel based on the x, y vertex	Colour
	11	BUFFER	Display Buffer	Reads buffer state	0 writes buffer to display, clears the buffer, and disables it, 1	Number

					enables writing to the buffer	
	12		Reserved			
	13		Reserved			
	14		Reserved			
	15	GSPECIAL	Graphics Special	User Defined	User Defined	
Text	16	ASCII8	8-Bit ASCII	Takes in an 8-Bit Ascii character	Displays an 8-bit ascii character	Number, Character
	17	CHAR5	5-Bit Char	Takes in a 5-bit character	Displays a 5 bit character	Number, Character
	18	CHAR6	6-Bit Char	Takes in a 6-bit character	Displays a 6 bit character	Number, Character
	19	ASCII7	7-Bit ASCII	Takes in a 7-Bit Ascii character	Displays a 7-bit ascii character	Number, Character
	20	UTF8	UTF-8	Takes in a UTF-8 character (1-4 bytes)	Displays a UTF-8 character	Number, Character
	21		Reserved			
	22		Reserved			
	23	TSPECIAL	Text Special	User Defined	User Defined	
Numbers	24	INT	Signed Integer	Takes in a signed integer	Displays a signed integer	Number
	25	UINT	Unsigned Integer	Takes in an unsigned integer	Displays an unsigned integer	Number
	26	BIN	Binary	Takes in a binary number	Displays a binary number	Number
	27	HEX	Hexadecimal	Takes in a hexadecimal number	Displays a hexadecimal number	Number
	28	FLOAT	Floating Point Number	Takes in a floating-point number	Displays a floating-point number	
	29	FIXED	Fixed Point Number	Takes in a fixed-point number	Displays a fixed-point number	

	30		Reserved			
	31	N-SPECIAL	Numbers Special	User Defined	User Defined	
Storage	32	ADDR	Address	Tells address	Sets address	Number
	33	BUS	Bus	Reads the data at the address	Writes data to that address	Number
	34	PAGE	Page	Reads the page number	Sets the page number	Number
	35		Reserved			
	36		Reserved			
	37		Reserved			
	38		Reserved			
	39	SSPECIAL	Storage Special	User Defined	User Defined	
Miscellaneous	40	RNG	RNG Device	Reads a random number	Sets a seed or device specific	Number
	41	NOTE	Note	Reads sound device pitch	Sets sound device pitch	Number, Note
	42	INSTR	Instrument	Reads sound device instrument	Sets sound device instrument	Number, Instrument
	43	NLEG	Note length	Device specific	Sets sound device note length and plays that note (in tenths?)	Number
	44	WAIT	Wait	Returns 1 after the wait period	Sets wait period (in tenths of seconds)	Number
	45	NADDR	Network Address	Reads the current address	Sets the network address	Number
	46	DATA	Network Data	Reads network data	Sends network data	Number
	47	MSPECIAL	Miscellaneous Special	User Defined	User Defined	
User defined	48	UD1	User Defined			

	49	UD2	User Defined			
	50	UD3	User Defined			
	51	UD4	User Defined			
	52	UD5	User Defined			
	53	UD6	User Defined			
	54	UD7	User Defined			
	55	UD8	User Defined			
	56	UD9	User Defined			
	57	UD10	User Defined			
	58	UD11	User Defined			
	59	UD12	User Defined			
	60	UD13	User Defined			
	61	UD14	User Defined			
	62	UD15	User Defined			
	63	UD16	User Defined			

CODE FAULTS

This section contains most common faults and the most likely causes as well as possible solutions.

Faults within the code can be detected using a URCL emulator.

It is important to test for and fix these faults before deploying the code onto a target CPU. The code is unable to detect these errors itself when deployed.

Pre-Runtime Faults

These are faults which can be detected before running the code.

These faults can be detected by static analysis before execution, such as part of a generic URCL code optimiser.

Invalid Number of Operands

Potential Cause	Potential Fix
Too many operands were given for an instruction.	Rewrite the instruction in question, making sure that the number of operands match the expected value given in the Instruction section.
Too few operands were given for an instruction.	Rewrite the instruction in question, making sure that the number of operands match the expected value given in the Instruction section.

Invalid Operand Types

Potential Cause	Potential Fix
The types of operands that were given for an instruction are not in the valid operand types which are listed in the Instruction section.	Rewrite the instruction in question, making sure that the types are valid as per the operand type tables provided in the Instruction section.

Unrecognised Identifier

Potential Cause	Potential Fix
Invalid instruction name.	Rewrite the instruction in question, making sure that the name is spelt exactly as given in the Instruction section.
Invalid operand name.	Rewrite the instruction in question, making sure that the operands have the correct prefix if required. These are specified in the Prefix section.
Comment that has not be marked as a comment.	Add `//` or `/*` and `*/` as appropriate to the comment.
Invalid header name.	Rewrite the header in question, making sure that the name is spelt exactly as given in the Header section.
Label without prefix.	Add the prefix `.` on to the label in question.

Unsupported Number of Registers

Potential Cause	Potential Fix
The value of MINREG is higher than 2 to the power of CPU Word Length. (256 on an 8 bit CPU)	Lower the number of registers the program requires by using the Heap and Stack or increase the word length. This may require rewriting large parts of the program.
The program uses a register which is larger than the value set in the MINREG header.	Increase the value of the MINREG header to match the minimum required by the program.

Unsupported Heap Size

Potential Cause	Potential Fix
The value of MINHEAP is higher than 2 to the power of CPU Word Length. (256 on an 8 bit CPU)	Lower the number of words of heap the program requires or increase the word length. This may require rewriting large parts of the program.

Unsupported Stack Size

Potential Cause	Potential Fix
The value of MINSTACK is higher than 2 to the power of CPU Word Length. (256 on an 8 bit CPU)	Lower the number of stack values the program requires or increase the word length. This may require rewriting large parts of the program.

Invalid Label Name

Potential Cause	Potential Fix
Label name contains invalid characters.	Rename the label in question, making sure that the label name consists of only letters, numbers, and underscore.

Duplicate Label Definition

Potential Cause	Potential Fix
The same label is defined multiple times.	Remove or rename one of the label definitions. Each label should only be defined once.

Runtime Faults

These are faults which can only be detected by running the code.

Non-Instruction Execution

Potential Cause	Potential Fix
Branched to a value which does not point towards an instruction.	Rewrite the code in question to make sure the branch address always points to a valid instruction.
A non-instruction pointer is loaded into the program counter.	Rewrite the code in question to make sure the load target always points to a valid instruction.
The program fails to branch around DW values.	Add JMP instructions to branch around the DW value or move the DW values elsewhere in the program.

Stack Underflow

Potential Cause	Potential Fix
Popping from the stack while it is empty.	Rewrite code to ensure the POP instruction is only called when there are values on the stack.
The stack pointer lost sync with the actual size of the stack.	Avoid editing the stack pointer directly in the code as this can cause it to be desynced if not handled properly.

Stack Overflow

Potential Cause	Potential Fix
The stack overlaps the heap.	Rewrite code to ensure the heap and stack do not get too big or increase the values of the MINHEAP and MINSTACK headers.

The stack pointer lost sync with the actual size of the stack.	Avoid editing the stack pointer directly in the code as this can cause it to be desynced if not handled properly.

Invalid RAM Location

Potential Cause	Potential Fix
Attempted to write to a RAM location which does not exist.	Rewrite code to ensure that invalid RAM addresses are not written to or increase the value of the MINHEAP header to include that RAM address.
Attempted to read from a RAM location which does not exist.	Rewrite code to ensure that invalid RAM addresses are not read from or increase the value of the MINHEAP header to include that RAM address.

INTERPRETING URCL

Due to how simple it is to translate from URCL to a target CPU's assembly, it is possible to store the raw URCL code on the target CPU and get the CPU to interpret and translate the code itself during execution.

This is a lot slower than running compiled code but can offer some advantages, such as being able to modify the instructions more easily during execution.

Bitwise Representation

There are several different ways to represent URCL code, some of which are given here.

Each operand requires one full word length, and the operand type must be stored separately.

There are only two types that an operand can be, and the exact pair of types depend on the instruction. This means that the operand type can be represented using a single bit for each operand.

There are 59 URCL instructions. This means that 6 bits are required to be able to represent them all.

> *To represent just the Core instructions, only 3 bits are required.*

This means that 1 to 5 words are required to represent each instruction.

The first 1 or 2 words represent the instruction and the operand types. Then the remaining 0 to 3 words represent the number of operands.

On an 8 bit CPU this could look like:

8 Bit (All instructions + Separate Types)	Bitwise Representation	Key
First Word	AAAAAA XX	A = Instruction X = Unused
Second Word	BCD XXXXX	B = First operand type C = Second operand type D = Third operand type
Third Word	EEEEEEEE	E = First Operand
Fourth Word	FFFFFFFF	F = Second Operand
Fifth Word	GGGGGGGG	G = Third Operand

Each letter represents 1 bit within each word.

A 4 Byte version can be done with 8 bits which has fewer unused bits, but only if the number of instructions is cut down to 32. This means that all the Complex and some of Basic instructions must be removed to make it work with only 5 bits for the instruction:

8 Bit (Cut down to 4 Bytes + Separate Types)	Bitwise Representation	Key
First Word	AAAAA BCD	A = Instruction B = First operand type C = Second operand type D = Third operand type
Second Word	EEEEEEEE	E = First Operand
Third Word	FFFFFFFF	F = Second Operand
Fourth Word	GGGGGGGG	G = Third Operand

Since there are only 159 possible combinations of instructions and operand types, the operand types can be combined with the instructions to fit in one byte. This will, however, make it harder to interpret the instruction.

So, on an 8 bit CPU it could look like:

8 Bit (Cut down to 4 Bytes + Combined Types)	Bitwise Representation	Key
First Word	AAAAAAAA	A = Instruction and Types
Second Word	BBBBBBBB	B = First Operand
Third Word	CCCCCCCC	C = Second Operand
Fourth Word	DDDDDDDD	D = Third Operand

On a 16 bit CPU it could look like:

16 Bit (All instructions)	Bitwise Representation	Key
First Word	AAAAAA BCD XXXXXXX	A = Instruction B = First operand type C = Second operand type D = Third operand type X = Unused
Second Word	EEEEEEEEEEEEEEEE	E = First Operand
Third Word	FFFFFFFFFFFFFFFF	F = Second Operand
Fourth Word	GGGGGGGGGGGGGGGG	G = Third Operand

Lastly, on a 4 bit CPU if the only the Core and a couple of the Basic instructions were kept, it could look like:

4 Bit (Cut down to 4 bits)	Bitwise Representation	Key
First Word	AAAA	A = Instruction
Second Word	BCD X	B = First operand type C = Second operand type D = Third operand type X = Unused
Third Word	EEEE	E = First Operand
Fourth Word	FFFF	F = Second Operand
Fifth Word	GGGG	G = Third Operand

EXAMPLE PROGRAMS

Simple Fibonacci

```
BITS == 8
MINREG 2
MINHEAP 0
MINSTACK 0
RUN ROM

IMM R1 0
IMM R2 1
.loop
    ADD R1 R1 R2
    ADD R2 R1 R2
    JMP .loop
```

This program has no escape condition so it will keep going forever.

This program does not output the answers.

FizzBuzz

```
BITS == 8
MINREG 4
MINHEAP 0
MINSTACK 0
RUN ROM
```

```
.setup
    IMM R1 0        // current value = 0
    IMM R2 3        // fizz counter = 3
    IMM R3 5        // buzz counter = 5

.loop
    OUT %TEXT '#'   // draw a new line character to the char display
    INC R1 R1
    IMM R4 0        // R4 is used to tell if fizz activated
    DEC R2 R2
    BRZ .fizz R2    // branch to .fizz if fizz counter == 0
.return
    DEC R3 R3
    BRZ .buzz R3    // branch to .buzz if buzz counter == 0
    BNZ .loop R4    // branch to .loop if R4 != 0
    OUT %TEXT R1    // draw current value to the char display
    JMP .loop

.fizz
    IMM R4 1        // R4 = 1
    OUT %TEXT 'F'   // draw "FIZZ" on the char display
    OUT %TEXT 'I'
    OUT %TEXT 'Z'
    OUT %TEXT 'Z'
    IMM R2 3        // fizz counter = 3
    JMP .return

.buzz
    OUT %TEXT 'B'   // draw "BUZZ" on the char display
    OUT %TEXT 'U'
    OUT %TEXT 'Z'
    OUT %TEXT 'Z'
    IMM R3 5        // buzz counter = 5
    JMP .loop
```

This program starts at 1 and it increments this value once per loop. It prints out "FIZZ" if the value is divisible by 3, "BUZZ" if the value is divisible by 5, "FIZZBUZZ" if the value is divisible by both 3 and 5 or the original value if not divisible by 3 or 5.

This program has no escape condition so it will keep going forever.

Bubble Sort

```
BITS == 8
MINREG 5
MINHEAP 5
MINSTACK 0
RUN ROM

.setup
    MOV R2 R0                   // R2 = list pointer
    .rng
        IN R1 %RNG              // R1 = random number
        STR R2 R1
        INC R2 R2
        OUT %TEXT '#'
        OUT %TEXT R1
        BNE .rng R2 5           // stop when 5 numbers have been generated
.main
    MOV R5 R0                   // R5 = switch check
    DEC R3 R0                   // R3 = low pointer
    MOV R4 R0                   // R4 = high pointer
    .loop
        INC R3 R3               // R3 += 1
        INC R4 R4               // R4 += 1
        LOD R1 R3               // R1 = low value
        LOD R2 R4               // R2 = high value
        BRL .switch R2 R1       // go to .switch if high less than low
        BNE .loop R4 4          // branch to .loop if not at end of list
        BRE .main R5 1          // loop again if any switches occurred
    .out
        MOV R1 R0               // R1 = pointer for printing outputs
        .outLoop
            LOD R2 R1
            OUT %TEXT '#'
            OUT %TEXT R2
            INC R1 R1
            BNE .outLoop R1 5   // loop until all 5 values are printed
        HLT
    .switch
        IMM R5 1                // set switch check
        STR R3 R2
```

```
        STR R4 R1
        BNE .loop R4 4              // branch to .loop if not at end of list
        JMP .main                   // loop again
```

This program generates a list of 5 random numbers and prints them. Then it sorts the numbers using a bubble sort algorithm, afterwards it prints the sorted list.

More example programs can be found in the URCL Discord which is linked in the Links section.

ACKNOWLEDGEMENTS

URCL would not have been possible without all of the URCL community contributing towards it. The community has contributed by voting on every part of the language, testing the language on real CPUs, creating tools such as emulators and compilers, writing many URCL programs and more.

Biggest Contributors

Name (If they wish to give it)	Discord Username / Minecraft Username	Contributions
Ben Aitken	Mod Punchtree / ModPunchtree	• One of the original founders of URCL • Created the Flagless fork of URCL • Managed the Google Sheet documentation for both Main URCL and Flagless • Ran Flagless URCL on the MPU6 • Hosted polls • Made several emulators • Made a B to URCL compiler • Made a generic URCL code optimiser • Made a Discord bot so the emulator and compiler are easily accessible • Created and maintained this formal URCL documentation
	Haku /	• Created a tool to translate URCL into target CPU's assembly
	Kuggo / Kuggo	• One of the original founders of URCL • Began working on math and string libraries • Hosted a poll
	Lucida Dragon /	• Created FlapStacks and URCL.NET • Created the URCL highlight extension for VSCode • Created a Discord bot
	URCL's Official Gay Lady / IAmLesbian	• One of the original founders of URCL • Managed the URCL discord

		• Hosted polls • Created the URCL logo • Kept Tuke under control
	Verlio_H /	• One of the original founders of URCL • Created URCL OS • Made the current ports documentation • Created the complex numbers library
	Tuke /	• Created URCL OS • Made the current ports documentation • Created a compiler • Beans
	sammyuri / sammyuri	• Ran URCL on several of their CPUs
	GLS / GamingLiamStudios	• Made a C to URCL compiler
	Qwerasd /	• Made a URCL emulator
	Tape / TapeDispenser69	• Made a URCL emulator in scratch • worked on an OS • Whined about the existence of Flagless URCL