# Generic decompiler documentation

Michael Madsen

December 26, 2014

## Contents

## 1 Overview

The decompilation process consists of a few different steps:

- Disassembly
- Code flow analysis
- Code generation

Of these steps, the code flow analysis is engine-independent, while disassembly and code generation require engine-specific code.

### 1.1 Reading guide

Names used in code are written in a `monospaced typewriter font`.

Actual code snippets have basic syntax highlighting on a light gray background, and lines are numbered, like below:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello world!");
    return 0;
}
```

In this document, the terms *control flow analysis* and *code flow analysis* are used interchangably.

### 1.2 Limitations

The decompiler is targeted for stack-based instruction sets, and may contain assumptions to that effect. If you want to add an engine which does not use a stack-based instruction set, parts of this document may not apply directly, and additional work to the generic parts may be necessary.

# 2 Engine

The `Engine` class represent a single engine. It works as a factory for the engine-specific classes required for each step of the process.

As a minimum, engines must provide a disassembler and a code generator. All other steps are optional, but you can implement them for additional processing.

If you need to store metadata about the script, you can add the necessary fields to your engine class and store the information there, as the same instance will be used throughout the decompilation process.

## 2.1 Adding a new engine

In order to make the decompiler use the code you write to decompile code for some engine, it must be registered in the program. To do so, include the header file for your engine in `decompiler.cpp`, and use the `ENGINE` macro defined there to register your engine with the program.

This macro takes 3 parameters: the engine ID, a description of the engine, and the name of the `Engine` subclass used to create the classes used for the various steps of the process. The ID is entered by the user to signify the engine where the script originates from, and the description is a descriptive text which will be shown when the user requests a list of the supported engines. In general, you should place the files for your engine in a folder with the same name as the engine ID you use.

The methods you need to implement in your `Engine` subclass are:

- `getDisassembler`, which takes a reference to the instruction vector to use for storage and creates a disassembler object and returns it. For more on disassemblers, see Section 3 on page 6.

- `getCodeGenerator`, which takes a reference to the `std::ostream` to output the code to and creates a code generator object and returns it. For more on code generators, see Section 5 on page 19.

Additional methods you can override are:

- `supportsCodeFlow` and `supportsCodeGen`, which can be used to stop the decompiler from going any further after disassembly or code flow analysis, respectively. This is helpful when working on a brand new engine, so you can take one step at a time without having to remember to use the right command-line switch. If you do not override these methods, the decompiler will go through all steps.

- `detectMoreFuncs` allows you to tell the control flow analysis to automatically detect functions based on reachability. See Section 4.1 on page 15 for details. By default, this is turned off; engines must opt-in to this feature.

- `postCFG`, which is a post-processing step called after control flow analysis. If you override `detectMoreFuncs` to return true, you must also override this function to process any newly found functions. A default implementation which does nothing is already provided in case you do not need to do any post-processing.

- `usePureGrouping` is used to toggle "pure" grouping. In pure grouping, stack levels are ignored during group generation in the control flow analysis. By default, this is turned off. See Section 4.2 on page 16 for details.

Additionally, if your engine is not stack-based, you may not wish to see the stack effect when reviewing the disassembly or code flow graph. You can disable this by calling `setOutputStackEffect(false)` from e.g. your Engine constructor. The method is defined in instruction.h, which you will have to include.

It is important to realize that you do not necessarily need to implement a completely new code generator and disassembler for every engine; for variations on the same engine, you can reuse the existing classes and simply send in any extra information required. In particular, code generators are likely to be reusable without change for different versions of the same engine – e.g., the Kyra2 code generator will likely work for all Kyra games.

For this purpose, the user may optionally specify an *engine variant*, which is a string that will be passed to your Engine. If you make use of this feature, you should also override `getVariants` to specify which variants your engine supports. This list will be displayed to the user if they specify an engine while using the `-h` option.

Note that the variant sent into your engine is not validated against your list of supported variants. This keeps the variant logic flexible, and allows you to implement your own fallback logic for unknown variant strings.

If you can auto-detect the variant from the script file, you should prefer this approach over asking the user to specify the variant.

## 2.2 Functions

Some engines allow multiple functions in a single script file. Each function must be analyzed separately, but in order to do that, it is of course necessary to know where the functions start and end, and when it is time to actually generate some code, you will want to know a bit about the function as well.

This information is stored in the engine, as a `std::map` of `Functions`, in the field `_functions`.

```
struct Function {
public:
    ConstInstIterator _startIt;
    ConstInstIterator _endIt;
    std::string _name;
    GraphVertex _v;
    uint32 _args;
    bool _retVal;
    std::string _metadata;
};
```

Each member of this struct has a specific purpose:

- `_startIt` is a const iterator pointing to the first instruction in the function, i.e. the entry point.
- `_endIt` is a const iterator pointing to the instruction immediately after the last instruction, similar to `end()` on STL collections.
- `_name` is the name of the functions.
- `_v` is the GraphVertex containing the entry point. This will be automatically assigned in the control flow analysis.
- `_args` is the number of arguments for the function. This is present as a convenience; usually, you will not know the names of the arguments in the function, so you can store the number of them here and use this during code generation to generate a method signature containing some default parameter names.
- `_retVal` should be true if your method returns a value, and false if it does not. Again, this is for your own convenience, to make it easier to handle calls.
- `_metadata` contains metadata about the function, so you know how to handle the arguments when the function is being called. It is up to you how you want to use this.

When you add a function to the map, you must use the address of the first instruction as the key.

Sometimes, you do not know where all of the functions begin or end. In that case, the control flow analysis can analyze the code for you and automatically detect missing functions or unknown end points. See Section 4.1 on page 15 for details.

## 3 Disassembler

The purpose of the disassembler is to read instructions from a script file and convert them to a common, machine-readable form for further analysis.

## 3.1 Instructions

Instructions are represented using a type hierarchy, with the `Instruction` struct as the base type.

```
struct Instruction : public RefCounted {
    uint32 _opcode;
    uint32 _address;
    int16 _stackChange;
    std::string _name;
    std::vector<ValuePtr> _params;
    std::string _codeGenData;

    friend std::ostream &operator<<(std::ostream &output, const Instruction *inst);
    virtual std::ostream &print(std::ostream &output) const;
    virtual bool isJump() const;
    virtual bool isCondJump() const;
    virtual bool isUncondJump() const;
    virtual bool isStackOp() const;
    virtual bool isFuncCall() const;
    virtual bool isReturn() const;
    virtual bool isKernelCall() const;
    virtual bool isLoad() const;
    virtual bool isStore() const;
    virtual uint32 getDestAddress() const;
    virtual void processInst(ValueStack &stack, Engine *engine, CodeGenerator *codeGen)
        = 0;
};
```

Each member of this struct has a specific purpose:

- `_opcode` is used to store the numeric opcode associated with the instruction. This is not used by the decompiler itself, but is for your reference during later parts of the decompilation process. Note that this field is declared as a 32-bit integer; if you need more than 4 bytes for your opcodes, you will need to figure out which bytes you want to store if you want to use this field.

- `_address` stores the absolute memory address where this instruction would be loaded into memory.

- `_stackChange` stores the net change of executing this instruction - for example, if the instruction pushes a byte on to the stack, this should be set to 1. This is used to determine when each statement ends. The count can be in any unit you wish - bytes, words, bits - as long as the same unit is used for all instructions. This means that if your stack only works with 16-bit elements, pushing an 8-bit value and pushing a 16-bit value should have the same net effect on the stack.

- `_name` contains the name of the instruction. This is mainly for use during code generation.

- `_params` contains the parameters given to the instruction - for example, if you have the instruction `PUSH 1`, there would be one parameter, with the value of 1. See Section 3.3 on the next page for details on the Value type.
- `_codeGenData` stores metadata to be used during code generation. For details, see Section 5 on page 19.

If some instructions do not have a fixed effect on the stack–that is, the instruction name alone does not determine the effect on the stack–set the field to some easily recognizable value when doing the disassembly. You will, however, have to determine the exact stack effect after disassembling the script, as the code flow analysis depends on this information to be accurate.

## 3.2 Instruction types

As mentioned previously, the different instructions are represented using a type hierarchy. This allows you to independently specify how each kind of instruction should be handled, while abstracting away the engine-specific information to allow for generic analysis.

This is particularly important during code flow analysis; since this part is completely engine-independent, the analysis must have some way of distinguishing the different types of instructions. For that purpose, a number of `is*` methods are defined which specify whether the instruction satisfies some specific purpose.

Each of the predefined instruction types have a class associated with it to make it simpler to add functionality to a specific class of instructions, as specified in Table 3.1 on the next page.

For some of these, an extra type exists which contains a default implementation of `processInst`, assuming a sensible default implementation exists. The default implementations are found in `BinaryOpStackInstruction`, `BoolNegateStackInstruction`, `DupStackInstruction`, `KernelCallStackInstruction`, `ReturnInstruction`, `UnarayOpPrefixStackInstruction`, `UnaryOpPostfixStackInstruction`, `UncondJumpInstruction`. Most of these are targeted at stack-based engines, but if your engine doesn't work with these, you can always create your own class with your own implementation of `processInst`.

`getDestAddress` must be implemented on jump instructions to allow the generic code to find the target of a jump. You must create subclassses for your jump instructions which override this method.

Most of the types are self-explanatory, with the possible exception of `KernelCallInstruction`. `KernelCallInstruction` should be used for "magic functions"–opcodes that perform some function specific to the engine, like playing a sound, drawing a graphic, or saving the game.

In a few cases, you may not know which instruction type is correct. For example, in Kyra, the same opcode is used for unconditional jumps and

| **Base class** | **Purpose** |
|---|---|
| `BinaryOpInstruction` | Binary operations (+, *, ==, etc.) |
| `BoolNegateInstruction` | Boolean negation |
| `CallInstruction` | Script function call |
| `CondJumpInstruction` | Conditional jumps |
| `DupInstruction` | Duplicate stack entry |
| `UncondJumpInstruction` | Unconditional jumps |
| `KernelCallInstruction` | Kernel function call |
| `LoadInstruction` | Load from memory |
| `ReturnInstruction` | Function return |
| `StackInstruction` | Stack allocation or deallocation |
| `StoreInstruction` | Store to memory |
| `UnaryOpPrefixInstruction` | Unary operation, prefixed operator |
| `UnaryOpPostfixInstruction` | Unary operation, postfixed operator |

Table 3.1: Predefined instruction types

script function calls, but the correct type depends on other instructions. You can handle this is by creating a new Instruction type which can work as both, depending on a flag you declare in your type, and set once you can correctly determine the type. Note that since `InstPtr` is not a raw pointer, you must first convert it to one before you can cast it to the pointer type of your choice. This can be done by calling `inst.get()`, where `inst` is your `InstPtr`.

When at all possible, you should inherit from one of the more specific types, rather than inheriting directly from `Instruction`.

## 3.3 Parameters and values

Instruction parameters are stored using a hierarchy of `Value` types. Several types are predefined in `value.h`, and you can declare new types if you need to (e.g. a list of values).

`Value` types are also used during code generation, so you can reuse your parameter values directly.

All `Value` types must define a `print` function which prints themselves to a `std::ostream`. This is used not only for code generation, but also for disassembly and control flow output.

For direct values, you should also override the dup function to create a copy of your class. The default implementation is tailored for values that represent expressions, and will therefore output an assignment to show that the result of an expression is being duplicated.

For more details, see Section 5.3 on page 19, where Values are discussed wrt. code generation, and a list of predefined value types is given.

## 3.4 The Disassembler class

All disassemblers must inherit, directly or indirectly, from the Disassembler class. This is an abstract class providing an interface for disassemblers.

```
class Disassembler {
protected:
    Common::File _f;
    InstVec &_insts;
    uint32 _addressBase;

    virtual void doDisassemble() throw(std::exception) = 0;
    virtual void doDumpDisassembly(std::ostream &output);

public:
    Disassembler(InstVec &insts);
    virtual ~Disassembler() {}

    void open(const char *filename);
    void disassemble();
    void dumpDisassembly(std::ostream &output);
};
```

_f represents the file you will be reading from. The file is opened using the open function.

_insts is a reference to an std::vector storing the instructions, passed in via the constructor. Whenever you have read an instruction fully, add it here.

_addressBase is provided as a convenience if your engine does not consider the first instruction to be located at address 0. Assign the expected base address to this field, and make sure that the addresses you assign to the instructions are relative to this base address. This is mainly useful if your engine supports jumps or other references to absolute addresses in the script; if only relative addresses are used, the base address will not be relevant.

doDisassemble is the method used to perform the actual disassembly, so this method must be implemented by all disassemblers.

disassemble simply calls the doDisassemble method to perform the disassembly. The result is cached, so if this method is called twice, it won't perform disassembly again.

Finally, dumpDisassembly is used to output the instructions in a human-readable format to a file or stdout, performing a disassembly first if required, and then calls `doDumpDisassembly` to perform the actual output. `doDumpDisassembly` simply outputs each instruction in turn, using the printing function associated with each instruction. If you want to customize the way instructions are output, you should ideally create new Instruction subclasses and override their printing function, as the same format is used when dumping a code flow graph, but if you just want to prepend or append some additional information to the dump, you can override this method to do so.

## 3.5 The SimpleDisassembler class

To simplify the development of disassemblers, another base class is provided for instruction sets where instructions are of the format `opcode [params]`, with opcode and parameters stored in distinct bytes.

`SimpleDisassembler` defines a number of macros which you can use for writing your disassembler, and provides a framework for reading instruction parameters.

Following is a guide on how to implement a disassembler using this class as its base class. The instruction set used for this example is described in Table 3.2. While not a very useful instruction set, it covers many different aspects.

| Instruction | Parameters | Description |
|---|---|---|
| `PUSH` (0x00) | uint8 | Pushes byte onto the stack. |
| `POP` (0x01) | | Pops a byte from the stack. |
| `PUSH2` (0x02) | int16 | Pushes two bytes onto the stack. |
| `POP2` (0x03) | | Pops two bytes from the stack. |
| `PRINT` (0x80) | C string | Prints string to standard output. |
| `HALT` (0xFF 0x00) | | Stops the machine. |

Table 3.2: Instruction set used in the SimpleDisassembler example.

For the purpose of this example, our instruction set will use little-endian values, and uses byte elements for the stack (so `POP` changes the stack pointer by 1 and `POP2` changes it by 2).

### 3.5.1 Opcode recognition

The first thing to do in the `doDisassemble` method is to read past any header which may be present in your script file. We will assume that our bytecode files do not have a header.

You must place your opcodes between two macros, `START_OPCODES` and `END_OPCODES`. These two macros define the looping required to read one byte at a time.

```
START_OPCODES;
END_OPCODES;
```

To define an opcode, use the OPCODE macro. This macro takes 5 parameters: the opcode value, the name of the instruction, the name of the Instruction type to use, the net effect on the stack, and a string describing the parameters that are part of the instruction. We will start by implementing the POP and POP2 opcodes:

```
START_OPCODES;
    OPCODE(0x01, "POP", MyStackInstruction, −1, "");
    OPCODE(0x03, "POP2", MyStackInstruction, −2, "");
END_OPCODES;
```

The OPCODE macro automatically stores the full opcode in the _opcode field of the generated Instruction.

### 3.5.2 Parameter reading

PUSH, PUSH2 and PRINT all take parameters as part of the instruction. To read these, you must specify them as part of the parameter string, using one character per parameter. The types understood by default are specified in Table 3.3.

| Character | Type |
|---|---|
| b | Signed 8-bit integer. |
| B | Unsigned 8-bit integer. |
| s | Signed 16-bit byte, little-endian. |
| S | Signed 16-bit byte, big-endian. |
| w | Unsigned 16-bit byte, little-endian. |
| W | Unsigned 16-bit byte, big-endian. |
| i | Signed 32-bit byte, little-endian. |
| I | Signed 32-bit byte, big-endian. |
| d | Unsigned 32-bit byte, little-endian. |
| D | Unsigned 32-bit byte, big-endian. |

Table 3.3: Type specifications recognized by SimpleDisassembler.

To help you remember these meanings, little-endian values are encoded using lower case ("small letters", i.e. little), while big-endian values are encoded using upper case ("big" letters). The exception here is a single byte, since endianness has no effect for individual bytes. Here, the mnemonic is that an unsigned byte ("B") has a larger maximum value. For the other letters, "s" was used because it is the first letter in "short", which is usually a 16-bit signed value in C. Similarly, "i" is short for "int". "w" and "d" come from the terms "word" and "dword", which are terms for 16-bit and 32-bit unsigned types on the x86 platform.

Note that strings are not supported by default. To add reading of a string type, you can override the readParameter function to add your own types:

```
    switch (type) {
    case 'c': //Character string
        {
        byte cmd;
        bool inStr = false;
        std::stringstream s;
        while ((cmd = _f.readByte()) != 0) {
            s << cmd;
            _address}};
        }
        s << '"';
        p->_type = kStringParamType;
        p->_value = s.str();
        }
        break;
    default: //Defer handling to parent implementation
        SimpleDisassembler::readParameter(p, type);
        break;
    }
```

Note that you will have to increment the `_address` variable manually when you read a byte. This variable is used to determine the address of the instruction, and must be kept in sync with your progress reading the file.

Now, we can add all three opcodes to the list:

```
START_OPCODES;
    OPCODE(0x00, "PUSH", MyStackInstruction, 1, "B");
    OPCODE(0x01, "POP", MyStackInstruction, -1, "");
    OPCODE(0x02, "PUSH", MyStackInstruction, 1, "w");
    OPCODE(0x03, "POP2", MyStackInstruction, -2, "");
    OPCODE(0x80, "PRINT", KernelCallStackInstruction, 0, "c");
END_OPCODES;
```

### 3.5.3 Multi-byte opcodes

There is only one opcode left to add, `HALT`. This one is a bit trickier, because it uses multiple bytes for the opcode - and the `OPCODE` macro only works for one byte at a time.

To solve this, you can define *subopcodes*. By defining 0xFF as the start of a multi-byte opcode, we can then specify 0x00 as representing a `HALT` instruction when it follows 0xFF.

Defining 0xFF is easily done using the `START_SUBOPCODE` macro. After that, specify the opcodes for this following byte, and finish the subopcode declarations with `END_SUBOPCODE`.

```
START_OPCODES;
    OPCODE(0x00, "PUSH", MyStackInstruction, 1, "B");
    OPCODE(0x01, "POP", MyStackInstruction, -1, "");
    OPCODE(0x02, "PUSH", MyStackInstruction, 1, "w");
    OPCODE(0x03, "POP2", MyStackInstruction, -2, "");
    OPCODE(0x80, "PRINT", KernelCallStackInstruction, 0, "c");
    START_SUBOPCODE(0xFF);
        OPCODE(0x00, "HALT", KernelCallStackInstruction, 0, "");
    END_SUBOPCODE;
END_OPCODES;
```

Subopcodes can be nested if the instruction set requires it. For subopcodes, the `_opcode` field stores the bytes in the order they appear in the file - i.e., the HALT instruction would have the opcode value 0xFF00. If the opcodes are longer than 4 bytes, only the last 4 bytes will be stored.

If all opcodes in a group of subopcodes share a prefix, you can use the `START_SUBOPCODE_WITH_PREFIX` macro instead of `START_SUBOPCODE`. This macro takes an additional string parameter containing the full prefix to use for the opcodes associated with this subopcode. The prefix is not propagated if you nest subopcodes, only the nearest prefix is used.

### 3.5.4 Code generation metadata

For each opcode, you will need to replicate its semantics during code generation. To assist you in generalizing your code, you can use the `OPCODE_MD` macro to add metadata to the instruction, which is then available during code generation.

For example, if you have an opcode for addition, you can store the addition operator as a string in the metadata field, and have that put to use during code generation to avoid having to check the opcode for each instruction of that type.

The arguments for the `OPCODE_MD` are the same as those for `OPCODE`, but with an extra parameter at the end for the metadata.

```
START_OPCODES;
    OPCODE_MD(0x14, "add", BinaryOpStackInstruction, −1, "", "+");
END_OPCODES;
```

For details, see Section 5 on page 19.

### 3.5.5 Advanced opcode handling

If you have one or two opcodes that do not quite fit into the framework provided, you can define your own specialized handling for these opcodes.

Instead of using the `OPCODE` macro, put your code between `OPCODE_BASE` and `OPCODE_END`. For example, if your opcode has the value 0x40, you would use this:

```
OPCODE_BASE(0x40);
    //Your code here
OPCODE_END;
```

`OPCODE_BASE` automatically keeps track of the current opcode value. You can access `full_opcode` to get the current full opcode. Alternatively, you can use the `OPCODE_BODY` macro to use the standard behavior for opcodes, and then follow that with the additional code you want. The `OPCODE_BODY` macro takes the same arguments as the `OPCODE_MD` macro.

For your convenience, a few additional macros are available: `ADD_INST`, which adds an empty instruction of the provided type to the vector, and `LAST_INST` which retrieves the last instruction in the vector. Additionally, you can use `INC_ADDR` as a shorthand for incrementing the address variable by 1, but note that you should *not* increment the address for the opcode itself - this is handled by the other macros.

## 4 Control flow analysis

The control flow analysis is a multi-step process:

- Create a graph with one instruction per vertex, and edges going from instructions to their possible successors
- Do a depth-first search to determine the expected stack level at each vertex
- Optionally detect functions
- Merge vertices to form groups
- Perform analysis on vertices

Calls to in-script functions are not represented with edges in the graph. This is done to keep functions separate from one another, so if your engine uses a jump as part of calling functions, you need to make sure you represent the jump using a `CallInstruction` instead of a `JumpInstruction`.

The first step is handled in the constructor, while the next three steps are handled by the `createGroups()` method. The last step is handled by the `analyze` method.

### 4.1 Function detection

Prior to grouping, the control flow can be used to detect new functions. This detection is automatically activated if the Engine method `detectMoreFuncs` returns true.

When function detection is enabled, unreachable blocks of code will be treated as functions, unless the presumed entry point is located within the range of another function. The end point of the function will then be the last instruction reachable from the entry point.

Functions detected this way will be given the name `auto_`. You can use this as a prefix to the actual name to signify that the function may not actually be a function, or you can ignore it and just replace it with the name you would normally use.

You can also have the function detection determine the end point of an existing function. To do so, `_endIt` must be the same as `_startIt` for that function. In this case, only the end point will be changed within the function; the name will stay the same.

Note that the control flow analysis has no way of determining an appropriate name, number of arguments, return value, or metadata. You will have to fill that in yourself using `postCFG` in the engine.

If this step is not enabled, and no functions have been defined before the control flow analysis is started, there will still be added a single function covering the entire script. This is done to avoid having a special case in the code, and it will not affect the output of your script in any way.

## 4.2 Group generation

Groups are initially formed according to these rules:

- If the next instruction is a jump or a return, end the group here.
- If the next instruction has multiple predecessors, end the group here.
- If the current instruction brings the stack to a lower level than the start of the current group, make the new level the expected stack level (to support clean-up after control structures).
- If the current instruction brings the stack level to the same as the start of the current group, and at least one instruction with a non-negative stack effect exists in the group, end the group here.

The final two rules are based on a property of stack-based engines: when the stack becomes balanced, as defined by these two rules, this indicates that the current group corresponds to a single line of code. This property of the groups can be helpful when performing code generation, since it adds some context – e.g., conditions are placed in a group of their own.

However, this only works for stack-based engines; for a non-stack-based engine, each instruction ends up in a group of its own, which is particularly bad for visualization of the code flow, since dot tends to choke on the large amount of vertices resulting from this. For this reason, engines are able to request *pure grouping* via the virtual `usePureGrouping` method, mentioned in Section 2.1 on page 4. In pure grouping, only the first two rules are applied, creating the minimum number of groups for any grouping algorithm.

Prior to applying these rules, a depth-first search is performed to calculate the expected stack level at each instruction. If multiple paths are found to the same instruction, a warning will be output if the expected stack level from each path differs.

Unreachable code will not be processed during the group generation, but will remain as individual instructions.

## 4.3 Short-circuit detection

*NOTE: This feature is currently disabled.*

As part of the group generation, the decompiler can combine multiple, consecutive groups if it detects them as being part of a single condition check that are merely split up due to short-circuiting.

The rules used to detect if two consecutive groups $A$ and $B$ are oart of the same short-circuited check are as follows:

- Both group $A$ and $B$ end with a conditional jump (they have two possible successors each)
- For each outgoing edge from A, the edge must either go to B, or to a successor of B – in other words, merging may not add any new successors. More formally, $\mathrm{succ}(A) \subseteq \{B\} \cup \mathrm{succ}(B)$, where $\mathrm{succ}(X)$ is the set of possible successors for the group $X$.

## 4.4 Construct detection

After the groups have been created, we then analyze the groups to find the various kind of control flow constructs. The constructs are detected in multiple steps, with one construct per step, in the following order:

- `do-while`
- `while`
- `break`
- `continue`
- `if`
- `else`

Each of these five constructs are marked using a `GroupType` member of the `Group` type, while `else` blocks are flagged using two booleans, `_startElse` and `_endElse`. If `_startElse` is true, then an `else` block starts with this group, and should be output prior to the code in this group. If `_endElse` is true, an `else` block ends with this group, and the end should be output after the code in this group.

Once a group has been flagged as being some construct, it is skipped for the other constructs.

The criteria used for each construct are as follows:

**Do-while detection** Group must end with a conditional jump (i.e., have two outgoing edges). Jump must go to an earlier place in the code.

**While detection** Group must end with conditional jump. Block must have an ingoing edge from some group later in the code, unless that edge comes from a do-while condition (in which case it is assumed to be an `if` instead).

**Break detection** Unconditional jump to some place later in the code. That place must either be the group immediately after a `do-while` condition, or the jump target of a `while` condition. Additionally, the jump is verified to go to the appropriate loop (so it does not exit multiple loops at once).

**Continue detection** Unconditional jump to a `while` or `do-while` condition, unless it is targeting a `while` condition which jumps to the next sequential group (in which case it is merely the end of the `while`-loop). Just as with `break`s, the jump is verified to go to the appropriate loop.

**If detection** All unflagged conditional jumps are flagged as `if`s.

**Else detection** All `if`s are processed to see if they may have an `else` attached. If the jump target of an `if` is immediately preceded by an unconditional jump, which is neither a break or a continue, and that jump goes to later in the code, this signifies a possible `else` block, starting with the jump target of the if condition and ending with the group immediately before the target of the jump immediately before the jump target of the if condition. To avoid false positives, this block is then validated to not cross block boundaries. If the check passes, the `else` block data is added to the graph.

## 4.5 Graph output

The graph can be output in DOT format by using the `-g` switch. In the graph, arrows on edges will be hollow if the edge is a jump, and filled if the edge represents the usual sequential order.

## 4.6 Limitations

Currently, only unconditional jumps are supported for `break` and `continue`; however, for code of the form `if (...) break;` or `if (...) continue;`, it is a pretty straight-forward optimization to use the `break`/`continue` jump as the conditional jump for the `if` condition check. Since `if`s are found last, it should be possible to simply check unmarked conditional jumps as well and see if they meet the other criteria for a `break`/`continue`, although there might be some false positives for an if that stretches to the end of the loop it is placed in.

It is currently assumed that all conditional jumps in `if` condition checks go to a later place in the code. If optimized continue statements are used in a while (as described above), this will cause the analysis to be incorrect.

## 5 Code generation

Having detected all of the various control flow constructs in the previous phase, it is now time to put that information to use and generate some code from the instructions.

Code generation is implemented as a two-step process:

- First, a DFS is performed to process all reachable groups. During this processing, the code for that group is generated.
- Next, the groups are iterated over sequentially, and the generated code is output.

This process is repeated for each function.

### 5.1 Function signature

For each function in the script, the `constructFuncSignature` method is called. By default, this will return the empty string, and this will cause the code to be output "freely", i.e. without anything surrounding it. If a non-empty string is returned, a `}` will be added after all of the code in the function.

If your engine uses methods, you will want to override this method to output your own signature.

*Note:* You must currently include a `{` at the end of your signature.

### 5.2 Group processing

During processing of a group, the instructions in the group are processed one at a time. This is done by calling `processInst` on each instruction. This method should emulate the effect of the instruction, and, if the instruction corresponds to a statement, call `addOutputLine` on the code generator to add this statement as a line of code.

If you need to access information about the group currently being processed, use the member variable `_curGroup` on the code generator.

### 5.3 The stack and stack entries

When generating the code, a stack is used to represent the state of the system. When data is pushed on the stack, a `Value` describing how that data was created is added; when data is popped, a `Value` describing the popped data is removed.

To manipulate the stack, use the `push` and `pop` methods to push or pop values. Unlike the STL stack, `pop` returns the value being popped from the stack, so you do not have to first get the top element and then pop it afterwards, but you can still call the `peek` method if you just want to look at the topmost element without removing it. Additionally, it has an `empty` method to check if the stack is empty.

Some engines require you to look further down the stack than just the topmost element. You can use the `peekPos` method to retrieve an element at an arbitrary position in the stack. This method takes an integer containing the number of stack entries to skip, i.e. passing the value 0 will give you the topmost element, while passing the value 2 will give you the third value on the stack.

*Note:* `peekPos` accesses the underlying STL container (`std::deque`) using the `at` function, which will throw an exception if the stack does not contain enough elements.

When working with values, you should use the `ValuePtr` type. This wraps the entry in a `boost::intrusive_ptr` to free the associated memory when it is no longer referenced.

Some value types contain references to an arbitrary number of values. This can be handled using an STL `deque`, typedef'ed as `ValueList`.

The following value types are predefined:

**Integers (IntValue)** Integers can use up to 32-bits, and be signed or unsigned. When creating an integer, you must specify its value and whether or not it is signed. This also contains additional methods to extract the value and signedness of the value, which may be of use in some situations.

**Addresses (AddressValue/RelAddressValue)** Addresses are implemented as a specialization of IntValue. When output to a string, hexadecimal notation is used instead of decimal, and for relative addresses, the sign is prefixed and the absolute offset value is used instead (i.e., -1 is shown as `-0x1` instead of `0xFFFFFFFF`).

Absolute addresses cannot be retrieved using `getSigned`, while relative addresses will return the offset with this method. To get the absolute address associated with either of these values, call `getUnsigned`.

**Variables (VarValue)** Variables are stored as a simple string. Subclasses of `CodeGenerator` must implement their own logic to determine a suitable variable name when given a reference.

**Binary operations (BinaryOpValue)** Binary operations stores the two stack entries used as operands, and a string containing the operator. Parentheses are added if the operator precedence requires it.

**Unary operations (UnaryOpValue)** Just like binary operations, except only a single operand is stored. The operator can be placed before (prefix) or after (postfix) the operand.

**Duplicated entries (DupValue)** Stores an index to distinguish between multiple duplicated entries. This index is automatically assigned and determined when calling the `dup` function to duplicate a stack entry.

**Array entries (ArrayValue)** Array entries are stored as a simple string containing the name of the array, and an EntryList of stack entries used as the indices, with the first element in the ValueList being output as the first index.

**Strings (StringValue)** A string is stored as... well, a string. The default implementation automatically surrounds the string with quotes.

**Negated values (NegatedValue)** `NegatedValue` represents an expression of the form `!value`, i.e. boolean negation. This type is used to ensure that `value->negate()->negate()` does not actually perform double negation, but simply uses the original value.

**Function calls (CallValue)** Function calls have the same underlying storage types as an array entry, but the output is formatted like a function call instead of an array access.

Each entry type knows how to output itself to an `std::ostream` supplied as a parameter to the `print` function, and the common base class `StackEntry` also overloads the « operator so any stack entry can be streamed directly to an output stream using that function.

## 5.4 Outputting code

When processing certain kinds of instructions, you will probably want to create a line of code as part of the output. To do that, call `addOutputLine` with a string containing the code you wish to output as an argument. This will then be associated with the group being processed.

If your line of code deals with specialized control flow, you will probably want to do something about the indentation. You can supply two extra boolean arguments to `addOutputLine` to state that the indentation should be decreased before outputting this line, and/or that the indentation should be increased for lines output after this line. If you leave out these arguments, no extra indentation is added.

Note: By default, if, while and do-while statements detected in the control flow graph are automatically output after processing the conditional jump. To fill in the condition, the topmost stack value is used.

Note: This indent handling is currently considered a temporary solution until there is time to implement something better. It may be replaced with a different form of indentation handling at a later time.

You will usually need to output assignments at some point. For that, you can use the `writeAssignment` method to generate an assignment statement. `writeAssignment` takes two parameters, the first being the Value representing the left-hand side of the assignment operator, and the second being the Value representing the right-hand side of the operator.

## 5.5 Default instruction handling and instruction metadata

Default handling exists for a number of instruction types, described below. See also Section 3.1 on page 7 which discusses instructions in more detail.

**DupStackInstruction** The topmost stack entry is popped, and two duplicated copies are pushed to the stack. If the entry being duplicated was not already a duplicate, an assignment will be output to assign the original stack entry to a special dup variable, to show that the original entry is not being recalculated.

**UnaryOpPrefixStackInstruction/UnaryOpPostfixStackInstruction** The topmost stack entry is popped, and a `UnaryOpEntry` is created and pushed to the stack, using the codegen metadata as the operator, and the previously popped entry as the operand. The exact type determines whether the operator is pre- or postfixed to the operand.

**BinaryOpStackInstruction** The two topmost stack entries are popped, and a BinaryOpEntry is created and pushed to the stack, using the codegen metadata as the operator and the previously popped entries as the operands. The order of the operands is determined by the value of the field `_binOrder`, as described in Section 5.6 on the following page.

**ReturnInstruction** This simply adds a line `return;` to the output.

*Note:* The default handling does not currently allow specifying a return value as part of the statement, as in `return 0;`. You will have to handle that yourself using a subclass.

**KernelCallStackInstruction** The metadata is treated similar to parameter specifications in `SimpleDisassembler` (see Section 3.5 on page 11). If the specification string starts with the character `r`, this signifies that the call returns a value, and processing starts at the next character. For each character in the metadata string, `processSpecialMetadata` is called with the instruction being processed, and the current metadata character to be handled. The default implementation only understands the character `p`, which pops an argument from the stack and adds it to the argument list. Once the metadata string has been processed fully, then an entry representing the function call is pushed to the stack if the call returns a value. Otherwise, the call is added to the output.

You can override the `processSpecialMetadata` method to add your own specification characters, just like you would override `readParameter` in `SimpleDisassembler`. Use the `addArg` method to add arguments.

Due to the conflict with the specification of a return value, it is recommended that you do not adopt `r` as a metadata character unless you provide your own `processInst` implementation for this purpose.

**UncondJumpInstruction** Nothing is output, as this should already be handled by the generic code.

In addition, certain instruction types trigger additional generic behavior:

**Conditional jumps** After processing the conditional jump, an if, while or do-while condition is output using the topmost stack entry as the condition. The condition is automatically negated for if and while conditons, so you only need to consider what the instruction itself does. If the jump is taken when the checked condition is false (e.g. a `jumpFalse` instruction), you must remember to negate the value representing the condition.

**Unconditional jumps** After processing the instruction, if the current group has been detected as a break or a continue, a break or continue statement is output. Otherwise, the jump is analyzed and output unless it is a jump back to the condition of a while-loop that ends there, or it is determined that the jump is unnecessary due to an else block following immediately after. A default, empty implementation of `processInst` is provided for this type.

**Other types** No default handling exists for types other than those mentioned above, so you must handle them yourself by creating new Instruction subclasses.

Note that this also includes `CallInstruction`. Although many engines might want to handle this in a manner similar to `KernelCallInstruction` opcodes, this is left to the engine-specific code so they can fully make sense of the metadata they choose to add to the function.

## 5.6 Order of arguments

The generic handling of binary operators (`BinaryOpStackInstruction`) and kernel functions (`KernelCallStackInstruction`) can be configured to display their arguments using FIFO or LIFO - respectively, the first and the last entry to be pushed onto the stack is used as the first (leftmost) argument. This is set as part of the constructor for the `CodeGenerator` class, using the enumeration values `kFIFOArgOrder` and `kLIFOArgOrder`.

To provide an example, consider the following sequence of instructions:

```
PUSH a
PUSH b
SUB
```

This can mean two different things, either `a - b` or `b - a`, depending on the order in which the operands should be evaluated. For the former, choose FIFO ordering, for the latter, choose LIFO.

For arguments to function calls, the same principle applies. You can use the `addArg` method to add an argument to the call currently being processed, using the chosen ordering.

In general, you might not know which ordering is more correct for function arguments; unless you have reason to believe otherwise, simply use the same ordering as for binary operators.

## 6 Current restrictions

This section describes restrictions and limitations that are currently placed on the individual engines that have been implemented.

### 6.1 SCUMMv6

roomObj scripts have a header specifying offsets for various verbs. This is currently ignored, and the script is treated as a single piece of code; it would be more appropriate to create functions from these and properly treat stopObjectCodeA/B as returns.

### 6.2 KYRA2

The decompiler has only been tested with English scripts (.EMC files) from the Kyra 2 demo. It is possible that some scripts from the full version misbehave somehow; as I do not own the game, I cannot be sure of this.

The setRetAndJmp opcode is not currently handled since I don't have any examples of it.

Only "regular" scripts are supported; animation scripts are not supported. It looks like the file names of animation scripts all start with either _Z or _IDL, but this is not necessarily an exhaustive list.

# 7 To-do list

During the course of developing the decompiler, various ideas have been considered for implementation in the decompiler, but not all of them could be implemented within the given timeframe.

Following is a description of known issues and enhancements that would be nice to see in the decompiler at some point in time.

Note that the order of the issues is not necessarily indicative of the priority that should be given to each issue.

## 7.1 Allow conditional jumps to be used for break/continue

Currently, only unconditional jumps are allowed for break/continue. However, where the code is similar to `if (...) break;`, it is a simple and obvious optimization to not output an unconditional jump and just make the proper address the target of the conditional jump from the if.

Support for this should be added to the analysis. Should be doable by also checking destinations of unmarked conditional jumps, since ifs are marked after breaks and continues - so simply add that to the initial filter and make sure we process the right (non-sequential) target vertex.

Classifying as an enhancement since it appears this optimization is not used in SCUMMv6 (there is an if (...) break in tentacle/room-13-206, but the break is a separate jump). Not sure if KYRA uses it - a cursory scan of the KYRA2 demo suggests that it could use an optimization, as I found only 1 occurence of a c1_ifNotJmp immediately followed by a c1_jump (skull\skull.emc, 0x0578) - but it did not look like a loop. That, however, does not prove that KYRA actually uses the optimization - it could simply be that such a piece of code was never used.

As far as I have been able to tell, this optimization really is not used in Kyra, so this will have to be deferred until we have an engine which needs it.

## 7.2 Re-enable short-circuit detection

Currently the short-circuit detection is disabled because it requires some extra handling in code generation which is not there yet (you have to analyze each jump more closely).

## 7.3 Refactor CFG design

The CFG anaylsis, while certainly functional, is not entirely pretty right now.

It would be a good idea to go over this and see if it can be made better somehow, e.g. by incorporating more of the syntax as nodes in the graph. This might also make it easier to get short-circuiting working correctly.

## 7.4 Refactor disassemblers to accept a SeekableReadStream

It would be desirable if disassemblers accepted a `Common::SeekableReadStream` instead of a `Common::File`, for easier integration with other tools and possibly ScummVM itself.

Unfortunately, streams do not currently exist in the tools repository. To get around this, it should be enough to move over `ReadStream`, `WriteStream`, and `SeekableReadStream` to begin with, and make `Common::File` subclass from the required streams.

This is very much an *optional* task, and only to be done near the end, if time permits - it is not really within the scope of the project, and it would be fairly easy to make the switch after GSoC.

## 7.5 SCUMM: Rewrite jump 0 at end of script to infinite loop

Several SCUMM scripts end with a jump 0, making them infinite loops. It would be nice if this could be expressed accordingly, but this does not appear to be a trivial task; some jump 0s in a script could be expressed as a continue, others cannot.

## 7.6 Proper getCondition method on Value

For now, it is assumed that conditional jumps leave their condition on the stack, so this can be retrieved by the generic code generation code. For non-stack-based engines, it would be a bit nicer if they could just give us a condition to use in an if/while/do-while, instead of currently requiring that the value is on the top of the stack.

A very simple way to do this would be to simply define a getCondition on Value that takes the same parameters as processInst, with a default implementation that just calls processInst and pops and returns the top value from the stack - this means no changes would be required to exisiting engines, and new engines can override this method as they see fit.