x86 assembler for JIT code generation

Design Document

18.11.18

Jari Korkala
jv.korkala@gmail.com

x86 assembler for JIT code generation
18.11.18
Jari Korkala

Table of Contents

x86 assembler for JIT code generation
18.11.18
Jari Korkala

1 Introduction

This project started as a spin-off of the software 3d renderer project in 2014. It had become apparent that some pipeline stages would have too many options for them to be hard-coded into the renderer. At that time Direct3D and OpenGL already had their specific programming languages and compilers for vertex and pixel shader, but I also found out that depth-stencil tests have a wide configuration vector:

```
struct DepthStencilTest
{
	Ceng::TEST_TYPE::value stencilTest;

	UINT32 stencilReadMask;
	UINT32 stencilWriteMask;

	UINT32 stencilRef;

	Ceng::STENCIL_ACTION::value stencilFail;
	Ceng::STENCIL_ACTION::value depthFail;
	Ceng::STENCIL_ACTION::value depthPass;
};

class DepthStencilDesc
{
public:
	Ceng::BOOL stencilEnable; // Do stencil tests
	Ceng::BOOL depthEnable; // Do depth tests

	Ceng::BOOL depthWrite; // Allow writes to depth buffer

	Ceng::TEST_TYPE::value depthTest;

	DepthStencilTest frontFace;
	DepthStencilTest backFace;
};
```

Where the enumerations are

```
namespace STENCIL_ACTION
{
	enum value
	{
		KEEP = 1 ,
		ZERO = 2 ,
		SET_REF = 3 ,
		BIT_INVERT = 4 ,

		INCREMENT = 5 ,
		INCREMENT_SAT = 6 ,
```

```
                DECREMENT = 7 ,
                DECREMENT_SAT = 8 ,

                FORCE32B = 1 << 30 ,
        };
};

namespace TEST_TYPE
{
        enum value
        {
                LESS = 1 ,
                LESS_EQ = 2 ,

                EQUAL = 3 ,
                NOT_EQUAL = 4 ,

                ABOVE = 5 ,
                ABOVE_EQ = 6 ,

                ALWAYS_PASS = 7 ,
                NEVER_PASS = 8 ,

                FORCE32B = 1 << 30 ,
        };
};
```

The total number of simultaneous options is

- Stencil test on/off (2)
- Separately for front and back face (2)
  - Stencil test mode (8)
  - Stencil buffer operation when result is fail (8)
  - Stencil buffer operation when stencil test pass but depth test fail (8)
  - Stencil buffer operation when stencil test pass and depth test pass (8)
- Depth test on/off (2)
- Depth buffer write on/off (2)
- Depth test mode (8)

If the goal is to have a branch-free implementation, naive calculation gives 524288 different functions to be implemented. Of course we can quickly see that the number of unique functions is much smaller. There are two unique functions with (stencil test off, depth test off), 32 with (stencil test off, depth test on), and 1024 functions with (stencil test on, depth test off).

But I digress: the point is that if all unique functions were hard-coded the library would be huge, and most clients would only use a small number of the configurations. However, a JIT assembler might allow the functions to be constructed from demand with less code and data used, by stitching together code snippets that correspond to specific configuration item values. As additional value, the JIT assembler would be a necessary back-end to any shader compiler, so this project was born.

I started from the code generator, but then proceeded to write a parser on top of that to make the assembler general purpose. This document will describe both components separately.

2 Parser component

This component is still incomplete. It can be split further into the tokenizer and the intermediate format generator. The parser is not necessary to use the code generator, but it allows for more compact representation of the input.

2.1 Tokenizer

TBA

2.2 Intermediate format generator

TBA

3 Code generator component

Since this is a JIT assembler, the code generator has been designed to be used independently.

The code generator is in principle ready, but I found the project in a state where it might not be deployable.

3.1 Program definition

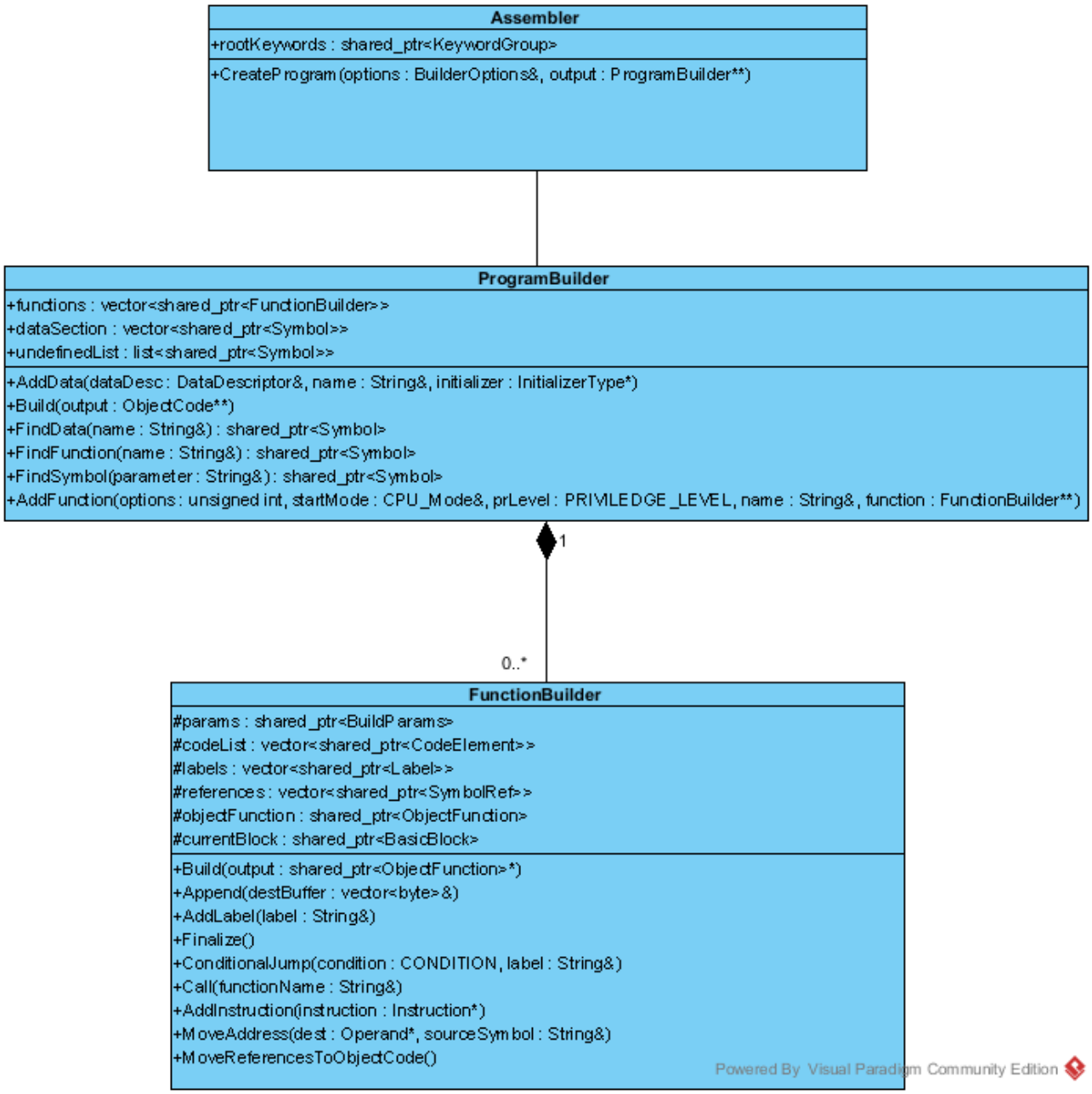

Illustration 1: UML class diagram of the top level classes.

In order to define the program to assemble, an intermediate format is used. Illustration 1 shows the top level classes of the assembler. Here ProgramBuilder and FunctionBuilder are used to construct the program to be assembled.

A program is similar to a compilation unit and its purpose is to produce object code. Therefore a new program object should be created for each .asm file. The program consists of functions (code

segment) and the data segment (dataSection in diagram). Code and data will be shared by the entire object code, and later in the linking phase by the entire output program. ProgramBuilder will keep track of undefined symbols for the linking phase.

Like the name suggests, FunctionBuilder is used to construct a function. Note that there aren't actually functions in the assembly language, only labels. Functions are treated as entities only for convenience.

Functions are defined as a sequence of labels and CodeElement objects. Illustration 2 shows the related class hierarchy.

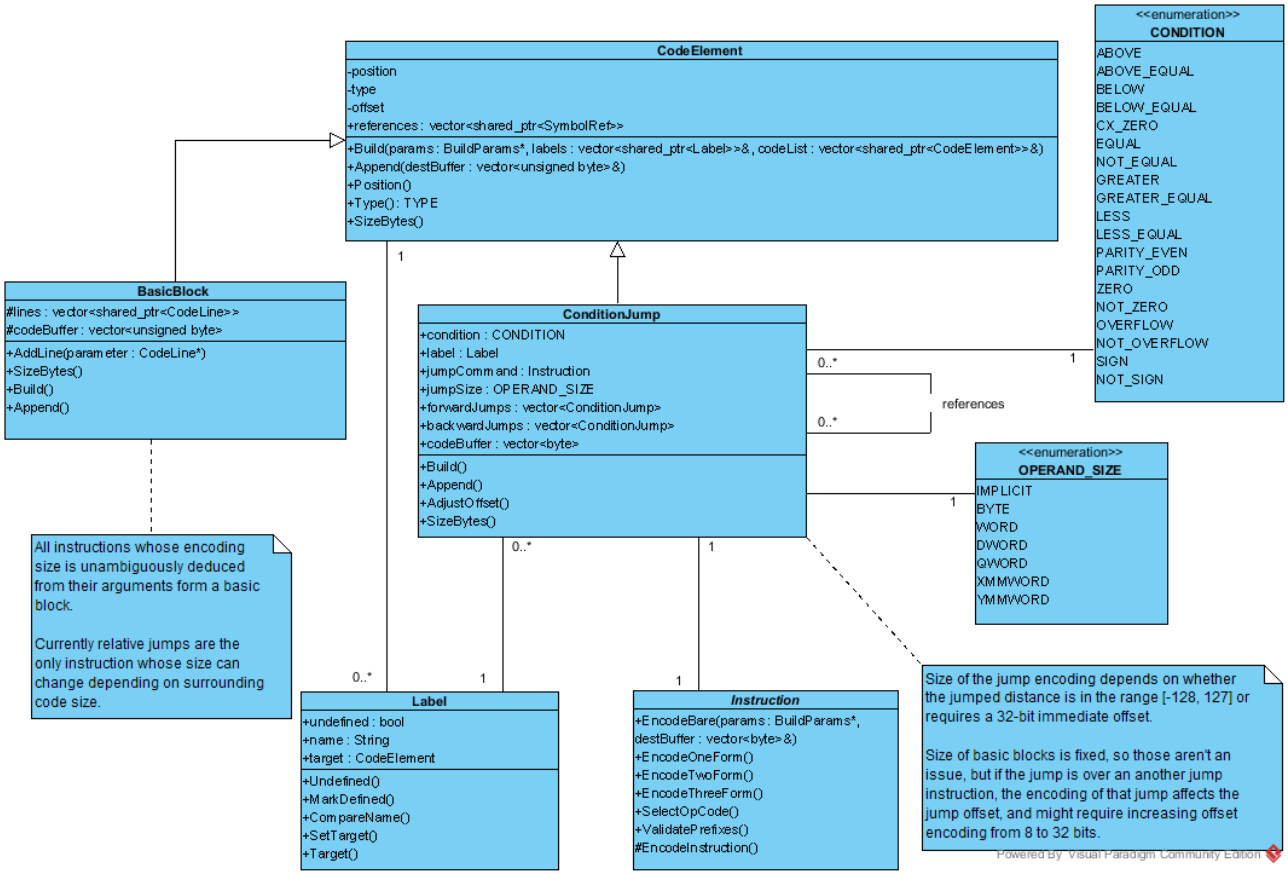

Illustration 2: UML class diagram for CodeElement.

3.1.1 Label

A label is a named symbol. When the function is assembled, it's value is the start address of the next basic block in the generated machine code. Therefore the label can be used as a jump target without having to know the actual address to jump to.

Use *FunctionBuilder.AddLabel()* to add label to the current position of the function.

3.1.2 Conditional jump

This class is used to define a conditional jump instruction. Since the encoded length of these instructions depends on the jump distance, it is useful to keep track of other jump instructions between this object and it's target label (start address of a basic block).

NOTE: Unconditional relative jumps are TBA feature, but would be handled in a similar manner.

Use *FunctionBuilder.ConditionJump()* to append a conditional jump instruction that targets a label in the function. The label doesn't have to be defined prior to use of this function. The label has to be defined before *ProgramBuilder.Build()* is called, however.

3.1.3 Basic blocks

This class defines a continuous segment of instructions whose encoded length is unambiguous. There are no jumps out of or into a basic block.

If a label or conditional jump is appended, the current basic block is immediately closed and assembled since the encoding format won't change later.

FunctionBuilder.AddInstruction() methods can be used to append most of the x86 instructions.

AddInstruction(Instruction)* : append instruction without parameters

AddInstruction(Instruction, Operand*)* : append with single operand

AddInstruction(Instruction, Operand *dest, Operand *source)* : append with two operands

AddInstruction(Instruction, Operand*, Operand*, Operand*)*: append with three operands

FunctionBuilder.MoveAddress() is a shorthand method to append a x86 instruction to move the address of a symbol to the destination operand. It is essential to use of pointers to global variables or functions.

NOTE: A similar method for stack variables would be useful but is a TBA feature.

FunctionBuilder.Call() is a shorthand method to append a x86 call instruction. Currently the method requires the name of a function as param, but in the future it should also accept a label.

Functions *Build()*, *Finalize()* and *MoveReferencesToObjectCode()* are part of the interface to ProgramBuilder and should not be used by the application. Potential future changes include moving these functions to protected and declaring ProgramBuilder as friend, or exposing limited view to application.

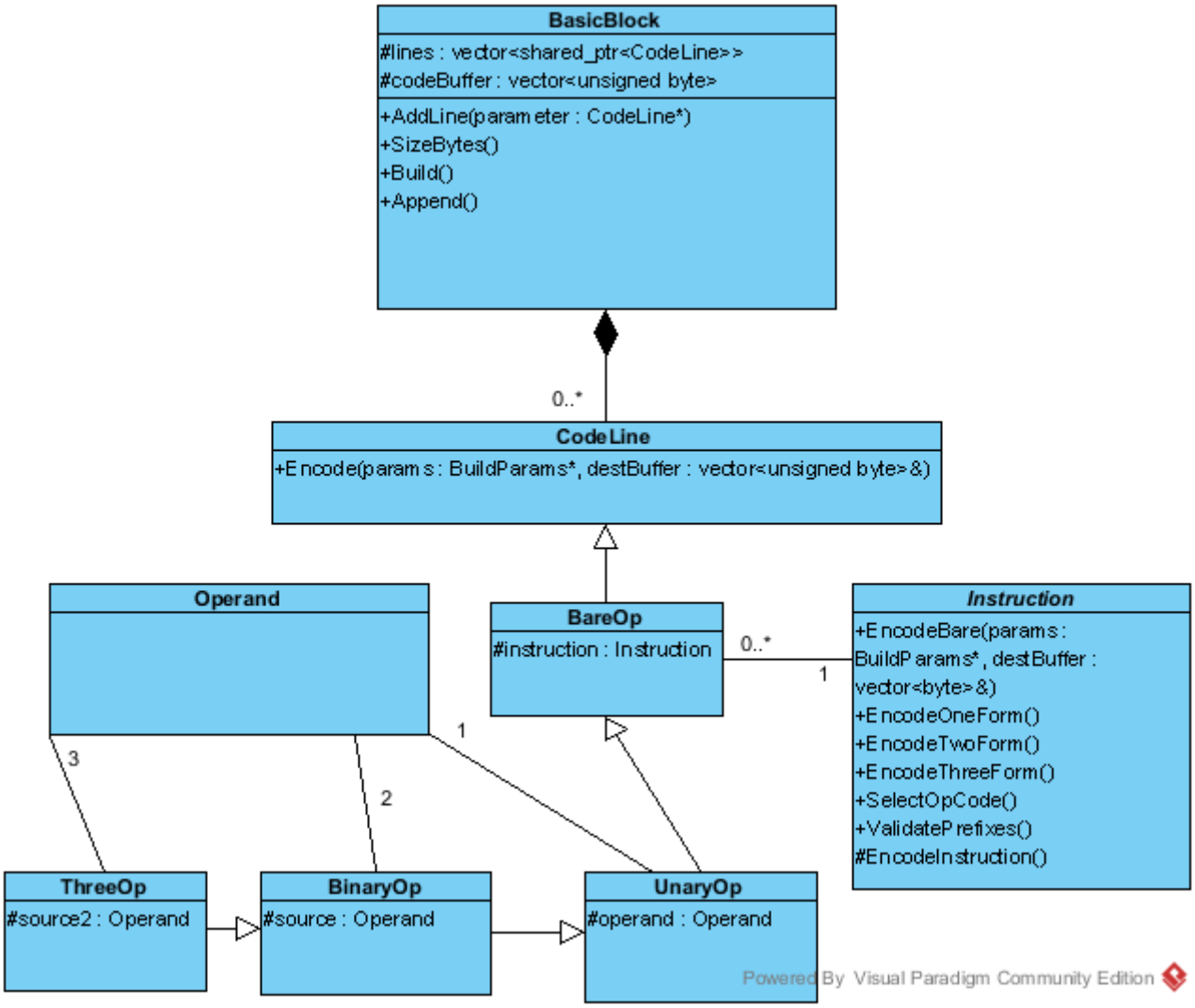

Illustration 3: UML class diagram for basic blocks.

Basic block consists of the CodeLine objects that store instructions with their operands. Illustration 3 shows the class diagram for basic blocks.

3.1.4 Operands

These classes provide operands for an instruction. Illustration 4 shows the operands supported by x86 instruction set. RegisterOperand can be used either as source, destination, or part of a memory address calculation in MemoryOperand.

ImmediateOperand is either a literal or a symbol. When symbol is used, the encoded immediate value field gets replaced with the memory address of the symbol.

MemoryOperand has a myriad of options. The most typical uses are:

[disp32] : a 32-bit signed integer is used as memory address. This is used to move value of a global variable to destination. *MemoryOperand(shared_ptr<Symbol>)* is a special case uses the address of the symbol as displacement.

> NOTE: Since the displacement is signed, this options is limited to the lowest 2 GB of the address space. It is unlikely, however, that an application has that many global variables.

[base+(disp8 | disp32)] : Use value of base register + optional offset as a memory address. Pointer with optional struct member or array offset.

[base + index(*scale) + (disp8 | disp32)]: Use value of base register + index register (optionally multiplied by scale) + optional offset as memory address. Useful for array access when size of array element is 1,2,4 or 8 bytes (possible values of scale).

[RIP + disp32] : Used in 64-bit mode for instruction pointer + offset addressing. RIP stands for register, instruction pointer.

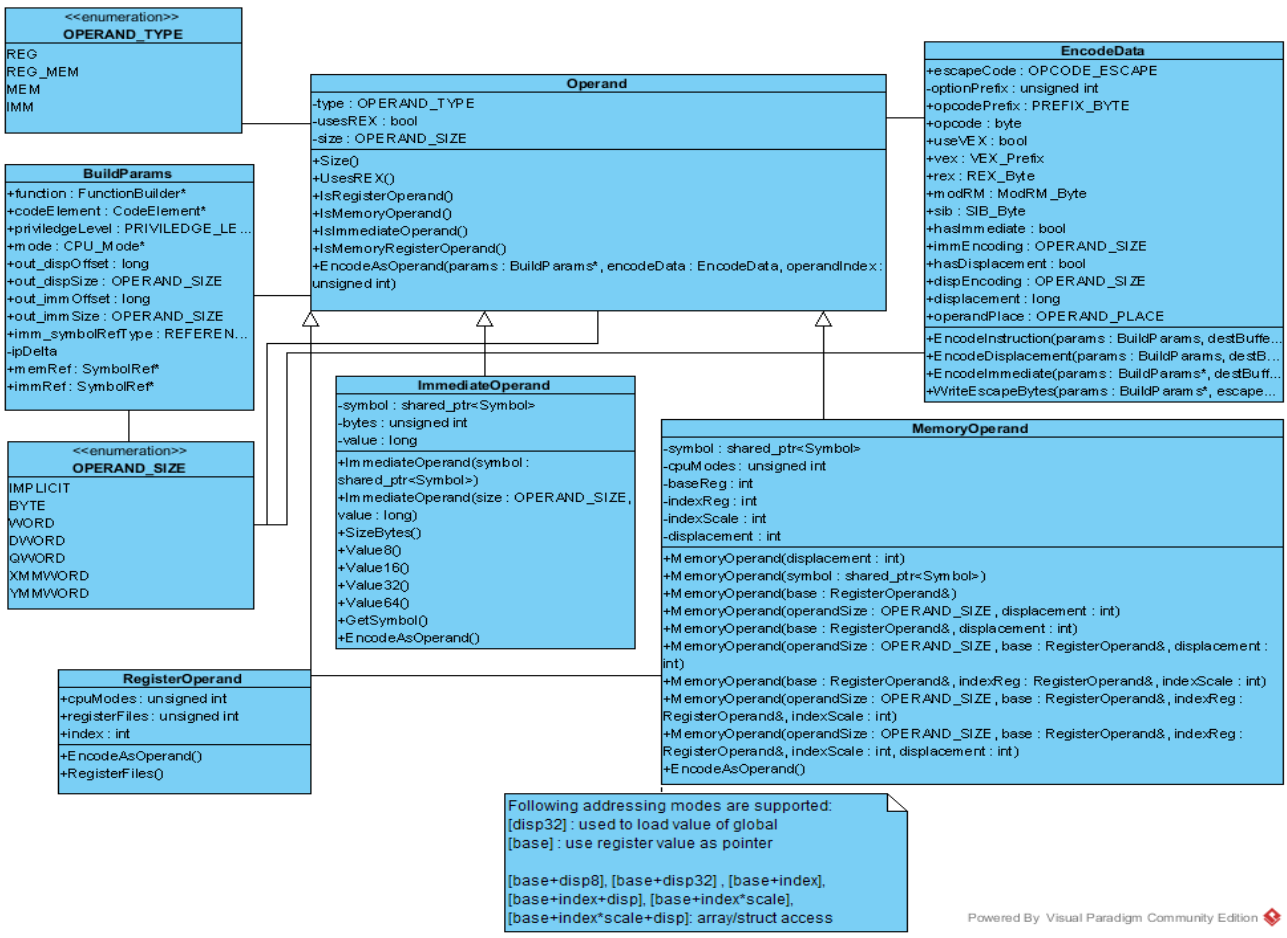

Illustration 4: UML diagram for instruction operands.

3.1.5 Instructions

These classes provide machine code generator configuration for each x86 instruction. Illustration 5 shows the class diagram. For each x86 instruction, there is a const instance of one of the classes in the hierarchy.

All of the configuration methods, such as EncodeOneForm(), configure an instance of EncodeData, which does the final machine code generation.

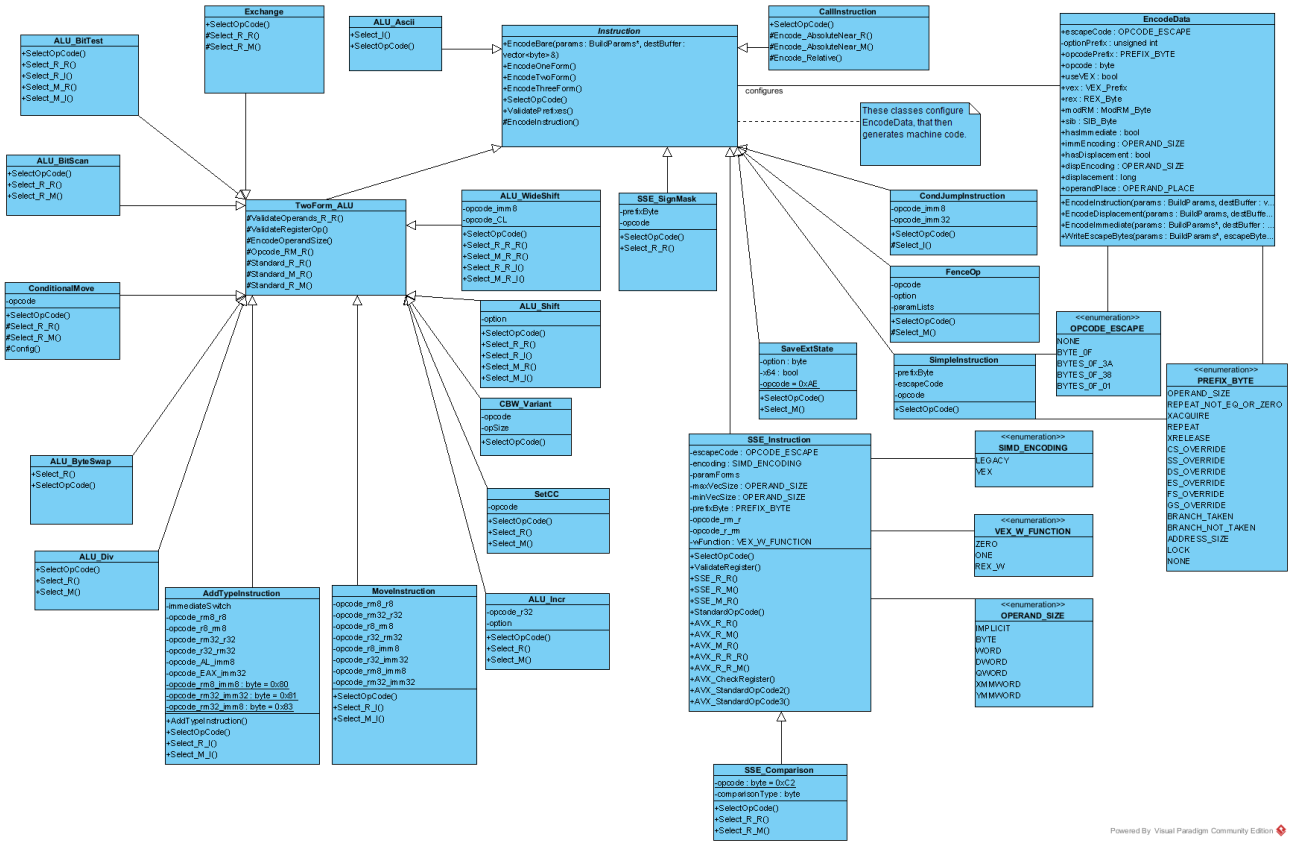

Illustration 5: UML class diagram of the Instruction class hierarchy.

The x86 instruction set is 30 years old, so the generation options are complex, as can be seen in Illustration 6. The actual instruction is selected by extension prefixes and a one byte opcode. Before x86-64, the ModRM and SIB bytes were used to indicate operands most of the time, but sometimes operands are part of the opcode, or encoded as an immediate operand.

AMD added REX prefix in x86-64, which indicates 64-bit operand size, and allows use of 16 registers per register file.

SSE floating point SIMD instructions were at first encoded with a prefix:

Datatype	Dimension	Prefix
Float	Packed (vector)	0x0F
Float	Scalar	0x0F 0xF3
Double	Packed	0x0F 0x66
Double	Scalar	0x0F 0xF2

This changed with the introduction of the VEX prefix in 2011 together with AVX instruction set. The VEX prefix combines the previous prefixes and REX into a more compact form.

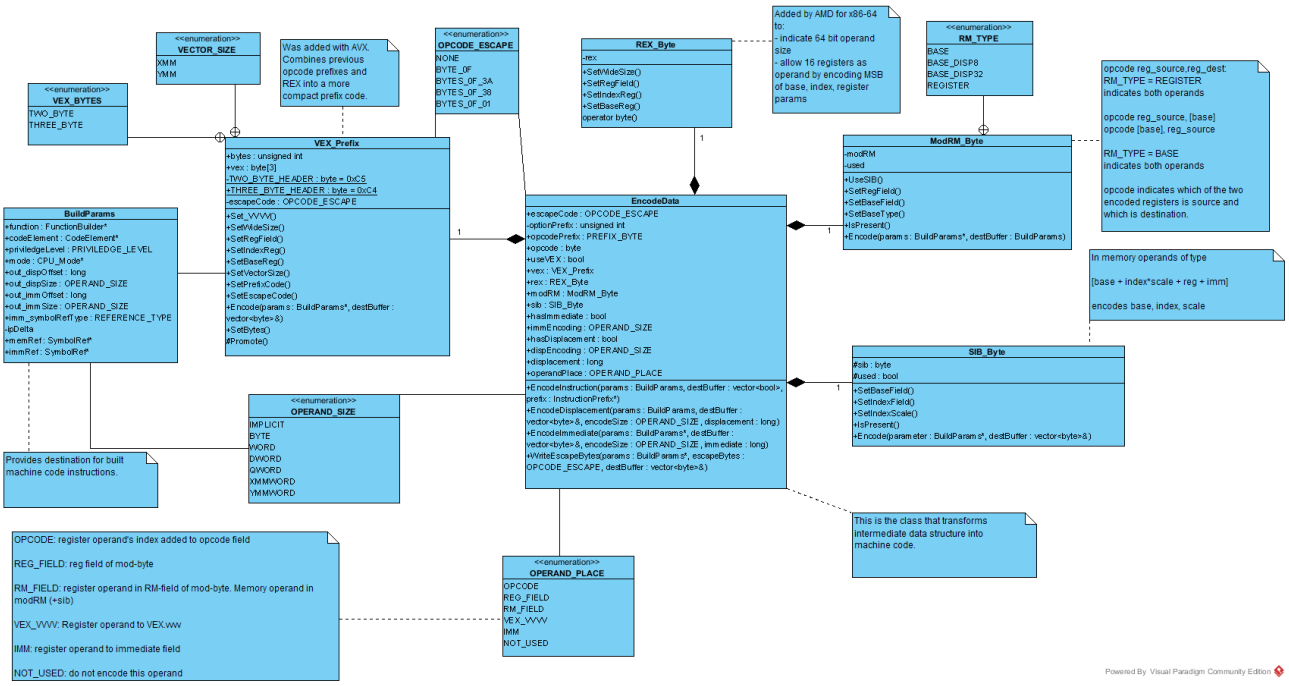

Illustration 6: UML diagram of the EncodeData class.

3.2 Object code generation

When the program has been defined, it is time to build it. This is done by calling *ProgramBuilder.Build()*. Output is the ObjectCode which is seen in Illustration 7. At this point each function is completely converted into a single block of machine code. All code segment sites where a symbol's address has to be written are gathered.

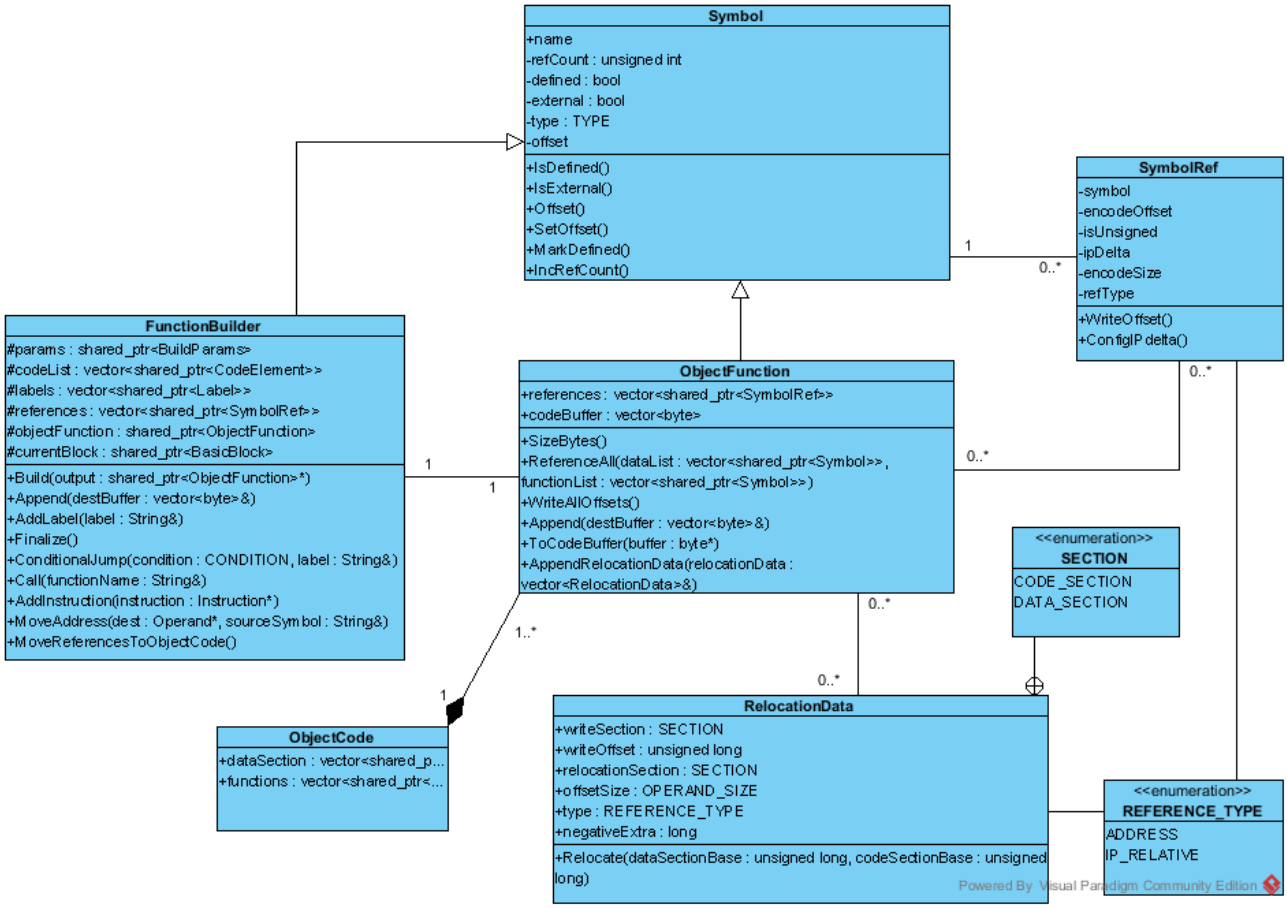

Illustration 7: UML class diagram of the generated object code.

4 Linker component

When all input units (.asm) have been converted to ObjectCode instances with ProgramBuilder.Build(), the linker can take any number of them as input and produce a linked program.

The linker first gathers all symbols used and finds the designated entry point function. Data segment is allocated and populated with initial values.

Code segment is filled with the entire linked program. Zero-based addresses of all symbols are written into the designated locations in the code segment. If the symbol is a variable, then the address is relative to start of data segment. If the symbol is function or label, address is relative to start of code segment.

Relocation information is generated so that base address of data and code segments can be added to these locations when generating an executable.

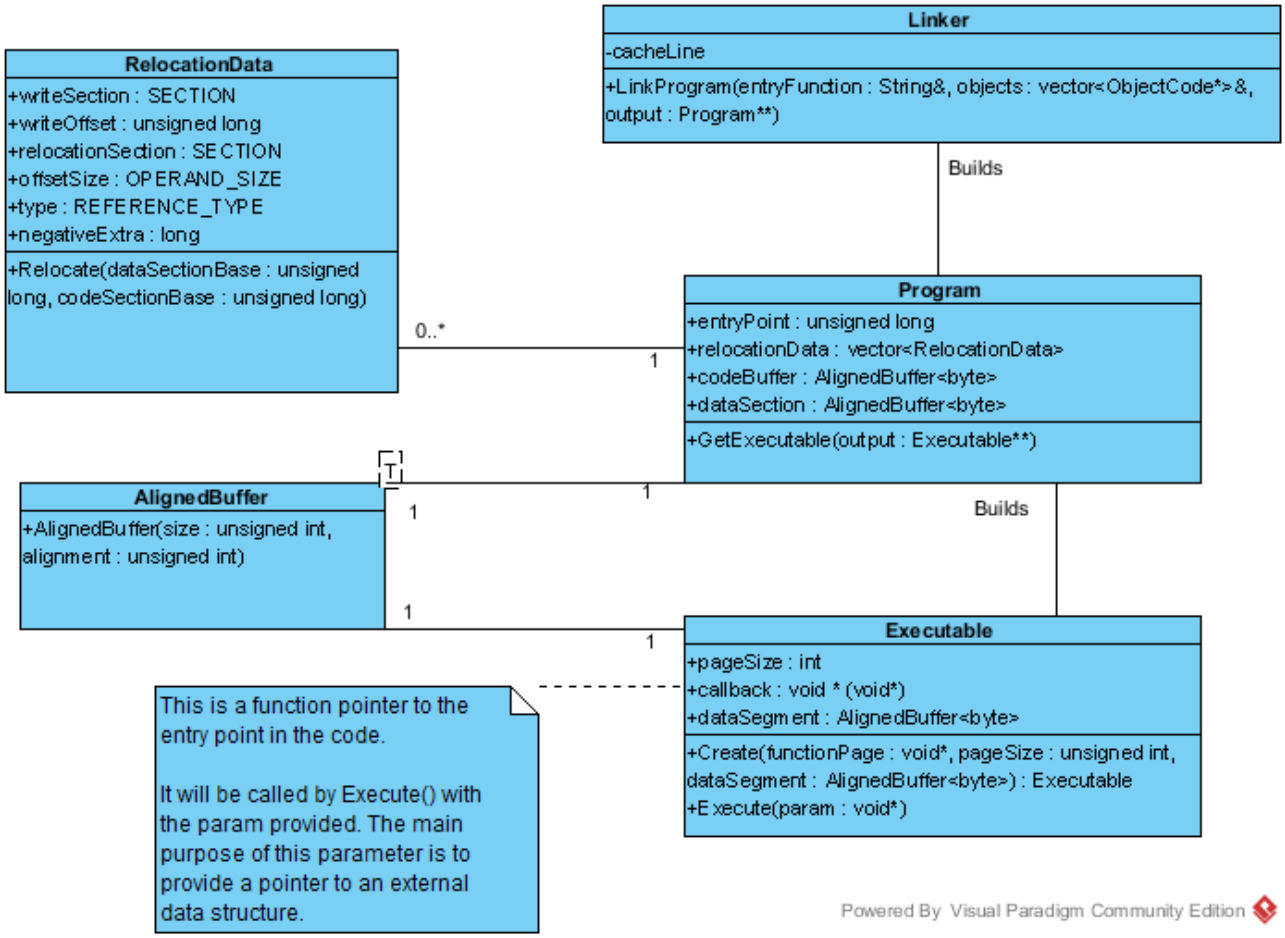

Illustration 8: UML diagram of the linker and execution parts.

5 Execution component

It is foreseen that multiple copies of the program can be in use simultaneously, so all need their own data. This means that all raw memory addresses embedded in the code must be adjusted by the start address of the executable's data segment. As a consequence a copy of the code must be made for each executable as well.

Linker generates relocation information so that this process of relocating the code is as streamlined as possible.

The generated executable is started with the *Execute(void *param)* method, where the parameter is currently used to provide a pointer to some external data structure for testing purposes. In the future additional Execute() methods could be added with different parameter numbers.

Potential future work could be to add helpers to read and write symbol values in the data segment.

6 Conclusions and future work

I found this project abandoned really close to completion. Unfortunately the current state is non-buildable since it seems I was in the process of adding the parser and streamlining the code generation components. Note to self to not do two major changes at the same time again.

Short term goal should be to get the code generator to work again, since that was the original goal of the project.

It seems the work on the tokenizer is quite far, but I think the intermediate format generator has to be overhauled. For x86 assembly, the shunting-yard algorithm should be adequate, but I'm also considering to test the approach I took on combo detection in my game engine project. In the game engine combos are stored as a tree that branches depending on the next input so that the entire input sequence so far is implicitly known based on the current node.

The string class I wrote for this project is my first extensive take on templates and generic programmin. It can be found at /include/datatypes/ce-string.h. It supports UTF-8, UTF-16, UTF-32 encodings, and differentiates between number of UTF codepoints and bytes in the string. It provides iterator, append, find, insert, erase, and substring functionality. It still has occasional compilation issues and would be a good excercise in unit testing.