A Quick Break

It’s been ages since I posted on this project, the last year has been a bit crazy for me, but I’ve picked it up again now that I have a (tiny) bit more time. So where were we and where are we? At the end of the last article we had a working VM with some minor optimisations and the start of a compiler. I’m going to break into the compiler project for a while to diverge onto something else which will be developed in parallel with the compiler: a debugger. Why? Because stepping the code line-by-line in C# is going to become increasingly burdensome. In the last project it was the C# implementation of the VM that usually needed debugging, from now on it will mostly be the code we write to run on the VM that needs debugging.

So, what’s the first thing to do on this tangent? Go off on another tangent of course! Spceifically, before we write the debugger we are going to write an assembler. There are two reasons for this, number one: the debugger will need a disassembler and if we are writing that we might as well have an assembler. Number two: I want to start using the compiler on test programs like the “What’s your name” program from the VM development articles. It will be a while before the compiler can manage that entire program so we need some assembly to fill in the blanks.

The first thing to do is clarify what assembly is and what it’s for. The basic idea is to make the binary a lot more human-readable than just the list of numbers we would see in the C# debugger or the program file. To do this we go through the program and replace those numbers with a human-readable counterpart. This text will be more or less equivalent to the C# enums that correspond to the numbers that make up a program.

The assembler is going to be much simpler than the main compiler as it has a simpler job to do, it will be a console app with a single class called Assembler. The Assembler class has one public function called ParseString which takes a string containing the program in assembly form and outputs a list of ints which is the binary representation of the program. Our compiler will then call this function:

public static List<int> ParseString(string asm)

{

// One instruction per line

string[] lines = asm.Split('\n');

// Used for error reporting

UInt32 lineCount = 0;

List<int> binaryStream = new List<int>();

foreach (var line in lines)

{

lineCount++;

// The components of an instruction can be split by spaces or tabs

string[] parts = SplitLine(line);

// Generate the binary for this line

ParseExecutionUnit(parts, lineCount, binaryStream);

}

return binaryStream;

}

This is a really simple function, it just takes the string of assembly splits it by line, then splits each line by whitespace and finally passes each line to ParseExecutionUnit. This imposes a couple of restrictions on our asm format. Each line represents a single instruction, instructions can’t share lines or use more than one line. Secondly each part of an instruction is separated by whitespace. Now we have to create the first part of our asm definition, we are going to say that the first string in an instruction defines which execution unit will process it. The next step is to call a function which reads that information and calls the appropriate function to parse an instruction for the given execution unit:

public static int ParseExecutionUnit(string[] parts, UInt32 line, List<int> binaryStream)

{

if (parts.Count() == 0)

{

// Empty string

return 0;

}

switch (parts[0])

{

case "ALU":

{

ParseALU(parts, line, binaryStream);

}

break;

case "Branch":

{

ParseBranch(parts, line, binaryStream);

}

break;

case "Interrupt":

{

ParseInterrupt(parts, line, binaryStream);

}

break;

case "Load":

{

ParseLoad(parts, line, binaryStream);

}

break;

case "Store":

{

ParseStore(parts, line, binaryStream);

}

break;

case "Stack":

{

ParseStack(parts, line, binaryStream);

}

break;

case "Data":

{

ParseData(parts, line, binaryStream);

}

break;

case @"//": // Comment line

case "": // Blank line

case "<": // Closing tag from asm inlined into source

break;

default:

{

Console.WriteLine("Error, didn't recognise instruction type " + parts[0] + " on line: " + line.ToString());

}

break;

}

return 0;

}

This works exactly as you would expect, it just does a string comparison on the first part of the line and calls the appropriate function. These functions all work in more or less the same way so I’m just going to go through the ParseALU function to give you an idea of what is happening. All of our instructions are stored in two ints and all ALU instructions must have five parts, so the first thing we do is create those ints, set the execution unit they contain to ALU, and test that we have the right number of data items:

static void ParseALU(string[] parts, UInt32 line, List<int> binaryStream)

{

int instruction0 = (int)Virtual\_Machine.UnitCodes.ALU;

int instruction1 = 0;

if (parts.Count() < 5)

{

Console.WriteLine("Error, not enough arguments on line " + line.ToString() + ", should be 5");

return;

}

Next we need to know exactly what instruction for the ALU we are parsing and this becomes the second part of our asm definition, the second string in an instruction is the instruction to execute. Parsing this is trivial and doesn’t require a separate function call, we just OR the correct bits for the instruction into the first int of our binary instruction:

switch (parts[1])

{

case "Add":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.Add;

}

break;

case "AddLiteral":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.AddLiteral;

}

break;

case "Sub":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.Subtract;

}

break;

case "SubtractLiteral":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.SubtractLiteral;

}

break;

case "Mul":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.Multiply;

}

break;

case "MulLiteral":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.MultiplyLiteral;

}

break;

case "Div":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.Divide;

}

break;

case "DivLiteral":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.DivideLiteral;

}

break;

case "Copy":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.Copy;

}

break;

case "SetLiteral":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.SetLiteral;

}

break;

case "Nop":

{

instruction0 |= (int)Virtual\_Machine.ALUOperations.Nop;

}

break;

default:

{

Console.WriteLine("Error, ALU instruction \"" + parts[1] + "\" on line " + line.ToString() + " not recognised");

return;

}

}

Believe it or not we are now most of the way through parsing an instruction, it already contains the bits for which execution unit to use and which operation that unit should perform, all we need to do now is extract any data from the instruction.

There are three possible pieces of data in one of our instructions, two eight bit values which are packed into the second half of the first int, and one thirty-two bit value which is stored as the second int. Extracting this is as simple as using int.TryParse and making sure we got something sensible, then ORing the results into the right places:

int register0 = 0;

int register1 = 0;

bool test = int.TryParse(parts[2], out register0);

if (!test || register0 > 255)

{

Console.WriteLine("Error, third argument on line " + line.ToString() + " must be UInt8.");

return;

}

test = int.TryParse(parts[3], out register1);

if (!test || register1 > 255)

{

Console.WriteLine("Error, fourth argument on line " + line.ToString() + " must be UInt8.");

return;

}

test = int.TryParse(parts[4], out instruction1);

if (!test)

{

Console.WriteLine("Error, fifth argument on line " + line.ToString() + " must be UInt32.");

return;

}

instruction0 |= register0 << 8;

instruction0 |= register1;

Finally we need to add our new instructions to the binary instruction stream:

binaryStream.Add(instruction0);

binaryStream.Add(instruction1);

And that’s it, with one of these functions for each execution unit we have a complete assembler. It’s not very neat but it does the job and it’s so simple that tidiness isn’t a major concern. There is, of course, an inverse operation to assembly, which turns human-readable instructions into binary, and that is the process of turning binary instructions into human readable ones called disassembly. I’m not going to talk about that now because it’s built into the debugger that will be the subject of the next article series.

One of the reasons I gave for writing an assembler was so that we could embed asm in our .tim files, to do that we need to add a few extensions to the compiler. Firstly, we need an asm token type in our lexer. We will use the xml tags <asm> and </asm> to define a stretch of assembly and we aren’t going to add the concept to our grammar as it cannot be mixed into the main code, all we can do is define a series of instructions to be inserted directly into the instruction stream at the point that the token is encountered. We add “<asm>” as a new keyword and look for it in the GenerateToken function:

case "<asm>":

token.type = TokenType.OpenASM;

break;

In the GetNextToken function we check if the token is an asm opening and if so we read all of the text until we find a closing tag into the token’s data variable:

if(token.type == TokenType.OpenASM)

{

const string closingTag = "</asm>";

bool foundClosingTag = false;

while(!foundClosingTag)

{

for(int i = 0; i < 6; i++)

{

foundClosingTag = true;

if(m\_text[m\_positionInCode + i] != closingTag[i])

{

foundClosingTag = false;

break;

}

}

token.data += m\_text[m\_positionInCode];

m\_positionInCode++;

}

m\_positionInCode += 6;

}

The parser changes are largely trivial, we just create a new ASTNodeType of ASM and when we encounter an asm token insert a node based on it into the AST:

private bool ParseASM(SyntaxNode parent)

{

if(m\_tokenStream[m\_tokenIndex].type == TokenType.OpenASM)

{

SyntaxNode asmNode = new SyntaxNode();

asmNode.m\_type = ASTType.ASM;

asmNode.m\_data = m\_tokenStream[m\_tokenIndex].data;

parent.m\_children.Add(asmNode);

m\_tokenIndex++;

return true;

}

return false;

}

Finally, for code generation of an asm node we just pass the node’s data to the assembler we’ve been talking about here and insert the resulting binary instructions directly into the code stream:

private void GenerateASM(SyntaxNode node)

{

m\_codeStream.AddRange(Assembler.Assembler.ParseString(node.m\_data));

}

This is parsed in the top level of the code generator, which means we can insert ASM as a line of code and it will appear in the code stream in between the compiled output of the line above and the line below. In other words it works exactly as expected but has to be on its own line and can’t be inserted into the middle of a statement to be compiled. We can now re-write the test program from before as assembly, insert it into a .tim file and have the compiler build and run it:

From now on, instead of adding the source files as a .zip I’m just going to link to my GitHub project with all the code on it: