The MiniC Compiler

Harrison Knox

Compiler Overview

The MiniC compiler (hereafter named "Minic") is written in Java and compiles most valid Mini files to ARM Assembly. There are four stages to compilation:

1. Parsing
2. Static Type Checking
3. Intermediate Representation Building
4. Code Generation and Register Allocation

Parsing

Minic uses Antlr for parsing Mini files. Antlr provides the framework for parsing through the source file, but Minic uses specialized file visiting functions. Minic's source file visitor is based on the one provided by Professor Keen with some minor modifications, such as "squashing" blocks that contain only one statement within them. The parser returns an abstract syntax tree containing the three types of declarations: structs, globals, and functions. Forward declaration is allowed: you can have mutually recursive functions or mutually dependent structs. Structs declare the type name and the types and names of each of the fields within. Globals declare the type and name of globally accessible variables. Functions have a name, return type, parameters and locals with types and unique names, and a list of statements. Statements can contain within them other statements, expressions, and (for only assignment statements) an Lvalue.

Static Semantics

There are many stages of type checking. First the type checker validates that all structs have unique names and all fields within a struct have valid types and unique names. Then the parser validates the types and names of the globals. Finally, most of the work is done in checking the functions.

First, the type checker validates that all functions have proper signatures: unique names, valid return types, and valid parameter types. If a function fails any of those then it is "removed" and not type checked. This is mostly because one of the premises for a valid function are wrong: a duplicated name means the second function will never be called, and invalid (non-declared) types in either the parameters or return means this function cannot be called in a valid source file. After checking the signatures, it verifies that there is a function `main` that takes no parameters and returns an int.

For each function it first checks that all local declarations (parameters and local variables) are valid, then it walks through the abstract syntax tree and checks each Lvalue, expression, and statement. Lvalues are just identifiers and member accesses used as targets for assignments, so the check for Lvalue validity is to ensure that a variable of that name exists, and if it can be indexed by any field names given.

Expressions are typed, so when the type checker checks an expression it returns the type of the expression. If the expression is invalid (for example, if the types of math operands are not what are expected) the type checker returns a value that is inherently invalid (null). A container expression will be

invalid if its contained expressions are invalid, so if any subexpressions cause the type checker to return null the type checker will return null for the expression itself.

Statements are not typed, so the type checker returns whether or not the statement is return-equivalent. Most statements are not return-equivalent. Only block statements and if-else statements can be, so they will propagate the return-equivalent flag return value to any containing statements. Ultimately the return-equivalent flag will make it up to the function checker, which will check that a non-void function is actually returning.

To manage keeping track of errors, all error printing is done through a static ErrorPrinter class that counts the number of errors it prints. That way Minic can type-check the file and stop later if any errors were printed.

Intermediate Representation

Within a function, only a few statements can affect the control-flow. Between control-flow statements, all statements are executed in the order they are written with no branches or jumps to interrupt the flow. By generating a control-flow graph of the code, it allows us to perform some analyses that would otherwise be tedious and difficult to do in the AST representation, such as seeing if a function returns on all paths or if a variable has a constant value at its uses.

Compilers use abstract intermediate representations like LLVM because it removes most of the nuances between different machine architectures. LLVM allows us to use an infinite number of registers, allowing us to defer register allocation until a later stage. LLVM also simplifies variable (virtual register) initializations by enforcing static-single assignment. By requiring all variables be defined in only one location it allows the compiler to know that all uses of the same register are the exact same value, perhaps allowing for some optimizations.

Optimizations

Minic does not implement any optimizations aside from a compiling to register-based assembly.

Abstractly interpreting instructions allows the compiler to evaluate constant values once and put those values directly into the executable, speeding up the executable since it will not need to run the same lines again and again to get the same, unchanged values. In a way, constant propagation is like caching values. A constant propagation optimization will not execute the program, so it will not enter functions or perform recursion: it will look at only the block or function level variables. Input values may differ as well, limiting the effectiveness of a constant propagation optimization (although, with no input values and completely deterministic functions, with an aggressive enough optimizer it would be possible to reduce a whole program down to a single print statement).

Code Generation and Register Allocation

When translating from LLVM to ARM most instructions have a one-to-one equivalent (such as arithmetic operations). Some LLVM instructions require more than one ARM instruction to fully translate, but for the most part, each translation is somewhat deterministic. The only LLVM instruction that complicated things was the function call instruction, because of the frame management: unpacking the arguments, pushing some arguments to the stack and moving others into the registers `r0` through `r3`, moving the return value out of `r0` if needed, and popping all the extra arguments after the function call.

The phi instructions are required to allow LLVM to assign one of many values to a virtual register without breaking its enforced SSA. If the value of a register needed to change based on the path it took to get to the point, LLVM would not allow the programmer to define the same register in multiple locations, regardless if they are in the different blocks. As a result, LLVM requires an instruction that selects a return value based on the block that just jumped into the current one. ARM, however, does not have such a feature; fortunately, ARM does not enforce SSA, so as long as the values in registers don't conflict, we can reassign the same register different values in different blocks. This is the strategy Minic uses for "coming out of SSA": for every phi instruction at the beginning of a block, put `mov` instructions at the end of every incoming block.

Other

One small feature of Minic that I think is particularly convenient is the options system to control what stages of the compilation to perform and where to write the files to. By default (no options provided), Minic will write the Assembly to a file with the same name as the input file but with the extension `.s` instead of `.mini`. If you provide the option `-llvm`, it outputs the LLVM to a file of the same name with a `.ll` extension. You can also append an equals sign to print to stdout (`-llvm=` and `-arm=`) or provide the name of a path to save the output to (`-llvm=file.ll`). You can also compile directly with Clang using the `-clang` option. This will produce a file with the same name but with a `.clang` extension.

Analysis

As Minic is able to produce 90% accurate ARM Assembly code, the twenty benchmarks were tested on the Raspberry Pis using four configurations:

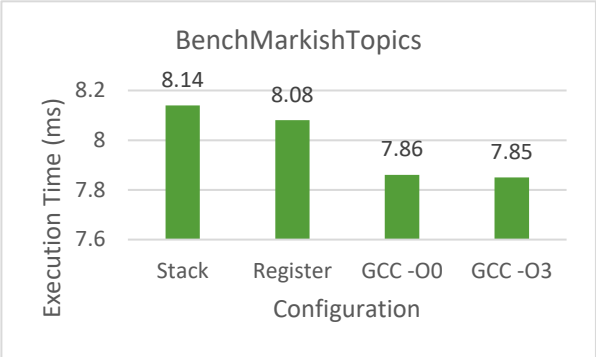
1. **Stack**: Minic-produced stack-based ARM
2. **Register**: Minic-produced register-based ARM
3. **Unoptimized**: GCC-produced executable with optimizations off (`GCC -O0`)
4. **Optimized**: GCC-produced executable with optimizations on (`GCC -O3`)

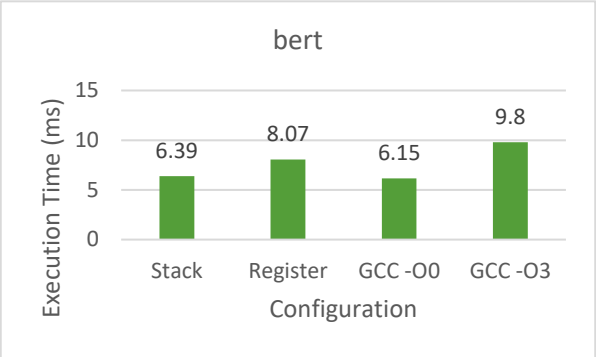
Minic is able to produce valid output for eighteen of the twenty benchmarks. The only two it fails in are stats and killerBubbles. For the stats benchmark, only one line of output is incorrect: the maximum value of the longer input. My suspicion is that there is an issue with the phi instruction in the loop to calculate the min and max values. The killerBubbles output is backwards, in that instead of printing ascending values, the executable created by Minic will output descending values.

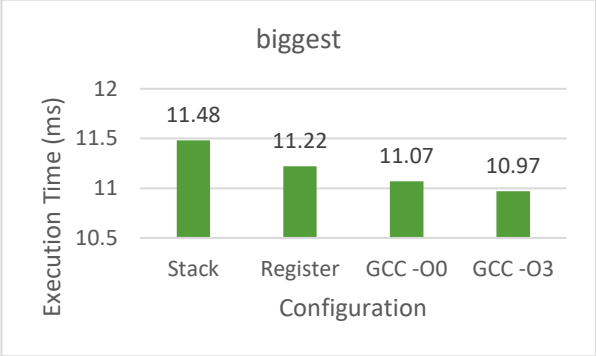
Each benchmark was tested by executing each configuration with the provided input file piped in and output piped to `/dev/null`. To get a more accurate average, I ran each executable one hundred times and ran the GNU `time` utility across the one hundred runs.

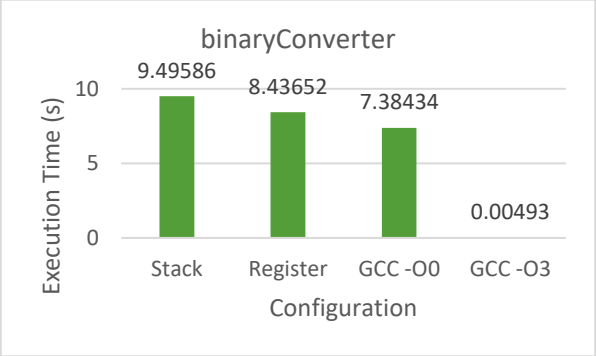
The following pages contain charts, one chart for each benchmark, that display how well the Minic-produced executables compare with the GCC-produced executables. The heights of the bars are the average time taken for each configuration to run; the time axis is scaled to the best magnitude of time to represent the data (seconds or milliseconds). As expected, Stack remained on par with Unoptimized. Surprisingly, Stack managed to beat Optimized in a few benchmarks. Register, unfortunately, did not seem to add as much of a benefit as I was expecting. Only in the bert benchmark did Register beat the optimized GCC executable.

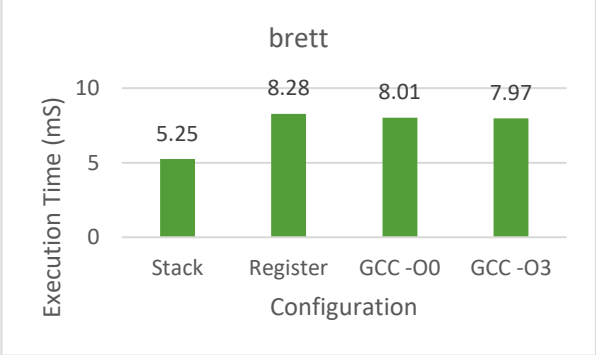
Following the charts is a table of the LLVM instruction counts for each benchmark (labels, function declarations, structs, and other non-instructions are not included). As expected, Register cut the number of instructions needed by a significant amount (averaging around a 46% reduction). Even with the smaller code size, Register still was not significantly faster than Stack.

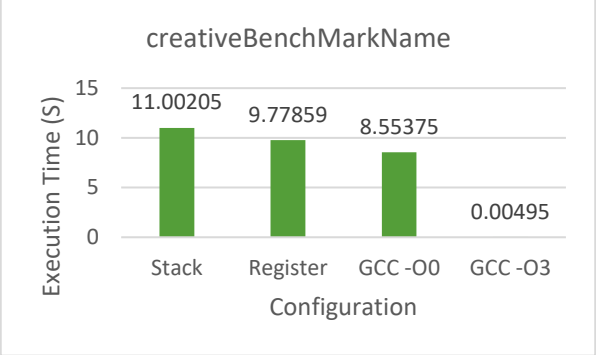

BenchMarkishTopics
Execution Time (ms)
8.2
8
7.8
7.6
8.14
8.08
7.86
7.85
Stack
Register
GCC -O0
GCC -O3
Configuration

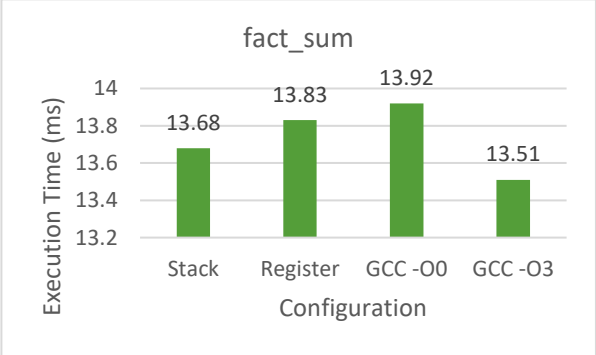

bert
Execution Time (ms)
15
10
5
0
6.39
8.07
6.15
9.8
Stack
Register
GCC -O0
GCC -O3
Configuration

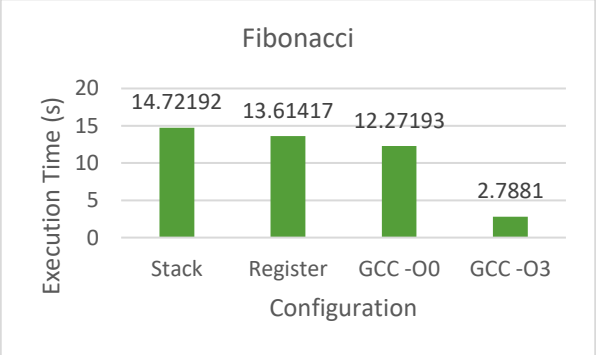

biggest
Execution Time (ms)
12
11.5
11
10.5
11.48
11.22
11.07
10.97
Stack
Register
GCC -O0
GCC -O3
Configuration

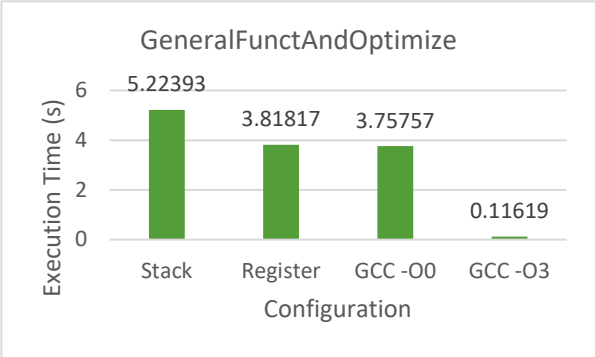

binaryConverter
Execution Time (s)
10
5
0
9.49586
8.43652
7.38434
0.00493
Stack
Register
GCC -O0
GCC -O3
Configuration

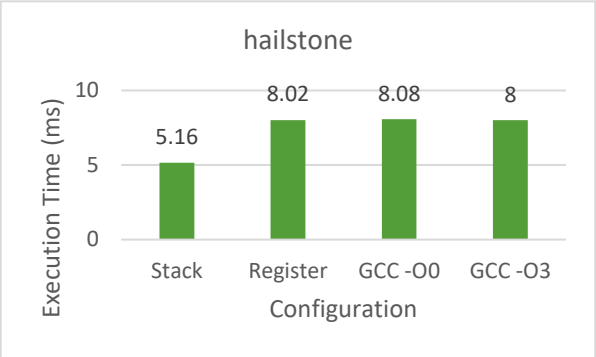

brett
Execution Time (mS)
10
5
0
5.25
8.28
8.01
7.97
Stack
Register
GCC -O0
GCC -O3
Configuration

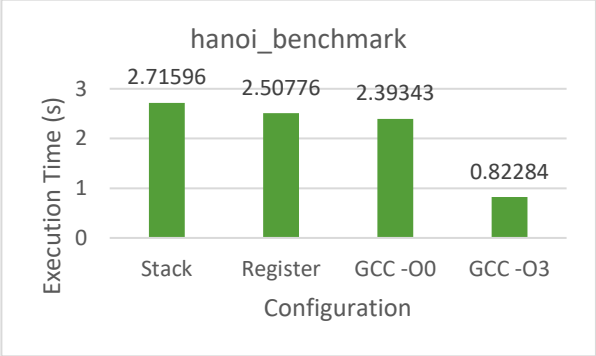

creativeBenchMarkName
Execution Time (S)
15
10
5
0
11.00205
9.77859
8.55375
0.00495
Stack
Register
GCC -O0
GCC -O3
Configuration

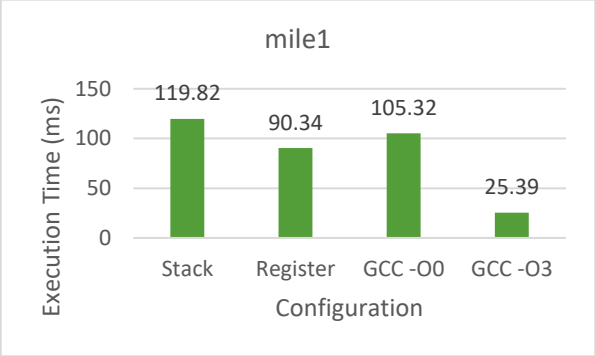

fact_sum
Execution Time (ms)
14
13.8
13.6
13.4
13.2
13.68
13.83
13.92
13.51
Stack
Register
GCC -O0
GCC -O3
Configuration

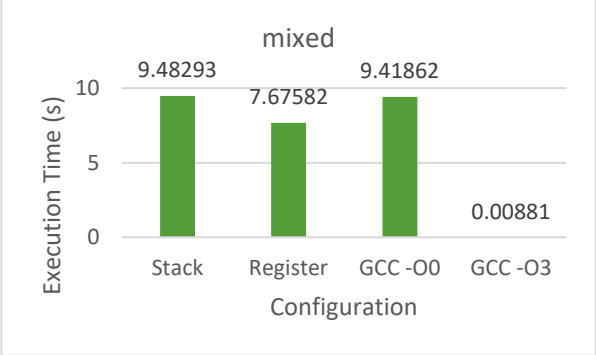

Fibonacci
Execution Time (s)
20
15
10
5
0
14.72192
13.61417
12.27193
2.7881
Stack
Register
GCC -O0
GCC -O3
Configuration

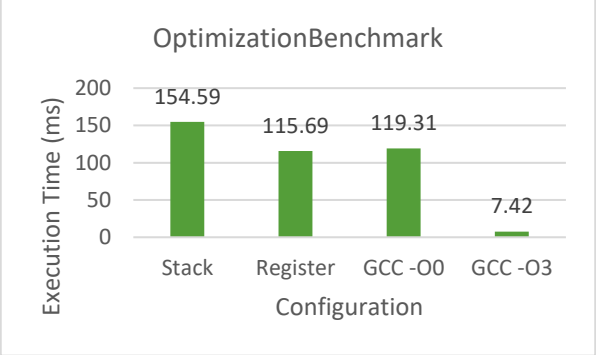

GeneralFunctAndOptimize
Execution Time (s)
6
4
2
0
5.22393
3.81817
3.75757
0.11619
Stack
Register
GCC -O0
GCC -O3
Configuration

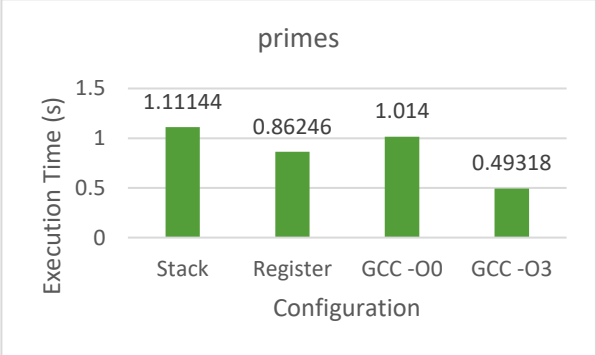

hailstone
Execution Time (ms)
10
5
0
5.16
8.02
8.08
8
Stack
Register
GCC -O0
GCC -O3
Configuration

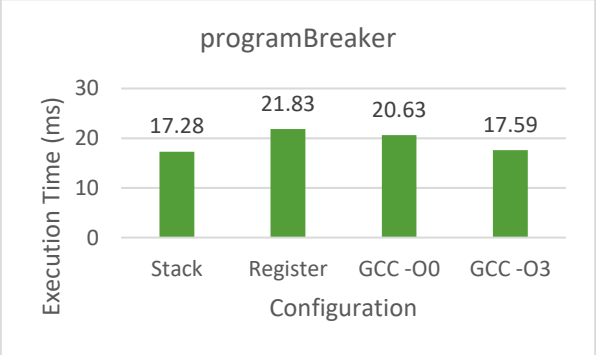

hanoi_benchmark
Execution Time (s)
3
2
1
0
2.71596
2.50776
2.39343
0.82284
Stack
Register
GCC -O0
GCC -O3
Configuration

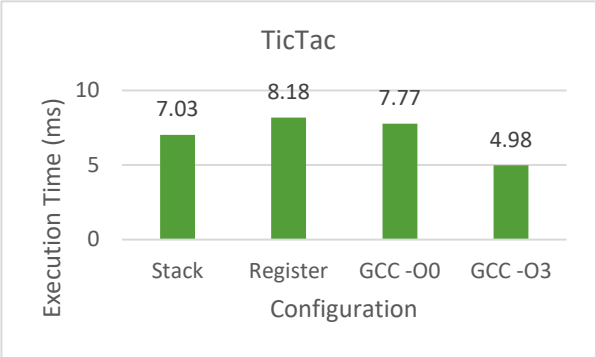

mile1
Execution Time (ms)
150
100
50
0
119.82
90.34
105.32
25.39
Stack
Register
GCC -O0
GCC -O3
Configuration

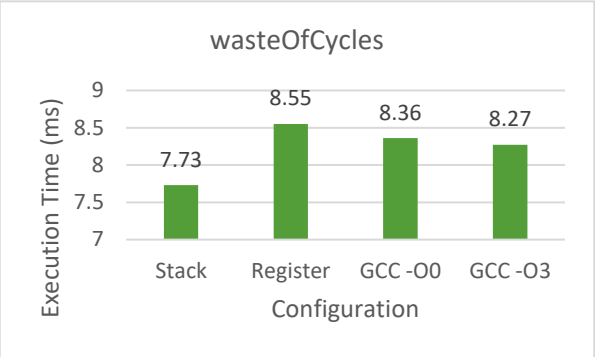

mixed
Execution Time (s)
10
5
0
9.48293
7.67582
9.41862
0.00881
Stack
Register
GCC -O0
GCC -O3
Configuration


OptimizationBenchmark
Execution Time (ms)
200
150
100
50
0
154.59
115.69
119.31
7.42
Stack
Register
GCC -O0
GCC -O3
Configuration


primes
Execution Time (s)
1.5
1
0.5
0
1.11144
0.86246
1.014
0.49318
Stack
Register
GCC -O0
GCC -O3
Configuration


programBreaker
Execution Time (ms)
30
20
10
0
17.28
21.83
20.63
17.59
Stack
Register
GCC -O0
GCC -O3
Configuration





Instruction Counts	Stack	Register	% Reduction
BenchMarkishTopics	128	73	42.96875
bert	650	362	44.30769
biggest	91	44	51.64835
binaryConverter	126	49	61.11111
brett	649	499	23.11248
creativeBenchMarkName	243	128	47.3251
fact_sum	74	34	54.05405
Fibonacci	35	19	45.71429
GeneralFunctAndOptimize	145	90	37.93103
hailstone	51	26	49.01961
hanoi_benchmark	190	124	34.73684
mile1	75	38	49.33333
mixed	207	118	42.99517
OptimizationBenchmark	1018	383	62.37721
primes	92	44	52.17391
programBreaker	82	38	53.65854
TicTac	425	288	32.23529
wasteOfCycles	46	22	52.17391