CS4212 Programming Assignment 3 Report

JULIUS PUTRA TANU SETIAJI (A0149787E)

Contents

Contents 1
1 Compiler Usage 1
1.1 Using GCC to assemble the assembly source code 1
1.2 Using gem5 to run the output binary 2
2 General Compilation Flow 2
2.1 Lexer and Parser 2
2.2 Typechecker and IR3 Codegen 2
2.3 Lower Codegen 3
2.4 Optional Optimisation Passes 7
2.5 Register Allocation 7
2.6 ARM Code Generation 9
2.7 Optional Peephole Optimisation Passes 9
3 Lower IR Optimisations 9
3.1 Algebraic Identities 9
3.2 Deadcode Elimination 10
3.3 Barrel-shifter Operand2 10
3.4 Constant Propagation and Constant Folding 10
4 ARM Code Peephole Optimisations 10
4.1 Algebraic Identities 11
4.2 Useless `MOV` 11
4.3 Useless `LDR` 11
4.4 Remove `MOV` to itself 11
4.5 Consecutive `MOV` of immediate values to the same register 11
5 Additional Notes 11

1 COMPILER USAGE

To compile a JLite source code into ARM assembly code without optimisation, run:

```
./jlitec arm <input file>
```

To compile a JLite source code into ARM assembly code with optimisations run on the Lower IR as well as peephole optimisations on the ARM code, run:

```
./jlitec arm -O3 <input file>
```

To compile a JLite source code into ARM assembly code with optimisations run on the Lower IR but without the peephole optimisations on the ARM code, run:

```
./jlitec arm -O2 <input file>
```

1.1 Using GCC to assemble the assembly source code

Redirect the output of my compiler into a file, and then pass this file to a GCC (cross) compiler for ARM. For example, if your input file is `a.j`, in `bash`, run:

```
./jlitec arm a.j > /tmp/a.s
case "$(uname -s)" in
  Linux*)  CC_ARM=arm-linux-gnueabihf-gcc;;
  Darwin*) CC_ARM=arm-unknown-linux-gnueabi-gcc;;
esac
[[ CC_ARM != "" ]] && $CC_ARM /tmp/a.s --static || echo Unknown platform
```

1.2 Using gem5 to run the output binary

Assuming the output binary is in `/tmp/a.out`, change directory to where gem5 is located, then run:

```
./build/ARM/gem5.opt configs/example/se.py -c /tmp/a.out -i /dev/tty
```

By specifying input file as `/dev/tty`, you can provide input to the resulting binary from `stdin` for any `readln` uses.

2 GENERAL COMPILATION FLOW

Besides IR3, I defined my own machine-specific intermediate representation (IR) for use in my compiler, called Lower. With reference to GCC, IR3 is similar to GIMPLE, while Lower is similar to RTL.

My compiler can be divided into several parts:

(1) Lexer
(2) Parser
(3) Typechecker
(4) IR3 Codegen
(5) Lower Codegen
(6) Optional optimisation passes
(7) Register Allocation
(8) ARM code generation
(9) Optional peephole optimisation passes

2.1 Lexer and Parser

Lexer and parser were done as part of Programming Assignment 1.

Lexing is done using the lexer generator JFlex. It was relatively straightforward as the tokens in JLite is a subset of the tokens in Java, and JFlex provided a sample Java lexer. The one difference between Java tokens are JLite tokens is the liberty that I took to disallow representing the empty string as `""`, and instead forcing the use of `null` to represent empty string.

Parsing is done using the LALR parser generator CUP. This part is more complicated as the provided grammar is ambiguous, due to the type-specificity of each operator, the overloading of the operator + to represent both arithmetic addition and string concatenation, and the associativity ambiguity of the string concatenation operator. Thus, I specified a less strict grammar rules, and verified the abstract syntax tree during construction in order to make sure it follows the full JLite grammar rules.

The generated abstract syntax tree is then passed on to the typechecker.

2.2 Typechecker and IR3 Codegen

Typechecker and IR3 code generation were done as part of Programming Assignment 1.

Typechecking is performed on the abstract syntax tree recursively, and the types are attached to the abstract syntax tree. My compiler supports function overloading based on types of the parameters of a method, together with a check that there is no two method with duplicate signature (defined as the types of the parameters). The method resolution is then based on the types of the parameters to yield the correct return type for the method.

Generating IR3 from the abstract syntax tree is quite trivial through recursion with the information yielded from the typechecker, i.e. the type information and the method resolution information. IR3 does not have a notion of classes anymore, only structs and functions that act on structs. The generated IR3 code is then passed on to the Lower codegen.

2.3 Lower Codegen

Lower is my machine-specific where I perform all the passes on. Unlike Jlite and IR3, Lower does not have any notion of expression, only statements. These statements are designed to very closely match ARM instructions, while still maintaining IR3 semantics as much as possible to still allow reasonable optimization at this IR level. Lower also allows statements to make use of real physical hardware ARM registers, besides variables. The struct `Addressable` in Lower is designed to be the representation of a "register", both real physical hardware registers as well as pseudo-registers (which program variables are).

The pass that lowers IR3 to Lower is the file `LowerPass.java`.

There are many types of statements in Lower (located in `src/main/java/jlitec/backend/passes/lower/stmt/`):

- `BINARY`, representing instructions in ARM whose operands are one register and one Operand2 that is either Immediate or Register without barrel-shifter operation.
- `BINARY_BIT`, representing instructions in ARM whose operands are one register and one Register with barrel-shifter operation.
- `BIT`, representing barrel-shifter instructions in ARM.
- `BRANCH_LINK`, representing the `BL` instruction.
- `CMP`, representing an if-goto instruction (`CMP` followed by conditional `B` instructions).
- `FIELD_ACCESS`, representing a field access (`LDR` instruction).
- `FIELD_ASSIGN`, representing a field assignment (`STR` instruction).
- `GOTO`, representing unconditional branching (unconditional `B` instruction).
- `IMMEDIATE`, representing assignment into a register from an immediate value (either `MOV` instruction with an immediate operand2, or `LDR` from a constant pool, as is the case for literal strings).
- `LABEL`, representing a label.
- `LOAD_STACK_ARG`, representing `LDR` instruction from the stack for a function argument.
- `LOAD_LARGE_IMM`, representing assignment into a register from the constant pool for a large immediate value.
- `LDR_SPILL`, representing `LDR` instruction for a spilled variable.
- `STR_SPILL`, representing `STR` instruction for a spilled variable.
- `MOV`, representing assignment from one register into another (`MOV` instruction with two register operands).
- `RETURN`, representing returning from the current function (either `BX LR` instruction or `LDMFD SP!, {..., PC}`).
- `PUSH_PAD_STACK`, representing `SUB SP, SP, #4` to pad the stack by 4 bytes to maintain 8-byte alignment as required by the ARM calling convention.

- `PUSH_STACK`, representing `PUSH` pseudo-instruction (`STR Rs, [SP, #-4]!`) to push a register into the stack, which also updates the `SP` register.
- `POP_STACK`, representing popping the stack (`ADD SP, SP, #<number of bytes>`).
- `REG_BINARY`, representing instructions in ARM whose operands are 2 registers (`MUL` and `SDIV`).
- `REVERSE_SUBTRACT`, representing the `RSB` instruction whose operand2 is either an Immediate value or Register without barrel-shifter operation.
- `REVERSE_SUBTRACT_BIT`, representing the `RSB` instruction whose operand2 is a Register with barrel-shifter operation.
- `UNARY`, representing either the `MVN` instruction or `NEG` pseudo-instruction (`RSB Rd, Rs, #0`).

There could be some IR3 operations that do not fulfill the requirements of any Lower statement, e.g. `a = b * 3;` since the `MUL` instruction as represented by `REG_BINARY` Lower statement requires both operands to be registers. The pass that lowers IR3 to Lower will create new temporaries as required. The purpose of Lower is to make ARM code generation trivial given register allocation information.

`new` statements, `println` statements, `readln` statements, and string concatenation operations are also lowered into either a call into a helper function, or the corresponding `libc` functions:

- **new** statements are implemented by calling `calloc` with the correct number of bytes based on the number of fields of the class. Note that `calloc` is used here instead of `malloc` in order to perform the "shallow" initialization specified in Programming Assignment 1, since `calloc` will fill the memory regions allocated with zero bytes, while `malloc` makes no such promises. The types of the fields do not matter as all types in JLite are uniformly 32 bits (4 bytes):
  - `String` is represented as a pointer to the first element of a null-terminated array of characters. Since ARM is an ISA for a 32-bit machine, pointer lengths are 32-bits.
  - `Int` is represented as a 32-bit integer as we are using a 32-bit machine.
  - `Bool` is represented as an `Int` whose value is 1 for true, and `0` for false.
  - Classes are represented as a pointer, which in a 32-bit machine is 32-bits long.
- **println** statements are implemented by:
  - `String`: calling `puts(string);`.
  - `Int`: calling `printf("%d", integer);`.
  - `Bool`: calling helper function `println_bool(bool);`.
- **readln** statements are implemented by:
  - `Int` and `Bool`: calling `readln_int_bool();` and using the result.
  - `String`: calling `getline_without_newline();` and using the result.
- **String concatentation** is implemented by using `strlen` to find out the lengths of the two string, then `malloc` to allocate a buffer for the resulting string, then `strcpy` from the first string to the resulting string, followed by `strcat` from the second string to the resulting string. Note that it is safe to use `malloc` here as opposed to the use of `calloc` for **new** statements, as the allocated memory buffer will be immediately overwritten with `strcpy` and `strcat`.

Note that since we do not have a garbage collector, currently memory allocated for strings and classes are just leaked when they go out of scope.

Helper function `println_bool(bool);` prints either `true` or `false` depending on the boolean value. The implementation in C and ARM assembly:

```c
void println_bool(int a)
{
  puts(a == 1 ? "true" : "false");
}
```

```
println_bool:
LDR R2, .TRUE
LDR R1, .FALSE
CMP R0, #1
MOVEQ R1, R2
MOV R0, R1
b puts
```

Helper function `readln_int();` gets one whole new line and read an integer from it. The implementation in C and ARM assembly:

```
int readln_int()
{
  int a;
  char* result = NULL;
  size_t n = 0;
  getline(&result, &n, stdin);
  sscanf(result, "%d", &a);
  free(result);
  return a;
}
```

```
readln_int:
STMFD SP!, {R4, LR}
SUB SP, SP, #16
MOV R0, #0
STR R0, [SP, #8]
STR R0, [SP, #4]
LDR R0, .Lstdin
LDR R2, [R0]
ADD R0, SP, #8
ADD R1, SP, #4
BL getline
LDR R0, [SP, #8]
LDR R1, .PERCENTD
ADD R2, SP, #12
BL sscanf
LDR R0, [SP, #8]
BL free
LDR R0, [SP, #12]
ADD SP, SP, #16
LDMFD SP!, {R4, PC}
```

Helper function `readln_bool();` gets one whole new line and returns true (1) if and only if the input starts with the substring `"true"`, otherwise the function returns false (0). The implementation in C and ARM assembly:

```
bool readln_bool()
{
  bool a;
  char* result = NULL;
  size_t n = 0;
  getline(&result, &n, stdin);
  a = strncmp(result, "true", 4) == 0;
  free(result);
  return a;
}
```

```
readln_bool:
STMFD SP!, {R4, R5, R11, LR}
SUB SP, SP, #8
MOV R0, #0
STR R0, [SP, #4]
STR R0, [SP]
LDR R0, .Lstdin
LDR R2, [R0]
MOV R1, SP
ADD R0, SP, #4
BL getline
LDR R4, [SP, #4]
LDR R1, .TRUE
MOV R2, #4
MOV R0, R4
BL strncmp
MOV R5, R0
MOV R0, R4
BL free
RSBS R0, R5, #0
ADC R0, R5, R0
ADD SP, SP, #8
LDMFD SP!, {R4, R5, R11, PC}
```

Helper function `getline_without_newline()`; gets one whole new line and removes the newline from it. The implementation in C and ARM assembly:

```
char* getline_without_newline()
{
  char* result = NULL;
  size_t n = 0;
  ssize_t len = getline(&result, &n, stdin);
  result[len - 1] = 0;
  return realloc(result, len - 1);
}
```

```
getline_without_newline:
STMFD SP!, {R4, LR}
SUB SP, SP, #8
MOV R4, #0
STR R4, [SP]
STR R4, [SP, #4]
MOV R0, SP
ADD R1, R0, #4
LDR R3, .Lstdin
LDR R2, [R3]
BL getline
SUB R1, R0, #1
LDR R0, [SP]
```

```
STRB R4, [R0, R1]
BL realloc
ADD SP, SP, #8
LDMFD SP!, {R4, PC}
```

2.4 Optional Optimisation Passes

There are a couple of optimisations that I have implemented on the Lower IR. I will expound on this in a later section.

2.5 Register Allocation

My register allocation implementation is located in the file `RegAllocPass.java`. Live dataflow analysis implementation is located in file `LivePass.java`, while basic block analysis is located in file `FlowPass.java`.

Register allocation is performed on the Lower IR based on graph colouring of the interference graph, created using liveness dataflow analysis. Performing graph colouring based on the Lower IR rather than IR3 allows for more optimal register allocation. For example, variables that are not live across function calls can be allocated to registers `R0` to `R3`, as well as registers `R12` and `LR` (as long as `LR` is saved). Note that register `R12` is not safe to use across function calls as it is used as an Intra-Procedure-call scratch register.

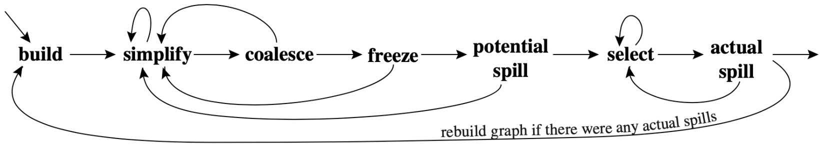

Fig. 1. Chaitin-Briggs-George-Appel register allocation algorithm based on graph-colouring

The register allocation method used is Chaitin-Briggs-George-Appel algorithm based on Kempe's graph-colouring algorithm. Let K be the number of available physical hardware register. The steps are as can be seen from Figure 1:

- **Build**: construct the interference graph, and categorise each pseudo-register as either move-related or non-move-related.
- **Simplify**: one at a time, remove non-move-related nodes of low-degree ($< K$) from the graph, similar to Kempe's graph-colouring algorithm.
- **Coalesce**: perform conservative coalescing on the reduced graph obtained in the simplification phase. Conservative coalescing means that nodes are only coalesced if it will not turn a K-colourable graph uncolourable. Here we use two heuristics for conservative coalescing:
  - Briggs: two nodes can be coalesced if the resulting node will have fewer than K neighbours of significant degree (having $\geq K$ edges).
  - George: Nodes a and b can be coalesced if, for every neighbour t of a, either t already interferes with b or t is of insignificant degree.
- **Freeze**: if neither simplify nor coalesce applies, freeze all the low-degree move-related nodes – this means that they are considered to be non-move-related, which enables more simplication. After this, simplify and coalesce are resumed
- **Potential Spill**: If there are no low-degree nodes, select a significant-degree node for potential spilling and push it on the stack.
- **Select**: Pop the entire stack, assigning colours if possible.

- **Actual spill**: If there is a node for which no colour can be applied during select, it is an actual spill. The program is rewritten with the spill, and the whole process is started from scratch.

Spilling priority is determined by the spilling priority score, where lower scores are more likely to be spilled. The heuristic used is score = (number of defs and uses) ÷ degree. This is because we want to spill variables that are least frequently defined and used, as well as those that interferes with many other variables. Temporaries created in IR3/Lower is given higher scores than user-defined variables, and spilled temporaries are given even higher scores then temporaries created in IR3/Lower.

Note that some operands of Lower statements are actual physical hardware registers, which means that they are precoloured in terms of the graph colouring problem. Using conservative coalescing, these registers can be coalesced with other pseudo-registers, resulting in a more optimal register allocation. Note that care should also be taken to ensure short live-range of physical hardware registers, as they cannot be spilled into memory, unlike pseudo-registers.

Another interesting implementation is on registers `R0` to `R3` used in the calling convention. Since they are caller-saved, their contents are clobbered after a function call. This can be indicated in the interference graph by making these registers interfere with one another, as well as with variables that are live across function calls. Implementation-wise, this is done by making `BRANCH_LINK` statement use registers `R0` to `R3` and defines (clobbers) registers `R0` to `R3` as well as `R12` and `LR`.

The standard convention call is represented in Lower using `MOV` and `IMMEDIATE` statements to move the parameters into registers `R0` to `R3`, as well pushing the rest of the parameters into stack if need be, then a `BRANCH_LINK` statement, followed by a `POP_STACK` statement if there were parameters on the stack, and finally, a `MOV` statement to move the result of the function call to a variable if the result is used.

For example, take the following JLite statements (with the class signature in the comment):

```
/* class Func {
 *   Func f(Int, Int, Int, Int, Int, Int); // mangled as Func_0
 *   Void g();                             // mangled as Func_1
 * }
 */
Func a; Int b; Int c; Int d; Int e; Int f;
a.f(b, 5, c, d, e, f).g();
// a, c, d, e, f are live here
```

Lowered to IR3, the equivalent statements are:

```
Func _t1;
_t1 = %Func_0(a, b, 5, c, d, e, f);
%Func_1(_t1);
// a, c, d, e, f are live here
```

Lowered to Lower, the equivalent statements are:

```
R0 <- a;      // MOV
R1 <- b;      // MOV
R2 = 5;       // IMMEDIATE
R3 <- c;      // MOV
PUSH (PAD);   // PUSH_STACK_PAD
PUSH f;       // PUSH_STACK
PUSH e;       // PUSH_STACK
PUSH d;       // PUSH_STACK
CALL Func_0; // BRANCH_LINK
```

```
POP 4;       // POP_STACK
_t1 <- R0;   // MOV
R0 <- _t1;   // MOV
CALL Func_1; // BRANCH_LINK
// a, c, d, e, f are live here
```

Assuming that result of the register allocation is that a is assigned to register R4, b to register R1, c to register R5, d to register R6, e to register R7, f to register R8, _t1 to register R0, then the generated ARM code is:

```
MOV R0, R4
MOV R2, #5
MOV R3, R5
SUB SP, SP, #4
STR R8, [SP, #4]!
STR R7, [SP, #4]!
STR R6, [SP, #4]!
BL Func_0
ADD SP, SP, #16
BL Func_1
```

Observe that MOV to the same registers are omitted, as this is the result of the coalescing phase done in the register allocator. With the liveness information used to construct the interference graph, the register allocator is able to tell that variables b and _t1 do not live across function calls. Thus, the register allocator is able to make a more optimal choice of assigning b to R1, given that it is the second parameter to Func_0, while _t1 is assigned to R0 as it is the first parameter of Func_1.

2.6 ARM Code Generation

My code for the ARM Code Generation is located in Global.java.

With register allocation from the register allocator, the only complicated thing for ARM code generation from Lower IR is the calculation of offsets of spilled variables and stack arguments. This is especially since the ARM calling convention specifies that stack must be 8-byte aligned, thus my compiler needs to add the appropriate padding to align the stack.

The rest of it is trivial, as most types of Lower statements correspond with exactly one ARM instruction, except for CMP, and comparison operators in BINARY and BINARY_BIT.

One addition that I performed was to make the main function return 0, to conform to the Unix convention.

2.7 Optional Peephole Optimisation Passes

There are a couple of optimisations that I have implemented for the ARM code generated. I will expound on this in a later section.

3 LOWER IR OPTIMISATIONS

A couple of optimisations can be performed on the Lower IR. These optimisations are run continuously until 2 runs produce the same Lower IR.

3.1 Algebraic Identities

The file for this is AlgebraicPass.java.

We make use of the following algebraic identities:

- $x + 0 = 0 + x = x$ and $x - 0 = x$.

- $x \wedge true = x$
- $x \wedge false = false$
- $x \vee true = true$
- $x \vee false = x$

3.2 Deadcode Elimination

The file for this is `DeadcodeOptimizationPass.java`. This optimization also makes use of live dataflow analysis implementation, located in file `LivePass.java`, and basic block analysis is located in file `FlowPass.java`.

There are a couple of dead code types that are detected in this pass:

- **Basic Block**: basic blocks with in-degrees = 0 except for the entry node is removed.
- **Uses**: statements that defines dead variables are removed.
- **Dead Gotos**: delete goto statemens pointing to a label that follows the goto statement directly, i.e. `GOTO L2; L2:`, and rather use the fallthrough behaviour.
- **Mov to itself**: delete mov statements with the same source and destination.
- **Remove useless labels**: remove labels that are not the destination of any goto or if-goto statements.
- **Remove double goto**: remove labels whose content is another goto statement, and ensuring all gotos to the removed label go to the destination of the goto statement in the removed label.
- **Replace label return**: Replace gotos where the destination only contains returns.

3.3 Barrel-shifter Operand2

Replace the sequence of `BIT` followed by `BINARY` with `BINARY_BIT`, and `BIT` followed by `REVERSE_SUBTRACT` with `REVERSE_SUBTRACT_BIT`. If the destination of the `BIT` statement is not used anymore anywhere else, then the **Deadcode Elimination** pass will pick it up to remove the `BIT` statements.

3.4 Constant Propagation and Constant Folding

The file for this is `ConstantFoldingOptimizationPass.java` and `ReachingPass.java`.

Using reaching definition dataflow analysis, resolve the value of variables into a constant. If all operands in a statement is identified, then perform constant folding: do the operation during compile time instead and turn the statement into an `IMMEDIATE` statement with the result of the operation.

This works wonderfully on the provided `test_booleans.j` and `test_ops.j` sample input files provided, and we are left only with print statements containing the results.

There are 2 additional passes performed on the resulting constant-propagated and constant-folded program:

- **Handling large immediate**: operand2 immediate values are limited to 8-bit values rotated by an even number of bits. Any immediate outside of that range must be loaded from a constant pool. This is handled by replacing those with `LOAD_LARGE_IMM` Lower statement.
- **Inlining of println_bool**: if the parameter to `println_bool` is constant-folded and known at compile-time, then inline the helper function to either `puts("true");` or `puts("false");`.

4 ARM CODE PEEPHOLE OPTIMISATIONS

There are a couple of peephole optimisations that can be performed on the resulting ARM code.

4.1 Algebraic Identities

4.1.1 Addition and subtraction with zero. Replace `ADD R0, R1, #0` or `SUB R0, R1, #0` with `MOV R0, R1`.

4.1.2 And with true. Replace `AND R0, R1, #1` with `MOV R0, R1`.

4.1.3 And with false. Replace `AND R0, R1, #0` with `MOV R0, #0`.

4.1.4 Or with true. Replace `ORR R0, R1, #1` with `MOV R0, #1`.

4.1.5 Or with false. Replace `ORR R0, R1, #0` with `MOV R0, R1`.

4.2 Useless MOV

Replace:

```
MOV R5, R0
MOV R0, R5
```

with:

```
MOV R5, R0
```

Note that we cannot remove `MOV R5, R0` entirely as the value of `R5` might be used later on, for example, to ensure a variable survives a function call.

4.3 Useless LDR

Replace:

```
STR R0, [SP]
LDR R0, [SP]
```

with:

```
STR R0, [SP]
```

Note that we cannot remove `STR R0, [SP]` entirely as that value in memory might be loaded later on.

4.4 Remove MOV to itself

Delete:

```
MOV R0, R0
```

4.5 Consecutive MOV of immediate values to the same register

Replace:

```
MOV R0, #5
MOV R0, #1
```

with:

```
MOV R0, #1
```

5 ADDITIONAL NOTES

Possible future work includes:

- Sub-expression elimination, which is probably best done on the IR3 level.
- Better spilling heuristic by changing the weight of the number of defs and uses inside a loop as opposed to outside a loop, for example $score = ((\text{number of defs and uses outside loops}) + 10 \times (\text{number of defs and uses inside loops})) \div \text{degree}$.

- Converting into Single Static Assignment form to unlock further optimisation, for example with sparse conditional constant propagation.

Just for fun, I also implemented a C backend. The output can then be passed to any C compiler. This can be invoked by running `./jlitec c <input file>`.