National University of Singapore
CS4212 Project Assignment 3

AY 2021/2022 Semester 1

Due Date: 21 November 2021 (Sunday, 23:59 Hrs)

This is a hard deadline with no possibility of extension due to marks submission deadline.

In this final assignment, you are required to construct the back-end of the compiler for JLite. You have already translated valid JLite programs into three-address code in *IR3* representation in Assignment 2. In this assignment, you will be required to produce two artifacts:

1. **You need to produce a** JLite **compiler.** Your starting point is your submission for *Assignment 2*. You need to complete the compiler implementation by adding its back-end. Specifically, you are required to translate *IR3*-programs (corresponding to JLite programs) into ARM assembly code.

   There is no model solution for Assignment 2 to start this assignment.

2. **You need to write a report.** This report describes the design of your JLite compiler. It should explain how you implemented all phases of the compiler (front-end, middle-end, back-end), any difficulties you have run into, and any optimizations you might have implemented. Make sure to explain your register allocation strategy.

Assessment

- **Code**: Your submission needs to pass a benchmark set of tests, like the previous assignments.
- **Design**: Your report should explain the design of your JLite compiler and how you implemented various features of the language.
- **Marks**: Your mark is primarily based on the correctness of the programs produced by your compiler. The effectiveness of your register allocation scheme will also be assessed, but with a lower weight. Any other **optimization** will be for *bonus points.*

1 Resources and Expected Implementations

You are given the following resources:

1. Instructions on how to set up your machine to run ARM assembly programs. This is provided in the PDF for **Tutorial 7**.
2. A subfolder named `testcases` that contains some JLite programs to be accepted by the JLite compiler, and their corresponding ARM code. Please note that the ARM code is only meant as a reference, not a benchmark. With a different register allocation strategy and optimizations, the ARM code produced by your compiler might look different.

In constructing the back-end compiler, you are required to implement the following:

1. Generation of offsets for each variable access;
2. Smart algorithms for register allocation and assignment. Your grade will partially depend on the efficiency of register allocation and assignment.
3. An option for users to turn on and off the set of optimizations you have implemented. (Note: make it possible to turn on/off all optimizations at once. We will not evaluate individual optimizations. **By default, optimizations should be turned off.**) The number of *bonus points* awarded will depend on the efficiency and effectiveness of these implemented optimizations.

As the ARM code that you are about the handle represents a subset of the actual ARM assembly instruction set, we include two simplifications in JLite programs to be compiled (which you are free to ignore, but without bonus points rewarded if you do):

1. You are not required to handle JLite programs that involve any division operation. The version of ARM assembly that we use does not have a native integer division instruction and we do not want to consider floating-point instructions. (If you want, you can emulate division in software, but you are not required to.)
2. You are not required to handle JLite programs that use the `readln` or `println` statements with objects as the argument, nor are you required to be able to read strings. This avoids trouble with having to define a way to represent objects as strings and with memory allocation.

   You should still support reading integers and printing integers and strings. You can implement this by generating a call to `scanf()` or `printf()` with the appropriate format string whenever you encounter a `readln` or `println` statement. Given the type of the variable or expression you are reading/printing, you can infer what format specifier to use.
3. Use C's `malloc()` to create new objects dynamically when needed. Try to `free()` them when appropriate.
4. You are not required to handle JLite programs that perform string concatenation. This opens a can of worms requiring you to perform memory management where you cannot determine at compile time how much memory to allocate.

2 Delivery

2.1 Suggestions

1. Start early. There is quite a bit of work to be done, more than it initially appears.
2. Start slow. Look through the testcases to see how a working ARM assembly program should look like.
3. Start simple. Generate simple, working assembly code for very small toy JLite programs. For example, no control flow, just straight line code. When you have something working, proceed with control flow. Finally, procedure calls and stack maintenance.

4. The tricky part is register allocation. For that you need to compute the live range of the 3-address code variables using the technique we discussed in class. One simple strategy to ease the implementation is to allocate a "home" location for the whole set of ARM registers in every stack frame. By offsetting the frame pointer, you can easily spill whichever register you need to spill. Again, get a simple scheme (like `GETREG`) to work first before you try advanced solutions.
5. Use GCC and QEMU to *test* the programs you generate. That you can produce well-formed ARM assembly does not mean that you are compiling programs correctly. In particular, procedure calls and stack maintenance are easy to get wrong, such that, e.g., nested calls crash your program.
6. You might find it helpful to install `gdb-multiarch` and use `qemu-arm -singlestep -g 1234 ./a.out` to start a remote debugging session of your program. You connect to the session from GDB by running `gdb-multiarch` and then typing `file a.out` followed by `target remote localhost:1234`. Using GDB's `layout regs` might help debugging stack maintenance issues.

2.2 Submission of your product

Please submit a ZIP archive, with your name and student ID as part of the ZIP's filename, containing the following files to `LumiNUS CS4212` website under the "`Project Assignment 3 Submissions`" folder.

1. The ZIP file should at the top level contain a single folder bearing your name and student ID. In that folder should be your complete project sources, including test files and expected output for these tests, as well as your project report in PDF format.
2. We will test your submission using Python 3.8.10 and the ARM GCC cross compiler toolchain, so you should ensure your code works with this software without requiring any changes, or additional package installations. Your sources should include the following files:
   (a) The files from the previous assignments;
   (b) `compile.py` – the main compiler program, which runs the entire compilation pipeline
   Invoking `./compile.py program.j` should print the resulting ARM assembly to stdout and also write a `program.s` file to disk in the directory the script is invoked from.
   The assembly files you generate should compile with the `arm-linux-gnueabi-gcc-10 program.s --static` command without requiring any modifications.
   (c) You can structure the rest of your compiler back-end in any way you find reasonable, as long as you describe the file structure in your report.
3. A subfolder called `test/` within the main folder. This should include at least 6 pairs of files (created by you) to test your code. Each pair consists of `TESTNAME.j` and `TESTNAME.s.gold`, where `TESTNAME.j` is the program to be compiled and `TESTNAME.s.gold` is the expected (correct) output from your compiler on that file. If you implement any optimizations, also include a `TESTNAME.s.opt` file for each test case, show-casing how your compiler optimises that program.
4. The main folder should include a document named `[YourFullName]_pa3_report.pdf` containing your **project report** that describes the design of your compiler, how your register allocation scheme works, the file structure of your project, what (if any) optimizations you implemented, and anything else you might want to share.

```
 1.    .data
 2. L1:
 3.    .asciz "Hello World"
 4.
 5.    .text
 6.    .global main
 7.    .type main, %function
 8. main:
 9.      stmfd sp!,{fp,lr,v1,v2,v3,v4,v5}
10.      add fp,sp,#24
11.      sub sp,fp,#32
12.      ldr a1,=L1
13.      bl printf(PLT)
14.
15. .L1exit:
16.      mov a4,#0
17.      mov a1,r3
18.      sub sp,fp,#24
19.      ldmfd sp!,{fp,pc,v1,v2,v3,v4,v5}
```

Figure 1: Hello World in ARM Assembler

A Information about ARM Assembly Code

A.1 Organization of ARM Assembly Code

An ARM assembly program is composed of a sequence of instructions. A simple "Hello World" program is shown in Figure A.1. The first part of the code (Lines 1-3) is optional and represents the data section of the program. The beginning of this section is denoted by the .data directive at Line 1. Line 5 denotes the start of the code section. Line 6 is mandatory and denotes the variable to be exported by the code. Line 7 sets the type of the exported variable as a function. Lines 8-19 represent the code section which is composed of the instructions for the main function.

A.1.1 Execution Environment

In order to understand the instructions in the code section you must first understand ARM registers and memory management. The memory model for an ARM program is composed of registers, the stack, and the heap. The registers are used to hold values which are operated on by general data processing instructions such as arithmetic operations, boolean operations, compare operations and so on. As ARM is a RISC load and store architecture, before using a value stored in memory, this value must first be loaded into a register. After updating the value the memory can be updated to the new value by a store operation.

Register	Name	Role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

Table 1: ARM registers

The stack is used to store local variables and temporary computations for functions as well as to create temporary register workspace for subroutines. Any registers that are needed can be pushed onto the stack at the start of the subroutine and popped off again at the end so as to restore them before return to the caller. The heap is used for dynamic data such as objects. Allocating objects on the stack can cause problems if objects are passed as arguments between functions. The memory is byte addressable with word (four bytes) addresses starting at multiples of four. In terms of registers, ARM has 15 general purpose registers in total, all of which are 32-bits long. These registers and their purpose are listed in Table 1.

A.1.2 Calling sequence

ARM defines a set of rules for function entry and exit so that:

- Object code generated by different compilers can be linked
- Procedures can be called between high-level languages and assembly languages.

These rules define:

- The mechanism for argument passing
- The usage of registers
- The usage of the stack
- The format of the stack-based data structure

The mechanism for argument passing involves passing the first four arguments in register a1-a4 and placing the rest, if the called function defines more than 5 parameters, on top of the stack. The return value in then placed in a1.

In terms of registers usage we can broadly classify all registers in two categories:

- Caller-saved that may change during a function call and thus the values stored in them prior to the method call must be saved by the caller before executing the method call.
- Callee-saved that must return unchanged after executing the body of a function.

The first category, caller-saved, contains registers r0-r3, or a1-a4. These registers, as the names in Table 1 suggest, are used to pass the first four arguments to a method call. Register a1 is also used to return the result. Lines 12 and 13 in the hello world example show a method call to a printf function and the passing of argument 1 in register a1. As registers a1 to a4 do not contain any values prior to the method call they are not saved by the caller before executing the call.

The second category, callee-saved, contains registers r4-r15. Registers r4-r8, also called v1-v5, are register variables that can be used as scratch registers to perform computations. Registers r9-r15 are registers for special purposes which can also be used as temporary variables if saved properly.

As callee-saved registers must be returned unchanged by a function they are saved in the prologue of a function and restored in the epilogue of the function. Line 9 in Figure A.1 saves registers v1-v5 and fp and lr using the instruction stmfd. The d suffix of stmfd denotes that saving is based on a descending stack which involves that the stack grows to small addresses (descending). The f in stmfd denotes that the stack pointer points to the last full location (full). The instruction at line 9 also post-updates the stack pointer (sp). Line 10 sets the frame pointer to point to the beginning of the stack frame for the function. Lines 18-19 restore the callee saved registers to their initial values. Line 18 sets the stack pointer to the top address (smallest address) pointing to a register saved in the epilogue. The ldmfd instruction at line 19, the reverse of stmfd, then loads from a full descending stack values into registers v1-v5 and fp and pc. By moving the value in lr (link register) in the pc register the control will be returned to the caller code.

In terms of stack management, traditionally, a stack grows down in memory, with the last "pushed" value at the lowest address. The ARM also supports ascending stacks, where the stack structure grows up through memory. The value of the stack pointer can either: point to the last occupied address (Full stack) and so needs pre-decrementing (ie before the push); or it can point to the next occupied address (Empty stack) and so needs post-decrementing (ie after the push). The stack type to be used is given by the postfix to the instruction: STMFD / LDMFD for Full Descending stack, STMED / LDMED for Empty Descending stack, etc. In our "Hello World" example we used a full descending stack.

A.1.3 Calling external functions

The JLite language memory model depends on dynamic memory as well as static memory. Objects will reside in dynamic memory, or the heap, while other primitive variables and partial computations will reside on the stack. In order to allocate heap space you can make use of the existing function _Znwj. Calling this function is done as follows:

```
1.    mov a1,#4
2.    bl _Znwj(PLT)
3.    mov v5,a1
```

As can be seen the function takes one parameter which is the size in bytes of the memory space to be allocated (Line 1). The return is the address of the memory space allocated, which is returned after the call in argument register a1 (Line 3).

Printing to the standard output can also be done using an external function. This function is printf. As can be seen from the hello world program in order to print a string literal the address of the string is loaded into register a1 (Line 12) and then a branch instruction to the external function is performed (Line 13).

```
12.     ldr a1,=L1
13.     bl printf(PLT)
```

Calling printf to output an integer value requires two parameters. The first parameter specifies the format of the output. For this scenario the format specifies that an integer value is to be output (Lines1-2, Line 2). The second parameter contains the value of the integer to be output which is 4 in this example (Line 4).

```
1. L4:
2.     .asciz "%i"
3.     ldr a1,=L4
4.     mov a2,#4
5.     bl printf(PLT)
```

B Samples of JLite Compilation Results

```
class Main {

Void main(){
Int a;
Int b;
Int i;
Int d;
Int t1;
Int t2;
Compute help;

a = 1;
b = 2;
i = 3;
d = 4;
help = new Compute();
t1 = help.addSquares(a,b) + help.square(i);
t2 = help.square(d); // Should be equal to 16
if(t2>t1){
println("Square of d larger than sum of squares");
// Should be the output
}
else{
println("Square of d smaller than sum of squares");
}
}
}

class Compute {

   Bool computedSquares;
   Int cachedValue;

   Int square(Int a){
     return a*a;
   }
   Int add(Int a, Int b){
    return a+b;
   }
   Int addSquares(Int a, Int b){
    if(computedSquares){
      return cachedValue;
    }
    else{
      computedSquares = true;
      return add(square(a),square(b));
    }
   }
}
}
```

```
.data

L1:
.asciz "Square of d larger than sum of squares"

L2:
.asciz "Square of d smaller than sum of squares"
.text
.global main

Compute_1:
stmfd sp!,{fp,lr,v1,v2,v3,v4,v5}
add fp,sp,#24
sub sp,fp,#28
add v3,a2,a3
mov a1,v3
b .L4exit

.L4exit:
sub sp,fp,#24
ldmfd sp!,{fp,pc,v1,v2,v3,v4,v5}

Compute_2:
stmfd sp!,{fp,lr,v1,v2,v3,v4,v5}
add fp,sp,#24
sub sp,fp,#48
ldr a3,[a1,#4]
cmp a3,#0
beq .3
ldr a1,[a1,#0]
mov a1,a1
b .L3exit
b .4

.3:
mov v5,#1
mov v5,v5
str v5,[a1,#4]
mov a1,a1
mov a2,a2
bl Compute_0(PLT)
mov a3,a1
str a3,[fp,#-44]
mov a1,a1
ldr a2,[fp,#-36]
```

```
bl Compute_0(PLT)
mov v5,a1
mov a1,a1
ldr a2,[fp,#-44]
mov a3,v5
bl Compute_1(PLT)
mov a2,a1
mov a1,a2
b .L3exit

.4:

.L3exit:
sub sp,fp,#24
ldmfd sp!,{fp,pc,v1,v2,v3,v4,v5}

Compute_0:
stmfd sp!,{fp,lr,v1,v2,v3,v4,v5}
add fp,sp,#24
sub sp,fp,#28
mul a4,a2,a2
mov a1,a4
b .L2exit

.L2exit:
sub sp,fp,#24
ldmfd sp!,{fp,pc,v1,v2,v3,v4,v5}

main:
stmfd sp!,{fp,lr,v1,v2,v3,v4,v5}
add fp,sp,#24
sub sp,fp,#68
mov v5,#1
mov a4,v5
mov v5,#2
mov a3,v5
mov v5,#3
mov a2,v5
mov v5,#4
mov v1,v5
str a4,[fp,#-28]
str a3,[fp,#-32]
str a2,[fp,#-36]
mov a1,#8
bl _Znwj(PLT)
mov a4,a1
```

```
str a4,[fp,#-52]
ldr a1,[fp,#-52]
ldr a2,[fp,#-28]
ldr a3,[fp,#-32]
bl Compute_2(PLT)
mov v5,a1
ldr a1,[fp,#-52]
ldr a2,[fp,#-36]
bl Compute_0(PLT)
mov a4,a1
add a1,v5,a4
str a1,[fp,#-44]
ldr a1,[fp,#-52]
mov a2,v1
bl Compute_0(PLT)
mov a3,a1
ldr a4,[fp,#-44]
cmp a3,a4
movgt a1,#1
movle a1,#0
cmp a1,#0
beq .1
ldr a1,=L1
bl printf(PLT)
b .2

.1:
ldr a1,=L2
bl printf(PLT)

.2:

.L1exit:
mov a4,#0
mov a1,a4
sub sp,fp,#24
ldmfd sp!,{fp,pc,v1,v2,v3,v4,v5}
```