National University of Singapore

CS4212 Project Assignment 3

AY 2020/2021 Semester 1

Due Date: 22^{th} November 2020 (Sunday, 23:59 Hrs)

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

1. **You need to produce a JLite compiler.** Your starting point is your submission *Assignment 2*. Specifically, you need to complete the compiler implementation by adding its backend. In particular, you are required to translate *IR3*-programs (corresponding to JLite programs) into ARM assembly codes. *There is no solution for Assignment 2 to start this assignment, you have to use your Assignment 2 codes as the starting point.*
2. **You need to write a report.** This report describes how you design those specific features available in JLite compiler.

Assessment

- **Code**: Your submissions need to pass a benchmark like the previous assignments.
- **Design**: Your report explains the intended design for handling the features of JLite language, be it a subset of JLite or a superset of JLite.
- **Marks**: Your mark is based on your correctness result of the benchmark. Any **optimization** added 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 produced. Please note that the ARM code are meant for your reference. As different teams will develop different optimizations, choose different register allocation strategy, the ARM code produced by different teams will be 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 used for register allocation and assignment. Your grade will depend on the efficiency of register allocation and assignment.
3. An option for users to turn on and turn off a set of optimizations you have implemented. (Note that there should be just one option to turn on/off all optimizations, not individual optimizations.) Your **bonus point** 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 simplications in JLite programs to be compiled (which you might wish to ignore, but without bonus rewarded):

1. As we don't consider ARM instructions that handles floating-point numbers, we do not allow the use of any operations related to floating-point operation. Thus, the accepted JLite programs should not involve any division operation.
2. We skip the complication involved in reading data from console and convert it to the desired type. Therefore, the accepted JLite program will not have `readln` statement. Consequently, your JLite program cannot read any input; the program will assume all data is available for its computation.

2 Delivery

2.1 Consultation

Email me if you have any questions.

2.2 Suggestions

1. Start early. There is quite a bit of work to be done.
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 "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.

2.3 Submission of your product

In addition to testing your program in person, you are also required to:

- Create a good set of sample JLite programs to test your product. Put in a subfolder those sample programs that you have created, and show their corresponding result.
- Create a `README` text file to describe how one can make use of your test suites to validate the correctness of your programs.

Please submit a zipped item, named after you, containing the following documents to `IVLE CS4212` website under the "`Project Assignment 3 Submission`" folder.

1. Your product.
2. **Five** sample programs that you have tried on your product
3. A document, named **CS4212 2019 Project Assignment 3 Details**, describing your solution, the content of your submission, and any important information which you would like to share with us.

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.

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.

```
 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

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

A.1.2 Selected ARM Instruction Set

The set of instructions to be used in your assembly code is defined in the Ocaml file arm_structs.ml in the workbin. This definition file is optional for your assignment. You can choose you define your own datatypes for ARM instructions.

The selected subset of arm instruction includes memory access instructions such LDR (Load) and STR (Store), general data processing instructions such as ADD, AND, CMP(compare), and control transfer or (branch instructions). The details of all these instructions can be found in section 4 in the
Arm_Assembler_Guide.pdf in the workbin. In order to understand how the datatypes defined in arm_structs.ml map to the instruction formats in Arm_Assembler_Guide.pdf let's look at the definitions.

1. General data processing instructions for performing arithmetic or boolean operations have the following format:

```
op{cond}{S} Rd, Rn, Operand2

where:
 op is one of ADD, SUB, RSB, ADC, SBC, or RSC.
 cond is an optional condition code.
 S is an optional suffix. If S is specified, the condition
   code flags are updated on the result of the operation.
 Rd is the ARM register for the result.
 Rn is the ARM register holding the first operand.
 Operand2 is a flexible second operand
```

This ARM format corresponds to the OCaml defined type:

```
data_instr_type =
 cond * bool * reg * reg * operand2_type
```

The Ocaml type definition is easy to read. For instance, the above type means that it consists of 5 components, **cond**, **bool**, **reg**, **reg**, **operand2_type**. If you have any problem reading Ocaml codes, please contact the TA.

2. Memory access instructions LDR and STR have the following format:

```
op{cond}{B}{T} Rd, [Rn]
op{cond}{B} Rd, [Rn, FlexOffset]{!}
op{cond}{B} Rd, label
op{cond}{B}{T} Rd, [Rn], FlexOffset
where:
 op is either LDR (Load Register) or STR (Store Register)
```

This ARM format corresponds to the OCaml defined types:

```
type mem_instr_type =
 cond * word_type * reg * address_type

type address_type =
| LabelAddr of string
| Reg of reg
| RegPreIndexed of reg * int * bool
| RegPostIndexed of reg * int

type word_type = string
```

3. Instructions to compare values have the following format:

```
CMP{cond} Rn, Operand2
CMN{cond} Rn, Operand2
TST{cond} Rn, Operand2
TEQ{cond} Rn, Operand2
```

This ARM format corresponds to the OCaml defined type:

```
type cmp_instr_type = cond * reg * operand2_type
```

4. Instructions to move a value from one register to another have with/without negating it have the following format:

```
MOV{cond}{S} Rd, Operand2
MVN{cond}{S} Rd, Operand2
```

This ARM format corresponds to the OCaml defined type:

```
type mov_instr_type = cond * bool * reg * operand2_type
```

5. Instructions to perform a jump with/without saving the return address in a link register have the following format:

```
B{cond} label
BL{cond} label
```

This ARM format corresponds to the OCaml defined constructors:

```
 type arm_instr =
  ...
  | B of cond * label
  | BL of cond * label
...
```

There are a few more instructions not presented here such as instructions to load and store a list of registers (stmfd, ldmfd), an instruction for multiplication. These are presented at large in the assembly guide and defined in arm_structs.ml in the workbin.

A.1.3 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.1.4 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}
```