CS5218: Assignment 1 – Initialized Variables

1 Introduction

There are several goals for this assignment:

- Designing a sample analysis using the principles of Data Flow Analysis.
- Gaining exposure to LLVM in general and the LLVM IR which is the intermediate representation used by LLVM.
- Using LLVM to perform a simple analysis.

This assignment is not a group assignment.

1.1 Initialized Variables Analysis

We define a simple analysis, initialized variables analysis, which returns the set of all the variables that are initialized at any program point. The initialized variables analysis returns the list of all variables which are initialized in each basic block. We consider a variable to be initialized in a basic block if it has been initialized in at least one path from the beginning of the program.

Example 1: For example, consider the c program below:

```
int main() {
    int a,b,c,d,e,f;
    b = 5;
    if (a > 0)
        f = -1;
    else
        c = b;
    e = d;
}
```

The generated LLVM IR will have four basic blocks with labels: entry, if.then, if.else and if.end. In the end of each of the basic blocks the list of initialized variables would be as follows (registers are skipped for brevity):

- entry: {b}
- if.then: {b, f}
- if.else: {b, c}
- if.end: {b, c, e, f}

Note 1: Although the value of d is unknown, still e is considered as an initialized variable.

Note 2: Note that the list of initialized variables for if.end contains both f and c which were initialized in its predecessor basic blocks it.then and if.end.

2 Task 1: Designing Initialized Variables Analysis

In this task, we first will design an initialized variables analysis based on principles of Data Flow Analysis. In order to do that the lattice, transfer functions and the monotone framework over a simple while language (slide 2 of CS5218-02-DFA.pdf) should be defined for the analysis.

3 Task 2: Implementing the Initialized Variables Analysis in LLVM

LLVM is a set of compiler infrastructure tools. You have seen `clang` and `clang++`, the LLVM C and C++ compilers in the demo. In this task, you will write an LLVM pass (similar to the LLVM pass in StackSet.cpp which was explained in the demo session) to perform the analysis on loop-free programs.

Note 1: You would need a map which contains the results of the analysis map for each program point. You can use a C++ map which maps the label name of each Basic Block to a set of initialized variables. Note that variables in LLVM are represented with Alloca instructions:

```
std::map<std::string,std::set<Instruction*>> analysisMap;
for (auto &BB: *F){
    std::set<Instruction*> emptySet;
    analysisMap[getSimpleNodeLabel(&BB)] = emptySet;
}
```

You can also print the map using the following function:

```
void printAnalysisMap(std::map<std::string,std::set<Instruction*>> analysisMap) {
    for (auto& row : analysisMap){
        std::set<Instruction*> initializedVars = row.second;
        std::string BBLabel = row.first;
        errs() << BBLabel << ":\n";
        for (Instruction* var : initializedVars){
            errs() << "\t";
            var->dump();
        }
        errs() << "\n";
    }
}
```

The function *dump* is a very useful function which can print LLVM basic blocks, instructions and values. For example, $var->dump()$ in the code above will print an initialized var which would be like the following:

```
%b = alloca i32, align 4
```

Note 2: You can loop through instructions in a basic block using a simple for loop:

```
 for (auto &I: *BB){ ... }
```

Note 3: The opcode of an instruction can be checked by the *isa* casting instruction which returns either false or true. For example, the following instruction will return true on all Load instructions and false on all other instructions:

```
if(isa<LoadInst>(I)) { ... }
```

Note 4: Each LLVM instruction may have some arguments. The arguments are of type LLVM Value which is a superset of all instructions, constant values and global variables in LLVM. You can load an instruction's argument using the *getOperand* function and then cast it to an LLVM instruction using *dyn_cast*:

```
Value* v = I.getOperand(1);  // reterieving second argument
Instruction* var = dyn_cast<Instruction>(v);
```

Note, in case an LLVM value is not an LLVM instruction, the casting will return NULL.

4 BONUS TASK - Task 3: Adding Support for Loops to the Analysis

In order to handle loops, the designed analysis should be continued until a fixpoint is reached. Change your analysis from task 2 to support loops.

Example 3: In this example, consider the c program below and its respective CFG in Figure 2:

```
int main() {
    int a,b,c,d,e,f;
    b = 5;
    int i = 0;
    while (i < 10) {
        if (i % 2 == 0)
            f = -1;
        else
            c = b;
        i++;
    }
    e = d;
}
```

Applying the initialized variables analysis will generate the following list of initialized variables in the end of each of the basic blocks:

- entry: {b, i}
- while.cond: {b, c, f, i}
- while.body: {b, c, f, i}
- if.then: {b, c, f, i}
- if.else: {b, c, f, i}
- if.end: {b, c, f, i}
- while.end: {b, c, e, f, i}

Note the difference in the list of initialized variables in if.then and if.end compared to the example in task 2. The difference is that the list of initialized variables from each basic block can reach the other basic block in the next iteration. As a result, both c and f are initialized in both basic blocks.

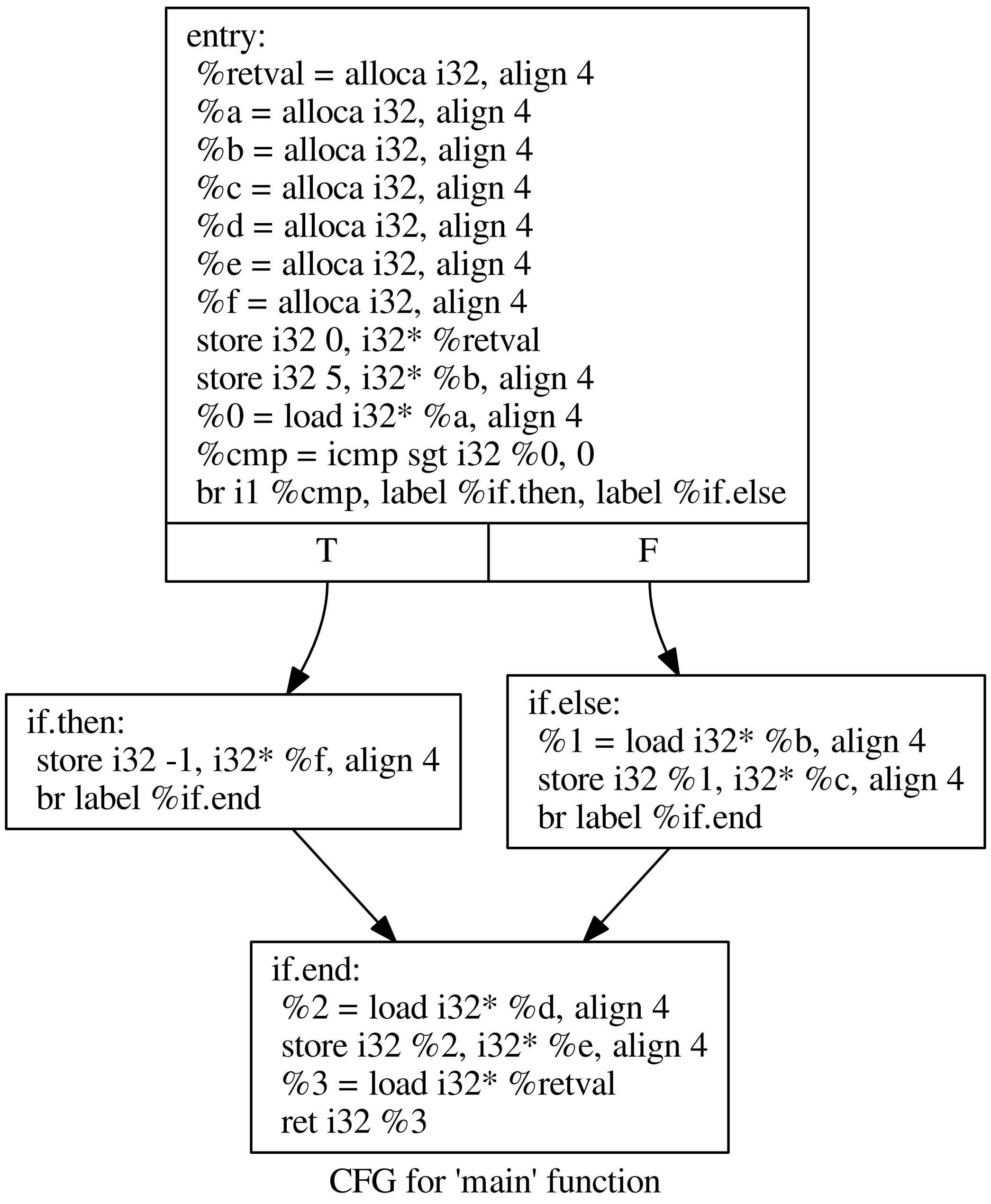
5 Submission - Due Date: Sun 24th Feb., 11.59pm

Please submit the following in a single archive file (`zip` or `tgz`):

1. A report in PDF (`asg1.pdf`). It should describe your design for task 1, LLVM version used, the implementation of tasks, the algorithm used. It should also contain the steps on how to build and run the pass and the output of examples programs tested.
2. Your source code.
3. Your test C files with corresponding `ll` files.

For technical support on LLVM, you can contact Rasool or Sanghu (`{rasool,sanghara}@comp.nus.edu.sg`). Make sure your subject line begins with “CS5218”.

Make sure your reports and source files contain information about your name, matric number and email. Your zip files should have the format *Surname-Matric*-asg1.zip (or `tgz`). Submit all files above to the appropriate IVLE workbin.



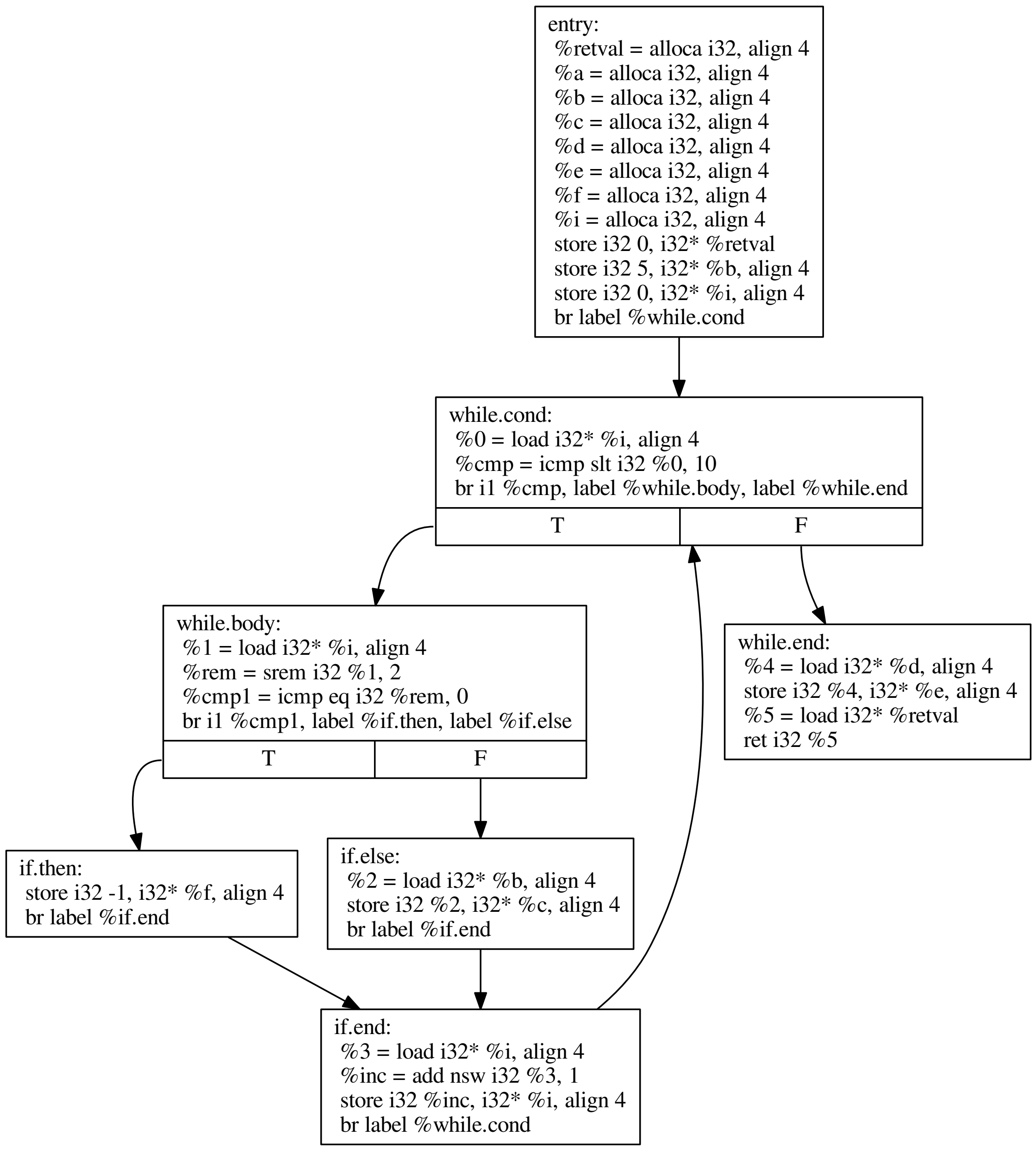
```
entry:
 %retval = alloca i32, align 4
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %d = alloca i32, align 4
 %e = alloca i32, align 4
 %f = alloca i32, align 4
 store i32 0, i32* %retval
 store i32 5, i32* %b, align 4
 %0 = load i32* %a, align 4
 %cmp = icmp sgt i32 %0, 0
 br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
 store i32 -1, i32* %f, align 4
 br label %if.end
```

```
if.else:
 %1 = load i32* %b, align 4
 store i32 %1, i32* %c, align 4
 br label %if.end
```

```
if.end:
 %2 = load i32* %d, align 4
 store i32 %2, i32* %e, align 4
 %3 = load i32* %retval
 ret i32 %3
```

Figure 1: CFG of C Program without Loop



```
entry:
 %retval = alloca i32, align 4
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %d = alloca i32, align 4
 %e = alloca i32, align 4
 %f = alloca i32, align 4
 %i = alloca i32, align 4
 store i32 0, i32* %retval
 store i32 5, i32* %b, align 4
 store i32 0, i32* %i, align 4
 br label %while.cond
```

```
while.cond:
 %0 = load i32* %i, align 4
 %cmp = icmp slt i32 %0, 10
 br i1 %cmp, label %while.body, label %while.end
```

```
while.body:
 %1 = load i32* %i, align 4
 %rem = srem i32 %1, 2
 %cmp1 = icmp eq i32 %rem, 0
 br i1 %cmp1, label %if.then, label %if.else
```

```
while.end:
 %4 = load i32* %d, align 4
 store i32 %4, i32* %e, align 4
 %5 = load i32* %retval
 ret i32 %5
```

```
if.then:
 store i32 -1, i32* %f, align 4
 br label %if.end
```

```
if.else:
 %2 = load i32* %b, align 4
 store i32 %2, i32* %c, align 4
 br label %if.end
```

```
if.end:
 %3 = load i32* %i, align 4
 %inc = add nsw i32 %3, 1
 store i32 %inc, i32* %i, align 4
 br label %while.cond
```

Figure 2: CFG of C Program with Loop