Assignment 4: Code Optimization & Target Code Generation

CS 464/5607: Compiler Design

Spring 2026

1 Introduction

In this final assignment, you will complete the compiler pipeline by implementing two critical phases:

1. **Phase 4b: Code Optimization**: Transforming the Intermediate Representation (IR) to improve efficiency.
2. **Phase 5: Target Code Generation**: Translating the optimized IR into MIPS assembly language executable on the SPIM simulator.

Provided IR Implementation

To assist you, a complete reference implementation of the IR generation phase is provided in the `lib/` directory (`ir_generator.cpp` and `ir.cpp`). Since both optimization and code generation require you to traverse and manipulate the IR linked list extensively, you must familiarize yourself with the `IRInst` structure defined in `ir.h`.

Note: You are free to replace these files with your own implementation from the previous assignment if you prefer.

2 Part 1: Code Optimization

2.1 Why Optimize?

Optimization is the process of modifying code to consume fewer resources (execution time, memory, or power) while preserving the original program's semantics. A naive compiler might generate redundant instructions; the optimizer's job is to clean this up.

Common optimization techniques include Loop Unrolling, Function Inlining, and Common Subexpression Elimination. In this assignment, for simplicity, we will focus on two foundational techniques: **Constant Folding** and **Dead Code Elimination**.

2.2 Implemented Optimizations

2.2.1 1. Constant Folding

Constant folding involves evaluating expressions with constant operands at compile-time rather than runtime. This reduces the number of instructions executed by the CPU.

Example: Consider the source code `x = 3 + 4;`.

- **Naive IR:**

```
ADD 3 4 t0     // t0 = 3 + 4
ASSIGN t0 x    // x = t0
```

- **Optimized IR:** The optimizer detects that both operands (3 and 4) are literals. It calculates the result (7) and replaces the instruction.

```
ASSIGN 7 t0    // Constant folded result
ASSIGN t0 x
```

Important: You must maintain the Three-Address Code structure. Do not assign directly to `x` in one step if the original IR used a temporary. Fold the calculation into an `ASSIGN` to the temporary variable first.

2.2.2 2. Dead Code Elimination (DCE)

DCE removes instructions that have no effect on the program output. You will implement two types:

- **Unreachable Code**: Instructions immediately following a `RETURN` or unconditional `GOTO` that are not targets of a label.
- **Unused Assignments**: Assignments to variables (temps or locals) that are never subsequently read (used as an argument or operand) before being redefined or the scope ending.

2.3 Optimizer API (`optimizer.h`)

The optimization logic is controlled by a driver function that repeatedly applies passes until the IR reaches a "fixed point" (no further changes occur). You will implement the logic in `optimizer.cpp`.

- `void optimize_ir(IRInst *head)`: The entry point. It calls the individual passes in a loop.
- `int constant_folding(IRInst *head)`: Scans the IR for arithmetic operations (`ADD`, `SUB`, `MUL`, `DIV`) on literals. Returns the number of folds performed.
- `int dead_code_elimination(IRInst *head)`: Identifies and removes unreachable or unused instructions. Returns the number of instructions deleted.

Debugging Tip: Use print statements inside your passes to track how many folds and eliminations occur in each pass. This will help you verify that your fixed-point loop is terminating correctly.

3 Part 2: Target Code Generation (MIPS)

3.1 Overview of MIPS

MIPS (Microprocessor without Interlocked Pipelined Stages) is a RISC architecture that uses a Load/Store model. This means operands must be loaded into registers before operations can be performed, and results must be stored back to memory.

You will implement the generator in `codegen.cpp`. The goal is to traverse the IR list and emit valid MIPS assembly to an output file.

3.2 Scope of Generation

Your generator must handle:

- **Control Flow**: `if-else` statements (branching) and `while` loops.
- **Functions**: Function headers, calls (`jal`), returns (`jr $ra`), and stack management.
- **Arithmetic**: `add`, `sub`, `mul`, `div`, `mod`.
- **Comparisons**: `eq`, `neq`, `lt`, `gt`, `le`, `ge`.
- **I/O**: `cout` (via syscalls). *Note: You may ignore* ***`cin`*** *handling for the automated testing purposes.*

3.3 Implementation Hints

You must implement the function:

```
1 void generate_mips(IRInst *ir_head, char *output_filename);
```

A standard approach involves two passes over the IR:

1. **Pass 1 (.data generation)**: Traverse the IR to identify all variables (user variables and compiler temporaries like `t0`). Emit them in the `.data` section initialized to 0.
2. **Pass 2 (.text generation)**: Traverse the IR again to emit instructions. The specific logic varies based on the instruction type, for example for arithmetic and logical operations:
    - Load operand `a` into a register (e.g., `$t0`).
    - Load operand `b` into a register (e.g., `$t1`).
    - Perform the MIPS `add` instruction.
    - Store the result register back to memory variable `t0`.

4 MIPS Syntax and Instructions

MIPS assembly files consist of a `.data` section for declaring variables and a `.text` section for code logic.

4.1 System Calls (Syscalls)

MIPS uses the `syscall` instruction to interact with the OS. The operation is determined by the value in register `$v0`.

Service	Code ($v0)	Arguments
print_int	1	$a0 = integer to print
print_float	2	$f12 = float to print
print_char	11	$a0 = character to print
exit	10	(none)

Implementation Note: To implement `cout << x`, load 1 into `$v0`, load `x` into `$a0`, and execute `syscall`.

4.2 Relevant Instruction Set

The following table summarizes the instructions you will need.

IR Op	MIPS Instruction	Example	Description
ASSIGN	`li` / `lw` + `sw`	`li $t0, 5`	Load immediate or word
ADD	`add`	`add $t2, $t0, $t1`	$t2 = $t0 + $t1
SUB	`sub`	`sub $t2, $t0, $t1`	$t2 = $t0 - $t1
MUL	`mul`	`mul $t2, $t0, $t1`	$t2 = $t0 * $t1
DIV	`div` + `mflo`	`div $t0, $t1`	Lo = $t0 / $t1 (Quotient)
MOD	`div` + `mfhi`	`div $t0, $t1`	Hi = $t0 % $t1 (Remainder)
IFZ	`beqz`	`beqz $t0, label`	Branch if $t0 == 0
GOTO	`j`	`j label`	Unconditional jump
LABEL	(none)	`label:`	Define target
FUNC	(none)	`main:`	Define function entry
CALL	`jal`	`jal func`	Jump and link (save PC to $ra)
RET	`jr`	`jr $ra`	Jump register (return)
PARAM	`sw` (stack)	`sw $t0, ($sp)`	Push argument to stack
POP	`lw` (stack)	`lw $t0, ($sp)`	Pop argument from stack
EQ	`seq`	`seq $t2, $t0, $t1`	Set $t2=1 if $t0 == $t1
NEQ	`sne`	`sne $t2, $t0, $t1`	Set $t2=1 if $t0 != $t1
LT	`slt`	`slt $t2, $t0, $t1`	Set $t2=1 if $t0 ¡ $t1
GT	`sgt`	`sgt $t2, $t0, $t1`	Set $t2=1 if $t0 ¿ $t1
LTE	`sle`	`sle $t2, $t0, $t1`	Set $t2=1 if $t0 ¡= $t1
GTE	`sge`	`sge $t2, $t0, $t1`	Set $t2=1 if $t0 ¿= $t1
PRINT	`syscall`	`li $v0, 1`	Refer to section 4.1

4.3 Reserved Keywords

You strictly cannot use MIPS reserved keywords as variable names in your `.data` section. You may implement name mangling (e.g., prefixing variables) to avoid conflicts. **Forbidden names include**:

- Register names: `zero, v0, a0, t0-t9, s0-s7, sp, fp, ra, gp, at, k0, k1`
- Opcodes: `add, sub, div, mul, lw, sw, beq, bne, b, j, jal, syscall`

5 Simulating with SPIM

SPIM is a self-contained simulator that runs MIPS32 assembly language programs. You must ensure SPIM is installed on your development environment (e.g., via `sudo apt install spim` on Linux/WSL) before attempting to run your generated code.

5.1 Running the Simulator

To simulate a generated MIPS assembly file, use the `-file` flag followed by the filename:

```
spim -file output.s
```

5.2 Understanding SPIM Output

When you run SPIM, it prints several lines of startup information (version, copyright, and memory loading status) before executing your program's instructions.

- **Startup Header**: The first 5 lines of output typically contain SPIM's initialization messages (e.g., "SPIM Version...", "Loaded: ...").
- **Program Output**: Your program's actual output (e.g., from `cout` syscalls) appears immediately after these header lines.

Note: The provided testing script (`run_tests.py`) automatically strips these first 5 lines to compare only your program's actual output against the expected results.

5.3 Driver Output Behavior

The driver (`driver.cpp`) determines the name of the output MIPS file based on the command-line arguments provided.

- **Automatic Naming (1 argument)**: If you run the compiler with only the input file path, the driver replaces the input extension with `.s`.

```
./build/compiler input.txt
# Result: Generates 'input.s'

```

- **Manual Naming (2 arguments)**: If you provide a second argument, the driver uses it strictly as the output filename.

```
./build/compiler input.txt my_custom_output.s
# Result: Generates 'my_custom_output.s'

```

6 Building and Testing

6.1 Running Tests

We have provided an automated Python script to facilitate testing. This script:

1. Runs `make` to build the compiler.
2. Compiles inputs from `tests/inputs/`.
3. Compares generated IR against expected output.
4. Simulates the generated MIPS code using `spim`.
5. Compares the runtime output against `tests/expected_output/`.

To run the full suite:

```
python run_tests.py
```

6.2 Testing with Custom Inputs

You are encouraged to create your own test cases to debug specific issues or test edge cases.

1. Create a text file (e.g., `my_test.txt`) with your source code.
2. Open your terminal and run `make` to ensure your compiler is built.
3. Run the compiler manually by passing the file path as an argument:

   ```
   ./build/compiler my_test.txt
   ```

   This will generate an assembly file (e.g., `my_test.s`).
4. Simulate the generated code using SPIM (refer to Section 5):

   ```
   spim -file my_test.s
   ```

6.3 Platform Specifics (Windows)

If you are developing natively on Windows (using MinGW/Git Bash) rather than WSL or Linux, you must make two small modifications to handle file extensions correctly.

1. Modify Makefile

Windows executables require the `.exe` extension. Update the `TARGET` variable:

```
# Change:
TARGET = $(BUILD_DIR)/compiler

# To:
TARGET = $(BUILD_DIR)/compiler.exe
```

2. Modify run_tests.py

The Python automation script needs to know the exact name of the executable to launch it. Update the `COMPILER_BIN` variable:

```
# Change:
COMPILER_BIN = "./build/compiler"

# To:
COMPILER_BIN = "./build/compiler.exe"
```

6.4 Grading Distribution

- **60% Visible Test Cases**: The visible tests provided in the `tests/` folder.
- **40% Hidden Test Cases**.

7 Submission Instructions