Project 2: Terminators

Youssef Amin, Zhuo Li, Yue Hu

November 2025

1 Task 1

For Task 1, we reviewed all components of our Project 1 implementation, and confirmed that they are working correctly and required no major changes.

2 Task 2

First, we recursively traverse the Abstract Syntax Tree (AST) and assigns a unique integer label to every executable node. The label numbers start from **1** and increase sequentially.

Labels are assigned to the following statement nodes:

- **Assign**
- **Skip**
- **If**
- **While**

For **If** and **While** nodes, the label represents their conditional test point. The label information is stored in each node's `label` field for later use in pretty-printing and visualization.

Then the Pretty_Printer module generates human-readable source code from the AST structure. For each node that has a label, the pretty printer outputs the label using the following format:

```
/*l = k*/ <statement>
```

It also supports indentation and keyword-based formatting, such as:

- `if ... then ... else ... fi`
- `while ... do ... od`

By combining the output from the Labeler, the Pretty Printer produces a WHILE program in which every program point has a unique label (Fig.1).

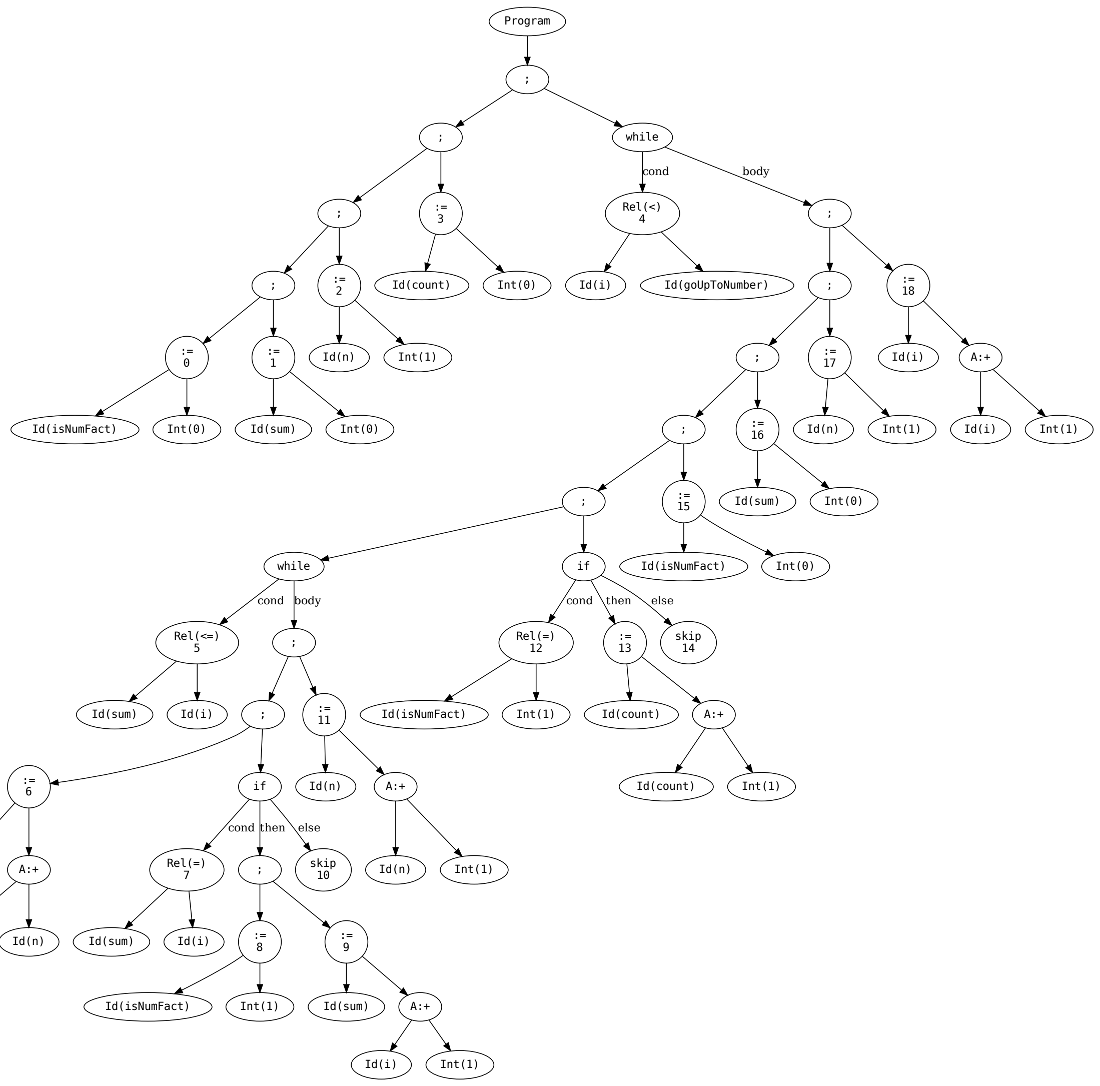

Figure 1: Labeled AST generated from the same WHILE program in Task 2, Project 1

3 Task 3

For this task we needed a concrete control-flow graph (CFG) representation and a conversion from the labeled AST into that CFG.

CFG data structure. We represent a CFG as a mapping from labels to nodes. Each node corresponds to one labeled basic command in the WHILE program:

- a unique integer label ℓ,
- a pointer to the AST node for the command at ℓ (assignment, `skip`, `if`, or `while`),
- and a list of successor pointers for outgoing control-flow edges.

In C++ terms, the CFG stores `CFGNode` objects in a dictionary keyed by ℓ, and each `CFGNode` owns its AST pointer and a small vector of successor pointers. This makes it easy to iterate over nodes by label, and to follow successors when constructing analyses such as liveness.

Building the CFG from the labeled AST. Starting from the Task 2 labeled AST, we recursively traverse the syntax tree and create CFG nodes as follows:

- For an annotated assignment $[x := a]_\ell$ or $[skip]_\ell$, we create a single node labeled ℓ.
- For a sequence $c_1; c_2$, we recursively build CFG fragments for c_1 and c_2 and connect every exit of the first fragment to the entry of the second.
- For an `if` $[b]_\ell$ `then` c_1 `else` c_2 `fi`, we create a header node at ℓ whose successors are the entries of the then- and else-branches. The exits of both branches are then connected to a common successor (the command that follows the `if`).
- For a `while` $[b]_\ell$ `do` c `od`, we create a header node at ℓ with an edge to the loop body and an edge to the command that follows the loop. The exits of the body point back to the header, forming the back edge of the loop.

The result is a graph in which each node represents a single labeled command and edges represent all possible next commands during execution.

Displaying the CFG. To display the CFG we emit a DOT file that lists each node label and its successors. Using `dot` we can then generate a PDF or PNG diagram. This visualization was especially helpful when debugging later analyses (such as liveness and dead-code elimination) because we could directly see the control-flow structure for each test program.

Basic WHILE

```
x := 4;
if [x <= 3] then
  y := 10
else
  y := 20
fi;
z := y + 1
```

Labeled WHILE

```
/*l=0*/ x := 4;
/*l=1*/ if ((x <= 3)) then
  /*l=2*/ y := 10;
else
  /*l=3*/ y := 20;
fi
/*l=4*/ z := (y + 1);
```

Generated CFG

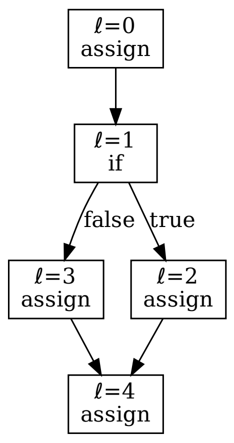

Figure 2: Basic Program, Labeled Program, and Generated Control-Flow Graph.

4 Task 4

In Task 4 we generate virtual machine code from the CFG. The code is written in a RISC-V–like assembly language, but at this stage we still assume an unbounded number of `s`-registers: each source-level variable gets its own dedicated register. Later tasks (Tasks 9–10) refine this into a real register allocation with spills.

Variable-to-register mapping. We first build a symbol table that assigns each program variable a unique index. In the Task 4 version of the compiler, variable number i is mapped to register `s(i+1)`. This gives a simple one-to-one mapping from variables to registers and keeps expression code generation straightforward. The mapping is also embedded into comments in the assembly code, which latter helps debugging subsequent tasks (liveness analysis and register allocation)

Prologue and epilogue. The generated code follows the calling convention used in Project 1: the compiled program receives a pointer in `a0` to an array holding the values of all WHILE variables. The function prologue:

- saves any `s`-registers we plan to use on the stack, and
- loads initial variable values from the array into their corresponding `s`-registers.

The epilogue performs the inverse steps: it writes the final values of the `s`-registers back into the variable array, restores the saved `s`-registers from the stack, and then returns.

Translating commands and expressions. Each CFG node is translated into a small sequence of assembly instructions:

- Arithmetic expressions are evaluated into `t0`, using instructions such as `add`, `sub`, and `mul`. Variable reads move from `s`-registers into `t0`, and integer literals become `li` instructions.
- Assignments $[x := a]_\ell$ evaluate the right-hand side into `t0` and then move the result into the register assigned to `x`.
- Boolean expressions and comparisons generate code that normalizes results to 0/1 in `t0` and then use `beqz` branches.
- For `if` and `while` commands, the CFG edges directly determine which labels we branch to: the header node computes the boolean condition and emits a conditional branch to one successor and an unconditional jump to the other.

We also emit comments in the assembly that record the source label and the kind of command (assign, `if`, `while`, or `skip`), so that the generated code can be traced back to the CFG and the original WHILE program. This virtual RISC-V code serves as the baseline back end that we later optimize in Tasks 9 and 10.

Prologue (Loading Variables)

```
# s1 <- input (x)
ld s1, 0(t2)
addi t2, t2, 8
# s2 <- input (y)
ld s2, 0(t2)
addi t2, t2, 8
# s3 <- input (z)
ld s3, 0(t2)
addi t2, t2, 8
j L0
```

Epilogue (Writing Back Registers)

```
# output (x) <- s1
sd s1, 0(t2)
addi t2, t2, 8
# output (y) <- s2
sd s2, 0(t2)
addi t2, t2, 8
# output (z) <- s3
sd s3, 0(t2)
addi t2, t2, 8
```

CFG Node Emission (Label $\ell = 1$: if-statement)

```
L1:
# l=1 if
mv t0, s1
mv t1, t0
li t0, 3
slt t0, t0, t1
xori t0, t0, 1
beqz t0, L3 # false -> else
j L2 # true -> then
```

Figure 3: Prologue, Epilogue, and CFG-Based Code Emission.

5 Task 5

Our team had already implemented a comprehensive testing framework during `Project 1`, and we reused and extended this framework for Project 2. This framework performs the following:

- Automatically discovers all `WHILE` programs from `WhileFiles/`
- Compiles each file using our compiler `./whilec` and checks that compilation completes without crashes.
- Verifies that the compiler generated all output files are produced (e.g., `out_program.s`).
- Validates the structure of the pseudo RISC-V assembly, which ensures:
  - Correct prologue: `mv t2, a0`
  - Load/store for variable initialization: `ld` or `sd`
  - Control-flow instructions when the source contains `if`/`while`

6 Task 6

This task compares the performance of our Project 1 compiler (which uses memory slots for all variables) with our Project 2 compiler (which stores program variables directly in RISC-V registers). The main goal is to measure the performance impact of reducing the number of memory loads and stores. Since memory access is slower than register access, we expect that reducing the number of `ld`/`sd` instructions will improve performance, especially for workloads that involve many arithmetic operations or loops.

We evaluated a set of stress tests on `risc-machine-2.cs.unm.edu`. These tests include factorial by repeated addition, Euclidean GCD, prime counting, and other loop-intensive workloads. Table 1 summarizes the times for both compilers:

Workload	**Input**	**Time Perform**	**Project 1**	**Project 2 (Release2)**
Factorial (Addition)	12	real	2.796s	1.399s
		user	2.794s	1.394s
		sys	0.001s	0.005s
Factorial (Addition)	13	real	36.282s	18.148s
		user	36.270s	18.144s
		sys	0.005s	0.009s
Square	10^5	real	1m13.995s	40.665s
		user	1m13.974s	40.649s
		sys	0.009s	0.008s
GCD(Sub)	$(1, 10^9)$	real	10.744s	5.733s
		user	10.742s	5.726s
		sys	0.000s	0.000s
GCD(Mod)	(102334155, 102334156)	real	0.672s	0.370s
		user	0.671s	0.365s
		sys	0.001s	0.005s
Prime Count	10^4	real	3.218s	1.509s
		user	3.216s	1.499s
		sys	0.001s	0.009s

Table 1: Execution Time Comparison for Workloads between Project 1 and Project 2.

Across all workloads, Project 2 consistently runs faster than Project 1. Table 1 shows that the speedup is especially large for programs with heavy arithmetic workload, such as factorial (addition based), and prime counting. These programs perform many repeated operations, and replacing repeated memory accesses with register-to-register instructions makes a clear difference in execution time.

To furtherly illustrate why our compiler in release 2 substantially faster than Project 1, we compared the generated assembly code for the inner loop of the Listing 6 benchmark. Although both compilers implement the same `WHILE` program, the structure of the generated code differs a lot.

In Figure 4, we could see that, for memory based code generator, every iteration of the inner loop reloads k from memory, performs an update, and write it back. Even a simple command `k:=k-1` requires recomputing the address of k and loading it from memory on each iteration. In contrast, for a register based code generator, k is mapped into a saved register (`s5`), the loop body would therefore perform the update using only register-to-register instructions, no address computation or memory access occurs inside the loop body,

Square

```
i := input;
j := input;
sum := 0;
while i > 0 do
  k := j;
  while k > 0 do
    sum := sum + 1;
    k := k - 1
  od;
  i := i - 1
od;
output := sum
```

Memory-Based Codegen

```
# k := k - 1
li t2, 4
slli t2, t2, 3
add t2, a0, t2
ld t0, 0(t2)

mv t1, t0
li t0, 1
sub t0, t1, t0

sd t0, 0(t2)
```

Register-Based Codegen

```
# k := k - 1
mv t0, s5
mv t1, t0
li t0, 1
sub t0, t1, t0
mv s5, t0
```

Figure 4: Comparison of the Inner Loop: Source-Level WHILE code, Project 1 Memory-Based Assembly, and Project 2 Register-Based Assembly.

Task 7: Definition of `gen`/`kill` Sets and Liveness Equations

Liveness analysis is a backward dataflow analysis. A variable is live at a program point if its value may be used along some path before being overwritten.

We assume that each WHILE statement corresponds to a CFG node. For an expression e, let $FV(e)$ denote the set of variables occurring freely in e.

Formal Definition of `gen` and `kill` For each CFG node n:

$$\begin{aligned}
\text{Assignment } (x := a) : &\quad \text{gen}[n] = FV(a), \qquad \text{kill}[n] = \{x\},\\
\texttt{skip} : &\quad \text{gen}[n] = \varnothing, \qquad \text{kill}[n] = \varnothing,\\
\text{Condition } (b) : &\quad \text{gen}[n] = FV(b), \qquad \text{kill}[n] = \varnothing,\\
\text{Loop guard } (b) : &\quad \text{gen}[n] = FV(b), \qquad \text{kill}[n] = \varnothing.
\end{aligned}$$

All statements inside conditional or loop bodies use the appropriate rule for their basic form above.

Liveness Dataflow Equations For each CFG node n, we compute:

$$\text{in}[n] \quad \text{(variables live before node } n), \textit{and} \qquad \text{out}[n] \quad \text{(variables live after node } n).$$

The liveness equations are:

$$\boxed{\text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])}$$

$$\boxed{\text{out}[n] = \bigcup_{s \in \text{succ}(n)} \text{in}[s]}$$

The list of variables assumed live at program exit is used to initialize the out set of the exit node. All other sets begin empty.

Data Structure For each CFG label ℓ, we store:

$$\text{gen}[\ell],\ \text{kill}[\ell],\ \text{succ}[\ell],\ \text{in}[\ell],\ \text{out}[\ell].$$

These collectively form the full system of liveness equations, which are printed as required by Task 7.

To illustrate how our implemented liveness analysis works, we present a complete example in this section.

First, the WHILE code:

```
n := in;

if n > 100 then
    y := 5
else
    if not [n <= 20] then
        if n < 41 then
            y := 1
        else
            if [n > 80 and n <= 100] then
                y := 4
            else
                if [n < 51 or n <= 60] then
                    y := 2
                else
                    y := 3
                fi
            fi
        fi
    else
        y := 0
    fi
fi;

out := y
```

then, the CFG:

```
digraph CFG {
  node [shape=box];
  12 [label="l=12\nout := y"];
  11 [label="l=11\ny := 0"];
  10 [label="l=10\ny := 3"];
  9 [label="l=9\ny := 2"];
  8 [label="l=8\nif (((n < 51) or (n <= 60))) then ..."];
  7 [label="l=7\ny := 4"];
  6 [label="l=6\nif (((n > 80) and (n <= 100))) then ..."];
  5 [label="l=5\ny := 1"];
  4 [label="l=4\nif ((n < 41)) then ..."];
  3 [label="l=3\nif ((not (n <= 20))) then ..."];
  2 [label="l=2\ny := 5"];
  1 [label="l=1\nif ((n > 100)) then ..."];
  0 [label="l=0\nn := in"];

  11 -> 12;
  10 -> 12;
  9 -> 12;
  8 -> 9 [label="true"];
  8 -> 10 [label="false"];
  7 -> 12;
  6 -> 7 [label="true"];
  6 -> 8 [label="false"];
  5 -> 12;
  4 -> 5 [label="true"];
  4 -> 6 [label="false"];
  3 -> 4 [label="true"];
  3 -> 11 [label="false"];
  2 -> 12;
  1 -> 2 [label="true"];
  1 -> 3 [label="false"];
  0 -> 1;
}
```

We use code to generate the following sets for each node in the CFG: (1) gen set; (2) kill set; (3)Equations for IN[n] and OUT[n] (not yet solved)

```
l=0: n := in
  gen = {in}
  kill = {n}
  OUT[0] = IN[1]
  IN[0] = gen  (OUT[0] \ kill)

l=1: if ((n > 100)) then ...
  gen = {n}
  kill = {}
  OUT[1] = IN[2]  IN[3]
  IN[1] = gen  (OUT[1] \ kill)

l=2: y := 5
  gen = {}
  kill = {y}
  OUT[2] = IN[12]
  IN[2] = gen  (OUT[2] \ kill)

l=3: if ((not (n <= 20))) then ...
  gen = {not, n}
  kill = {}
  OUT[3] = IN[4]  IN[11]
  IN[3] = gen  (OUT[3] \ kill)

l=4: if ((n < 41)) then ...
  gen = {n}
  kill = {}
  OUT[4] = IN[5]  IN[6]
  IN[4] = gen  (OUT[4] \ kill)

l=5: y := 1
  gen = {}
  kill = {y}
  OUT[5] = IN[12]
  IN[5] = gen  (OUT[5] \ kill)

l=6: if (((n > 80) and (n <= 100))) then ...
  gen = {n, and}
  kill = {}
  OUT[6] = IN[7]  IN[8]
  IN[6] = gen  (OUT[6] \ kill)

l=7: y := 4
  gen = {}
  kill = {y}
  OUT[7] = IN[12]
  IN[7] = gen  (OUT[7] \ kill)

l=8: if (((n < 51) or (n <= 60))) then ...
  gen = {n, or}
  kill = {}
  OUT[8] = IN[9]  IN[10]
  IN[8] = gen  (OUT[8] \ kill)

l=9: y := 2
  gen = {}
  kill = {y}
  OUT[9] = IN[12]
  IN[9] = gen  (OUT[9] \ kill)

l=10: y := 3
  gen = {}
  kill = {y}
  OUT[10] = IN[12]
  IN[10] = gen  (OUT[10] \ kill)

l=11: y := 0
  gen = {}
  kill = {y}
  OUT[11] = IN[12]
  IN[11] = gen  (OUT[11] \ kill)

l=12: out := y
  gen = {y}
  kill = {out}
  OUT[12] = EXIT_lIVE
  IN[12] = gen  (OUT[12] \ kill)
```

Task 8: Solving the Liveness Analysis Equations

To compute the liveness sets, we solve the equations defined in Task 7 using an iterative fixed-point algorithm for backward dataflow analysis. In the notation used in class, the liveness sets at label ℓ are sometimes written as $\mathrm{LA}^{\circ}(\ell)$ (live before) and $\mathrm{LA}^{\bullet}(\ell)$ (live after), but in this report we use the more standard dataflow notation $\mathrm{IN}(\ell)$ and $\mathrm{OUT}(\ell)$.

Initialization

- For all CFG nodes ℓ:
$$\mathrm{IN}(\ell) = \varnothing, \qquad \mathrm{OUT}(\ell) = \varnothing.$$
- For the exit node ℓ_{exit}:
$$\mathrm{OUT}(\ell_{\text{exit}}) = (\text{variables assumed live at program exit}).$$

Fixed-Point Iteration The algorithm repeatedly updates the liveness sets until a fixed point is reached:

1. Update the "live-after" set:
$$\mathrm{OUT}(\ell) = \bigcup_{s \in \mathrm{succ}(\ell)} \mathrm{IN}(s).$$
2. Update the "live-before" set:
$$\mathrm{IN}(\ell) = \mathrm{gen}[\ell] \cup \big(\mathrm{OUT}(\ell) - \mathrm{kill}[\ell]\big).$$
3. Continue until no IN or OUT set changes.

Based on the CFG and dataflow equations of task7, this is the output liveness_in and liveness_out.

```
=== Task 8: liveness Analysis Result ===

l=0 : n := in
  IN = {in}
  OUT = {n}

```

```
l=1 : if ((n > 100)) then ...
  IN = {n}
  OUT = {n}

l=2 : y := 5
  IN = {}
  OUT = {y}

l=3 : if ((not (n <= 20))) then ...
  IN = {n}
  OUT = {n}

l=4 : if ((n < 41)) then ...
  IN = {n}
  OUT = {n}

l=5 : y := 1
  IN = {}
  OUT = {y}

l=6 : if (((n > 80) and (n <= 100))) then ...
  IN = {n}
  OUT = {n}

l=7 : y := 4
  IN = {}
  OUT = {y}

l=8 : if (((n < 51) or (n <= 60))) then ...
  IN = {n}
  OUT = {}

l=9 : y := 2
  IN = {}
  OUT = {y}

l=10 : y := 3
  IN = {}
  OUT = {y}

l=11 : y := 0
  IN = {}
  OUT = {y}

l=12 : out := y
  IN = {y}
  OUT = {}
```

7 Task 9

Task 9 uses the results of liveness analysis to eliminate dead code. Intuitively, an assignment is dead if it computes a value that is never used along any path from that point to the end of the program. Removing such assignments reduces the amount of work performed by the final code, and also reduces pressure on registers and memory.

Dead assignment detection. From Task 8 we have, for each program point ℓ, the liveness sets IN and OUT, describing the variables live before and after that point.

For a labeled command of the form $[x := a]_\ell$, we classify the assignment as dead if the assigned variable x is *not* in OUT. In that case, the value computed for x cannot affect any future computation or the program's final output, and therefore the assignment can safely be removed.

In the implementation we traverse all CFG nodes and collect the labels of dead assignments into a set. This set is later passed to the code generator.

Code elimination during generation. Rather than physically deleting nodes from the CFG, we integrate dead-code elimination into the code generation phase. When emitting code for a CFG node that corresponds to an assignment, we check whether its label is in the dead-assignment set:

- If the label is dead, we do not emit any arithmetic or move instructions for the assignment; we only preserve the control-flow structure (the label and outgoing jump).
- If the label is live, we emit the normal assignment code.

This approach keeps the CFG structure intact while still eliminating dead computations. In our tests, dead-code elimination reduced both the number of instructions and the number of memory operations, especially in programs with intentionally redundant assignments or unused branches.

8 Task 10

In Task 10 we use liveness information to perform register allocation and introduce spilling. Unlike Task 4, where we pretended to have one distinct `s`-register per variable, here we are restricted to the real RISC-V callee-saved registers `s1`–`s11`. Any additional variables must be stored in memory.

Interference graph and coloring. We first build an interference graph whose vertices are program variables. Two variables interfere if they are simultaneously live at some program point. Concretely, for each label ℓ we look at the liveness-after set OUT; every pair of distinct variables in this set gets an undirected edge in the interference graph. Intuitively, variables that are live together cannot share the same register.

Once we have the interference graph, we run a simple greedy graph-coloring algorithm. Variables are ordered by decreasing degree, and we assign each variable the lowest-numbered color (register) that is not already used by its neighbors. Colors range from 1 to 11, corresponding to registers `s1`–`s11`. If a variable cannot be assigned any free color, we mark it as spilled.

The result is a map from each variable to either a physical register number or a special value indicating that the variable is stored in memory.

Spills and memory layout. For spilled variables we reuse the same array that already holds WHILE variables on entry to the program (the array pointed to by `a0`). Each variable has a fixed slot in this array. When reading a spilled variable, we generate a small sequence that computes its address from `a0` and loads it into `t0`. When writing a spilled variable, we store the value in `t0` back into its array slot. Register-allocated variables, by contrast, live entirely in their assigned `s`-registers.

Prologue, epilogue, and code generation. The function prologue and epilogue were updated to respect the allocation:

- The prologue saves only the `s`-registers that are actually used by some variable, then loads initial values for those variables from the array into their assigned registers. Spilled variables stay in memory and are not loaded.
- The epilogue writes back only the register-allocated variables from their `s`-registers to the array, restores the saved `s`-registers, and returns. Spilled variables are already stored in memory each time they are updated, so no extra work is needed at the end.

The expression and command translators were refactored to use helper operations that either read from or write to a register (for allocated variables) or perform the appropriate load/store sequence (for spilled variables). Combined with the dead-code elimination from Task 9, this register allocation significantly reduces the number of live registers and memory operations in the final RISC-V code.

9 Task 11

9.1 Live-in–Based Automatic Harness Generation

To support performance testing and simplify command-line usage, our testing framework automatically generates a C harness for every `WHILE` program. The harness initializes only the variables that are live at program entry, all other variables will be set to zero.

We first scan the labeled program and identify all source variables in alphabetical order, excluding the keywords. Then the compiler reads the set of variables that are live at the program entry point. This information is computed during the previous liveness analysis and written to a file named `liveness_entry.txt`.

```
auto var_order = read_var_order_from_labeled("labeled_program.while");
auto livein_vars = read_livein("liveness_entry.txt");
```

Based on those two lists, the harness then emits a `C` file that declares a `vars[]` array for all source variables, but only assigning values from `argv` to the live-in variables in alphabetic order. The resulting `C` program prints the initial state of all variables, calls the compiled function and prints the updated final state for all variables. The testing pipeline is shown in Fig 5 below.

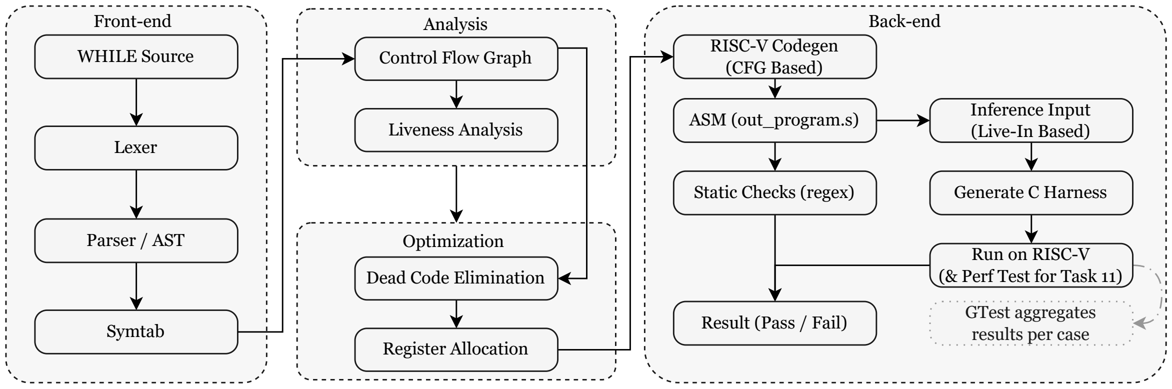

Figure 5: Testing pipeline of the compiler

9.2 Performance Tests

For the performance evaluation in Project 2, all WHILE benchmark programs are located in the `WhileFiles/perf_test` directory. For each `.while file`, the automatically generated `C harness` and `RISC-V assembly` files are placed in the `src/build` folder. To run and time a specific while file, use the following command:

```
// 1. generate the harness and assembly (place in src/build)
~/src$ ../tests/test_runner --gtest_filter=*/<WHILE_Name>
// 2. compile into an executable
~/src$ gcc -o wh build/<WHILE Name>.c build/<WHILE  Name>.s
// 3. run and measure execution time
~/src$ time ./wh <arg>
```

Across the full benchmark suite (Table 2), most of the workloads show only a small from Release 2 to Release 3, because those stress tests doesn't contain any dead assignments, thus liveness analysis and dead code elimination do not change their generated code size.

In this task, we extend the previous performance evaluation by developing a new benchmark `dead_while`, which intentionally includes several dead assignments in the while loop. This benchmark is designed to measure how liveness analysis, dead code elimination and register allocation (Task 7-10) improve the execution time of the compiler's generated code.

9.2.1 Performance Analysis

The following analysis focuses on analyzing the `dead_while` benchmark, which most clearly illustrates how each optimization affects final execution time. Inside the loop, junk1, junk2, junk3, junk4 are computed every iteration, but none of them are ever read later.

```
n := input;
i := 0;
x := 1;
y := 2;
sum := 0;
```

```
while [i < n] do
  junk1 := x * x + y * y + i * i; -- dead assignment
  junk2 := x + y + n + i; -- dead assignment
  junk3 := x * y + 1234; -- dead assignment
  junk4 := x * x * x + y * y * y; -- dead assignment
  sum := sum + x + y;
  i := i + 1
od;
output := sum
```

As shown in the table 2, the compiler of Project 1 performs the worst due to the excessive memory operations. Even a simple dead assignment would produce long stretches of load instructions and repeated address computation. Like discussed before in section 6, in Release 2, the compiler eliminates this by placing all variables in registers. However, the compiler still evaluates all arithmetic expressions for `junk1-junk4` on every iteration, the loop body is therefore remains large.

Release 3 applies liveness analysis to determine that all four junk variables are dead assignments at loop exit, allowing the compiler to skip their assignments when generates the final assembly. As a result, the large arithmetic blocks that appear in Release 2 are replaced by empty label transitions in Release 3 (Fig. 6), leaving only the updates to `sum` and `i`. Furthermore, register allocation reduces the live working set to just the registers required for the loop, avoiding spills and minimizing prologue/epilogue overhead.

Release 2 Codegen

```
L6:
  # l=6 assign
  mv t0, s4
  ...
  j L7

L7:
  # l=7 assign
  mv t0, s4
  ...
  j L8
```

```
L8:
  # l=8 assign
  mv t0, s4
  ...
  j L9

L9:
  # l=9 assign
  mv t0, s4
  ...
  j L10
```

Release 3 Codegen

```
L5:
  ...
  j L6 # true -> body

L6:

L7:

L8:

L9:

L10:
  ...
```

Figure 6: Comparison of loop-body code generation in Release 2 (with dead assignments) and Release 3 (after dead code elimination).

Workload	Input	Dead Code Lines	Time Perform	Project 1	Project 2	
					Release2	Release3
closest_prime	387096383	0	real	5m21.974s	2m33.711s	2m33.619s
			user	5m21.967s	2m33.650s	2m33.613s
			sys	0.005s	0.036s	0.004s
Square	10^5	0	real	1m13.995s	40.665s	40.645s
			user	1m13.974s	40.649s	40.562s
			sys	0m0.009s	0.008s	0.001s
GCD(Sub)	(1, 10^9)	0	real	10.744s	5.733s	5.013s
			user	10.742s	5.726s	5.008s
			sys	0.000s	0.005s	0.005s
GCD(Mod)	(102334155, 102334156)	0	real	0.672s	0.370s	0.371s
			user	0.671s	0.365s	0.365s
			sys	0.001s	0.005s	0.005s
Prime Count	10^4	0	real	3.218s	1.509s	1.505s
			user	3.216s	1.499s	1.499s
			sys	0.001s	0.009s	0.005s
Factorial (Addition)	12	0	real	2.796s	1.399s	1.399s
			user	2.794s	1.394s	1.398s
			sys	0.001s	0.005s	0.001s
dead_while	10^9	4	real	38.777s	17.416s	4.776s
			user	38.660s	17.406s	4.774s
			sys	0.001s	0.013s	0.001s

Table 2: Execution Time Comparison for Workloads between Project 1 and Project 2.

Overall, our final version of the compiler achieves most of its performance gains by eliminating dead assignments and ensuring that all live state remains in registers, resulting in significantly more efficient performance execution.

10 Contributions

Release	Tasks	Contributor
Release 1	Task 2	Yue Hu
Release 2	Task 3 & Task 4	Youssef Amin
	Task 5 & Task 6	Zhuo Li
Release 3	Task 7 & Task 8	Yue Hu
	Task 9 & Task 10	Youssef Amin
	Task 11	Zhuo Li

All members participated in debugging throughout the project. The project report was also revised and finalized by all team members. The GitHub repository we worked together can be found here: Terminators.