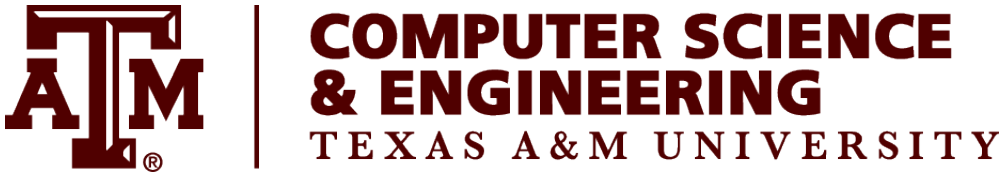
Instruction:

1. You must use the LaTeX template: https://www.overleaf.com/read/qqntsyfqwwdb. Make a copy of the template, and you can edit it. Overleaf is the recommended platform.
    - The template includes further guidance on the content of the report. A snapshot of the report template is included here.
    - Important: Remove all instructions and placeholder text before you submit your final report.
    - Ask questions on Campuswire for clarification.
2. There is a *preliminary* report due on **November 16, 2025**:
    - At a minimum, you must report what is working and what is not working with your compiler before going to the evaluation session.
    - As a preliminary report, you should not spend a lot of time on this. If you are short on time, focus on coding and finishing the requirements.
3. The *final* report is due with your compiler's code on **December 7, 2025**:
    - All Sections are required.
    - We expect the "Update after Evaluation" Section.
    - The report is mandatory. If you do not submit this, we will penalize you heavily (up to half of the credit).
    - A reminder that **no late submission is allowed** for both the report and the code.



CSCE 434/605: Compiler Design

Fall 2025

Project Report

December 4, 2025

Author	*UIN*
Nitish Malluru	932007196

Contents

1. Summary

1.1 IR Generator

To create the unoptimized IR we need to traverse the AST. Each function is a different CFG. Each CFG consists of a set of basic blocks. A basic block is a set of instructions that are guaranteed to run together, so whenever a branch is encountered we create a new basic block. This occurs in loops and if statements, so they are broken down into condition, and body blocks. For the individual instructions I use 4 parameter TACS that best match up with the DLX instruction set. I have a few helper TACS like MOV, CALL, and RET that are substituted with actual instructions during code generation. One other responsibility of the IR Generator is to set aside space for locals and temps. For both local and global variable declarations, 4 byte space are allocated in memory for all types except array type. ArrayTypes dimensions are multiplied together and then by 4 to allocate space for.

To make implementing optimizations easier I implemented SSA. In a first pass I do dominator analysis to find the merge points for separate branches that eventually merge. With this information I could proceed with SSA. For every declaration of the same variable I increment its version. then when two different versions of merge I place a phi instruction associating to each branch's definition of the variable I am merging.

The point of SSA is to make optimization easier as now variable are only declared once. Common Subexpression Elimination becomes a lot easier to compute since variables are only declared once, so we can more easily eliminate redundant expressions. Copy and Constant Propagations work similarly in that they use lattices to map if a value can have a constant value propagated to it. Using Data Flow analysis we can use the lattice to trace if a variable will always be a certain variable or constant. DCE works by tracking what is live at the out block and tracking what is used to define variable in the live out set. Constant Folding tries to eliminate impossible branches and evaluates expressions where all arguments are constants.

1.2 Register Allocator

Moving on to the Register Allocator I first eliminate all Phi Instructions. Then I use the Chaitin Briggs algorithm to allocate all the registers. I first calculate the live in and out for every block until they converge on a consistent definition. Then I iterate through the cfg in reverse, when I hit a variable definition I can add an edge from all the variables that are live out in the block. Then I can begin actually allocating registers. I first start by popping the first node with less than K degrees. I update the graph to remember this removal from the graph and continue. If it is impossible to pop all the nodes I spill the node with the highest number of edges.

1.3 Code Generator

The Code Generator uses a two passes. In the first pass I replace all TACs with there actual instructions. I also create function prologues to save arguments, return address, stack pointer, and frame pointer. For function epilogues I save the return value, and restore return address, stack pointer, and frame pointer. In the second pass I assign actual PC counts for where all the blocks are located since I now know the length of each block.

Feature Check List

- ✓ SSA
- ✓ Type Checker
- ✓ Constant Folding
- ✓ Constant Propagation
- ✓ Copy Propagation
- ✓ Orphan Function Elimination
- ✓ Uninitialized Variables
- ✓ Common Subexpression Elimination
- ✓ Dead Code Elimination
- ✓ Graph-coloring Register Allocation
- ✗ Linear Scan Register Allocation
- ✗ Register Allocation - Coalescing

KLOC

Code:
Comment:
Total:

2. Implementation

2.1 Design and Implementation Details

- **Uninitialized variables:** During IR Generation I track all initialized globals and initialized locals in distinct sets. Whenever an instruction needs uses a variable I check if they exists within theses sets. If absent a warning and save to a list to add. After I am done processing the entire function AST I add initialization instructions to a block after the entry block. When a variable is defined it is added to the proper set.
- **IR instruction:** IR is generated using the visitor pattern to to transform the Abstract Syntax Tree nodes to CFGs constructed from TAC instructions. Basic arithmetic and comparison nodes directly map to tacs, however IF, WHILE, and REPEAT nodes require more complex instructions. For example in IF nodes, the generator will create a thenBlock, joinBlock, and an optional elseBlock, using different branch instructions. Loops are constructed using a headerBlock, bodyBlock, and exitBlock.
- **Stack Frame Layout and Epilogues:** Function calls and returns have specific TACs that notify the code generator when to add prologue and epilogue blocks. These blocks modify the stack frame. The stack is laid out with the return value slot located at (FP+8), sitting immediately above the saved Frame Pointer (FP+4) and Return Address (FP+0).
- **Function Calls and Return Values:** At every function call push params onto the stack and save all our registers, then in the called function pop all the params. After every return we read the saved return value in the stack.

```
FP+8:  Return value slot

FP+4:  Saved Frame Pointer (old FP)

FP+0:  Saved Return Address (old RA)

FP-4:  Local variable 1

FP-8:  Local variable 2

...

FP-N:  Temporary variables
```

- **Heuristics used for Graph Coloring:** I simply remove the node with the greatest number of edges to reduce the amount of variables that would need to be spilled in future iterations of graph coloring.
- **Spilled variables:** I reserve R26 and R27 to be used for spilling, so R1–25 can be used as data registers. When a spilled register is used, its address is loaded into the first available spilled register. The next instruction then loads the value into the same register. This does not clobber the register since the address is no longer needed again as the value lives directly in the register.
- **Global Load and Commit Strategy** Global Variables are loaded at function entry and committed

before function end or function call. This allows us to optimize them as any other regular variable. Global Variables exists at -4 offsets from the Global Pointer.

- **Float Detection for Instructions** There exists many float alternatives for instructions so we need to know when to choose the correct one. To accomplish this TACS that have both an integer and float form have an isfloat instance variable. When generating a TAC we can see if one of the operands is a tac and set the flag to true if so.
- **Array Handling** ArrayType tracks element type and dimensions. Arrays live contiguously in memory, either GP-relative (globals) or FP-relative (locals). The IR generator uses AddaGP/AddaFP + Load/Store to compute and use element addresses. Arrays are passed as pointers; no extra loads for array params. Multi-dimensional arrays use element size computations to handles sub-array sizes, so index scaling is correct across dimensions.

2.2 Testcases

Demonstrate that the compiler is working, focusing on optimizations: Either pick from public testcases or create one yourself, list the tests (see here: https://www.overleaf.com/learn/latex/Code_listing) and similar to PA4 dotgraph assignment, include pre- and post-optimization Dot graphs. In the post-optimization graph, noted where the optimation takes place

* If you have used generative AI for any coding, you have to disclose it here. For example, the tool used, the prompt used, the code generated, and the part of the compiler

I used Github Copilot tab completions. Outside of that I used AI to architect SSA Elimination aswell as Code Gen transformation. I used Gemini with the following prompts.

Updates after Evaluation

Only include this in the final report - not the prelim

List any fixes/updates you made after the evaluation session, e.g., if any tests fail earlier, show that they work now. If there is none, say so.

3. Reflection

What went well

The frontend went really well, I think PA3 was very enjoyable in building the AST. I did not have too many hiccups while building either the Type Checker or Interpreter. And it was very cool to finally get some output from our code.

IR Generation before I had to refactor it for code generation, went pretty smooth. This was the first time we could visualize the compiler output because we created SketchViz graphs. This greatly helped while building the optimization since you could see what was being changed in each pass.

The satisfaction of seeing the compiler work at the end, is definitely worth it. It demystifies one of the major pillars of Computer Science, and combines knowledge of Programming Languages, Data Structures, and Micro Architecture.

What did not go well

Debugging code generation was extremely painful to diagnose since it was difficult to map program code to register operations. Using a debugger with DLX in debug mode helped but still was extremely tedious. The ASM output was the key here but you had to combine it with register allocations to see how data traversed around the registers.

Although I did not do a group project, from the Syllabus I felt like I was the stoic. I was able to figure out things on my own, but it would require way less effort if I actually asked for help. I recommend that you start early so you have enough time to get help.

Advice

I would read the entire Project document front from back, so that you are actively thinking about design considerations you can implement now, that will make your life easier in the future. I made a mistake of not thinking about memory when I began backend development which forced me to make many changes during the development of code generation. I also completely overlooked global variables, which are handled differently then I originally expected.

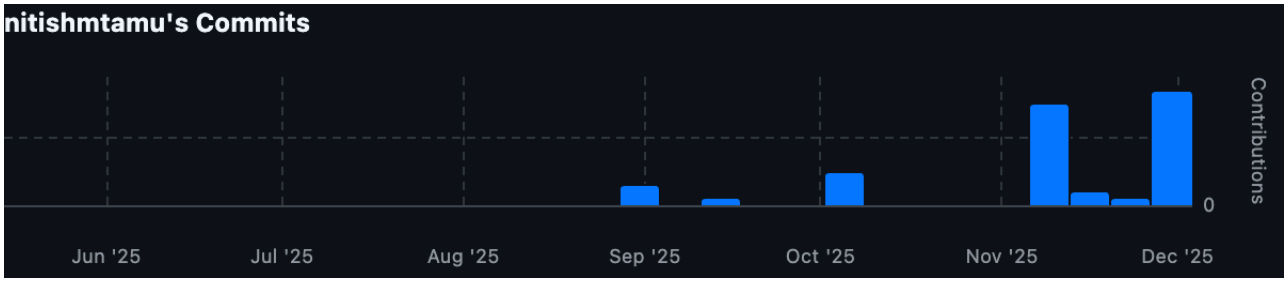
Start backend development earlier than you think. PA4 really snuck up on me and I had to rush to be able to hit all the requirements I wanted to fit into my compiler.

Also SSA is not to difficult to implement and make life a lot easier when doing optimizations. It is definitely worth the time!

4. Work Arrangement and Contribution

Code Contribution Graph

The graph goes back to June since I just used the same repo I was pulling all the Starter Code and Tests from.



Solo version

I affirm that this report describes the correct status of my submission as of December 4, 2025.

ON MY HONOR, AS AN AGGIE, I HAVE NEITHER GIVEN NOR RECEIVED UNAUTHORIZED AID ON THIS ACADEMIC WORK.

{Nitish Malluru}