

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH



NGUYỄN LÝ NGÔN NGỮ LẬP TRÌNH
HỌC PHẦN MỞ RỘNG (CO300C)

Phase 3
Sinh mã MIPS cho toàn bộ ngôn ngữ MT22

GVHD: TS Nguyễn Hứa Phùng
SV thực hiện: Kha Sang – 2010576

Tp. Hồ Chí Minh, Tháng 05/2023

Mục lục

1	Thiết kế sinh mã trong ngôn ngữ MIPS	3
1.1	Ngôn ngữ MIPS	3
1.1.1	Giới thiệu	3
1.1.2	Một số đặc trưng	3
1.2	Giải thuật sắp xếp thanh ghi đề xuất	4
1.2.1	Sinh danh sách các danh hiệu	4
1.2.2	Phân tích thời gian sống của danh hiệu	6
1.2.3	Tối ưu hóa cấp phát thanh ghi	8
2	Kiến trúc bộ sinh mã MT22 sang MIPS	10
2.1	Tổng quan kiến trúc sinh mã	10
2.2	Chi tiết module sinh mã	12
2.2.1	Identifier Collector	12
2.2.2	Liveness Analysis	12
2.2.3	Graph Allocator	13
3	Các bước hiện thực và kiểm thử	14
3.1	Các bước hiện thực kiến trúc	14
3.2	Chiến lược kiểm thử việc quản lý thanh ghi	14
3.2.1	Phép cộng cơ bản	14
3.2.2	Phép cộng nhiều biến, và có biến không cần sử dụng	15
3.2.3	Phép cộng nhiều biến, vượt số lượng thanh ghi	16
4	Tiến độ hiện tại và hướng phát triển	18
4.1	Tiến độ hoàn thành	18
4.2	Hướng phát triển	18
	References	19

Danh sách hình vẽ

1	Chức năng các thanh ghi trong MIPS	3
2	Lưu đồ giải thuật sinh dãy danh sách danh hiệu	5
3	Lưu đồ giải thuật sinh Flow Graph của khối chương trình	7
4	Lưu đồ giải thuật sinh bảng ánh xạ thanh ghi	9
5	Kiến trúc sinh mã MIPS từ mã MT22	10
6	Sơ đồ lớp mô tả kiến trúc CodeGen	11
7	Lớp IdentifierCollector	12
8	Lớp LivenessAnalysis	13
9	Lớp GraphAllocator	13
10	Test 500	14
11	Check 500	15
12	Test 501	15
13	Check 501	16
14	Test 502	16
15	Check 502	17

1 Thiết kế sinh mã trong ngôn ngữ MIPS

1.1 Ngôn ngữ MIPS

1.1.1 Giới thiệu

MIPS là viết tắt của Microprocessor without Interlocked Pipeline Stages, là một kiến trúc vi xử lý (processor architecture) được sử dụng trong thiết kế vi xử lý dựa trên bộ lệnh RISC (Reduced Instruction Set Computing). MIPS là một kiến trúc vi xử lý phổ biến trong các thiết bị nhúng, máy tính cá nhân và máy chủ.

Ngôn ngữ lập trình MIPS rất phổ biến trong giảng dạy các khái niệm lập trình và kiến trúc máy tính, nhất là trong các khóa học về hệ thống máy tính, hệ điều hành, và các khóa học về lập trình nhúng. MIPS sử dụng các chỉ thị đơn giản và dễ hiểu, và có số lượng chỉ thị giới hạn, giúp cho người mới học dễ dàng tiếp cận và hiểu được cách thức hoạt động của máy tính.

Để lập trình trên MIPS, người dùng thường sử dụng các trình biên dịch (compiler) hoặc trình dịch ngược (assembler) để chuyển đổi các lệnh mã nguồn thành mã máy để thực thi trên vi xử lý MIPS.

Một số dạng lệnh thường được sử dụng trong ngôn ngữ lập trình MIPS bao gồm các lệnh số học (arithmetic), các lệnh logic, các lệnh nhảy (jumps) và các lệnh thao tác với bộ nhớ (memory operations). Các lệnh này được thực hiện bằng cách sử dụng các thanh ghi (registers) trong vi xử lý MIPS, và các địa chỉ bộ nhớ để lưu trữ dữ liệu và chỉ thị.

1.1.2 Một số đặc trưng

MIPS là một kiến trúc vi xử lý RISC (Reduced Instruction Set Computing), nghĩa là chỉ sử dụng một tập hợp nhỏ các lệnh máy đơn giản và cơ bản để giảm độ phức tạp của vi xử lý. Điều này đưa đến một số lợi ích, bao gồm tốc độ xử lý nhanh hơn, tiêu thụ điện năng thấp hơn và dễ dàng để thiết kế và xây dựng.

MIPS có 32 thanh ghi (registers), mỗi thanh ghi có độ rộng 32 bit. Những thanh ghi này được sử dụng để lưu trữ dữ liệu tạm thời và kết quả tính toán. Các thanh ghi này được đánh số từ 0 đến 31, và mỗi thanh ghi có một tên gọi đặc biệt như *zero*, *at*, *v0*, *v1*, *a0* đến *a3*, *t0* đến *t9*, *s0* đến *s7*, *k0* và *k1*, *gp*, *sp*, *fp* và *ra*.

Assembly Programming

MIPS Registers		
Register name	Number	Usage
zero	0	constant 0
at	1	reserved for assembler
v0, v1	2 ~ 3	expression evaluation and results of a function
a0 ~ a3	4 ~ 7	arguments 1 - 4
t0 ~ t7	8 ~ 15	temporary (not preserved across call)
s0 ~ s7	16 ~ 23	saved (preserved across call)
t8, t9	24, 25	temporary (not preserved across call)
k0, k1	26, 27	reserved for OS kernel
gp	28	pointer for global area
sp	29	stack pointer
fp	30	frame pointer
ra	31	return address (used by function call)

UOU 1-5

Hình 1: Chức năng các thanh ghi trong MIPS

MIPS cũng có một bộ nhớ chính (main memory) để lưu trữ dữ liệu và chương trình. Địa chỉ bộ nhớ được biểu diễn bằng các số nguyên không dấu, có độ rộng 32 bit, và được đánh số từ 0 đến $2^{32} - 1$.

MIPS cung cấp một tập hợp các lệnh đơn giản để thực hiện các thao tác cơ bản như thực hiện các phép tính số học, các phép logic, các phép nhảy, các phép thao tác với bộ nhớ, v.v. Mỗi lệnh trong MIPS có độ dài cố định là 32 bit, với các trường khác nhau để xác định loại lệnh, thanh ghi đích và các giá trị hằng số hoặc địa chỉ.

Một số ứng dụng của MIPS bao gồm các máy tính nhúng, thiết bị điện tử tiêu dùng và các máy tính cá nhân. Ngoài ra, MIPS cũng được sử dụng trong các hoạt động nghiên cứu và giảng dạy về kiến trúc máy tính và lập trình hệ thống.

1.2 Giải thuật sắp xếp thanh ghi đề xuất

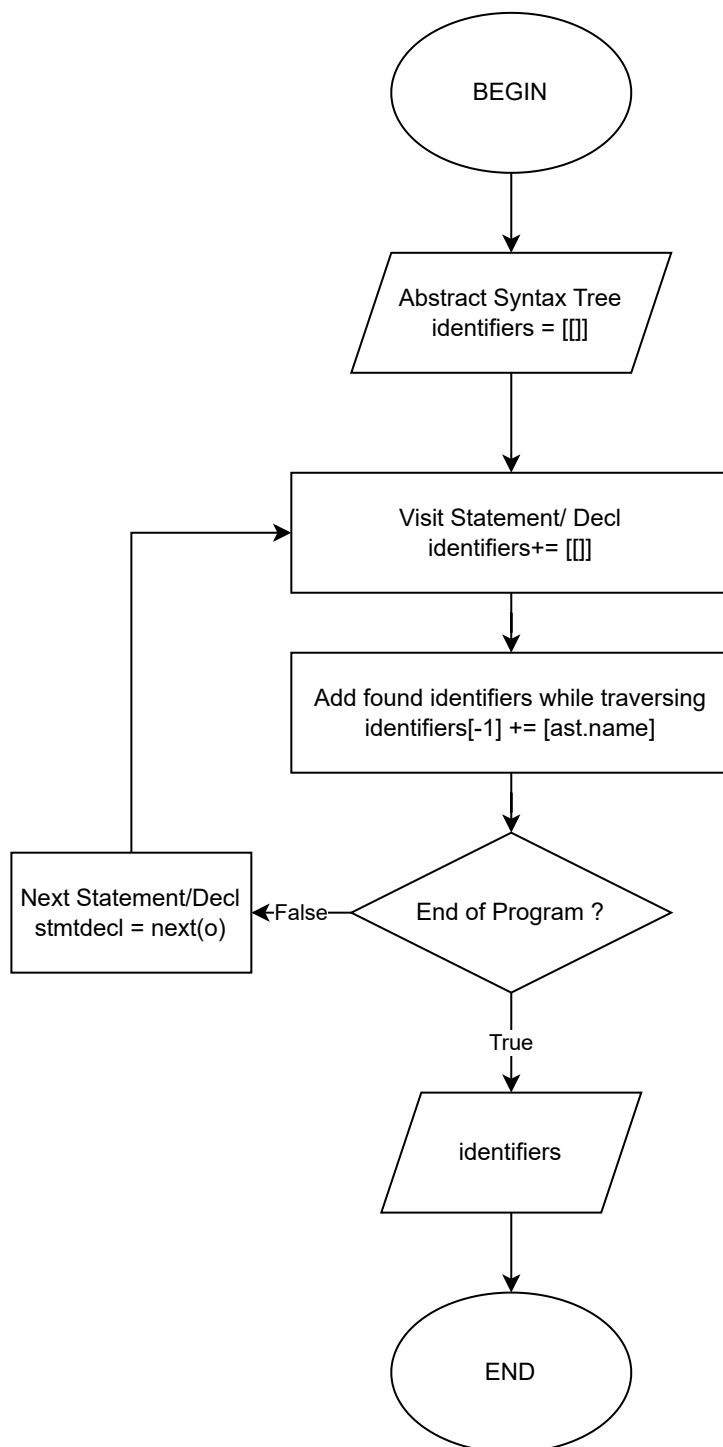
Do số lượng thanh ghi trên ngôn ngữ MIPS là có hạn (rất hạn chế so với các ngôn ngữ Stack-based như Java, Python,...), chúng ta cần phải thiết kế giải thuật cấp phát thanh ghi cho các khai báo, hàm gán,... nhằm tối ưu hóa số lượng biến có thể lưu trữ trong chương trình mà vẫn đảm bảo chương trình chạy được và đúng như yêu cầu.

Quá trình cấp phát thanh ghi hoạt động với đầu vào là Cây cú pháp trừu tượng (Abstract Syntax Tree) và đầu ra là Danh sách ánh xạ các thanh ghi ứng với mỗi danh hiệu được sử dụng trong chương trình.

Có 3 bước lớn để hiện thực giải thuật, bao gồm (1) Sinh danh sách các danh hiệu, (2) Phân tích thời gian sống của danh hiệu, (3) Tối ưu hóa cấp phát thanh ghi

1.2.1 Sinh danh sách các danh hiệu

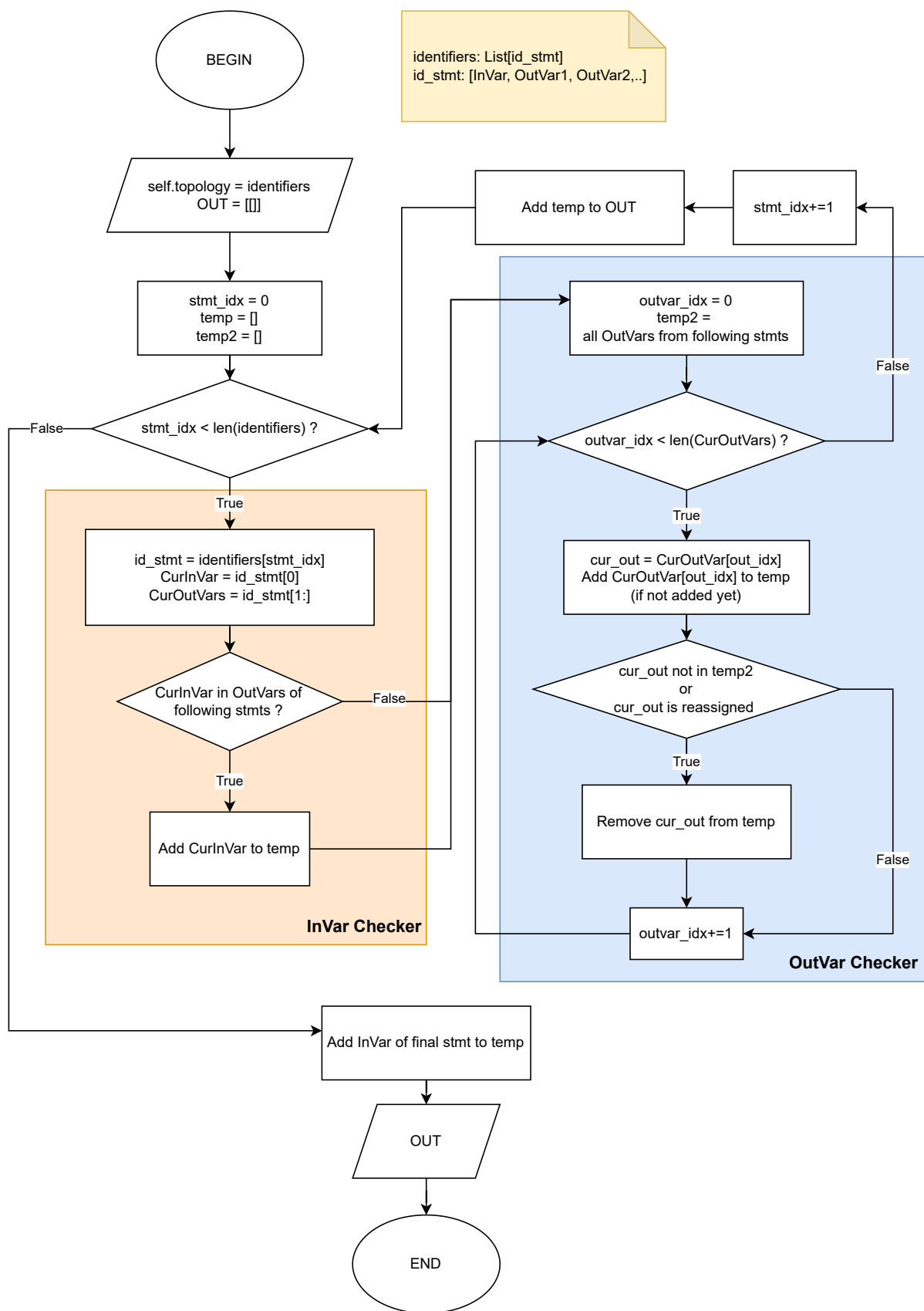
- **Input:** Cây AST từ bước ASTGeneration.
- **Output:** Danh sách của danh sách các danh hiệu được sử dụng hoặc khai báo ở mỗi câu lệnh.
- **Mô tả giải thuật:**
 - Sử dụng Visitor Pattern, ta tiến hành duyệt các node trên cây cú pháp.
 - Với mỗi dòng lệnh (Stmt/ VarDecl), ta tiến hành thu thập tên tất cả các danh hiệu vào trong 1 danh sách con, thứ tự duyệt từ trái sang phải. Đầu ra của chương trình là dãy các danh sách con đó.
 - Do hiện tại các biến đều là Scalar Variable, ta có thể quy ước danh hiệu đầu tiên của mỗi danh sách con là InVar (biến được gán) và các danh hiệu còn lại là OutVar (biến được sử dụng). Ta sẽ áp dụng tính chất này để áp dụng cho các bước tiếp theo.
- **Lưu đồ giải thuật:**



Hình 2: Lưu đồ giải thuật sinh dãy danh sách danh hiệu

1.2.2 Phân tích thời gian sống của danh hiệu

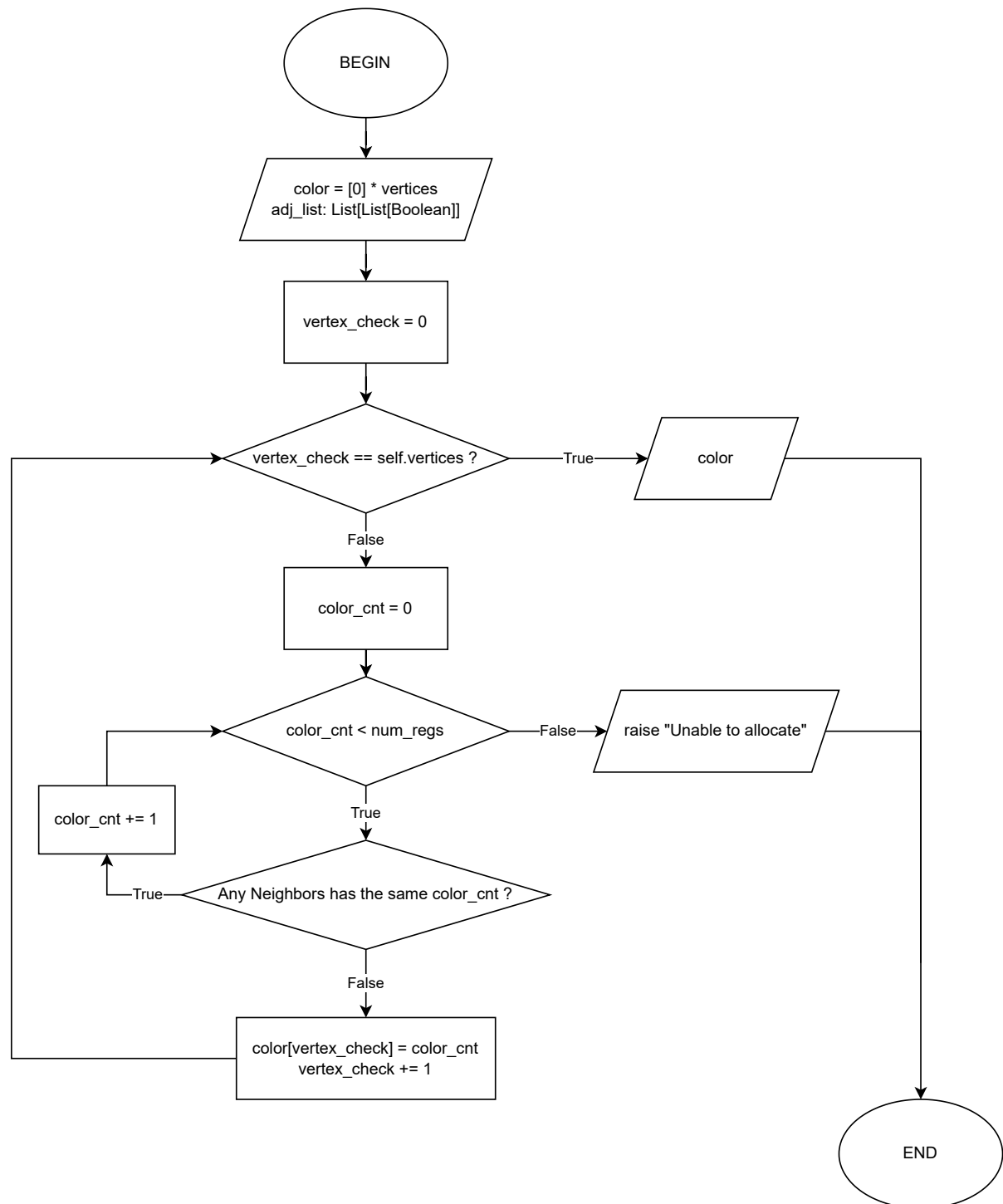
- **Input:** Danh sách của danh sách các danh hiệu có mặt tại mỗi dòng lệnh
- **Output:** Danh sách của danh sách các danh hiệu cần "tồn tại" tại mỗi dòng sử dụng hoặc khai báo.
- **Mô tả giải thuật:**
 - Ta tiến hành hai tác vụ thực hiện tuần tự: xem xét InVar, sau đó là các OutVar của mỗi dòng lệnh.
 - Với InVar, ta kiểm tra nó có được sử dụng lại hay không (tức kiểm tra trong OutVar của các câu lệnh tiếp theo). Nếu có ta thêm nó vào danh sách sống tạm thời.
 - Với mỗi OutVar, ta kiểm tra xem nó có được khai báo lại không (tức kiểm tra trong InVar của các khai báo tiếp theo) (a) hoặc có được sử dụng lại hay không (tức kiểm tra trong OutVar của các khai báo tiếp theo) (b). Nếu (a) đúng còn (b) sai thì loại nó ra khỏi danh sách sống tạm thời (nếu có tồn tại).
 - Ngoài ra, quy ước rằng InVar cuối cùng sẽ được sử dụng (cho hàm in giá trị), do đó tiến hành thêm vào danh sách sống tạm thời.
 - Thực hiện tuần tự các bước trên cho đến khi kết thúc chương trình. Đầu ra của bước này là dãy các danh sách sống của các danh hiệu ứng với từng câu lệnh.
- **Lưu đồ giải thuật:**



Hình 3: Lưu đồ giải thuật sinh Flow Graph của khối chương trình

1.2.3 Tối ưu hóa cấp phát thanh ghi

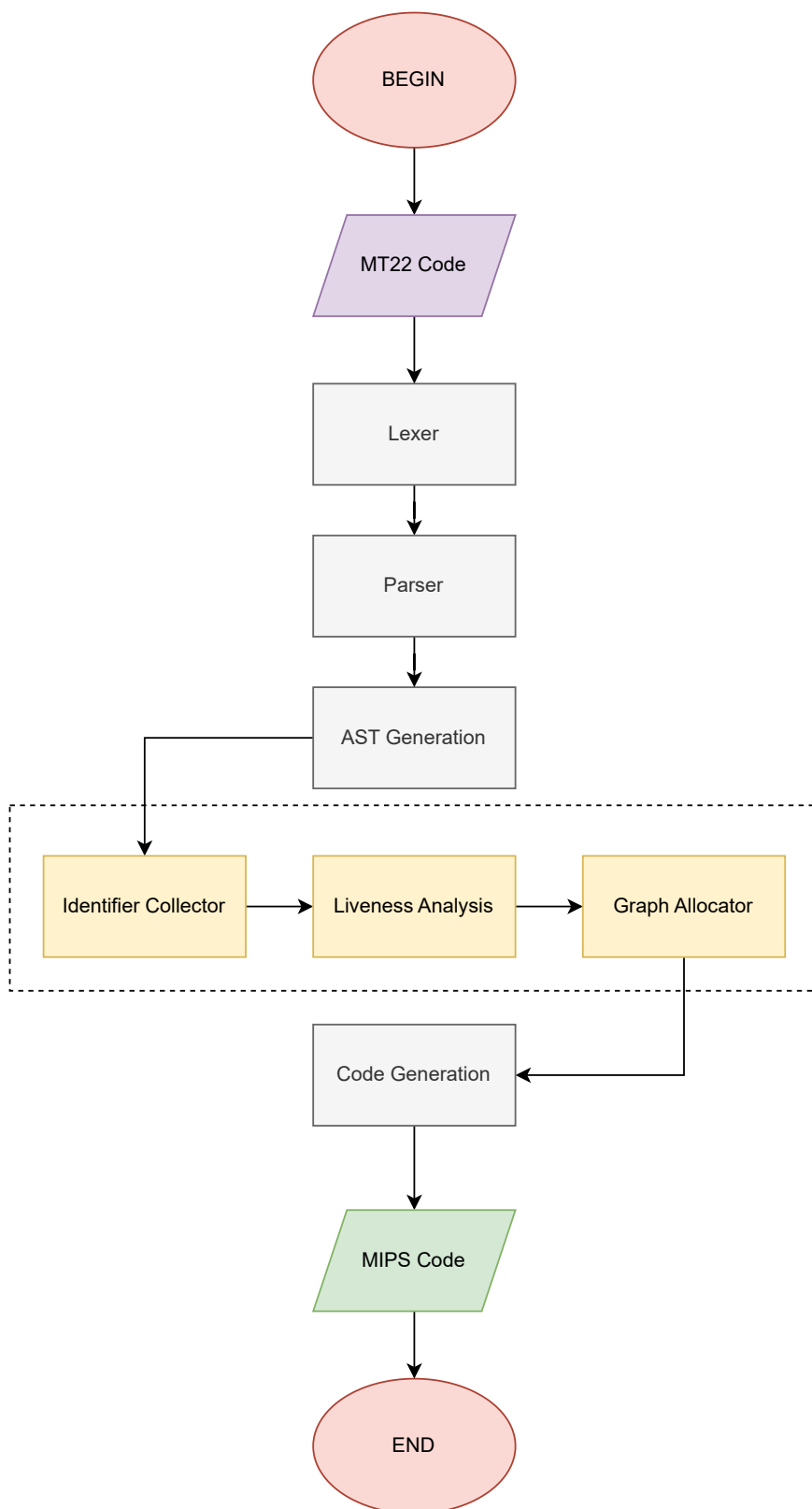
- **Input:** Danh sách của danh sách các danh hiệu sống tại mỗi dòng lệnh và số thanh ghi cho phép
- **Output:** Bảng ánh xạ các biến và thanh ghi
- **Mô tả giải thuật:**
 - Ta áp dụng [Giải thuật Chaitin](#) để giải quyết bài toán, tuy nhiên có biến đổi đôi chút để đơn giản hóa việc hiện thực (có đánh đổi về tính tối ưu nhưng nhìn chung vẫn chấp nhận được).
 - Xây dựng danh sách liên kế bằng dãy các danh sách sống tạm thời. Từ đây ta sẽ tính được số giá trị danh hiệu phân biệt (cũng là số nút của đồ thị)
 - Tiến hành tạo mảng 1 chiều có kích thước bằng số đỉnh của đồ thị với giá trị đồng loạt là 0. Đây sẽ là nơi lưu trữ giá trị ánh xạ thanh ghi của từng node.
 - Duyệt từng phần tử trong mảng vừa tạo, lần lượt gán giá trị thanh ghi từ 1 đến khi số lượng tối đa. Dừng lại khi tìm được giá trị thanh ghi mà các node trực tiếp lân cận không có cùng giá trị.
 - Thực hiện cho đến khi đến cuối mảng. Lưu ý rằng nếu không tìm được thanh ghi phù hợp cho danh hiệu bất kì, chương trình sẽ trả về lỗi.
- **Lưu đồ giải thuật:**



Hình 4: Lưu đồ giải thuật sinh bảng ánh xạ thành ghi

2 Kiến trúc bộ sinh mã MT22 sang MIPS

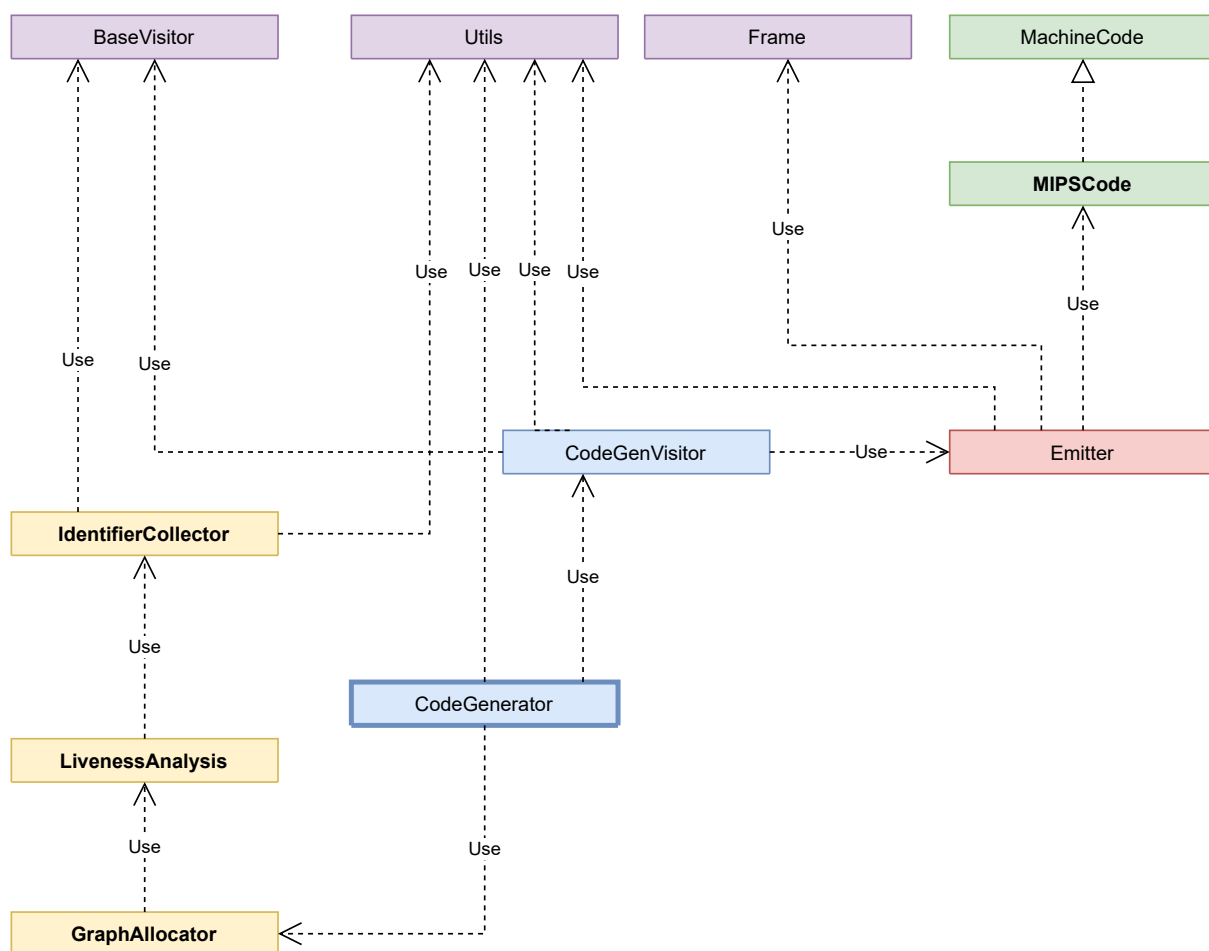
2.1 Tổng quan kiến trúc sinh mã



Hình 5: Kiến trúc sinh mã MIPS từ mã MT22

Sơ lược các module trong kiến trúc CodeGen:

- **BaseVisitor:** lớp trừu tượng cho Visitor Pattern.
- **Utils:** cung cấp các hàm phụ trợ
- **Frame:** cung cấp Frame Indexing (không sử dụng ở Phase 3) và Label Indexing (sử dụng để tạo label cho các cấu trúc rẽ nhánh và vòng lặp).
- **MachineCode:** lớp trừu tượng cho các ngôn ngữ lập trình mã đích.
- **MIPSCode:** lớp hiện thực các lệnh từ MachineCode, ứng với hợp ngữ MIPS.
- **CodeGenVisitor:** lớp giao tiếp giữa bên ngoài và bộ sinh mã CodeGenerator (Module Test gọi đối tượng thuộc lớp này để kiểm thử kết quả).
- **Emitter:** module trung gian dùng để điều hướng giữa bộ sinh mã CodeGen để gọi về mã đích thích hợp.
- **IdentifierCollector, LivenessAnalysis, GraphAllocator:** các module hỗ trợ tính toán việc sắp xếp thanh ghi trên từng danh hiệu, thực hiện trước bước sinh mã diễn ra.
- **CodeGenerator:** là trung tâm của bộ sinh mã, nhận cây AST từ mã nguồn và tiến hành gọi các hàm Emitter để xuất mã đích.



Hình 6: Sơ đồ lớp mô tả kiến trúc CodeGen

2.2 Chi tiết module sinh mã

2.2.1 Identifier Collector

- **self.identifiers**: khởi tạo dãy danh sách danh hiệu rỗng
- **Các hàm Visitors**: Các hàm visit dùng để duyệt các nút trên cây AST. Tiến hành thêm danh hiệu sao cho phù hợp

```
1  ✓ from Utils import *
2  ✓ from Visitor import *
3
4  ✓ class IdentifierCollector(BaseVisitor, Utils):
5  ✓     def __init__(self):
6      self.identifiers = [[]]
7
8  >     def visitBinExpr(self, ast, param): ...
11
12 >     def visitUnExpr(self, ast, param): ...
14
15 >     def visitId(self, ast, param): ...
17
18 >     def visitAssignStmt(self, ast, param): ...
22
23 >     def visitBlockStmt(self, ast, param): ...
26
27 >     def visitVarDecl(self, ast, param): ...
31
32 >     def visitFuncDecl(self, ast, param): ...
34
35 >     def visitProgram(self, ast, param): ...
```

Hình 7: Lớp *IdentifierCollector*

2.2.2 Liveness Analysis

- **self.topology**: Dãy danh sách các danh hiệu sử dụng ở mỗi câu lệnh, là output của giai đoạn Identifier Collector.
- **self.flow**: Dãy danh sách danh hiệu sống tạm thời
- **self.ids**: Dãy các danh hiệu đôi một khác nhau có mặt trong toàn bộ *self.flow*.
- **getUniqueIds(self, out)**: Hàm hỗ trợ tính toán *self.ids*
- **createFlowGraph(self)**: Hàm tính toán *self.flow*

```
src > main > mt22 > codegen > LivenessAnalysis.py > LivenessAnalysis > createFlowGraph
1  class LivenessAnalysis:
2      def __init__(self, topology):
3          self.topology = topology
4          self.flow = []
5          self.ids = []
6
7  >  def getUniqueIds(self, out): ...
13
14 >  def createFlowGraph(self): ...
```

Hình 8: Lớp *LivenessAnalysis*

2.2.3 Graph Allocator

- **self.unique_id_list**: Danh sách các node của đồ thị
- **self.flow**: Dãy của danh sách sống tạm thời của các node trong mỗi câu lệnh
- **self.V**: Số đỉnh của đồ thị
- **self.color_global**: Mảng ánh xạ Danh hiệu - Thanh ghi, là đầu ra của module này.
- **self.graph**: Ma trận liên kề, đại diện cho đồ thị danh hiệu.
- **flowToAdj(self)**: Hàm tính toán *self.graph* từ *self.flow*.
- **isNeighborColor(self)**: Hàm hỗ trợ, trả về giá trị Boolean kiểm tra xem các node lân cận có cùng màu (dùng cùng thanh ghi) hay không.
- **graphColorRecursionCheck(self, num_regs, color, v)**: Hàm hỗ trợ tính toán đệ quy để lần lượt thay đổi gán thanh ghi trên biến "color" được truyền tham chiếu (đối số là color_global).
- **graphColoring(self, num_regs)**: Hàm tính toán *self.color_global*.

```
1  class GraphAllocator():
2
3      def __init__(self, unique_id_list, flow):
4          self.unique_id_list = unique_id_list
5          self.flow = flow
6          self.V = len(unique_id_list)
7          self.color_global = [0] * self.V
8          self.graph = [[0 for _ in range(self.V)] for _ in range(self.V)]
9
10 >  def flowToAdj(self): ...
19
20      # Check for Neighbors if they have the same color
21 >  def isNeighborColor(self, v, color, c): ...
26
27      # Recursion to get number of colors needed
28 >  def graphColorRecursionCheck(self, num_regs, color, v): ...
38
39      # Coloring operation for graph
40 >  def graphColoring(self, num_regs): ...
```

Hình 9: Lớp *GraphAllocator*

3 Các bước hiện thực và kiểm thử

3.1 Các bước hiện thực kiến trúc

- **Giai đoạn 1:** Sinh mã JVM với các phép khai báo và gán
 - Tiến hành định nghĩa Token và Grammar trong MT22.g4 có sẵn.
 - Hiện thực các visitor hỗ trợ duyệt VarDecl và AssignStmt.
 - Tiến hành mở rộng file CodeGeneration.py để có thể sinh mã JVM.
 - Viết các testcase đảm bảo rằng việc sinh và chạy code là hợp lệ.
- **Giai đoạn 2:** Tìm hiểu thuật toán sắp xếp thanh ghi
 - Hiện thực các module cần thiết (các file mới) để hiện thực giải thuật.
 - Tiến hành tích hợp các module vào CodeGenerator.py và kiểm thử việc sinh bằng ánh xạ.
- **Giai đoạn 3:** Sinh mã MIPS với các phép khai báo và gán cho phép "+" số nguyên.
 - Tạo class MIPSCode trong MachineCode để quy định format của 1 file MIPS.
 - Tiến hành thay đổi CodeGeneration.py và Emitter.py (nếu cần) để hỗ trợ việc sinh mã.
 - Tạo các testcase và tiến hành chạy mã sinh ra trên MIPS Assembler để đối chiếu kết quả.
- **Giai đoạn 4:** Chuyển đổi toàn bộ ngôn ngữ MT22 sang MIPS Code
 - Tích hợp code từ BTL1 - BTL4 của MT22 gốc vào codebase của học phần mở rộng.
 - Tiến hành thay đổi CodeGeneration.py, Emitter.py và MachineCode.py và các module quản lý thanh ghi để hỗ trợ việc sinh mã cho toàn bộ các loại câu lệnh của ngôn ngữ.
 - Tạo các testcase và tiến hành chạy mã sinh ra trên MIPS Assembler để đối chiếu kết quả.

3.2 Chiến lược kiểm thử việc quản lý thanh ghi

Ta quy ước chỉ có 8 thanh ghi được sử dụng để cấp phát (từ `s0` đến `s7`), hàm `putInt` sẽ được chuyển sang `syscall` để in số nguyên.

3.2.1 Phép cộng cơ bản

```
7      def test_add_simple(self):
8          """simple addition"""
9          print("Test 500 - Simple Addition")
10         input = """void main() {
11             a,b,c: int = 1,2,3;
12             b = c; // [c]
13             a = a + b; // [b]
14             putInt(a); // [a,b]
15         }"""
16         expect = "4"
17         self.assertTrue(TestCodeGen.test(input, expect, 500))
```

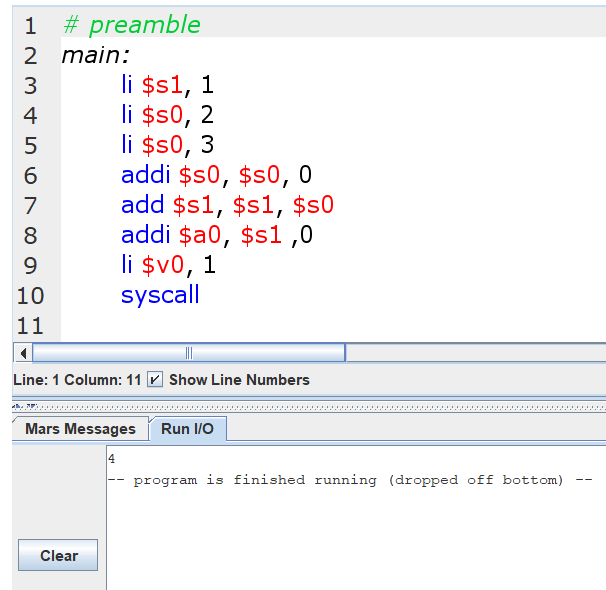
Hình 10: Test 500

```

1 # preamble
2 main:
3     li $s1, 1
4     li $s0, 2
5     li $s0, 3
6     addi $s0, $s0, 0
7     add $s1, $s1, $s0
8     addi $a0, $s1, 0
9     li $v0, 1
10    syscall

```

Listing 1: Output 500



Hình 11: Check 500

3.2.2 Phép cộng nhiều biến, và có biến không cần sử dụng

```

18     def test_add_unused(self):
19         """c is never used again → no need to allocate"""
20         print("Test 501 - Unused Variable Addition: c")
21         input = """void main() {
22             a,b,c: int = 1,2,3;
23             d: int = 4; // [a,b]
24             d = a + b; // [a,b]
25             putInt(d); // [a,b,d]
26         }"""
27         expect = "3"
28         self.assertTrue(TestCodeGen.test(input, expect, 501))

```

Hình 12: Test 501

```

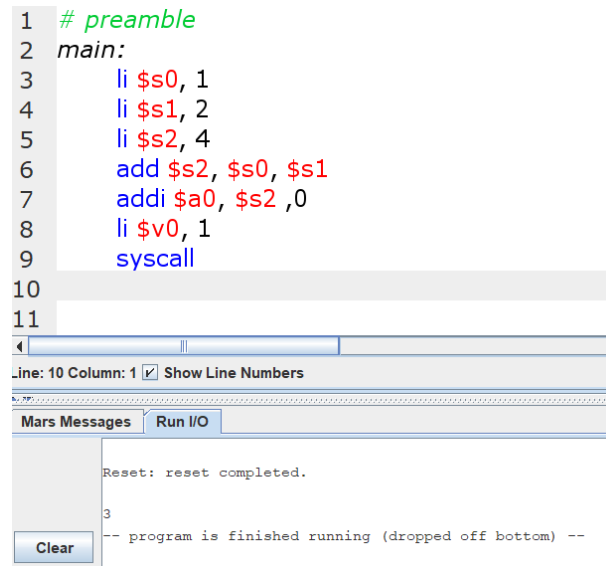
1 # preamble
2 main:
3     li $s0, 1
4     li $s1, 2
5     li $s2, 4

```



```
6 add $s2, $s0, $s1
7 addi $a0, $s2, 0
8 li $v0, 1
9 syscall
```

Listing 2: Output 501



Hình 13: Check 501

3.2.3 Phép cộng nhiều biến, vượt số lượng thanh ghi

```
29 def test_add_many_variables(self):
30     """test num_vars(12) > num_regs(8)"""
31     print("Test 502 - Mega Variable Addition")
32     input = """void main() {
33         a,b,c: int = 1,2,3; [] [a] [a,b]
34         d: int = a + b; //3 [a,b,c]
35         e: int = a + c; //4 [c,a,d]
36         f: int = d + a; //5 [c,a,d]
37         g: int = d; //3 [c,a,d,f]
38         e = c; //3 [c,a,d,f]
39         h: int = a + d; //4 [c,a,d,f]
40         i: int = f + 1; //6 [c,h,f]
41         j,l: int = h,i; //4,6 [c,h,i] [c,i]
42         k: int = c; //3 [c]
43         h = h + k; //7 [k]
44         putInt(h); [h,k]
45     }"""
46     expect = "7"
47     self.assertTrue(TestCodeGen.test(input, expect, 502))
```

Hình 14: Test 502

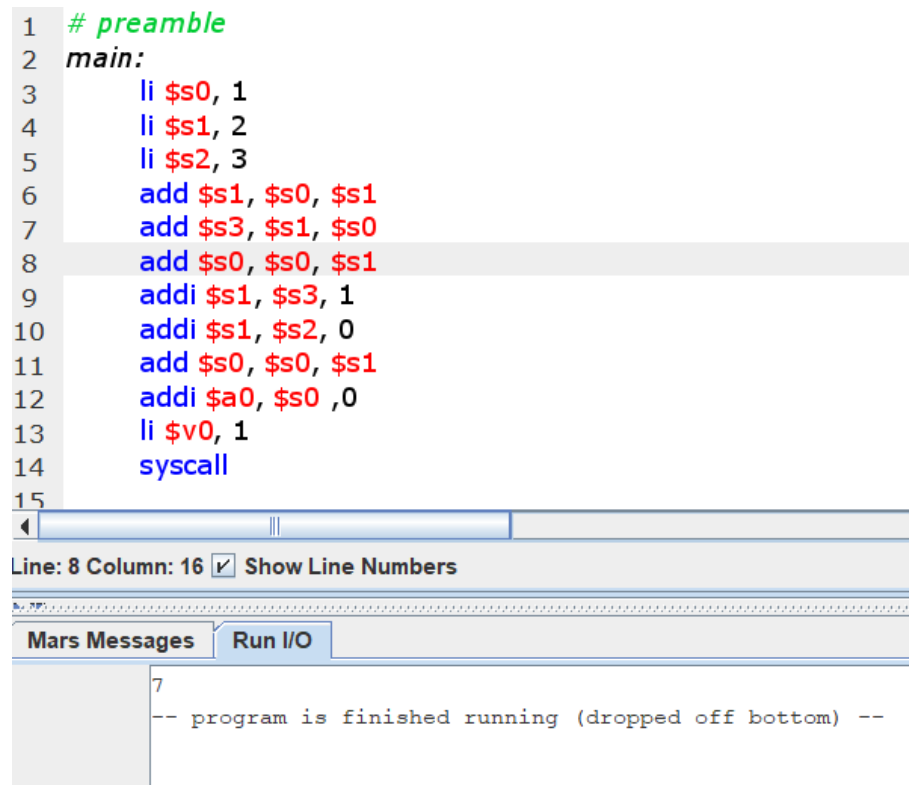
```
1 # preamble
```

```

2 main:
3   li $s0, 1
4   li $s1, 2
5   li $s2, 3
6   add $s1, $s0, $s1
7   add $s3, $s1, $s0
8   add $s0, $s0, $s1
9   addi $s1, $s3, 1
10  addi $s1, $s2, 0
11  add $s0, $s0, $s1
12  addi $a0, $s0, 0
13  li $v0, 1
14  syscall

```

Listing 3: Output 502



```

1  # preamble
2  main:
3      li $s0, 1
4      li $s1, 2
5      li $s2, 3
6      add $s1, $s0, $s1
7      add $s3, $s1, $s0
8      add $s0, $s0, $s1
9      addi $s1, $s3, 1
10     addi $s1, $s2, 0
11     add $s0, $s0, $s1
12     addi $a0, $s0, 0
13     li $v0, 1
14     syscall
15

```

Line: 8 Column: 16 ☒ Show Line Numbers

Mars Messages Run I/O

```

7
-- program is finished running (dropped off bottom) --

```

Hình 15: Check 502

4 Tiến độ hiện tại và hướng phát triển

4.1 Tiến độ hoàn thành

Hạn chế của giải thuật đề xuất là chỉ thực hiện được trong tầm vực cục bộ (cụ thể là hàm main). Do thời gian em hoàn thành bài làm khá ngắn (vì vừa phải hoàn thành BTL4 và các bài thi trên trường), em chỉ hoàn thiện được một số chức năng cơ bản sau đây:

- Các phép tính Cộng/ Trừ/ Nhân/ Chia trên kiểu số nguyên, chưa có tính kết hợp.
- Các phép tính Logic (And/ Or) trên kiểu luận lý, chưa có tính kết hợp.
- Toán tử hai ngôi và một ngôi cho các kiểu dữ liệu phù hợp (không hiện thực đối với kiểu String/ Float/ Array vì bắt buộc phải lưu trữ ở Memory).
- Cấu trúc rẽ nhánh (if/ if-else/ nested if-else).
- Cấu trúc lặp (while/ do-while/ for/ nested loops/ mixed-type loops).
- Các lệnh syscall in giá trị.

4.2 Hướng phát triển

Qua dự án mở rộng lần này (cả 3 Phase), em đã học được rất nhiều kiến thức bổ ích về cách thức, nguyên lý hoạt động của các loại ngôn ngữ lập trình khác nhau, những ưu nhược điểm của từng loại,... Ngoài ra, kỹ năng lập trình và debug đã được cải thiện đáng kể sau khi trải qua môn học này.

Do hạn chế về thời gian và kinh nghiệm, phiên bản hiện tại của dự án vẫn chưa đáp ứng một cách toàn vẹn yêu cầu của Phase 3. Tuy nhiên em đề xuất một số hướng phát triển sau đây (nếu có đủ thời gian để thực hiện):

- **Triển khai tính kết hợp cho các phép tính:** Do ngôn ngữ MIPS thuộc dạng TAC (Three Address Code), các thao tác phải được chia nhỏ thành biểu thức 2 ngôi và 1 ngôi. Các phép tính có tính kết hợp đòi hỏi phải thiết kế việc lưu trữ kết quả trung gian vào thanh ghi tạm sao cho kết quả đầu ra là chính xác.
- **Triển khai việc lưu trữ vào vùng nhớ để hỗ trợ cho lưu trữ thanh ghi:** Trên tinh thần của dự án là tập trung quản lý việc lưu trữ các giá trị thực trên thanh ghi, em tạm gác lại việc lưu trữ giá trị trên vùng *.data*. Xét đến giải thuật đề xuất, khi các thanh ghi cung cấp không thể cung cấp, ta sẽ có thể chuyển giá trị 1 thanh ghi nào đó sang bộ nhớ (Node Spilling), từ đó chương trình sẽ đảm bảo luôn hoạt động. Ngoài ra, nếu phát triển thành công, MT22 sẽ có thể lưu trữ dạng số chấm động, chuỗi ký tự và mảng (mà giá trị thực vốn cần phải khai báo trước ở vùng bộ nhớ).
- **Triển khai việc gọi hàm và thao tác hàm:** Do thiết kế của giải thuật, phiên bản hiện tại chỉ có thể đáp ứng đối với tầm vực cục bộ. Nếu phát triển thêm việc gọi hàm, ta cần phải làm mới thanh ghi để thực hiện các thao tác trong tầm vực cục bộ của hàm đó, sau đó phải quay lại giá trị cũ (các giá trị đã có ở hàm main) khi hàm thực thi xong. Đi kèm đó là vấn đề truyền tham số theo dạng *pass by value-result* được theo thiết kế của MT22, tuy nhiên vấn đề này có thể được giải quyết dễ dàng nếu như thao tác gọi-trả về được hiện thực thành công.



Tài liệu

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.