

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



NGUYÊN LÝ NGÔN NGỮ LẬP TRÌNH - Mở rộng (CO300C)

Báo cáo Bài tập lớn mở rộng

Giai đoạn 2 CẤP PHÁT THANH GHI VÀ SINH MÃ MIPS CHO PHÉP CỘNG SỐ NGUYÊN DƯƠNG

GVHD: Nguyễn Hứa Phùng

Sinh viên: Nguyễn Thái Tân 2112256

Thành phố Hồ Chí Minh, 04/2024



Mục lục

1	Giới thiệu	2
2	Cơ sở lý thuyết	3
2.1	Register-based	3
2.1.1	Register	3
2.1.2	Register-based	4
2.2	Microprocessor without Interlocked Pipelined Stages (MIPS)	4
2.3	So sánh stack-based và register-based	6
3	Hiện thực	8
3.1	Cấu trúc chương trình	8
3.2	Cấp phát thanh ghi	9
3.3	Hiện thực (1)	9
3.4	Hiện thực (2)	14
3.5	Hiện thực các hàm visit cần thiết	18
4	Kiểm thử	20
5	Kết luận	24

1 Giới thiệu

Trong ngôn ngữ lập trình, quá trình biên dịch là một quá trình vô cùng quan trọng. Các giai đoạn trong quá trình biên dịch có thể được chia thành hai nhóm chính: front-end và back-end. Đầu ra của front-end là mã trung gian, mã này có thể là AST. Đầu ra của back-end là mã máy. Ở bài tập trước đó, Java byte code (đoạn mã sử dụng để chạy trên Java Virtual Machine (JVM)) được sử dụng làm đầu ra của giai đoạn back-end. JVM là một stack-based và chấp nhận mô hình hướng đối tượng. Việc tạo mã stack-based sẽ dễ dàng hơn, tuy nhiên sẽ cần sự đánh đổi về hiệu suất. Vì vậy trong bài tập này, chúng ta sẽ khắc phục nhược điểm của stack-based bằng cách sử dụng register-based để làm đầu ra của giai đoạn back-end. Có nhiều kiến trúc hỗ trợ cho register-based, trong bài tập này chúng ta sẽ chọn hiện thực Microprocessor without Interlocked Pipelined Stages, hay tên thường gọi là MIPS.

Register-based là kiến trúc mà CPU sẽ sử dụng các thanh ghi để thực hiện các thao tác trên đó. Vấn đề của register-based là chỉ tồn tại một số lượng hữu hạn các thanh ghi, nên khi thực hiện các lệnh, các thao tác, chúng ta phải cấp phát thanh ghi một cách hợp lý. Trong bài tập này, chúng ta cũng sẽ hiện thực quá trình cấp phát thanh ghi. Vì *initial code* được hiện thực cho một ngôn ngữ đơn giản không sử dụng biến, nên nếu sử dụng *initial code* này sẽ không thể hiện thực quá trình cấp phát thanh ghi. Do đó trong bài tập này chúng ta sử dụng ngôn *ZCode* đã được mô tả trong các bài tập trước đó.

Trong phần còn lại của bài viết này, chúng ta sẽ tìm hiểu các phần chính cụ thể như sau:

1. Cơ sở lý thuyết: Chương này sẽ trình bày chi tiết về các cơ sở lý thuyết cần thiết trước khi hiện thực. Qua việc tìm hiểu từng bước về Register-based và MIPS, người đọc sẽ nắm rõ những kiến thức cơ bản về kiến trúc sử dụng thanh ghi, đồng thời rút ra so sánh về sự khác nhau giữa register-based và stack-based.
2. Hiện thực: Chương này sẽ trình bày chi tiết về cách hiện thực việc quản lý cấp phát thanh ghi và sinh mã MIPS cho phép cộng số nguyên dương trên ngôn ngữ ZCode. Qua đó, người đọc sẽ hiểu rõ về ý tưởng từng bước hiện thực và mã nguồn chi tiết.
3. Kiểm thử: Chương này sẽ trình bày chi tiết về quá trình kiểm thử của việc hiện thực trong chương trước đó. Qua đó người đọc sẽ hiểu rõ chiến lược thiết kế cho testcase để kiểm thử chương trình.
4. Kết luận: Từ việc hiện thực và kiểm thử, nêu được kết luận cho bài tập, từ đó rút ra kinh nghiệm và hướng phát triển tiếp theo cho chương trình.

2 Cơ sở lý thuyết

Chương này sẽ trình bày chi tiết về các cơ sở lý thuyết cần thiết trước khi hiện thực. Chúng ta sẽ lần lượt tìm hiểu về Register-based và MIPS. Qua đó, người đọc có thể rút ra được sự khác biệt giữa register-based và stack-based.

2.1 Register-based

2.1.1 Register

Register (thanh ghi) là một loại bộ nhớ máy tính được tích hợp trực tiếp vào processor (bộ xử lý) hoặc CPU, dùng để lưu trữ và thao tác dữ liệu trong quá trình thực hiện các lệnh. Một register có thể chứa một lệnh, một địa chỉ hoặc bất kỳ loại dữ liệu nào (chẳng hạn như một chuỗi bit hoặc các ký tự riêng lẻ).

Số lượng và kích thước của các thanh ghi trong CPU được xác định bởi thiết kế bộ xử lý, tác động đáng kể đến hiệu suất và khả năng của thanh ghi. Các kích thước thanh ghi phổ biến là 8 bit, 16 bit, 32 bit và 64 bit. Các máy tính hiện đại ngày nay thường có các thanh ghi 32 bit hoặc 64 bit.

Trong một số bộ xử lý hoặc kiến trúc chuyên dụng, ta cũng có thể tìm thấy kích thước thanh ghi lớn hơn, chẳng hạn như 128 bit, 256 bit, Những thanh ghi lớn hơn này thường được sử dụng cho các mục đích cụ thể như xử lý vectơ hoặc các hoạt động mã hóa, trong đó có liên quan đến tính song song và các tập dữ liệu lớn.

Các loại thanh ghi:

- Program Counter (PC): theo dõi địa chỉ bộ nhớ của lệnh tiếp theo được nạp và thực thi.
- Instruction Register (IR): chứa lệnh hiện đang được thực thi.
- Accumulator (ACC): thanh ghi đa năng được sử dụng cho các phép tính số học và logic. Nó lưu trữ kết quả trung gian trong quá trình tính toán.
- General-Purpose Registers (R0, R1, R2...): lưu trữ dữ liệu trong quá trình tính toán và thao tác dữ liệu.
- Address Registers (AR): lưu trữ địa chỉ bộ nhớ để truy cập dữ liệu hoặc để truyền dữ liệu giữa các vị trí bộ nhớ khác nhau.
- Stack Pointer (SP): trỏ đến đầu stack (là vùng bộ nhớ được sử dụng để lưu trữ tạm thời trong các lệnh gọi hàm).
- Data Registers (DR): lưu trữ dữ liệu được lấy từ bộ nhớ hoặc thu được từ các hoạt động input/output.
- Status Register/Flags Register (SR): chứa các bit riêng lẻ cho biết kết quả của các hoạt động, chẳng hạn như carry, overflow, zero result, và các hoạt động khác.
- Control Registers (CR): quản lý các cài đặt và thông số điều khiển khác nhau liên quan đến hoạt động của CPU, chẳng hạn như xử lý ngắt, quản lý bộ nhớ và cấu hình hệ thống.

2.1.2 Register-based

Register-based là kiến trúc mà CPU sẽ sử dụng các thanh ghi để thực hiện các thao tác trên đó.

Với một chương trình theo register-based, vấn đề của chúng ta là không có vô hạn thanh ghi để lưu trữ các dữ liệu. Vì lý do đó, chúng ta cần cấp phát thanh ghi một cách hợp lý để với một số lượng thanh ghi hữu hạn, chương trình vẫn chạy được bình thường.

Theo [1], ta có thuật toán để cấp phát thanh ghi như sau :

1. Từ *live ranges* của biến, xây dựng một *interference graph* với nguyên tắc hai node (biến) sẽ được nối bởi một cạnh nếu *live ranges* của chúng lồng nhau.
Live ranges của biến là thời gian từ lúc biến được khởi tạo cho đến lần cuối cùng nó được sử dụng.
2. Tô màu *interference graph* trên bằng k màu, sao cho không có hai node nào được nối với nhau có cùng màu.
3. Nếu không thể tô với k màu, có nhiều cách xử lý khác nhau, nhưng trong bài tập này, chúng ta chọn cách *spill*.
Chọn node có *live ranges* thấp nhất để *spill*, tức là đưa biến đó vào vùng *.data*. Sau đó loại node đó ra khỏi graph, thực hiện tô màu lại cho đến khi thành công.
4. Gán tương ứng màu với thanh ghi. Như vậy, mỗi biến sẽ được lưu trữ thành công.

2.2 Microprocessor without Interlocked Pipelined Stages (MIPS)

MIPS là mã register-based, được tạo ra vào năm 1981 bởi J. L. Hennessy tại Stanford University. Trong MIPS, CPU chỉ có thể thực hiện các thao tác trên các thanh ghi và các giá trị tức thời đặc biệt.

MIPS là kiến trúc máy tính Reduced Instruction Set Computer (RISC), tức là nó sẽ không gán các lệnh riêng lẻ cho các nhiệm vụ phức tạp, chuyên sâu về mặt logic (khác với kiến trúc máy tính Complex instruction set computer (CISC)). Vì vậy,

- MIPS dùng thanh ghi 32 bit nên tất cả các lệnh MIPS đều dài 32 bit.
- Điều này làm cho phần cứng để truy cập và giải mã các lệnh trở nên đơn giản.
- Điều này cũng có nghĩa là chỉ có một số lượng lệnh hữu hạn.

MIPS có tổng cộng 32 thanh ghi (register) để lưu giá trị, được đánh số từ 0 đến 31. Chúng có thể được truy cập thông qua cú pháp \$ + số thứ tự thanh ghi (\$0, \$1, \$10,...). Tuy nhiên, MIPS có quy ước mục đích sử dụng của mỗi thanh ghi, vì thế, người ta thường truy cập thanh ghi thông qua tên của chúng:

Tên	Thanh ghi	Ý nghĩa
\$zero	0	Thanh ghi này luôn chứa giá trị 0
\$at	1	Dùng cho các mục đích khác, nên hạn chế dùng thanh ghi này
\$v0, \$v1	2, 3	Lưu giá trị trả về của hàm
\$a0 - \$a3	4 - 7	Lưu tham số truyền vào của hàm
\$t0 - \$t7	8 - 15	Lưu biến tạm
\$s0 - \$s7	16 - 23	Lưu biến
\$t8, \$t9	24, 25	Như các \$t ở trên
\$k0, \$k1	26, 27	Được dùng cho nhân HĐH sử dụng
\$gp	28	Pointer to global area
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address, sử dụng cho việc gọi hàm

Bảng 1: Quy ước về tên thanh ghi

Một số lệnh cơ bản của MIPS:

- Lệnh load/store:
 - lw Rd, RAM_src: Chép 1 word (4 byte) tại vị trí trong bộ nhớ RAM vào thanh ghi
 - lb Rd, RAM_src: Chép 1 byte tại vị trí trong bộ nhớ RAM vào byte thấp của thanh ghi
 - sw Rs, RAM_dest: Lưu 1 word trong thanh ghi vào vị trí trong bộ nhớ RAM
 - sb Rs, RAM_dest: Lưu 1 byte thấp trong thanh ghi vào vị trí trong bộ nhớ RAM
 - li Rd, value: Khởi tạo thanh ghi với giá trị
 - la Rd, label: Khởi tạo thanh ghi với địa chỉ của nhãn
- Lệnh số học:
 - add Rd, Rs, Rt: $Rd = Rs + Rt$ (kết quả có dấu)
 - addi Rd, Rs, imm: $Rd = Rs + imm$
 - addu Rd, Rs, Rt: $Rd = Rs + Rt$ (kết quả không dấu)
 - sub Rd, Rs, Rt: $Rd = Rs - Rt$
 - subu Rd, Rs, Rt: $Rd = Rs - Rt$ (kết quả không dấu)
 - mult Rs, Rt: $(Hi, Lo) = Rs * Rt$
 - div Rs, Rt: $Lo = Rs / Rt$ (thương), $Hi = Rs \% Rt$ (số dư)
 - mfhi Rd: $Rd = Hi$
 - mflo Rd: $Rd = Lo$
 - move Rd, Rs: $Rd = Rs$
- Lệnh nhảy:
 - j label: Nhảy không điều kiện đến nhãn 'label'
 - jal label: Lưu địa chỉ trở về vào \$ra và nhảy đến nhãn 'label' (dùng khi gọi hàm)
 - jr Rs: Nhảy đến địa chỉ trong thanh ghi Rs (dùng để trở về từ lời gọi hàm)
 - bgez Rs, label: Nhảy đến nhãn 'label' nếu $Rs \geq 0$
 - bgtz Rs, label: Nhảy đến nhãn 'label' nếu $Rs > 0$
 - blez Rs, label: Nhảy đến nhãn 'label' nếu $Rs \leq 0$
 - bltz Rs, label: Nhảy đến nhãn 'label' nếu $Rs < 0$
 - beq Rs, Rt, label: Nhảy đến nhãn 'label' nếu $Rs = Rt$
 - bne Rs, Rt, label: Nhảy đến nhãn 'label' nếu $Rs \neq Rt$

- **Lệnh syscall:** Lệnh syscall làm treo sự thực thi của chương trình và chuyển quyền điều khiển cho HĐH (được giả lập bởi MARS). Sau đó, HĐH sẽ xem giá trị thanh ghi \$v0 để xác định xem chương trình muốn nó làm việc gì.

Dịch vụ	Giá trị trong \$v0	Đối số	Kết quả
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (trong \$v0)
read_float	6		float (trong \$f0)
read_double	7		double (trong \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (trong \$v0)
exit	10		
print_character	11	\$a0 = char	
read_character	12		char (trong \$v0)

Bảng 2: Bảng syscall

2.3 So sánh stack-based và register-based

Với việc đã tìm hiểu qua stack-based ở bài tập trước và register-based ở bài tập này, ta rút ra một vài nhận xét về chúng:

Đặc điểm	Stack-based	Register-based
Tổ chức bộ nhớ	Các toán hạng được đẩy lên stack trước khi thực hiện một thao tác. Các hoạt động và kết quả của chúng thường liên quan đến top của stack	Các toán hạng được lưu trữ trong các thanh ghi trong CPU. Các hoạt động và kết quả của chúng được xử lý và lưu trực tiếp trên thanh ghi.
Tốc độ truy cập	Chậm hơn	Nhanh hơn vì xử lý trực tiếp trên CPU
Cách dùng bộ nhớ	Có xu hướng sử dụng tổng bộ nhớ ít hơn vì chúng dựa vào một stack duy nhất. Tuy nhiên, stack có thể bị tràn (overflow), do đó có khả năng xảy ra hiện tượng phân mảnh.	Có thể sử dụng nhiều bộ nhớ hơn vì chúng cần phân bổ các thanh ghi cho toán hạng. Tuy nhiên, chúng không gặp phải vấn đề phân mảnh.
Tốc độ truy cập	Chậm hơn	Nhanh hơn vì xử lý trực tiếp trên CPU
Hiệu suất	Yếu hơn, do hoạt động thao tác trên stack thường xuyên	Mạnh hơn, đặc biệt là với các hoạt động liên quan đến việc truy cập thường xuyên vào toán hạng.
Hiện thực	Dễ hơn	Phức tạp hơn

Bảng 3: So sánh stack-based và register-based

3 Hiện thực

Sau khi đã tìm hiểu đầy đủ về cơ sở lý thuyết, chương này sẽ trình bày về quá trình hiện thực sinh mã MIPS cho phép cộng số nguyên dương trên ngôn ngữ ZCode. Đồng thời cũng trình bày chi tiết mã nguồn hiện thực.

3.1 Cấu trúc chương trình

Vì *initial code* được hiện thực để sinh mã và chạy JVM, nên chúng ta cần thay đổi cấu trúc các thư mục và thay đổi các file cần thiết cho việc sinh mã và chạy MIPS. Ở bước này, ta cũng thay đổi việc phân tích từ vựng, ngữ pháp từ ngôn ngữ của *initial code* trong file *.g4* và *AST.py* thành ngôn ngữ ZCode.

Trước hết, ta cần thay đổi việc thực thi chương trình bằng *jasmin* thành thực thi chương trình bằng *Mars_5*. Ta nhận thấy, *class TestCodeGen()* ở trong *src/test/TestUtils.py*, có phương thức *def check(soldir, asttree, num)* dùng để kiểm tra CodeGeneration. Để ý dòng

```
1 subprocess.call("java -jar "+ JASMIN_JAR + " " + path + "/ZCodeClass
  .j", shell=True, stderr=subprocess.STDOUT)
2 cmd = "java -cp ./lib" + os.pathsep + ". ZCodeClass"
3 subprocess.run(cmd, shell=True, stdout = f, timeout=10)
```

với *JASMIN_JAR* là đường dẫn đến file *./external/jasmin.jar*. Đây là lệnh dùng để gọi một process mới, thực thi file *.jar* để chạy được chương trình JVM. Do đó, ta cần thay đổi dòng lệnh này để thực thi file *.jar* chạy được chương trình MIPS.

Theo [2], để chạy trên command line thì ta cần lệnh

```
1 java -jar mars.jar [options] program.asm [more files...] [ pa arg1 [
  more args...]]
```

trong đó *program.asm* là chương trình chính, *more files* là các chương trình phụ, *more args* là các đối số đi kèm (được trình bày rõ trong [2]). Để thông báo bản quyền sẽ không được hiển thị, ta sẽ dùng thêm đối số *nc*.

Vì ngôn ngữ của chúng ta được đặc tả là có phương thức *writeNumber()* để thực hiện việc Outstream, do đó ta cần thêm file *io.asm* để lưu trữ sẵn các lệnh IO cần thiết. Hơn nữa, đối với MIPS, ta cũng cần thêm một phương thức *exit()* để kết thúc chương trình.

```
1 .globl writeNumber
2 .globl exit
3 .text
4
5 writeNumber:
6     li $v0, 1
7     syscall
8     jr $ra
9
10 exit:
11     li $v0, 10
12     syscall
```

Kết hợp các ý trên, ta có dòng lệnh cuối cùng như sau

```
1 cmd = "java -jar "+ MARS_JAR + " " + path + "/ZCodeClass.asm" + " "  
  + IO_ASM + " nc"  
2 subprocess.run(cmd, shell=True, stdout = f, timeout=10)
```

trong đó *MARS_JAR* = *./external/Mars4_5.jar* là đường dẫn đến file *Mars4_5.jar* và *IO_ASM* = *./lib/io.asm* là đường dẫn đến file *io.asm*.

3.2 Cấp phát thanh ghi

Như đã trình bày, với một chương trình theo register-based, vấn đề của chúng ta là không có vô hạn thanh ghi để lưu trữ các dữ liệu. Vì lý do đó, chúng ta cần cấp phát thanh ghi một cách hợp lý để với một số lượng thanh ghi hữu hạn, chương trình vẫn chạy được bình thường.

Trước khi hiện thực, chúng ta có một số nhận xét sau:

- Khi cấp phát thanh ghi, cần cố gắng hạn chế số lượng thanh ghi cấp phát nhất có thể. (1)
Ví dụ, ta thực hiện phép cộng $a + 1 + 2$. Nếu suy nghĩ theo hướng thông thường, ta load từng giá trị vào các thanh ghi, load a vào $\$t0$, 1 vào $\$t1$ và 2 vào $\$t2$, sau đó tính $\$t3 = \$t0 + \$t1$, $\$t4 = \$t3 + \$t2$. Kết quả nằm ở $\$t4$.
Tuy nhiên ta cần hạn chế thanh ghi nhất có thể, vì thế ta có thể thực hiện load a vào $\$t0$, 1 vào $\$t1$, tính $\$t1 = \$t1 + \$t0$, sau đó load 2 vào $\$t2$, tính $\$t1 = \$t1 + \$t2$. Kết quả nằm ở $\$t1$.
- Nếu số lượng biến quá nhiều, không thể lưu trữ sẵn vào thanh ghi, ta bắt buộc phải lưu trữ ở phần *.data*, khi dùng đến thì mới load vào thanh ghi cần thiết. (2)

Như vậy, ta có hướng hiện thực như sau: Ta sẽ duyệt qua toàn bộ chương trình, cố gắng cấp phát thanh ghi theo (1). Nếu không thể cấp phát được nữa, ta mới cấp phát theo (2).

3.3 Hiện thực (1)

Trong *class CodeGenVisitor*, ta thêm một số thuộc tính sau để hỗ trợ việc cấp phát thanh ghi.

```
1 class CodeGenVisitor(BaseVisitor, Utils):  
2     def __init__(self, astTree, env, dir_):  
3         # other attributes  
4         self.allVars = {}  
5         self.register = {  
6             't0': [], 't1': [], 't2': [], 't3': [], 't4': [], 't5':  
              [], 't6': [], 't7': []  
7         }  
8         self.register_dest = []  
9         self.register_for_num = 0
```

Trong đó,

- *allVars*: một dictionary chứa thông tin của tất cả các biến có trong chương trình.
- *register*: một dictionary chứa thông tin của tất cả các thanh ghi.

- *register_dest*: một list chứa thông tin các thanh ghi được dùng làm thanh ghi đích (ví dụ $\$t4 = \$t3 + \$t2$ thì $\$t4$ là thanh ghi đích); vậy nếu thanh ghi nằm trong list này thì cũng có thông tin trong *register*.
- *register_for_num*: Sẽ có một số thanh ghi chỉ dùng để lưu số hoặc giá trị của phép tính (ví dụ trong phép cộng $a + 2$ thì cần một thanh ghi để lưu trữ số 2). Do đó thuộc tính *register_for_num* lưu trữ số lượng thanh ghi được dùng cho việc này.

Nhận xét: Các phương thức cấp phát thanh ghi sẽ chỉ được gọi nếu node đang được duyệt trên cây là node *VarDecl* hoặc node *NumberLiteral*.

Nếu là *VarDecl*

1. Kiểm tra xem biến được khai báo global hay không bằng *c.frame*.
2. Tìm kiếm một thanh ghi đang trống bằng phương thức *getFreeRegister*.
Phương thức *getFreeRegister*: duyệt qua tất cả các phần tử trong *self.register*, nếu có phần tử nào còn thỏa mãn thì trả về thanh ghi tương ứng.
3. Cập nhật *self.allVars* thông tin về biến vừa khai báo. Nếu là global thì lưu thông tin scope là 'GLOBAL', nếu không global thì lưu thông tin scope là *c.frame.name.name*.

```
1 def getFreeRegister(self):
2     for key, value in self.register.items():
3         if len(value) == 0:
4             return key
5
6 def visitVarDecl(self, ast, c):
7     varName = ast.name.name
8     varInit = ast.varInit
9     varType = ast.varType
10
11     if c.frame is None:
12         # Global scope
13         register = self.getFreeRegister()
14         self.allVars.update({ast.name.name: [register, 'GLOBAL',
15             varInit.value]})
16         self.register[register].append(ast.name.name)
17
18         val = CName(self.className)
19         self.emit.printout(self.emit.emitATTRIBUTE(register, varInit.
20             value, False))
21     else:
22         register = self.getFreeRegister()
23         self.allVars.update({ast.name.name: [register, c.frame.name.
24             name, varInit.value]})
25         self.register[register].append(ast.name.name)
26
27         val = CName(c.frame.name.name)
28         self.emit.printout(self.emit.emitATTRIBUTE(register, varInit.
29             value, True))
```

```
26     return Symbol(varName, varType, val)
27
```

Nếu là *NumberLiteral*

1. Tìm kiếm một thanh ghi đang trống cho việc lưu trữ số bằng phương thức *getFreeRegisterForNumber*.

Phương thức *getFreeRegisterForNumber*:

- (a) Duyệt qua tất cả các phần tử trong *self.register*, nếu đã có thanh ghi nào lưu trữ giá trị tương tự thì trả về thanh ghi đó.
 - (b) Trước hết, giải phóng các thanh ghi còn thừa. Sau đó duyệt lại qua tất cả các phần tử trong *self.register*, nếu thanh ghi nào còn trống thì trả về thanh ghi đó.
2. Như đã nói, sẽ có một số thanh ghi chỉ dùng để lưu số hoặc giá trị của phép tính. Đây chính là một thanh ghi như vậy. Vì thế ta vừa cập nhật *self.register_for_num*, vừa cập nhật *self.allVars* (vì không có tên biến nên thông tin về tên biến trong *self.allVars* ta có thể theo quy luật chuỗi các ký tự '*').

```
1 def getFreeRegisterForNumber(self, val):
2     for key, value in self.register.items():
3         if key not in self.register_dest and val in value:
4             return key
5
6     if self.register_dest != []:
7         for key, value in self.register.items():
8             if key not in self.register_dest:
9                 self.register[key] = [item for item in value if not
10                     isinstance(item, (int, float))]
11
12     for key, value in self.register.items():
13         if len(value) == 0:
14             return key
15
16 def visitNumberLiteral(self, ast, o):
17     register = self.getFreeRegisterForNumber(ast.value)
18     self.register[register].append(ast.value)
19     t = any([register, o.frame.name.name, self.count, self.count, 0]
20             == value for _, value in self.allVars.items())
21     if not t:
22         self.register_for_num += 1
23         string = '*' * self.register_for_num
24         self.allVars[string] = [register, o.frame.name.name, 0]
25     return self.emit.emitPUSHICONST(ast.value, register), NumberType
26     (), register
```

Nhận xét: Các phương thức dùng để load thanh ghi sẽ chỉ được gọi nếu node đang được duyệt trên cây là node *Id*.

1. Dùng phương thức *getRegOfId* để kiểm tra xem biến này đã được khai báo hay chưa.

Phương thức *getRegOfId*: duyệt qua tất cả phần tử trong *self.allVars*; nếu có phần tử trùng khớp thì trả về thanh ghi tương ứng; nếu không có phần tử trùng khớp thì trả về *None*.

```
1 def getRegOfId(self, id, scope: str):
2     # in scope first
3     for key, value in self.allVars.items():
4         if id == key and value[1] == scope:
5             return value[0]
6     # in global later
7     for key, value in self.allVars.items():
8         if id == key:
9             return value[0]
10    return
11
12 def visitId(self, ast, c):
13     sym = next(filter(lambda x: x.name == ast.name, c.sym))
14     name = sym.name
15     register = self.getRegOfId(name, c.frame.name.name)
16     if register:
17         return self.emit.emitGETATTRIBUTE("", c.frame), sym.mtype,
18             register
19     return self.emit.emitGETATTRIBUTE(register, c.frame), sym.mtype
```

Cuối cùng, ta hiện thực việc cấp phát thanh ghi cho phép cộng như sau:

1. Như đã nói, ta cần cố gắng hạn chế số lượng thanh ghi cấp phát nhất có thể, vì vậy theo ý tưởng ở (1), trước khi thực hiện phép cộng, ta giải phóng các thanh ghi cho các số bằng phương thức *releaseRegisterForNumber(self, is_release_reg_dest: bool)*

Phương thức *releaseRegisterForNumber(self, is_release_reg_dest: bool)*:

- Nếu *is_release_reg_dest* là *True*, ta duyệt qua tất cả các phần tử trong *self.register*, giải phóng tất cả các thanh ghi lưu trữ số.
- Nếu *is_release_reg_dest* là *False*, ta duyệt qua tất cả các phần tử trong *self.register*, giải phóng tất cả các thanh ghi lưu trữ số, trừ các thanh ghi đích trong *self.register_dest*.

Với trường hợp phép tính nhị phân nói chung và phép cộng nói riêng, ta gọi phương thức *releaseRegisterForNumber(is_release_reg_dest = False)*.

Ví dụ ở (1), với $\$t1 = \$t1 + \$t0$ thì $\$t1$ là thanh ghi đích, ta không giải phóng $\$t1$, để khi duyệt tiếp đến $\$t1 = \$t1 + \$t2$, giá trị của $\$t1$ vẫn ở đó, dùng để cộng dồn vào kết quả cuối cùng.

2. Duyệt cây con bên trái trước, nếu cây con bên trái này:

- là *NumberLiteral*: chọn thanh ghi đích cho phép cộng này là thanh ghi được trả về từ *visitNumberLiteral*. Đồng thời cập nhật *self.register_dest*.
- là *Id*: *visitId* sẽ tự động được gọi và trả về thanh ghi tương ứng của biến. Ghi nhớ lại biến này để sinh mã MIPS ở bước 4.

- không phải là *NumberLiteral* hay *Id*: sẽ gọi đệ quy các phương thức *visit*, vì thế ta không cần quan tâm đến trường hợp này.
3. Duyệt cây con bên phải sau, nếu cây con bên phải này:
- là *NumberLiteral*: Nếu chưa chọn thanh ghi đích thì chọn thanh ghi đích cho phép cộng này là thanh ghi được trả về từ *visitNumberLiteral*. Đồng thời cập nhật *self.register_dest*.
 - là *Id*: *visitId* sẽ tự động được gọi và trả về thanh ghi tương ứng của biến. Ghi nhớ lại biến này để sinh mã MIPS ở bước 4.
 - không phải là *NumberLiteral* hay *Id*: sẽ gọi đệ quy các phương thức *visit*, vì thế ta không cần quan tâm đến trường hợp này.
4. Nếu vẫn chưa chọn được thanh ghi đích, ta buộc phải chọn một thanh ghi khác để làm thanh ghi đích cho phép cộng này bằng phương thức *getFreeRegister* (đã được trình bày ở trên). Đồng thời cập nhật *self.register_dest*.
5. Cuối cùng là sinh mã MIPS bằng phương thức *self.emit.emitADDOP*.

```
1 def visitBinaryOp(self, ast, c):
2     self.releaseRegisterForNumber(False)
3
4     reg1 = reg2 = ""
5     flag = False
6     access = Access(c.frame, c.sym, False, True)
7
8     lCode, lType, lReg = self.visit(ast.left, access)
9     for item in self.register[lReg]:
10         if isinstance(item, (int, float)):
11             if lReg not in self.register_dest:
12                 self.register_dest.append(lReg)
13             flag = True
14
15     rCode, rType, rReg = self.visit(ast.right, access)
16     for item in self.register[rReg]:
17         if isinstance(item, (int, float)):
18             if flag == False:
19                 if rReg not in self.register_dest:
20                     self.register_dest.append(rReg)
21             reg1, reg2 = rReg, lReg
22
23     if flag == True:
24         reg1, reg2 = lReg, rReg
25
26     if reg1 == "":
27         reg1, reg2 = lReg, rReg
28         temp = self.getFreeRegister()
29
30         t = any([temp, c.frame.name.name, self.count, self.count, 0]
31                 == value for _, value in self.allVars.items())
32         if not t:
```

```
32         self.register_for_num += 1
33         string = '*' * self.register_for_num
34         self.allVars[string] = [temp, c.frame.name.name, self.
35             count, self.count, 0]
36
37         if temp not in self.register_dest:
38             self.register_dest.append(temp)
39         self.register[self.register_dest[-1]].append(0)
40
41         if reg1 in self.register_dest and reg2 in self.register_dest:
42             self.register_dest.remove(reg1)
43             self.register_dest.remove(reg2)
44             self.register_dest.append(reg1)
45         if ast.op == '+':
46             return lCode + rCode + self.emit.emitADDOP(reg1, reg2, self.
47                 register_dest[-1], NumberType()), NumberType(), self.
48                 register_dest[-1]
```

3.4 Hiện thực (2)

Như đã trình bày, nếu số lượng biến quá nhiều, không thể lưu trữ sẵn vào thanh ghi, ta bắt buộc phải lưu trữ ở phần *.data*, khi dùng đến thì mới load vào thanh ghi cần thiết.

Với cơ sở lý thuyết đã trình bày ở phần 2.1.2, ta cần thêm một số thuộc tính sau ở *class CodeGenVisitor* để hỗ trợ việc cấp phát thanh ghi.

```
1 class CodeGenVisitor(BaseVisitor, Utils):
2     def __init__(self, astTree, env, dir_):
3         # other attributes
4         self.count = 0
5         self.number_of_spill = 0
6         self.need_to_reallocate = False
7         self.rules = None
```

Trong đó,

- *count*: đánh thời gian cho các lệnh.
Mỗi biến sẽ dùng *count* này để lưu thông tin về lần đầu tiên và lần cuối cùng được gọi, trừ hai giá trị này là tính được *live ranges*.
- *number_of_spill*: lưu trữ số lượng biến cần phải được spill
- *need_to_reallocate*: Như đã trình bày, ta sẽ thực hiện cấp phát theo (1), nếu không thành công mới thực hiện cấp phát theo (2). *need_to_reallocate* là biến bool thể hiện việc cấp phát theo (1) có thành công hay không.
- *rules*: một list lưu trữ các luật gán tương ứng màu với thanh ghi.

Ta cũng hiện thực cấu trúc dữ liệu cho Node và Graph như sau:

```
1 class Node():
2     def __init__(self, var, reg, start_time, end_time, val):
3         self.var = var
4         self.reg = reg
5         self.start_time = start_time
6         self.end_time = end_time
7         self.val = val
8         self.adjacent_nodes = []
9         self.color = None
10
11     def add_edge(self, node):
12         self.adjacent_nodes.append(node)
13
14     def live_ranges(self):
15         return self.end_time - self.start_time
```

```
1 class Graph:
2     def __init__(self):
3         self.nodes = []
4         self.rules = []
5
6     def add_node(self, var, reg, start_time, end_time, val):
7         node = Node(var, reg, start_time, end_time, val)
8         self.nodes.append(node)
9
10    def add_edge(self, var1, var2):
11        node1 = self.find_node_by_var(var1)
12        node2 = self.find_node_by_var(var2)
13        if self.check_overlap(node1, node2):
14            node1.add_edge(node2)
15            node2.add_edge(node1)
16
17    def find_node_by_var(self, var):
18        for node in self.nodes:
19            if node.var == var:
20                return node
21        return None
22
23    def check_overlap(self, node1, node2):
24        return not(node1.end_time < node2.start_time or node2.
25                    end_time < node1.start_time)
26
27    def sort_nodes(self):
28        self.nodes.sort(key=lambda node: (node.var.count('*'), node.
29            live_ranges()), reverse=True)
29
30    def color_graph(self):
31        colored = self.color_util(8)
```



```
31         while not colored:
32             self.rules.append({'data': [self.nodes[-1].var, self.
33                                     nodes[-1].reg, self.nodes[-1].val]})
34             self.nodes.pop()
35             colored = self.color_util(8)
36
37     def color_util(self, num_colors):
38         for node in self.nodes:
39             available_colors = set(range(1, num_colors + 1))
40             for adj_node in node.adjacent_nodes:
41                 if adj_node.color in available_colors:
42                     available_colors.remove(adj_node.color)
43             if not available_colors:
44                 return False
45             node.color = min(available_colors)
46         return True
47
48     def gen_rules(self):
49         for node in self.nodes:
50             self.rules.append({'node.reg': 't' + str(node.color - 1)})
51         return self.rules
```

Trong đó,

- Phương thức *add_node*: thêm node vào graph
- Phương thức *add_graph*: duyệt qua tất cả các node, tạo ra các cạnh dựa vào phương thức *check_overlap*.
- Phương thức *check_overlap*: kiểm tra xem 2 node có live ranges lồng nhau hay không, nếu lồng nhau tức là có cạnh giữa 2 node này.
- Phương thức *sort_nodes*: Sắp xếp lại list *self.nodes* theo thứ tự ưu tiên: các biến dùng để lưu số sẽ được ưu tiên nhất → các biến có live ranges dài nhất.

Việc sắp xếp này sẽ giúp cho việc xóa một nodes ra khỏi đồ thị mô tả trong phương thức *color_graph* dễ dàng hơn.

- Phương thức *color_graph*: Được sử dụng để chuẩn bị và kiểm tra việc tô màu đồ thị. Gọi *color_util* để tô màu, nếu không tô màu thành công, xóa một nodes ra khỏi đồ thị, cập nhật *rule* vào *self.rules* và tô màu lại. Lặp lại quá trình trên cho đến khi tô màu thành công.

Vì MIPS có 8 thanh ghi từ \$t0 đến \$t7 để lưu giá trị biến, nên ta chọn tô 8 màu cho đồ thị này.

- Phương thức *color_util*: Được sử dụng để tô màu đồ thị.
- Phương thức *gen_rules*: cập nhật *self.rules* các luật gán tương ứng màu với thanh ghi.

Ta phải cập nhật các phương thức *getFreeRegister* và *getFreeRegisterForNumber* để khi không tìm được thanh ghi trống, chúng phải cập nhật số lượng *self.number_of_spill* và *self.need_to_reallocate* thành *True*, đồng thời trả về một thanh ghi ảo nào đó (ta đặt tên cho thanh ghi ảo này theo quy luật '*SPILL*' + *str(self.number_of_spill)*), cập nhật *self.register*.

```
1 def getFreeRegister(self):
2     for key, value in self.register.items():
3         if len(value) == 0:
4             return key
5     self.number_of_spill += 1
6     result = 'SPILL' + str(self.number_of_spill)
7     self.register.update({result: []})
8     self.need_to_reallocate = True
9     return result
10
11 def getFreeRegisterForNumber(self, val):
12     for key, value in self.register.items():
13         if key not in self.register_dest and val in value:
14             return key
15
16     if self.register_dest != []:
17         for key, value in self.register.items():
18             if key not in self.register_dest:
19                 self.register[key] = [item for item in value if not
20                                     isinstance(item, (int, float))]
21
22     for key, value in self.register.items():
23         if len(value) == 0:
24             return key
25     self.number_of_spill += 1
26     result = 'SPILL' + str(self.number_of_spill)
27     self.register.update({result: []})
28     self.need_to_reallocate = True
29     return result
```

Hiện thực thuật toán (3):

```
1 def reallocate(self, allVars: dict):
2     graph = Graph()
3     for key, value in allVars.items():
4         graph.add_node(key, value[0], value[2], value[3], int(value
5                             [4]))
6     for i in range(0, len(graph.nodes)):
7         for j in range(i+1, len(graph.nodes)):
8             graph.add_edge(graph.nodes[i].var, graph.nodes[j].var)
9
10    graph.sort_nodes()
11    graph.color_graph()
12    self.rules = graph.gen_rules()
```

3.5 Hiện thực các hàm visit cần thiết

Đầu tiên, ta cập nhật *visitFuncDecl*. Vì MIPS không thể tự động kết thúc chương trình, ta cần phải thêm lệnh *jal exit* ở cuối hàm *main*. Hơn nữa, sau khi hoàn thành việc visit qua một hàm nào đó, ta cũng cần giải phóng các thanh ghi cần thiết (ví dụ như các thanh ghi đích đã đề cập) bằng phương thức *releaseRegisterInScope(self, scope: str)*.

Phương thức *releaseRegisterInScope(self, scope: str)*:

1. Duyệt qua tất cả các phần tử trong *self.allVars* để lấy tất cả các biến trong scope tương ứng
2. Sau đó thực hiện remove các phần tử thanh ghi khỏi *self.register*

```
1 def releaseRegisterInScope(self, scope: str):
2     allRegsToRelease = []
3     for key, value in self.allVars.items():
4         if value[1] == scope:
5             allRegsToRelease.append((key, value[0]))
6     for reg in allRegsToRelease:
7         if reg[0] in self.register[reg[1]]:
8             self.register[reg[1]].remove(reg[0])
9
10 def visitFuncDecl(self, ast, o):
11     subctxt = o
12     frame = Frame(ast.name, None)
13
14     self.emit.printout(self.emit.emitMETHOD(ast.name.name))
15
16     glenv = subctxt.sym
17     body = ast.body
18
19     for ele in body.stmt:
20         self.count += 1
21         glenv = [self.visit(ele, SubBody(frame, glenv))] + glenv
22
23     self.releaseRegisterInScope(ast.name.name)
24     if ast.name.name == "main":
25         self.emit.printout("    jal exit")
26     return SubBody(None, [Symbol(ast.name, MType(list(), None), CName
        (self.className))] + subctxt.sym)
```

Tiếp đó, ta cập nhật phương thức *visitCallStmt(self, ast, o)*. Như đã trình bày, với mã MIPS ta cần đến thủ tục *syscall* để gọi các lệnh nhập xuất. Như vậy, trước hết ta kiểm tra xem hàm đang được gọi có phải là một trong các hàm nhập xuất hay không (trong bài tập này là hàm *writeNumber*), nếu đúng là hàm nhập xuất thì sinh mã bằng phương thức *self.emit.emitLOADSYSCALL(reg_des)*.

Phương thức *emitLOADSYSCALL(self, reg_des)* trong *class Emitter()*: Để thực hiện *syscall*, ta cần load đúng thanh ghi cần thiết vào thanh ghi \$a0. Với *writeNumber* là hàm in ra một số, ta cần load thanh ghi chứa giá trị cần in vào \$a0.

Vì ta đã lưu các thanh ghi đích cần thiết vào *in_* trong *visitCallStmt*, nên ta chỉ cần gọi *self.emit.emitLOADSYSCALL(in_[2])*. Hơn nữa, tương tự như *visitFuncDecl*, ta cũng cần giải phóng các thanh ghi cần thiết (ví dụ như các thanh ghi đích đã đề cập). Với *visitCallStmt* ta giải phóng bằng phương thức *releaseRegisterForNumber(True)*.

```
1 class MipsCode(MachineCode):
2     def emitLOADSYSCALL(self, reg_des):
3         return MipsCode.INDENT + "move $a0, $" + reg_des + MipsCode.
4             END
5
6 class Emitter():
7     def emitLOADSYSCALL(self, reg_des):
8         return self.mips.emitLOADSYSCALL(reg_des)
9
10 def visitCallStmt(self, ast, o):
11     ctxt = o
12     frame = ctxt.frame
13     nenv = ctxt.sym
14
15     in_ = ("", list(), "")
16     for x in ast.args:
17         str1, typ1, reg1 = self.visit(x, Access(frame, nenv, False,
18             True))
19         in_ = (in_[0] + str1, in_[1].append(typ1), in_[2] + reg1)
20
21     if ast.name.name == 'writeNumber':
22         self.emit.printout(self.emit.emitLOADSYSCALL(in_[2]))
23         self.emit.printout(self.emit.emitINVOKEFUNCTION(ast.name.name))
24         self.releaseRegisterForNumber(True)
```

Cuối cùng, trong *visitProgram* ta duyệt qua tất cả các lệnh trong chương trình, sinh ra mã MIPS (tạm gọi là *mã tạm*)

1. Kiểm tra *self.need_to_reallocate*, nếu không cần cấp phát lại thanh ghi, kết thúc.
2. Nếu cần cấp phát lại thanh ghi, *reallocate* các biến và thanh ghi, chỉnh sửa lại *mã tạm* bằng phương thức *self.emit.emitMIPS(self.rules)*.
Phương thức *self.emit.emitMIPS(self.rules)*:
 - (a) Sinh ra vùng *.data* cho các *spill*.
 - (b) Sinh ra vùng *.text* (tức là chương trình chính) bằng cách thay các thanh ghi trong *mã tạm* thành các thanh ghi đúng thông qua list *self.rules*.

```
1 def visitProgram(self, ast, c):
2     # visit the declarations ...
3     if self.need_to_reallocate == True:
4         self.reallocate(self.allVars)
5         self.emit.emitMIPS(self.rules)
6     return c
```

4 Kiểm thử

Sau khi đã hiện thực, ta tiến hành kiểm thử chương trình có chạy đúng hay không. Đầu tiên, ta kiểm thử với một số testcase đơn giản.

Cộng hai số: Ta chọn testcase là $1 + 1$ để kiểm tra mã sinh ra có giống với mong muốn hay không.

```
1 def test_501(self):
2     input = ""func main()
3     begin
4         writeNumber(1+1)
5     end
6     ""
7     expect = "2\n"
8     self.assertTrue(TestCodeGen.test(input, expect, 501))
```

Với testcase trên, trước khi chạy chương trình, ta phân tích trước kết quả đầu ra:

- Đầu tiên, \$t0 chưa được sử dụng nên ta gán giá trị 1 cho thanh ghi \$t0.
- Tiếp theo, \$t0 đã được sử dụng nên ta gán giá trị 2 cho thanh ghi \$t1.
- Mã sinh ra cho phép cộng là *add \$t0, \$t0, \$t1*

Sau khi chạy chương trình, mã sinh ra là

```
1 main:
2     li $t0, 1
3     li $t1, 1
4     add $t0, $t0, $t1
5     move $a0, $t0
6     jal writeNumber
7     jal exit
```

Chú ý vào dòng 2, 3, 4 trong đoạn mã trên, ta nhận thấy kết quả đã như mong muốn.

Cộng số với biến: Ta chọn testcase là $a + 3$ để kiểm tra mã sinh ra có giống với mong muốn hay không.

```
1 def test_512(self):
2     input = ""
3     number a <- 2
4     func main()
5     begin
6         writeNumber(a+3)
7     end
8     ""
9     expect = '5\n'
10    self.assertTrue(TestCodeGen.test(input, expect, 512))
```

Tương tự trên, ta phân tích trước kết quả đầu ra:

- Đầu tiên, \$t0 chưa được sử dụng nên ta gán giá trị 2 cho thanh ghi \$t0.
- Tiếp theo, \$t0 đã được sử dụng nên ta gán giá trị 3 cho thanh ghi \$t1.
- Mã sinh ra cho phép cộng là *add \$t0, \$t0, \$t1*

Sau khi chạy chương trình, mã sinh ra là

```
1 li $t0, 2
2
3 main:
4     li $t1, 3
5     add $t1, $t1, $t0
6     move $a0, $t1
7     jal writeNumber
8     jal exit
```

Chú ý vào dòng 1, 4, 5 trong đoạn mã trên, ta nhận thấy kết quả đã như mong muốn.

Cộng nhiều biến: Ta chọn testcase cộng giữa 11 biến để kiểm tra mã sinh ra có giống với mong muốn hay không.

```
1 def test_526(self):
2     input = ""
3     func main()
4     begin
5         number a <- 2
6         number b <- 2
7         number c <- 2
8         number d <- 2
9         number e <- 2
10        number f <- 2
11        number g <- 2
12        number h <- 2
13        number i <- 2
14        number j <- 2
15        number k <- 2
16        writeNumber(a+b+c+d+e+f+g+h+i+j+k)
17    end
18    ""
19    expect = '22\n'
20    self.assertTrue(TestCodeGen.test(input, expect, 526))
```

Tương tự trên, ta phân tích trước kết quả đầu ra:

- Có đến 11 biến, nên chắc chắn có *spill*.
- Cần 1 biến làm thanh ghi đích, nên cần có 11+1 thanh ghi, vậy sẽ có 11+1-8 *spill*.
- Nhận thấy *live ranges* của k, j, i, h là ngắn nhất, nên 4 *spill* chính là k, j, i, h.

Sau khi chạy chương trình, mã sinh ra là

```
1 .data
2     k: .word 2
3     j: .word 2
4     i: .word 2
5     h: .word 2
6
7 .text
8     main:
9         li $t1, 2
10        li $t2, 2
11        li $t3, 2
12        li $t4, 2
13        li $t5, 2
14        li $t6, 2
15        li $t7, 2
16        #li $t7, 2
17        #li $SPILL1, 2
18        #li $SPILL2, 2
19        #li $SPILL3, 2
20        add $t0, $t1, $t2
21        add $t0, $t0, $t3
22        add $t0, $t0, $t4
23        add $t0, $t0, $t5
24        add $t0, $t0, $t6
25        add $t0, $t0, $t7
26        lw $t8, h
27        add $t0, $t0, $t8
28        lw $t8, i
29        add $t0, $t0, $t8
30        lw $t8, j
31        add $t0, $t0, $t8
32        lw $t8, k
33        add $t0, $t0, $t8
34        move $a0, $t0
35        jal writeNumber
36        jal exit
```

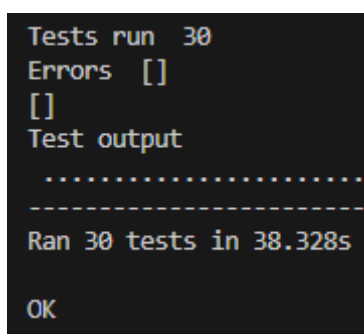
Chú ý vào vùng *.data* trong đoạn mã trên, ta nhận thấy kết quả đã như mong muốn.

Sau khi kiểm tra testcase đơn giản trên thành công, ta thiết kế nhiều testcase khác. Các testcase sẽ được thiết kế để kiểm thử cho các trường hợp khác nhau, với các trường hợp và số lượng cụ thể tương ứng như bảng bên dưới.

STT	Phép cộng	Số lượng
1	Giữa hai số bất kỳ	5
2	Giữa nhiều số bất kỳ	5
3	Giữa nhiều biến toàn cục và nhiều số bất kỳ - không <i>spill</i>	5
4	Giữa nhiều biến trong hàm main và nhiều số bất kỳ - không <i>spill</i>	5
5	Giữa nhiều biến bất kỳ và nhiều số bất kỳ - không <i>spill</i>	5
5	Giữa nhiều biến bất kỳ và nhiều số bất kỳ - có <i>spill</i>	5

Bảng 4: Thiết kế testcase

Sau khi chạy kiểm thử, kết quả chương trình đã đúng tất cả các testcase.



```
Tests run 30
Errors []
[]
Test output
.....
-----
Ran 30 tests in 38.328s
OK
```

Hình 1: Kết quả khi chạy kiểm thử

5 Kết luận

Như vậy, ta đã hoàn thành việc cấp phát thanh ghi và sinh mã MIPS cho phép cộng số nguyên dương. Sau bài tập này, chúng ta đã được ôn tập lại và nắm vững hơn các kiến thức cơ bản về ngôn ngữ lập trình, về kiến trúc máy tính và mô hình hóa toán học.

Với các testcase trên, ta tạm thời có thể kết luận chương trình chạy tốt với phép cộng số nguyên dương. Tuy nhiên, chúng ta vẫn chưa thể kết luận được chiến lược cấp phát thanh ghi như trên có phù hợp với các phép tính khác (như $-$, $*$, $/$) hay không. Hơn nữa, chúng ta cũng chỉ hiện thực trên một hàm *main* duy nhất, chưa hiện thực trên nhiều hàm khác nhau.

Chúng ta sẽ tiếp tục phát triển và giải quyết tất cả các vấn đề vừa nêu trên ở bài tập tiếp theo. Khi đó, ta sẽ hoàn thiện việc sinh mã MIPS cho một chương trình ZCode hoàn chỉnh.



Tài liệu tham khảo

- [1] Hugh Leather. *Compiler Optimisation 7 – Register Allocation*. <https://www.inf.ed.ac.uk/teaching/courses/copt/lecture-7.pdf>.
- [2] missouristate. *MarsHelpCommand*. https://courses.missouristate.edu/kenvollmar/mars/Help/Help_4_1/MarsHelpCommand.html.