Kỹ thuật dịch ngược

Ngày 1 tháng 12 năm 2016

Lời cam đoan

Lời cảm ơn

Tóm tắt luận văn

Mục lục

Danh sách hình vẽ

Listings

Chương 1

Giới thiệu

Chương này nhằm mục đích giới thiệu về bài toán sẽ được giải quyết trong luận văn và các khái niệm liên quan. Đầu chương sẽ nói về kỹ thuật dịch ngược, các ứng dụng của nó và những khó khăn trong quá trình dịch ngược. Phần tiếp theo trình bày bài toán đặt ra và các thách thức khi giải quyết bài toán. Phần cuối cùng tóm tắt cấu trúc của luận văn.

1.1 Kỹ thuật dịch ngược và ứng dụng

Trong khi kỹ thuật dịch phổ biến hiện nay là dịch từ mã viết bằng ngôn ngữ cấp cao xuống mã ngôn ngữ cấp thấp hơn, kỹ thuật dịch ngược thực hiện dịch từ mã ngôn ngữ cấp thấp lên mã ngôn ngữ cấp cao hơn. Kỹ thuật dịch ngược được sử dụng rất nhiều để hỗ trợ trong quá trình phát triển và sử dụng phần mềm:

- Vì một lý do nào đó, mã nguồn của một phần mềm bị mất đi. Để tiếp tục phát triển phần mềm hoặc bảo trì phần mềm đó, ta cần phải khôi phục lại mã nguồn. Nếu viết lại một chương trình mới hoàn toàn từ các tài liệu sẵn có sẽ rất mất thời gian và không đảm bảo sẽ tương đương được phần mềm cũ. Vì vậy một giải pháp phổ biến hiện nay là dựa vào file thực thi dịch ngược lại và hiệu chỉnh để có được mã nguồn mới hoàn chỉnh.

- Các phần mềm độc hại như virus, malware thường sẽ được giấu kín mã nguồn, nếu có được mã nguồn của chúng thì việc tìm ra phương pháp giải trừ sẽ rất dễ dàng. Ta có thể ứng dụng kỹ thuật dịch ngược để làm việc đó.

- Chuyển đổi chương trình chạy trên một phần cứng này sang chương trình chạy trên một phần cứng khác. Ví dụ: ta có mã đang chạy trên chip 8051, nhưng vì chip sẽ bị ngừng sản xuất trong một vài năm nữa, nên yêu cầu đề ra là tạo ra mã tương đương chạy trên một con chip khác hiện đại hơn. Để làm được điều đó, ta có thể dùng trình dịch ngược dịch mã viết cho 8051 lên một ngôn ngữ cấp cao, và sử dụng tiếp trình biên dịch để dịch mã nguồn đó thành mã của chip thay thế. Vì mã viết trong các hệ thống nhúng thường là mã assembly, nên cụ thể trình dịch ngược trong hệ thống chuyển đổi này sẽ chuyển từ mã assembly lên mã ngôn ngữ cấp cao. Và đó cũng là đối tượng nghiên cứu của luận văn này.

- Phần mềm viết bằng ngôn ngữ A bắt buộc phải chuyển đổi sang ngôn ngữ B để tiếp tục bảo trì và phát triển. Ngôn ngữ A có thể là một ngôn ngữ đã ra

Hình 1.1: Một ứng dụng của trình dịch ngược: chuyển đổi mã nguồn giữa các kiến trúc máy khác nhau

đời từ rất lâu (ví dụ: COBOL, Basic...), hiện nay không còn người hiểu biết về ngôn ngữ đó để lập trình phần mềm. Vì vậy, cần phải chuyển đổi phần mềm sang một ngôn ngữ khác mới hơn, có nhân lực để viết tiếp (ví dụ: Java, C#...). Quá trình này cũng được xem là dịch ngược, vì thường ngôn ngữ A ra đời trước sẽ có mức độ trừu tượng thấp hơn là các ngôn ngữ B được phát triển sau này.

Để xây dựng một công cụ dịch ngược thành công, có nhiều vấn đề cần phải giải quyết. Nhưng tựu chung lại, bài toán cơ bản nhất vẫn là khôi phục các thông tin. Khi dịch một chương trình từ mã nguồn viết bằng ngôn ngữ cấp cao xuống mã máy, có nhiều thông tin sẽ bị mất đi vì không còn cần thiết ở các ngôn ngữ cấp thấp nữa. Tuy nhiên, khi dịch lại lên ngôn ngữ cấp cao, nếu như không có những thông tin đó thì chương trình sẽ rất khó đọc, từ đó dẫn đến khó bảo trì và sửa chữa; hoặc phải chỉnh sửa chương trình đầu ra bằng tay rất nhiều. Các thông tin tiêu biểu cần khôi phục là:

- Kiểu dữ liệu của biến: Đối với các chương trình viết bằng ngôn ngữ cấp cao, kiểu dữ liệu của biến có thể xem như một ràng buộc khi gán giá trị cho biến và sử dụng biến. Ví dụ khi ta khai báo một biến có kiểu dữ liệu là integer, thì ta phải gán cho biến các giá trị là số nguyên (1, 2,...) và sử dụng biến trong các phép toán có toán tử là số nguyên. Nếu ta gán cho biến một giá trị khác số nguyên (số thực, chuỗi, boolean...) hoặc sử dụng biến trong các phép toán không chấp nhận toán tử là số nguyên thì trình biên dịch sẽ phát hiện lỗi ngay ở giai đoạn đầu. Tuy nhiên, đối với mã máy, dữ liệu được lưu trong các thanh ghi hoặc vùng nhớ, nên kiểu dữ liệu không còn cần thiết và sẽ được loại bỏ trong quá trình biên dịch. Khi dịch ngược, nếu không khôi phục được kiểu dữ liệu, người lập trình sẽ rất khó khăn trong quá trình sử dụng biến vì không biết nên gán giá trị nào và sử dụng ở đâu.

- Tên của biến: Trong quá trình lập trình, tên biến có hai chức năng chính: gợi

nhớ tác dụng của biến và truy xuất đến biến đó. Người lập trình khi đặt tên cho một biến thường sẽ dựa theo công dụng của biến đó để dễ dàng trong quá trình phát triển và bảo trì sau này. Một chương trình sẽ bị đánh giá là viết không tốt nếu tên biến được đặt lung tung và không có ý nghĩa nào. Ngoài ra, lập trình viên thường sẽ không quan tâm tới việc biến được lưu ở chỗ nào của vùng nhớ, khi cần truy xuất thì họ chỉ cần gọi tên biến. Ngược lại, với mã máy, để truy xuất một biến thì cần phải có tên thanh ghi hoặc địa chỉ vùng nhớ lưu biến đó. Vì vậy, tên biến đối với mã máy là không cần thiết và cũng bị loại bỏ. Tương tự như kiểu dữ liệu, nếu chương trình đầu ra không giữ được tên biến của chương trình gốc thì rất khó để phát triển và bảo trì. Giải pháp hiện nay của các trình dịch ngược là dựa vào các tài liệu sẵn có để chỉnh sửa tên biến bằng tay sau khi đã có chương trình đầu ra.

- Phân biệt giữa dữ liệu và mã điều khiển: Đặc điểm của một số mã máy (trừ mã máy chạy trên máy ảo) là dữ liệu và các câu lệnh điều khiển có cùng một định dạng mã nhị phân và được lưu trong cùng một vùng nhớ. Vì vậy, khi dịch ngược từ mã máy lên cần phải phân biệt được phần nào của vùng nhớ là lưu các dữ liệu và phần nào là câu lệnh của chương trình.

Từ khái niệm của kỹ thuật dịch ngược, ta thấy có nhiều mức độ dịch ngược, tương ứng với những bài toán khác nhau cần giải quyết. Nếu lấy đầu ra của quá trình dịch ngược là một chương trình viết bằng ngôn ngữ cấp cao, thì đầu vào của nó có thể là: mã nhị phân, mã assembly hoặc mã của một ngôn ngữ lập trình cấp cao khác nhưng mức độ trừu tượng thấp hơn. Tùy vào mức độ trừu tượng của ngôn ngữ đầu vào, các thông tin bị mất ở mã đầu vào sẽ khác nhau. Với mã máy thì tất cả các thông tin nêu trên đều không còn. Với mã assembly, tên biến vẫn xuất hiện trong chương trình vì một số assembler cho phép có các câu lệnh khai báo biến ở mã assembly. Còn với mã ngôn ngữ cấp cao thì gần như tất cả thông tin đều có ở chương trình gốc, và vấn đề cần giải quyết là tìm ra các cấu trúc tương đương ở ngôn ngữ đích.

1.2 Bài toán đặt ra

Như đã đề cập ở phần trên, mục tiêu của luận văn là nghiên cứu về trình dịch ngược từ mã assembly lên mã cấp cao, các bài toán cần phải giải quyết và hiện thực giải pháp. Vì mã assembly cho kiến trúc máy khác nhau có những đặc điểm khác nhau, và đi cùng với đó là những vấn đề khác nhau cần giải quyết, nên giới hạn của luận văn sẽ là trình dịch ngược từ mã assembly 8051. Việc chọn kiến trúc máy 8051 là do 2 nguyên nhân sau:

- Chip 8051 đã xuất hiện trên thị trường từ lâu, hiện tại sắp không còn được sản xuất. Tuy nhiên, vẫn còn nhiều hệ thống được chạy trên đây và cần phải chuyển đổi chúng sang một kiến trúc máy khác hiện đại hơn. Như vậy, nhu cầu đặt ra là có thực.

- Chip 8051 có một số đặc điểm khác biệt so với các con chip khác trên thị trường. Vì vậy việc dịch ngược từ mã 8051 sẽ gặp nhiều khó khăn hơn, vấn đề phải giải quyết phức tạp hơn.

Các đặc điểm khác biệt của 8051 gồm có:

- Trong khi hầu hết các kiến trúc máy khác sử dụng kiểu dữ liệu byte là kiểu dữ liệu nhỏ nhất, thì 8051 cho phép lập trình viên truy xuất tới mức bit trong một số thanh ghi và kèm theo đó là các câu lệnh xử lý bit. Tuy nhiên, các thanh ghi này của 8051 cũng có thể được truy xuất ở mức byte bình thường. Xem ví dụ ở đoạn mã 1.3, câu lệnh số 1 gán giá trị ở vùng nhớ có địa chỉ 38H cho toàn bộ thanh ghi ACC, trong khi câu lệnh số 2 chỉ sử dụng biến số 1 của thanh ghi ACC.

  Listing 1.1: Một đoạn mã 8051 sử dụng cả biến bit và biến byte của thanh ghi ACC

```
MOV ACC, 38H #1
SETB ACC.1 #2
```

- Một số assembler của 8051 cho phép sử dụng tên biến. Biến này dùng để lưu các giá trị hằng số, hằng số này thường là địa chỉ một vùng nhớ kích thước 1 byte (trong luận văn này sẽ gọi tắt là biến byte) hoặc đại diện cho bit của thanh ghi (gọi tắt là biến bit). Khi lập trình, người ta thường sử dụng biến byte và biến bit này theo bộ, nghĩa là chỉ khi thanh ghi được load vào giá trị vùng nhớ quy định bởi biến byte, thì các biến bit cùng bộ mới được sử dụng (xem ví dụ ở đoạn mã 1.2) (từ nay, khi luận văn sử dụng từ "nguyên tắc sử dụng bộ biến", nghĩa là đang đề cập đến nguyên tắc này).

  Listing 1.2: Một đoạn mã 8051 tuân theo nguyên tắc sử dụng bộ biến

```
#DEFINE OPTIONS #38H
#DEFINE TESTSUP ACC.1
public AA
AA:
MOV ACC, OPTIONS
JB TESTSUP, BB
```

Từ các đặc điểm trên, ta có thể thấy bài toán lớn nhất đặt ra trong luận văn này sẽ là tìm ra được mối liên hệ giữa biến byte và biến bit trong chương trình, lấy được các bộ biến byte và biến bit đúng. Có 2 cách để biết được điều này:

- Đưa ra quy định về việc khai báo biến byte và biến bit. Hiện nay, ở phần khai báo, các lập trình viên có thể khai báo các biến theo thứ tự tuỳ ý, và cũng không có quy định nào bắt buộc họ phải có phần comment chỉ rõ các biến byte và biến bit nào là cùng một bộ. Ta có thể đưa ra các mẫu khai báo cho biến byte và biến bit để trình dịch ngược có thể biết được các bộ biến bằng cách đọc theo mẫu mà không cần phân tích gì thêm. Tuy nhiên, sau khi đã xác định được các bộ biến này, cần có thêm một bước kiểm tra mã chương trình để đảm bảo rằng nguyên tắc sử dụng biến byte và biến bit được tuân thủ. Vì vậy, ta sẽ gọi giải pháp này là Kiểm tra kiểu - Type checking. Giải pháp này có ưu điểm là đơn giản, dễ hiện thực nhưng gây bất tiện cho người dùng vì phải chuyển đổi bộ mã hiện tại về theo mẫu quy định.

- Dựa vào phân tích luồng dữ liệu của chương trình, tìm ra được địa chỉ vùng nhớ được load vào thanh ghi tại thời điểm sử dụng biến bit và từ đó suy ra biến byte cùng bộ với biến bit đó. Giải pháp này được đặt tên là Suy luận

kiểu - Type inference. Với cách làm này, không cần phải thay đổi đoạn mã gốc. Tuy nhiên cách hiện thực sẽ phức tạp hơn nhiều vì có rất nhiều cách load dữ liệu vùng nhớ vào thanh ghi như: dùng trực tiếp hằng số, dùng trực tiếp biến byte, trung gian qua một thanh ghi khác, dùng một biểu thức toán học có 2 vế... (xem ví dụ ở đoạn mã **??**)

Listing 1.3: Ví dụ một số mẫu câu lệnh load vùng nhớ vào thanh ghi trong 8051

```
MOV ACC,  38H
MOV ACC,  OPTIONS
MOV ACC,  @DPTR
MOV ACC,  OPTIONS+1
```

Cả hai giải pháp này đều được hiện thực trong từng giai đoạn của luận văn và sẽ được trình bày trong các chương tiếp theo.

Ngoài ra, một công việc khác cần phải làm đó là giữ nguyên tên biến trong quá trình dịch ngược. Hiện tại trình dịch ngược Boomerang chỉ cho phép ta định nghĩa trước một số thanh ghi trong một kiến trúc máy, và đoạn mã đầu vào chỉ được sử dụng các thanh ghi đó, nếu sử dụng một cái tên nằm ngoài danh sách thanh ghi thì sẽ báo lỗi. Vì vậy, ta sẽ phải điều chỉnh cơ chế này, cho phép việc sử dụng tên biến khác và giữ nguyên chúng khi dịch ra đoạn mã ngôn ngữ cấp cao.

1.3 Cấu trúc luận văn

Luận văn sẽ gồm 6 chương như sau:

- Chương 1: Giới thiệu về kỹ thuật dịch ngược, bài toán đặt ra trong luận văn và cấu trúc của luận văn.
- Chương 2: Nêu lên một số kiến thức cơ bản và các nghiên cứu liên quan đến luận văn. Đặc biệt, trong chương này sẽ trình bày một số kiến thức cơ bản về Boomerang, giúp người đọc dễ dàng hiểu các phần sau hơn.
- Chương 3: Trình bày giải pháp Kiểm tra kiểu - Type checking. Ngoài ra, trong chương này sẽ trình bày cơ chế cho phép đọc tên biến khác thanh ghi và lưu trữ chúng trong trình dịch ngược, vì đây là bước đầu tiên trong quá trình thực hiện các giải pháp.
- Chương 4: Trình bày giải pháp Suy luận kiểu - Type inference.
- Chương 5: Đánh giá kết quả của luận văn thông qua các mẫu thử (testcase).
- Chương 6: Kết luận.

Chương 2

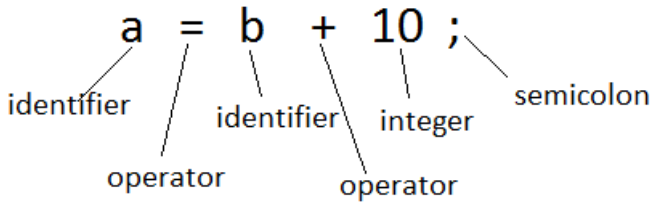
Các kiến thức nền tảng và nghiên cứu liên quan

2.1 Trình biên dịch

Trình biên dịch (compiler) là một chương trình hoặc một bộ chương trình máy tính, có nhiệm vụ biến đổi mã nguồn được viết bằng một ngôn ngữ lập trình này (ngôn ngữ lập trình gốc) sang một ngôn ngữ lập trình khác (ngôn ngữ lập trình đích), thường sẽ có dạng nhị phân và thực thi được.
Các bước của trình biên dịch gồm có:

- Phân tích từ vựng (lexical analysis): Đây là quá trình chuyển hóa một chuỗi các ký tự (ví dụ như các câu lệnh trong một chương trình máy tính) thành chuỗi các từ tố (token), ví dụ như: định danh, số nguyên, số thực... Giai đoạn này kiểm tra các lỗi về từ vựng, chính tả của chương trình.

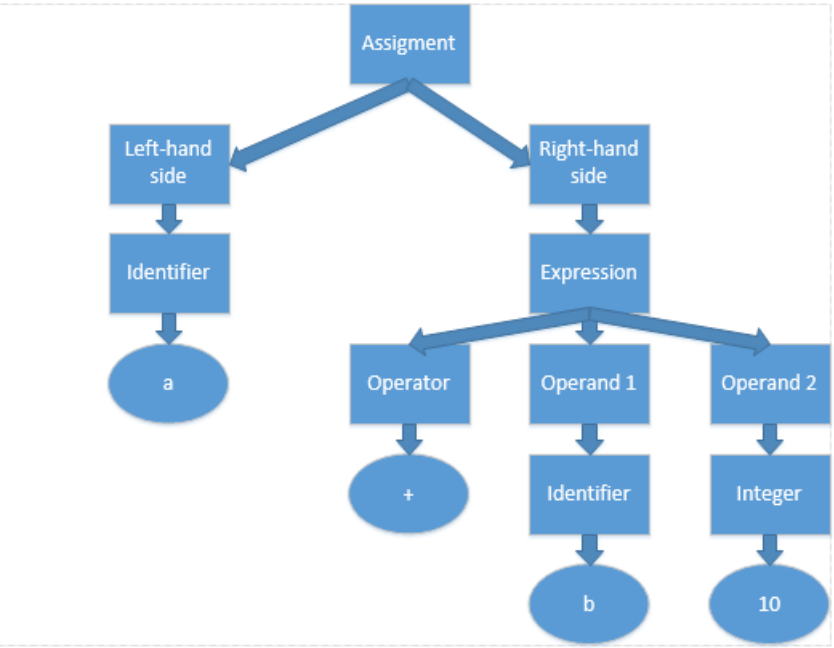
  Trong hình 2.1, các ký tự *a*, *b* được nhận diện là các định danh (identifier), *10* được nhận diện là số nguyên (integer), = và + được nhận diện là các phép tính (operator), *;* là ký tự đặc biệt kết thúc một câu lệnh.



Hình 2.1: Phân tích một câu lệnh thành các token

- Phân tích cú pháp (syntax analysis): Từ chuỗi các từ tố được tạo ra ở giai đoạn trên, một chương trình gọi là parser sẽ tạo ra một cấu trúc dữ liệu, thường là parse tree hoặc abstract syntax tree. Giai đoạn này sẽ kiểm tra các lỗi về cấu trúc ngữ pháp.

Câu lệnh gán từ ví dụ trên sẽ được phân tích thành cây cấu trúc như hình 2.2



Hình 2.2: Cây cấu trúc cho một câu lệnh gán

- Phân tích ngữ nghĩa (sematic analysis): Trong giai đoạn này, từ cây cấu trúc đã có, trình biên dịch sẽ áp dụng các luật về ngữ nghĩa để kiểm tra tính đúng đắn của chương trình. Thường sẽ là các luật về kiểu dữ liệu, kiểm tra tầm vực của biến và object binding.
  Tiếp tục ví dụ ở trên, trong gian đoạn này, trình biên dịch sẽ kiểm tra xem các biến a và b đã được khai báo chưa, tầm vực của các biến có phủ tới vị trí của câu lệnh không (ví dụ: có những biến được khai báo ở hàm A thì sẽ không có tầm vực ở bên ngoài hàm A), kiểu của biến có phù hợp với câu lệnh gán không (ví dụ: nếu b có kiểu là **string** thì câu lệnh trên không hợp lệ).

```
int a, b;
b = 20;
a = b + 10;
```

```
int a;
string b = "hello";
a = b + 10;
```

Hình 2.3: Ví dụ về lỗi kiểu biến

Trong hình 2.3, cả hai đoạn mã đều hợp lệ tính đến cuối giai đoạn phân tích cú pháp. Tuy nhiên, giai đoạn phân tích ngữ nghĩa sẽ phát hiện ra đoạn mã ở bên phải không hợp lệ vì nó vi phạm các ràng buộc về kiểu.

- Tạo ra mã trung gian: Sau khi trải qua các giai đoạn phân tích và kiểm tra, trình biên dịch sẽ tiến hành sinh mã trung gian từ mã nguồn. Đặc điểm của mã trung gian là đơn giản và rất gần với mã đích, tuy nhiên con người vẫn có thể đọc và hiểu được. Việc sinh mã trung gian nhằm giảm thiểu chi phí cho trình biên dịch khi phải sinh mã đích cho nhiều kiến trúc máy khác nhau. Thay vì với mỗi kiến trúc máy, trình biên dịch phải tạo ra công cụ riêng để

dịch từ mã nguồn sang mã đích, thì ở đây chỉ cần tạo ra công cụ để dịch từ mã trung gian - vốn đã rất gần với mã đích.

- Tạo mã đích: Từ mã trung gian, tùy vào kiến trúc máy sẽ thực thi chương trình, trình biên dịch sẽ tạo ra mã đích tương ứng. Giai đoạn này sẽ thực hiện các công việc như: lựa chọn câu lệnh trung gian sẽ thực hiện, quyết định các giá trị được lưu trong thanh ghi, sắp xếp thứ tự thực hiện các câu lệnh. Đầu ra của giai đoạn là mã máy có thể thực thi được.
- Tối ưu mã đích: Để tăng tốc độ thực hiện chương trình cũng như giảm các chi phí chạy chương trình, giai đoạn tối ưu mã đích sẽ kiểm tra và áp dụng các kỹ thuật nhằm loại bỏ mã chết, tối ưu vòng lặp, loại bỏ dư thừa... Giai đoạn này không nhất thiết chỉ thực hiện ở cuối quá trình biên dịch mà có thể nằm ở bất cứ đâu.

2.2 Trình dịch ngược

Mục tiêu của trình dịch ngược là chuyển đổi chương trình được viết bằng một ngôn ngữ cấp thấp (thường là mã máy) lên một ngôn ngữ cấp cao hơn (như C, C++...). Vì vậy, trình dịch ngược (decompiler) có thể xem như một quá trình đảo ngược của trình biên dịch (compiler). Chương trình đầu ra phải thực hiện được những chức năng tương đương như chương trình đầu vào.

```c
#include<stdio.h>

main()
{
    printf("Hello World");
}
```

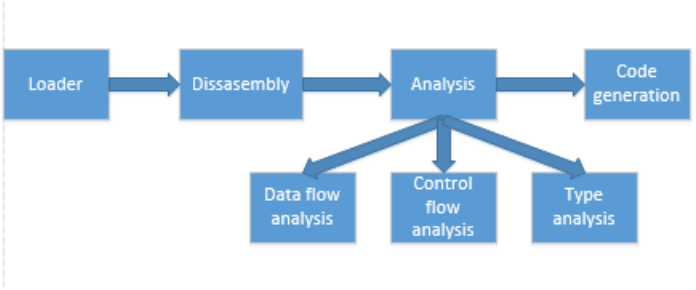
```c
// address: 0x804841d
int main(int argc, char *argv[], char *envp[]) {
    __size32 eax;                // r24

    printf("Hello World");
    return eax;
}
```

Hình 2.4: Đoạn mã gốc và đoạn mã được dịch ngược bởi trình dịch ngược Boomerang

Qúa trình dịch ngược có thể chia thành các giai đoạn sau:

- Loader: Load file cần dịch ngược, đọc từ file ra các thông tin như: loại file, loại kiến trúc máy... và xác định được ngõ vào của chương trình (tương đương với hàm main trong C).
- Disassembly: Mã gốc sẽ được chuyển thành mã trung gian, mã trung gian là gì thì tùy vào trình dịch ngược. Ví dụ Boomerang sẽ dùng mã trung gian là Register Transfer Language.
- Analysis: Sau khi đã chuyển sang mã trung gian, chương trình sẽ đi qua các bước phân tích để khôi phục lại thông tin đã mất trong quá trình biên dịch. Các phân tích thường phải có là: lan truyền biểu thức, loại bỏ mã chết, xác định nguyên mẫu hàm (function prototype), xác định kiểu dữ liệu...

- Code generation: Trải qua các kỹ thuật phân tích để xác định được thông tin cần thiết về dữ liệu, kiểu và luồng điều khiển chương trình, giai đoạn cuối cùng của dịch ngược là sinh ra mã chương trình bằng ngôn ngữ bậc cao.



Hình 2.5: Các bước cơ bản của một trình dịch ngược

2.3 Một số kỹ thuật tiêu biểu được sử dụng trong các công cụ dịch ngược

2.3.1 Lan truyền biểu thức

Lan truyền biểu thức (Expression propagation) là biến đổi phổ biến nhất trong quá trình dịch ngược một đoạn code. Nguyên tắc truyền biểu thức cũng rất đơn giản: Với các câu lệnh sử dụng giá trị của một biến nào đó, ta có thể thay tên biến đó bằng biểu thức nằm bên phải câu lệnh gán biến đó.
Hình 2.6 và 2.7 là một ví dụ cho lan truyền biểu thức. Trong hình 2.6, ta có các câu

```
          0 esp0 := esp          ; Save esp; see text
80483b0   1 esp := esp - 4
          2 m[esp] := ebp         ; push ebp
80483b1   3 ebp := esp
80483b3   4 esp := esp - 4
          5 m[esp] := esi         ; push esi
80483b4   6 esp := esp - 4
          7 m[esp] := ebx         ; push ebx
80483b5   8 esp := esp - 4
          9 m[esp] := ecx         ; push ecx
80483b6  10 tmp1 := esp
         11 esp := esp - 8        ; sub esp, 8
80483b9  13 edx := m[ebp+8]       ; Load n to edx
```

Hình 2.6: Một đoạn mã trước khi thực hiện lan truyền biểu thức

lệnh ở dạng mã trung gian trước khi thực hiện lan truyền biểu thức. Hình 2.7 là kết quả sau khi thực hiện lan truyền biểu thức. Ở đây ta giả sử có một biến đặc biệt là *esp0* được gán giá trị là giá trị ban đầu của biến *esp*. Ta sẽ thực hiện một thay thế đặc biệt ở câu lệnh số 1, thay vế phải của câu lệnh gán này - *esp* - bằng biến tương đương với nó là *esp0*. Sau đó, ở các câu lệnh tiếp theo, ta sẽ tiếp tục thay thế biến *esp* bằng các biểu thức tương đương. Ví dụ: Ở câu lệnh số 2, biểu thức tương đương của *esp* là *esp0 - 4*, còn ở câu lệnh số 5, biểu thức tương đương của *esp* là *esp0 - 8*

```
         0 esp0 := esp
80483b0  1 esp := esp0 - 4
         2 m[esp0-4] := ebp
80483b1  3 ebp := esp0-4
80483b3  4 esp := esp0 - 8
         5 m[esp0-8] := esi
80483b4  6 esp := esp0 - 12
         7 m[esp0-12] := ebx
80483b5  8 esp := esp0 - 16
         9 m[esp0-16] := ecx
80483b6 10 tmp1 := esp0 - 16
        11 esp := esp0 - 24
80483b9 13 edx := m[esp0+4]
```

Hình 2.7: Đoạn mã ở hình 2.6 sau khi thực hiện lan truyền biểu thức

(do *esp* đã được gán một giá trị mới ở câu lệnh số 4). Tuy nhiên, biểu thức *esp0 - 8* không thể được dùng để thay thế cho biến *esp* ở câu lệnh số 7 được, vì lúc đó *esp* đã mang giá trị khác.

Như vậy, qua ví dụ trên, ta có thể thấy việc lan truyền biểu thức từ câu lệnh a có dạng $x := exp$ đến một câu lệnh b chỉ có thể được thực hiện nếu đáp ứng hai điều kiện sau:

- a phải là câu lệnh gán có vế trái là x ở gần b nhất. Nói cách khác, giữa a và b không được có bất cứ câu lệnh gán nào khác có vế trái là x
- Trên tất cả các luồng đi của chương trình từ a tới b, không có câu lệnh gán nào có vế trái là bất kỳ biến nào được sử dụng trong a

```
1       a := b + 10;
2       m[b] := c * 20;
3       c := m[a] - 4;
4       d := m[b] + 3;
```

Hình 2.8: Đoạn mã trung gian với bốn câu lệnh gán đơn giản

Ở đoạn code hình 2.8, ta có thể thực hiện lan truyền biểu thức với biến a ở câu lệnh số 3, kết quả sẽ là *c := m[b + 10] - 4;* và câu lệnh gán biến a sẽ được loại bỏ bằng kỹ thuật loại bỏ mã chết được bàn ở phần tiếp theo. Tuy nhiên, ta không thể thực hiện lan truyền biểu thức với *m[b]* ở câu lệnh số 4, vì biến c được sử dụng trong câu lệnh gán số 2 đã được sử dụng làm vế trái trong câu lệnh gán số 3.

Với mã trung gian như trên, để kiểm tra hai điều kiện thỏa mãn việc lan truyền biểu thức phải mất rất nhiều thời gian, ta phải xét hết tất cả các luồng chương trình từ câu lệnh gán đến câu lệnh sử dụng biến, kiểm tra tất cả các biến được sử dụng trong câu lệnh gán. Tuy nhiên, với mã SSA - sẽ được nói đến ở mục 2.3.3, hai điều kiện trên sẽ được tự động thỏa mãn và không cần bất kỳ kiểm tra gì thêm.

2.3.2 Loại bỏ mã chết

Mã chết bao gồm các câu lệnh gán mà biến ở vế trái của nó không bao giờ được dùng. Cần phân biệt mã chết (dead code) với mã không bao giờ được chạy (unreachable code), là những câu lệnh mà không có bất kỳ luồng điều khiểu hợp lệ nào của chương trình đi qua (ví dụ: câu lệnh ở dưới một vòng lặp vô hạn). Việc lan truyền biểu thức sẽ dẫn đến việc có một số biến không được sử dụng, từ đó sinh ra mã chết.
Từ đoạn mã đã được lan truyền biểu thức ở hình 2.7, ta thấy biến *esp* không được sử dụng ở bất kỳ câu lệnh nào. Vì vậy, các câu lệnh gán có *esp* ở vế trái sẽ được xem là mã chết và được loại bỏ. Kết quả là hình 2.9.

```
80483b0  2 m[esp0-4] := ebp
80483b3  5 m[esp0-8] := esi
80483b4  7 m[esp0-12] := ebx
80483b5  9 m[esp0-16] := ecx
80483b9 13 edx := m[esp0+4]
```

Hình 2.9: Đoạn mã ở hình 2.8 sau khi loại bỏ mã chết

Để kiểm tra xem một biến có được sử dụng hay không, ta phải xem xét tất cả các luồng chạy hợp lệ của chương trình từ câu lệnh gán biến đến cuối chương trình, điều này phức tạp và mất nhiều thời gian. Tuy nhiên, mã SSA sẽ giúp việc kiểm tra mã chết dễ dàng hơn.

2.3.3 Mã SSA

Mã SSA là một dạng mã trung gian có tính chất là: Mỗi biến hoặc vùng nhớ được định nghĩa duy nhất một lần trong toàn bộ chương trình.

Để chuyển từ mã RTL sang mã SSA, các biến cần phải được thay đổi tên, thường là sẽ được đánh số thứ tự đằng sau tên biến gốc. Ví dụ, nếu biến *a* xuất hiện ở vế trái của 3 câu lệnh gán, thì sẽ được đánh số lần lượt là *a1*, *a2* và *a3* như ví dụ ở hình 2.11

```
1       a := b + c;
2       m := a;
3       a := e + f;
4       n := a;
5       a := 10;
6       k := a;
```

Hình 2.10: Đoạn mã trung gian với 3 lần định nghĩa biến *a*

```
1       a1 := b0 + c0;
2       m1 := a1;
3       a2 := e0 + f0;
4       n1 := a2;
5       a3 := 10;
6       k1 := a3;
```

Hình 2.11: Đoạn mã ở hình 2.11 đã được chuyển sang dạng mã SSA

Với tính chất của mã SSA, việc lan truyền biểu thức và loại bỏ mã chết sẽ được hiện thực rất dễ dàng.

Đối với lan truyền biểu thức, hai điều kiện đã được tự động thỏa mãn. Điều kiện đầu tiên thỏa mãn vì mỗi biến đều được định nghĩa duy nhất một lần, không có việc có nhiều định nghĩa cho cùng một tên biến (nếu ở mã gốc có việc đó, thì khi chuyển sang mã SSA, biến đó sẽ được đánh số để trở thành những biến khác nhau ở mỗi câu lệnh gán). Điều kiện thứ hai thỏa mãn vì chắc chắn từ câu lệnh gán một biến đến bất kỳ câu lệnh nào sử dụng biến đó, biến sẽ không được định nghĩa lại.

Việc loại bỏ mã chết cũng có thể thực hiện dễ dàng nhờ vào hành động thu thập thông tin về định nghĩa và sử dụng của một biến. Trong quá trình biến đổi từ mã RTL sang SSA, ta có thể xây dựng nên một bảng vị trí câu lệnh gán của một biến và vị trí các câu lệnh sử dụng biến đó. Trải qua các quá trình phân tích, nhất là lan truyền biểu thức, bảng này sẽ được cập nhật lại. Đến cuối cùng, các biến được định nghĩa nhưng không được sử dụng ở bất kỳ câu lệnh nào sẽ được xác định và loại bỏ các câu lệnh gán dư thừa đi.

2.4 Tình hình phát triển trình dịch ngược hiện nay

Hiện nay, có rất nhiều trình dịch ngược đã và đang được phát triển. Hầu hết đều hỗ trợ việc dịch ngược từ mã máy và có thể chia làm hai loại: trình dịch ngược nhận đầu vào là mã máy chạy trên máy ảo (ví dụ: mã máy được dịch từ các chương trình viết bằng Java, C#) và trình dịch ngược nhận đầu vào là mã máy chạy trên máy thật. Loại thứ nhất có số lượng nhiều hơn, lý do có thể vì mã máy ảo còn lưu giữ được khá nhiều thông tin từ chương trình gốc, điển hình như tên biến toàn cục (xem hình 2.12). Vì vậy, bài toán cần giải quyết để xây dựng một trình dịch ngược dạng này không nhiều. Ngược lại, mã máy thật đã bị mất hầu hết các thông tin từ chương trình gốc, nên việc khôi phục thông tin ở các trình dịch ngược từ mã máy thật khá phức tạp. Trong hình 2.13, tên biến ở chương trình gốc đã bị mất đi. Ngoài ra, cấu trúc vòng lặp cũng thay đổi từ *for* sang *while*. Đó là do ở mã máy, cấu trúc vòng lặp ở ngôn ngữ cấp cao đã được dịch thành các câu lệnh kiểm tra điều kiện và jump, nên Boomerang phải sử dụng các thuật toán phân tích luồng điều khiển để tạo ra cấu trúc vòng lặp mới, đôi khi có thể không trùng khớp với cấu trúc vòng lặp ban đầu.

Một số trình dịch ngược phổ biến có thể kể đến là:

- dcc: Là một trình dịch ngược từ mã máy, đây được xem là một trong những trình dịch ngược đầu tiên và vẫn còn được phát triển tới bây giờ.
- ILSpy: Là một trình dịch ngược cho .NET, input là các file assembly được dịch từ chương trình .NET, được phát triển bởi isharpcode. Hiện nay ILSpy vẫn đang được tiếp tục phát triển và thêm các tính năng mới.
- Procyon: Là một trình dịch ngược cho Java. Trước đây lựa chọn hàng đầu để dịch ngược mã Java là JAD (Java decompiler), tuy nhiên hiện nay JAD đã

```
using System;

namespace TestILSpy
{
    internal class Program
    {
        public static string abc = "Hello, World";

        private static void Main(string[] args)
        {
            Console.Write(Program.abc + 9);
            Console.ReadLine();
        }
    }
}
```

Hình 2.12: Một đoạn mã được dịch ngược bởi trình dịch ngược ILSpy. Tên biến static *abc* được giữ nguyên như mã gốc

```
#include<stdio.h>

main()
{
        int a=0;
        for (int i=0; i<10; i++){
                a++;
        }
}
```

```
// address: 0x80483ed
int main(int argc, int argv, int envp) {
    __size32 local0;        // m[esp - 12]
    int local1;             // m[esp - 8]

    local0 = 0;
    local1 = 0;
    while (local1 <= 9) {
        local0++;
        local1++;
    }
    return 0;
}
```

Hình 2.13: Một đoạn mã được dịch bởi Boomerang

ngừng phát triển và mã nguồn không còn mở nữa. Một số trình dịch ngược khác được phát triển và Procyon là một đại diện tiêu biểu.

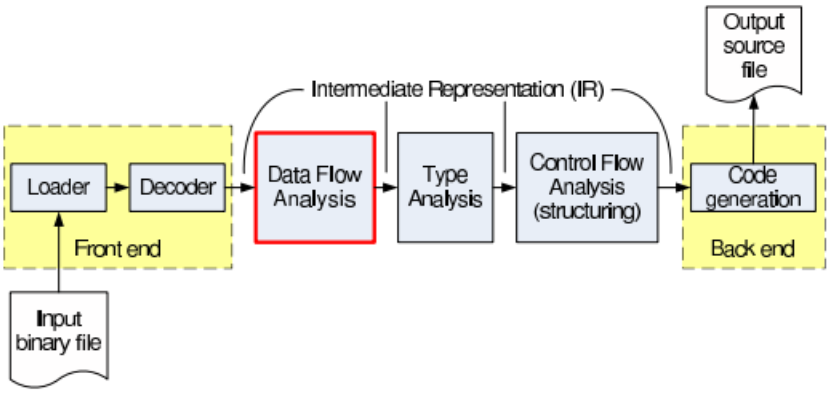
- Boomerang: Là trình dịch ngược từ mã máy, mục tiêu là tạo ra một trình dịch ngược không quan tâm tới ngôn ngữ viết ra chương trình gốc. Boomerang đã ngừng phát triển từ năm 2006 do hai lập trình viên chính bắt đầu làm việc cho một công ty mà lĩnh vực nghiên cứu của họ trùng lặp với Boomerang.

2.5 Cấu trúc của Boomerang

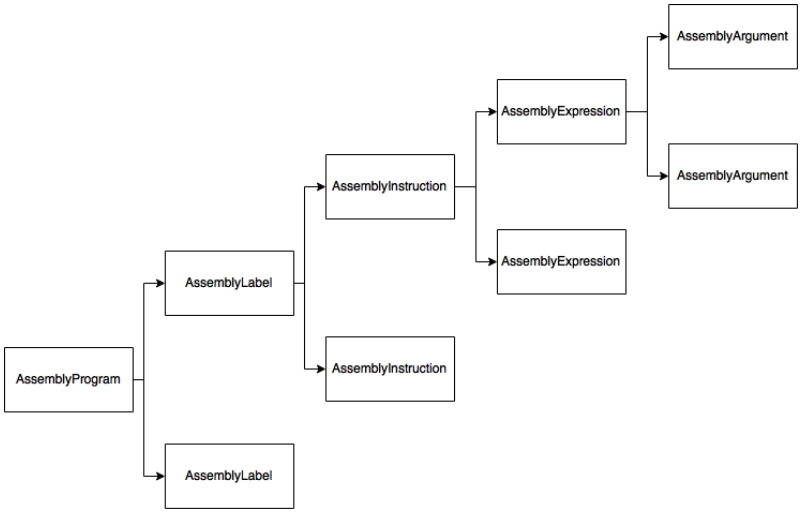
Ở giai đoạn thực tập tốt nghiệp, sau quá trình nghiên cứu một số trình dịch ngược hiện nay, người viết đã quyết định sử dụng trình dịch ngược Boomerang vì sử dụng nhiều kỹ thuật phân tích giúp chất lượng mã đầu ra cao và có cấu tạo kiểu module dễ thay đổi, thêm mới. Phần này sẽ giới thiệu về cấu trúc code của Boomerang, giúp ích cho việc trình bày các giải pháp của bài toán ở chương kế.

Về mặt tổng thể, Boomerang gồm có các phần sau:

Khi đọc vào một chương trình assembly, Boomerang sẽ lưu trữ chúng dưới cấu trúc sau:



Hình 2.14: Cấu trúc các khối lớn của Boomerang



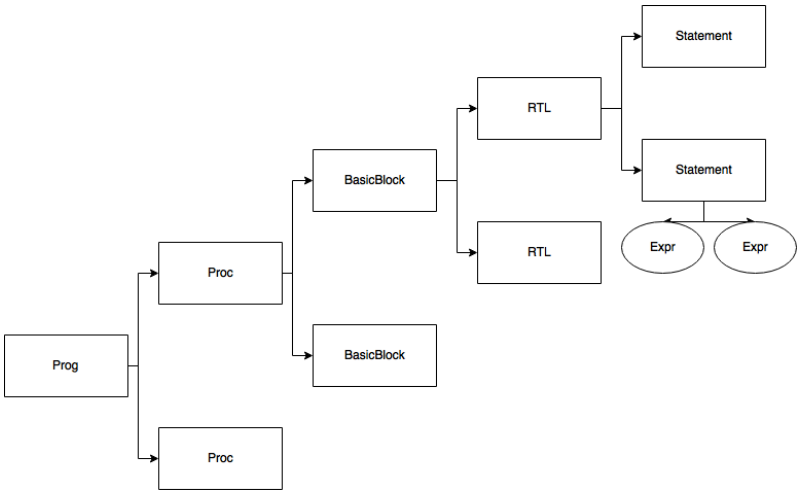
Hình 2.15: Cấu trúc dữ liệu lưu trữ mã assembly trong Boomerang

Lưu ý, trong AssemblyArgument, giá trị thực sự của tham số được lưu vào một union có tên là Arg. Giá trị này có thể là một chuỗi (đối với trường hợp thanh ghi hoặc tên biến), một số nguyên, một số thực hoặc một structure đại diện cho bit (bao gồm tên thanh ghi và vị trí của bit).

Listing 2.1: Đoạn mã mô tả cách biểu diễn giá trị của tham số trong Boomerang

```
struct bits{
        char* reg;
        int pos;
}
union Arg{
        int i;
        float f;
        char* c;
        bits bit;
}
```

Khi giải mã lên ngôn ngữ trung gian, cấu trúc Boomerang dùng để thể hiện là: Prog là tương ứng với toàn bộ chương trình. Một Proc là một hàm, BasicBlock



Hình 2.16: Cách lưu trữ một chương trình dưới dạng mã trung gian của Boomerang

đại diện cho một khối cơ bản mà ở đó không có một câu lệnh rẽ nhánh nào (ví dụ như if, hoặc vòng lặp...). Statement là một câu lệnh và Expr là các biểu thức trong chương trình. Ngoài ra còn có các class đại diện cho kiểu dữ liệu. Việc thực hiện các phân tích chủ yếu diễn ra tại Proc, vì vậy, các thay đổi trong luận văn này cũng chủ yếu được thực hiện bằng các hàm của Proc.

Như vậy, để thực hiện việc xử lý mã 8051, ta cần phải chỉnh sửa các phần sau đây của Boomerang:

- Phần parser để chấp nhận câu lệnh #DEFINE, cũng như lưu trữ biến và giá trị của biến vào một bảng dữ liệu.
- Phần giải mã từ mã assembly lên mã trung gian để đưa các biến không phải thanh ghi thành một biểu thức trung gian phù hợp.
- Phần chuyển đổi từ biểu thức trung gian thành tên biến để giải quyết các trường hợp biến không phải thanh ghi. Hiện tại, Boomerang chỉ cho phép người dùng định nghĩa sẵn một danh sách thanh ghi của kiến trúc máy và dựa vào đó để chuyển đổi. Cần có cơ chế để Boomerang có thể chuyển đổi các biến ngoài danh sách thanh ghi đó.
- Phần phân tích dữ liệu của Boomerang để thêm vào một phân tích hỗ trợ cho việc nhận biết các bộ biến. Mỗi giải pháp sẽ có một phương pháp phân tích khác nhau và được trình bày ở các chương kế.

Chương 3

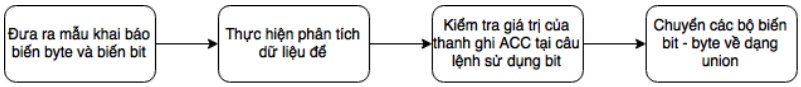
Kiểm tra kiểu - Type checking

Như đã giới thiệu ở phần trước, trước khi thực hiện các giải pháp để tìm ra bộ biến phù hợp với chương trình, ta cần phải chỉnh sửa Boomerang để nó chấp nhận đọc vào đoạn mã có các tên biến không thuộc tập tên thanh ghi, và giữ nguyên tên sau khi giải ra đoạn mã phù hợp. Việc chỉnh sửa này sẽ được trình bày trong phần đầu tiên của chương.

Riêng về giải pháp Kiểm tra kiểu gồm có các bước sau đây:

- Đưa ra mẫu khai báo biến byte và biến bit. Chỉnh sửa chỉnh dịch ngược để đọc vào mẫu khai báo này và biết được những biến byte và biến bit nào cùng một bộ.
- Thực hiện phân tích dữ liệu để biết được giá trị của thanh ghi ACC tại một thời điểm nào đó có phải là giá trị của một vùng nhớ cố định hay không. Tại giai đoạn này của luận văn, kỹ thuật phân tích dữ liệu được sử dụng là Reaching definitions.
- Tại các câu lệnh sử dụng bit, nếu giá trị của thanh ghi ACC là của một vùng nhớ cố định, thì kiểm tra xem giá trị địa chỉ của vùng nhớ đó đang được lưu bởi biến nào. Nếu biến đó được khai báo cùng bộ với biến bit thì xem như tuân thủ đúng nguyên tắc sử dụng, nếu không thì báo lỗi.
- Nếu toàn bộ chương trình đều tuân thủ đúng nguyên tắc sử dụng biến byte - biến bit thì đưa các bộ biến byte - biến bit về dạng union ở ngôn ngữ cấp cao.

Các phần tiếp theo của chương sẽ trình bày lần lượt các bước này.



Hình 3.1: Sơ đồ các bước thực hiện giải pháp Kiểm tra kiểu

3.1 Chỉnh sửa để Boomerang chấp nhận việc sử dụng biến không phải thanh ghi

3.1.1 Cơ chế lưu trữ tên thanh ghi hiện nay của Boomerang

Khi chuyển đổi từ mã assembly sang mã trung gian, Boomerang sẽ dùng một class con của Expr để biểu diễn thanh ghi. Cụ thể là class Location, và gọi phương thức static của class Location là Location::regOf(int num). Ta sẽ truyền vào phương thức này một con số đại diện cho thanh ghi đó. Cặp số - tên thanh ghi này được lưu vào một từ điển, để sau này khi thực hiện phân tích xong thì sẽ chuyển lại từ thanh ghi thành biến cục bộ.

Trong phần giải mã từ mã assembly sang mã trung gian, có một hàm để map giữa tên thanh ghi và con số đại diện cho nó, đó là hàm map_sfr(string name).

Listing 3.1: Một số phần mã trong hàm map_sfr

```
if (name == "R0") return 0;
else if (name == "R1") return 1;
else if (name == "R2") return 2;
...
else return -1;
```

Sau khi trải qua các quá trình phân tích và đến giai đoạn in ra mã đầu ra, trình dịch ngược sẽ gọi hàm getRegName trong class FrontEnd để trả lại tên ban đầu của thanh ghi. Trong hàm getRegName sẽ lấy từ điển tên thanh ghi - số đại diện được quy định sẵn của mỗi phần giải mã cho các kiến trúc máy khác nhau, tìm tên thanh ghi tương ứng với con số đó và trả về.

Listing 3.2: Phần mã trong hàm getRegName

```
std::map<std::string, int, std::less<std::string> >::iterator it;
for (it = decoder->getRTLDict().RegMap.begin(); it != decoder->getRTLDic
if ((*it).second == idx)
return (*it).first.c_str();
return NULL;
```

Như vậy, có thể thấy với các tên biến không được quy định trước, hàm map_sfr sẽ trả về giá trị -1, và vì giá trị -1 sẽ không có trong từ điển của phần giải mã, nên hàm getRegName sẽ trả về NULL, dẫn đến trình dịch ngược sẽ bị lỗi runtime và dừng ngay lập tức.

Vì số lượng tên biến là rất nhiều, nên ta không thể sử dụng phương pháp thêm mới các tên biến vào từ điển được quy định sẵn được, mà phải có cách để trình dịch ngược linh động hơn, chấp nhận bất kỳ các tên nào được sử dụng trong mã assembly. Giải pháp đưa ra là ngoài việc sử dụng từ điển thanh ghi được quy định sẵn, ta sẽ lập thêm một bảng tên biến, thành phần bao gồm các cặp tên biến - số đại diện. Trong giai đoạn giải mã, khi hàm map_sfr được gọi, nếu tên truyền vào nằm trong các thanh ghi đã quy định sẵn, thay vì trả về giá trị -1 thì ta sẽ tạo ra một giá trị random và đưa chúng vào bảng tên biến ở trên. Ngoài ra, còn có một đoạn mã kiểm tra biến được sử dụng đã được khai báo bằng câu lệnh #DEFINE chưa (ngoại trừ một số biến đặc biệt được tự sinh).

Listing 3.3: Phần mã mới được bổ sung trong hàm map_sfr

```
bool isDefined = false;
map<char*, AssemblyArgument*>::iterator it;
for (it = replacement.begin(); it!=replacement.end(); it++){
if(strcmp((*it).first, name.c_str()) == 0 ){
isDefined = true;
break;
}
}
if (isDefined || name.find("specbits") != string::npos ){
if (symbolTable->find(name) == symbolTable->end()){
bool existed = false;
int num;
do{
num = std::rand()%200+31;
map<string, int>::iterator it;
for (it = symbolTable->begin(); it!=symbolTable->end(); it++){
bool cond1 = (*it).second == num;
bool cond2 = (byteVar != -1 && byteVar>=num);
bool cond3 = (bit != -1 && bit>=num);
if (cond1 || cond2 || cond3){
existed = true;
continue;
} else {
existed = false;
}
}
} while (existed);
(*symbolTable)[name] = num;
if (name.find("specbits") != string::npos){
std::cout<<"Name: "<<name<<", "<<num<<endl;
}
return num;
} else {
return symbolTable->find(name)->second;
}
}
else {
std::cout<<"ERROR: "<<name<<" HAS NOT BEEN DEFINED YET"<<endl;
exit(1);
}
```

Tương ứng với sự thay đổi ở hàm map_sfr, ở hàm getRegName, ngoài việc dò trong từ điển quy định trước, ta sẽ thêm một đoạn mã để dò trong bảng tên biến.

Listing 3.4: Phần mã mới được bổ sung trong hàm getRegName

```
std::map<string,int>::iterator symIt;
for (symIt = decoder->getSymbolTable().begin(); symIt != decoder->getSyml
        if ((*symIt).second == idx){
```

```
                return (*symIt).first.c_str();
        }
}
```

Như vậy, vấn đề giữ nguyên tên biến được giải quyết mà không ảnh hưởng nhiều tới trình dịch ngược.

3.2 Mẫu khai báo biến byte và biến bit

Sau khi đã thực hiện phần giữ nguyên tên biến, bước bắt buộc cho cả hai giải pháp, ta sẽ đi vào tìm hiểu bước đầu tiên của giải pháp Kiểm tra kiểu là đưa ra mẫu khai báo. Mẫu khai báo biến byte và biến bit được quy định như sau:

Listing 3.5: Mẫu khai báo bộ biến

```
;BEGIN DEFINE
;BYTE VAR
[byte var declare]
;BIT VAR
[eight bit var declares]
;END DEFINE
```

Đây là mẫu khai báo khi muốn gom các biến vào cùng một bộ, ngoài ra, chương trình vẫn chấp nhận việc khai báo riêng lẻ từng biến. Như vậy, cần chỉnh lại phần parser của trình dịch ngược để chấp nhận cấu trúc mới này. Ngoài ra, trong trình dịch ngược, cần có một cấu trúc mới để lưu trữ thông tin của các bộ biến byte và biến bit này. Trong đoạn mã 3.6, có một trường byteVar để lưu tên biến byte, và một map lưu thứ tự của biến bit và tên biến bit tương ứng. Hiện tại, cấu trúc này đã đủ cho giải pháp Kiểm tra kiểu. Tuy nhiên, nó sẽ được mở rộng ở giải pháp sau.

Listing 3.6: Cấu trúc dữ liệu để lưu trữ một bộ biến

```
class UnionDefine{
        public:
        char* byteVar;
        map<int, char*>* bitVar;
        void prints(){
                cout << "Byte var: " << byteVar <<endl;
                cout << "Bit vars: "<<endl;
                map<int, char*>::iterator mi;
                for (mi = bitVar->begin(); mi != bitVar -> end(); mi++){
                        cout << (*mi).second << ": "
                                << (*mi).first << endl;
                }
        }
};
```

Như vậy, ta cần phải điều chỉnh lại phần parser của trình dịch ngược, sao cho nó vừa chấp nhận mẫu khai báo biến trên, vừa đưa từng bộ biến được khai báo vào trong một thực thể của class UnionDefine. Kết quả là, sau khi kết thúc phần parser, ta sẽ thu được một danh sách các class UnionDefine tương ứng với các khai báo ở trên. Đoạn mã 3.7 là phần parser viết dựa trên thư viện Bison++ để thực hiện chức

năng trên.

Listing 3.7: Đoạn mã parser cho phần khai báo bộ biến

```
definebit: BEGINDEFINE END_LINE DEFINEBYTE END_LINE define DEFINEBITS END
defineeachbit defineeachbit defineeachbit defineeachbit defineeachbit def
{
        UnionDefine* ut = new UnionDefine();
        $5 -> expList -> pop_back();
        ut->byteVar = $5 -> expList -> back() -> argList.back()->value.c;
        ut->bitVar = bitVar;
        unionDefine1 -> push_back(ut);
        bitVar = new map<char*, int>();
}
;
defineeachbit: DEFINE ID bit END_LINE {
        std::string temp($3->value.c);
        char c =  temp.at(temp.size()-1);
        int num = c - '0';
        (*bitVar)[$2] = num;
}
;
```

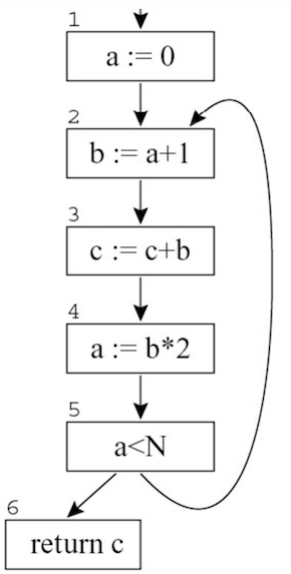
Như vậy, sau giai đoạn parser, ta đã thu được bộ biến theo khai báo của người dùng. Tuy nhiên, nguyên tắc sử dụng bộ biến này là không bắt buộc trong quá trình lập trình 8051, và các assembler cũng không hề kiểm tra việc người dùng có tuân thủ nguyên tắc này không. Vì vậy, trước khi chuyển hoá các bộ biến này sang cấu trúc union ở mã đầu ra, ta cần phải kiểm tra đoạn mã của người dùng sử dụng các bộ biến như thế nào.

3.3 Phân tích Reaching definitions

Mục đích của phân tích Reaching definitions là biết được ở một thời điểm của chương trình, các câu lệnh khai báo nào đang còn hiệu lực, hay nói cách khác là gía trị của các biến đang được khai báo bởi những câu lệnh nào. Ví dụ như ở đoạn chương trình 3.2, ta cần biết được giá trị của biến a ở câu lệnh số 2 được khai báo ở câu lệnh này. Đối với con người thì rất dễ dàng biết được là biến a được sử dụng có thể khai báo ở câu lệnh số 1 hoặc câu lệnh số 5. Tuy nhiên, cần có một phương pháp phân tích để cho máy tính cũng biết được điều đó, và đó chính là phương pháp Reaching definitions. Như vậy, khi áp dụng vào trình dịch ngược, ta sẽ biết được tại thời điểm sử dụng biến bit, giá trị của thanh ghi ACC đang được định nghĩa ở câu lệnh nào. Từ đó tiến hành các bước kiểm tra tiếp theo.

Để thực hiện Reaching definitions, ta phải làm quen với các định nghĩa sau:

- Nếu một biến được khai báo ở câu lệnh def1, sau đó được khai báo lại ở câu lệnh def2 sau đó, thì ta nói là câu lệnh def1 đã bị giết (killed) bởi câu lệnh def2.



Hình 3.2: Một đoạn chương trình mẫu

- Nếu có một đường thực thi chương trình đi từ câu lệnh khai báo def1 đến một điểm p của chương trình, mà trên đó def1 không bị giết bởi bất kỳ câu lệnh nào, thì ta nói là def1 đã đến được điểm p. Khái niệm một câu lệnh đến được một khối cơ bản cũng tương tự như vậy.

Ngoài ra, ta phải quy định một số khái niệm mới cho một khối cơ bản B:

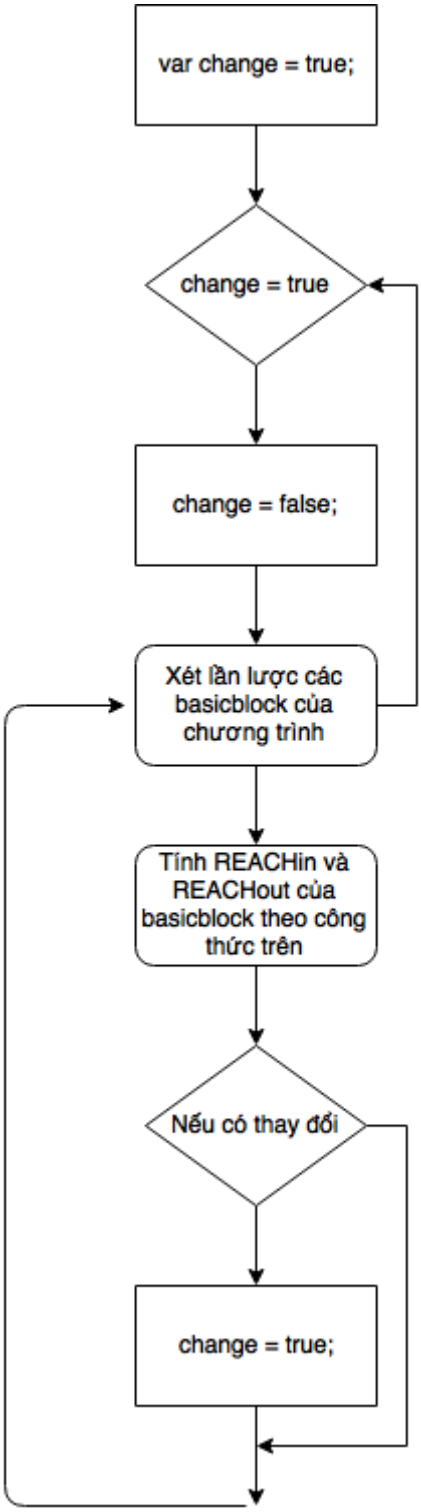
- REACHin(B): Tập hợp các câu lệnh khai báo đến được đầu vào (entry) của B.
- REACHout(B): Tập hợp các câu lệnh khai báo đến được ngõ ra (exit) của B.
- GEN(B): Tập hợp các câu lệnh khai báo xuất hiện trong B và có thể đến được ngõ ra (exit) của B, nghĩa là biến được khai báo trong câu lệnh đó không được khai báo lại ở các câu lệnh đằng sau nó.
- KILL(B): Tập hợp các câu lệnh khai báo mà biến được khai báo đã được khai báo lại trong B.

Như vậy, mục tiêu của phân tích Reaching definitions ở cấp độ khối cơ bản là tìm ra được tập hợp REACHin và REACHout của từng khối. Công thức được áp dụng là;

$$REACHout(B) = GEN(B) \cup (REACHin(B) - KILL(B)) \tag{3.1}$$

$$REACHin(B) = \cup_{j \in Pred(B)} REACHout(j) \tag{3.2}$$

Từ hai công thức 3.1 và 3.2, ta thấy ở phân tích này, tập hợp các giá trị ra (REACHin) được quyết định bởi các giá trị vào (REACHout), ngược lại với phân tích liveness (tìm tập hợp biến đang sống tại một thời điểm của chương trình). Như vậy, luồng đi của phân tích là cùng chiều với luồng đi của chương trình. Theo mẫu thường thấy của một phân tích dữ liệu, ta sẽ lần lượt tính toán các tập hợp vào và tập hợp ra của từng khối cơ bản cho đến khi không còn thay đổi nào được ghi nhận. Xem sơ đồ khối bên dưới.



Hình 3.3: Giải thuật tính Reaching definitions cho khối cơ bản

Tuy nhiên, trong trường hợp của bài toán cần giải quyết, ta cần phải biết tập hợp ra vào của từng câu lệnh một, chứ không chỉ của toàn bộ khối cơ bản, vì vậy, khi ứng dụng vào Boomerang, giải thuật sẽ được điều chỉnh lại để tìm tập REACHin và REACHout của từng câu lệnh. Việc điều chỉnh này là khá nhỏ, và các bước vẫn sẽ giữ nguyên, không thay đổi nhiều.

Tuy nhiên, phân tích Reaching definitions chỉ cho biết được câu lệnh khai báo có hiệu lực của một biến tại một thời điểm chương trình, chứ không cho biết được giá trị thực sự của biến đó. Nếu vế phải của câu lệnh khai báo này chỉ đơn giản là trỏ đến một vùng nhớ có địa chỉ được quy định bởi một biến byte, thì ta sẽ dễ dàng kiểm tra được. Nhưng ngoài ra, biểu thức quy định địa chỉ vùng nhớ được gán cho ACC có thể là các trường hợp sau đây:

- Một hằng số.
- Một thanh ghi, giá trị của thanh ghi có thể được khai báo ở các câu lệnh trước đó.
- Một biểu thức có hai vế, các vế của biểu thức có thể là một biến, một thanh ghi hoặc một hằng số.

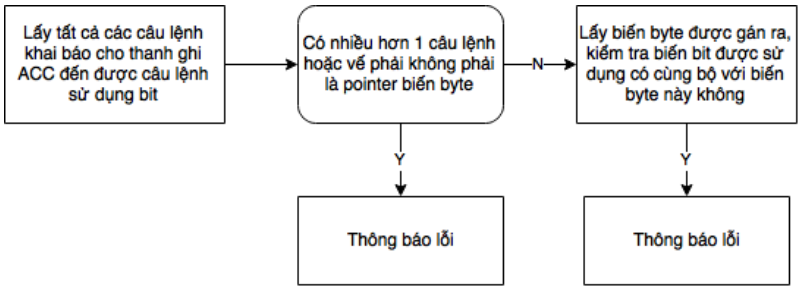
Listing 3.8: Một số câu lệnh gán mà phương pháp Suy luận kiểu sử dụng Reaching definitions không xử lý được

```
MOV A, 38H
MOV A, @DPTR
MOV A, OPTION+1
```

Với phương pháp Reaching definitions, ta sẽ không thể xử lý được với các trường hợp phức tạp hơn như các câu lệnh trong đoạn mã 3.8. Ta sẽ không biết được biến byte nào đã được khai báo giá trị 38H, hoặc thanh ghi DPTR đã được gán giá trị gì trước đó, hoặc OPTION+1 có bằng giá trị của một biến byte nào khác không... Ngoài ra, với trường hợp trong tập hợp REACHin của câu lệnh có nhiều câu lệnh khai báo cho thanh ghi ACC, ta sẽ không thể kiểm tra được vế phải của tất cả các câu lệnh khai báo đó có cùng một giá trị hay không mà chỉ đơn giản xử lý là câu lệnh đã vi phạm nguyên tắc sử dụng bộ biến. Ví dụ như đoạn mã 3.9, tại thời điểm câu lệnh sử dụng biến bit TESTSUPS, có hai câu lệnh khai báo biến a (là biến đại diện cho thanh ghi ACC tại ngôn ngữ trung gian). Đối với phương pháp Reaching definitions, nó sẽ xem như có hai giá trị mà biến a có thể mang, và sẽ báo lỗi vì vi phạm nguyên tắc sử dụng bộ biến. Tuy nhiên, nếu xét kỹ hơn, thì sẽ thấy là cả hai giá trị đó đều là biến OPTIONS, và thực chất tại thời điểm này a chỉ mang một giá trị, cho dù luồng đi của chương trình có như thế nào. Như vậy, phương pháp Reaching definitions sẽ bỏ qua những trường hợp như thế này và báo lỗi, dẫn đến việc độ chính xác sẽ không được cao.

Listing 3.9: Đoạn mã có nhiều câu lệnh khai báo cho ACC đến được một điểm của chương trình nhưng tất cả đều cùng giá trị

```
if (...){
        a = *(OPTIONS);
        ...
} else {
```



Hình 3.4: Quá trình kiểm tra một câu lệnh sử dụng bit

```
        a = *(OPTIONS);
        ...
}
TESTSUPS = 1;
```

Một phân tích khác sẽ được áp dụng trong giai đoạn tiến hành giải pháp Suy luận kiểu. Còn với giai đoạn sơ khai này của luận văn, ta sẽ giả thuyết rằng ở đoạn mã đầu vào chỉ có các câu lệnh gán vùng nhớ đơn giản cho ACC với một biến byte duy nhất như đoạn mã 3.10.

Listing 3.10: Mẫu câu lệnh gán cho thanh ghi ACC được chấp nhận hiện giờ

```
MOV A, OPTIONS
```

3.4 Kiểm tra cách sử dụng bộ biến trong toàn bộ chương trình và sinh mã

Sau khi đã có thông tin về tập hợp các câu lệnh khai báo có hiệu lực ở tất cả các câu lệnh của chương trình, ta sẽ chuyển sang kiểm tra các ở các câu lệnh sử dụng bit.

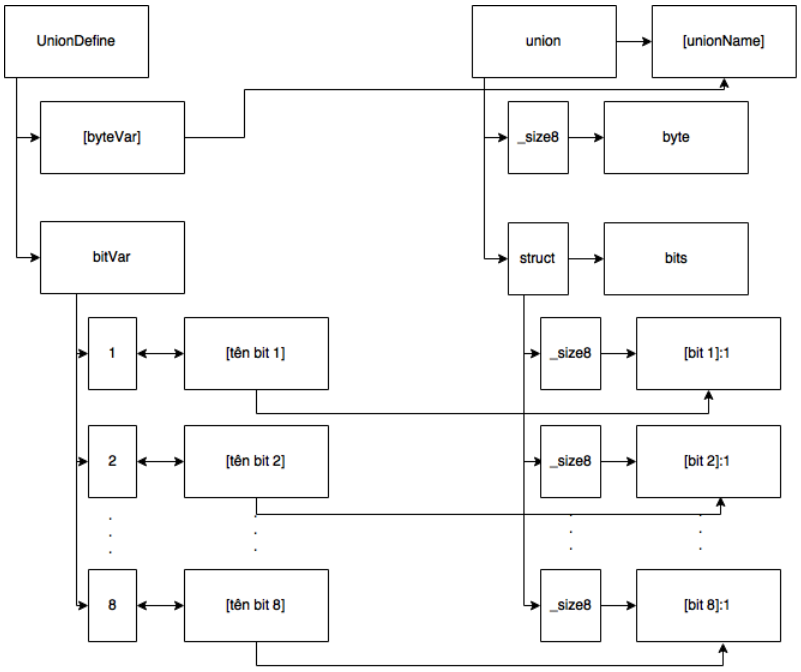
Quá trình kiểm tra cho 1 câu lệnh sử dụng bit gồm các bước sau:

Nếu ở bất cứ câu lệnh sử dụng biến bit nào, nguyên tắc đưa ra bị vi phạm, thì trình dịch ngược sẽ hiện thông báo lỗi trên console cho người dùng. Sau đó, đoạn mã kiểm tra lỗi vẫn tiếp tục chạy để kiểm tra các câu lệnh khác nhằm tạo thuận lợi cho người dùng trong quá trình sửa lỗi chương trình đầu vào. Tuy nhiên, nếu có lỗi thì trình dịch ngược sẽ không sinh mã đầu ra vì sẽ không thể xử lý được sinh các union như thế nào.

Kết thúc quá trình kiểm tra lỗi, nếu không có vi phạm nào xảy ra, trình dịch ngược sẽ tiến hành sinh ra thêm một số khai báo cho bộ biến, cũng như thực hiện các thay thế cần thiết. Các bộ biến được khai báo được lưu bằng cấu trúc UnionDefine, và ta cần thể hiện các bộ biến đó bằng một cấu trúc dữ liệu ở ngôn ngữ cấp cao của mã đầu ra. Như đã phân tích từ trước ở chương 1, cấu trúc dữ liệu đó là union. Hình 3.5 thể hiện việc chuyển đổi của class UnionDefine sang cấu trúc union.

Sau khi sinh ra các union như trên, ta cần tiến hành thực hiện các thay thế sau:

- Thay thế các biểu thức thể hiện biến bit dưới dạng thanh ghi độc lập thành một biểu thức truy xuất đến một thành phần của union tương ứng với bộ biến mà biến bit đó thuộc về. Vì ở giai đoạn giải mã, ta không thực hiện kiểm tra



Hình 3.5: Hình minh hoạ việc chuyển đổi từ class UnionDefine sang cấu trúc union ở mã đầu ra

biến nào là biến bit, biến nào là biến byte, nên ta sẽ xem tất cả các biến như những thanh ghi độc lập nhau. Sau khi đã trải qua các quá trình phân tích và sinh ra các union, ta cần thay thế để thể hiện rõ mối liên hệ giữa biến bit và biến byte.

- Thay thế các biểu thức truy xuất trực tiếp bit của thanh ghi ACC thành biểu thức truy xuất đến một thành phần của union tương ứng với bộ biến chứa biến byte mà thanh ghi ACC đang mang giá trị trỏ đến. Vì một số lý do, có đôi khi người lập trình viên không sử dụng biến bit mà sử dụng một truy xuất trực tiếp đến bit trong thanh ghi ACC, ví dụ như: ACC.1. Với trường hợp này, vì ta đã có trong tay bộ biến và giá trị của biến byte thanh ghi ACC đang được trỏ đến, ta sẽ thay thế để đoạn mã đầu ra thống nhất hơn, và có thể tiến hành bước thay thế thanh ghi ACC được trình bày bên dưới. Lưu ý: trong giai đoạn giải mã, ghi gặp biểu thức ACC.x, trình giải mã sẽ chuyển chúng về một thanh ghi đặc biệt có tên là specbitsx, với x là số thứ tự của bit muốn truy xuất (xem ví dụ đoạn mã 3.11).

Listing 3.11: Mã đầu ra trước khi thực hiện bước thay thế biến bit

```
TESTSUPS = 1;
if (specbits5 == 1){
        ...
}
```

Listing 3.12: Mã đầu ra sau khi thực hiện bước thay thế biến bit

```
OPTIONS.bits.TESTSUPS = 1;
if (OPTIONS.bits.bit1 == 1){
        ...
```

```
}
```

- Thay thế các vị trí sử dụng thanh ghi ACC bằng biến byte tương ứng. Khi lập trình ở dạng mã assembly, lập trình viên không được phép xử lý các vùng nhớ trực tiếp mà phải thông qua thanh ghi, tuy nhiên, khi đã chuyển đổi về dạng ngôn ngữ cấp cao, ta có thể sử dụng trực tiếp tên biến trong các câu lệnh mà không cần trung gian qua thanh ghi nữa. Điều này giúp mã đầu ra dễ hiểu và trong sáng hơn.

Listing 3.13: Mã đầu ra trước khi thực hiện bước thay thế thanh ghi ACC

```
a = *OPTIONS;
OPTIONS.bits.TESTSUPS1 = 1;
return a;
```

Listing 3.14: Mã đầu ra sau khi thực hiện bước thay thế thanh ghi ACC

```
OPTIONS.bits.TESTUPS1 = 1;
return OPTIONS.byte;
```

Như vậy, với giải pháp Kiểm tra kiểu, ta đã có sẵn thông tin về bộ biến ngay từ đầu và chỉ cần kiểm tra xem người lập trình có tuân thủ đúng quy tắc không trước khi sinh ra mã ở ngôn ngữ cấp cao. Giải pháp này yêu cầu can thiệp vào trình dịch ngược ít và hiện thực dễ dàng. Tuy nhiên, giải pháp còn nhiều hạn chế như phương pháp phân tích dữ liệu chưa đạt độ chính xác cao, cần người dùng phải chỉnh sửa lại khai báo theo mẫu quy định... Chính vì vậy, giai đoạn sau của luận văn đã phát triển một giải pháp mới có độ chính xác cao hơn và không cần chỉnh sửa mã đầu vào của người dùng, đó là giải pháp Suy luận kiểu. Giải pháp này sẽ được trình bày ở chương tiếp theo.

Chương 4

Suy luận kiểu - Type inference

Sau khi đã hiện thực giải pháp Kiểm tra kiểu, nhận thấy rằng giải pháp này còn nhiều hạn chế và có thể cải tiến thêm, luận văn đã phát triển thêm một giải pháp mới tốt hơn, không bắt buộc người dùng phải thay đổi code của mình theo mẫu khai báo, đó là Suy luận kiểu. Với giải pháp này, bằng các phép phân tích dữ liệu, trình dịch ngược sẽ tự động tìm ra được các bộ biến được sử dụng trong chương trình. Ngoài bước chung của hai giải pháp là chỉnh sử trình dịch ngược để giữ nguyên tên biến đã được trình bày ở giải pháp trước, các bước của giải pháp Suy luận kiểu gồm có:

1. Dùng một phương pháp phân tích luồng dữ liệu để biết được giá trị của thanh ghi ACC tại mỗi điểm của chương trình.

2. Đi qua các câu lệnh sử dụng biến bit, ghi nhận giá trị hiện tại của thanh ghi ACc tại câu lệnh đó và đưa biến bit đó vào bộ biến phù hợp.

3. Thêm vào các union tương ứng với bộ biến tìm ra được, thay thế các thanh ghi đại diện cho biến bit bằng truy xuất đến union tương ứng, cũng như thay thế vị trí sử dụng thanh ghi ACC.

Như vậy, giải pháp này đã bỏ qua được bước đầu tiên, quy định mẫu khai báo, của giải pháp Suy luận kiểu. Ngoài ra, kỹ thuật phân tích Reaching definitions như đã trình bày ở chương trước có một số khuyết điểm, vì vậy, ở giải pháp này, chúng ta sẽ phân tích và tìm ra một kỹ thuật khác toàn diện hơn. Điều này sẽ được trình bày ở phần đầu tiên của chương, phần tiếp theo sẽ nói về cách quét các câu lệnh sử dụng biến bit và đưa thông tin vào một cấu trúc dữ liệu phù hợp.

4.1 Phân tích Constant propagation

Ở chương trước, phương phán phân tích Reaching definitions đã được đề cập đến, tuy nhiên, phương pháp này chỉ áp dụng được cho trường hợp gán một biến byte trực tiếp cho thanh ghi ACC, vì vậy ta cần tìm một phương pháp khác phù hợp hơn. Phương pháp đạt yêu cầu cần phải tính toán được chính xác giá trị hiện có của thanh ghi ACC cho dù biểu thức bên phải của phép gán là gì. Ngoài ra, nếu thanh ghi ACC có thể có mang những giá trị khác nhau ở một câu lệnh sử dụng bit, thì mặc nhiên nguyên tắc bị vi phạm. Như vậy, ta chỉ xét tới các trường hợp giá trị ở thanh ghi ACC là một giá trị cố định, có thể tính toán được trước khi thực thi

chương trình. Từ các yêu cầu trên, ta kết luận được phương pháp phân tích phù hợp nhất trong trường hợp này là Lan truyền hằng số - Constant propagation. Phương pháp này cho phép tính toán giá trị của các biến, cho biết được gía trị đó có phải là một hằng số tại một thời điểm của chương trình hay không. Ví dụ như đoạn mã ban đầu 4.1, có thể rõ ràng thấy giá trị của biến x là 14, nhưng ta không biết được giá trị thực sự của biến y, cũng như biểu thức trả về là bao nhiêu. Nhờ vào việc lan truyền hằng số, các giá trị này sẽ được tính toán, như trong đoạn mã 4.2 và 4.3.

Listing 4.1: Đoạn mã trước khi thực hiện lan truyền hằng số

```
 int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Listing 4.2: Đoạn mã sau khi thực hiện lan truyền hằng số cho biến y

```
int x = 14;
int y = 0;
return y * (28 / x + 2);
```

Listing 4.3: Đoạn mã sau khi thực hiện lan truyền hằng số cho biểu thức trả về

```
int x = 14;
int y = 0;
return 0;
```

Với phương pháp này, một biến có thể có ba giá trị sau:

- Top: Nghĩa là chưa biết được biến có giá trị gì.
- Hằng số: Nghĩa là đã xác định được giá trị của biến là một hằng số.
- Bottom: Nghĩa là biến có thể mang những giá trị khác nhau, tuỳ thuộc vào luồng chạy của chương trình.

Ở bước khai báo ban đầu của giải thuật, tất cả các biến đều được truyền vào giá trị top (chưa biết), sau đó, trải qua quá trình phân tích thì giá trị của một biến có thể được xác định là hằng số (như giá trị của biến a tại câu lệnh số 3, đoạn mã 4.4) hoặc là bottom (như giá trị biến a tại câu lệnh số 9, đoạn mã 4.6).

Listing 4.4: Đoạn mã ví dụ biến có giá trị là hằng số

```
int a;
a = 4;
b = a*4;
```

Listing 4.5: Đoạn mã ví dụ biến có giá trị là bottom

```
int a;
int b;
cout<<"Enter b: ";
cin >> b;
if (b>15)
        a = 4;
else
```

```
        a = 5;
return a;
```

Như vậy, khi áp dụng vào trình dịch ngược Boomerang, mục tiêu của giải thuật này là để tìm ra được ở mỗi điểm của chương trình, giá trị thật sự của địa chỉ vùng nhớ mà thanh ghi ACC đang mang là gì.

Có nhiều cách thực hiện Constant propagation, vì trong Boomerang, có một giai đoạn code trung gian được giữ ở dạng SSA, nên ta sẽ chọn cách phân tích Sparse constant propagation để giảm thiểu thời gian xử lý. Và việc phân tích này sẽ được thực hiện ở cuối giai đoạn SSA, khi các phân tích khác đã hoàn tất. Giải thuật của phân tích Sparse constant propagation gồm có các bước được trình bày ở hình 4.1

Để thể hiện giá trị của một biến có thể thuộc ba loại là top, hằng số hoặc bottom, ta sẽ tạo ra một class mới trong Boomerang, đó là ConstantVariable. Đoạn mã của class này được trình bày bên dưới.

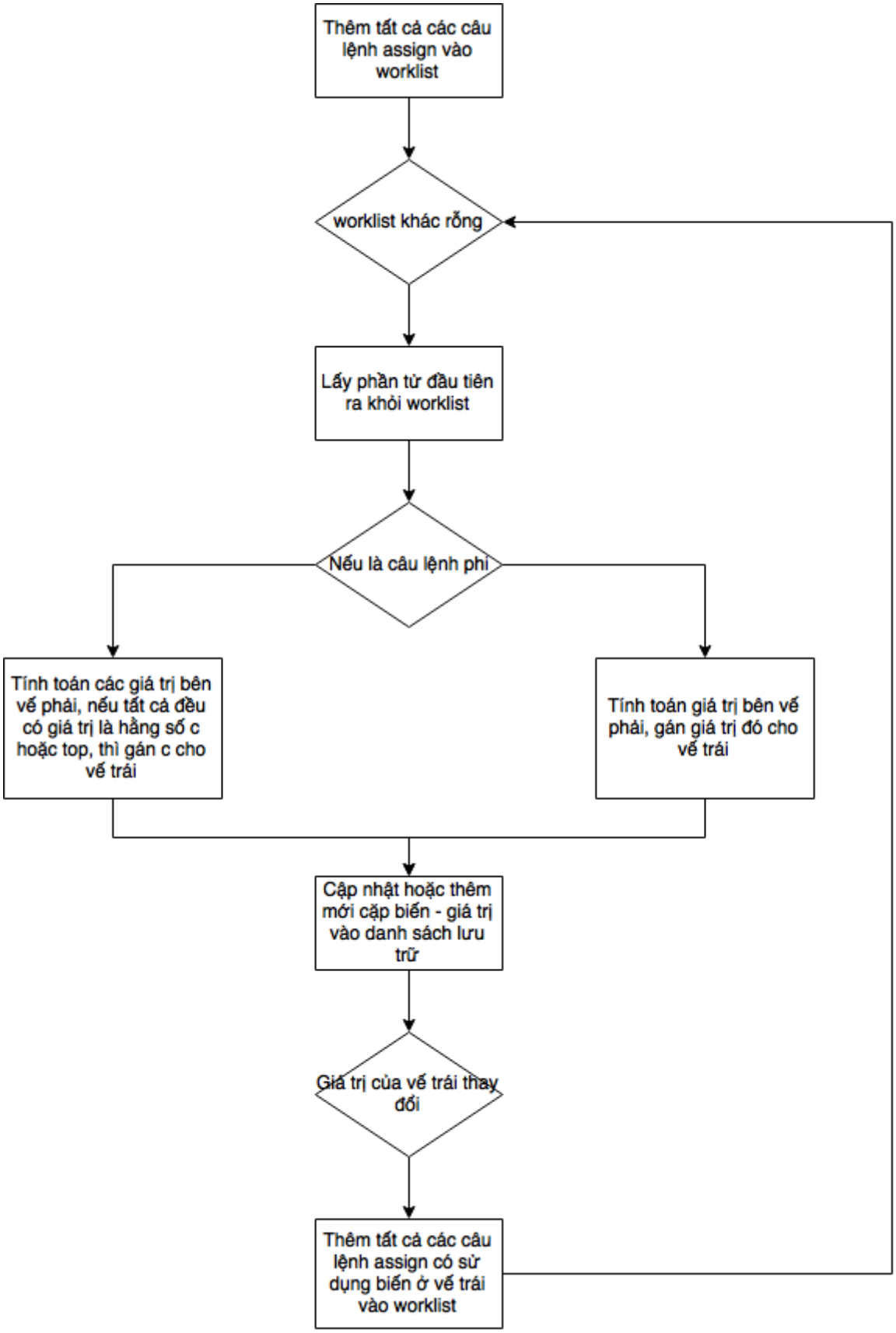
Listing 4.6: Đoạn mã thể hiện class ConstantVariable

```
class ConstantVariable{
        public:
        int type; //1: top, 2: constant, 3: bottom
        Exp* variable;
        ConstantVariable(){
                type = 3;
        }
};
```

Như vậy, ta có thể thấy kết quả của giải thuật này là tạo ra được một sơ đồ map giữa một biến SSA và một thực thể ConstantVariable thể hiện giá trị của biến đó.

Ngoài ra, ta sẽ áp dụng code pattern Visitor để tính toán được giá trị của các biểu thức nằm ở vế phải của lệnh gán. Visitor đó được đặt tên là EvalExpressionVisitor. Vì các biểu thức có thể viết ở mức assembly khá đơn giản, nên ta chỉ cần viết hàm visit cho các loại biểu thức sau:

- Const: Là biểu thức hằng số. Hàm visit này chỉ đơn giản trả về giá trị hằng số nếu đây là một hằng số nguyên.
- Binary: Là biểu thức có 2 vế. Hàm visit sẽ visit từng vế của biểu thức, và nếu cả hai vế đều là hằng số, thì sẽ thực hiện phép tính cộng trừ nhân chia hai hằng số đó để ra được kết quả cuối cùng.
- RefExp: Loại biểu thức này chứa một biểu thức khác, kèm theo câu lệnh khai báo biểu thức đó. Cụ thể, ta chỉ xét loại RefExp có biểu thức con là một biến. Ta sẽ tìm trong sơ đồ hiện tại để lấy ra giá trị của biến đó. Nếu không có trong sơ đồ, ta sẽ tìm trong bảng lưu trữ dữ liệu các câu lệnh #DEFINE để xem đó có phải là một biến đã được khai báo trước trong chương trình đầu vào không.
- TypedExp: Loại biểu thức để ép kiểu một biểu thức nào đó thành kiểu mong muốn. Với trường hợp này, ta sẽ visit biểu thức con và trả về giá trị của biểu thức đó.



Hình 4.1: Giải thuật Constant propagation đã được điều chỉnh phù hợp với yêu cầu của trình dịch ngược

Như vậy, với phương pháp phân tích này, ta có thể giải quyết được vấn đề về phải của phép gán thanh ghi ACC không chỉ đơn giản là một biến byte. Ngoài ra, nó còn nhận biết được các biểu thức có giá trị giống nhau mặc dù hình thức bên ngoài khác nhau. Xem ví dụ các câu lệnh ở đoạn mã 4.7. Câu lệnh gán số 1 và số 2 thực chất đều gán cho ACC giá trị vùng nhớ có địa chỉ quy định bởi biến OPTIONS. Nếu thực hiện phân tích Reaching definitions ở giải pháp trước, trình dịch ngược sẽ không thể biết được điều này. Tuy nhiên, ở giai đoạn này, vì trình dịch ngược sẽ tính toán được ở cả hai câu lệnh, ACC đều mang giá trị của vùng nhớ có địa chỉ là 38H. Và ở những bước tiếp theo, trình dịch ngược sẽ đối chiếu giá trị 38H với bảng lưu trữ dữ liệu và biết được biến OPTIONS đại diện cho giá trị đó.

Listing 4.7: Một số câu lệnh gán cho ACC có giá trị vế phải bằng nhau

```
#DEFINE OPTIONS #38H
...
MOV ACC, OPTIONS
MOV ACC, 38H
```

Một lưu ý là trong trường hợp này, ta không xét đến giá trị của thanh ghi ACC, mà ta chỉ xét đến giá trị của địa chỉ vùng nhớ thanh ghi ACC đang lưu giữ. Như vậy, chỉ có các câu lệnh dạng MOV ACC, [biểu thức] sẽ được xét đến. Khi gặp câu lệnh gán có dạng MOV ACC, #[biểu thức] thì đoạn mã phân tích sẽ xem như giá trị của ACC không phải là hằng số (bottom). Như vậy, với các biến khác, giá trị lưu trong thực thể ConstantExpression tương ứng với biến đó là giá trị thực sự của biến, còn riêng với thanh ghi ACC, giá trị đó được hiểu là giá trị địa chỉ vùng nhớ mà thanh ghi ACC được load vào.

4.2 Quét các câu lệnh sử dụng biến bit

Sau khi đã biết được giá trị địa chỉ vùng nhớ mà thanh ghi ACC đang nắm giữ, ta sẽ chuyển sang bước quét các câu lệnh sử dụng biến bit. Quá trình thực hiện bước này là khá giống nhau giữa hai giải pháp. Tuy nhiên, điểm khác biệt là nếu ở giải pháp Kiểm tra kiểu, do thực hiện phân tích Reaching definitions, nên ta sẽ biết chính xác được là tại câu lệnh sử dụng biến bit, thanh ghi ACC giữ giá trị của vùng nhớ có địa chỉ quy định bởi biến byte nào, còn với giải pháp Suy luận kiểu, do sử dụng phân tích Constant propagation, ta chỉ biết chính xác giá trị địa chỉ vùng nhớ thanh ghi ACC đang được load vào, mà không biết biến byte nào đại diện cho giá trị đó. Để biết được cụ thể biến byte nào tương ứng, ta sẽ phải dò trên bảng lưu trữ dữ liệu từ các câu lệnh #DEFINE, và nếu mỗi lần gặp một câu lệnh sử dụng bit đều làm vậy thì tốc độ sẽ không cao.

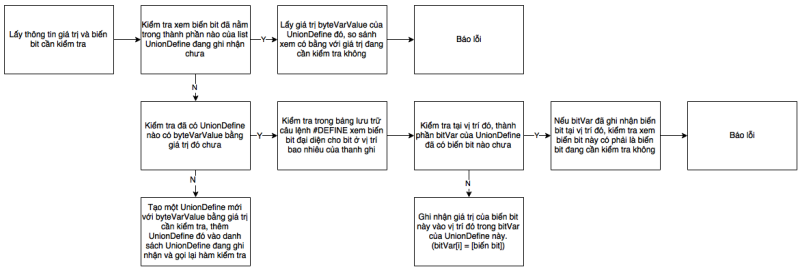
Giải pháp để nâng cao tốc độ xử lý là trong quá trình quét câu lệnh sử dụng bit, ta sẽ không tìm kiếm mối quan hệ giữa biến bit - biến byte, mà chỉ tìm kiếm mối quan hệ giữa biến bit - một giá trị trực tiếp nào đó. Sau khi tìm ra được tất cả các bộ này trong chương trình đầu vào, ta sẽ lần lượt tìm kiếm các biến byte tương ứng với giá trị trực tiếp và thay thế vào. Để thực hiện được điều đó, ta vẫn sẽ sử dụng cấu trúc lưu trữ UnionDefine, nhưng ngoài thành phần byteVar, ta sẽ thêm vào một thành phần mới là byteVarValue. Thành phần này sẽ lưu trữ giá trị trực tiếp ứng với các biến bit lưu ở bitVar. Ở phần cuối của giai đoạn phân tích, ta sẽ

tìm ra được biến byte tương ứng và đưa vào trường byteVar.

Listing 4.8: Đoạn mã mới của class UnionDefine

```
class UnionDefine{
public:
char* byteVar;
map<int, char*>* bitVar;
int byteVarValue;
void prints(){
cout << "Byte var: " << byteVar <<endl;
cout << "Bit vars: "<<endl;
map<int, char*>::iterator mi;
for (mi = bitVar->begin(); mi != bitVar -> end(); mi++){
cout << (*mi).second << ": " << (*mi).first << endl;
} } };
```

Quá trình quét câu lệnh và thu thập dữ liệu của giải pháp Suy luận kiểu cũng tương tự như giải pháp Kiểm tra kiểu, nhưng thay vì ghi nhận biến bit và biến byte rồi kiểm tra chúng có cùng một bộ như khai báo hay không, ta sẽ kiểm tra trước hai biến đó có vi phạm nguyên tắc sử dụng hay không (ví dụ như biến bit đó đã thuộc một bộ khác trước đó, hoặc ở bit vị trí đó của bộ biến của biến byte đã ghi nhận một biến bit khác...), nếu không thì ta sẽ ghi nhận mối quan hệ này vào danh sách UnionDefine đang lưu trữ.



Hình 4.2: Các bước kiểm tra và ghi nhận dữ liệu vào danh sách UnionDefine

Sau khi đã quét hết các câu lệnh ở các procedure, trước khi chuyển đổi các UnionDefine thành các khai báo ở ngôn ngữ C và thêm vào danh sách các biến toàn cục của program như giải pháp trước, ta phải thêm vào một bước chuyển đổi từ giá trị thành biến byte đại diện cho giá trị đó. Điều này có thể được thực hiện bằng cách chạy vòng lặp qua bảng lưu trữ các câu lệnh #DEFINE đã được thiết lập từ quá trình parse mã đầu vào. Nếu như có một giá trị nào đó chưa được khai báo ở câu lệnh #DEFINE, ta sẽ đặt tên mới cho biến byte đó là LOCATION_[giá trị]. Ví dụ như LOCATION_56

Bước cuối cùng của giải pháp này là thay thế các biến bit thành truy xuất tới union tương ứng, loại bỏ câu lệnh gán thanh ghi ACC và thay thế các vị trí sử dụng thanh ghi ACC thành các biến byte tương ứng. Cách thực hiện tương tự như trong giải pháp Suy luận kiểu đã trình bày ở phần 3.4 chương 3

Như vậy, giải pháp này đã giải quyết phần lớn các vấn đề đặt ra của bài toán.

Chương 5

Kiểm thử

Bất kỳ một sản phẩm nào đều cần phải được kiểm tra trước khi công bố, luận văn này cũng không phải là một ngoại lệ. Để đảm bảo chất lượng được đánh giá một các khách quan nhất, một hệ thống testcase với các loại tình huống được phân bổ một cách khoa học nhất sẽ được đưa ra, sau đó cho chạy thử qua cả 2 giải pháp của luận văn. Phần một của chương này sẽ trình bày về các testcase đó, phần hai sẽ trình bày cách kiểm tra chúng trên 2 giải pháp, và phần cuối sẽ là kết quả kiểm tra.

5.1 Hệ thống testcase

Có các tiêu chí phân loại testcase như sau:

- Loại biểu thức được gán vào thanh ghi ACC (tiêu chí I)
- Cách truy xuất bit của thanh ghi (tiêu chí II)
- Có vi phạm nguyên tắc sử dụng bộ biến hay không (tiêu chí III)

Với mỗi tiêu chí, ta sẽ có các trường hợp sau đây:
Tiêu chí I:

1. Một giá trị trực tiếp
2. Giá trị ở một vùng nhớ có địa chỉ là một biến byte
3. Giá trị ở một vùng nhớ có địa chỉ là một giá trị trực tiếp
4. Giá trị ở một vùng nhớ có địa chỉ là một thanh ghi
5. Giá trị ở một vùng nhớ có địa chỉ là một biểu thức 2 vế. Mỗi vế có thể là một biến byte, một thanh ghi, hoặc một giá trị trực tiếp

Tiêu chí II:

1. Truy xuất dựa vào một biến bit.
2. Truy xuất bằng cấu trúc truy xuất trực tiếp một bit của thanh ghi. Ví dụ: ACC.5

Tiêu chí III:

1. Không vi phạm nguyên tắc sử dụng. Nghĩa là: mỗi một biến bit chỉ được sử dụng khi thanh ghi ACC đang mang giá trị vùng nhớ có địa chỉ quy định bởi một biến byte duy nhất. Và không có hai biến bit nào cùng vị trí được sử dụng khi thanh ghi ACC đang mang giá trị của một biến byte nào đó.

2. Vi phạm nguyên tắc sử dụng. Một biến bit được sử dụng ở nhiều chỗ, trong các chỗ đó thanh ghi ACC mang giá trị của những biết byte khác nhau.

3. Vi phạm nguyên tắc sử dụng. Một biến bit được sử dụng ở một vị trí, tại vị trí đó thanh ghi ACC có thể mang giá trị của nhiều vùng nhớ khác nhau, không thể xác định trước khi thực thi chương trình.

4. Vi phạm nguyên tắc sử dụng. Một biến byte được sử dụng ở nhiều vị trí, sau câu lệnh gán biến byte, có 2 biến bit cùng một vị trí được sử dụng.

Dựa vào các tiêu chí và trường hợp trên, ta sẽ có tổng cộng 5x2x4 = 40 loại testcase. Ngoài ra, sẽ có một testcase phức tạp được lấy từ một đoạn chương trình thực của doanh nghiệp được đưa vào kiểm thử, nhằm đảm bảo tính thực tế của luận văn này.

5.2 Phương thức kiểm thử

Vì số lượng testcase không quá lớn, và output ra của trình dịch ngược có khá nhiều thông tin khác ngoài phạm vi luận văn, nên ta sẽ dùng cách kiểm tra bằng tay. Hai bộ source code sẽ được đưa vào vòng lặp, chạy từ testcase 1 đến testcase 50. Output ở console sẽ được lưu vào file có định dạng [số thứ tự testcase]console.txt và đoạn mã đầu ra (nếu có) sẽ được lưu vào file có định dạng [số thứ tự testcase]code.txt. Sau đó người viết sẽ trực tiếp kiểm tra hai file này để xác định kết quả có đúng như mong muốn hay không.

5.3 Kết quả chạy thử

Kết quả chạy thử của 2 phương pháp được thể hiện ở bảng dưới.

Như vậy, có thể thấy giải pháp đầu tiên ra kết quả không chính xác rất nhiều, còn giải pháp Suy luận kiểu thì ra được kết quả chấp nhận được. Điều này đã được dự báo trước vì giải pháp Kiểm tra kiểu còn nhiều hạn chế.

Chương 6

Kết luận

Chương này sẽ tổng kết lại các kết quả đã đạt được của luận văn và đưa ra hướng phát triển trong tương lai.

6.1 Kết quả đạt được, khó khăn, điểm hạn chế

Nhìn chung, luận văn đã hoàn thành mục tiêu đề ra ban đầu, giải quyết được bài toán về kiểu dữ liệu bit và câu lệnh xử lý bit trong mã assembly của 8051. Ngoài ra, luận văn đã chứng minh được tính thực tiễn của đề tài, khả năng áp dụng vào thực tế của các doanh nghiệp có nhu cầu dịch ngược. Cuối cùng, luận văn cũng đưa ra một phương pháp lập testcase và kiểm thử khoa học, đảm bảo đưa ra được các trường hợp có thể xảy ra ngoài thực tế và đặc biệt có một testcase là một đoạn mã thật của doanh nghiệp. Kết quả kiểm thử trên bộ testcase dành cho phương pháp Suy luận kiểu là chấp nhận được.

6.2 Hướng phát triển trong tương lai

Các hướng phát triển trong tương lai của trình dịch ngược gồm có:

- Tiếp tục mở rộng khả năng dịch ngược cho nhiều máy khác nhau
- Phân tích và sửa lỗi sai của giải thuật phân tích dòng dữ liệu
- Cải tiến chức năng nhận dạng kiểu của Boomerang

Tài liệu tham khảo