TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

BỘ MÔN MẠNG MÁY TÍNH
VÀ TRUYỀN THÔNG DỮ LIỆU

Trần Trung Tín

HƯỚNG DẪN THỰC HÀNH TCMT
HỌC PHẦN: HỢP NGỮ MIPS

TP. HỒ CHÍ MINH
2023

Mục lục

Bài thực hành LAB 1

GIỚI THIỆU MIPS

"Everybody should learn to program a computer, because it teaches you how to think." - Elon Musk, the CEO of Tesla Inc.

Ngôn ngữ mức cao đóng vai trò quan trọng trong việc phát triển phần mềm khi cung cấp các phát biểu rất gần với cấu trúc ngôn ngữ của con người, điều mà thúc đẩy lập trình viên sáng tạo và nhanh chóng thể hiện ý tưởng của họ. Về phía máy tính, bộ vi xử lý cần giải mã và thực thi từ lệnh một bằng cách mạch điện mà nó được trang bị, chẳn hạn như các phép tính số học, lượng giá các biểu thức logic, thực hiện các cấu trúc điều khiển chương trình và truy cập bộ nhớ chính.

Trong lập trình máy tính, Hợp ngữ (assembly) thường được viết tắt là asm là bất kỳ ngôn ngữ lập trình cấp thấp nào có sự tương ứng rất mạnh giữa các tập lệnh trong ngôn ngữ và tập lệnh mã máy của kiến trúc. Bởi vì hợp ngữ phụ thuộc vào tập lệnh mã máy, mỗi trình biên dịch có hợp ngữ riêng được thiết kế cho chính xác một kiến trúc máy tính cụ thể. Hợp ngữ cũng có thể được gọi là mã máy tượng trưng (symbolic machine code)

1.1 Giới thiệu MARS

MARS là một bộ giả lập hợp ngữ và thực thi, sẽ "lắp ráp" và mô phỏng việc thực thi các chương trình hợp ngữ MIPS. MARS được phân phối dưới dạng tệp JAR và có thể được sử dụng từ dòng lệnh hoặc thông qua môi trường phát triển tích hợp (IDE). MARS được viết bằng Java và yêu cầu ít nhất Bản phát hành 1.5 của Môi trường chạy thi hành Java J2SE (JRE) để hoạt động.

1.1.1 Tải về và chạy phần mềm MARS?

Người dùng Windows và Mac:

- Download and install JRE through below link: `https://www.java.com/en/download/manual.jsp`

- Download MARS from following link: `hhttp://courses.missouristate.edu/kenvollmar/mars/download.htm`

1.1.2 Chương trình đầu tiên

Yêu cầu 1. Sử dụng đoạn mã được cung cấp tại 1.1 và chương trình MARS MIPS để in ra màn hình lời chào Hello World.

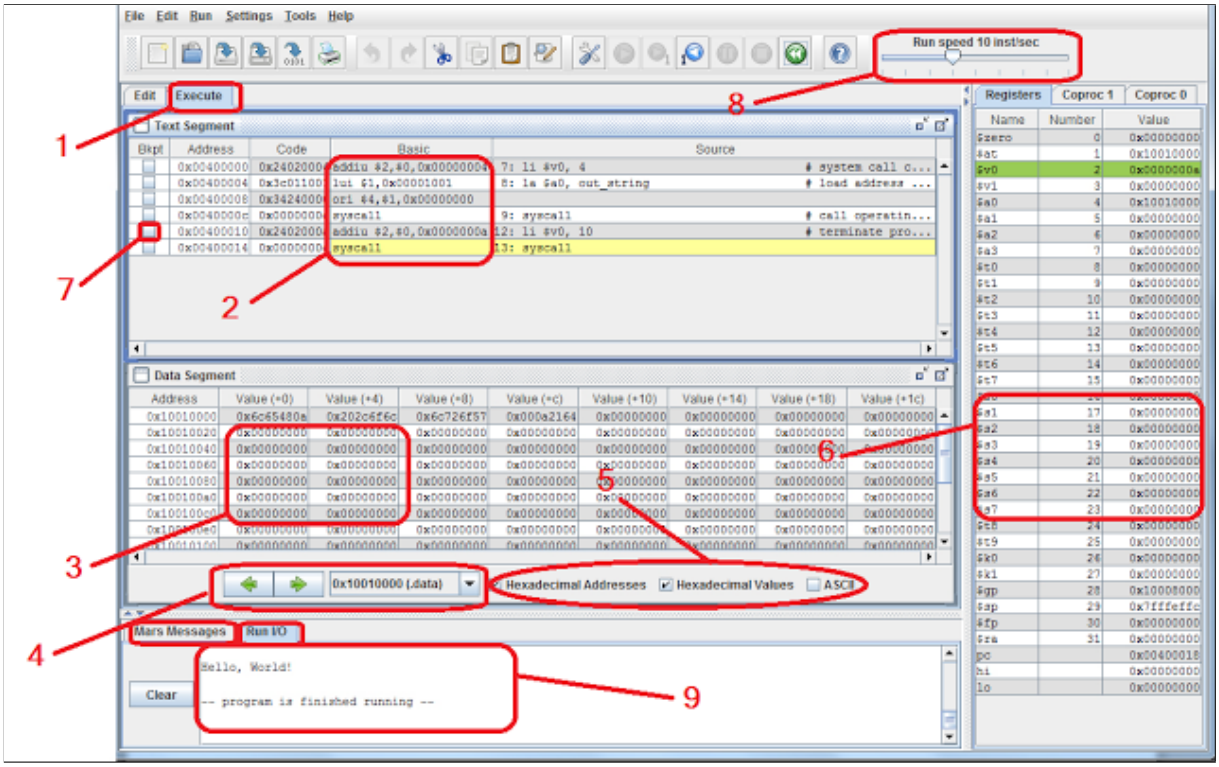
1. **Biên tập** một chương trình mới bằng cách chọn **File** → **New**. Nhập chương trình đầu tiên như dưới đây. Sau khi hoàn thành, chọn **File** → **Save as** để lưu chương trình của bạn vào đĩa cứng với phần mở rộng ".asm". Sau khi lưu tập tin lần đầu tiên, bạn có thể chọn **File** → **Save** để lưu thay đổi mà không cần chỉ định tên file.

```
1 .data
2 out_string: .asciiz "\nHello, World!\n"
3 .text
4 li $v0, 4
5 la $a0, out_string
6 syscall
7 li $v0, 10
8 syscall
```

Đoạn mã 1.1: Hello World in MIPS

2. **Lắp ráp** chương trình khi chúng đã được tạo và lưu vào đĩa cứng, nó có thể được tập lắp ráp lại (được dịch sang ngôn ngữ máy MIPS). Chọn **Run** → **Assemble**. Nếu không có lỗi, khung Execute (Thực thi) sẽ xuất hiện, hiển thị nội dung bộ nhớ và thanh ghi trước khi thực thi. Nhấp vào tab **"Edit"** nếu bạn muốn quay lại khung chỉnh sửa. Nếu có lỗi cú pháp trong chương trình của bạn, chúng sẽ xuất hiện trong cửa sổ thông báo ở cuối màn hình. Mỗi thông báo lỗi chứa dòng và vị trí trên dòng xảy ra lỗi.

3. **Đóng chương trình** Chọn **File** → **Close** để đóng chương trình hiện tại. Luôn đóng chương trình trước khi tháo đĩa chương trình hoặc thoát khỏi MARS. Thoát khỏi sao Hỏa bằng **File** → **Exit**.



Hình 1.1: Mạch logic được xây dựng trên Logisim

Hình 2.1 thể hiện môi trường của chương trình "HelloWorld" sau khi hoàn thành.

1. Màn hình thực thi được biểu thị bằng tab được đánh dấu.
2. Mã lắp ráp được hiển thị cùng với địa chỉ, mã máy, mã lắp ráp và dòng tương ứng từ tập tin mã nguồn.
3. Các giá trị được lưu trong Bộ nhớ có thể chỉnh sửa trực tiếp.
4. Cửa sổ trên Màn hình bộ nhớ được điều khiển theo nhiều cách: mũi tên trước/tiếp theo và menu các vị trí chung (ví dụ: trên cùng của ngăn xếp).
5. Cơ sở số được sử dụng để hiển thị các giá trị và địa chỉ dữ liệu (bộ nhớ và thanh ghi) có thể được chọn giữa thập phân và thập lục phân.
6. Các giá trị được lưu trong Register File có thể chỉnh sửa trực tiếp.
7. Điểm dừng được đặt bằng hộp kiểm cho mỗi lệnh lắp ráp. Các hộp kiểm này luôn được hiển thị và có sẵn.
8. Tốc độ thực thi có thể lựa chọn cho phép người dùng "xem hành động" thay vì trực tiếp kết thúc chương trình lắp ráp.
9. Tin nhắn MARS được hiển thị trên tab Tin nhắn MARS của vùng tin nhắn ở cuối màn hình. Đầu vào và đầu ra của bảng điều khiển thời gian chạy được xử lý trong tab Run I/O.

1.2 Cấu trúc một chương trình hợp ngữ MIPS

Các thành phần của chương trình MIPS như sau:

- Bình luận (Comment): Bình luận bắt đầu bằng dấu #. Mọi thứ từ # đến cuối dòng đều là một chú thích.
- Nhãn (Label): Nhãn là tên do người dùng xác định, được gán cho các lệnh và biến. Nhãn là một địa chỉ bắt đầu bằng một chữ cái, tiếp theo là các chữ cái và/hoặc chữ số và kết thúc bằng dấu hai chấm (:).
- Biến: Các biến được xác định ở đầu chương trình. Lệnh `.data` bắt đầu phần khai báo biến.
- Mã (Code): Mã xuất hiện sau các biến. Có hai chỉ thị cho mã. Lệnh .globl chỉ định tên bên ngoài của hàm. Tiếp theo là lệnh `.text`, lệnh này bắt đầu phần mã. Các nhãn chính: xuất hiện ngay sau lệnh `.text` để cho biết nơi thực thi sẽ bắt đầu.

Bố cục của một chương trình MIPS như sau:

```
1 # comments describing the program
2 .data
3 # Subsequent items put in user data segment
4 .text
5 main:
6 # Subsequent items put in user text segment
```

Đoạn mã 1.2: Bố cục chương trình MIPS

Các mã định danh phân đoạn `.data` và `.text` là bắt buộc, nhưng nhãn `main:` về mặt kỹ thuật là tùy chọn. Bố cục chương trình trong bộ nhớ: Để thực thi bộ nhớ chương trình MIPS phải được phân bổ. Máy tính MIPS có thể xử lý 4 Gigabyte bộ nhớ, từ địa chỉ `0x00000000` đến `0xffffffff`. Bộ nhớ người dùng bị giới hạn ở các vị trí dưới `0x7fffffff`.

Mục đích của các phân đoạn bộ nhớ khác nhau:

- Mã cấp độ người dùng được lưu trữ trong đoạn văn bản.
- Dữ liệu tĩnh (dữ liệu biết tại thời điểm biên dịch) do chương trình người dùng sử dụng được lưu trữ trong phân đoạn dữ liệu.
- Dữ liệu động (dữ liệu được cấp phát trong thời gian chạy) bởi chương trình người dùng được lưu trữ trong heap.

- Ngăn xếp được chương trình người dùng sử dụng để lưu trữ dữ liệu tạm thời trong chương trình con chẳng hạn cuộc gọi.
- Mã cấp độ hạt nhân (trình xử lý ngoại lệ và ngắt) được lưu trữ trong đoạn văn bản hạt nhân.
- Dữ liệu tĩnh được kernel sử dụng được lưu trữ trong phân đoạn dữ liệu kernel.
- Các thanh ghi được ánh xạ bộ nhớ cho các thiết bị IO được lưu trữ trong phân đoạn IO được ánh xạ bộ nhớ.
- Các hằng số: được lưu trữ trong bộ nhớ thay vì được tính toán khi chạy: Các hằng ký tự được đặt trong dấu ngoặc đơn, ví dụ 'a', 'Q', '4', '&'
- Các hằng số được viết ở cơ số 10, với dấu đầu dòng tùy chọn, ví dụ: 5, -17 Các hằng chuỗi được đặt trong dấu ngoặc kép, ví dụ "This is an example string"

1.3 Bài tập: chương trình cộng 2 số nguyên

Yêu cầu 2. Để bắt đầu, chúng ta sẽ viết một chương trình hợp ngữ có tên add.asm để tính tổng của 1 và 2 và lưu kết quả vào thanh ghi $t0.

1.3.1 Chú thích

Tuy nhiên, trước khi bắt đầu viết các hướng dẫn thực thi của chương trình, chúng ta cần viết chú thích mô tả những gì chương trình phải làm. Trong hợp ngữ MIPS, bất kỳ văn bản nào giữa dấu thăng # và dòng mới tiếp theo đều được hiểu là một chú thích. Ba hình thức chú thích rất cần thiết:

1. Tiêu đề chương trình: đây là mô tả tổng thể về chương trình hoặc dự án, xuất hiện ở phía trước danh sách mã.
2. Khối mã: các chú thích xác định một khối mã bên trong chương trình, chẳng hạn như vòng lặp hoặc chương trình con.
3. Lệnh quan trọng: cần có một chú thích cho mỗi lệnh hợp ngữ. Chú thích rất quan trọng đối với các chương trình hợp ngữ vì chúng rất khó đọc trừ khi chúng được ghi lại đúng cách.

Do đó, chúng ta bắt đầu bằng cách viết Tiêu đề chương trình sau:

Mặc dù chương trình này chưa làm được gì, nhưng ít nhất bất kỳ ai đọc chương trình của chúng ta cũng sẽ biết chương trình này phải làm gì và ai là tác giả của nó.

```
# Your Name -- DATE
# add.asm-- Chuong trinh tinh tong cua 1 va 2,
# ket qua duoc luu tai thanh ghi $t0.
# Registers used:
# t0 - holds the result.
# end of add.asm
```

Đoạn mã 1.3: Chú thích ở tiêu đề chương trình

1.3.2 Sử dụng các lệnh phù hợp

Tiếp theo, chúng ta cần tìm ra những lệnh nào máy tính sẽ cần thực hiện để cộng hai số. Tài liệu về tập lệnh MIPS có thể được tìm thấy trong Phụ lục A và trong Hướng dẫn của MARS. Một trong các giải pháp là dùng lệnh `addi`, là một lệnh cộng hai số với nhau, và có ba toán hạng:

1. Một thanh ghi sẽ được sử dụng để lưu trữ kết quả của phép cộng. Trong đoạn code này, chúng ta sử dụng $t0.

2. Thanh ghi chứa số đầu tiên được thêm vào. Chúng ta chọn $t1 cho mục đích này và ghi chú điều này. Vì vậy, chúng ta sẽ phải đặt giá trị 1 vào $t1 trước khi có thể sử dụng lệnh `addi`.

3. Hằng số 16 bit. Trong trường hợp này, vì 2 là hằng số dễ dàng khớp với 16 bit, nên chúng ta chỉ có thể sử dụng 2 làm toán hạng thứ ba của lệnh `addi`.

Bây giờ chúng ta đã biết cách cộng các số, nhưng chúng ta phải tìm ra cách lấy giá trị 1 vào thanh ghi $t1. Chúng ta có thể sử dụng lệnh `li` (tải giá trị tức thời - load immediate) để tải hằng số 16 bit vào thanh ghi. `li $t1, 1` là một lệnh giả cho `addi $t1, $zero, 1`. Do đó, chúng ta có đoạn lệnh 1.4 hoặc 1.5.

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
li $t1, 1 # load 1 into $t1.
addi $t0, $t1, 2 # $t0 = $t1 + 2.
# end of add.asm
```

Đoạn mã 1.4: Chương trình add.asm sử dụng lệnh *addi*

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
li $t1, 1 # load 1 into $t1.
li $t2, 2 # load 1 into $t1.
add $t0, $t1, $t2 # $t0 = $t1 + $t2.
# end of add.asm
```

Đoạn mã 1.5: Chương trình add.asm sử dụng lệnh *add*

1.3.3 Biên dịch chương trình

Hai lệnh này thực hiện phép tính mà chúng ta muốn nhưng chúng không tạo thành một chương trình hoàn chỉnh. Giống như C, một chương trình hợp ngữ phải chứa một số thông tin bổ sung cho trình biên dịch biết nơi chương trình bắt đầu và kết thúc. Hình thức chính xác của thông tin này thay đổi tùy theo trình biên dịch chương trình (lưu ý rằng có thể có nhiều trình biên dịch cho một kiến trúc nhất định và có một số trình biên dịch cho kiến trúc MIPS). Hướng dẫn này sẽ giả định rằng MARS đang được sử dụng làm môi trường biên dịch chương trình và thời gian chạy. Để chạy chương trình hợp ngữ, trình biên dịch sẽ tạo ra các bảng thông tin mã đối tượng. Điều này được thực hiện bằng cách đọc và sử dụng các lệnh, liên kết các tên tùy ý cho nhãn hoặc ký hiệu với các vị trí bộ nhớ, tạo ngôn ngữ máy và tạo một tệp đối tượng. Lưu ý rằng các lệnh giả là các lệnh mà trình biên dịch mã hiểu được nhưng không hiểu bằng ngôn ngữ máy. Nói cách khác, để có thể thực hiện các tác vụ phức tạp hơn, chúng ta cần sử dụng các macro trình biên dịch mã được gọi là lệnh giả. Ví dụ: `li $t1, 1` là lệnh giả cho `addi $t1, $zero, 1`.

Nhãn và nhãn `main`

Để bắt đầu, chúng ta cần cho trình biên dịch biết nơi chương trình bắt đầu. Trong MARS, việc thực thi chương trình bắt đầu ở đầu phân đoạn `.text`, có thể được xác định bằng nhãn chính. Nhãn là tên tượng trưng cho một địa chỉ trong bộ nhớ. Trong tập hợp MIPS, nhãn là tên ký hiệu (tuân theo các quy ước tương tự như tên ký hiệu C), theo sau là dấu hai chấm. Nhãn phải là mục đầu tiên trên một dòng. Một vị trí trong bộ nhớ có thể có nhiều nhãn. Do đó, để báo cho MARS biết rằng nó nên gán nhãn main cho lệnh đầu tiên của chương trình, chúng ta có thể viết như sau:

Khi một nhãn xuất hiện một mình trên một dòng, nó sẽ đề cập đến vị trí bộ nhớ sau. Vì vậy, chúng ta cũng có thể viết dòng này với nhãn main trên dòng riêng của nó. Đây thường là phong cách tốt hơn nhiều vì nó cho phép sử dụng các nhãn mô tả dài mà không làm gián đoạn việc thụt lề của chương trình. Nó cũng để lại nhiều khoảng trống

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
main: li $t1, 1 # load 1 into $t1.
addi $t0, $t1, 2 # $t0 = $t1 + 2.
# end of add.asm
```

Đoạn mã 1.6: Chương trình add.asm có nhãn main

trên dòng để lập trình viên viết nhận xét mô tả nhãn được sử dụng để làm gì, điều này rất quan trọng vì ngay cả các chương trình hợp ngữ tương đối ngắn cũng có thể có số lượng lớn nhãn. Lưu ý rằng trình biên dịch MARS không cho phép sử dụng tên của các lệnh làm nhãn. Do đó, nhãn có tên add không được phép vì có lệnh có cùng tên. (Tất nhiên, vì các tên lệnh đều rất ngắn và khá chung chung nên dù sao chúng cũng không tạo ra các tên nhãn có tính mô tả cao.) Việc đặt cho nhãn chính dòng riêng (và nhận xét riêng của nó) sẽ dẫn đến chương trình sau:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
main: # MARS starts execution at main.
li $t1, 1 # load 1 into $t1.
addi $t0, $t1, 2 # $t0 = $t1 + 2.
# end of add.asm
```

Đoạn mã 1.7: Chương trình add.asm có nhãn main trên dòng riêng

Lời gọi hệ thống `syscall`

Sự kết thúc của một chương trình được xác định cụ thể. Tương tự như C, trong đó hàm exit có thể được gọi để tạm dừng việc thực thi một chương trình, một cách để tạm dừng chương trình MIPS là sử dụng một cách tương tự như gọi exit trong C. Tuy nhiên, không giống như C, nếu bạn quên "gọi exit " chương trình của bạn sẽ không thoát ra một cách dễ dàng khi nó kết thúc chức năng chính. Thay vào đó, trong thực tế, nó có thể phạm lỗi thông qua bộ nhớ, diễn giải bất cứ điều gì nó tìm thấy dưới dạng hướng dẫn thực hiện. Nói chung, điều này có nghĩa là nếu bạn may mắn, chương trình của bạn sẽ chấm dứt

ngay lập tức; nếu bạn không may, nó sẽ làm điều gì đó ngẫu nhiên và sau đó bị hỏng. Cách để báo cho MARS rằng nó sẽ ngừng thực thi chương trình của bạn và cũng thực hiện một số việc hữu ích khác là dùng một lệnh đặc biệt gọi là `syscall`. Lệnh syscall tạm dừng việc thực thi chương trình của bạn và chuyển quyền điều khiển sang hệ điều hành. Sau đó, hệ điều hành sẽ xem xét nội dung của thanh ghi giá trị `$v0` để xác định xem chương trình của bạn đang yêu cầu nó làm gì. Lưu ý rằng các cuộc gọi tổng thể MARS không thực sự chuyển quyền điều khiển sang hệ điều hành. Thay vào đó, họ chuyển quyền điều khiển sang một hệ điều hành mô phỏng rất đơn giản là một phần của chương trình MARS. Trong trường hợp này, điều chúng tôi muốn là hệ điều hành làm bất cứ điều gì cần thiết để thoát khỏi chương trình của chúng tôi. Nhìn vào các hàm syscall có sẵn trong MARS, chúng ta thấy rằng điều này được thực hiện bằng cách đặt giá trị 10 (số cho lệnh thoát syscall) vào $v0 trước khi thực hiện lệnh `syscall`. Như trước đây, chúng ta có thể sử dụng lệnh `li` để thực hiện việc này:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0 - used to hold the result.
# t1 - used to hold the constant 1.
main:               # MARS starts execution at main.
li $t1, 1           # load 1 into $t1.
addi $t0, $t1, 2    # $t0 = $t1 + 2.
                    # put it into $t0.
li $v0, 10          # syscall code 10 is for exit.
syscall             # make the syscall.
# end of add.asm
```

Đoạn mã 1.8: Chương trình add.asm kết thúc bằng lời gọi mã "10"

1.4 Thao tác số học và logic

Thao khảo Phụ lục A.3 để thực hiện các Yêu cầu bên dưới.

Yêu cầu 3. Viết một chương trình hợp ngữ có tên calculator.asm để tính lần lượt tổng, hiệu, tích, thương và số dư trong phép chia của 4357 và 325; lưu kết quả vào thanh ghi $t0 đến $t4.

Yêu cầu 4. Viết một chương trình hợp ngữ có tên myinfo.asm để in ra màn hình họ tên, mã số sinh viên, ngày sinh, và email của sinh viên. Mỗi thông tin được khai báo trong từng chuỗi riêng biệt.

Yêu cầu 5. Viết một chương trình hợp ngữ so sánh hai số nguyên được thiết lập trong thanh ghi $t5 và $t6. Kết quả so sánh ghi ở thanh ghi $t7: nếu giá trị $t5 bé hơn giá trị $t6 thì $t7 được gán là 1; ngược lại, $t7 được gán là 0.

Yêu cầu 6. Xem xét đoạn chương trình sau và trả lời các câu hỏi bên dưới.

```
.data
out_string: .asciiz "\nHello, World!\n"
.text
li $v0, 4
la $a0, out_string
syscall
li $v0, 10
syscall
```

Đoạn mã 1.9: Chương trình Hello World

(a) Kí tự 'n' trong chuỗi msg được lưu vào byte có địa chỉ nào?

(b) Thứ tự lưu trữ của chuỗi là Little-Edian hay Big-Edian?

Bài thực hành LAB 2

NHẬP XUẤT VÀ RẼ NHÁNH

"Simplicity, carried to the extreme, becomes elegance." - Jon Franklin, computer scientist.

Rẽ nhánh và vòng lặp là hai cấu trúc kỳ diệu để tạo thành những chương trình máy tính thông minh và mạnh mẽ bởi khả năng ra quyết định theo ngữ cảnh khi vận hành và năng lực lặp lại một công việc với tốc độ rất cao. Tuy vậy, một chương trình hợp ngữ, hay một đoạn mã máy được đặt liên tiếp với nhau trong bộ nhớ chính trong quá trình chúng được thực thi. Bộ đếm sẽ giúp cho CPU nhảy đến câu lệnh liền kề bằng cách tăng địa chỉ lệnh thêm 4 đơn vị (với hệ thống sử dụng lệnh 4-byte) sau mỗi chu kỳ lệnh. Việc phá vỡ cấu trúc tuần tự này được thực hiện bởi các lệnh "nhảy", khi mà địa chỉ lệnh được tăng (hay giảm) một lượng giá trị phù hợp.

2.1 Nhập từ bàn phím và xuất ra màn hình

Yêu cầu 1. Viết chương trình add2.asm để tính tổng của 2 số nguyên được nhập vào tại thời điểm thực thi.

Thuật toán mà chương trình này sẽ tuân theo là:

1. Đọc hai số từ người dùng. Chúng ta sẽ cần hai thanh ghi để chứa hai số này. Chúng ta có thể sử dụng $t0 và $t1 cho việc này.
   (a) Nhận số đầu tiên từ người dùng, nhập vào $t0.
      i. tải syscall read_int vào $v0, tham khảo Phụ lục A.2 để biết mã này.
      ii. thực hiện cuộc gọi hệ thống,
      iii. di chuyển số đọc vào $t0.
   (b) Nhận số thứ hai từ người dùng, đưa vào $t1.
2. Tính tổng. Chúng ta sẽ cần một sổ đăng ký để lưu giữ kết quả của phép cộng này. Chúng ta có thể sử dụng $t2 cho việc này.

3. In tổng vừa tính được ra màn hình.

4. Kết thúc chương trình.

```
# Your Name -- DATE
 # add2.asm-- A program that computes and prints the sum
 # of two numbers specified at runtime by the user.
 # Registers used:
 # $t0 - used to hold the first number.
 # $t1 - used to hold the second number.
 # $t2 - used to hold the sum of the $t1 and $t2.
 # $v0 - syscall parameter and return value.
 # $a0 - syscall parameter.
 main:
 ## Get first number from user, put into $t0.
 li $v0, ?
 syscall
 move $t0, $v0
 #-------------------------------------#
 ## Get second number from user, put into $t1.
 #-------------------------------------#
 # Compute the sum.
 #-------------------------------------#
 li $v0, ?
 move $a0, $t2
 syscall

 li $v0, ?   # syscall code for exit.
 syscall      # make the syscall.
 # end of add2.asm.
```

Đoạn mã 2.1: Input and output in MIPS

Yêu cầu 2. Viết chương trình add2.asm để tính tổng của 2 số nguyên được nhập vào tại thời điểm thực thi dưới dạng số thập lục phân.

2.2 Cấu trúc rẽ nhánh

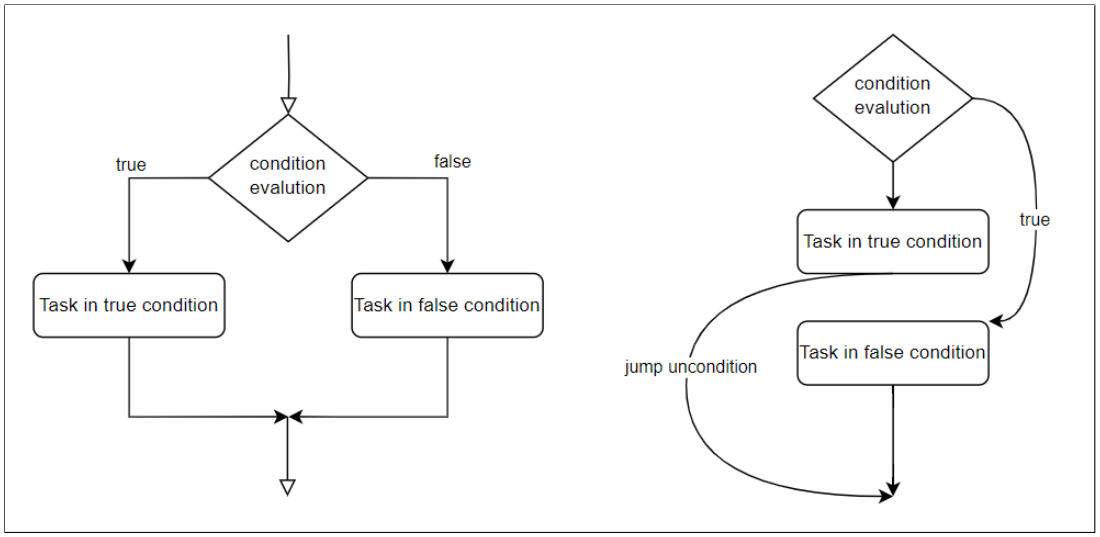
Trong các ngôn ngữ lập trình mức cao, cấu trúc rẽ nhánh dễ dàng được cài đặt thông qua các từ khóa if-else hay switch-case và các cấu trúc khối lệnh. Các chương trình máy tính sau khi được biên dịch thành mã máy đều được sắp đặt thành các lệnh liên tiếp nhau và được nạp vào bộ nhớ như là mảng một chiều mà mỗi phần tử là một lệnh có kích thước 4-byte. Vì vậy, để thực hiện cấu trúc rẽ nhánh trong MIPS, các lệnh nhảy đã được thiết kế để phá vỡ cấu trúc tuần tự. Trong đó bao gồm cả lệnh nhảy không điều kiện và lệnh nhảy khi một sự so sánh giá trị chứa trong các thanh ghi chỉ định được đánh giá là "đúng".

1. Lệnh nhảy không điều kiện `j label` là một lệnh sẽ nạp địa chỉ bộ nhớ (mà đang được định danh bằng nhãn label) vào trong thanh ghi lệnh của bộ xử lý. Bằng việc đó, lệnh đang được gán nhãn label sẽ được thực thi trong chu kỳ lệnh tiếp theo.

   (a) Nhãn được sử dụng để đánh dấu vị trí nào đó trong chương trình. Các nhãn được đặt tên tùy ý vì sự thuận tiện của lập trình viên, nhưng không được sử dụng các kí tự đặt biệt, khoảng cách và có kí số đứng đầu nhãn.

   (b) Địa chỉ được định danh bằng nhãn label được xác định tương đối trong quá trình lắp ráp ra mã máy, và tính toán tuyệt đối khi nạp vào bộ nhớ chính.

2. Lệnh nhảy có điều kiện, bao gồm `beq, bne, bgt, bge, blt, ble`, có 2 thanh ghi và 1 nhãn trong cấu trúc lệnh. Giá trị của 2 thanh ghi sẽ được so sánh và nếu sự lượng giá là "đúng/true" thì chương trình sẽ nhảy đến nhãn chỉ định. Ngược lại, lệnh nằm liền sau lệnh nhảy này sẽ được thực thi theo quy tắc của cấu trúc tuần tự.



Hình 2.1: Lược đồ cấu trúc rẽ nhánh (trái) và bố cục trong bộ nhớ chính (phải)

Yêu cầu 3. Viết chương trình test_zero.asm để người dùng nhập một giá trị nguyên vào bàn phím và thông báo ra màn hình nếu số nhập vào có giá trị là 0.

Gợi ý.

```
.data
msg: .asciiz "This is a zero"

.text
.globl main
main:

li $v0, 5      # input a integer
syscall        # make a system call
move $t0, $v0 # copy input value to desired register

bne $t0, $zero, exit  # if not a zero, skip print out and exit

li $v0, 4             # code for print a string
la $a0, msg           # address of the string
syscall

exit:
li $v0, 10            # code for exit
syscall
```

Đoạn mã 2.2: Cấu trúc rẽ nhánh if trong MIPS

Yêu cầu 4. Viết chương trình test_zero.asm để người dùng nhập một giá trị nguyên vào bàn phím và thông báo ra màn hình số nhập có giá trị là 0 hay khác 0.

Gợi ý.

```
.data
msgtrue: .asciiz "This is a zero"
msgfalse: .asciiz "This is not a zero"

.text
.globl main
main:

li $v0, 5
syscall
move $t0,$v0

beq $t0, $zero, true  # if not a zero, by-pass false statement

false:
li $v0, 4             # code for print a string
la $a0, msg2          # not a zero
syscall
j exit                # by-pass true statement

true:
li $v0, 4             # code for print a string
la $a0, msg1          # is a zero
syscall

exit:
li $v0, 10            # code for exit
syscall
```

Đoạn mã 2.3: Cấu trúc rẽ nhánh if-else trong MIPS

Yêu cầu 5. Viết chương trình test_zero.asm để người dùng nhập giá trị của tháng và chương trình in ra số ngày trong tháng đó.

Gợi ý.

```
.data
msg: .asciiz "This is a zero"

.text
.globl main
main:

li $v0, 5
syscall
move $t0,$v0

beq $t0, $zero, case1  # jump to case_1

case_1:
li $v0, 4              # code for print a string
la $a0, msg            # address of the string
syscall
j exit                 # break and go to exit

case_2:

exit:
li $v0, 10             # code for exit
syscall
```

Đoạn mã 2.4: Cấu trúc rẽ nhánh switch-case trong MIPS

Yêu cầu 6. Viết chương trình nhập vào 2 số nguyên a và b, xuất ra màn hình kết luận a lớn hơn b, hay a bé hơn b, hay a bằng giá trị với b.

Bài thực hành LAB 3

VÒNG LẶP VÀ MẢNG

"A good programmer is someone who always looks both ways before crossing a one-way street." - Doug Linder, computer scientist.

Trên nền tảng của phân tích top-down, phần lớn các thao tác số học và luận lý đều có thể hoàn tất bằng cách phân tích chúng thành những thao tác đơn giản hơn, và lặp lại nhiều lần các thao tác đó. Với số transistor tăng theo cấp số nhân, sức mạnh của bộ vi xử lý có thể giải quyết hàng tỷ phép tính trong một giây, và không hề mệt mỏi hay nhàm chán với những công việc trùng lắp.

3.1 Vòng lặp

Vòng lặp được hiện thực trong MIPS bằng ít nhất là 2 lệnh nhảy: một lệnh nhảy trở lại nhãn bắt đầu công việc cần lặp và một lệnh nhảy thoát khỏi đoạn mã lặp. Cấu trúc chung được thể hiện ở đoạn mã 3.1.

```
1 #các lệnh trước vòng lặp
2 #các thanh ghi cần khởi tạo giá trị
3 loop:
4  bne $t0, $t1, exit
5  #các công việc lặp
6  #thay đổi giá trị thanh ghi sau mỗi lần lặp
7  j loop
8  exit:
9  #các lệnh tiếp theo sau vòng lặp
```

Đoạn mã 3.1: Cấu trúc lặp trong MIPS

Yêu cầu 1. Viết chương trình loop_n.asm để in ra màn hình các số nguyên từ 1 đến n với n được nhập vào từ bàn phím tại thời điểm thực thi.

Gợi ý.

```
.data
   space:      .asciiz " "
.text
   # các lệnh trước khi vào vòng lặp, bao gồm nhập một số nguyên từ  bàn phím với
   # các lệnh gán giá trị khởi tạo cho biến index i
   li $t0, 1   # i bắt đầu từ 1
loop:
   blt $t0, $t5, exit
   move $a0, $t0        # in i ra màn hình
   li $v0, 1
   syscall

   li $v0, 4            # in một khoảng trắng sau mỗi số nguyên
   la $a0, space        # load address of space
   syscall

   addi $t0, $t0, 1     # tăng i lên 1
   j loop                # quay về đầu vòng lặp
exit:
   # các lệnh sau khi kết thúc vòng lặp
```

Đoạn mã 3.2: Một số lệnh cần thiết trong vòng lặp

Yêu cầu 2. Viết chương trình sum_n.asm để tính tổng các số nguyên từ a đến b với a và b được nhập vào từ bàn phím tại thời điểm thực thi và in giá trị tổng đó ra màn hình.

Yêu cầu 3. Viết chương trình continue_k.asm để in ra màn hình các số từ 1 đến 100 nhưng bỏ qua những số nguyên chia hết cho k, với k được nhập vào từ bàn phím tại thời điểm thực thi.

Yêu cầu 4. Viết chương trình break_k.asm để cho phép người dùng nhập liên tiếp các số nguyên vào từ bàn phím, chương trình dừng việc nhập khi người dùng nhập giá trị 0, chương trình tính tổng các số nguyên đã nhập vào và nó ra màn hình.

3.2 Mảng

Bộ thanh ghi có dung lượng rất hạn chế và không thể đáp ứng cho các cấu trúc dữ liệu hoặc là các dữ liệu cần lưu trữ trong suốt thời gian chương trình thực thi. Để giải quyết vấn đề này, bộ nhớ sẽ được cấp phát và các lệnh MIPS sẽ được cung cấp để đọc hoặc ghi lên bộ nhớ. Bộ nhớ có kích thước lớn đến hàng Gigabytes nhưng trong mỗi lượt thao tác chỉ là 1 byte (1 kí tự) hay 4 byte (1 số nguyên) được truy cập. Vì vậy, mỗi byte riêng biệt trong bộ nhớ được gán một định danh duy nhất, mà chúng ta gọi là địa chỉ.

3.2.1 Yêu cầu bộ nhớ và khởi tạo mảng

Hệ thống sẽ cấp phát một vùng nhớ có kích thước theo yêu cầu để lưu trữ dữ liệu. Không có khái niệm "kiểu dữ liệu" trong MIPS, việc xử lý ra các biểu diễn dữ liệu theo yêu cầu là trách nhiệm của lập trình viên. Hơn nữa, cấu trúc mảng trong MIPS được hiểu là các phần tử được lưu trữ liên tiếp nhau và chỉ số phần từ được tham khảo bởi độ dời (offset) trong các thao tác truy cập bộ nhớ.
MIPS cung cấp các khai báo sau đây: (các ví dụ liệt kê trong 3.4)

- **.space** Cấp phát vùng nhớ có kích thước xác định bằng số nguyên khai báo. Trong ví dụ 1, hệ thống sẽ cấp 1000 byte và gán nhãn tham khảo là "storingspace".
- **.asciiz** Cấp phát vùng nhớ vừa đủ chứa chuỗi kí tự khai báo sau đó. Trong ví dụ 2, hệ thống sẽ cấp 14 byte để chứa 13 kí tự khai báo và kí tự kết chuỗi '\0' vào một địa chỉ mà có thể truy vấn thông qua nhãn "stringname".
- **.byte** Cấp phát vùng nhớ vừa đủ chứa các giá trị (có thể là 1 kí tự hoặc 1 số nguyên thuộc [-128, 127]) được liệt kê sau đó. Trong ví dụ 3, hệ thống sẽ cấp 3 byte có địa chỉ bắt đầu được gán cho nhãn "arraybyte", và gán các giá trị 'a', -10, 84 vào từng ô 1-byte theo thứ tự liệt kê.
- **.word** Cấp phát vùng nhớ vừa đủ chứa các giá trị số nguyên 4-byte được liệt kê sau đó. Trong ví dụ 4, hệ thống sẽ cấp 12 byte có địa chỉ bắt đầu được gán cho nhãn "arrayint", và gán các giá trị 20, -10, 19842023 vào từng ô 4-byte theo thứ tự liệt kê.

```
1 .data
2    storingspace: .space 1000           # ví dụ 1
3    stringname: .asciiz "Hello, world!" # ví dụ 2
4    arraybyte: .byte 'a', -10, 84       # ví dụ 3
5    arrayint: .word 20, -10, 19842023   # ví dụ 4
```

Đoạn mã 3.3: Các hình thức yêu cầu cấp phát bộ nhớ

3.2.2 Các lệnh nạp vào thanh ghi

```
.data
   arrayint: .word 20, -10, 19842023
   size: .word 10
.text
   li $t1, 0          # load immediate: nạp giá trị nguyên vào thanh ghi
   la $s0, arrayint   # load address: nạp địa chỉ đầu tiên của mảng
                      # khai báo ở nhãn arrayint
   lw $s1, size       # đọc 4 byte đầu tiên của khai báo nhãn size,
                      # ghi vào s4
   lw $s2, size($t1)  # đọc 4 byte trong khai báo nhãn size kể từ vị trí là
                      # giá trị của t1, ghi vào s2
   lw $s4, ($s3)      # load word: đọc 4 byte trong bộ nhớ tại địa chỉ
                      # đang lưu trong thanh ghi s3, ghi vào s4
   lw $s6, 12($s5)    # load word: đọc 4 byte trong bộ nhớ tại địa chỉ
                      # đang lưu trong thanh ghi s5 dịch đi 12 byte,
                      # ghi vào s6
   lb ___             # load byte, tương tự các lệnh lw nhưng chỉ thao tác đọc
                      # đọc 1 byte thay vì 4 byte
```

Đoạn mã 3.4: Các lệnh nạp vào thanh ghi

Yêu cầu 5. Khai báo một mảng số nguyên có 10 phần tử được khởi tạo giá trị nguyên ngẫu nhiên và xuất mảng này ra màn hình.

Gợi ý.

```
.data
   list: .word 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
   size: .word 10

.main
   lw $t3, size # $t3 chứa số phần tử của mảng (N = 10)
   la $t1, list # địa chỉ của "list" truyền và cho
   $t1 li $t2, 0 # biến đếm vòng lặp i
   loop:
      beq $t2, $t3, exit # lặp cho đến khi i == N
      lw $a0, ($t1) # in giá trị phần tử i ra màn hình
      li $v0, 1 syscall
      addi $t2, $t2, 1 # i++
      addi $t1, $t1, 4 # tăng địa chỉ chứa trong $t1 lên 4
      j loop # lặp
   exit:
```

Đoạn mã 3.5: Đoạn mã in giá trị các phần tử trong một mảng khai báo sẵn

3.2.3 Các lệnh lưu giá thanh ghi ra bộ nhớ

Khi lưu giá trị thanh ghi ra bộ nhớ, chúng ta sử dụng các lệnh sb (store byte) và sw (store word) với cấu trúc tương tự các lệnh lb và lw.

Yêu cầu 6. Nhập một chuỗi kí từ vào từ bàn phím (giả sử không vượt 100 kí tự). In ra màn hình kí tự "in hoa" đầu tiên của chuỗi, hoặc thông báo "Khong tim thay".

Yêu cầu 7. Nhập mảng và tìm số lớn nhất, bé nhất rồi in ra màn hình.

Yêu cầu 8. Sắp xếp tăng dần một mảng số nguyên được khai báo sẵn.

Yêu cầu 9. Hoán vị 2 phần tử ở vị trí a và b của một mảng số nguyên. Với a và b là 2 giá trị được nhập vào từ bàn phím.

Phụ lục A

TRA CỨU MIPS

A.1 MIPS register names and conventions

Chỉ số	Tên gọi	Sử dụng	Giữ lại
$0	$zero	hằng số 0x00000000	N/A
$1	$at	biến tạm cho bộ lắp ráp	
$2 - $3	$v0 - $v1	giá trị trả về của hàm	
$4 - $7	$a0 - $a3	tham số truyền vào hàm	
$8 - $15	$t0 - $t7	các biến số tạm thời	
$16 - $23	$s0 - $s7	lưu trữ biến vào và ra	Y
$24 - $25	$t8 - $t9	các biến số tạm thời	
$26 - $27	$k0 - $k1	dành riêng cho nhân HĐH	N/A
$28	$gp	con trỏ toàn cục	Y
$29	$sp	con trỏ ngăn xếp	Y
$30	$fp	con trỏ khung	Y
$31	$ra	địa chỉ trả về	Y

Bảng A.1: Các thanh ghi trong kiến trúc MIPS

A.2 SYSCALL functions in MARS

Dịch vụ	**Mã**	**Đối số truyền vào / Kết quả trả về**
In số nguyên	1	$a0 = integer to print
In số thực float	2	$f12 = float to print
In số thực double	3	$f12 = double to print
In chuỗi kí tự	4	$a0 = address of null-terminated string to print
Đọc số nguyên	5	$v0 contains integer read
Đọc số thực float	6	$v0 contains integer read
Đọc số thực double	7	$f0 contains float read
Đọc chuỗi kí tự	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Cấp phát động	9	$a0 = number of bytes to allocate $v0 contains address of allocated memory
Kết thúc ch. trình	10	
In kí tự	11	$a0 = character to print
Đọc kí tự	12	$v0 contains character read
Mở tập tin	13	$a0 = address of null-terminated string contains filename $a1 = flags $a2 = mode $v0 contains file descriptor (negative if error)
Đọc từ tập tin	14	$a0 = file descriptor $a1 = address of input buffer $a2 = maximum number of characters to read $v0 contains number of characters read.
Ghi vào tập tin	15	$a0 = file descriptor $a1 = address of output buffer $a2 = number of characters to write $v0 contains number of characters written.
Đóng tập tin	16	$a0 = file descriptor
Kết thúc với mã	17	$a0 = termination result

Bảng A.2: Mã nạp cho thanh ghi $v0

A.3 Nhóm lệnh luận lý và số học (Định dạng R)

*** Lệnh số học và ý nghĩa**

```
add $t1, $t2, $t3    # $t1 = $t2 + $t3
sub $t1, $t2, $t3    # $t1 = $t2 - $t3
and $t1, $t2, $t3    # $t1 = $t2 & $t3 (bitwise and)
or $t1, $t2, $t3     # $t1 = $t2 | $t3 (bitwise or)
xor
nor
```

*** Lệnh thiết lập sau khi so sánh**

```
seq $t1, $t2, $t3     # $t1 = $t2 == $t3 ? 1 : 0
slt $t1, $t2, $t3     # $t1 = $t2 < $t3 ? 1 : 0
sle $t1, $t2, $t3     # $t1 = $t2 <= $t3 ? 1 : 0
sne      # presedou code
sgt
sge
```

A.4 Nhóm lệnh luận lý và số học (Định dạng I)

Nhóm lệnh I cũng giống như các lệnh nhóm R, nhưng toán hạng thứ hai thay là một hằng số 16-bit.

*** Lệnh số học và ý nghĩa**

```
addi $t1, $t2, 4        # $t1 = $t2 + 4
subi $t1, $t2, 15       # $t1 = $t2 - 15
andi $t1, $t2, 0x00FF   # $t1 = $t2 & 0x00FF
ori
xori
slti $t1, $t2, 42       # $t1 = $t2 < 42 ? 1 : 0
```

A.5 Nhóm lệnh nhân và chia số nguyên

Lệnh nhân tạo ra kết quả 64-bit được lưu trữ lần lượt trong 2 thanh ghi hi và lo.

```
mult $t1, $t2 # hi,lo = $t1 * $t2
mflo $t0 # $t0 = lo
mfhi $t3 # $t3 = hi
#Shortcut (macro instruction):
mul $t0, $t1, $t2 # hi,lo = $t1 * $t2; $t0 = lo
#Expands to:
mult $t1, $t2
mflo $t0
```

Lệnh chia số nguyên tạo ra kết quả chia nguyên và số dư được lưu trữ lần lượt trong thanh thi lo và hi.

```
div $t1, $t2
     # lo = $t1 / $t2;
     # hi = $t1 % $t2;
mflo $t2 # $t2 = lo
mfhi $t3 # $t3 = hi
```

A.6 Nhóm lệnh thao tác thanh ghi

```
move $t1, $t2 # $t1 = $t2
li $t1, 42 # $t1 = 42
li $t1, 'k' # $t1 = 0x6B
la $t1, label # $t1 = label
```

A.7 Nhóm lệnh nhảy

```
beq $t0,$t1,target # branch to target if $t0 = $t1
blt $t0,$t1,target # branch to target if $t0 < $t1
ble $t0,$t1,target # branch to target if $t0 <= $t1
bgt $t0,$t1,target # branch to target if $t0 > $t1
bge $t0,$t1,target # branch to target if $t0 >= $t1
bne $t0,$t1,target # branch to target if $t0 <> $t1
```