Laboratory Exercise 1

Giới thiệu về công cụ MARS

Goals

Sau bài thực hành này, bạn sẽ cài đặt được công cụ MARS, viết thử chương trình đơn giản để thử nghiệm công cụ MARS như lập trình hợp ngữ, chạy giả lập, gỡ rối, và các phương tiện nhằm hiểu rõ hơn về bản chất và các hoạt động thực sự đã xảy $ratrong bộ xử lý MIPS.

Literature

- Tài liệu Kien truc MIPS.pptx
- Tài liệu MARS features, file .doc
- Tài liệu MARS Tutorial, file .doc

About MIPSIT

- MIPS (**Microprocessor without Interlocked Pipeline Stages**) hình thành trên cơ sở RISC.
- Năm 1981: John L. Hennessy đứng đầu một nhóm bắt đầu một công trình nghiên cứu về *bộ xử lý MIPS* đầu tiên tại **Stanford University**
- Một thiết kế chủ chốt trong MIPS là yêu cầu các câu lệnh phải hoàn thành trong 1 chu kì máy.
- Một số ứng dụng
    - Pioneer DVR-57-H
    - Kenwood HDV-810 Car Navigation System
    - HP Color Laser Jet 2500 Printer

Kick-off

Tải về và chạy

1. Tải về Java Runtime Environment, JRE, để chạy công cụ MARS
   http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Cài đặt JRE
3. Tải về công cụ MARS, bao gồm
    a. Phiên bản mới nhất của MARS, và nên lấy thêm 2 tài liệu:
    b. MARS features, và
    c. MARS Tutorial

   ở URL sau
   http://courses.missouristate.edu/KenVollmar/MARS/download.htm

Xong. Công cụ MARS có thể thực hiện ngay mà không cần cài đặt. Click đúp vào file Mars.jar để chạy.

Chú ý: để tiện lợi, các sinh viên có thể tải về các tài nguyên liên quan tại URL duy nhất sau: ftp://dce.hust.edu.vn/tiennd/ict4/Mars/

Cơ bản về giao diện lập trình IDE

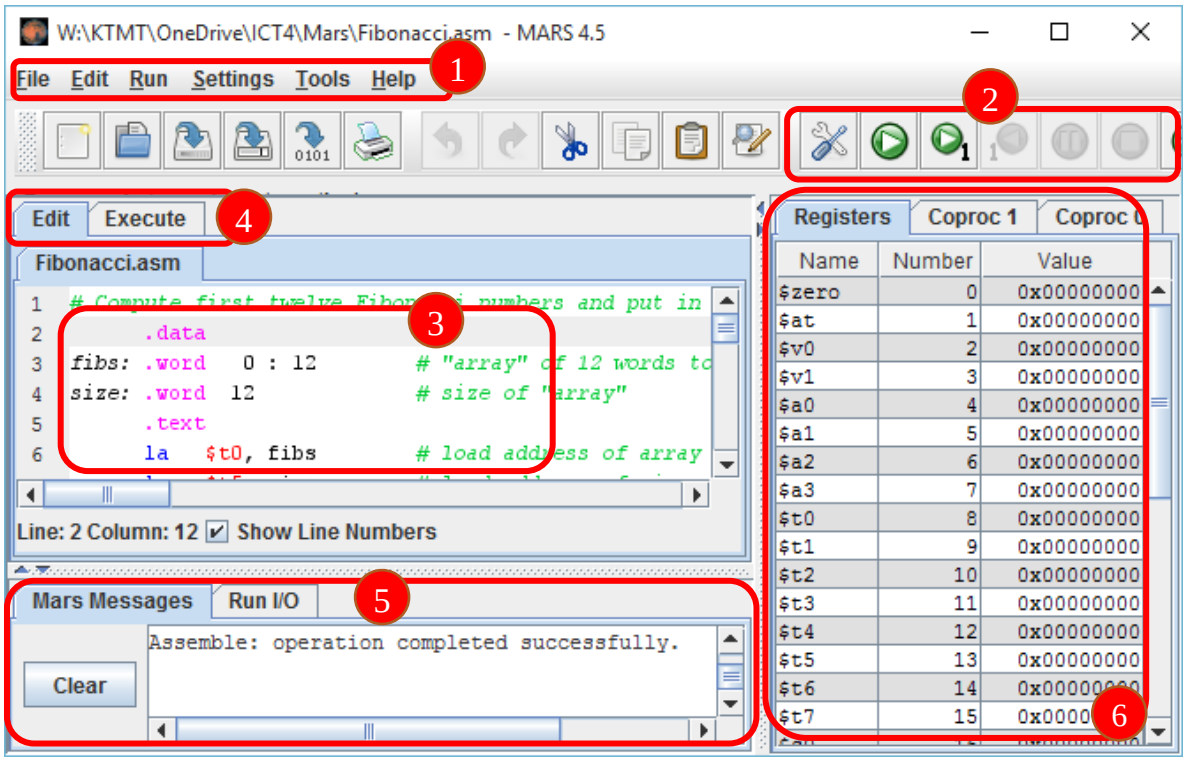


Figure 1. IDE of MARS Tool

1. **Menus**: Hầu hết các mục trong menu đều có các icon tương ứng
   - Di chuyển chuột lên trên của icon → tooltip giải thích về chức năng tương ứng sẽ hiển thị.

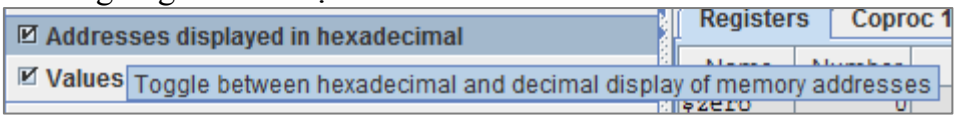


Figure 2. Tooltip giải thích chức năng trong các Menu

   - Các mục trong menu cũng có phím tắt tương ứng.
2. **Toolbar:**
   - Chứa một vài tính năng soạn thảo cơ bản như: copy, paste, open..
   - Các tính năng gỡ rỗi (trong hình chữ nhật màu đỏ)
     - Run: chạy toàn bộ chương trình
     - Run one step at a time: chạy từng lệnh và dừng (rất hữu ích)
     - Undo the last step: khôi phục lại trạng thái ở lệnh trước đó (rất hữu ích)
     - Pause: tạm dừng quá trình chạy toàn bộ (Run)
     - Stop: kết thúc quá trình gỡ rối
     - Reset MIPS memory and register: Khởi động lại
3. **Edit tab:** MARS có bộ soạn thảo văn bản tích hợp sẵn với tính tô màu theo cú pháp, giúp sinh viên dễ theo dõi mã nguồn. Đồng thời, khi sinh

viên gõ lệnh mà chưa hoàn tất, một popup sẽ hiện $rađể trợ giúp. Vào menu Settings / Editor… để thay đổi màu sắc, font…

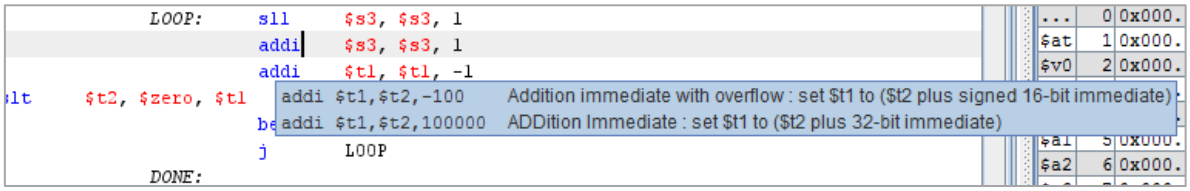


Figure 3. Popup trợ giúp hoàn tất lệnh và giải thích lệnh

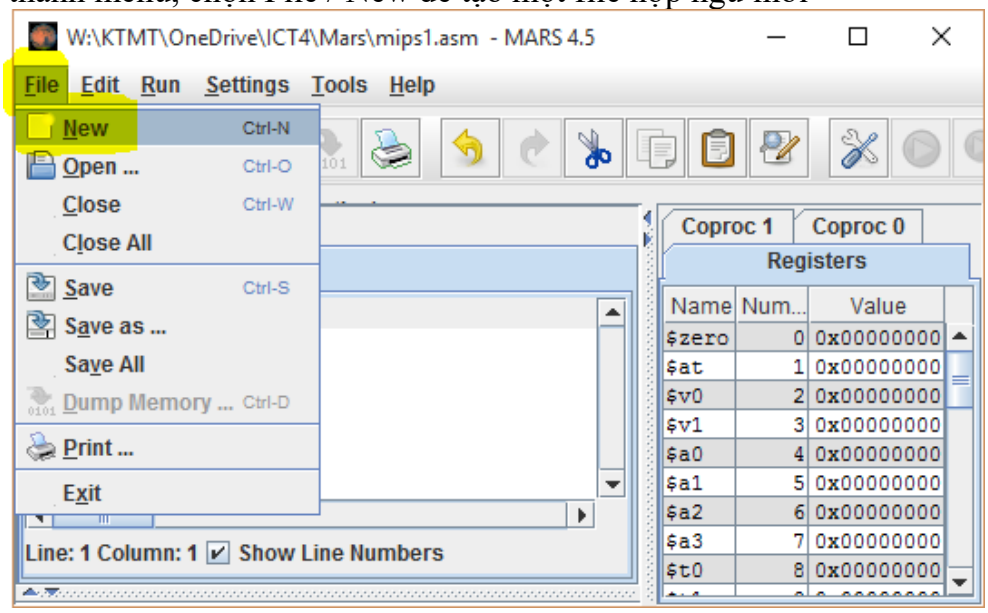
4. **Edit/Execute**:
   Mỗi file mã nguồn ở giao diện soạn thảo có 2 cửa sổ - 2 tab: Edit và Execute
   - **Edit tab:** viết chương trình hợp ngữ với tính năng tô màu theo cú pháp.
   - **Execute tab:** biên dịch chương trình hợp ngữ đã viết ở Edit tab thành mã máy, **chạy** và **gỡ rối**.
5. **Message Areas**: Có 2 cửa sổ message ở cạnh dưới của giao diện IDE
   - **The *Run I/O* tab** chỉ có tác dụng khi đang chạy run-time
     - Hiển thị các kết quả xuất $raconsole, và
     - Nhập dữ liệu vào cho chương trình qua console.

     MARS có tùy chọn để bạn có thể mọi thông tin nhập liệu vào qua console sẽ được hiển thị lại $ramessage area.
   - ***MARS Messages* tab** được dùng để hiển thị cho các thông báo còn lại như là các báo lỗi trong quá trình biên dịch hay trong quá trình thực hiện run-time. Bạn có thể click vào thông báo lỗi để chương trình tự động nhảy tới dòng lệnh gây $ralỗi.
6. **MIPS Registers**: Bảng hiển thị giá trị của các thanh ghi của bộ xử lý MIPS, luôn luôn được hiện thị, bất kể chương trình hợp ngữ có được chạy hay không. Khi viết chương trình, bảng này sẽ giúp bạn nhớ được tên của các thành ghi và địa chỉ của chúng. Có 3 tab trong bảng này:
   - **the Register File:** các thanh ghi số nguyên với địa chỉ từ $0 tới $31, và cả 3 thanh ghi đặc LO, HI và thanh ghi Program Counter
   - **the Coprocesor 0 registers:** các thanh ghi của bộ đồng xử lý C0, phục vụ cho xử lý ngắt
   - **the Coprocesor 1 registers:** các thanh ghi số dấu phẩy động

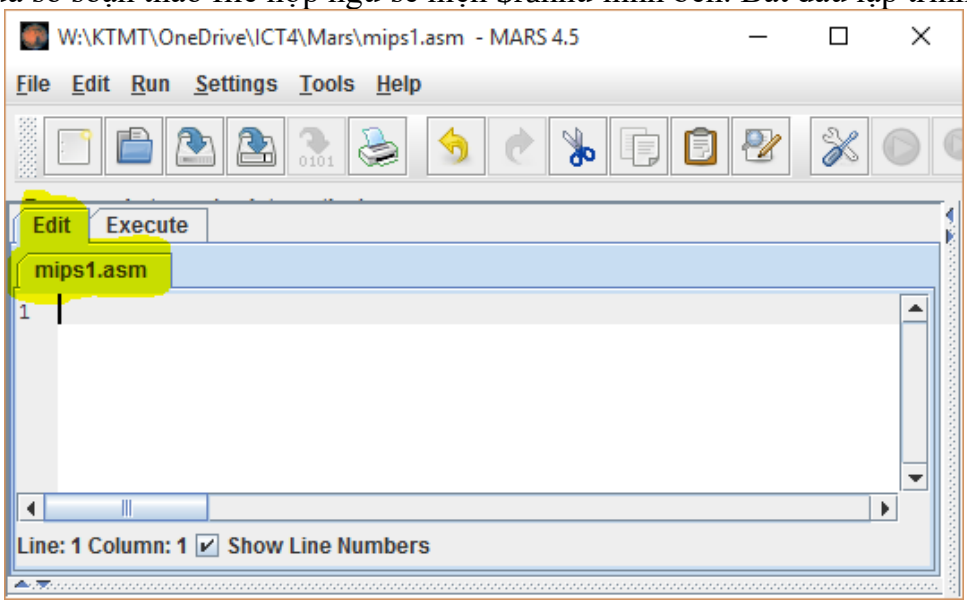
Bắt đầu lập trình và hiểu các công cụ với chương trình Helloworld

1. Click vào file mars.jar để bắt đầu chương trình

2. Ở thanh menu, chọn File / New để tạo một file hợp ngữ mới



3. Cửa sổ soạn thảo file hợp ngữ sẽ hiện $ranhư hình bên. Bắt đầu lập trình



4. Hãy gõ đoạn lệnh sau vào cửa sổ soạn thảo

```
.data                    # Vung du lieu, chua cac khai bao bien
x:        .word     0x01020304     # bien x, khoi tao gia tri
message:  .asciiz   "Bo mon Ky thuat May tinh"
.text                  # Vung lenh, chua cac lenh hop ngu
   la $a0, message       #Dua dia chi bien mesage vao thanh ghi a0
   li   $v0, 4             #Gan thanh ghi $v0 = 4
   syscall                 #Goi ham so v0, ham so 4, la ham print

   addi $t1,$zero,2        #Thanh ghi $t1 = 2
   addi $t2,$zero,3        #Thanh ghi $t2 = 3
   add  $t0, $t1, $t2      #Thanh ghi t- = $t1 + $t2
```

Kết quả như sau.

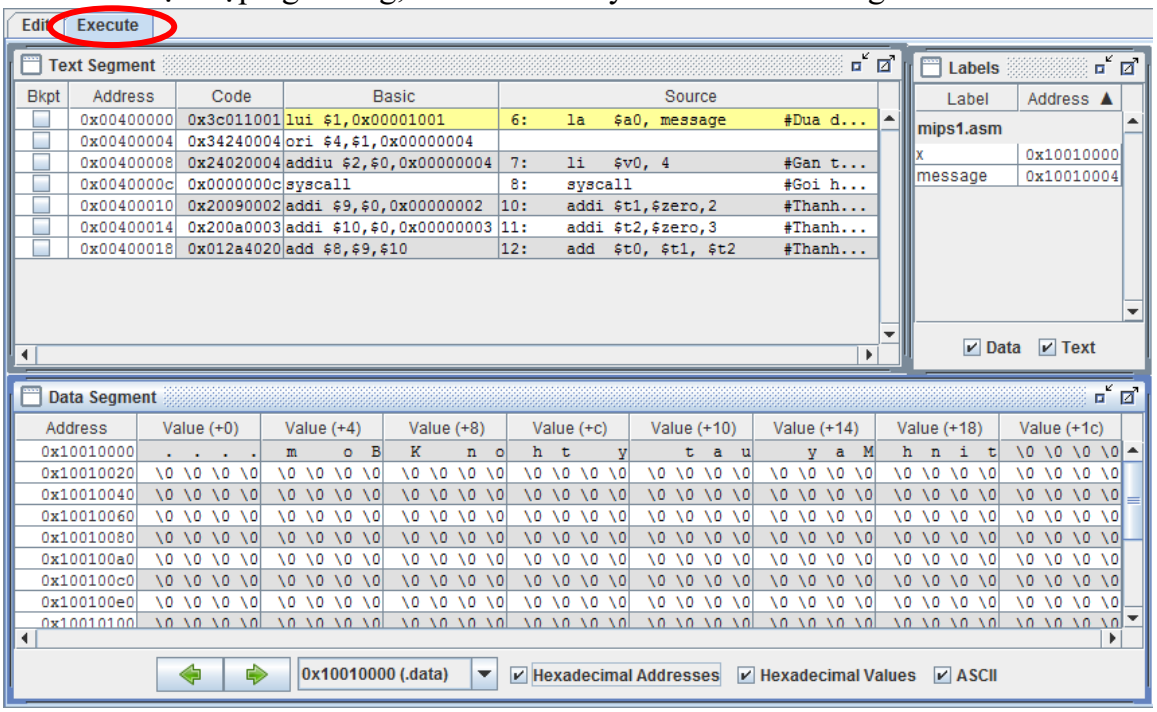
Edit | Execute

mips1.asm*

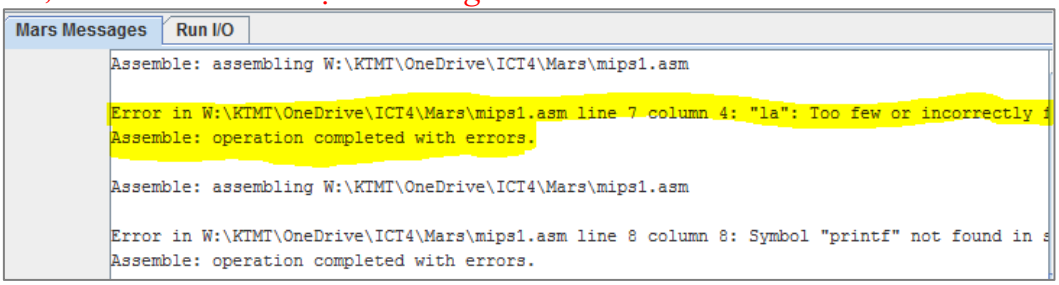
```
1   .data                    # Vung du lieu, chua cac khai bao bien
2   x: .word     0x01020304    # bien x, khoi tao gia tri
3   message:  .asciiz  "Bo mon Ky thuat May tinh"
4
5   .text                    # Vung lenh, chua cac lenh hop ngu
6      la    $a0, message      #Dua dia chi bien mesage vao thanh ghi a0
7      li    $v0, 4            #Gan thanh ghi v0 = 4
8      syscall                 #Goi ham so v0, ham so 4, la ham print
9
10     addi $t1,$zero,2        #Thanh ghi t1 = 2
11     addi $t2,$zero,3        #Thanh ghi t2 = 3
12     add  $t0, $t1, $t2      #Thanh ghi t- = t1 + t2
```

5. Để biên dịch chương trình hợp ngữ trên thành mã máy, thực hiện một trong các cách sau:
   - vào menu Run / Assemble, hoặc
   - trên thanh menu, bấm vào biểu tượng , hoặc
   - bấm phím tắt F3.

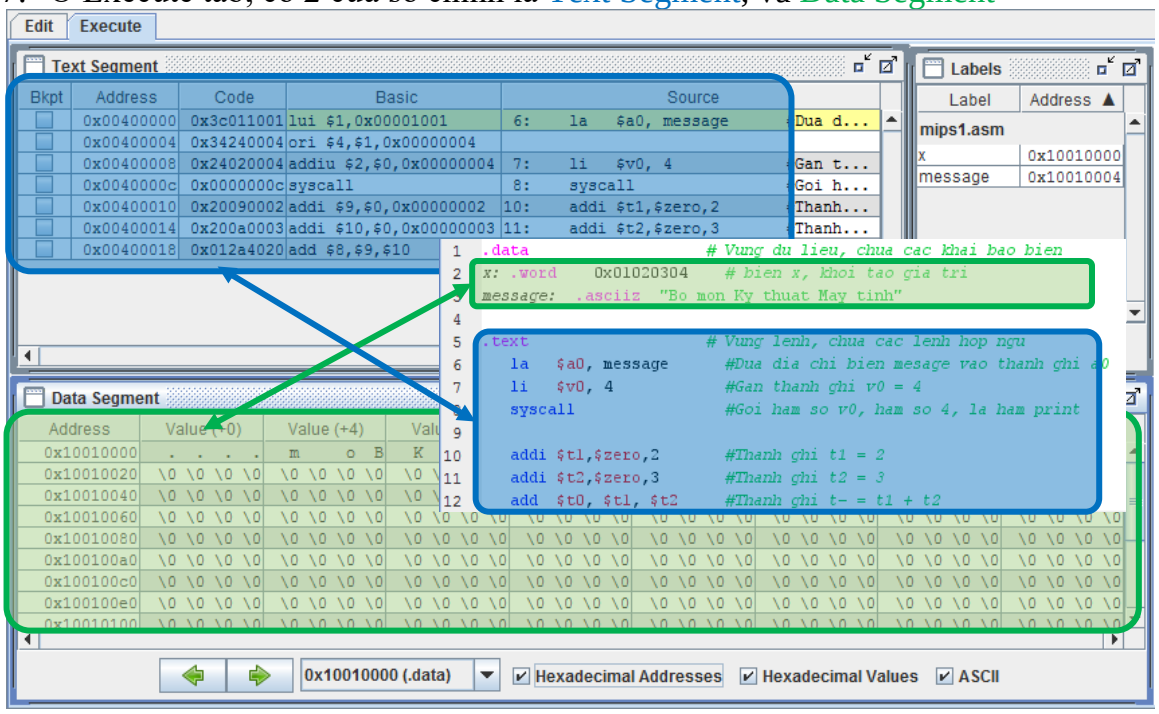
6. Nếu đoạn hợp ngữ đúng, MARS sẽ chuyển từ Edit tab sang Execute tab.



Chú ý: nếu đoạn hợp ngữ có lỗi, cửa sổ Mars Messages sẽ hiển thị chi tiết lỗi. Bấm vào dòng thông báo lỗi để trình soạn thảo tự động nhảy tới dòng code bị lỗi, rồi tiến hành sửa lại cho đúng.



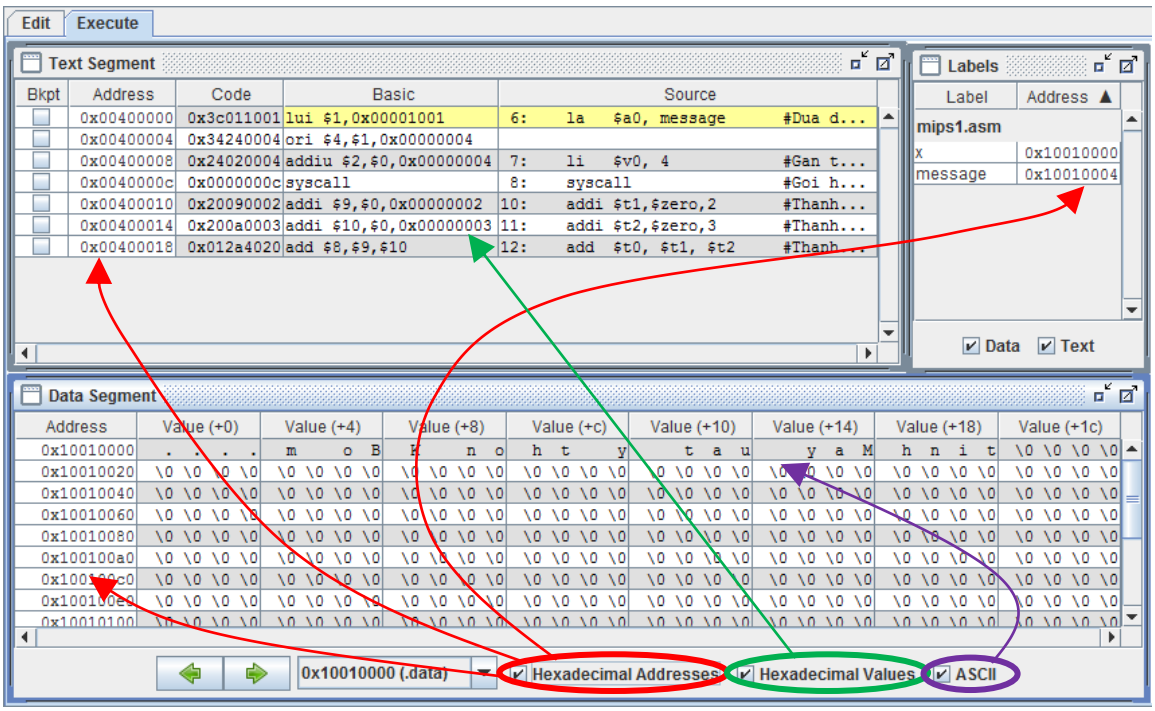
7. Ở Execute tab, có 2 cửa sổ chính là Text Segment, và Data Segment



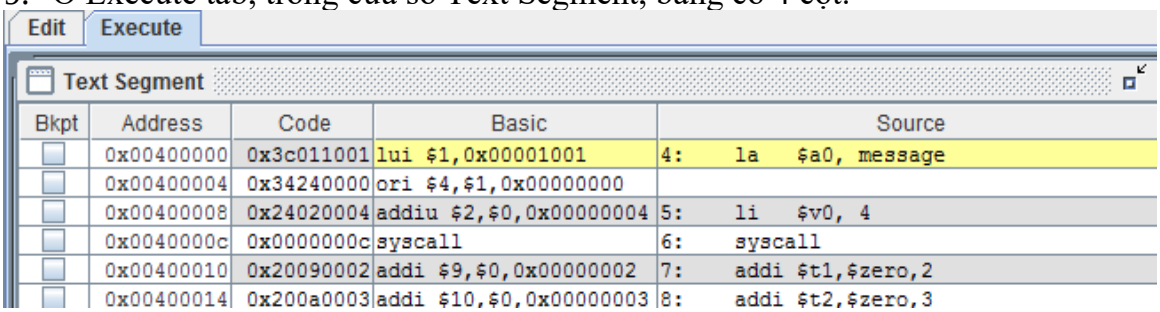
- **Text Segment:** là vùng không gian bộ nhớ chứa các mã lệnh hợp ngữ. Tương ứng với mã nguồn hợp ngữ, các dòng nào viết sau chỉ thị **.TEXT** tức là lệnh và sẽ thuộc Text Segment.
- **Data Segment:** là vùng không gian bộ nhớ chứa các biến. Tương ứng với mã nguồn hợp ngữ, các dòng nào viết sau chỉ thị **.DATA** tức là lệnh và sẽ thuộc Text Segment.

Chú ý: vì lý do nào đó, nếu ta khai báo biến sau chỉ thị .TEXT hoặc ngược lại thì trình biên dịch sẽ báo lỗi hoặc bỏ qua khai báo sai đó.

8. Ở Execute tab, sử dụng checkbox bên dưới để thay đổi cách hiển thị dữ liệu cho dễ quan sát
   - Hexadecimal Addresses : hiển thị địa chỉ ở dạng số nguyên hệ 16
   - Hexadecimal Values : hiển thị giá trị thanh ghi ở dạng số nguyên hệ 16
   - ASCII : hiển thị giá trị trong bộ nhớ ở dạng kí tự ASCII

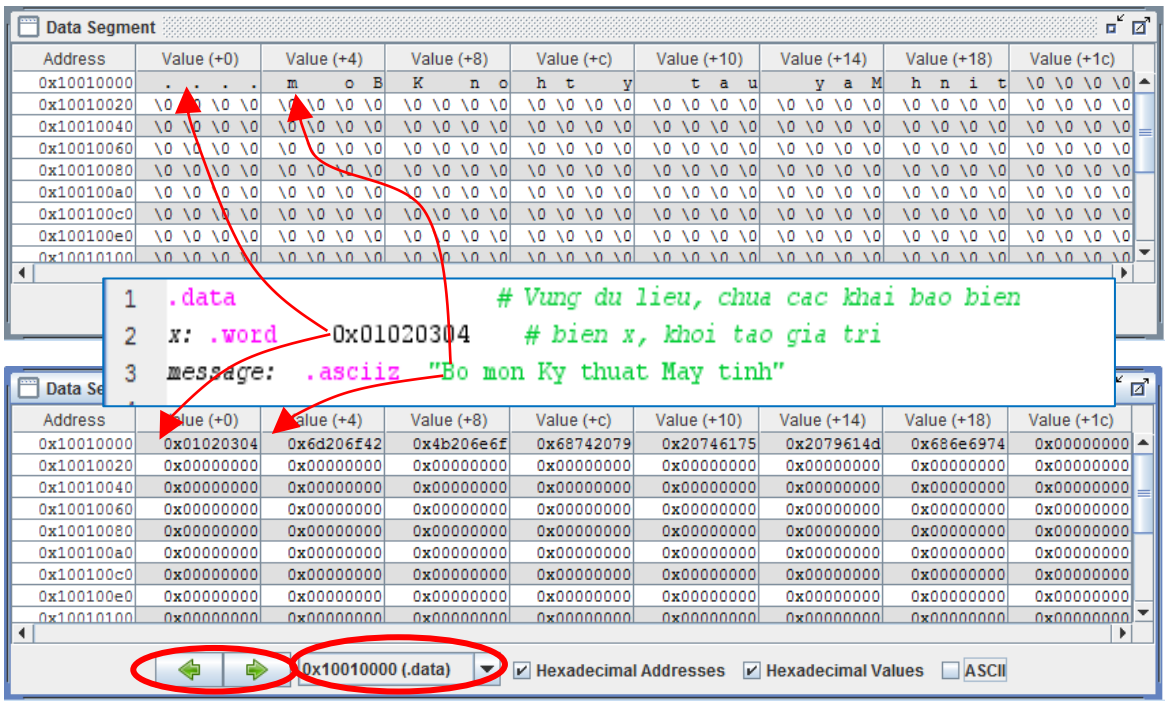


9. Ở Execute tab, trong cửa sổ Text Segment, bảng có 4 cột.



- **Bkpt**: Breakpoint, điểm dừng khi chạy toàn bộ chương trình chương trình bằng nút .
- **Address**: địa chỉ của lệnh ở dạng số nguyên *(xem thêm hướng dẫn về cửa sổ Label)*
- **Code**: lệnh ở dạng mã máy
- **Basic**: lệnh ở dạng hợp ngữ thuần, giống như qui định trong tập lệnh. Ở đây, tất cả các nhãn, tên gọi nhớ.. đều đã được chuyển đổi thành hằng số.
- **Source**: lệnh ở dạng hợp ngữ có bổ sung các macro, nhãn.. giúp lập trình nhanh hơn, dễ hiểu hơn, không còn giống như tập lệnh nữa. Trong ảnh minh họa bên dưới:
    i. Lệnh **la** trong cột Source là lệnh giả, không có trong tập lệnh → được dịch tương ứng thành 2 lệnh lui và ori trong cột Basic.
    ii. Nhãn message trong lệnh la $a0, message trong cột Source → được dịch thành hằng số 0x00001001 *(xem thêm hướng dẫn về cửa sổ Label)*

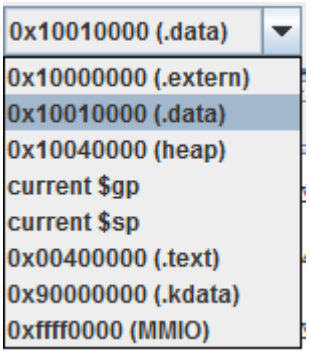
10. Ở Execute tab, trong cửa sổ Data Segment, bảng có 9 cột



- **Address**: địa chỉ của dữ liệu, biến ở dạng số nguyên. Giá trị mỗi dòng tăng 32 đơn vị (ở hệ 10, hoặc $20_{(16)}$) bởi vì mỗi dòng sẽ trình bày 32 byte ở các địa chỉ liên tiếp nhau
- **Các cột Value***:* mỗi cột Value chứa 4 byte, và có 8 cột Value, tương ứng với 32 byte liên tiếp nhau.

Trong hình ảnh trên, có thể thấy rõ giá trị của biến x = 0x01020304 được hiển thị chính xác trong Data Segment khi hiển thị dữ liệu ở dạng số ☐ ASCII, và giá trị của chuỗi "Bo mon Ky thuat May tinh" khi hiển thị ở dạng kí tự ☑ ASCII. *Lưu ý rằng việc lưu trữ chuỗi trong bộ nhớ ở dạng little-endian là do cách lập trình hàm phần mềm syscall, chứ không phải do bộ xử lý MIPS qui định. Có thể thấy, ở công cụ giả lập MIPS IT, hàm print lại qui định chuỗi theo kiểu big-endian.*

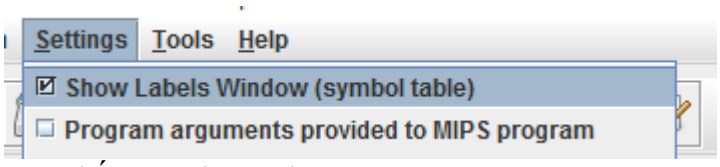
Bấm vào cặp nút để dịch chuyển tới vùng địa chỉ lân cận



Bấm vào combobox để dịch tới vùng bộ nhớ chứa loại dữ liệu được chỉ định. Trong đó lưu ý

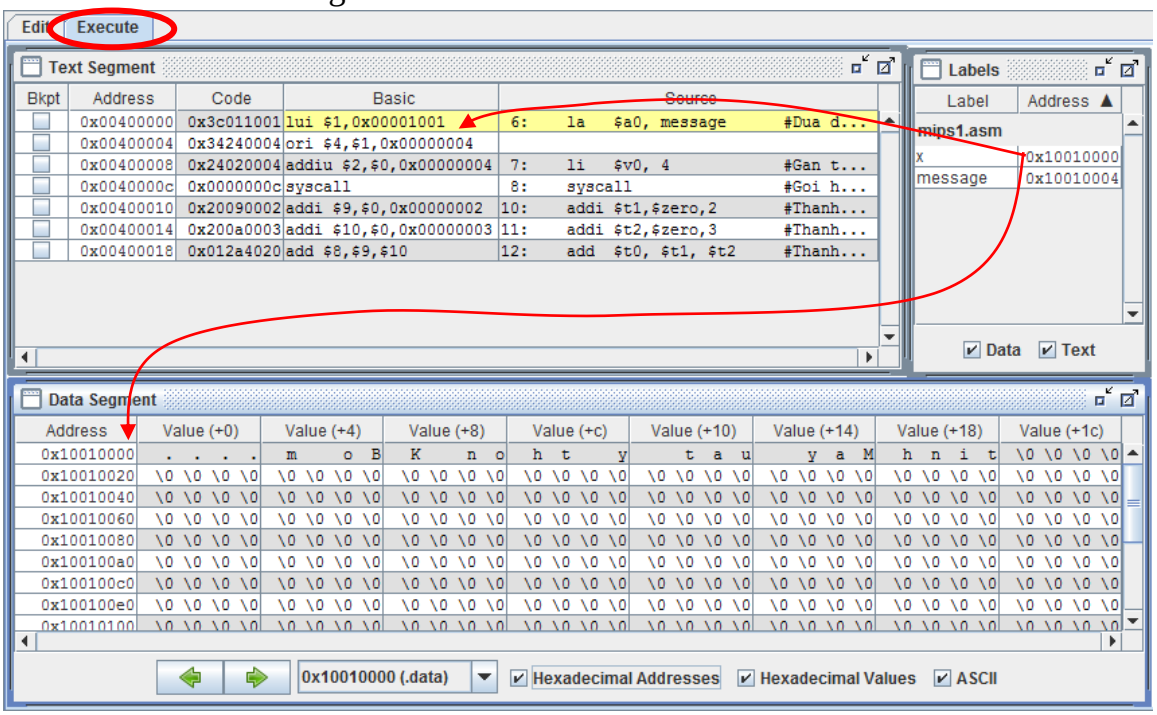
- .data: vùng dữ liệu
- .text: vùng lệnh
- $sp: vùng ngăn xếp

11. Cửa sổ Label: hiển thị tên nhãn và hằng số địa chỉ tương ứng với nhãn khi được biên dịch $ramã máy.
    Cửa sổ Label không tự động hiển thị. Phải vào menu Settings / chọn Show Labels Windows



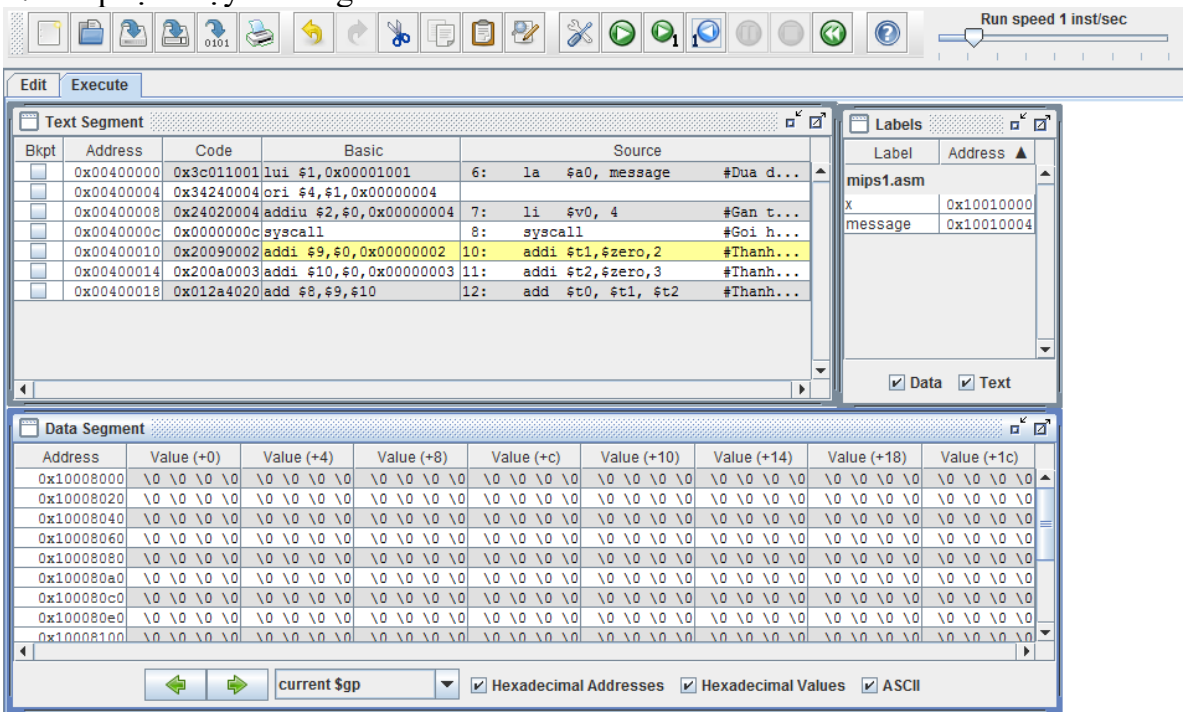
Trong ảnh sau, ta thấy các thông tin quan trọng :

- Trong cửa sổ Labels cho biết :
  - X chỉ là tên gợi nhớ, x sẽ được qui đổi thành hằng số 0x10010000.
  - Message cũng chỉ là tên gợi nhớ, sẽ được qui đổi thành hằng số 0x10010004
  - Click đúp vào tên biến, sẽ tự động chuyển sang vị trí tương ứng trong cửa sổ Data Segment.
- Trong cửa sổ Text Segment cho biết
  - Quả thực, ở lệnh gán *la $a0, message* tên gợi nhớ message đã được chuyển thành hằng số 0x1001004 thông qua cặp lệnh lui, ori
- Trong cửa sổ Data Segment cho biết
  - Quả thực, để giám sát giá trị của biến X, ta mở Data Segment ở hằng số 0x10010000 sẽ nhìn thấy giá trị của X.
  - Quả thực, để giám sát giá trị của biến message, ta mở Data Segment ở hằng số 0x10010004 sẽ nhìn thấy giá trị của message.

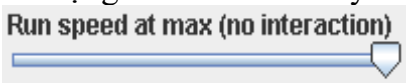


Chạy giả lập

1. Tiếp tục chạy chương trinh Hello World ở trên.



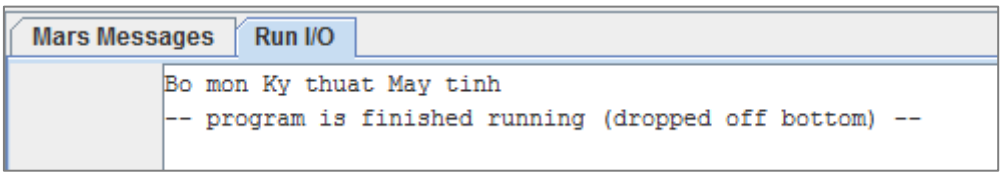
2. Sử dụng slider bar để thay đổi tốc độ thực thi lệnh hợp ngữ.



Mặc định, tốc độ thực thi là tối đa, và ở mức này, ta không thể can thiệp được nhiều vào quá trình hoạt động của các lệnh và kiểm soát chúng. Có thể dịch chuyển slider bar về khoảng 2 lệnh/giây để dễ quan sát.

3. Ở Execute tab, chọn cách để thực thi chương trình

- Bấm vào icon , Run, để thực hiện toàn bộ chương trình. Khi sử dụng Run, ta quan sát dòng lệnh được tô màu vàng cho biết chương trình hợp ngữ đang được xử lý tới chỗ nào. Đồng thời, quan sát sự biến đổi dữ liệu trong cửa sổ Data Segment và cửa sổ Register.
- Bấm vào icon , Reset, để khởi động lại trình giả lập về trạng thái ban đầu. Tất cả các ngăn nhớ và các thanh ghi đều được gán lại về 0.
- Bấm vào icon , Run one step, để thực thi chỉ duy nhất 1 lệnh rồi chờ bấm tiếp vào icon đó, để thực hiện lệnh kế tiếp.
- Bấm vào icon , Run one step backwards, để khôi phục lại trạng thái và quay trở lại lệnh đã thực thi trước đó.
- Sau khi chạy xong tất cả các lệnh của phần mềm Hello Word, sẽ thấy cửa sổ Run I/O hiển thị chuỗi.



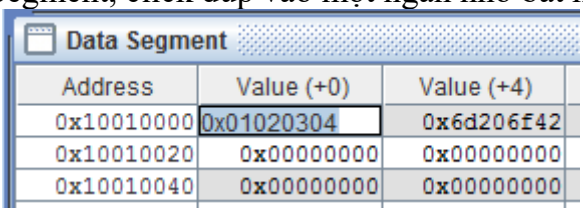
Giả lập & gỡ rối: quan sát sự thay đổi của các biến

Trong quá trình chạy giả lập, hãy chạy từng lệnh với chức năng Run one step. Ở mỗi lệnh, quan sát sự thay đổi giá trị trong cửa sổ Data Segment và cửa sổ Register, và hiểu rõ ý nghĩa của sự thay đổi đó.

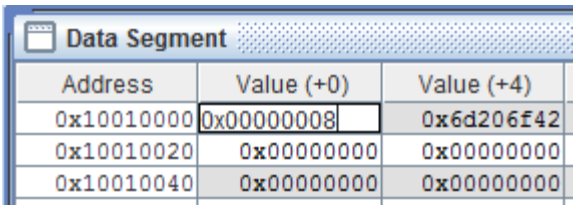
Giả lập & gỡ rối: thay đổi giá trị biến khi đang chạy run-time

Trong khi đang chạy giả lập, ta có thể thay đổi giá trị của một ngăn nhớ bất kì bằng cách

1. Trong Data Segment, click đúp vào một ngăn nhớ bất kì



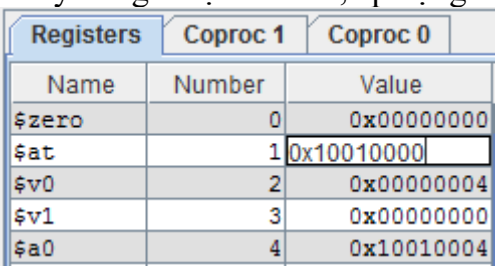
2. Nhập giá trị mới



3. Tiếp tục chạy chương trình

Giả lập & gỡ rối: thay đổi giá trị thanh ghi khi đang chạy run-time

Cách làm tương tự như thay đổi giá trị của biến, áp dụng cho cửa sổ Registers



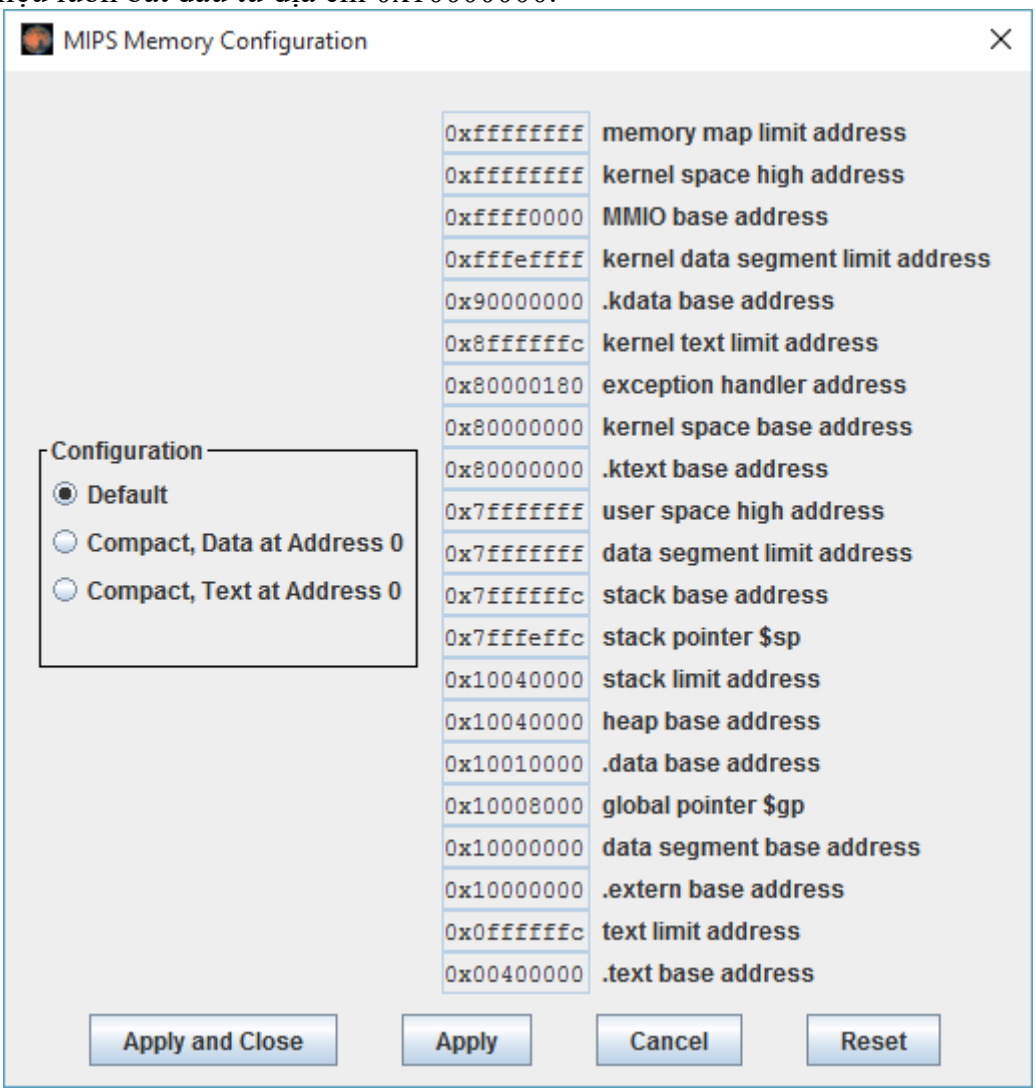
Tra cứu Help

Bâm nút Help để xem giải thích các lệnh của MIPS, các giả lệnh, chỉ dẫn biên dịch và các hàm của syscalls.

Các hằng địa chỉ

Chọn menu Settings / Memory Configuration…
Cửa sổ MIPS Memory Configuration chứa bảng qui định các hằng địa chỉ mà công cụ MARS sử dụng.

Theo bảng này, có thể thấy các mã lệnh luôn bắt đầu từ địa chỉ 0x00400000, còn dữ liệu luôn bắt đầu từ địa chỉ 0x10000000.



Laboratory Exercise 2

Instruction Set, Basic Instructions, Directives

Goals

Sau bài thực hành này, sinh viên sẽ nắm được nguyên lý cơ bản về tập lệnh của bộ xử lý MIPS; sử dụng được các lệnh hợp ngữ cơ bản và sử dụng công cụ gỡ rối để kiểm nghiệm lại các kiến thức về tập lệnh và hợp ngữ . Sinh viên cũng thành thạo với các chỉ thị biên dịch (Directives) để công cụ MARS có thể dịch hợp ngữ thành mã máy một cách đúng đắn.

Literature

- Tài liệu tóm tắt về Kien truc MIPS[1], file pptx
- Bảng tra cứu tập lệnh MIPS[2], file .doc

Assignments at Home and at Lab

Home Assignment 1

Đọc tài liệu về Kiến trúc MIPS và ghi nhớ các kiến thức cơ bản sau

- Tên và ý nghĩa của 32 thanh ghi
- Các thanh ghi đặc biệt PC, HI, LO
- Khuôn dạng của 3 loại lệnh I, J, R

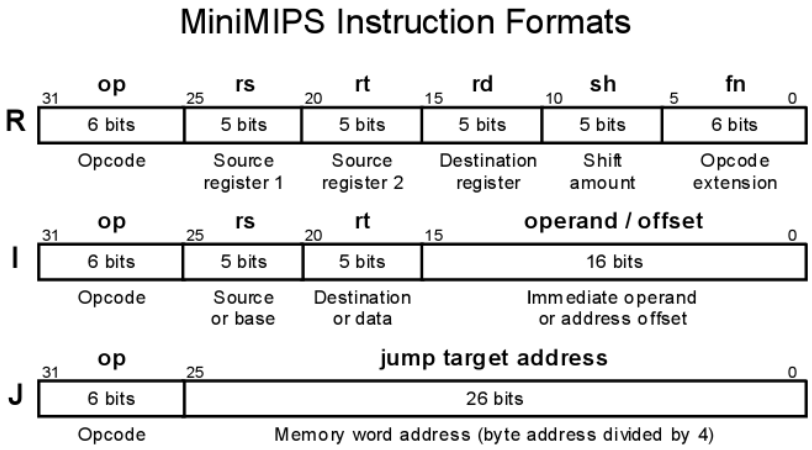
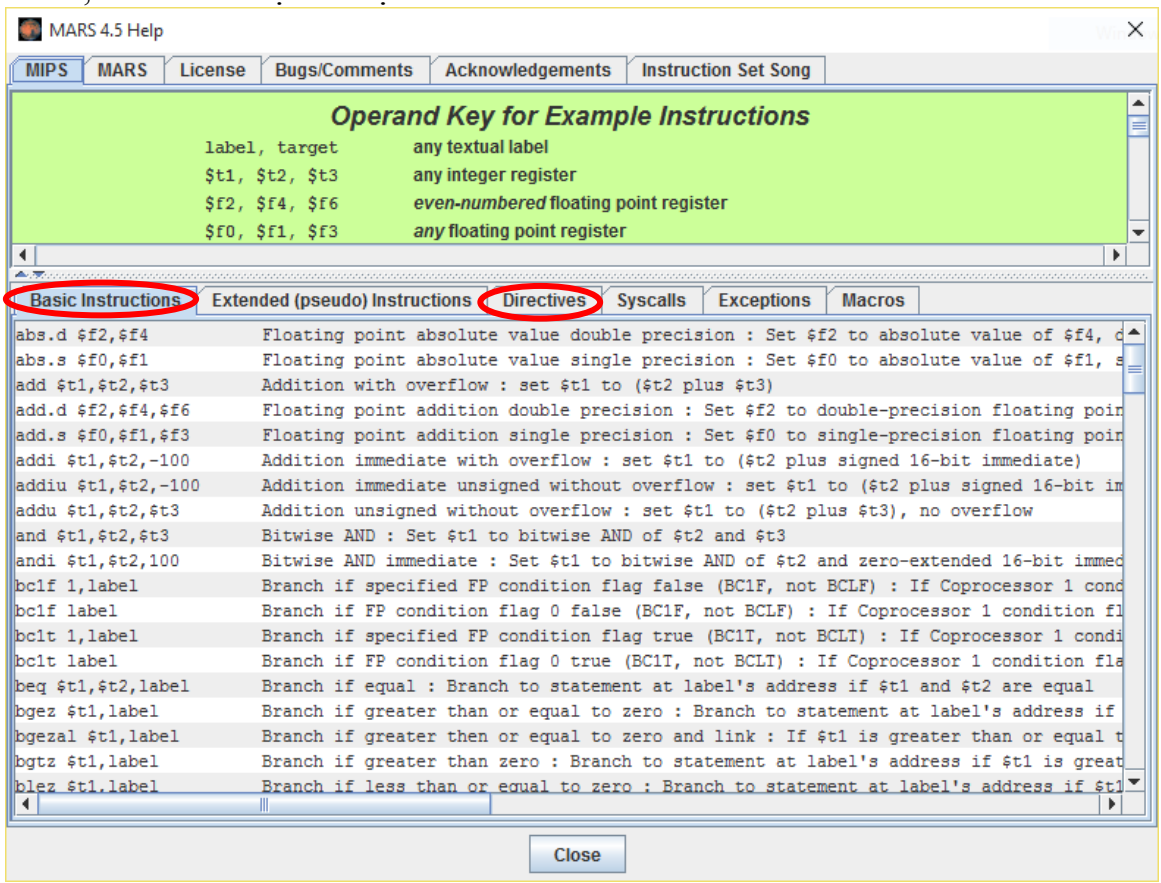


Figure 5.4 MiniMIPS instructions come in only three formats: register (R), immediate (I), and jump (J).

[1] Download tại: ftp://dce.hust.edu.vn/tiennd/ict4/References/Kien truc MIPS.pptx
[2] Download tại: ftp://dce.hust.edu.vn/tiennd/ict4/References/MIPS32-InstructionSet-QuickReference.pdf

Home Assignment 2

Sử dụng công cụ MARS, tra cứu Help và tìm hiểu về các lệnh cơ bản trong MIPS, và các chỉ thị biên dịch



Assignment 1: lệnh gán số 16-bit

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 1
.text
      addi    $s0, $zero, 0x3007 # $s0 = 0 + 0x3007 = 0x3007 ;I-type
      add     $s0, $zero, $0     # $s0 = 0 + 0 = 0            ;R-type
```

Sau đó:

- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của thanh ghi $s0
    - Sự thay đổi giá trị của thanh ghi $pc
- Ở cửa sổ Text Segment, hãy so sánh mã máy của các lệnh trên với khuôn dạng lệnh để chứng tỏ các lệnh đó đúng như tập lệnh đã qui định
- Sửa lại lệnh lui như bên dưới. Chuyện gì xảy ra sau đó. Hãy giải thích

```
      addi   $s0, $zero, 0x2110003d
```

Assignment 2: lệnh gán số 32-bit

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 2
```

```
.text
      lui    $s0,0x2110         #put upper half of pattern in $s0
      ori    $s0,$s0,0x003d     #put lower half of pattern in $s0
```

Sau đó:

- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của thanh ghi $s0
    - Sự thay đổi giá trị của thanh ghi $pc
- Ở cửa sổ Data Segment, hãy click vào hộp combo để chuyển tới quan sát các byte trong vùng lệnh .text.
    - Kiểm tra xem các byte đầu tiên ở vùng lệnh trùng với cột nào trong cửa sổ Text Segment.

Assignment 3: lệnh gán (giả lệnh)

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 3
.text
      li     $s0,0x2110003d #pseudo instruction=2 basic instructions
      li     $s1,0x2      #but if the immediate value is small, one ins
```

Sau đó:

- Biên dịch và quan sát các lệnh mã máy trong cửa sổ Text Segment. Giải thích điều bất thường?

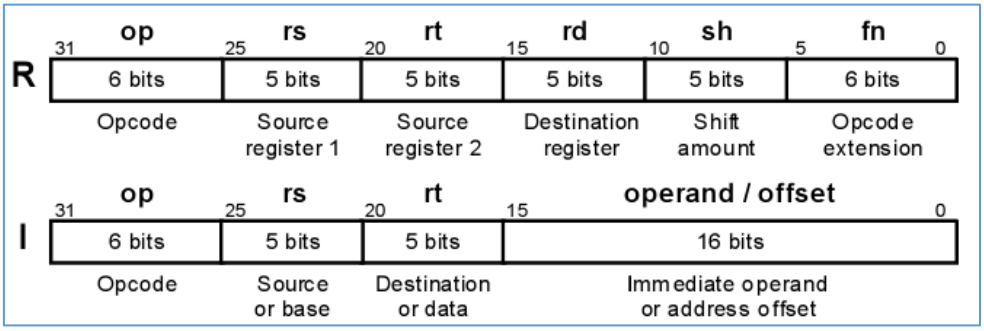
Assignment 4: tính biểu thức 2x + y = ?

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 4
.text
     #  Assign X, Y
      addi  $t1, $zero, 5     # X = $t1 = ?
      addi  $t2, $zero, -1    # Y = $t2 = ?
     #  Expression Z = 2X + Y
     add    $s0, $t1, $t1     # $s0 = $t1 + $t1 = X + X = 2X
     add    $s0, $s0, $t2     # $s0 = $s0 + $t2 = 2X + Y
```

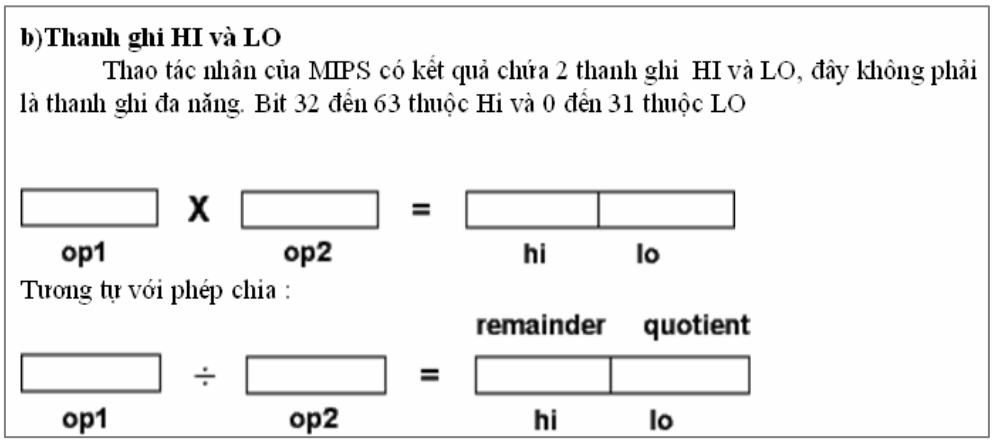
Sau đó:

- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của các thanh ghi
    - Sau khi kết thúc chương trình, xem kết quả có đúng không?
- Ở cửa sổ Text Segment, xem các lệnh **addi** và cho biết điểm tương đồng với hợp ngữ và mã máy. Từ đó kiểm nghiệm với khuôn mẫu của kiểu lệnh I



- Ở cửa sổ Text Segment, chuyển mã máy của lệnh **add** sang hệ 2. Từ đó kiểm nghiệm với khuôn mẫu của kiểu lệnh R.

Assignment 5: phép nhân



Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 5
.text
    #  Assign X, Y
    addi  $t1, $zero, 4     # X = $t1 = ?
    addi  $t2, $zero, 5     # Y = $t2 = ?
    #  Expression Z = 3*XY
    mul   $s0, $t1, $t2     # HI-LO = $t1 * $t2 = X * Y  ; $s0 = LO
    mul   $s0, $s0, 3       # $s0 = $s0 * 3 = 3 * X * Y
    #  Z' = Z
    mflo  $s1
```

Sau đó:

- Biên dịch và quan sát các lệnh mã máy trong cửa sổ Text Segment. Giải thích điều bất thường?
- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của các thanh ghi, đặc biệt là Hi, Lo
    - Sau khi kết thúc chương trình, xem kết quả có đúng không?

Assignment 6: tạo biến và truy cập biến

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 6
.data                       # DECLARE VARIABLES
X :  .word    5               # Variable X, word type, init value =
```

```
Y :  .word   -1            # Variable Y, word type, init value =
Z :  .word                 # Variable Z, word type, no init value

.text                      # DECLARE INSTRUCTIONS
     #  Load X, Y to registers
     la    $t8, X           # Get the address of X in Data Segment
     la    $t9, Y           # Get the address of Y in Data Segment
     lw    $t1, 0($t8)      # $t1 = X
     lw    $t2, 0($t9)      # $t2 = Y

     #  Calcuate the expression Z = 2X + Y with registers only
     add   $s0, $t1, $t1    # $s0 = $t1 + $t1 = X + X = 2X
     add   $s0, $s0, $t2    # $s0 = $s0 + $t2 = 2X + Y

     # Store result from register to variable Z
     la    $t7, Z           # Get the address of Z in Data Segment
     sw    $s0, 0($t7)      # Z = $s0 = 2X + Y
```

Sau đó:

- Biên dịch và quan sát các lệnh mã máy trong cửa sổ Text Segment.
    - Lệnh **la** được biên dịch như thế nào?
- Ở cửa sổ Label và quan sát địa chỉ của X, Y, Z.
    - So sánh chúng với hằng số khi biên dịch lệnh **la** thành mã máy
    - Click đúp vào các biến X, Y, Z để công cụ tự động nhảy tới vị trí của biến X, Y, Z trong bộ nhớ ở cửa sổ Data Segment. Hãy bảo đảm các giá trị đó đúng như các giá trị khởi tạo.
- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của các thanh ghi
    - Xác định vai trò của lệnh **lw** và **sw**
- Ghi nhớ qui tắc xử lý
    - Đưa tất cả các biến vào thanh ghi bằng cặp lệnh **la**, **lw**
    - Xử lý dữ liệu trên thanh ghi
    - Lưu kết quả từ thanh ghi trở lại biến bằng cặp lệnh **la**, **sw**
- Tìm hiểu thêm các lệnh **lb**, **sb**

Laboratory 3

Load/ Store , Jump & Branch instructions

Goals

After this laboratory exercise, you should know how to use load/store, jump and branch instructions. You should also be able to implement high level programming language structures such as conditional statement (if-then-else), loop and selection statement (switch-case)

Literature

Behrooz Parhami (CAMS): Section 5.4, 5.5

Preparation

Before start this laboratory, you should review textbook and read the entire laboratory exercise in detail. You also need to review Laboratory Exercise 2 and try to experience MARS by yourself.

Assignments at Home and at Lab

Home Assignment 1

This home assignment implements "if-then-else" statement using some fundamental instructions, such as slt, addi, jump and branch.

if (i<=j)
x=x+1;
z=1;
else
y=y-1;
z=2*z;

At first, you should draw the algorithm chart for this statement. After that, you read this example carefully, try to clarify the function of each intructions.

```
#Laboratory Exercise 3, Home Assignment 1
start:
      slt   $t0,$s2,$s1        # j<i
      bne   $t0,$zero,else     # branch to else if j<i
      addi  $t1,$t1,1          #     then part: x=x+1
      addi  $t3,$zero,1        # z=1
      j     endif              # skip "else" part
else: addi  $t2,$t2,-1         # begin else part: y=y-1
      add   $t3,$t3,$t3        # z=2*z
endif:
```

Home Assignment 2

The following example demonstrates how to implement loop statement. This program computes the sum of elements of array A.

loop: ***i=i+step;***
Sum=sum+A[i];
If(I !=n) goto loop;

Assuming that the index i, the starting address of A, the comparison constant n, step and sum are found in registers $s1, $s2, $s3, $s4 and $s5, respectively. You should try to understand each line in this code.

```
#Laboratory 3, Home Assigment 2
.text
loop: add   $s1,$s1,$s4       #i=i+step
      add   $t1,$s1,$s1       #t1=2*s1
      add   $t1,$t1,$t1       #t1=4*s1
      add   $t1,$t1,$s2       #t1 store the address of A[i]
      lw    $t0,0($t1)        #load value of A[i] in $t0
      add   $s5,$s5,$t0       #sum=sum+A[i]
      bne   $s1,$s3,loop      #if i != n, goto loop
```

Home Assignment 3

A switch/case statement allows multiway branching based on the value of an integer variable. In the following example, the switch variable test can assume one of the three values in [0, 2] and a different action is specified for each case.

switch(test) {
case 0:
a=a+1; break;
case 1:
a=a-1; break;
case 2:
*b=2*b; break;*
}

Assuming that **a** and **b** are stored in registers $s2 and $s3. You should read this code section carefully, understand how to implement switch/case statement.

```
#Laboratory Exercise 3, Home Assignment 3
.data
test: .word 1
.text
            la    $s0, test   #load the address of test variable
            lw    $s1, 0($s0) #load the value of test to register $t1
            li    $t0, 0      #load value for test case
            li    $t1, 1
            li    $t2, 2
            beq   $s1, $t0, case_0
            beq   $s1, $t1, case_1
            beq   $s1, $t2, case_2
            j     default
case_0:     addi  $s2, $s2, 1       #a=a+1
            j     continue
case_1:     sub   $s2, $s2, $t1     #a=a-1
            j     continue
case_2:     add   $s3, $s3, $s3     #b=2*b
            j     continue
default:
continue:
```

Assignment 1

Create a new project to implement the code in Home Assignment 1. Initialize for i and j variable. Compile and upload to the simulator. Run this program step by step, observe the changing of memory and the content of registers at each step.

Assignment 2

Create a new project implementing the code in Home Assignment 2. Initialize for i, n, step, sum variables and array A. Compile and upload to the simulator. Run this program step by step, observe the changing of memory and the content of registers by each step. Try to test with some more cases (change the value of variables).

Assignment 3

Create a new project implementing the code in Home Assignment 3. Compile and upload to the simulator. Run this program step by step; observe the changing of memory and the content of registers by each step. Change the value of test variable and run this program some times to check all cases.

Assignment 4

Modify the Assignment 1, so that the condition tested is

a. i <j
b. i >= j
c. i+j <= 0
d. i+j > m+n

Assignment 5

Modify the Assignment 2, so that the condition tested at the end of the loop is

a. i < n
b. i <= n
c. sum >= 0
d. A[i] == 0

Assignment 6

Using all of above instructions and statements, create a new project to implement this function: find the element with the largest absolute value in a list of integers. Assuming that this list is store in an integer array and we know the number of elements in it.

Conclusions

Before you pass the laboratory exercise, think about the questions below:

- Which registers are affected by a branch instruction?

Laboratory Exercise 4

Arithmetic and Logical operation

Goals

After this laboratory exercise, you should know how to use arithmetic, logical and shift instructions. In addition, you should also understand overflow in arithmetic operation and how to detect it.

Literature

Behrooz Parhami (CAMS): Section 5.3

Preparation

Before you start the exercise, you should review the textbook, section 5.3 and read this laboratory carefully. You should also review the Laboratory Exercise 2.

Assignments at Home and at Lab

Home Assignment 1

The sum of two 32-bit integers may not be representable in 32 bits. In this case, we say that an overflow has occurred. Overflow is possible only with operands of the same sign. For two nonnegative (negative) operands, if the sum obtained is less (greater) than eitheir operand, overflow has occurred. The following program dectects overflow based on this rule. Two operands are stored in register $s1 and $s2, the sum is stored in register $s3. If overflow occur, $t0 register is set to 1 and clear to 0 in otherwise.

```
#Laboratory Exercise 4, Home Assignment 1
.text
start:
      li    $t0,0             #No Overflow is default status
      addu  $s3,$s1,$s2       # s3 = s1 + s2
      xor   $t1,$s1,$s2       #Test if $s1 and $s2 have the same sign

      bltz  $t1,EXIT          #If not, exit
      slt   $t2,$s3,$s1
      bltz  $s1,NEGATIVE      #Test if $s1 and $s2 is negative?
      beq   $t2,$zero,EXIT    #s1 and $s2 are positive
           # if $s3 > $s1 then the result is not overflow
      j     OVERFLOW
NEGATIVE:
      bne   $t2,$zero,EXIT    #s1 and $s2 are negative
           # if $s3 < $s1 then the result is not overflow
OVERFLOW:
      li    $t0,1             #the result is overflow
EXIT:
```

Home Assignment 2

The following program demonstrates how to use logical instructions to extract information from one register. We can extract one bit or more according to the mask we use. Read this example carefully and explain each lines of code.

```
#Laboratory Exercise 4, Home Assignment 2
.text
      li    $s0, 0x0563       #load test value for these function
      andi  $t0, $s0, 0xff    #Extract the LSB of $s0
      andi  $t1, $s0, 0x0400  #Extract bit 10 of $s0
```

Home Assignment 3

This example show how the shift operations used to implement other instructions, such as multiply by a small power of 2.

```
#Laboratory Exercise 4, Home Assignment 3
.text
      li    $s0,1             #s0=1
      sll   $s1,$s0,2         #s1=s0*4
```

Assignment 1

Create a new project to implement the Home Assigment 1. Compile and upload to simulator. Initialize two operands (register $s1 and $s2), run this program step by step, observe memory and registers value.

Assignment 2

Write a program to do the following tasks:

- Extract MSB of $s0
- Clear LSB of $s0
- Set LSB of $s0 (bits 7 to 0 are set to 1)
- Clear $s0 (s0=0, must use logical instructions)

MSB: Most Significant Byte
LSB: Least Significant Byte

s0 = 0x 1 2 3 4 5 6 7 8

MSB LSB

Assignment 3

Pseudo instructions in MIPS are not-directly-run-on-MIPS-processor instructions which need to be converted to real-instructions of MIPS. Re-write the following pseudo instructions using real-instructions understood by MIPS processors:

a. `abs  $s0,s1`
   `s0 <= | $s1 |`

b. `move  $s0,s1`
   `s0 <= $s1`

c. `not  $s0`
   `s0 <= bit invert (s0)`

d. `ble  $s1,s2,L`

```
if (s1 <= $s2)
     j L
```

Assignment 4

To dectect overflow in additional operation, we also use other rule than the one in Assignment 1. This rule is: when add two operands that have the same sign, overflow will occur if the sum doesn't have the same sign with either operands. You need to use this rule to write another overflow detection program.

Assignment 5

Write a program that implement multiply by a small power of 2. (2, 4, 8, 16, etc for example).

Conclusions

Before you pass the laboratory exercise, think about the questions below:

- What is the difference between SLLV and SLL instructions?
- What is the difference between SRLV and SRL instructions?

Laboratory Exercise 5

Character string with SYSCALL function, and sorting

Goals

After this laboratory exercise, you should understand the mechanism of storing ASCII and Unicode string. You will be able to program to process string and put string to console. In addition, you should know how to sort a list of elements.

Literature

Patterson, Henessy (COD): section 2.8, 2.13

Preparation

Before you start the exercise, you should review the textbook, section 6.1 and read this laboratory carefully. You should also read the Mips Lab Environment Reference to find the usage of printf, putchar procedures … and so on.

About SYSCALL

A number of system services, mainly for input and output, are available for use by your MIPS program. They are described in the table below.

MIPS register contents are not affected by a system call, except for result registers as specified in the table below.

How to use SYSCALL system services

1. Load the service number in register $v0.
2. Load argument values, if any, in $a0, $a1, $a2, or $f12 as specified.
3. Issue the SYSCALL instruction.
4. Retrieve return values, if any, from result registers as specified.

Example: display the value stored in $t0 on the console

```
li   $v0, 1            # service 1 is print integer
li   $a0, 0x307        # the interger to be printed is 0x307
syscall                # execute
```

Table of Frequently Available Services

Service	Code in $v0	Arguments	Result
print decimal integer	1	$a0 = integer to print	

print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
exit	10	(terminate execution)	
print character	11	$a0 = character to print	*See note below table*
read character	12		$v0 contains character read
open file	13	$a0 = address of null-terminated string containing filename $a1 = flags $a2 = mode	$v0 contains file descriptor (negative if error). *See note below table*
read from file	14	$a0 = file descriptor $a1 = address of input buffer $a2 = maximum number of characters to read	$v0 contains number of characters read (0 if end-of-file, negative if error). *See note below table*
write to file	15	$a0 = file descriptor $a1 = address of output buffer $a2 = number of characters to write	$v0 contains number of characters written (negative if error). *See note below table*
close file	16	$a0 = file descriptor	
exit2 (terminate with value)	17	$a0 = termination result	*See note below table*
time (system time)	30		$a0 = low order 32 bits of system time $a1 = high order 32 bits of system time. *See note below table*
MIDI out	31	$a0 = pitch (0-127) $a1 = duration in milliseconds $a2 = instrument (0-127) $a3 = volume (0-127)	Generate tone and return immediately. *See note below table*
sleep	32	$a0 = the length of time to sleep in milliseconds.	Causes the MARS Java thread to sleep for (at least) the specified number of milliseconds. This timing will not be precise, as the Java implementation will add some overhead.
MIDI out synchronous	33	$a0 = pitch (0-127) $a1 = duration in milliseconds $a2 = instrument (0-127) $a3 = volume (0-127)	Generate tone and return upon tone completion. *See note below table*
print integer in hexadecimal	34	$a0 = integer to print	Displayed value is 8 hexadecimal digits, left-padding with zeroes if necessary.
print integer in binary	35	$a0 = integer to print	Displayed value is 32 bits, left-padding with zeroes if necessary.
print integer as unsigned	36	$a0 = integer to print	Displayed as unsigned decimal value.
(not used)	37-39		
set seed	40	$a0 = i.d. of pseudorandom	No values are returned. Sets the seed of the corresponding underlying Java

		number generator (any int). $a1 = seed for corresponding pseudorandom number generator.	pseudorandom number generator (`java.util.Random`). *See note below table*
random int	41	$a0 = i.d. of pseudorandom number generator (any int).	$a0 contains the next pseudorandom, uniformly distributed int value from this random number generator's sequence. *See note below table*
random int range	42	$a0 = i.d. of pseudorandom number generator (any int). $a1 = upper bound of range of returned values.	$a0 contains pseudorandom, uniformly distributed int value in the range 0 = [int] [upper bound], drawn from this random number generator's sequence. *See note below table*
ConfirmDialog	50	$a0 = address of null-terminated string that is the message to user	$a0 contains value of user-chosen option 0: Yes 1: No 2: Cancel
InputDialogInt	51	$a0 = address of null-terminated string that is the message to user	$a0 contains int read $a1 contains status value 0: OK status -1: input data cannot be correctly parsed -2: Cancel was chosen -3: OK was chosen but no data had been input into field
InputDialogString	54	$a0 = address of null-terminated string that is the message to user $a1 = address of input buffer $a2 = maximum number of characters to read	*See Service 8 note below table* $a1 contains status value 0: OK status. Buffer contains the input string. -2: Cancel was chosen. No change to buffer. -3: OK was chosen but no data had been input into field. No change to buffer. -4: length of the input string exceeded the specified maximum. Buffer contains the maximum allowable input string plus a terminating null.
MessageDialog	55	$a0 = address of null-terminated string that is the message to user $a1 = the type of message to be displayed: 0: error message, indicated by Error icon 1: information message, indicated by Information icon 2: warning message, indicated by Warning icon 3: question message, indicated by Question icon other: plain message (no icon displayed)	N/A

MessageDialogInt	56	$a0 = address of null-terminated string that is an information-type message to user $a1 = int value to display in string form after the first string	N/A
MessageDialogString	59	$a0 = address of null-terminated string that is an information-type message to user $a1 = address of null-terminated string to display after the first string	N/A

1. print decimal integer

print an integer to standard output (the console).

Argument(s):

$v0 = 1
$a0 = number to be printed

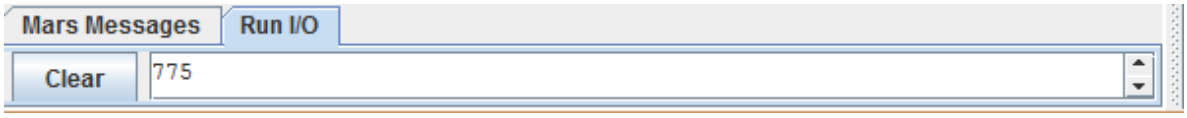
Return value:

none

Example:

```
   li   $v0, 1            # service 1 is print integer
   li   $a0, 0x307        # the interger to be printed is 0x307
   syscall                # execute
```

and result is



2. MessageDialogInt

show an integer to a information-type message dialog.

Argument(s):

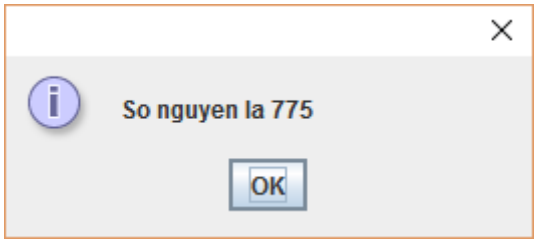
$v0 = 56
$a0 = address of the null-terminated message string
$a1 = int value to display in string form after the first string *Return value:*

none

Example:

```
.data
Message: .asciiz "So nguyen la "
.text
   li   $v0, 56
   la   $a0, Message
   li   $a1, 0x307        # the interger to be printed is 0x307
   syscall                # execute
```

and result is



3. print string

Formatted print to standard output (the console).

Argument(s):

$v0 = 1
$a0 = value to be printed

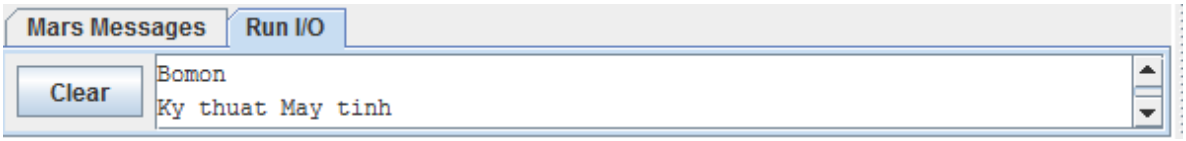
Return value:

none

Example:

```
.data
Message: .asciiz "Bomon \nKy thuat May tinh"
.text
   li  $v0, 4
   la  $a0, Message
   syscall
```

and result is



4. MessageDialogString

Show a string to a information-type message dialog

Argument(s):

$v0 = 59
$a0 = address of the null-terminated message string
$a1 = address of null-terminated string to display

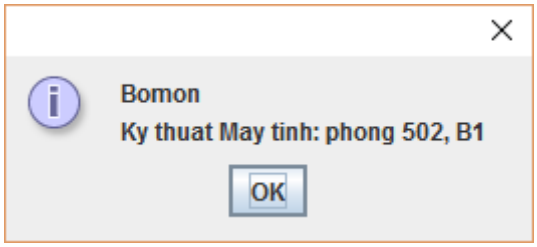
Return value:

none

Example:

```
.data
Message: .asciiz "Bomon \nKy thuat May tinh:"
Address: .asciiz " phong 502, B1"
.text
   li  $v0, 59
   la  $a0, Message
   la  $a1, Address
   syscall
```

and result is



5. read integer

Get a integer from standard input (the keyboard).

Argument(s):

$v0 = 5

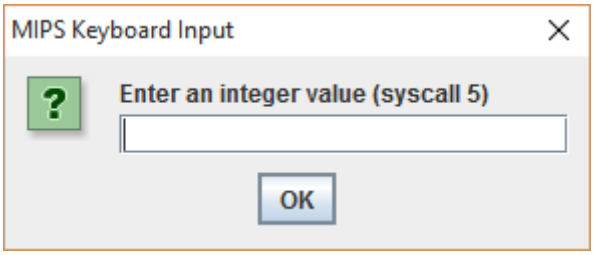
Return value:

$v0 = contains integer read

Example:

```
    li      $v0, 5
    syscall
```

and result is



6. InputDialogInt

Show a message dialog to read a integer with content parser

Argument(s):

$v0 = 51

$a0 = address of the null-terminated message string

Return value:

$a0 = contains int read

$a1 contains status value

0: OK status

-1: input data cannot be correctly parsed

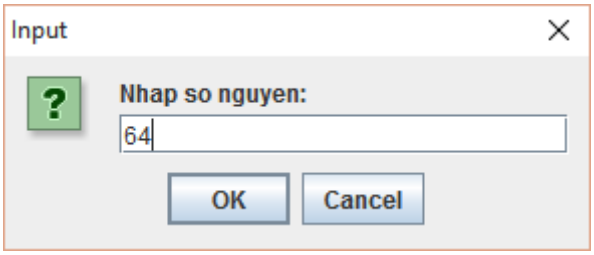
-2: Cancel was chosen

-3: OK was chosen but no data had been input into field

Example:

```
.data
Message: .asciiz "Nhap so nguyen:”
.text
   li    $v0, 51
   la    $a0, Message
   syscall
```

and result is



7. read string

Get a string from standard input (the keyboard).

Argument(s):

$v0 = 8
$a0 = address of input buffer
$a1 = maximum number of characters to read

Return value:

none

Remarks:

For specified length n, string can be no longer than n-1.

- If less than that, adds newline to end.
- In either case, then pads with null byte

Just in special cases:

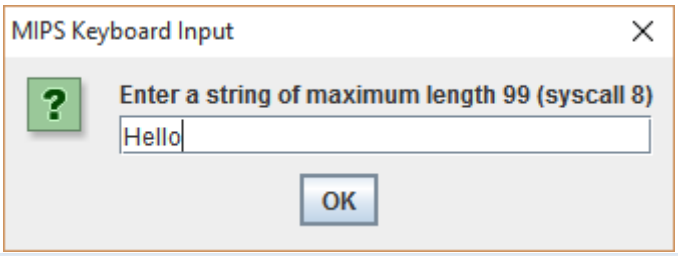
If n = 1, input is ignored and null byte placed at buffer address.

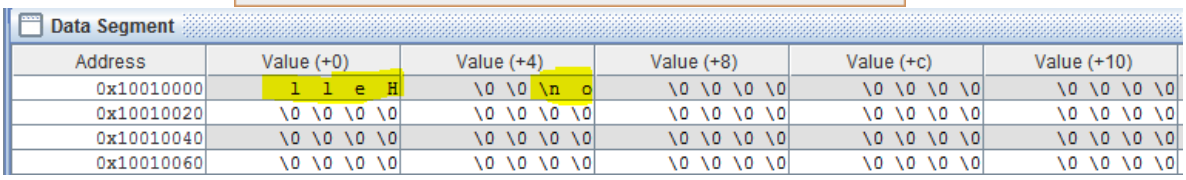
If n < 1, input is ignored and nothing is written to the buffer.

Example:

```
.data
Message: .space 100     # Buffer 100 byte chua chuoi ki tu can
.text
   li   $v0, 8
   la   $a0, Message
   li   $a1, 100
   syscall
```

and result is





Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	l l e H	\0 \0 \n o	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

8. InputDialogString

Show a message dialog to read a string with content parser

Argument(s):

$v0 = 54
$a0 = address of the null-terminated message string
$a1 = address of input buffer
$a2 = maximum number of characters to read

Return value:

$a1 contains status value

0: OK status

-2: OK was chosen but no data had been input into field.
No change to buffer.

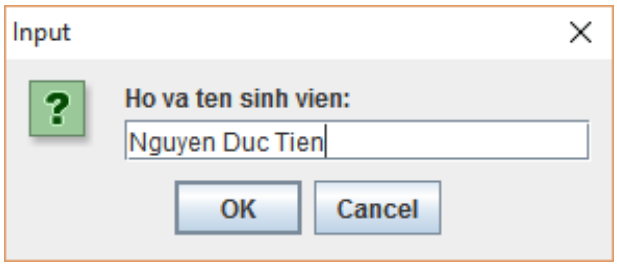
-3: OK was chosen but no data had been input into field

-4: length of the input string exceeded the specified maximum. Buffer contains the maximum allowable input string plus a terminating null.

Example:

```
.data
Message: .asciiz "Ho va ten sinh vien:"
string:  .space   100
.text
   li    $v0, 54
   la    $a0, Message
   la    $a1, string
   la    $a2, 100
   syscall
```

and result is



9. print character

Print a character to standard output (the console).

Argument(s):

$v0 = 11
$a0 = character to print (at the lowest significant byte)

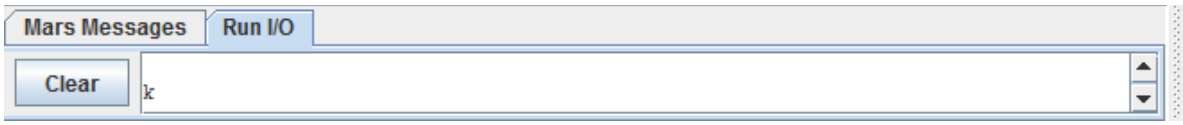
Return value:

none

Example:

```
   li  $v0, 11
   li  $a0, 'k'
   syscall
```

and result is



10. read character

Get a character from standard output (the keyboard).

Argument(s):

$v0 = 12

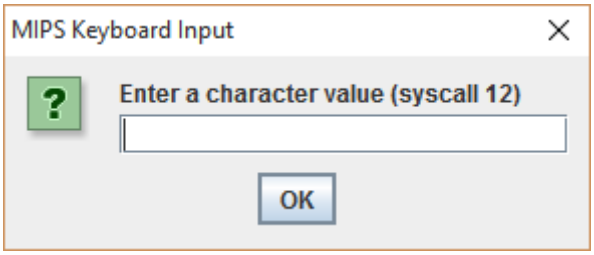
Return value:

$v0 contains character read

Example:

```
   li  $v0, 12
   syscall
```

and result is



11. ConfirmDialog

Show a message bog with 3 button: Yes | No | Cancel

Argument(s):

$v0 = 50
$a0 = address of the null-terminated message string

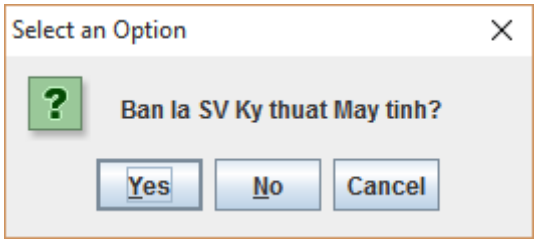
Return value:

$a0 = contains value of user-chosen option
0: Yes
1: No
2: Cancel

Example:

```
.data
Message: .asciiz "Ban la SV Ky thuat May tinh?"
.text
   li    $v0, 50
   la    $a0, Message
   syscall
```

and result is



12. MessageDialog

Show a message bog with icon and button OK only

Argument(s):

$v0 = 55
$a0 = address of the null-terminated message string
$a1 = the type of message to be displayed:
0: error message, indicated by Error icon
1: information message, indicated by Information icon
2: warning message, indicated by Warning icon
3: question message, indicated by Question icon
other: plain message (no icon displayed)

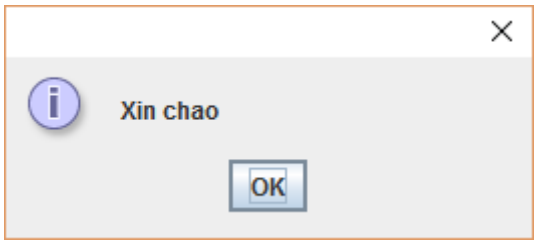
Return value:

none

Example:

```
.data
Message: .asciiz "Xin chao"
.text
   li    $v0, 55
   la    $a0, Message
   syscall
```

and result is



13. MIDI out

Make a sound

Argument(s):

$v0 = 31
$a0 = pitch (0-127)
$a1 = duration in milliseconds
$a2 = instrument (0-127)
$a3 = volume (0-127)

Return value:

Generate tone and return immediately

Example:

```
   li $v0, 33
   li $a0, 42     #pitch
   li $a1, 2000   #time
   li $a2, 0      #musical instrusment
   li $a3, 212    #volume
```

14. MIDI out synchronous

Make a sound

Argument(s):

$v0 = 33

$a0 = pitch (0-127)
$a1 = duration in milliseconds
$a2 = instrument (0-127)
$a3 = volume (0-127)

Return value:

Generate tone and return upon tone completion

Example:

```
li $v0, 33
li $a0, 42     #pitch
li $a1, 2000   #time
li $a2, 0      #musical instrusment
li $a3, 212    #volume
syscall
```

15. Exit

Terminated the software. Make sense that there is no EXIT instruction in the Instruction Set of any processors. Exit is a service belongs to Operating System.

Argument(s):

$v0 = 10

Return value:

none

Example:

```
li        $v0, 10     #exit
syscall
```

16. Exit with code

Terminated the software. Make sense that there is no EXIT instruction in the Instruction Set of any processors. Exit is a service belongs to Operating System.

Argument(s):

$v0 = 17
$a0 = termination result

Return value:

none

Example:

```
li      $v0, 17      # exit
li      $a0, 3       # with error code = 3
syscall
```

Assignments at Home and at Lab

Home Assignment 1

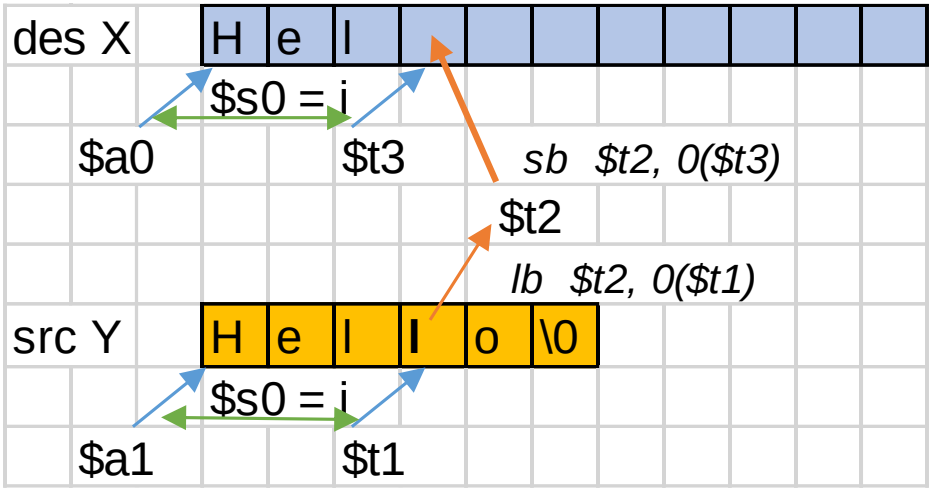
The following simple assembly program will display a welcome string. We use printf function for this purpose. Read this example carefully, pay attention to the way to pass parameters for printf function. Read Mips Lab Enviroment Reference for details.

```
#Laboratory Exercise 5, Home Assignment 1
.data
test: .asciiz "Hello World"
.text
```

```
   li   $v0, 4
   la   $a0, test
   syscall
```

Home Assignment 2

Procedure strcpy copies string y to string x using the null byte termination convention of C. Read this example carefully, try to understand all of this code section.




```
#Laboratory Exercise 5, Home Assignment 2
.data
x: .space 1000                      # destination string x, empty
y: .asciiz "Hello"                  # source string y

.text
strcpy:
      add    $s0,$zero,$zero          #s0 = i=0
L1:
      add    $t1,$s0,$a1              #t1 = s0 + a1 = i + y[0]
                                      #   = address of y[i]
      lb     $t2,0($t1)               #t2 = value at t1 = y[i]
      add    $t3,$s0,$a0              #t3 = s0 + a0 = i + x[0]
                                      #   = address of x[i]
      sb     $t2,0($t3)               #x[i]= t2 = y[i]
      beq    $t2,$zero,end_of_strcpy #if y[i]==0, exit
      nop
      addi   bb$s0,$s0,1                #s0=s0 + 1 <-> i=i+1
      j      L1                       #next character
      nop
end_of_strcpy:
```

Home Assignment 3

The following program count the length of a null-terminated string. Read this example carefully, analyse each line of code.

```
#Laboratory Exercise 5, Home Assignment 3
.data
string:       .space   50
Message1:     .asciiz "Nhap xau:”
Message2:     .asciiz "Do dai la "
.text
main:
get_string:    # TODO
```

```
get_length:   la   $a0, string          # a0 = Address(string[0])
              xor  $v0, $zero, $zero    # v0 = length = 0
              xor  $t0, $zero, $zero    # t0 = i = 0
check_char:   add  $t1, $a0, $t0        # t1 = a0 + t0
                                        #= Address(string[0]+i)
              lb   $t2, 0($t1)          # t2 = string[i]
              beq  $t2,$zero,end_of_str # Is null char?
              addi $v0, $v0, 1          # v0=v0+1->length=length+1
              addi $t0, $t0, 1          # t0=t0+1->i = i + 1
              j    check_char
end_of_str:
end_of_get_length:
print_length: # TODO
```

Assignment 1

Create a new project to implement the program in Home Assignment 1. Compile and upload to simulator. Run and observe the result. Go to data memory section, check how test string are stored and packed in memory.

Assignment 2

Create a new project to print the sum of two register $s0 and $s1 according to this format:

"The sum of (s0) and (s1) is (result)"

Assignment 3

Create a new project to implement the program in Home Assignment 2. Add more instructions to assign a test string for y variable, and implement *strcpy* function. Compile and upload to simulator. Run and observe the result.

Assignment 4

Accomplish the Home Assignment 3 with syscall function to get a string from dialog, and show the length to message dialog.

Assignment 5

Write a program that let user input a string. Input process will be terminated when user press Enter or then length of the string exceed 20 characters. Print the reverse string.

Conclusions

Before you pass the laboratory exercise, think about the questions below:

- What the difference between the string in C and Java?
- In C, with 8 bytes, how many characters that we can store?
- In Java, with 8 bytes, how many characters that we can store?

Laboratory Exercise 6

Array and Pointer

Goals

After this laboratory Exercise, you should understand how array and pointer are represented and you will be able to differentiate index and pointer using in stepping through an array of list.

Literature

Patterson, Henessy (COD): section 2.8, 2.13

Preparation

Before you start the exercise, you should review the textbook, section 6.1 and read this laboratory carefully.

Array and Pointer

In a wide variety of programming tasks, it becomes necessary to step through an array or list, examining each of its elements in turn. For example, to determine the largest value in a list of integers, every element of the list mus be examined. There are two basic ways of accomplishing this:

1. Index: use a register that holds the index I and increment the register in each step to effect moving from element i of the list to element i+1
2. Pointer: use a register that points to (holds the address of) the list elements being examined and update it in each step to point to the next element.

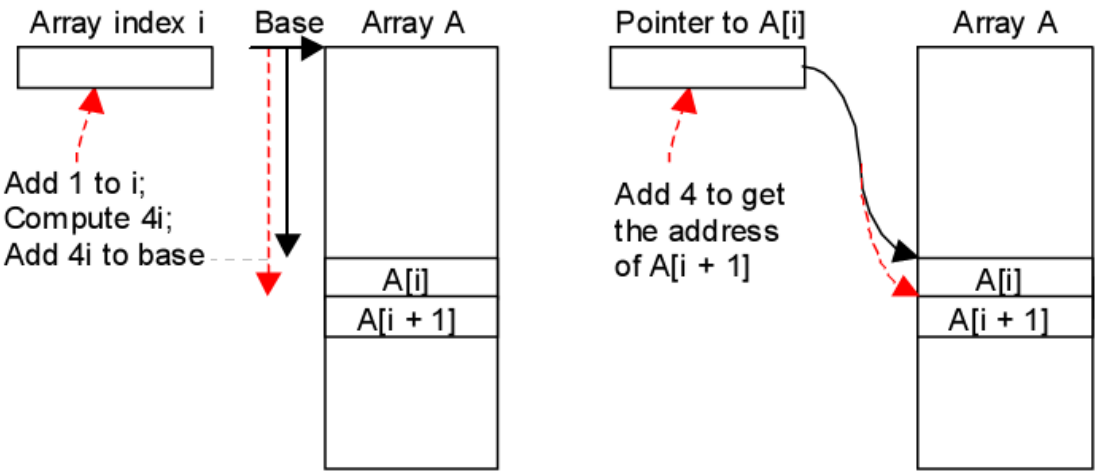
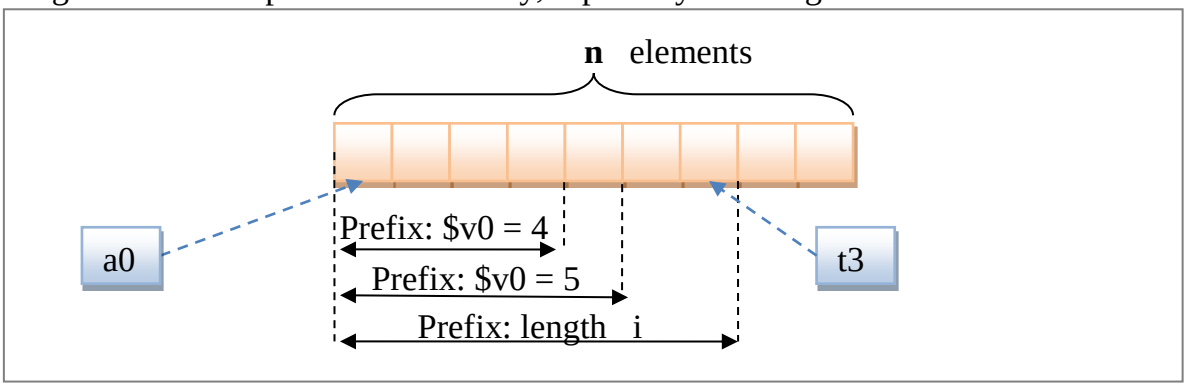


Figure 4. Using the indexing method and the pointer updating method to step through the elements of an array

Assignments at Home and at Lab

Home Assignment 1

Consider a list of integers of length n. A prefix of length I for the given list consist of the first i integers in the list, where 0 <= i <=n. A maximum-sum prefix, as the name implies, is a prefix for which the sum of elements is the largest among all prefixes. For example, if the list is (2,-3,2,5,-4), its maximum-sum prefix consists of the first four elements and the associated sum is 2-3+2+5=6; no other prefix of the given list has a larger sum. The following procedure uses indexing method to find the maximum-sum prefix in a list of integers. Read this procedure carefully, especially indexing method.

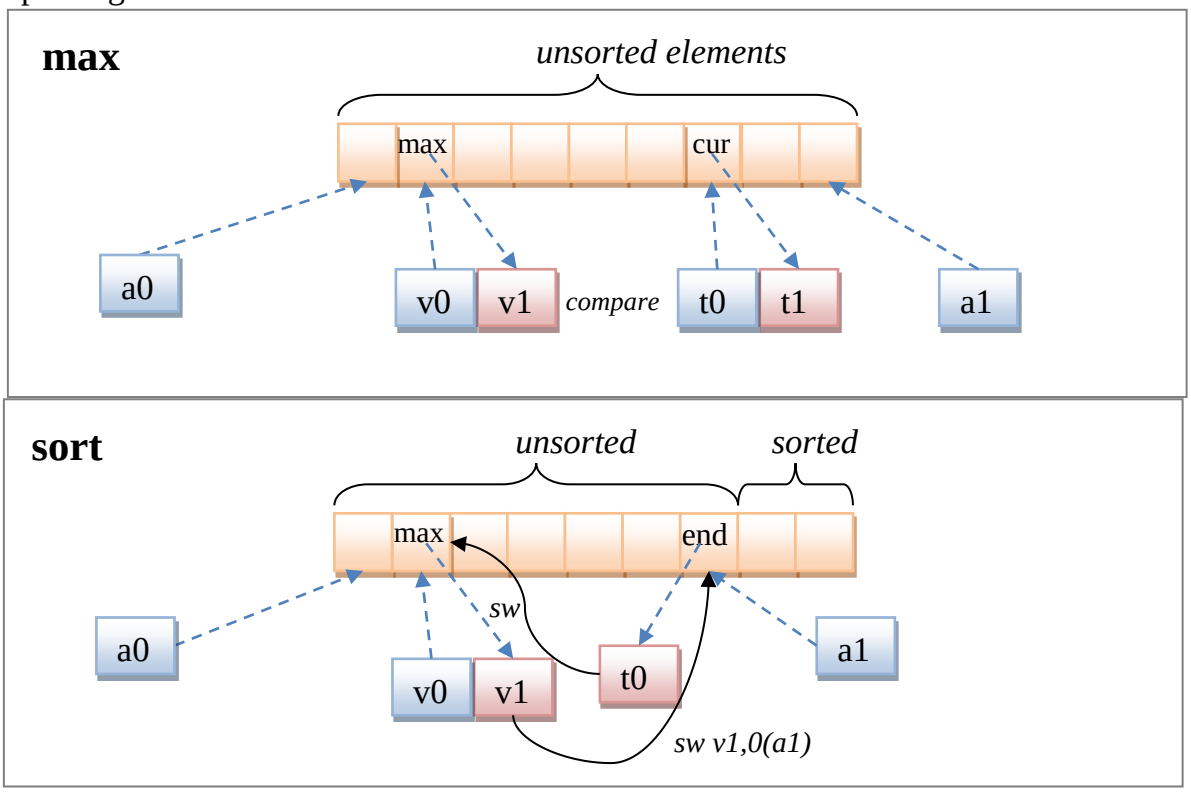



```
.data
A: .word -2, 6, -1, 3, -2

.text
main:        la     $a0,A
             li     $a1,5
             j      mspfx
             nop
continue:
lock:        j  lock
             nop
end_of_main:

#-----------------------------------------------------------------
#Procedure mspfx
# @brief    find the maximum-sum prefix in a list of integers
# @param[in] a0   the base address of this list(A) need to be
processed
# @param[in] a1   the number of elements in list(A)
# @param[out] v0  the length of sub-array of A in which max sum
reachs.
# @param[out] v1  the max sum of a certain sub-array
#-----------------------------------------------------------------
#Procedure mspfx
#function: find the maximum-sum prefix in a list of integers
#the base address of this list(A) in $a0 and the number of
#elements is stored in a1
mspfx:  addi      $v0,$zero,0 #initialize length in $v0 to 0
        addi      $v1,$zero,0 #initialize max sum in $v1to 0
        addi      $t0,$zero,0 #initialize index i in $t0 to 0
        addi      $t1,$zero,0 #initialize running sum in $t1 to 0
loop: add   $t2,$t0,$t0       #put 2i in $t2
      add   $t2,$t2,$t2       #put 4i in $t2
      add   $t3,$t2,$a0       #put 4i+A (address of A[i]) in $t3
```

```
      lw    $t4,0($t3)          #load A[i] from mem(t3) into $t4
      add   $t1,$t1,$t4         #add A[i] to running sum in $t1
      slt   $t5,$v1,$t1         #set $t5 to 1 if max sum < new sum
      bne   $t5,$zero,mdfy      #if max sum is less, modify results
      j     test                #done?
mdfy: addi  $v0,$t0,1           #new max-sum prefix has length i+1
      addi  $v1,$t1,0           #new max sum is the running sum
test: addi  $t0,$t0,1           #advance the index i
      slt   $t5,$t0,$a1         #set $t5 to 1 if i<n
      bne   $t5,$zero,loop      #repeat if i<n
done: j     continue
mspfx_end:
```

Home Assignment 2

A given list of n numbers can be sorted in ascending order as follows. Find a largest number in the list (there may be more than one) and swap it with the last element in the list. The new last element is now in its proper position in sorted order. Now, sort the remaining n-1 elements using the same step repeatedly. When only one element is left, sorting is complete. This method is known as selection sort. This example demonstrates this kind of sorting using pointer updating method. Read this procedure carefully and pay attention to pointer updating method.




```
.data
A: .word 7, -2, 5, 1, 5,6,7,3,6,8,8,59,5
Aend: .word

.text
main:       la   $a0,A            #$a0 = Address(A[0])
```

```
            la   $a1,Aend
            addi $a1,$a1,-4      #$a1 = Address(A[n-1])
            j    sort            #sort
after_sort: li   $v0, 10         #exit
            syscall
end_main:
#-----------------------------------------------------------------
#procedure sort (ascending selection sort using pointer)
#register usage in sort program
#$a0 pointer to the first element in unsorted part
#$a1 pointer to the last element in unsorted part
#$t0 temporary place for value of last element
#$v0 pointer to max element in unsorted part
#$v1 value of max element in unsorted part
#-----------------------------------------------------------------
sort:       beq     $a0,$a1,done    #single element list is sorted
            j       max             #call the max procedure
after_max:  lw      $t0,0($a1)      #load last element into $t0
            sw      $t0,0($v0)      #copy last element to max location
            sw      $v1,0($a1)      #copy max value to last element
            addi    $a1,$a1,-4      #decrement pointer to last element
            j       sort            #repeat sort for smaller list
done:       j       after_sort

#-------------------------------------------------------------------
---
#Procedure max
#function: fax the value and address of max element in the list
#$a0 pointer to first element
#$a1 pointer to last element
#-------------------------------------------------------------------
---
max:
      addi  $v0,$a0,0         #init max pointer to first element
      lw    $v1,0($v0)        #init max value to first value
      addi  $t0,$a0,0         #init next pointer to first
loop:
      beq   $t0,$a1,ret       #if next=last, return
      addi  $t0,$t0,4         #advance to next element
      lw    $t1,0($t0)        #load next element into $t1
      slt   $t2,$t1,$v1       #(next)<(max) ?
      bne   $t2,$zero,loop    #if (next)<(max), repeat
      addi  $v0,$t0,0         #next element is new max element
      addi  $v1,$t1,0         #next value is new max value
      j     loop              #change completed; now repeat
ret:
      j     after_max
```

Assignment 1

Create a new project to implement procedure in Home Assignment 1. Add code the main program and initialize data for the integer list. Compile and upload to simulator. Run this program step by step, observe the process of explore each element of the integer list using indexing method.

Assignment 2

Create a new project to implement procedure in Home Assignment 2. Add code the main program and initialize data for the integer list. Compile and upload to simulator. Run this program step by step, observe the process of explore each element of the integer list using pointer updating method.

Assignment 3

Write a procedure to implement bubble sort algorithm.

Assignment 4

Write a procedure to implement insertion sort algorithm.

Conclusions

Before you pass the laboratory exercise, think about the questions below:

- What the advantage and disadvantage of two methods: indexing and updating pointer?

Laboratory Exercise 7

Procedure calls, stack and parameters

Goals

After this laboratory exercise, you should understand how to invoke a procedure and the mechanism of stack. In addition, you will be able to write some procedures that use stack to pass parameters and return results also.

Literature

Behrooz Parhami (CAMS): Section 6.1

Preparation

Before you start the exercise, you should review the textbook, section 6.1 and read this laboratory carefully.

Procedure calls

A procedure is a subprogram that when called (initiated, invoked) performs a specific task, perhaps leading to one or more results, based on the input parameters (arguments) with which it is provided and returns to the point of call, having perturbed nothing else. In assembly language, a procedure is associated with a symbolic name that denotes its starting address. The jal instruction in MIPS is inteded specifically for procedure calls: it performs the control transfer (unconditional jump) to the starting address of the procedure, while also saving the return address in register ra.

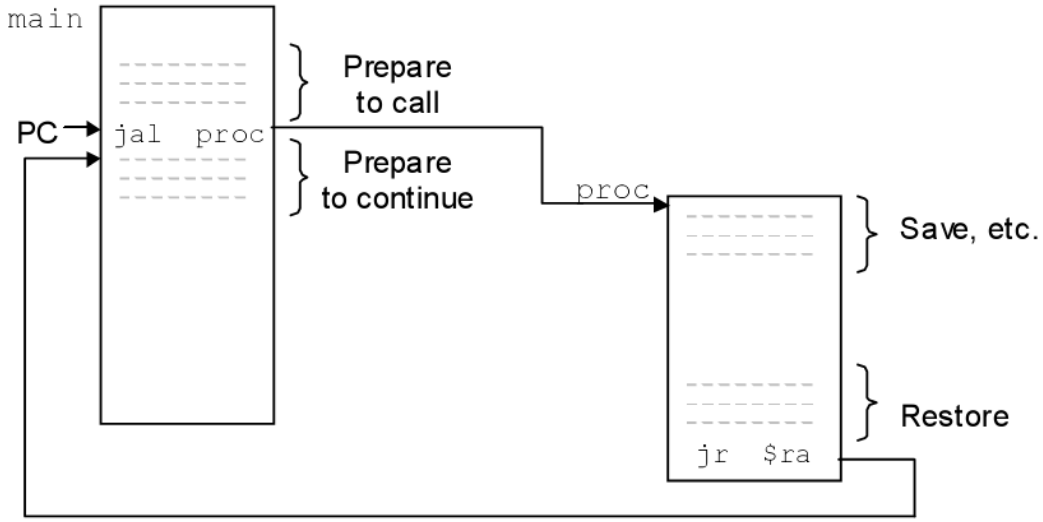


Figure 5. Relationship between the main program and a procedure

Assignments at Home and at Lab

Home Assignment 1

The following program uses abs procedure. This procedure is used to find the absolute value in an integer. This program uses two registers, $a0 for input argument and $v0 for result. You should read this example carefully, pay attention to how to write and invoke a procedure.

```
#Laboratory Exercise 7 Home Assignment 1
.text
main:   li      $a0,-45         #load input parameter
        jal     abs             #jump and link to abs procedure
        nop
        add     $s0, $zero, $v0
        li      $v0,10          #terminate
        syscall
endmain:
#-------------------------------------------------------------------
# function abs
# param[in]    $a1     the interger need to be gained the absolute
value
# return       $v0     absolute value
#-------------------------------------------------------------------
abs:
      sub   $v0,$zero,$a1     #put -(a0) in v0; in case (a0)<0

      bltz  $a1,done          #if (a0)<0 then done
      nop
      add   $v0,$a1,$zero     #else put (a0) in v0
done:
      jr    $ra
```

Home Assignment 2

In this example, procedure max has the function to find the largest elements in three of integers. These intergers are passed to max procedure through $a0, $a1 and $a2 registers. Read this example carefully and try to explain each line of code.

```
#Laboratory Exercise 7, Home Assignment 2
.text
main:  li    $a0,2            #load test input
       li    $a1,6
       li    $a2,9
       jal   max              #call max procedure
       nop
endmain:
#-------------------------------------------------------------------
-
#Procedure max: find the largest of three integers
#param[in]  $a0  integers
#param[in]  $a1  integers
#param[in]  $a2  integers
#return     $v0   the largest value
#-------------------------------------------------------------------
-
max:  add     $v0,$a0,$zero   #copy (a0) in v0; largest so far
      sub     $t0,$a1,$v0     #compute (a1)-(v0)
```

```
      bltz    $t0,okay        #if (a1)-(v0)<0 then no change
      nop
      add     $v0,$a1,$zero   #else (a1) is largest thus far
okay: sub     $t0,$a2,$v0     #compute (a2)-(v0)
      bltz    $t0,done        #if (a2)-(v0)<0 then no change
      nop
      add     $v0,$a2,$zero   #else (a2) is largest overall
done: jr      $ra             #return to calling program
```

Home Assignment 3

The following program demonstrates the push and pop operations with stack. The value of two register $s0 and $s1 will be swap using stack. Read this example carefully, pay attention to adjust operation and the order of push and pop (sw and lw instructions).

```
#Laboratory Exercise 7, Home Assignment 3
.text
push: addi   $sp,$sp,-8      #adjust the stack pointer
      sw     $s0,4($sp)      #push $s0 to stack
      sw     $s1,0($sp)      #push $s1 to stack
work: nop
      nop
      nop
pop:  lw     $s0,0($sp)      #pop from stack to $s0
      lw     $s1,4($sp)      #pop from stack to $s1
      addi   $sp,$sp,8       #adjust the stack pointer
```

Parameters and results

In this section, we answer the following unresolved questions relating to procedures

1. How to pass more than four input parameters to a procedure or receive more than two results from it?
2. Where does a procedure save its own parameters and immediate results when calling another procedure (nested calls)?

The stack is used in both cases. Before a procedure call, the calling program pushed the contents of any register that need to be save onto the top of the stack and follow these with any additional arguments for the procedure. The procedure can access these arguments in the stack. After the procedure terminates, the calling procedure expects to find the stack pointer undisturbed, thus allowing it to restore the save registers to their original states and proceed with its own computations. Thus, a procedure that uses the stack by modifying the stack pointer must save the content of the stack pointer at the outset and , at the end, restore sp to it original state. This is done by copying the stack pointer into the frame pointer register fp. Before doing this, however, the old contents of the frame pointer must be saved. Hence, while a procedure is executing, fp may hold the content of fp right before it was called; fp and sp together “frame” the area of the stack that is in use by the current procedure.

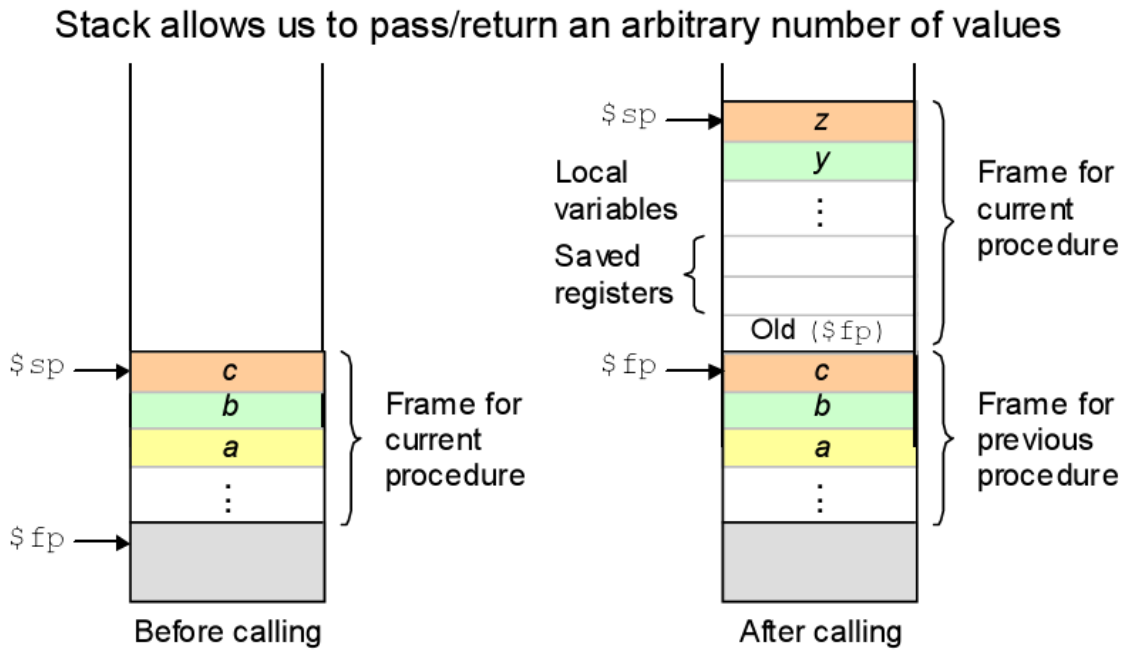


Figure 6. Use of the stack by a procedure

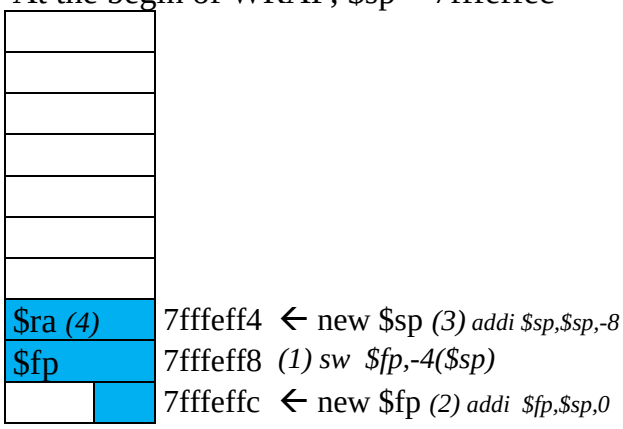
Home Assignment 4

The following program is a recursive procedure for computing n!. Read this program carefully and pay attention to register sp and fp at the start and the end of each recursive step.

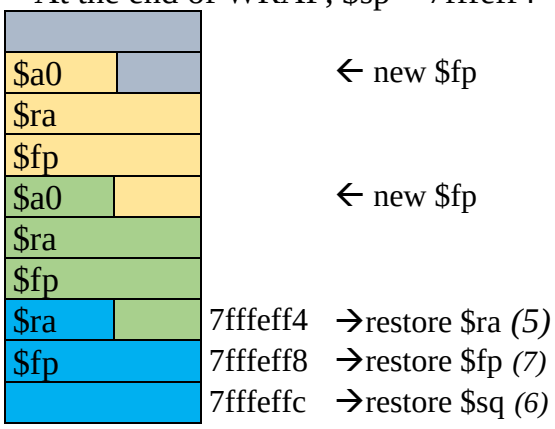
```
#Laboratory Exercise 7, Home Assignment 4
.data
Message: .asciiz  "Ket qua tinh giai thua la: "

.text
main:  jal    WARP

print: add    $a1, $v0, $zero # $a0 = result from N!
       li     $v0, 56
       la     $a0, Message
       syscall
quit:  li     $v0, 10         #terminate
       syscall
endmain:

#-----------------------------------------------------------------------
-
#Procedure WARP: assign value and call FACT
#-----------------------------------------------------------------------
-
WARP:   sw     $fp,-4($sp)    #save frame pointer (1)
        addi   $fp,$sp,0      #new frame pointer point to the top (2)
        addi   $sp,$sp,-8     #adjust stack pointer (3)
        sw     $ra,0($sp)     #save return address (4)

        li     $a0,6          #load test input N
        jal    FACT           #call fact procedure
        nop

        lw     $ra,0($sp)     #restore return address (5)
        addi   $sp,$fp,0      #return stack pointer  (6)
```

```
        lw      $fp,-4($sp)    #return frame pointer   (7)
        jr      $ra
wrap_end:

#-----------------------------------------------------------------
-
#Procedure FACT: compute N!
#param[in]  $a0  integer N
#return     $v0   the largest value
#-----------------------------------------------------------------
-
FACT:   sw      $fp,-4($sp)         #save frame pointer
        addi    $fp,$sp,0           #new frame pointer point to stack's
top
        addi    $sp,$sp,-12         #allocate space for $fp,$ra,$a0 in
stack
        sw      $ra,4($sp)          #save return address
        sw      $a0,0($sp)          #save $a0 register

        slti    $t0,$a0,2           #if input argument N < 2
        beq     $t0,$zero,recursive#if it is false ((a0 = N) >=2)
        nop
        li      $v0,1               #return the result N!=1
        j       done
        nop
recursive:
        addi    $a0,$a0,-1          #adjust input argument
        jal     FACT                #recursive call
        nop
        lw      $v1,0($sp)          #load a0
        mult    $v1,$v0             #compute the result
        mflo    $v0
done:   lw      $ra,4($sp)          #restore return address
        lw      $a0,0($sp)          #restore a0
        addi    $sp,$fp,0           #restore stack pointer
        lw      $fp,-4($sp)         #restore frame pointer
        jr      $ra                 #jump to calling
fact_end:
```

At the begin of WRAP, $sp = 7fffeffec



At the end of WRAP, $sp = 7fffeff4



Assignment 1

Create a new project to implement the program in Home Assignment 1. Compile and upload to simulator. Change input parameters and observe the memory when run the program step by step. Pay attention to register pc, $rato clarify invoking procedure process (Refer to figure 7).

Assignment 2

Create a new project to implement the program in Home Assignment 2. Compile and upload to simulator. Change input parameters (register a0, a1, a2) and observe the memory when run the program step by step. Pay attention to register pc, $rato clarify invoking procedure process (Refer to figure 7).

Assignment 3

Create a new project to implement the program in Home Assignment 3. Compile and upload to simulator. Pass test value to registers $s0 and $s1, observe run process, pay attention to stack pointer. Goto memory space that pointed by sp register to view push and pop operations in detail.

Assignment 4

Create a new project to implement the program in Home Assignment 4. Compile and upload to simulator. Pass test input through register a0, run this program and test result in register v0. Run this program in step by step mode, observe the changing of register pc, ra, sp and fp. Draw the stack through this recursive program in case of n=3 (compute 3!).

Assignment 5

Write a procedure to find the largest, the smallest and these positions in a list of 8 elements that are stored in regsiters $s0 through s7. For example:

Largest: 9,3 -> The largest element is stored in $s3, largest value is 9

Smallest: -3,6 -> The smallest element is stored in s6, smallest value is -3

Tips: using stack to pass arguments and return results.

Conclusions

Before you pass the laboratory exercise, think about the questions below:

- What registers that the Caller need to save by convention?
- What registers that the Callee need to save by convention?
- In push label of Home Assignment 3, could we change the order of adjust stack and store word operations? If yes, what should we have to modify?
- What is stack pointer?
- What is frame pointer?

Laboratory Exercise 8, 9

Mini-projects

Goals

Review elementary knowledge about assembly programming to solve whole problems. Learn how to work in a group to promote group spirit.

Time

Week 8	: stay at home and do mid-term practice exercises
Mid-term Week	: stay at home and do mid-term practice exercises
Week 9	: teachers check the practice exercises

Laboratory Exercise 10

Goals

After this laboratory exercise, you should understand the method to control pheripheral devices via simulators.

Literature

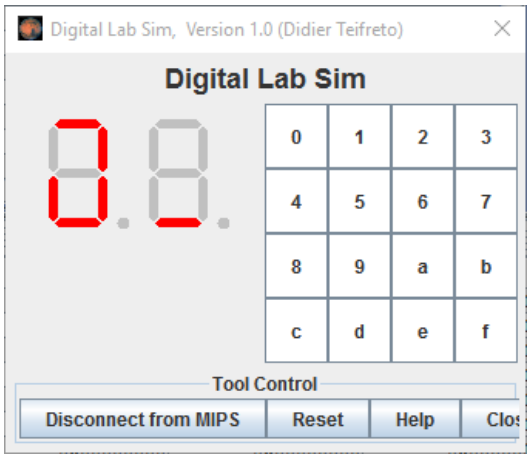
How does the CPU communicate with input and output devices such as the monitor or keyboard?

There are several ways. Intel machines have special instructions named in and out that communicate with I/O ports. These instructions are usually disabled for ordinary users, but they are used internally for communicating with I/O devices. This is called port-mapped I/O. However, we are going to look at a di_erent method in which I/O devices have access to memory. The CPU can place data in memory that can be read by the I/O devices; likewise, the I/O devices can place data in memory for the CPU. This is called memory-mapped I/O or MMIO. (For more information, see P&H page 588 or Appendix B.8, or look it up online!)

Assignments at Home and at Lab

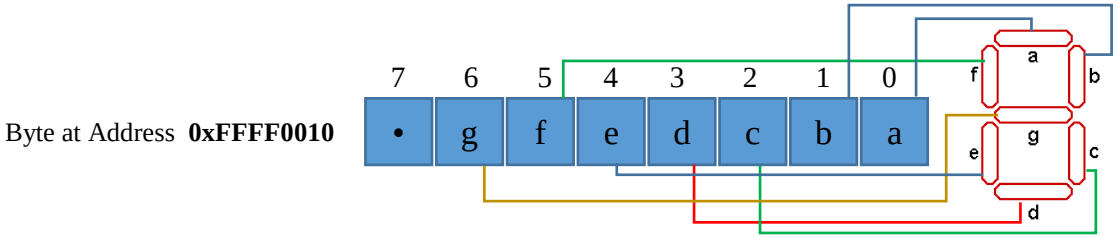
Home Assignment 1 – LED PORT

Write a program using assembly language to show numbers from 0 to F to the 7-seg led.



To view the 7-segs, at the menu bar, click /Tools/Digi Lab Sim

Click Help to understand how to turn on the 7-seg led.

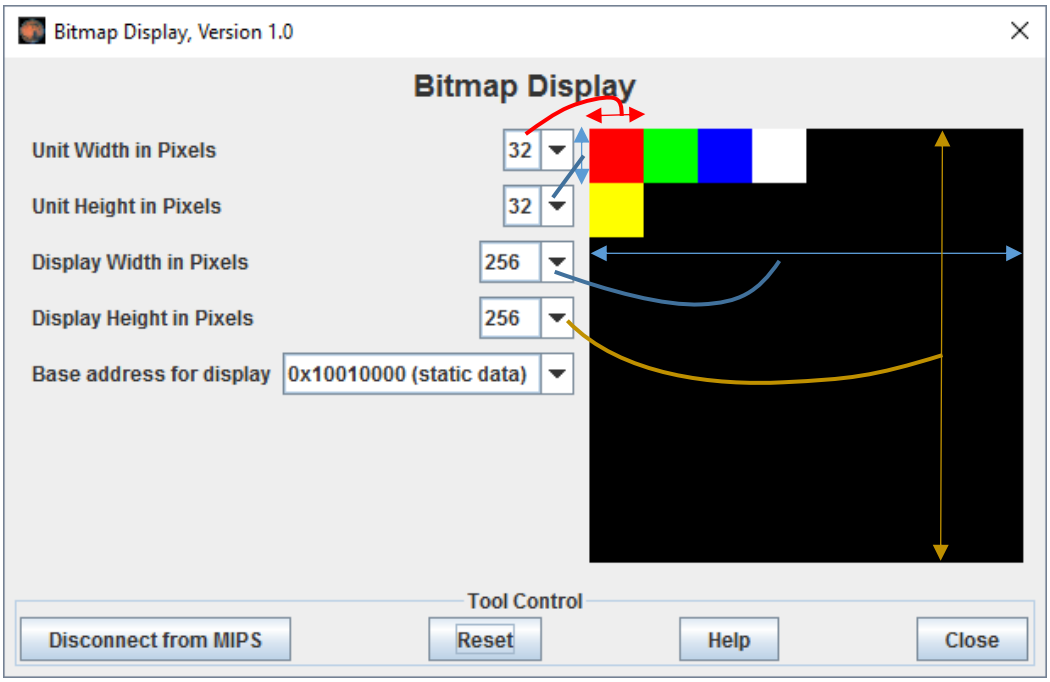



```
.eqv SEVENSEG_LEFT    0xFFFF0011    # Dia chi cua den led 7 doan trai.
                                    #     Bit 0 = doan a;
                                    #     Bit 1 = doan b; ...
                                    #     Bit 7 = dau .
.eqv SEVENSEG_RIGHT   0xFFFF0010    # Dia chi cua den led 7 doan phai

.text
main:
          li    $a0,  0x8               # set value for segments
          jal   SHOW_7SEG_LEFT          # show
          nop
          li    $a0,  0x1F              # set value for segments
          jal   SHOW_7SEG_RIGHT         # show
          nop
exit:     li    $v0, 10
          syscall
endmain:

#---------------------------------------------------------------
# Function  SHOW_7SEG_LEFT : turn on/off the 7seg
# param[in]  $a0   value to shown
# remark     $t0 changed
#---------------------------------------------------------------
SHOW_7SEG_LEFT:  li   $t0,  SEVENSEG_LEFT # assign port's address
                 sb   $a0,  0($t0)        # assign new value
                 nop
                 jr   $ra
                 nop

#---------------------------------------------------------------
# Function  SHOW_7SEG_RIGHT : turn on/off the 7seg
# param[in]  $a0   value to shown
# remark     $t0 changed
#---------------------------------------------------------------
SHOW_7SEG_RIGHT: li   $t0,  SEVENSEG_RIGHT # assign port's address
                 sb   $a0,  0($t0)         # assign new value
                 nop
                 jr   $ra
                 nop
```

Home Assignment 2 – BITMAP DISPLAY

Bitmap Display like the graphic monitor, in which Windows OS draws windows, start button… In order to to that, developer should calculate color of all bitmap pixels on thee screen and store these color value to the screen memory. Wherever we change a value in screen memory, the color of the respective pixel on the screen will be changed.

In MARS, in menu bar, click Tools / Bitmap Display to open the screen simulator



b

0	R	G	B	
00	FF	00	00	0x10010000 - pixel 0
00	00	FF	00	0x10010004 - pixel 1
00	00	00	00	0x10010008 - pixel 2
00	FF	FF	FF	0x1001000C - pixel 3

Each rectangular unit on the display represent s one memory word in a contiguous address space starting with the specified base address (in above figure, base address is 0x10010000)

Value stored in that word will be interpreted as a 24-bit RGB

```
.eqv MONITOR_SCREEN 0x10010000   #Dia chi bat dau cua bo nho man hinh
.eqv RED            0x00FF0000   #Cac gia tri mau thuong su dung
.eqv GREEN          0x0000FF00
.eqv BLUE           0x000000FF
.eqv WHITE          0x00FFFFFF
.eqv YELLOW         0x00FFFF00
.text
   li $k0, MONITOR_SCREEN        #Nap dia chi bat dau cua man hinh

   li $t0, RED
   sw  $t0, 0($k0)
   nop

   li $t0, GREEN
   sw  $t0, 4($k0)
   nop

   li $t0, BLUE
   sw  $t0, 8($k0)
   nop

   li $t0, WHITE
   sw  $t0, 12($k0)
```

```
   nop

   li $t0, YELLOW
   sw  $t0, 32($k0)
   nop

   li $t0, WHITE
   lb $t0, 42($k0)
   nop
```

Home Assignment 3 – MARSBOT RIDER

The MarsBot is a virtual robot that has a very simple mode of operation. It travels around in two-dimensional space, optionally leaving a trail, or track, as it goes. It uses five words in memory:[3]

Name	Address	Meaning
HEADING	0xffff8010	Integer: An angle between 0 and 359
LEAVETRACK	0xffff8020	Boolean (0 or non-0): whether or not to leave a track
WHEREX	0xffff8030	Integer: Current x-location of the MarsBot
WHEREY	0xffff8040	Integer: Current y-location of the MarsBot
MOVING	0xffff8050	Boolean: whether or not to move

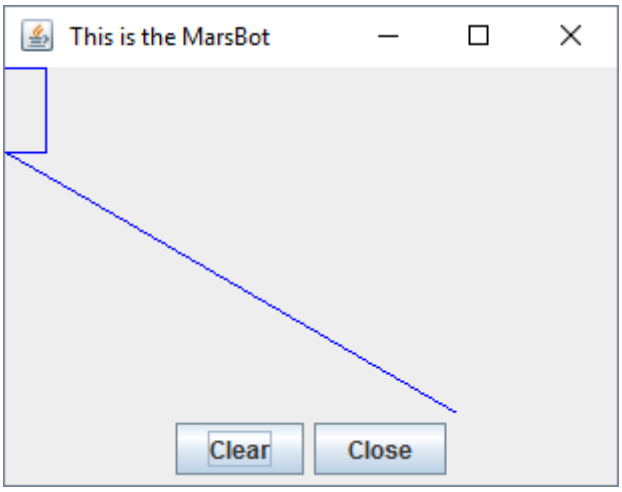
The CPU can place commands in the HEADING, LEAVETRACK, and MOVE locations; the robot can then change its direction of travel (using the HEADING value), turn on or turn off the pen" drawing the line (using the LEAVE-TRACK value), and can halt or resume moving (using the MOVING value).

```
.eqv  HEADING    0xffff8010    # Integer: An angle between 0 and 359
                               # 0 : North (up)
                               # 90: East (right)
                               # 180: South (down)
                               # 270: West  (left)
.eqv  MOVING     0xffff8050    # Boolean: whether or not to move
.eqv  LEAVETRACK 0xffff8020    # Boolean (0 or non-0):
                               #    whether or not to leave a track
.eqv  WHEREX     0xffff8030    # Integer: Current x-location of
MarsBot
.eqv  WHEREY     0xffff8040    # Integer: Current y-location of
MarsBot


.text
main:   jal     TRACK           # draw track line
        nop
        addi    $a0, $zero, 90  # Marsbot rotates 90* and start
running
        jal     ROTATE
        nop
        jal     GO
        nop
```

[3] http://cs.allegheny.edu/~rroos/cs210f2013

```
sleep1: addi    $v0,$zero,32    # Keep running by sleeping in 1000 ms
        li      $a0,1000
        syscall

        jal     UNTRACK         # keep old track
        nop
        jal     TRACK           # and draw new track line
        nop

goDOWN: addi    $a0, $zero, 180 # Marsbot rotates 180*
        jal     ROTATE
        nop

sleep2: addi    $v0,$zero,32    # Keep running by sleeping in 2000 ms
        li      $a0,2000
        syscall

        jal     UNTRACK         # keep old track
        nop
        jal     TRACK           # and draw new track line
        nop

goLEFT: addi  $a0, $zero, 270   # Marsbot rotates 270*
        jal   ROTATE
        nop

sleep3: addi    $v0,$zero,32    # Keep running by sleeping in 1000 ms
        li      $a0,1000
        syscall

        jal     UNTRACK         # keep old track
        nop
        jal     TRACK           # and draw new track line
        nop

goASKEW:addi  $a0, $zero, 120   # Marsbot rotates 120*
        jal   ROTATE
        nop

sleep4: addi    $v0,$zero,32    # Keep running by sleeping in 2000 ms
        li      $a0,2000
        syscall

        jal     UNTRACK         # keep old track
        nop
        jal     TRACK           # and draw new track line
        nop
end_main:

#-----------------------------------------------------------
# GO procedure, to start running
# param[in]    none
#-----------------------------------------------------------
GO:     li    $at, MOVING     # change MOVING port
        addi  $k0, $zero,1    # to  logic 1,
        sb    $k0, 0($at)     # to start running
        nop
        jr    $ra
        nop
#-----------------------------------------------------------
# STOP procedure, to stop running
# param[in]    none
#-----------------------------------------------------------
```

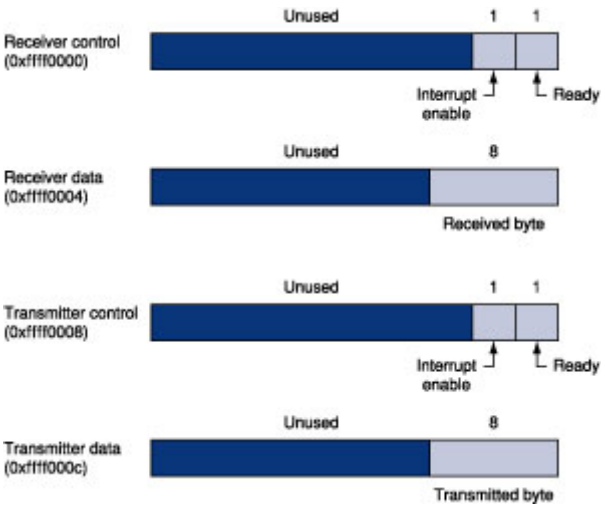
```
STOP:   li    $at, MOVING     # change MOVING port to 0
        sb    $zero, 0($at)   # to stop
        nop
        jr    $ra
        nop
#-----------------------------------------------------------
# TRACK procedure, to start drawing line
# param[in]    none
#-----------------------------------------------------------
TRACK:  li    $at, LEAVETRACK # change LEAVETRACK port
        addi  $k0, $zero,1    # to  logic 1,
        sb    $k0, 0($at)     # to start tracking
        nop
        jr    $ra
        nop
#-----------------------------------------------------------
# UNTRACK procedure, to stop drawing line
# param[in]    none
#-----------------------------------------------------------
UNTRACK:li    $at, LEAVETRACK # change LEAVETRACK port to 0
        sb    $zero, 0($at)   # to stop drawing tail
        nop
        jr    $ra
        nop
#-----------------------------------------------------------
# ROTATE procedure, to rotate the robot
# param[in]    $a0, An angle between 0 and 359
#                   0 : North (up)
#                   90: East  (right)
#                  180: South (down)
#                  270: West  (left)
#-----------------------------------------------------------
ROTATE: li    $at, HEADING    # change HEADING port
        sw    $a0, 0($at)     # to rotate robot
        nop
        jr    $ra
        nop
```



Home Assignment 4 – KEYBOARD and DISPLAY MMIO

Use this program to simulate Memory-Mapped I/O (MMIO) for a keyboard input device and character display output device. It may be run either from MARS' Tools menu or as a stand-alone application.

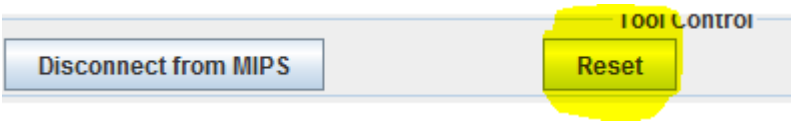
While the tool is connected to MIPS, each keystroke in the text area causes the corresponding ASCII code to be placed in the Receiver Data register (low-order byte of memory word 0xffff0004), and the Ready bit to be set to 1 in the Receiver Control register (low-order bit of 0xffff0000). The Ready bit is automatically reset to 0 when the MIPS program reads the Receiver Data using an 'lw' instruction.

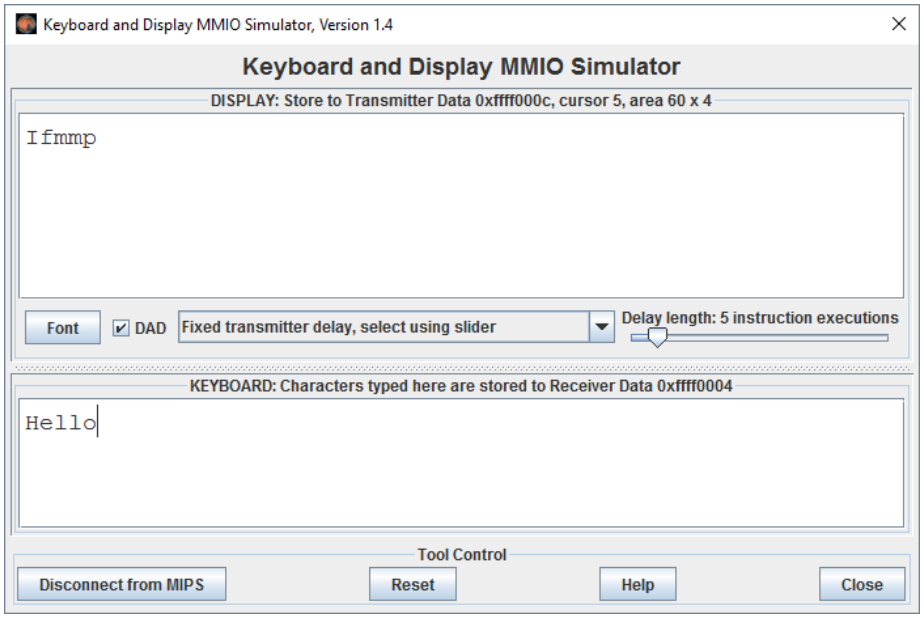



```
.eqv KEY_CODE   0xFFFF0004       # ASCII code from keyboard, 1 byte
.eqv KEY_READY  0xFFFF0000       # =1 if has a new keycode ?
                                 # Auto clear after lw

.eqv DISPLAY_CODE   0xFFFF000C   # ASCII code to show, 1 byte
.eqv DISPLAY_READY  0xFFFF0008   # =1 if the display has already to do
                                 # Auto clear after sw

.text
            li   $k0,  KEY_CODE
            li   $k1,  KEY_READY

            li   $s0, DISPLAY_CODE
            li   $s1, DISPLAY_READY

loop:       nop

WaitForKey: lw   $t1, 0($k1)            # $t1 = [$k1] = KEY_READY
            nop
            beq  $t1, $zero, WaitForKey # if $t1 == 0 then Polling
            nop
            #-----------------------------------------------------
ReadKey:    lw   $t0, 0($k0)            # $t0 = [$k0] = KEY_CODE
            nop
            #-----------------------------------------------------
WaitForDis: lw   $t2, 0($s1)            # $t2 = [$s1] = DISPLAY_READY
             nop
```

```
            beq  $t2, $zero, WaitForDis # if $t2 == 0 then Polling
            nop
            #-----------------------------------------------------
Encrypt:    addi $t0, $t0, 1            # change input key
            #-----------------------------------------------------
ShowKey:    sw $t0, 0($s0)              # show key
            nop
            #-----------------------------------------------------
            j loop
            nop
```

Warning: Must execute as below

1. Click Run
2. Click Reset 



Assignment 1

Create a new project, type in, and build the program of Home Assignment 1. Show different values on LED

Assignment 2

Create a new project, type in, and build the program of Home Assignment 2. Draw something.

Assignment 3

Create a new project, type in, and build the program of Home Assignment 3. Make the Bot run and draw a triangle by tracking

Assignment 4

Create a new project, type in, and build the program of Home Assignment 4. Read key char and terminate the application when receiving “exit” command.

Laboratory Exercise 11

Interrupts & IO programming

Goals

After this laboratory exercise, you should understand the basic principles of interrupts and how interrupts can be used for programming. You should also know the difference between polling and using interrupts and the relative merits of these methods.

Literature

■ Patterson and Hennessy: Chapter 2.7, 2.9, 2.10, 2.13, 5.7, Appendix A.6, A.7, A.10

Polling or Interrupts

A computer can react to external events either by polling or by using interrupts. One method is simpler, while the other one is more systematic and also more efficient. You will study the similarities and differences of these methods using a simple "toy" example program.

Each pheriperal device connects to the CPU via a few ports. CPU uses address to find out the respective port, and after that, CPU could read/write the new value to these port to get/control the device.

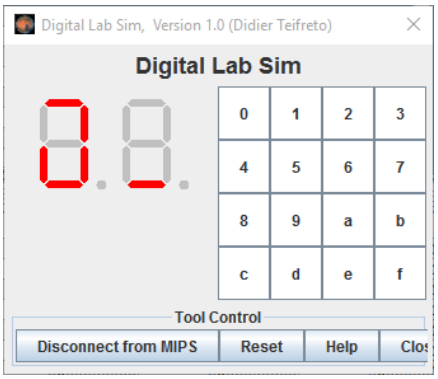
Preparation

Study literature and these home assignments before coming into the class.

Assignments at Home and at Lab

Home Assignment 1 - POOLING

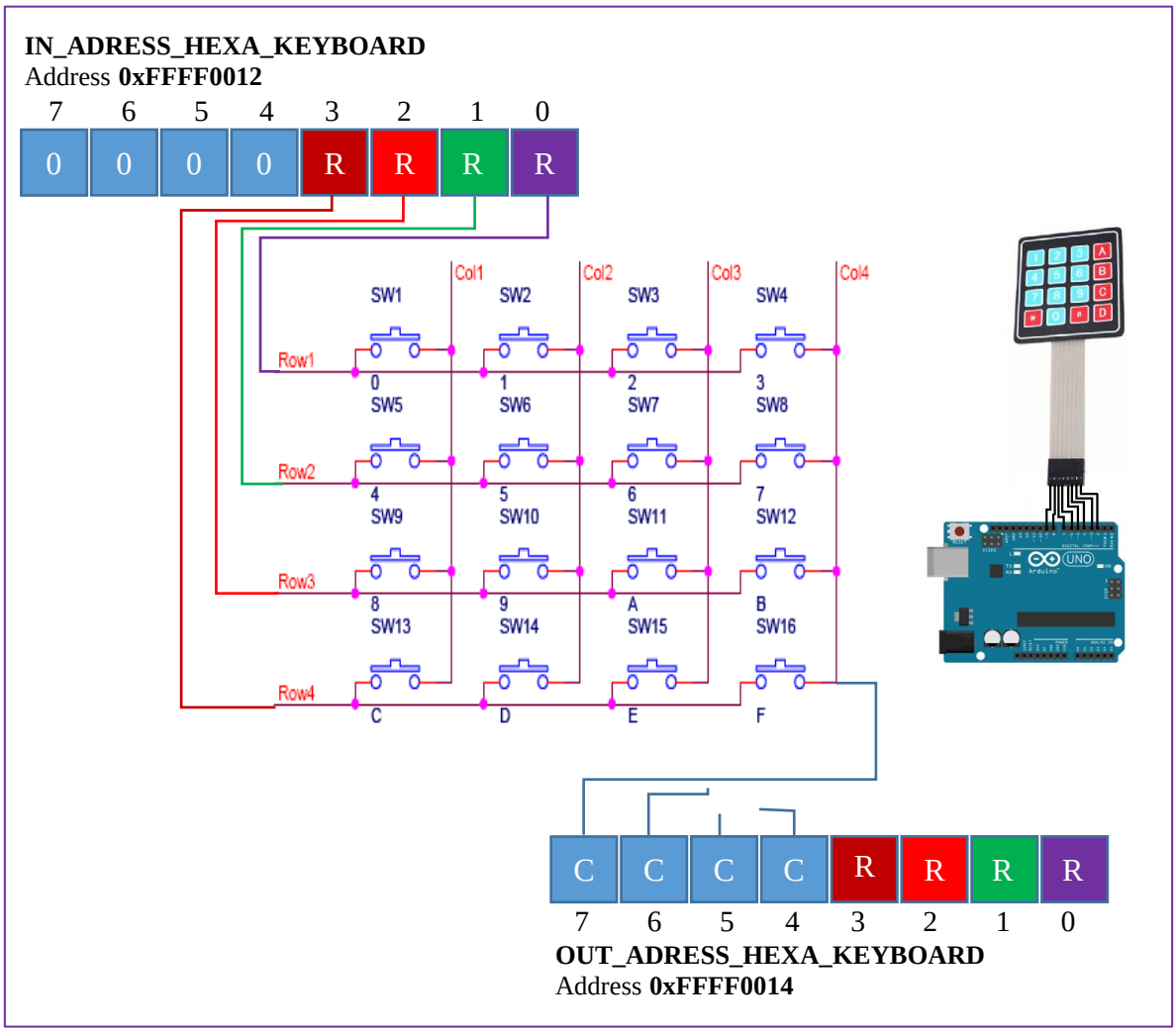
Write a program using assembly language to detect key pressed in Digi Lab Sim and print the key number to console.



The program has a unlimited loop, to read the scan code of key button. This is POLLING.

In order to use the key matrix[4], you should
1.assign experted row index into the byte at the address 0xFFFF0012
2.read byte at the address 0xFFFF0014, to detect which key button was pressed

[4] Key matrix animation: http://hackyourmind.org/public/images/keypad12keys_anim.gif




```
#-------------------------------------------------------
#            col 0x1     col 0x2     col 0x4      col 0x8
#
#  row 0x1      0           1           2            3
#             0x11        0x21        0x41         0x81
#
#  row 0x2      4           5           6            7
#             0x12        0x22        0x42         0x82
#
#  row 0x4      8           9           a            b
#             0x14        0x24        0x44         0x84
#
#  row 0x8      c           d           e            f
#             0x18        0x28        0x48         0x88
#
#-------------------------------------------------------
# command row number of hexadecimal keyboard (bit 0 to 3)
# Eg. assign 0x1, to get key button 0,1,2,3
#     assign 0x2, to get key button 4,5,6,7
# NOTE must reassign value for this address before reading,
# eventhough you only want to scan 1 row
.eqv IN_ADRESS_HEXA_KEYBOARD       0xFFFF0012

# receive row and column of the key pressed, 0 if not key pressed
# Eg. equal 0x11, means that key button 0 pressed.
# Eg. equal 0x28, means that key button D pressed.
.eqv OUT_ADRESS_HEXA_KEYBOARD      0xFFFF0014
```

```
.text
main:             li    $t1,   IN_ADRESS_HEXA_KEYBOARD
                  li    $t2,   OUT_ADRESS_HEXA_KEYBOARD
                  li    $t3,   0x08       # check row 4 with key C, D,
E, F

polling:          sb    $t3,   0($t1 )    # must reassign expected row
                  lb    $a0,   0($t2)     # read scan code of key button
    print:        li    $v0,   34         # print integer (hexa)
                  syscall
    sleep:        li    $a0,   100        # sleep 100ms
                  li    $v0,   32
                  syscall
back_to_polling: j      polling           # continue polling
```

Home Assignment 2 - INTERRUPT

Study the following assembly program, which **waits for an interrupt from keyboard matrix, and prints out a simple message**. Go over the code in detail and make sure that you understand everything, especially how to write and install an interrupt routine, how to enable an interrupt, and what happens when an interrupt is activated.

Vietnamese support:
Cũng như các bộ xử lý khác, MIPS có 3 service với cùng một nguyên lý, nhưng khác nhau về mục đích sử dụng

- *Exception: xảy ra khi có lỗi trong quá trình chạy, chẳng hạn tham chiếu bộ nhớ không hợp lệ.*
- *Trap: xẩy ra bởi cách lệnh kiểm tra*
- *Interrupt: do các thiết bị bên ngoài kích hoạt*

Cả 3 cơ chế trên đều được gọi chung là Exception.
Cách thức hoạt động: *khi một exception xảy ra*

- *Khi có một Exception xảy ra, MIPS sẽ luôn nhảy tới địa chỉ cố định 0x80000180 để thực hiện chương trình con phục vụ ngắt. Để viết chương trình con phục vụ ngắt, sử dụng chỉ thị* ***.ktext*** *để viết code ở địa chỉ 0x80000180 nói trên.*
- *Bộ đồng xử lý C0, thanh ghi $12 (status) sẽ bật bit 1*
- *Bộ đồng xử lý C0, thanh ghi $13 (cause) sẽ thay đổi các bit 2~6 cho biết nguyên nhân gây ra ngắt*
- *Bộ đồng xử lý C0, thanh ghi $14 (epc) sẽ chứa địa chỉ kế tiếp của chương trình chính, để quay trở về sau khi xử lý các đoạn mã Exception xong. (giống như thanh ghi $ra)*
- *Trường hợp thanh ghi $13 (cause) cho biết nguyên nhân làm tham chiếu địa chỉ bộ nhớ không hợp lệ, thanh ghi $8 (vaddr) sẽ chứa địa chỉ lỗi đó.*
- *Nếu không có mã lệnh nào ở địa chỉ 0x80000180 (**.ktext**), chương trình sẽ hiện thông báo lỗi và tự kết thúc.*
- *Sau khi kết thúc chương trình con, sử dụng lệnh **eret** để quay trở lại chương btrình chính. Lệnh **eret** sẽ gán nội dung thanh ghi PC bằng giá trị trong thanh ghi $14 (epc).*

Tuy nhiên, lưu ý rằng, trong MARS, thanh ghi PC vẫn chứa địa chỉ của lệnh mà ngắt xảy ra, tức là lệnh đã thực hiện xong, chứ không chứa địa chỉ của lệnh kế tiếp. Bởi vây phải tự lập trình để tăng địa chỉ chứa trong thanh ghi epc bằng cách sử dụng 2 lệnh mfc0 (để đọc thanh ghi trong bộ đồng xử lý C0) và mtc0 (để ghi giá trị vào thanh ghi trong bộ đồng xử lý C0)

```
        mfc0    $at, $14          # $at <=  Coproc0.$14 = Coproc0.epc
        addi    $at, $at, 4       # $at = $at + 4    (next instruction)
        mtc0    $at, $14          # Coproc0.$14 = Coproc0.epc <= $at
```

- *Các bit 8-15 của thanh ghi Cause, $13 được sử dụng để xác định nguyên nhân gây ra ngắt. Hãy đọc thanh ghi này, kết hợp với thông tin chi tiết trong hướng dẫn sử dụng của từng thiết bị giả lập để biết được nguồn gốc gây ra ngắt.*

Cách thức viết chương trình phục vụ ngắt: *để viết chương trình con phục vụ ngắt khi có sự kiện ngắt xảy ra, có thể dùng một trong các phương pháp sau:*

1. *Viết chương trình con phục vụ ngắt trong cùng một file nguồn*
2. *Viết chương trình con phục vụ ngắt trong file nguồn độc lập, và lưu trữ trong cùng một thư mục với chương trình chính. Sau đó, sử dụng tính năng trong mục Setting là "Assemble all files in directory"*
3. *Viết chương trình con phục vụ ngắt trong file nguồn độc lập, và lưu trữ trong cùng một thư mục bất kì. Sau đó, sử dụng tính năng trong mục Setting là "Exception Handler.."*

BUG: Ghi nhận các lỗi của công cụ MARS

1. *Giữa 2 lệnh syscall và lệnh jump, branch cần bổ sung thêm lệnh nop. Nếu không việc ghi nhận giá trị của thanh ghi PC vào EPC sẽ bị sai*
2. *Với các công cụ giả lập, nên bấm nút "Connect to MIPS" trước khi chạy giả lập. Nếu không, việc phát sinh sự kiện ngắt sẽ không xảy ra.*

```
.eqv IN_ADRESS_HEXA_KEYBOARD       0xFFFF0012

.data
Message: .asciiz "Oh my god. Someone's presed a button.\n"
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# MAIN Procedure
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.text
main:
        #---------------------------------------------------------
        # Enable interrupts you expect
        #---------------------------------------------------------
        # Enable the interrupt of Keyboard matrix 4x4 of Digital Lab
Sim
        li    $t1,    IN_ADRESS_HEXA_KEYBOARD
        li    $t3,    0x80  # bit 7 of = 1 to enable interrupt
```

```
        sb    $t3,   0($t1)

        #-------------------------------------------------------
        # No-end loop, main program, to demo the effective of
interrupt
        #-------------------------------------------------------
Loop:   nop
        nop
        nop
        nop
        b       Loop            # Wait for interrupt
end_main:

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# GENERAL INTERRUPT SERVED ROUTINE for all interrupts
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.ktext 0x80000180
        #-------------------------------------------------------
        # Processing
        #-------------------------------------------------------
IntSR:  addi    $v0, $zero,  4  # show message
        la      $a0, Message
        syscall
        #-------------------------------------------------------
        # Evaluate the return address of main routine
        # epc  <= epc + 4
        #-------------------------------------------------------
next_pc:mfc0    $at, $14        # $at <=  Coproc0.$14 = Coproc0.epc
        addi    $at, $at, 4     # $at = $at + 4   (next instruction)
        mtc0    $at, $14        # Coproc0.$14 = Coproc0.epc <= $at

return: eret                    # Return from exception
```

Home Assignment 3 – INTERRUPT & STACK

Study the following assembly program, in which

1. Main program enables 1 interrupt: from key matrix in Data Lab Sim
2. Main program only print a sequence number to console
3. Connect Data Lab Sim. Whenever user press a key button C, D, E, or F, an interrupt raises and print key scan-code to console

```
.eqv IN_ADRESS_HEXA_KEYBOARD       0xFFFF0012
.eqv OUT_ADRESS_HEXA_KEYBOARD      0xFFFF0014

.data
Message: .asciiz "Key scan code "

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# MAIN Procedure
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.text
main:
        #-------------------------------------------------------
        # Enable interrupts you expect
        #-------------------------------------------------------
        # Enable the interrupt of Keyboard matrix 4x4 of Digital Lab
Sim
        li    $t1,   IN_ADRESS_HEXA_KEYBOARD
        li    $t3,   0x80       # bit 7 = 1 to enable
        sb    $t3,   0($t1)
```

```
        #-------------------------------------------------------
        # Loop an print sequence numbers
        #-------------------------------------------------------
        xor     $s0, $s0, $s0  # count = $s0 = 0
Loop:   addi    $s0, $s0, 1    # count = count + 1
prn_seq:addi    $v0,$zero,1
        add     $a0,$s0,$zero  # print auto sequence number
        syscall
prn_eol:addi    $v0,$zero,11
        li      $a0,'\n'       # print endofline
        syscall
sleep:  addi    $v0,$zero,32
        li      $a0,300        # sleep 300 ms
        syscall
        nop                    # WARNING: nop is mandatory here.
        b       Loop           # Loop
end_main:

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# GENERAL INTERRUPT SERVED ROUTINE for all interrupts
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.ktext 0x80000180
        #-------------------------------------------------------
        # SAVE the current REG FILE to stack
        #-------------------------------------------------------
IntSR:  addi  $sp,$sp,4    # Save $ra  because we may change it later
        sw    $ra,0($sp)
        addi  $sp,$sp,4    # Save $ra  because we may change it later
        sw    $at,0($sp)
        addi  $sp,$sp,4    # Save $ra  because we may change it later
        sw    $v0,0($sp)
        addi  $sp,$sp,4    # Save $a0, because we may change it later
        sw    $a0,0($sp)
        addi  $sp,$sp,4    # Save $t1, because we may change it later
        sw    $t1,0($sp)
        addi  $sp,$sp,4    # Save $t3, because we may change it later
        sw    $t3,0($sp)
        #-------------------------------------------------------
        # Processing
        #-------------------------------------------------------
prn_msg:addi    $v0, $zero,  4
        la      $a0, Message
        syscall
get_cod:li      $t1,   IN_ADRESS_HEXA_KEYBOARD
        li      $t3,   0x88      # check row 4 and re-enable bit 7
        sb      $t3,   0($t1)    # must reassign expected row
        li      $t1,   OUT_ADRESS_HEXA_KEYBOARD
        lb      $a0,   0($t1)
prn_cod:li      $v0,34
        syscall
        li      $v0,11
        li      $a0,'\n'        # print endofline
        syscall
        #-------------------------------------------------------
        # Evaluate the return address of main routine
        # epc  <= epc + 4
        #-------------------------------------------------------
next_pc:mfc0    $at, $14        # $at <=  Coproc0.$14 = Coproc0.epc
        addi    $at, $at, 4     # $at = $at + 4   (next instruction)
        mtc0    $at, $14        # Coproc0.$14 = Coproc0.epc <= $at
        #-------------------------------------------------------
        # RESTORE the REG FILE from STACK
        #-------------------------------------------------------
```

```
restore:lw      $t3, 0($sp)      # Restore the registers from stack
        addi    $sp,$sp,-4
        lw      $t1, 0($sp)      # Restore the registers from stack
        addi    $sp,$sp,-4
        lw      $a0, 0($sp)      # Restore the registers from stack
        addi    $sp,$sp,-4
        lw      $v0, 0($sp)      # Restore the registers from stack
        addi    $sp,$sp,-4
        lw      $ra, 0($sp)      # Restore the registers from stack
        addi    $sp,$sp,-4

return: eret                      # Return from exception
```

Home Assignment 4 – MULTI INTERRUPT

Vietnamese support:

Thanh ghi số 13, status trong bộ đồng xử lý C0, chứa các thiết lập về tình trạng ngắt.

	15	14	13	12	11	10			6	5	4	3	2	1	0
										1	1	0	1	0	0
					KM	TC			Exception Code					K/U	IE
IE = 1 cho phép ngắt. IE = 0 vô hiệu hóa mọi hoạt động ngắt															
K/U=1 hoạt động ở chế độ Kernel. K/U=0 hoạt động ở chế độ User															
Ngoại lệ do syscall, overflow, lệnh tạo ngắt mềm như teq teqi…															
Time Counter bộ đếm thời gian															
Key Matrix															

Study the following assembly program, in which

1. Main program enables 2 interrupts simultaneously: from key matrix and time counter in Data Lab Sim
2. Main program do nothing with a deadloop
3. Connect Data Lab Sim. Whenever user press any key or time interval reachs, an interrupt raises and print key scan-code to console.

```
.eqv  IN_ADRESS_HEXA_KEYBOARD       0xFFFF0012
.eqv  COUNTER          0xFFFF0013          # Time Counter

.eqv  MASK_CAUSE_COUNTER   0x00000400      # Bit 10: Counter interrupt
.eqv  MASK_CAUSE_KEYMATRIX 0x00000800      # Bit 11: Key matrix
interrupt

.data
msg_keypress: .asciiz "Someone has pressed a key!\n"
msg_counter: .asciiz "Time inteval!\n"

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# MAIN Procedure
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.text
main:
        #-----------------------------------------------------------
        # Enable interrupts you expect
        #-----------------------------------------------------------
        # Enable the interrupt of Keyboard matrix 4x4 of Digital Lab
Sim
        li    $t1,   IN_ADRESS_HEXA_KEYBOARD
        li    $t3,   0x80      # bit 7 = 1 to enable
        sb    $t3,   0($t1)
```

```
        # Enable the interrupt of TimeCounter of Digital Lab Sim
        li    $t1,   COUNTER
        sb    $t1,   0($t1)

        #-----------------------------------------------------------
        # Loop an print sequence numbers
        #-----------------------------------------------------------
Loop:   nop
        nop
        nop
sleep:  addi    $v0,$zero,32   # BUG: must sleep to wait for Time
Counter
        li      $a0,200        # sleep 300 ms
        syscall
        nop                    # WARNING: nop is mandatory here.
        b       Loop
end_main:

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# GENERAL INTERRUPT SERVED ROUTINE for all interrupts
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.ktext 0x80000180
IntSR:  #-----------------------------------------------------------
        # Temporary disable interrupt
        #-----------------------------------------------------------
dis_int:li    $t1,   COUNTER   # BUG: must disable with Time Counter
        sb    $zero,   0($t1)
        # no need to disable keyboard matrix interrupt
        #-----------------------------------------------------------
        # Processing
        #-----------------------------------------------------------
get_caus:mfc0  $t1, $13              # $t1 = Coproc0.cause
IsCount:li    $t2, MASK_CAUSE_COUNTER# if Cause value confirm
Counter..
        and   $at, $t1,$t2
        beq   $at,$t2, Counter_Intr
IsKeyMa:li    $t2, MASK_CAUSE_KEYMATRIX # if Cause value confirm Key..
        and   $at, $t1,$t2
        beq   $at,$t2, Keymatrix_Intr
others: j     end_process           # other cases

Keymatrix_Intr: li      $v0, 4      # Processing Key Matrix Interrupt
        la      $a0, msg_keypress
        syscall
        j     end_process
Counter_Intr: li      $v0, 4        # Processing Counter Interrupt
        la    $a0, msg_counter
        syscall
        j     end_process
end_process:
        mtc0  $zero, $13            # Must clear cause reg
en_int: #-----------------------------------------------------------
        # Re-enable interrupt
        #-----------------------------------------------------------
        li    $t1,   COUNTER
        sb    $t1,   0($t1)
        #-----------------------------------------------------------
        # Evaluate the return address of main routine
        # epc  <= epc + 4
        #-----------------------------------------------------------
next_pc:mfc0    $at, $14        # $at <=  Coproc0.$14 = Coproc0.epc
        addi    $at, $at, 4     # $at = $at + 4   (next instruction)
        mtc0    $at, $14        # Coproc0.$14 = Coproc0.epc <= $at
```

```
return: eret                        # Return from exception
```

Home Assignment 5 – KEYBOARD

Vietnamese support:

- *Bộ xử lý MIPS cho phép tạo ra ngắt mềm, bằng lệnh teq, hoặc teqi*
- *Thiết bị Keyboard không tự động tạo ra ngắt khi có một phím được bấm, mà người lập trình phải tự tạo ngắt mềm.*

```
.eqv KEY_CODE   0xFFFF0004        # ASCII code from keyboard, 1 byte
.eqv KEY_READY  0xFFFF0000        # =1 if has a new keycode ?
                                  # Auto clear after lw

.eqv DISPLAY_CODE   0xFFFF000C    # ASCII code to show, 1 byte
.eqv DISPLAY_READY  0xFFFF0008    # =1 if the display has already to do
                                  # Auto clear after sw

.eqv  MASK_CAUSE_KEYBOARD   0x0000034     # Keyboard Cause

.text
            li   $k0,  KEY_CODE
            li   $k1,  KEY_READY

            li   $s0, DISPLAY_CODE
            li   $s1, DISPLAY_READY

loop:       nop
WaitForKey: lw   $t1, 0($k1)            # $t1 = [$k1] = KEY_READY
            beq  $t1, $zero, WaitForKey # if $t1 == 0 then Polling
MakeIntR:   teqi $t1, 1          # if $t0 = 1 then raise an Interrupt
            j loop

#-----------------------------------------------------------------
# Interrupt subroutine
#-----------------------------------------------------------------
.ktext 0x80000180
get_caus:   mfc0  $t1, $13              # $t1 = Coproc0.cause
IsCount:    li    $t2, MASK_CAUSE_KEYBOARD# if Cause value confirm
Keyboard..
            and   $at, $t1,$t2
            beq   $at,$t2, Counter_Keyboard
            j    end_process

Counter_Keyboard:
ReadKey:    lw   $t0, 0($k0)            # $t0 = [$k0] = KEY_CODE

WaitForDis: lw   $t2, 0($s1)            # $t2 = [$s1] = DISPLAY_READY
            beq  $t2, $zero, WaitForDis # if $t2 == 0 then Polling

Encrypt:    addi $t0, $t0, 1           # change input key

ShowKey:    sw $t0, 0($s0)             # show key
            nop

end_process:
next_pc:    mfc0    $at, $14       # $at <= Coproc0.$14 = Coproc0.epc
            addi    $at, $at, 4    # $at = $at + 4 (next instruction)
            mtc0    $at, $14       # Coproc0.$14 = Coproc0.epc <= $at
return:     eret                   # Return from exception
```

Assignment 1

Create a new project, type in, and build the program of Home Assignment 1. Upgrade the source code so that it could defect all 16 key buttons, from 0 to F.

Assignment 2

Create a new project, type in, and build the program of Home Assignment 2. Upgrade the source code so that it could defect all 16 key buttons, from 0 to F.

Assignment 3

Create a new project, type in, and build the program of Home Assignment 3. Upgrade the source code so that it could defect all 16 key buttons, from 0 to F.

Assignment 4

Create a new project, type in, and build the program of Home Assignment 4. Upgrade the source code so that it could defect all 16 key buttons, from 0 to F.

Assignment 5

Create a new project, type in, and build the program of Home Assignment 5.

Conclusions

Before you finish the laboratory exercise, think about the questions below:

- What is polling?
- What are interrupts?
- What are interrupt routines?
- What are the advantages of polling?
- What are the advantages of using interrupts?
- What are the differences between interrupts, exceptions and traps?

Laboratory Exercise 12

Cache Memory

Goals

After this laboratory exercise, you should understand the basic principles of cache memories and how different parameters of a cache memory affect the efficiency of a computer system.

Literature

■ Patterson and Hennessy: Chapter 7.1–7.3

Preparation

Read the literature and this laboratory exercise in detail and solve the home assignments.

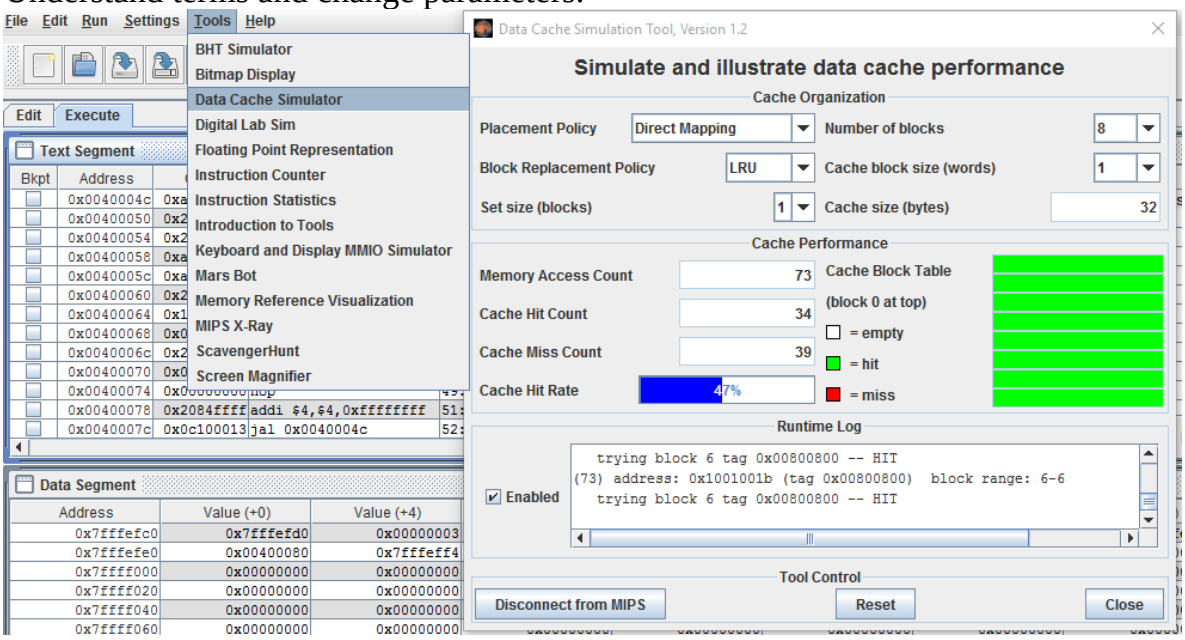
Assignments at Home and at Lab

Home Assignment 1

Looking for and research information about Cache in CPU.

Home Assignment 2

Try to use Data Cache Simulation Tool in MARS.
Understand terms and change parameters.



Function of a Cache Memory

A cache memory is a memory that is smaller but faster than the main memory. Due to the locality of memory references, the use of a cache memory can have the effect on the computer system that the apparent speed of the memory is that of the cache memory, while the size is that of the main memory.

Assignment 1

Create a project, type in the program of Laboratory 7, Home Assignment 4 , build it and upload it to the cache simulator.

Assignment 2

Run the program in the cache simulator and study how the instruction cache works. Then give *full* answers to the following questions.

■ How is the full 32-bit address used in the cache memory?
■ What happens when there is a cache miss?
■ What happens when there is a cache hit?
■ What is the block size?
■ What is the function of the tag?

Assignment 3

The parameters of the cache memory can be changed to test the effects of different cases. Investigate the effects of different parameter settings.

■ Explain the following: cache size, block size, number of sets, write policy and replacement policy.

■ If a cache is large enough that all the code within a loop fits in the cache, how many cache misses will there be during the execution of the loop? Is this good or bad?

■ What should the code look like that would benefit the most from a large block size?

Cache Efficiency

The actual efficiency gained by using a cache memory varies depending on cache size, block size and other cache parameters, but it also depends on the program and data.

Conclusions

Before you pass the laboratory exercise, think about the questions below:

■ What is the general idea with cache memory?
■ What is a block?
■ How does block size affect the efficiency of a cache?
■ How fast is a cache memory? How fast is a DRAM?
■ Do the optimal cache parameters depend on the program code?
■ How can one select good cache parameters?
■ Is it possible to change cache size on a PC? On a Mac?

Track Change

Ver	Date	Details
5.3	2013/04/23	More explanation for Lab Exercise 12, Home Assignment 2
6	2016/02	Using MARS
6.2	2018/02	Fix bug

Home Assignment 3

The following program sort a list of interger by ascending order. The swap procedure swap two successive elements. Read this example carefully, analyse each line of code, especially in sort procedure.

```
#Laboratory Exercise 5, Home Assignment 3
.data
test:       .word       3,6,2,1
.text
main:
      la    $a0,test
      li    $a1,4
      jal   sort
      nop
      la    $a0,test
      lw    $s0,0($a0)        #load test array after sorting
      lw    $s1,4($a0)
      lw    $s2,8($a0)
      lw    $s3,12($a0)
end_main:

swap:
      sll   $t1,$a1,2         #reg $t1=k*4
      add   $t1,$a0,$t1     #reg $t1=v+(k*4)
                              #reg $t1 has the address of v[k]
      lw    $t0,0($t1)        #reg $t0 (temp)=v[k]
      lw    $t2,4($t1)        #reg $t2=v[k+1]
                              #refers to next element of v
      sw    $t2,0($t1)        #v[k]=reg $t2
      sw    $t0,4($t1)        #v[k+1]=reg $t0 (temp)
      jr    $ra
end_swap:

#Procedure sort
#function: sort a list of element ascending
sort:
      addi  $sp,$sp,-20     #make room on stack for 5 registers
      sw    $ra,16($sp)     #save ra
      sw    $s3,12($sp)     #save $s3
      sw    $s2,8($sp)        #save $s2
      sw    $s1,4($sp)        #save $s1
      sw    $s0,0($sp)        #save $s0

      move  $s2,$a0         #copy parameter $$a0 into $s2 (save $a0)
      move  $s3,$a1         #copy parameter $a1 into $s3 (save a1)
      move  $s0,$zero         #i=0
for1tst:
      slt   $t0,$s0,$s3     #reg $t0=0 if $s0>=s3 (i>=n)
      beq   $t0,$zero,exit1   #go to exit1 if $s0>=s3 (i>=n)
      addi  $s1,$s0,-1        #j=i-1
for2tst:
      slti  $t0,$s1,0         #reg $t0=1 if $s1<0 (j<0)
      bne   $t0,$zero,exit2   #go to exit2 if $s1<0 (j<0)
      sll   $t1,$s1,2         #reg $t1=j*4
      add   $t2,$s2,$t1     #reg $t2=v+(j*4)
      lw    $t3,0($t2)        #reg $t3=v[j]
      lw    $t4,4($t2)        #reg $t4=v[j+1]
      slt   $t0,$t4,$t3     #reg $t0=0 if $t4>=t3
      beq   $t0,$zero,exit2   #go to exit2 if $t4>=t3
      move  $a0,$s2         #1st parameter of swap is v(old $a0)
      move  $a1,$s1         #2nd parameter of swap is j
```

```
      jal   swap
      addi  $s1,$s1,-1        #j=j-1
      j     for2tst           #jump to test of inner loop
exit2:
      addi  $s0,$s0,1         #i=i+1
      j     for1tst           #jump to test of outer loop
exit1:
      lw    $s0,0($sp)        #restore $s0
      lw    $s1,4($sp)        #restore $s1
      lw    $s2,8($sp)        #restore $s2
      lw    $s3,12($sp)     #restore $s3
      lw    $ra,16($sp)     #restore ra
      addi  $sp,$sp,20        #restore stack pointer
      jr    $ra
```

end_sort:

Assignment 4

Write a program that let user input a string (press Enter to terminate), sort this string by ascending order (based on ASCII code) and print the string after sorting.