

Thực hành Kiến trúc Máy tính Computer Architecture Lab

Ver 8.1 (09/2024)

Mục lục

Bài 1. Giới thiệu về công cụ RARS.....	3
Bài 1(mở rộng). Giới thiệu về các công cụ khác	13
Bài 2. Tập lệnh, các lệnh cơ bản, các chỉ thị biên dịch	14
Bài 3. Các lệnh nhảy và lệnh rẽ nhánh	22
Bài 4. Các lệnh số học và logic	27
Bài 5. Nhập xuất dữ liệu với hàm ECALL, xử lý chuỗi ký tự	31
Bài 6. Mảng và con trỏ	41
Bài 7. Lệnh gọi chương trình con, truyền tham số sử dụng ngăn xếp.....	46
Bài 8, 9. Project giữa kỳ	53
Bài 10. Giao tiếp với các thiết bị ngoại vi	54
Bài 11. Lập trình xử lý ngắn.....	59
Bài 12. Bộ nhớ đệm nhanh – Cache memory	72
Bài 13. Lập trình hợp ngữ với ESP32-C3 – Mô phỏng bằng Wokwi	75
Bài 14. Lập trình hợp ngữ với ESP32-C3 – Lắp mạch trên breadboard	83
Bài 15, 16. Project cuối kỳ	88

Tóm tắt thay đổi

Phiên bản	Nội dung thay đổi	Thực hiện
<= 4.x	Công cụ thực hành, giả lập MIPS-IT
5.x	Chuẩn hóa trình bày	VinhTT, 01/2013
6.x	Công cụ thực hành, giả lập MARS	TienND 02/2017
7.x	Bổ sung các bài tập về hệ thống vào ra như polling, ngắt	TienND 02/2019
8.0	Chuyển đổi sang kiến trúc RISC-V với công cụ thực hành giả lập chính RARS, kết hợp với Ripe.me	VuiLB 05/2024

Bài 1. Giới thiệu về công cụ RARS

Mục đích

Sau bài thực hành này, sinh viên sẽ cài đặt được công cụ RARS, viết thử chương trình đơn giản để thử nghiệm công cụ RARS 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 ra trong bộ xử lý.

Tài liệu

- Tài liệu về RISC-V, slide bài giảng học phần.
 - The RISC-V Instruction Set Manual: [riscv-spec-20191213.pdf](https://riscv.org/specifications/riscv-isam/)
-

About RARS

- Website chính thức: <https://github.com/TheThirdOne/rars>
 - Tải về nhanh bản RARS version 1.6:
https://github.com/TheThirdOne/rars/releases/download/v1.6/rars1_6.jar
-

Kick-off

Tải về và chạy

1. Tải về Java Runtime Environment, JRE, để chạy công cụ RARS
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. Cài đặt JRE
3. Tải về công cụ RARS ở URL sau
<https://github.com/TheThirdOne/rars>

Công cụ RARS có thể thực hiện ngay mà không cần cài đặt. Click đúp vào file **rars1_6.jar** để chạy.

Cơ bản về giao diện lập trình IDE

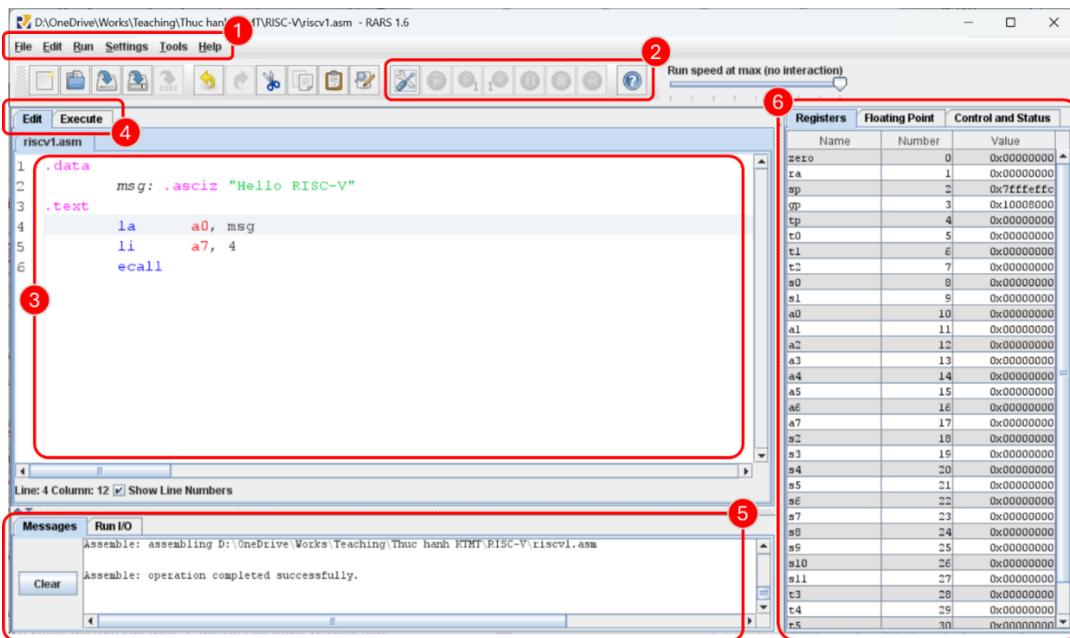


Figure 1. IDE of RARS Tool

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

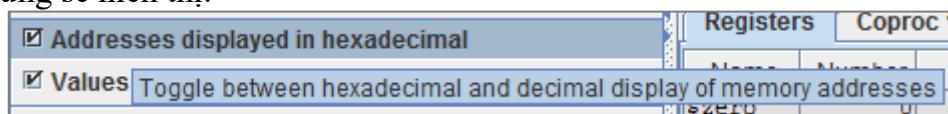


Figure 2. Tooltip giải thích chức năng trong các Menu

- o Các mục trong menu cũng có phím tắt tương ứng.
2. **Toolbar:**
 - o Chứa các tính năng soạn thảo cơ bản như copy, paste, open ...
 - o Các tính năng gỡ lỗi (trong hình chữ nhật màu đỏ)
 - Run: chạy toàn bộ chương trình
 - Run one step at a time: chạy từng lệnh rồi dừng
 - Undo the last step: khôi phục lại trạng thái ở lệnh trước đó
 - Pause: tạm dừng quá trình chạy toàn bộ (Run)
 - Stop: kết thúc quá trình gỡ lỗi
 - Reset memory and register: khởi động lại bộ nhớ và thanh ghi
3. **Edit tab:** RARS có bộ soạn thảo văn bản tích hợp sẵn với tính năng **tô màu theo cú pháp**, giúp người dùng dễ dàng theo dõi mã nguồn. Đồng thời, khi lệnh được nhập mà chưa hoàn tất, một popup sẽ hiện ra để trợ giúp. Vào menu Settings / Editor... để thay đổi các thiết lập liên quan đến chức năng soạn thảo.

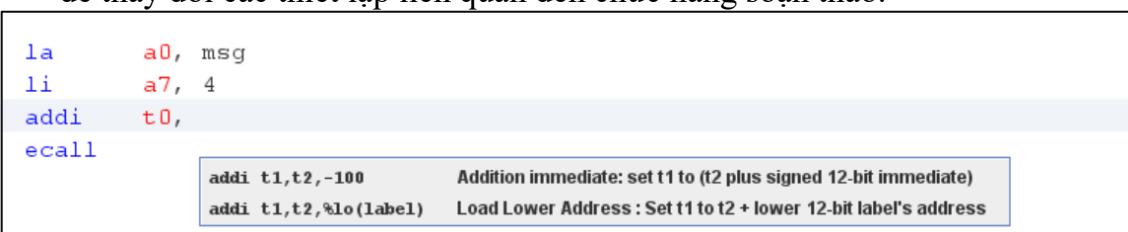


Figure 3. Popup trợ giúp hoàn tất lệnh và giải thích lệnh

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 lỗi**.

5. **Message Areas:** Có 2 cửa sổ ở cạnh dưới của giao diện IDE.

- **Run I/O** chỉ có tác dụng khi chạy chương trình.
 - **Hiển thị các kết quả xuất ra console**
 - **Nhập dữ liệu vào cho chương trình qua console.**

RARS có tùy chọn để mọi thông tin nhập liệu vào qua console sẽ được hiển thị lại ra message area.

- **Messages** được dùng để hiển thị cho các thông báo còn lại như là các thông báo lỗi trong quá trình biên dịch hay trong quá trình chạy chương trình. **Click vào thông báo lỗi để chương trình tự động nhảy tới dòng lệnh** gây ra lỗi.

6. **Registers:** Bảng hiển thị giá trị của các thanh ghi của bộ xử lý, luôn luôn được hiển thị, bất kể chương trình hợp ngữ có được chạy hay không. Khi viết chương trình, bảng này sẽ giúp sinh viên nhớ được tên của các thanh ghi và số hiệu của chúng. Có 3 tab trong bảng này:

- **Registers:** các thanh ghi số nguyên với có thứ tự từ 0 đến 31, và cả thanh ghi bộ đếm chương trình Program Counter.
- **Floating Point:** các thanh ghi số thực dấu phẩy động.
- **Control and Status:** các thanh ghi điều khiển và trạng thái, phục vụ cho xử lý ngắt.

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

1. Click vào file **rars1_6.jar** để bắt đầu chương trình.
2. Ở thanh menu, chọn **File / New** để tạo một file hợp ngữ mới.

```

D:\OneDrive\Works\Teaching\Thuc hanh KTMT\RISC-V\riscv1.asm - RARS 1.6
File Edit Run Settings Tools Help
New Ctrl-N Create a new file for editing
Close Ctrl-W
Close All
Save Ctrl-S
Save as ...
Save All
Dump Memory ... Ctrl-D
Exit
5      li    a0, msg
6      addi t0, a7, 4
7      ecall
.asciz "Hello RISC-V"

```

3. Cửa sổ soạn thảo file hợp ngữ sẽ hiện ra. Bắt đầu lập trình.

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

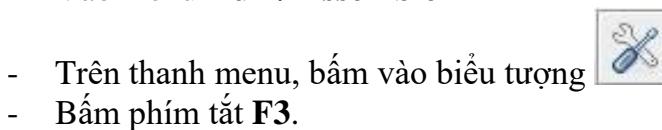
```
.data
    .word 0x01020304      # Bien x, khai tao gia tri
    msg: .asciz "Truong Cong nghe thong tin va Truyen thong"

.text
    la a0, msg            # Vung dia chi bien mesage vao thanh ghi a0
    li a7, 4              # Gan thanh ghi a7 = 4
    ecall                # Goi ham he thong in chuoi ky tu

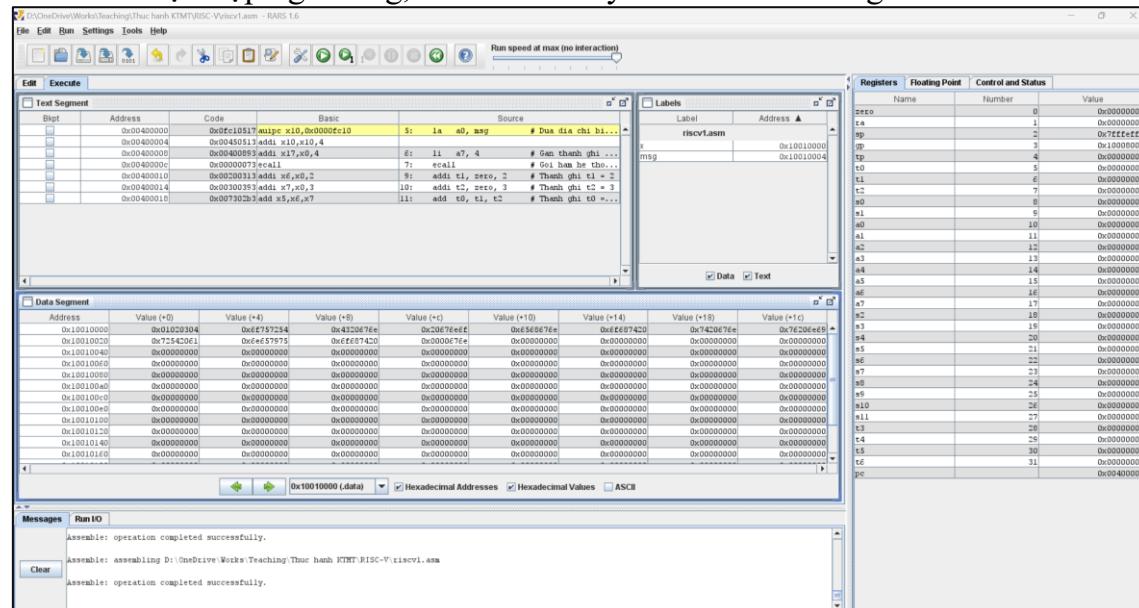
    addi t1, zero, 2      # Thanh ghi t1 = 2
    addi t2, zero, 3      # Thanh ghi t2 = 3
    add t0, t1, t2        # Thanh ghi t0 = t1 + t2
```

5. Để biên dịch chương trình hợp ngữ trên thành mã máy, thực hiện một trong các cách sau:

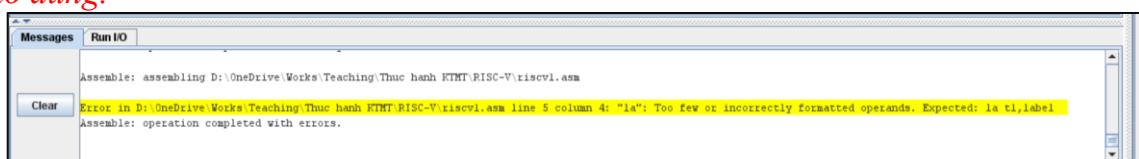
- Vào menu Run / Assemble



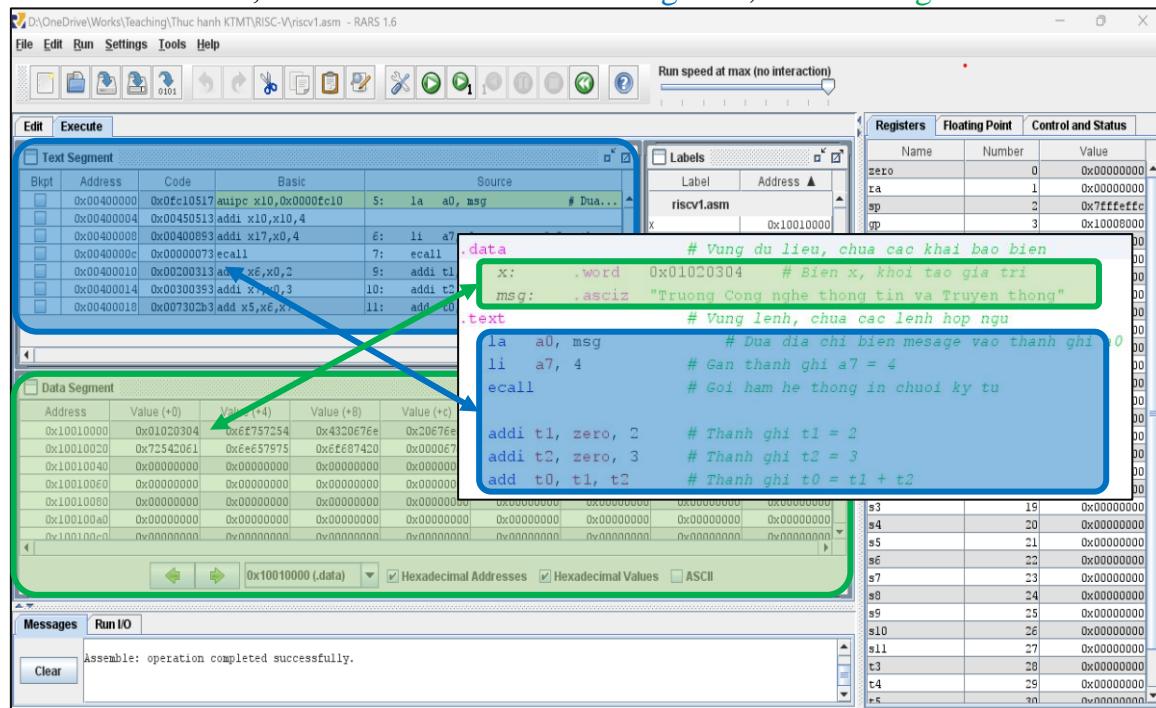
6. Nếu đoạn hợp ngữ đúng, RARS sẽ chuyển từ Edit tab sang Execute tab.



Chú ý: nếu đoạn hợp ngữ có lỗi, cửa sổ 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

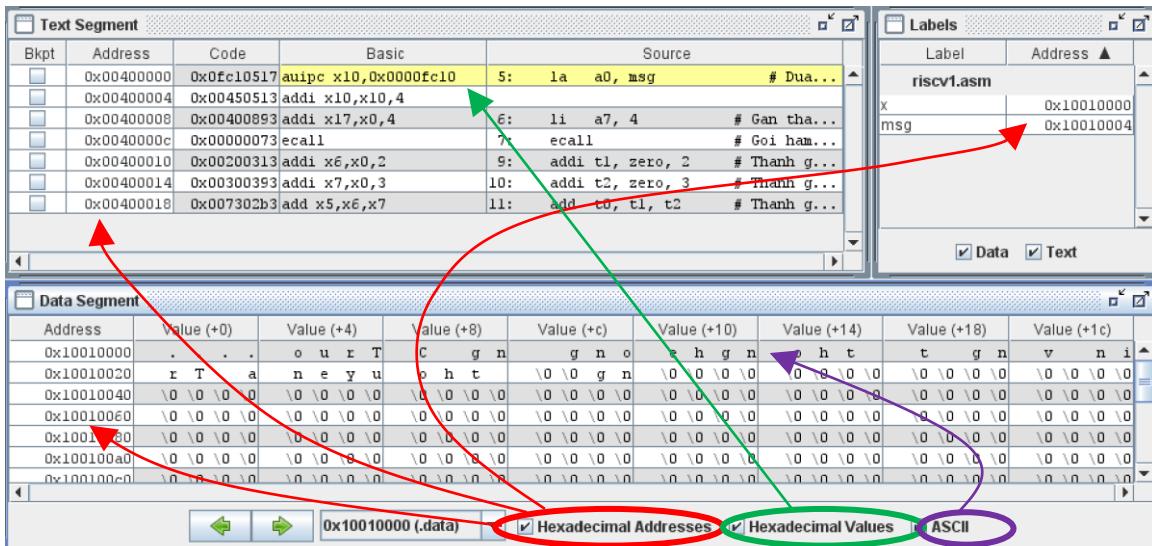


- Text Segment:** là vùng không gian bộ nhớ chứa các mã lệnh hợp ngữ. Tương ứng với mã nguồn hợp ngữ, các dòng nào viết sau chỉ thị `.text` tức là lệnh và sẽ thuộc Text Segment.
- Data Segment:** là vùng không gian bộ nhớ chứa các biến. Tương ứng với mã nguồn hợp ngữ, các dòng nào viết sau chỉ thị `.data`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

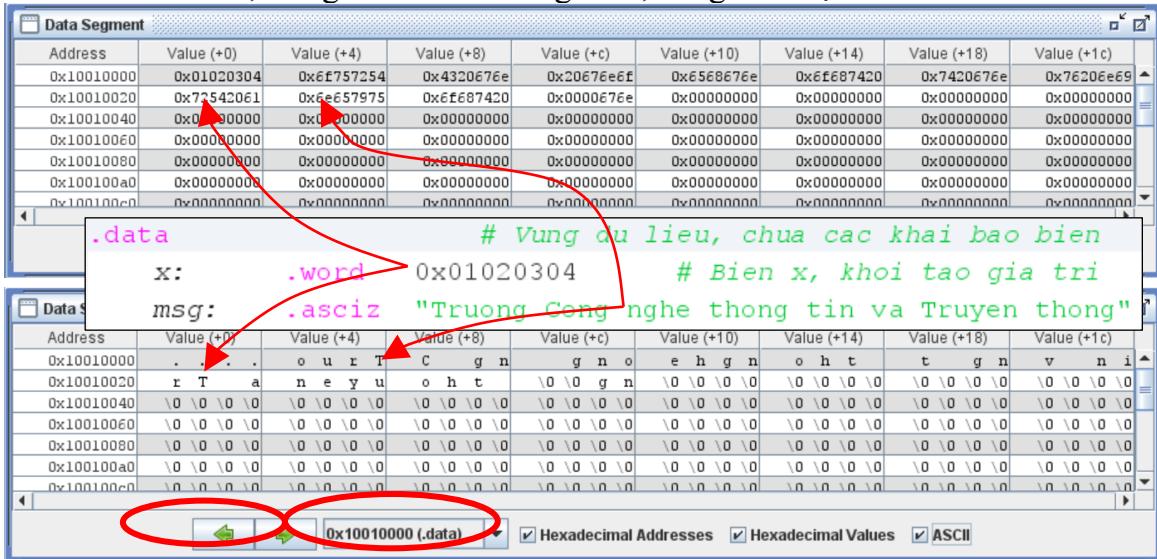


9. Ở Execute tab, trong cửa sổ Text Segment, bảng có 4 cột.

Text Segment			
Bkpt	Address	Code	Basic
	0x00400000	0x0fc10517	auipc x10,0x0000fc10
	0x00400004	0x00450513	addi x10,x10,4
	0x00400008	0x00400893	addi x17,x0,4
	0x0040000c	0x00000073	ecall
	0x00400010	0x00200313	addi x6,x0,2
	0x00400014	0x00300393	addi x7,x0,3
	0x00400018	0x007302b3	add x5,x6,x7

- **Bkpt:** Breakpoint, điểm dừng khi chạy toàn bộ 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 nhó ... 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:
 - Lệnh **la** trong cột Source là lệnh giả, không có trong tập lệnh → được dịch tương ứng thành 2 lệnh **auipc** và **addi** trong cột Basic.
 - Nhãn **msg** trong lệnh **la a0, msg** trong cột Source → địa chỉ của nhãn được thay bằng tham số cho lệnh **auipc** và **addi**.

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 tràn bày 32 bytes ở các địa chỉ liên tiếp nhau
- Các cột Value:** mỗi cột chứa 4 byte, và có 8 cột, tương ứng với 32 bytes liên tiếp nhau.

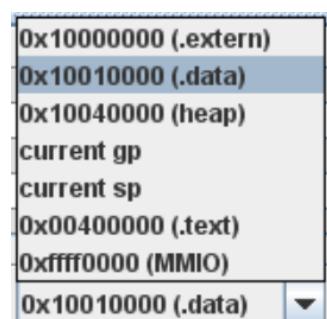
Trong hình ảnh trên, có thể thấy rõ giá trị của biến x = 0x01020304 được hiển thị chính xác trong Data Segment khi hiển thị dữ liệu ở dạng số ASCII, và giá trị của chuỗi “Truong Cong nghe thong tin va Truyen thong” khi hiển thị ở dạng kí tự ASCII.

Lưu ý rằng việc lưu trữ chuỗi trong bộ nhớ ở dạng little-endian là do cách lập trình hàm phần mềm ecalll, chứ không phải do bộ xử lý RISC-V qui định. Có thể thấy, công cụ giả lập qui định hiển thị 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

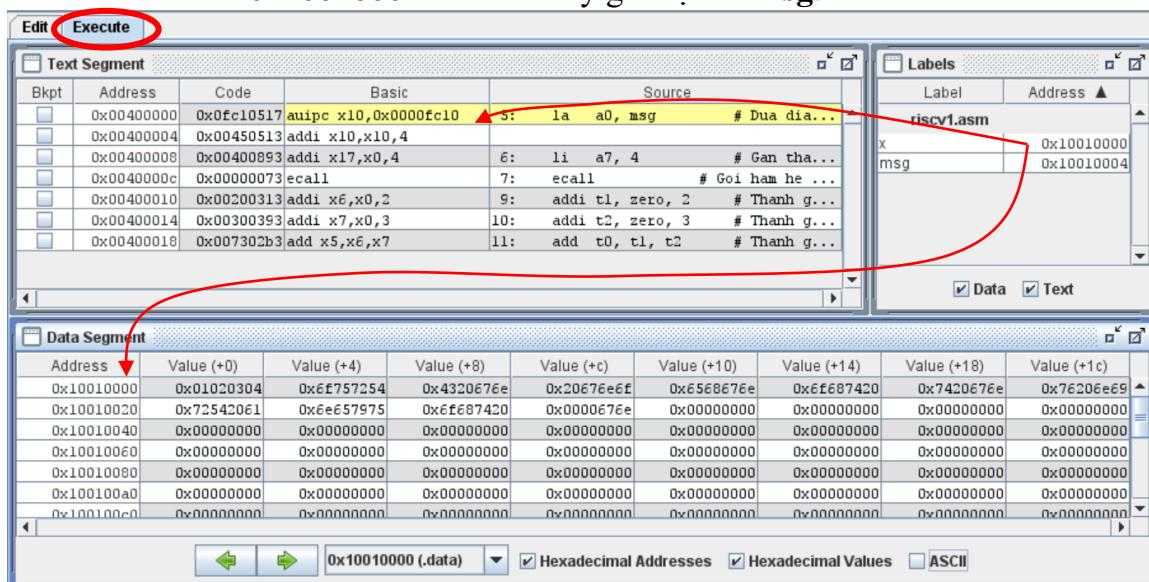


11. Cửa sổ Label: hiển thị tên nhãn và hằng số địa chỉ tương ứng với nhãn khi được biên dịch ra mã máy.

Cửa sổ Label không tự động hiển thị. Vào menu Settings / chọn Show Labels Windows để bật tắt hiển thị cửa sổ Label.

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

- Trong cửa sổ **Labels** cho biết:
 - o **x** chỉ là tên gọi nhó, x sẽ được qui đổi thành hằng số 0x10010000.
 - o **msg** cũng chỉ là tên gọi nhó, sẽ được qui đổi thành hằng số 0x10010004
 - o 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:
 - o Ở lệnh gán **la a0**, **msg** tên gọi nhó **msg** đã được chuyển thành hằng số 0x10010004 thông qua cặp lệnh **auipc** và **addi**.
- Trong cửa sổ Data Segment cho biết:
 - o Để giám sát giá trị của biến **x**, ta mở Data Segment ở hằng số 0x10010000 sẽ nhìn thấy giá trị của **x**.
 - o Để giám sát giá trị của biến **msg**, ta mở Data Segment ở hằng số 0x10010004 sẽ nhìn thấy giá trị của **msg**.



Chạy giả lập

1. Tiếp tục chạy chương trình 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 Registers.

- 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 ví dụ 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 Registers, 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.

Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x01020304	0x6d206f42
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000

2. Nhập giá trị mới.

Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x00000008	0x6d206f42
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000

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.

Registers	Floating Point	Control and Status
Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7ffffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000005
t1	6	0x00000002
t2	7	0x00000003
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x10010004
a1	11	0x00000000
a2	12	0x00000000

Tra cứu Help



Bấm nút Help  để xem giải thích các lệnh của RISC-V, các giả lệnh, chỉ dẫn biên dịch và các hàm dịch vụ hệ thống.

Các hằng địa chỉ

Chọn menu Settings / Memory Configuration...

Cửa sổ **Memory Configuration** chứa bảng qui định các hằng địa chỉ mà công cụ RARS 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.

Configuration	
<input checked="" type="radio"/> Default	memory map limit address
<input type="radio"/> Compact, Data at Address 0	kernel space high address
<input type="radio"/> Compact, Text at Address 0	MMIO base address
	kernel space base address
	user space high address
	data segment limit address
	stack base address
	stack pointer (sp)
	stack limit address
	heap base address
	.data base address
	global pointer (gp)
	data segment base address
	.extern base address
	text limit address
	.text base address

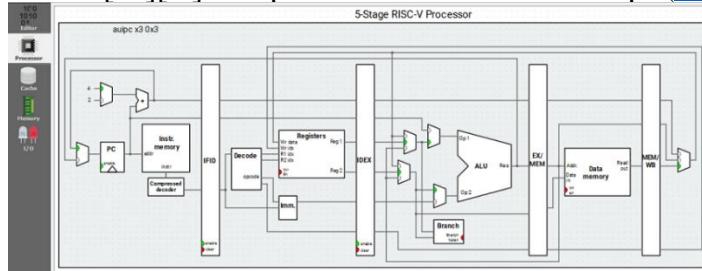
Bài 1(mở rộng). Giới thiệu về các công cụ khác

RISC-V là một kiến trúc đã rất nổi tiếng đã được nhiều dòng CPU/MCU sử dụng, trong đó phải kể đến:

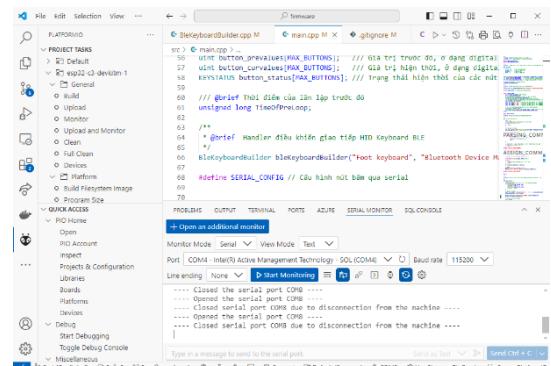
- ESP32 dòng C3, C6 của Espressif ([Xem chi tiết](#))
- CH32 của WCH ([Xem chi tiết](#))
- Allwinner D1 với bo mạch MangoPi ([Xem chi tiết](#))

Về mô phỏng, giả lập, có rất nhiều công cụ lựa chọn như:

1. **Ripes** được sử dụng để chạy mô phỏng các đoạn mã lập trình viết bằng ngôn ngữ C, ngôn ngữ Asm, trên bộ xử lý theo kiến trúc RISC-V với tập lệnh RV32I[M][C]. Có phiên bản Web và Desktop. ([Xem chi tiết](#))



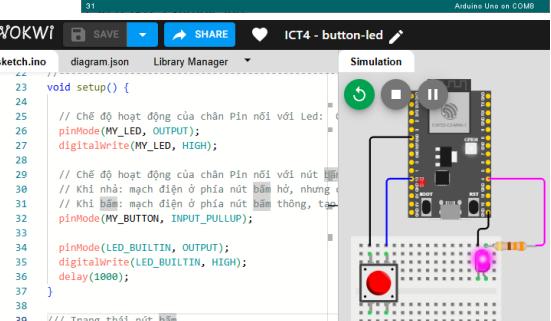
2. **Visual Studio Code**, kết hợp với extension **PlatformIO** được sử dụng để lập trình C, Asm, trên bộ xử lý thuật sử dụng kiến trúc RISC-V như ESP32-C3. Công cụ này cũng có phép chạy debug trực tiếp trên chip RISC-V. Không mô phỏng. Có phiên bản Desktop.



3. **ArduinoIDE** được sử dụng để lập trình C, Asm, trên bộ xử lý thuật sử dụng kiến trúc RISC-V như ESP32-C3. Công cụ này không debug trực tiếp trên chip RISC-V. Không mô phỏng.



4. **Wowki** là công cụ trực quan để thiết kế board mạch với nhiều cảm biến, chấp hành, cho phép mô phỏng cả thiết kế IoT. Đồng thời, cũng cho phép tạo và lưu trữ các dự án cho từng tài khoản. Có phiên bản web ([Xem chi tiết](#)). Mô phỏng đẹp.



Bài 2. Tập lệnh, các lệnh cơ bản, các chỉ thị biên dịch

Mục đích

Sau bài thực hành này, sinh viên sẽ nắm được nguyên lý cơ bản về cách CPU thực hiện lệnh, tập lệnh của bộ xử lý kiến trúc RISC-V; sử dụng các lệnh hợp ngữ cơ bản và sử dụng công cụ gõ rồi để kiểm nghiệm lại các kiến thức về tập lệnh và hợp ngữ. Sinh viên cũng thành thạo với các chỉ thị biên dịch (Directive) để công cụ RARS có thể dịch thành mã máy một cách đúng đắn.

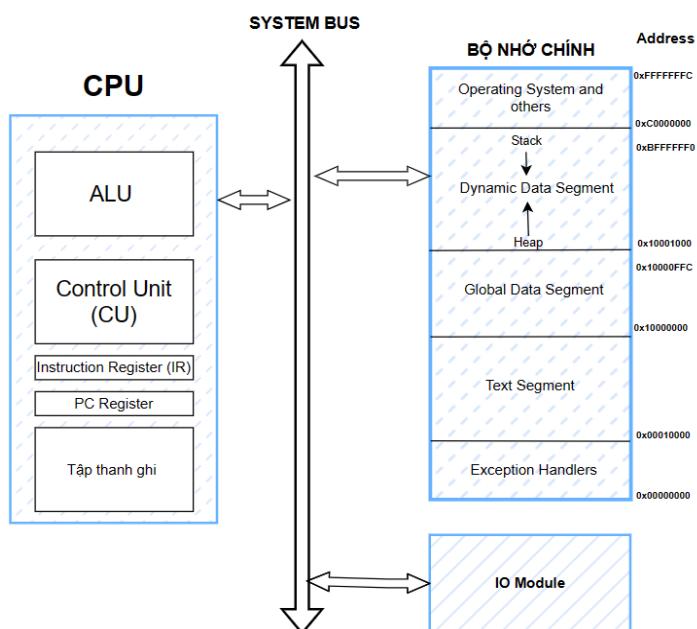
Tài liệu

- Tài liệu về RISC-V, slide bài giảng học phần.
- The RISC-V Instruction Set Manual

Bài chuẩn bị ở nhà và bài thực hành trên lớp

Home Assignment 1

Tìm hiểu các thành phần cơ bản của máy tính, mô hình lập trình của máy tính - luồng thực thi của chương trình (Cách chương trình thực hiện trong máy tính) và kiến trúc tập lệnh (Instruction set architecture).

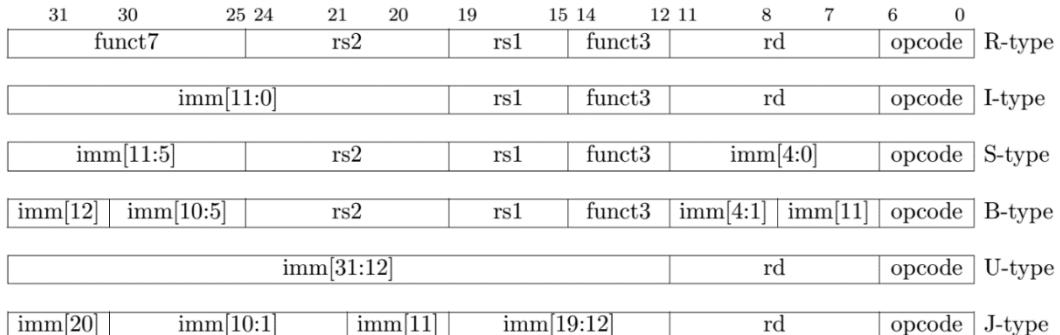


Home Assignment 2

Từ mô hình lập trình của máy tính, đọc tài liệu về kiến trúc RISC-V và ghi nhớ các kiến thức cơ bản sau:

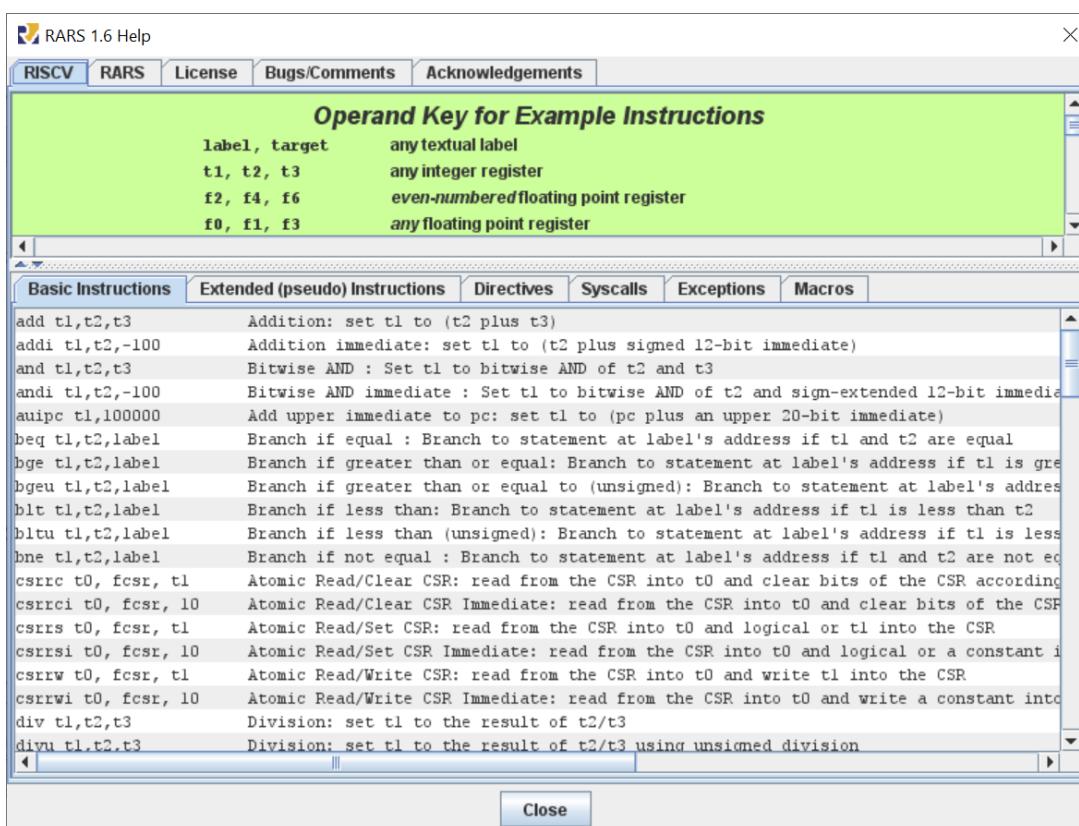
- Luồng hoạt động chương trình trong máy tính, để hiểu rõ:
 - o Tên và ý nghĩa của 32 thanh ghi.
 - o Các chỉ thị biên dịch (directives)

- Các thanh ghi đặc biệt: thanh ghi PC, thanh ghi zero, (thanh ghi IR)
- Kiến trúc tập lệnh cơ bản RV32I, và extension mở rộng
 - + M: hỗ trợ phép nhân (Multiplication Extension)
 - + C : hỗ trợ các lệnh nén 16-bit
- Khuôn dạng lệnh của các loại lệnh: R, I, S, B, U, J.



Home Assignment 3

Mở menu Help trong RARS để tìm hiểu về các lệnh cơ bản và các chỉ thị biên dịch.



Assignment 1: Lệnh gán số nguyên nhỏ 12-bit

Support: Trong kiến trúc RV32I không có lệnh gán giá trị trực tiếp vào thanh ghi, mà có thể sử dụng lệnh cộng để gán. Ví dụ muốn gán giá trị `0x123` vào thanh ghi `s0`, sử dụng lệnh cộng `addi s0, zero, 0x123`.

Mặt khác, lệnh **addi** thuộc khuôn dạng lệnh I, sử dụng 12 bits để biểu diễn số nguyên có dấu (imm[11:0]). Do đó lệnh **addi** chỉ có thể sử dụng để gán số nguyên có dấu trong phạm vi 12 bits (từ -2048 đến 2047).

Nhập chương trình sau vào công cụ giả lập RARS:

```
# Laboratory Exercise 2, Assignment 1
.text
    addi s0, zero, 0x512    # s0 = 0 + 0x512; I-type: chỉ có thể lưu
                            # được hằng số có dấu 12 bits
    add  s0, x0, zero       # s0 = 0 + 0 ; R-type: có thể sử dụng số
                            # hiệu thanh ghi thay cho tên thanh ghi
```

Yêu cầu:

- Sử dụng công cụ gỡ lỗi, chạy từng lệnh và dừng lại.
- Ở mỗi lệnh, quan sát cửa sổ Registers và chú ý:
 - o Sự thay đổi giá trị thanh ghi **s0**.
 - o Sự thay đổi giá trị thanh ghi **pc** (thanh ghi pc nằm ở vị trí dưới cùng trong cửa sổ Registers).
- Trong 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 quy định.
- Sửa lại lệnh **addi** như bên dưới. Chuyện gì xảy ra sau đó. Hãy giải thích.

```
addi s0, zero, 0x20232024
```

Assignment 2: Lệnh gán số 32-bit

Support: Để gán một số 32-bit vào thanh ghi 32-bit, sử dụng lệnh load upper immediate instruction (**lui**) kết hợp với lệnh **addi**. Lệnh **lui**, thuộc khuôn dạng lệnh U-type chứa hằng số 20-bit, nạp một hằng số 20 bits vào 20 bits có trọng số cao (most significant bit) của thanh ghi đích. Lệnh **lui** được sử dụng để nạp 20 bits trọng số cao, và lệnh **addi** được sử dụng để nạp 12 bits trọng số thấp (least significant bit).

Nhập chương trình sau vào công cụ giả lập RARS:

```
# Laboratory Exercise 2, Assignment 2
# Load 0x20232024 to s0 register
.text
    lui  s0, 0x20232        # s0 = 0xABCD000
    addi s0, s0, 0x024      # s0 = s0 + 0x123
```

Yêu cầu:

- Sử dụng công cụ gỡ lỗi, chạy từng lệnh và dừng lại.
- Ở mỗi lệnh, quan sát cửa sổ Registers và chú ý:
 - o Sự thay đổi giá trị thanh ghi **s0**.

- Sự thay đổi giá trị thanh ghi pc.
- Trong cửa sổ Data Segment, nhấn vào Combo Box để chọn và hiển thị dữ liệu trong vùng nhớ chứa lệnh (.text).
 - So sánh dữ liệu trong vùng Data Segment và mã máy trong Text Segment.

Chú ý:

- Hằng số (immediate) trong RISC-V luôn là số bù 2 12-bit (12-bit two's complement numbers), nên khi thực hiện chúng phải mở rộng dấu thành 32-bit với kiểu **sign-extended**. Một số bộ xử lý khác có cũng có kiểu mở rộng dấu **zero-extend**. Cần phân biệt hai kiểu mở rộng dấu này.
- Khi muốn nạp một hằng số lớn vào thanh ghi, nếu số 12-bit trong lệnh **addi** là số âm (bit thứ 11 bằng 1), thì cần mở rộng dấu thành 32 bit, và số 20-bit trong lệnh **lui** cần phải tăng lên 1. Giải thích tại sao?

```
# Laboratory Exercise 2, Assignment 2
# IN HIGH LEVEL LANGUAGE
# int a = 0xFEEDB987;
# IN ASSEMBLY LANGUAGE
.text
    lui s0, 0xFEEDC          # s0 = 0xFEEDC000
    addi s0, s0, 0xFFFFF987 # s0 = 0xFEEDB987
```

Assignment 3: Lệnh gán (giả lệnh)

Support: Chú ý rằng RISC-V được thiết kế dựa trên kiến trúc RISC, nên kích thước lệnh và độ phức tạp của phần cứng được thiết kế sao cho tối ưu nhất. Tuy nhiên, các công cụ giả lập cung cấp một số lệnh giả (Extended/Pseudo Instructions), các lệnh giả này không nằm trong tập lệnh của RISC-V, nó phổ biến đối với compiler/assemblers và lập trình viên. Khi chuyển sang mã máy, các lệnh giả được dịch sang một hoặc nhiều lệnh chính thống.

Nhập chương trình sau vào công cụ giả lập RARS:

```
# Laboratory Exercise 2, Assignment 3
.text
    li s0, 0x20232024
    li s0, 0x20
```

Yêu cầu:

- Biên dịch, quan sát và so sánh các lệnh ở cột Source và cột Basic trong cửa sổ Text Segment. Giải thích kết quả.

Assignment 4: Tính biểu thức $2x + y = ?$

Gõ chương trình sau vào công cụ RARS.

```
# Laboratory Exercise 2, Assignment 4
.text
```

```

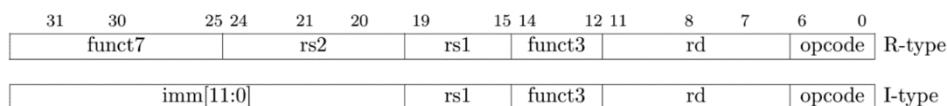
# Assign X,Y into t1,t2 register
addi t1, zero, 5      # X = t1 = ?
addi t2, zero, -1     # Y = t2 = ?

# Expression Z = 2X + Y
add s0, t1, t1      # s0 = t1 + t1 = X + X = 2X
add s0, s0, t2      # s0 = s0 + t2 = 2X + Y

```

Yêu cầu:

- Sử dụng công cụ gõ lỗi, chạy từng lệnh và dừng lại.
- Ở mỗi lệnh, quan sát cửa sổ Registers và chú ý:
 - o Sự thay đổi giá trị các thanh ghi.
 - o Sau khi kết thúc chương trình, kiểm tra xét quả có đúng không?
- Trong cửa sổ Text Segment, quan sát mã máy của các lệnh. Từ đó kiểm nghiệm mã máy với khuôn mẫu của lệnh **addi** (I-type) và lệnh **add** (R-type).
- Để hiểu hơn bản chất của việc này, hãy chuyển đổi mã basic của hai lệnh **addi** và **add** dựa vào khuôn dạng lệnh tương ứng, và so sánh với mã máy của từng lệnh (ở cột Code trong cửa sổ Text Segment).



Assignment 5: Phép nhân

Support: Phép nhân khác so với các lệnh toán học còn lại, bởi vì khi nhân hai số 32-bit, kết quả có thể là số 64-bit. Kiến trúc RISC-V cung cấp các lệnh khác nhau để thực hiện phép nhân, có thể ghi kết quả là 32-bit hoặc 64-bit, tùy lệnh. Các lệnh này không nằm trong kiến trúc RV32I, mà được nằm trong extension RV32M (RISC-V multiply/divided extension).

- Lệnh multiply **mul rd, rs1, rs2**: Nhân hai toán hạng nằm trong thanh ghi rs1, rs2 và ghi 32 bits có trọng số thấp của kết quả vào thanh ghi rd.
- Để ghi 32-bit trọng số cao của kết quả phép nhân, RISC-V cung cấp lệnh **mulh**, **mulhsu**, **mulhu**.
 - o **mulh**: multiply high signed signed, xử lý trên cả hai toán hạng có dấu.
 - o **mulhsu**: multiply high signed unsigned, xử lý trên một toán hạng có dấu, một toán hạng không dấu.
 - o **mulhu**: multiply high unsigned unsigned, xử lý trên cả hai toán hạng không dấu.

Nhập chương trình sau vào công cụ giả lập RARS:

```
# Laboratory Exercise 2, Assignment 5
.text
    # Assign X, Y into t1, t2 register
    addi t1, zero, 4          # X = t1 =?
    addi t2, zero, 5          # Y = t2 =?

    # Expression Z = X * Y
    mul s1, t1, t2           # s1 chứa 32 bit thấp
```

Yêu cầu:

- Biên dịch và quan sát các lệnh mã máy.
- Sử dụng công cụ gõ lỗi, chạy từng lệnh và quan sát sự thay đổi của các thanh ghi. Kiểm tra kết quả xem có đúng không?
- Giải thích lại chi tiết đoạn lệnh trên.
- Tìm hiểu lệnh chia, lấy code minh họa, giải thích luồng hoạt động cũng như kết quả của các thanh ghi.

Assignment 6: Tạo biến và truy cập biến

Support: Chỉ định biên dịch (assembler directives) không phải là lệnh máy. Chỉ thị biên dịch, như tên gọi, là các hướng dẫn cung cấp thêm thông tin cho bộ biên dịch assembler để dịch chương trình cho chính xác.

Đối với máy tính, lệnh và dữ liệu không có sự khác biệt. Tất cả đều là mã nhị phân. Một chuỗi bit có mã hex là 0xF0028293 là một lệnh, hay một biến? Hãy thử xem xét

- Với vai trò của 1 CPU, là phần cứng:
 - o Nếu thanh ghi con trỏ lệnh PC đang trỏ tới chuỗi bit đó, thì đó là một lệnh và được giải mã là addi x5, x5, -256.
 - o Nếu PC không trỏ đến, chuỗi bit đó là một biến số. Hoàn toàn có thể đọc và thay đổi giá trị của biến đó.
- Với vai trò của 1 trình biên dịch, là phần mềm:
 - o Nếu chuỗi bit đó được khai báo sau chỉ thị **.data**, đó là một biến số
 - o Nếu chuỗi bit đó được khai báo sau chỉ thị **.text**, đó là một lệnh và được giải mã là addi x5, x5, -256

Chỉ thị **.data** hay **.text** cũng chỉ là một đánh dấu/bookmark, báo hiệu địa chỉ bắt đầu của 1 vùng nhớ trong RAM, rất đơn giản, sẽ được trình biên dịch xác định là nơi đặt biến số hay lệnh đầu tiên vào đó. Điểm bắt đầu này hoàn toàn chỉ mang tính qui ước nhằm dễ kiểm soát tài nguyên, nên mỗi một CPU, một hệ điều hành, trình biên dịch có thể thiết lập lại các điểm bắt đầu này.

Ví dụ dưới đây đưa ra 2 chỉ thị biên dịch phổ biến nhất là **.data** và **.text** trong việc cấp phát và khởi tạo biến toàn cục, định nghĩa hằng số, và đoạn lệnh được nạp vào phần text

segment, biến được nạp vào data segment, ... (Hiểu rõ luồng chương trình hoạt động để hiểu rõ hơn về assembler directives).

Nhập chương trình sau vào công cụ giả lập RARS:

```
# Laboratory Exercise 2, Assignment 6

.data          # Khởi tạo biến (declare memory)
    X: .word 5      # Biến X, kiểu word (4 bytes), giá trị khởi tạo = 5
    Y: .word -1     # Biến Y, kiểu word (4 bytes), giá trị khởi tạo = -1
    Z: .word 0      # Biến Z, kiểu word (4 bytes), giá trị khởi tạo = 0

.text          # Khởi tạo lệnh (declare instruction)
    # Nạp giá trị X và Y vào các thanh ghi
    la  t5, X      # Lấy địa chỉ của X trong vùng nhớ chứa dữ liệu
    la  t6, Y      # Lấy địa chỉ của Y
    lw  t1, 0(t5)   # t1 = X
    lw  t2, 0(t6)   # t2 = Y

    # Tính biểu thức Z = 2X + Y với các thanh ghi
    add s0, t1, t1
    add s0, s0, t2

    # Lưu kết quả từ thanh ghi vào bộ nhớ
    la  t4, Z      # Lấy địa chỉ của Z
    sw  s0, 0(t4)   # Lưu giá trị của Z từ thanh ghi vào bộ nhớ
```

Yêu cầu:

- Biên dịch và quan sát các lệnh trong cửa sổ Text Segment.
 - o Lệnh **la** (load address) được biên dịch như thế nào? Giải thích cơ chế hoạt động của lệnh **la**. (Gợi ý, để ý tới giá trị thanh ghi **pc**, địa chỉ của nhãn, đọc tài liệu để hiểu cách hoạt động của lệnh **auipc** và **la**).
- Trong cửa sổ Labels, xem địa chỉ của biến X, Y, Z được lưu trong bộ nhớ.
 - o Double click vào nhãn X, Y, Z trong cửa sổ Label để di chuyển đến vị trí của biến tương ứng trong bộ nhớ trong Data Segment. Xác nhận giá trị của biến trong bộ nhớ và giá trị khởi tạo trong mã nguồn.
- Sử dụng công cụ gõ lỗi, chạy từng lệnh và dừng lại.
- Ở mỗi lệnh, quan sát cửa sổ Registers và chú ý:
 - o Sự thay đổi các thanh ghi.
 - o Xác định vai trò của lệnh **lw** và **sw**.
- Tìm hiểu thêm các lệnh **lb**, **sb**.
- **Ghi nhớ qui tắc xử lý:**
 - o Nạp giá trị của biến từ vùng nhớ vào thanh ghi bằng cặp lệnh **la**, **lw**.
 - o Xử lý dữ liệu trên thanh ghi.

- Lưu kết quả từ thanh ghi trở lại vùng nhớ của biến bằng lệnh **la**, **sw**.

Assignment 7: Tạo biến và đặt lệnh ở địa chỉ chính xác

Support: các chỉ thị .data, .text có thể kèm địa chỉ tuyệt đối phía sau, nhằm thay đổi giá trị mặc định. Bằng cách này, có thể yêu cầu chính xác trình biên dịch đặt 1 biến số, hoặc 1 lệnh ở vị trí cố định mong muốn.

Hệ quả:

- Nội dung/Giá trị của 1 biến số có thể thay đổi, nhưng địa chỉ của biến thì luôn là hằng số cố định. (*Không xét tình huống ảo hóa, hoặc thông qua hệ điều hành*)
- Địa chỉ của 1 lệnh luôn không đổi.

Lưu ý:

- Nếu dò tìm được địa chỉ của 1 biến số của phần mềm nào đó, bạn có thể dùng phương pháp này để truy cập trái phép vào chính biến đó và đổi giá trị. Đó là **hack**.
- Nếu dò tìm được địa chỉ của 1 lệnh của phần mềm nào đó, bạn có thể dùng phương pháp này để sửa lệnh đó thành lệnh nhảy tới đoạn mã mà bạn viết. Đó là **virus máy tính**.

```
# Yêu cầu trình biên dịch đặt biến tĩnh bắt đầu từ địa chỉ 0x10011234
.data 0x10011234
x: .word
# Yêu cầu trình biên dịch đặt biến tĩnh bắt đầu từ địa chỉ 0x10014320
.data 0x10014320
y: .word
# Yêu cầu trình biên dịch đặt lệnh tiếp theo bắt đầu từ địa chỉ 0x00408000
.text 0x00408000
    addi x1, zero, 2
```

Kết luận

Bài 3. Các lệnh nhảy và lệnh rẽ nhánh

Mục đích

Sau bài thực hành này, sinh viên sẽ hiểu và biết cách sử dụng các lệnh nhảy và lệnh rẽ nhánh. Từ đó, sinh viên có thể thực hiện các câu lệnh điều khiển, vòng lặp trong ngôn ngữ lập trình bậc cao bằng hợp ngữ, hiểu bản chất của các lệnh đó khi thực hiện trong máy tính.

Tài liệu

Chuẩn bị

Trước khi bắt đầu bài này, sinh viên cần xem trước tài liệu tham khảo và bài thực hành chi tiết, cẩn thận. Để làm tốt, sinh viên cần xem lại và hiểu rõ bài thực hành 2, sử dụng công cụ mô phỏng thành thạo.

Assignments at Home and at Lab

- Lệnh nhảy có điều kiện (khi điều kiện thỏa mãn thì nhảy đến nhãn): RISC-V hỗ trợ 6 lệnh nhảy có điều kiện.

breq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA

- Lệnh nhảy không điều kiện: RISC-V hỗ trợ hai lệnh dưới đây.

jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
jal rd, label	jump and link	PC = JTA, rd = PC + 4

Tuy nhiên ta sử dụng giả lệnh jump (`j label`), lệnh này nhảy đến label được đánh dấu trong mã nguồn.

- Lệnh `slt t1, t2, t3`, thiết lập $t1 = 1$ nếu $t2 < t3$, ngược lại xóa $t1 = 0$.
- Các ví dụ dưới đây minh họa cách thực hiện các cấu trúc thường sử dụng trong ngôn ngữ lập trình bậc cao,
 - Cấu trúc rẽ nhánh if/else
 - Cấu trúc lặp for/while
 - Cấu trúc lựa chọn switch/case

Home Assignment 1

Ví dụ minh họa dưới đây thực hiện cấu trúc rẽ nhánh if-then-else sử dụng các lệnh hợp ngữ được đề cập ở trên.

Cấu trúc cần thực hiện như sau:

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

Trước hết, sinh viên nên vẽ lưu đồ thuật toán cho ví dụ này. Sau đó đọc kỹ đoạn mã hợp ngữ sau đây, hiểu rõ chức năng từng lệnh.

```
# Laboratory Exercise 3, Home Assignment 1
.text
start:
# TODO:
# Initialize i to s1
# Initialize j to s2

# Cách 1:
# blt s2, s1, else    # if j < i then jump else

# Cách 2:
slt    t0, s2, s1      # set t0 = 1 if j < i else clear t0 = 0
bne    t0, zero, else  # t0 != 0 means t0 = 1, jump else

then:  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

Ví dụ dưới đây minh họa cách thực hiện một cấu trúc lặp để tính tổng các phần tử của mảng A.

Mô tả thuật toán với ngôn ngữ lập trình bậc cao (ngôn ngữ lập trình C):

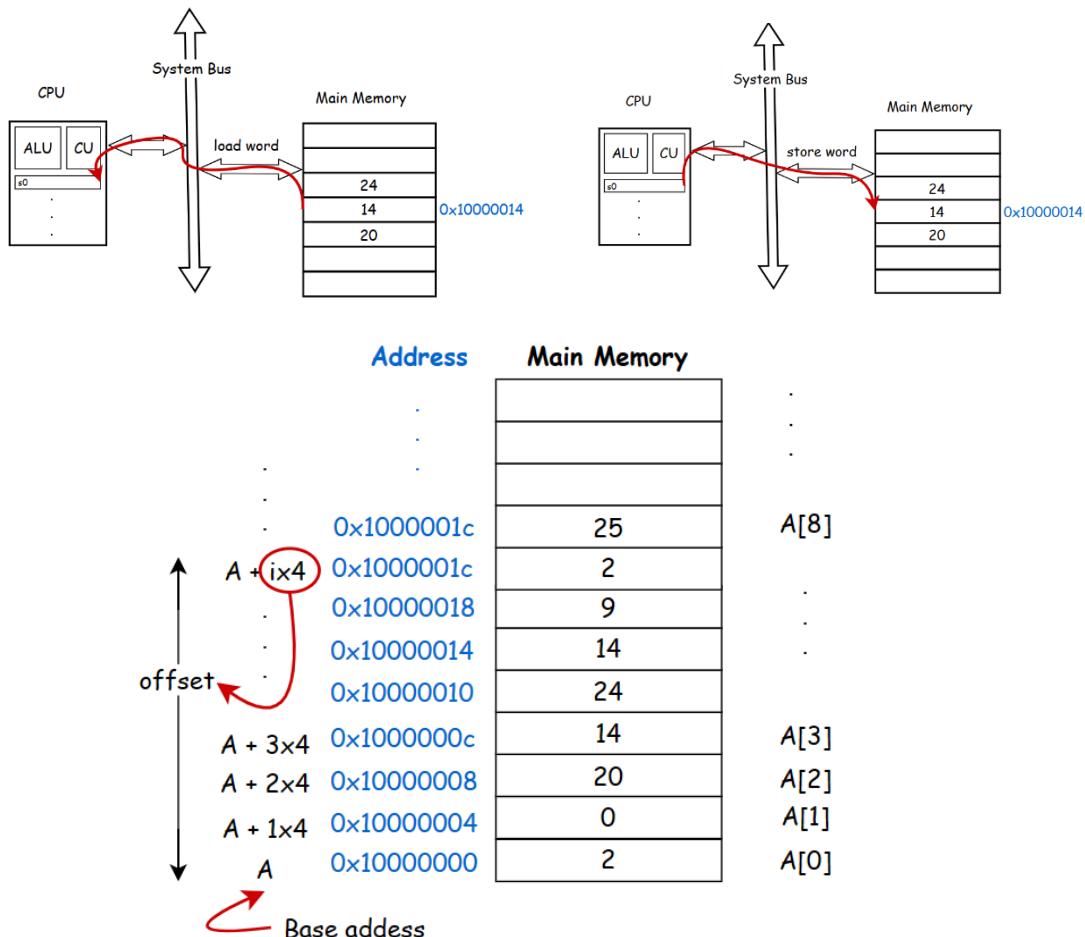
```
sum = 0;
for (int i = 0; i < n; i += step)
    sum += A[i];
```

Mô tả thuật toán với lập trình hợp ngữ:

```
sum = 0
i = 0
loop:   if (i >= n) goto endloop
        sum = sum + A[i]
        i = i + step
        goto loop
endloop:
```

Cách sử dụng mảng trong lập trình hợp ngữ: **A** là biến lưu trữ địa chỉ đầu tiên của mảng – địa chỉ của phần tử **A[0]**, vậy khi muốn truy cập tới phần tử **A[i]**, ta cần truy

cập nhật ô nhớ có địa chỉ $A + 4 \times i$. Sử dụng lệnh **lw** để đọc phần tử mảng vào thanh ghi, và lệnh **sw** để ghi giá trị từ thanh ghi vào phần tử mảng.



Giả sử ta có chỉ số i , địa chỉ cơ sở của mảng A , số phần tử mảng n , bước lặp và kết quả tính tổng được lưu trữ tương ứng các trong thanh ghi $s1, s2, s3, s4, s5$. Đọc kỹ đoạn mã nguồn sau để hiểu cách chương trình hoạt động.

```
# Laboratory 3, Home Assignment 2
.data
    A: .word 1, 3, 2, 5, 4, 7, 8, 9, 6
.text
    # TODO: Initialize s2, s3, s4 registers
    li s1, 0      # i = 0
    li s5, 0      # sum = 0
loop:
    slt t2, s1, s3  # check loop condition i < n
    beq t2, zero, endloop # if i >= n then end loop
    add t1, s1, s1  # t1 = 2 * s1
    add t1, t1, t1  # t1 = 4 * s1 => t1 = 4*i
    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]
    add s1, s1, s4 # i = i + step
    j loop          # go to loop
endloop:
```

Home Assignment 3

Cấu trúc switch/case cho phép rẽ nhánh theo nhiều hướng dựa vào giá trị của một biến số nguyên. Trong ví dụ dưới đây, biến test có thể mang một trong các giá trị 0, 1, 2 và tùy thuộc vào giá trị cụ thể của biến test mà các câu lệnh khác nhau sẽ được thực hiện.

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

Giả sử a và b được lưu trữ trong thanh ghi s2 và s3, giá trị của biến test được khai báo trong bộ nhớ. Đọc kỹ đoạn mã sau và hiểu các thực hiện biểu thức lựa chọn switch/case bằng hợp ngữ.

```
# Laboratory Exercise 3, Home Assignment 3
.data
    test: .word 0
.text
    la s0, test      # Nạp địa chỉ của biến test vào s0
    lw s1, 0(s0)    # Nạp giá trị của biến test vào s1
    li t0, 0          # Nạp giá trị cần kiểm tra
    li t1, 1          # Nạp giá trị cần kiểm tra
    li t2, 2          # Nạp giá trị cần kiểm tra
    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

Tạo project để thực hiện đoạn mã trong Home Assignment 1. Khởi tạo các biến cần thiết. Dịch và mô phỏng với RARS. Chạy chương trình ở chế độ từng dòng lệnh, kiểm tra sự thay đổi của bộ nhớ và nội dung các thanh ghi ở mỗi bước chạy.

Assignment 2

Tạo project để thực hiện đoạn mã trong Home Assignment 2. Khởi tạo các biến cần thiết và mảng A. Dịch và mô phỏng với RARS. Chạy chương trình ở chế độ từng dòng lệnh, quan sát sự thay đổi của bộ nhớ và nội dung các thanh ghi ở mỗi bước chạy. Thay bộ giá trị khác để kiểm tra sự đúng đắn của chương trình.

Assignment 3

Tạo project để thực hiện đoạn mã trong Home Assignment 3. Dịch và mô phỏng với RARS. Chạy chương trình ở chế độ từng dòng lệnh, quan sát sự thay đổi của bộ nhớ và nội dung các thanh ghi ở từng bước chạy. Thay đổi bộ giá trị và chạy lại chương trình một vài lần để kiểm tra tất cả các trường hợp.

Assignment 4

Thay thế điều kiện rẽ nhánh trong Home Assignment 1 bằng các điều kiện mới sau đây:

- a. $i < j$
- b. $i \geq j$
- c. $i + j \leq 0$
- d. $i + j > m + n$ (với m và n được lưu trong các thanh ghi khác)

Assignment 5

Thay thế điều kiện nhảy trong Home Assignment 2 bằng các điều kiện nhảy mới sau đây:

- a. $i \leq n$
- b. $\text{sum} \geq 0$
- c. $A[i] \neq 0$

Assignment 6

Tạo project để thực hiện chương trình sau: Tìm phần tử có giá trị tuyệt đối lớn nhất từ một danh sách các số nguyên 32-bit. Giả sử danh sách số nguyên được lưu trong một mảng biết trước số phần tử.

Kết luận

Trước khi hoàn thành bài thực hành này, suy nghĩ cách giải quyết vấn đề sau: Các lệnh branch khác gì so với các lệnh jump, tại sao chúng được chia thành 2 khuôn dạng khác nhau?

Bài 4. Các lệnh số học và logic

Mục đích

Sau bài thực hành này, sinh viên hiểu được cách sử dụng các lệnh số học (arithmetic instruction), lệnh logic (logical instruction) và lệnh dịch bit (shift instruction). Từ đó, sinh viên cũng hiểu về hiện tượng tràn số học (overflow, carry out) và cách xác định khi tràn số xảy ra.

Tài liệu

- Bài giảng slide về Số học máy tính

Chuẩn bị

Assignments at Home and at Lab

Home Assignment 1

Lệnh số học đã được giới thiệu ở Bài thực hành 2, bài này sẽ đề cập đến trường hợp đặc biệt khi thực hiện lệnh cộng hai số 32-bit, đó là hiện tượng tràn số.

Support: Tổng của hai số nguyên 32-bit có thể không biểu diễn được trong thanh ghi 32-bit (do bị tràn khi cộng module 2 có nhón). Trong trường hợp này, ta nói overflow (carry out) xảy ra. Overflow chỉ có thể xảy xa khi hai toán hạng đó cùng dấu (áp dụng cho số có dấu, còn carry-out là tràn số không dấu).

Khi cộng hai toán hạng không âm (âm), nếu tổng của chúng nhỏ hơn (lớn hơn) các toán hạng, overflow sẽ xảy ra.

Chú ý: RISC-V không có cơ chế tự bảo vệ như các kiến trúc khác (MIPS), vậy để phòng tránh overflow (carry out) xảy ra, phải cài đặt thuật toán phần mềm để bảo vệ dựa trên quy tắc trên.

Chương trình dưới đây xác định tràn số dựa vào quy tắc trên. Hai toán hạng nguồn được lưu vào thanh ghi **s1** và **s2**, toán hạng đích được lưu vào thanh ghi **s3**. Thanh ghi **t0** chứa kết quả Nếu tràn số xảy ra **t0 = 1**, nếu không xảy ra **t0 = 0**.

```
# Laboratory Exercise 4, Home Assignment 1
.text
    # TODO: Initialize s1 and s2 in different cases

    # Algorithm for determining orverflow condition
    li    t0, 0          # No overflow is default status
    add  s3, s1, s2      # s3 = s1 + s2
    xor  t1, s1, s2      # Test if s1 and s2 have the same sign
    blt  t1, zero, EXIT  # If not, exit
    slt  t2, s3, s1
```

```

    blt  s1, zero, 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

Các phép logic cơ bản bao gồm **and**, **or**, **xor** và **not**.

Các lệnh này thao tác trên từng bit của hai thanh ghi nguồn và ghi kết quả vào thanh ghi đích.

		Source registers				
		s1	0100 0110	1010 0001	1111 0001	1011 0111
		s2	1111 1111	1111 1111	0000 0000	0000 0000
Assembly code		Result				
and	s3, s1, s2	s3	0100 0110	1010 0001	0000 0000	0000 0000
or	s4, s1, s2	s4	1111 1111	1111 1111	1111 0001	1011 0111
xor	s5, s1, s2	s5	1011 1001	0101 1110	1111 0001	1011 0111

Các lệnh **and**, **or** và **xor** cũng có phiên bản immediate, lệnh sử dụng một thanh ghi nguồn và một số nguyên có dấu 12-bit.

Một số ứng dụng của các lệnh logic:

- Lệnh **and** thường được sử dụng để trích xuất một hoặc nhiều bit trong thanh ghi.
- Lệnh **and** thường được sử dụng để xóa một hoặc nhiều bit.
- Lệnh **or** thường được sử dụng thiết lập một hoặc nhiều bit.

Chương trình dưới đây minh họa cách trích xuất nội dung thanh ghi sử dụng các lệnh logic. Như đã nói, ta có thể trích xuất 1 bit hoặc nhiều bit có trong thanh ghi. Hãy đọc kỹ ví dụ sau đây để hiểu rõ từng dòng lệnh:

```

# Laboratory Exercise 4, Home Assignment 2
.text
    li   s0, 0x12345678 # Test value
    andi t0, s0, 0xff    # Extract LSB of s0
    andi t1, s0, 0x0400  # Extract bit 10 of s0

```

Home Assignment 3

Loại lệnh tiếp theo được xét trong bài này là lệnh dịch bit (shift instruction). Lệnh này dịch toàn bộ 32-bit của thanh ghi sang trái hoặc sang phải. Các lệnh dịch bit bao gồm **sll** (shift left logical), **srl** (shift right logical), and **sra** (shift right arithmetic). Đối

với lệnh shift right sẽ có hai phiên bản đó là logical (gán tất cả bit mới bằng 0) và arithmetic (gán tất cả các bit mới bằng bit đầu).

Source register			
Assembly code	Result		
slli t0, s5, 7	t0	1000 1110	0000 1000
srli s1, s5, 17	s1	0000 0000	0000 0000
srai t2, s5, 3	t2	1111 1111	1110 0111

RISC-V có hỗ trợ phiên bản immediate (slli, srli, and srai), với 5-bit hằng số. Chú ý rằng, khi dịch trái (phải) N-bit sẽ tương đương với nhân (chia) với chính nó 2^N nếu kết quả không vượt quá giới hạn 32-bit.

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

Assignment 1

Tạo project để thực hiện chương trình ở Home Assignment. Dịch và chạy mô phỏng với RARS. Khởi tạo các toán hạng cần thiết, chạy từng lệnh của chương trình, quan sát bộ nhớ và giá trị thanh ghi.

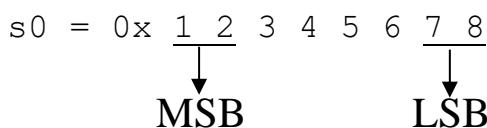
Assignment 2

Viết một chương trình thực hiện các công việc sau:

- Trích xuất MSB của thanh ghi s0
- Xóa LSB của thanh ghi s0
- Thiết lập LSB của thanh ghi s0 (bit 7 đến bit 0 được thiết lập bằng 1)
- Xóa thanh ghi s0 bằng cách dùng các lệnh logic ($s0 = 0$)

MSB: Most Significant Byte (Byte có trọng số cao)

LSB: Least Significant Byte (Byte có trọng số thấp)



Assignment 3

Như đã đề cập, giả lệnh không phải lệnh chính thống của RISC-V, khi biên dịch assembler sẽ chuyển chúng thành các lệnh chính thống. Viết chương trình thực thi các giả lệnh dưới đây sử dụng các lệnh chính thống mà RISC-V định nghĩa:

- a. abs s0, s1
 $s0 = \text{abs}(\$s1)$
- b. move s0, s1
 $s0 = s1$

```
c. not    s0
      s0 = bit_invert(s0)
d. ble    s1, s2, label
      if (s1 <= s2)
          j label
```

Assignment 4

Để xác định tràn số xảy ra khi thực hiện phép cộng, có một cách đơn giản hơn so với cách được mô tả trong Home Assignment 1. Giải thuật được mô tả như sau: Khi cộng hai toán hạng cùng dấu, tràn số xảy ra nếu tổng của chúng không cùng dấu với hai toán hạng nguồn. Hãy viết chương trình xác định tràn số theo giải thuật trên.

Assignment 5

Viết chương trình thực hiện nhân với một lũy thừa của 2 (2, 4, 8, 16, ...) mà không sử dụng lệnh nhân.

Kết luận

Ứng dụng phép dịch bit để thực hiện phép nhân có lợi gì so với sử dụng các lệnh nhân trong extension M (RV32M: RISC-V 32-bit Multiplier/Divider).

Bài 5. Nhập xuất dữ liệu với hàm ECALL, xử lý chuỗi ký tự

Mục đích

Sau bài thực hành này, sinh viên sẽ nắm được cách lưu trữ các chuỗi ký tự trong bộ nhớ. Sinh viên cũng có thể lập trình được ứng dụng xử lý chuỗi ký tự và đưa ra màn hình bằng cách sử dụng lời gọi các dịch vụ hệ thống.

Tài liệu

Chuẩn bị

About ECALL

Một số các dịch vụ hệ thống, chủ yếu liên quan đến nhập xuất dữ liệu, có thể được sử dụng trong các chương trình hợp ngữ. Khi gọi các dịch vụ hệ thống, nội dung các thanh ghi không ảnh hưởng, ngoại trừ các thanh ghi chưa kết quả trả về. Danh sách các dịch vụ mà RARS cung cấp có thể được tra cứu trong menu Help, thẻ Syscalls.

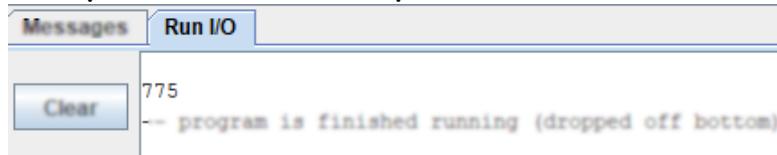
Các bước thực hiện một lời gọi hệ thống

1. Nạp số hiệu dịch vụ vào thanh ghi a7.
2. Nạp các tham số nếu có vào các thanh ghi được chỉ định, ví dụ a0, a1, a2, ...
3. Thực hiện lệnh ecall.
4. Nhận kết quả, nếu có, từ các thanh ghi được chỉ định.

Ví dụ: Hiển thị một số nguyên ra cửa sổ Run I/O

```
.text
    li    a7, 1      # 1 là số hiệu dịch vụ in số nguyên
    li    a0, 0x307 # giá trị cần in
    ecall           # thực hiện lời gọi hệ thống
```

Kết quả được hiển thị ở cửa sổ Run I/O ở cạnh dưới màn hình như sau



Một số dịch vụ thường dùng

1. In số nguyên

Hiển thị một số nguyên ra cửa sổ Run I/O.

Tham số:

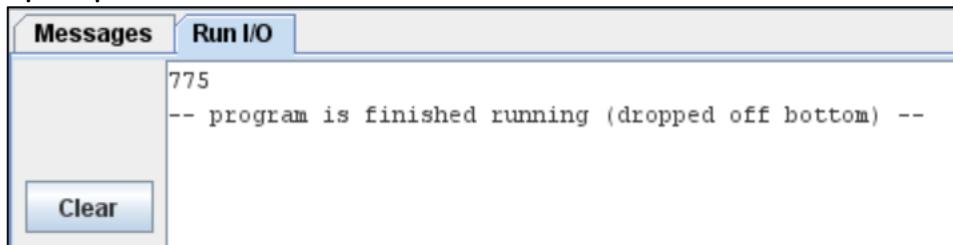
a7 = 1

a0 = số nguyên cần in
Giá trị trả về:
Không có

Ví dụ:

```
.text
    li a7, 1
    li a0, 0x307
    ecall
```

Kết quả thực hiện:



2. In số nguyên trong hộp thoại

Hiển thị số nguyên trong hộp thoại kèm theo thông điệp.

Tham số:

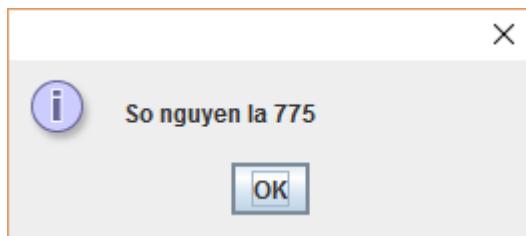
a7 = 56
a0 = địa chỉ chuỗi ký tự thông điệp
a1 = số nguyên cần in

Giá trị trả về:
Không có

Ví dụ:

```
.data
message: .asciz "So nguyen la: "
.text
    li a7, 56
    la a0, message
    li a1, 0x307
    ecall
```

Kết quả thực hiện:



3. In chuỗi ký tự

In chuỗi ký tự ra màn hình Run I/O.

Tham số:

a7 = 4
a0 = địa chỉ chuỗi cần in

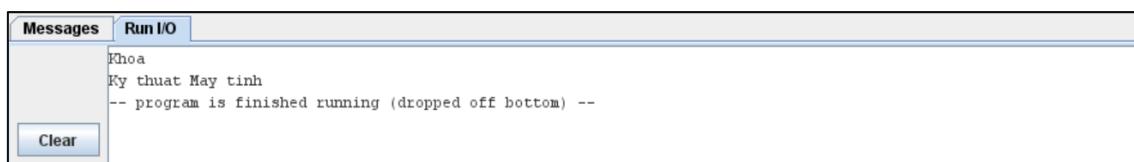
Kết quả trả về:

Không có

Ví dụ:

```
.data
message: .asciz "Khoa \nKy thuat May tinh"
.text
    li a7, 4
    la a0, message
    ecall
```

Kết quả thực hiện:



4. In chuỗi ký tự trong hộp thoại

Hiển thị chuỗi ký tự trong hộp thoại kèm theo chuỗi thông điệp.

Tham số:

a7 = 59
a0 = địa chỉ của chuỗi thông điệp
a1 = địa chỉ chuỗi cần in

Kết quả trả về:

Không có

Ví dụ:

```
.data
message: .asciz "Truong: "
address: .asciz "Cong nghe thong tin va Truyen thong"
.text
    li a7, 59
    la a0, message
    la a1, address
    ecall
```

Kết quả thực hiện



5. Nhập số nguyên

Nhập một số nguyên từ bàn phím.

Tham số:

a7 = 5

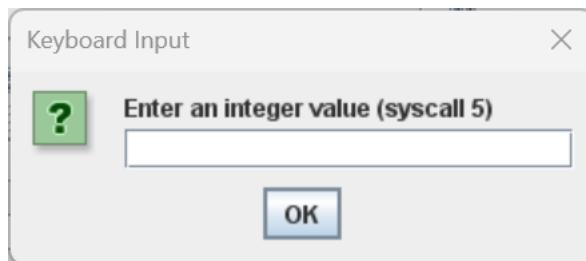
Giá trị trả về:

a0 = số nguyên được nhập

Ví dụ:

```
.text
    li a7, 5
    ecall
```

Kết quả thực hiện:



Chú ý: Các dịch vụ 5, 6, 7, 8, 12 có thể nhập dữ liệu trong cửa sổ Run I/O hoặc trong hộp thoại popup bằng cách thay đổi lựa chọn ở menu Settings / Popup dialog for input syscalls (5, 6, 7, 8, 12)

6. Nhập chuỗi ký tự

Nhập một chuỗi ký tự từ bàn phím.

Tham số:

a7 = 8

a0 = địa chỉ buffer chứa chuỗi ký tự được nhập

a1 = độ dài tối đa được nhập

Giá trị trả về:

Không có

Chú ý:

Với độ dài tối đa được nhập là n (a1), chuỗi ký tự có độ dài không vượt quá n-1.

- Nếu độ dài ngắn hơn, ký tự xuống dòng được chèn vào cuối.
- Với trường hợp còn lại, giá trị null được đặt ở cuối chuỗi (vị trí n-1).

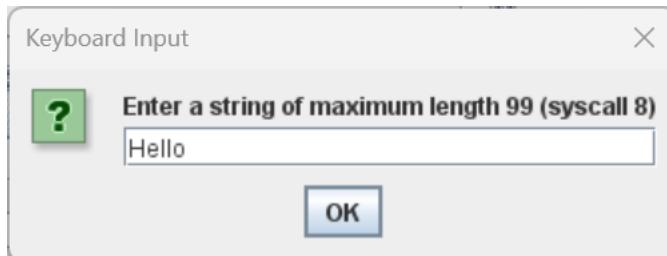
Trường hợp đặc biệt:

- Nếu n = 1, việc nhập bị bỏ qua, giá trị null được đặt tại địa chỉ đầu của buffer.
- Nếu n < 1, việc nhập bị bỏ qua, không giá trị nào được ghi vào buffer.

Ví dụ:

```
.data
buffer: .space 100      # Buffer 100 byte chua chuoi ki tu
.text
    li  a7, 8
    la  a0, buffer
    li  a1, 100
    ecall
```

Kết quả thực hiện:



Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	1 l e H	\0 \0 \n \0	\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
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100A0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100C0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100E0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010100	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

7. Nhập chuỗi ký tự với hộp thoại

Hiển thị hộp thoại thông điệp để nhập chuỗi ký tự, cho phép kiểm tra kết quả nhập dữ liệu.

Tham số:

- a7 = 54
- a0 = địa chỉ chuỗi thông điệp
- a1 = địa chỉ buffer chứa dữ liệu nhập được
- a2 = số ký tự tối đa cần nhập

Giá trị trả về:

a1 chứa trạng thái kết quả:

- 0: Nhập dữ liệu thành công.
- 2: Nút Cancel được nhấn. Buffer không thay đổi.
- 3: Nút OK được nhấn nhưng không có dữ liệu nhập vào. Buffer không thay đổi.
- 4: Độ dài chuỗi ký tự nhập vào vượt quá độ dài tối đa đã khai báo. Buffer chứa số ký tự tối đa cho phép kèm theo ký tự null kết thúc xâu.

Ví dụ:

```
.data
message: .asciz "Ho va ten sinh vien:"
buffer:   .space   100
.text
    li    a7, 54
    la    a0, message
    la    a1, buffer
    li    a2, 100
    ecall
```

Kết quả thực hiện:



8. In ký tự

In một ký tự ra cửa sổ Run I/O.

Tham số:

a7 = 11
a0 = Ký tự cần in (có thể sử dụng số nguyên chứa mã ASCII của ký tự hoặc bản thân ký tự trong cặp dấu nháy đơn)

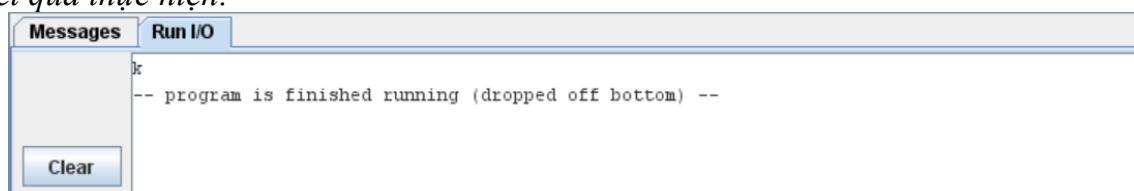
Giá trị trả về:

Không có

Ví dụ:

```
.text
    li    a7, 11
    li    a0, 'k'
    ecall
```

Kết quả thực hiện:



9. Nhập ký tự

Nhập một ký tự từ bàn phím.

Tham số:

a7 = 12

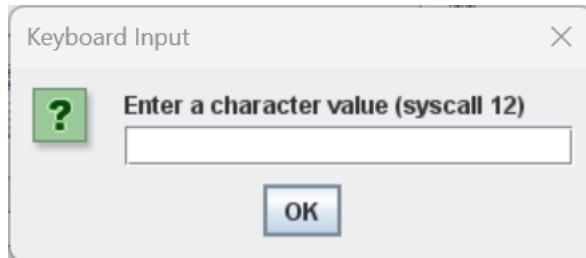
Giá trị trả về:

a0 = Ký tự được nhập

Ví dụ:

```
.text
    li    a7, 12
    ecall
```

Kết quả thực hiện:



10. ConfirmDialog

Hiển thị hộp thoại thông điệp với 3 lựa chọn Yes | No | Cancel
Tham số:

a7 = 50
a0 = địa chỉ chuỗi thông điệp

Giá trị trả về:

a0 = chứa giá trị người dùng đã chọn
0: Yes
1: No
2: Cancel

Ví dụ:

```
.data
question: .asciz "Ban co phai la SV Ky thuat May tinh?"
.text
    li    a7, 50
    la    a0, question
    ecall
```

Kết quả thực hiện:



11. MessageDialog

Hiển thị hộp thoại chứa thông điệp kèm theo biểu tượng.

Tham số:

a7 = 55
a0 = địa chỉ chuỗi thông điệp
a1 = kiểu thông điệp được hiển thị:
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)

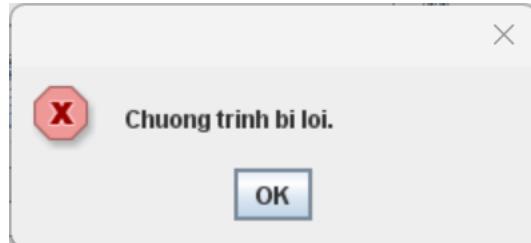
Giá trị trả về:

Không có

Ví dụ:

```
.data
message: .asciz "Chuong trinh bi loi."
.text
    li    a7, 55
    la    a0, message
    li    a1, 0
    ecall
```

Kết quả thực hiện:



12. Exit

Kết thúc chương trình. Không có lệnh Exit trong tập lệnh của bất kỳ bộ xử lý nào. Exit là một dịch vụ của hệ điều hành.

Tham số:

a7 = 10

Giá trị trả về:

Không có

Ví dụ:

```
li    a7, 10
ecall
```

Assignments at Home and at Lab

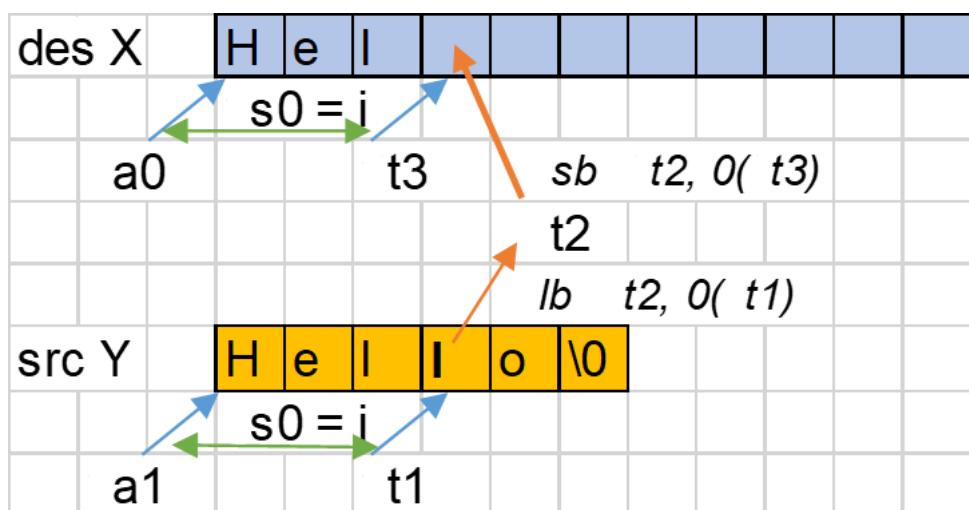
Home Assignment 1

Chương trình hợp ngữ đơn giản sau in ra màn hình một dòng ký tự “Hello world”. Ta dùng hàm **printf** để làm điều đó. Đọc kỹ đoạn mã này, chú ý đến cách truyền tham số cho hàm **printf**.

```
# Laboratory Exercise 5, Home Assignment 1
.data
test: .asciz "Hello World"
.text
    li  a7, 4
    la  a0, test
    ecall
```

Home Assignment 2

Hàm `strcpy` trong ngôn ngữ C, thực hiện việc sao chép chuỗi y vào chuỗi x sử dụng ký tự null '\0' để đánh dấu kết thúc chuỗi. Chương trình dưới đây minh họa việc thực hiện hàm này. Đọc kỹ và hiểu chương trình này.



```
# Laboratory Exercise 5, Home Assignment 2
.data
x: .space 32          # destination string x, empty
y: .asciz "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
    addi  s0, s0, 1        # s0=s0 + 1 <-> i=i+1
    j     L1                # next character
end_of_strcpy:
```

Home Assignment 3

Chương trình sau thực hiện đếm số ký tự của một chuỗi ký tự kết thúc bằng ký tự null. Đọc hiểu đoạn mã sau đây.

```
# Laboratory Exercise 5, Home Assignment 3
```

```
.data
string: .space 50
message1: .asciz "Nhập xau: "
message2: .asciz "Do dai xau la: "

.text
main:
get_string:
    # TODO Input string from keyboard
get_length:
    la    a0, string          # a0 = address(string[0])
    li    t0, 0                # 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  t0, t0, 1            # t0 = t0 + 1 -> i = i + 1
    j     check_char
end_of_str:
end_of_get_length:
print_length:
    # TODO print result to console
```

Assignment 1

Tạo project thực hiện Home Assignment 1. Dịch và nạp chương trình lên trình mô phỏng. Chạy và quan sát kết quả. Chuyển đến Data Segment, kiểm tra cách chuỗi ký tự được lưu trữ trong bộ nhớ.

Assignment 2

Tạo project thực hiện chương trình in tổng của hai toán hạng nằm trong thanh ghi s0 và s1 theo định dạng sau:

“The sum of (s0) and (s1) is (result)”

Assignment 3

Tạo project thực hiện Home Assignment 2. Đọc hiểu mã nguồn, khởi tạo các biến cần thiết cho chương trình, thực hiện hàm **strcpy**. Dịch và nạp lên mô phỏng, chạy và quan sát kết quả.

Assignment 4

Tạo project thực hiện Home Assignment 3, sử dụng ecall để nhập chuỗi ký tự cần đếm, và in kết quả ra màn hình.

Assignment 5

Viết chương trình cho phép người dùng nhập chuỗi ký tự bằng cách nhập từng ký tự. Việc nhập sẽ kết thúc khi người dùng nhấn Enter hoặc khi độ dài chuỗi ký tự vượt quá 20. In chuỗi đã nhập theo chiều ngược lại.

Kết luận

Bài 6. Mảng và con trỏ

Mục đích

Sau bài thực hành này, sinh viên có thể hiểu cách thức mảng và con trỏ được biểu diễn và phân biệt được cách sử dụng mảng và con trỏ để duyệt các phần tử của mảng.

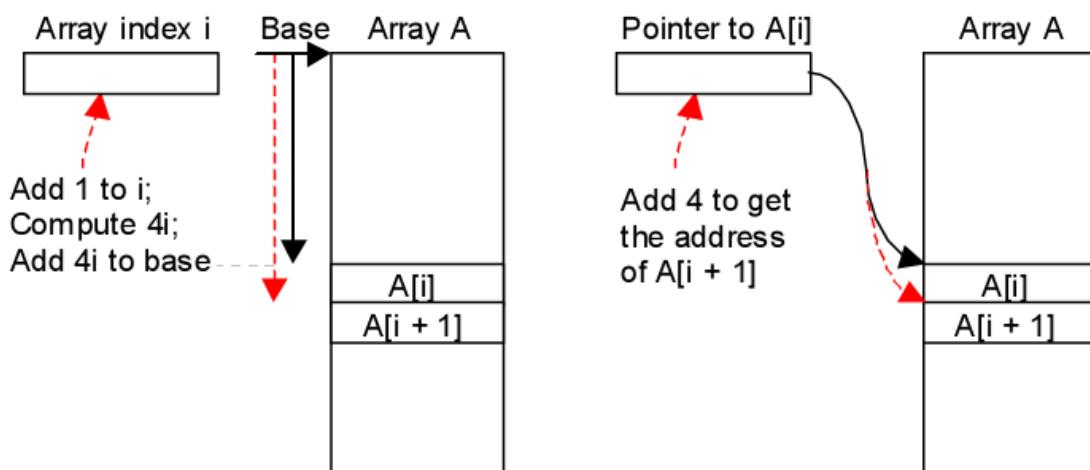
Tài liệu

Chuẩn bị

Mảng và con trỏ

Trong các bài toán lập trình, có nhiều bài toán liên quan đến dữ liệu kiểu mảng, cần phải duyệt các phần tử của mảng. Ví dụ, tìm phần tử có giá trị lớn nhất trong mảng, sắp xếp các phần tử của mảng. Có hai phương pháp cơ bản để duyệt các phần tử của mảng thông qua chỉ số mảng (index) hoặc con trỏ (pointer).

1. Index: dùng thanh ghi để lưu trữ giá trị chỉ số i của phần tử cần truy nhập và tăng hoặc giảm chỉ số để truy nhập đến các phần tử khác của mảng.
2. Pointer: dùng thanh ghi để lưu trữ địa chỉ của phần tử cần truy nhập, và thay đổi giá trị của thanh ghi để trỏ đến các phần tử khác của mảng.



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

Assignments at Home and at Lab

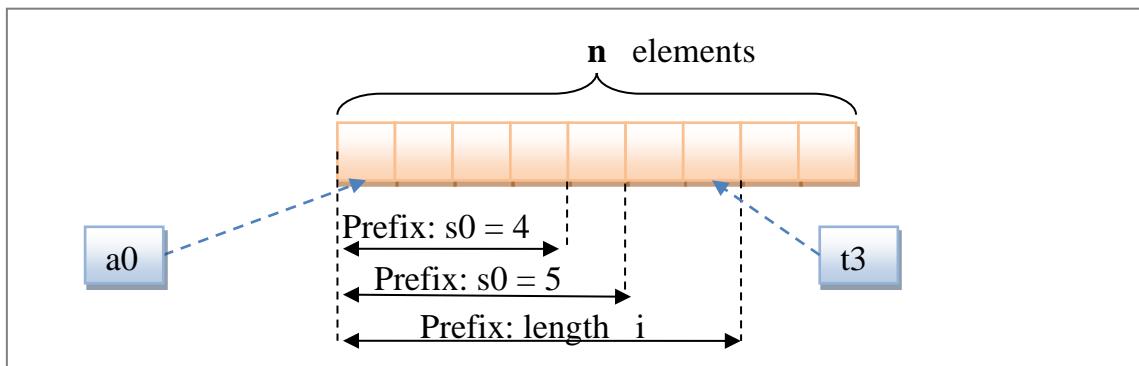
Home Assignment 1

Thuật toán tìm tổng tiền tố lớn nhất (maximum prefix-sum).

Cho dãy số nguyên có chiều dài n , tiền tố có chiều dài i bao gồm i phần tử đầu tiên trong mảng với $1 \leq i \leq n$. Một tổng tiền tố là tổng của i phần tử đầu tiên trong dãy số nguyên đó. Tìm tổng tiền tố của dãy số đã cho.

Ví dụ: Cho dãy số nguyên $(2, -3, 2, 5, -4)$, tổng tiền tố lớn nhất chứa 4 phần tử đầu tiên và tổng tương ứng là $(2+ -3 + 2 + 5) = 6$.

Chương trình dưới đây thực hiện thuật toán trên bằng phương pháp chỉ số. Hãy đọc kỹ và hiểu cách chương trình hoạt động.



```

.data
A: .word -2, 6, -1, 3, -2
.text
main:
    la    a0, A
    li    a1, 5
    j     mspfx
continue:
exit:
    li    a7, 10
    ecall
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) needs to be processed
# @param[in] a1 the number of elements in list(A)
# @param[out] s0 the length of sub-array of A in which max sum reaches.
# @param[out] s1 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:
    li    s0, 0          # initialize length of prefix-sum in s0 to 0
    li    s1, 0x80000000 # initialize max prefix-sum in s1 to smallest int
    li    t0, 0           # initialize index for loop i in t0 to 0

```

```

    li    t1, 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
    blt s1, t1, mdfy    # if(s1 < t1) modify results
    j    next
next:
    addi s0, t0, 1       # new max-sum prefix has length i+1
    addi s1, t1, 0       # new max sum is the running sum
done:
    j    continue
mspx_end:

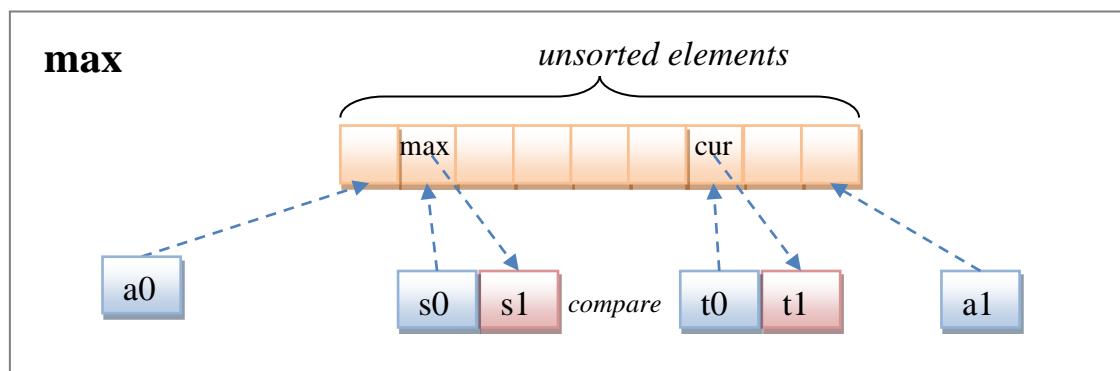
```

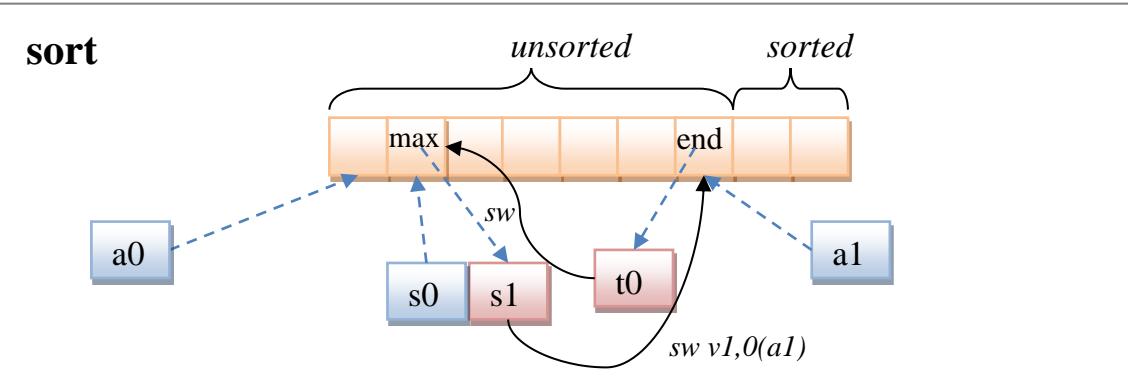
Home Assignment 2

Thuật toán sắp xếp lựa chọn (selection sort).

Một mảng số nguyên gồm n phần tử có thể được sắp xếp theo thứ tự tăng dần như sau. Tìm phần tử có giá trị lớn nhất trong danh sách và đổi chỗ nó với phần tử cuối cùng trong dãy. Phần tử cuối cùng đã được đặt đúng vị trí. Tiếp tục thực hiện các bước trên với $n - 1$ phần tử chưa được sắp xếp cho đến khi chỉ còn lại 1 phần tử. Khi đó thuật toán kết thúc, mảng được sắp xếp theo thứ tự tăng dần.

Chương trình dưới đây minh họa việc thực hiện thuật toán sắp xếp lựa chọn bằng phương pháp truy nhập kiểu con trỏ. Hãy đọc kỹ và hiểu cách thực hiện của chương trình.





```

.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    a7, 10
    ecall
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
# s0 pointer to max element in unsorted part
# s1 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(s0)        # copy last element to max location
    sw    s1, 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 s0, a0, 0    # init max pointer to first element
    lw    s1, 0(s0)   # 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
    blt  t1, s1, loop # if (next)<(max), repeat
    addi s0, t0, 0    # next element is new max element
    addi s1, t1, 0    # next value is new max value
    j     loop        # change completed; now repeat
ret:
    j     after_max
```

Assignment 1

Tạo project thực hiện chương trình trong Home Assignment 1. Khởi tạo bộ giá trị mới cho mảng, dịch và nạp lên mô phỏng. Chạy chương trình từng bước một và quan sát sự thay đổi các thanh ghi để kiểm nghiệm chương trình hoạt động đúng với thuật toán.

Assignment 2

Tạo mới một project thực hiện chương trình trong Home Assignment 2. Khởi tạo bộ giá trị mới cho mảng, dịch và nạp lên mô phỏng. Chạy chương trình từng bước một và quan sát sự thay đổi các thanh ghi để kiểm nghiệm chương trình hoạt động đúng với thuật toán. Viết thêm chương trình con để in ra mảng sau mỗi lượt sắp xếp.

Assignment 3

Viết chương trình thực hiện thuật toán sắp xếp nổi bọt (bubble sort).

Assignment 4

Viết chương trình thực hiện thuật toán sắp xếp chèn (insertion sort).

Kết luận

Bài 7. Lệnh gọi chương trình con, truyền tham số sử dụng ngăn xếp

Mục đích

Sau bài thực hành này, sinh viên có thể hiểu cách gọi chương trình con và cơ chế hoạt động của ngăn xếp. Ngoài ra sinh viên có thể tự viết được các chương trình con sử dụng ngăn xếp để truyền tham số và trả về kết quả.

Tài liệu

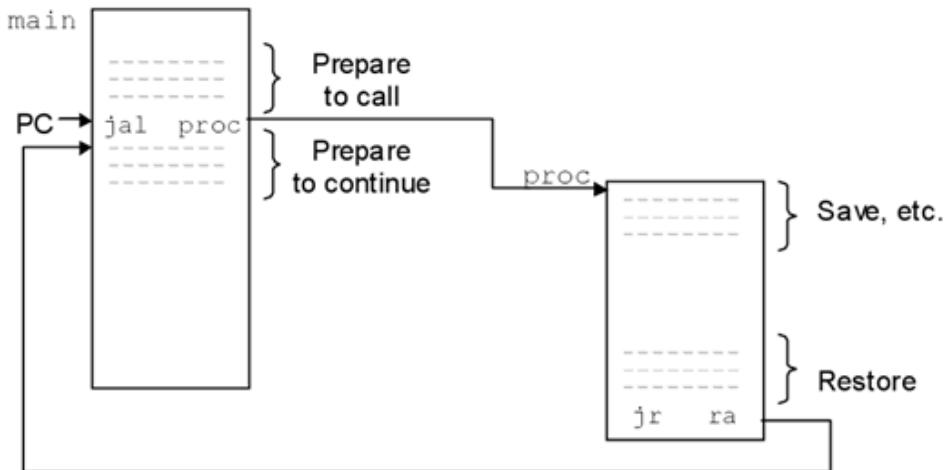
Chuẩn bị

Gọi chương trình con

Thủ tục (chương trình con) là một đoạn mã nguồn thực hiện một công việc cụ thể nào đó, có thể trả về một hoặc nhiều kết quả, dựa trên các tham số đầu vào. Khi chương trình con kết thúc, bộ xử lý sẽ quay trở lại vị trí gọi chương trình con để thực hiện các công việc tiếp theo sau lời gọi chương trình con.

Trong lập trình hợp ngữ, một chương trình con thường kết hợp với một nhãn để đánh dấu địa chỉ bắt đầu của chương trình con. Có 2 lệnh được sử dụng khi làm việc với chương trình con.

- Lệnh **jal rd, label** (jump and link), thực hiện lưu địa chỉ của lệnh tiếp theo ($pc + 4$) vào thanh ghi **rd** và nhảy đến câu lệnh sau nhãn **label**. Địa chỉ này sẽ được sử dụng để quay trở về chương trình chính khi kết thúc chương trình con. Giả lệnh **jal label** (tương đương với lệnh **jal ra, label**) lưu địa chỉ trả về vào thanh ghi **ra**, thường được sử dụng để gọi chương trình còn.
- Lệnh **jalr rd, rs1, imm** (jump and link register), thực hiện lưu địa chỉ của lệnh tiếp theo ($pc + 4$) vào thanh ghi **rd** và nhảy đến lệnh có địa chỉ **rs1 + imm**. Giả lệnh **jr ra** (tương đương với lệnh **jalr zero, ra, 0**), nhảy đến lệnh có địa chỉ lưu trong thanh ghi **ra**, thường được sử dụng để quay trở lại chương trình chính.



Relationship between the main program and a procedure

Assignments at Home and at Lab

Home Assignment 1

Chương trình dưới đây minh họa việc khai báo và sử dụng hàm **abs** để tính giá trị tuyệt đối của một số nguyên. Hàm sử dụng 2 thanh ghi, **a0** chứa tham số vào và **s0** chứa kết quả. Đọc kỹ chương trình và hiểu cách khai báo và gọi chương trình con.

```
# Laboratory Exercise 7 Home Assignment 1
.text
main:
    li  a0, -45      # load input parameter
    jal abs          # jump and link to abs procedure

    li  a7, 10        # terminate
    ecall
end_main:
# -----
# function abs
# param[in]  a0      the interger need to be gained the absolute
# value
# return      s0      absolute value
# -----
abs:
    sub s0, zero, a0    # put -a0 in s0; in case a0 < 0
    blt a0, zero, done  # if a0<0 then done
    add s0, a0, zero    # else put a0 in s0
done:
    jr  ra
```

Home Assignment 2

Trong ví dụ này, chương trình con max được khai báo và sử dụng để tìm phần tử lớn nhất trong 3 số nguyên. Các tham số này được truyền vào chương trình con qua các thanh ghi a0, a1, a2. Kết quả được lưu vào thanh ghi s0. Đọc kỹ chương trình và hiểu cách khai báo và gọi chương trình con.

```

# 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

    li    a7, 10     # terminate
    ecall
end_main:

# -----
# Procedure max: find the largest of three integers
# param[in]  a0  integers
# param[in]  a1  integers
# param[in]  a2  integers
# return    s0  the largest value
# -----
max:
    add   s0, a0, zero    # copy a0 in s0; largest so far
    sub   t0, a1, s0      # compute a1 - s0
    blt   t0, zero, okay  # if a1 - v0 < 0 then no change
    add   s0, a1, zero    # else a1 is largest thus far
okay:
    sub   t0, a2, s0      # compute a2 - v0
    bltz  t0, zero, done  # if a2 - v0 < 0 then no change
    add   s0, a2, zero    # else a2 is largest overall
done:
    jr   ra                # return to calling program

```

Home Assignment 3

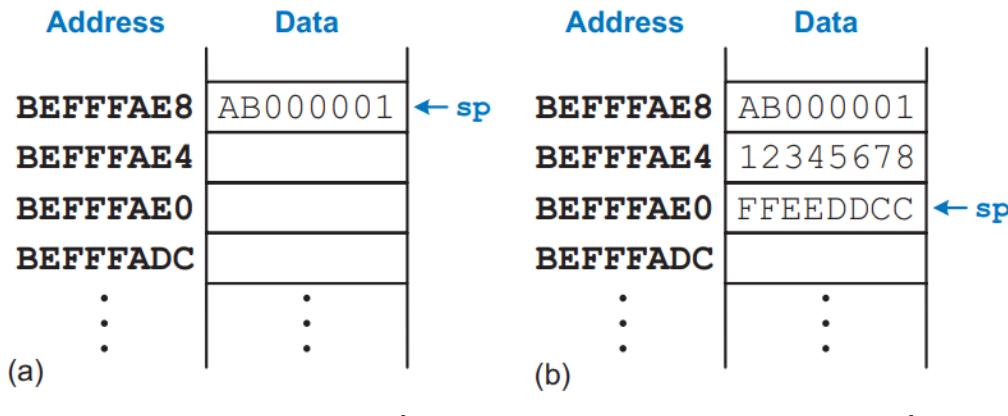
Support:

Vùng nhớ ngăn xếp bản chất là một **mảng** hoạt động dựa trên nguyên lý **Last In First Out (LIFO)**, thường được quản lý bởi thanh ghi **sp** (stack pointer). Thanh ghi **sp** lưu trữ địa chỉ (trỏ) phần tử ở đỉnh ngăn xếp, sử dụng thanh ghi **sp** để thực hiện hai thao tác **push** và **pop**.

Cách sử dụng ngăn xếp:

1. Tạo khoảng trống để lưu trữ nội dung một hoặc nhiều thanh ghi (giảm giá trị của thanh ghi **sp** ứng với số byte cần cấp phát).
2. Lưu trữ nội dung các thanh ghi cần lưu trữ vào ngăn xếp (dùng lệnh **sw**).
3. Thực hiện chương trình sử dụng các thanh ghi này.
4. Khôi phục lại giá trị ban đầu cho các thanh ghi (dùng lệnh **lw**)
5. Trả lại vùng nhớ ngăn xếp đã cấp phát (khôi phục giá trị ban đầu cho thanh ghi **sp**)

Trong RISC-V, đáy của ngăn xếp được lưu ở địa chỉ cao, các phần tử tiếp theo được lưu trữ ở địa chỉ thấp hơn.



Chương trình hợp ngữ dưới đây minh họa cách sử dụng vùng nhớ ngăn xếp (stack) với hai phép toán **push** và **pop** bằng cách sử dụng lệnh **lw** và **sw**. Giá trị của hai thanh ghi **s0** và **s1** sẽ được hoán đổi với nhau sử dụng ngăn xếp.

```
# 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
```

Support:

Với các quy ước trong RISC-V, các tham số vào thường được lưu trong các thanh ghi a0-a7, còn giá trị trả về được lưu trong thanh ghi a0. Các câu hỏi cần giải quyết:

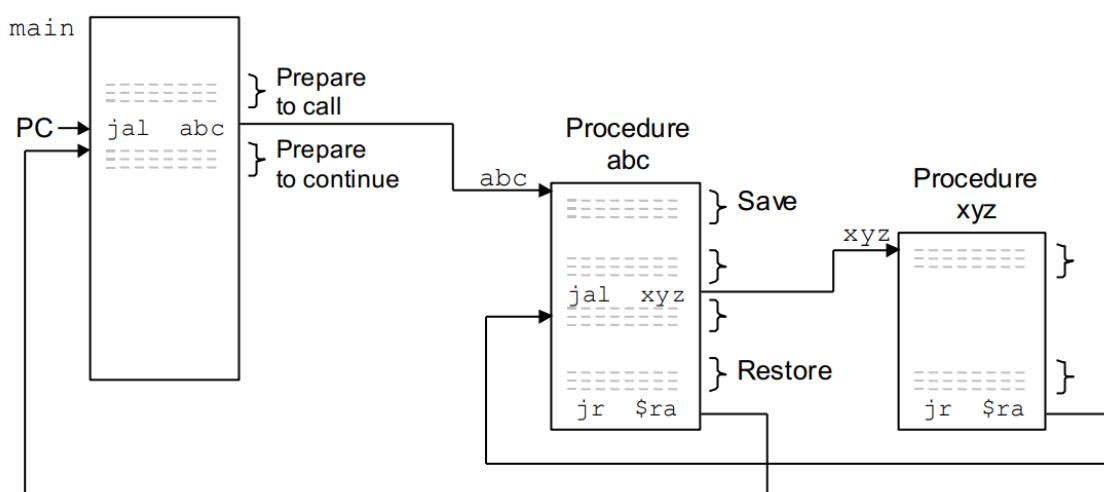
1. Khi hàm có nhiều hơn 8 tham số hoặc có nhiều hơn 1 giá trị trả về?
 2. Khi hàm cần lưu giá trị tham số đầu vào và trạng thái của nó để gọi một hàm khác?
 3. Khi chương trình có quá nhiều biến local không lưu trữ đủ trong 32 thanh ghi?
- Sử dụng vùng nhớ **Ngăn xếp (Stack)**

Quy ước với hàm cha (caller) và hàm con (callee)

1. **Caller rule:** Trước khi gọi callee, caller lưu nội dung các thanh ghi chứa các tham số đầu vào và thanh ghi tạm của nó vào đỉnh stack (a0-a7 và t0-t6), để dành các thanh ghi này cho callee sử dụng. Sau khi callee thực hiện xong, khôi phục nội dung các thanh ghi đã được lưu trữ trước đó.

2. **Callee rule:** Trước khi thực hiện, phải lưu trữ nội dung các thanh ghi mà callee muốn sử dụng (`s0-s11` và `ra`). Trước khi quay trở lại caller, phải khôi phục nội dung thanh ghi đã sử dụng trước đó.
 → Hai quy ước này sẽ đảm bảo chương trình con được sử dụng nhiều thanh ghi phục vụ cho chương trình nhất có thể.

Chú ý đối với thủ tục lồng nhau: khi gọi thủ tục `xyz` cần phải lưu trữ thanh ghi `ra` (hiện thời chưa địa chỉ trả về của hàm `abc`) vào stack rồi mới nhảy tới hàm `xyz`, vì nếu không khi nhảy tới `xyz` thì giá trị hiện thời `ra` được ghi đè bởi địa chỉ trả về của `xyz` nên hàm sẽ không thể quay trở lại được chương trình chính.



Home Assignment 4

Chương trình sau đây sử dụng thuật toán đệ quy để tính $n!$. Đọc kỹ chương trình để hiểu cách sử dụng ngăn xếp để lưu trữ và khôi phục các thanh ghi.

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

.text
main:
    jal    WARP

print:
    add    a1, s0, zero      # a0 = result from N!
    li     a7, 56
    la     a0, message
    ecall

quit:
    li     a7, 10            # terminate
    ecall
end_main:

#
# Procedure WARP: assign value and call FACT
# -----
```

```

WARP:
    addi    sp, sp, -4    # adjust stack pointer
    sw      ra, 0(sp)    # save return address

    li      a0, 3        # load test input N
    jal     FACT         # call fact procedure

    lw      ra, 0(sp)    # restore return address
    addi    sp, sp, 4    # return stack pointer
    jr      ra

wrap_end:

# -----
# Procedure FACT: compute N!
# param[in]  a0  integer N
# return     s0  the largest value
# -----
FACT:
    addi    sp, sp, -8  # allocate space for ra, a0 in stack
    sw      ra, 4(sp)  # save ra register
    sw      a0, 0(sp)  # save a0 register

    li      t0, 2
    bge   a0, t0, recursive
    li      s0, 1        # return the result N!=1
    j       done

recursive:
    addi   a0, a0, -1  # adjust input argument
    jal    FACT         # recursive call
    lw     s1, 0(sp)    # load a0
    mul   s0, s0, s1

done:
    lw      ra, 4(sp)  # restore ra register
    lw      a0, 0(sp)  # restore a0 register
    addi   sp, sp, 8    # restore stack pointer
    jr      ra          # jump to caller

fact_end:

```

Assignment 1

Tạo project để thực hiện Home Assignment 1. Dịch và chạy mô phỏng. Thay đổi các tham số chương trình (thanh ghi **a0**) và quan sát kết quả thực hiện. Chạy chương trình ở chế độ từng dòng lệnh và chú ý sự thay đổi của các thanh ghi, đặc biệt là thanh ghi **pc** và **ra**.

Assignment 2

Tạo project để thực hiện Home Assignment 2. Dịch và chạy mô phỏng. Thay đổi các tham số chương trình (thanh ghi **a0, a1, a2**) và quan sát kết quả thực hiện. Chạy chương trình ở chế độ từng dòng lệnh và chú ý sự thay đổi của các thanh ghi, đặc biệt là thanh ghi **pc** và **ra**.

Assignment 3

Tạo project để thực hiện Home Assignment 3. Dịch và chạy mô phỏng. Thay đổi tham số chương trình (thanh ghi **s0, s1**), quan sát quá trình và kết quả thực hiện. Chú ý sự thay

đổi giá trị của thanh ghi **sp**. Quan sát vùng nhớ được trỏ bởi thanh ghi **sp** trong cửa sổ Data Segment.

Assignment 4

Tạo project để thực hiện Home Assignment 4. Dịch và chạy mô phỏng. Thay đổi tham số ở thanh ghi **a0** và kiểm tra kết quả ở thanh ghi **s0**. Chạy chương trình ở chế độ từng dòng lệnh và quan sát sự thay đổi giá trị của các thanh ghi **pc**, **ra**, **sp**, **a0**, **s0**. Liệt kê các giá trị trong vùng nhớ ngăn xếp khi thực hiện chương trình với $n = 3$.

Assignment 5

Viết chương trình con tìm giá trị lớn nhất, nhỏ nhất và vị trí tương ứng trong danh sách gồm 8 số nguyên được lưu trữ trong các thanh ghi từ $a0$ đến $a7$. Ví dụ:

Largest: 9, 3 => Giá trị lớn nhất là 9 được lưu trữ trong $a3$

Smallest: -3, 6 => Giá trị nhỏ nhất là -3 được lưu trữ trong $a6$

Gợi ý: Sử dụng ngăn xếp để truyền tham số.

Kết luận

Bài 8, 9. Project giữa kỳ

Mục đích

Sinh viên ôn lại các kiến thức đã được học trong các bài thực hành để lập trình giải quyết các bài toán cơ bản.

Time

Week 8: Sinh viên làm bài project trên lớp.

Week 9: Sinh viên trình bày kết quả đã thực hiện với giảng viên.

Bài 10. Giao tiếp với các thiết bị ngoại vi

Mục đích

Sau bài thực hành này, sinh viên có thể hiểu được cách giao tiếp với các thiết bị ngoại vi của công cụ giả lập.

Tài liệu

Cách thức CPU giao tiếp với các thiết bị nhập và xuất ví dụ như màn hình hoặc bàn phím?

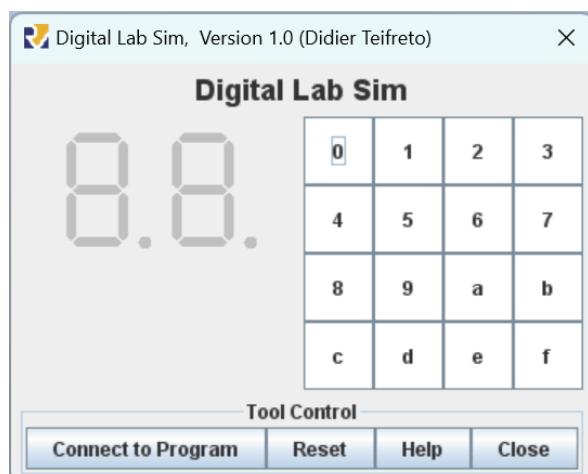
Memory-mapped I/O (MMIO) và port-mapped I/O (PMIO) là 2 phương pháp bổ sung cho nhau để thực hiện các thao tác vào ra giữa CPU và các thiết bị ngoại vi trong máy tính.

MMIO sử dụng cùng một không gian địa chỉ để gán địa chỉ cho bộ nhớ chính và các thiết bị vào ra. Bộ nhớ và các thanh ghi của thiết bị vào ra được ánh xạ với các giá trị địa chỉ, do đó một địa chỉ nhớ có thể tham chiếu đến một phần của bộ nhớ vật lý (RAM) hoặc bộ nhớ và thanh ghi của thiết bị vào ra. Vì vậy, các lệnh của CPU được sử dụng để truy nhập bộ nhớ cũng có thể được sử dụng để truy nhập các thiết bị ngoại vi.

Assignments at Home and at Lab

Home Assignment 1 – LED PORT

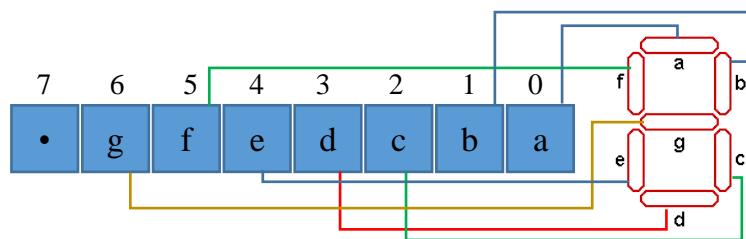
Viết chương trình hợp ngữ để hiển thị các chữ số từ 0 đến F trên đèn LED 7 đoạn.



Để mở thiết bị LED 7 đoạn, chọn **Digital Lab Sim** ở menu Tools.

Mở Help để biết cách hoạt động của đèn LED 7 đoạn.

Byte ở địa chỉ **0xFFFF0010**



```
.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, 0x06                  # set value for segments
    jal    SHOW_7SEG_LEFT            # show
    li      a0, 0x3F                  # set value for segments
    jal    SHOW_7SEG_RIGHT           # show
exit:
    li      a7, 10
    ecall
end_main:

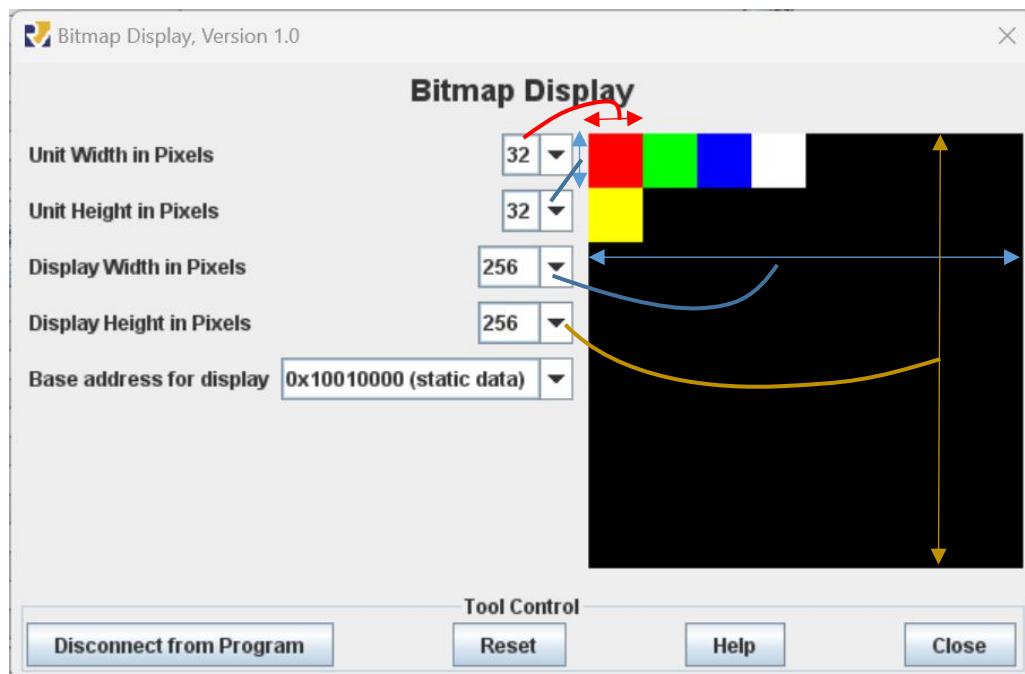
# -----
# Function SHOW_7SEG_LEFT : turn on/off the 7seg
# param[in] a0  value to shown
# remark   t0 changed
#
SHOW_7SEG_LEFT:
    li      t0, SEVENSEG_LEFT    # assign port's address
    sb    a0, 0(t0)                # assign new value
    jr    ra

# -----
# 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
    jr    ra
```

Home Assignment 2 – BITMAP DISPLAY

Bitmap Display là thiết bị giả lập màn hình máy tính. Bằng cách thiết lập màu cho từng điểm ảnh, các hình ảnh có thể được hiển thị trên màn hình. Mỗi điểm ảnh bao gồm 3 thành phần màu RGB được gắn với 1 từ nhớ (4 bytes) trong bộ nhớ. Để thiết lập màu cho một điểm ảnh, giá trị tương ứng với các thành phần màu sẽ được ghi vào từ nhớ tương ứng trong bộ nhớ.

Trong phần mềm giả lập RARS, nhấn vào Bitmap Display trong menu Tools để mở công cụ màn hình giả lập.



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

Mỗi đơn vị (unit) trên màn hình tương ứng với một từ nhớ (4 bytes) trong bộ nhớ. Từ nhớ chứa thông tin về thành phần màu RGB (byte MSB không sử dụng) của đơn vị. Các đơn vị được lưu trữ liên tiếp nhau với địa chỉ cơ sở là 0x10010000.

```
.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  a0, MONITOR_SCREEN      # Nap dia chi bat dau cua man hinh

    li  t0, RED
    sw  t0, 0(a0)

    li  t0, GREEN
    sw  t0, 4(a0)

    li  t0, BLUE
    sw  t0, 8(a0)

    li  t0, WHITE
    sw  t0, 12(a0)

    li  t0, YELLOW
```

```

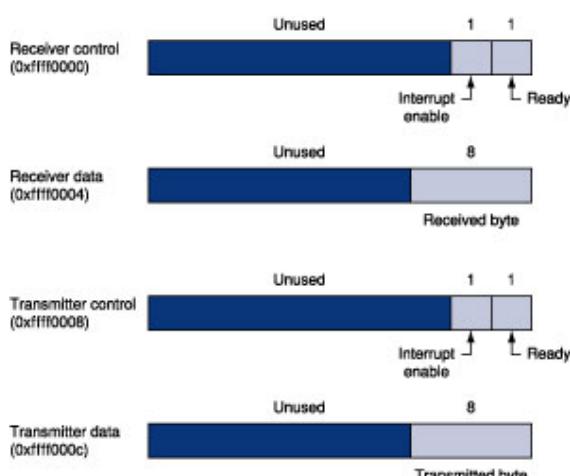
    sw  t0, 32(a0)
    li  t0, WHITE
    lb  t0, 42(a0)

```

Home Assignment 3 – KEYBOARD and DISPLAY MMIO

Công cụ này được sử dụng để giả lập ánh xạ bộ nhớ (MMIO – Memory-Mapped I/O) cho thiết bị nhập từ bàn phím và thiết bị xuất ra màn hình.

Khi kết nối với chương trình, mỗi phím được nhấn trong cửa sổ KEYBOARD sẽ ghi mã ASCII tương ứng vào từ nhớ Receiver Data (có địa chỉ 0xFFFF0004), và bit Ready được thiết lập trong từ nhớ Receiver Control (có địa chỉ 0xFFFF0000). Bit ready tự động được xóa về 0 khi chương trình đọc dữ liệu từ từ nhớ Receiver Data sử dụng lệnh **lw**.



```

.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

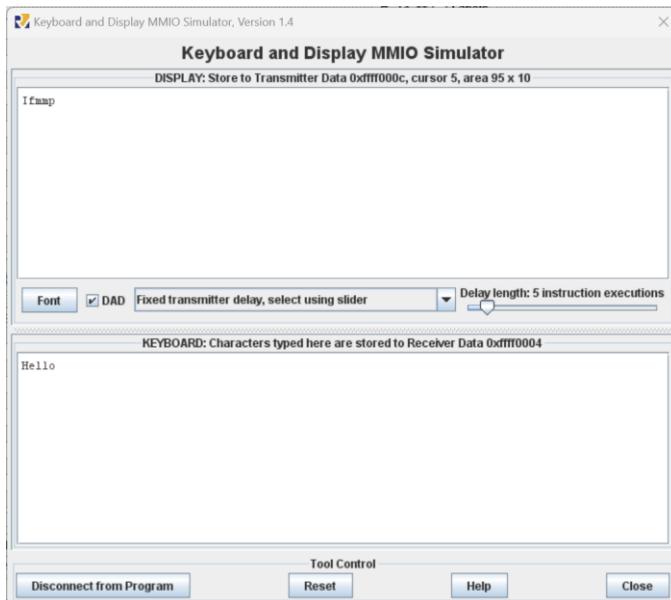
.text
    li  a0, KEY_CODE
    li  a1, KEY_READY
    li  s0, DISPLAY_CODE
    li  s1, DISPLAY_READY

loop:
WaitForKey:
    lw   t1, 0(a1)                # t1 = [a1] = KEY_READY
    beq t1, zero, WaitForKey    # if t1 == 0 then Polling
ReadKey:
    lw   t0, 0(a0)                # t0 = [a0] = 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($0)       # show key
    j       loop
```

Kết quả khi thực hiện:



Assignment 1

Tạo project để thực hiện Home Assignment 1. Thay đổi các giá trị hiển thị trên LED 7 đoạn (ví dụ: 2 chữ số cuối MSSV, 2 chữ số cuối mã ASCII của ký tự nhập từ bàn phím, ...)

Assignment 2

Tạo project để thực hiện Home Assignment 2. Thêm mã nguồn để vẽ các hình trên màn hình (vẽ bàn cờ vua, vẽ hình chữ nhật, ...)

Assignment 3

Tạo project để thực hiện Home Assignment 3. Thêm mã nguồn để hoàn thành yêu cầu sau: Nhập ký tự thường => hiển thị ký tự hoa tương ứng, nhập ký tự hoa => hiển thị ký tự thường tương ứng, nhập ký tự số thì giữ nguyên, nhập ký tự khác => hiển thị ký tự *. Khi nhập chuỗi ký tự "exit" thì kết thúc chương trình.

Bài 11. Lập trình xử lý ngắn

Mục đích

Sau bài thực hành này sinh viên có thể hiểu được nguyên lý cơ bản của ngắn và các sử dụng ngắn trong lập trình hợp ngữ. Sinh viên cũng có thể phân biệt được sự khác nhau giữa cách sử dụng kỹ thuật thăm dò và ngắn khi giao tiếp với các thiết bị ngoại vi.

Tài liệu

Kỹ thuật thăm dò và xử lý ngắn

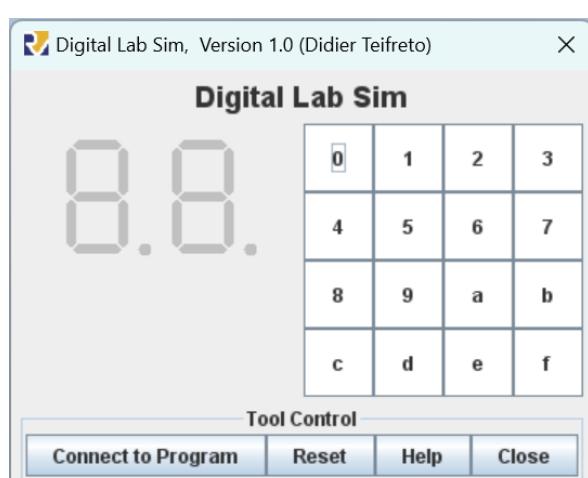
Một máy tính có thể phản hồi các sự kiện từ thiết bị ngoại vi bằng kỹ thuật thăm dò hoặc xử lý ngắn. Trong khi kỹ thuật thăm dò tương đối đơn giản, thì kỹ thuật xử lý ngắn mang tính hệ thống và hoạt động hiệu quả hơn. Sinh viên sẽ được tìm hiểu điểm giống và khác nhau giữa các kỹ thuật này.

Mỗi thiết bị ngoại vi kết nối với CPU thông qua các cổng. CPU sử dụng địa chỉ gắn liền với các cổng này, khi đó CPU có thể đọc/ghi giá trị đến các cổng này để kiểm soát hoạt động của thiết bị ngoại vi.

Chuẩn bị

Assignments at Home and at Lab

Home Assignment 1 – Kỹ thuật thăm dò (Pooling)



Viết chương trình hợp ngữ xác định phím được nhấn của thiết bị ma trận phím trong công cụ Digital Lab Sim và in ra mã phím ra màn hình.

Chương trình sử dụng vòng lặp vô hạn, liên tục quét mã phím và in ra màn hình. Đây được gọi là kỹ thuật thăm dò.

Các bước cần thực hiện để xác định mã phím được nhấn trên thiết bị ma trận phím (kỹ thuật quét ma trận phím):

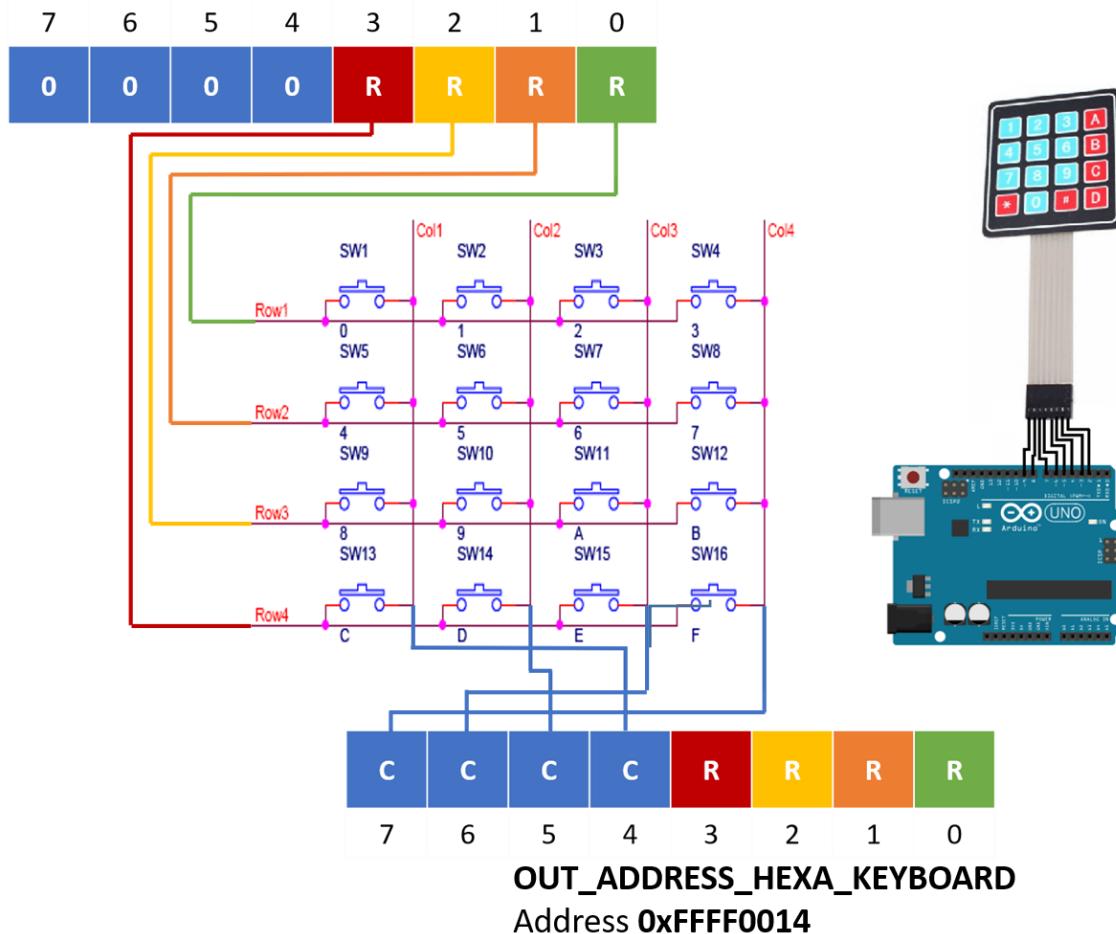
1. Gán dòng cân kiểm tra vào byte tại địa chỉ 0xFFFF0012.

2. Đọc byte tại địa chỉ 0xFFFF0014 để biết phím nào được nhấn trên dòng đã chọn.

Chú ý: chạy chương trình ở tốc độ 30 lệnh/s để tránh hiện tượng phần mềm RARS bị treo.

IN_ADDRESS_HEXA_KEYBOARD

Address 0xFFFF0012



OUT_ADDRESS_HEXA_KEYBOARD

Address 0xFFFF0014

```
# -----
#          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_ADDRESS_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_ADDRESS_HEXA_KEYBOARD    0xFFFF0014

.text
main:
    li  t1, IN_ADDRESS_HEXA_KEYBOARD
    li  t2, OUT_ADDRESS_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  a7, 34            # print integer (hexa)
    ecall
sleep:
    li  a0, 100           # sleep 100ms
    li  a7, 32
    ecall
back_to_polling:
    j   polling           # continue polling
```

Home Assignment 2 – Kỹ thuật xử lý ngắt

Giới thiệu về ngắt và xử lý ngắt

Ngắt là cơ chế cho phép các thiết bị ngoại vi gửi thông báo đến CPU sự kiện các thiết bị này cần chú ý. Khi ngắt xảy ra, thiết bị ngoại vi gửi tín hiệu đến CPU, khi CPU nhận được tín hiệu này sẽ thực hiện các công việc theo trình tự sau:

1. Sao lưu ngữ cảnh của chương trình hiện tại.
2. Thực hiện chương trình con phục vụ ngắt.
3. Khôi phục ngữ cảnh và tiếp tục thực hiện chương trình chính.

Ngắt có thể xảy ra do nhiều nguồn: ngắt ngoài do thiết bị ngoại vi, ngắt do bộ định thời, ngắt do lệnh (ngắt mềm) hoặc ngắt do chương trình xảy ra lỗi (exception) mà không được kiểm soát.

Kiến trúc tập lệnh RISC-V định nghĩa 3 mức độ quyền truy nhập bao gồm **User/Application**, **Supervisor**, và **Machine**. Mức độ quyền truy nhập định nghĩa những tài nguyên nào (thanh ghi, lệnh, ...) được truy nhập bởi phần mềm. Cơ chế này sẽ hạn chế việc thực thi của các phần mềm và bảo vệ hệ thống bởi các phần mềm có tình thực hiện các thao tác không được phép. Quyền **Machine** có mức độ truy nhập cao nhất, quyền **User/Application** có mức độ truy nhập thấp nhất. **Chương trình giả lập RARS mô phỏng mức độ User/Application.**

Các thanh ghi được sử dụng để xử lý ngắt

Kiến trúc RISC-V định nghĩa các thanh ghi điều khiển và trạng thái (**Control and Status Registers, CSRs**) cho biết trạng thái của CPU và cho phép phần mềm điều khiển hành vi của CPU. Kiến trúc tập lệnh RISC-V cũng bao gồm một tập hợp các lệnh cho phép phần mềm đọc và ghi nội dung các thanh ghi CSRs.

Các thanh ghi CSRs liên quan đến xử lý ngắt:

- **mstatus**: Thanh ghi trạng thái, các trường trong thanh ghi này cho biết thông tin hoặc điều khiển cơ chế xử lý ngắt. Trường **UIE** (bit 0) cho phép hoặc không cho phép ngắt (ở chế độ User).
- **mcause** (Machine Interrupt Cause): Thanh ghi nguyên nhân ngắt, bao gồm 2 trường: **INTERRUPT** (bit 31) chỉ ra nguyên nhân có phải do ngắt hay do ngoại lệ (exception), trường **EXCCODE** (bit 0 đến 30) chỉ ra nguyên nhân ngắt (hoặc ngoại lệ).
- **mtvec** (Machine Trap Vector): Thanh ghi chứa thông tin về chương trình con mà CPU sẽ thực hiện khi xảy ra ngắt.
- **mie** (Machine Interrupt Enable): Thanh ghi thiết lập cho phép hoặc không cho phép các nguồn ngắt cụ thể.
- **mip** (Machine Interrupt Pending): Thanh ghi chứa thông tin về các ngắt đang chưa được xử lý bởi CPU.
- **mepc** (Machine Exception Program Counter): Thanh ghi chứa giá trị của thanh ghi PC khi ngắt xảy ra.

31	30	WPRI								23	22	21	20	19	18	17
SD		1	8						TSR	TW	TVM	MXR	SUM	MPRV		
1									1	1	1	1	1	1	1	
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X[1:0]	FS[1:0]	MPP[1:0]	WPRI	SPP	MPIE	WPRI	SPIE	UPIE	MIE	WPRI	SIE	UIE				
2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.6: Machine-mode status register (**mstatus**) for RV32.

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0			
WIRI	MEIP	WIRI	SEIP	UEIP	MTIP	WIRI	STIP	UTIP	MSIP	WIRI	SSIP	USIP				
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.11: Machine interrupt-pending register (**mip**).

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0			
WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE				
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.12: Machine interrupt-enable register (**mie**).

Luồng xử lý ngắt (Đối với chương trình giả lập RARS, mức độ truy nhập User)

1. Khai báo chương trình con xử lý ngắt (Interrupt Service Routine - ISR), nội dung của ISR thường bao gồm:
 - a. Sao lưu các thanh ghi sử dụng trong chương trình con.
 - b. Phân loại ngắt, tùy thuộc vào kiểu ngắt thực hiện việc xử lý tương ứng.
 - c. Khôi phục các thanh ghi đã sao lưu.
 - d. Quay trở lại chương trình chính
2. Nạp địa chỉ chương trình con xử lý ngắt vào thanh ghi **mtvec**.
3. Tùy vào chương trình, thiết lập nguồn ngắt trong thanh ghi **mie**.

4. Cho phép ngắt toàn cục, thiết lập bit **UIE** của thanh ghi **mstatus**.
5. Thiết lập công cụ giả lập để mô phỏng ngắt (Keypad, Timer Tool, ...)

Chú ý:

- *RARS đã đổi tên các thanh ghi CSRs thành **ustatus**, **ucause**, **utvec**, **ui**, **uip**, **uepc** để nhấn mạnh việc mô phỏng ở chế độ truy nhập User.*
- *Với các công cụ giả lập, nên bấm nút “Connect to Program” trước khi chạy giả lập. Nếu không, việc phát sinh sự kiện ngắt sẽ không xảy ra.*
- *Việc sử dụng breakpoint để dừng chương trình khi ngắt xảy ra sẽ không hiệu quả, có thể sử dụng lệnh **ebreak** để dừng chương trình.*

Ví dụ dưới đây minh họa việc thiết lập và xử lý ngắt do công cụ Keypad tạo ra. Đọc kỹ và hiểu cách chương trình hoạt động.

```
.eqv IN_ADDRESS_HEXA_KEYBOARD      0xFFFF0012
.data
    message: .asciz "Someone's presed a button.\n"
#
# MAIN Procedure
#
.text
main:
    # Load the interrupt service routine address to the UTVEC register
    la      t0, handler
    csrrs  zero, utvec, t0

    # Set the UEIE (User External Interrupt Enable) bit in UIE register
    li      t1, 0x100
    csrrs  zero, uie, t1      # uie - ueie bit (bit 8)
    # Set the UIE (User Interrupt Enable) bit in USTATUS register
    csrrsi zero, ustatus, 0x1  # ustatus - enable uie (bit 0)

    # Enable the interrupt of keypad of Digital Lab Sim
    li      t1, IN_ADDRESS_HEXA_KEYBOARD
    li      t3, 0x80  # bit 7 = 1 to enable interrupt
    sb      t3, 0(t1)

    #
    # No-end loop, main program, to demo the effective of interrupt
    #

loop:
    nop
    # Delay 10ms
    li      a7, 32
    li      a0, 10
    ecall
    nop
    j       loop
```

```
end_main:

# -----
# Interrupt service routine
# -----
handler:
    # ebreak # Can pause the execution to observe registers
    # Saves the context
    addi    sp, sp, -8
    sw     a0, 0(sp)
    sw     a7, 4(sp)

    # Handles the interrupt
    # Shows message in Run I/O
    li     a7, 4
    la     a0, message
    ecall

    # Restores the context
    lw     a7, 4(sp)
    lw     a0, 0(sp)
    addi   sp, sp, 8

    # Back to the main procedure
    uret
```

Home Assignment 3 – INTERRUPT & STACK

Ngăn xếp được sử dụng để sao lưu và phục hồi các thanh ghi được sử dụng trong chương trình con phục vụ ngắn, tránh ảnh hưởng đến hoạt động của chương trình chính.

Chương trình dưới đây thực hiện các chức năng sau:

1. Chương trình chính thiết lập ngắn từ thiết bị keypad của công cụ Digital Lab Sim.
2. Chương trình chính in ra chuỗi các số nguyên liên tiếp ở màn hình Run I/O.
3. Mỗi khi người dùng nhấn một trong các phím C, D, E hoặc F, ngắn được tạo ra, chương trình con phục vụ ngắn in ra mã phím ở màn hình Run I/O.

Đọc kỹ và hiểu cách chương trình hoạt động.

```
.eqv IN_ADDRESS_HEXA_KEYBOARD      0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD     0xFFFF0014
.data
    message: .asciz "Key scan code: "
# -----
# MAIN Procedure
# -----
.text
main:
    # Load the interrupt service routine address to the UTVEC register
    la     t0, handler
    csrrs zero, utvec, t0
```

```
# Set the UEIE (User External Interrupt Enable) bit in UIE register
li      t1, 0x100
csrrs  zero, uie, t1      # uie - ueie bit (bit 8)
# Set the UIE (User Interrupt Enable) bit in USTATUS register
csrrsi zero, ustatus, 0x1 # ustatus - enable uie (bit 0)

# Enable the interrupt of keypad of Digital Lab Sim
li      t1, IN_ADDRESS_HEXA_KEYBOARD
li      t3, 0x80 # bit 7 = 1 to enable interrupt
sb      t3, 0(t1)

# -----
# Loop to print a sequence numbers
# -----
xor    s0, s0, s0      # count = s0 = 0
loop:
addi   s0, s0, 1       # count = count + 1
prn_seq:
addi   a7, zero, 1
add    a0, s0, zero    # Print auto sequence number
ecall
addi   a7, zero, 11
li     a0, '\n'        # Print EOL
ecall
sleep:
addi   a7, zero, 32
li     a0, 300         # Sleep 300 ms
ecall
j      loop
end_main:

# -----
# Interrupt service routine
# -----
handler:
# Saves the context
addi   sp, sp, -16
sw    a0, 0(sp)
sw    a7, 4(sp)
sw    t1, 8(sp)
sw    t2, 12(sp)

# Handles the interrupt
prn_msg:
addi   a7, zero, 4
la    a0, message
ecall
get_key_code:
```

```
    li      t1,    IN_ADDRESS_HEXA_KEYBOARD
    li      t2,    0x88      # Check row 4 and re-enable bit 7
    sb      t2,    0(t1)     # Must reassign expected row
    li      t1,    OUT_ADDRESS_HEXA_KEYBOARD
    lb      a0,    0(t1)

prn_key_code:
    li      a7,  34
    ecall
    li      a7,  11
    li      a0,  '\n'        # Print EOL
    ecall

    # Restores the context
    lw      t2,  12(sp)
    lw      t1,  8(sp)
    lw      a7,  4(sp)
    lw      a0,  0(sp)
    addi   sp,  sp,  16

    # Back to the main procedure
    uret
```

Home Assignment 4 – Xử lý nhiều nguồn ngắt

Trong trường hợp nhiều ngắt được kích hoạt, khi ngắt xảy ra CPU thực hiện chung một chương trình con xử lý ngắt. Do đó bên trong chương trình con, cần phân biệt nguồn ngắt để thực hiện việc xử lý tương ứng.

Thanh ghi CSR mcause cho biết thông tin về nguồn ngắt. Thanh ghi này gồm 2 trường:

- **INTERRUPT** (bit 31): mang giá trị 1 nếu nguyên nhân do ngắt, giá trị 0 nếu do ngoại lệ.
- **EXCCODE** (bit 0 đến 30): chỉ ra nguyên nhân ngắt (nguồn gây ngắt), được mô tả trong bảng sau.

mcause fields		Cause
INTERRUPT	EXCCODE	
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	11	Machine external interrupt
1	12-15	Reserved for future standard use
1	≥ 16	Reserved for platform use

Chương trình sau thực hiện các chức năng:

1. Chương trình chính kích hoạt đồng thời 2 ngắt từ thiết bị keypad (Digital Lab Sim) và từ bộ định thời (Timer Tool).
2. Chương trình chính chạy vòng lặp vô hạn.
3. Khi chạy giả lập, sau mỗi khoảng thời gian hoặc người dùng nhấn vào nút trên keypad, chương trình sẽ in ra thông báo tương ứng ở màn hình Run I/O.

Ghi chú: Cách sử dụng bộ định thời.

- Từ nhút ở địa chỉ 0xFFFF0018 trả về giá trị hiện tại của bộ định thời (đơn vị ms).
- Từ nhút ở địa chỉ 0xFFFF0020 chứa giá trị so sánh (đơn vị ms). Ngắt xảy ra khi giá trị hiện tại của bộ định thời vượt quá giá trị so sánh.
- Trong chương trình con xử lý ngắt của bộ định thời, cần cập nhật giá trị so sánh nếu muốn ngắt tiếp theo xảy ra.
- Công cụ Timer Tool (ảnh minh họa phía dưới) được sử dụng để điều khiển bộ định thời.



Đọc kỹ chương trình và hiểu cách chương trình hoạt động.

.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
--

```

.eqv TIMER_NOW          0xFFFF0018
.eqv TIMER_CMP          0xFFFF0020
.eqv MASK_CAUSE_TIMER   4
.eqv MASK_CAUSE_KEYPAD  8

.data
    msg_keypad: .asciz "Someone has pressed a key!\n"
    msg_timer: .asciz "Time inteval!\n"

# -----
# MAIN Procedure
# -----
.text
main:
    la      t0, handler
    csrrs  zero, 5, t0

    li      t1, 0x100
    csrrs  zero, 4, t1      # uie - ueie bit (bit 8) - external interrupt
    csrrsi zero, 4, 0x10    # uie - utie bit (bit 4) - timer interrupt

    csrrsi zero, 0, 0x1     # ustatus - enable uie - global interrupt

# -----
# Enable interrupts you expect
# -----
# Enable the interrupt of keypad of Digital Lab Sim
    li      t1, IN_ADDRESS_HEXA_KEYBOARD
    li      t2, 0x80 # bit 7 of = 1 to enable interrupt
    sb      t2, 0(t1)

# Enable the timer interrupt
    li      t1, TIMER_CMP
    li      t2, 1000
    sw      t2, 0(t1)

# -----
# No-end loop, main program, to demo the effective of interrupt
# -----
loop:
    nop
    li  a7, 32
    li  a0, 10
    ecall
    nop
    j   loop
end_main:

# -----

```

```
# Interrupt service routine
# -----
handler:
    # Saves the context
    addi    sp, sp, -16
    sw     a0, 0(sp)
    sw     a1, 4(sp)
    sw     a2, 8(sp)
    sw     a7, 12(sp)

    # Handles the interrupt
    csrr    a1, ucause
    li      a2, 0xFFFFFFFF
    and    a1, a1, a2      # Clear interrupt bit to get the value

    li      a2, MASK_CAUSE_TIMER
    beq    a1, a2, timer_isr
    li      a2, MASK_CAUSE_KEYPAD
    beq    a1, a2, keypad_isr
    j      end_process

timer_isr:
    li      a7, 4
    la      a0, msg_timer
    ecall

    # Set cmp to time + 1000
    li      a0, TIMER_NOW
    lw      a1, 0(a0)
    addi   a1, a1, 1000
    li      a0, TIMER_CMP
    sw      a1, 0(a0)

    j      end_process

keypad_isr:
    li      a7, 4
    la      a0, msg_keypad
    ecall
    j      end_process

end_process:

    # Restores the context
    lw      a7, 12(sp)
    lw      a2, 8(sp)
    lw      a1, 4(sp)
    lw      a0, 0(sp)
    addi   sp, sp, 16
```

uret

Home Assignment 5 – Xử lý ngoại lệ

Ngoại lệ (Exception) là các sự kiện được tạo ra bởi CPU để phản hồi các điều kiện ngoại lệ khi thực hiện các câu lệnh. Các ngoại lệ thường kích hoạt cơ chế xử lý để đảm bảo các điều kiện ngoại lệ được kiểm soát trước khi CPU tiếp tục thực hiện chương trình. Cơ chế xử lý ngoại lệ được sử dụng để bảo vệ hệ thống trước việc thực hiện các câu lệnh không hợp lệ. Việc xử lý ngoại lệ tương tự với xử lý ngắt, bao gồm các bước:

1. Lưu trữ ngữ cảnh của chương trình.
 2. Xử lý điều kiện ngoại lệ.
 3. Khôi phục ngữ cảnh và tiếp tục thực hiện chương trình (Tùy thuộc vào kiểu ngoại lệ, hệ thống sẽ quyết định việc tiếp tục thực hiện chương trình hay không).
- Ví dụ sau minh họa việc thực hiện cấu trúc **try-catch-finally** trong các ngôn ngữ lập trình bậc cao để xử lý các ngoại lệ.

```
.data
    message: .asciz "Exception occurred.\n"
.text
main:
try:
    la      t0, catch
    csrrw  zero, 5, t0 # Set utvec (5) to the handlers address
    csrrsi zero, 0, 1 # Set interrupt enable bit in ustatus (0)

    lw zero, 0          # Trigger trap for Load access fault
finally:
    li a7, 10          # Exit the program
    ecall

catch:
    # Show message
    li a7, 4
    la a0, message
    ecall
    # Load finally address to uepc
    la t0, finally
    csrrw zero, 65, t0
    uret
```

Assignment 1

Tạo project thực hiện Home Assignment 1. Cập nhật mã nguồn để chương trình có thể in ra mã của tất cả 16 nút bấm trên keypad.

Assignment 2

Tạo project để thực hiện và thử nghiệm Home Assignment 2. Chạy ở chế độ từng dòng lệnh, quan sát giá trị của các thanh ghi để hiểu cách chương trình hoạt động.

Assignment 3

Tạo project để thực hiện và thử nghiệm Home Assignment 3. Chạy ở chế độ từng dòng lệnh, quan sát giá trị của các thanh ghi để hiểu cách chương trình hoạt động. Cập nhật mã nguồn để chương trình có thể in ra mã của tất cả 16 nút bấm trên keypad.

Assignment 4

Tạo project để thực hiện và thử nghiệm Home Assignment 4. Chạy ở chế độ từng dòng lệnh, quan sát giá trị của các thanh ghi để hiểu cách chương trình hoạt động.

Assignment 5

Tạo project để thực hiện và thử nghiệm Home Assignment 5. Chạy ở chế độ từng dòng lệnh, quan sát giá trị của các thanh ghi để hiểu cách chương trình hoạt động.

Assignment 6: Ngắt mềm

Ngắt mềm có thể được kích hoạt bằng cách thiết lập bit **USIP** của thanh ghi **mip**. Viết chương trình kích hoạt ngắt mềm nếu xảy ra tràn số khi thực hiện việc cộng 2 số nguyên có dấu (xem lại Bài thực hành 4), chương trình con xử lý ngắt sẽ in ra thông báo lỗi tràn số và kết thúc chương trình.

Kết luận

Trước khi kết thúc bài thực hành, sinh viên trả lời các câu hỏi sau:

- Kỹ thuật thăm dò là gì?
- Ngắt là gì?
- Chương trình con xử lý ngắt là gì?
- Ưu điểm của kỹ thuật thăm dò?
- Ưu điểm của kỹ thuật xử lý ngắt?
- Nhược điểm khác nhau giữa ngắt, ngoại lệ và traps?

Bài 12. Bộ nhớ đệm nhanh – Cache memory

Mục đích

Sau bài thực hành này, sinh viên có thể hiểu được nguyên lý cơ bản của bộ nhớ đệm nhanh và sự ảnh hưởng của các tham số đến hiệu quả hoạt động của bộ nhớ đệm.

Tài liệu

Patterson and Hennessy: Chapter 7.1–7.3

Chuẩn bị

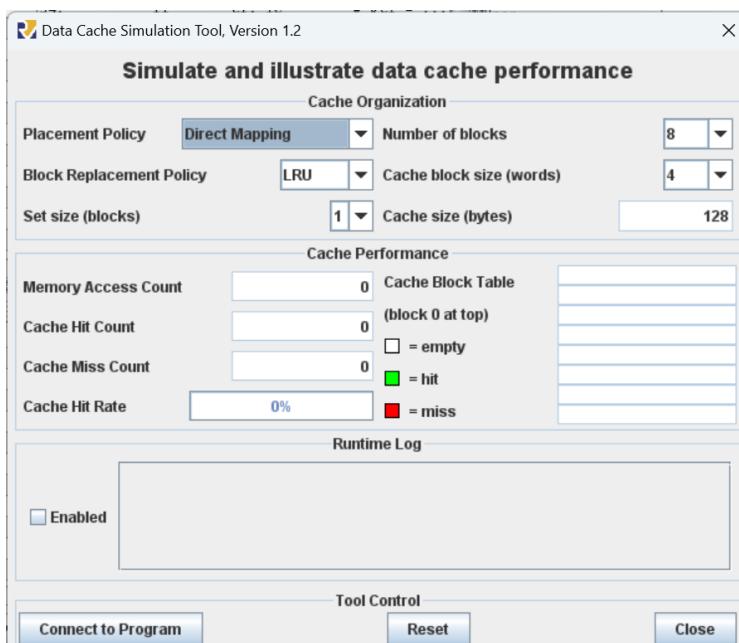
Assignments at Home and at Lab

Assignment 1 – Sử dụng công cụ Data Cache Simulator

1. Mở chương trình **row-major.asm** trong thư mục Lab12. Chương trình này duyệt ma trận kích thước 16x16 theo thứ tự hàng chính (duyệt từng hàng), gán các giá trị từ 0 đến 255. Chương trình thực hiện thuật toán sau:

```
for (row = 0; row < 16; row++)
    for (col = 0; col < 16; col++)
        data[row][col] = value++;
```

2. Biên dịch chương trình.
3. Từ menu Tools, mở công cụ Data Cache Simulator.



Đây là công cụ giả lập việc sử dụng và hiệu năng của bộ nhớ đệm nhanh khi chương trình được thực hiện. Bao gồm các vùng chức năng chính:

- *Cache Organization:* gồm các thiết lập mô tả cách bộ nhớ đệm được cấu hình.
- *Cache Performance:* với mỗi lần truy nhập bộ nhớ khi chương trình hoạt động, chương trình giả lập sẽ xác định việc truy nhập dữ liệu có thỏa mãn từ bộ nhớ đệm hay không và cập nhật việc hiển thị hiệu năng tương ứng.

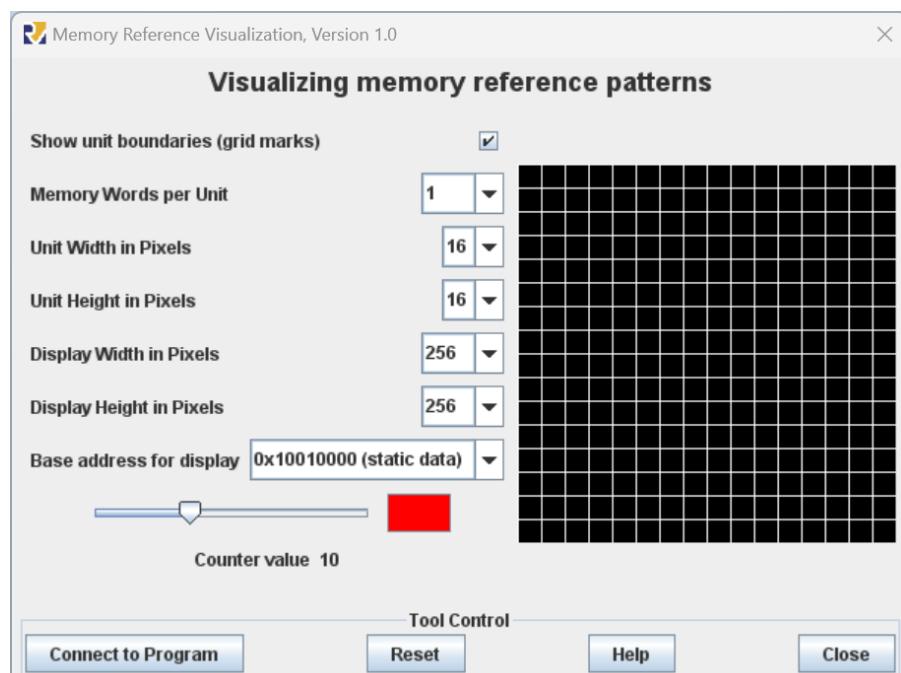
4. Nhấn vào nút **Connect to Program**.
5. Quay trở lại RARS, điều chỉnh tốc độ thực hiện là 30 lệnh/s. Thiết lập này làm chậm lại quá trình thực thi chương trình, giúp sinh viên quan sát được hiệu ứng minh họa trong vùng Cache Performance.
6. Nhấn nút chạy chương trình, theo dõi Cache Performance được cập nhật mỗi lần truy nhập bộ nhớ.
7. *Tỷ lệ cache hit sau khi chạy chương trình?* Với mỗi lần cache miss, một khối gồm 4 từ nhớ được ghi vào bộ nhớ đệm. Với cách duyệt theo dòng, các phần tử của ma trận được truy nhập theo thứ tự lưu trữ trong bộ nhớ. Vì vậy sau mỗi lần cache miss là 3 lần cache hit khi 3 phần tử tiếp theo được lưu trữ trong cùng một khối dữ liệu lưu vào bộ đệm. Lượt truy nhập tiếp theo sẽ là cache miss khi chế độ ánh xạ trực tiếp (Direct Mapping) ánh xạ tới khối tiếp theo. Do đó, 3 trong 4 lần truy nhập bộ nhớ sẽ được tìm thấy ở bộ nhớ cache.
8. Với cách giải thích như trên, dự đoán tỷ lệ cache hit nếu kích thước khối block tăng từ 4 thành 8 từ nhớ? Hoặc giảm từ 4 thành 2 từ nhớ?
9. Kiểm nghiệm kết quả bằng cách thay đổi kích thước khối block và chạy lại chương trình từ bước 6.
Chú ý: Khi thay đổi các cài đặt trong Cache Organization, giá trị hiệu năng tính toán trước đó sẽ được khởi tạo lại.
10. Thực hiện lại các bước trên với chương trình **column-major.asm** trong thư mục Lab12. Chương trình này sẽ duyệt các phần tử trong ma trận kích thước 16x16 theo thứ tự cột chính (duyệt từng cột) và lần lượt gán các giá trị từ 0 đến 255. Chương trình thực hiện thuật toán sau:

```
for (col = 0; col < 16; col++)
    for (row = 0; row < 16; row++)
        data[row][col] = value++;
```
11. *Hiệu năng truy nhập bộ nhớ cache đổi với chương trình này là bao nhiêu?* Vẫn đề được xác định ở đây là các từ nhớ không được truy nhập tuần tự như trong chương trình trước. Mỗi lần truy nhập cách nhau 16 từ nhớ. Với những thiết lập hiện tại trong công cụ giả lập, không có 2 lần truy nhập liên tiếp nào cùng chung một khối block, do đó lần truy nhập nào cũng là cache miss.
12. Thay đổi kích thước block thành 16. Khởi tạo lại công cụ giả lập.
13. Mở thêm 1 cửa sổ công cụ Cache Simulator, đặt bên cạnh cửa sổ hiện tại đang mở. Kết nối với chương trình và thay đổi kích thước block thành 16 và số block là 16.
14. Chạy lại chương trình. *Hiệu năng truy nhập được tính trên cửa sổ cũ là bao nhiêu?* Tăng kích thước block thành 16 từ nhớ không làm tăng hiệu năng bởi vì chỉ có một lần truy nhập với mỗi block, trước khi block được thay thế bởi block khác. *Hiệu năng truy nhập được tính trên cửa sổ mới là bao nhiêu?*

nhiêu? Trong trường hợp này, toàn bộ ma trận sẽ vừa với bộ nhớ cache và một khi block được đọc, nó sẽ không bị thay thế bởi block khác. Chỉ có lần truy nhập block đầu tiên trả về kết quả là cache miss.

Assignment 2 – Sử dụng công cụ Memory Reference Visualization

1. Mở chương trình **row-major.asm** trong thư mục **Lab12**.
2. Dịch chương trình.
3. Từ menu **Tools**, mở công cụ **Memory Reference Visualization**.



Công cụ này sẽ tô màu vào mỗi ô trong lưới ở bên phải mỗi khi một từ nhớ được tham chiếu. Địa chỉ cơ sở (ứng với chỉ thị biên dịch .data) tương ứng với ô ở góc trên bên trái. Địa chỉ được gán theo thứ tự hàng (từ trái qua phải, từ trên xuống dưới).

Màu các ô phụ thuộc vào số lần từ nhớ tương ứng được tham chiếu. Màu đen là 0, màu xanh lam là 1, màu xanh lá là 2, màu vàng là 4, màu cam từ 5 đến 9, màu đỏ là 10 và cao hơn.

4. Nhấn vào nút **Connect to Program**.
5. Điều chỉnh tốc độ thực hiện lệnh trong RARS là 30 lệnh/s.
6. Chạy chương trình. Quan sát công cụ minh họa việc truy nhập bộ nhớ.
7. Lặp lại các bước 1 đến 6 với chương trình **column-major.asm**.
8. Lặp lại các bước 1 đến 6 với chương trình **fibonacci.asm**.

Kết luận

Bài 13. Lập trình hợp ngữ với ESP32-C3 – Mô phỏng bằng Wokwi

Mục đích

Trong bài thực hành này, sinh viên được làm quen với kit ESP32-C3 dựa trên kiến trúc RISC-V. Sinh viên có thể sử dụng hợp ngữ để lập trình các ứng dụng đơn giản điều khiển các cổng vào ra của kit và chạy mô phỏng bằng công cụ giả lập Wokwi.

Tài liệu

- ESP32-C3 Technical Reference Manual

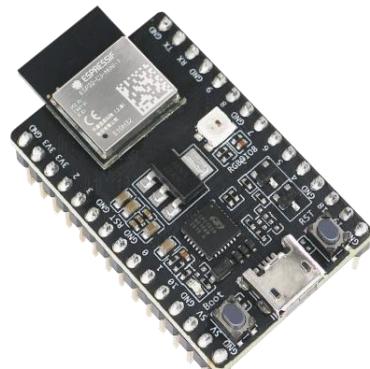
Chuẩn bị

Giới thiệu về MCU ESP32-C3

ESP32-C3 là một hệ thống trên chip (System on Chip – SoC) của hãng Espressif, tích hợp vi điều khiển dựa trên kiến trúc mã nguồn mở RISC-V. Nó đạt được sự cân bằng hợp lý về sức mạnh, khả năng kết nối ngoại vi và bảo mật, từ đó cung cấp giải pháp tiết kiệm chi phí tối ưu cho các thiết bị được kết nối. Vi điều khiển RISC-V 32-bit có thể hoạt động với tốc độ xung nhịp tối đa 160 MHz. Với 22 chân GPIO (General Purpose Input/Output) có thể cấu hình, kích thước bộ nhớ RAM 400 KB và hỗ trợ chế độ năng lượng thấp, nó có thể được sử dụng trong nhiều trường hợp khác nhau liên quan đến các thiết bị được kết nối.

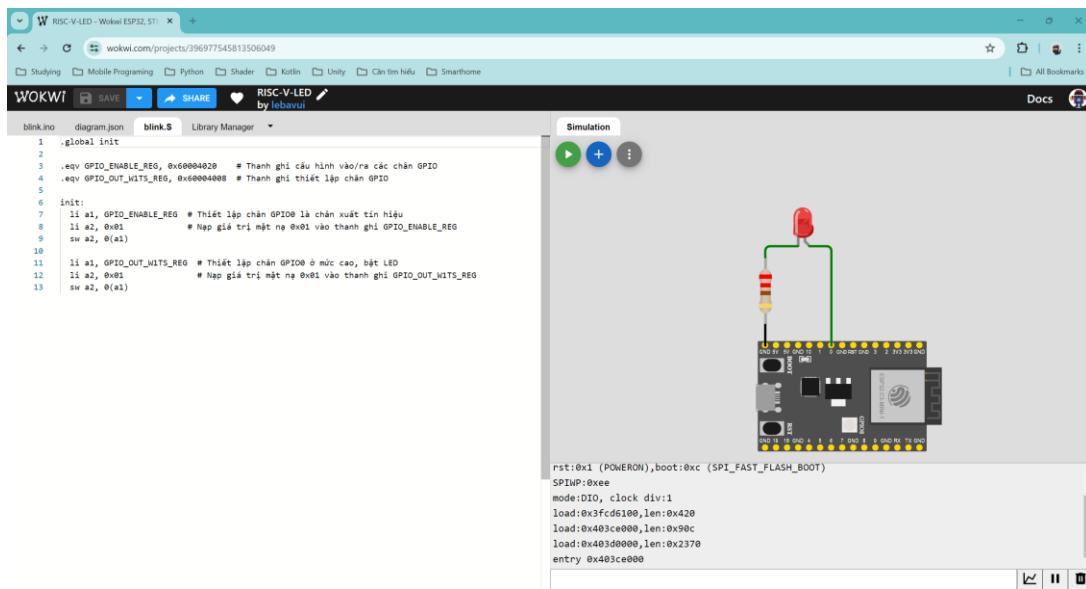
Các hãng điện tử thường chế tạo các mạch phát triển (còn gọi là kit). Bên cạnh module vi điều khiển chính, các kit này còn tích hợp các mạch điện tử giúp cho việc sử dụng được thuận tiện, ví dụ như mạch nạp chương trình qua cổng USB, mạch điều khiển điện áp, các LED trạng thái, ... Hình bên phải là kit ESP32-C3-DevKitM-1 của hãng Espressif chế tạo. Ngoài ra, còn có các phiên bản của các hãng điện tử khác.

Khi làm việc với các mạch phát triển này, sinh viên cần đọc datasheet để biết cách sử dụng, sơ đồ chân, chế độ hoạt động, ... của mạch.



Giả lập kit ESP32-C3 với Wokwi

Wokwi là một ứng dụng nền web cho phép giả lập các mạch phát triển hệ nhúng quen thuộc như Arduino, STM32 và ESP32. Trang web cung cấp giao diện cho phép lập trình viên viết mã nguồn và chạy thử trên các hệ nhúng giả lập. Nó cũng cung cấp các thiết bị điện tử cơ bản đóng vai trò như các thiết bị ngoại vi kết nối với hệ nhúng như LED, LED 7 đoạn, màn hình LCD, nút bấm, công tắc, điện trở, ... để mô phỏng thử nghiệm các ứng dụng hệ nhúng đơn giản.



Giao diện soạn thảo mã nguồn và mô phỏng của Wokwi

Lập trình hợp ngữ trên ESP32-C3 sử dụng Wokwi

Các bước tạo project, soạn thảo mã nguồn và mô phỏng sử dụng Wokwi như sau:

- Tạo project mới sử dụng template ESP32-C3.
 - Vào trang chủ Wokwi <https://wokwi.com>
 - Chọn ESP32
 - Chọn ESP32-C3
- Tại giao diện soạn thảo mã nguồn, đổi lại tên file ino nếu cần (nhấn vào mũi tên bên phải **Library Manager**, chọn **Rename**). Thay đổi nội dung file **ino** như sau:

```
void setup() {}
void loop() {}
```

- Thêm mới file mã nguồn hợp ngữ (phần mở rộng .S), nhấn vào mũi tên bên phải **Library Manager**, chọn **New file ...**, đặt tên file trùng với file **ino** nhưng với phần mở rộng là **.S** (ví dụ file ino là **sketch.ino** thì file hợp ngữ là **sketch.S**).
- Soạn thảo mã nguồn hợp ngữ, khung chương trình hợp ngữ như sau:

```
# Căn định nghĩa hàm init để Wokwi thực hiện chương trình hợp ngữ
.global init

# (Tùy chọn) Sử dụng chỉ thị .eqv để định nghĩa các hằng
.eqv CONST1, 0x0001

# (Tùy chọn) Sử dụng chỉ thị .data để khai báo dữ liệu trong bộ nhớ
.data

# Chỉ thị .text đánh dấu bắt đầu đoạn lệnh
# Nếu không có .data thì cũng không cần .text
.text
init:
    # Chương trình hợp ngữ bắt đầu từ đây!!!
```

5. Nhấn nút + để thêm các linh kiện điện tử cần thiết, nối chân các linh kiện với các chân của kit ESP32-C3 theo sơ đồ nguyên lý.
6. Nhấn nút Start để dịch và chạy mô phỏng.

Các thanh ghi điều khiển các chân vào ra GPIO trên ESP32-C3

ESP32-C3 bao gồm 22 chân vào ra đa năng (General Purpose Input/Output – GPIO). Các chân này có thể được cấu hình để xuất tín hiệu (Output) hoặc đọc tín hiệu (Input). Mỗi chân ngoài chức năng GPIO còn có thể thực hiện các chức năng khác (ví dụ như kết nối ngoại vi theo giao thức I2C). ESP32-C3 cung cấp các thanh ghi để cấu hình chức năng cho các chân GPIO. Việc cấu hình này thường được thực hiện ở bước khởi tạo.

- **GPIO_ENABLE_REG** (GPIO output enable register), địa chỉ **0x60004020**, thanh ghi cho phép xuất tín hiệu. Bit 0 đến bit 21 tương ứng với GPIO0 đến GPIO21. Giá trị bit bằng 1 cho phép chân GPIO tương ứng là chân xuất tín hiệu (output).
- **GPIO_ENABLE_W1TS_REG** (GPIO output enable set register), địa chỉ **0x60004024**, thanh ghi hỗ trợ việc thiết lập bit trên thanh ghi GPIO_ENABLE_REG. Giá trị bit bằng 1 sẽ thiết lập bit tương ứng trong thanh ghi GPIO_ENABLE_REG, các bit khác không thay đổi.
- **GPIO_ENABLE_W1TC_REG** (GPIO output enable clear register), địa chỉ **0x60004028**, thanh ghi hỗ trợ việc xóa bit trên thanh ghi GPIO_ENABLE_REG. Giá trị bit bằng 1 sẽ xóa bit tương ứng trong thanh ghi GPIO_ENABLE_REG, các bit khác không thay đổi.
- **GPIO_IN_REG** (GPIO input register), địa chỉ **0x6000403C**, thanh ghi được sử dụng để đọc trạng thái các chân GPIO là chân nhập tín hiệu (input). Bit 0 đến bit 21 tương ứng với GPIO0 đến GPIO21. Giá trị bit là 1/0 ứng với mức logic của chân GPIO là cao/thấp.
- **GPIO_OUT_REG** (GPIO output register), địa chỉ **0x60004004**, thanh ghi xuất tín hiệu (output). Bit 0 đến bit 21 tương ứng với GPIO0 đến GPIO21. Giá trị bit là 1/0 ứng với mức logic của chân GPIO là cao/thấp.
- **GPIO_OUT_W1TS_REG** (GPIO output set register), địa chỉ **0x60004008**, thanh ghi hỗ trợ việc thiết lập bit trên thanh ghi GPIO_OUT_REG. Giá trị bit bằng 1 sẽ thiết lập bit tương ứng trong thanh ghi GPIO_OUT_REG, các bit khác không thay đổi.
- **GPIO_OUT_W1TC_REG** (GPIO output clear register), địa chỉ **0x6000400C**, thanh ghi hỗ trợ việc xóa bit trên thanh ghi GPIO_OUT_REG. Giá trị bit bằng 1 sẽ xóa bit tương ứng trong thanh ghi GPIO_OUT_REG, các bit khác không thay đổi.
- **IO_MUX_GPIOn_REG** (n từ 0 đến 21), địa chỉ **0x60009004+4*n**, thanh ghi cấu hình chức năng cho chân GPIOn (từ GPIO0 đến GPIO21). Các chân của ESP32-C3 có thể được sử dụng như là các chân GPIO, hoặc cũng có thể được kết nối với tín hiệu ngoại vi. Các thanh ghi này được sử dụng để chọn chức năng cũng như cấu hình hoạt động cho các chân GPIO.

Chú ý:

- Nên sử dụng thanh ghi **GPIO_ENABLE_W1TS_REG** và **GPIO_ENABLE_W1TC_REG** để cấu hình các bit cần thiết cho thanh ghi **GPIO_ENABLE_REG**, tránh thay đổi các bit không liên quan.
 - Nên sử dụng thanh ghi **GPIO_OUT_W1TS_REG** và **GPIO_OUT_W1TC_REG** để cấu hình các bit cần thiết cho thanh ghi **GPIO_OUT_REG**, tránh thay đổi các bit không liên quan.
 - Cần tra cứu tài liệu để biết các chức năng của chân GPIO, từ đó chọn cấu hình phù hợp.
- ⇒ Sinh viên cần tra cứu trong tài liệu hoặc datasheet để nắm được rõ hơn về các thanh ghi này.

Home Assignment 1 – Bật/tắt LED

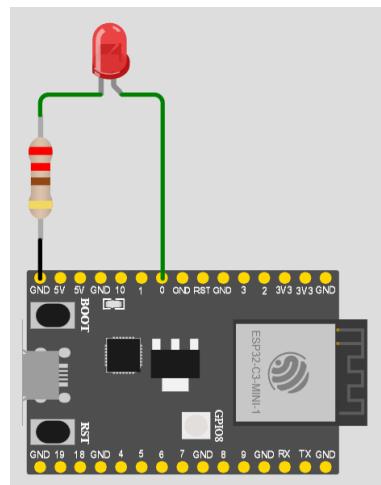
Ví dụ sau minh họa thực hiện mạch điều khiển bật/tắt đèn LED.

Sơ đồ ghép nối bao gồm một đèn LED nối cực dương vào chân GPIO0, cực âm nối tiếp điện trở 220 Ohm nối với chân GND.

Khi chân GPIO0 ở mức logic 0, đèn LED tắt. Khi chân GPIO0 ở mức logic 1, đèn LED sáng.

Sử dụng Wokwi để lắp mạch như hình minh họa.

Mã nguồn hợp ngữ của chương trình:



```
.global init

.eqv GPIO_ENABLE_REG, 0x60004020    # Thanh ghi cấu hình vào/ra các chân GPIO
.eqv GPIO_OUT_W1TS_REG, 0x60004008  # Thanh ghi thiết lập chân GPIO

init:
    li a1, GPIO_ENABLE_REG    # Thiết lập chân GPIO0 là chân xuất tín hiệu
    li a2, 0x01                # Nạp mặt nạ 0x01 vào thanh ghi GPIO_ENABLE_REG
    sw a2, 0(a1)

    li a1, GPIO_OUT_W1TS_REG  # Thiết lập chân GPIO0 ở mức cao, bật LED
    li a2, 0x01                # Nạp mặt nạ 0x01 vào thanh ghi GPIO_OUT_W1TS_REG
    sw a2, 0(a1)
```

Nạp chương trình vào Wokwi, chạy thử và quan sát kết quả.

Home Assignment 2 – Nhấp nháy LED

Ví dụ sau minh họa thực hiện mạch điều khiển đèn LED nhấp nháy. Sơ đồ lắp mạch giống Home Assignment 1. Để tạo hiệu ứng nhấp nháy, chương trình luân phiên thực hiện việc bật/tắt đèn LED và chờ một khoảng thời gian.

Mã nguồn hợp ngữ của chương trình:

```
.global init

.eqv GPIO_ENABLE_REG, 0x60004020
.eqv GPIO_OUT_W1TS_REG, 0x60004008
.eqv GPIO_OUT_W1TC_REG, 0x6000400C

init:
    li a1, GPIO_ENABLE_REG      # Thiết lập GPIO0 là chân output
    li a2, 0x01
    sw a2, 0(a1)

main_loop:
    li a1, GPIO_OUT_W1TS_REG  # Thiết lập GPIO0 ở mức cao
    li a2, 0x01
    sw a2, 0(a1)
    call delay_asm            # Delay

    li a1, GPIO_OUT_W1TC_REG  # Xóa GPIO0 về mức thấp
    li a2, 0x01
    sw a2, 0(a1)
    call delay_asm            # Delay

    j main_loop               # Loop

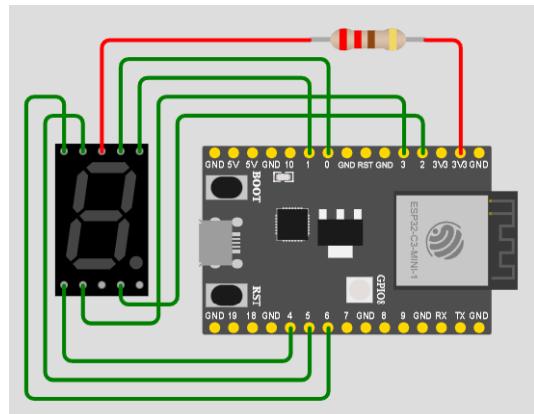
# Chương trình con delay, chờ một khoảng thời gian
delay_asm:
    li a3, 0                  # Giá trị biến đếm
    li a4, 5000000             # Thời gian chờ (số lần đếm)

loop_delay:
    addi a3, a3, 1
    blt a3, a4, loop_delay
    ret
```

Nap chuong trinh vào Wokwi, chay thử và quan sát kết quả.

Home Assignment 3 – Hiển thị số trên LED 7 đoạn

Ví dụ sau minh họa thực hiện mạch hiển thị số 0 trên LED 7 đoạn chung cực dương (Common Anode). Sơ đồ lắp mạch giống như hình minh họa. Các chân led a, b, c, d, e, f, g được nối tương ứng với các chân GPIO0, GPIO1, GPIO2, GPIO3, GPIO4, GPIO5, GPIO6. Các chân GPIO này cần được cấu hình để xuất tín hiệu.

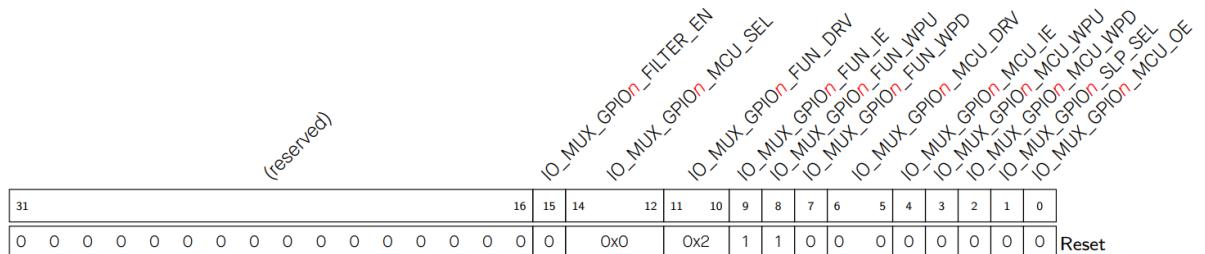


Các chân GPIO4, GPIO5, GPIO6, GPIO7 có chức năng mặc định thực hiện giao thức SPI. Để xuất tín hiệu ra các chân này, cần chọn chức năng sử dụng thanh ghi **IO_MUX_GPIO_n_REG** (n từ 4 đến 7).

Table 2-3. IO MUX Pin Functions

Pin No.	IO MUX / GPIO Name	IO MUX Function ^{1,4}				
		0	Type ³	1	Type	2
4	GPIO0	GPIO0	I/O/T	GPIO0	I/O/T	
5	GPIO1	GPIO1	I/O/T	GPIO1	I/O/T	
6	GPIO2	GPIO2	I/O/T	GPIO2	I/O/T	FSPIQ ^{5d}
8	GPIO3	GPIO3	I/O/T	GPIO3	I/O/T	
9	GPIO4	MTMS ^{5a}	I1	GPIO4	I/O/T	FSPIHD
10	GPIO5	MTDI	I1	GPIO5	I/O/T	FSPIWP
12	GPIO6	MTCK	I1	GPIO6	I/O/T	FSPICLK
13	GPIO7	MTDO	O/T	GPIO7	I/O/T	FSPID

Register 5.21. IO_MUX_GPIO_n_REG (n: 0-21) (0x0004+4*n)



Trường **IO_MUX_GPIO_n_MCU_SEL** (bit 12 – 14) được sử dụng để chọn chức năng cho chân GPIO_n. Trường này cần thiết lập giá trị là 1 để chọn chức năng là GPIO.

Chú ý: Đôi với mô phỏng bằng Wokwi không nhất thiết phải chọn chức năng cho các chân GPIO4, GPIO5, GPIO6, GPIO7. Nhưng khi sử dụng mạch phát triển thì cần phải thiết lập chức năng.

Mã nguồn hợp ngữ của chương trình:

```

.global init

.eqv GPIO_ENABLE_REG, 0x60004020      # Cho phép xuất tín hiệu các chân GPIO
.eqv GPIO_OUT_REG, 0x60004004          # Thiết lập mức logic đầu ra

.eqv IO_MUX_GPIO4_REG, 0x60009014      # Thiết lập chức năng chân GPIO4
.eqv IO_MUX_GPIO5_REG, 0x60009018      # Thiết lập chức năng chân GPIO5
.eqv IO_MUX_GPIO6_REG, 0x6000901C      # Thiết lập chức năng chân GPIO6
.eqv IO_MUX_GPIO7_REG, 0x60009020      # Thiết lập chức năng chân GPIO7

.text

init:
    li a1, GPIO_ENABLE_REG
    li a2, 0xFF                  # Xuất tín hiệu các chân GPIO0 đến GPIO7 (thiết lập 8 bits)
    sw a2, 0(a1)                 # Thiết lập các bits trong GPIO_ENABLE_REG

    # Thiết lập chức năng các chân GPIO4, GPIO5, GPIO6, GPIO7
    # Mặc định các chân này được sử dụng cho giao thức SPI

```

```
# Cần chuyển chức năng sang GPIO

li a2, 0x1000

li a1, IO_MUX_GPIO4_REG
sw a2, 0(a1)

li a1, IO_MUX_GPIO5_REG
sw a2, 0(a1)

li a1, IO_MUX_GPIO6_REG
sw a2, 0(a1)

li a1, IO_MUX_GPIO7_REG
sw a2, 0(a1)

# a1 chứa địa chỉ thanh ghi xác định mức logic đầu ra các chân GPIO
li a1, GPIO_OUT_REG
li a2, 0xC0
sw a2, 0(a1)          # Đưa tín hiệu ra cổng GPIO
```

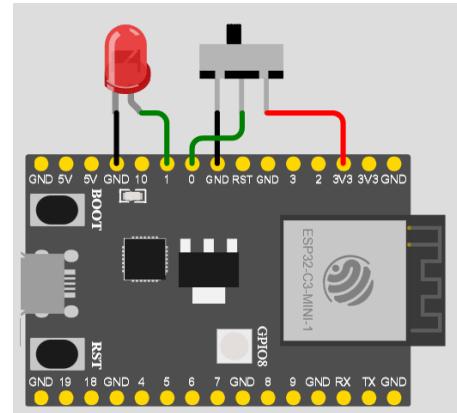
Home Assignment 4 - Đọc trạng thái công tắc/nút bấm

Ví dụ sau minh họa thực hiện mạch đọc tín hiệu vào tại chân GPIO0 và điều khiển bật tắt đèn LED tại chân GPIO1.

Sơ đồ mạch ghép nối như hình minh họa sau.

Thanh ghi **GPIO_IN_REG** chứa giá trị đầu vào các chân GPIO. Mặc định các chân GPIO là chân đầu vào nhận tín hiệu. Tuy nhiên cần thiết lập để cho phép nhận tín hiệu bằng cách thiết lập bit **IO_MUX_GPIOx_FUN_IE** trong thanh ghi **IO_MUX_GPIOx_REG** ứng với chân **GPIOx**.

Mã nguồn hợp ngữ của chương trình:



```
.global init

.eqv GPIO_OUT_W1TS_REG, 0x60004008 # Thanh ghi thiết lập
.eqv GPIO_OUT_W1TC_REG, 0x6000400C # Thanh ghi xóa
.eqv GPIO_ENABLE_REG, 0x60004020 # Thanh ghi cho phép xuất tín hiệu
.eqv GPIO_IN_REG, 0x6000403C # Thanh ghi đọc trạng thái GPIO
.eqv IO_MUX_GPIO0_REG, 0x60009004 # Thanh ghi thiết lập chức năng GPIO0

init:
    li a1, GPIO_ENABLE_REG # Thiết lập GPIO1 là chân xuất tín hiệu
    li a2, 0x02
    sw a2, 0(a1)

    li a1, IO_MUX_GPIO0_REG # Thiết lập cho phép GPIO0 nhận tín hiệu
    lw a2, 0(a1)
    ori a2, a2, 0x200      # Thiết lập bit IO_MUX_GPIO0_FUN_IE
    sw a2, 0(a1)
```

```
loop:  
    li a1, GPIO_IN_REG      # Đọc trạng thái các chân GPIO  
    lw a2, 0(a1)  
    andi a3, a2, 0x01      # Kiểm tra mức tín hiệu GPIO0  
    beq a3, zero, clear    # Nếu GPIO0 = 0 => Tắt LED  
set:  
    li a1, GPIO_OUT_W1TS_REG # Bật LED: Thiết lập GPIO1 = 1  
    li a2, 0x02  
    sw a2, 0(a1)  
    j next  
clear:  
    li a1, GPIO_OUT_W1TC_REG # Tắt LED: Xóa GPIO1 = 0  
    li a2, 0x02  
    sw a2, 0(a1)  
next:  
    j loop                  # Loop
```

Assignment 1

Tạo project để thực hiện và thử nghiệm Home Assignment 1. Cập nhật mã nguồn để thử nghiệm với các cổng GPIO khác (GPIO2, GPIO3, GPIO4).

Assignment 2

Tạo project để thực hiện và thử nghiệm Home Assignment 2. Cập nhật mã nguồn để thử nghiệm với các cổng GPIO khác (GPIO2, GPIO3, GPIO4) và thay đổi thời gian nhấp nháy đèn LED.

Assignment 3

Tạo project để thực hiện và thử nghiệm Home Assignment 3. Cập nhật mã nguồn để hiển thị các chữ số khác nhau (từ 0 đến 9).

Assignment 4

Tạo project để thực hiện và thử nghiệm Home Assignment 4. Cập nhật mã nguồn để sử dụng các cổng GPIO khác làm chân nhận tín hiệu (GPIO2, GPIO3, GPIO4).

Assignment 5

Tạo project để thực hiện mạch đếm từ 0 đến 9 trên LED 7 đoạn.

Bài 14. Lập trình hợp ngữ với ESP32-C3 – Lắp mạch trên breadboard

Mục đích

Trong bài thực hành này, sinh viên được giới thiệu và làm việc với kit phát triển ESP32-C3 thực tế. Sinh viên biết cách lắp các mạch đơn giản trên bảng mạch breadboard, sử dụng công cụ Visual Studio Code và PlatformIO để lập trình, chạy thử nghiệm và gỡ lỗi trên kit.

Tài liệu

Chuẩn bị

Sinh viên cần nhớ lại cách sử dụng breadboard để lắp các mạch thử nghiệm.

Giới thiệu phần mềm Visual Studio Code

Visual Studio Code (VS Code) là một trình soạn thảo mã nguồn được Microsoft phát triển. Nó hỗ trợ các chức năng gỡ lỗi, quản lý mã nguồn bằng Git, nổi bật cú pháp, tự hoàn thành mã nguồn. Nó cũng cho phép tùy chỉnh, người dùng có thể thay đổi theme, phím tắt, và các tùy chọn khác. Người dùng có thể tải và cài đặt miễn phí trên các hệ điều hành Windows, macOS, hoặc Ubuntu.

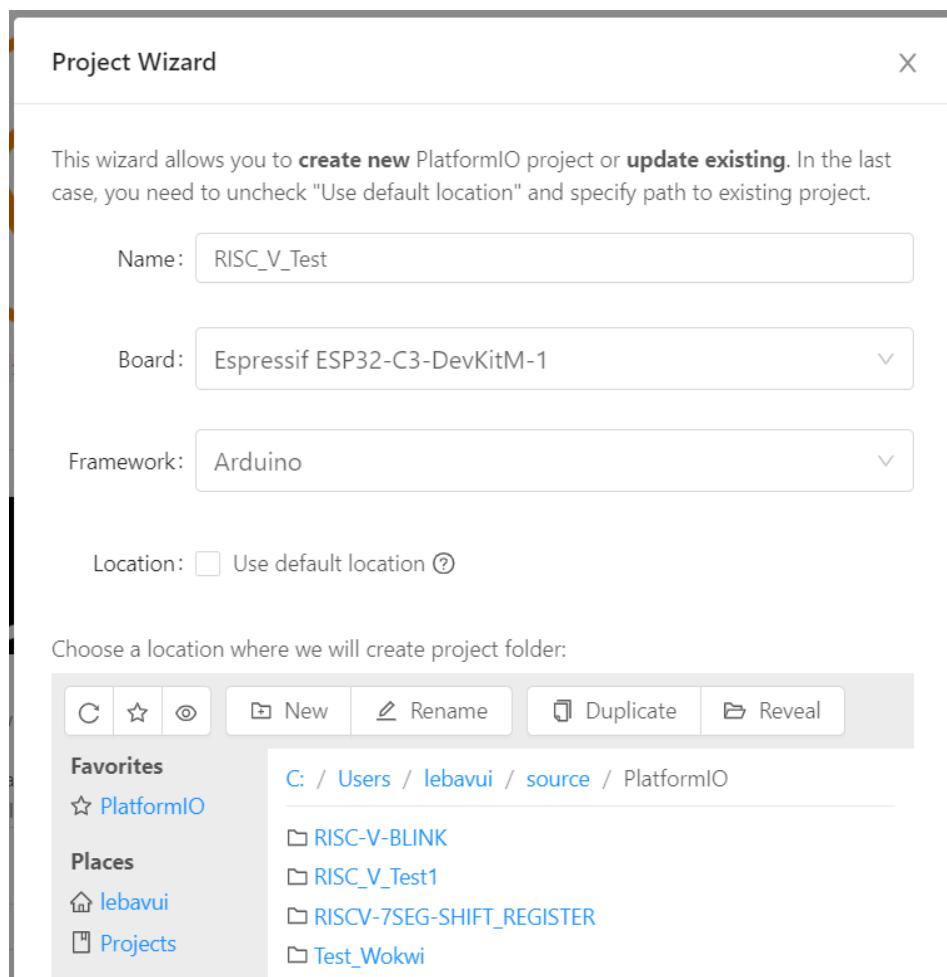
VS Code có thể được mở rộng thông qua việc cài thêm các plugin. Điều này giúp bổ sung chức năng cho trình biên tập và hỗ trợ thêm ngôn ngữ.

Giới thiệu công cụ PlatformIO

Lập trình hợp ngữ trên ESP32-C3 sử dụng VS Code và PlatformIO

Để lập trình hợp ngữ và chạy thử trên mạch phát triển (giả sử mạch đã được lắp trên breadboard, kit được kết nối với máy tính), các bước thực hiện như sau:

1. Tạo project mới.
 - a. Mở PlatformIO Home: Nhấn vào biểu tượng PlatformIO ở thanh công cụ bên trái, chọn PIO Home > Open
 - b. Chọn New Project
 - c. Trong cửa sổ Project Wizard đặt tên cho project, chọn board là **ESP32-C3-DevKitM-1**, chọn framework là **Arduino**, chọn thư mục lưu trữ project. Nhấn **Finish**.



d. Khi project được tạo xong, mở file **main.cpp** cập nhật nội dung như sau:

```
void setup() {}  
void loop() {}
```

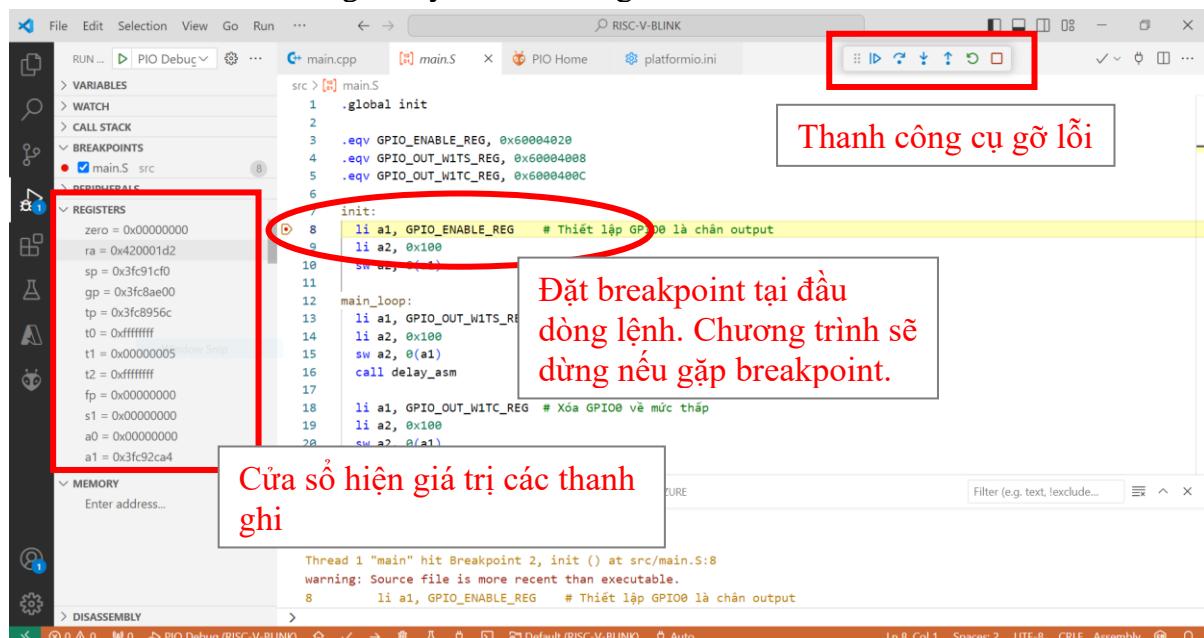
- e. Tạo file mới với phần mở rộng là .S. File này sẽ chứa mã nguồn hợp ngữ.
2. Soạn thảo mã nguồn hợp ngữ, khung chương trình hợp ngữ như sau (*khung chương trình này giống với project mô phỏng bằng Wokwi*):

```
# Cần định nghĩa hàm init để thực hiện chương trình hợp ngữ  
.global init  
  
# (Tùy chọn) Sử dụng chỉ thị .data để khai báo dữ liệu trong bộ nhớ  
.data  
  
# Chỉ thị .text đánh dấu bắt đầu đoạn lệnh  
# Nếu không có .data thì cũng không cần .text  
.text  
init:  
    # Chương trình hợp ngữ bắt đầu từ đây!!!  
    li a0, 1  
    li a1, 2  
    add a2, a1, a0
```

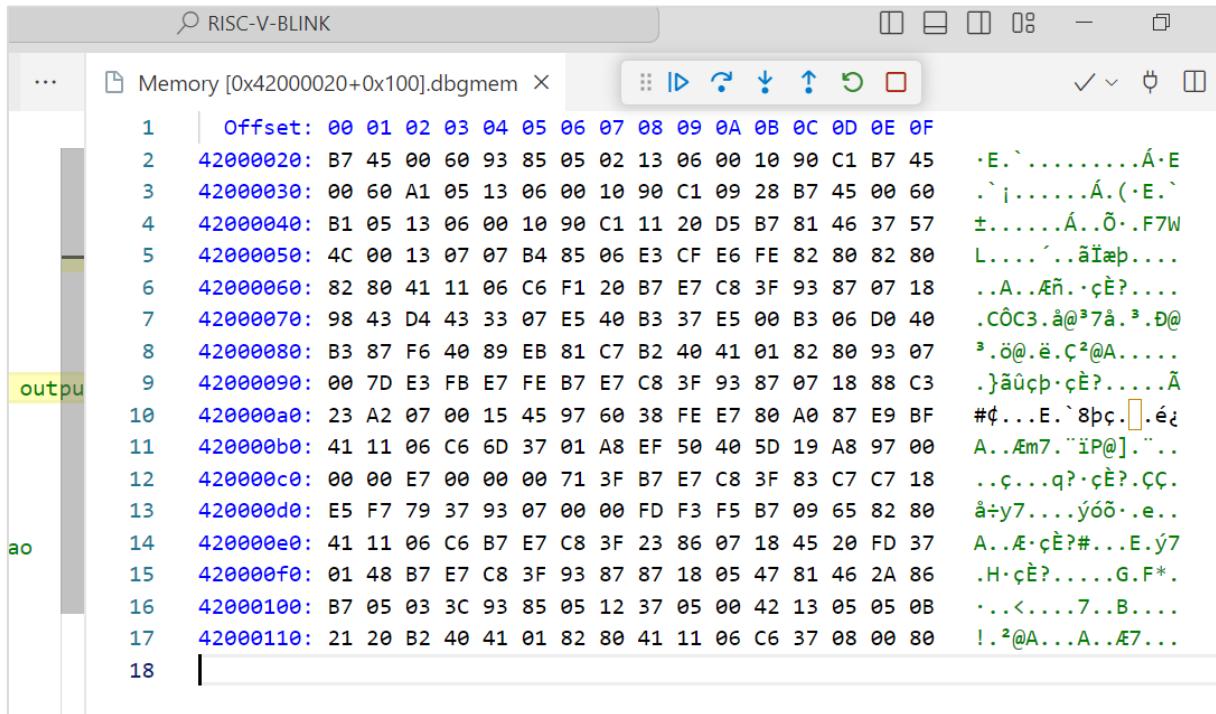
3. Dịch chương trình: nhấn nút Build (ở thanh công cụ bên dưới hoặc góc trên bên phải) hoặc nhấn tổ hợp phím tắt Ctrl+Alt+B. Quá trình dịch sẽ được thể hiện trong cửa sổ Terminal.

4. Nạp chương trình vào kit: nhấn nút Upload (ở thanh công cụ bên dưới hoặc góc trên bên phải) hoặc nhấn tổ hợp Ctrl+Alt+U. Chương trình sẽ được dịch lại và nạp vào kit. Quá trình dịch và nạp mã nguồn sẽ được thể hiện trong cửa sổ Terminal. Sau khi nạp chương trình xong, kit sẽ tự khởi động lại và hoạt động theo chương trình đã nạp.
5. Gỡ lỗi chương trình: kit ESP32-C3 tích hợp giao thức JTAG kết nối qua cổng USB cho phép gỡ lỗi chương trình (chạy từng lệnh, xem được giá trị các thanh ghi, dữ liệu được lưu trữ trong bộ nhớ). Để thực hiện quá trình gỡ lỗi, các bước cần thực hiện như sau:
 - a. Mở file **platformio.ini**, thêm vào dòng


```
debug_tool = esp-builtin
```
 - b. Nhấn vào nút Run and Debug (ở thanh công cụ bên trái)
 - c. Chọn kiểu debug là PIO Debug (chương trình sẽ được dịch, nạp và thiết lập chế độ gỡ lỗi)
 - d. Nhấn nút Start Debugging (hoặc nhấn F5)
 - e. Khi vào chế độ gỡ lỗi chương trình sẽ dừng lại. Tại thời điểm này, có thể thực hiện việc dừng lệnh tại breakpoint, chạy từng lệnh, theo dõi giá trị các thanh ghi, lấy dữ liệu trong bộ nhớ.



- f. Lấy dữ liệu từ bộ nhớ: Để lấy dữ liệu từ bộ nhớ của vi điều khiển, trong cửa sổ MEMORY nhấn vào Enter address, sau đó nhập địa chỉ bắt đầu và số bytes của vùng nhớ cần lấy. Một cửa sổ mới hiển thị dữ liệu từ vùng nhớ đã trích xuất. Chú ý: Dữ liệu này chỉ đọc, không ghi trực tiếp được vào bộ nhớ như chương trình giả lập.



The screenshot shows a memory dump from address 0x42000020 to 0x42000110. The left column lists memory addresses, and the right column shows the raw hex data followed by its ASCII representation. The memory contains various command bytes and data, including the instruction sequence for the LED blink pattern.

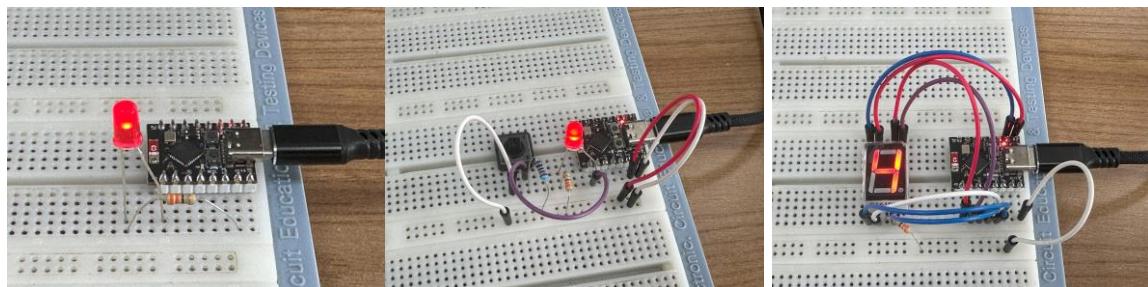
```

RISC-V-BLINK
Memory [0x42000020+0x100].dbgmem
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
42000020: B7 45 00 60 93 85 05 02 13 06 00 10 90 C1 B7 45
42000030: 00 60 A1 05 13 06 00 10 90 C1 09 28 B7 45 00 60
42000040: B1 05 13 06 00 10 90 C1 11 20 D5 B7 81 46 37 57
42000050: 4C 00 13 07 07 B4 85 06 E3 CF E6 FE 82 80 82 80
42000060: 82 80 41 11 06 C6 F1 20 B7 E7 C8 3F 93 87 07 18
42000070: 98 43 D4 43 33 07 E5 40 B3 37 E5 00 B3 06 D0 40
42000080: B3 87 F6 40 89 EB 81 C7 B2 40 41 01 82 80 93 07
42000090: 00 7D E3 FB E7 FE B7 E7 C8 3F 93 87 07 18 88 C3
420000a0: 23 A2 07 00 15 45 97 60 38 FE E7 80 A0 87 E9 BF
420000b0: 41 11 06 C6 6D 37 01 A8 EF 50 40 5D 19 A8 97 00
420000c0: 00 00 E7 00 00 00 71 3F B7 E7 C8 3F 83 C7 C7 18
420000d0: E5 F7 79 37 93 07 00 00 FD F3 F5 B7 09 65 82 80
420000e0: 41 11 06 C6 B7 E7 C8 3F 23 86 07 18 45 20 FD 37
420000f0: 01 48 B7 E7 C8 3F 93 87 87 18 05 47 81 46 2A 86
42000100: B7 05 03 3C 93 85 05 12 37 05 00 42 13 05 05 0B
42000110: 21 20 B2 40 41 01 82 80 41 11 06 C6 37 08 00 80

```

Chú ý: Để thực hiện được tính năng gõ lỗi với hệ điều hành Windows, cần phải cài đặt driver libusbK cho USB JTAG.

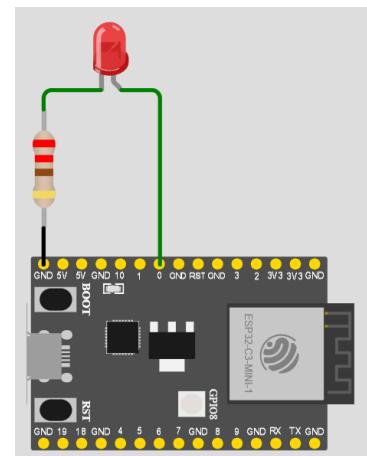
Một số hình ảnh thử nghiệm kit ESP32-C3 Super Mini trên breadboard.



Assignment 1

Lắp mạch đèn LED theo sơ đồ bên phải sử dụng breadboard, kit ESP32-C3 và các linh kiện cần thiết.

Tạo project lập trình hợp ngữ trong Arduino IDE. Viết mã nguồn điều khiển LED nhấp nháy (tham khảo Bài thực hành 13). Dịch, nạp chương trình vào kit và quan sát kết quả.

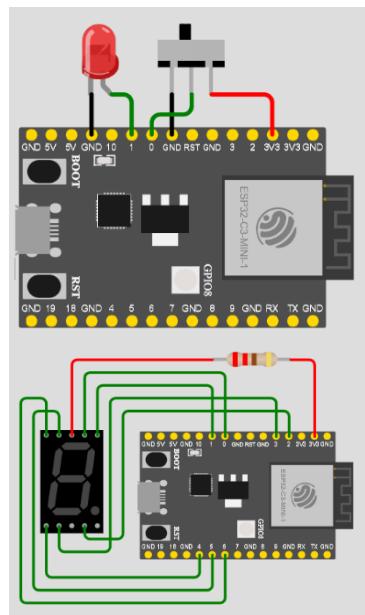


Assignment 2

Lắp mạch LED và công tắc gạt theo sơ đồ bên phải sử dụng breadboard, kit ESP32-C3 và các linh kiện cần thiết.

Tạo project lập trình hợp ngữ trong Arduino IDE. Viết mã nguồn đọc trạng thái công tắc và điều khiển LED bật/tắt tương ứng (tham khảo Bài thực hành 13).

Dịch, nạp chương trình vào kit và quan sát kết quả.



Assignment 3

Lắp mạch LED 7 đoạn sử dụng breadboard, kit ESP32-C3 và các linh kiện cần thiết.

Tạo project lập trình hợp ngữ trong Arduino IDE. Viết mã điều khiển LED 7 đoạn đếm và hiển thị từ 0 đến 9.

Dịch, nạp chương trình vào kit và quan sát kết quả.

Assignment 4

Lắp mạch LED 7 đoạn và công tắc gạt sử dụng breadboard, kit ESP32-C3 và các linh kiện cần thiết.

Tạo project lập trình hợp ngữ trong Arduino IDE. Viết mã nguồn đọc trạng thái công tắc và điều khiển LED 7 đoạn đếm tăng/giảm dần tùy thuộc vào trạng thái của công tắc. Dịch, nạp chương trình vào kit và quan sát kết quả.

Bài 15, 16. Project cuối kỳ

Mục đích

Sinh viên ôn lại các kiến thức đã được học trong các bài thực hành để lập trình giải quyết các bài toán cơ bản.

Time

Tuần 12: Sinh viên được giao đề tài project cuối kỳ

Tuần 15, 16: Sinh viên trình bày kết quả thực hiện với giảng viên hướng dẫn.