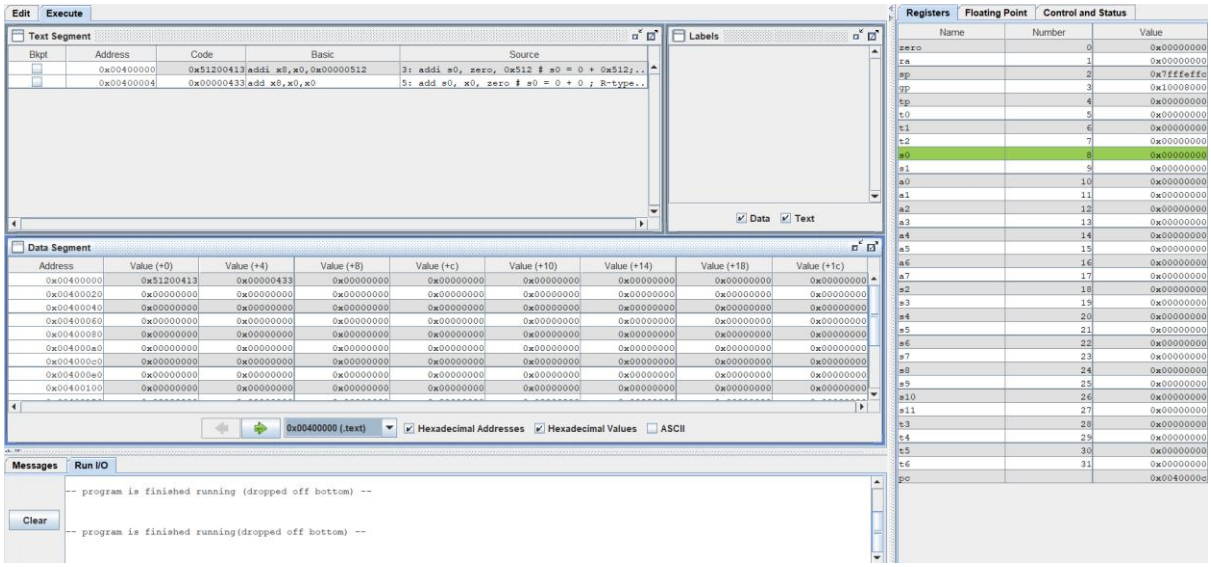
Bài 2. Tập lệnh, các lệnh cơ bản, các chỉ thị biên dịch

Nguyễn Thành Duy

MSSV: 20235696

Assignment 1: Lệnh gán số nguyên nhỏ 12-bit

Kết quả sau khi chạy chương trình:



Sau câu lệnh này:

```
addi s0, zero, 0x512    # s0 = 0 + 0x512; I-type: chỉ có thể lưu
                        # được hằng số có dấu 12 bits
```

Thanh ghi so thay đổi giá trị như mong muốn:

s0	8	0x00000512

Thanh ghi pc: tăng từ 0x00400000 -> 0x00400004

pc		0x00400004

Tiếp theo:

```
add  s0, x0, zero       # s0 = 0 + 0 ; R-type: có thể sự dụng số
                        # hiệu thanh ghi thay cho tên thanh ghi
```

Thanh ghi so thay đổi giá trị như mong muốn:

s0	8	0x00000000

Thanh ghi pc: 0x00400004 -> 0x00400008

pc		0x00400008

0x51200413

0101 0001 0010 |0000 0|000 |0100 0|001 0011

op: 001 0011

funct3: 000

➔ Addi

rd: 01000 => x8 => so

rs1: 00000 => x0 => zero

imm: 010100010010 => 512

⇨ addi so, zero, 0x512 (đúng như khuôn dạng lệnh)

0x00000433

0000 000|0 0000 |0000 0|000 |0100 0|011 0011

op: 011 0011

funct3: 000

funct7: 00000

➔ add

rd: 01000 => x8 => so

rs1: 00000 => x0 => zero

rs2: 00000 => x0

⇨ add so, xo, zero (đúng như khuôn dạng lệnh)

Sửa lại lệnh addi

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

Chương trình sẽ báo lỗi:

```
Error in C:\New folder\riscvl.asm line 5 column 16: "0x20232024": operand is out of range
Assemble: operation completed with errors.
```

Giải thích:

Do giá trị 0x20232024 vượt quá giới hạn giới hạn addi chỉ dùng để biểu diễn số 12-bit nên chương trình sẽ báo lỗi và không biên dịch được.

Assignment 2: Lệnh gán số 32-bit

Sau câu lệnh này:

```
lui  s0, 0x20232    # s0 = 0x20232
```

- Thanh ghi so:

s0	8	0x20232000

- Thanh ghi pc:

pc		0x00400004

Tiếp theo:

```
addi s0, s0, 0x024  # s0 = s0 + 0x024
```

- Thanh ghi so:

s0	8	0x20232024

- Thanh ghi pc:

pc		0x00400008

Ở bảng Data Segment:

Address	Value (+0)	Value (+4)
0x00400000	0x20232437	0x02440413

Ở bảng Text Segment:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20232437	lui x8,0x00020232	4: lui s0, 0x20232 # s0 = 0x20232
	0x00400004	0x02440413	addi x8,x8,0x00000024	5: addi s0, s0, 0x024 # s0 = s0 + 0x024

Ta thấy được rằng cách giá trị ở byte thứ nhất và thứ tư của địa chỉ 0x00400000 ở bảng Text Segment giống với mã máy(Code) ở bảng Text Segment.

Khi nạp một số 32-bit 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?

Giải thích:

Vì các Hằng số (immediate) trong RISC-V luôn là số bù 2 12-bit nên khi thực hiện cần mở rộng dấu thành 32-bits ở kiểu sign-extend.

Khi mở rộng dấu thành 32-bit để tránh tràn số thì ta bổ sung các F ở trước số cần mở rộng. Do vậy nên số 20-bit trong lệnh lui cần phải tăng lên 1 do các giá trị FF.. (-1) nên khi cộng ta chủ động cộng 1 vào số 20-bit.

Assignment 3: Lệnh gán (giả lệnh)

Sau khi chạy chương trình:

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20232437	lui x8,0x00020232	3: li s0, 0x20232024
	0x00400004	0x02440413	addi x8,x8,0x00000024	
	0x00400008	0x02000413	addi x8,x0,0x00000020	4: li s0, 0x20

Quan sát các lệnh ở cột Source và cột Basic trong cửa sổ Text Segment ta thấy:

Câu lệnh: li so, 0x20232024 được tách thành 2 lệnh chính thống là:

lui x8, 0x00020232 và addi x8, x8, 0x00000024

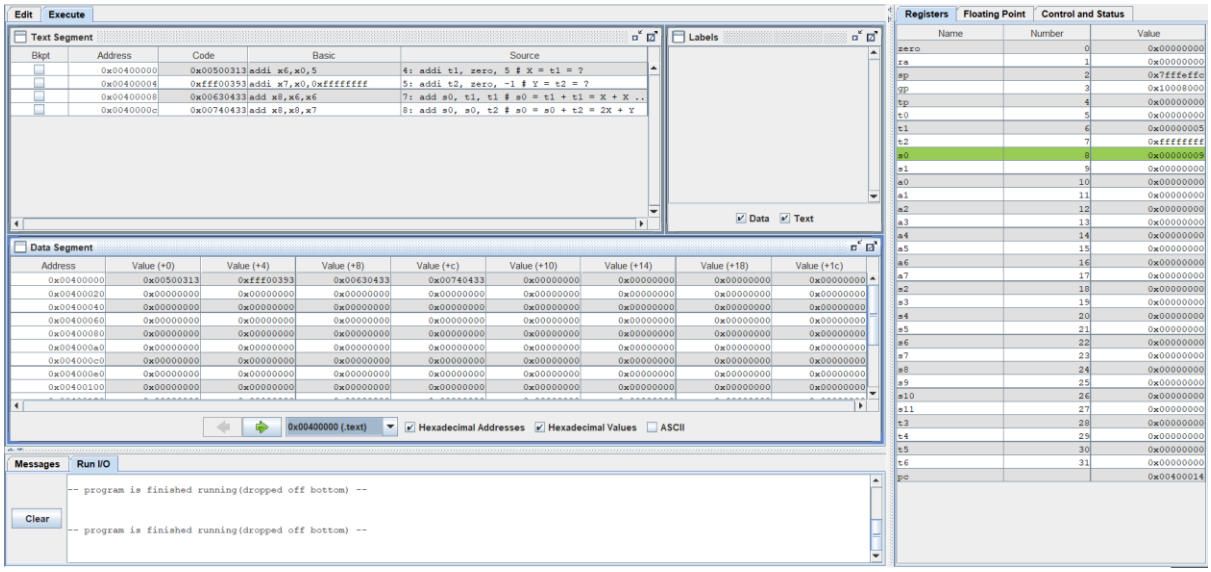
Câu lệnh: li so, 0x20 được biên dịch câu lệnh:

addi x8, x0, 0x00000020

Thực chất: li là câu lệnh gán giá trị nhưng tùy vào giá trị cần gán thì sẽ có cách gán hay được biên dịch thành các câu lệnh chính thống khác nhau.

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

Kết quả sau khi chạy chương trình:



Câu lệnh đầu tiên: X = 0 + 5 = 5

```
addi t1, zero, 5      # X = t1 = ?
```

Thanh ghi:

t1	6	0x00000005

Câu lệnh thứ hai: Y = 0 – 1 = - 1

```
addi t2, zero, -1     # Y = t2 = ?
```

Thanh ghi:

t2	7	0xffffffff

Câu lệnh thứ ba: so = X + X = 5 + 5 = 10

```
add s0, t1, t1     # s0 = t1 + t1 = X + X = 2X
```

Thanh ghi:

s0	8	0x0000000a

Câu lệnh thứ tư: so = 2X + Y = 10 – 1 = 9

```
add s0, s0, t2     # s0 = s0 + t2 = 2X + Y
```

Thanh ghi:

Kết thúc chương trình: Kết quả đúng như mong muốn

addi t1, zero, 5=> addi x6, x0, 5

op: 0010011

rd: x6 => 00110

funct3: 000

rs1: x0 => 00000

imm: 0000 0000 0101

⇨ 0x00500313 (giống mã máy)

addi t2, zero, -1 => addi x7,x0, -1

op: 0010011

rd: x7 => 00111

funct3: 000

rs1: x0 => 00000

imm: 1111 1111 1111

⇨ 0xfff00393 (giống mã máy)

Tương tự với: add

Assignment 5: Phép nhân

Câu lệnh thứ nhất:

```
addi t1, zero, 4          # X = t1 =?
```

- Thanh ghi:

Câu lệnh thứ hai:

```
addi t2, zero, 5          # Y = t2 =?
```

- Thanh ghi:

Câu lệnh thứ ba:

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

- Thanh ghi:

Kết quả của đoạn chương trình đúng như mong muốn

Giải thích:

- Ở câu lệnh đầu tiên để gán giá trị 4 vào t1
- Ở câu lệnh thứ hai để gán giá trị 5 vào t2
- Ở câu lệnh thứ ba để tính tích t1 và t2 và lưu vào s1

Lệnh chia:

riscv1.asm*

```
1  .text
2  addi t1, zero, 20
3  addi t2, zero, 5
4  div s1, t1, t2
```

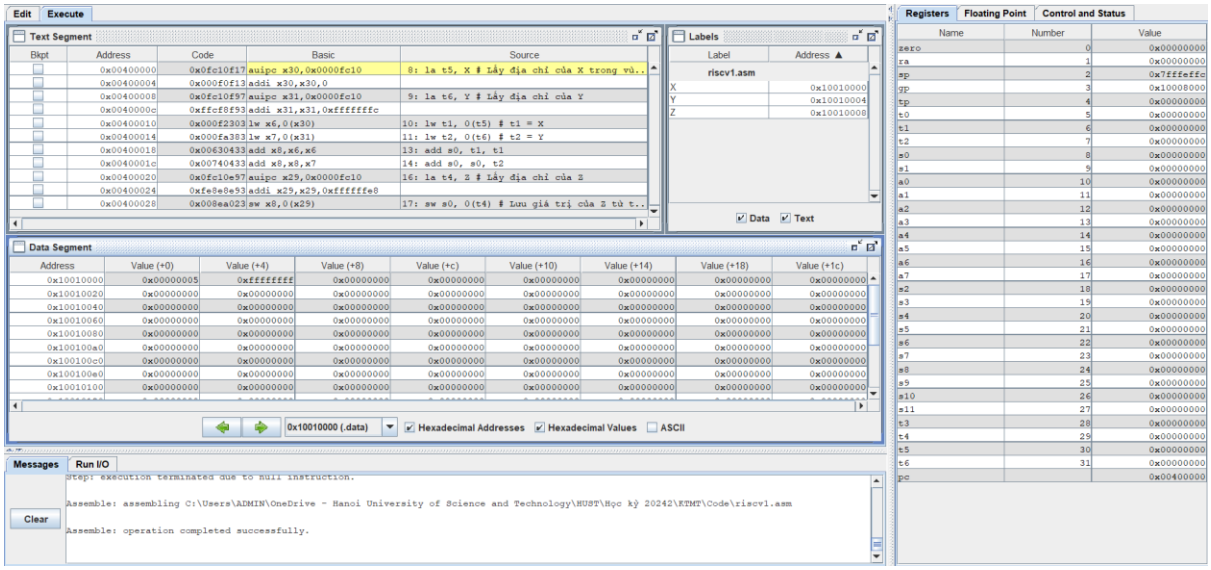
Tương tự như phép nhân thì chia dùng lệnh div

Kết quả của thanh ghi:

Kết quả s1 = 20 : 5 = 4 (giống với value ở thanh ghi)

Assignment 6: Tạo biến và truy cập biến

Kết quả sau khi chạy chương trình:



- Lệnh la được biên dịch thành 2 câu lệnh: auipc và addi

+ Địa chỉ tuyệt đối được chia thành hai phần: phần cao (20-bit) và phần thấp (12-bit). Kết hợp hai câu lệnh này để tải một địa chỉ đầy đủ 32-bit.

+ auipc: giúp lấy phần trên (20-bit) của địa chỉ nhãn dựa vào pc, auipc tính toán phần cao của địa chỉ label bằng cách lấy giá trị pc hiện tại và cộng thêm một giá trị offset.

+ addi: bổ sung phần thấp (12-bit) để hoàn chỉnh địa chỉ

⇨ Giúp tải địa chỉ của nhãn (label) vào thanh ghi
⇨ Cách này giúp chương trình có thể chạy độc lập với vị trí của bộ nhớ

- Dòng lệnh đầu tiên:

```
la t5, X        # Lấy địa chỉ của X trong vùng nhớ chứa dữ liệu
```

Khi chạy câu lệnh: Thanh ghi t5 có value ở thanh ghi là địa chỉ của X(đã định nghĩa ở trên)

t5	30	0x10010000

- Dòng lệnh thứ hai:

```
la t6, Y        # Lấy địa chỉ của Y
```

Tương tự như t5 thì t6 có value ở thanh ghi là địa chỉ của Y.

- Dòng lệnh thứ ba:

```
lw t1, 0(t5)    # t1 = X
```

Khi chạy câu lệnh: Giá trị value của t1 được lấy từ giá trị từ địa chỉ t5 mà giá trị của địa chỉ t5 là 5 (X = 5)

t1	6	0x00000005

- Dòng lệnh thứ tư: tương tự với dòng lệnh thứ ba (lấy giá trị từ địa chỉ t6 gán cho value của t2)
- Dòng lệnh thứ năm:

```
la t5, X        # Lấy địa chỉ của X trong vùng nhớ chứa dữ liệu
```

- Dòng lệnh thứ sáu, bảy:

```
add s0, t1, t1
add s0, s0, t2
```

Dùng để cộng 2t1 và t2 và gán cho value của s0 (s0 = 2t1 + t2 = 10 – 1 = 9)

s0	8	0x00000009

- Dòng lệnh thứ tám:

```
la t4, Z        # Lấy địa chỉ của Z
```

Được dùng để lấy địa chỉ của Z (xem trong cửa sổ Labels) và gán vào value của t4

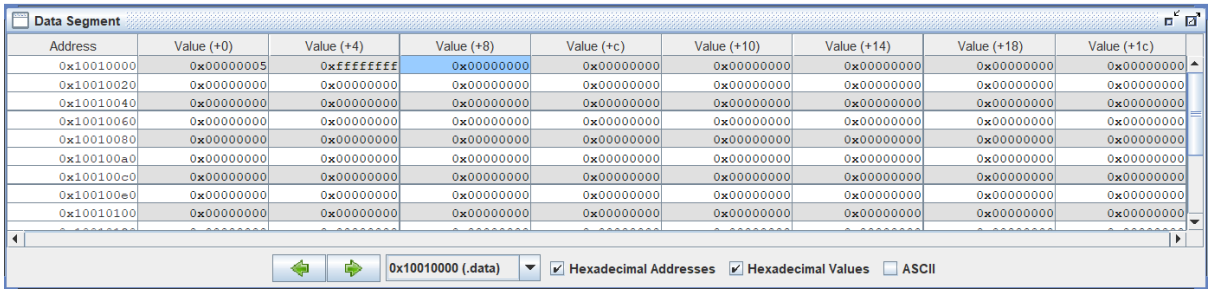
t4	29	0x10010008

- Dòng lệnh thứ chín:

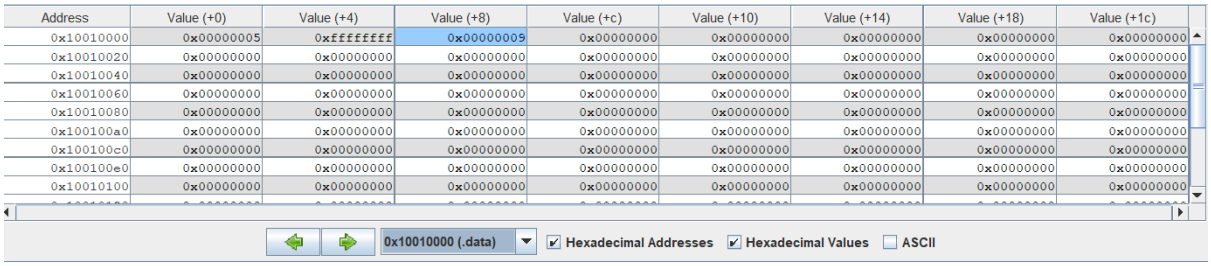
```
sw s0, 0(t4)    # Lưu giá trị của Z từ thanh ghi vào bộ nhớ
```

Sau khi chạy câu lệnh thì sẽ lấy giá trị từ thanh ghi s0 và ghi vào bộ nhớ của địa chỉ t4:

Bộ nhớ trước khi chạy:



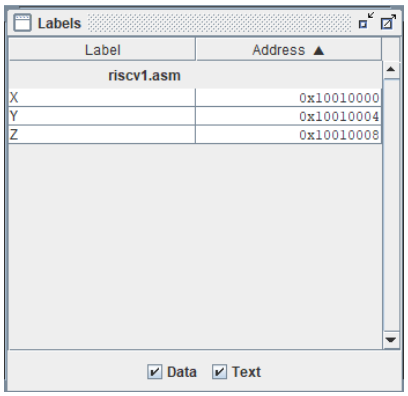
Bộ nhớ sau khi chạy:



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000005	0xffffffff	0x00000009	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Ở cửa số Labels:



Labels

Label	Address ▲
riscv1.asm	
X	0x10010000
Y	0x10010004
Z	0x10010008

☑ Data ☑ Text

- Vị trí của X, Y, Z trong bộ nhớ Data Segment:

X:

Address	Value (+0)
0x10010000	0x00000005

Y:

Address	Value (+0)	Value (+4)
0x10010000	0x00000005	0xffffffff

Z:

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x00000005	0xffffffff	0x00000000

Ta thấy giá trị của biến trong bộ nhớ giống với giá trị khởi tạo trong mã nguồn

- Vai trò câu lệnh lw, sw:

+) Về lw:

lw t1, o(t5) → Tải **giá trị tại địa chỉ t5** vào t1

(o(t5) là offset(độ dịch) tính từ t5)

lw t2, 0(t6) → Tải **giá trị tại địa chỉ t6** vào t2

(o(t6) là offset(độ dịch) tính từ t6)

Kết quả: t1 = X, t2 = Y

+) Về sw: Dùng để lưu một từ(32-bit) từ thanh ghi vào bộ nhớ

VD: # ao = 42 và s0 = 0x10010000

sw ao, o(so) # địa chỉ 0x10010000 sẽ chứa giá trị 42.

- Lệnh lb và sb:

+) Về lb: Nạp một byte (8-bit) từ bộ nhớ vào thanh ghi

+) Về sb: Lưu một byte (8-bit) từ thanh ghi vào bộ nhớ