BÁO CÁO THỰC HÀNH KIẾN TRÚC MÁY TÍNH TUẦN 2

Họ và tên: Hoàng Văn Thắng

MSSV: 20235828

Assignment 1: Lệnh gán số nguyên nhỏ 12-bit

Nhập chương trình:

```
1  # Laboratory Exercise 2, Assignment 1
2  .text
3          addi s0, zero, 0x512     # s0 = 0 + 0x512; I-type: chỉ có thể lưu
4                                   # được hằng số có dấu 12 bits
5          add s0, x0, zero         # s0 = 0 + 0; R-type: có thể sử dụng số
6                                   # hiệu thanh ghi thay cho tên thanh ghi
7
```

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:
    - Sự thay đổi giá trị thanh ghi **s0**.

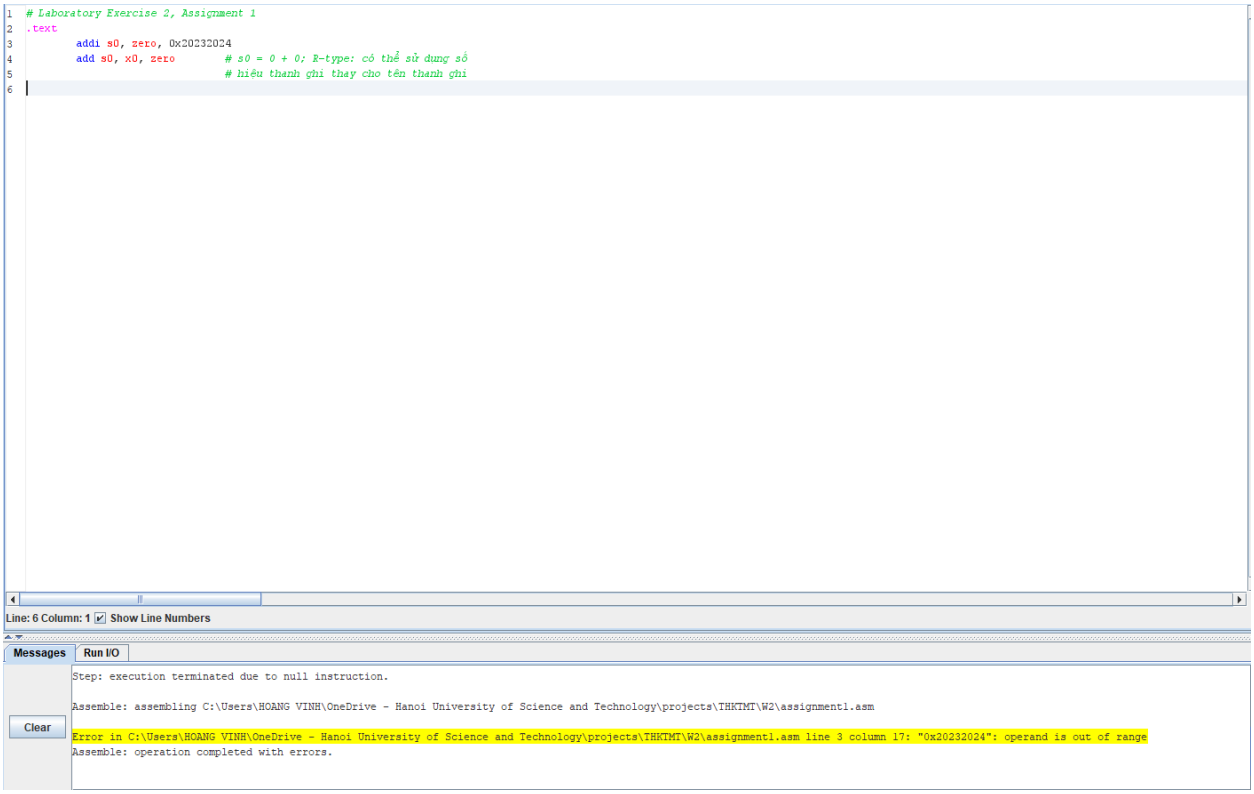
s0	8	0x00000512
s0	8	0x00000000

        - Sự thay đổi của thanh ghi **s0**: từ giá trị 0x00000000 chuyển thành 0x00000512 sau lệnh thứ nhất, rồi trở lại giá trị ban đầu sau lệnh thứ hai
    - Sự thanh đổi giá trị thanh ghi **pc** (thanh ghi pc nằm ở vị trí dưới cùng trong cửa sổ Registers).

pc		0x00400004
pc		0x00400008
pc		0x0040000c

        - Sự thanh đổi của thanh ghi **pc**: tăng thêm một khoảng có giá trị là 0x00000004 sau mỗi giá trị
- Sửa lại lệnh **addi** như bên dưới. Chuyện gì xảy ra sau đó. Hãy giải thích

Khi ta sử lại lệnh addi s0, zero, 0x512 thành add1 s0, 0x20232024 công cụ sẽ báo lỗi out of range vì giá trị đó vượt quá giới hạn của một số nguyên 12 bit có dấu.



Giải thích: Lệnh addi là lệnh I-Type, chỉ hỗ trợ gán các số nguyên có dấu trong phạm vi từ trong 12 bit có dấu, tức từ -2048 đến 2047 (12 bit). Nếu ta cố gán giá trị lớn hơn 2047 hoặc nhỏ hơn -2048, lỗi sẽ xuất hiện vì giá trị đó không thể biểu diễn bằng 12 bit

Assignment 2: Lệnh gán số 32-bit

Nhập chương trình:

```
1  # Laboratory Exercise 2, Assignment 2
2  # Load 0x20232024 to s0 register
3  .text
4          lui s0, 0x20232 # s0 = 0x20232
5          addi s0, s0, 0x024 # s0 = s0 + 0x024
6
```

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:
    - Sự thay đổi giá trị thanh ghi **s0**.

s0	8	0x20232000
s0	8	0x20232024

- Sự thay đổi của thanh ghi **s0**: sau lệnh *lui s0, 0x20232* giá trị của **s0** là 0x20232000. Sau lệnh *addi s0, s0, 0x024* giá trị của **s0** là 0x20232024

- Sự thanh đổi giá trị thanh ghi **pc** (thanh ghi pc nằm ở vị trí dưới cùng trong cửa sổ Registers).

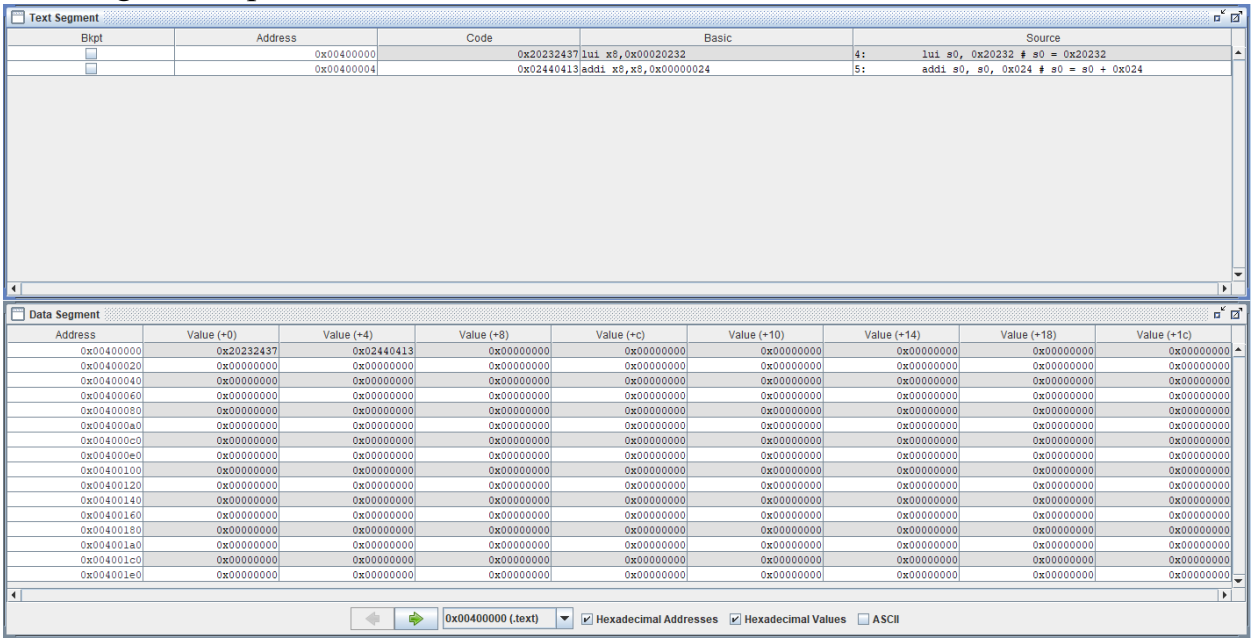
pc		0x00400004
pc		0x00400008
pc		0x0040000c

- Sự thanh đổi của thanh ghi **pc**: tăng thêm một khoảng có giá trị là 0x00000004 sau câu lệnh *lui s0, 0x20232* và *addi s0, s0, 0x024*

- 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.

Các byte đầu tiên ở vùng lệnh trùng với cột Code (mã máy theo Hexa) trong cửa sổ Text Segment ở phần thực thi



Chú ý:

- **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?**

Khi giá trị 12-bit của *addi* là số âm, mở rộng dấu (sign-extension) làm thay đổi giá trị. Vì vậy, cần tăng giá trị trong lệnh *lui* để bù đắp phần mở rộng dấu, đảm bảo số 32-bit đúng

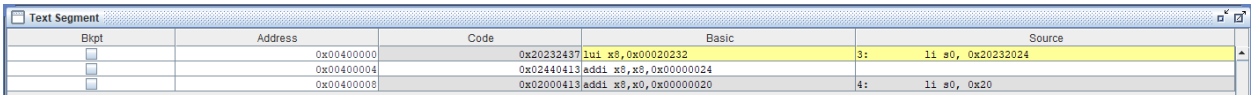
Assignment 3: Lệnh gán (giả lệnh)

Nhập chương trình:

```
1  # Laboratory Exercise 2, Assignment 3
2  .text
3          li s0, 0x20232024
4          li s0, 0x20
5
```

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.
    - Biên dịch



    - Quan sát và so sánh các lệnh ở cột Source và cột Basic trong cửa sổ Text Segment

Basic	Source
lui x8,0x00020232	3: li s0, 0x20232024
addi x8,x8,0x00000024	
addi x8,x0,0x00000020	4: li s0, 0x20

Ta dễ thấy được trong chương trình chỉ có hai câu lệnh nhưng khi biên dịch ở phần Basic và Source có 3 dòng. Đó là khi biên dịch lệnh *li s0, 0x20232024* được tách thành 2 lệnh

Tại sao lệnh *li s0, 0x20232024* lại tách thành 2 lệnh

Lệnh *li s0, 0x20232024*: Lệnh này được giả lập bởi các lệnh thực:

- *lui x8, 0x20232*: Nạp phần 20-bit cao của số 0x20232024 vào thanh ghi s0 (được dịch là x8 trong mã máy).
- *addi x8, x8, 0x24*: Cộng thêm phần 12-bit thấp của số 0x20232024 vào thanh ghi s0.

Lệnh *li s0, 0x20*: Đây là một giá trị nhỏ (trong phạm vi giá trị 12 bit có dấu), vì vậy chỉ cần một lệnh thực:

- *addi x8, x0, 0x20*: Nập giá trị 0x20 vào thanh ghi s0 (tương ứng với x8)

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

Nhập chương trình:

```
# 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 gỡ lỗi, chạy từng lệnh và dừng lại
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý:
    - Sự thay đổi giá trị các thanh ghi

t1	6	0x00000005
t2	7	0xffffffff
s0	8	0x0000000a
s0	8	0x00000009

    - Thanh **pc**:

pc		0x00400004
pc		0x00400008
pc		0x0040000c
pc		0x00400010
pc		0x00400014

Kết quả chạy:

s0	8	0x00000009

Sự thay đổi của các thanh ghi:

- Thanh t1 có giá trị trở thành 0x0000005
- Thành t2 có giá trị trở thành 0xffffffff
- Thanh s0 thay đổi theo như kết quả giống khi thực hiện phép tính đã được giải thích như trong mã nguồn trên
- Thanh ghi pc tăng thêm giá trị 0x00000004 sau mỗi câu lệnh
- **Kết quả chạy của chương trình đúng**

Assignment 5: Phép nhân

Nhập chương trình:

```
# 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 gỡ lỗi, chạy từng lệnh và quan sát sự thanh đổi của các thanh ghi. Kiểm tra kết quả xem có đúng không?
    - Sự thanh đổi của các thanh ghi

t1	6	0x00000004
t2	7	0x00000005
s1	9	0x00000014

    - Sự thay đổi của thanh **pc**

pc		0x00400004
pc		0x00400008
pc		0x0040000c
pc		0x00400010

    - Kết quả chạy

t1	6	0x00000004
t2	7	0x00000005
s0	8	0x00000000
s1	9	0x00000014

Nhận xét:

- t1 = 0x00000004 (tương đương giá trị 4 của hệ thập phân) – đây là giá trị của biến X
- t2 = 0x00000005 (tương đương giá trị 5 của hệ thập phân) – đây là giá trị của biến Y
- s1 = 0x00000014 (tương đương giá trị 20 của hệ thập phân) – đây là kết quả của phép nhân $4 \times 5 = 20$, giá trị lưu trong s1

- **Kết quả chính xác**

- Giải thích lại chi tiết đoạn lệnh trên.

addi t1, zero, 4: Gán giá trị 4 vào thanh ghi t1 (X = 4)

addi t2, zero, 5: Gán giá trị 5 vào thanh ghi t2 (Y = 5)

mul s1, t1, t2: Nhân t1 với t2, lưu kết quả 32-bit thấp vào thanh ghi s1

- Giải thích từng câu lệnh:

addi t1, zero, 4:

- Đây là lệnh cộng thức thời (I-type), gán giá trị 4 vào thanh ghi t1. Thanh ghi zero luôn có giá trị bằng 0 nên lệnh này thực tế là t1 = 0 + 4. Kết quả t1 = 4

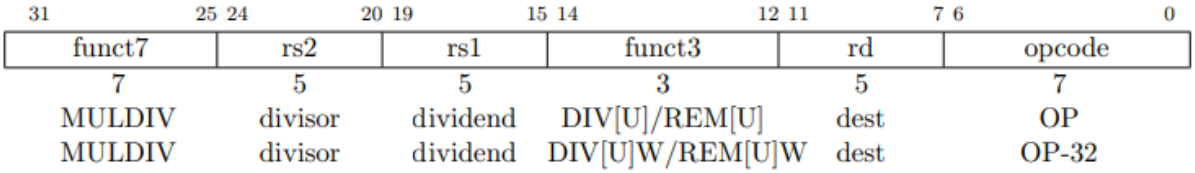
addi t2, zero, 5:

- Đây là lệnh cộng thức thời (I-type), gán giá trị 5 vào thanh ghi t2. Thanh ghi zero luôn có giá trị bằng 0 nên lệnh này thực tế là t2 = 0 + 4. Kết quả t1 = 5

mul s1, t1, t2:

- Lệnh multiply *mul rd, rs1, rs2*: Nhân hai toán hạng nằm trong thanh ghi rs1, rs2 và ghi 32-bit trọng số thấp của kết quả vào thanh ghi rd.

- Tìm hiểu lệnh chia trong RISC-V



Kiến trúc RISC-V cung cấp các lệnh khác nhau để thực hiện phép chia, các lệnh nằm trong extension RV32M (RISC-V multiply/divided extension).

+ Lệnh *div rd, rs1, rs2*: Chia hai toán hạng có dấu (signed) nằm trong thanh ghi rs1, rs2 và ghi kết quả vào thanh ghi rd

+ Lệnh *divu rd, rs1, rs2*: Chia hai toán hạng không dấu (unsigned) nằm trong thanh ghi rs1, rs2 và ghi kết quả vào thanh ghi rd

+ Lệnh *rem rd, rs1, rs2*: Lấy phần dư của phép chia có dấu và ghi kết quả vào thanh ghi rd

+ Lệnh *remu rd, rs1, rs2*: Lấy phần dư của phép chia không dấu và ghi kết quả vào thanh ghi rd

Ngoài ra còn một số trường hợp như phép chia cho số 0 và phép chia tràn số được tóm tắt ở bảng sau

Condition	Dividend	Divisor	DIVU	REMU	DIV	REM
Division by zero	x	0	$2^{XLEN}-1$	x	-1	x
Overflow (signed only)	-2^{XLEN-1}	-1	–	–	-2^{XLEN-1}	0

Ví dụ minh họa:

```
addi t1, zero, 11 # t1 = 11
addi t2, zero, 4 # t2 = 4
div s1, t1, t2 # s1 = t1 / t2 (s1 = 11/4 = 2)
rem s2, t1, t2 # s2 = t1 % t2 (s2 = 11 % 4 = 3)
```

Giải thích từng lệnh:

***addi t1, zero, 11*:**

- Gán giá trị 11 cho thanh ghi t1
- Kết quả: t1 = 11

***addi t2, zero, 4*:**

- Gán giá trị 3 cho thanh ghi t2
- Kết quả: t2 = 4

***div s1, t1, t2*:**

- Thực hiện phép chia có dấu, chia t1 cho t2 (11 / 4). Kết quả của phép chia là thương số 2, lưu vào thanh ghi s1
- Kết quả: s1 = 2

***rem s2, t1, t2*:**

- Thực hiện phép lấy phần dư của phép chia t1 cho t2 (11 % 4). Phần dư là 3, lưu vào thanh ghi s2
- Kết quả: s2 = 3

Luồng hoạt động của lệnh chia

1. Thực hiện phép chia 11/4: Kết quả thương số là 2, được lưu trong thanh ghi s1
2. Thực hiện phép chia lấy dư 11 % 4: Phần dư là 3, được lưu trong thanh ghi s2.

Kết quả của các thanh ghi:

+ Thanh ghi s1: Chứa giá trị 2

+ Thanh ghi s2: Chứa giá trị 3

Assignment 6: Tạo biến và truy cập biến

Nhập chương trình:

```
# Laboratory Exercise 2, Assignment 6
.data                   # Khởi tạo biến (declared 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 (declared 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.
    - 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**)

Ví dụ: lệnh *la t5, X*:

Lệnh **la** (load address) được dùng để lấy địa chỉ của một biến từ bộ nhớ vào thanh ghi. Nó thực hiện việc nạp giá trị của biến X vào thanh ghi t5

Cách biên dịch: Lệnh này thường được dịch thành một hoặc hai lệnh như **aiupc** (add upper immediate to PC) hoặc **addi** để tính toán địa chỉ thực trong bộ nhớ

auipc (add upper immediate to PC): Lệnh này lấy 20 bit cao của một giá trị tức thời, dịch trái 12 bit (tương được nhân với 4096), và cộng kết quả với giá trị hiện tại của PC. Kết quả được lưu vào thanh ghi đích

Công thức: rd = PC + (immediate << 12)

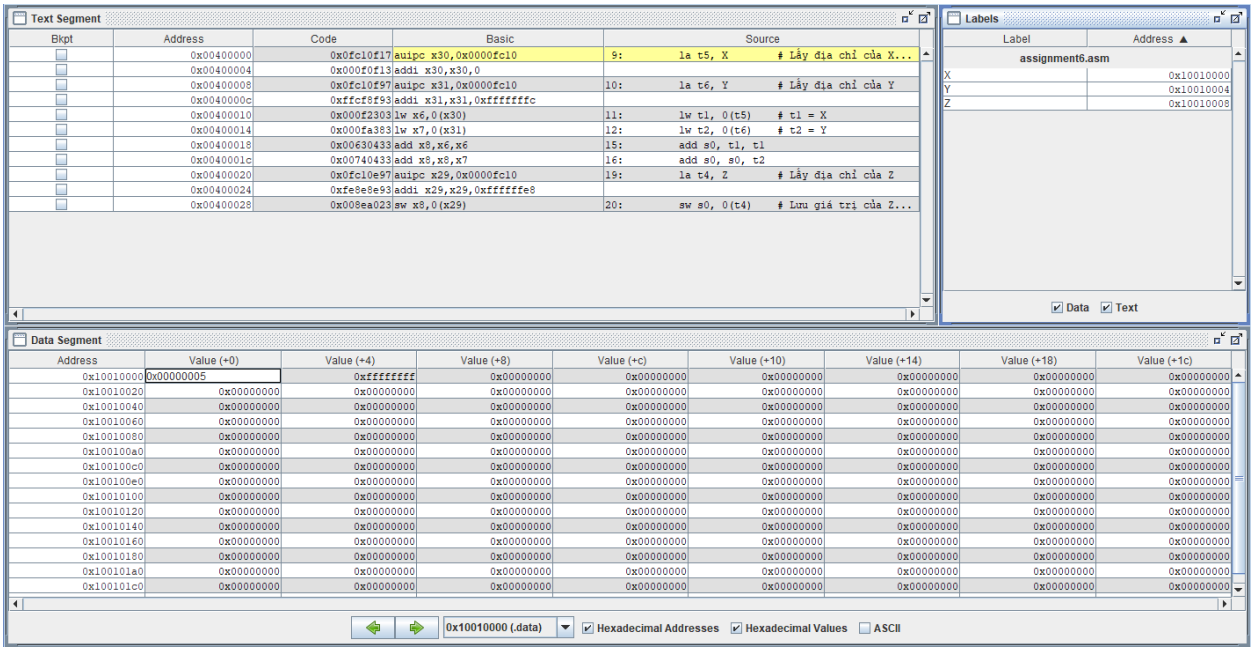
auipc được sử dụng để tính toán một địa chỉ gần với địa chỉ nhãn.

addi (Add Immediate) / lw (Load Word): Lệnh thứ hai được sử dụng để điều chỉnh giá trị trong thanh ghi đích để đạt được địa chỉ chính xác của nhãn.

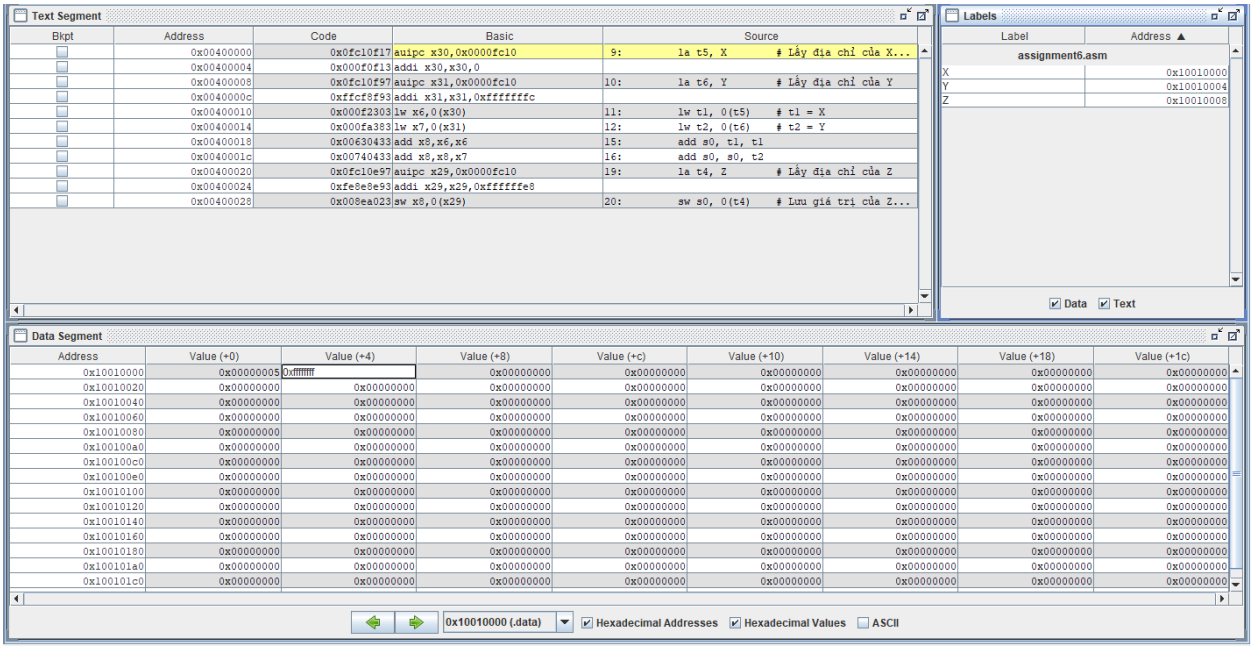
- addi: Thường được sử dụng nếu khoảng cách giữa PC (sau khi thực hiện auipc) và địa chỉ của nhãn đủ nhỏ (nằm trong khoảng -2048 đến +2047 byte). addi cộng một giá trị tức thời 12-bit (có dấu) với thanh ghi nguồn.
- lw: Đôi khi được sử dụng nếu la đang được sử dụng để nạp địa chỉ của một biến trong bộ nhớ, và lệnh tiếp theo là truy cập vào biến đó. Trong trường hợp này, lw sẽ nạp giá trị từ địa chỉ được tính toán (bởi auipc và phần bù) vào thanh ghi đích.

- Trong cửa sổ Labels, xem địa chỉ của biến X, Y, Z được lưu trong bộ nhớ.
  - 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.

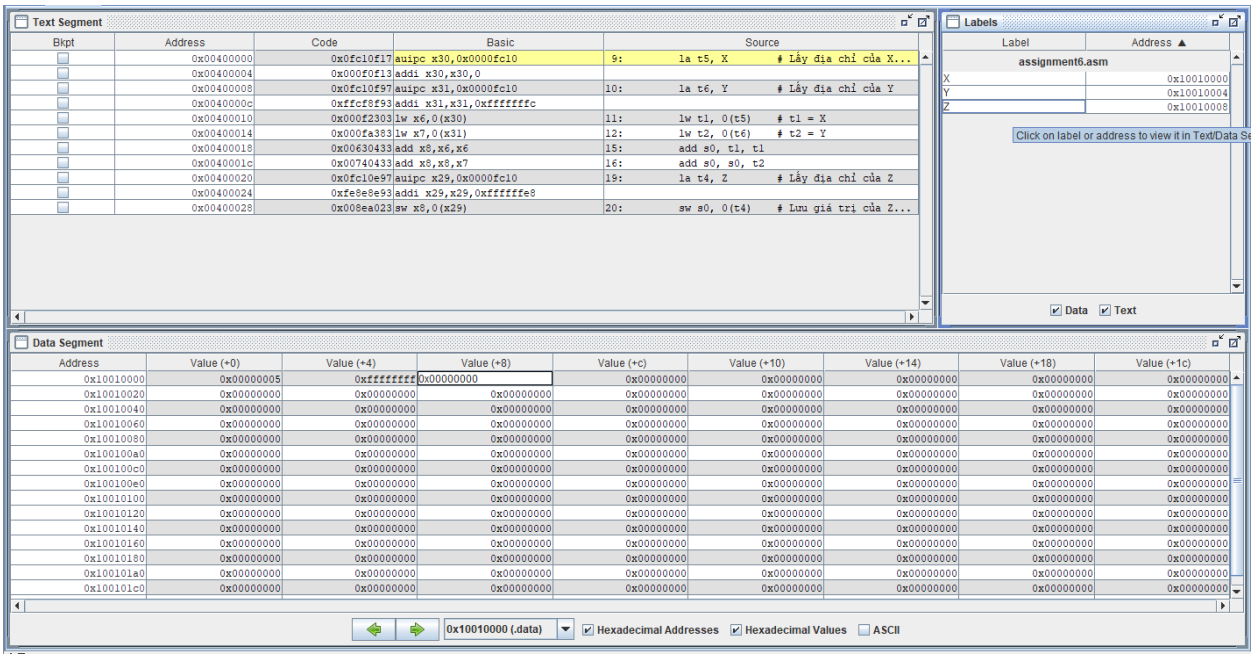
Biến X:


Text Segment
Bkpt Address Code Basic Source
0x00400000 0x0fc10f17 auipc x30,0x0000fc10 9: la t5, X # Lấy địa chỉ của X...
0x00400004 0x000f0f13 addi x30,x30,0
0x00400008 0x0fc10f97 auipc x31,0x0000fc10 10: la t6, Y # Lấy địa chỉ của Y
0x0040000c 0xffcf8f93 addi x31,x31,0xfffffffc
0x00400010 0x000f2303 lw x6,0(x30) 11: lw t1, 0(t5) # t1 = X
0x00400014 0x000fa383 lw x7,0(x31) 12: lw t2, 0(t6) # t2 = Y
0x00400018 0x00630433 add x8,x6,x6 15: add s0, t1, t1
0x0040001c 0x00740433 add x8,x8,x7 16: add s0, s0, t2
0x00400020 0x0fc10e97 auipc x29,0x0000fc10 19: la t4, Z # Lấy địa chỉ của Z
0x00400024 0xfe8e8e93 addi x29,x29,0xfffffff8
0x00400028 0x008ea023 sw x8,0(x29) 20: sw s0, 0(t4) # Lưu giá trị của Z...
Labels
Label Address
assignment6.asm
X 0x10010000
Y 0x10010004
Z 0x10010008
Data Text
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+c) Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000 0x00000005 0xffffffff
0x10010020 0x10010040 0x10010060 0x10010080 0x100100a0 0x100100c0 0x100100e0 0x10010100 0x10010120 0x10010140 0x10010160 0x10010180 0x100101a0 0x100101c0
0x00000000
0x10010000 (.data)
Hexadecimal Addresses
Hexadecimal Values
ASCII

Biến Y:


Text Segment
Bkpt Address Code Basic Source
0x00400000 0x0fc10f17 auipc x30,0x0000fc10 9: la t5, X # Lấy địa chỉ của X...
0x00400004 0x000f0f13 addi x30,x30,0
0x00400008 0x0fc10f97 auipc x31,0x0000fc10 10: la t6, Y # Lấy địa chỉ của Y
0x0040000c 0xffcf8f93 addi x31,x31,0xfffffffc
0x00400010 0x000f2303 lw x6,0(x30) 11: lw t1, 0(t5) # t1 = X
0x00400014 0x000fa383 lw x7,0(x31) 12: lw t2, 0(t6) # t2 = Y
0x00400018 0x00630433 add x8,x6,x6 15: add s0, t1, t1
0x0040001c 0x00740433 add x8,x8,x7 16: add s0, s0, t2
0x00400020 0x0fc10e97 auipc x29,0x0000fc10 19: la t4, Z # Lấy địa chỉ của Z
0x00400024 0xfe8e8e93 addi x29,x29,0xfffffff8
0x00400028 0x008ea023 sw x8,0(x29) 20: sw s0, 0(t4) # Lưu giá trị của Z...
Labels
Label Address
assignment6.asm
X 0x10010000
Y 0x10010004
Z 0x10010008
Data Text
Data Segment
Address Value (+0) Value (+4) Value (+8) Value (+c) Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000 0x00000005 0xffffffff
0x10010020 0x10010040 0x10010060 0x10010080 0x100100a0 0x100100c0 0x100100e0 0x10010100 0x10010120 0x10010140 0x10010160 0x10010180 0x100101a0 0x100101c0
0x00000000
0x10010000 (.data)
Hexadecimal Addresses
Hexadecimal Values
ASCII

Biến Z:



- Sử dụng công cụ gỡ lỗi, chạy từng lệnh và dừng lại
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý:
    - Sự thay đổi các thanh ghi

t5	30	0x10010000
t5	30	0x10010000
t6	31	0x10010008
t6	31	0x10010004
t1	6	0x00000005
t2	7	0xffffffff
s0	8	0x0000000a
s0	8	0x00000009
t4	29	0x10010020
t4	29	0x10010008

    - Sự thay đổi các thanh pc

pc		0x00400004
pc		0x00400008
pc		0x0040000c
pc		0x00400010

pc		0x00400014
pc		0x00400018
pc		0x0040001c
pc		0x00400020
pc		0x00400024
pc		0x00400028
pc		0x0040002c
pc		0x00400030

Các thanh t5, t6, t1, t2, s0, t4 thay đổi giá trị, thanh pc cứ mỗi bước tăng lên 4 đơn vị sau mỗi bước tính

Giải thích sự thay đổi:

- Thanh ghi t5: được dùng khi gọi lệnh *la t5, X*; có tác dụng nạp địa chỉ của biến X vào thanh t5
- Thanh ghi t6: được dùng khi gọi lệnh *la t6, Y*; có tác dụng nạp địa chỉ của biến Y vào thanh t6
- Thanh ghi t1: được dùng khi gọi lệnh *lw t1, 0(t5)*; lấy giá trị của biến t1 lưu vào bộ nhớ (5)
- Thanh ghi t2: được dùng khi gọi lệnh *lw t2, 0(t6)*; lấy giá trị của biến t2 lưu vào bộ nhớ (-1)
- Thanh ghi s0: được dùng khi gọi lệnh *add s0, t1, t1*; *add s0, s0, t2*; lưu giá trị của biểu thức Z = 2X + Y (10 khi gọi lệnh đầu, sau đó bằng 9 khi gọi lệnh tiếp)
- Thanh ghi t4: được dùng khi gọi lệnh *la t4, Z*; có tác dụng nạp địa chỉ của biến Z vào thanh t4
- Lệnh *sw s0, 0(t4)*: Lấy giá trị lưu trong thanh s0 (9), lưu vào địa chỉ thanh t4 đang trỏ tới (biến Z); lệnh này không làm thay đổi giá trị thanh ghi trừ thanh pc

    - Xác định vai trò của lệnh **lw** và **sw**.
- Lệnh lw: Lấy địa chỉ của biến kiểu word và lưu vào 1 thanh ghi
- Lệnh sw: Lấy địa chỉ của biến kiểu word lưu vào bộ nhớ

- Tìm hiểu thêm các lệnh **lb**, **sb**.

- lb (load byte): Nạp 1 byte (8 bit) từ bộ nhớ vào thanh ghi, với việc mở rộng dấu (sign-extended) thành 32 bit
- sb (store byte): Lưu 1 byte từ thanh ghi vào bộ nhớ