BÁO CÁO THỰC HÀNH KIẾN TRÚC MÁY TÍNH

Lab 4 : Các lệnh số học và logic

Họ tên	MSSV
Phạm Minh Hiển	20235705

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.

- **Home assignments 1:**
  - **TH1** : Không tràn số khi cộng 2 số dương :

```
.text
        li s1, 10
        #li s1, 0x7FFFFFFF
        #li s1, 0x80000000
        #li s2, -1
        li s2, 1
        li t0, 0
        add s3, s1, s2
        xor t1, s1, s2
        blt t1, zero, EXIT
        blt s1, zero, NEGATIVE
        bge s3, s1, EXIT
        j OVERFLOW
NEGATIVE:
        bge s1, s3, EXIT
OVERFLOW:
        li t0, 1
EXIT:
```

Sự thay đổi của thanh ghi:

- Ban đầu các thanh ghi đều có giá trị 0.
- Sau các lệnh gán **li,** thanh ghi **s1, s2, t0** có giá trị lần lượt là 10, 1, 0.

	s1	s2	t0	t1	s3
Sau lệnh **add**	0x0000000a (Giữ nguyên)	0x00000001 (Giữ nguyên)	0x00000000 (Giữ nguyên)	0x00000000 (Giữ nguyên)	0x0000000b
Sau lệnh **xor**	Giữ nguyên	Giữ nguyên	Giữ nguyên	0x0000000b	Giữ nguyên
Sau các lệnh tiếp theo	Giữ nguyên	Giữ nguyên	Giữ nguyên	Giữ nguyên	Giữ nguyên

- Chương trình chạy đến lệnh **bge** thì ta thấy, lúc này do s3 = 11 > 0 thoả mãn điều kiện nên sẽ trỏ đến *EXIT* và khi chạy tiếp sẽ kết thúc chương trình.
- Không xảy ra tràn số, kết quả s3 là 11 -> Đúng logic.
- Ta thấy thanh ghi pc cũng sẽ thay đổi như Lab trước, tăng 4 vào giá trị khi thực hiện các lệnh liên tiếp và sẽ thay đổi giá trị cũng như là trỏ đến vị trí mới khi rẽ nhánh hoặc nhảy.

- **TH2** : Không tràn số khi cộng 2 số âm :

```
.text
        li s1, -10
        #li s1, 0x7FFFFFFF
        #li s1, 0x80000000
        li s2, -1
        #li s2, 1
        li t0, 0
        add s3, s1, s2
        xor t1, s1, s2
        blt t1, zero, EXIT
        blt s1, zero, NEGATIVE
        bge s3, s1, EXIT
        j OVERFLOW
NEGATIVE:
        bge s1, s3, EXIT
OVERFLOW:
        li t0, 1
EXIT:
```

Sự thay đổi của thanh ghi:

- Tương tự với **TH1**, sau các lệnh **li,** các thanh ghi **s1, s2, t0** sẽ có giá trị lần lượt là -10, -1, 0.

	s1	s2	t0	t1	s3
Sau lệnh **add**	0xfffffff6 (Giữ nguyên)	0xffffffff (Giữ nguyên)	0x00000000 (Giữ nguyên)	0x00000000 (Giữ nguyên)	0xfffffff5
Sau lệnh **xor**	Giữ nguyên	Giữ nguyên	Giữ nguyên	0x00000009	Giữ nguyên
Sau các lệnh tiếp theo	Giữ nguyên	Giữ nguyên	Giữ nguyên	Giữ nguyên	Giữ nguyên

- Khi chương trình thực hiện lệnh **blt** thứ 2, do s1 = -10 < 0, thoả mãn điều kiện rẽ nhánh nên chương trình sẽ trỏ đến lệnh **bge** trong *NEGATIVE.*
- Lúc này do s1 = -10 > s3 = -11 nên sẽ rẽ nhánh sang *EXIT* và kết thúc chương trình.
- Không có tràn số, kết quả phép cộng là s3 = -11 -> Đúng logic.
- Ta thấy thanh ghi pc cũng sẽ thay đổi như Lab trước, tăng 4 vào giá trị khi thực hiện các lệnh liên tiếp và sẽ thay đổi giá trị cũng như là trỏ đến vị trí mới khi rẽ nhánh hoặc nhảy.

- **TH3** : Không tràn số khi cộng 2 số trái dấu :

```
.text
        li s1, 10
        #li s1, 0x7FFFFFFF
        #li s1, 0x80000000
        li s2, -1
        #li s2, 1
        li t0, 0
        add s3, s1, s2
        xor t1, s1, s2
        blt t1, zero, EXIT
        blt s1, zero, NEGATIVE
        bge s3, s1, EXIT
        j OVERFLOW
NEGATIVE:
        bge s1, s3, EXIT
OVERFLOW:
        li t0, 1
EXIT:
```

Sự thay đổi của thanh ghi:

- Tương tự với **TH1**, sau các lệnh **li,** các thanh ghi **s1, s2, t0** sẽ có giá trị lần lượt là 10, -1, 0.

	s1	s2	t0	t1	s3
Sau lệnh **add**	0x0000000a (Giữ nguyên)	0xffffffff (Giữ nguyên)	0x00000000 (Giữ nguyên)	0x00000000 (Giữ nguyên)	0x00000009
Sau lệnh **xor**	Giữ nguyên	Giữ nguyên	Giữ nguyên	0xfffffff5	Giữ nguyên
Sau các lệnh tiếp theo	Giữ nguyên	Giữ nguyên	Giữ nguyên	Giữ nguyên	Giữ nguyên

- Do vẫn thoả mãn điều kiện của lệnh **blt** thứ nhất, t1 = -11 < 0, nên sẽ trỏ đến *EXIT* và sẽ kết thúc chương trình khi chạy tiếp.
- Không xảy ra tràn số, kết quả phép cộng là s3 = 9 -> Đúng logic.
- Ta thấy thanh ghi pc cũng sẽ thay đổi như Lab trước, tăng 4 vào giá trị khi thực hiện các lệnh liên tiếp và sẽ thay đổi giá trị cũng như là trỏ đến vị trí mới khi rẽ nhánh hoặc nhảy.

- **TH4 + 5** : Tràn số khi cộng 2 số dương HOẶC 2 số âm :

```
.text
        #li s1, -10
        li s1, 0x7FFFFFFF
        #li s1, 0x80000000
        #li s2, -1
        li s2, 1
        li t0, 0
        add s3, s1, s2
        xor t1, s1, s2
        blt t1, zero, EXIT
        blt s1, zero, NEGATIVE
        bge s3, s1, EXIT
        j OVERFLOW
NEGATIVE:
        bge s1, s3, EXIT
OVERFLOW:
        li t0, 1
EXIT:
```

```
.text
        #li s1, -10
        #li s1, 0x7FFFFFFF
        li s1, 0x80000000
        li s2, -1
        li t0, 0
        add s3, s1, s2
        xor t1, s1, s2
        blt t1, zero, EXIT
        blt s1, zero, NEGATIVE
        bge s3, s1, EXIT
        j OVERFLOW
NEGATIVE:
        bge s1, s3, EXIT
OVERFLOW:
        li t0, 1
EXIT:
```

- Ta gán s1 = $2^{31} - 1$ và s2 = 1 (s1 = -2^{31} và s2 = -1 đối với trường hợp cộng 2 số âm), giá trị các thanh ghi sẽ thay đổi tương tự các trường hợp trước.
- Thanh ghi s3 có giá trị 0x80000000 (-2^{31}) đối với trường hợp cộng 2 số dương, và 0x7fffffff ($2^{31} - 1$) đối với trường hợp cộng 2 số âm.
- Với TH 2 số dương, khi chạy đến **bge** do s3 = -2^{31} < s1 = $2^{31} - 1$, nên sẽ chạy lệnh nhảy **j** OVERFLOW. Tiếp đó gán giá trị 1 (mang ý nghĩa tràn số) cho t0 và kết thúc chương trình.
- Với TH 2 số âm, khi chạy đến **blt** thứ 2, do s1 = -2^{31} < 0 nên sẽ rẽ nhánh *NEGATIVE*, thực hiện **bge**, do s1 = -2^{31} < s3 = $2^{31} - 1$ nên sẽ chạy tiếp *OVERFLOW* và gán giá trị 1 cho t0 và kết thúc chương trình.
- Kết luận, phép tính tràn số và cho kết quả s3 = -2^{31} và $2^{31} - 1$ tương ứng 2 TH.

- **Home assignments 2:**

```
.text
        li s0, 0x12345678
        andi t0, s0, 0xff
        andi t1, s0, 0x0400
```

- Lệnh **lui** gồm 2 lệnh gán giá trị 0x12345678 cho s0.
- Sau lệnh **andi** thứ nhất, chương trình thực hiện phép **and** cho s0 và 0x000000ff, ta thấy do logic phép and thực hiện trên từng bit nên ta sẽ lấy được LSB của s0 là 78 lưu dưới dạng thập lục phân 0x00000078 vào t0.
- Lệnh **andi** thứ 2 thực hiện phép and cho s0 và 0x00000400 (0000 0000 0000 0000 0000 0100 0000 0000) cho phép lấy bit thứ 10 của s0 là 1 lưu trong t1 là 0x00000400.

- **Home assignments 3:**

```
.text
        li s0, 1
        li t1, 2
        sll s1, s0, t1
```

- Hai lệnh **li** gán giá trị 1 và 2 lần lượt cho t0, t1.
- Lệnh **sll** dịch trái giá trị lưu trong s0 2 bit (t1) là 0000 0000 0000 0000 0000 0000 0000 0001 thành 0000 0000 0000 0000 0000 0000 0000 0100 tương ứng giá trị 4 trong thập phân và lưu giá trị 0x00000004 vào thanh ghi s1.

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)

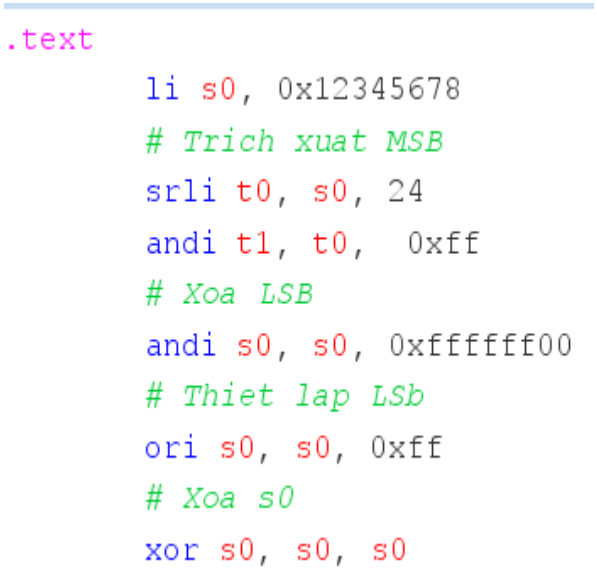
MSB: Most Significant Byte (Byte có trọng số cao)

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

SOURCE CODE :

.text

```
li s0, 0x12345678
# Trich xuat MSB
srli t0, s0, 24
andi t1, t0,  0xff
# Xoa LSB
andi s0, s0, 0xffffff00
# Thiet lap LSB
ori s0, s0, 0xff
# Xoa s0
xor s0, s0, s0
```



- Lệnh **lui** gán giá trị 0x12345678 cho s0.
- Để trích xuất MSB, **srli** thực hiện dịch phải logical giá trị trong s0 24 bit và lưu vào t0. Lúc này t0 có giá trị 0000 0000 0000 0000 0000 0000 0001 0010 trong hệ nhị phân tương ứng với 0x00000012 trong hệ thập lục phân. Lệnh **andi** lấy 8 bit cuối từ trái sang của t0 (tương ứng MSB của s0) và gán giá trị cho t1 là 0x00000012.Ta thu được MSB là 12.
- Để xoá LSB, ta thực hiện **andi** s0 với 1111 1111 1111 1111 1111 1111 0000 0000, từ đó xoá đi 8 bit cuối cùng từ trái sang tương ứng với LSB của s0 và lưu kết quả 0x12345600 vào s0.

- Để thiết lập LSB, **ori** thực hiện phép or s0 và 0000 0000 0000 0000 0000 0000 1111 1111, thiết lập 8 bit cuối từ trái sang đều là 1 cho s0 có giá trị 0x123456ff.
- Để xoá s0, **xor** s0 với chính nó, do đặc tính phép xor nên 2 bit giống nhau sẽ ra 0, s0 có giá trị 0x00000000.

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:

SOURCE CODE:

```
.text
        li s1, 10
        li s2, 11
        sub s0, zero, s1 # neg s0, s1
        addi s0, s1, 0 # mv s0, s1
        xori s0, s0, 0xFFFFFFFF # not s0, s0
        bge s2, s1, label # ble s1, s2, label
START: # dung de test bge
        addi s1, s1, 10
EXIT: # label
```

```
.text
        li s1, 10
        sub s0, zero, s1 # neg s0, s1
        addi s0, s1, 0 # mv s0, s1
        xori s0, s0, 0xFFFFFFFF # not s0 ( -1 )
        bge s2, s1, EXIT # ble s1, s2, label
START:
        addi s1, s1, 10
EXIT:
```

a. **neg s0, s1 # s0 = -s1**
   - Lệnh **sub** thực hiện phép trừ 0 – s1 lấy được kết quả -s1 lưu vào s0. Từ đó lấy được giá trị âm của s1.
   - Ở đây ta lấy ví dụ s1 = 10, khi thực hiện **sub** sẽ cho ra s0 = -10.

b. **mv s0, s1 # s0 = s1**
   - Lệnh **addi** thực hiện phép cộng s1 + 0 rồi gán vào s0, khi này ta thu được giá trị s0 = s1.
   - Ở đây ta thực hiện 10 + 0 và lưu vào s0, s0 = s1 = 10.

c. **not s # s0 = bit_invert(s0)**
   - Lệnh **xori** cho phép thực hiện phép or s0 với 1111 1111 1111 1111 1111 1111 1111 1111 trên từng bit và lấy được giá trị đảo từng bit của s0 (do 1 với 1 ra 0 và 1 với 0 ra 1).
   - Ví dụ ở đoạn code trên, ta lấy được giá trị đảo của s0 là 0000 0000 0000 0000 0000 0000 1010 tương ứng với giá trị 0xfffffff5 (-11).

d. **ble s1, s2, label # if (s1 <= s2) j label**

- Lệnh **bge** xét điều kiện s2 >= s1 tương đương với giả lệnh **ble** xét điều kiện s1 <= s2.
- Ở đây ta, lấy ví dụ cho 2 nhãn là *START* và *EXIT*, do s2 = 11 > s1 = 10, nên chương trình sẽ rẽ nhánh *EXIT* và kết thúc chương trình sau lệnh **bge**.

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.

SOURCE CODE:

```
.text
    li s1, 100
    # li s1, 0x7fffffff
    # li s1, 0x80000000
    # li s2, 1
    li s2, -1
    li t0, 0 # Danh gia tran so
    add s3, s1, s2
    xor s0, s1, s2 # Kiem tra dau s1 va s2
    blt s0, zero, EXIT # Neu s1 va s2 khac dau, re nhanh EXIT
    xor s0, s1, s3 # Kiem tra dau s3 va s1
    bgt s0, zero, EXIT # Neu dau s3 giong s1, re nhanh EXIT
OVERFLOW:
    li t0, 1 # Tran so
EXIT:
```

❖ Chạy thử code với các trường hợp :

- **Hai số khác dấu (không tràn số):**
  - s1 = 100 (0x00000064), s2 = -1 (0xffffffff).
  - s3 = 99 (0x00000063).
  - Chương trình chạy đến **blt** kiểm tra s1 và s2 khác dấu nên rẽ nhánh *EXIT* và kết thúc chương trình.
  - t0 = 0, chương trình không tràn số, cho kết quả s3 = 99 -> Hoạt động đúng logic.

- **<u>Hai số cùng dương (tràn số):</u>**
  - s1 = 2^{31} -1 (0x7fffffff), s2 = 1 (0x00000001).
  - s3 = -2^{31} (0x80000000).
  - Chương trình chạy đến **bgt** kiểm tra s3 và s1 không cùng dấu nên tiếp tục chạy lệnh **li** trong *OVERFLOW*, đặt t0 = 1 và kết thúc chương trình.
  - Vậy phép cộng tràn số và cho kết quả s3 = -2^{31} -> Hoạt động như dự đoán.
- **<u>Hai số cùng âm (tràn số):</u>**
  - s1 = -2^{31} (0x80000000), s2 = -1 (0xffffffff).
  - s3 = $2^{31} – 1$ (0x7fffffff).
  - Chương trình chạy đến **bgt** kiểm tra s3 và s1 không cùng dấu nên tiếp tục chạy lệnh **li** trong *OVERFLOW,* đặt t0 = 1 và kết thúc chương trình.
  - Vậy phép cộng tràn số và cho kết quả s3 = $2^{31} – 1$ -> Hoạt động như mong đợi.

Assignment 5:

Viết chương trình thực hiện nhân một số nguyên bất kỳ với một lũy thừa của 2 (2, 4, 8, 16, …) mà không sử dụng lệnh nhân. Ví dụ: Cho 2 thanh ghi t1 = 6, t2 = 8. Yêu cầu viết chương trình tính tích của 2 thanh ghi này mà không sử dụng lệnh nhân.

SOURCE CODE:

```
.text
    li t1, 6 # So nguyen
    li t2, 8 # Luy thua cua 2
    li t3, 0 # Bien dem
COUNT:
    srli t2, t2, 1 # Dich phai 1 bit (Chia 2)
    beq t2, zero, END # Neu t2 = 0 ket thuc dem
    addi t3, t3, 1 # Tang bien dem
    j COUNT
END:
    addi t5, t1, 0 # Khoi tao tong
LOOP:
    beq t3, zero, ENDLOOP # Neu dem ve 0 thi dung
    slli t5, t5, 1 # Dich trai 1 bit (Nhan 2)
    addi t3, t3, -1 # Giam bien dem
    j LOOP
ENDLOOP:
```

- Thử code với số nguyên t1 = 6 và t2 = 8 là luỹ thừa của 2.
- Sau vòng lặp COUNT, ta thu được số luỹ thừa t3 = 3, tức là phép tính sẽ là 6 x 2^3 = 48, hay ta sẽ dịch trái t1 3 bit.
- Vòng lặp LOOP cho ta dịch trái t5 (lấy giá trị ban đầu là t1 = 6) 3 bit thu được kết quả t5 = 48 (0x00000030). -> Chương trình hoạt động như kỳ vọng.
- Khi thử với t1 = 7 và t2 = 16, chương trình cho ra kết quả tương tự, t5 = 112 (0x00000070) -> Chương trình hoạt động chính xác.

Kết luận :

- Phép dịch bit có điểm lợi hơn phép nhân là không phụ thuộc vào Extension M.
- Trong phép nhân luỹ thừa của 2 thì việc sử dụng phép dịch bit sẽ có hiệu suất cao hơn phép nhân (đỡ tốn tài nguyên hơn).