1. Instruction使用 請看PDF

2. 可用的reg如右圖 , 一般寫ABI name

注意事項: 假設有fuction A , B ,C
且 形式為 A call B call C

A ,B中都使用了:
reg t0 (屬於caller)
reg s2 (屬於callee)

那麼在A中，當A call B 時
他需要存 reg t0 (用sp存)
而不需要存reg s2 (B如果用到 B會幫A存)

同理
在B中，當B被A call 但他也call C時
他也需要存 reg t0 (用sp存)
而不需要存reg s2 (C如果用到 C會幫B存)

存法在下一頁

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

舉裡來說 在 fuc A中: 使用t0, t1 (caller) s2 (callee)

XXXX code XXXX 我要呼叫B了
1. 先存好 caller type reg (不然B有可能會改到)

addi sp, sp , -16 (-16代表存2 sp指向stack address最高位
)
sd t0 , 0(sp) //把t0存到0(sp)
sd t1 , 8(sp) //把t1存到8(sp)

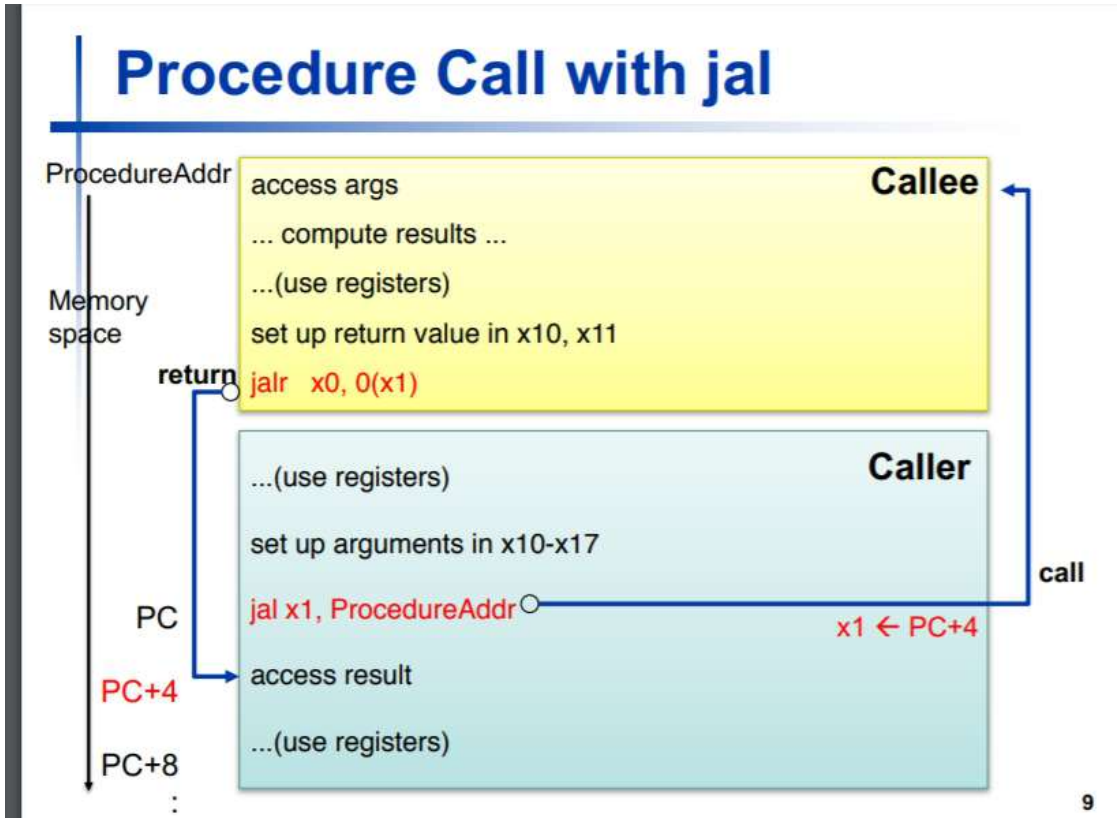
2. 叫B (注意有固定形式 x1=ra)

jal x1 , B (x1 PC +4 就是現在位置的下一行) (下頁有重點
)

3. 叫完之後把 caller type reg 值恢復

ld t0 , 0(sp) //把0(sp)存回t0
ld t1 , 8(sp) //把8(sp)存回t1
addi sp, sp , 16 (恢復stack 為最高位 避免浪費)

Note 1 : 不用存s2 反正 如果B有用到他需要負責存(也就是
A call 完之後 s2的值會跟A call 前一樣)



Note 2 : 以上都是約定成俗的習慣，但如果你確定B、C都用到 t0,t1當然你也可以不要存

在B裡面 (被A call 的)
XXXX code XXXX

最後做完我要回A了 ? A在哪? 在x1 = ra

所以在B的最後會寫出:

jarl x0 , 0(x1) = ret = jr ra (三種寫法都一樣 回到x1所指的位置)

注意注意 : 如果你B在結束前又call C
jal x1 , C (那很抱歉你的x1就被蓋掉了妳回不去A了 幫你QQ)

所以請記得先存x1 (在B中)

```
addi  sp, sp , -8
sd  x1 , 0(sp)
jal  x1 , C
sd  x1 , 0(sp)
ld  x1 , 0(sp)
addi  sp, sp , 8
```

下一頁開始
part A
Part B說明

PART A : 把下面這個用組合語言寫出來

```
uint64_t lfsr64(uint64_t status, uint64_t feedback){
    uint64_t lfsr=status;
    lfsr = (lfsr & 1) ? (lfsr >> 1) ^ feedback : (lfsr >> 1); //cycle one step of LFSR
    return lfsr;

}
```

1. a ? b : c ⍰ a = 1 時執行b 否則執行c
2. a 是 lfsr&1 . b是 (lfsr>>1)^feedback . C是(lfsr>>1)

下一頁

```
.section .data
.align 8
# There are two global variables:
#  * seed: address of seed for this lfsr64 function call
#  * feedback: address of FEEDBACK for this lfsr64 function call
seed: .dword  SEED
feedback: .dword FEEDBACK

.section .text
.global main
main:

  # your code goes here

  # do not modify following
  li  a0, 0       #assign 0 to a0
  ret             #return (jump to the addr store in register ra)
```

Code 寫這邊
其他我也看不
懂不好意思

2. Remember to store the result of the function back to seed in your assembly code.

注意事項 (seed = status、feedback = feedback)

1. 取得 seed , feedback

法一 :
ld t0 , seed (小寫) 此時t0是seed這個address位置得值
ld t1 , feedback 此時t1是feedback這個address位置

法二: 拜託用這個 等等你就知道為啥
la 即 load address
la t0 , seed 此時t0是seed這個address本身
la t1 , feedback 此時t1是feedback這個address本身
ld t2 , 0(t0) 此時t2是seed這個address位置得值
ld t3 , 0(t1) 此時t3是feedback這個address位置得值

2. 存result 到 seed (是個address) , 假設result = t5
用法二時 :
sd t5 , 0(t0) [只有這樣會成功]

用法一時 :
sd t5 , t0 [失敗] 法一存不進去
sd t5 , seed [失敗]
sd t5 , 0(seed) [失敗] 拜託我也不知道為啥不過不要

PART B :重寫lfsr64 還有 generate_maze

```
########################################
# Make 'lfsr64' a function:
#   Similar to the previous task, the only difference is lfsr64 now becomes a function.
#   That is,
#   a0 --> status
#   a1 --> feedback
#       the return value should be put in a0 before ret
#       uint64_t lfsr64(uint64_t status, uint64_t feedback){
#               uint64_t lfsr=status;
#               lfsr = (lfsr & 1) ? (lfsr >> 1) ^ feedback : (lfsr >> 1);
#               return lfsr;
#       }
#   as designated in declaraction of lfsr64() in generate_maze.h
########################################

.section .data
.align 1

.section .text
.global lfsr64
lfsr64:

    # your code goes here
    # remember to save register such as s0 or ra onto the stack if you used it in the assembly.

    ret     #return, same as "jr ra"
```

跟剛剛part A一樣
但是直接把a0當作 status = seed , a1 當作feed back
所以不用ld , 最後把要return的值 assign 給a0就好

sd a0 , result

Note : a0 ,a1 是 caller type 所以你也不用幫call lfsr64的fu
他們在call lfsr64的時候會自己存，然後你也沒call 其他人
以都不用管 可以亂蓋調a0 a1的值(最後要assign 給a0)

Code 寫這邊 Reg用t0~t6 (caller type)就好 這樣不用
因為lfsr64不會再叫其他人了，所以他不用存t0~t6

PART B :generate_maze 一言難盡 這我真的無法

只能提醒你們幾點(你們自己看我的CODE啦 然後啥時換我躺啦)

1. Fuction input =a0 ,a1 , a2 … / output[即return值]=a0 按照順序排列

```
#ifndef GENERATE_MAZE_H
#define GENERATE_MAZE_H
#include <stdio.h>
#define MAXSIZE 1000

        void generate_maze(uint64_t* maze, int h, int w, int i_i, int i_j, uint64_t* seed);
        int available_dir(uint64_t* maze, int i_i, int i_j);
        int random_dir(uint64_t* seed, int range);
        int choose_dir(uint64_t* maze, int i_i, int i_j, int r);
        uint64_t random_lfsr(uint64_t value, uint64_t range);
        uint64_t lfsr64(uint64_t status, uint64_t feedback);

#endif
```

maze = a0 ,h=a1 ,w=a2, i_i= a3 , i_j =a4 , seed =a5

maze = a0 , i_i= a1 , i_j =a2 return 值為a0

maze = a0 , i_i= a1 , i_j =a2 , r =a3 return 值為a0

看下一頁

所以 請記得

```
do {
    range = available_dir(maze, i_i, i_j);
    if(range==-1)  break; // leave the loop
    r=random_dir(seed, range);
    direction=choose_dir(maze, i_i, i_j, r);
    next_cell(maze,i_i, i_j, direction);
    generate_maze(maze, h, w, next_i, next_j, seed);
} while (1);
```

1. 先存好 a0~a5 x1** (用sp)
2. 把maze丟到a0 , i_i丟到a1 , i_j丟到a2
3. 呼叫 jal x1 , available_dir
4. 呼叫完後先把a0(=range)存好,再視情況(看下個fuc會用到啥) assign 值給a0~.....(同第2.)

NOTE : 每次call fuction x1都會被改到
所以請每call 一次 generate_maze就存一次x1 (jal x1 , available_dir這種不用)
只有 jal x1 , generate_maze才要 (但是存的地方請在一開始就存 -> 不知道我說啥的看code)

我要躺啦我不管啦我程式很爛ㄟ.....你們快點做事我要打LOL