



IMAGINATION大學計劃

RVfpga實驗4

函數呼叫

1. 簡介

函數呼叫對於任何程式都至關重要，因為可以實現模組化與程式碼重複使用，從而簡化程式碼的編寫和偵錯。C程式設計語言還包含常用C函數（例如隨機數產生器和常用數學函數）的標準函數庫以及處理器/開發板專用函數庫。高階函數按照*呼叫慣例*轉換為組合語言。本實驗將展示如何在C程式中編寫和使用函數（程式設計師編寫的兩個函數以及C函數庫中包含的函數），以及如何用組合語言實作這些函數。在實驗的末尾，我們提供了程式編寫練習，該程式會使用函數和呼叫函數庫

2. 編寫使用函數的C程式

函數也稱為子常式或程序，是指封裝到具有指定操作和介面的程式碼區塊中的一段程式碼。這種模組化結構可降低複雜性並支援程式碼重複使用，從而提高效率。可以從程式的任何位置呼叫函數，且呼叫的函數執行完畢後即可立即恢復執行程式。函數可由其他函數（稱為*巢狀函數*）甚至同一函數（稱為*遞迴呼叫*）呼叫。

要編寫使用函數的RISC-V程式，請按照實驗2和實驗3中所述的通用步驟進行操作：

1. 建立RVfpga專案
2. 編寫C程式
3. 將RVfpgaNexys下載到Nexys A7 FPGA開發板（請記住，也可以使用Verilator或Whisper模擬這些程式）
4. 編譯、下載和執行/偵錯程式

有關這些步驟的詳細說明，請參閱實驗2。下面簡要說明了各項步驟。

第1步. 建立RVfpga專案

在以下資料夾中建立名為project1的專案：

```
[RVfpgaPath]/RVfpga/Labs/Lab4
```

第2步. 編寫C程式

現在需要將C程式新增到專案中。建立一個新檔案，然後在專案中輸入或複製/貼上以下C程式。下列檔案也提供該程式：

```
[RVfpgaPath]/RVfpga/Labs/Lab4/LedsSwitches_functions.c
```

```
// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

void IOsetup();
```

```

unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main ( void )
{
    unsigned int switches_val;

    IOsetup();
    while (1) {
        switches_val = getSwitchVal();
        writeValtoLEDs(switches_val);
    }

    return(0);
}

void IOsetup()
{
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal()
{
    unsigned int val;

    val = READ_GPIO(GPIO_SWs);    // read value on switches
    val = val >> 16;    // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val)
{
    WRITE_GPIO(GPIO_LEDs, val);    // display val on LEDs
}

```

將檔案儲存到專案的src目錄下，並將檔案命名為**LedsSwitches_Functions.c**。

第3步. 將RVfpgaNexys下載到Nexys A7 FPGA開發板

遵循RVfpga實驗2和實驗3中的說明，將RVfpgaNexys下載到Nexys A7開發板。

第4步. 編譯、下載和執行程式

現在即可在RVfpgaNexys上編譯、下載和執行/偵錯程式。

按下「Run」（執行）和「Start Debugging」（開始偵錯）按鈕，然後按兩次「Step Over」（單步跳過）按鈕（位於頂端工具列中）或按兩次F10，直至到達呼叫getSwitchVal()函數的第19行。然後按下「Step Into」（單步進入）按鈕（或F11），進入getSwitchVal()函數。如果無法檢視val變數，請展開左側工具列中的「VARIABLES」（變數）→「Local」（區域）欄位。此時，val變數在程式中可能會被列為「optimized out」（優化輸出）。執行一次單步（單步跳過或單步進入）操作即可看到val變數變為開關的值，如圖1所示。

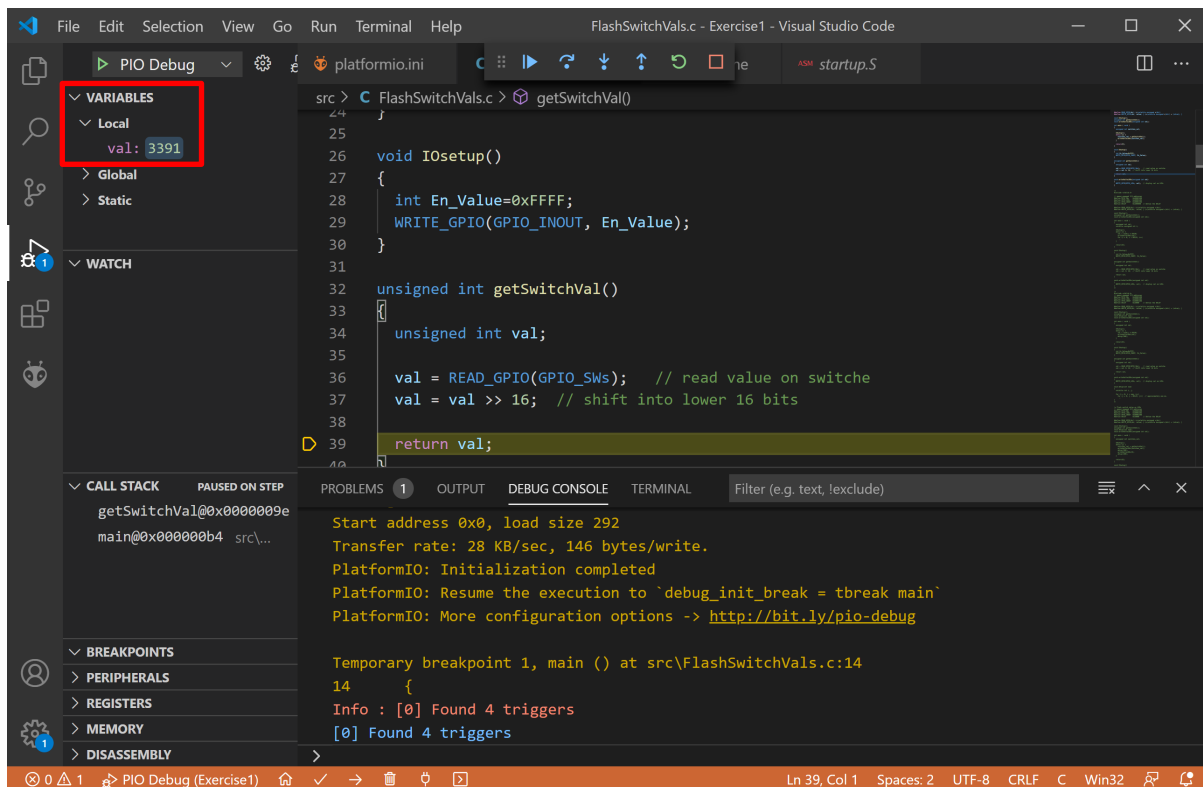


圖1. 單步進入getSwitchVal()函數

現在，通過按一下第19行的左側設定一個中斷點。該行左側將出現一個紅點，表明此時已設定中斷點，如圖2所示。

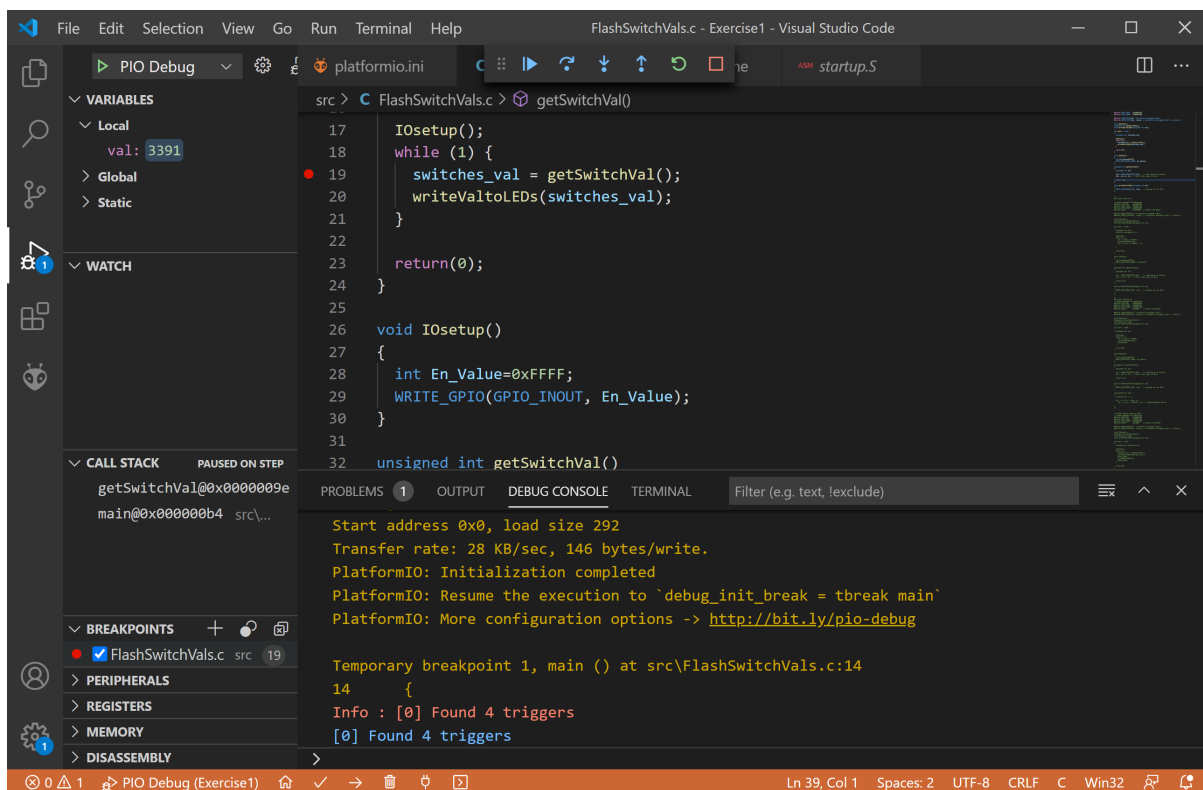




圖2. 設定中斷點

按下「Continue」（繼續）按鈕 （或F5）。程式將在到達第19行的中斷點後停止。此時，按下「Step Over」（單步跳過）按鈕 （或F10）。函數隨即執行，但偵錯工具不會進入函數，只會顯示函數的效果。需要注意的是，`switches_val`變數將變為開關的值，如圖3所示。

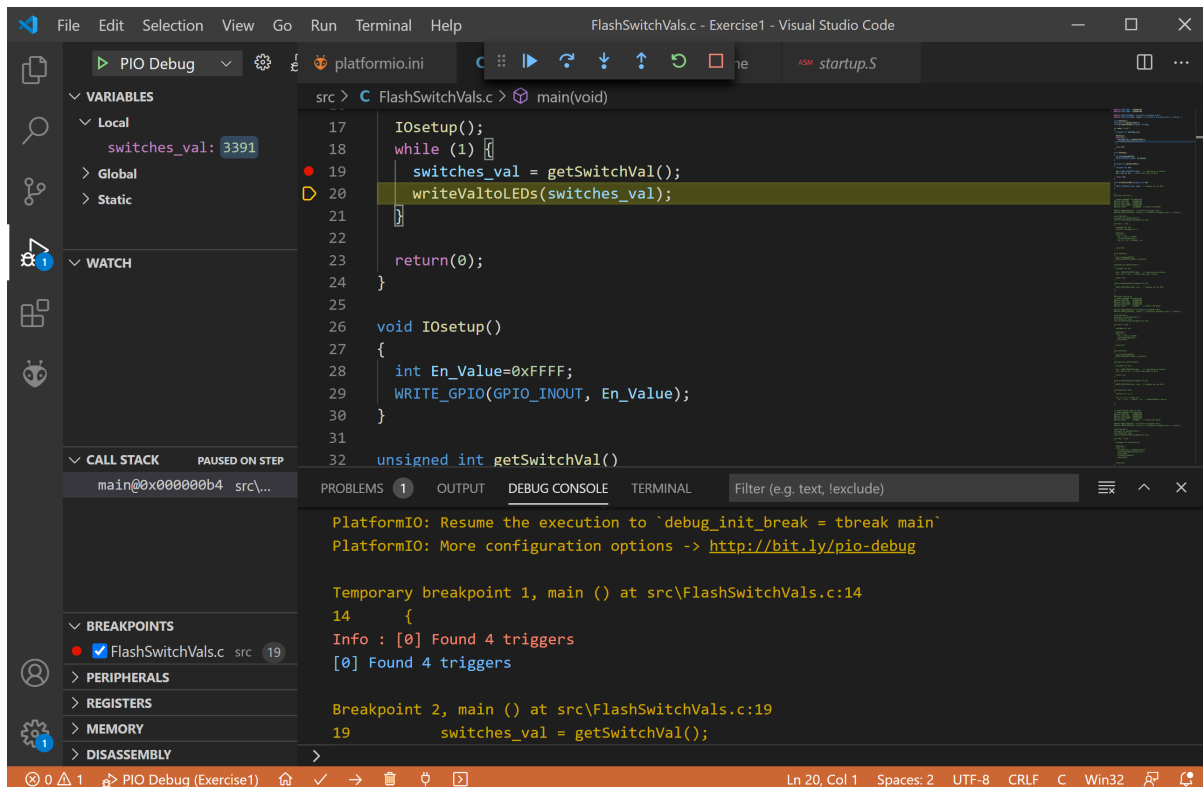


圖3. 單步跳過函數

3. 編寫呼叫函數庫函數的C程式

高階程式設計語言（如C語言）中包含程式設計師常用的函數庫。**google**搜尋「C標準函數庫」，可以找到一系列常用的C函數庫。通過包含標頭檔，在標頭檔中提供相應的函數陳述式，即可使用這些函數庫。具體方法是將以下行新增到C程式檔案的頂端：

```
#include <libraryname>
```

“libraryname”需替換為函數庫的名稱。舉例來說，數學函數庫（`math.h`）中提供多種常用函數，比如`fabs()`可計算浮點數的絕對值，`fmax()`可傳回兩個浮點數中的較大值。

另一個常用的函數庫是C標準函數庫（`stdlib.h`）。該庫中包含的某些函數可產生隨機數。例如，以下程式可在LED上顯示隨機數，具體方法是包含`stdlib.h`標頭檔（`#include <stdlib.h>`），然後呼叫`rand()`函數來傳回隨機數。將以下程式複製並貼到PlatformIO RVfpga專案中，然後在Nexys A7 FPGA開發板的RVfpgaNexys上執行該程式。

```
#include <stdlib.h>

// memory-mapped I/O addresses
#define GPIO_SWs      0x80001400
#define GPIO_LEDs     0x80001404
#define GPIO_INOUT    0x80001408
#define DELAY         0x1000000 // Define the DELAY

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

void IOsetup();
unsigned int getSwitchVal();
void writeValtoLEDs(unsigned int val);

int main(void)
{
    unsigned int val;
    volatile unsigned int i;

    IOsetup();
    while (1) {
        val = rand() % 65536;
        writeValtoLEDs(val);
        for (i = 0; i < DELAY; i++)
            ;
    }
    return(0);
}

void IOsetup() {
    int En_Value=0xFFFF;
    WRITE_GPIO(GPIO_INOUT, En_Value);
}

unsigned int getSwitchVal() {
    unsigned int val;

    val = READ_GPIO(GPIO_SWs); // read value on switches
    val = val >> 16; // shift into lower 16 bits

    return val;
}

void writeValtoLEDs(unsigned int val) {
    WRITE_GPIO(GPIO_LEDs, val); // display val on LEDs
}
```

下列檔案也提供該程式：

[RVfpgaPath]/RVfpga/Labs/Lab4/RandomNumberLEDs.c

除了這些C標準函數庫外，Western Digital (WD) 在其韌體套件 (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) 中還提供了專用於SweRV EH1處理器的函數庫 (PSP，可在系統中找到，位置為~/.platformio/packages/framework-wd-riscv-sdk/psp/) 和專用於Nexys A7開發板的函數庫 (BSP，可在系統中找到，位置為~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp/)。正如「入門指南」(第6.F節 - *HelloWorld_C-Lang*程式) 中所述，通過在platformio.ini中新增適當的行並在C程式的開頭包含適當的檔案，即可將這些函數庫包含在專案中。

這些函數庫提供了函數和巨集，程式設計師可借此來使用中斷、列印字串、讀/寫各個暫存器等。「RVfpga入門指南」和各實驗的範例和練習中會用到其中的很多函數。

4. RISC-V呼叫慣例

本節將介紹RISC-V呼叫慣例，其定義了如何將高階函數轉換為RISC-V組合語言。此呼叫慣例是應用程式二進位介面 (Application Binary Interface, ABI) 的一部分。通過定義慣例，可以跨程式使用由不同程式設計師編寫或包含在函數庫中的函數。在RISC-V中，**跳轉和連結指令** (jal) 會對函數進行叫用。例如，以下程式碼會呼叫函數func1：

```
jal func1
```

該指令可跳轉到標籤func1並將jal後面的指令的位址儲存到傳回位址暫存器中 (ra = x1)。函數隨後會使用傳回 (ret) 偽指令 (或跳轉暫存器指令jr ra) 跳轉到ra中儲存的位址，以此實作傳回。

可以使用輸入引數呼叫函數，也可以向呼叫函數傳回值。根據RISC-V慣例，輸入引數會傳遞給暫存器a0-a7中的函數。如果需要其他引數，則將其放入堆疊。根據RISC-V慣例，傳回值將放置在暫存器a0和a1中。使用暫存器傳遞引數和傳回值的相關協議由RISC-V呼叫慣例定義。

為了能從程式中的任何位置安全地呼叫函數，函數必須保留機器的架構狀態 (即程式設計師可以看到的暫存器內容)。假設有一個程式，其main函數的一個迴圈使用暫存器t0來儲存迴圈的索引。迴圈主體呼叫一個名為SortVector的函數，SortVector函數使用暫存器t0來儲存向量A的位址 (參見圖4)。因此，暫存器t0在SortVector函數中會被改寫，這會產生修改迴圈索引並導致其執行出錯的副作用。

```

main:
    add t2, zero, M
    add t0, zero, zero
    ...
loop1:
    bge t0, t2, endloop1
    ...
    jal SortVector
    ...
    add t0, t0, 1
    j loop1
endloop1:
    ...
    ret

SortVector:
    ...
    la t0, A
    ...
    ret

```

圖4. 主程式與函數使用暫存器時的衝突範例

如果main程式的程式設計師選擇了另一個暫存器來實作迴圈索引（例如，t1），顯然就不會發生這種衝突。但在呼叫函數之前，程式設計師很難瞭解函數實作時的一切內部細節（在某些情況下甚至不可行）。

對於各函數來說，更可行的解決方案是在所有將要修改的暫存器的記憶體中建立一個臨時副本，並在返回呼叫者程式前恢復暫存器的原始值。該解決方案需通過呼叫堆疊實作，呼叫堆疊是使用LIFO（後進先出）策略存取資料的儲存區域。該區域用於儲存與程式的即時函數（即已開始但尚未執行完畢的函數）相關的所有資訊，這些資訊從可用儲存空間的末端開始儲存（即從較高的位址向低位址儲存）。

函數通常分為三部分：

- ➔ 入口程式碼（序言）
- ➔ 函數主體
- ➔ 出口程式碼（尾聲）

序言必須建立函數的堆疊架構，並根據需要將暫存器內容儲存在堆疊中。堆疊架構是函數執行時使用的儲存區域。尾聲用於恢復呼叫者程式的架構狀態並釋放堆疊架構占用的儲存空間，使堆疊與執行序言前完全相同。

堆疊的存取由指標管理，該指標稱為堆疊指標（sp = x2），用於儲存堆疊中最後被占用的儲存位置的位址。程式開始前，必須使用堆疊的基本位址（即堆疊區域最高位址）對sp進行初始化。在RVfpga系統中，sp暫存器由_start函數初始化，該函數在~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/startup.S中實作。在初始化時，堆疊為空。第二個指標為架構指標（fp = x8），該指標指向活動函數堆疊架構的基本位址（即最高位址）。

函數使用**堆疊架構**作為私有儲存區域，只能由函數自身存取該區域。**堆疊架構**的一部分專門用於儲存要由函數修改的架構暫存器的副本，在某些情況下，堆疊架構還可以透過記憶體位置將參數傳遞給函數。

表1描述了RISC-V慣例為每個整數暫存器分配的預期角色。通過表1還可以看出，某些暫存器必須由呼叫的函數保留，而另一些暫存器則可能被函數改寫（即未保留）。

- 如果函數需要改寫任何保留的暫存器，必須首先在其**堆疊架構**中複製該暫存器，並在返回到**呼叫者**（即發起呼叫的函數）前恢復暫存器的值。除堆疊指標（sp）和傳回位址暫存器（ra）外，各呼叫之間還會保留**12**個整數暫存器（s0–s11），如果**被呼叫者**要使用這些暫存器，則必須儲存這些暫存器。
- 另一方面，**呼叫者**必須意識到某些暫存器無需由**被呼叫者**保留，因此這些暫存器可能會在呼叫後丟失。請注意，除參數和傳回值暫存器（a0–a7）外，還有**7**個整數暫存器（t0–t6）在各呼叫之間被用作臨時暫存器，這些暫存器為揮發性暫存器，如果**呼叫者**在函數呼叫後要再次使用這些暫存器，則必須儲存這些暫存器。

表1. RISC-V整數暫存器

名稱	暫存器編號	用途	保留
zero	x0	常數值0	-
ra	x1	返回位址	是
sp	x2	堆疊指標	是
gp	x3	全域指標	-
tp	x4	執行緒指標	-
t0–2	x5–7	臨時變數	否
s0/fp	x8	儲存暫存器/架構指標	是
s1	x9	儲存暫存器	是
a0–1	x10–11	函數引數/傳回值	否
a2–7	x12–17	函數引數	否
s2–11	x18–27	儲存暫存器	是
t3–6	x28–31	臨時變數	否

在圖4的範例中，根據該慣例將有兩種解決方案：

- main程式可使用一個暫存器作為迴圈索引（例如s0），而非使用t0，SortVector函數保證會保留該暫存器。
- main函數也可繼續使用t0，但在呼叫SortVector之前，必須將自身內容除存在堆疊中，並在從SortVector傳回後恢復原有內容。

函數執行時，由於堆疊架構需要更多儲存空間，堆疊會擴大，待函數執行完畢後，堆疊則會縮小。堆疊是向下（從高位址向低位址）儲存的，並且在進入程序後，堆疊指標應與**16**位元組邊界對齊。在標準ABI中，堆疊指標必須在整個程序執行期間保持對齊。

範例

下面舉例說明了如何使用C語言（圖5）和RISC-V組合語言（圖6）實作排序演算法。輸入為由N個元素組成的陣列A，每個元素都是大於0的整數。輸出為另一個陣列B，該陣列以降序儲存陣列A的元素。

在C程式中，main函數會呼叫SortVector函數，後者將接收陣列A和B的位址及大小（N），並將A的元素按降序逐個儲存到B中。SortVector函數則會呼叫另一個函數MaxVector，後者將接收陣列A的位址及大小、傳回陣列A的最大值並重設該值，從而使後續迭代中不再考慮該值。

```
#define N 8

int MaxVector(int A[], int size)
{
    int max=0, ind=0, j;
    for(j=0; j<size; j++){
        if(A[j]>max){
            max=A[j];
            ind=j;
        }
    }
    A[ind]=0;
    return(max);
}

int SortVector(int A[], int B[], int size)
{
    int max, j;
    for(j=0; j<size; j++){
        max=MaxVector(A, size);
        B[j]=max;
    }
    return(0);
}

int main ( void )
{
    int A[N]={7,3,25,4,75,2,1,1}, B[N];
    SortVector(A, B, N);
    return(0);
}
```

圖5. 用C語言實作的排序演算法

圖6說明以組合語言編寫的相同演算法。我們在分析該程式時會考慮前面各節所介紹的概念。

- main函數

○ 序言

- 首先，在堆疊中預留出空間，以儲存函數中使用的保留暫存器：
add sp, sp, -16。請注意，根據RISC-V慣例，sp暫存器必須始終與16位元組邊界保持對齊，以確保與RISC-V的128位元版本RV128I相容。
- 由於該函數未使用儲存的暫存器，暫存器s0-s11不需要儲存在堆疊中。但是，必須儲存暫存器ra，因為main函數會呼叫SortVector函數，而後者會更新ra中儲存的值。

- 函數主體
 - SortVector函數需透過指令jal SortVector呼叫。在呼叫該函數之前，根據呼叫慣例，應將三個輸入參數分別放入暫存器a0（陣列A的位址）、暫存器a1（陣列B的位址）和暫存器a2（陣列A和B的大小）。
- 尾聲
 - 在序言期間儲存在堆疊中的暫存器（ra）現已恢復。
 - 堆疊指標（sp）也需恢復為初始位置：add sp, sp, 16。
- **SortVector函數**
 - 序言
 - 首先，在堆疊中預留出空間，以儲存函數中使用的保留暫存器：add sp, sp, -32。
 - 隨後，函數所使用的儲存暫存器（s1-s3）將逐一儲存在堆疊中。
 - 同時必須儲存暫存器ra，因為SortVector會呼叫MaxVector函數，後者會改寫ra中儲存的值。
 - 函數主體
 - 首先，將輸入參數（a0、a1和a2）移入保留暫存器（s1、s2和s3），以便在執行函數MaxVector後使用這些參數。
 - 為計算向量B，需實作一個迴圈，以在每次迭代中計算A的最大值並將其儲存到B中。為計算A的最大值，需在迴圈的每次迭代中呼叫MaxVector函數：jal MaxVector。呼叫該函數之前，根據呼叫慣例，需將函數的輸入參數移入暫存器a0和a1。函數執行完畢後，將在暫存器a0中傳回A的最大值。
 - 請注意，迴圈主要使用儲存的暫存器來儲存變數。RISC-V呼叫慣例保證，這些暫存器在執行MaxVector後會保留自身的值（即函數必須保留自身的值）。
 - 函數可對暫存器a0和a1進行修改。因此，上述暫存器在每次呼叫前必須做好準備。
 - 從MaxVector傳回後，需要重複使用暫存器t1。因此，在呼叫函數（sw t1, 16(sp)）之前，必須將該暫存器保留在SortVector的堆疊中，並在執行結束後恢復（lw t1, 16(sp)）。
 - 尾聲
 - 在序言期間儲存在堆疊中的暫存器現已恢復。
 - 堆疊指標（sp）也需恢復為初始位置：add sp, sp, 32。
- **MaxVector函數**
 - 序言
 - 首先，在堆疊中預留出空間，以儲存函數中使用的保留暫存器：add sp, sp, -16。

- 然後，將函數所使用的儲存暫存器（即暫存器s1）儲存到堆疊中：sw s1, 0(sp)。請注意，如果該函數未儲存該暫存器，則*呼叫者*函數（SortVector）將無法執行，因為後者同樣使用該暫存器儲存向量A的位址。
- 由於呼叫者函數不會再呼叫其他函數（即屬於葉子函數），此時無需儲存ra。
- 函數主體
 - 該函數使用s1和一些臨時暫存器來計算陣列A的最大值。
- 尾聲
 - 在返回*呼叫者*函數之前，必須準備傳回值：mv a0, t2。
 - 在序言期間儲存在堆疊中的暫存器（s1）現已恢復。
 - 堆疊指標（sp）也需恢復為初始位置：add sp, sp, 16。

```
.globl main

.equ N, 8

.data
A: .word 7,3,25,4,75,2,1,1

.bss
B: .space 4*N

.text

MaxVector:
    add sp, sp, -16
    sw s1, 0(sp)

    mv s1, zero
    mv t2, zero
loop2:
    beq s1, a1, endloop2
    lw t1, (a0)
    ble t1, t2, else2
    mv t2, t1
    mv t3, a0
else2:
    add a0, a0, 4
    add s1, s1, 1
    j loop2
endloop2:
    sw zero, (t3)

    mv a0, t2
    lw s1, 0(sp)
    add sp, sp, 16
    ret

SortVector:
    add sp, sp, -32
    sw s1, 0(sp)
    sw s2, 4(sp)
    sw s3, 8(sp)
    sw ra, 12(sp)

    mv s1, a0                # Address of vector A
    mv s2, a1                # Address of vector B
    mv s3, a2                # Size of vectors A and B
```

```

mv t1, zero

loop1:
    beq t1, s3, endloop1
    mv a0, s1
    mv a1, s3
    sw t1, 16(sp)
    jal MaxVector
    lw t1, 16(sp)
    sw a0, (s2)
    add s2, s2, 4
    add t1, t1, 1
    j loop1
endloop1:

lw s1, 0(sp)
lw s2, 4(sp)
lw s3, 8(sp)
lw ra, 12(sp)
add sp, sp, 32
ret

main:
    add sp, sp, -16
    sw ra, 0(sp)

    la a0, A
    la a1, B
    add a2, zero, N
    jal SortVector

    lw ra, 0(sp)
    add sp, sp, 16
    ret

.end

```

圖6. 用組合語言實作的排序演算法

圖7 展示了執行MaxVector函數的主體時堆疊的狀態。

- 藍色部分為main函數的堆疊架構，其中包含該函數的傳回位址（ra）。
- 綠色部分為SortVector函數的堆疊架構，其中包含該函數所使用的儲存暫存器（s1-s3）、暫存器t1以及暫存器ra。
- 最後，黃色部分為MaxVector函數的堆疊架構，該堆疊架構為活動堆疊架構（正在執行的函數的堆疊架構），其中包含該函數所使用的儲存暫存器（s1）。

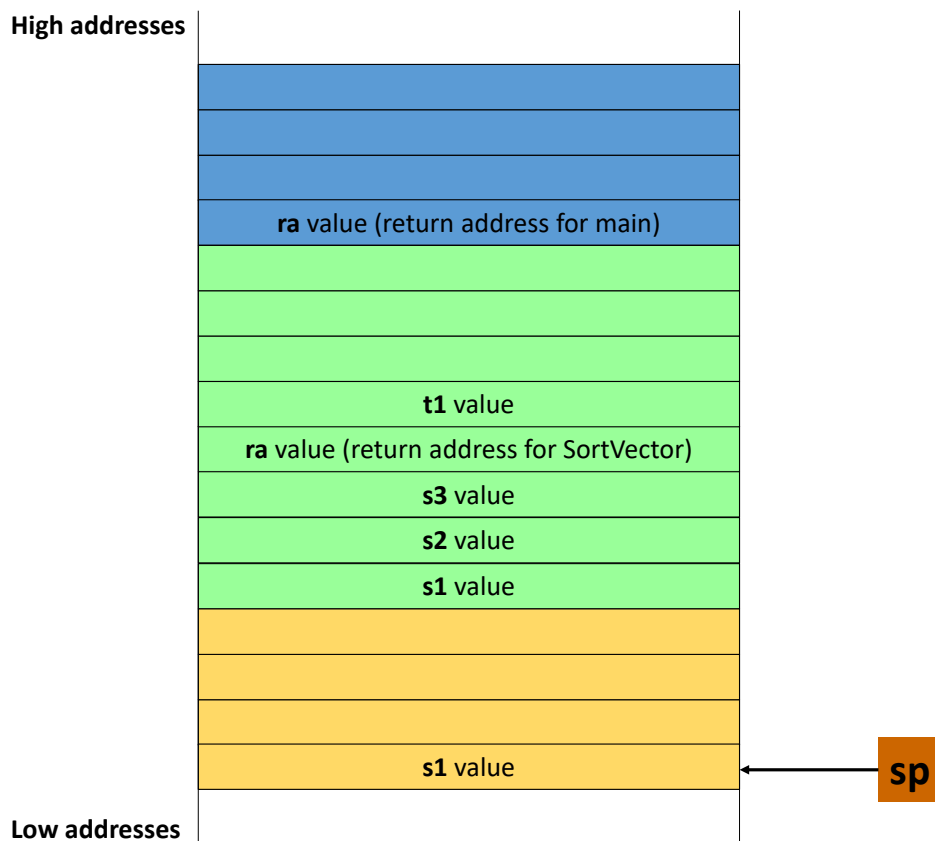


圖7. 圖6中組合語言程式的**MaxVector**函數主體的堆疊狀態。

任務：圖6中的組合語言程式在PlatformIO專案中提供，檔案位置為：

[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions。使用逐步除錯工具選項在開發板上（或在ISS模擬器上）執行該程式，以分析儲存在各暫存器（s、ra、a等）中的值，以及根據RISC-V呼叫慣例儲存在堆疊中的值。

- 程式完成編譯後，可檢視PlatformIO所產生的**.pio/build/swervolf_nexys/firmware.dis**檔來瞭解程式中各指令的位址。

- 可使用記憶體控制台分析堆疊的變化以及陣列A和B的內容。

- 在本專案中，我們使用調整後的**link.ld**指令碼，**sp**暫存器在其中被強制與16位元組邊界對齊。該指令碼位於以下位置：

[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions/ld/link.ld。sp暫存器使用ALIGN()命令強制對齊：

```
.stack :
{
    _heap_end = .;
    . = . + __stack_size;
    /* Force 16-B alignment of SP register */
    . = ALIGN(16);
    _sp = .;
} > ram : ram_load
```

5. 練習

現在，可通過以下練習建立包含函數呼叫的C程式/組合語言程式。

請注意，如果使Nexys A7開發板與電腦保持連接並維持通電狀態，則在切換不同程式時，無需將RVfpgaNexys重新重新載入到開發板上。但是，如果關閉Nexys A7開發板，則需要使用PlatformIO將RVfpgaNexys重新載入到開發板上。

還要記住，可以使用Verilator或Whisper模擬這些程式。

練習1. 編寫一個C程式，在LED上顯示開關值反轉後的值。將程式命名為**DisplayInverse_Functions.c**。

例如，如果開關的值（二進位）為：0101010101010101，則LED應顯示：1010101010101010；如果開關的值為：1111000011110000，則LED應顯示：0000111100001111；依此類推。包含一個getSwitchesInvert()函數，該函數會傳回開關值反轉後的值。函數宣告為：

```
unsigned int getSwitchesInvert();
```

練習2. 編寫一個C程式，使開關的值在LED上閃爍。將程式命名為**FlashSwitchesToLEDs_Functions.c**。

開關的值應以約2秒的週期進行脈衝開關。包含一個名為delay()的函數，該函數會產生num毫秒的延遲。可根據經驗進行設定，週期無需恰好為2秒。函數宣告如下所示：

```
void delay(int num);
```

練習3. 編寫一個測量回應時間的C程式。程式應能測量所有LED點亮後，操作員接通最右側的開關（SW[0]）所用的時間。需使用stdlib.h庫中的rand()函數產生一個隨機的時間量，以在使用者每次嘗試測試回應時間時進行延遲。將程式命名為**ReactionTime.c**。

該程式應按如下方式工作。

該程式應按如下方式工作。

1. 使用者斷開最右側的開關（向下撥），指示準備開始。
2. 該程式將關閉所有LED，然後等待一段隨機的時間（不超過3秒）。可使用練習2中的delay()函數進行延遲。
3. 所有LED隨後都會亮起，程式開始計算使用者接通最右側的開關所經過的時間（ms）。
4. 當使用者接通最右側的開關（SW[0]）時，LED上將以二進位形式顯示接通開關（向上撥）所經過的時間（ms），序列控制台上則以十進位形式顯示該時間。
5. 如果使用者斷開（向下撥）最右側的開關，此遊戲將重複進行。

練習4. `rand()`函數的一個問題是該函數使用可預測的隨機數序列。也就是說，每次執行該程式時，都會以同樣的隨機數開頭，然後遵循同樣的隨機數序列。在練習3中多次執行該程式，就可以證實這一點。

但是，如果先使用`srand()`函數，即可使`rand()`函數採用隨機的起始值。唯一的問題是，必須給`srand()`一個輸入引數，一個隨機的不帶正負號整數。例如，可將使用者斷開開關以開始遊戲所用的時間，作為隨機數賦值給`srand()`。

請重新編寫練習3的程式，以便在LED點亮之前產生一個真正隨機的時間序列。盡可能使用函數。將程式命名為**ReactionTimeTrulyRandom.c**。

練習5. 重新編寫練習4的程式，使點亮的LED的長度與回應時間成正比。這樣，可以更容易地判斷回應時間是否變快，無需解讀毫秒數的二進位表示形式。可使不同的LED點亮範圍與不同的回應時間範圍相對應。例如，回應時間較快時，僅應點亮右側的少數LED。當回應時間逐漸增加，左側應有越來越多的LED點亮。如果回應時間非常慢，則將點亮所有LED。將程式命名為**ReactionTimeBar.c**。

練習6. 編寫一個C程式以實作「Simon says」遊戲。應產生以下效果：

1. 程式會使最右側的三個LED以某種模式閃爍，然後等待使用者使用最右側的三個開關重複對應的序列。開關[2:0]對應於LED[2:0]，其中LED[0]是最右側的LED，而開關[0]是最右側的開關。
2. 隨機模式應首先點亮1個LED，然後點亮2個LED，再點亮3個，依此類推。
3. 然後，使用者嘗試使用最右側的三個開關重複該序列。當使用者將開關向上撥時，相應的LED應點亮（當使用者將開關向下撥時，相應的LED熄滅）。
4. 如果使用者在暫停後輸入正確的序列，則應顯示下一個模式，序列中會多出一個LED。
5. 如果使用者輸入的序列錯誤，則LED將保持點亮狀態，不會出現新的序列。
6. 將最左側的開關（開關[15]）先向上撥（接通）再向下撥（斷開），可以重設遊戲。

選用函數以實作模組化的程式，使其更易於編寫、偵錯和理解。不要忘記根據需要使用標準C函數庫來編寫程式。將程式命名為**SimonSays.c**。

練習7. 給定一個包含 $3*N$ 個元素的向量A，需要得出一個包含N個元素的向量B，使B的每個元素都是A中三個連續元素之和的絕對值。例如：

$$B[0] = |A[0] + A[1] + A[2]|, \quad B[1] = |A[3] + A[4] + A[5]|, \quad \dots$$

編寫一個名為**Triplets.S**的RISC-V組合語言程式（該程式必須符合RISC-V呼叫慣例）：

- main程式根據以下高階虛擬程式碼實作B的計算：

```
#define N 4

int A[3*N] = {a list of 3*N values};
int B[N];
int i, j=0;

void main (void)
{
    for (i=0; i<N; i++){
        B[i] = res_triplet(A,j);
        j=j+3;
    }
}
```

- 函數res_triplet會從位置p開始，傳回向量V中每3個連續元素之和的絕對值。該函數是根據以下高階虛擬程式碼指定的規格實作的：

```
int res_triplet(int V[ ], int pos)
{
    int i, sum=0;
    for (i=0; i<3; i++){
        sum = sum + V[pos+i];
    }
    sum=abs(sum);
    return sum;
}
```

- 函數abs(int x)會傳回其輸入引數的絕對值。

練習8. 編寫一個名為**Filter.S**的RISC-V組合語言程式（該程式必須符合之前學習過的函數管理標準）。可以使用以下虛擬程式碼：

```
#define N 6
int i, j=0, A[N]={48,64,56,80,96,48}, B[N];
for (i=0; i<(N-1); i++){
    if( (myFilter(A[i],A[i+1])) == 1){
        B[j]=A[i]+ A[i+1] + 2;
        j++;
    }
}
```

- 編寫等效的RISC-V組合語言程式碼，其中應包含預留儲存空間所需的任何指令，並宣告相應的部分（.data、.bss和.text）。如果第一個引數是16的倍數，而第二個引數大於第一個引數，函數myFilter會傳回值1；其他情況下則傳回0。
- 編寫myFilter函數的組合語言程式碼。

練習9. 需要構建一個名為**Coprimes.S**的RISC-V組合語言程式（該程式必須符合之前學習過的函數管理標準），使其可在給定的一系列整數對（>0）中找出由互質數組成的整數對。眾所周知，如果兩個數的公約數只有1，則稱為互質數。

假定輸入資料位於以下形式的陣列D中：

$$D = (x_0, y_0, c_0, x_1, y_1, c_1, \dots, x_{N-1}, y_{N-1}, c_{N-1})$$

每三個元素為一組（ x_i 、 y_i 、 c_i ），含義如下： x_i 和 y_i 表示一對整數， c_i 的初始值為0。執行程式後， c_i 的值必須按以下方式修改：如果 x_i 和 y_i 互質，則 $c_i = 2$ ；其他情況下 $c_i = 1$ 。

例如：

對於以下輸入向量：D = (3,5,0, 6,18,0, 15,45,0, 13,10,0, 24,3,0, 24,35,0)

最終結果應為：D = (3,5,2, 6,18,1, 15,45,1, 13,10,2, 24,3,1, 24,35,2)

- 編寫一個RISC-V組合語言程式，該程式應遍歷陣列D並根據下方左方塊中給定的規格產生結果。該程式將呼叫函數check_coprime (int D [], int i)，其輸入引數為D的起始位址和待檢查的整數對的編號（從0到M-1）。該函數會檢查陣列D中的第i對整數是否為互質數，並將結果儲存到相應的記憶體位置。
- 根據下方右方塊中給定的規格編寫函數check_coprime的程式碼。請記住，函數gcd(int a, int b)是根據歐幾里得演算法在實驗3中實作的，該函數會傳回兩個輸入引數的最大公約數（greatest common divisor，gcd）。如果gcd為1，則兩個數為互質數。

<pre>#define M 6 int D[] = {a list of M*3 int values} void main () { int i; for (i=0; i<M; i++) check_coprime(D,i); }</pre>	<pre>void check_coprime (int A[], int pos) { int res; res = gcd(A[3*pos], A[(3*pos)+1]); if (res == 1) A[(3*pos)+2] = 2; else A[(3*pos)+2] = 1; }</pre>
---	--