



Imagination大学计划

RVfpga实验4

函数调用

1. 简介

函数调用对于任何程序都至关重要，因为可以实现模块化与代码重用，从而简化代码的编写和调试。**C**编程语言还包含常用**C**函数（例如随机数生成器和常用数学函数）的标准库以及处理器/开发板专用库。高级函数按照 *调用约定* 转换为汇编语言。本实验将展示如何在**C**程序中编写和使用函数（程序员编写的两个函数以及**C**库中包含的函数），以及如何用汇编语言实现这些函数。在实验的末尾，我们提供了关于编写使用函数和调用库的程序的练习。

2. 编写使用函数的C程序

函数也称为子例程或过程，是指封装到具有指定操作和接口的代码块中的一段代码。这种模块化结构可降低复杂性并支持代码重用，从而提高效率。函数可以从程序中的任何位置调用，而且调用的函数执行完毕后可立即恢复程序执行。函数可由其他函数（称为*嵌套函数*）甚至同一函数（称为*递归调用*）调用。

要编写使用函数的**RISC-V**程序，请按照实验2和实验3中所述的通用步骤进行操作：

1. 创建RVfpga项目
2. 编写C程序
3. 将RVfpgaNexys下载到Nexys A7 FPGA开发板（请记住，也可以使用Verilator或Whisper仿真这些程序）
4. 编译、下载和运行/调试程序

有关这些步骤的详细说明，请参见实验2。下面简要说明了各项步骤。

步骤1. 创建RVfpga项目

在以下文件夹中创建名为project1的项目：

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

步骤2. 编写C程序

现在，需要将**C**程序添加到项目中。创建一个新文件，然后在项目中输入或复制/粘贴以下**C**程序。以下文件中也提供该程序：

```
[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
}

```


将文件保存到项目的src目录下，并将文件命名为LedsSwitches_Functions.c。

步骤3. 将RVfpgaNexys下载到Nexys A7 FPGA开发板

遵循RVfpga实验2和实验3中的说明，将RVfpgaNexys下载到Nexys A7开发板。

步骤4. 编译、下载和运行程序

现在即可在RVfpgaNexys上编译、下载和运行/调试程序。

按下“Run”（运行）和“Start Debugging”（开始调试）按钮，然后单击两次“Step Over”（单步跳过）按钮（位于顶部工具栏中）或按两次F10，直到到达调用getSwitchVal()函数的第19行。然后按下“Step Into”（单步进入）按钮（或F11），进入getSwitchVal()函数。如果无法查看val变量，请展开左侧工具栏中的“VARIABLES”（变量）→“Local”（本地）字段。此时，val变量在程序中可能会被列为“optimized out”（优化输出）。执行一次单步（单步跳过或单步进入）操作即可看到val变量变为开关的值，如图1所示。

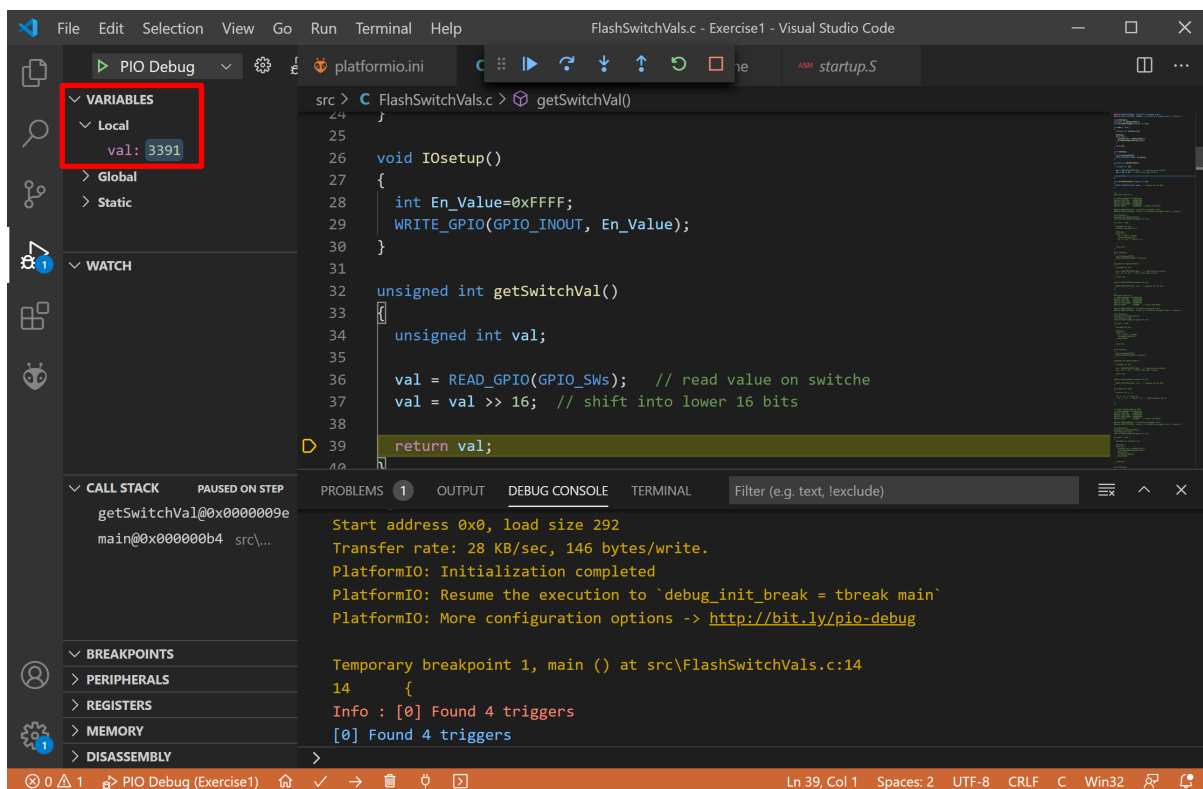


图1. 单步进入getSwitchVal()函数

现在，通过单击第19行的左侧设置一个断点。该行左侧将出现一个红点，表明此时已设置断点，如图2所示。

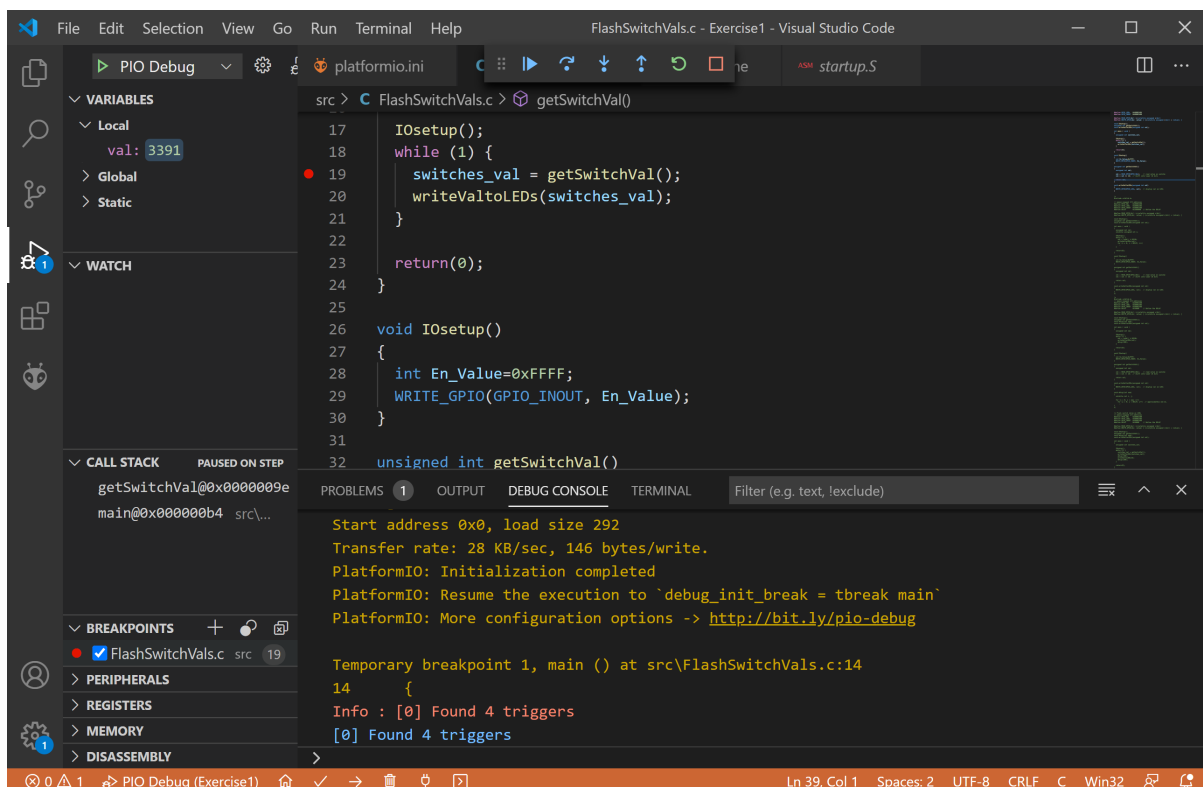




图2. 设置断点

按下“Continue”（继续）按钮 （或F5）。程序将在到达第19行的断点后停止。此时，按下“Step Over”（单步跳过）按钮 （或F10）。函数随即执行，但调试器不会进入函数，只会显示函数的效果。需要注意的是，`switches_val`变量将变为开关的值，如图3所示。

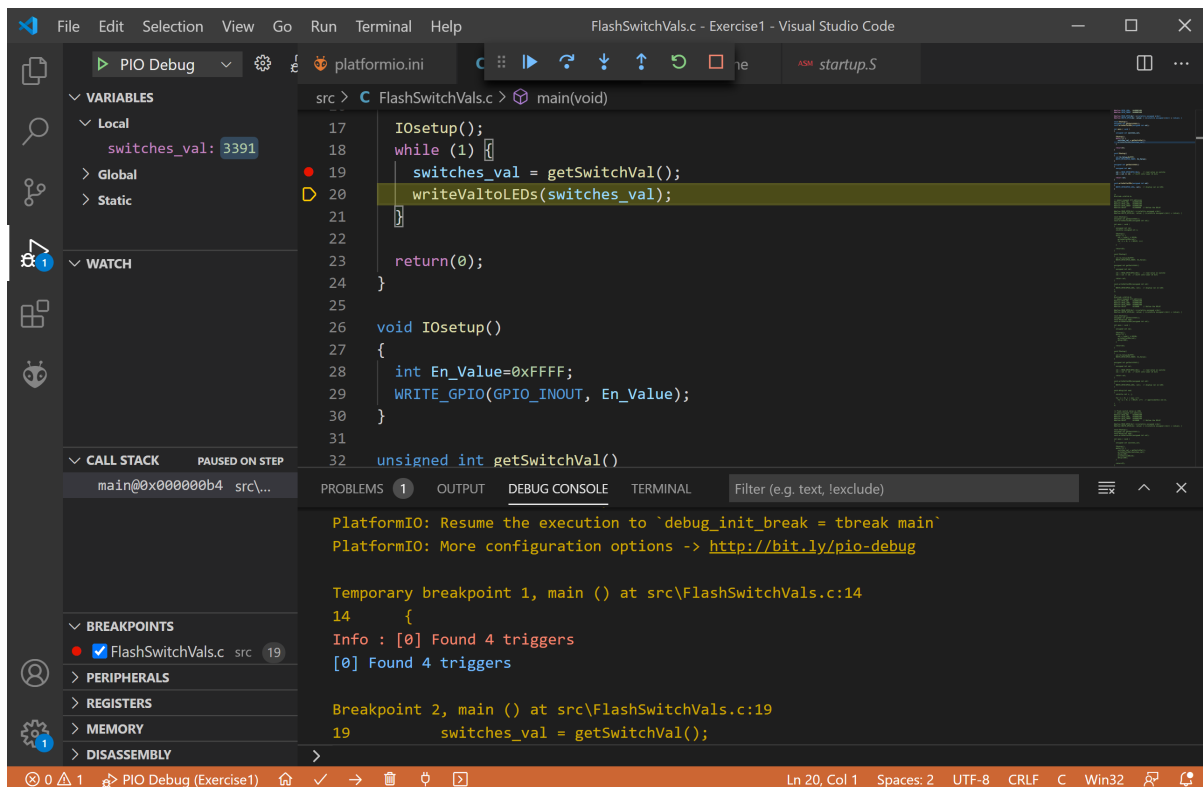


图3. 单步跳过函数

3. 编写调用库函数的C程序

高级编程语言（如C语言）中包含程序员常用的函数库。google搜索“C标准库”，可以找到一系列常用的C库。通过包含头文件，在头文件中给出相应的函数声明，即可使用这些函数库。具体方法是以下行添加到C程序文件的顶部：

```
#include <libraryname>
```

“libraryname”需替换为库的名称。举例来说，数学库（`math.h`）中提供多种常用函数，比如`fabs()`可计算浮点数的绝对值，`fmax()`可返回两个浮点数中的较大值。

另一个常用的库是C标准库（`stdlib.h`）。该库中包含的某些函数可生成随机数。例如，以下程序可在LED上显示随机数，具体方法是包含`stdlib.h`头文件（`#include <stdlib.h>`），然后调用`rand()`函数来返回随机数。将以下程序复制并粘贴到PlatformIO RVfpga项目中，然后在Nexys A7 FPGA开发板上的RVfpgaNexys上运行该程序。

```
#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
}
```

以下文件中也提供该程序：

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

除了这些C标准库外，Western Digital（WD）在其固件包（<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>）中还提供了专用于SweRV EH1处理器的库（PSP，可在系统中找到，位置为~/.platformio/packages/framework-wd-riscv-sdk/psp/）和专用于Nexys A7开发板的库（BSP，可在系统中找到，位置为~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp/）。正如“入门指南”（第6.F节 - *HelloWorld_C-Lang*程序）中所述，通过在platformio.ini中添加适当的行并在C程序的开头包含适当的文件，即可将这些库包含在项目中。

这些库中提供了函数和宏，程序员可借此来使用中断、打印字符串、读/写各个寄存器等。“RVfpga入门指南”和各实验的示例和练习中会用到其中的很多函数。

4. RISC-V调用约定

本部分将介绍RISC-V调用约定，其定义了如何将高级函数转换为RISC-V汇编语言。此调用约定是应用程序二进制接口（Application Binary Interface, ABI）的一部分。通过定义约定，可以跨程序使用由不同程序员编写或包含在库中的函数。在RISC-V中，**跳转和链接指令**（jal）会发起对函数的调用。例如，以下代码会调用函数func1：

```
jal func1
```

该指令可跳转到标签func1并将jal后面的指令的地址保存到返回地址寄存器中（ra = x1）。函数随后会使用返回（ret）伪指令（或跳转寄存器指令jr ra）跳转到ra中存储的地址，以此实现返回。

可以使用输入参数调用函数，也可以向调用函数返回值。根据RISC-V约定，输入参数会传递给寄存器a0-a7中的函数。如果需要其他参数，则将其放入堆栈。根据RISC-V约定，返回值将放置在寄存器a0和a1中。使用寄存器**传递参数和返回值的相关协议**由RISC-V调用约定定义。

为了能从程序中的任何位置安全地调用函数，函数必须保留机器的架构状态（即程序员可以看到的寄存器内容）。假设有一个程序，其main函数的一个循环使用寄存器t0来存储循环的索引。循环主体调用一个名为SortVector的函数，SortVector函数使用寄存器t0来存储向量A的地址（参见图4）。因此，寄存器t0在SortVector函数中会被改写，这会产生修改循环索引并导致其执行出错的副作用。

```

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

```

图4. 主程序与函数使用寄存器时的冲突示例

如果main程序的程序员选择了另一个寄存器来实现循环索引（例如，t1），显然就不会发生这种冲突。但在调用函数之前，程序员很难了解函数实现时的一切内部细节（在某些情况下甚至不可行）。

对于各函数来说，更切实际的解决方案是在所有将要修改的寄存器的存储器中创建一个临时副本，并在返回调用者程序前恢复寄存器的原始值。该解决方案需通过调用堆栈实现，调用堆栈是使用LIFO（后进先出）策略存取数据的存储区域。该区域用于存储与程序的实时函数（即已开始但尚未执行完毕的函数）相关的所有信息，这些信息从可用存储空间的末端开始存储（即从高地址向低地址存储）。

函数通常分为三部分：

- ➔ 入口代码（序言）
- ➔ 函数主体
- ➔ 出口代码（尾声）

序言必须创建函数的堆栈帧，并根据需要将寄存器内容存储在堆栈上。堆栈帧是函数执行时使用的存储区域。尾声用于恢复调用者程序的架构状态并释放堆栈帧占用的存储空间，使堆栈与执行序言前完全相同。

堆栈的访问由指针管理，该指针称为堆栈指针（ $sp = x2$ ），用于存储堆栈中最后被占用的存储单元的地址。程序开始前，必须使用堆栈的基址（即栈区最高地址）对 sp 进行初始化。在RVfpga系统中， sp 寄存器由 $_start$ 函数初始化，该函数在 $\sim/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/startup.S$ 中实现。在初始化时，堆栈为空。第二个指针为帧指针（ $fp = x8$ ），该指针指向活动函数堆栈帧的基址（即最高地址）。

函数使用**堆栈帧**作为私有存储区域，只能由函数自身访问该区域。**堆栈帧**的一部分专门用于保存要由函数修改的架构寄存器的副本，在某些情况下，堆栈帧还可以用作通过存储单元将参数传递给函数的一种方式。

表1描述了RISC-V约定为每个整数寄存器分配的预期角色。通过表1还可以看出，某些寄存器必须由调用的函数保留，而另一些寄存器则可能被函数改写（即未保留）。

- 如果函数需要改写任何保留的寄存器，必须首先在其**堆栈帧**中复制该寄存器，并在返回到**调用者**（即发起调用的函数）前恢复寄存器的值。除堆栈指针（**sp**）和返回地址寄存器（**ra**）外，各调用之间还会保留**12**个整数寄存器（**s0-s11**），如果**被调用者**要使用这些寄存器，则必须保存这些寄存器。
- 另一方面，**调用者**必须意识到某些寄存器无需由**被调用者**保留，因此这些寄存器可能会在调用后丢失。请注意，除参数和返回值寄存器（**a0-a7**）外，还有**7**个整数寄存器（**t0-t6**）在各调用之间被用作临时寄存器，这些寄存器为易失性寄存器，如果**调用者**在函数调用后要再次使用这些寄存器，则必须保存这些寄存器。

表1. RISC-V整数寄存器

名称	寄存器编号	用途	保留
zero	x0	常数值0	-
ra	x1	返回地址	是
sp	x2	堆栈指针	是
gp	x3	全局指针	-
tp	x4	线程指针	-
t0-2	x5-7	临时变量	否
s0/fp	x8	保存寄存器/帧指针	是
s1	x9	保存寄存器	是
a0-1	x10-11	函数参数/返回值	否
a2-7	x12-17	函数参数	否
s2-11	x18-27	保存寄存器	是
t3-6	x28-31	临时变量	否

在图4的示例中，根据该约定将有两种解决方案：

- main**程序可使用一个寄存器作为循环索引（例如**s0**），而非使用**t0**，**SortVector**函数保证会保留该寄存器。
- main**函数也可继续使用**t0**，但在调用**SortVector**之前，必须将自身内容保存在堆栈中，并在从**SortVector**返回后恢复原有内容。

函数执行时，由于堆栈帧需要更多存储空间，堆栈会扩大，待函数执行完毕后，堆栈则会缩小。堆栈是向下（从高地址向低地址）存储的，并且在进入过程后，堆栈指针应与**16**字节边界对齐。在标准**ABI**中，堆栈指针必须在整个过程执行期间保持对齐。

示例

下面举例说明了如何使用C语言（图5）和RISC-V汇编语言（图6）实现排序算法。输入为由N个元素组成的数组A，每个元素都是大于0的整数。输出为另一个数组B，该数组以降序存储数组A的元素。

在C程序中，main函数会调用SortVector函数，后者将接收数组A和B的地址及大小（N），并将A的元素按降序逐个存储到B中。SortVector函数则会调用另一个函数MaxVector，后者将接收数组A的地址及大小、返回数组A的最大值并复位该值，从而使后续迭代中不再考虑该值。

```
#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);
}
```

图5. 用C语言实现的排序算法

图6所示为用汇编语言编写的相同算法。我们在分析该程序时会考虑前面部分中介绍的概念。

- main函数

o 前言

- 首先，在堆栈中预留出空间，以存储函数中使用的保留寄存器：add sp, sp, -16。请注意，根据RISC-V约定，sp寄存器必须始终与16字节边界保持对齐，以确保与RISC-V的128位版本RV128I兼容。
- 由于该函数未使用保存的寄存器，寄存器s0-s11不需要存储在堆栈中。但是，必须保存寄存器ra，因为main函数会调用SortVector函数，而后者会更新ra中存储的值。

- 函数主体

- SortVector函数需通过指令jal SortVector调用。在调用该函数之前，根据调用约定，应将三个输入参数分别放入寄存器a0（数组A的地址）、寄存器a1（数组B的地址）和寄存器a2（数组A和B的大小）。

- 结语

- 在序言期间保存在堆栈中的寄存器（ra）现已恢复。
- 堆栈指针（sp）也需恢复为初始位置：add sp, sp, 16。

- SortVector函数

- 前言

- 首先，在堆栈中预留出空间，以存储函数中使用的保留寄存器：add sp, sp, -32。
- 随后，函数所使用的保存寄存器（s1-s3）将逐一存储在堆栈中。
- 同时必须保存寄存器ra，因为SortVector会调用MaxVector函数，后者会改写ra中存储的值。

- 函数主体

- 首先，将输入参数（a0、a1和a2）移入保留寄存器（s1、s2和s3），以便在执行函数MaxVector后使用这些参数。
- 为计算向量B，需实现一个循环，以在每次迭代中计算A的最大值并将其存储到B中。为计算A的最大值，需在循环的每次迭代中调用MaxVector函数：jal MaxVector。调用该函数之前，根据调用约定，需将函数的输入参数移入寄存器a0和a1。函数执行完毕后，将在寄存器a0中返回A的最大值。
- 请注意，循环主要使用保存的寄存器来存储变量。RISC-V调用约定保证，这些寄存器在执行MaxVector后会保留自身的值（即函数必须保留自身的值）。
- 函数可对寄存器a0和a1进行修改。因此，上述寄存器在每次调用前必须做好准备。
- 从MaxVector返回后，需要重用寄存器t1。因此，在调用函数（sw t1, 16(sp)）之前，必须将该寄存器保留在SortVector的堆栈中，并在执行结束后恢复（lw t1, 16(sp)）。

- 结语

- 在序言期间保存在堆栈中的寄存器现已恢复。
- 堆栈指针（sp）也需恢复为初始位置：add sp, sp, 32。

- MaxVector函数

- 前言

- 首先，在堆栈中预留出空间，以存储函数中使用的保留寄存器：add sp, sp, -16。

- 然后，将函数所使用的保存寄存器（即寄存器s1）存储到堆栈中：sw s1, 0(sp)。请注意，如果该函数未保存该寄存器，则调用者函数（SortVector）将无法执行，因为后者同样使用该寄存器存储向量A的地址。
- 由于调用者函数不会再调用其他函数（即属于叶子函数），此时无需保存ra。
- 函数主体
 - 该函数使用s1和一些临时寄存器来计算数组A的最大值。
- 结语
 - 在返回调用者函数之前，必须准备返回值：mv a0, t2。
 - 在序言期间保存在堆栈中的寄存器（s1）现已恢复。
 - 堆栈指针（sp）也需恢复为初始位置：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

```

图6. 用汇编语言实现的排序算法

图7展示了执行MaxVector函数的主体时堆栈的状态。

- 蓝色部分为main函数的堆栈帧，其中包含该函数的返回地址（ra）。
- 绿色部分为SortVector函数的堆栈帧，其中包含该函数所使用的保存寄存器（s1-s3）、寄存器t1以及寄存器ra。
- 最后，黄色部分为MaxVector函数的堆栈帧，该堆栈帧为活动堆栈帧（正在执行的函数的堆栈帧），其中包含该函数所使用的保存寄存器（s1）。

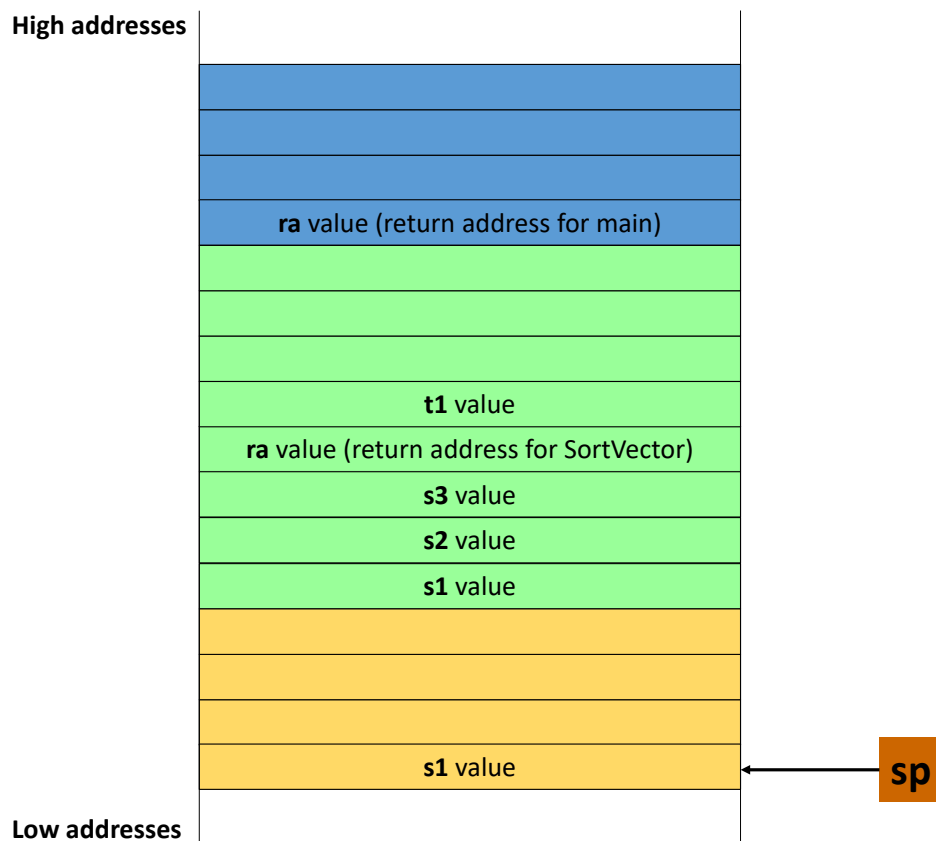


图7. 图6中汇编程序的MaxVector函数主体的堆栈状态。

任务：图6中的汇编程序在Platformio项目中提供，文件位置为：

[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions.使用逐步调试器选项在开发板上（或在ISS仿真器上）执行该程序，以分析存储在各寄存器（s、ra、a等）中的值以及根据RISC-V调用约定存储在堆栈中的值。

- 程序完成编译后，可查看PlatformIO所生成的**.pio/build/swervolf_nexys/firmware.dis**文件来了解程序中各指令的地址。

- 可使用存储器控制台分析堆栈的变化以及数组A和B的内容。

- 在本项目中，我们使用调整后的**link.lds**脚本，**sp**寄存器在其中被强制与16字节边界对齐。

该脚本位于以下位置：

[RVfpgaPath]/RVfpga/Labs/Lab4/SortingAlgorithm_Functions/ld/link.lds。sp寄存器使用

ALIGN()命令强制对齐：

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

5. 练习

现在，可通过以下练习创建包含函数调用的C程序/汇编程序。

请记住，如果使Nexys A7开发板与计算机保持连接并保持上电状态，则在切换运行不同程序时，无需将RVfpgaNexys重新加载到开发板上。但是，如果关闭Nexys A7开发板，则需要使用PlatformIO将RVfpgaNexys重新加载到开发板上。

还要记住，可以使用Verilator或Whisper仿真这些程序。

练习1. 编写一个C程序，在LED上显示开关值取反后的值。将程序命名为**DisplayInverse_Functions.c**。

例如，如果开关的值（二进制）为：0101010101010101，则LED应显示：1010101010101010；如果开关的值为：1111000011110000，则LED应显示：0000111100001111；依此类推。包含一个getSwitchesInvert()函数，该函数会返回开关值取反后的值。函数声明为：

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

练习2. 编写一个C程序，使开关的值在LED上闪烁。将程序命名为**FlashSwitchesToLEDs_Functions.c**。

开关的值应以约2秒的周期进行脉冲开关。包含一个名为delay()的函数，该函数会产生num毫秒的延时。可根据经验进行设置，周期无需恰好为2秒。函数声明如下所示：

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

练习3. 编写一个测量响应时间的C程序。程序应能测量所有LED点亮后，操作员接通最右侧的开关（SW[0]）所用的时间。需使用stdlib.h库中的rand()函数生成一个随机的时间量，以在用户每次尝试测试响应时间时进行延迟。将程序命名为**ReactionTime.c**。

该程序应按如下方式工作。

该程序应按如下方式工作。

1. 用户断开最右侧的开关（向下拨），指示准备开始。
2. 该程序将关闭所有LED，然后等待一段随机的时间（不超过3秒）。可使用练习2中的delay()函数进行延迟。
3. 然后所有LED都将亮起，程序开始计算用户接通最右侧的开关所经过的时间（ms）。
4. 当用户接通最右侧的开关（SW[0]）时，LED上将以二进制形式显示接通开关（向上拨）所经过的时间（ms），串行控制台上则以十进制形式显示该时间。
5. 如果用户断开（向下拨）最右侧的开关，此游戏将重复进行。

练习4. `rand()`函数的一个问题是该函数使用可预测的随机数序列。也就是说，每次运行该程序时，都会以同样的随机数开头，然后遵循同样的随机数序列。在练习3中多次运行该程序，就可以证实这一点。

但是，如果先使用`srand()`函数，即可使`rand()`函数采用随机的起始值。唯一的问题是，必须给`srand()`一个输入参数，一个随机的无符号整数。例如，可将用户断开开关以开始游戏所用的时间作为随机数赋值给`srand()`。

请重新编写练习3的程序，以实现在LED点亮之前生成一个真正随机的时间序列。尽可能使用函数。将程序命名为**ReactionTimeTrulyRandom.c**。

练习5. 重新编写练习4的程序，使点亮的LED的长度与响应时间成正比。这样，可以更容易地判断响应时间是否变快，无需解读毫秒数的二进制表示形式。可使不同的LED点亮范围与不同的响应时间范围相对应。例如，响应时间较快时，仅应点亮右侧的少数LED。当响应时间逐渐增加，左侧应有越来越多的LED点亮。如果响应时间非常慢，则将点亮所有LED。将程序命名为**ReactionTimeBar.c**。

练习6. 编写一个C程序以实现“Simon says”游戏。应产生以下效果：

1. 程序会使最右侧的三个LED以某种模式闪烁，然后等待用户使用最右侧的三个开关重复对应的序列。开关[2:0]对应于LED[2:0]，其中LED[0]是最右侧的LED，而开关[0]是最右侧的开关。
2. 随机模式应首先点亮1个LED，然后点亮2个LED，再点亮3个，依此类推。
3. 然后，用户尝试使用最右侧的三个开关重复该序列。当用户将开关向上拨时，相应的LED应点亮（当用户将开关向下拨时，相应的LED熄灭）。
4. 如果用户在暂停后输入正确的序列，则应显示下一个模式，序列中会多出一个LED。
5. 如果用户输入的序列错误，则LED将保持点亮状态，不会出现新的序列。
6. 将最左侧的开关（开关[15]）先向上拨（接通）再向下拨（断开），可以重置游戏。

选用函数以实现模块化的程序，使其更易于编写、调试和理解。不要忘记根据需要使用标准C库来编写程序。将程序命名为**SimonSays.c**。

练习7. 给定一个包含 $3 \times N$ 个元素的向量A，需要得出一个包含N个元素的向量B，使B的每个元素都是A中三个连续元素之和的绝对值。例如：

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

编写一个名为**Triplets.S**的RISC-V汇编程序（该程序必须符合RISC-V调用约定）：

- main程序根据以下高级伪代码实现B的计算：

```
#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;
    }
}
```

- 函数res_triplet会从位置p开始，返回向量V中每3个连续元素之和的绝对值。该函数是根据以下高级伪代码指定的规范实现的：

```
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;
}
```

- 函数abs(int x)会返回其输入参数的绝对值。

练习8. 编写一个名为**Filter.S**的RISC-V汇编程序（该程序必须符合之前学习过的函数管理标准）。可以使用以下伪代码：

```
#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++;
    }
}
```

- 编写等效的RISC-V汇编代码，其中应包含预留存储空间所需的任何指令，并声明相应的部分（.data、.bss和.text）。如果第一个参数是16的倍数，而第二个参数大于第一个参数，函数myFilter会返回值1；其他情况下则返回0。
- 编写myFilter函数的汇编代码。

练习9. 需要构建一个名为**Coprimes.S**的RISC-V汇编程序（该程序必须符合之前学习过的函数管理标准），使其可在给定的一系列整数对（>0）中找出由互质数组成的整数对。众所周知，如果两个数的公约数只有1，则称为互质数。

假定输入数据位于以下形式的数组D中：

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

每三个元素为一组（ x_i, y_i, c_i ），含义如下： x_i 和 y_i 表示一对整数， c_i 的初始值为0。运行程序后， c_i 的值必须按以下方式修改：如果 x_i 和 y_i 互质，则 $c_i = 2$ ；其他情况下 $c_i = 1$ 。

例如：

对于以下输入向量：D = (3,5,0, 6,18,0, 15,45,0, 13,10,0, 24,3,0, 24,35,0)

最终结果应为：D = (3,5,2, 6,18,1, 15,45,1, 13,10,2, 24,3,1, 24,35,2)

- 编写一个RISC-V汇编程序，该程序应遍历数组D并根据下方左框中给定的规范生成结果。该程序将调用函数check_coprime (int D [], int i)，其输入参数为D的起始地址和待检查的整数对的编号（从0到M-1）。该函数会检查数组D中的第i对整数是否为互质数，并将结果存储到相应的存储单元。
- 根据下方右框中给定的规范编写函数check_coprime的代码。请记住，函数gcd(int a, int b)是根据欧几里得算法在实验3中实现的，该函数会返回两个输入参数的最大公约数（greatest common divisor, gcd）。如果gcd为1，则两个数为互质数。

<pre>#define M 6 int D[] = {a list de 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>
---	--