THE IMAGINATION UNIVERSITY PROGRAMME

# RVfpgaEL2 Lab 3
## Function Calls

## 1. Introduction

Function calls are a critical part of any program because they allow for modularity and code reuse and, thus, make writing and debugging code easier. The C programming language also includes standard libraries, as well as processor/board specific libraries, of commonly used C functions, such as random number generators and common math functions. High-level functions are translated into assembly following a *Calling Convention*. This lab shows how to write and use functions in C programs – both functions written by the programmer as well as functions contained in C libraries. It also shows how functions are implemented in assembly language. At the end of the lab, we provide exercises on writing programs that use functions and library calls.

## 0. Writing a C program that uses functions

A function – also called a subroutine or procedure – is code that is packaged into a block of code that has a defined operation and interface (inputs and outputs). This modularity increases efficiency by decreasing complexity and supporting code reuse. A function can be called from any point in the program in such a way that, when the function finishes, program execution is resumed just after the function call. Functions may be called from another function (which are called *nested* functions), or even by the same function (called *recursive* functions).

To write a RISC-V program with functions, follow the same general steps as described in Labs 1 and 2:

1. Create an RVfpgaEL2 project
2. Write a C program
3. Download RVfpgaEL2-Boolean onto the Boolean Board and compile, download, and run a C program on RVfpgaEL2-Boolean
4. Compile, download, and run a C program on RVfpgaEL2-ViDBo

Refer to Lab 1 for detailed instructions for these steps. Below is a brief description of each step.

### Step 1. Create an RVfpgaEL2 project

Follow the same steps as in Lab 1 for creating the project in Catapult. In this case, in folder *[RVfpgaBooleanPath]/Labs/Lab03*, create a directory called *Project3*.

You can use the sources provided in Lab 1 (*CMakeLists.txt* file and *commandLine* folder), which you can find at *[RVfpgaBooleanPath]/Labs/Lab01/ProjectSources*.

### Step 2. Write a C program

Now you will add a C program to the project. Start by creating directory **src** in folder *[RVfpgaBooleanPath]/Labs/Lab03/Project3*.

Then, create a new file and type or copy/paste the following C program in the project.

```
// 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
}
```

This program is also available in the following file:

*[RVfpgaBooleanPath]/Labs/Lab03/LedsSwitches_functions.c*

Save the file into the `src` directory of your project and name the file Test.c.

## Step 3. Download RVfpgaEL2-Boolean onto the Boolean Board and compile, download, and run a C program on RVfpgaEL2-Boolean

You will now run the example program on RVfpgaEL2-Boolean. For that purpose, download the bitstream onto the board, compile the program, and launch it as explained in detail in the Getting Started Guide. You can do as many tests as you wish: run step-by-step, inspect memory, the peripherals and the registers, view the disassembly code, etc.

**Step 4. Compile, download, and run a C program on RVfpgaEL2-ViDBo**

You can also test execution on RVfpgaEL2-ViDBo. For that purpose, compile the program and launch it as explained in detail in the GSG.

## 1. Writing a C program with calls to library functions

High-level programming languages such as C include libraries of functions that are commonly used by programmers. You can google “C standard libraries” to find a list of commonly used C libraries. These libraries of functions can be used by including the header file that gives the declaration of the included functions. This is done by adding the following line to the top of the C program file:

```
#include <libraryname>
```

“`libraryname`” is replaced by the name of the library. For example, the math library (`math.h`) provides common functions such as `fabs()`, which computes the absolute value of a floating-point number, `fmax()`, which returns the largest of two floating point numbers, etc.

Another common library is the C standard library (`stdlib.h`). Some of the functions included in this library generate random numbers. For example, the program below displays a random number on the LEDs by including the `stdlib.h` header file (`#include <stdlib.h>`) and calling the `rand()` function that returns a random number.

Create a new project with the program below and run it on the Boolean FPGA board and on RVfpga-ViDBo.

```
#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
}
```

This program is also available in the following file:

*[RVfpgaBooleanPath]/Labs/Lab03/RandomNumberLEDs.c*

In addition to these C standard libraries, ChipsAlliance provides specific libraries for the VeeR EL2 processor (PSP, which you can find at `[RVfpgaBooleanPath]/common/drivers/psp`) and for the Boolean board (BSP, which you can find at `[RVfpgaBooleanPath]/common/drivers/bsp`). These libraries provide functions and macros that allow programmers to use interrupts, print a string, read/write individual registers, among other things. In the RVfpgaEL2 Getting Started Guide and in these labs, you will use many of these functions in the examples and exercises.

## 2. RISC-V Calling Convention

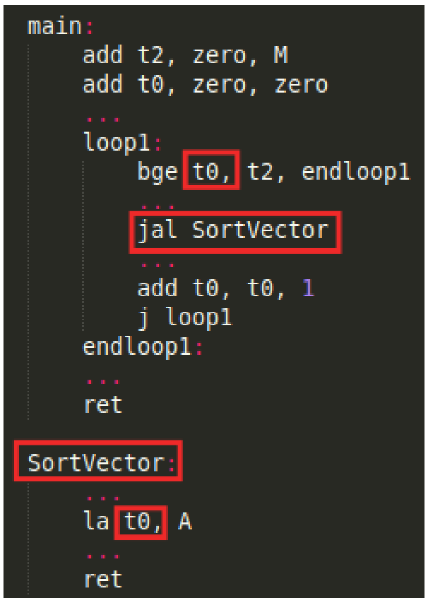
This section describes the RISC-V Calling Convention, which defines how high-level functions are translated into RISC-V assembly language. This calling convention is a part of the **Application Binary Interface** (ABI). By defining a convention, functions written by different programmers or contained in libraries can be used across programs. In RISC-V, the **jump and link** instruction (`jal`) invokes a call to a function. For example, the following code calls the function `func1`:

```
jal func1
```

This instruction both jumps to the label `func1` *and* saves the address of the instruction after `jal` into the return address register (`ra` = `x1`). The function then returns by using the return (`ret`) pseudo-instruction (or jump register instruction: `jr ra`), which jumps to the address stored in `ra`.

Functions may be called with input arguments and may also return a value to the calling function. By RISC-V convention, input arguments are passed to the function in registers `a0`–`a7`. If additional arguments are needed, they are placed on the stack. Again by convention, return values are placed in registers `a0` and `a1`. The agreement about which registers are used to pass arguments and return values is defined by **the RISC-V Calling Convention**.

In order to safely invoke a function from any location in the program, it is essential that the function preserves the architectural state of the machine (i.e. the contents of those registers than can be seen by the programmer). Suppose that we have a program with a `main` function that has a loop that uses register `t0` for storing the index of the loop. In the body of the loop, a function called `SortVector` is invoked, and this function `SortVector` uses register `t0` for storing the address of vector `A` (see Figure 1). Thus, register `t0` is overwritten in function `SortVector`, which has the undesirable side effect of modifying the index of the loop and causing its execution to be incorrect.



```
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
```

**Figure 1. Example register use conflict between main and SortVector functions**

Obviously, this wouldn't have happened if the programmer of the `main` function had selected another register to implement the loop index (for example, `t1`). However, it is not reasonable (and in some cases, not even possible) for the programmer to know the internal details of a function's implementation before calling it.

A more practical solution is for every function to create a temporary copy in memory of all those registers that will be modified, and to restore their original values before returning to the *caller* program. This solution is implemented by means of the **Call Stack**, which is a memory region that is accessed using a LIFO (Last-In-First-Out) policy. This region is used to store all the information related to the live functions of the program (i.e., those functions that have started but not finished their execution), and which begins at the end of the available memory (i.e. in the higher addresses), and grows towards lower addresses.

A function is normally structured into three parts:

- ➔ `Entry code (`**`Prologue`**`)`
- ➔ `Function `**`Body`**
- ➔ `Exit code (`**`Epilogue`**`)`

The *Prologue* must create the function's **stack frame** and store registers on the stack, if needed. The *stack frame* is the memory region used by a function during its execution. The *Epilogue* restores the architectural state of the *caller* program and releases the memory space occupied by the *stack frame*, thus leaving the stack exactly as it was before executing the *Prologue*.

Accesses to the stack are managed by means of a pointer, called the *stack pointer* (`sp` = `x2`), which stores the address of the last occupied location of the stack. Before a program begins, `sp` must be initialized with the address of the base of the stack (i.e. the highest address of the stack region). At initialization, the stack is empty. A second pointer, the *frame pointer* (`fp` = `x8`) points to the base address (i.e. the highest address) of the active function's *stack frame*.

Functions use the ***stack frame*** as a private memory region, which can only be accessed from the function itself. A part of the ***stack frame*** is devoted to save a copy of the architectural registers that are to be modified by the function and, in some cases, it can also be used as a way of passing parameters to the function through memory locations.

Table 1 describes the intended role that the RISC-V convention assigns to each integer register. As also illustrated in Table 1, some registers must be preserved by a called function whereas some others may be overwritten by the function (i.e., they are not preserved).

- If the function needs to overwrite any preserved registers, it must first make a copy of such a register in its *stack frame* and restore the value before returning to the *caller* (i.e., the function that called it). In addition to the stack pointer (`sp`) and return address register (`ra`), twelve integer registers `s0`–`s11` are preserved across calls and must be saved by the *callee* if used by it.

- On the other hand, the *caller* must be aware that some registers need not be preserved by the *callee* and, thus, could be lost after the call. Note that, in addition to the argument and return value registers (`a0`–`a7`), seven integer registers `t0`–`t6` are temporary registers that are volatile across calls and must be saved by the *caller* if used again after the function invocation.

**Table 1. RISC-V integer registers**

| Name | Register Number | Use | Preserved |
|---|---|---|---|
| **zero** | **x0** | Constant value 0 | - |
| **ra** | **x1** | Return address | Yes |
| **sp** | **x2** | Stack pointer | Yes |
| **gp** | **x3** | Global pointer | - |
| **tp** | **x4** | Thread pointer | - |
| **t0-2** | **x5-7** | Temporary variables | No |
| **s0/fp** | **x8** | Saved register / Frame pointer | Yes |
| **s1** | **x9** | Saved register | Yes |

| | | | |
|---|---|---|---|
| **a0-1** | **x10-11** | Function arguments / Return values | No |
| **a2-7** | **x12-17** | Function arguments | No |
| **s2-11** | **x18-27** | Saved registers | Yes |
| **t3-6** | **x28-31** | Temporary variables | No |

In the example from Figure 1, there would be two solutions according to this convention:

- The `main` program could use a register for the loop index that is guaranteed to be preserved by the `SortVector` function (such as `s0`) instead of `t0`.
- The `main` function could keep using `t0`, but then it has to preserve its content on the stack before calling `SortVector` and restore it after returning from `SortVector`.

The stack expands as more memory is needed by functions' stack frames and contracts as those functions complete. The stack grows downwards (towards lower addresses) and, according to RISC-V convention, the stack pointer must be aligned to a 16-byte boundary.

## Example

The following example implements a sorting algorithm, first in C (Figure 2) and then in RISC-V assembly language (Figure 3). The input is an array `A` of `N` elements, each being an integer greater than 0. The output is another array, `B`, that stores the elements of `A` in decreasing order.

In C, the `main` function calls function `SortVector`, which receives the addresses of arrays `A` and `B`, and their size (`N`), and stores the elements of `A` into `B` element-by-element, in decreasing order. This `SortVector` function calls another function, `MaxVector`, which receives the address of array `A` and its size and returns the maximum value of array `A` and resets that value, so that it is no longer considered in the following iterations.

```
#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);
}
```

**Figure 2. Sorting algorithm in C language**

Figure 3 illustrates the same algorithm written in assembly. We analyse the program, taking into account the concepts explained in the previous sections.

- **`main` function**
  - Prologue
    - First, space is reserved in the stack for storing the preserved registers that are used in the function: `add sp, sp, -16`. Note that, according to the convention, the stack pointer, register `sp`, must always be 16-byte aligned to maintain compatibility with the 128-bit version of RISC-V, RV128I.
    - Given that no saved register is used by this function, `s0`-`s11` registers need not be stored in the stack. However, register `ra` must be saved, given that `main` calls function `SortVector`, which call will overwrite the value in `ra`.
  - Function Body
    - The `SortVector` function is invoked using instruction `jal SortVector`. Before calling the function, according to the Calling Convention, the 3 input parameters are placed in registers `a0` (address of `A`), `a1` (address of `B`), and `a2` (size of `A` and `B` arrays).
  - Epilogue
    - The register that was saved in the stack at the prologue (`ra`) is now restored.
    - The stack pointer (`sp`) is also restored to its initial position: `add sp, sp, 16`.

- **`SortVector` function**
  - Prologue
    - First, space is reserved in the stack for storing the preserved registers that are used in the function: `add sp, sp, -32`.
    - Then, the saved registers used by the function (`s1`-`s3`) are stored in the stack, one by one.
    - Register `ra` must also be saved, because `SortVector` calls the `MaxVector` function, which overwrites the value stored in `ra`.
  - Function Body
    - First, the input parameters (`a0`, `a1`, and `a2`) are moved into preserved registers (`s1`, `s2`, and `s3`), so that they can be used after the execution of function `MaxVector`.
    - For computing vector `B`, a loop is implemented that, in each iteration, computes the maximum value of `A` and stores it into `B`. For computing the maximum value of `A`, the `MaxVector` function is invoked in each iteration of the loop: `jal MaxVector`. Before calling the function, according to the Calling Convention, the input parameters to this function are moved into registers `a0` and `a1`. When the function finishes execution, it returns the maximum value of `A` in register `a0`.
    - Note that the loop mostly uses the saved registers to store variables. These registers are guaranteed by the RISC-V Calling Convention to

preserve their value after the execution of the `MaxVector` (i.e. the function must preserve their values).

        - Registers `a0` and `a1` may be modified by the function. Thus, they must be initialized before each function call.
        - Register `t1` needs to be reused after `MaxVector` returns. Thus, it must be preserved in `SortVector`'s stack before calling the function (`sw t1, 16(sp)`) and restored after executing it (`lw t1, 16(sp)`).
    - Epilogue
        - The registers that were saved in the stack during the prologue, are now restored.
        - The stack pointer (`sp`) is also restored to its initial position: `add sp, sp, 32`.

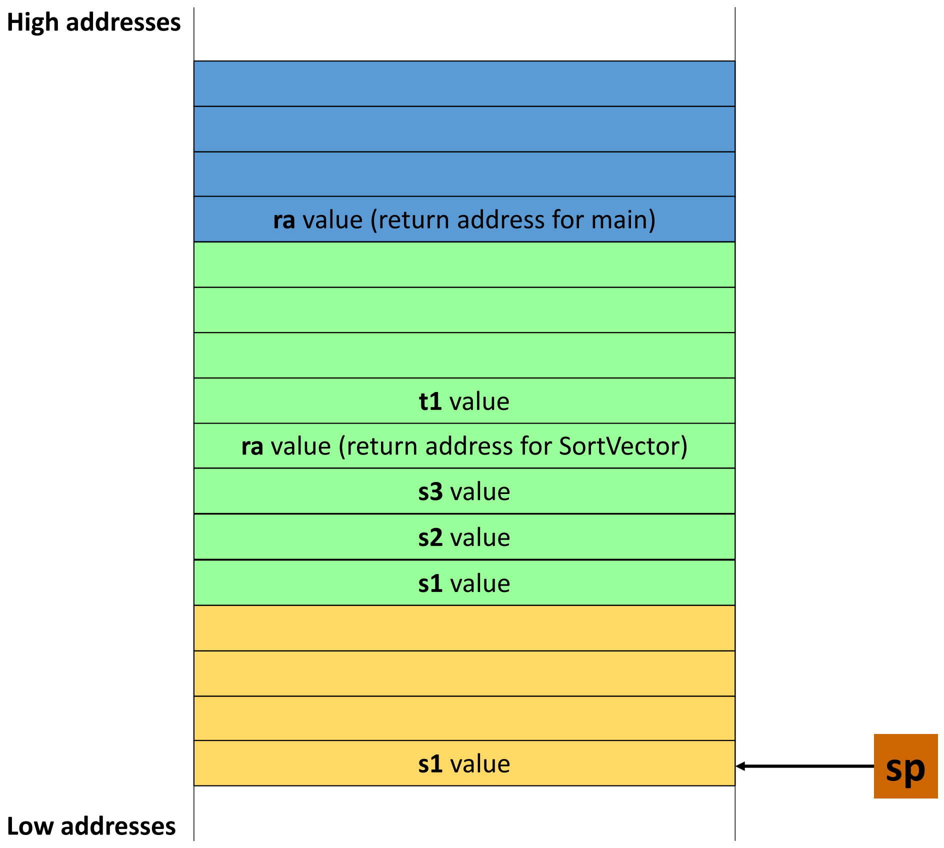
- **`MaxVector` function**
    - Prologue
        - First, space is made on the stack for storing the preserved registers that are used in the function: `add sp, sp, -16`.
        - Then, the saved register used by the function (i.e., register `s1`) is stored in the stack: `sw s1, 0(sp)`. Note that, if this register were not saved by this function, the execution of the *caller* function (`SortVector`) would fail, as it is also using this register for storing the address of vector `A`.
        - Because this function does not invoke another one (it is a *leaf* function); thus, `ra` need not be saved in this case.
    - Function Body
        - The function uses `s1` and some temporary registers to calculate the maximum value of array `A`.
    - Epilogue
        - The function must prepare the return value before returning to the *caller*: `mv a0, t2`.
        - The register that was saved on the stack during the prologue (`s1`), is now restored.
        - The stack pointer (`sp`) is also restored to its initial position: `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
```

**Figure 3. Sorting algorithm in assembly language**

Figure 4 illustrates the state of the stack at the point of executing the body of the `MaxVector` function.

- The *stack frame* of the `main` function is shown in blue, and it includes the return address (`ra`) for that function.
- The *stack frame* of the `SortVector` function is shown in green, and it includes the saved registers used by this function (`s1`-`s3`), register `t1`, and *`ra`*.
- Finally, the *stack frame* of the `MaxVector` function, which is the *active stack frame* (the *stack frame* of the function that is currently executing), is shown in yellow, and it includes the saved register used by this function (`s1`).



High addresses
ra value (return address for main)
t1 value
ra value (return address for SortVector)
s3 value
s2 value
s1 value
s1 value
sp
Low addresses

**Figure 4. Stack state during execution of the `MaxVector` function from Figure 3**

<u>**TASK**</u>**:** The assembly program from Figure 3 is provided in a Catapult project available at: *[RVfpgaBooleanPath]/Labs/Lab03/SortingAlgorithm_Functions*. Execute this program using the step-by-step debugger option for analysing the value stored in the various registers (`s`, `ra`, `a`, etc.) as well as the values stored in the stack, according to the RISC-V Calling Convention. You can use the Memory Console for analysing the evolution of the stack as well as the contents of arrays `A` and `B`.

## 3. Exercises

Now create your own C/Assembly programs that include function calls by completing the following exercises.

Remember that if you leave the Boolean board connected to your computer and powered on, you do not need to reload RVfpgaEL2-Boolean onto the board between different programs. However, if you turn off the Boolean board, you will need to reload RVfpgaEL2-Boolean onto the board using Catapult.

Remember as well that you can run these programs in simulation, using the different Verilator-based simulators and the RVfpgaEL2-Whisper Instruction Set Simulator.

**Exercise 1.** Write a C program that displays the inverse of the switches on the LEDs. Name the program **DisplayInverse_Functions.c**.

For example, if the switches are (in binary): 0101010101010101, then the LEDs should display: 1010101010101010; if the switches are: 1111000011110000, then the LEDs should display: 0000111100001111; and so on. Include a `getSwitchesInvert()` function that returns the inverted value of the switches. The function declaration is:

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

**Exercise 2.** Write a C program that flashes the value of the LEDs onto the switches. Name the program **FlashSwitchesToLEDs_Functions.c**

The value should pulse on and off about every two seconds. Include a function called `delay()` that causes a delay of `num` milliseconds. This can be done empirically and does not need to be exact. The function declaration looks like this:

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

**Exercise 3.** Write a C program that measures reaction time. Your program should time how long it takes for a person to switch on the right-most switch (SW[0]) after all of the LEDs light up. You will use the `rand()` function from the `stdlib.h` library to generate a random amount of time to delay between each time the user attempts to test their reaction time. Name the program **ReactionTime.c**.

The program should work as follows.

1. The user toggles the right-most switch off (down) to indicate they'd like to begin.
2. The program turns off all of the LEDs, then waits for a random amount of time (but no longer than about 3 seconds). You will want to use the delay() function from Exercise 2.
3. Then all the LEDs turn on and the program begins counting the number of milliseconds until a user switches the right-most switch on.
4. When the user toggles the right-most switch (SW[0]) on, the number of milliseconds it took to toggle the switch up (on) is displayed in binary on the LEDs and in decimal on the serial console.
5. The game then repeats by the user toggling the right-most switch down (off).

**Exercise 4.** One issue with the `rand()` function is that it uses a predictably random sequence of numbers. That is, each time you run the program it will start with the same random number and follow the same sequence of random numbers. Run your program from Exercise 3 several times to see that it starts with the same random number and follows the same random sequence.

However, if you use the `srand()` function first, it will seed the `rand()` function with a random starting point. The only issue is that `srand()` must be given an input argument, an unsigned integer, that itself is random. Give `srand()` a random number, for example, the number of milliseconds until the user toggles the switch off to begin the game.

Rewrite Exercise 3 to produce a truly random sequence of times before the LEDs turn on. Use functions when possible. Name the program **ReactionTimeTrulyRandom.c**.

**Exercise 5.** Rewrite Exercise 4 so that the LEDs display a growing bar of LEDs, proportional to reaction time. This way, the person viewing their reaction time can more easily tell if they are getting faster – without having to interpret the binary representation of the number of milliseconds. You may choose the range of reaction times corresponding to each range of lit LEDs. For example, for quick reaction times, only a few LEDs on the right should light up. An increasing number of LEDs to the left should light up as reaction times increase. A very slow reaction time would light up all of the LEDs. Name the program **ReactionTimeBar.c**.

**Exercise 6.** Write a C program that implements a "Simon says" game. The following should happen:

1. The program blinks a pattern on the three right-most LEDs and waits for the user to press the corresponding sequence of switches using the three right-most switches. Switches[2:0] correspond to LED[2:0], with LED[0] being the right-most LED and Switches[0] being the right-most switch.
2. The random patterns should start by lighting 1 LED, then 2 LEDs, then 3, etc.
3. The user then tries to repeat the sequence using the three right-most switches. The corresponding LED should light up as the user toggles the switches up (and turn off when the user toggles the switch back down).
4. If the user enters the correct sequence, after a pause, the next pattern should display, with one more LED in the sequence.
5. If the user enters the wrong sequence, the LEDs stay lit and no new sequence is played.
6. The game is reset by pushing the left-most switch (Switches[15]) up (on) and then down (off).

Use functions of your choice to modularize the program and make it easier to write, debug, and understand. Remember to use standard C libraries as desired to write your program. Name the program **SimonSays.c**.

**Exercise 7.** Given a vector, `A`, of 3*`N` elements, we want to obtain a new vector, `B`, of `N` elements, so that each element of `B` is the absolute value of the sum a triplet of consecutive elements of `A`. For example:

```
B[0] = |A[0]+A[1]+A[2]|,   B[1] = |A[3]+A[4]+A[5]|, ...
```

Write a RISC-V assembly program called **Triplets.S** (the program must conform to the RISC-V calling convention):

- The `main` program computes the values in `B`, according to the following high-level pseudo-code:

```
#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;
      }
}
```

- Function `res_triplet` returns the absolute value of the sum of 3 consecutive elements of a vector `V`, starting at position `pos`. It is implemented according to the specification given by the following high-level pseudo-code:

```
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;
}
```

- Function `abs(int x)` returns the absolute value of its input argument.

**Exercise 8.** Write a RISC-V assembly program called **Filter.S** (the program must be compliant with the standard for function management studied before). You can use the following pseudo-code:

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

- Write the equivalent RISC-V assembly code, including any directives required to reserve memory space, and declaring the corresponding sections (`.data`, `.bss` and `.text`). Function `myFilter` returns the value 1 if the first argument is a multiple of 16 and the second is greater than the first; otherwise, it returns a 0.

- Write the assembly code of the function `myFilter`.

**Exercise 9.** We want to build a RISC-V assembly program called **Coprimes.S** (the program must be compliant with the standard for function management studied before), such that given a list of pairs of integers (>0) finds which pairs are composed of coprime (or mutually prime) numbers. It is understood that two numbers are coprime if the only common divisor they have is 1.

We assume that the input data are contained in an array, `D`, of the form:

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

Each triplet $(x_i, y_i, c_i)$ is interpreted as follows: $x_i$ and $y_i$ represent a pair of numbers, and $c_i$ is initially 0. After running the program, the value of each $c_i$ must have been modified in such a way that $c_i$ = 2, if $x_i$ and $y_i$ are coprime; and $c_i$ = 1, otherwise.

For example:

For the following input vector: `D` = (3,5,0, 6,18,0, 15,45,0, 13,10,0, 24,3,0, 24,35,0)
The final result should be: `D` = (3,5,2, 6,18,1, 15,45,1, 13,10,2, 24,3,1, 24,35,2)

- Write a RISC-V assembly program that traverses the array `D` and generates the result according to the specification given in the left box below. The program calls the function `check_coprime (int D [], int i)`, whose input arguments are the base address of `D` and the number of the pair that we want to check (from 0 to `M`-1). The function checks if the numbers of the `i`th pair of array `D` are coprime and stores the result in the corresponding memory location.

- Write the code for the functions `check_coprime`, according to the specification given in the right box below. Remember that function `gcd(int a, int b)` was implemented in Lab 2 according to the Euclidean algorithm, and it returns the greatest common divisor (GCD) of the two input arguments. If the GCD is 1, then the numbers are coprime.

```
#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);
}
```

```
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;
}
```