CS2053 - Computer Architecture

Department of Computer Science and Engineering - University of Moratuwa

Lab3: Stack and Function Arguments

Learning Outcomes:

After completing this lab, you will be able to:

- Explain the implementation of stack in an assembly program.
- Use the stack to store and retrieve values in LIFO order.
- Implement functions and link them.
- Explain how to preserve the function context and pass arguments using the stack.

- ➢ There is a Moodle quiz which will be opened at 8.15 am and closed at 10.15 am based on this lab.
- ➢ Download the `Lab03_Example.s` file from the Moodle course page and open it in the Ripes tool.

Introduction

Function calls play a crucial role in programming since they enable us to create modular and reusable code. This simplifies the process of writing and debugging code. High-level programming languages also come with libraries with standard functions, along with libraries tailored to specific processors or boards. The C Standard Library is a good example of that.

The functions, which are written in high-level languages, are converted into assembly code according to the Calling Convention of the architecture. This lab provides insight into the implementation of functions in assembly language.

The lab also introduces the concept of the stack. The stack is a crucial element in computer memory, especially in function calls. It is a memory area that handles the program flow and stores function-related information. This includes return addresses and local variables. Understanding the stack is vital for managing function execution and program states.

RISC-V Calling Convention

The calling convention defines how the return address, return values, function arguments, and stack are placed in the memory for function calls. Even though the RISC-V architecture technically supports passing arguments and return values using the stack, it expects the function arguments to be directly passed using a0-a7 registers and return values to be directly passed using a0-a1 registers whenever possible.

1. Register Bank

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Figure 1: Register usage of RISC-V calling convention
Source: [riscv-spec.pdf]

The register `ra (x1)` holds the return address of the current function. The register `sp (x2)` contains the address of the stack top. More details about the stack will be covered in a later section. The registers `a0-a7 (x10-x17)` are used to pass the function arguments. The `a0` and `a1` registers are used to return values to the caller. If you have more than eight arguments or more than two return values, those should be passed using the stack. This lab does not cover that. The temporary registers `t0-t6` (`x5-x7` and `x28-x31`) registers are used to store temporary values within a function. We should not expect the temporary registers to preserve their values after calling another function. The values stored in saved registers `s0-s11` (`x8`, `x9`, `x18-x27`) should be preserved (ie. the values in these registers at the start of the function, will be the same when it returns).

In summary,

Preserved registers: `ra`, `sp`, `s0-s11`
Non-preserved registers: `t0-t6`, `a0-a7`

★ Any modifications done to the preserved registers within a function should be restored when it returns. We will see how this happens in the upcoming sections.

2. Calling and Returning

Since a function call does not guarantee preserving the data in the `a0-a7` and `t0-t6` registers, you need to make sure all the required values are placed in the saved registers `s0-s11` before calling the function. Remember that the values in the saved registers are preserved during a function call.

A function call may take some arguments. Therefore, the arguments to be passed to the function should be stored in registers `a0-a7` first. The function call can be done in multiple ways:

- Use the `jal ra <memory_label>` J-type instruction to call the function labelled <memory_label>. It will take the following actions:

```
ra = pc + 4 # return address = next instruction
pc = <memory_label> # jump to function
```

- Use the `jalr ra rd <displacement>` I-type instruction to call the function located at <displacement> number of bytes to the address stored in the rd register. It will take the following actions:

```
ra = pc + 4 # return address = next instruction
pc = rd + <displacement> # jump to function
```

- The `jal` and `jalr` pseudo instructions can be used for the same purpose. These pseudo instructions do not require the `ra` register to be specified in the command itself. Also, the `jalr` pseudo instruction does not take a displacement as the `jalr` I-type instruction does.

To return from the function, call the `ret` pseudo instruction after placing the required return values in the `a0` and/or `a1` registers if any. See the example given below and observe how the functions are called and returned.

3. The Stack

The stack is a Last In First Out (LIFO) type of data structure. The stack memory area is generally used to allocate memory for static variables. As you will see in the next section, the stack frame of a function is also saved onto the stack.

In the Ripes simulator, the stack starts at a higher memory location and grows downwards. The register `sp` is the stack pointer. It holds the memory address of the current top of the stack. When you place some values on top of the stack, you have to decrement the stack pointer since it grows downward. Similarly, when you are removing (or abandoning) values from the top of the stack, you have to increment the stack pointer. Before the program starts, the `sp` register will be initialised with the address of the base of the stack.

★ **What is the base address of the stack in the Ripes simulator?** See the value of the register `sp` at the beginning of an assembly program.

4. The Prologue

Within a function call, we have to preserve the values in the `ra`, `sp`, and `s0-s11` registers. Therefore, we have to save these values in some way and then restore them later. A Call Stack is used as the solution for this.

The prologue of a function creates the stack frame for that function. It executes before the function body. What it effectively does is to push the values of the preserved registers, that will be altered by the function body, to the stack. Then it will decrement the stack pointer to point to the new top of the stack (remember that the stack grows downwards). See the prologue section of the example given below.

Also, the current return address should be saved to the stack if it calls another function. Let's say `func1` calls `func2`. The return address of the `func1` is stored in the `ra` register before the call to `func2`. It is essential to save that address since when we call `func2`, its return address will override the value in the register `ra` which is the return address of `func1`.

By convention, the stack frame is aligned to 16 bytes. That means when you are allocating memory in the stack for a function frame, you have to allocate memory in a multiple of 16 bytes. See in the example that the `MaxVector` function allocates 16 bytes on the stack and the `SortVector` function allocates 32 bytes on the stack even though both functions do not use the complete allocated space.

```
def prologue ():
    decrement sp by num s registers + local var space
    Store any saved registers used
    Store ra if a function call is made
```

Figure 2: The general structure of the prologue
Source: [RISCV_Calling_Convention.pdf (berkeley.edu)]

5. **The Epilogue**

The epilogue will be placed after the function body. It will restore the preserved values saved in the stack frame back to those registers. Also, if the function did call another function during its execution time, the return address should be restored from the stack. Then it will increment the stack pointer to point to the original stack top. This will restore the architectural state of the system before the function call. Finally, it calls the `ret` instruction to jump back to the return address.

```
def epilogue ():
    Reload any saved registers used
    Reload ra (if necessary)
    Increment sp back to previous value
    Jump back to return address
```

Figure 3: The general structure of the epilogue
Source: [RISCV_Calling_Convention.pdf (berkeley.edu)]

In summary, a function call has three major sections:

1. **Prologue**
    - Entry point to the function.
    - Creates the stack frame and saves the state of the preserved registers.
2. **Function Body**
3. **Epilogue**
    - Restores the original state.
    - Returns from the function.

➢ **You should always follow the calling convention, even if you are able to write a working solution without adhering to it.**

Example

The following example implements a sorting algorithm, first in C (Figure 4) and then in RISC-V assembly language (provided in Moodle). 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.

```c
#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 4: The sorting algorithm in C language

In the given C program, the primary function, "`main`", invokes the "`SortVector`" function. This "`SortVector`" function takes the addresses of arrays `A` and `B`, along with their size denoted as `N`. The purpose of this function is to transfer the elements of array `A` into array `B`, one element at a time, arranging them in descending order.

Within the "`SortVector`" function, there exists a call to another function named "`MaxVector`". This secondary function, "`MaxVector`", accepts the address of array `A` and its size as parameters. Its role is to determine the maximum value present in array `A` and subsequently reset that particular value. This action is performed to exclude the maximum value from future considerations.

Download the `Lab03_Example.s` file from the Moodle course page and open it in the Ripes tool.

First, 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: `addi sp, sp, -16`. Note that, according to the convention, the `sp` register must always be kept 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 the `main` calls function `SortVector`, which updates the value stored in `ra`.
  - Function Body
    - The `SortVector` function is invoked using the 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: `addi sp, sp, 16`.

- **`SortVector` function**
  - Prologue
    - First, space is reserved in the stack for storing the preserved registers that are used in the function: `addi 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 in 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` can be modified by the function. Thus, they must be prepared before every invocation.
    - 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: `addi sp, sp, 32`.

- **`MaxVector` function**
    - Prologue
        - First, space is made on the stack for storing the preserved registers that are used in the function: `addi 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), `ra` needs not to 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: `addi sp, sp, 16`.

The below Figure 5 shows the stack 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 returning 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 executing), is shown in yellow, and it includes the saved register used by this function (`s1`).

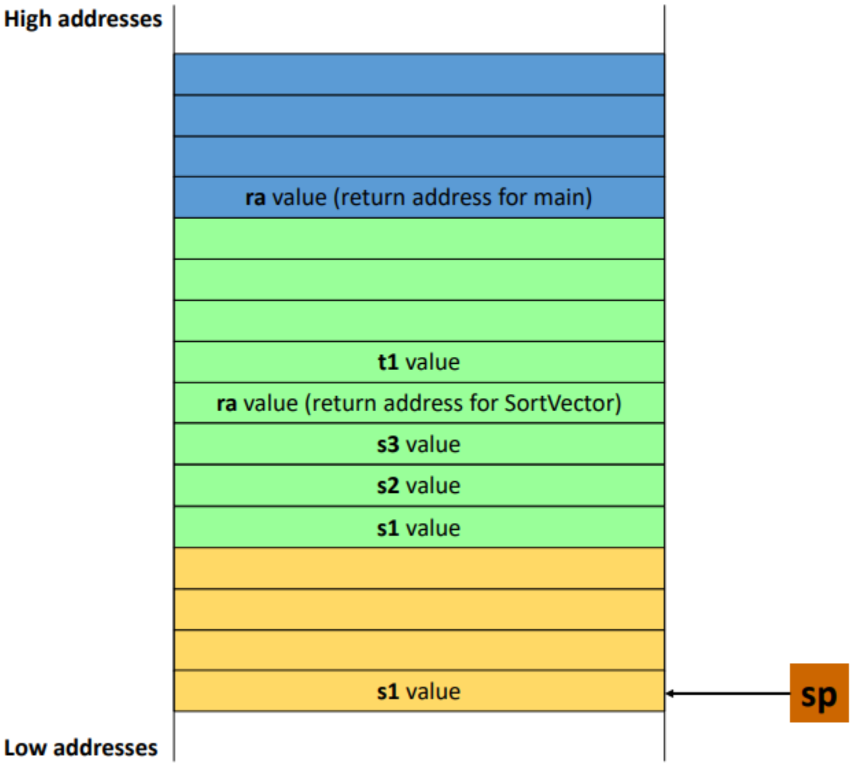

Figure 5: Stack state at the body of function MaxVector

Execute this program on the Ripes simulator and analyse 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.

Exercise 1

Given a vector, A, of 3*N elements, obtain a new vector, B, of N elements, so that each element of B is the absolute value of the sum of 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 confirm to the RISC-V calling convention)

The main program implements the computation of 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 the vector V, starting at position p. 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 2

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`.

Additional Exercise

Coprime numbers, also known as mutually prime numbers, are two positive integers that have 1 as the only common divisor. In other words, their greatest common divisor (GCD) is equal to 1.

Build a RISC-V assembly program called **Coprimes.S**, such that given a list of pairs of integers (>0) finds which pairs are composed of coprime numbers.

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`, ... , `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)`

1. 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 starting 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.

2. Write the code for the functions `check_coprime`, according to the specification given in the right box below. Remember that the function `gcd(int a, int b)` was implemented in Lab 2 according to the Euclidean algorithm.

```c
#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);
}
```

```c
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;
}
```

References

- Calling Convention - https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf.
- Understanding RISC-V Calling Convention by Nick Riasanovsky - https://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Calling_Convention.pdf.
- Lab03 - Imagination University Programme Version 2.2 – 9th May 2022 © Copyright Imagination Technologies