MarieSimR: The MARIE Computer Simulator Revision

Xuejun Liang
Department of Computer Science
California State University – Stanislaus
Turlock, CA 95382, USA
xliang@cs.scustan.edu

Abstract – MarieSim is the MARIE computer simulator [1] and MARIE is an accumulator-based computer architecture model used in the textbook [2]. MarieSimR, a revision of MarieSim, is developed to support the use of stack and recursive subroutine. Users can learn and practice more assembly language programming skills with using MarieSimR. In this revision, the stack pointer stored in a reserved memory location and the stack-relative addressing mode are added. The subroutine call and return instructions are revised to use the stack for the subroutine return address. A stack frame can be created for a subroutine to hold the return address, input arguments, output results, local variables and so on. A new instruction for increasing or decreasing the value of the stack pointer is added to facilitate the push and pop operations. In addition, a new instruction for loading an immediate constant into the accumulator is added to replace the clear instruction. Finally, a new assembler directive is also added to support for defining a label to hold the address of another label symbolically. In this paper, the design and implementation of this revision are presented. Two programming applications are also discussed to illustrate how to utilize these new instructions.

Keywords - Computer Organization and Architecture, Simulator, Accumulator Machine, Assembly Language Programming

I. INTRODUCTION

Assembly language programming and writing, using and modifying processor simulators are major hands-on assignment categories in an undergraduate computer architecture course [3]. There are many computer architectures with different instruction formats such as stack-based, accumulator-based, register-based, two-address, or three-address machine. Linda Null and Julia Lobur designed an accumulator-based computer architecture model called MARIE (the Machine Architecture that is Really Intuitive and Easy) and its simulator called MarieSim [1] which is used in their popular textbook [2] to teach beginning computer organization and architecture. Users can assemble, execute, and debug their assembly language program and track the values of internal CPU registers and the memory contents in MarieSim environment during the execution of each instruction. But, MarieSim is too simple and thus unable to support some important concepts in computer architecture such as the immediate addressing mode, the stack, and thus recursive subroutines. Meanwhile, MarieSim assembler does not support for defining a label (pointer) to hold the address of another label symbolically. Programmers have to compute (count) the address of a label manually and then hardcode it into their code.

In order to solve these problems, MarieSimR, a revision of MarieSim, is developed. In MarieSimR, an assembler directive *Lab* is added for defining a label to hold the address of another label symbolically. An instruction *Limm* is added for loading an immediate constant into the accumulator *AC*. This instruction is an extension of MARIE instruction *Clear* which loads zero into *AC*. More importantly, a stack pointer that is stored in a reserved memory location *SP* and the stack-relative addressing mode are added. Meanwhile, a new subroutine call instruction *Call* and a new subroutine return instruction *JR* are added to use the stack for saving and getting the return address, respectively. They will replace the MARIE subroutine call and return instructions *JNS* and JUMPI, respectively, which use the memory slot just before the first instruction of the subroutine for the return address. In addition, a new instruction *IncSP* for increasing or decreasing the value of the stack pointer is added to facilitate the stack operations. With this new revision, a stack frame can be built for a subroutine to hold the return address, input arguments, output results, local variables, etc. and thus recursive subroutines can be supported.

In the rest of the paper, the MARIE instruction set and its revision are described in Section II. Two example applications are discussed to illustrate how to utilize the new revision and to compare MarieSim and MarieSimR in Section III. The first application is to compute the sum of an array of numbers and the second is to compute the Fibonacci numbers in which different programming techniques are used so that students can have a better understanding of global variables, local variables, stack, subroutine, and recursion. Finally, a conclusion is given in Section IV.

II. MARIE INSTRUCTION SET AND ITS REVISION

The MARIE architecture is an accumulator-based computer architecture model and has the following characteristics: (1) 2's complement data representation, (2) fixed word length data and instructions, (3) 4K words of word-addressable main memory, (4) 16-bit data words, (5) 16-bit instructions, 4 for the opcode and 12 for the address, (6) A 16-bit arithmetic logic unit (ALU), and (7) Seven registers for control and data movement. The goal of this work is to revise MARIE to support stack and recursive subroutines without changing its above characteristics and its microarchitecture for control unit. Note that the seven registers are not accessible to assembly language programmers.

In order to support stack, a user-accessible stack pointer (the address of top of stack) must be added into MARIE. Instead of using a dedicated register, a reserved memory location is used to store the stack pointer. As shown in Figure 1, stack pointer is stored at the reserved memory location *SP* which is 3071. Here, Mem[*SP*] means the memory content at the location *SP*. User programs and data occupy the low address region from 0 to 3070 and stack occupies the high address region from 3072 to 4095. Note that the stack uses 1K out of 4K MARIE memory space and grows towards the higher address end.

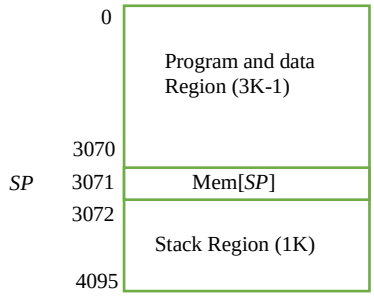

Figure 1: MARIE Memory Map After Extension

There are two reasons for using a reserved memory location rather than a dedicated register to store the stack pointer. First, there is no space to add one more register in the MARIE datapath without changing its microarchitecture for control unit. Second, if a dedicated stack pointer register is used, the user interface of MARIE simulator has to be changed to display the value of the stack pointer register. On the other hand, if a reserved memory location is used, users can observe the value of the stack pointer from the memory monitor panel in the current MARIE simulator user interface without any changes.

The MARIE instructions and the revised/new instructions (in red color) are shown in Table 1. MARIE has 15 instructions currently. The MARIE subroutine call and return instructions JnS and JumpI are replaced by Call and JR, respectively. The new subroutine call instruction Call pushes the return address on top of stack, while the new subroutine return instruction JR gets the return on top of stack. The MARIE instruction Clear which resets *AC* to zero is replaced by the new instruction Limm which loads an immediate number into *AC*. Finally, a new instruction IncSP is added to increase or decrease the value of stack pointer by a constant amount.

Note that in Table 1, X can be either a hexadecimal literal or a label (symbol) and is used as a memory address in MarieSim. Mem[X] represents the content at the memory location X. *PC* is the program counter which holds the address of the instruction to be executed next. *Imm* is a decimal constant integer. Note that the range of X is from 0 to 4095, while the range of *Imm* is from -2048 to 2047. They both occupy 12 bits inside an instruction and an opcode takes 4 bits inside an instruction. The total length of an instruction is 16 bits.

In MarieSimR, when using direct or indirect addressing, X is the same as that in MarieSim. But when using stack-relative addressing, X has the format: *$* ± *Offset*, where *$* represents the value of stack pointer which is stored in the reserved memory location *SP* and *Offset* is a 10-bit decimal constant integer. The 2 bits right before *Offset* inside an instruction are set to be 11. This indicates that stack-relative addressing is used. The address represented by $ ± *Offset* is equal to Mem[*SP*] ± *Offset*. Please note that both the value of the stack pointer and the contents in stack can be observed from the memory monitor panel in the MarieSim user interface.

Table 1: MARIE Instruction Set and Its Revision

Opcode	Instruction	Meaning
0000	JnS X	Mem[X] ← PC & PC ← X+1
	Call X	Push PC & PC ← X
0001	Load X	AC ← Mem[X]
0010	Store X	Mem[X] ← AC
0011	Add X	AC ← AC + Mem[X]
0100	Subt X	AC ← AC – Mem[X]
0101	Input	AC ← value from keyboard
0110	Output	Display value in AC on screen
0111	Halt	Terminate program
1000	Skipcond 000	Skip next instruction if AC < 0
	Skipcond 400	Skip next instruction if AC = 0
	Skipcond 800	Skip next instruction if AC > 0
1001	Jump X	PC ← X
1010	Clear	AC ← 0
	Limm *Imm*	AC ← *Imm*
1011	AddI X	AC ← AC + Mem[Mem[X]]
1100	JumpI X	PC ← Mem[X]
	JR	POP PC
1101	LoadI X	AC ← Mem[Mem[X]]
1110	StoreI X	Mem[Mem[X]] ← *AC*
1111	IncSP Imm	Mem[SP] += Imm

Two important operations on stack are push and pop. Push X operation will push memory content at memory address X, i.e. Mem[X], on top of stack and Pop X operation will store the content on top of stack in memory location X and remove it from stack. These two operations can both be implemented by three assembly instructions as shown in Table 2. Push X will increase the value of stack pointer (Mem[SP]) by one first and then stores Mem[X] on top of stack. On the other hand, Pop X will store the content on top of stack in memory location X first and then decreases the value of stack pointer by one.

Table 2: Push and Pop Operations

Push X	Pop X
IncSP 1	loadI *SP*
Load X	Store X
StoreI *SP*	IncSP -1

Note that the Push X and Pop X operations will be used in the assembly language programs in this paper for the sake of saving space. In order to assemble and to run the programs, you must expand Push X and Pop X, i.e., replace them with their corresponding sequences of instructions as shown in Table 2, respectively.

A. *MARIE Directives and One New Directive*

Directives are instructions to assemblers. MarieSim has five directives. Directive ORG defines the starting address of the program. Directives DEC, OCT, and HEX define named constant (or variable) in decimal, octadecimal, and hexadecimal, respectively. Directive END indicates the end of the program. MarieSimR adds one more directive LAB which defines a named hexadecimal constant (or variable) specified either by a hexadecimal literal or a label symbolically.

III. Assembly Language Program Examples Using MarieSim and MarieSimR

In the following subsections, two simple examples are used to illustrate how to write assembly language programs to deal with array, loop, function, and recursion by using MarieSimR. Meanwhile, comparisons between MarieSim and MarieSimR are also discussed. The first example is to compute sum of all values in an array. The second example is to compute Fibonacci number. Note that very similar examples were used to show how to write assembly language programs for different instruction formats in [4, 5].

A. *Sum of an Array of Numbers*

This example is to compute sum of an array of numbers. As shown in Figure 2, The code in the middle is for MarieSim but it works fine in MarieSimR as well, the code on the right is for MarieSimR only, and the left is the memory address of each corresponding instruction or data on its right.

```
/ Address

0x100
0x101
0x102
0x103
0x104
0x105
0x106
0x107
0x108
0x109
0x10A
0x10B
0x10C
0x10D
0x10E
0x10F
0x110
0x111
0x112
0x113
0x114
0x115
0x116
0x117
0x118
0x119
0x11A
0x11B
```

```
/ Code for MarieSim
        ORG   100
        Load  Addr
        Store Next
        Load  Num
        Subt  One
Store   Ctr
Loop,   Load  Sum
        AddI  Next
        Store Sum
        Load  Next
        Add   One
        Store Next
        Load  Ctr
        Subt  One
        Store Ctr
        Skipcond 000
        Jump  Loop
        Halt
Addr,   Hex   117
Next,   Hex   0
Num,    Dec   5
Sum,    Dec   0
Ctr,    Hex   0
One,    Dec   1
Dat,    Dec   -10
        Dec   15
        Dec   -20
        Dec   25
        Dec   30
```

```
/ Code for MarieSimR
        ORG   100
        Load  Addr
        Store Next
        Limm  -1
        Add   Num
        Store Ctr
Loop,   Load  Sum
        AddI  Next
        Store Sum
        Limm  1
        Add   Next
        Store Next
        Limm  -1
        Add   Ctr
        Store Ctr
        Skipcond 000
        Jump  Loop
        Halt
Addr,   Lab   Dat
Next,   Hex   0
Num,    Dec   5
Sum,    Dec   0
Ctr,    Hex   0
Dat,    Dec   -10
        Dec   15
        Dec   -20
        Dec   25
        Dec   30
```

Figure 2: Compute Sum of Numbers in An Array

It can be seen that both codes use indirect addressing to access each number in the array stored in the memory starting from the memory location *Dat*. Using MarieSim, users must calculate the memory address of *Dat* manually and then use this constant address where is needed. Note that the memory address of *Dat* is hexadecimal 0x117 because the code starts from 0x100 and each instruction occupies one memory location and each data occupies one memory location as well. On the other hand, using MarieSimR, user can use the label *Dat* where is needed by using the new assembler directive *Lab*. Then, the assembler will compute the memory address of *Dat*. The code related with this comparison is highlighted in red color in Figure 2.

Another comparison is about how to deal with immediate constant operand. The related code in this compassion is highlighted in the blue color in Figure 2. The new instruction *Limm* in MarieSimR allows to load a constant number into the accumulator *AC*, while in MarieSim, a constant number must be stored in memory before it can be loaded into *AC*. Therefore, the code for MarieSim must define a named constant One to hold the constant number 1 in order to add 1 or subtract 1. But the code for MarieSimR does not need to define such a named constant.

B. *Compute Binonacci Numbers*

The Fibonacci number can be computed by the formula

$$Fib(N) = \begin{cases} N & if\ N < 2 \\ Fib(N-1) + Fib(N-2) & if\ N \geq 2 \end{cases}$$

Four methods will be used to compute it. The first is using a main program with a loop, the second using a non-recursive function with global variables, the third using a non-recursive function with local variables, and the fourth using a recursive function. Note that the first two methods can be applied in both MarieSim and MarieSimR. But the last two methods can only be applied in MarieSimR.

1) *Using a Loop*

Figure 3 shows a C++ code that computes the Fibonacci number using the main program with a loop, where the number N is a user input. The control constructs used in this C++ code will be used (translated) in the assembly language programs later so that comparisons can be made. Note that variable A is used for storing *Fib*(N-2), B for storing *Fib*(N-1), C for storing *Fib*(N), and I is the loop control variable.

```
int main() {                    //compute Fib(N)
    int I, A, B, C, N
    std::cin >> N;              //get input N
    if (N < 2)
        C = N;
    else {
        A = 0; B = 1;
        for (I = 2; I <= N; I++) {
            C = B + A; A = B; B = C;
        }
    }
    std::cout << C;             //display result
    return 0;
}
```

Figure 3: C++ Code: Compute Fibonacci Number

Figure 4 shows the MarieSim code that computes Fibonacci numbers using a loop. It works fine in MarieSimR as well. But the label One is not necessary when using MarieSimR. It can be removed and hence instructions Load I and Add One should be replaced by Limm 1 and Add I. The variables I, N, C, B, and A have the same roles as those in C++ code. The program reads the input and stores it in both variables N and C. Then, it computes N-1. If N ≥ 2, it goes to Loop. Otherwise, it goes to Done. From the label Loop, it computes C=B+A, A=B, and B=C, increases I by 1, computes I-N, if I=N, it goes to Loop again, otherwise, it goes to Done. From the label Done, it loads and prints the result.

```
// Code for MarieSim and MarieSimR
            ORG     100
            Input               // Read input
            Store   N           // N=input
            Store   C           // C=N
            Subt    I           // AC = N-1
            SKIPCOND 800        // If N-1 > 0, goto Loop
            JUMP    Done        // Otherwise, goto Done
Loop,       Load    B           // AC=B
            Add     A           // AC=B+A
            Store   C           // C=B+A
            Load    B           // AC=B
            Store   A           // A=B
            Load    C           // AC=C
            Store   B           // B=C
            Load    I           // AC=1
            Add     One         // AC=I+1
            Store   I           // I=I+1
            SUBT    N           // AC=I-N
            SKIPCOND 400        // If I=N, goto Done
            Jump    Loop        // Otherwise, goto Loop
Done,       Load    C           // Load result
            Output              // Print result
            Halt                // Terminate program
// Variable Declarations
I,          DEC     1           // Loop control variable
N,          DEC     0           // N
C,          DEC     0           // Fib(N)
B,          DEC     1           // Fib(N-1)
A,          DEC     0           // Fib(N-2)
One,        DEC     1           // Constant 1
```

Figure 4: Compute Fibonacci Number Using a Loop

2) Using a Non-Recursive Function

Using Global Variables: Figure 5 shows the MarieSim code on the left and the MarieSimR code on the right. They both compute the Fibonacci numbers using a non-recursive function *Fib*. The global variables like in Figure 4 are defined and used in both the main routine and the subroutine *Fib*. The main routine gets the user input and stores it in N. Then it calls the subroutine *Fib* which gets its input from N and saves its result in C. After *Fib* returns, the main routine loads the result from C and prints the result. Because the global variables are used, the MarieSim code of the subroutine *Fib* shown in Figure 5 is very similar with the code shown in Figure 4.

Note that in MarieSim, the subroutine call instruction *JnS Fib* stores the return address at the memory location *Fib* and thus the first instruction of *Fib* is at the memory location *Fib*+1. So, the subroutine return instruction *JumpI Fib* gets the return address at the memory location *Fib* as well. On the other hand, in MarieSimR, the subroutine call instruction *Call Fib* stores the return address on top of stack and the subroutine return instruction *JR* also gets the return address from top of stack. So, the first instruction of *Fib* is just at the memory location *Fib*.

```
// Code for MarieSim
            ORG     100
            Input
            Store   N
            JnS     Fib
            Load    C
            Output
            Halt
// Subroutine Fib
Fib,        HEX     0
            Load    N
            Store   C
            Subt    I
            SKIPCOND 800
            JUMP    Done
Loop,       Load    B
            Add     A
            Store   C
            Load    B
            Store   A
            Load    C
            Store   B
            Load    I
            Add     One
            Store   I
            SUBT    N
            SKIPCOND 400
            Jump    Loop
Done,       JumpI   Fib
// Global Variable Declarations
I,          DEC     1
N,          DEC     0
C,          DEC     0
B,          DEC     1
A,          DEC     0
One,        DEC     1
```

```
// Code for MarieSimR
            ORG     100
            Input
            Store   N
            Call    Fib
            Load    C
            Output
            Halt
// Subroutine Fib

Fib,        Load    N
            Store   C
            Subt    I
            SKIPCOND 800
            JUMP    Done
Loop,       Load    B
            Add     A
            Store   C
            Load    B
            Store   A
            Load    C
            Store   B
            Limm    1
            Add     I
            Store   I
            SUBT    N
            SKIPCOND 400
            Jump    Loop
Done,       JR
// Global Variable Declarations
I,          DEC     1
N,          DEC     0
C,          DEC     0
B,          DEC     1
A,          DEC     0
```

Figure 5: Using Subroutine and Global Variables

Using Local Variables: Local variables are used inside a function only. When a function is called, its local variables are created and when the function is returned, its local variables are no longer available. In a general-purpose register architecture, the CPU registers are often the first choice for local variables. When there are not enough registers, local variables will be allocated their memory spaces from stack. In the accumulator-based architecture, there is no any general-purpose registers. So, the stack is the only choice for local variables. In fact, a stack frame can be created for each function to hold the return address, input arguments, local variables, output results, and so on. In general, the stack frame of a function will be created, maintained, and destroyed by both the calling routine and the called subroutine. Note that MarieSim does not support stack. So, the following discussion is for MarieSimR only.

Table 3 shows a stack frame of non-recursive function *Fib*. The stack frame contains six memory words in stack. The input N and the output result share one memory space in stack at $-1 or Mem[*SP*] -1. The return address is stored in the next location and it is followed by the local variables I, A, B, and C.

Table 3: Stack Frame of Non-Recursive Function Fib

Location in Stack	Memory Address	Used for
$-1	Mem[*SP*]-1	Input N / Output *Fib*(N)
$	Mem[*SP*]	Return Address
$+1	Mem[*SP*]+1	Local Variable I
$+2	Mem[*SP*]+2	Local Variable A
$+3	Mem[*SP*]+3	Local Variable B
S+4	Mem[*SP*]+4	Local Variable C

```
//Main code for MarieSimR
        ORG     100
        Input                   //Read input
        Store   N               //N=input
        Push*   N               //Push N on stack
        Call    Fib             //Call subroutine Fib
        Pop*    C               //Pop result into C
        Load    C               //Load result
        Output                  //Print result
        Halt                    //Terminate program
//Subroutine Fib
Fib,    Limm    1               //AC=1
        Store   $+1             //I=1
        Store   $+3             //B=1
        Limm    0               //AC=0
        Store   $+2             //A=0
        load    $-1             //AC=N
        Subt    $+1             //AC=N-1
        SKIPCOND 800            //If N-1 > 0 goto Loop
        Jump    Done            //Otherwise, goto Done
Loop,   Load    $+3             //AC=B
        Add     $+2             //AC=B+A
        Store   $+4             //C=B+A
        Load    $+3             //AC=B
        Store   $+2             //A=B
        Load    $+4             //AC=C
        Store   $+3             //B=C
        Limm    1               //AC=1
        Add     $+1             //AC=I+1
        Store   $+1             //I=I+1
        Subt    $-1             //AC=I-N
        SKIPCOND 400            //If I=N, goto Done
        Jump    Loop            //Otherwise, goto Loop
        Load    $+4             //AC=C
        Store   $-1             //Save result
Done,   JR
//Global Variable Declarations
N,      DEC     0               //N -- Input to Fib
C,      DEC     0               //Fib(N)-- Output from Fib
```

Figure 6: Using Subroutine and Local Variables

** Push N and Pop C should be expanded according to Table 2 before assembling and running*

Figure 6 shows the code that computes Fibonacci numbers using a non-recursive function with local variables. As shown Figure 6, the main routine gets the user input and stores it in N. Then it pushes the input N on top of stack right before it calls the subroutine Fib. Note that when a subroutine is called, its return address is pushed on top of stack by the subroutine call instruction Call. Because the push operation increases $ (or Mem[SP]) by 1 (see Table 2), right after entering the subroutine Fib, the return address is at $ (or Mem[PS]) and the input N is at $-1 (or Mem[*SP*]-1). Then, the subroutine Fib uses four memory locations $+1, $+2, $+3, and $+4 in the stack area for its local variables I, A, B, and C, respectively. Now, the stack frame for the subroutine Fib as shown in Table 3 is created.

The code of subroutine Fib in Figure 6 has three parts: (1) Initialize the local variables I, A, and B (highlighted in red color), (2) Compute Fib(N) (highlighted in blue color), and (3) Copy the result to memory location $-1, which overwrites the input N, right before the return (highlighted in crison color).

Note that the subroutine return instruction JR copies the return address from top of stack into *PC* and then decreases the stack pointer $ (or Mem[SP]) by 1. Therefore, right after the subroutine Fib returns, the return result is on top of stack (at $). So, the main routine can pop the result from top of stack. Please note that the stack frame of Fib is now completely destroyed. This is, the stack pointer now has the value as that right before the input N is pushed on stack. Finally, the main routine loads and prints the result.

3)Using a Recursive Function

Now, the subroutine Fib will be implemented as a recursive function. Like for the non-recursive function, we need to design a stack frame for the recursive function. First, we are going to need one memory slot of stack to store both input N and output Fib(N). Before calling the function Fib, the input N must be on top of stack (at $). But, right after calling Fib, stack grows and the return address is now on top of stack (at $). So, N now is at $-1. When N < 2, Fib(N) = N. This means that the output is the input and it is already in the right location in stack as desired. When N>2, Fib(N) = Fib(N-1) + Fib(N-2). This means that we need to call Fib twice to compute Fib(N-1) and Fib(N-2) inside the Fib(N). Therefore, we need two more local memory spaces inside the stack frame of Fib(N) to store (N-1)/Fib(N-1) and (N-2)/Fib(N-2). So, the stack frame for Fib(N) needs four memory spaces in stack region as shown in Table 4.

Table 4: Stack Frame of Recursive Function Fib(N) right after calling Fib(N) and right before returning from Fib(N)

Location in Stack	Memory Address	Used for
$-1	Mem[*SP*]-1	Input N / Output *Fib*(N)
$	Mem[*SP*]	Return Address
$+1	Mem[*SP*]+1	Input N-1 / Output *Fib*(N-1)
$+2	Mem[*SP*]+2	Input N-2 / Output *Fib*(N-2)

Figure 7 shows the code that computes Fibonacci numbers using a recursive function. As shown in Figure 7, the main code and the global variable declarations are exactly the same as those in Figure 6.

One important fact is that right before calling Fib(N-1) inside subroutine Fib(N), N-1 must be pushed on stack. This will increase the value of $ by 1. This means that right before calling Fib(N-1), the memory addresses of the stack frame for Fib(N) are shifting in terms of $ as shown in Table 5. The code to push N-1 on stack are highlighted in blue code as shown in Figure 7. Please note that we need to make sure that right after Fib(N-1) returns, its output Fib(N-1) overwrites its input N-1 and remains on top of stack, i.e., at $, as shown in Table 5.

Similarly, right before calling Fib(N-2) inside Fib(N), N-2 must be pushed on top of stack. So, the memory address of the stack frame for Fib(N) is shifting in terms of $ again as shown in Table 6. Meanwhile, when Fib(N-2) returns, its result Fib(N-2) replaces its input N-2 on top of stack as shown in Table 6 again. The code to push N-2 on stack is highlighted in red color as shown in Figure 7.

```
//Main code for MarieSimR
            ORG     100
            Input               //Read input
            Store   N           //N=input
            Push*   N           //Push N on stack
            Call    Fib         //Call subroutine Fib
            Pop*    C           //Pop result into C
            Load    C           //Load result
            Output              //Print result
            Halt                //Terminate program
Fib,        Limm    -1          //AC=-1
            Add     $-1         //AC = N-1
            Skipcond 800        //If N > 1 do Recursion
            jump    Done        //Otherwise, goto Done
Recurs,     Store   $+1         //Store N-1 to $+1
            IncSP   1           //Increase SP ($) by 1
            Call    Fib         //Call F(N-1)
            Limm    -2          //AC = -2
            Add     $-2         //AC = N-2
            Store   $+1         //Store N-2 to $+1
            IncSP   1           //Increase SP ($) by 1
            Call    Fib         //Call F(N-2)
            IncSP   -2          //Restore SP (decrease by 2)
            Load    $+1         //AC = Fib(N-1)
            Add     $+2         //AC = F(N-1) + F(N-2) = F(N)
            Store   $-1         //Store Fib(N) to $-1
Done,       JR
//Global Variable Declarations
N,          DEC     0           //N -- Input to Fib
C,          DEC     0           //f(N)-- Output from Fib
```

Figure 7: Using Recursive Subroutine
** Push N and Pop C should be expanded according to Table 2 before assembling and running*

Finally, right before the Fib(N) returns, the stack frame must be restored as shown in Table 4. This means that its return address must be on top of stack, i.e., at $ and its result Fib(N) which is Fib(N-1) + Fib(N-2) must be stored at $-1. The code to accomplish these is highlighted in crimson color as shown in Figure 7.

Table 5: Stack Frame of Recursive Function Fib(N) right before calling Fib(N-1) and right after returning from Fib(N-1)

Location in Stack	Memory Address	Used for
$-2	Mem[*SP*]-2	Input N / Output *Fib*(N)
$-1	Mem[*SP*]-1	Return Address
$	Mem[*SP*]	Input N-1 / Output *Fib*(N-1)
$+1	Mem[*SP*]+1	Input N-2 / Output *Fib*(N-2)

Therefore, after Fib(N) returns, its result is on top of stack because the subroutine return instruction pops off the return address from top of stack and thus the value of stack pointer $ is decreased by 1. Then, the main code can pop off the result from top of stack.

Table 6: Stack Frame of Recursive Function Fib(N) right before calling Fib(N-2) and right after returning from Fib(N-2)

Location in Stack	Memory Address	Used for
$-3	Mem[*SP*]-3	Input N / Output *Fib*(N)
$-2	Mem[*SP*]-2	Return Address
$-1	Mem [*SP*]-1	Input N-1 / Output *Fib*(N-1)
$	Mem [*SP*]	Input N-2 / Output *Fib*(N-2)

As you may already notice, after the main code pops off the result from the stack, the value of stack pointer $ is restored like nothing happened. The stack frame of Fib(N) is now destroyed.

IV. Conclusions

In this paper, MarieSimR, a revision to MarieSim [1], is presented. It adds the support of the stack by using a reserved memory slot to store the stack pointer. The revised instruction for subroutine call will push the return address on top of stack and the revised instruction for subroutine return will pop off the return address from top of stack. Meanwhile, the stack-relative address is added. Therefore, a stack frame can be created for a subroutine to support using local variables and recursions. A new instruction for increasing or decreasing the stack pointer is also added to facilitate the stack operations. In addition, a new instruction for loading an immediate constant integer into the accumulator AC is also added to replace the MARIE instruction Clear which loads 0 into the accumulator AC. Finally, a new assembler directive is added to support to define a label to hold the address of another label symbolically.

This new revision of MarieSim makes it closer to a real machine. At the same time, they do not require to change the MarieSim user interface. Users can edit, assemble, run, and debug their code for MarieSimR just like for MarieSim. Users can observe the stack pointer and the contents in stack using the memory monitor panel of MarieSim environment.

Please note that when the stack-relative addressing is used, the operand address displayed in the instruction panel of the MarieSim user interface will be $+*Offset*, where *Offset* is a hexadecimal number which is equal to a 10-bit integer in 2's complement representation. For examples, the operand address of Load $+29 is $+01D, and the operand address of Load $-29 is $+3E3. Please also note that the machine instruction of Load $+29 is 16-bit hexadecimal 1B1D and that machine instruction of Load $-29 is 16-bit hexadecimal 1FE3.

Similarly, the immediate operand in the instructions Limm and IncSP is displayed as a hexadecimal number which is equal to a 12-bit integer in 2's complement representation.

A webpage [6] has been created for students to study this revised MARIE computer simulator. Users can use it for more interesting assembly language programming assignments that requires stack and/or recursion. Students can also modify it to make it better. For examples, adding pseudo-instructions push and pop. Checking if the user program will overwrite the stack. Please note that the new instruction IncSP Imm is not necessary because it can be achieved by the following three instructions: Limm imm, Add SP, and Store SP. This means that we could add a pseudo-instruction to perform IncSP Imm.

REFERENCES

[1] Linda Null and Julia Lobur, MarieSim: The MARIE computer simulator, Journal on Educational Resources in Computing, Volume 3, Issue 2, June 2003, pp 1–29.

[2] Linda Null and Julia Lobur, The essentials of computer organization and architecture, 5th Edition, Jones & Bartlett Learning, 2019

[3] Xuejun Liang, A survey of hands-on assignments and projects in undergraduate computer architecture courses, in Proceedings of International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 07), December 3-12, 2007.

[4] Xuejun Liang, Computer Architecture Simulators for Different Instruction Formats, in the proceedings of The 6th Annual Conference on Computational Science and Computational Intelligence (CSCI 2019), pp. 806-811, Las Vegas, Nevada, USA, Dec 05-07, 2019

[5] Xuejun Liang, More on Computer Architecture Simulators for Different Instruction Formats, in the proceedings of 2020 International Conference on Computational Science and Computational Intelligence (CSCI 2020), pp. 910-916, Las Vegas, Nevada, USA, Dec 16-18, 2020.

[6] Xuejun Liang, MarieSimR: The MARIE Computer Simulator Revision Webpage, the last access date: Nov 10, 2021, available at https://www.cs.csustan.edu/~xliang/Courses/MarieSimRWeb