# MarieSimR: The MARIE Computer Simulator Revision

Xuejun Liang
Department of Computer Science
California State University - Stanislaus

Fall 2021

# Introduction

- MarieSim is the MARIE computer simulator
  - MARIE is an accumulator-based computer model used in the popular textbook “The essentials of computer organization and architecture”.
  - Used for assembly language programming exercises.
- But, MarieSim is too simple and thus unable to support some important concepts in computer architecture:
  - No immediate addressing mode
  - No stack and thus its subroutine has no local variables and can not be recursive.
  - Can not define a variable to hold the address of another variable symbolically.
- In order to solve these problems, a revision to MarieSim, called MarieSimR, is developed.

# Outline

- Design and Implementation of MarieSimR
- Assembly Language Program Examples
  - to illustrate how to use MarieSimR
  - to compare MarieSim and MarieSimR

# The Goal of This Revision

- The goal of this work is to revise MARIE to support the stack and recursive subroutines without changing its architectural characteristics and its microarchitecture for control unit
- The MARIE architecture has the following characteristics:
  - Binary, two's complement data representation.
  - Stored program, fixed word length data and instructions.
  - 4K words of word-addressable main memory.
  - 16-bit data words.
  - 16-bit instructions, 4 for the opcode and 12 for the address.
  - A 16-bit arithmetic logic unit (ALU).
  - Seven registers for control and data movement.

# MarieSimR: What's New?

- 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. So, recursive subroutines are supported.
- A new instruction for increasing or decreasing the value of the stack pointer is added to facilitate the push and pop operations.
- A new instruction for loading an immediate constant into the accumulator is added to replace the clear instruction.
- A new assembler directive is added to support for defining a label to hold the address of another label symbolically

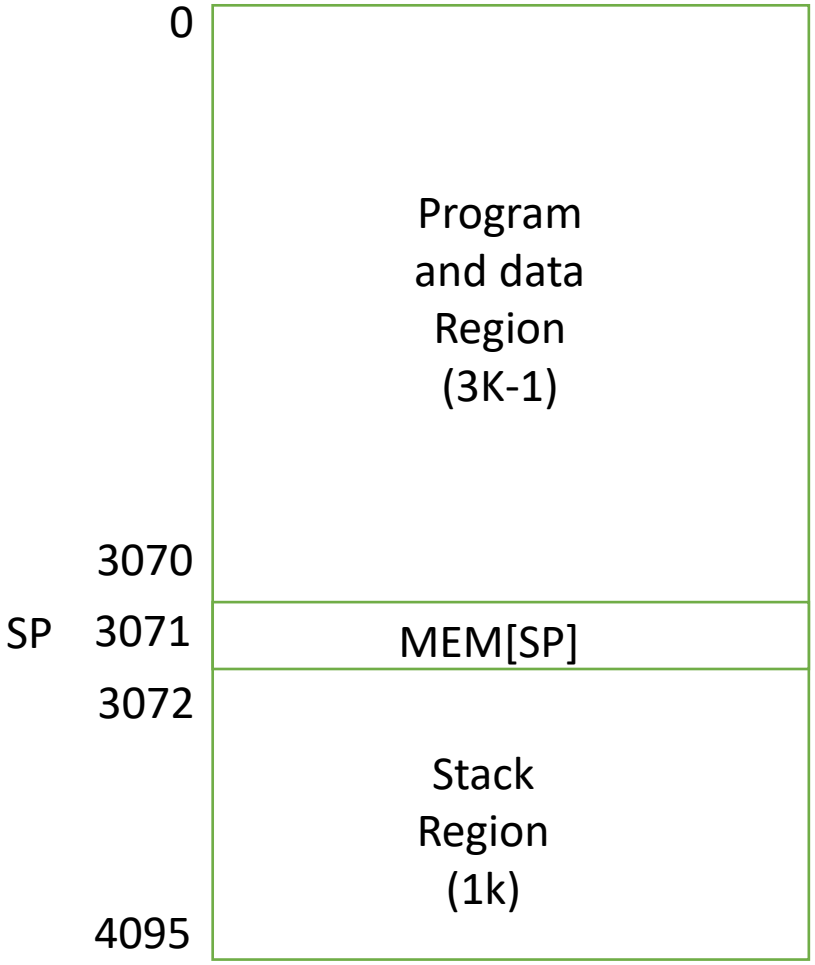
# MARIE Instruction Set and Revision

| 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 |

- X is either a hexadecimal literal or a label (symbol) and is used as a memory address
- Mem[X] represents the content at the memory location X.
- Imm is a 12-bit decimal constant integer.
- When using stack-relative addressing, X has the format: $±offset
  - $ represents the value of the stack pointer
  - Offset is a 10-bit decimal constant integer.

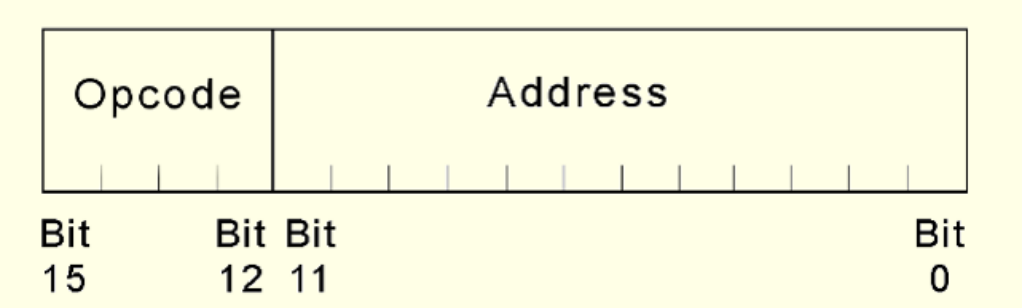
# MARIE Memory Map with Stack

- The MARIE architecture has 4K words of word-addressable main memory.
- The stack grows towards the high memory address end and starts from 3072.
- The stack occupies 1K out of 4K MARIE word-addressable memory space.
- The stack pointer is located at memory location 3071

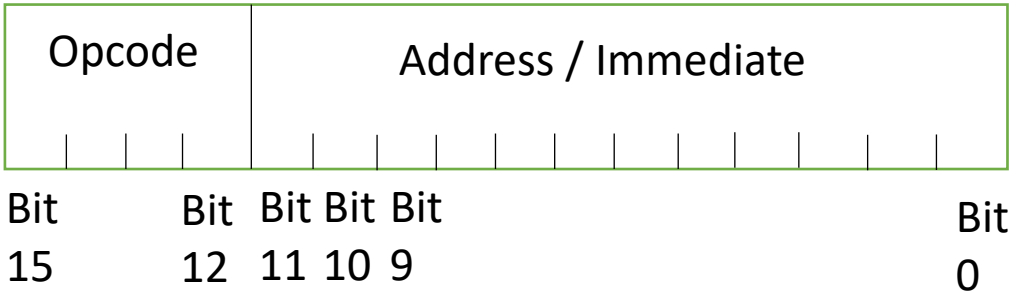


# Instruction Encoding Format

MARIE




MARIE Revision




Instructions Limm Imm and IncSP Imm store the 12-bit integer in 2's complement inside the instruction from Bit 0 to Bit 11.

If Bit 11 is 1 and Bit 10 is 1, then the stack-relative addressing is used and the offset address is stored inside the instruction from Bit 0 to Bit 9.

# Push and Pop Operations

| Push X | Meaning |
|---|---|
| IncSP 1 | Mem[SP] ← Mem[SP] + 1 |
| Load X | AC ← Mem[X] |
| StoreI SP | Mem[Mem[SP]] ← AC |

| Pop X | Meaning |
|---|---|
| loadI SP | AC ← Mem[Mem[SP]] |
| Store X | Mem[X] ← AC |
| IncSP -1 | Mem[SP] ← Mem[SP] - 1 |

The instruction IncSP Imm is not necessary as
It can be replaced by three other instructions

| IncSP Imm |
|---|
| Limm Imm |
| Add SP |
| Store SP |

# MARIE Directives and Extension

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

# Two Programming Examples

- Example 1: Compute sum of numbers in an array.
  - Loop through an array
  - Compare MarieSim and MarieSimR
- Example 2: Compute Fibonacci number Fib(N)
  - Using Loop
  - Using Subroutine with global variables
  - Using Subroutine with local variables
  - Using Recursive subroutine

Need to use Stack frame

# Example 1: Using loop to add five numbers in array, save result to Sum

```
/ Code for MarieSim and MarieSimR
      ORG      100    /Program starts from 0x100
      Load     Addr   /Load address of first number to be added
      Store    Next   /Store this address is our Next pointer
      Load     Num    /Load the number of items to be added
      Subt     One    /Decrement
      Store    Ctr    /Store this value in Ctr to control looping
Loop, Load     Sum    /Load the Sum into AC
      AddI     Next   /Add the value pointed to by location Next
      Store    Sum    /Store this sum
      Load     Next   /Load Next
      Add      One    /Increment by one to point to next address
      Store    Next   /Store in our pointer Next
      Load     Ctr    /Load the loop control variable
      Subt     One    /Subtract one from the loop control variable
      Store    Ctr    /Store this new value in loop control variable
      Skipcond 000    /If control variable < 0, skip next instruction
      Jump     Loop   /Otherwise, go to Loop
      Halt            /Terminate program
Addr, Hex      117    /Numbers to be summed start at location 117
Next, Hex      0      /A pointer to the next number to add
Num,  Dec      5      /The number of values to add
Sum,  Dec      0      /The sum
Ctr,  Hex      0      /The loop control variable
One,  Dec      1      /Used to increment and decrement by 1
Dat,  Dec      -10    /The values to be added together
      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
```

# Example 2: Compute Fibonacci Number fib(N), Where N is an Input

Mathematics formula

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

C++ code using a loop

```
int I, A, B, C, N

cin >> N;
if (N < 2)
  C = N;
else {
  A = 0; B = 1;
  for (I = 2; I <= N; I++) {
     C = B + A; A = B; B = C;
  }
}
cout << C;
```

C = fib(N)
B = fib(N-1)
A = fib(N-2)

# Method 1 (Using Loop)

```
// 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
    JUMP L3
L2, 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, done
    Jump      L2        // otherwise, continue
L3, Load      C         // print result
    Output
    Halt                // stop
// Variable Declarations
I,   DEC      1         // index
N,   DEC      0         // N
C,   DEC      0         // f(N)
B,   DEC      1         // f(N-1)
A,   DEC      0         // f(N-2)
One, DEC      1         // Used for increment by 1
```

```
// Code for 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
    JUMP      L3
L2, 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
    LIMM      1         // AC=1
    ADD       I         // AC=I+1
    Store     I         // I=I+1
    SUBT      N         // AC=I-N
    SKIPCOND  400       // if I=N, done
    Jump      L2        // otherwise, continue
L3, Load      C         // print result
    Output
    Halt                // stop
// Variable Declarations
I,   DEC      1         // index
N,   DEC      0         // N
C,   DEC      0         // f(N)
B,   DEC      1         // f(N-1)
A,   DEC      0         // f(N-2)
```

# Example 2: C++ Code with Using Function

```
int N, C;

int fib(int N) {
  int I, A, B, C;
  if (N < 2)
    C = N;
  else {
    A = 0; B = 1;
    for (I = 2; I <= N; I++) {
      C = B + A; A = B; B = C;
    }
  }
  return C;
}

cin >> N;
C = fib(N);
cout << C;
```

```
int N, C;

int fib(int N) {
  if (N < 2)
    return N;
  else
    return fib(N-1)+fib(N-2);
}

cin >> N;
C = fib(N);
cout << C;
```

Recursive function

Non-recursive function

# Method 2 (Using Function and Global Variables)

```
// Code for MarieSim
    ORG 100
    Input           //Read input
    Store    N      //N=input
    JNS      Fib    //Call subroutine Fib
    Load     C      //Print result
    Output
    Halt            //Terminate
// Subroutine Fib
Fib, HEX     0      //for storing return addr
    Load     N
    Store    C      //C=N
    Subt     I      //AC = N-1
    SKIPCOND 800    //if N-1 > 0
    JUMP     L3
L2, 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, done
    Jump     L2     //otherwise, continue
L3, JumpI    Fib
// Global Variable Declarations
I,  DEC      1      //index
N,  DEC      0      //N -- input to Fib
C,  DEC      0      //f(N)-- output from Fib
B,  DEC      1      //f(N-1)
A,  DEC      0      //f(N-2)
One, DEC      1     //used for increment by 1
```

```
// Code for MarieSimR
    ORG 100
    Input           //Read input
    Store    N      //N=input
    Call     Fib    //Call subroutine Fib
    Load     C      //Print result
    Output
    Halt            //Terminate
// Subroutine Fib
Fib, Load    N
    Store    C      //C=N
    Subt     I      //AC = N-1
    SKIPCOND 800    //if N-1 > 0
    JUMP     L3
L2, 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
    Limm     1      //AC=1
    ADD      I      //AC=I+1
    Store    I      //I=I+1
    SUBT     N      //AC=I-N
    SKIPCOND 400    //if I=N, done
    Jump     L2     //otherwise, continue
L3, JR
// Global Variable Declarations
I,  DEC      1      //index
N,  DEC      0      //N -- input to Fib
C,  DEC      0      //f(N)-- output from Fib
B,  DEC      1      //f(N-1)
A,  DEC      0      //f(N-2)
```

# Method 3 (Using Function and Local Variables)

Stack frame of Fib(N)

| Location in Stack | Memory Address | Used for |
|---|---|---|
| **$-1** | Mem[SP]-1 | Input N / Output Fib(N) |
| **$** | Mem[SP] | Return Address |
| **$+1** | Mem[SP]+1 | Local Variable I |
| **$+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 (need to expand)
    Call    Fib     //Call subroutine Fib
    Pop     C       //Pop result into C (need to expand)
    Load    C       //print result
    Output
    Halt            //terminate program
```

```
//Subroutine Fib
Fib, Limm     1
     Store    $+1     //I=1
     Store    $+3     //B=1
     Limm     0
     Store    $+2     //A=0
     load     $-1     //AC=N
     SUBT     $+1     //N-1
     SKIPCOND 800     //if N-1 > 0
     JUMP     L3
L2,  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
     ADD      $+1     //AC=I+1
     Store    $+1     //I=I+1
     SUBT     $-1     //AC=I-N
     SKIPCOND 400     //if I=N, done
     Jump     L2      //otherwise, continue
     Load     $+4     //AC=C
     Store    $-1     //Save result
L3, JR
//Global Variable Declarations
N,   DEC      0       //N -- input to Fib
C,   DEC      0       //f(N)-- output from Fib
```

Initialize the local variables

Compute Fib(N)

Copy the result to memory location $-1

# The values in stack at several important timestamps

| (1) Right before calling Fib | Memory address | Used for |
|---|---|---|
| $ | Mem[SP] | Input N |

| (2) Right after calling Fib | Memory address | Used for |
|---|---|---|
| $-1 | Mem[SP]-1 | Input 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 |
| $+4 | Mem[SP]+4 | Local variable C |

| (3) Right before Fib return | Memory address | Used for |
|---|---|---|
| $-1 | Mem[SP]-1 | 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 |
| $+4 | Mem[SP]+4 | Local variable C |

| (4) Right after Fib return | Memory address | Used for |
|---|---|---|
| $ | Mem[SP] | Output Fib(N) |

# Method 4 (Using Recursive Function)

Stack Frame of 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) |

```
//Main code for MarieSimEx
    ORG     100
    Input           //Read input
    Store   N       //N=input
    Push    N       //Push N on stack (need to expand)
    JnS     Fib     //Call subroutine Fib
    Pop     C       //Pop result into C (need to expand)
    Load    C       //print result
    Output
    Halt            //terminate program
```

```
//Subroutine Fib
Fib, Limm -1
    Add       $-1     // AC = N-1
    Skipcond  800     // if N > 1
    jump      L1      // Done
    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      // Decrease SP ($) by 2 (restore SP)
    Load      $+1     // AC = Fib(N-1)
    Add       $+2     // AC = F(N-1) + F(N-2) = F(N)
    Store     $-1     // store Fib(N) to $-1
L1, JR
//Global Variable Declarations
N,  DEC       0       //N -- input to Fib
C,  DEC       0       //f(N)-- output from Fib
```

# The values in stack at several important timestamps (1)

```
Push    N       //Push N on stack (need to expand)
```

| (1) Right before calling Fib(N) | Memory address | Used for |
|---|---|---|
| $ | Mem[SP] | Input N |

```
Call    Fib     //Call subroutine Fib
```

| (2) Right after calling Fib(N) | Memory address | Used for |
|---|---|---|
| $-1 | Mem[SP]-1 | Input N |
| $+0 | Mem[SP] | Return address of Fib(N) |
| $+1 | Mem[SP]+1 | |
| $+2 | Mem[SP]+2 | |

```
Store   $+1     // Store N-1 to $+1
IncSP   1       // Increase SP ($) by 1
```

| (3) Right before calling Fib(N-1) | Memory address | Used for |
|---|---|---|
| $-2 | Mem[SP]-2 | Input N |
| $-1 | Mem[SP]-1 | Return address of Fib(N) |
| $ | Mem[SP] | N-1 |
| $+1 | Mem[SP]+1 | |

# The values in stack at several important timestamps (2)

| (4) Right after Fib(N-1) return | Memory address | Used for |
|---|---|---|
| $-2 | Mem[SP]-2 | Input N |
| $-1 | Mem[SP]-1 | Return address |
| $ | Mem[SP] | Fib(N-1) |
| $+1 | Mem[SP]+1 | |

```
Limm    -2    // AC = -2
Add     $-2   // AC = N-2
Store   $+1   // Store N-2 to $+1
IncSP   1     // Increase SP ($) by 1
```

| (5) Right before calling Fib(N-2) | Memory address | Used for |
|---|---|---|
| $-3 | Mem[SP]-3 | Input N |
| $-2 | Mem[SP]-2 | Return address |
| $-1 | Mem[SP]-1 | Fib(N-1) |
| $ | Mem[SP] | N-2 |

| (6) Right after Fib(N-2) return | Memory address | Used for |
|---|---|---|
| $-3 | Mem[SP]-3 | Input N |
| $-2 | Mem[SP]-2 | Return address |
| $-1 | Mem[SP]-1 | Fib(N-1) |
| $ | Mem[SP] | Fib(N-2) |

# The values in stack at several important timestamps (3)

| (6) Right after Fib(N-2) return | Memory address | Used for |
|---|---|---|
| $-3 | Mem[SP]-3 | Input N |
| $-2 | Mem[SP]-2 | Return address |
| $-1 | Mem[SP]-1 | Fib(N-1) |
| $ | Mem[SP] | Fib(N-2) |

```
IncSP    -2     // Decrease SP ($) by 2 (restore SP)
Load     $+1    // AC = Fib(N-1)
Add      $+2    // AC = F(N-1) + F(N-2) = F(N)
Store    $-1    // store Fib(N) to $-1
```

| (7) Right before Fib(N) return | Memory address | Used for |
|---|---|---|
| $-1 | Mem[SP]-1 | Fib(N) |
| $ | Mem[SP] | Return address |
| $+1 | Mem[SP]+1 | Fib(N-1) |
| $+2 | Mem[SP]+2 | Fib(N-2) |

```
L1, JR
```

| (8) Right after Fib(N) return | Memory address | Used for |
|---|---|---|
| $ | Mem[SP] | Fib(N) |

```
Pop      C      //Pop result into C (need to expand)
```