More on Computer Architecture Simulators for Different Instruction Formats

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

***Abstract*—Several simple computer architecture simulators are developed and implemented for four different instruction formats, including stack-based, accumulator-based, two-address, and three-address machines. The simulators for the first two machines have been reported in [1]. This paper will present details for the remaining two instruction formats. These simulators can be used to assemble and run assembly language programs on these architectures. Several examples are given to illustrate how to develop assembly language programs to deal with arrays, loops, subroutines, and recursions on these different computer architectures. Sizes and performances of these assembly language programs are discussed. Students will have a better understanding of computer architectures by using these simulators on their assembly language programming assignments. In addition, students can also modify these simulators to add more instructions or pseudo-instructions.**

***Keywords*—Computer Architecture, Simulator, Instruction Format, 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 [2]. There are many computer architectures with different instruction formats such as stack-based, accumulator-based, two-address, or three-address machine. But, in general, only one architecture will be chosen for teaching assembly language programming in a computer architecture class or textbook. David A. Patterson and John L. Hennessy uses MIPS in their textbook [3]. Kip Irvine teaches x86 in his textbook [4]. Linda Null and Julia Lobur uses the accumulator-based architecture and the MARIE simulator [5]. On the other hand, although there are numerous processor simulators available [6], most simulators are for the research purpose and using them needs a big learning curve. It is certainly desirable to have various simple simulators, each for one major computer processor architecture, so that students can program and compare these processors.

To this end, six simple computer architecture simulators are designed and implemented for four different instruction formats, including stack-based, accumulator-based, two-address (2A), and three-address (3A) machines. The simulators for the first two instruction formats have been reported in [1]. This paper will present details for the remaining two instruction formats. Both memory-to-memory (M2M) and register-to-register (R2R) architectures are implemented for both 2A and 3A machines. These simulators can be used to assemble and run assembly language programs on these simulated computer architectures. Several simple applications are used to illustrate how to develop assembly language programs to deal with arrays, loops, stacks, subroutines, and recursions on these computer architectures. At the meantime, comparisons of sizes and performances among these assembly language programs are discussed.

Students will have a better understanding of computer architectures by using these simulators on their assembly language programming exercises. Students can also modify these simulators to add more instructions or pseudo-instructions. In addition, these simulated machines can also serve as the compiler's target machines for the code generation practice.

In the rest of this paper, the simulated instruction sets are presented in Second II. Assembly language programming examples using these simulators are described in Second III. Comparisons of sizes and performances among these assembly language programs are reported in Section IV. Finally, Second V concludes the paper.

II. Instruction Sets of Simulated Machines

In simulated machines, all data are 32 bits and all addresses and immediate data are 16 bits. All instructions in one simulated machine are of the fixed word length which may be different for different machines. Two separate memories are used for data and instructions. Data are word addressable and a datum word is 32 bits. Instruction is also word addressable, but an instruction word may not be 32 bits and it will depend on its particular instruction format of simulated machine. So, each simulated machine has 64K 32-bit words of data memory and 64K instruction words of instruction memory.

In this paper, the notation M[A] represents the memory content at memory address A. The acronym *Imm* stands for 16-bit immediate number, *PC* for program counter, and *SP* for stack pointer.

In all simulated machines, stack will grow toward higher memory address. The stack pointer *SP* is a register in R2R architectures, while it is a reserved memory location in M2M architectures. Meanwhile, there are three additional reserved memory locations in M2M architectures. They are *ZERO* for constant 0, *INPUT* for input data, and *OUTOUT* for output data.

A. Two-Address Memory-to-Memory Instruction Set

The instruction set of the simulated 2A M2M machine contains 19 instructions as shown in Table 1. It includes 1 load immediate (LI) instruction, 6 integer arithmetic instructions, 1 load and 1 store (GET and PUT) instructions, 5 branch instructions, 1 subroutine call and 1 subroutine return (JNS and JR) instructions, 1 input and 1 output (READ and PRNT) instructions, and finally, 1 stop instruction to terminate the program.

In Table 1: 2A M2M Instruction Set, the symbol $\leftarrow$ means assignment. The letters A and B indicate memory locations. They can be a global variable name or a local variable in the form of \$+*Imm* whose memory address is M[*SP*]+*Imm*. So, the instruction ADD A \$+4 means M[A] $\leftarrow$ M[A] + M[M[*SP*]+4]. Note that M[*SP*] is the content of *SP* and is usually pointing to the top of stack. JNS Label will save *PC* on stack and assign Label to *PC*. JR will assign the top content on stack to *PC* and remove it from stack.

Table 1: 2A M2M Instruction Set

	Instruction	Meaning
0	LI A *Imm*	M[A] $\leftarrow$ *Imm*
1	ADDI A *Imm*	M[A] $\leftarrow$ M[A]+*Imm*
2	ADD A B	M[A] $\leftarrow$ M[A]+M[B]
3	SUB A B	M[A] $\leftarrow$ M[A]-M[B]
4	MUL A B	M[A] $\leftarrow$ M[A]*M[B]
5	DIV A B	M[A] $\leftarrow$ M[A]/M[B]
6	REM A B	M[A] $\leftarrow$ M[A]%M[B]
7	GET A B	M[A] $\leftarrow$ M[M[B]]
8	PUT A B	M[M[B]] $\leftarrow$ M[A]
9	GOTO Label	*PC* $\leftarrow$ Label
10	BEQZ A Label	If M[A] = 0 GOTO Label
11	BNEZ A Label	If M[A] $\neq$ 0 GOTO Label
12	BGEZ A Label	If M[A] $\geq$ 0 GOTO Label
13	BLTZ A Label	If M[A] < 0 GOTO Label
14	JNS Label	M[*SP*] = M[*SP*]+1, M[M[*SP*]] = *PC*, & *PC* $\leftarrow$ Label
15	JR	*PC* $\leftarrow$ M[M[*SP*]] & M[*SP*] = M[*SP*]-1
16	READ	M[*INPUT*] $\leftarrow$ Input
17	PRNT	Display M[*OUTPUT*] on screen
18	STOP	Terminate program

Table 2 shows four pseudo-instructions in which MOVE and NEG are implemented by using four instructions each, while PUSH and POP using 2 instructions each.

Table 2: Pseudo-Instructions of 2A M2M Machine

	Pseudo-Instruction	Meaning
1	MOVE C A	M[C] $\leftarrow$ M[A]
2	NEG A	M[A] $\leftarrow$ -M[A]
3	POP A	M[A] $\leftarrow$ M[M[*SP*]] M[*SP*] $\leftarrow$ M[*SP*] - 1
4	PUSH A	M[*SP*] $\leftarrow$ M[*SP*] + 1 M[M[*SP*]] $\leftarrow$ M[A]

B. Three-Address Memory-to-Memory Instruction Set

The instruction set of the simulated 3A M2M machine has 19 instructions in which instructions 0, 9, 14-18 are the exactly same with those of the simulated 2A M2M machine and the remaining instructions are listed in Table 3. Note that if the memory location C in instructions 1-6 in Table 3 is all replaced by A and the C in instructions 7, 8, 10-13 in Table 3 is all replaced by the reserved memory location *ZERO*, these instructions are the same with those of the 2A M2M machine. Therefore, the 2A M2M instruction set is a subset of the 3A M2M instruction set.

Please note that pseudo-instructions in Table 2 are also implemented in the 3A M2M machine, but MOVE and NEG are implemented by using only one instruction each. In addition, four more pseudo-instructions BEQZ, BNEZ, BGEZ, and BLTZ are introduced as a short version of responding instructions. For example, BEQZ A Label is really BEQ A ZERO Label.

Table 3: Remaining Instructions of 3A M2M Instruction Set

	Instruction	Meaning
1	ADDI A C *Imm*	M[A] $\leftarrow$ M[C]+*Imm*
2	ADD A C B	M[A] $\leftarrow$ M[C]+M[B]
3	SUB A C B	M[A] $\leftarrow$ M[C]-M[B]
4	MUL A C B	M[A] $\leftarrow$ M[C]*M[B]
5	DIV A C B	M[A] $\leftarrow$ M[C]/M[B]
6	REM A C B	M[A] $\leftarrow$ M[C]%M[B]
7	GET A C B	M[A] $\leftarrow$ M[C+M[B]]
8	PUT A C B	M[C+M[B]] $\leftarrow$ M[A]
10	BEQ A C Label	If M[A] = M[C] GOTO Label
11	BNE A C Label	If M[A] $\neq$ M[C] GOTO Label
12	BGE A C Label	If M[A] $\geq$ M[C] GOTO Label
13	BLT A C Label	If M[A] < M[C] GOTO Label

C. Two-Address Register-to-Register Instruction Set

In the R2R architectures, 32 general purpose registers and the MIPS register convention [4] as shown in Table 4 are used.

Table 4: MIPS Register Convention [4]

Name	Number	Usage
\$zero	\$0	The constant value 0
\$at	\$1	Reserved for assembler
\$v0-\$v1	\$2-\$3	Expression evaluation and results of a function
\$a0-\$a3	\$4-\$7	Argument 1-4
\$t0-\$t7	\$8-\$15	Temporary (not preserved across call)
\$s0-\$s7	\$16-\$23	Saved temporary (preserved across call)
\$t8-\$t9	\$24-\$25	Temporary (not preserved across call)
\$k0-\$k1	\$26-\$27	Reserved for OS kernel
\$gp	\$28	Pointer to global area
\$sp	\$29	Stack pointer
\$fp	\$30	Frame pointer
\$ra	\$31	Return address (used by function call)

The instruction set of the simulated 2A R2R machine has 19 instructions which can be simply obtained by replacing memory locations A and B in the Instruction column of Table 1 with registers R and R1 respectively, and replacing M[A] and M[B] in the Meaning column of Table 1 with R and R1 respectively as well. For examples, ADD R R1 means R $\leftarrow$ R+R1 and GET R R1 means R $\leftarrow$ M[R1]. But, the instructions 14-17 in 2A R2R have different meaning with their counterparts in Table 1 and are listed in Table 5. The major differences between R2R and M2M architectures are (1) the subroutine call instruction JNS saves the return address in the register \$ra in R2R, while it saves the return address on stack in M2M, (2) the subroutine return instruction JR gets the return address from \$ra in R2R, while it gets the return address from stack in M2M, (3) the input goes to the register \$v0 in R2R, while it goes to the reserved memory location *INPUT* in M2M, and (4) the content for display is in the register \$a0 in R2R, while it is in the reserved memory location *OUTPUT*. Note that this architecture is called R2R because all arithmetic instructions are not allowed to access memory.

Table 5: Four Instructions 14-17 in 2A R2R Instruction Set

	Instruction	**Meaning**
14	JNS Label	\$ra $\leftarrow$ *PC* & *PC* $\leftarrow$ Label
15	JR	*PC* $\leftarrow$ \$ra
16	READ	\$v0 $\leftarrow$ Input
17	PRNT	Print \$a0

The four pseudo-instructions in Table 2 are implemented in their register version in 2A R2R machine. MOVE and NEG use four instructions each and PUSH and POP use two instructions each. One additional pseudo-instruction LA R Var which loads the address of the variable Var into the register R is implemented by using one instruction only.

D. Three-Address Register-to-Register Instruction Set

The instruction set of the simulated 3A R2R machine has 19 instructions in which instructions 14-17 are the same with those in Table 5 and other instructions except instructions 7-8 can be simply obtained by replacing memory locations in Instruction column and memory contents at memory locations in Meaning column of Table 3 (Table 1) with registers. The two exceptional instructions GET and PUT are listed in Table 6. The offset in these two instructions is a 16-bit integer which can be a constant or a variable name. It is easy to see that the instruction set of 3A R2R is a superset of that of 2A R2R. 3A R2R also implements all pseudo-instructions of 2A R2R.

Table 6: Two Instructions 7-8 in 3A R2R Instruction Set

	Instruction	**Meaning**
7	GET R R1 Offset	R $\leftarrow$ M[R1+Offset]
8	PUT R R1 Offset	M[R1+Offset] $\leftarrow$ R

III. Assemly Language Program Examples

Any assembly language program of all simulated machines consists of three parts: data section (optional), code section, and input section (optional) separated by a key word END.

The data section is used for declaring variables in memory. Each declaration takes one line and consists of ID, Type, and Value. ID is a variable name, Type indicates number of words the variable value has, and Value is optional initial value(s) of the variable. The code section consists of assembly language instructions. Each instruction takes one line and precedes an optional label immediately followed by ':' symbol. The input section is used for providing user input data. One line contains only one word (integer). In addition, users can add comments starting from // symbol and until to the end of line. A comment cannot cross multiple lines.

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 for the simulated machines. The first example is to compute sum of absolute values of all elements in an array. The second example is to compute Fibonacci number.

A. Sum of Absolute Values of Elements in Array

In this subsection, an array of integers and the length of the array are given in the data section, the sum of absolute values of elements in the array is computed, saved in the data section, and displayed on screen. Figures 1-4 show the assembly language programs for the four different computer architectures 2A M2M, 2A R2R, 3A M2M, and 3A R2R, respectively.

```
//Decorations
PDAT    1        DAT        //Pointer to the array DAT
SUM     1        0          //Sum
NUM     1        9          //Number of elements in the array
TMP     1        0          //Temporary location
DAT     9        10 20 30 -40 50 60 70 80 -90    //Array data
END
//Instructions
L1:     BEQZ     NUM  L3          //If NUM=0, done
        GET      TMP  PDAT        //Get an element from array
        BGEZ     TMP  L2          //If positive, skip
        NEG      TMP              //TMP = -TMP
L2:     ADD      SUM  TMP         //Add to sum
        ADDI     NUM  -1          //Decrease NUM by one
        ADDI     PDAT  1          //Point to next array element
        GOTO     L1               //Next iteration
L3:     MOVE     OUTPUT  SUM      //Print sum on screen
        PRNT
        STOP                      //Terminate program
END
```

Figure 1: 2A M2M Code Using Array

These programs demonstrate how to declare variables in the data section and how to write assembly instructions to access array elements one by one and to control the loop iterations in the code section.

For the array access, a pointer to the array is used for 2A machines, while an array index and the array base address are used for 3A machines. In each loop iteration, the point will be increased to point to the next array element in 2A machine, while the array index will be increased for the next array element in 3A machines. The variable PDAT in Figure 1 and the register \$s0 in Figure 2 are such pointers. The variable IND in Figure 3 and the register \$t0 in Figure 4 are the array index. Note that the array variable DAT reparents the base address of this array.

```
//Declaration
SUM     1         0          //Sum
NUM     1         9          //Number of elements in the array
DAT     9         10 20 30 -40 50 60 70 80 -90     //Array data
END
//Instructions
        LA        $s0 NUM   //$s0 = address of NUM
        GET       $t0 $s0   //$t0 = NUM
        LA        $s0 DAT   //$s0 = address of DAT
        LI        $a0 0     //$a0(sum) = 0
L1:     BEQZ      $t0 L3    //If no remaining element, done
        GET       $t1 $s0   //Get an array element into $t1
        BGEZ      $t1 L2    //If positive, skip
        NEG       $t1       //Else, negate $t1
L2:     ADD       $a0 $t1   //Add to $a0(sum)
        ADDI      $t0 -1    //Decrease num of remaining elements
        ADDI      $s0 1     //Next element
        GOTO      L1        //Next iteration
L3:     LA        $s0 SUM   //$s0 = address of SUM
        PUT       $a0 $s0   //Save the sum to SUM
        PRNT                //Print sum on screen
        STOP                //Terminate program
END
```

Figure 2: 2A R2R Code Using Array

```
//Declaration
IND     1         0          //Array index
//SUM, NUM, TMP, and DAT are the same as those in Figure 1
END
//Instructions
L1:     BEQ       IND NUM L3         //If IND=NUM, done
        GET       TMP DAT IND        //Get an array element
        BGE       TMP ZERO L2        //If positive, skip
        NEG       TMP                //Else, negate
L2:     ADD       SUM SUM TMP        //Add to sum
        ADDI      IND IND 1          //Increase index IND by one
        GOTO      L1                 //Next iteration
L3:     MOVE      OUTPUT SUM         //Move sum into OUTPUT
        PRNT                         //Print sum on screen
        STOP                         //Terminate program
END
```

Figure 3: 3A M2M Code Using Array

```
//Declaration
//SUM, NUM, and DAT are the same as those in Figure 2
END
//Instructions
        LI        $t0 0              //$t0(index) = 0
        GET       $t1 $zero NUM      //$t1 = NUM
        LI        $a0 0              //$a0(sum)=0
L1:     BEQ       $t0 $t1 L3         //If index=NUM, done
        GET       $t2 $t0 DAT        //Get an array element
        BGE       $t2 $zero L2       //If positive, skip
        NEG       $t2                //Else, negate
L2:     ADD       $a0 $a0 $t2        //Add $t2 to $a0(sum)
        ADDI      $t0 $t0 1          //Increase index
        GOTO      L1                 //Next iteration
L3:     PUT       $a0 $zero SUM      //Save the sum to SUM
        PRNT                         //Print sum on screen
        STOP                         //Terminate program
END
```

Figure 4: 3A R2R Code Using Array

For the loop control, the length of the array is used for 2A machines. In each iteration, the length is deceased by one. When the length becomes 0, the loop ends. The variable NUM in Figure 1 and the register $t0 Figure 2 are the array length. On the other hand, the array index that is used for accessing array is also used for the loop control in 3A machines. In each iteration, the index is increased by one. When the index becomes equal to the array length, the loop ends.

Please note that the array access and loop control methods used for 2A machines can be used for 3A machines as well, but not vice versa. 2A machines can neither access the memory using a base address plus offset nor branch based on comparing two non-zero quantities. Please also note that 3A machine could use the array length as the array index for both the loop control like 2A machines and the array access like 3A machines. In this case, the index variable can be omitted and the array element is processed from the end to the beginning.

B. *Compute Binonacci Numbers*

Three methods will be used to compute Fibonacci numbers, which is defined as bellow. The first is using a main program with a loop, the second using a non-recursive function, and the third using a recursive function.

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

Figure 5 shows C++ code that computes 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.

```
int main() {                        //compute Fib(N)
      int I, A, B, C, N
      std::cin >> N;                //get input N, say 10.
      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;
      return 0;
}
```

Figure 5: C++ Code: Compute Fibonacci Number Using Loop

In the following subsections, assembly language programs for 2A machines will be illustrated only because programs for 3A machines are very similar with their 2A counterparts except they use less instructions.

1) *Using a Loop*

Figure 6 shows the 2A M2M code computing the Fibonacci number using a loop. In the data section, the same variables as those in C++ code are declared. The program reads the input and stores it in variable N. Then, it computes N-2 and checks if N ≥ 2. If no, it stores the result in C, which is N itself, and goes to print result. Otherwise, it computes the loop body (C = B+A;

A = B; B = C;), decreases N, and checks if N ≥ 0. If yes, it goes to next loop iteration. If no, it exists the loop and prints the result.

```
//Declarations
N       1  0                              //N
C       1  0                              //Fib(N)
B       1  1                              //Fib(N-1)
A       1  0                              //Fib(N-2)
END
//Instruction
        READ                              //INPUT = Input N
        MOVE    N  INPUT                  //N = Input
        MOVE    C  N                      //C = N
        ADDI    N  -2                     //N = N-2
        BLTZ    N  L2                     //If N<2, done
L1:     MOVE    C  B                      //C = B
        ADD     C  A                      //C = C + A
        MOVE    A  B                      //A = B
        MOVE    B  C                      //B = C
        ADDI    N  -1                     //N--
        BGEZ    N  L1                     //If N >=0
L2:     MOVE    OUTPUT  C                 //OUTPUT = C
        PRNT                              //Print C
        STOP                              //Terminate
END
10
```

Figure 6: 2A M2M Code Using Loop

Figure 7 shows the 2A R2R code computing the Fibonacci number using a loop. This code utilizes registers for variables instead of memory locations. In fact, $v0 is N, $t1 is A, $t2 is B, and $a0 is C. Note that the READ instruction will save user input in $v0 and the PRNT instruction will display $a0.

```
END
//Instruction
        READ                              //$v0 = N
        MOVE    $a0  $v0                  //$a0(C) = N
        ADDI    $v0  -2                   //$v0 = $v0-2
        BLTZ    $v0  L2                   //If N<2, done
        LI      $t1  0                    //$t1(A) = 0
        LI      $t2  1                    //$t2(B) = 1
L1:     MOVE    $a0  $t2                  //$a0(C) = B,
        ADD     $a0  $t1                  //C = B+A
        MOVE    $t1  $t2                  //A = B
        MOVE    $t2  $a0                  //B = C
        ADDI    $v0  -1                   //N--
        BGEZ    $v0  L1                   //if N >= 0, continue
L2:     PRNT                              //Print $a0 (C)
        STOP                              //Terminate
END
10
```

Figure 7: 2A R2R Code Using Loop

2) *Using a Non-Recursive Function*

Figure 8 shows the 2A M2M code computing the Fibonacci number using a non-recursive function. It takes an integer N as input and compute Fib(N) as output. Note that the input N should be pushed on stack just before calling the function and the result Fib(N) should be stored on stack so that right after the function returns, only the function result remains on the stack.

```
END
//Instruction
        READ                              //INPUT = input N
        PUSH    INPUT                     //Push N on stack
        JNS     Fib                       //call Fib
        POP     OUTPUT                    //Pop result into OUTPUT
        PRNT                              //Print result
        STOP                              //Terminate
Fib:    MOVE    $+3  $-1                  //S = N
        ADDI    $+3  -2                   //S = N-2
        BLTZ    $+3  L2                   //If S < 0, done
        LI      $+1  0                    //A = 0
        LI      $+2  1                    //B = 1
L1:     MOVE    $-1  $+2                  //C = B
        ADD     $-1  $+1                  //C = C + A
        MOVE    $+1  $+2                  //A = B
        MOVE    $+2  $-1                  //B = C
        ADDI    $+3  -1                   //S = S-1
        BGEZ    $+3  L1                   //If S >= 0, continue
L2:     JR
END
10
```

Figure 8: 2A M2M Code Using Non-Recursive Function

Table 7 shows the stack frame of function Fib in Figure 8. Right after the function call instruction JNS and right before the function return instruction JR are executed, the stack pointer is (should be) pointing to the return address on stack. One slot below the return address on stack is used for input N and output Fib(N) or C. The next three slots on the stack are for local variables A, B, and S. Remember, the stack is growing toward higher memory address. Here, local variable S at $+3 is used to hold N-2 initially and then used for controlling the loop like variable N as shown Figure 6.

Table 7: Stack Frame of Function Fib in 2A M2M Code

Address	Content	Explanation
$-1	N/Fib(N)/C	Iuput/Output
$	RA	Return Address
$+1	A	Local Variable
$+2	B	Local Variable
$+3	S	Local Variable

Figure 9 shows the 2A R2R code computing the Fibonacci number using a non-recursive function. It takes an integer N saved in $a0 as input and compute Fib(N) and save it in $v0 as output. Note that there is no stack frame necessary for the function Fib in Figure 9 because the function uses registers for passing the input and output as well as local variables. For examples, $a0 is N, $t1 is A, $t2 is B, and $v0 is C. Note that we would use $v0 for N and $a0 for C like in Figure 7. In this case, we can also eliminate two MOVE instructions before and after the function call JNS in Figure 9. However, according to the register convention, $a0-$a4 are used for function inputs and $v0-$v1 are used for function outputs. It is important to follow the convention so that other programmers are able to understand your program.

```
END
//Instruction
            READ                        //$v0 = N
            MOVE $a0 $v0                //$a0 = $v0
            JNS Fib                     //$v0 = Fib(N)
            MOVE $a0 $v0                //$a0=$v0
            PRNT                        //Print $a0
            STOP                        //Terminate
Fib:        MOVE    $v0  $a0            //$v0(C) = $a0(N)
            ADDI    $a0  -2             //N = N-2
            BLTZ    $a0  L2             //If N<2, done
            LI      $t1  0              //$t1(A) = 0
            LI      $t2  1              //$t2(B) = 1
L1:         MOVE    $v0  $t2            //C = B
            ADD     $v0  $t1            //C = C+A
            MOVE    $t1  $t2            //A = B
            MOVE    $t2  $v0            //B = C
            ADDI    $a0  -1             //N = N-1
            BGEZ    $a0  L1             //If N ≥ 0, continue
L2:         JR
END
10
```

Figure 9: 2A R2R Code Using Non-Recursive Function

3) Using a Recursive Function

Figure 10Figure 11 shows 2A M2M code computing the Fibonacci number using a recursive function. The main routine in Figure 10 is the same as that in Figure 8. Only the recursive function Fib(N) is shown in Figure 10.

```
END
//Instruction
//The main routine is the same as that in Figure 8
Fib:    MOVE    $+1     $-1     //$+1 = N
        ADDI    $+1     -2      //$+1 = N-2
        BLTZ    $+1     L1      //If N < 2, done
        ADDI    SP      1       //PUSH N-2
        JNS     Fib             //Call F(N-2)
        MOVE    $+1     $-2     //$+1 = N
        ADDI    $+1     -1      //S+1 = N-1
        ADDI    SP      1       //PUSH N-1
        JNS     Fib             //Call F(N-1)
        ADDI    SP      -2      //restore SP (or $)
        MOVE    $-1     $+1     //F(N) = F(N-2) + F(N-1)
        ADD     $-1     $+2
END
10
```

Figure 10: 2A M2M Code Using Recursive Function

Table 8: Stack Frame of Recursive Function Fib in 2A M2M Code

Address			**Content**	**Explanation**
Fib(N)	Fib(N-2)	Fib(N-1)		
$-1	$-2	$-3	N/Fib(N)	Input/output
$	$-1	$-2	RA	Return Address
$+1	$	$-1	(N-2)/Fib(N-2)	Input/output
$+2	$+1	$	(N-1)/Fib(N-1)	Input/output

Table 8 shows the stack frame of recursive function Fib in Figure 10. When Fib(N) is called, the input N is at $-1, the return address of Fib(N) is at $. Inside Fib(N), Fib(N-2) and Fib(N-1) are called recursively when N ≥ 2. Before Fib(N-2) is called, N-2 must be pushed on stack. So, the value $ of the stack point SP is increased by one. Therefore, right after calling Fib(N-2), the input N is at $-2 and the return address of Fib(N) is at $-1. It is Similar when Fib(N-1) is called inside Fib(N). Please note that the value $ of the stack pointer must be restored before the function returns. Therefore, SP is decreased by two and the result Fib(N) is stored at $-1 before the Fib(N) returns.

Figure 11 shows 2A R2R code computing the Fibonacci number using a recursive function. The main routine in Figure 11 is the same as that in Figure 10. Only the recursive function Fib(N) is shown in Figure 11. It takes an integer N saved in $a0 as input and compute Fib(N) and save it in $v0 as output. Note that there must be a stack frame for any recursive function to save the return address at least. Aside from $a0 and $v0, the register $s0 is used to hold the result of Fib(N-2). So, the stack frame for this function contains three slots for saving $a0, $s0, and $ra, respectively. At the beginning of the recursive function shown in Figure 11, registers $a0, $s0, and $ra are pushed on stack. Right before the function returns, the three registers are restored and the stack remains the same as one before calling.

```
END
//Instruction
//The main routine is the same as that in Figure 9
//compute Fib(N)
Fib:    PUSH    $a0
        PUSH    $s0
        PUSH    $ra
        MOVE    $v0     $a0     /$v0 = N
        ADDI    $a0     -2      //$a0 = N-2
        BLTZ    $a0     L2      //If N<2
        JNS     Fib             //Compute $v0 = Fib(N-2)
        MOVE    $s0     $v0     //$s0 = Fib(N-2)
        ADDI    $a0     1       //$a0 = N-1
        JNS     Fib             //Compute $v0 = Fib(N-1)
        ADD     $v0     $s0     //$v0 = Fib(N-1) + Fib(N-2)
L2:     POP             $ra
        POP             $s0
        POP             $a0
        JR
END
10
```

Figure 11: 2A R2R Code Using Recursive Function

IV. Comparison of Sizes and Performances

In general, M2M programs use less instructions than their counterpart R2R programs. However, M2M programs have a larger size and much more memory accesses than R2R. So R2R architectures have much better performance than M2M.

In the following tables, the acronym NI is for number of instructions of programs, LI for length of instruction in bits, PS for program size in bits, NIMA for number of instructions fetched from memory during execution, NDMA for number of data memory accesses during execution, and TNMA for total number of memory accesses during execution. Note that one pseudo-instruction may include more than one instruction. Each instruction set is encoded in a fixed instruction length. The actual bits required for encoding instructions are used to compute the size of a program for a particular instruction set.

Table 9, Table 10, Table 11, and Table 12 show comparison results of the four programs using array, four programs using loop, four programs using non-recursive function, and four programs using recursive functions for 2A M2M, 2A R2R, 3A M2M, and 3A R2R architectures, respectively. For computing the sum of absolute values of elements in an array, the size of the array is 9. For computing the Fibonacci number, the user input N is 10.

Table 9: Comparison Results of the Four Programs Using Array

	NI	LI	PS	NIMA	NDMA	TNMA
2A M2M	17	39	663	106	190	296
2A R2R	19	26	494	108	11	119
3A M2M	10	56	560	67	141	208
3A R2R	13	31	403	70	11	81

Table 10: Comparison Results of the Four Programs Using Loop

	NI	LI	PS	NIMA	NDMA	TNMA
2A M2M	32	39	1,248	152	299	451
2A R2R	26	26	676	146	0	146
3A M2M	13	56	728	53	132	185
3A R2R	13	31	403	53	0	53

Table 11: Comparison Results for Using Non-Recursive Function

	NI	LI	PS	NIMA	NDMA	TNMA
2A M2M	32	39	1,248	152	303	455
2A R2R	36	26	936	156	0	156
3A M2M	16	56	896	56	143	199
3A R2R	17	31	527	57	0	57

Table 12: Comparison Results for Using Recursive Function

	NI	LI	PS	NIMA	NDMA	TNMA
2A M2M	30	39	1,170	2,567	5,751	8,318
2A R2R	43	26	1,118	2,400	525	2,925
3A M2M	18	56	1,008	1,155	3,282	4,437
3A R2R	22	31	686	1,419	525	1,944

According to the data in these tables, it is clear that R2R architecture is better than M2M architecture because R2R programs have smaller sizes and much less memory accesses. For the same reason, 3A machine is better than 2A machine. 3A R2R is the best among the four architectures. That is why almost all modern processor architectures are 3A R2R. From Table 10, Table 11, and Table 12, it can be seen that using a non-recursive function causes a small overhead, but using a recursive function causes a very large overhead.

V. CONCLUSIONS

In this paper, the four computer architecture simulators are presented as a continuation of paper [1] which presented stack-based and accumulator-based computer architecture simulators. Several example assembly language programs are also given. These examples illustrate many basic programming concepts and techniques at the assembly language level. These include dealing with array, loop, stack, function call, function return, parameter passing, local variables, stack frame, and recursion. The sizes (numbers of instructions) and costs (numbers of instructions executed and numbers of data memory accesses performed) of these example programs are compared among different computer instruction formats. It is clear that R2R (3A) machine programs have smaller sizes and costs than M2M (2A) machine programs. The elegance of recursive functions come with a great cost of larger memory and longer execution time compared with non-recursive functions. In addition, recursive programming at assembly language level is very difficult.

Currently, the pseudo-instructions MOVE and NEG use four instructions each in 2A machines, while they use one instruction each in 3A machines. So, adding MOVE and NEG in 2A instruction sets will reduce the size of 2A programs.

A webpage [8] has been created for students to study and use these simulators for their assembly language programming assignments. Students can also modify these simulators to add more instructions and/or pseudo-instructions. The webpage contains a simple introduction to these simulators, the steps to use these simulators, the assembly language program structure, syntax, and examples, the simulators source and Jar files, and assignments.

REFERENCES

[1] 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

[2] 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.

[3] David A. Patterson and John L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 5th Edition, Morgan Kaufmann Publishers, 2014.

[4] Kip Irvine, Assembly language for x86 processors – access card, 8th Edition, Pearson, 2020

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

[6] Luke Yen, Min Xu, Milo Martin, Doug Burger, and Mark Hill, "WWW Computer Architecture Page," available from: http://pages.cs.wisc.edu/~arch/www/

[7] Xuejun Liang, Loretta A. Moore, and Jacqueline Jackson, Programming at different levels: a teaching module for undergraduate computer architecture course, in Proceedings of the 2014 International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'14), pp.77-83, Las Vegas, Nevada, USA, July 21-24, 2014.

[8] Xuejun Liang, Computer Architecture Simulators Webpage, available at https://www.cs.csustan.edu/~xliang/Courses/SimulatorWeb/index.htm