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 different instruction formats, including stack-based, accumulator-based, two-address, and three-address machines. These simulators can be used to assemble and run assembly language programs on the above computer architectures. Several simple applications are used to illustrate how to develop assembly language programs to deal with arrays, subroutines, and recursions on these different computer architectures. 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, debugging functions, and etc.

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 [1]. 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 [2]. Kip Irvine teaches x86 in his textbook [3]. Linda Null and Julia Lobur uses the accumulator-based architecture and the MARIE simulator [4]. On the other hand, although there are numerous processor simulators available [5], 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 different instruction formats, including stack-based, accumulator-based, two-address, and three-address machines. Both memory-to-memory and register-to-register architectures are considered for the two-address and three-address machines. These simulators can be used to assemble and run assembly language programs on the above simulated computer architectures. Several simple applications are used to illustrate how to develop assembly language programs to deal with arrays, subroutines, and recursions on these computer architectures. Using these simulators to perform their hands-on assembly language programming exercises, students will be able to have a better understanding of computer architectures. Students can also modify these simulators to add more instructions, debugging functions, and etc. In addition, these simulated machines can serve as the compiler's target machines for the code generation practice.

For the simplicity, the microarchitectures that will support the execution of instructions of these simulated machines are not considered. The instruction sets implemented in these simulators contain only basic integer arithmetic, branch, stack, load, store, subroutine call and return, input, and output. Due to the limit of space, only stack-based and accumulator-based machines will be reported in detail in this paper. In the rest of this paper, the simulated instruction sets are presented in Second II. Several assembly language programming examples using these simulators are described in Second III. Finally, Second IV will conclude the papers.

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 is 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, *SP* for stack pointer, *FP* for frame pointer, and *AC* for accumulator.

In all simulated machines, stack will grow towards higher memory address. *SP* and *FP* are registers in stack-based, and two-address register-to-register, and three-address register-to-register machines, while *SP* is a reserved memory location and *FP* is not available in accumulator-based, two-address memory-to-memory, and three-address memory-to-memory machines.

A. Stack-Based (Zero-Address) Instruction Set

Table 1 lists all instructions of the simulated stack-based (or zero-address) machine. This instruction set includes 5 integer arithmetic instructions, 5 branch instructions, 1 subroutine call and 1 return instructions, 10 stack operations, 2 instructions to manipulate with *SP* and *FP*, 1 input and 1 output instructions, and finally, 1 stop instruction to terminate the program.

The operational stack and activation record (stack frame) for subroutine calls share the same stack inside the data memory. The notation *FP*+*Imm* is used to indicate a local variable inside an activation record (stack frame). It is a memory address in the stack frame with offset *Imm*.

Table 1: Stack-Based Instruction Set

op	Instruction	Explanation
0	ADD	Pop the top two addends, add, and push the sum
1	SUB	Pop the subtrahend and minuend, subtract, and push the difference
2	MUL	Pop the multiplicand and multiplier, multiply, and push the product
3	DIV	Pop the dividend and divisor, divide, and push the quotient
4	REM	Pop the dividend and divisor, divide, and push the remainder
5	GOTO Label	Unconditionally jump to the instruction at address Label
6	BEQZ Label	Pop the top item and jump to Label if the popped item is zero
7	BNEZ Label	Pop the top item and jump to Label if the popped item is not zero
8	BGEZ Label	Pop the top item and jump to Label if the popped item is greater than or equal to 0
9	BLTZ Label	Pop the top item and jump to Label if the popped item is less than 0
10	JNS Label	Push the return address and transfer the control to the instruction at address Label
11	JR nLoc	Pop the return address into *PC* and decrement *SP* by nLoc
12	PUSH *FP*	Push the content of *FP* on stack
13	PUSH *FP*+*Imm*	Push M[*FP*+*Imm*] on stack
14	PUSH *Imm*	Push a 16-bit integer value *Imm* on stack
15	PUSH Var	Push M[Var] on stack
16	PUSHI Var	Push M[M[Var]] on stack
17	POP *FP*	Pop the top item into *FP* from stack
18	POP *FP*+*Imm*	Pop the top item into M[*FP*+*Imm*] from stack
19	POP Var	Pop the top item into M[Var] from stack
20	POPI Var	Pop the top item into M[M[Var]] from stack
21	SWAP	Swaps the top two items on the stack
22	MOVE	Copy content of *SP* into *FP*
23	ISP nLoc	Increase/decrease *SP* by nLoc
24	READ	Read an input and push it on stack
25	PRNT	Print the top item on stack
26	STOP	Terminate the program

B. Accumulator-Based (One-Address) Instruction Set

Table 2 lists all instructions of the simulated accumulator-based (or one-address) machine. This instruction set includes 6 integer arithmetic instructions, 1 load immediate instruction, 5 branch instructions, 1 subroutine call and 1 return instructions, 1 GET and 1 GETI instructions, 1 PUT and 1 PUTI instructions, 1 input and 1 output instructions, and finally, 1 stop instruction to terminate the program.

The symbol ← in Table 2 means assignment. Var in Table 2 indicates a memory location. It 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 $+4 means *AC* ← *AC* + *M[M[SP]*+4]. Note that *M[SP]* is the content of *SP* and is usually pointing to the top of stack.

Table 2: Accumulator-Based Instruction Set

Op	Instruction	Meaning
0	LIMM Imm	*AC* ← *Imm*
1	AIMM Imm	*AC* ← *AC*+*Imm*
2	ADD Var	*AC* ← *AC*+*M[*Var*]*
3	SUB Var	*AC* ← *AC*-*M[*Var*]*
4	MUL Var	*AC* ← *AC***M[*Var*]*
5	DIV Var	*AC* ← *AC*/*M[*Var*]*
6	REM Var	*AC* ← *AC*%*M[*Var*]*
7	GET Var	*AC* ← *M[*Var*]*
8	PUT Var	*M[A]* ← *AC*
9	GOTO Label	*PC* ← Label
10	BEQZ Label	If *AC* = 0 then *PC* ← Label
11	BNEZ Label	If *AC* ≠ 0 then *PC* ← Label
12	BGEZ Label	If *AC* ≥ 0 then *PC* ← Label
13	BLTZ Label	If *AC* < 0 then *PC* ← Label
14	JNS Label	Push the return address and *PC* ← Label
15	JR	Pop the return address into *PC*
16	READ	Read an input and save it to *AC*
17	PRNT	Print *AC*
18	STOP	Terminate the program
19	GETI Var	*AC* ← *M[M[*Var*]]*
20	PUTI Var	*M[M[*Var*]]* ← *AC*

The assembler of simulated one-address machine provides three pseudo-instructions. POP will remove the top item of stack by reducing the stack pointer *SP*'s value by 1. TOP A will only return the value of the top item of stack to A without changing stack. PUSH A will increase the stack pointer *SP*'s value by 1 first and then save the value of A on the top of stack.

III. Assemly Language Program Examples

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

The data part 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 values of the variable. The code part is for assembly language instructions. Each instruction takes one line and precedes an optional label immediately followed by ':' symbol. The input part is used for providing user input data. One input 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, 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, which is defined by

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

A. Sum of Absolute Values of Elements in Array

Figure 1 Shows stack-based assembly language program to compute the sum of absolute values of array elements. In the data section, an array variable DAT with 9 integers is declared and initialized. Five other variables are also declared. The loop starts by checking if N-I = 0. If yes, the program will exit the loop and print the result. Otherwise, the program adds one array element's absolute value to SUM and then increase the array index I and the array pointer PDAT for the next loop iteration.

```
//Data
I       1       0               //array index
SUM     1       0               //sum
N       1       9               //number of elements in the array
TMP     1       0               //temporary location
PDAT    1       DAT             //pointer to the array DAT
DAT     9       10 20 30 -40 50 60 70 80 -90    //array DAT
END
//Code
L1:     PUSH    N
        PUSH    I
        SUB                     //N-I
        BEQ     L3              //if (N-I)=0, done
        PUSHI   PDAT            //get an array element
        PUSHI   PDAT            //get the array element again
        BGEZ    L2              //if positive, skip
        PUSH    0               //else, negate
        SWAP
        SUB
L2:     PUSH    SUM             //add to sum
        ADD
        POP     SUM
        PUSH    I               //increase index I by one
        PUSH    1
        ADD
        POP     I
        PUSH    PDAT            //increase array address by one
        PUSH    1
        ADD
        POP     PDAT
        GOTO    L1              //next element
L3:     PUSH    SUM             //print sum
        PRNT
        STOP                    //stop
END
```

Figure 1: Stack-Based (Zero-Address) Code Using Array

```
//Data
//Same as that in Figure 1.
END
//Code
L1:     GET     N
        SUB     I       //N-I
        BEQZ    L3      //if (N-I)=0, done
        GETI    PDAT    //get an array element into AC
        BGEZ    L2      //if positive, skip
        PUT     TMP     //else, negate
        LIMM    0
        SUB     TMP
L2:     ADD     SUM     //add to sum
        PUT     SUM
        GET     I       //increase index I by one
        AIMM    1
        PUT     I
        GET     PDAT    //increase array address by one
        AIMM    1
        PUT     PDAT
        GOTO    L1      //next element
L3:     GET     SUM     //print sum
        PRNT
        STOP            //terminate program
END
```

Figure 2: Accumulator-Based Code Using Array

Figure 2 shows accumulator-based assembly language code to compute the sum of absolute values of array elements. This program has the same data section as that in the stack-based program. It also applys the same algorithm to compute the sum. The difference is that the stack-based program needs to push the the two input data on stack for an operation and to get the result from stack, while the accumulator-based program needs to load one of the two input data for an operation into accumulator *AC* and to get the result from *AC*.

B. Binonacci Numbers

Three methods will be used to compute Fibonacci numbers. The first is using a loop, the second using a function, and the third using a recursive function.

1) Using a Loop,

Figure 3 shows C++ code that computes Fibonacci number using a loop. The algorithm 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 3: C++ Code: Compute Fibonacci Number

Figure 4 shows the stack-based code to compute 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 $N \geq 2$. 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;), increases loop index I, and checks if $I \leq N$. If yes, it goes to next loop iteration. If no, it exists the loop and prints the result.

```
//Data
I       1       1       //index
N       1       0       //N
C       1       0       //Fib(N)
B       1       1       //Fib(N-1)
A       1       0       //Fib(N-2)
END
//Code
        READ            //read input
        POP     N       //N=input
        PUSH    N
        PUSH    2
        SUB             //N-2
        BGEZ    L1      //if N>=2 go to loop body
        PUSH    N
        POP     C       //C=N
        GOTO    L2      //go to print result
L1:     PUSH    B       //beginning of loop body
        PUSH    A
        ADD
        POP     C       //C = B+A
        PUSH    B
        POP     A       //A = B
        PUSH    C
        POP     B       //B = C
        PUSH    I
        PUSH    1
        ADD
        POP     I       //I = I+1
        PUSH    I
        PUSH    N
        SUB             //I-N
        BLTZ    L1      //if I<N go to loop body
L2:     PUSH    C       //print result
        PRNT
        STOP            //terminate program
END
//Input
10              //N
```

Figure 4: Stack-Based Code Using a Loop

Figure 5 Shows accumulator-based assembly language code to compute Fibonacci number. This program has the same data section as that in the stack-based program. It applies the same loop algorithm. Here, N-1 is computed to check if $N < 2$, instead of computing N-2. Another difference is that the stack-based program needs to push the data on stack for computation, while the accumulator-based program needs to load one of the input data into accumulator *AC* for computation.

2) Using a Non-Recursive Function

Now, consider writing a non-recursive function Fib to compute Fibonacci number. 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

```
//Data
//Same as that in Figure 4
//Code
        READ            //read input
        PUT     N       //N=input
        PUT     C       //C=N
        SUB     I       //AC=N-1
        BLTZ    L2      //if N<1 done
        BEQZ    L2      //if N=1 done
L1:     GET     B       //beginning of loop body
        ADD     A       //AC=B+A
        PUT     C       //C=B+A
        GET     B       //AC=B
        PUT     A       //A=B
        GET     C       //AC=C
        PUT     B       //B=C
        GET     I       //AC=I
        AIMM    1       //AC=I+1
        PUT     I       //I=I+1
        SUB     N       //AC=I-N
        BLTZ    L1      //if I<N go to loop body.
L2:     GET     C       //print result
        PRNT
        STOP            //terminate program
        END
//inputs
10              //N
```

Figure 5: Accumulator-Based Code Using a Loop

Fib(N) should be stored on stack so that right after the function returns, only the function result remains on the stack. Table 3 shows the stack frame of the function Fib. It contains input N/output Fib(N), return address, and four local variables.

Table 3: Stack Frame of Function Fib

Address	**Content**	**Explanation**
FP-1	N/Fib(N)	Before call Fib(N)/after Fib(N) return
FP	RA	Return Address
FP+1	I	Local variable
FP+2	A	Local variable
FP+3	B	Local variable
FP+4	C	Local variable

Figure 6 shows stack-based code to compute Fibonacci number using a non-recursive function. The read instruction reads input and push it on top of stack. So, this program reads input and then calls the function Fib immediately. After the function returns, the print instruction prints the result on top of stack.

Note that JNS instruction pushes return address on stack. The first instruction of the subroutine Fib copies *SP* into *FP*. So, *FP* points to the function return address and the input N is pointed by *FP*-1. The next four sluts on stack are used for the four local variables. Note that I, A, B, and C in the content column in Table 3 are used to compare with variable names of program in Figure 4. No local variable names can be defined, instead, *FP+Imm* is used to represent local variable location.

Note that Fib(N)=N when N<2. So, there is no need to store result on stack when $N < 2$ as it is already there. Meanwhile,

right before Fib returns, the result should be stored on stack at *FP*-1 and *SP* should point to the return address. Instructions POP FP-1 and ISP -3 in Figure 6 are to achieve these.

```
//No data
END
//Code
        READ
        JNS     Fib     //call Fib
        PRNT            //print result
        STOP            //terminate program
Fib:    MOVE    FP SP
        PUSH    1       //I=1
        PUSH    0       //A=0
        PUSH    1       //B=1
        PUSH    0       //C=0
        PUSH    FP-1    //N
        PUSH    2       //2
        SUB             //N-2
        BGEZ    L1      // if N>=2 go to loop body
        GOTO    L2      //Fib(N)=N if N<2
L1:     PUSH    FP+3    //beginning of loop body
        PUSH    FP+2
        ADD
        POP     FP+4    //C=B+A
        PUSH    FP+3
        POP     FP+2    //A=B
        PUSH    FP+4
        POP     FP+3    //B=C
        PUSH    FP+1
        PUSH    1
        ADD
        POP     FP+1    //I=I+1
        PUSH    FP+1
        PUSH    FP-1
        SUB             //I-N
        BLTZ    L1      //if I<N go to loop body
        POP     FP-1    //Store result at FP-1
L2:     ISP     -3      //restore the stack
        JR      0       //return
END
//Inputs
10                      //N=10
```

Figure 6: Stack-Based Code Using Non-Recursive Function

```
//Data
N       1       0       //N: input
C       1       0       //C: result of fib(N)
END
//Code
//main program
        READ            //read input
        PUT     N       //N=input
        PUSH    N       //push N on stack
        JNS     Fib     //call Fib
        TOP     C       //get and save fib(N)
        POP             //restore stack
        GET     C       //get Fib(N) from C
        PRNT            //print Fib(N)
        STOP            //terminate program
```

Figure 7: Accumulator-Stack-Based Code Using Non-Recursive Function (1)

```
Fib:    LIMM    1
        PUT     $+1     //I=1
        PUT     $+3     //B=1
        AIMM    -1
        PUT     $+2     //A=0
        GET     $-1     //AC=N
        SUB     $+1     //AC=N-1
        BLTZ    L3      //if N<1 done
        BEQZ    L3      //if N=1 done
L2:     GET     $+3     //beginning of loop body
        ADD     $+2     //AC=B+A
        PUT     $+4     //C=B+A
        GET     $+3     //AC=B
        PUT     $+2     //A=B
        GET     $+4     //AC=C
        PUT     $+3     //B=C
        GET     $+1     //AC=I
        AIMM    1       //AC=I+1
        PUT     $+1     //I=I+1
        SUB     $-1     //AC=I-N
        BLTZ    L2      //if I<N go to loop body
        GET     $+4     //AC=C
        PUT     $-1     //store result on stack
L3:     JR
END
10
```

Figure 8: Accumulator-Stack-Based Code Using Non-Recursive Function (2)

Figure 7 shows the main program of accumulator-based code to compute Fibonacci number using a non-recursive function. Figure 8 shows the function Fib. This program has the same stack frame of function Fib as shown in Table 3 except the frame pointer *FP* is replaced by the $ symbol. The main program reads input, pushes it on stack, and then calls the subroutine Fib. When the subroutine returns, it pops the result off stack and prints the result. The subroutine Fib code is very much like the code shown in Figure 5.

Note that the subroutine Fib shown in Figure 8 does not use any push instructions. Therefore, the stack point *SP* does not change inside the subroutine and then there is no to use pop instructions to restore the stack.

3) Using a Recursive Function

Table 4 shows the stack frame of recursive function Fib. Before the function call, the input N should be pushed on stack. Then function call pushes the return address on stack. As shown in Figure 9, the subroutine will push the previous frame point *FP* on stack to save it as successive calls will change *FP*. The next two sluts in the stack frame at *FP*+1 and *FP*+2 are for successive calls. Before the function returns, the result is stored on stack frame at *FP*-2 and previous *FP* is restored.

Table 4: Stack Frame of Recursive Function Fib (Stack)

Address	**Content**	**Explanation**
FP-2	N/Fib(N)	Before call Fib(N)/after Fib(N) return
FP-1	RA	Return Address
FP	Prior FP	Previous FP
FP+1	(N-1)/Fib(N-1)	Before call Fib(N-1)/after Fib(N-1) return
FP+2	(N-2)/Fib(N-2)	Before call Fib(N-2)/after Fib(N-2) return

```
//compute Fib(N)
Fib:    PUSH    FP
        MOVE    FP SP     //FP=SP
        PUSH    FP-2      //N
        PUSH    2         //2
        SUB               //N-2
        BGEZ    L1        //N>=2
        GOTO    L2        //Fib(N)=N if N<2
L1:     PUSH    FP-2      //N
        PUSH    1         //1
        SUB               //N-1
        JNS     Fib       //call Fib(N-1)
        PUSH    FP-2      //N
        PUSH    2         //2
        SUB               //N-2
        JNS     Fib       //call Fib(N-2)
        ADD               //Fib(N-1)+Fib(N-2)
        POP     FP-2      //store result at FP-2
L2:     POP     FP        //restore previous FP
        JR      0
        END
//Inputs
10                        //N=10
```

Figure 9: Stack-Based Recursive Function Fib

Table 5 shows the stack frame of recursive function Fib for accumulator-based machine. Before the function call, the input N should be pushed on stack. Then the function call from main program pushes the return address on stack. The next two sluts in the stack frame are for successive calls Fib(N-1) and Fib(N-2), respectively. The first column of address in Table 5 shows the use of stack frame for function Fib(N).

Note that in order to call Fib(N-1) within Fib(N), the stack pointer should point to N-1. This requires that *SP* increase by 1 before calling Fib(N-1). So, the value of $ is also increased by 1 as shown in the second column of address. Similarly, right before calling Fib(N-2), *SP* and therefore $ are increased by 1 again as shown in the third column of address. Finally, the last column in Table 5 is similar to the last column in Table 4.

Table 5: Stack Frame of Recursive Function Fib (Accumulator)

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

IV. CONCLUSIONS

In this paper, several computer architecture simulators are presented. 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 and return, parameter passing, local variables, and recursion. Because these simulated machines contain very simple similar instruction set and have the same assembly language program structure, it is convenient to compare assembly language programming details among different computer instruction formats. For example, to

```
//compute Fib(N)
Fib:    GET     $-1       //AC=N
        BNEZ    L1        //Fib(N)=0 if N=0
        GOTO    L3        //done
L1:     AIMM    -1        //AC=N-1
        BNEZ    L2        //Fib(N)=1 if N=1
        GOTO    L3        //done
L2:     PUT     $+1       //store N-1 to $+1
        GET     SP        //AC=SP
        AIMM    1         //AC=SP+1
        PUT     SP        //increase SP ($) by 1
        JNS     Fib       //call Fib(N-1)
        GET     $-2       //AC=N
        AIMM    -2        //AC=N-2
        PUT     $+1       //store N-2 to $+1
        GET     SP        //AC=SP
        AIMM    1         //AC=SP+1
        PUT     SP        //increase SP ($) by 1
        JNS     Fib       //Call Fib(N-2)
        GET     SP        //AC=SP
        AIMM    -2        //AC=SP-2
        PUT     SP        //restore SP
        GET     $+1       //AC=Fib(N-1)
        ADD     $+2       //AC=Fib(N-1)+Fib(N-2)
        PUT     $-1       //store Fib(N) to $-1
L3:     JR
END
10                        //Input 10
```

Figure 10: Accumulator-Based Recursive Function Fib

support the use of array, the processor without general purpose registers should provide indirect memory access instructions such as PUSHI and GETI. Owing to the space, two-address and three-address (memory-to-memory and register-to-register) machine simulators are not presented here. In register-to-register machine, stack frame may not be needed for a non-recursive subroutine. In a three-address machine, an assembly language program uses a smaller number of instructions than zero- or one-address machines.

Students can use these simulators for assembly language programming assignments. They can also modify these simulators to add more instructions and debugging tools. In addition, these simulated machines can serve as the compiler's target machines for the code generation practice.

REFERENCES

[1] 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.

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

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

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

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

[6] 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.