Computer Architecture Simulators for Different Instruction Formats

Xuejun Liang
Department of Computer Science
California State University – Stanislaus
Turlock, CA 95382, USA

Motivation

- MARIE is an accumulator-based machine simulator
  - Used in The Essentials of Computer Organization and Architecture, by Linda Null and Julia Lobur for assembly language programming
- But, MARIE simulator has some problems:
  - Programmers can not define a variable to hold the address of another variable symbolically.
  - Conditional branch can only skip the next instruction. It takes numbers as its operands for indicating different conditions. Programmers have to remember these numbers.
  - MARIE do not have the stack pointer. So there is no stack frame and its subroutine has no local variables and can not be recursive.
- How to solve these problems?
  - Fix it or create a new one.

Purpose

- To solve these problems, and furthermore, to be able to compare different computer architectures, a set of computer simulators for different instruction formats are developed.
- Using these simulators
  - Program different computer processors using assembly languages.
- Modifying these simulators
  - Add more instructions
  - Add more debugging functions
  - Add user interfaces
  - Design microarchitectures to support the execution of instructions of simulated machines
- Serving as compiler's target computers

Outlines

- Simulated Instruction Sets
  - Stack-based (Zero address) Machine
  - Accumulator-based (one address) Machine
  - Two address Machine
    - Memory-to-Memory and Register-to-Register
  - Three address Machine
    - Memory-to-Memory and Register-to-Register
- Assembly Language Programming Examples
  - Compute sum of absolute values of elements in an array
  - Compute Fibonacci numbers
    - Using loop, function, and recursive function

Zero-address machine

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 64K words of word-addressable data memory.
- 64K words of word-addressable instruction memory.
- 32-bit data words.
- 32-bit instructions.
- A 32-bit arithmetic logic unit (ALU).
  - Only arithmetic operations are implemented
- Registers
  - PC: 32-bit Program counter
  - SP: Stack pointer pointing to the top of the stack
  - FP: Stack frame pointer holding the base address of AR

Implementations

- Three separate 32-bit integer arrays are used for instructions, memory data, and input data, respectively.
- Each instruction takes 21 bits
  - 5-bit opcode, and 16-bit operand (address).
  - But, using one 32-bit word to hold one instruction.
- Instruction memory address is 16 bits. So, Instructions will take up to 64K 32-bit words (or integers).
- The branch instructions use 16-bit absolute address. The instructions JNS also uses 16-bit absolution address
- Each datum occupies 32 bits. The data address is 16 bits. So, we have 64K words of data memory.
- Stack is growing towards higher data memory address

Instruction Set (1)

Imm	16-bit 2's complement
PC	Program counter
Var	Variable
Lab	Label
M[A]	Memory content of variable A
SP	Reserved location, Stack pointer
FP	Reserved location, Frame pointer

op	Instruction	Explanation
0	ADD	Pop the top two locations, add, and push the result
1	SUB	Pop subtrahend and minuend, subtract, and push the result
2	MUL	Pop the multiplicand and multiplier, multiply, and push the result
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 Lab	Unconditionally jump to the instruction at address Lab
6	BEQZ Lab	Pop the top location and jump to Label if the popped location is zero
7	BNEZ Lab	Pop the top location and jump to Lab if the popped location is not zero
8	BGEZ Lab	Pop the top location and jump to Lab if the popped location is greater than or equal to 0
9	BLTZ Lab	Pop the top location and jump to Lab if the popped location is less than 0

Instruction Set (2)

op	Instruction	Explanation
10	JNS Lab	Push the return address and transfer the control to the instruction at address Lab
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]
14	PUSH imm	Push a 16-bit integer value
15	PUSH Var	PUSH M[Var]
16	PUSHI Var	Push M[M[Var]]
17	POP FP	FP ← POP()
18	POP FP+imm	M[FP+imm] ← POP()
19	POP Var	M[Var] ← POP()
20	POPI Var	M[M[Var]] ← POP()
21	SWAP	Swaps two top words on the stack
22	MOVE	FP ← SP
23	ISP nLoc	Increase/decrease SP by nLoc
24	READ	Read an input and push it on stack
25	PRNT	Pop the top location and print it
26	STOP	Terminate the program

One Address Machine

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 64K words of word-addressable data memory.
- 64K words of word-addressable instruction memory.
- 32-bit data words.
- 32-bit instructions.
- A 32-bit arithmetic logic unit (ALU).
  - Only arithmetic operations are implemented
- Registers
  - AC: 32-bit Accumulator
  - PC: 32-bit Program counter
  - SP: Stack pointer pointing to the top of the stack

Implementations

- Three separate 32-bit integer arrays are used for instructions, memory data, and input data, respectively.
- Each instruction takes 22 bits
  - 5-bit opcode, and 16-bit operand (address).
  - 1-bit indicating if the operand is local or global
  - But, using one 32-bit word to hold one instruction.
- Instruction memory address is 16 bits. So, Instructions will take up to 64K 32-bit words (or integers).
- The branch instructions use 16-bit absolute address. The instructions JNS also use 16-bit absolution address
- Each datum occupies 32 bits. The data address is 16 bits. So, we have 64K words of data memory.
- Stack is growing towards higher data memory address

Instruction Set

Imm:	16-bit 2's complement
AC:	Accumulator
PC:	Program counter
Var:	Variable
Lab:	Label
M[A]:	Memory content of variable A
PUSH PC:	Push PC on stack
POP:	Remove top content on stack
SP:	Reserved location, Stack pointer
ZERO:	Reserved location, M[ZERO]=0

$+Imm:	Local variable Its address is M[SP]+Imm, where Imm is a 16-bit integer.
Example:	ADD $+4 means AC ← AC + M[M[SP]+4]

op	Instruction	Explanation
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[Var] ← AC
9	GOTO Lab	PC ← Lab
10	BEQZ Lab	If AC = 0 GOTO Lab
11	BNEZ Lab	If AC ≠ 0 GOTO Lab
12	BGEZ Lab	If AC ≥ 0 GOTO Lab
13	BLTZ Lab	If AC < 0 GOTO Lab
14	JNS Lab	PC ←L & PUSH PC
15	JR	PC ← M[M[SP]] & POP
16	READ	AC ← Input
17	PRNT	Print AC
18	STOP	Stop
19	GETI Var	AC ←M[M[Var]]
20	PUTI Var	M[M[Var]] ← AC

Pseudo-Instructions

	Pseudo-instruction	Meaning	Instruction
1	POP	M[SP] = M[SP] - 1	GET SP AIMM -1 PUT SP
2	TOP A	M[A] ← M[M[SP]]	GETI SP PUT A
3	PUSH A	M[SP] = M[SP] + 1 M[M[SP]] ← M[A]	GET SP AIMM 1 PUT SP GET A PUTI SP

Two-address machine (M-to-M)

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 64K words of word-addressable data memory.
- 64K double words of word-addressable instruction memory.
- 32-bit data words.
- 64-bit instructions.
- A 32-bit arithmetic logic unit (ALU).
  - Only arithmetic operations are implemented
- Registers
  - PC: 32-bit Program counter
  - SP: Stack pointer pointing to the top of the stack

Implementations (1)

- Two separate 32-bit integer arrays are used for memory data, and input data, respectively.
- One 64-bit integer arrays are used for instructions.
- But, Each instruction takes only 39 bits
  - 5-bit opcode, and two 16-bit operand (address).
  - Two 1-bit indicating if the operand is local or global.
- Instruction memory address is 16 bits. So, Instructions will take up to 64K 64-bit words (or integers).
- The branch instructions use 16-bit absolute address. The instructions JNS also uses 16-bit absolution address
- Each datum occupies 32 bits. The data address is 16 bits. So, we have 64K words of data memory.
- Stack is growing towards higher data memory address

Instruction Set

Two M2M

Imm:	16-bit 2's compliment
PC:	Program counter
M[A]:	Memory content of variable A
PUSH PC:	Push PC on stack
POP:	Remove top content on stack
SP:	Reserved location, Stack pointer
ZERO:	Reserved location, M[ZERO]=0
INPUT:	Reserved location for input
OUTPUT:	Reserved location for output

$+Imm:	Local variable Its address is M[SP]+Imm, where Imm is a 16-bit integer.
Example:	ADD Var $+4 means M[Var] = M[Var] + M[M[SP]+4]

op	Instruction	Meaning
0	LIMM C Imm	M[C] ← Imm
1	AIMM C Imm	M[C] ← M[C]+Imm
2	ADD C A	M[C] ← M[C]+M[A]
3	SUB C A	M[C] ← M[C]-M[A]
4	MUL C A	M[C] ← M[C]*M[B]
5	DIV C A	M[C] ← M[C]/M[B]
6	REM C A	M[C] ← M[C]%M[B]
7	GET C A	M[C] ← M[M[A]]
8	PUT B A	M[M[B]] ← M[A]
9	GOTO L	PC ← L
10	BEQZ A L	If M[A] = 0 GOTO L
11	BNEZ A L	If M[A] ≠ 0 GOTO L
12	BGEZ A L	If M[A] ≥ 0 GOTO L
13	BLTZ A L	If M[A] < 0 GOTO L
14	JNS L	PC←L & PUSH PC
15	JR	PC ← M[M[SP]] & POP
16	READ	M[*INPUT*] ← input
17	PRNT	Print M[*OUTPUT*]
18	STOP	Stop

Pseudo-Instructions

	Pseudo-instruction	Meaning	Instruction
1	MOVE B A	M[B] ← M[A]	LIMM B 0 ADD B A
2	NEG A	M[A] ←-M[A]	SUB ZERO A LIMM A 0 ADD A ZERO SUB ZERO A
3	POP	M[SP] ← M[SP] – 1	AIMM SP -1
4	TOP A	M[A] ← M[M[SP]]	GET A SP
5	PUSH A	M[SP] ← M[SP] + 1 M[M[SP]] ← M[A]	AIMM SP 1 PUT SP A

Three-address machine (M-to-M)

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 64K words of word-addressable data memory.
- 64K double words of word-addressable instruction memory.
- 32-bit data words.
- 64-bit instructions.
- A 32-bit arithmetic logic unit (ALU).
  - Only arithmetic operations are implemented
- Registers
  - PC: 32-bit Program counter
  - SP: Stack pointer pointing to the top of the stack

Implementations (1)

- Two separate 32-bit integer arrays are used for memory data, and input data, respectively.
- One 64-bit integer arrays are used for instructions.
- But, Each instruction takes only 56 bits
  - 5-bit opcode, and three 16-bit operand (address).
  - Three 1-bit indicating if the operand is local or global.
- Instruction memory address is 16 bits. So, Instructions will take up to 64K 64-bit words (or integers).
- The branch instructions use 16-bit absolute address. The instructions JNS also uses 16-bit absolution address
- Each datum occupies 32 bits. The data address is 16 bits. So, we have 64K words of data memory.
- Stack is growing towards higher data memory address

Instruction Set

Three M2M

Imm:	32-bit 2's compliment
PC:	Program counter
M[A]:	Memory content of variable A
PUSH PC:	Push PC on stack
POP:	Remove top content on stack
SP:	Reserved location, Stack pointer
ZERO:	Reserved location, M[ZERO]=0
INPUT:	Reserved location for input
OUTPUT:	Reserved location for output

$+Imm:	Local variable Its address is M[SP]+Imm, where Imm is a 16-bit integer.
Example:	ADD A B $+4 means M[A] = M[B] + M[M[SP]+4]

op	Instruction	Meaning
0	LIMM C Imm	M[C] ← Imm
1	AIMM C A Imm	M[C] ← M[A]+Imm
2	ADD C A B	M[C] ← M[A]+M[B]
3	SUB C A B	M[C] ← M[A]-M[B]
4	MUL C A B	M[C] ← M[A]*M[B]
5	DIV C A B	M[C] ← M[A]/M[B]
6	REM C A B	M[C] ← M[A]%M[B]
7	GET C A I	M[C] ← M[A+M[I]]
8	PUT B I A	M[B+M[I]] ← M[A]
9	GOTO L	PC ← L
10	BEQ A B L	If M[A] = M[B] GOTO L
11	BNE A B L	If M[A] ≠ M[B] TO L
12	BGE A B L	If M[A] ≥ M[B] GOTO L
13	BLT A B L	If M[A] < M[B] GOTO L
14	JNS L	PC←L & PUSH PC
15	JR	PC ← M[M[SP]] & POP
16	READ	M[INPUT] ← Input
17	PRNT	Print M[OUTPUT]
18	STOP	Stop

Pseudo-Instructions

	Pseudo-instruction	Meaning	Instruction
1	MOVE B A	M[B] ← M[A]	ADD B A ZERO
2	GETI B I	M[B] ← M[M[I]]	GET B ZERO I
3	PUTI I A	M[M[I]] ← M[A]	PUT ZERO I A
4	BEQZ A L	If M[A] = 0 GOTO L	BEQ A ZERO L
5	BNEZ A L	If M[A] ≠ 0 GOTO L	BNE A ZERO L
6	BGEZ A L	If M[A] ≥ 0 GOTO L	BGE A ZERO L
7	BLTZ A L	If M[A] < 0 GOTO L	BLT A ZERO L
8	NEG A	M[A] ← - M[A]	SUB A ZERO A
9	POP	M[SP] = M[SP] - 1	AIMM SP SP -1
10	TOP A	M[A] ← M[M[SP]]	GET A ZERO SP
11	PUSH A	M[SP] = M[SP] + 1 M[M[SP]] ← M[A]	AIMM SP SP 1 PUT ZERO SP A

Two-address machine (R-to-R)

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 64K words of word-addressable data memory.
- 64K words of word-addressable instruction memory.
- 32-bit data words.
- 32-bit instructions.
- A 32-bit arithmetic logic unit (ALU).
  - Only arithmetic operations are implemented
- Registers
  - PC: 32-bit Program counter
  - SP: Stack pointer pointing to the top of the stack

Registers

- There are 32 registers: $0-$31, Similar to MIPS

• $zero	=	$0	Constant 0
• $v0-$v1	=	$2-$3	Results of a function,
• $a0-a3	=	$4-$7	Argument 1-4
• $t0-$t7	=	$8-$15	Temporary
• $sp	=	$29	Stack pointer
• $ra	=	$31	Return address

Implementations (R2R)

- Three separate 32-bit integer arrays are used for instructions, memory data, and input data, respectively.
- But, Each instruction takes only 26 bits
  - 5-bit opcode, 5-bit registers, and 16-bit operand (address).
- Instruction memory address is 16 bits. So, Instructions will take up to 64K 32-bit words (or integers).
- The branch instructions use 16-bit absolute address. The instructions JNS also uses 16-bit absolution address
- Each datum occupies 32 bits. The data address is 16 bits. So, we have 64K words of data memory.
- Stack is growing towards higher data memory address

Instruction Set

Two R2R

Imm:	16-bit 2's compliment
PC:	Program counter
R, R1:	Registers
L:	Label
M[R]:	Memory content at address R

op	Instruction			Meaning
0	LI	R	Imm	R ← Imm
1	ADDI	R	Imm	R ← R+Imm
2	ADD	R	R1	R ← R+R1
3	SUB	R	R1	R ← R-/R1
4	MUL	R	R1	R ← R*R1
5	DIV	R	R1	R ← R/R1
6	REM	R	R1	R ← R%R1
7	GET	R	R1	R ← M[R1]
8	PUT	R	R1	M[R1] ← R
9	GOTO	R	L	PC ← L
10	BEQZ	R	L	If R = 0 GOTO L
11	BNEZ	R	L	If R ≠ 0 GOTO L
12	BGEZ	R	L	If R ≥ 0 GOTO L
13	BLTZ	R	L	If R < 0 GOTO L
14	JNS		L	$ra ← & PC←L
15	JR			PC ← $ra
16	READ			$v0 ← Input
17	PRNT			Print $a0
18	STOP			Stop

Pseudo-Instructions

	Pseudo-instruction	Meaning	Instruction
1	LA R Var	R ← &Var	LI R Var
2	MOVE R2 R1	R2 ← R1	LI $at 0 ADD $at R1 LI R2 0 ADD R2 $at
3	NEG R	R ← -R	LI $at 0 SUB $at R LI R 0 ADD R $at
4	GETI R2 imm	R2 ← M[imm]	LI $at imm GET R2 $at
5	PUTI R2 imm	M[imm] ← R2	LI $at imm PUT R2 $at
6	GETV R2 Var	R2 ← M[Var]	LI $at var GET R2 $at
7	PUTV R2 Var	M[Var] ← R2	LI $at var PUT R2 $at
8	POP R	R ← M[$sp] $sp = $sp - 1	GET R $sp ADDI $sp -1
9	PUSH R	$sp = $sp + 1 M[$sp] ← R	ADDI $sp 1 PUT R $sp

Three-address machine (R2R16)

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 64K words of word-addressable data memory.
- 64K words of word-addressable instruction memory.
- 32-bit data words.
- 32-bit instructions.
- A 32-bit arithmetic logic unit (ALU).
  - Only arithmetic operations are implemented
- Registers
  - PC: 32-bit Program counter
  - SP: Stack pointer pointing to the top of the stack

Registers

- There are 32 registers: $0-$31, Similar to MIPS

• $zero	=	$0	Constant 0
• $v0-$v1	=	$2-$3	Results of a function,
• $a0-a3	=	$4-$7	Argument 1-4
• $t0-$t7	=	$8-$15	Temporary
• $sp	=	$29	Stack pointer
• $ra	=	$31	Return address

Implementations (R2R16)

- Three separate 32-bit integer arrays are used for instructions, memory data, and input data, respectively.
- But, Each instruction takes at most 31 bits
  - 5-bit opcode, two 5-bit registers, and 16-bit operand (address).
- Instruction memory address is 16 bits. So, Instructions will take up to 64K 32-bit words (or integers).
- The branch instructions use 16-bit absolute address. The instructions JNS also uses 16-bit absolution address
- Each datum occupies 32 bits. The data address is 16 bits. So, we have 64K words of data memory.
- Stack is growing towards higher data memory address

Instruction Set

Three R2R

Imm:	16-bit 2's compliment
offset:	Imm or address of variable
PC:	Program counter
R, R1:	Registers
L:	Label
M[R]:	Memory content at address R

op	Instruction	Meaning
0	LI R Imm	R ← Imm
1	ADDI R2 R1 Imm	R2 ← R1+Imm
2	ADD R3 R1 R2	R3 ← R1 + R2
3	SUB R3 R1 R2	R3 ← R1 + R2
4	MUL R3 R1 R2	R3 ← R1 + R2
5	DIV R3 R1 R2	R3 ← R1 + R2
6	REM R3 R1 R2	R3 ← R1 + R2
7	GET R2 R1 offset	R2 ← M[R1+ offset]
8	PUT R2 R1 offset	M[R1+ offset] ← R2
9	GOTO L	PC ← L
10	BEQ R1 R2 L	If R1 = R2 GOTO L
11	BNE R1 R2 L	If R1 ≠ R2 GOTO L
12	BGE R1 R2 L	If R1 ≥ R2 GOTO L
13	BLT R1 R2 L	If R1 < R2 GOTO L
14	JNS L	$ra ← PC & PC ←L
15	JR	PC ← $ra
16	READ	$v0 ← Input
17	PRNT	Print $a0
18	STOP	Stop

Pseudo-Instructions

	Pseudo-instruction	Meaning	Instruction
0	LA R Var	R ← &Var	Var: variable, 16-bit address
1	MOVE R2 R1	R2 ← R1	ADD R2 R1 $zero
2	NEG R	R ← - R	SUB R $zero R
3	GETR R2 R1	R2 ← M[R1]	GET R2 R1 0
4	PUTR R2 R1	M[R1] ← R2	PUT R2 R1 0
5	GETI R2 imm	R2 ← M[imm]	GET R2 $zero imm
6	PUTI R2 imm	M[imm] ← R2	PUT R2 $zero imm
7	GETV R2 Var	R2 ← M[Var]	GET R2 $zero var
8	PUTV R2 Var	M[Var] ← R2	PUT R2 $zero var
9	BEQZ R L	If R = 0 GOTO L	BEQ R $zero L
10	BNEZ R L	If R ≠ 0 GOTO L	BNE R $zero L
11	BGEZ R L	If R ≥ 0 GOTO L	BGE R $zero L
12	BLTZ R L	If R < 0 GOTO L	BLT R $zero L
13	POP R	R ← M[SP] SP ← SP – 1	GET R SP 0 ADDI SP SP -1
14	PUSH R	SP ← SP + 1 M[SP] ← R	ADDI SP SP 1 PUT R SP 0

Program Structure and Syntax (1)

- Every program contains three sections and separated by END
  - Data (optional)
  - Code
  - Input (optional)
- Data (Declarations)
  - One variable definition per line
  - **ID** (identifier) is the variable name.
  - **Type** is a positive integer
    - **Type** = 1, ID is a scalar variable
    - **Type** > 1, ID is an array variable
  - **Value** is up to **Type** initial integers of **ID.** If less than **Type** initial values are provided, default initial values are used.

[Data]
END
Code
END
[Input]

ID Type [Value]

[Label:] Instruction

Number (integer)

Program Structure and Syntax (2)

- Code (Instructions)
  - One instruction per line
  - **Label** is optional. It must be followed by ':' immediately. There is no space between Label and ':'.
  - **Instruction** is any instruction, including pseudo-instruction.
- Input:
  - One input value per line.
  - **Number** is any integer.
- Comments:
  - Any text starting from // to the end of the line will be considered as comments

[Data]
END
Code
END
[Input]

ID Type [Value]

[Label:] Instruction

Number (integer)

Example 1: Compute sum of absolute values of elements in an array

C++ code

```
int main() {
	int DAT [ 9] = {10, 20, 30, -40, 50, 60, 70, 80, -90}  //array
	int N = 9;		//number of elements in array
	int SUM = 0;	//sum
	int I;
	for (I = 0; I < N; I++)
		if (DAT < 0)
			DAT[I] = - DAT[I];
		SUM = SUM + DAT[I];
	std::cout << SUM;
	return 0;
}
```

Example 1: Stack-Based Code 1/2

```
//Declaration
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                //A pointer to the array DAT
DAT   9 10 20 30 -40 50 60 70 80 -90      //array DAT
END
//Instructions
L1:   PUSH      N
      PUSH      I
      SUB
      BEQZ      L3          //if (N-I)=0, done
      PUSHI     PDAT        //Get an array element
      PUSHI     PDAT        //Get the array element again for testing
      BGEZ      L2          //if positive, skip
```

Stack-Based Code 2/2

```
      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               //terminate program
END
```

Example 1: Accumulator-based Code 1/2

```
//Declaration
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                   //A pointer to the array DAT
DAT   9 10 20 30 -40 50 60 70 80 -90      //array DAT
END
//Instructions
L1:   GET       N
      SUB       I
      BEQZ      L3          //if (N-I)=0, done
      GETI      PDAT        //Get an array element into AC
      BGEZ      L2          //if positive, skip
```

Accumulator-based Code 2/2

```
      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            //stop
END
//Inputs
//None
```

Example 1: Two-Address M2M Code 1/2

```
I       1   0              //Array Index
SUM     1   0              //Sum
N       1   9              //Number of elements in the array
TMP     1   0              //Temporary location
N_I     1   0
PDAT    1   DAT            //PDAT is a pointer that contains address of DAT
DAT     9   10 20 30 -40 50 60 70 80 -90 //THE DATA ARRAY
END
```

Two-Address M2M Code

```
//Instructions
L1: MOVE    N_I N
    SUB     N_I I       //N_I = N-I
    BEQZ    N_I L3      //if N-I=0
    GET     TMP PDAT    //Get an element from array
    BGEZ    TMP L2      //if positive, skip
    NEG     TMP         //else, negate
L2: ADD     SUM TMP     //Add to sum
    AIMM    I 1         //increase index I by one
    AIMM    PDAT 1      //Pointing to next array element
    GOTO    L1          //next element
L3: MOVE    OUTPUT SUM //print sum
    PRNT
    STOP                //stop
END
//Inputs
//None
```

Example 1: Three-Address M2M Code 1/2

```
//Declaration
I      1     0              //Array Index
SUM 1      0              //Sum
N      1     9              //Number of elements in the array
TMP 1      0              //Temporary location
DAT  9     10 20 30 -40 50 60 70 80 -90  //DATA ARRAY
END
```

Three-Address M2M Code 2/2

```
//Instructions
L1:   BEQ     I     N     L3        //if I=N, done
      GET     TMP   DAT   I         //Get an array element into tmp
      BGEZ    TMP   L2              //if positive, skip
      NEG     TMP                   //else, negate
L2:   ADD     SUM   SUM   TMP       //add to sum
      AIMM    I     I     1         //increase index I by one
      GOTO    L1                    //next element
L3:   MOVE    OUTPUT SUM            //print average
      PRNT                          //Print
      STOP                          //stop
END
//Inputs
//None
```

Example 1: Two-Address R2R Code 1/2

```
//Declaration
NUM     1     9
DAT     9     10 20 30 -40 50 60 70 80 -90        //DATA ARRAY
SUM     1     0
END
```

Two-Address R2R Code

```
//Instructions
        GETV   $t0  NUM          //$t0 = NUM
        LA       $s0  DAT        //$s0 address of DAT
        MOVE $t3 $zero           //SUM=$t3=0
L1:   BEQZ   $t0  L3             //if finish reading the array
        GET     $t2  $s0         //Get an array element into %t2
        BGEZ   $t2  L2           //if positive, skip
        NEG    $t2  $t2          //else, negate
L2:   ADD    $t3  $t2          //add
        ADDI   $t0  -1           //$t0--
        ADDI   $s0  1            //$s0++
        GOTO  L1                 //next element
L3:   PUTV  $t3 SUM          //save SUM
        MOVE $a0 $t3             //Print SUM
        PRNT
        STOP                     //stop
END
//Input
//None
```

Example 1: Three-Address R2R Code 1/2

```
//Declaration
NUM     1     9
DAT     9     10 20 30 -40 50 60 70 80 -90        //DATA ARRAY
SUM     1     0
END
```

Three-Address R2R Code 2/2

```
//Instructions
        GETV   $t0 NUM          //$t0 = NUM
        LA     $s0 DAT          //$s0 = address of DAT
        MOVE $t3 $zero          //SUM=$t3=0
L1:     BEQZ   $t0 L3           //if finish reading the array
        GETR   $t2 $s0          //Get an array element into %t2
        BGEZ   $t2 L2           //if positive, skip
        NEG    $t2                      //else, negate
L2:     ADD    $t3 $t3 $t2      //to sum
        ADDI   $t0 $t0 -1       //$t0--
        ADDI   $s0 $s0 1        //$s0++
        GOTO   L1               //next element
L3:     PUTV   $t3 SUM
        MOVE  $a0 $t3
        PRNT
        STOP                    //stop
END
//Input
//None
```

Example 1: Compute sum of absolute values of elements in an array: Comparison

	Loop-control and array access	Abs values	Vars	insts
Stack	Compute N-I, exit loop if N-I = 0. Use pointer PDAT to access array. Increase I and PDAT for next iteration	Top of stack	6	25
Accumulator	The same as those for Stack	AC	6	20
2-Addr (M2M)	The same as those for Stack	Variable	7	18
3-Addr (M2M)	Exit loop if N=I. Use array base address DAT + index I to access array. Increase I for next iteration	Variable	5	10
2-Addr (R2R)	Exit loop if N=0 ($t0=N), Use pointer ($s0) to access array. Increase $s0 and decrease $t0 for next iteration	Register	3	25
3-Addr (R2R)	The same as those for 2-Addr (R2R)	Register	3	15

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

Mathematics formula

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

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;
  }
}
std::cout << C;
```

Example 2: Loop Solution

Zero Address Code

```
//Declarations
I 1 1          //index
N 1 0          //N
C 1 0          //f(N)
B 1 1          //f(N-1)
A 1 0          //f(N-2)
END
//Instructions
READ           //Read input
POP  N         //N=input
PUSH N
PUSH 2
SUB            //N-2
BGEZ L1        //if N>=2
PUSH N
POP C          //C=N, if N<=1
GOTO L2
```

```
L1:   PUSH B
      PUSH A
      ADD
      POP C          //C = B+A
      PUSH B         //A = B
      POP A
      PUSH C         //B = C
      POP B
      PUSH I         //I = I+1
      PUSH 1
      ADD
      POP I          //I = I+1
      PUSH I         //I-N
      PUSH N
      SUB            //I-N
      BLTZ L1        //if I<N, continue
L2:   PUSH C         //print result
      PRNT
      STOP           //stop
END
//Input
10                   //N
```

Example 2: Loop Solution

Accumulator-based Code

```
//Declarations
I 1 1      //index
N 1 0      //N
C 1 0      //f(N)
B 1 1      //f(N-1)
A 1 0      //f(N-2)
END
//Instructions
READ       //Read input
PUT N      //N=input
PUT C      //C=N
SUB I      //AC=N-1
BLTZ L2    //if N<1
BEQZ L2    //if N=1
```

```
L1: GET B      //AC=B
    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, cont.
L2: GET C      //print result
    PRNT
    STOP       //stop
    END
//inputs
10 //N
```

Example 2: Loop Solution

Two Address M2M Code

```
//Declarations
N     1  0        //N
C     1  0        //f(N)
B     1  1        //f(N-1)
A     1  0        //f(N-2)
N_I  1 0          //N_I=0
END
//Instruction
      READ              //Input
      MOVE  N  INPUT
      BNEZ  N   L1
      LIMM C   0      //C=0
      GOTO L3
```

```
L1:   MOVE N_I N          //N_I = N
      AIMM N_I -1         //N_I = N-1
      BNEZ N_I L2         //If N != 1
      LIMM C 1            //C=1
      GOTO L3
L2:   AIMM N_I -1         //N_1 = N-2
L4:   MOVE C B            //C = B
      ADD C A             //C = C + A
      MOVE A B            //A = B
      MOVE B C            //B = C
      AIMM N_I -1         //N_I --
      BGEZ N_I L4
L3:   MOVE OUTPUT C       //print result
      PRNT
      STOP                //stop
END
10
```

Example 2: Loop Solution

Three Address M2M Code

```
//Declarations
N     1  0        //N
C     1  0        //f(N)
B     1  1        //f(N-1)
A     1  0        //f(N-2)
END
```

```
//Instruction
      READ                  //Read Input
      MOVE  N  INPUT        //N = Input
      BNEZ   N  L1          //if N != 0
      LIMM   C  0           //C=0
      GOTO   L4
L1:   AIMM  N  N  -1        //N--
      BNEZ   N  L2          //if N != 1
      LIMM   C  1           //C=1
      GOTO   L4
L2:   AIMM  N  N  -1        //N--
L3:   ADD    C   B   A      //C = B + A
      MOVE  A  B            //A = B
      MOVE  B  C            //B = C
      AIMM  N  N  -1        //N--
      BGEZ   N  L3          //if N >=0 continue
L4:   MOVE  OUTPUT  C       //Print result
      PRNT
      STOP                  //stop
END
//Input
10
```

Example 2: Loop Solution

Two Address R2R Code

```
//Declarations
//None
END
//Instructions
      READ            //$v0 = N
      BNEZ $v0 L1     //if N != 0
      GOTO L4
L1:   LI $t0 1
      SUB $t0 $v0
      BNEZ $t0 L2     //if N != 1
      GOTO L4
```

```
L2:   LI      $t1  0        //$t1=A
      LI      $t2  1        //$t2=B
      MOVE $v1 $v0          //$v1=N,
      ADDI  $v1 -2          //$v1=N-2
L3:   ADD   $t1 $t2         //A=A+B
      MOVE $v0 $t1          //C=A
      MOVE $t1 $t2          //A=B
      MOVE $t2 $v0          //B=C
      ADDI  $v1 -1          //$v1 --
      BGEZ  $v1 L3          //if $v1 >= 0
L4:   MOVE $a0 $v0          //$a0 = Fib(N)
      PRNT                  //print result
      STOP                  //stop
END
10
```

Register assignment
$v1: N, $t1: A, $t2: B, $v0: C, $t0:

Example 2: Loop Solution

Three Address R2R Code

```
//Declarations
//None
END
//Instructions
      READ              //$v0 = N
      BNEZ $v0 L1       //if N != 0
      GOTO L4
L1:   LI $v1 1
      BNEZ $v0 $v1 L2 //if N != 1
      GOTO L4
```

```
L2:   LI      $t1  0          //$t1=A
      LI      $t2  1          //$t2=B
      ADDI  $v1  $v0 -2       //$v1=N-2
L3:   ADD   $v0  $t2 $t1      //$v0=C=B+A
      MOVE $t1  $t2           //A=B
      MOVE $t2  $v0           //B=C
      ADDI   $v1  $v1 -1      //$v1 --
      BGEZ   $v1  L3          //if $v1 >= 0
L4:   MOVE $a0  $v0           //$a0 = result
      PRNT                    //print result
      STOP                    //stop
END
//Input
10
```

Register assignment
$v1: N, $t1: A, $t2: B, $v0: C, $t0:

Example 2: Compute Fibonacci using a loop: Comparison (1)

3-Address	2-Address	1-address	0-address
C = A + B A = B B = C	A = A + B C = A A = B B = C	GET A ADD B PUT C GET B PUT A GET C PUT B	PUSH A PUSH B ADD POP C PUSH B POP A PUSH C POP B

Example 2: Compute Fibonacci using a loop: Comparison (2)

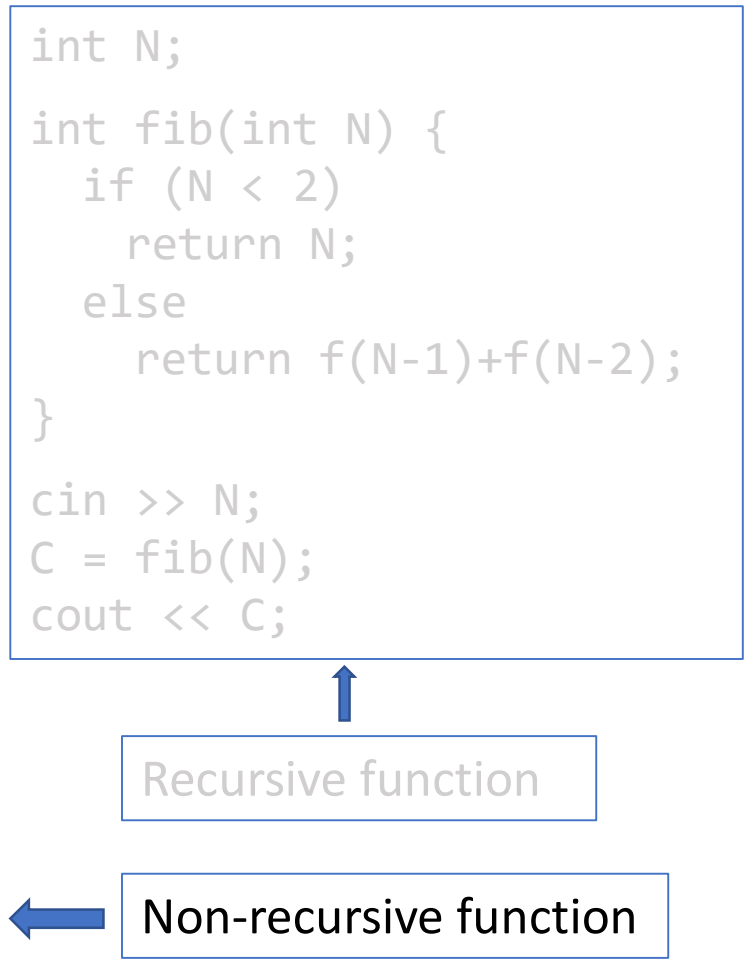
	Loop-control	Vars	insts
Stack	Compute I-N, continue loop if I-N < 0. Increase I for next iteration	5	28
Accumulator	The same as those for Stack	5	21
2-Addr (M2M)	Assign N to N_I, continue loop if N_I >= 0. Decrease N_I for next iteration	5	26
3-Addr (M2M)	Continue loop if N >= 0. Decrease N for next iteration	4	20
2-Addr (R2R)	Assign N to $v1, continue loop if $v1 >= 0. Decrease $v1 for next iteration	Registers used	35
3-Addr (R2R)	The same as those for 2-Addr (R2R)	Registers used	17

Example 2: C++ Code Using Function

```
int N;

int fib(int N) {
  int I, A, B, C;
  if (N < 2)
    return 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;
```



Example 2: Using Function

Zero Address Code 1/2

```
//No memory data declaration
END
        READ
        JNS  Fib        //call Fib
        PRNT
        STOP
Fib:    MOVE FP SP
        PUSH 1          //i=1
        PUSH 0          //a=0
        PUSH 1          //b=1
        PUSH 0          //c=0
```

Activation Record

Addr	Name	Explanation
FP-1	N/Fib(N)	Input/output
FP	RA	Return Address
FP+1	I	Local variable
FP+2	A	Local variable
FP+3	B	Local variable
FP+4	C	Local variable

Zero Address Code 2/2

```
    PUSH FP-1  //N
    PUSH 2     //2
    SUB        //N-2
    BGEZ L1    //if N>=2
    GOTO L2    //f(N)=N if N<2
L1: PUSH FP+3  //B
    PUSH FP+2  //A
    ADD        //B+2
    POP FP+4   //C=B+A
    PUSH FP+3  //B
    POP FP+2   //A=B
    PUSH FP+4  //C
    POP FP+3   //B=C
```

```
    PUSH FP+1  //I
    PUSH 1     //1
    ADD        //I+1
    POP FP+1   //I=I+1
    PUSH FP+1  //I
    PUSH FP-1  //N
    SUB        //I-N
    BLTZ L1    //if i<N, continue
    POP FP-1
L2: ISP -3
    JR 0
    END
//Inputs
10             N=10
```

Example 2: Using Function

Accumulator-based Code 1/2

```
N 1 0   //N: input
C 1 0   //C: result fib(N)
END
//main code
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
```

Stack Frame or Activation Record

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	I	Local variable
$+2	A	Local variable
$+3	B	Local variable
$+4	C	Local variable

Accumulator-based Code 2/2

```
Fib: LIMM 1
    PUT $+1     //I=1
    PUT $+3     //B=1
    AIMM -1
    PUT $+2     //A=0
    GET $-1     //AC=N
    SUB $+1     //N-1
    BLTZ L3     //N<1
    BEQZ L3     //N=1
```

```
L2: GET $+3     //AC=B
    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, cont.
    GET $+4     //AC=C
    PUT $-1     //copy result
L3: JR
END
10
```

Example 2: Using Function

Two Address M2M Code 1/2

```
N     1     0     //N
C     1     0     //f(N)
END
      READ
      MOVE      N     INPUT
      PUSH      N
      JNS       Fib             //call Fib
      TOP       C
      POP
      MOVE      OUTPUT   C
      PRNT
      STOP
```

Activation Record

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	A	Local variable
$+2	B	Local variable
$+3	C	Local variable
$+4	N_I	Local variable

Two Address M2M Code 2/2

```
Fib:  BNEZ    $-1  L1      //if N != 0
      GOTO    L4
L1:   MOVE    $+4  $-1     //N_I = N
      AIMM    $+4  -1      //N_I = N-1
      BNEZ    $+4  L2      //if N != 1
      GOTO    L4
L2:   AIMM    $+4  -1      //N_I = N-2
      LIMM    $+1  0       //A = 0
      LIMM    $+2  1       //B = 1
L3:   MOVE    $+3  $+2     //C = B
      ADD     $+3  $+1     //C = C + A
      MOVE    $+1  $+2     //A = B
      MOVE    $+2  $+3     //B = C
      AIMM    $+4  -1      //N_I --
      BGEZ    $+4  L3      //if N_I >= 0
      MOVE    $-1  $+3     //copy result
L4:   JR
END
10
```

Example 2: Using Function

Three Address M2M Code 1/2

```
//Declaration
N     1     0     //N
C     1     0     //f(N)
END
//Instruction
      READ                  //Read Input
      MOVE  N  INPUT        //N = Input
      PUSH   N
      JNS     Fib           //call Fib
      TOP     C             //POP C
      POP
      MOVE  OUTPUT  C       //print C
      PRNT
      STOP
```

Activation Record

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	N_I	Local variable
$+2	A	Local variable
$+3	B	Local variable
$+4	C	Local variable

Three Address M2M Code 2/2

```
Fib:  BNEZ  $-1  L1              //if N != 0
      GOTO L4
L1:   AIMM $+1 $-1 -1            //N_I = N-1
      BNEZ  $+1  L2              //if N != 1
      GOTO L4
L2:   AIMM $+1  $+1 -1           //N_I--
      LIMM $+2  0                //A = 0
      LIMM $+3  1                //B = 1
L3:   ADD$+4 $+3 $+2             //C = B + A
      MOVE $+2 $+3               //A = B
      MOVE $+3 $+4               //B = C
      AIMM $+1 $+1 -1            //N_I--
      BGEZ   $+1  L3             //if N_I >= 0 continue
      MOVE $-1  $+4              //copy result
L4:   JR
//Inputs
END
10
```

Example 2: Using Function

No Activation Record Is needed

Two Address R2R Code

```
//Declarations
//None
END
//Instructions
        READ                    //$v0 = N
        MOVE $a0 $v0            //$a0 = $v0
        JNS Fib                 //$v0 = Fib(N)
        MOVE $a0 $v0            //$a0=$v0
        PRNT                    //Print $a0
        STOP
```

Register assignment

$a0: input to function

$v0: output from function

$t3: N/N-1, $t0: A, $t1: B, $v0: C, $t2: I

```
Fib: MOVE   $v0 $a0     //$v0=$a0
     LI     $t2 2       //$t2=2
     MOVE   $t3 $a0     //$t3=N
     SUB    $t3 $t2     //$t3=N-2
     BLTZ   $t3 L2      //N<2
     LI     $t0 0       //$t0=A
     LI     $t1 1       //$t1=B
L1:  ADD    $t0 $t1     //A=B+A
     MOVE   $v0 $t0     //$v0=C=A
     MOVE   $t0 $t1     //A=B
     MOVE   $t1 $v0     //B=C
     ADDI   $t2 1       //$t2++
     MOVE   $t3 $a0     //$t3=N
     SUB    $t3 $t2     //$t3=N-$t2
     BGEZ   $t3 L1
L2:JR
END
10
```

Example 2: Using Function

No Activation Record Is needed

Three Address R2R Code

```
//Declarations
//None
END
//Instructions
        READ                    //$v0 = N
        MOVE $a0 $v0            //$a0 = $v0
        JNS Fib                 //$v0 = Fib(N)
        MOVE $a0 $v0            //$a0=$v0
        PRNT                    //Print $a0
        STOP
```

Register assignment
$a0: input to function
$v0: output from function
$t0: A, $t1: C, $v0: B/C, $t2: I

```
Fib:  MOVE $v0  $a0          //$v0=$a0
      LI     $t2  2          //$t2=2
      BLT    $a0  $t2 L2     //N<2
      LI     $t0  0          //$t0=0
      LI     $v0  1          //$v0=1
L1:   ADD    $t1  $t0  $v0   //$t1=$t0+$v0
      MOVE $t0  $v0          //$t0=$v0
      MOVE $v0  $t1          //$v0=$t1
      ADDI   $t2  $t2 1      //$t2++
      BGE    $a0  $t2 L1     //if $a0 >= $t2
L2:   JR
//Input
END
10
```

Example 2: Compute Fibonacci using a function: Comparison

	Stack Frames	Local Vars	insts
Stack	1 input/output, 1 return address, 4 local variables, accessed via FP	4	34
Accumulator	1 input/output, 1 return address, 4 local variables, accessed via $	4	41
2-Addr (M2M)	Same as Accumulator	4	34
3-Addr (M2M)	Same as Accumulator	4	24
2-Addr (R2R)	No. Using registers to pass input/output, use registers for local variables	Registers used	52
3-Addr (R2R)	Same as 2-Addr (R2R)	Registers used	17

Parameter passing and local variables for architecture with in general-purpose registers can utilize these registers. Otherwise, stack is usually required.

Example 2: C++ Code with Using Recursive Function

```
int N;

int fib(int N) {
  int I, A, B, C;
  if (N < 2)
    return 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;

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

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


Recursive function

Non-recursive function

Example 2: Using Recursive Function

Zero Address Code 1/2

```
//No memory data declaration
END
     READ
     JNS      Fib      //call Fib
     PRNT
     STOP
Fib: PUSH     FP
     MOVE     FP SP    //FP=SP
     PUSH     FP-2     //N
     PUSH     2        //2
     SUB               //N-2
     BGEZ     L1       //N>=2
     GOTO     L2       //f(N)=N if N<2
```

Activation Record

Addr	Name	Explanation
FP-2	N/Fib(N)	Input/output
FP-1	RA	Return Address
FP	Prior FP	Previous FP
FP+1	(N-1)/Fib(N-1)	Input/output
FP+2	(N-2)/Fib(N-2)	Input/output

Zero Address Code 2/2

```
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
L2: POP     FP
    JR      0
    END
//Inputs
10                  //N=10
```

Activation Record

Addr	Name	Explanation
FP-2	N/Fib(N)	Input/output
FP-1	RA	Return Address
FP	Prior FP	Previous FP
FP+1	(N-1)/Fib(N-1)	Input/output
FP+2	(N-2)/Fib(N-2)	Input/output

Example 2: Using Recursive Function

Accumulator-based Code 1/3

```
N 1 0   //N: input
C 1 0   //C: result fib(N)
END
//main code
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
```

Activation Record of Fib(N)

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	(N-1)/Fib(N-1)	Input/output
$+2	(N-2)/Fib(N-2)	Input/output

```
//compute Fib(N)
Fib:    GET     $-1     //AC=N
        BNEZ    L1      //Fib(N)=0 if N=0
        GOTO    L3
L1:     AIMM    -1      //AC=N-1
        BNEZ    L2      //Fib(N)=1 if N=1
        GOTO    L3
L2:     PUT     $+1     //N-1 save to $+1
        GET     SP
        AIMM    1
        PUT     SP
        JNS     Fib     //call Fib(N-1)
```

Accumulator-based Code 2/3

Right Before
Fib(N-1) is called
Right After
Fib(N-1) is returned

Activation Record of Fib(N)

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	(N-1)/Fib(N-1)	Input/output
$+2	(N-2)/Fib(N-2)	Input/output

Addr	Name	Explanation
$-2	N/Fib(N)	Input/output
$-1	RA	Return Address
$	(N-1)/Fib(N-1)	Input/output
$+1	(N-2)/Fib(N-2)	Input/output

Accumulator-based Code 3/3

```
    GET  $-2    //AC=N
    AIMM -2     //AC=N-2
    PUT  $+1    //N-2 save to $+1
    GET  SP
    AIMM 1
    PUT  SP
    JNS  Fib    //Call Fib(N-2)
    GET  SP     //POP 2 times
    AIMM -2
    PUT  SP
    GET  $+1    //AC=F(N-1)
    ADD  $+2    //AC=F(N-1)+F(N-2)
    PUT  $-1
L3: JR
END
10              //Input 10
```

Addr	Name	Explanation
$-2	N/Fib(N)	Input/output
$-1	RA	Return Address
$	(N-1)/Fib(N-1)	Input/output
$+1	(N-2)/Fib(N-2)	Input/output

Right Before
Fib(N-2) is called
Right After
Fib(N-2) is returned

Addr	Name	Explanation
$-3	N/Fib(N)	Input/output
$-2	RA	Return Address
$-1	(N-1)/Fib(N-1)	Input/output
$	(N-2)/Fib(N-2)	Input/output

Example 2: Using Recursive Function

Two Address M2M Code 1/2

```
N     1     0     //N
C     1     0     //f(N)
END
      READ
      MOVE      N     INPUT
      PUSH      N
      JNS       Fib             //call Fib
      TOP       C
      POP
      MOVE      OUTPUT   C
      PRNT
      STOP
```

Activation Record

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	(N-1)/Fib(N-1)	Input/output
$+2	(N-2)/Fib(N-2)	Input/output

Two Address M2M Code 2/2

```
Fib: BNEZ  $-1 L1
    GOTO   L3          //if N != 0
L1: MOVE   $+1 $-1     //$+1 = N-1
    AIMM   $+1 -1
    BNEZ   $+1 L2      // if N != 1
    GOTO   L3
L2: AIMM   SP  1       //PUSH N-1
    JNS    Fib         //compute F(N-1)
    AIMM   SP  1       //PUSH N-2
    MOVE   $+0 $-3
    AIMM   $+0 -2
    JNS    Fib         //Compute F(N-2)
    AIMM   SP  -2      //POP 2 times
    MOVE   $-1 $+1     //F(N) = F(N-1) + F(N-2)
    ADD    $-1 $+2
L3: JR
END
10
```

After Fib(N) is called

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	(N-1)/Fib(N-1)	Input/output
$+2	(N-2)/Fib(N-2)	Input/output

After Fib(N-1) is called inside Fib(N)

Addr	Name	Explanation
$-2	N/Fib(N)	Input/output
$-1	RA	Return Address
$	(N-1)/Fib(N-1)	Input/output
$+1	(N-2)/Fib(N-2)	Input/output

After Fib(N-2) is called inside Fib(N)

Addr	Name	Explanation
$-3	N/Fib(N)	Input/output
$-2	RA	Return Address
$-1	(N-1)/Fib(N-1)	Input/output
$	(N-2)/Fib(N-2)	Input/output

Example 2: Using Recursive Function

Three Address M2M Code 1/2

```
//Declaration
N     1     0     //N
C     1     0     //f(N)
END
//Instruction
      READ                  //Read Input
      MOVE  N  INPUT        //N = Input
      PUSH   N
      JNS     Fib           //call Fib
      TOP     C             //POP C
      POP
      MOVE  OUTPUT  C  //print C
      PRNT
      STOP
```

Activation Record

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	(N-1)/Fib(N-1)	Input/output
$+2	(N-2)/Fib(N-2)	Input/output

Three Address M2M Code 2/2

```
//Compute f(N)
Fib:  BNEZ  $-1  L1
      GOTO L3                  //if N != 0
L1:   AIMM $+1  $-1  -1        //$+1 = N-1
      BNEZ   $+1  L2           // if N != 1
      GOTO  L3
L2:   AIMM  SP  SP  1          //PUSH N-1
      JNS  Fib                 //compute F(N-1)
      AIMM  SP  SP  1          //PUSH N-2
      AIMM  $+0  $-3  -2
      JNS  Fib                 //Compute F(N-2)
      AIMM  SP  SP  -2         //POP 2 times
      ADD  $-1  $+1  $+2       //F(N) = F(N-1) + F(N-2)
L3:   JR
END
//Input
10
```

After Fib(N) is called

Addr	Name	Explanation
$-1	N/Fib(N)	Input/output
$	RA	Return Address
$+1	(N-1)/Fib(N-1)	Input/output
$+2	(N-2)/Fib(N-2)	Input/output

After Fib(N-1) is called inside Fib(N)

Addr	Name	Explanation
$-2	N/Fib(N)	Input/output
$-1	RA	Return Address
$	(N-1)/Fib(N-1)	Input/output
$+1	(N-2)/Fib(N-2)	Input/output

After Fib(N-2) is called inside Fib(N)

Addr	Name	Explanation
$-3	N/Fib(N)	Input/output
$-2	RA	Return Address
$-1	(N-1)/Fib(N-1)	Input/output
$	(N-2)/Fib(N-2)	Input/output

Example 2: Using Recursive Function

Two Address R2R Code 1/2

```
//Declarations
//None
END
//Instructions
        READ                    //$v0 = N
        MOVE $a0 $v0            //$a0 = $v0
        JNS Fib                 //$v0 = Fib(N)
        MOVE $a0 $v0            //$a0=$v0
        PRNT                    //Print $a0
        STOP
```

Activation Record

	Saved register	Used for
	$a0	Input N
	$s0	Fib(N-1)
	$ra	Return address

Two Address R2R Code 2/2

```
//compute $v0 = Fib(N)
Fib: PUSH $a0          //save $a0
    PUSH $s0           //save $s0
    PUSH $ra           //save $ra
    MOVE $v0 $a0       //$v0 = $a0
    ADDI $a0 -2        //$a0 = N-2
    BLTZ $a0 L2        //if N<2
    ADDI $a0 1         //$a0 = N-1
    JNS Fib            //compute $v0 = Fib(N-1)
    MOVE $s0 $v0       //$s0 = Fib(N-1)
    ADDI $a0 -1        //$a0 = N-2
    JNS Fib            //compute $v0 = Fib(N-2)
    ADD $v0 $s0        //$v0 = Fib(N-2) + Fib(N-1)
L2: POP $ra
    POP $s0
    POP $a0
    JR
    END
```

Example 2: Using Recursive Function

Three Address R2R Code 1/2

```
//Declarations
//None
END
//Instructions
        READ                    //$v0 = N
        MOVE $a0 $v0            //$a0 = $v0
        JNS Fib                 //$v0 = Fib(N)
        MOVE $a0 $v0            //$a0=$v0
        PRNT                    //Print $a0
        STOP
```

Activation Record

	Saved register	Used for
	$a0	Input N
	$s0	Fib(N-1)
	$ra	Return address

Three Address R2R Code 2/2

```
//Compute $v0 = Fib(N)
Fib:   PUT    $a0  $sp  1          //push $a0
       PUT    $s0  $sp  2          //push $s0
       PUT    $ra  $sp  3          //push $ra
       ADDI   $sp  $sp 3
       MOVE $v0  $a0               //compute f(N)
       ADDI   $a0  $a0  -2         //$a0 = N-2
       BLTZ   $a0  L2              //if N<2
       ADDI   $a0  $a0  1          //$a0 = N-1
       JNS    Fib                  //compute $v0 = fib(N-1)
       MOVE $s0  $v0               //save fib(N-1) to $s0
       ADDI   $a0  $a0 -1          //$a0=N-2
       JNS    Fib                  //compute $v0 = fib(N-2)
       ADD    $v0  $v0  $s0        //$v0=fib(N-2)+fib(N-1)
L2:    ADDI   $sp  $sp  -3
       GET    $a0  $sp  1          //pop $a0
       GET    $s0  $sp  2          //pop $s0
       GET    $ra  $sp  3          //pop $ra
       JR
END
//inputs
10
```

Example 2: Compute Fibonacci using a recursive function: Comparison (1)

	Stack frame	Local Vars	insts
Stack	1 input/output, 1 return address, 1 previous FP, and 2 recursive calls Fib(N-1) and Fib(N-2). Accessed via FP.	4	34
Accumulator	1 input/output, 1 return address, and 2 calls Fib(N-1) and Fib(N-2). Accessed via $. But value of $ is changing when push or pop	4	41
2-Addr (M2M)	Same as Accumulator	4	34
3-Addr (M2M)	Same as Accumulator	4	24
2-Addr (R2R)	3 saved registers, $a0, $s0, $ra	Registers used	52
3-Addr (R2R)	Same as 2-Addr (R2R)	Registers used	24

Example 2: Compute Fibonacci using a recursive function: Comparison (2)

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

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

CONCLUSIONS

- Several computer architecture simulators are presented.
  - Similar instruction set
  - Similar assembly language program structure and syntax
- Several example assembly language programs are also given to illustrate many basic programming concepts and techniques at the assembly language level
  - Dealing with array, loop, function call and return, parameter passing, local variables, stack frame, and recursion
  - Register selection and usage
  - Compute using different instruction formats
  - Access array using either indirect or based memory addressing modes.