Virtual Machine Tutorial

CSA2201 – Compiler Techniques

Gordon Mangion

Virtual Machine

- A software implementation of a computing environment in which an operating system or program can be installed and run.

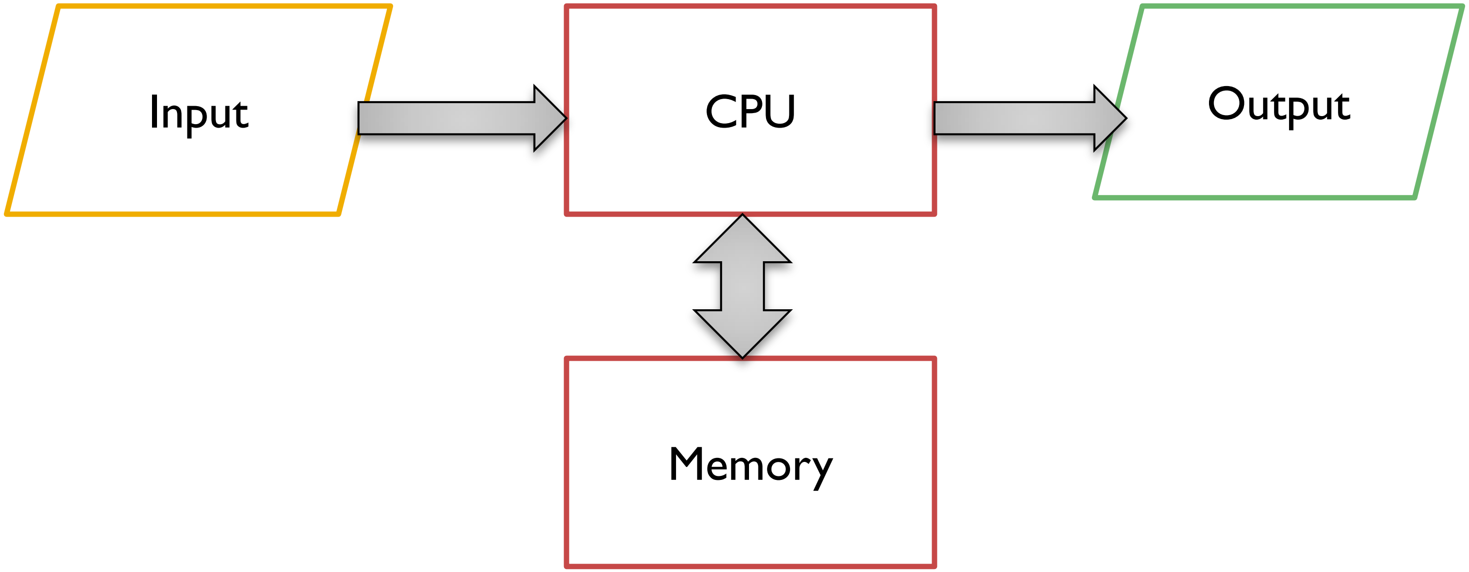
http://searchservervirtualization.techtarget.com/definition/virtual-machine

- A **process** virtual machine is designed to run a single process (program).

http://en.wikipedia.org/wiki/Virtual_machine

Computing Environment

- Computing Environment
  - CPU
  - Memory
  - I/O



Central Processing Unit

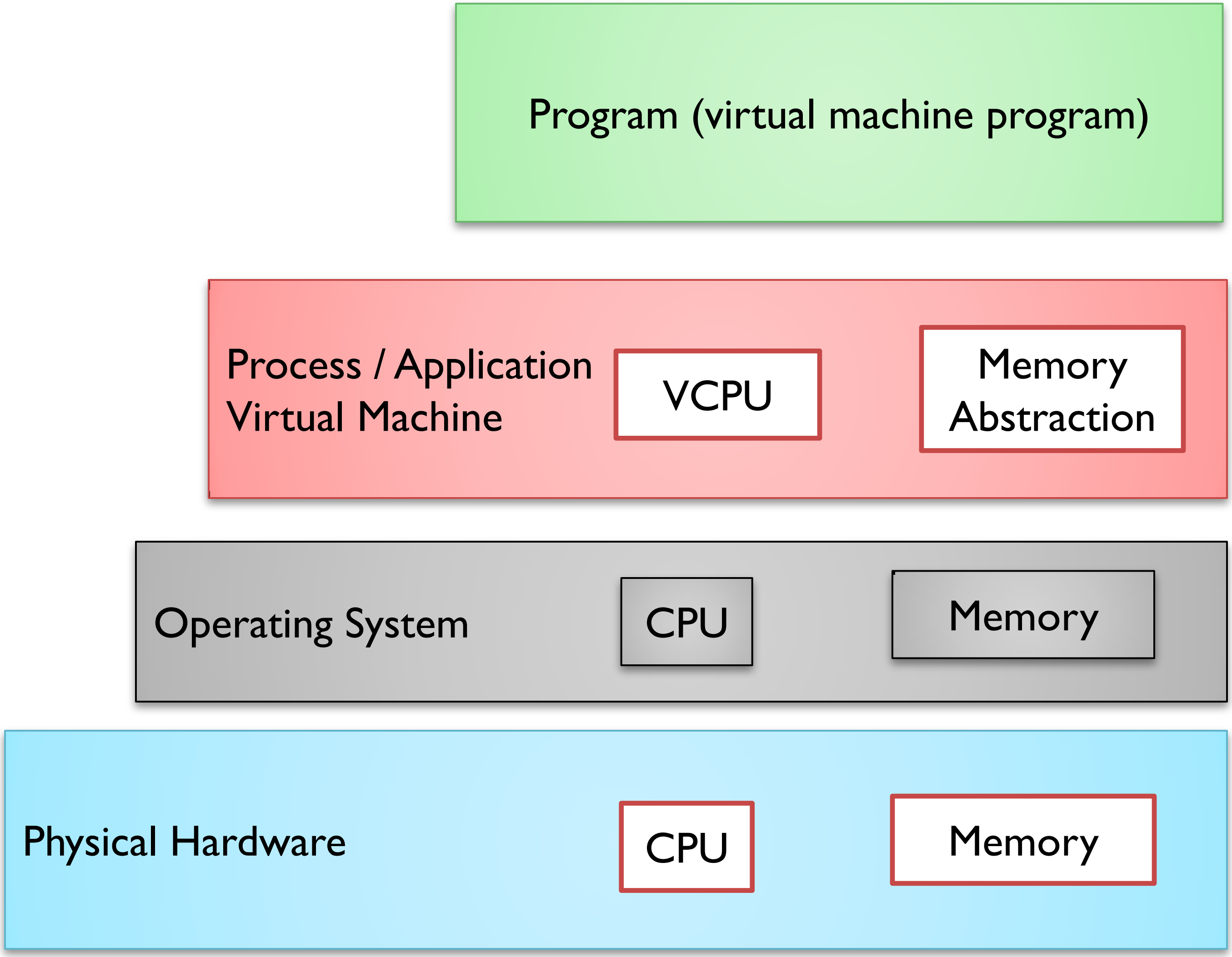
- CPU
  - Registers
  - Arithmetic Operations
  - Logic Operations
  - Flow Control of Programs

- Programs for one CPU usually do not work with another type of CPU

Virtual Machine

- Level Of Abstraction
  - Virtual CPU
  - Virtual Memory Model
  - Controlled Input / Output

Virtual Machine Architecture



Program (virtual machine program)

Process / Application Virtual Machine

VCPU

Memory Abstraction

Operating System

CPU

Memory

Physical Hardware

CPU

Memory

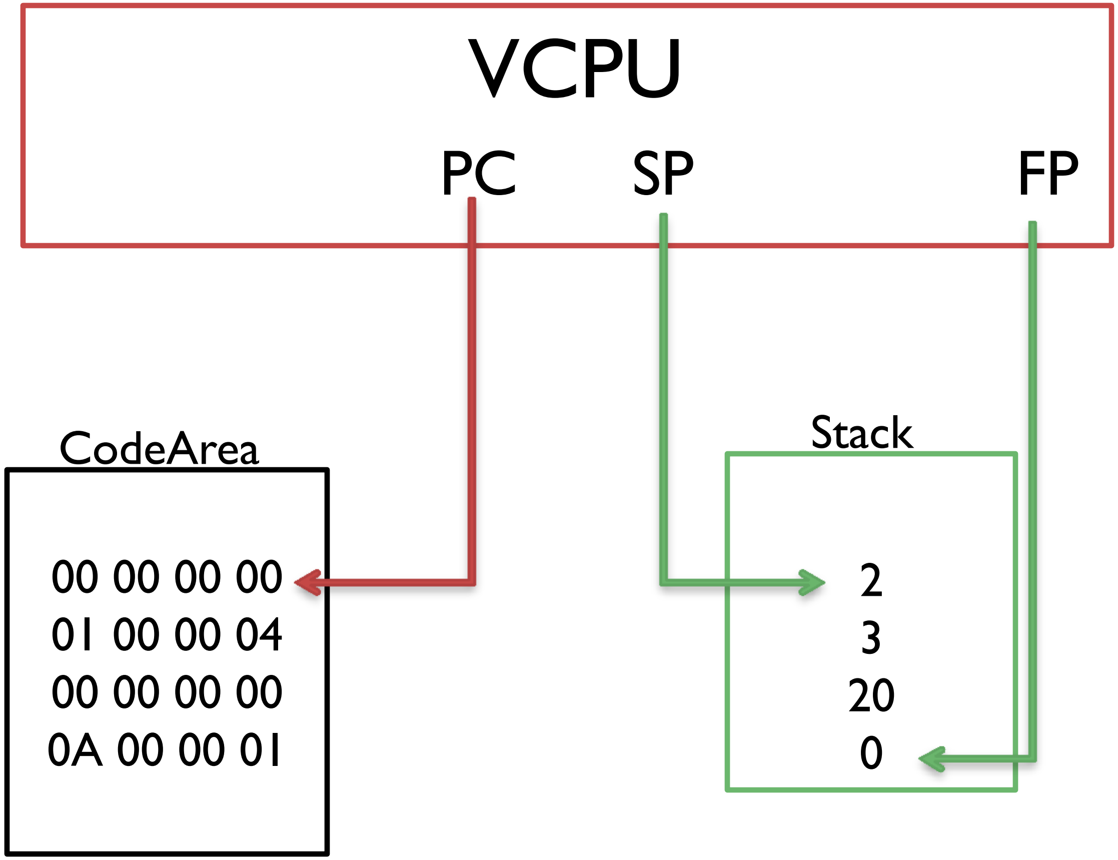
Virtual Machine Architecture

- VCPU
  - Either Register based
  - Or Stack based
- Code Area
  - Where Program(s) is loaded and executed
- Data Area
  - Where program data is manipulated
- Stack

S-Machine

- VCPU
  - Stack based
  - 3 registers
    - PC, Program Counter
    - SP, Stack Pointer
    - FP, Frame Pointer
- Code Area
  - Where S-Machine Programs are loaded
- Stack Area
  - All the operations are stack based
  - Data is kept in the machine stack

S-Machine



```
00 00 00 00
01 00 00 04
00 00 00 00
0A 00 00 01
```

```
2
3
20
0
```

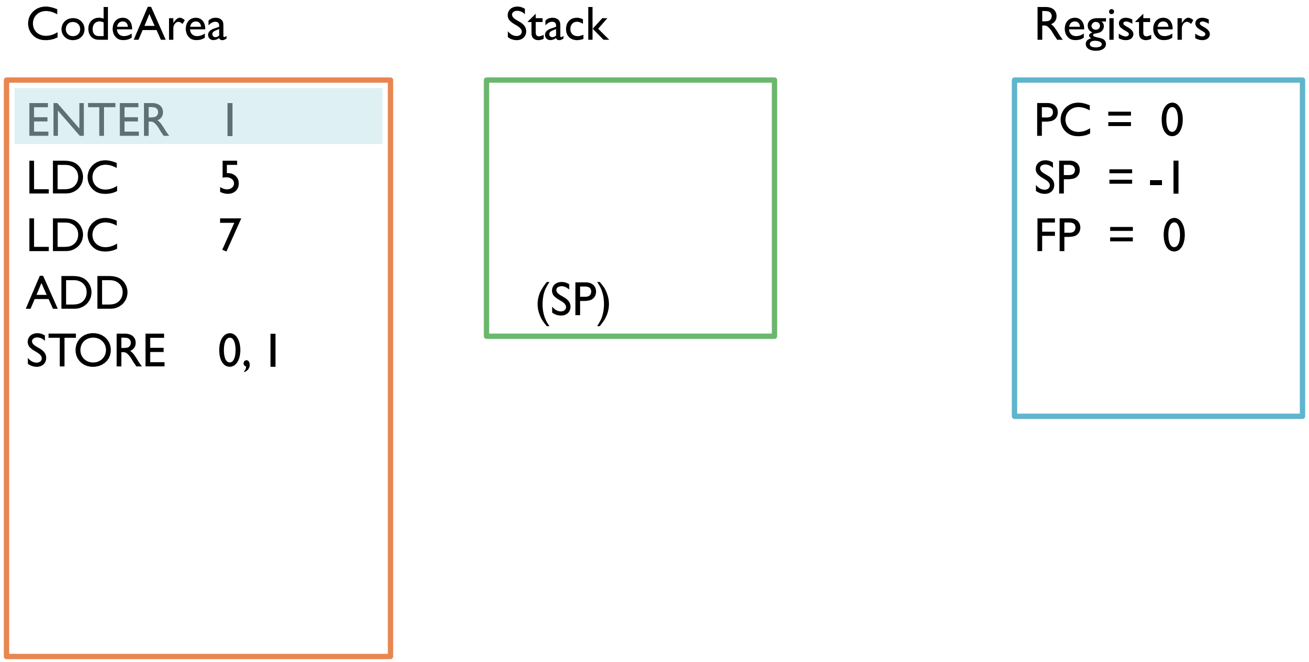
S-Machine

- Initial Conditions
  - PC = 0x0000
  - SP = 0xFFFF
  - FP = 0x0000

  - Code Area empty
  - Stack empty

S-Machine Instructions

- A = 5 + 7



CodeArea

```
ENTER    1
LDC      5
LDC      7
ADD
STORE    0, 1
```

Stack

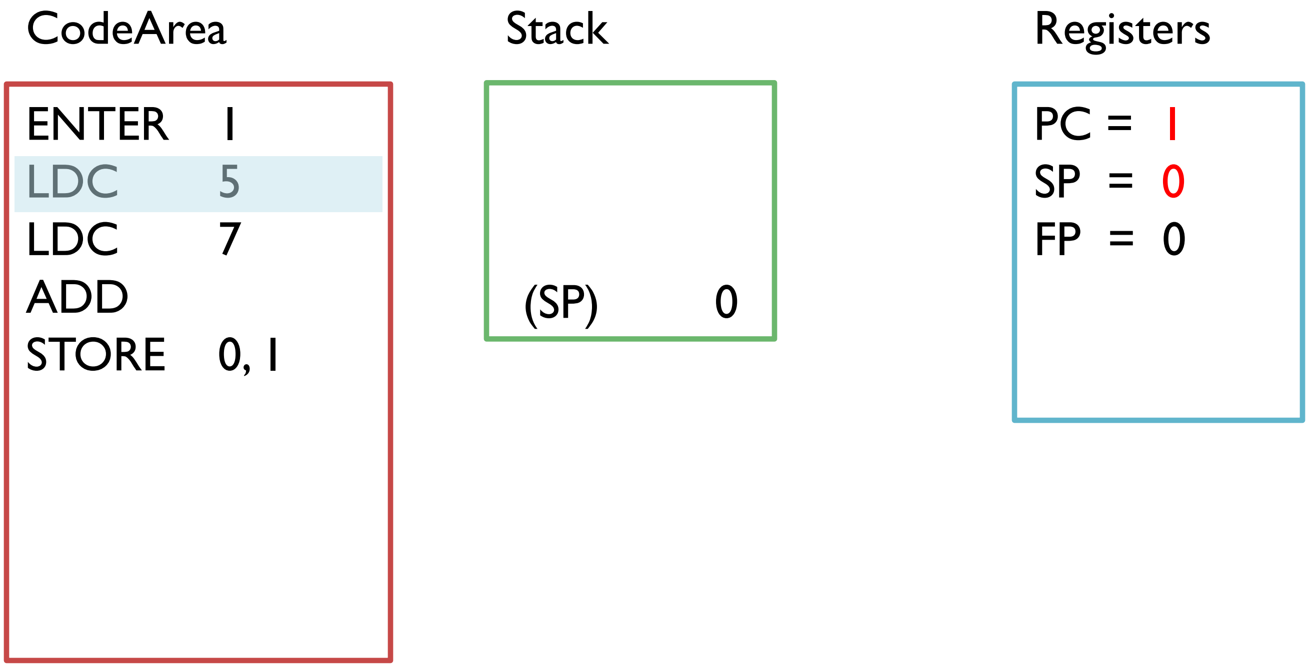
(SP)

Registers

PC = 0
SP = -1
FP = 0

S-Machine Instructions

- A = 5 + 7



CodeArea

```
ENTER    1
LDC      5
LDC      7
ADD
STORE    0, 1
```

Stack

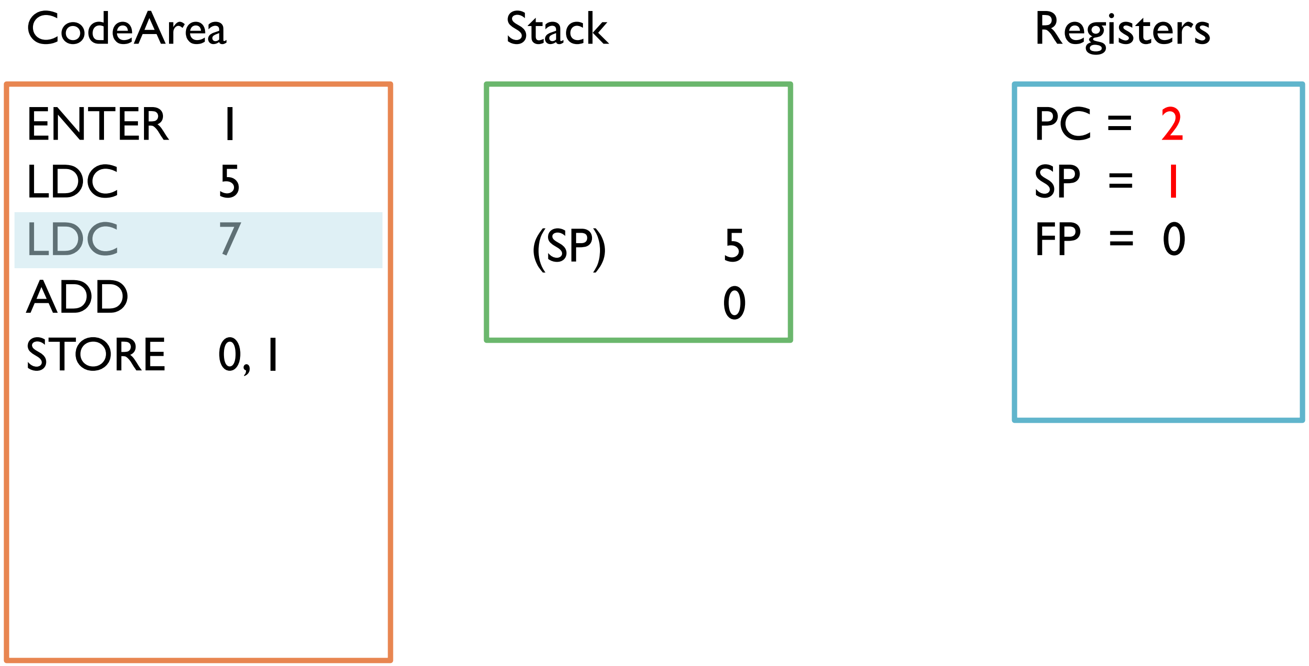
(SP) 0

Registers

PC = 1
SP = 0
FP = 0

S-Machine Instructions

- A = 5 + 7



CodeArea

```
ENTER   1
LDC     5
LDC     7
ADD
STORE   0, 1
```

Stack

(SP) 5

0

Registers

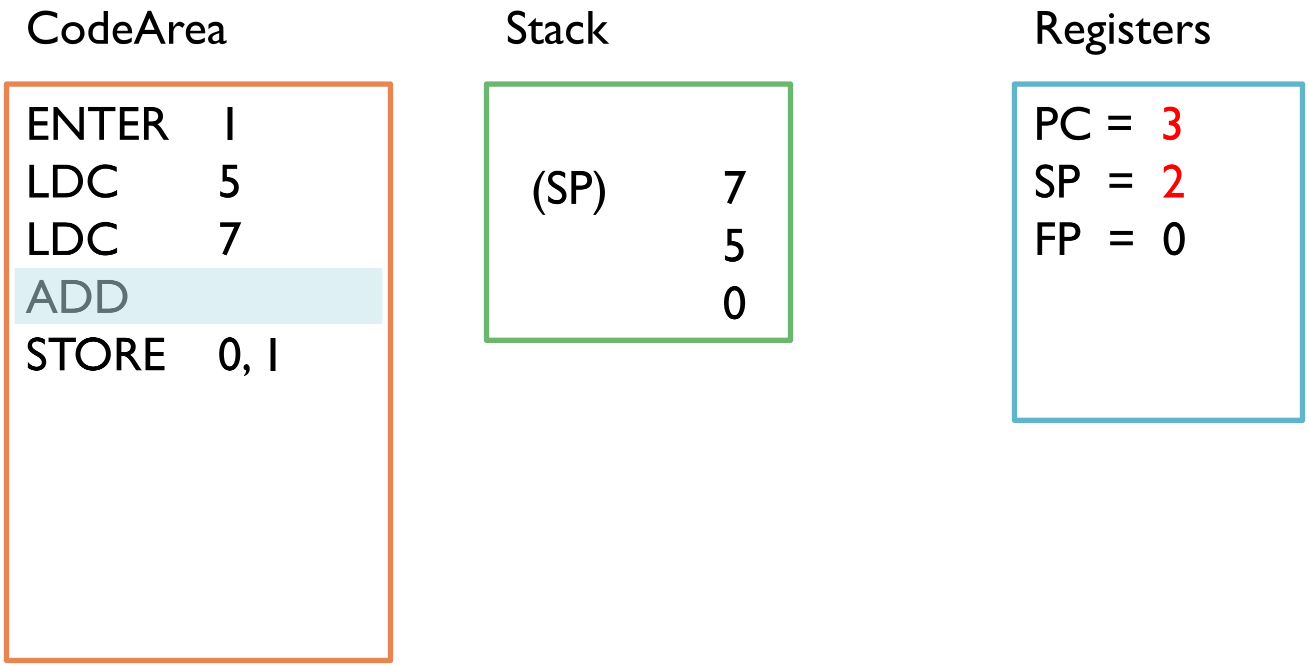
PC = 2

SP = 1

FP = 0

S-Machine Instructions

- A = 5 + 7



CodeArea

```
ENTER    1
LDC      5
LDC      7
ADD
STORE    0, 1
```

Stack

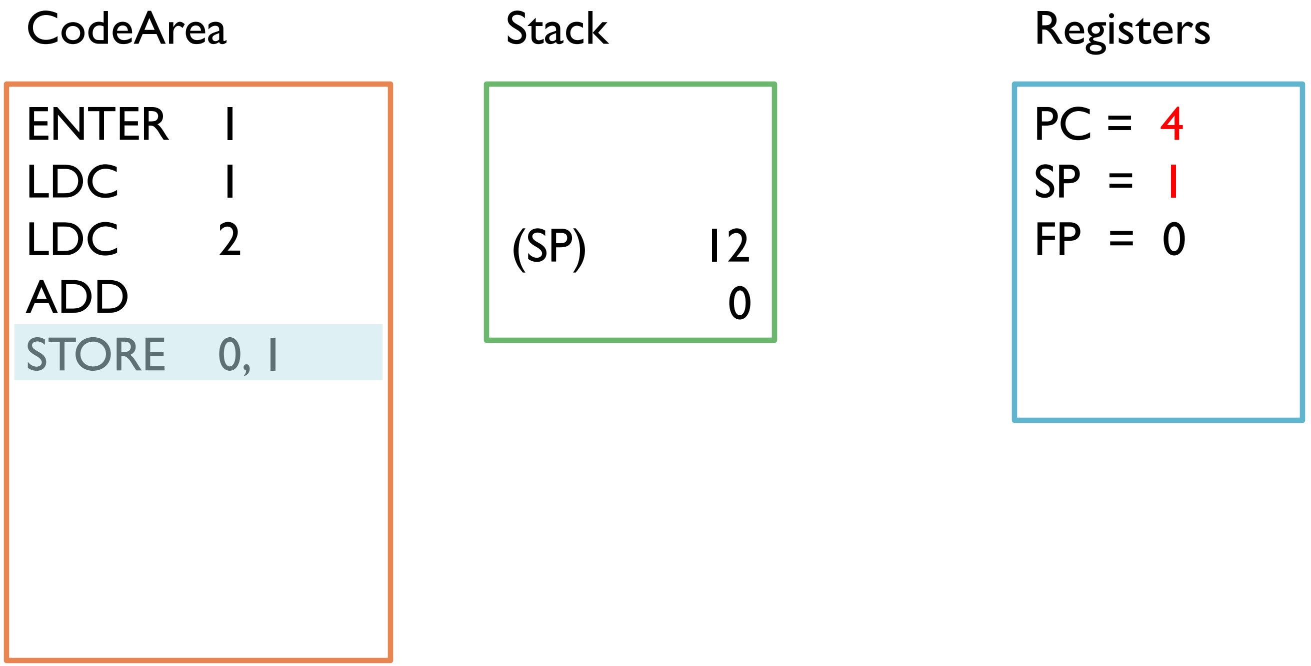
```
(SP)     7
         5
         0
```

Registers

```
PC =  3
SP =  2
FP =  0
```

S-Machine Instructions

- A = 5 + 7



CodeArea

```
ENTER    1
LDC      1
LDC      2
ADD
STORE    0, 1
```

Stack

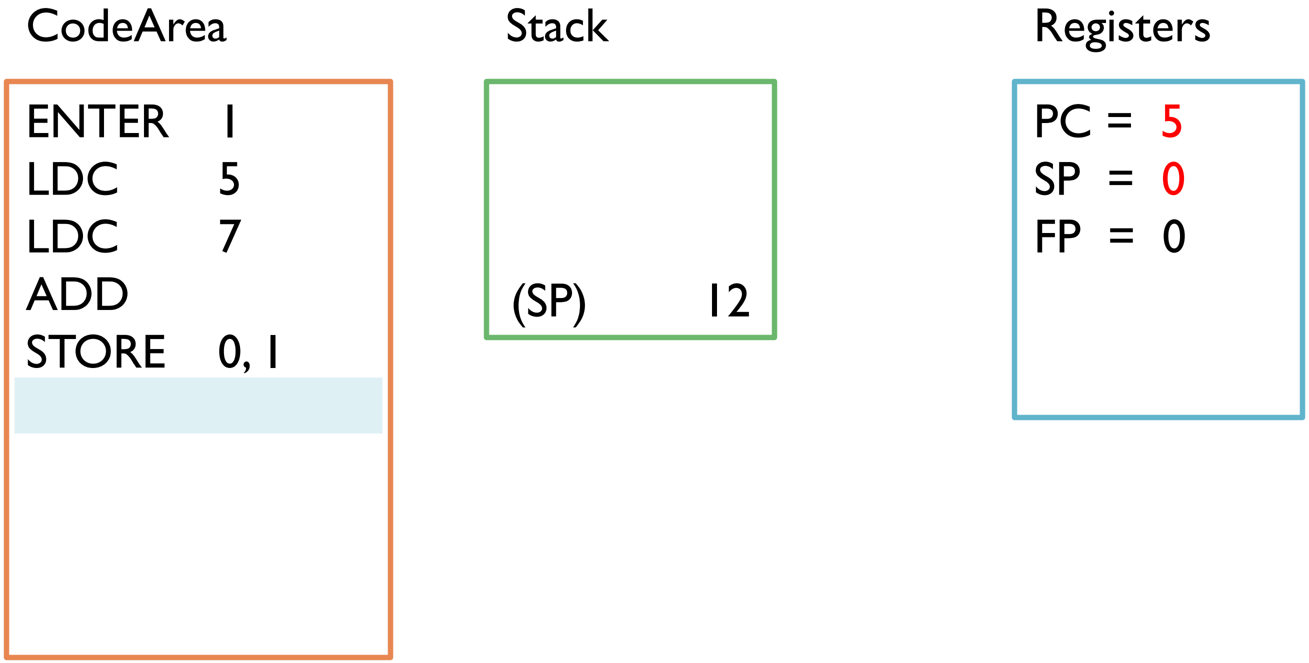
```
(SP)          12
               0
```

Registers

PC = 4
SP = 1
FP = 0

S-Machine Instructions

- A = 5 + 7



CodeArea

```
ENTER    1
LDC      5
LDC      7
ADD
STORE    0, 1
```

Stack

(SP) 12

Registers

```
PC =  5
SP  =  0
FP  =  0
```

S-Machine Instruction Set

- All Instructions are 32bits long (8 bytes)

- Format
  - Opcode (8 bits)
  - Scope Operand (8 bits)
  - Operand (16bits)

  - II ss nnnn

Instruction Set Examples

- 03000001
  - 03 / 00 / 0001
  - STORE 0, 1
- 01000a00
  - 01 / 00 / 000a
  - LDC 10
- 00000000
  - 00 / 00 / 0000
  - NOP

Note:- Operands are in **big-endian** form.

Instruction Set - Arithmetic

- Arithmetic Operations
  - ADD (+) (opcode 0x0F)
  - SUB (-) (opcode 0x10)
  - MUL (*) (opcode 0x11)
  - DIV (/) (opcode 0x12)
  - MOD (%) (opcode 0x13)

```
Stack(SP-1) = Stack(SP-1) Op Stack(SP);
SP = SP - 1;
PC = PC + 1;
```

Instruction Set - Relational

- Relational Operations
  - EQ (==) (opcode 0x0F)
  - NE (!=) (opcode 0x10)
  - LT (<) (opcode 0x11)
  - GT (>) (opcode 0x12)
  - LE (<=) (opcode 0x13)
  - GE (>=) (opcode 0x13)

```
Stack(SP-1) = Stack(SP-1) Op Stack(SP);
SP = SP - 1;
PC = PC + 1;
```

Instruction Set

- Operations
  - NOP (opcode 0x00)
    - No Operation
  - HALT (opcode 0x19)
    - Stop program execution
  - WRITE (opcode 0x1B)
    - Pop and Output the number on top of stack

Instruction Set – Stack

- Stack Operations
  - LDC n (opcode 0x01 00 nnnn)
    - Load/Push Constant **n** on Stack
  - DUP (opcode 0x04)
    - Duplicate the top of stack item
  - POP (opcode 0x05)
    - Removes the item from the top of stack

Instruction Set – Flow Control

- Flow Control Instructions
  - JMP n (opcode 0x06 00 nnnn)
    - Jump to location **n** in the code area
  - JZ n (opcode 0x07 00 nnnn)
    - If the top of stack item is 0 then jump to location **n** in the code area
  - JNZ n (opcode 0x08 00 nnnn)
    - If the top of stack item is NOT 0 then jump to location **n** in the code area

Instruction Set – Flow Control

- Flow Control Instructions
  - CALL n (opcode 0x0C 00 nnnn)
    - Push [PC + 1] (next instruction location) on stack
    - Jump to location **n** in the code area
  - RET (opcode 0x0D)
    - Pop PC from the top of stack and jump to that address in the code area
  - RETN n (opcode 0x0E 00 nnnn)
    - Pop PC from the top of stack and jump to that address in the code area
    - Pop **n** items from the stack

Instruction Set

- Remaining
  - ENTER n (opcode 0x0A 00 nnnn)
  - LEAVE (opcode 0x0B)

  - LD s, n (opcode 0x02 ss nnnn)
  - STORE s, n (opcode 0x03 ss nnnn)

  - READ s, n (opcode 0x1A ss nnnn)

Consider

```
var int x = 0;
x = x + 1;
write x;
```

Variables and Scope

```
var int x = 0;
x = x + 1;
write x;
```

```
ENTER       1

LDC         0
STORE       0, 1

LD          0, 1
LDC         1
ADD
STORE       0, 1

LD          0, 1
WRITE

LEAVE
```

Variables and Scope

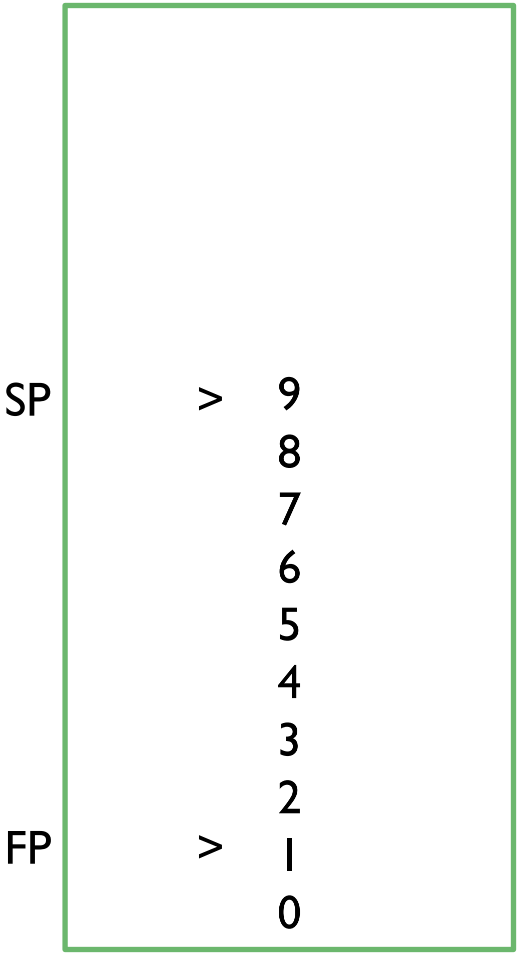
ENTER 1

```
SP = SP + 1;
Stack(SP) = FP;
FP = SP;
SP = SP + n;
PC = PC + 1;
```

Assumptions

SP = 10

FP = 1



SP > 9

8

7

6

5

4

3

2

FP > 1

0

Variables and Scope

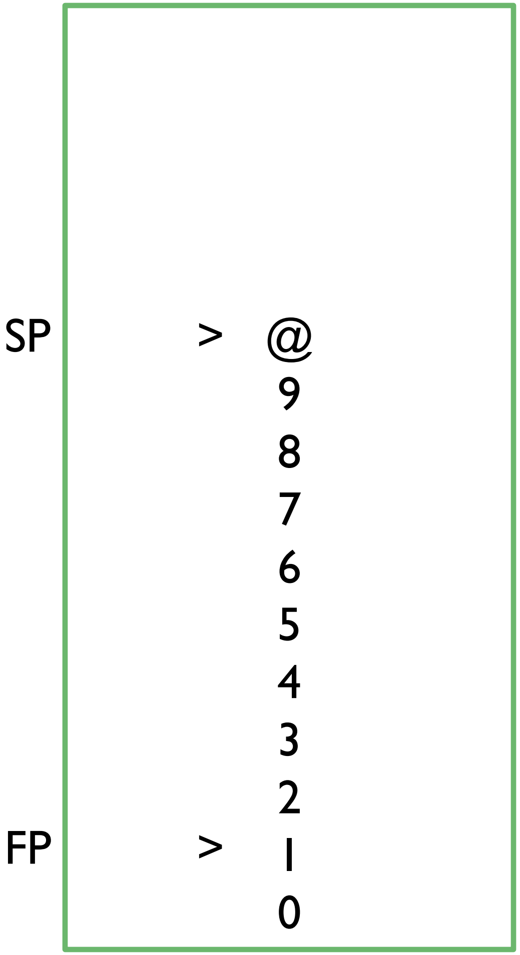
ENTER 1

```
SP = SP + 1;
Stack(SP) = FP;
FP = SP;
SP = SP + n;
PC = PC + 1;
```

Assumptions

SP = 10

FP = 1



SP > @

9

8

7

6

5

4

3

2

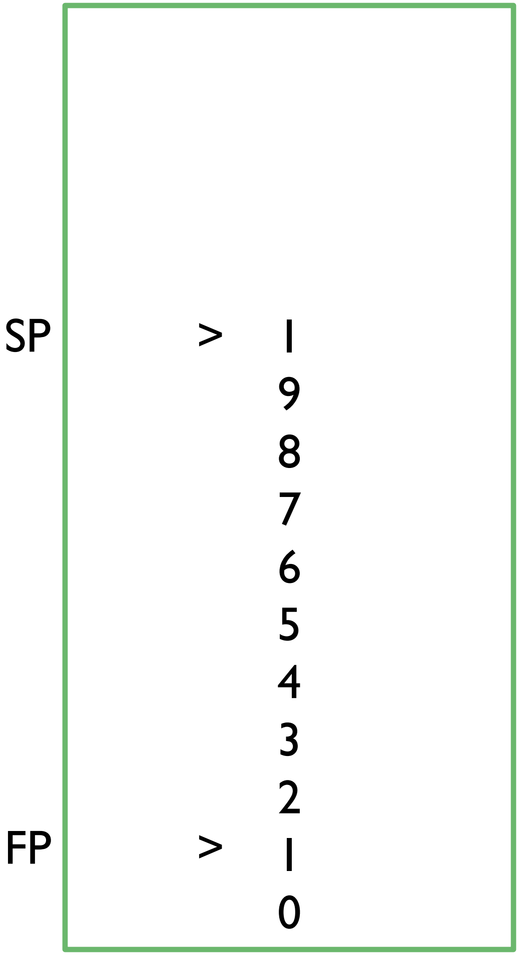
FP > 1

0

Variables and Scope

ENTER 1

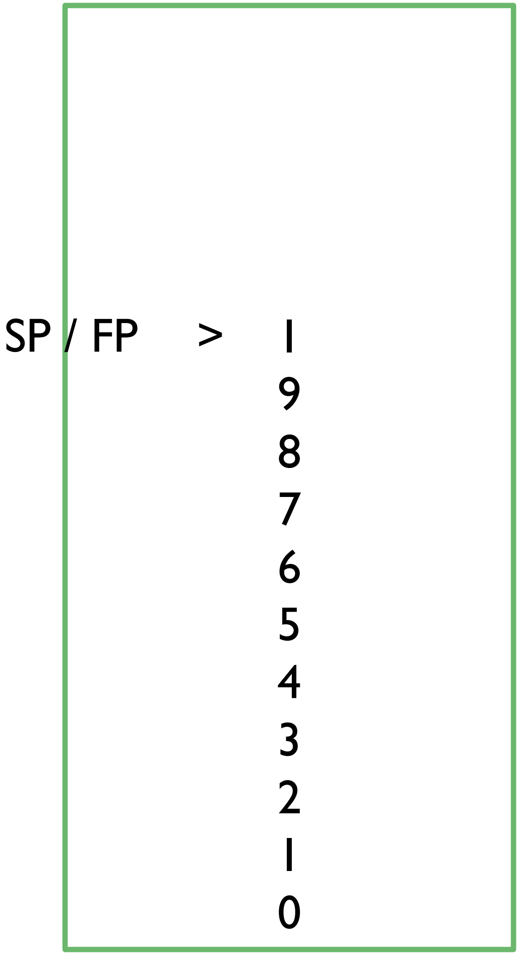
```
SP = SP + 1;
Stack(SP) = FP;
FP = SP;
SP = SP + n;
PC = PC + 1;
```



Variables and Scope

ENTER 1

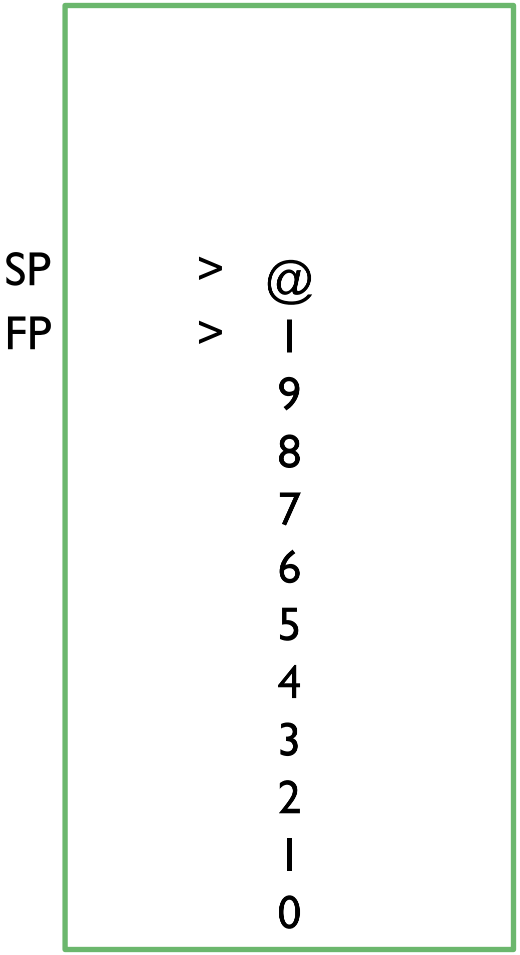
```
SP = SP + 1;
Stack(SP) = FP;
FP = SP;
SP = SP + n;
PC = PC + 1;
```



Variables and Scope

ENTER 1

```
SP = SP + 1;
Stack(SP) = FP;
FP = SP;
SP = SP + n;
PC = PC + 1;
```



SP > @

FP > 1

9

8

7

6

5

4

3

2

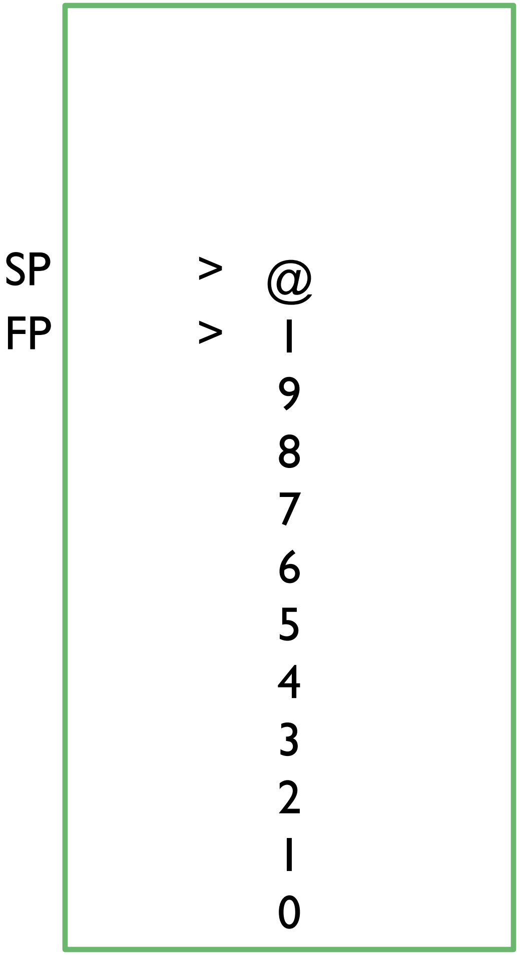
1

0

Variables and Scope

ENTER 1

```
SP = SP + 1;
Stack(SP) = FP;
FP = SP;
SP = SP + n;
PC = PC + 1;
```

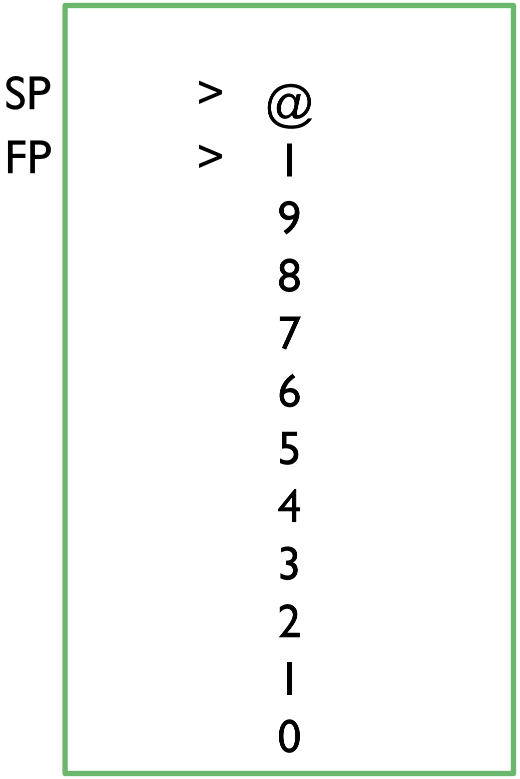


Variables and Scope

```
var int x = 0;
x = x + 1;
write x;
```

Location of x = Stack(FP + 1)

```
ENTER    1
-------------
LDC      0
STORE    0, 1
-------------
LD       0, 1
LDC      1
ADD
STORE    0, 1
-------------
LD       0, 1
WRITE
-------------
LEAVE
```

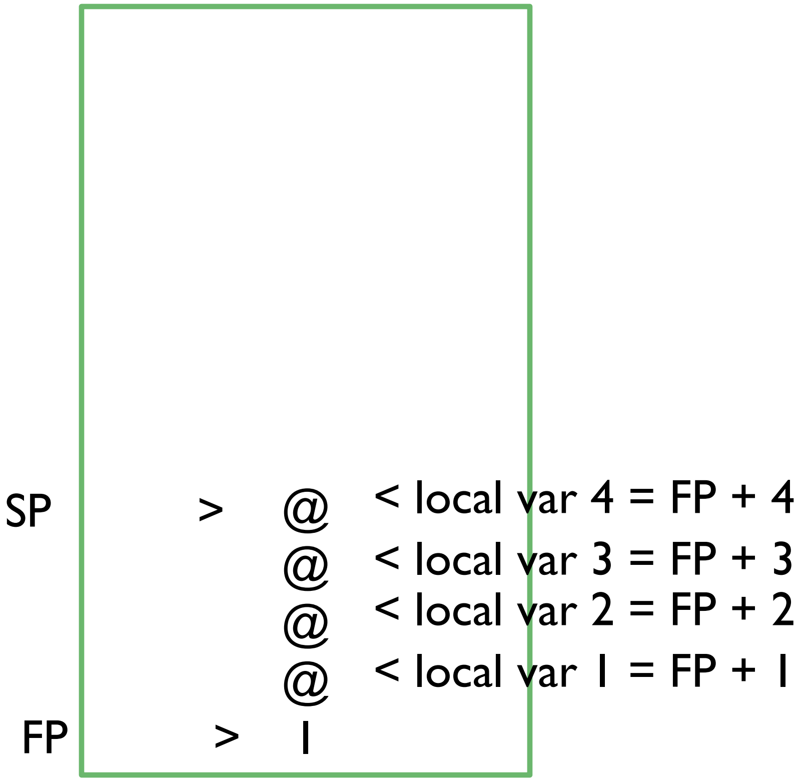


```
SP  >  @
FP  >  1
       9
       8
       7
       6
       5
       4
       3
       2
       1
       0
```

Variables and Scope

- Local Variables are stored as positive offsets from the FP register

- Only the ENTER instruction modifies the FP register



SP > @ < local var 4 = FP + 4

@ < local var 3 = FP + 3

@ < local var 2 = FP + 2

@ < local var 1 = FP + 1

FP > 1

Variables and Scope

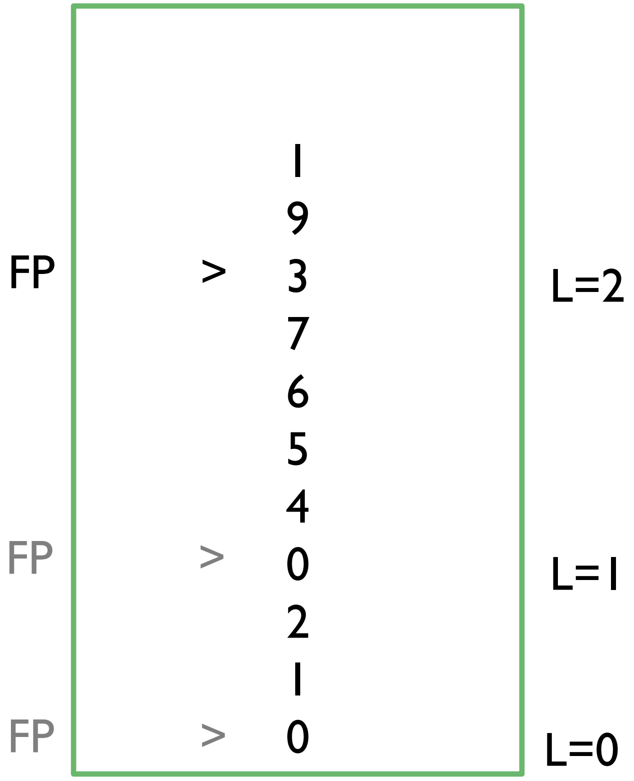
- What about the local scope operand of
  - LD s, n (opcode 0x02 ss nnnn)
  - STORE s, n (opcode 0x03 ss nnnn)
  - READ s, n (opcode 0x1A ss nnnn)

Variables and Scope

- ENTER instructions are executed FP changes but previous one is stored on the stack

- FP values are chained so the VM can backtrack to previous scopes

- Variables are referenced by the pair (scope, offset)



1

9

FP > 3 L=2

7

6

5

4

FP > 0 L=1

2

1

FP > 0 L=0

Variables and Scope

- What about the local scope operand of
  - LD s, n (opcode 0x02 ss nnnn)
    - Pushes the value of variable (s,n) on the stack
  - STORE s, n (opcode 0x03 ss nnnn)
    - Stores the value on top of stack in the variable (s,n)
  - READ s, n (opcode 0x1A ss nnnn)
    - Reads an integer from standard input and stores the value in the variable (s,n)

Functions

```
int Add(int a, int b)
{
        var int n = 0;
        n = a + b;
        return n;
}
```

Functions

- Definition

```
int Add( int a, int b )
{
    var int n = 0;
    n = a + b;
    return n;
}
```

- Call

```
x = Add( 5, 7 );
```

Functions

- Definition

```
int Add( int a, int b )
{
    var int n = 0;
    n = a + b;
    return n;
}
```

- Call

```
    x = Add( 5, 7 );
```

Assumptions

- Let us assume that the Add function has been compiled,
- hence the type checker is happy and code generation is ready
- We know the address of the Add function
- We know that the Add function has a return value, 2 parameters and 1 local variable

Functions

- Signature
  - int Add(int a, int b)
- Consider Call

  Add(5, 7);

- Prepare space for the return value
- Load/Push Parameters on the stack
- Perform the Function call

Instructions

```
LDC     0               // Prepare space

LDC     5               // LD param 1
LDC     7               // LD param 2

CALL    Add location    // Call the function
```

Functions

- Remember
  - CALL instruction pushes the PC on the stack
  - Functions have a new scope so the generated code starts with an ENTER instruction

Functions

- Definition

```
int Add( int a, int b )
{
    var int n = 0;
    n = a + b;
    return n;
}
```

- Call

```
    x = Add( 5, 7 );
```

Instructions

```
ENTER   1
LDC     0
…



LEAVE
RETN    2

--------------------------------

LDC     0
LDC     5
LDC     7

CALL    Add
...
```

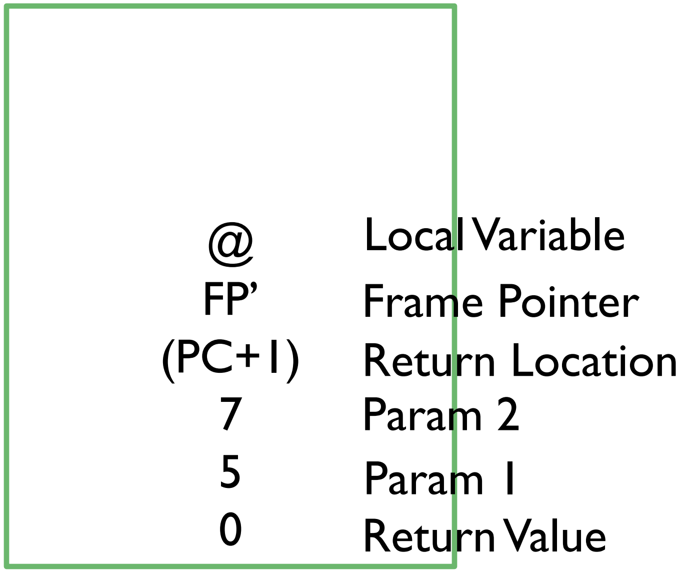
Functions

- Definition: int Add(int a, int b)
- Call: Add(5, 7);

Instructions

```
ENTER    1
LDC      0
…
LEAVE
RETN     2
-------------------------------
LDC      0
LDC      5
LDC      7

CALL     Add
…
```



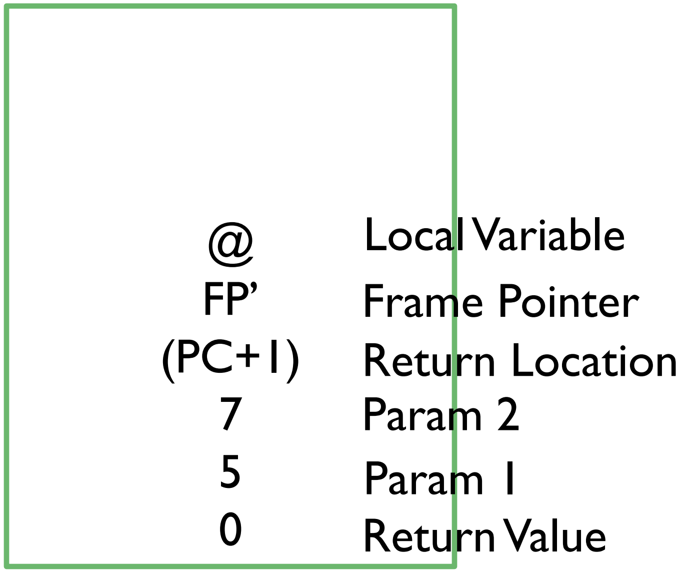
@	Local Variable
FP’	Frame Pointer
(PC+1)	Return Location
7	Param 2
5	Param 1
0	Return Value

Functions

- Accessing variables inside Functions
  - Local Variable → FP + 1

  - Param 1 → FP – 3
  - Param 2 → FP – 2
  - Return Value → FP – 4

- Do Not Access
  - FP – 1
  - FP



@	Local Variable
FP’	Frame Pointer
(PC+1)	Return Location
7	Param 2
5	Param 1
0	Return Value

Functions

- Definition

```
int Add( int a, int b )
{
    var int n = 0;
    n = a + b;
    return n;
}
```

- Call

```
    x = Add( 5, 7 );
```

Instructions

```
ENTER   1

LDC     0
STORE   0, 1

LD      0,  -3
LD      0,  -2
ADD
STORE   0, 1

LD      0, 1
STORE   0, -4
LEAVE
RETN    2

--------------------------------
LDC     0
LDC     5
LDC     7

CALL    Add
...
```

@	Local Variable
FP’	Frame Pointer
(PC+1)	Return Location
7	Param 2
5	Param 1
0	Return Value