COP 3402 Systems Software

Euripides Montagne
University of Central Florida

COP 3402 Systems Software

Virtual Machines as instruction interpreters

Outline

1. Virtual machines as software interpreters
2. P-code: instruction set architecture
3. The instruction format
4. Assembly language

Virtual Machine: P-code

The Pseudo-code (P-code) is a virtual machine.

A virtual machine is a software implementation of an instruction set architecture.

P-code was implemented in the 70s to generate intermediate code for Pascal compilers.

Another example of a virtual machine is the JVM (Java Virtual Machine) whose intermediate language is commonly referred to as Java bytecode.

The P-machine Instruction format (PM/0)

The ISA of the PM/0 has 24 instructions and the instruction format has three components [OP, L, M]:

OP is the operation code.

L indicates the lexicographical level.

M depending of the opcode it indicates:
- A number (instructions: LIT, INC).

- A program address (instructions: JMP, JPC, CAL).

- A data address (instructions: LOD, STO)

- The identity of the operator OPR(i.e. OPR 0, 2 (ADD) or OPR 0, 4 (MUL)).
2 means ADD, 4 means MUL.

Virtual Machine: P- code

The interpreter of the P-machine(PM/0) consists of a process address space(PAS), a memory area which consists of two segments:

text
stack
PAS

A **“text”** segment

A segment called the **“stack”**.

“text” segment that contains the instructions.

The CPU has four registers:

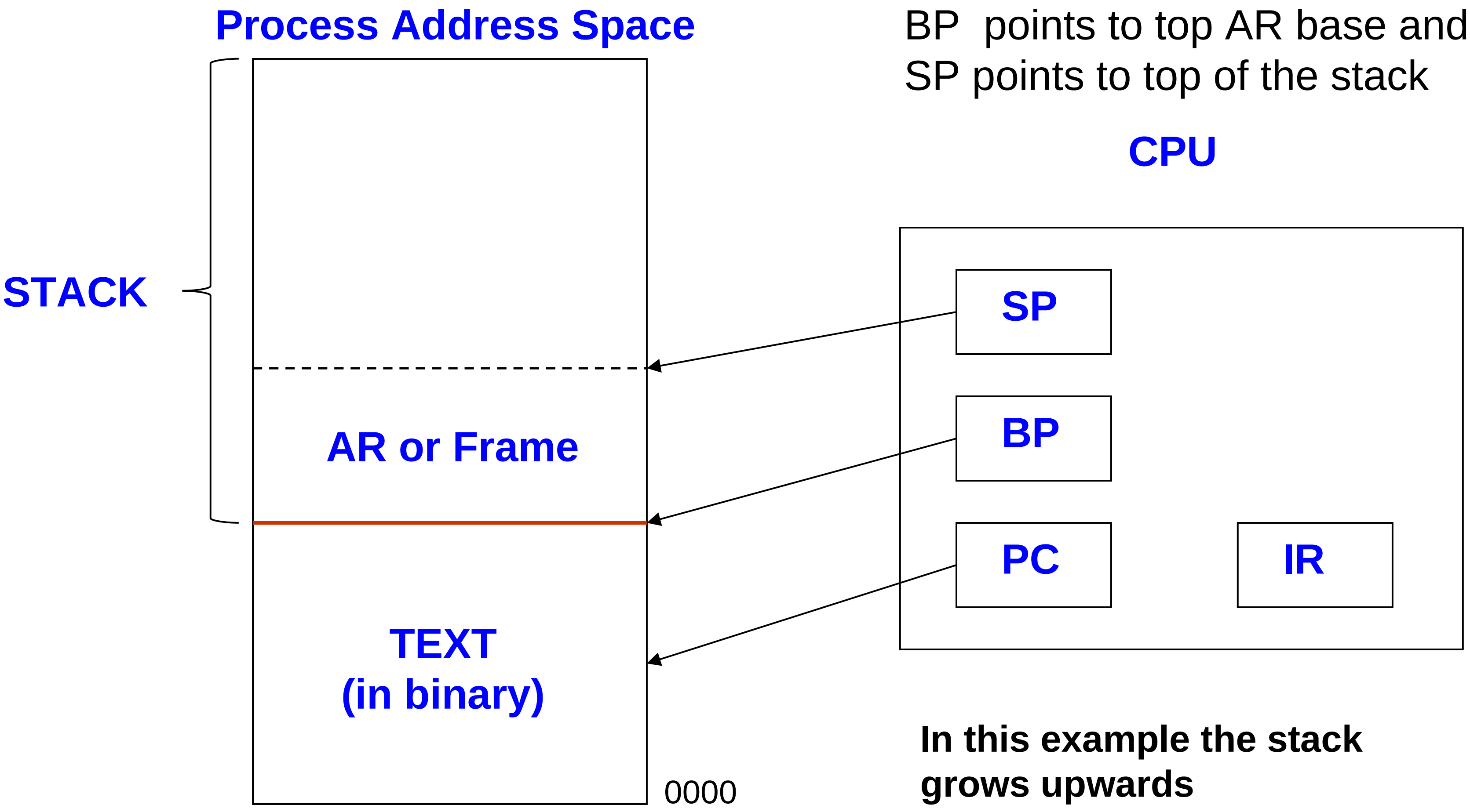
-Base pointer **(bp)** which points to the base of the current **<u>activation record (AR)</u>** in the stack. (AR is also known as stack frame)

-Stack pointer **(sp)** which points to the top of the stack

- Program counter or instruction pointer (**pc**), which points to the next instruction to be executed

- Instruction register (**ir**). Where instruction fetched from memory are placed into.

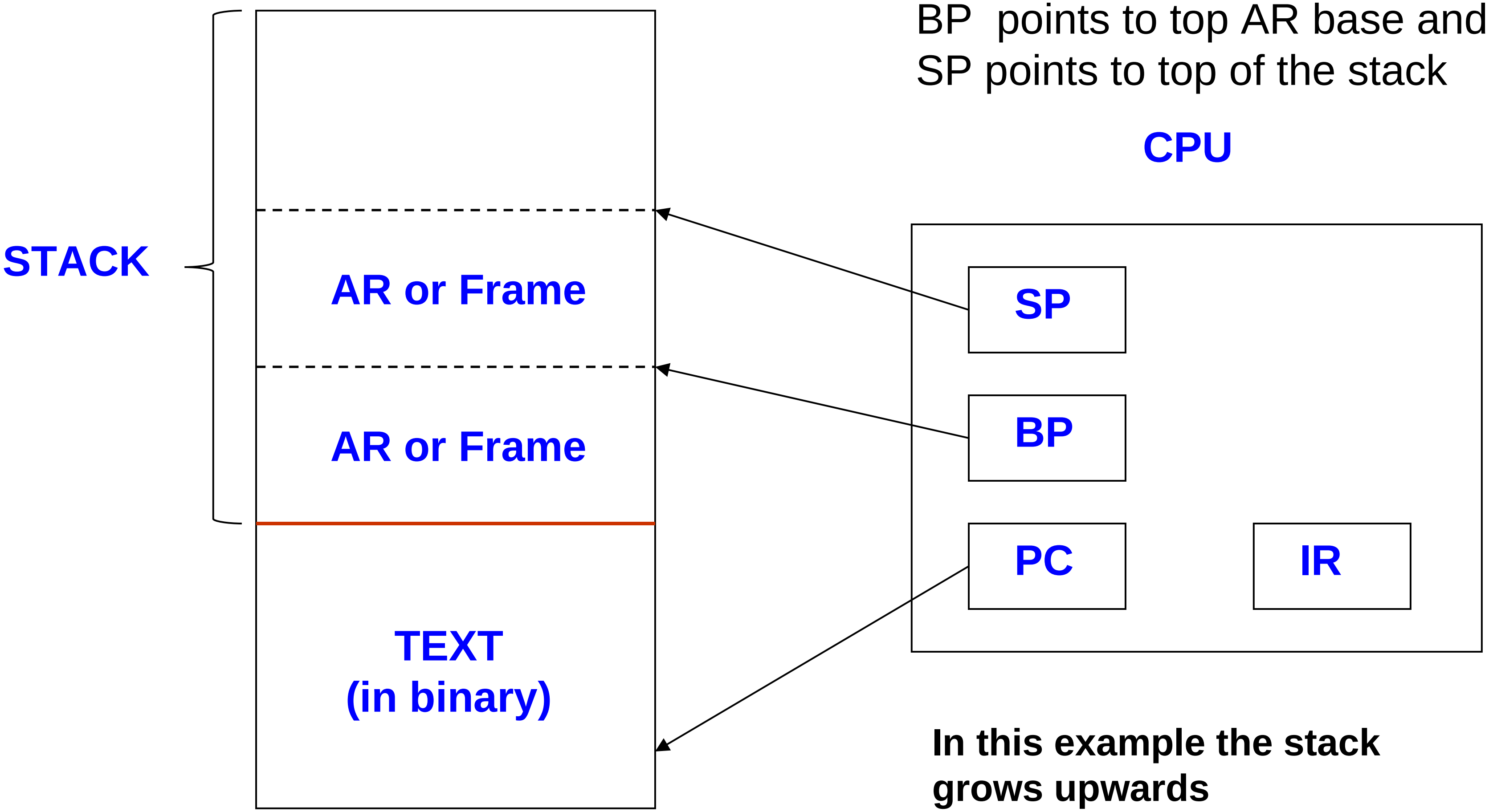
Virtual Machine: P- code



BP points to top AR base and
SP points to top of the stack

In this example the stack grows upwards

When a function is called a new AR is created for that function, and BP and SP are updated to point to the newly created AR.



BP points to top AR base and
SP points to top of the stack

In this example the stack grows upwards

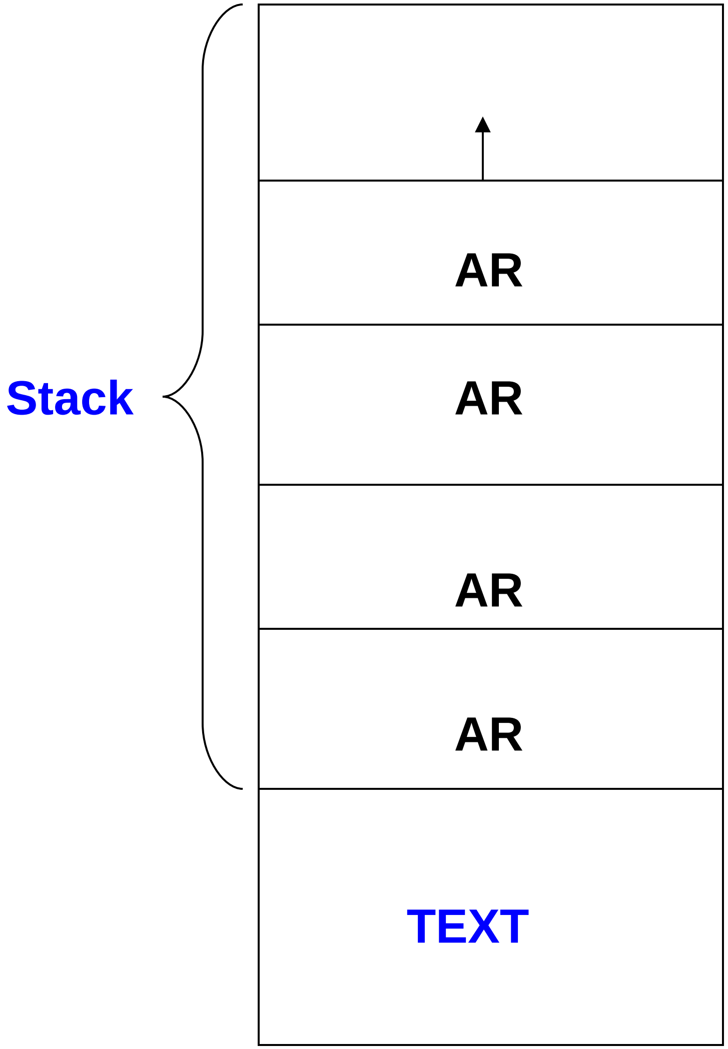
Activation Records (AR)

What is an activation record?

Activation record or stack frame is the name given to a data structure which is inserted in the stack, each time a procedure or function is called.

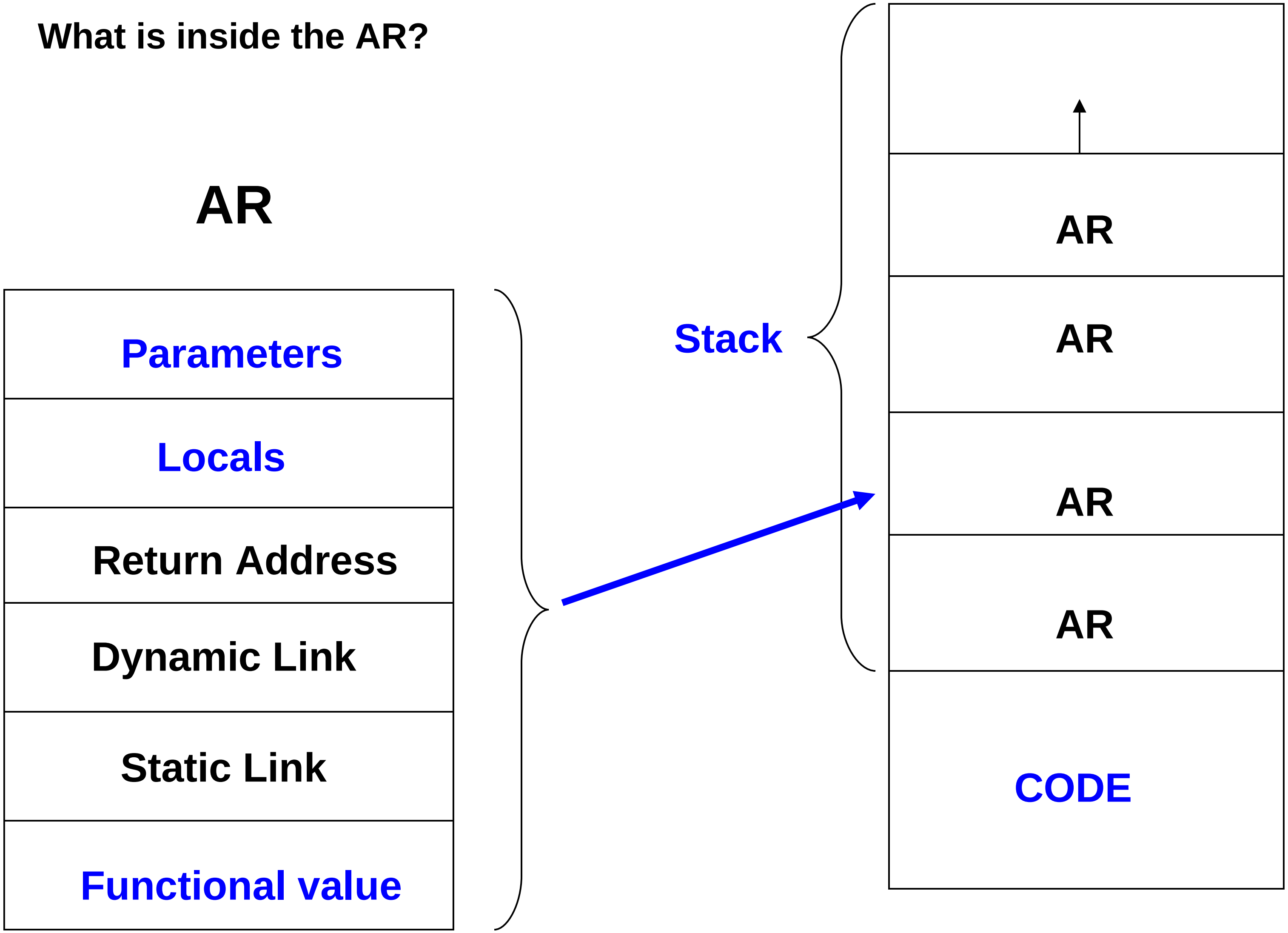
The data structure contains information to control sub-routine linkage when a program is in execution.

In the AR there is as well reserve space for local variables and parameters.



Activation records (AR)

What is inside the AR?



AR

Parameters

Locals

Return Address

Dynamic Link

Static Link

Functional value

Stack

AR

AR

AR

AR

CODE

Activation records (AR)

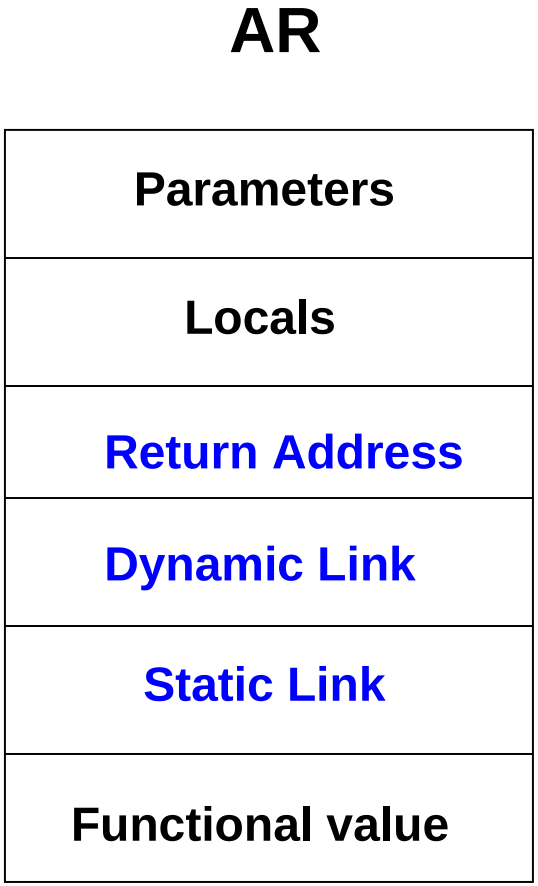
Control Information:

Return Address (RA): Points, in the text segment, to the next instruction to be executed in the calling function after termination of the current function or procedure.

Dynamic Link (DL): Points to the calling function stack frame base

Static Link (SL): Points to the stack frame of the procedure that statically encloses the current function or procedure. (More details will be presented shortly)

DL and **RA** restore the callers environment



AR

Parameters

Locals

Return Address

Dynamic Link

Static Link

Functional value

Activation records (AR)

Functional value: Location to store the function returned value.

Parameters: Space reserved to store the actual parameters of the function.

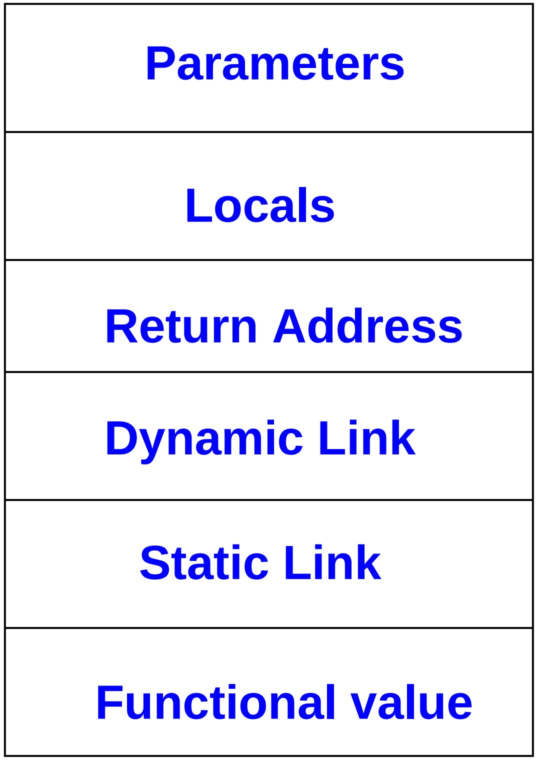
Locals: Space reserved to store local variables declared within the function or procedure.

Return Address: Points, in the code segment, to the next instruction (in the calling function) to be executed after termination of the current function or procedure.

Dynamic Link: Points to the base of the previous stack frame (AR = stack frame)

Static Link: Points to the stack frame of the the procedure that statically encloses the current function or procedure

AR



Parameters

Locals

Return Address

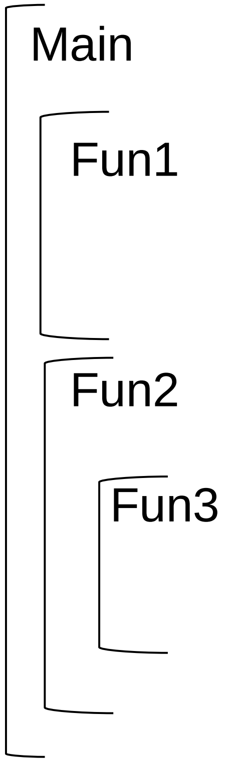
Dynamic Link

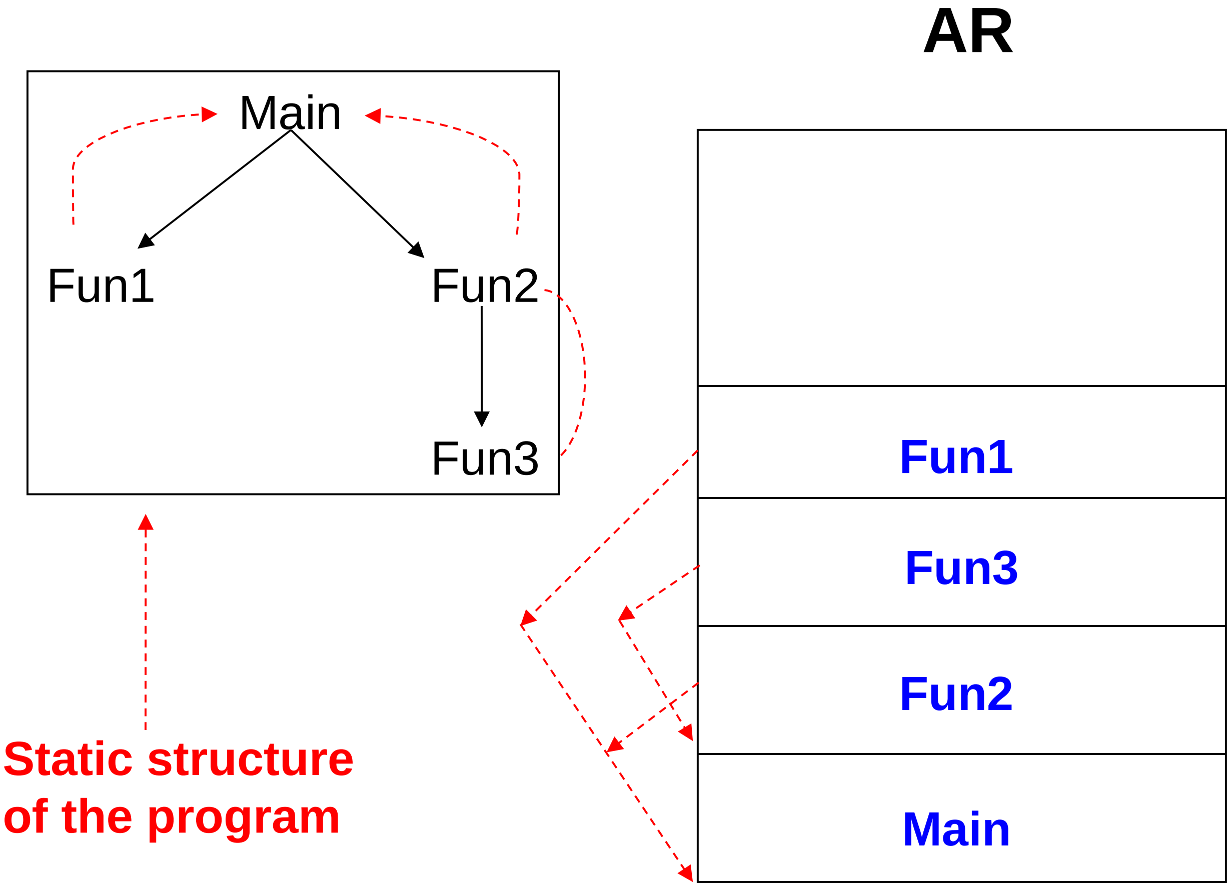
Static Link

Functional value

Activation records (AR)

Static Link: Points to the stack frame of the procedure that statically encloses the current function or procedure:





Static structure of the program

Back to the P-machine!!
Instruction cycle

The machine instruction cycle has two step known as fetch and execute.

Fetch step:
In the fetch step an instruction is fetch from the text segment (ir ← text[pc]) and the program counter is incremented by one (pc ← pc + ?).

FETCH

ir ← text[pc]
pc ← pc + ?   ? = 1 or 2 or 3 or 4

Execute step:
In this step ir.op indicates the operation to be executed. In case ir.op = OPR then the field ir.m is used to identified the operator and execute the appropriate arithmetic or logical instruction

opcode

P-machine ISA

01– **LIT** **0, M** → Push constant value (literal) **M** onto stack

02 – **OPR** **(to be defined later, after the examples)**

03 – **LOD** **L, M** → Push from location at offset **M** in stack frame **L** levels down.

04 – **STO** **L, M** → Store value on top of stack at offset **M** in stack frame
L levels down.

05 – **CAL** **L, M** → Call procedure at M (generates new AR and pc = **M**).

06 – **INC** **0, M** → Increment sp by M

07 – **JMP** **0, M** → pc = **M**

08 – **JPC** **0, M** → Jump to M if top of stack element is 0

09 – **SYS** **0, 1** → Write the top stack element to the screen
SYS **0, 2** → Read in input from the user and store it on top of the stack
SYS **0, 3** → End of program (**Set Halt flag to zero**)

P-machine ISA

We will show, with examples, how some instructions work

01 – **LIT 0, M** → Push constant value (literal) **M** onto stack

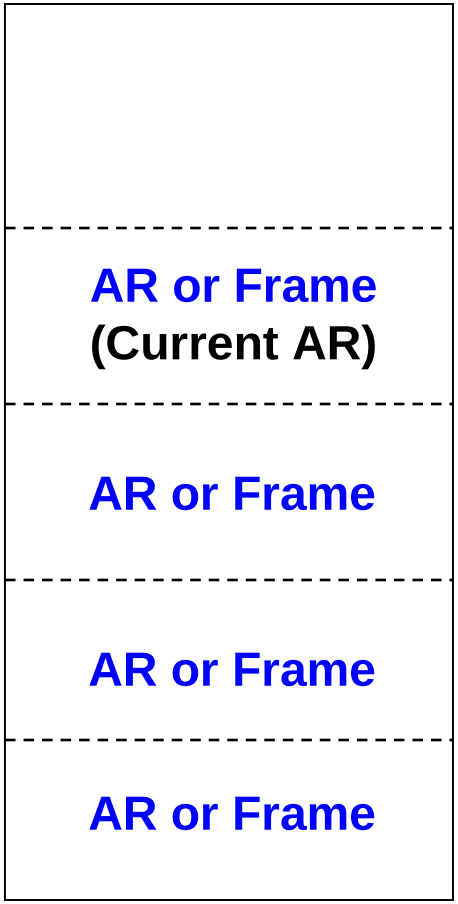
03 – **LOD L, M** → Push from location at offset **M** in stack frame **L** levels down.

04 – **STO L, M** → Store top of stack at offset **M** in stack frame **L** levels down.

Pay attention to theses details: L = 0 in the examples for LOD and STO because a value will be loaded or stored in the current activation record (the active one).

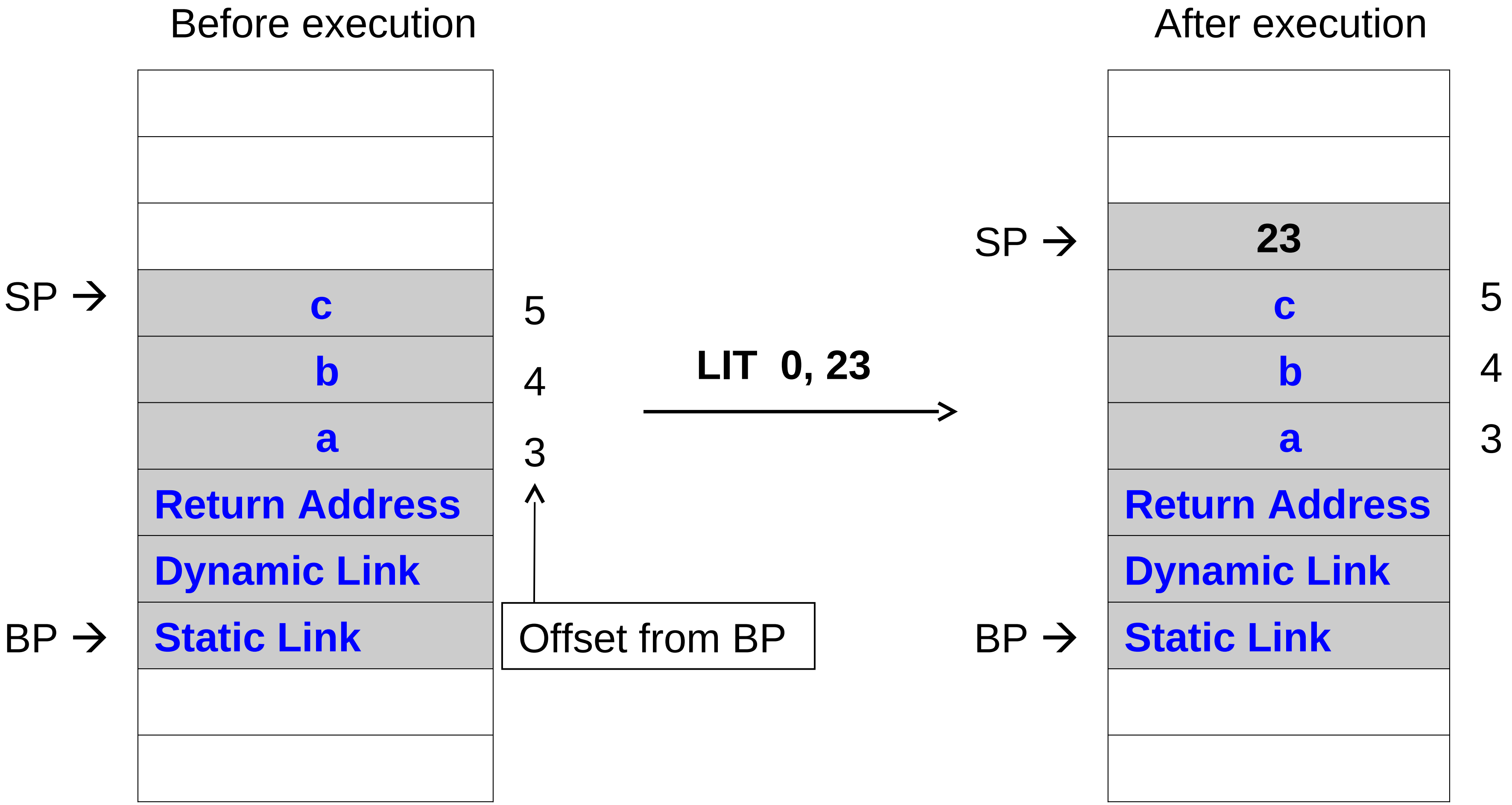
We need to clarify this because eventually several ARs might be coexisting in the stack and the AR on top of the stack is current or active AR.

L != 0 means that a value has to be LOD or STO in a different AR (when L != 0, it is a **Natural** number)


AR or Frame
(Current AR)
AR or Frame
AR or Frame
AR or Frame

01– **LIT 0, M** → Push constant value (literal) **M** onto stack

LIT 0, M steps: sp ← sp + 1; stack[sp] ← M;



Before execution

SP →
c 5
b 4
a 3
Return Address
Dynamic Link
BP →
Static Link

Offset from BP

LIT 0, 23

After execution

SP →
23
c 5
b 4
a 3
Return Address
Dynamic Link
BP →
Static Link

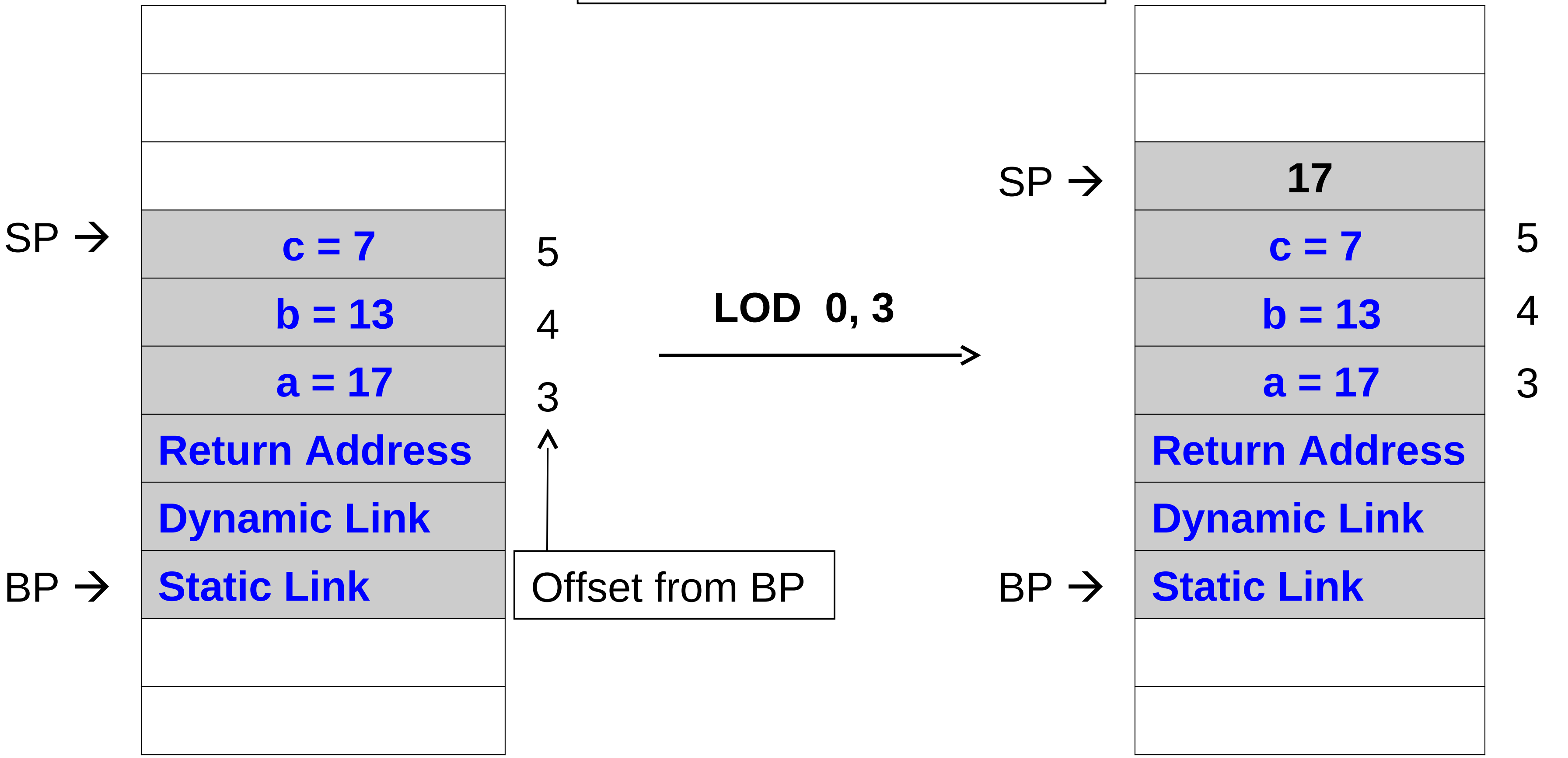
03– **LOD 0, M** → Push from location at offset **M** in current AR

LOD 0, M: sp ← sp +1; stack[sp] ← stack[base(**L**) + **M**];

Before execution

As L = 0 then base(L) = 0

After execution



SP →

c = 7

b = 13

a = 17

Return Address

Dynamic Link

BP →

Static Link

5

4

3

Offset from BP

LOD 0, 3

SP →

17

c = 7

b = 13

a = 17

Return Address

Dynamic Link

BP →

Static Link

5

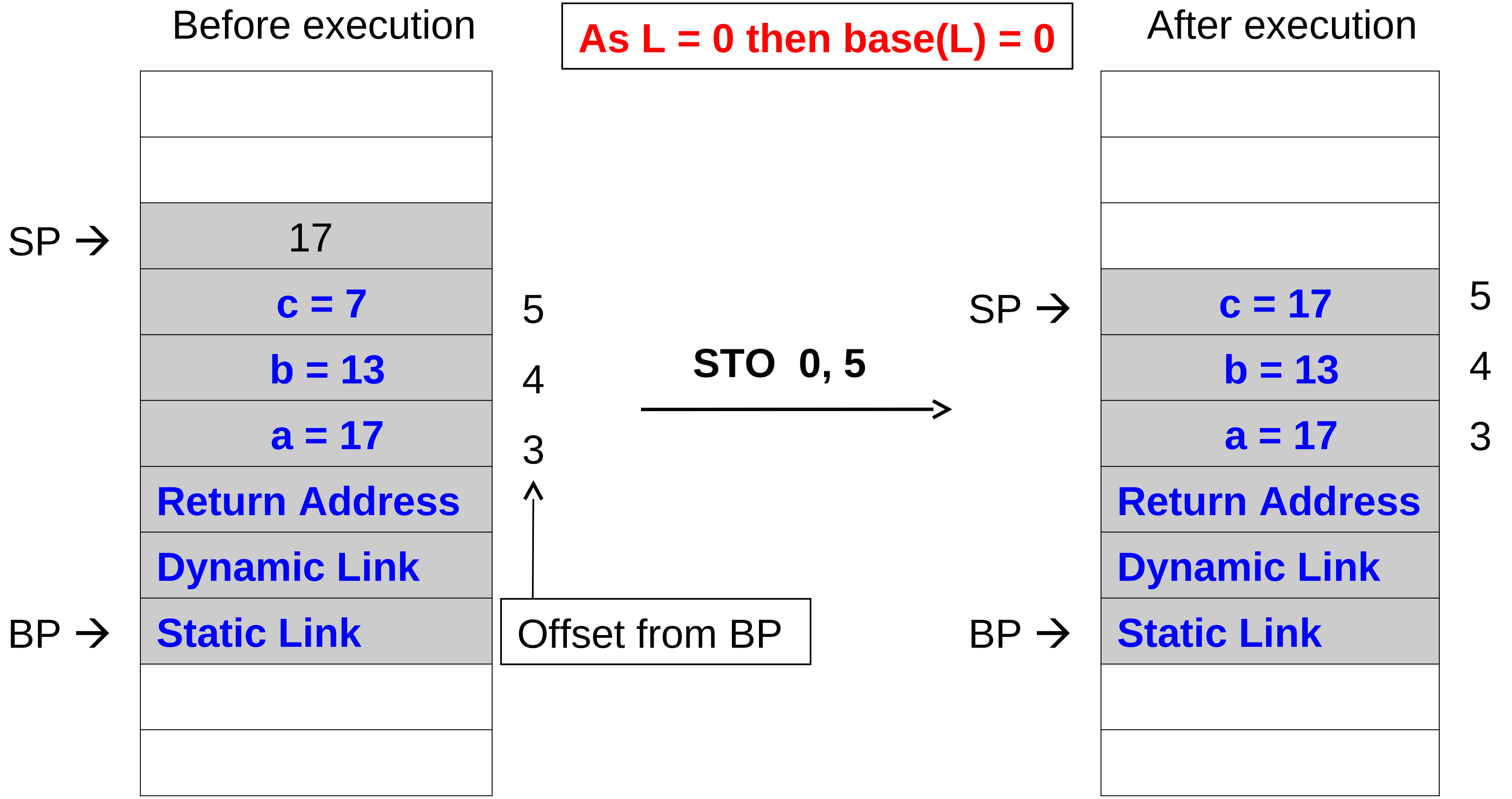
4

3

“base(L) is a function which locates the base of an ARs”

03– **STO 0, M** → Store top of stack at offset **M** in current AR.

STO 0, M: stack[base(**L**) + **M]** ← stack[sp]; sp ← sp - 1;



Before execution

As L = 0 then base(L) = 0

After execution

	Before execution	Offset			After execution	Offset
SP →	17					
	c = 7	5		SP →	**c = 17**	5
	b = 13	4	**STO 0, 5** →		**b = 13**	4
	a = 17	3			**a = 17**	3
	Return Address				**Return Address**	
	Dynamic Link				**Dynamic Link**	
BP →	**Static Link**	Offset from BP		BP →	**Static Link**	

"base(L) is a function which locates the base of an ARs"

P-machine ISA

opcode

↓

02 - OPR:

RTN 0,0 → Return operation (i.e. return from subroutine)

OPR 0,1 → NEG (- stack[sp])

OPR 0,2 → ADD (sp← sp – 1 and stack[sp] ← stack[sp] + stack[sp + 1])

OPR 0,3 → SUB (sp← sp – 1 and stack[sp] ← stack[sp] - stack[sp + 1])

OPR 0,4 → MUL (sp← sp – 1 and stack[sp] ← stack[sp] * stack[sp + 1])

OPR 0,5 → DIV (sp← sp – 1 and stack[sp] ← stack[sp] div stack[sp + 1])

OPR 0,6 → ODD (stack[sp] ← stack mod 2) or ord(odd(stack[sp]))

OPR 0,7 → MOD (sp← sp – 1 and stack[sp] ← stack[sp] mod stack[sp + 1])

OPR 0,8 → EQL (sp← sp – 1 and stack[sp] ← stack[sp] = =stack[sp + 1])

OPR 0,9 → NEQ (sp← sp – 1 and stack[sp] ← stack[sp] != stack[sp + 1])

OPR 0,10 → LSS (sp← sp – 1 and stack[sp] ← stack[sp] < stack[sp + 1])

OPR 0,11 → LEQ (sp← sp – 1 and stack[sp] ← stack[sp] <= stack[sp + 1])

OPR 0,12 → GTR (sp← sp – 1 and stack[sp] ← stack[sp] > stack[sp + 1])

OPR 0,13 → GEQ (sp← sp – 1 and stack[sp] ← stack[sp] >= stack[sp + 1])

P-machine ISA

opcode

↓

```
01 -  LIT    0, M →  sp ← sp +1;
                     stack[sp] ← M;

02 – RTN   0, 0 →   sp ← bp -1;
                    cpu.pc ← stack[sp + 3];   // cpu.pc = RA
                    cpu.bp ← stack[sp + 2];   // cpu.bp = DL

03 – LOD   L, M → sp ← sp +1;
                  stack[sp] ← stack[ base(L) + M];

04 – STO   L, M →  stack[ base(L) + M] ← stack[sp];
                    sp ← sp -1;
```

“base(L) is a function which locates the base of an ARs”

opcode
↓

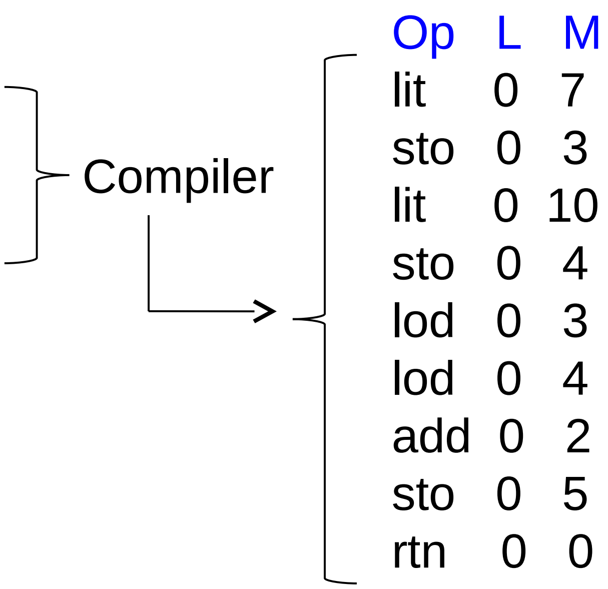
P-machine ISA

```
05 - CAL   L, M → stack[sp + 1]  ←  base(L);          /* static link (SL)
                  stack[sp + 2]  ← cpu.bp;            /*  dynamic link (DL)
                  stack[sp + 3]  ← cpu.pc             /*  return address (RA)
                  bp ← sp + 1;
                  pc ← M;

06 – INC   0, M → sp ← sp + M;


07 – JMP  0, M →  pc = M;


08 – JPC  0, M →  if stack[sp] == 0 then  pc ← M;
                 sp ← sp - 1;


09 – WRT 0, 0  →  print (stack[sp]);
                  sp ← sp – 1;
```

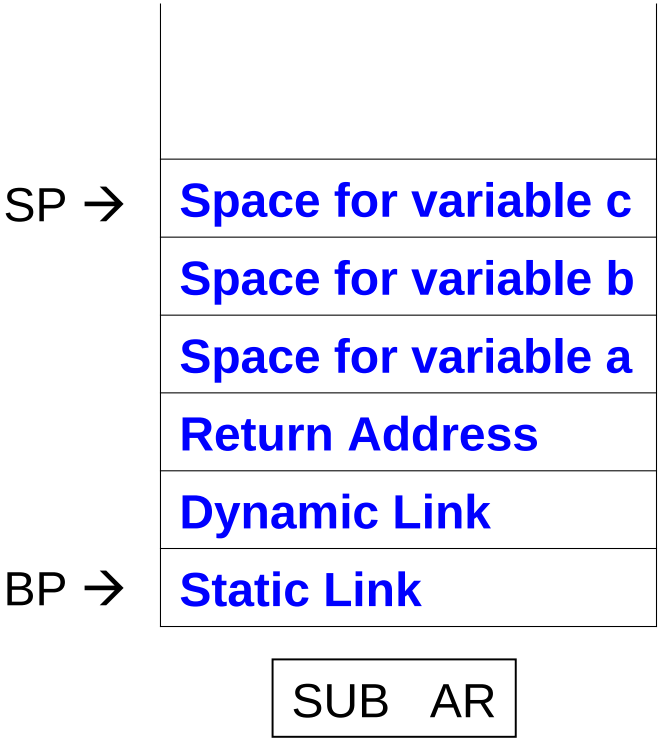
Associated to each function there is an Activation Record

Let us assume we have a function called SUB, and we are executing the instruction inside the function

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```



Op	L	M
lit	0	7
sto	0	3
lit	0	10
sto	0	4
lod	0	3
lod	0	4
add	0	2
sto	0	5
rtn	0	0



SP → Space for variable c

Space for variable b

Space for variable a

Return Address

Dynamic Link

BP → Static Link

SUB AR

Associated to each function there is an Activation Record

Literal 7 loaded on top of stack.

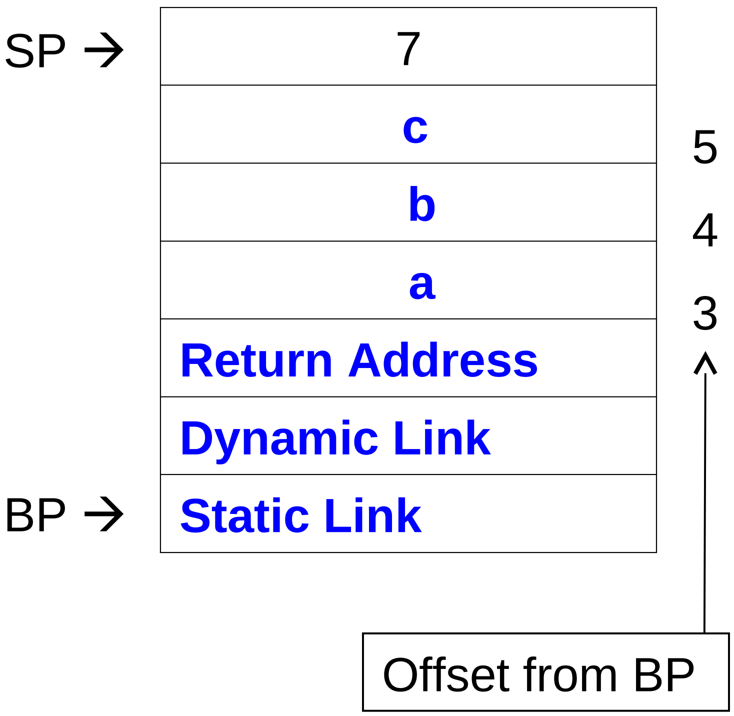
```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let’s run the program

```
Op   L   M
lit  0   7
sto  0   3   ← PC
lit  0  10
sto  0   4
lod  0   3
lod  0   4
add  0   2
sto  0   5
rtn  0   0
```

IR = LIT 0, M → sp ← sp +1;
stack[sp] ← **M;**

Stack after fetch-execute



SP →

7

c 5

b 4

a 3

Return Address

Dynamic Link

BP →

Static Link

Offset from BP

Associated to each function there is an Activation Record

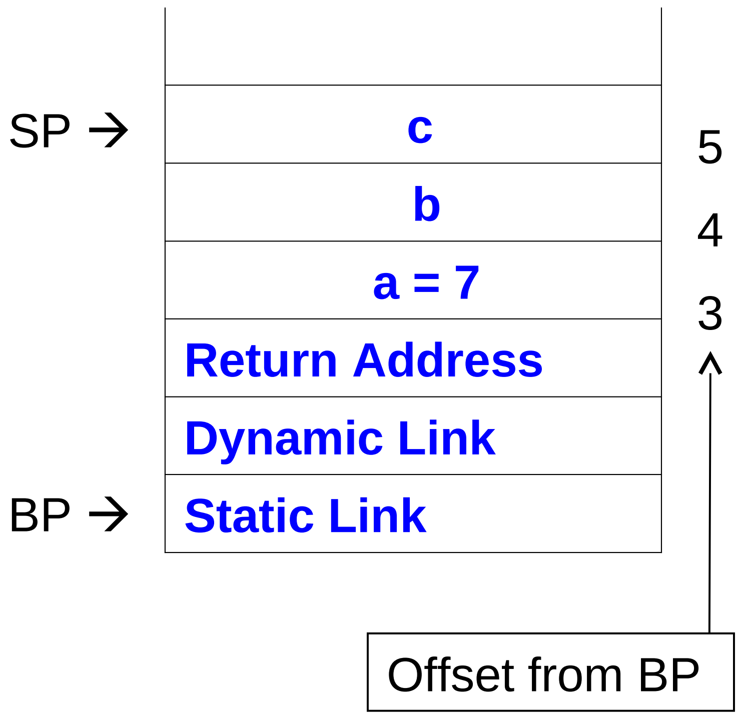
Value on top of stack (7) store in variable a.

Stack after fetch-execute ↓

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let’s run the program

```
Op   L   M
lit  0   7
sto  0   3
lit  0  10   ← PC
sto  0   4
lod  0   3
lod  0   4
add  0   2
sto  0   5
rtn  0   0
```



IR = STO L, M → stack[BP + **M]** ← stack[sp];
sp ← sp -1;

Associated to each function there is an Activation Record

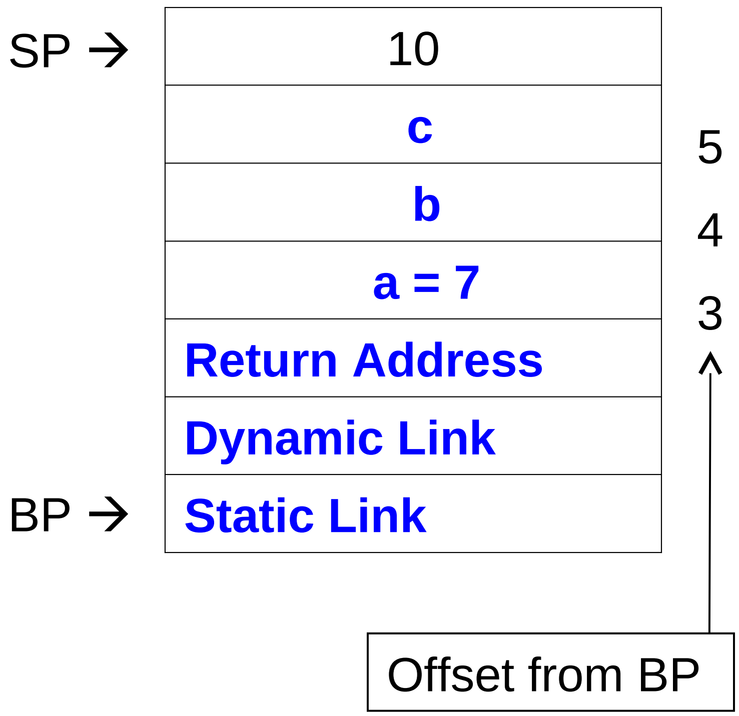
Literal 10 loaded on top of stack.

Stack after fetch-execute

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let's run the program

Op	L	M	
lit	0	7	
sto	0	3	
lit	**0**	**10**	
sto	0	4	← PC
lod	0	3	
lod	0	4	
add	0	2	
sto	0	5	
rtn	0	0	



SP → 10

c 5

b 4

a = 7 3

Return Address

Dynamic Link

BP → Static Link

Offset from BP

IR = LIT 0, M → sp ← sp +1;
stack[sp] ← **M;**

Associated to each function there is an Activation Record

Value on top of stack (10) store in variable b.

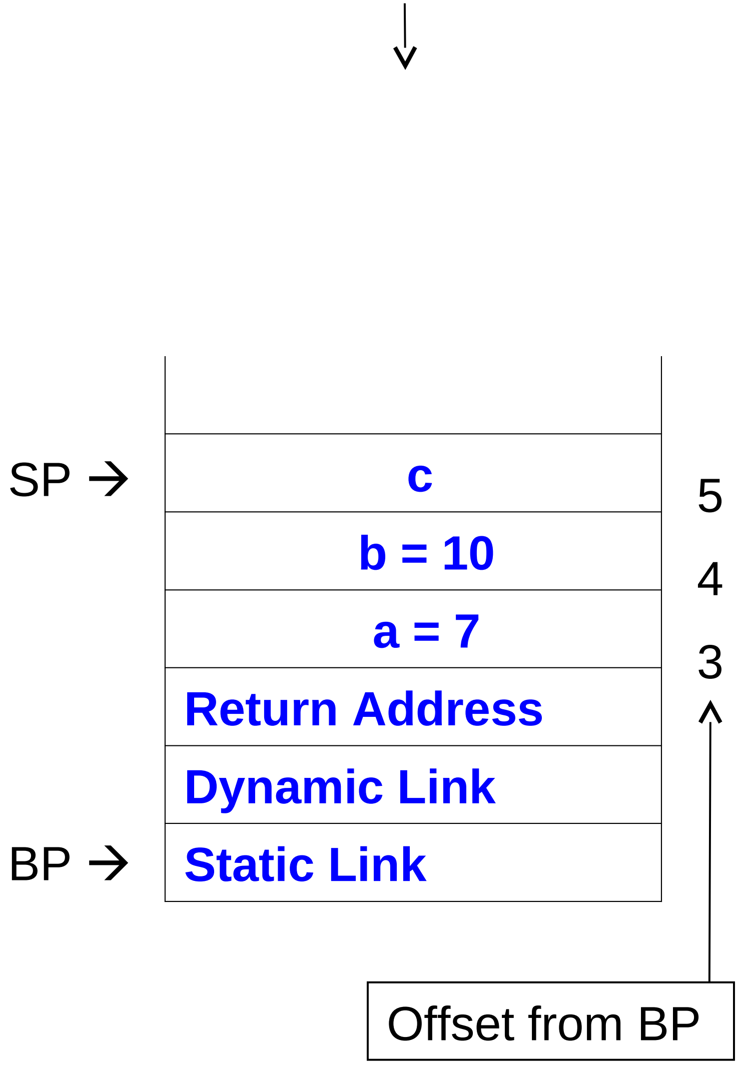
```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let’s run the program

```
Op    L   M
lit   0   7
sto   0   3
lit   0  10
sto   0   4
lod   0   3   ← PC
lod   0   4
add   0   2
sto   0   5
rtn   0   0
```

IR = STO L, M → stack[BP + **M]** ← stack[sp];
sp ← sp -1;

Stack after fetch-execute



Associated to each function there is an Activation Record

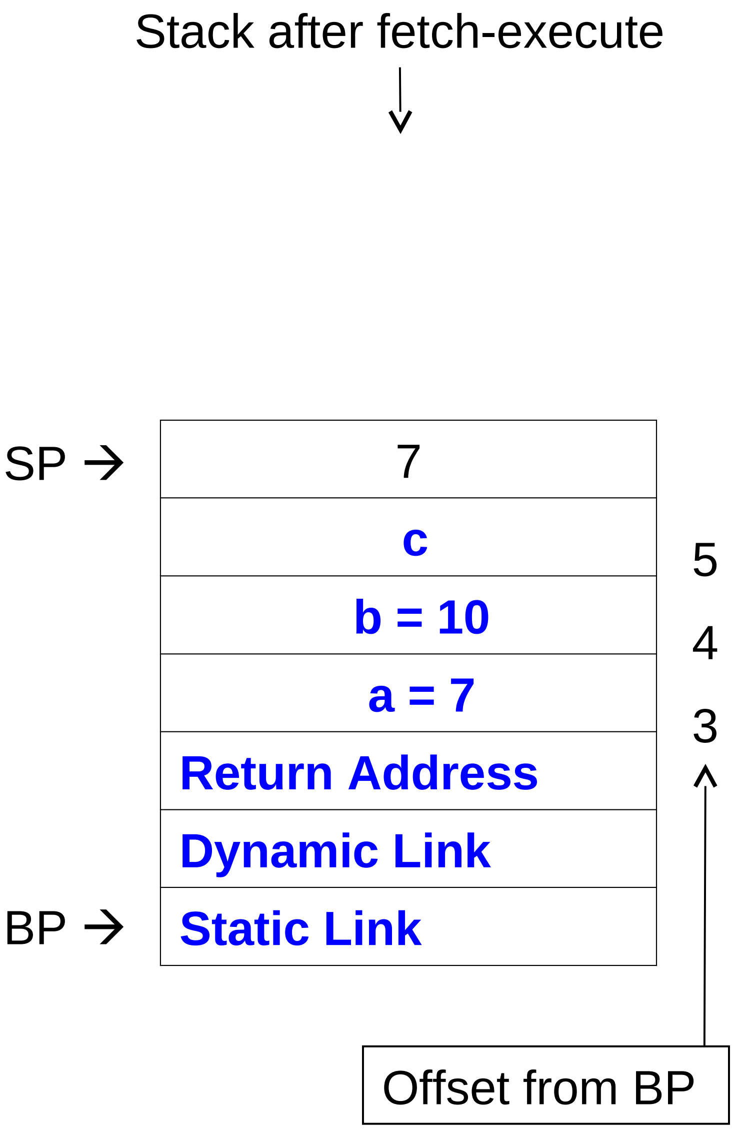
Variable a value (7) load on top of the stack.

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let’s run the program

Op	L	M	
lit	0	7	
sto	0	3	
lit	0	10	
sto	0	4	
lod	**0**	**3**	
lod	0	4	← PC
add	0	2	
sto	0	5	
rtn	0	0	

IR = LOD L, M → sp ← sp +1;
stack[sp] ← stack[BP + **M**];



Stack after fetch-execute

SP →

7
c
b = 10
a = 7
Return Address
Dynamic Link
Static Link

BP →

Offset from BP

Associated to each function there is an Activation Record

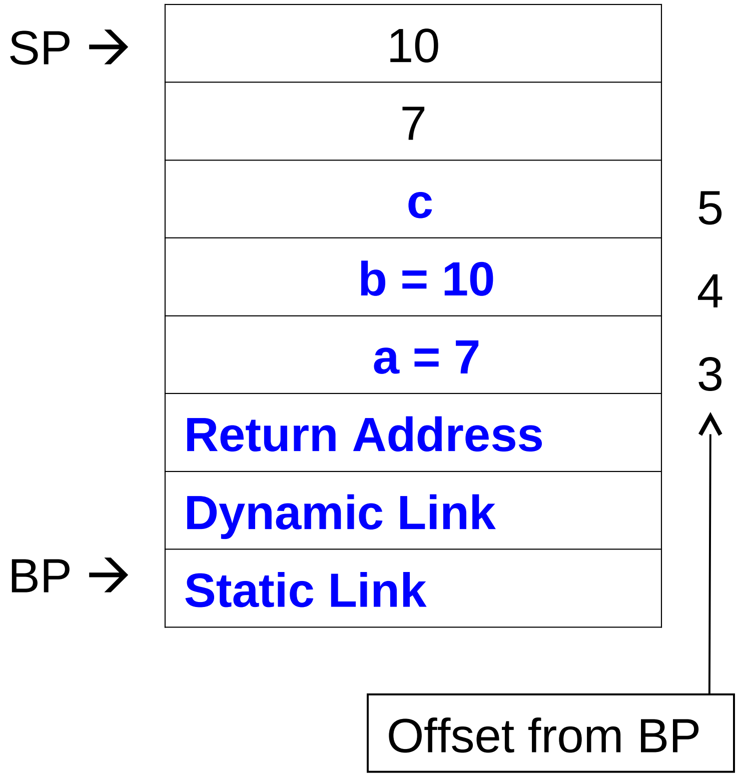
Variable b value (10) load on top of the stack.

Stack after fetch-execute

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let’s run the program

```
Op   L   M
lit  0   7
sto  0   3
lit  0  10
sto  0   4
lod  0   3
lod  0   4
add  0   2    ← PC
sto  0   5
rtn  0   0
```



SP →

10

7

c — 5

b = 10 — 4

a = 7 — 3

Return Address

Dynamic Link

BP →

Static Link

Offset from BP

IR = LOD L, M → sp ← sp +1;
stack[sp] ← stack[BP + **M**];

Associated to each function there is an Activation Record

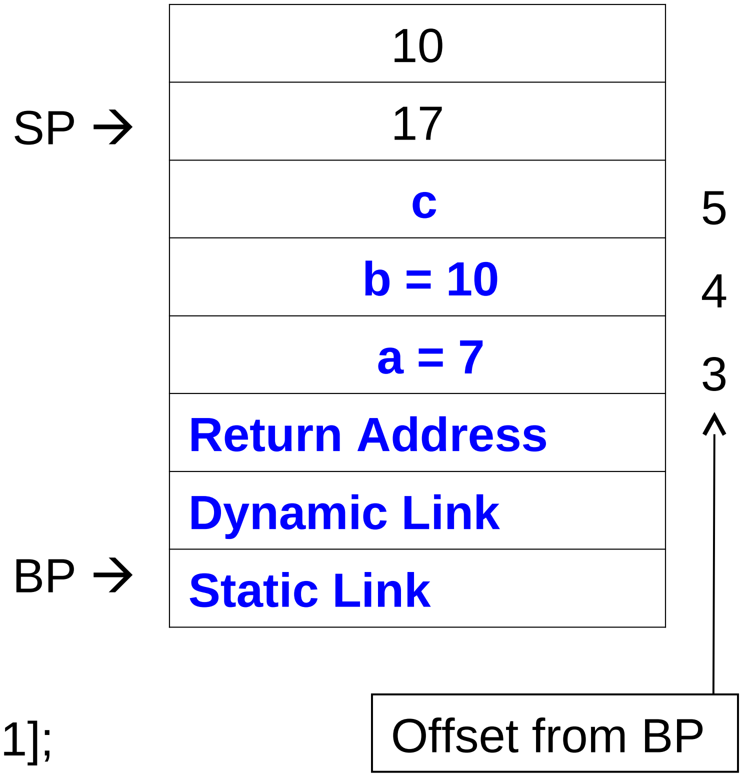
top of stack values (a and b) are added up

Stack after fetch-execute

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let's run the program

```
Op   L   M
lit  0   7
sto  0   3
lit  0  10
sto  0   4
lod  0   3
lod  0   4
add  0   2
sto  0   5   ← PC
rtn  0   0
```



SP →

BP →

Stack	Offset
10	
17	
c	5
b = 10	4
a = 7	3
Return Address	
Dynamic Link	
Static Link	

Offset from BP

IR = ADD L, M → sp ← sp -1;
stack[sp] ← stack[sp] + stack[sp + 1];

Associated to each function there is an Activation Record

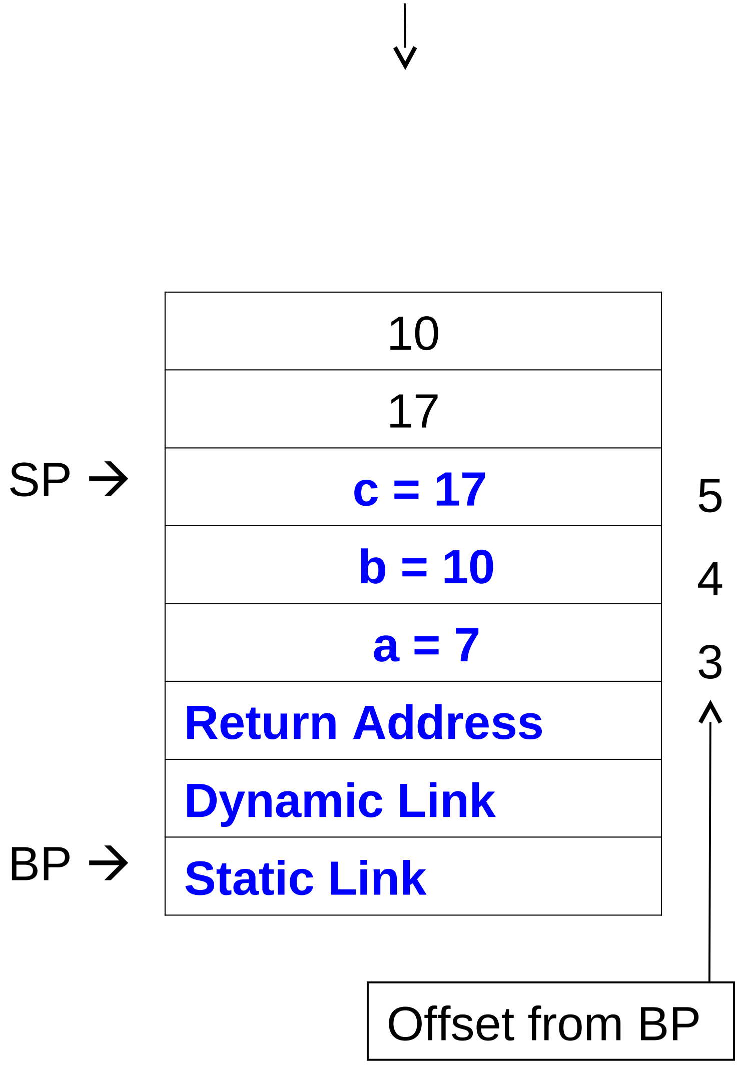
Top of stack value (17) is store in variable "c"

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let's run the program

```
Op   L   M
lit  0   7
sto  0   3
lit  0  10
sto  0   4
lod  0   3
lod  0   4
add  0   2
sto  0   5
rtn  0   0   ← PC
```

Stack after fetch-execute



	10	
	17	
SP →	**c = 17**	5
	b = 10	4
	a = 7	3
	Return Address	
	Dynamic Link	
BP →	**Static Link**	

Offset from BP

IR = STO L, M → stack[BP + **M]** ← stack[sp];
sp ← sp -1;

Associated to each function there is an Activation Record

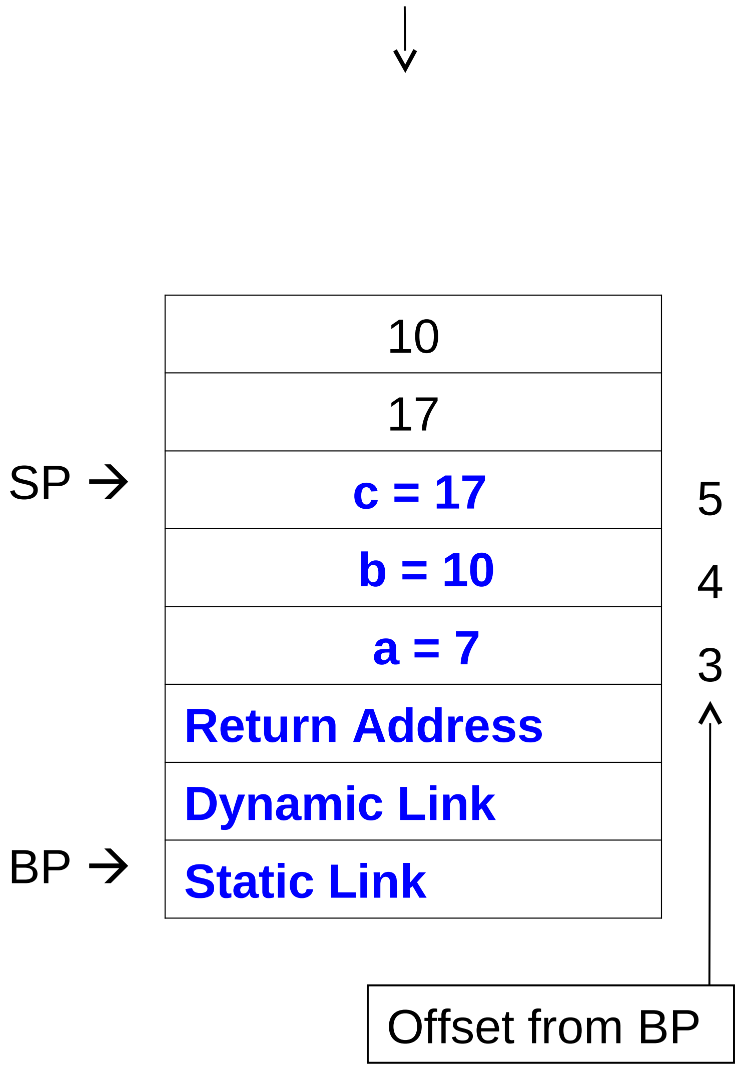
Return from Subroutine (RTN)

```
procedure SUB;
var a, b, c;
begin
  a:= 7;
  b:= 10;
  c:= a + b;
end;
```

Let’s run the program

```
Op   L   M
lit  0   7
sto  0   3
lit  0  10
sto  0   4
lod  0   3
lod  0   4
add  0   2
sto  0   5
rtn  0   0
???  ?   ?   ← PC
```

Stack after fetch-execute



10

17

SP → c = 17 5

b = 10 4

a = 7 3

Return Address

Dynamic Link

BP → Static Link

Offset from BP

IR = OPR 0, 0 → AR is deleted

P-machine: Code generation

In this example “ functional value field” is not considered

Programming example using PL/0

```
const n = 13;         /* constant declaration
var i,h;              /* variable declaration
procedure sub;
  const k = 7;
  var j,h;
  begin                  /* procedure
    j:=n;                /* declaration
    i:=1;
    h:=k;
  end;
begin  /* main starts here
  i:=3;
  h:=9;
  call sub;
end.
```

P-code for the program on the left

```
 0 jmp 0 10
 1 jmp 0 2
 2 inc 0 5
 3 lit 0 13
 4 sto 0 3
 5 lit 0 1
 6 sto 1 3
 7 lit 0 7
 8 sto 0 4
 9 opr 0 0
10 inc 0 5
11 lit 0 3
12 sto 0 3
13 lit 0 9
14 sto 0 4
15 cal 0 2
16 opr 0 0
```

Running a program on PM/0

		pc	bp	sp	stack
Initial values		0	1	0	0 0 0 0 0
0 jmp	0, 10	10	1	0	0 0 0 0 0
10 inc	0, 5	11	1	5	0 0 0 0 0
11 lit	0, 3	12	1	6	0 0 0 0 0 3
12 sto	0, 3	13	1	5	0 0 0 3 0
13 lit	0, 9	14	1	6	0 0 0 3 0 9
14 sto	0, 4	15	1	5	0 0 0 3 9
15 cal	0, 2	2	6	5	0 0 0 3 9 | 1 1 16
2 inc	0, 5	3	6	10	0 0 0 3 9 | 1 1 16 0 0
3 lit	0, 13	4	6	11	0 0 0 3 9 | 1 1 16 0 0 13
4 sto	0, 3	5	6	10	0 0 0 3 9 | 1 1 16 13 0
5 lit	0, 1	6	6	11	0 0 0 3 9 | 1 1 16 13 0 1
6 sto	1, 3	7	6	10	0 0 0 1 9 | 1 1 16 13 0
7 lit	0, 7	8	6	11	0 0 0 1 9 | 1 1 16 13 0 7
8 sto	0, 4	9	6	10	0 0 0 1 9 | 1 1 16 13 7
9 opr	0, 0	16	1	5	0 0 0 1 9

code	
0 jmp	0 10
1 jmp	0 2
2 inc	0 5
3 lit	0 13
4 sto	0 3
5 lit	0 1
6 sto	1 3
7 lit	0 7
8 sto	0 4
9 opr	0 0
10 inc	0 5
11 lit	0 3
12 sto	0 3
13 lit	0 9
14 sto	0 4
15 cal	0 2
16 sio	0 3

The P-machine (Register Version)

The ISA of the PM/0 (register version) has 22 instructions, a register file, and the instruction format has four components <op, r, l, m>:

OP **is the operation code**

R **Refers to a register.**

L **indicates the lexicographical level or a register in arithmetic and relational instructions. (L or R)**

M **depending of the operators it indicates:**

- A number (instructions: LIT, INC).
- A program address (instructions: JMP, JPC, CAL).
- A data address (instructions: LOD, STO)
- A register in arithmetic and logic instructions.
(e.g. ADD R[1], R[2], R[3] → R[1] = R[2] + R[3])

Instruction Set Architecture (ISA)

There are 13 arithmetic/logical operations that manipulate the data within the register file. These operations will be explained after the 11 basic instructions of PM/0.

ISA:

01 – **LIT**	**R, 0, M**	Loads a constant value (literal) **M** into Register **R**
02 – **RTN**	0, **0, 0**	Returns from a subroutine and restore the caller environment
03 – **LOD**	**R, L, M**	Load value into a selected register from the stack location at offset **M** from **L** lexicographical levels down
04 – **STO**	R, **L, M**	Store value from a selected register in the stack location at offset **M** from **L** lexicographical levels down
05 – **CAL**	**0, L, M**	Call procedure at address **M** (generates new Activation Record and pc ← **M**)
06 – **INC**	**0, 0, M**	Allocate **M** words (increment sp by M). First four are **Functional Value, Static Link (SL)**, **Dynamic Link (DL)**, and **Return Address (RA)**
07 – **JMP**	**0, 0, M**	Jump to instruction **M**
08 – **JPC**	**R, 0, M**	Jump to instruction **M** if **R** = 0
09 – **SIO**	R, **0, 1**	Write a register contents to the screen
10 **– SIO**	**R, 0, 2**	Read in a value from the user program and store it in a register
11– SIO	**0, 0, 3**	End of program (program stops running and call the OS)

ISA Pseudo Code

```
01 – LIT    R, 0,  M          R[i] ← M;

02 – RTN  0, 0, 0             sp ← bp - 1;
                              bp ← stack[sp + 3];
                              pc ← stack[sp + 4];

03 – LOD R, L, M              R[i] ← stack[ base(L, bp) + M];

04 – STO R, L, M              stack[ base(L, bp) + M] ← R[i];

05 - CAL   0, L, M            stack[sp + 1]  ← 0;              /* space to return value
                              stack[sp + 2]  ←  base(L, bp);   /* static link (SL)
                              stack[sp + 3]  ← bp;             /* dynamic link (DL)
                              stack[sp + 4]  ← pc;             /* return address (RA)
                              bp ← sp + 1;
                               pc ← M;

06 – INC   0, 0, M            sp ← sp + M;

07 – JMP   0, 0, M            pc ← M;

08 – JPC   R, 0, M            if R[i] == 0 then { pc ← M; }

09 – SIO   R, 0, 1            print(R[i]);

10 – SIO   R, 0, 2            read(R[i]);

11 – SIO   R, 0, 3            Set Halt flag to zero, assuming that Halt = 1 controls the Fetch-execute loop; (End of
                              program)
```

ISA Pseudo Code (arithmetic and relational instructions)

12 - **NEG**	(R[i] ← -R[i])
13 - **ADD**	(R[i] ← R[j] + R[k])
14 - **SUB**	(R[i] ← R[j] - R[k])
15 - **MUL**	(R[i] ← R[j] * R[k])
16 - **DIV**	(R[i] ← R[j] / R[k])
17 - **ODD**	(R[i] ← R[i] mod 2) or ord(odd(R[i]))
18 - **MOD**	(R[i] ← R[j] mod R[k])
19 - **EQL**	(R[i] ← R[j] = = R[k])
20 - **NEQ**	(R[i] ← R[j] != R[k])
21 - **LSS**	(R[i] ← R[j] < R[k])
22 - **LEQ**	(R[i] ← R[j] <= R[k])
23 - **GTR**	(R[i] ← R[j] > R[k])
24 - **GEQ**	(R[i] ← R[j] >= R[k])

P-machine: Code generation (Regs)

Programming example using PL/0

```
const n = 13;          /* constant declaration
var i,h;               /* variable declaration
procedure sub;
  const k = 7;
  var j,h;
  begin
    j:=n;                 /* procedure
    i:=1;                 /* declaration
    h:=k;
  end;
begin  /* main starts here
  i:=3;
  h:=9;
  call sub;
end.
```

P-code for the program on the left

```
0 jmp  0 0 10
1 jmp  0 0 2
2 inc   0 0 6
3 lit     0 0 13
4 sto   0 0 4
5 lit     0 0 1
6 sto   0 1 4
7 lit     0 0 7
8 sto   0 0 5
9 opr   0 0 0
10 inc   0 0 6
11 lit     0 0 3
12 sto  0 0 4
13 lit    0 0 9
14 sto  0 0 5
15 cal  0 0 2
16 sio  0 0 3
```

Program on PM/0 with registers

		pc	bp	sp	stack
Initial values		0	1	0	0 0 0 0 0
0 jmp	0, 0, 10	10	1	0	0 0 0 0 0
10 inc	0, 0, 6	11	1	6	0 0 0 0 0
11 lit	0, 0, 3	12	1	6	0 0 0 0 0
R0 = 3, R1 =0, R2 = 0, etc.					
12 sto	0, 0, 4	13	1	6	0 0 0 0 3
13 lit	0, 0, 9	14	1	6	0 0 0 0 3
14 sto	0, 0, 5	15	1	6	0 0 0 0 3 9
15 cal	0, 0, 2	2	7	6	0 0 0 0 3 9
2 inc	0, 0, 6	3	7	12	0 0 0 0 3 9| 0 1 1 16
3 lit	0, 0, 13	4	7	12	0 0 0 0 3 9| 0 1 1 16
4 sto	0, 0, 4	5	7	12	0 0 0 0 3 9| 0 1 1 16 13
5 lit	0, 0, 1	6	7	12	0 0 0 0 3 9| 0 1 1 16 13
6 sto	0, 1, 4	7	7	12	0 0 0 0 1 9| 0 1 1 16 13
7 lit	0, 0, 7	8	7	12	0 0 0 0 1 9| 0 1 1 16 13
8 sto	0, 0, 5	9	7	12	0 0 0 0 1 9| 0 1 1 16 13 7
9 rtn	0, 0, 0	16	1	6	0 0 0 0 1 9
16sio	0, 0, 3	17	1	6	0 0 0 0 1 9

code

```
0 jmp   0 0 10
1 jmp   0 0 2
2 inc   0 0 6
3 lit   0 0 13
4 sto   0 0 4
5 lit   0 0 1
6 sto   0 1 4
7 lit   0 0 7
8 sto   0 0 5
9 opr   0 0 0
10 inc  0 0 6
11 lit  0 0 3
12 sto  0 0 4
13 lit  0 0 9
14 sto  0 0 5
15 cal  0 0 2
16 sio  0 0 3
```

COP 3402 Systems Software

Virtual Machines as instruction interpreters

(The End)