Lecture 2: PM/0 Virtual Machine

Outline

- Virtual Machines as software interpreters
- P-code: Instruction Set Architecture
- Instruction Format
- Assembly Language

Virtual Machine: P-code

- The Pseudo-code machine is a software (virtual) machine that implements the instruction set architecture of a stack-based computer.
- 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 different instructions.

The instruction format has three components <**op, l, m**>:

- **op** **Operation Code** (op or opcode)
- **l** **Lexicographical Level** (level)
- **m** **Modifier** indicates depending on op (mnemonic)

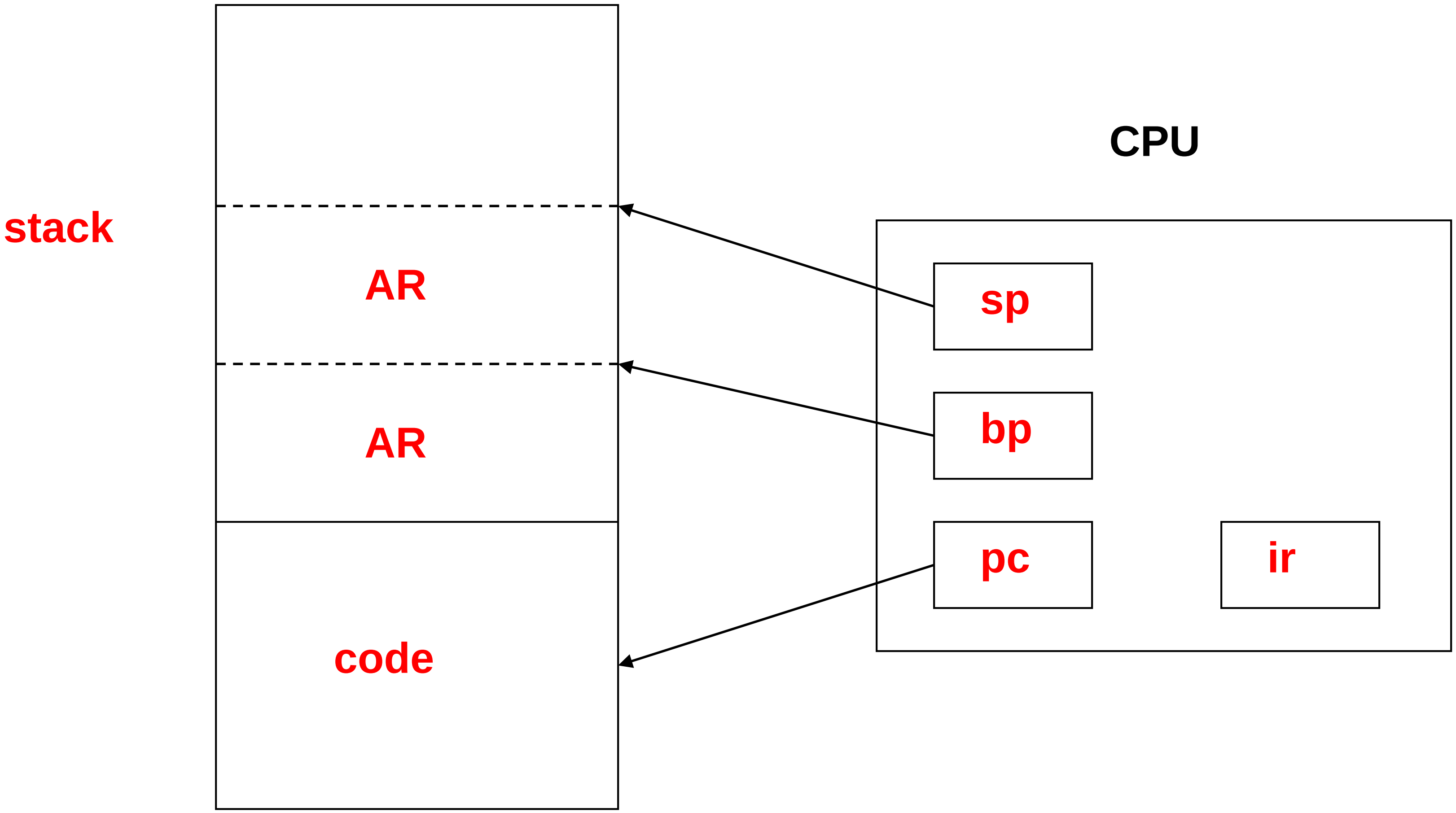
o **Number**	LIT, INT
o **Program Address**	JMP, JPC, CAL
o **Data Address**	LOD, STO
o **Identity of the operator**	OPR

Virtual Machine: P- code

The P-machine (PM/0) consists of:

- **stack** a store organized as a stack
- **code** a store that contains the instructions
- **CPU** with four registers:
  - **bp** points to the base of the current **Activation Record (AR)** in the stack
  - **sp** points to the top of the stack
  - **pc** program counter or instruction pointer
  - **ir** instruction register

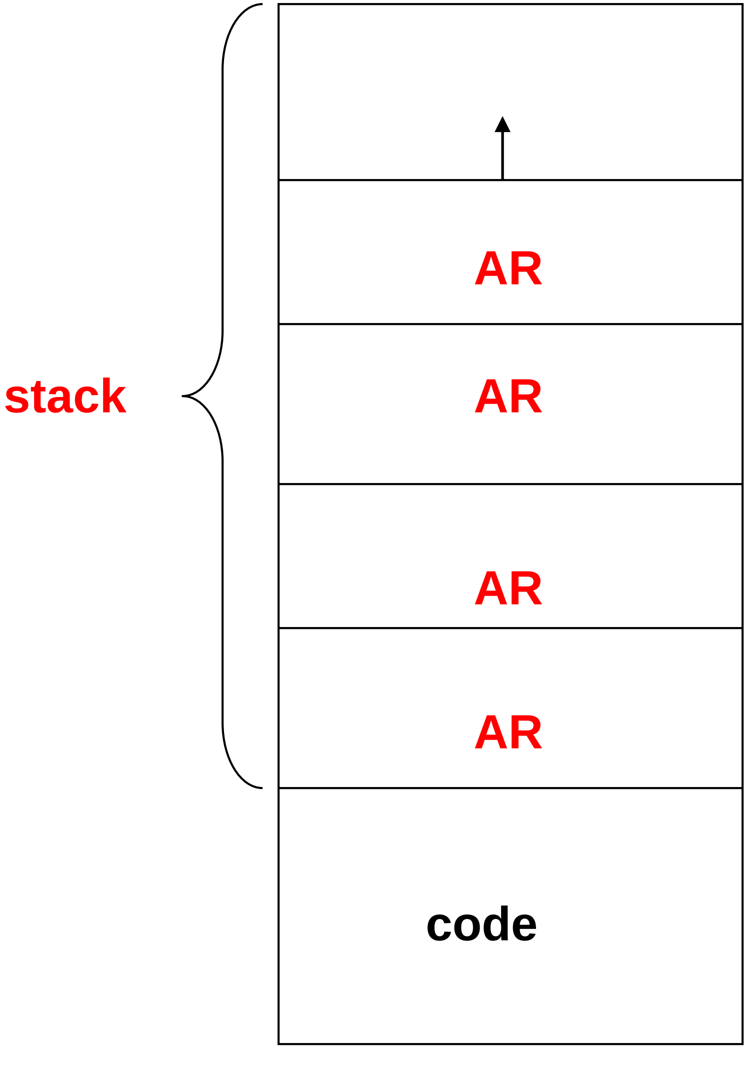
Virtual Machine: P- code



Activation Records (AR)

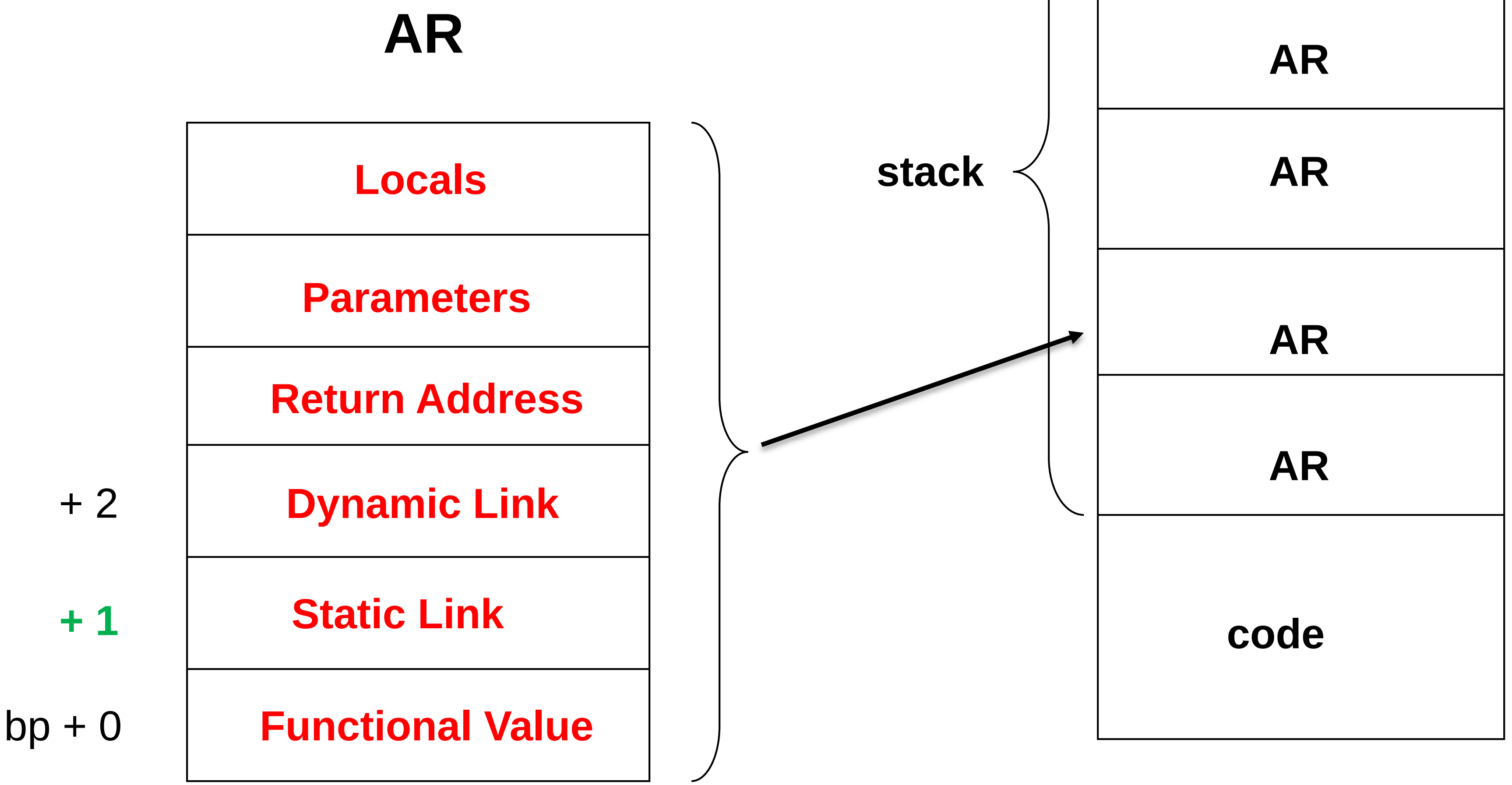
- **Activation Record** or **Stack Frame**: data structure that push onto stack, each time a procedure/function is called

- AR contains all information necessary to control the execution of the subroutine



Activation Records (AR)

The order of FV, **SL**, DL, RA is consistent with **+ 1** in the **base function** (useful for the implementation of the PM/0 machine).



Activation Records (AR)

Control Information

- **Return Address** points to the next instruction of the **caller** to be executed after returning from the **callee**, that is, the current function/procedure

- **Dynamic Link** points to the base of the previous AR, that is, the AR of the caller

- **Static Link** points to the AR of the procedure/function that statically encloses the callee

Note that the procedure/function that statically encloses the callee is not necessarily the caller.

For instance, A statically encloses B and B calls itself recursively.

Accessing Values in Activation Records

How to compute the base of activation record **L** levels down

```
int base( int level, int b ) {
  while (level > 0) {
    b = stack[ b + 1 ];
    level--;
  }
  return b;
}
```

The order of FV, **SL**, DL, RA is consistent with the **+ 1** in the base function.

Activation Records (AR)

Control Information

- **Functional Value** is the location storing the return value of the callee
- **Parameters** are the locations storing the parameters of the callee passed by the caller
- **Locals** are the locations storing the local variables declared within the callee

Instruction Cycle

The instruction cycle consists of two steps:

- **Fetch Cycle**
  - an instruction is fetched from the code store

    ```
    ir ← code[ pc ];
    ```

  - the program counter is incremented by one

    ```
    pc ← pc + 1;
    ```

- **Execute Cycle**
  - `ir.op` indicates the operation to be executed
  - When the opcode `ir.op` equals 02 (OPR) or 09 (SIO), then the modifier `ir.m` further identifies the instruction

Informal Description of ISA

03 **LOD** **L** **M** Get value at offset **M** in frame **L** levels down and push it

opcode mnemonic level modifier

Informal Description

01 **LIT**	**0**	**M**	Push value **M** onto stack	
02 **OPR**	**0**	**M**	Perform return (**RET**) or arithmetic/logical operation (**ADD**, **EQL**, …)	
03 **LOD**	**L**	**M**	Get value at offset **M** in frame **L** levels down and push it	
04 **STO**	**L**	**M**	Pop stack and insert value at offset **M** in frame **L** levels down	
05 **CAL**	**L**	**M**	Call procedure at **M** (generates new stack frame)	
06 **INC**	**0**	**M**	Allocate **M** locals on stack	
07 **JMP**	**0**	**M**	Jump to **M**	
08 **JPC**	**0**	**M**	Pop stack and jump to **M** if value is equal to 0	
09 **SIO**	**0**	**0**	Pop stack and print out value	**OUT**
09 **SIO**	**0**	**1**	Read in input from user and push it	**INP**
09 **SIO**	**0**	**2**	Halt the machine	**HLT**

Formal Definition of ISA

01 **LIT** **0** **M** Push value **M** onto stack

```
sp ← sp + 1;
stack[ sp ] ← M;
```

03 **LOD** **L** **M** Get value in frame **L** levels down at offset **M** and push it

```
sp ← sp + 1;
stack[ sp ] ← stack[ base( L, bp ) + M ];
```

04 **STO** **L** **M** Pop stack and insert value in frame **L** levels down at offset **M**

```
stack[ base( L, bp ) + M ] ← stack[ sp ];
sp ← sp – 1;
```

Formal Definition

05 **CAL** L M Call procedure at M (generates new stack frame)

```
stack[ sp + 1 ] ← 0;                 // functional value (FV)
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;
```

Why + 2 and not + 1 ?

The order of FV, **SL**, DL, RA is consistent with the **+ 1** in the base function.

06 **INC** 0 M Allocate M locals on stack

```
sp ← sp + M;
```

Formal Definition

07 **JMP** **0** **M** Jump to **M**

```
pc ← sp + M;
```

08 **JPC** **0** **M** Pop stack and jump to **M** if value is equal to 0

```
if ( stack[ sp ] == 0 ) then { pc ← M; }
sp ← sp – 1;
```

Formal Definition

09 SIO 0 M

Recall that when the opcode is equal to 09 (mnemonic SIO), the operation to be executed is further determined by the modifier **M**

0 OUT Pop stack and print out value

```
print( stack[ sp ] );
sp ← sp – 1;
```

1 INP Read in input from user and push it

```
sp ← sp + 1;
read( stack[ sp ] );
```

2 HLT Halt the machine (your virtual machine stops)

```
halt;
```

Formal Definition

02 OPR 0 M

Recall that when the opcode is equal to 02 (mnemonic OPR), the operation to be executed is further determined by the modifier **M**

Formal Definition

02 OPR 0 M

The only operation with **no** argument

0 RTN Return from function or procedure

```
sp ← bp – 1;
pc ← stack[ sp + 4 ]; // return address (RA)
bp ← stack[ sp + 3];  // dynamic link (DL)
```

Formal Definition

2 OPR 0 M

Operations with **one** argument

M

1 NEG `stack[ sp ] ← - stack[ sp ];`

6 ODD `stack[ sp ] ← stack[ sp ] mod 2;`

Formal Definition

02 OPR 0 M

Operations with **two** arguments:

M for all operations below, perform first

```
sp ← sp – 1;
```

```
2  ADD stack[sp] ← stack[ sp ] + stack[ sp + 1 ];
3  SUB stack[sp] ← stack[ sp ] - stack[ sp + 1 ];
4  MUL stack[sp] ← stack[ sp ] * stack[ sp + 1 ];
5  DIV stack[sp] ← stack[ sp ] div stack[ sp + 1 ];
6  MOD stack[sp] ← stack[ sp ] mod stack[ sp + 1 ];

8  EQL stack[sp] ← stack[ sp ] == stack[ sp + 1 ];
9  NEQ stack[sp] ← stack[ sp ] != stack[ sp + 1 ];
10 LSS stack[sp] ← stack[ sp ] <  stack[ sp + 1 ];
11 LEQ stack[sp] ← stack[ sp ] <= stack[ sp + 1 ];
12 GTR stack[sp] ← stack[ sp ] >  stack[ sp + 1 ];
13 GEQ stack[sp] ← stack[ sp ] >= stack[ sp + 1 ];
```

P-machine: Code Generation

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;
    i:=1;
    h:=k;
  end;
begin  /* main starts here
  i:=3;
  h:=0;
  call sub;
end;
```

/* procedure
/* declaration

P-code

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

Running a program on PM/0

				pc	bp	sp	stack
Initial values				0	1	0	
0	jmp	0	10	10	1	0	
10	inc	0	6	11	1	6	0 0 0 0 0 0
11	lit	0	3	12	1	7	0 0 0 0 0 0 3
12	sto	0	4	13	1	6	0 0 0 0 3 0
13	lit	0	0	14	1	7	0 0 0 0 3 0 0
14	sto	0	5	15	1	6	0 0 0 0 3 0
15	cal	0	2	2	7	6	0 0 0 0 3 0
2	inc	0	6	3	7	12	0 0 0 0 3 0 | 0 1 1 16 0 0
3	lit	0	13	4	7	13	0 0 0 0 3 0 | 0 1 1 16 0 0 13
4	sto	0	4	5	7	12	0 0 0 0 3 0 | 0 1 1 16 13 0
5	lit	0	1	6	7	13	0 0 0 0 3 0 | 0 1 1 16 13 0 1
6	sto	1	4	7	7	12	0 0 0 0 1 0 | 0 1 1 16 13 0
7	lit	0	7	8	7	13	0 0 0 0 1 0 | 0 1 1 16 13 0 7
8	sto	0	5	9	7	12	0 0 0 0 1 0 | 0 1 1 16 13 7
9	opr	0	0	16	1	6	0 0 0 0 1 0
16	sio	0	2	17	1	6	0 0 0 0 1 0