Lecture 4

COP3402 FALL 2015 – DR. MATTHEW GERBER – 9/2/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- Virtual Machines and P-Machines
- PM/0

Virtual Machines

A **virtual machine** is simply a logical computer created by software on another computer.

This can involve completely emulating the hardware of another actual machine:

- Virtual PC, VirtualBox, Parallels, VMWare, etc.
- Mobile device development target emulation
- Video game emulation

This can also involve "emulating" a *notional* machine:

- The Java Virtual Machine
- The Dalvik runtime for Android, and its replacement ART
- The .NET Common Language Runtime
- The P-Machine
  - …wait, what?

P-Machines

P-Machines, or P-Code Machines, or Portable Code Machines, or any of several other terms, are machines intended to execute code for notional computers.

They have existed since 1966. Nothing about them is new.

The P-Machine (then “p-Machine”) concept was codified for the **Pascal-P** system.

- The first really good version was Pascal-P2, which Nikalus Wirth pulled together himself in 1974
- This was a genuine attempt to create a complete portable development and computing environment
- The similarities to Java are obvious – and so are some of the problems
- P-Machine code was slower and less capable than native code
- Native systems eventually won, and **Turbo Pascal** replaced the variant UCSD P-System as the most popular version of Pascal

We are going to become acquainted with a simple stack-based P-machine.

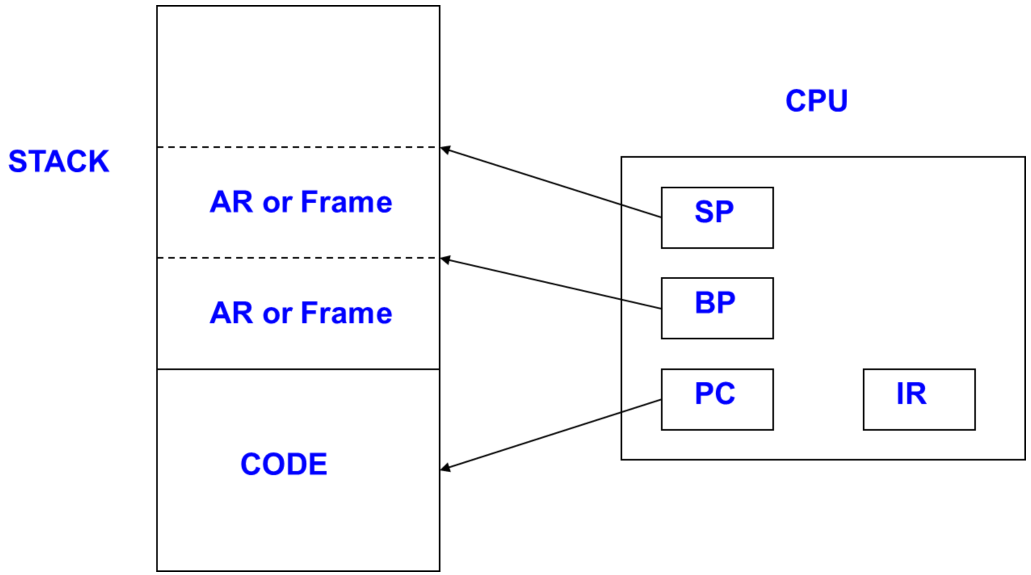
PM/0

PM/0

- …is a *stack architecture*
  - All data storage (yes, *all* data storage) is handled by a single stack
  - Code is stored separately
- …contains features designed to support function and procedure calls
  - We will see that each function call results in a new *activation record* on the stack
    - An activation record is also known as a *stack frame*
    - Executing the program in the first place counts as a function call
  - Activation records contain all the information necessary to pass data and control between the caller and the called subprogram
- …has a small number of registers, that are *not* general-purpose
  - Arithmetic is done on the stack

Registers

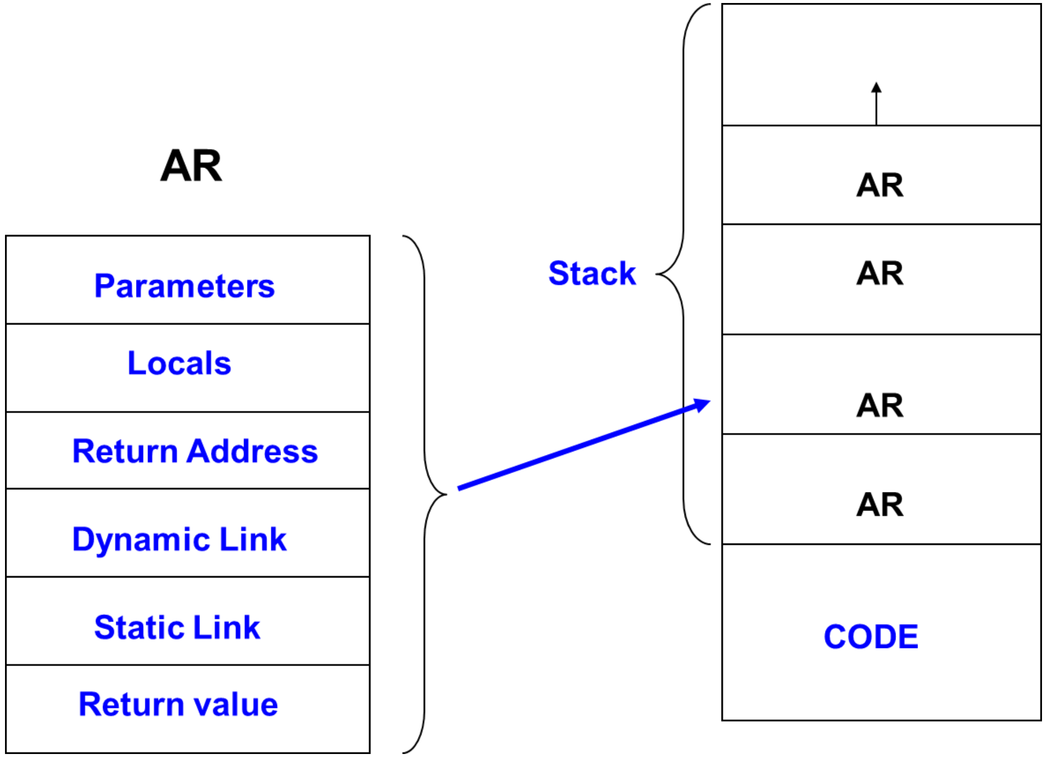
- The **Stack Pointer (SP)** always points to the very top of the stack
- The **Base Pointer (BP)** points to the base of the current activation record
- The **Program Counter (PC)** and **Instruction Register (IR)** are familiar



Activation Records

Again, an AR is created each time a function or procedure is called.

- There are as many ARs as there are functions currently pending
- ARs contain six important values
- We'll discuss the **Return value** (or **FV**, for **Function Value**) and **Parameters** when it's time to actually implement parameterized subprograms
- The other four we need now



The Activation Record

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

Return Address: A pointer to the next instruction to be executed after the current function or procedure ends.

Dynamic Link: A pointer to the caller's frame.

Static Link: A pointer to the frame of the procedure that statically encloses the current function or procedure.

- *We'll clarify this further when we get to subprograms – don't worry too much about it now.*

Activation Record
Parameters
Locals
Return Address
Dynamic Link
Static Link
Return Value

The Instruction Cycle

Like any other von Neumann machine, the P-Machine uses a fetch-execute cycle.

- **Fetch Cycle**
  - Fetch an instruction from the code store `ir ← code[pc]`
  - Increment the program counter `pc ← pc + 1`
- **Execute Cycle**
  - Each instruction is of the format <**OP**, **L**, **M**>
  - **OP** is the opcode
  - **L** is the *lexicographical level* – the number of frames to walk back when performing the instruction
  - **M** is the parameter, and means different things depending on the instruction type

The Instruction Set

Op	Mnemonic	Description
01	LIT 0, M	Push the literal value M onto the stack.
02	OPR 0, 0	Return from a procedure call.
02	OPR 0, M	Perform an ALU operation, specified by M.
03	LOD L, M	Read the value at offset M from L levels down (if L=0, our own frame) and push it onto the stack.
04	STO L, M	Pop the stack and write the value into offset M from L levels down – if L=0, our own frame.
05	CAL L, M	Call the procedure at M.
06	INC 0, M	Allocate enough space for M local variables. We will always allocate at least four.
07	JMP 0, M	Branch to M.
08	JPC 0, M	Pop the stack and branch to M if the result is 0.
09	SIO 0, 1	Pop the stack and write the result to the screen.
10	SIO 0, 2	Read an input from the user and store it at the top of the stack.
11	SIO 0, 3	Stop the machine.

ALU Operations

Operation	Name	Description
OPR 0, 1	NEG	Pop the stack and push the negation of the result.
OPR 0, 2	ADD	Pop the stack twice, add the values, and push the result.
OPR 0, 3	SUB	Pop the stack twice, subtract the top value from the second value, and push the result.
OPR 0, 4	MUL	Pop the stack twice, multiply the values, and push the result.
OPR 0, 5	DIV	Pop the stack twice, divide the second value by the top value, and push the quotient.
OPR 0, 6	ODD	Pop the stack, push 1 if the value is odd, and push 0 otherwise.
OPR 0, 7	MOD	Pop the stack twice, divide the second value by the top value, and push the remainder.
OPR 0, 8	EQL	Pop the stack twice and compare the top value t with the second value s. Push 1 if $s = t$ and 0 otherwise.
OPR 0, 9	NEQ	Pop the stack twice and compare the top value t with the second value s. Push 1 if $s \neq t$ and 0 otherwise.
OPR 0, 10	LSS	Pop the stack twice and compare the top value t with the second value s. Push 1 if $s < t$ and 0 otherwise.
OPR 0, 11	LEQ	Pop the stack twice and compare the top value t with the second value s. Push 1 if $s \leq t$ and 0 otherwise.
OPR 0, 12	GTR	Pop the stack twice and compare the top value t with the second value s. Push 1 if $s > t$ and 0 otherwise.
OPR 0, 13	GEQ	Pop the stack twice and compare the top value t with the second value s. Push 1 if $s \geq t$ and 0 otherwise.

Instruction Pseudocode

Op	Mnemonic	Pseudocode
01	LIT 0, M	sp ← sp + 1; stack[sp] ← M;
02	OPR 0, 0 (Return)	sp ← bp -1; pc ← stack[sp + 4]; bp ← stack[sp + 3];
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;
05	CAL L, M	/* FV, SL, DL, RA */ stack[sp+1] ← 0; stack[sp+2] ← base(L); stack[sp+3] ← bp; stack[sp+4] ← pc; bp ← sp + 1; pc ← M;

Op	Mnemonic	Pseudocode
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	SIO 0, 1	print (stack[sp]); sp ← sp – 1;
10	SIO 0, 2	sp ← sp + 1; read (stack[sp]);
11	SIO 0, 3	halt;

Base(L) is the base of the stack frame L levels down from ours.
If L is 0, it's our own frame.

Code Generation: A Compiled Example

```
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.
```

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	3

Running a Program

				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	3	0	0	0	

Next Time:
More on Subprograms