Interpreters

- An interpreter is like a CPU, only in software.
- The compiler generates virtual machine (VM) code rather than native machine code.
- The interpreter executes VM instructions rather than native machine code.

Interpreters are

- **slow** Often 10–100 times slower than executing machine code directly.
- **portable** The virtual machine code is not tied to any particular architecture.

Interpreters work well with

very high-level, dynamic languages (APL, Prolog, ICON) where a lot is unknown at compile-time (array bounds, etc).
Actions in an Interpreter...

Kinds of Interpreters...

Stack-Based Instruction Sets

- Many virtual machine instruction sets (e.g., Java bytecode, Forth) are stack based.

- **add** pop the two top elements off the stack, add them together, and push the result on the stack.

- **push X** push the value of variable X.

- **pusha X** push the address of variable X.

- **store** pop a value V, and an address A off the stack. Store V at memory address A.

Actions in an Interpreter

- Internally, an interpreter consists of
  1. The interpreter *engine*, which executes the VM instructions.
  2. *Memory* for storing user data. Often separated as a heap and a stack.
  3. A stream of VM instructions.
Register-Based Instruction Sets...

- Here’s an example of a small program and the corresponding register code:

<table>
<thead>
<tr>
<th>Source Code</th>
<th>VM Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR X, Y, Z : INTEGER; BEGIN X := Y + Z; END;</td>
<td>pusha X load R₁, Y load R₂, Z add R₃, R₁, R₂ loada R₄, X store R₄, R₃</td>
</tr>
</tbody>
</table>

Stack-Based Instruction Sets...

- Here’s an example of a small program and the corresponding stack code:

<table>
<thead>
<tr>
<th>Source Code</th>
<th>VM Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR X, Y, Z : INTEGER; BEGIN X := Y + Z; END;</td>
<td>pusha X push Y push Z add store</td>
</tr>
</tbody>
</table>

- Stack codes are *compact*. If we don’t worry about code size, we can use any intermediate code (tuples, trees). Example: RISC-like VM code with ∞ number of virtual registers R₁, · · ·:

Add VM registers R₂ and R₃ and store in VM register R₁.

load R₁, X R₁ := value of variable X.

loada R₁, X R₁ := address of variable X.

store R₁, R₂ Store value R₂ at address R₁.
**Switch Threading**

- Instructions are stored as an array of integer tokens. A switch selects the right code for each instruction.

```c
typedef enum {add,load,store,...} Inst;
void engine () {
    static Inst prog[] = {load,add,...};
    Inst *pc = &prog;
    int Stack[100]; int sp = 0;
    for (;;) {
        switch (*pc++) {
            case add: Stack[sp-1]=Stack[sp-1]+Stack[sp]; sp--; break;
        }
    }
}
```

**Switch Threading in Java**

- Let’s look at a simple Java switch interpreter.
- We have a stack of integers `stack` and a stack pointer `sp`.
- There’s an array of bytecodes `prog` and a program counter `pc`.
- There is a small memory area `memory`, an array of 256 integers, numbered 0–255. The LOAD, STORE, ALOAD, and ASTORE instructions access these memory cells.

```java
void engine () {
    static Inst prog[] = {load,add,...};
    Inst *pc = &prog;
    int Stack[100]; int sp = 0;
    for (;;) {
        switch (*pc++) {
            case add: Stack[sp-1]=Stack[sp-1]+Stack[sp]; sp--; break;
        }
    }
}
```
### Example programs

This program prints a newline character and then exits:

```
PRINTLN
EXIT
```

Or, in binary: \(\langle 8,9 \rangle\)

This program prints the number 10, then a newline character, and then exits:

```
PUSHB 10
PRINT
PRINTLN
EXIT
```

Or, in binary: \(\langle 6,10,7,8,9 \rangle\)

### Bytecode semantics

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>opcode</th>
<th>stack-pre</th>
<th>stack-post</th>
<th>side-effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
<td>([A,B])</td>
<td>([A+B])</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>([A,B])</td>
<td>([A-B])</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>2</td>
<td>([A,B])</td>
<td>([A*B])</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
<td>([A,B])</td>
<td>([A-B])</td>
<td></td>
</tr>
<tr>
<td>LOAD X</td>
<td>4</td>
<td>([])</td>
<td>([\text{Memory}[X]])</td>
<td></td>
</tr>
<tr>
<td>STORE X</td>
<td>5</td>
<td>([A])</td>
<td>([])</td>
<td>\text{Memory}[X] = A</td>
</tr>
<tr>
<td>PRINT</td>
<td>7</td>
<td>([A])</td>
<td>([])</td>
<td>Print A</td>
</tr>
<tr>
<td>PRINTLN</td>
<td>8</td>
<td>([])</td>
<td>([])</td>
<td>Print a newline</td>
</tr>
<tr>
<td>EXIT</td>
<td>9</td>
<td>([])</td>
<td>([])</td>
<td>The interpreter exits</td>
</tr>
<tr>
<td>PUSHW X</td>
<td>11</td>
<td>([])</td>
<td>([X])</td>
<td></td>
</tr>
</tbody>
</table>

### Example programs...

This program pushes two values on the stack, then performs an \texttt{ADD} instruction which pops these two values off the stack, adds them, and pushes the result. \texttt{PRINT} then pops this value off the stack and prints it:

```
PUSHB 10
PUSHB 20
ADD
PRINT
PRINTLN
EXIT
```

Or, in binary: \(\langle 6,10,6,20,0,7,8,9 \rangle\)

### Bytecode semantics...

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>opcode</th>
<th>stack-pre</th>
<th>stack-post</th>
<th>side-effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ L</td>
<td>12</td>
<td>([A,B])</td>
<td>([])</td>
<td>if (A = B) then (PC++L)</td>
</tr>
<tr>
<td>BNE L</td>
<td>13</td>
<td>([A,B])</td>
<td>([])</td>
<td>if (A \neq B) then (PC++L)</td>
</tr>
<tr>
<td>BLT L</td>
<td>14</td>
<td>([A,B])</td>
<td>([])</td>
<td>if (A &lt; B) then (PC++L)</td>
</tr>
<tr>
<td>BGT L</td>
<td>15</td>
<td>([A,B])</td>
<td>([])</td>
<td>if (A &gt; B) then (PC++L)</td>
</tr>
<tr>
<td>BLE L</td>
<td>16</td>
<td>([A,B])</td>
<td>([])</td>
<td>if (A \leq B) then (PC++L)</td>
</tr>
<tr>
<td>BGE L</td>
<td>17</td>
<td>([A,B])</td>
<td>([])</td>
<td>if (A \geq B) then (PC++L)</td>
</tr>
<tr>
<td>BRA L</td>
<td>18</td>
<td>([])</td>
<td>([])</td>
<td>(PC++L)</td>
</tr>
<tr>
<td>ALOAD</td>
<td>19</td>
<td>([X])</td>
<td>([\text{Memory}[X]])</td>
<td></td>
</tr>
<tr>
<td>ASTORE</td>
<td>20</td>
<td>([A,X])</td>
<td>([])</td>
<td>\text{Memory}[X] = A</td>
</tr>
<tr>
<td>SWAP</td>
<td>21</td>
<td>([A,B])</td>
<td>([B,A])</td>
<td></td>
</tr>
</tbody>
</table>
Example program...

This program uses the `LOAD` and `STORE` instructions to store a value in memory cell number 7:

```
PUSHB 10
STORE 7
PUSHB 10
LOAD 7
MUL
PRINT
PRINTLN
EXIT
```

Or, in binary: (6,10,5,7,6,10,4,7,2,7,8,9)

# Print the numbers 1 through 9.
# i = 1; while (i < 10) do {print i; println; i++;}
PUSHB 1  # mem[1] = 1;
STORE 1
LOAD 1   # if mem[1] < 10 goto exit
PUSHB 10
BGE
LOAD 1   # print mem[i] value
PRINT
PRINTLN
PUSHB 1  # mem[i]++
LOAD 1
ADD
STORE 1
BRA    # goto top of loop
EXIT
### Bytcode Description...

**BEQ L**: Pop the two top integers $A$ and $B$ off the stack, if $A == B$ then continue with instruction $PC + L$, where $PC$ is address of the instruction following this one. Otherwise, continue with the next instruction.

**BNE L**: As above, but branch if $A \neq B$.

**BLT L**: As above, but branch if $A < B$.

**BGT L**: As above, but branch if $A > B$.

**BLE L**: As above, but branch if $A \leq B$.

**BGE L**: As above, but branch if $A \geq B$.

**BRA L**: Continue with instruction $PC + L$, where $PC$ is the address of the instruction following this one.

---

### Switch Threading in Java

```java
public class Interpreter {
    static final byte ADD = 0;
    static final byte SUB = 1;
    static final byte MUL = 2;
    static final byte DIV = 3;
    static final byte LOAD = 4;
    static final byte STORE = 5;
    static final byte PUSHB = 6;
    static final byte PRINT = 7;
    static final byte PRINTLN = 8;
    static final byte EXIT = 9;
    static final byte PUSHW = 11;

    static final byte BEQ = 12;
    static final byte BNE = 13;
    static final byte BLT = 14;
    static final byte BGT = 15;
    static final byte BLE = 16;
    static final byte BGE = 17;
    static final byte BRA = 18;
    static final byte ALOAD = 19;
    static final byte ASTORE = 20;
    static final byte SWAP = 21;

    static void interpret (byte[] prog) throws Exception {
        int[] stack = new int[100];
        int[] memory = new int[256];
        int pc = 0;
        int sp = 0;
        while (true) {
            switch (prog[pc]) {
                case ADD : {
                    stack[sp-2]+=stack[sp-1]; sp--;
                    pc++; break;
                }
                case SUB, MUL, DIV : {
                    stack[sp-2]+=stack[sp-1]; sp--;
                    pc++; break;
                }
            }
        }
    }
}
```
case PRINTLN: {
    System.out.println(); pc++; break; }

case EXIT : {return;}

case BEQ : { /*Same for BNE,BLT,BGT,BLE,BGE*/
    pc+= ((stack[sp-2]==stack[sp-1])?
    2+(int)prog[pc+1]:2;
    sp-=2; break; }

case BRA : {
    pc+= 2+(int)prog[pc+1]; break; }

case SWAP : {
    int tmp = stack[sp-1];
    stack[sp-1] = stack[sp-2];
    stack[sp-2]=tmp;
    pc++; break; }

/* Similar for PUSHW. */

switch (e) {
    case 1: S1; break;
    case 3: S2; break; ⇒ Lab1: S1; goto Lab4;
    default: S3;
    }

JumpTab = {0,&Lab1,&Lab3,&Lab2};

if ((e < 1) || (e > 3)) goto Lab3;
    goto *JumpTab[e];

Lab1: S1; goto Lab4;
Lab2: S2; goto Lab4;
Lab3: S3;
Lab4:

int tmp = stack[sp-1];
stack[sp-1] = stack[sp-2];
stack[sp-2]=tmp;
pc++; break; }

case PUSHB : {
    stack[sp] = (int)prog[pc+1];
    sp++; pc+=2; break; }

/* Similar for PUSHW. */

case PRINT : {
    System.out.print(stack[--sp]);
    pc++; break; }

Direct Call Threading...

- Every instruction is a separate function.
- The program `prog` is an array of pointers to these functions.
- I.e. the `add` instruction is represented as the address of the `add` function.
- `pc` is a pointer to the current instruction in `prog`.
- `pc` jumps to the function that `pc` points to, then increments `pc` to point to the next instruction.
- Hard to implement in Java.

```c
typedef void (*Inst)();
Inst prog[] = {&load,&add,···};

Inst *pc = &prog;
int Stack[100]; int sp = 0;

void add() {  
    Stack[sp-1]=Stack[sp-1]+Stack[sp];
    sp--;
}

void engine () {  
    for (;;) (*pc++)();
}
```
Direct Threading...

- Direct threading is the most efficient method for instruction dispatch.

VM Code Program (32/64-bit address)

- Each instruction is represented by the address (label) of the code that implements it.
- At the end of each piece of code is an indirect jump `goto *pc++` to the next instruction.
- `&&` takes the address of a label. `goto *V` jumps to the label whose address is stored in variable `V`. This is a gcc extensions to C.

Indirect Threading

- Unfortunately, a direct threaded program will be large since each instruction is an address (32 or 64 bits).
- At the cost of an extra indirection, we can use byte-code instructions instead.
- `prog` is an array of bytes.
- `jtab` is an array of addresses of instructions.
- `goto *jtab[*pc++]` finds the current instruction (what `pc` points to), uses this to index `jtab` to get the address of the instruction, jumps to this code, and finally increments `pc`.

```c
typedef void *Inst
static Inst prog[]={{&&add,&&sub},...};

void engine() {
    Inst *pc = &prog;
    int Stack[100]; int sp=0;
    goto **pc++;

    add: Stack[sp-1]=Stack[sp]; sp--; goto **pc++;
    sub: Stack[sp-1]=Stack[sp]; sp--; goto **pc++;
}
```
Minimizing Stack Accesses

- To reduce the cost of stack manipulation we can keep one or more of the Top-Of-Stack elements in registers.
- In the example below, TOS holds the top stack element. Stack[sp] holds the element second to the top, etc.

```c
void engine() {
    static Inst prog[]={&add,&load,...};
    Inst *pc = &prog; int sp; register int TOS;
    goto *pc++;
    add:   TOS+=Stack[sp]; sp--; goto *pc++;
    store: Memory[Stack[sp]]=TOS; TOS=Stack[sp-1]; sp-=2;
           goto *pc++;
}
```

Instruction Sets Revisited

- We can (sometimes) speed up the interpreter by being clever when we design the VM instruction set:
  1. Combine often used code sequences into one instruction.
     E.g. `muladd a,b,c,d` for $a := b \times c + d$. This will reduce the number of instructions executed, but will make the VM engine larger.
  2. Reduce the total number of instructions, by making them simple and RISC-like. This will increase the number of instructions executed, but will make the VM engine smaller.
- A small VM engine may fit better in the cache than a large one, and hence yield better overall performance.

```c
typedef enum {add,load,...} Inst;
typedef void *Addr;
static Inst prog[]={add,sub,...};

void engine() {
    static Addr jtab[] = {&add,&load,...};
    Inst *pc = &prog;
    int Stack[100]; int sp=0;
    goto *jtab[*pc++];
    add:  Stack[sp-1]+=Stack[sp]; sp--; goto *jtab[*pc++];
    store: Memory[Stack[sp]]=TOS; TOS=Stack[sp-1]; sp-=2;
           goto *jtab[*pc++];
}
```

Indirect Threading...

- We can (sometimes) speed up the interpreter by being clever when we design the VM instruction set:
  1. Combine often used code sequences into one instruction. E.g. `muladd a,b,c,d` for $a := b \times c + d$. This will reduce the number of instructions executed, but will make the VM engine larger.
  2. Reduce the total number of instructions, by making them simple and RISC-like. This will increase the number of instructions executed, but will make the VM engine smaller.
- A small VM engine may fit better in the cache than a large one, and hence yield better overall performance.
**Summary**

- Direct threading is the most efficient dispatch method. It cannot be implemented in ANSI C. Gnu C’s “labels as values” do the trick.
- Indirect threading is almost as fast as direct threading. It may sometimes even be faster, since the interpreted program is smaller and may hence fits better in the cache.
- Call threading is the slowest method. There is overhead from the jump, save/restore of registers, the return, as well as the fact that VM registers have to be global.

**Just-In-Time Compilation**

- Used to be called *Dynamic Compilation* before the marketing department got their hands on it. Also a verb, *jitting*.
- The VM code is compiled to native code just prior to execution. Gives machine independence (the bytecode can be sent over the net) and speed.
- When? When a class/module is loaded? The first time a method/procedure is called? The 2nd time it’s called?

**Readings and References**

- Louden, pp. 4–5.