Switch Threading

Switch Threading

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

```
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.

Bytecode semantics

mnemonic	opcode	stack-pre	stack-post	side-effects
ADD	0	[A,B]	[A+B]	
SUB	1	[A,B]	[A-B]	
MUL	2	[A,B]	[A*B]	
DIV	3	[A,B]	[A-B]	
LOAD X	4	[]	[Memory[X]]	
STORE X	5	[A]	[]	Memory[X] = A
PUSHB X	6	[]	[X]	
PRINT	7	[A]	[]	Print A
PRINTLN	8	[]	[]	Print a newline
EXIT	9	[]	[]	The interpreter exits
PUSHW X	11	[]	[X]	

Bytecode semantics. . .

mnemonic	opcode	stack-pre	stack-post	side-effects
BEQ L	12	[A,B]	[]	if A=B then PC+=L
BNE L	13	[A,B]	[]	if A!=B then PC+=L
BLT L	14	[A,B]	[]	if A<B then PC+=L
BGT L	15	[A,B]	[]	if A>B then PC+=L
BLE L	16	[A,B]	[]	if A<=B then PC+=L
BGE L	17	[A,B]	[]	if A>=B then PC+=L
BRA L	18	[]	[]	PC+=L
ALOAD	19	[X]	[Memory[X]]	
ASTORE	20	[A,X]	[]	Memory[X] = A
SWAP	21	[A,B]	[B,A]	

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: $\langle 6, 10, 7, 8, 9 \rangle$

Example programs…

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

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

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

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: $\langle 6, 10, 5, 7, 6, 10, 4, 7, 2, 7, 8, 9 \rangle$

Example programs. . .

```
# 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[1]++
LOAD 1
ADD
STORE 1
BRA           # goto top of loop
EXIT
```

Bytecode Description

$\boxed{\texttt{ADD}}$: Pop the two top integers A and B off the stack, then push $A + B$.

$\boxed{\texttt{SUB}}$: As above, but push $A - B$.

$\boxed{\texttt{MUL}}$: As above, but push $A * B$.

$\boxed{\texttt{DIV}}$: As above, but push A/B.

$\boxed{\texttt{PUSHB}\ \ X}$: Push X, a signed, byte-size, value, on the stack.

$\boxed{\texttt{PUSHW}\ \ X}$: Push X, a signed, word-size, value, on the stack.

$\boxed{\texttt{PRINT}}$: Pop the top integer off the stack and print it.

$\boxed{\texttt{PRINTLN}}$: Print a newline character.

$\boxed{\texttt{EXIT}}$: Exit the interpreter.

Bytecode Description...

$\boxed{\texttt{LOAD}\ \ X}$: Push the contents of memory cell number X on the stack.

$\boxed{\texttt{STORE}\ \ X}$: Pop the top integer off the stack and store this value in memory cell number X.

$\boxed{\texttt{ALOAD}}$: Pop the address of memory cell number X off the stack and push the value of X.

$\boxed{\texttt{ASTORE}}$: Pop the address of memory cell number X and the value V off the stack and store the V in X.

$\boxed{\texttt{SWAP}}$: Exchange the two top elements on the stack.

Bytecode Description. . .

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

$\boxed{\texttt{BNE}\ L}$: As above, but branch if $A \neq B$.

$\boxed{\texttt{BLT}\ L}$: As above, but branch if $A < B$.

$\boxed{\texttt{BGT}\ L}$: As above, but branch if $A > B$.

$\boxed{\texttt{BLE}\ L}$: As above, but branch if $A \leq B$.

$\boxed{\texttt{BGE}\ L}$: As above, but branch if $A \geq B$.

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

Switch Threading in 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;
         }
       /* Same for SUB, MUL, DIV. */
```

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

case STORE   : {
  memory[prog[pc+1]] = stack[sp-1];
  sp-=1; pc+=2; break;}

case ALOAD    : {
  stack[sp-1] = memory[stack[sp-1]];
  pc++; break;}

 case ASTORE   : {
   memory[stack[sp-1]] = stack[sp-2];
   sp-=2; pc++; break;}
```

```
case SWAP : {
   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; }
```

```
      case PRINTLN: {
        System.out.println(); pc++; break; }

      case EXIT : {return;}

      case BEQ  : {/*Same for BNE,BLT,...*/
        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; }

      default : throw new Exception("Illegal")
}}}}
```

Switch Threading...

- Switch (case) statements are implemented as indirect jumps through an array of label addresses (a *jump-table*). Every switch does 1 range check, 1 table lookup, and 1 jump.

```
switch (e) {
  case 1:  S1; break;
  case 3:  S2; break;
  default:  S3;
}
```

$\Rightarrow$

```
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:
```

Faster Operator Dispatch

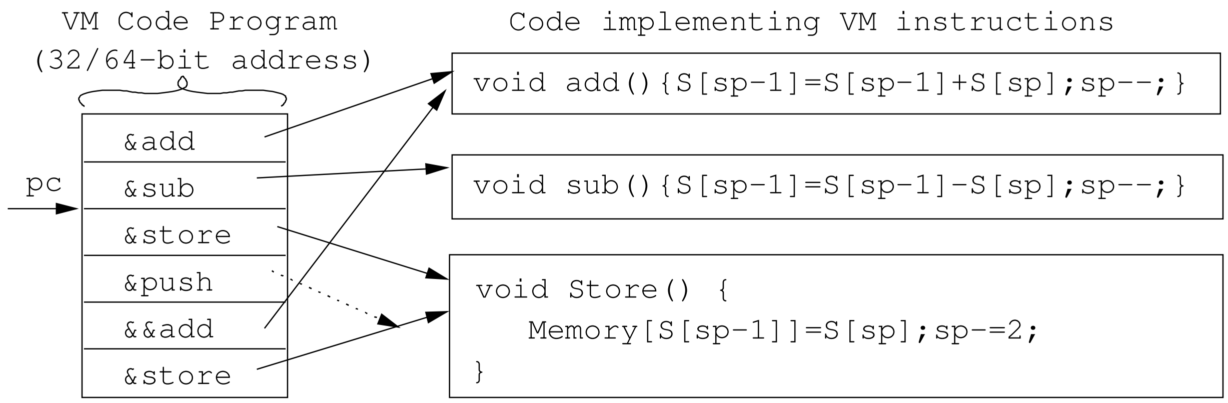
Direct Call Threading

- 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.

Direct Call Threading...

```
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 Call Threading...



Direct Call Threading…

- In direct call threading all instructions are in their own functions.
- This means that VM registers (such as `pc, sp`) must be in global variables.
- So, every time we access `pc` or `sp` we have to load them from global memory. $\Rightarrow$ Slow.
- With the switch method `pc` and `sp` are local variables. Most compilers will keep them in registers. $\Rightarrow$ Faster.
- Also, a direct call threaded program will be large since each instruction is represented as a 32/64-bit address.
- Also, overhead from call/return sequence.

Direct Threading

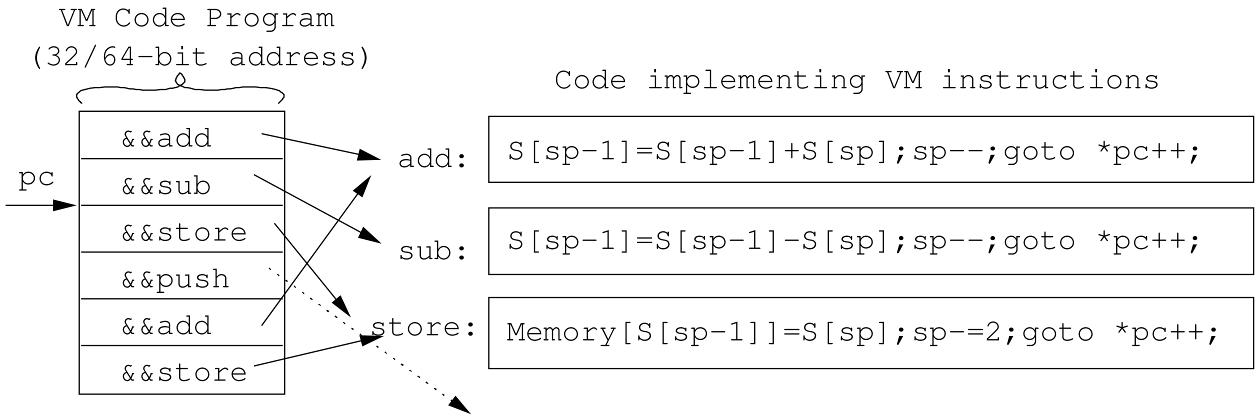
- Each instruction is represented by the address (label) of the code that implements it.
- At the end of each piece of code is an indirect jump `goto *pc++` to the next instruction.
- `"&&"` takes the address of a label. `goto *V` jumps to the label whose address is stored in variable `V`. This is a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

Direct Threading...

```
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++;
}
```

Direct Threading...

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



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`.

Indirect Threading…

```
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++];
}
```

Indirect Threading. . .