# CS410P/510 Programming Language Compilation

## Winter 2024

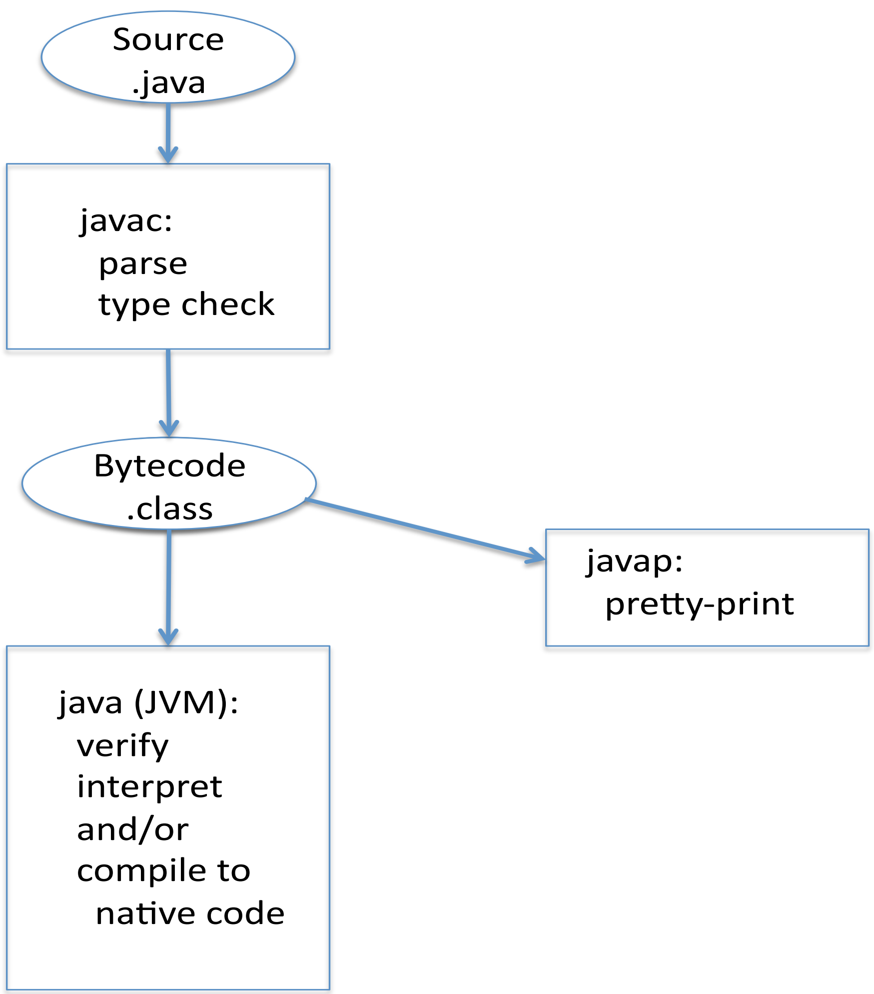
## Lecture on Java Virtual Machine and Interpreters

# Virtual Machines

- Widely used at both language and whole-system level.
- Offer enhanced portability, by abstracting away from specifics of underlying target platform.
- VM code is a well-specified intermediate representation that can be processed in many useful ways:

- transmitted
- interpreted
- compiled
- linked
- verified
- ...

# Java Architecture



# Java Architecture Features

• Mandated separation of front end and back end with precisely specified intermediate code.

• Back end doesn't trust provider of bytecode; hence verification step in JVM.

• Focus on high-speed compilation:

- JIT ("just-in-time") compilers
- mixed interpreter/compiler (eg HotSpot)
- feedback-directed optimization

• Focus on resource-bounded compilation and execution environment.

• Dynamic loading (and even reloading) of class definitions.

# Java Architecture issues

• Except for the need to support dynamic loading, we could dispense with bytecode and JVM, and use standard compiler architecture for Java too; some experimental systems do.

• Bytecode is a relatively high-level IR (can recover source from it), and is better suited to being interpreted than to being optimized. So compiler in JVM often uses lower-level IR.

• We can essentially dispense with front-end and just treat bytecode as source.

• JVM bytecode sometimes used as target for other source languages (e.g. Scala), although not really designed for this purpose.

- Microsoft's .NET explicitly intends its bytecode (CIL) as a multi-language common ground.

# JAVA EXAMPLE: SOURCE CODE

```
class Example {
  public static void main(String [] argv) {
    int i;
    int a = 507;
    for (i = 0 ; i < 10; i++)
      a = (a + i) - f(i * 2);
    System.out.println(a);
  }

  private static int f(int x) {
    return x+42;
  }
}
```

# Bytecode for Example

```
% javac Example.java
% java Example
42
% javap -c -p Example
Compiled from "Example.java"
class Example {
  Example();
    Code:
       0: aload_0
       1: invokespecial #1     // Method java/lang/Object."<init>":()V
       4: return

  private static int f(int);
    Code:
       0: iload_0
       1: bipush        42
       3: iadd
       4: ireturn
```

```
public static void main(java.lang.String[]);
  Code:
     0: sipush        507
     3: istore_2
     4: iconst_0
     5: istore_1
     6: iload_1
     7: bipush        10
     9: if_icmpge     29
    12: iload_2
    13: iload_1
    14: iadd
    15: iload_1
    16: iconst_2
    17: imul
    18: invokestatic  #2      // Method f:(I)I
    21: isub
    22: istore_2
    23: iinc          1, 1
    26: goto          6
    29: getstatic     #3      // Field java/lang/System.out:Ljava/io/PrintStream;
    32: iload_2
    33: invokevirtual #4      // Method java/io/PrintStream.println:(I)V
    36: return
```

# VM ARCHITECTURE: STACKS AND FRAMES

The VM stack consists of a sequence of **frames**; frames need not be contiguous in memory. Frame size and overall stack size may be limited by implementations. (There is actually one stack per VM thread.)

One frame is associated with each method invocation. Each frame contains two areas, each of statically **fixed** size (per method):

- **local variable** storage associated with the method, and
- an **operand stack** for evaluating expressions within the method and for communicating arguments and results with other methods.

The local variable area is an array of words, addressed by word offset from the array base. The arguments to a method (including `this`, for instance methods) always appear as its initial local variables.

The operand stack is a stack of words.

# Types and Verification

The JVM directly supports each of the primitive Java types (except `boolean`, which is mapped to `int`). Floating-point arithmetic follows IEEE 754. Values of reference types (classes,interfaces,arrays) are pointers to heap records, whose layout is implementation-dependent.

Data values are not tagged with type information, but instructions are. When executing, the JVM assumes that instructions are always operating on values of the correct type. The instruction set is designed to make it possible to **verify** that any given method is type-correct, without executing it. The JVM performs verification on any bytecode derived from an untrusted source (e.g., over the network).

At any given point of execution, each entry in the local variable area and the operand stack must have a well-defined **type state**; i.e., it must be possible to deduce the type of each entry unambiguously.

To enforce this property, JVM code must be generated with care. For example, when there are two execution paths to the same PC, they must arrive with identical type state. So, for example, it is impossible to to use a loop to copy an array onto the stack.

# Instruction Set

Each JVM instruction consists of a one-byte **op code** followed by zero or more **parameters**.

The inner loop of the JVM execution engine (ignoring exceptions) is effectively:

```
do {
     fetch opcode;
     if (there are parameters) fetch parameters;
     execute action for opcode;
} while (more to do);
```

Most instructions take their operands from the top of the stack (popping them in the process) and push their result back on the top of the stack. A few operate directly on local variables.

# Instruction Set Organization

Most instructions encode the type of their operands; thus, many instructions have multiple versions distinguished by their prefix (`i,l,f,d,b,s,c,a`).

Instructions group into families. Each family does the same basic operation, but has a variety of members distinguished by operand type and built-in arguments.

The instruction set is not totally orthogonal; in particular, few operations are provided for bytes, shorts, and chars, and integer comparisons are much simpler than non-integer ones. In all, 201 out of 255 possible op-code values are used.

# Example family: push local variable onto stack

*Load 1-word integer from local variable $n$:*

`iload` $n$ $(0 \leq n \leq 255)$
`iload_`$n$ $(0 \leq n \leq 3)$
`wide iload` $n$ $(0 \leq n \leq 65535)$

*Load 2-word long from local variables $n$ and $n+1$:*

`lload` $n$ $(0 \leq n \leq 255)$
`lload_`$n$ $(0 \leq n \leq 3)$
`wide lload` $n$ $(0 \leq n \leq 65535)$

*Load 1-word float from local variables $n$:*

`fload` $n$ $(0 \leq n \leq 255)$
`fload_`$n$ $(0 \leq n \leq 3)$
`wide fload` $n$ $(0 \leq n \leq 65535)$

*Load 2-word double from local variables $n$ and $n+1$:*

`dload` $n$ $(0 \leq n \leq 255)$
`dload_`$n$ $(0 \leq n \leq 3)$
`wide dload` $n$ $(0 \leq n \leq 65535)$

*Load 1-word object reference from local variable $n$:*

`aload` $n$ $(0 \leq n \leq 255)$
`aload_`$n$ $(0 \leq n \leq 3)$
`wide aload` $n$ $(0 \leq n \leq 65535)$

# Families of Operations (1)

## Load and Store

- `load` - push local variable onto stack
- `store` - pop top-of-stack into local variable
- `push`,`ldc`,`const` - push constant onto stack
- `wide` - modify following `load` or `store` to have wider parameter.

## Arithmetic and Logic

- `add`,`sub`,`mul`, `div`, `rem`, `neg`
- `shl`,`shr`, `ushr`
- `or`, `and`, `xor`
- `iinc` - increment local variable

## Conversions

- `i2l`,`i2f`,`i2d`,`l2f`,`l2d`,`f2d`.
- `i2b`,`i2c`,`i2s`, etc. - never raise exception.

# More Operations (2)

## Stack management

- `pop`,`dup`,`dup_x`,`swap`

## Control transfer

- `if_icmpeq`,`if_icmplt`, etc. – compare ints and branch
- `ifeq`,`iflt`, etc. – compare int with zero and branch
- `if_acmpeq`, `if_acmpne` – compare refs and branch
- `ifnull`,`ifnonnull` – compare ref with null and branch
- `cmp` – compare (non-integer) values and push result code (-1,0,1)
- `tableswitch`,`lookupswitch` – for switch statements
- `goto` – target is offset in method code
- `jsr`,`ret` – intended for `finally`
- `athrow` – throw explicit exception

# More Operations (3)

## Objects

- `new` – create new class instance
- `newarray` – creates new array
- `getfield`,`putfield` – access instance variables
- `getstatic`,`putstatic` – access class variables
- `aload`, `astore` – push, pop array elements to,from stack
- `arraylength`
- `instanceof`, `checkcast` – runtime narrowing checks

## Method invocation

- `invokevirtual` – for ordinary instance methods
- `invokeinterface` – for interface methods
- `invokespecial` – for constructor (`<init>`),`private`, or superclass methods
- `invokestatic` – for static methods
- `return`

# Bytecode Summary

JVM Bytecode is intended to be both easy to interpret and easy to use as compiler IR. As an IR, it's fairly high-level (largely for **safety** reasons).

It makes the following **explicit**:

- Parameter and local variable offsets
- Temporaries (using stack)
- Order of evaluation
- Control flow within procedures
- Exceptions

But it leaves the following **implicit**:

- Object layout and field offsets
- Array access
- Method calls (virtual or otherwise)
- Inheritance hierarchy

All these must be resolved *inside* the JVM implementation.

# MAKING INTERPRETERS EFFICIENT

- Many systems (not just Java) use VM's with an explicitly specified binary program representation (conventionally called **bytecode** even if instructions aren't byte-sized).
- Most VM's can execute bytecode directly by **interpretation**.
- Interpretation is typically 1-2 orders of magnitude slower than compilation (but of course this depends on interpreter, compiler, target machine)
- So serious VM's usually do JIT compilation too
- Still, it is worthwhile to make interpreters **efficient**
- But it is also desirable to keep them **portable** (e.g. stick to standard C)

# Costs of Interpretation

Interpreting an instruction requires:

- Dispatching the instruction: getting control to the code corresponding to the instruction
- Accessing the operands: getting the values of the parameters and arguments (and storing the result)
- Actually performing the computation. (Note: the longer this takes, the smaller the percentage overhead of interpretation!)

# Naive Java Interpreter: Sample Instructions

```
u4 stack[STACKSIZE];
void interp (Method *method,u4 *sp) {
  u1 *pc = method->code;
  u4 *locals = sp - method->nargs + 1;
  sp = locals + method->max_locals - 1;
  while (1) {
    switch (*pc) {
    case ICONST_3:   // push the constant 3 onto the operand stack
      {  *(++sp) = 3;
         pc++;
         break;  }
    case ISTORE_1:   // pop the top of the operand stack into local var #1
      {  locals[1] = *(sp--);
         pc++;
         break;  }
    case IADD:      // replace top two elements of stack with their sum
      {  int32_t v2 = (int32_t) (*(sp--));
         int32_t v1 = (int32_t) (*sp);
         *sp = v1 + v2;
         pc++;
         break; }
    ...
    }}}
```

# Operand Access

First, let's consider just the cost of accessing stack elements: loads/stores to memory and `sp` adjustment.

C code:

```
case ICONST_3: { *(++sp) = 3; pc++; break; }
```

X86 (64-bit) machine code (obtained using clang -S)

```
// %rbx holds sp;  %r14 holds pc
movl $3, 4(%rbx)  // *(new sp) = 3
addq $4, %rbx     // new sp = sp + 4
addq $1, %r14     // pc++
jmp top
```

This code is pretty tight, assuming that the stack must be held in memory.

# Operand Access

C code:

```
case ISTORE_1: {  locals[1] = *(sp--);  pc++;  break;  }
```

X86 (64-bit) machine code (obtained using clang -S)

```
// %rbx holds sp; %r14 holds pc; %r13 points to base of locals
movl (%rbx), %eax   // *old_sp
addq $-4, %rbx      // new sp = sp - 4
movl %eax, 4(%r13)  // locals[1] = *old_sp
addq $1, %r14       // pc++
jmp top
```

Again, hard to do better, assuming that locals are held in memory.

# Operand Access

C code:

```
case IADD: { int32_t v2 = (int32_t) (*(sp--));
             int32_t v1 = (int32_t) (*sp);
             *sp = v1 + v2;
             pc++; break; }
```

X86-64 code:

```
// %rbx holds sp; %r14 holds pc
movl (%rbx), %eax     // *sp
addl %eax, -4(%rbx)   // *(new_sp) = *(new_sp) + *sp
addq -4, %rbx         // new_sp = sp - 4
addq $1, %r14         // pc++
jmp top
```

Most obvious problem is that nearly every instruction loads and/or stores stack entries.

# STACK CACHING

Idea: what if we **cache** the top-of-stack in a local variable `s0`?

(Assume that `sp` points to the top of the *remainder* of the stack.)

This saves one load and one store for `IADD`:

```
case IADD: {int32_t v2 = (int32_t) s0;
            int32_t v1 = (int32_t) (*(sp--));
            s0 = v1+v2; pc++; break; }
```

Approximate X86-64 code:

```
// %rbx holds sp (pointer to slot1); %r14 holds pc;
// %r10d holds slot0
movl (%rbx), %eax   // load *sp
addl %eax, %r10d    // slot0 = *sp + slot0
addq -4, %rbx       // new_sp = sp - 4
addq $1, %r14       // pc++
jmp    top
```

# Caching One Slot

But it is a wash for the other two instructions because we have to keep `s0` up-to-date.

```
case ICONST_3: { *(++sp) = s0; s0 = 3; pc++; break; }
```

Approximate X86-64 code (still one stack store)

```
// %rbx holds sp (pointer to slot1) ; %r14 holds pc;
// %r10d holds slot0
addq  $4, %rbx           // new_sp = sp + 4
movl  %r10d, (%rbx)      // *new_sp = slot0
movl  $3, %r10d          // slot0 = 3
addq  $1, %r14           // pc++
jmp   top
```

# CACHING ONE SLOT (CONTINUED)

```
case ISTORE_1: { locals[1] = s0; s0 = *(sp--); pc++; break; }
```

Approximate X86-64 code (still one stack load)

```
// %rbx holds sp (pointer to slot1);
// %r13 points to base of locals
// %r14 holds pc; %r10d holds slot0
movl %r10d, 4(%r13)  // locals[1] = slot0
movl (%rbx), %r10d   // slot0 = *sp
addq -4, %rbx        // sp = sp - 4
addq $1, %r14        // pc++
jmp top
```

# Caching Two Slots

What if we keep **two** elements in local variables (registers) named `s1` (top of stack) and `s0` (next-to-top of stack)?

```
case ISTORE_1: { locals[1] = s1; s1 = s0; s0 = *(sp--);
                 pc++; break;}

case ICONST_3: { *(++sp) = s0; s0 = s1; s1 = 3;
                 pc++; break; }

case IADD: {  s1 = s1+s0; s0 = *(sp--); pc++; break;}
```

This just pushes off the problem: no improvement in number of loads and stores needed.

New idea: let's keep a **different** number of cached stack slots at different points during execution.

# Generalized Stack Caching

• Interpreter operates in one several different **states** corresponding to how many stack slots are cached.

• Each instruction (potentially) causes transition to a different state, according to what it does to the stack.

• For example:

`ICONST_3` moves to a state where *more* slots are cached;

`ISTORE_1` moves to one where *fewer* slots are cached.

`IADD` moves to a state where one slot is cached.

# Generalized Stack Caching (2)

For JVM, 3 states are sufficient to handle all instruction types.

State 0: no slots cached.

State 1: top of stack is cached in variable `s0`.

State 2: top of stack is cached in variable `s1`; next-to-top in `s0`.

In all states, `sp` points to remainder of stack beyond cached slots.

Sample code follows (in practice we may organize it differently)...

```
case IADD: {
  switch (state) {
  case 0: { int32 v2 = (int32) (*(sp--)); int32 v1 = (int32) (*(sp--));
            s0 = (v1+v2); state = 1; break; }
  case 1: { int 32 v2 = (int32) s0; int32 v1 = (int32) (*(sp--));
            s0 = (v1+v2); state = 1; break; }
  case 2: { int 32 v2 = (int32) s1; int32 v1 = (int32) s0;
            s0 = (v1+v2); state = 1; break; }
  pc++; break; }

case ICONST_3: {
  switch (state) {
  case 0: s0 = 3; state = 1; break;
  case 1: s1 = 3; state = 2; break;
  case 2: *(++sp) = s0; s0 = s1; s1 = 3; state = 2; break; }
  pc++; break; }

case ISTORE_1: {
  switch (state) {
  case 0: locals[1] = *(sp--); state = 0; break;
  case 1: locals[1] = s0; state = 0; break;
  case 2: locals[1] = s1; state = 1; break; }
  pc++; break; }
```

# Example Sequence

Consider a typical expression like

```
b = a + 3
```

where we assume `a` is local variable 0 and `b` is local variable 1.

(Assume we start with state = 0.)

```
Bytecode        Corresponding executed code

ILOAD_0         s0 = locals[0]; state = 1;
ICONST_3        s1 = 3; state = 2;
IADD            s0 = s1 + s0; state = 1;
ISTORE_1        locals[1] = s0; state = 0;
```

We do only the essential loads and stores – no stack traffic at all!

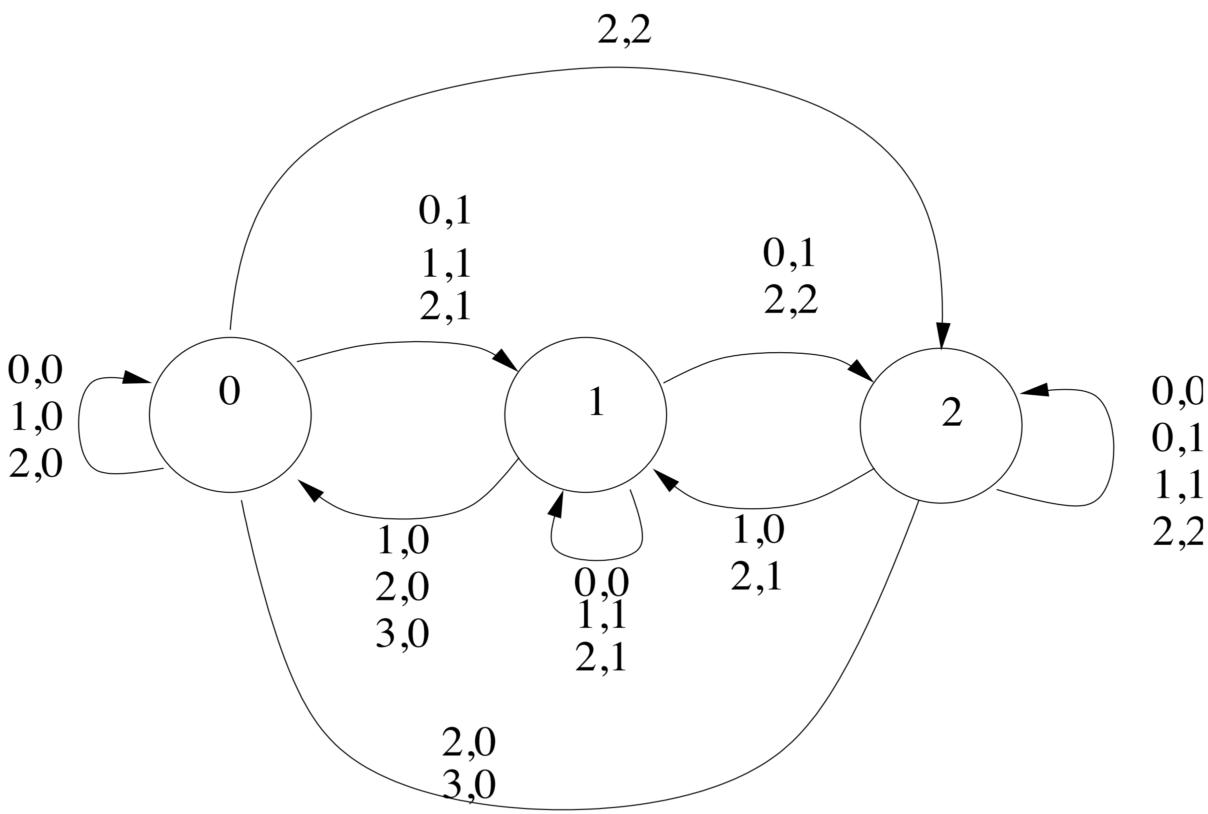
# 3-State Transition Diagram

More generally, instructions are classified by a pair:

(# of stack slots they consume, # of stack slots they produce)

For example:

```
ISTORE_0    1,0
ICONST_0    0,1
IADD        2,1
```



# DYNAMIC VS. STATIC CACHING

So far we've described **dynamic** stack caching, where the interpreter keeps track of its current state.

• In practice, we implement this by having three complete sets of instruction implementations and dispatching to the correct one based on current state as well as opcode (more on this later).

• But it may seem like we should be able to predict the state at each program point **statically** (before execution). If so, we could simply have three variants of each opcode, and select the right one at compile time. This would be more efficient.

• Only problem: at **join points** in the code, the state may differ depending on the path by which the join point was reached. Must choose a convention for which state to use there, and add **compensation code** to the other branches; this is complex in practice.

# Aside: Why use stack-based VM's?

Nearly all hardware processors use **registers**

• Each HW instruction is parameterized by its argument/result registers.

• Why is this good for hardware? Because the opcode and the argument registers can be decoded in **parallel**, and values can quickly be fetched from a small, fast register file.

Why not try this in software machines too?

• Parameters must be fetched from the byte stream and decoded **serially**; for stack instructions, parameters are implicit.

• Instructions with parameters take more space.

• Software registers cannot easily be stored in hardware registers, because the latter can't be indexed. So software registers end up living in an in-memory array (just like stack slots).

• On the other hand, register architectures require fewer instructions; hence less **dispatch**. So maybe a worthwhile idea after all...

# Speeding Up Instruction Dispatch

What does X86-64 code look like now?

```
// %r12 holds table; %r14 holds pc
top: movzbl (%r14), %rax         // fetch opcode at pc
     cmpq $tablesize, %rax       // compare against jump table size
     ja  undefined               // if out of range branch to "undefined"
     movslq (%r12,%rax,4), %rax // get table entry=snippet address-table base
     addq %r12,%rax              // add to table base
     jmpq *%rax                  // jump to snippet
table:
     .word nop_snippet-table
     .word aconst_null_snippet-table
     .word iconst_m1_snippet-table
     .word iconst_0_snippet-table
     ...
     .word goto_w_snippet-table
undefined:
     ...issue error and die...
```

# THEADED CODE

Obvious performance problems:

- Unnecessary bounds check.
- Two jumps per dispatch (counting the one back to `top` at the end of the previous instruction).

First fix: **(Indirect) Threaded Code**

If we can code our own indirect jumps, could

- Remove bounds check.
- Replicate dispatch at end of every snippet, thus removing one jump.
- This is not possible in ANSI Standard C, but can do in `gcc` using the `&&` operator.

# Indirect Threaded Code

```
interp(Method method) {
  static void *dispatch_table[] =
  {&&NOP,
   &&ACONST_NULL,
   &&ICONST_M1,
   ...,
   &&JSR_W };
   char *pc = method->code;
   ...
   goto *(dispatch_table[*pc]);

  NOP:
   pc++;
   goto *(dispatch_table[*pc]);

  ACONST_NULL:
   *(++sp) = (u4) 0;
   pc++;
   goto *(dispatch_table[*pc]);
  ...
}
```

# Direct Threading

Each instruction dispatch still requires two fetches: one to get the byte code and a second to get the snippet address.

New idea: what if we represent each instruction opcode by the address of its snippet?

```
interp() {
  char *codeaddrs[] = ...;  /* fill this with snippet addrs */
  char *pc = codeaddrs;     /* initialize to start */
  goto **pc;

  ACONST_NULL:
    *(++sp) = (u4) 0;
    pc++;
    goto **pc;
  ...
```

Now need only one fetch per instruction!

# Rewriting Byte Code

But notice that we're no longer interpreting the original bytecode any more.

Must rewrite before execution

Simple in principle, but there are details. e.g.

- What should we do with the parameter bytes following the opcode?

If we're going to rewrite the bytecode, there are **many** opportunities to improve things, e.g.

- Combine code for similar opcodes (e.g. constant loading).
- Short-circuit constant pool references (important in full language)
- Perform static stack caching
- Etc, etc.

A more radical rewrite idea: dispatch to each snippet using a **subroutine** call instruction. May pay off on processors that pre-fetch from the return address on the hardware stack!

# REDUCING DISPATCHES

Another way to reduce dispatch time is to do **fewer** dispatches.

One basic approach is to **combine** sequences of instructions that occur frequencly into into “macro” or “super”-instructions.

For example, the following sequence pattern is very common:

```
ILOAD n
ICONST i
IADD
ISTORE n
```

In fact, the JVM designers already invented a combined instruction for this (`IINC`) but the same idea works for other sequences.

Another approach is to use a **register** architecture, which typically requires many fewer instructions (although each instruction gets more parameters).

# BUILDING COMBINED INSTRUCTIONS

This can be done in several ways:

• Statically, for multiple programs:

- Essentially a refinement of the VM definition, possibly tuned to workload from a particular set of programs.

- Can construct such specialized VM's semi-automatically from a generic VM.

- Specialized VM can be compiled with "cross-snippet" optimization.

• Statically, for a single program

- Encoding is sent with the program.

Static encodings also have the benefit of reducing the program size, allowing quicker transmission.

• Dynamically, by building superinstructions "on the fly" from snippet code.

- This is beginning to resemble a compiler!