**CPSC 213**

**Introduction to Computer Systems**

**Unit 1b**

**Static Scalars and Arrays**

### Examining Java and C Bit by Bit

- Reading writing and arithmetic on Variables
  - static base types (e.g., int, char)
  - static and dynamic arrays of base types
  - dynamically allocated objects and object references
  - object instance variables
  - procedure locals and arguments
- Control flow
  - static intra-procedure control flow (e.g., if, for, while)
  - static procedure calls
  - dynamic control flow and polymorphic dispatch

### Design Tasks

- Design Instructions for SM213 ISA
  - design instructions necessary to implement the languages
  - keep hardware simple/fast by adding few/simple instructions possible
- Design Compilation Strategy
  - determine how compiler will compile each language feature it sees
  - which instructions will it use?
  - in what order?
- Consider Static and Dynamic Phases of Computation
  - the static phase of compilation (compilation) happens just once
  - the dynamic phase (running the program) happens many times
  - thus anything the compiler computes, saves execution time later

### The Simple Machine (SM213) ISA

- Architecture
  - Register File
    8, 32-bit general purpose registers
  - CPU
    a cycle per instruction (fetch + execute)
  - Main Memory
    byte addressed, Big Endian integers

### Machine and Assembly Syntax

- Machine code
  - `addr` sets starting address for subsequent instructions
  - `r0` hex value of instruction with opcode x and operands 0 and 1
  - `v vv vvvvvv vv` hex value of optional extended value part instruction
- Assembly code
  - `label [ | directive | [comment] ]` (includes i.e., `::`
    - directive:
      - `(sex number | C long number)`
      - `instruction :: opeand`
    - operand:
      - `S I (reg | offset) (reg,reg,4)`
      - `reg`
      - `r [0-7]`
      - `literal`
      - `number`
      - `number` decimal | 0 hex

### Static Variable Allocation

- Static Variables of Built-In Types
  - Java
    - static data members are allocated to a class, not an object
  - they can store built-in scalar types or references to arrays or objects (references later)
  - `private`
  - `public`
  - `protected`
  - `static`
  - `volatile`
- Static Variable Access
  - (static)
    - `m[0x1000 + r[d]*4] ← r[s]` and `r[s] ← m[0x1000 + r[d]*4]`
    - `m[r[x] + r[y]*4] ← r[z]` and `r[z] ← m[r[x] + r[y]*4]`

### Designing ISA for Static Variables

- Requirements for scalars
  - `a = 0;
  - load constant into register` `r0`
  - `store value in register into memory at constant address`
  - `m[r[x]] ← r[s]`
  - `load value in memory at constant address into a register` `r[x] ← m[r[x]]`
- Additional requirements for arrays
  - `m[b[a]] = a;
  - load value in register into memory at address in register A plus constant`
  - `m[r[x]+r[y]*4] ← r[z]`
    - `load value in memory at address in register A plus constant into register` `r[x] ← m[r[x]+r[y]*4]`
    - `Generalizing and simplifying we get`
      - `r[x] ← m[r[x]+r[y]*4]` and `r[x] ← m[r[x]+r[y]*4]`

### Static Memory Layout

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>a = 0</td>
</tr>
<tr>
<td>0x2000</td>
<td>b[a] = a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>a = 0</td>
</tr>
<tr>
<td>0x2000</td>
<td>b[a] = a</td>
</tr>
</tbody>
</table>

```

### The Big Picture

- Build model of execution
  - for Java and C programs
  - by examining language features
  - and deciding how they are implemented by the machine
- What is required
  - design an ISA into which programs can be compiled
  - implement the ISA in the hardware simulator
- Our approach
  - examine code snippets that exemplify each language feature in turn
  - look at Java and C, pausing to dig deeper when C is different from Java
  - design and implement ISA as needed
- The simulator is an important tool
  - machine execution is hard to visualize without it
  - this visualization is really our WHOLE POINT here
The compiler's assembly translation

```c
int a;
int b[10];
void foo () {
  a = 0;
  b[a] = a;
}
```

ld $0, r0
ld ...

```
r2 ← m[r1]
r3 ← 0x2000
m[r3+r2*4] ← r2
```

### Example

Example: The compiler's assembly translation

```
void foo () {
  a = 0;
  b[a] = a;
}
```

ld $0, r0
ld ...

```
r2 ← m[r1]
r3 ← 0x2000
m[r3+r2*4] ← r2
```

### Global Dynamic Array

#### Java

- An array variable stores reference to array allocated dynamically with new statement

```
public class a {
  static int b[] = new int[10];
  ...
}
```

#### C

- An array can store static or dynamic contents
- Pointer to a dynamic array allocated with malloc at run-time

```
int a[] = malloc(10*sizeof(int));
```

#### In assembly language

```
Static Array
Dynamic Array
```

### Comparing static and dynamic arrays

- What is the benefit of static arrays?
- What is the benefit of dynamic arrays?

### Pointers in C

#### Its purpose

- An alternative way to access dynamic arrays to the address of an integer
- Results in a new pointer of the same type
- Value of the pointer is offset by index times size of pointer's elements
- For example, asking in an array yields a pointer value 10 bigger than the original

#### Subtracting two pointers of the same type

- Results in an integer
- Gives number of elements between the two pointers
- For example:
  - \( 0x12345678 - 0x02345678 = 0x12345678 - 0x02345678 \)

#### Other operators

- \( *x \)
  - The address of \( x \)
- \( x * \)
  - The value \( x \) points to

#### Addressing Modes

In these instructions:

- Immediate
  - Constant value stored in instruction
- Register
  - Operand is register number, register stores value
- Base-offset
  - Operand is register number, register stores memory address of value (+/offset)
- Indexed
  - Two register number operands; store base memory address and index of value

### How C Arrays are Different from Java

#### Terminology

- Use the term `pointer` instead of `reference`; they mean the same thing

#### Declaration

- The type is a pointer to the type of its elements, indicated with a *

#### Allocation

- malloc allocates a block of bytes; no type; no constructor

#### Type Safety

- Any pointer can be type cast to any pointer type

#### Bounds checking

- C performs no array bounds checking
- Out-of-bounds access manipulates memory that is not part of array
- This is the major source of virus vulnerabilities in the world today

#### Question: Can array bounds checking be performed statically?

- What does this say about a tradeoff that Java and C take different?

### Static vs Dynamic Arrays

- Declared and allocated differently, but accessed the same
- For static arrays, the compiler allocates the array
- For dynamic arrays, the compiler allocates a pointer

#### Allocation

- malloc allocates a block of bytes; no type; no constructor
- For example:
  - \( a \) is `malloc(10*sizeof(int))`
- Dynamic arrays are just like all other pointers
  - Can store or access with either `a[0]` or `*(a+x)

### Example

The following two C programs are identical

- For array access, the compiler would generate this code

```
for (int i = 0; i < 10; i++) {
  b[i] = i;
}
```

- Multiplying the index 4 by 4 (size of integer) to compute the array offset

### C and Java Arrays and Pointers

#### In both languages

- An array is a list of elements of the same type
- Array elements are named by non-negative integers that start with 0
- Syntax for accessing element i of array a is `a[i]`

#### In Java

- Variable `a` stores a reference to the array
- `a[0] = 0` means `m[0] = x * sizeof(array-element)`
- In C
  - Variable `a` can store a pointer to the array or the array itself
  - `a[0] = 0` means `m[0] = x * sizeof(array-element)`

#### Dynamic arrays

- Dynamic arrays are just like all other pointers
- Can store or access with either `a[0]` or `*(a+x)`
- Can store or access with either `a[0]` or `*(a+x)`
And in assembly language

```assembly
r[0] ← 0x2000  # r[0] = &c
r[1] ← m[r[0]]  # r[1] = c
m[r[0]] ← r[2]  # c = c + 3
m[r[2]] ← r[4]  # c[0] = c[3]
ld $0x2000, r0  # r0 = &c
ld (r0), r1  # r1 = c
ld $12, r2  # r2 = 3*sizeof(int)
add r1, r2  # r2 = c + 3
st r2, (r0)  # c = c + 3
ld $3, r3  # r3 = 3
ld (r2, r3, 4), r4  # r4 = c[3]
st r4, (r2)  # c[0] = c[3]
```

Summary: Static Scalar and Array Variables

- **Static variables**
  - the compiler knows the address (memory location) of variable

- **Static scalars and arrays**
  - the compiler knows the address of the scalar value or array

- **Dynamic arrays**
  - the compiler does not know the address the array

- **What C does that Java doesn’t**
  - static arrays
  - arrays can be accessed using pointer dereferencing operator
  - arithmetic on pointers

- **What Java does that C doesn’t**
  - typesafe dynamic allocation
  - automatic array-bounds checking