The Big Picture

- Build machine 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
Examine 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 as few/simple instructions possible

- Develop Compilation Strategy
  - determine how compiler will compile each language feature it sees
  - which instructions will it use?
  - in what order?
  - what can compiler compute statically?

- Consider Static and Dynamic Phases of Computation
  - the static phase of computation (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: one cycle per instruction (fetch + execute)
  - Main Memory: byte addressed, Big Endian integers

- Instruction Format
  - 2 or 6 byte instructions (each character is a hexit)
    - x-01, xx-01, x0vv or x-01 vvvvvvv
  - where
    - x is opcode (unique identifier for this instruction)
    - 0 means unused
    - 0 and 1 are operands
    - vv vvvvvv are immediate / constant values

Machine and Assembly Syntax

- Machine code
  - [addr:] x-01 [vvvvvv]
    - addr: sets starting address for subsequent instructions
    - x-01 hex value of instruction with opcode x and operands 0 and 1
    - vvvvvv hex value of optional extended value part instruction

- Assembly code
  - ([label:] [instruction | directive] [# comment] | )*
    - directive :: (.pos number) | (.long number)
    - instruction :: opcode operand+
    - operand :: $literal | reg | offset (reg) | (reg,reg,4)
    - reg :: r 0..7
    - literal :: number
    - offset :: number
    - number :: decimal | 0x hex
Register Transfer Language (RTL)

- **Goal**
  - a simple, convenient pseudo language to describe instruction semantics
  - easy to read and write, directly translated to machine steps

- **Syntax**
  - each line is of the form LHS ← RHS
  - LHS is memory or register specification
  - RHS is constant, memory, or arithmetic expression on two registers

- **Register and Memory are treated as arrays**
  - m[a] is memory location at address a
  - r[i] is register number i

- **For example**
  - r[0] ← 10
  - r[1] ← m[r[0]]
  - r[2] ← r[0] + r[1]

Static Variables, Built-In Types (S1-global-static)

- **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)

```java
public class Foo {
    static int a;
    static int[] b;  // array is not static, so skip for now

    public void foo () {
        a  = 0;
    }
}
```

- **C**
  - global variables and any other variable declared static
  - they can be static scalars, arrays or structs or pointers (pointers later)

```c
int a;
int b[10];

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

Static Variable Allocation

- **Allocation is**
  - assigning a memory location to store variable's value
  - assigning the variable an address (its name for reading and writing)

- **Key observation**
  - global/static variables can exist before program starts and live until after it finishes

- **Static vs dynamic computation**
  - compiler allocates variables, giving them a constant address
  - no dynamic computation required to allocate the variables, they just exist

Static Memory Layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>value of a</td>
</tr>
<tr>
<td>0x2000</td>
<td>value of b[0]</td>
</tr>
<tr>
<td>0x2004</td>
<td>value of b[1]</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x2020</td>
<td>value of b[9]</td>
</tr>
</tbody>
</table>
**Designing ISA for Static Variables**

- **Requirements for scalars**
  - load constant into register
    - \( r[x] \leftarrow v \)
  - store value in register into memory at constant address
    - \( m[0x1000] \leftarrow r[x] \)
  - load value in memory at constant address into a register
    - \( r[x] \leftarrow m[0x1000] \)

- **Additional requirements for arrays**
  - store value in register into memory at address in register*4 plus constant
    - \( m[0x2000+4\times r[x]+4] \leftarrow r[y] \)
  - load value in memory at address in register*4 plus constant into register
    - \( r[y] \leftarrow m[0x2000+4\times r[x]+4] \)

- **Generalizing and simplifying we get**
  - \( r[x] \leftarrow constant \)
  - \( m[r[x]] \leftarrow r[y] \) and \( r[y] \leftarrow m[r[x]] \)
  - \( m[r[x]+r[y]\times 4] \leftarrow r[z] \) and \( r[z] \leftarrow m[r[x]+r[y]\times 4] \)

**Key Observation**
- address of \( a, b[0], b[1], b[2], ... \) are constants known to the compiler
- Use RTL to specify instructions needed for \( a = 0 \)

**Generalizing**
- What if it’s \( a = a + 2 \) or \( a = b \) or \( a = foo() \)?
- What about reading the value of \( a \)?

**The compiler’s semantic translation**
- it uses these instructions to compile the program snippet

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>load immediate</td>
<td>( r[d] \leftarrow v )</td>
<td><code>ld $v, rd</code></td>
<td>0d- vvvvvvv</td>
</tr>
<tr>
<td>load base-offset</td>
<td>( r[d] \leftarrow m[r[1]] )</td>
<td><code>ld $(r[0], r[1])</code></td>
<td>1sd</td>
</tr>
<tr>
<td>load indexed</td>
<td>( r[d] \leftarrow m[r[1]+4\times r[i]] )</td>
<td><code>ld (r[i], r[1], r[0])</code></td>
<td>2sid</td>
</tr>
<tr>
<td>store base+offset</td>
<td>( m[r[d]] \leftarrow r[s] )</td>
<td><code>st r[d], $r[s]</code></td>
<td>3sidd</td>
</tr>
<tr>
<td>store indexed</td>
<td>( m[r[d]+4\times r[i]] \leftarrow r[s] )</td>
<td><code>st r[s], (r[i], r[0], r[1])</code></td>
<td>4sdi</td>
</tr>
</tbody>
</table>

**Static Memory Layout**
- value of \( a \) at \( 0x1000 \)
- value of \( b \) at \( 0x2000, 0x2004, ..., 0x2020 \)

**Static Variable Access (static arrays)**
- compiler does not know address of \( b[a] \)
  - unless it can know the value of \( a \) statically, which it can here by looking at \( a = 0 \), but not in general
- Use RTL to specify instructions for \( b[a] = a \), not knowing \( a \)?
Addressing Modes

- In these instructions
  - **load immediate**: \( r[d] \leftarrow v \)\n    - Assembly: `ld $v, rd`\n    - Machine: `0d-- vvvvvvv`
  - **load base+offset**: \( r[d] \leftarrow m[r[s]] \)\n    - Assembly: `ld $r(s), rd`\n    - Machine: `1?sd`
  - **load indexed**: \( r[d] \leftarrow m[r[s]+4*r[i]] \)\n    - Assembly: `ld (r,s,i,4), rd`\n    - Machine: `2sdi`
  - **store base+offset**: \( m[r[d]+4*r[i]] \leftarrow r[s] \)\n    - Assembly: `st r[s], ?(rd)`\n    - Machine: `3s?d`
  - **store indexed**: \( m[r[d]+4*r[i]] \leftarrow r[s] \)\n    - Assembly: `st r[s], (rd,i,4)`\n    - Machine: `4sdi`

- We have specified 4 *addressing modes* for operands
  - **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

Basic Arithmetic, Shifting, NOP and Halt

- **Arithmetic**
  - **register move**: \( r[d] \leftarrow r[s] \)\n    - Assembly: `mov rs, rd`
    - Machine: `60sd`
  - **add**: \( r[d] \leftarrow r[d] + r[s] \)\n    - Assembly: `add rs, rd`
    - Machine: `61sd`
  - **and**: \( r[d] \leftarrow r[d] & r[s] \)\n    - Assembly: `and rs, rd`
    - Machine: `62sd`
  - **inc**: \( r[d] \leftarrow r[d] + 1 \)\n    - Assembly: `inc rd`
    - Machine: `63-d`
  - **dec**: \( r[d] \leftarrow r[d] - 1 \)\n    - Assembly: `dec rd`
    - Machine: `65-d`
  - **not**: \( r[d] \leftarrow \sim r[d] \)\n    - Assembly: `not rd`
    - Machine: `67-d`

- **Shifting, NOP and Halt**
  - **shift left**: \( r[d] \leftarrow r[d] << S = s \)\n    - Assembly: `shl rd, s`
    - Machine: `7dSs`
  - **shift right**: \( r[d] \leftarrow r[d] >> S = s \)\n    - Assembly: `shr rd, s`
    - Machine: `7dSs`
  - **halt**: halt machine\n    - Assembly: `halt`
    - Machine: `f0--`
  - **nop**: do nothing\n    - Assembly: `nop`
    - Machine: `ff--`
Global Dynamic Array

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 perform statically?*
*what does this say about a tradeoff that Java and C take differently?*

Static vs Dynamic Arrays

- **Declared and allocated differently, but accessed the same**

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

  ```
  int a;
  int* b;
  void foo () {
    b = (int*) malloc (10*sizeof(int));
    b[a] = a;
  }
  ```

- **Static allocation**
  - for static arrays, the compiler allocates the array
  - for dynamic arrays, the compiler allocates a pointer

  ```
  0x2000: value of b[0]
  0x2004: value of b[1]
  ...
  0x2024: value of b[9]
  ```
• Then when the program runs
  • the dynamic array is allocated by a call to malloc, say at address 0x3000
  • the value of variable b is set to the memory address of this array

- Generating code to access the array
  • for the dynamic array, the compiler generates an additional load for b

- In assembly language

  **Static Array**
  ```
  ld $a_data, r0       # r0 = address of a
  ld (r0), r1         # r1 = a
  ld $b_data, r2      # r2 = address of b
  st r1, (r2,r1,4)    # b[a] = a
  ```

  **Dynamic Array**
  ```
  ld $a_data, r0       # r0 = address of a
  ld (r0), r1         # r1 = a
  ld $b_data, r2      # r2 = address of b
  st r1, (r2,r1,4)    # b[a] = a
  ```

- Comparing static and dynamic arrays
  • what is the benefit of static arrays?
  • what is the benefit of dynamic arrays?

**C and Java Arrays and Pointers**

- In both languages
  • an array is a list of items of the same type
  • array elements are named by non-negative integers start with 0
  • syntax for accessing element i of array b is b[i]

- In Java
  • variable a stores a pointer to the array
  • b[x] = 0 means m[m[b] + x * sizeof(array-element)] ← 0

- In C
  • variable a can store a pointer to the array or the array itself
  • b[x] = 0 means m[b + x * sizeof(array-element)] ← 0
  • dynamic arrays are just like all other pointers
    - stored in TYPE*
    - access with either a[x] or *(a+x)
The following two C programs are identical:

```
int *a;
a[4] = 5;
```

```
int *a;
*(a+4) = 5;
```

For array access, the compiler would generate this code:

```
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]
```

Adding X to a pointer of type Y*, adds X * sizeof(Y) to the pointer's memory-address value.

Question (from S3-C-pointer-math.c)

```
int *c;
void foo () {
  // ...
  c = (int *) malloc (10*sizeof(int));
  // ...
  c = &c[3];
  *c = *&c[3];
  // ...
}
```

What is the equivalent Java statement to

- [b] c[0] = c[3];
- [g] c[3] = c[6];
- [r] there is no typesafe equivalent
- [y] not valid, because you can't take the address of a static in Java

Looking more closely

```
c = &c[3];
*c = *&c[3];
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]
```

Before

```
0x2000: 0x3000
0x3000: 0
0x3000: 1
0x3000: 2
0x3000: 3
0x3000: 4
0x3000: 5
0x3000: 6
0x3000: 7
0x3000: 8
```

After

```
0x2000: 0x3000
0x3000: 0
0x3000: 1
0x3000: 2
0x3000: 3
0x3000: 4
0x3000: 5
0x3000: 6
0x3000: 7
0x3000: 8
```

Example

So, what does this tell you about pointer arithmetic in C?

Adding X to a pointer of type Y*, adds X * sizeof(Y) to the pointer's memory-address value.
And in assembly language

```
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]
```

### 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