CPSC 213

Introduction to Computer Systems

Unit 1b

Static Scalars and Arrays
Reading

- Companion
  - 2.2.3, 2.3, 2.4.1-2.4.3, 2.6

- Textbook
  - *Array Allocation and Access*
    - 1ed: 3.8
    - 2ed: 3.8
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
Design Plan
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 hex digit)
  - x−sd, xsd−, xxsd, xsvv, xxvs, or xs-- vvvvvvvv
- where
  - x or xx is opcode (unique identifier for this instruction)
  - − means unused
  - s and d are operands (registers), sometimes left blank with −
  - vv and vvvvvvvv are immediate / constant values
Machine and Assembly Syntax

- **Machine code**
  - `[ addr: ] x–01 [ vvvvvvvv ]`
    - `addr:` sets starting address for subsequent instructions
    - `x–01` hex value of instruction with opcode x and operands 0 and 1
    - `vvvvvvv` 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 \leftarrow 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] \leftarrow 10
- r[1] \leftarrow m[r[0]]
- r[2] \leftarrow r[0] + r[1]
Implementing the ISA
The CPU Implementation

- Internal state
  - pc: address of next instruction to fetch
  - instruction: the value of the current instruction
    - insOpCode
    - insOp0
    - insOp1
    - insOp2
    - insOpImm
    - insOpExt

- Operation
  - fetch
    - read instruction at pc from memory, determine its size and read all of it
    - separate the components of the instruction into sub-registers
    - set pc to store address of next instruction, sequentially
  - execute
    - use insOpCode to select operation to perform
    - read internal state, memory, and/or register file
    - update memory, register file and/or pc
Static Variables of Built-In Types
Static Variables, 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)

  ```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;
  }
  ```
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 Variable Access (scalars)

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

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

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`?

Static Memory Layout

- 0x1000: value of `a`
- 0x2000: value of `b[0]`
- 0x2004: value of `b[1]`
- ...
- 0x2024: value of `b[9]`
When is space for \texttt{a} allocated (when is its address determined)?

- [A] The program locates available space for \texttt{a} when program starts
- [B] The compiler assigns the address when it compiles the program
- [C] The compiler calls the memory to allocate \texttt{a} when it compiles the program
- [D] The compiler generates code to allocate \texttt{a} before the program starts running
- [E] The program locates available space for \texttt{a} when the program starts running
- [F] The program locates available space for \texttt{a} just before calling \texttt{foo()}

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

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

- compiler does not know address of \texttt{b[a]}
  - unless it can knows the value of \texttt{a} statically, which it could here by looking at \texttt{a=0}, but not in general

Array access is computed from base and index

- address of element is \texttt{base} plus \texttt{offset}; \texttt{offset} is \texttt{index} times element size
- the base address (0x2000) and element size (4) are static, the index is dynamic

Use RTL to specify instructions for \texttt{b[a] = a}, not knowing \texttt{a}?
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
  - \( m[0x2000+r[x]*4] \leftarrow r[y] \)
- load value in memory at address in register*4 plus constant into register
  - \( r[y] \leftarrow m[0x2000+r[x]*4] \)

Generalizing and simplifying we get

- \( r[x] \leftarrow \text{constant} \)
- \( m[r[x]] \leftarrow r[y] \) and \( r[y] \leftarrow m[r[x]] \)
- \( m[r[x] + r[y]*4] \leftarrow r[z] \) and \( r[z] \leftarrow m[r[x] + r[y]*4] \)
The compiler’s semantic translation

- it uses these instructions to compile the program snippet

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

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

ISA Specification for these 5 instructions

<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] ← v</td>
<td>ld $v, rd</td>
<td>0d-- vvvvvvvv</td>
</tr>
<tr>
<td>load base+offset</td>
<td>r[d] ← m[r[s]]</td>
<td>ld ?(r[s]), rd</td>
<td>1?srd</td>
</tr>
<tr>
<td>load indexed</td>
<td>r[d] ← m[r[s]]+4*r[i]</td>
<td>ld (r[s],ri,4), rd</td>
<td>2sid</td>
</tr>
<tr>
<td>store base+offset</td>
<td>m[r[d]] ← r[s]</td>
<td>st rs, ?(rd)</td>
<td>3s?d</td>
</tr>
<tr>
<td>store indexed</td>
<td>m[r[d]]+4*r[i] ← r[s]</td>
<td>st rs, (rd,ri,4)</td>
<td>4sdi</td>
</tr>
</tbody>
</table>
The compiler’s assembly translation

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

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

```assembly
r[0] ← 0
r[1] ← 0x1000
m[r[1]] ← r[0]

r[2] ← m[r[1]]
r[3] ← 0x2000
```

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

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

```assembly
ld $0, r0
ld $0x1000, r1
st r0, (r1)

ld (r1), r2
ld $0x2000, r3
st r2, (r3,r2,4)
```
If a human wrote this assembly
• list static allocations, use labels for addresses, add comments

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

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

```assembly
ld $0, r0    # r0 = 0
ld $a_data, r1    # r1 = address of a
st r0, (r1)    # a = 0

ld (r1), r2    # r2 = a
ld $b_data, r3    # r3 = address of b
st r2, (r3,r2,4)    # b[a] = a
```

```assembly pos 0x1000
a_data:
.long 0    # the variable a
```

```assembly pos 0x2000
b_data:
.long 0    # the variable b[0]
.long 0    # the variable b[1]
...    
.long 0    # the variable b[9]
```
In these instructions

- **Immediate**
  - Constant value stored in instruction

- **Register**
  - Operand is register number, register stores value

- **Base+offset**
  - Operand in register number
  - Register stores memory address of value

- **Indexed**
  - Two register-number operands
  - Store base memory address and index of value

---

### Addressing Modes

<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>(ld \ $v, r_d)</td>
<td>(0\text{d-- vvvvvvvv})</td>
</tr>
<tr>
<td>load base+offset</td>
<td>[r[d]] (\leftarrow m[r[s]])</td>
<td>(ld \ ?(r_s), r_d)</td>
<td>(1\text{s}d)</td>
</tr>
<tr>
<td>load indexed</td>
<td>[r[d]] (\leftarrow m[r[s]+4\times r[i]])</td>
<td>(ld \ (r_s,ri,4), r_d)</td>
<td>(2\text{sid})</td>
</tr>
<tr>
<td>store base+offset</td>
<td>(m[r[d]] \leftarrow r[s])</td>
<td>(st \ r_s, ?(r_d))</td>
<td>(3\text{s}d)</td>
</tr>
<tr>
<td>store indexed</td>
<td>(m[r[d]+4\times r[i]] \leftarrow r[s])</td>
<td>(st \ r_s, (r_d,ri,4))</td>
<td>(4\text{sdi})</td>
</tr>
</tbody>
</table>
Basic Arithmetic, Shifting NOP and Halt

### Arithmetic

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>register move</td>
<td><code>r[d] ← r[s]</code></td>
<td>mov rs, rd</td>
<td>60sd</td>
</tr>
<tr>
<td>add</td>
<td><code>r[d] ← r[d] + r[s]</code></td>
<td>add rs, rd</td>
<td>61sd</td>
</tr>
<tr>
<td>and</td>
<td><code>r[d] ← r[d] &amp; r[s]</code></td>
<td>and rs, rd</td>
<td>62sd</td>
</tr>
<tr>
<td>inc</td>
<td><code>r[d] ← r[d] + 1</code></td>
<td>inc rd</td>
<td>63–d</td>
</tr>
<tr>
<td>inc address</td>
<td><code>r[d] ← r[d] + 4</code></td>
<td>inca rd</td>
<td>64–d</td>
</tr>
<tr>
<td>dec</td>
<td><code>r[d] ← r[d] - 1</code></td>
<td>dec rd</td>
<td>65–d</td>
</tr>
<tr>
<td>dec address</td>
<td><code>r[d] ← r[d] - 4</code></td>
<td>deca rd</td>
<td>66–d</td>
</tr>
<tr>
<td>not</td>
<td><code>r[d] ← ~ r[d]</code></td>
<td>not rd</td>
<td>67–d</td>
</tr>
</tbody>
</table>

### Shifting NOP and Halt

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift left</td>
<td><code>r[d] ← r[d] &lt;&lt; S = s</code></td>
<td>shl rd, s</td>
<td>7dSS</td>
</tr>
<tr>
<td>shift right</td>
<td><code>r[d] ← r[d] &gt;&gt; S = −s</code></td>
<td>shr rd, s</td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td><code>halt machine</code></td>
<td>halt</td>
<td>f0--</td>
</tr>
<tr>
<td>nop</td>
<td><code>do nothing</code></td>
<td>nop</td>
<td>ff--</td>
</tr>
</tbody>
</table>
Global Dynamic Array
Global Dynamic Array

- **Java**
  - array variable stores reference to array allocated dynamically with `new` statement

  ```java
  public class Foo {
      static int a;
      static int b[] = new int[10];

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

- **C**
  - array variables can store static arrays or pointers to arrays allocated dynamically with call to `malloc` library procedure

  ```c
  int a;
  int* b;

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

  *malloc does not assign a type*  
  *# of bytes to allocate*
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

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

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

```c
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]

0x2000: value of b
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

\[
\begin{align*}
0x2000: & \text{ value of } b[0] \\
0x2004: & \text{ value of } b[1] \\
& \ldots \\
0x2024: & \text{ value of } b[9] \\
0x3000: & \text{ value of } b[0] \\
0x3004: & \text{ value of } b[1] \\
& \ldots \\
0x3024: & \text{ value of } b[9]
\end{align*}
\]
In assembly language

### Static Array

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

.pos 0x1000
a_data:
.long 0        # the variable a

.pos 0x2000
b_data:
.long 0        # the variable b[0]
.long 0        # the variable b[1]
... 
.long 0        # the variable b[9]
```

### Dynamic Array

```
ld $a_data, r0  # r1 = address of a
ld (r0), r1     # r2 = a
ld $b_data, r2  # r2 = address of b
ld (r2), r3     # r3 = b
st r1, (r3,r1,4) # b[a] = a

.pos 0x1000
a_data:
.long 0        # the variable a

.pos 0x2000
b_data:
.long 0        # the b
```

Comparing static and dynamic arrays

- what is the benefit of static arrays?
- what is the benefit of dynamic arrays?
Pointers in C
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 \( m[m[b] + x \times \text{sizeof(array-element)}] \leftarrow 0 \)

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

```c
int *a;
a[4] = 5;
int *a;
*(a+4) = 5;
```

For array access, the compiler would generate this code

```
r[0] ← a
r[1] ← 4
r[2] ← 5
m[r[0]+4*r[1]] ← r[2]
```

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

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.
Pointer Arithmetic in C

- Its purpose
  - an alternative way to access dynamic arrays to the a[i]

- Adding or subtracting an integer index to a pointer
  - results in a new pointer of the same type
  - value of the pointer is offset by index times size of pointer’s referent
  - for example
    - adding 3 to an int* yields a pointer value 12 larger than the original

- Subtracting two pointers of the same type
  - results in an integer
  - gives number of referent-type elements between the two pointers
  - for example
    - (& a[7]) − (& a[2])) == 5 == (a+7) − (a+2)

- other operators
  - & X the address of X
  - * X the value X points to
What is the equivalent Java statement to

- [C] there is no typesafe equivalent
- [D] not valid, because you can’t take the address of a static in Java
Looking more closely

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

\[
\begin{align*}
& r[0] \leftarrow 0x2000 \quad \# r[0] = &c \\
& r[1] \leftarrow m[r[0]] \quad \# r[1] = c \\
& r[2] \leftarrow 12 \quad \# r[2] = 3 \times \text{sizeof(int)} \\
& m[r[0]] \leftarrow r[2] \quad \# c = c + 3 \\
& r[3] \leftarrow 3 \quad \# r[3] = 3 \\
& m[r[2]] \leftarrow r[4] \quad \# c[0] = c[3]
\end{align*}
\]

Before

\[
0x2000: 0x3000 \\
0x3000: 0 \\
0x3004: 1 \\
0x3008: 2 \\
0x300c: 3 \\
0x3010: 4 \\
0x3014: 5 \\
0x3018: 6 \\
0x301c: 7 \\
0x3020: 8
\]

After

\[
0x2000: 0x300c \\
0x3000: 0 \\
0x3004: 1 \\
0x3008: 2 \\
0x300c: 6 \\
0x3010: 4 \\
0x3014: 5 \\
0x3018: 6 \\
0x301c: 7 \\
0x3020: 8
\]

\[
\text{c[0] = c[3]}
\]
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]
```

```
    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