a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

CPSC 213
Introduction to Computer Systems

Unit 1b: Static Scalars and Arrays

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

Announcements

- Google doc for lecture questions
  - See Canvas for link (Modules -> Resources)
  - https://docs.google.com/document/d/1wxjVBLQbLmbRLXrTAzygaPMhovRsYJsETHP_yEU-9o8/edit

  - Add your question anonymously (at the top)
  - Help answer questions too!

Overview

- Reading
  - Companion: 1, 2.1-2.3, 2.4.1-2.4.3
  - Textbook: 3.1-3.2.1
  - Reference (textbook, as needed): 3.1-3.5, 3.8, 3.9.3
- Learning objectives:
  - list the basic components of a simple computer and describe their function
  - describe ALU functionality in terms of inputs and outputs
  - describe the exchange of data between ALU, registers, and memory
  - identify and describe the basic components of a machine instruction
  - outline the steps a RISC machine performs to do arithmetic on numbers stored in memory
  - outline the steps a simple CPU performs when it processes an instruction
  - translate between array-element offsets and indices
  - distinguish static and dynamic computation for access to global scalars and arrays in C
  - translate between C and assembly language for access to global scalars and arrays
  - translate between C and assembly for code that performs simple arithmetic
  - explain the difference between arrays in C and Java

Our approach

- Develop a model of computation
  - that is rooted in what the machine actually does
  - by examining C, bit-by-bit (comparing to Java as we go)

- The processor
  - we will design (and you will implement) a simple instruction set
  - based on what we need to compute C programs
  - similar to a real instruction set (MIPS)

- The language
  - we will act as compiler to translate C into machine language/assembly
  - bit by bit, then putting the bits together to do interesting things
  - edit, debug, and run using simulated processor to visualize execution

The CPU

- CPUs execute instructions, not C or Java code
- Execution proceeds in stages
  - Fetch: load the next instruction from memory
  - Decode: figure out from the instruction what needs to be done
  - Execute: do what the instruction specifies
  - There can be more or fewer stages than this, depending on your model/implementation (CPSC 121, 313)

- These stages are looped over and over again forever

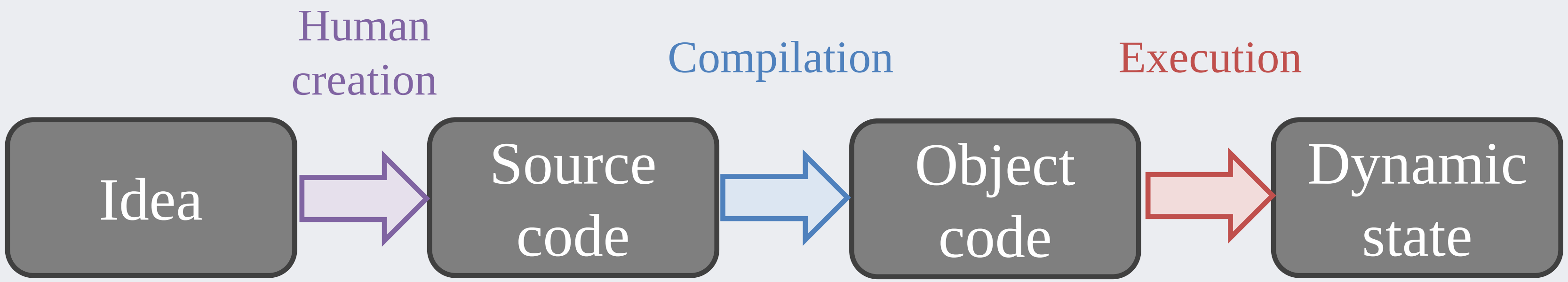
CPU instructions

- CPU instructions are very simple
  - Read (load) a value from memory
  - Write (store) a value to memory
  - Add/subtract/AND/OR/etc. two numbers
  - Shift a number
  - Control flow (see these in unit 1d)

- Some of these operations are carried out by an ALU (Arithmetic & Logic Unit)
  - ALU is used in the execute stage

Phases of computation

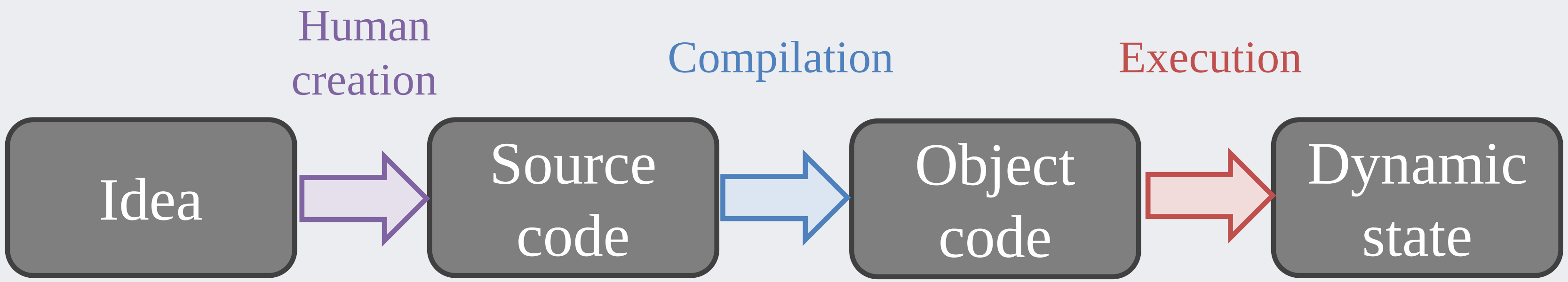
- Human creation: design program and describe it in high-level language
- Compilation: convert high-level human description into machine-executable text
- Execution: a physical machine executes the code text



Static vs dynamic computation

- Execution
  - Parameterized by input values unknown at compilation
  - Producing output values that are unknowable at compilation

- Anything the compiler can compute is called *static*
- Anything that can only be discovered during execution is called *dynamic*



iClicker 1b.1

- Consider the code below:

```
int a;
a = 5;
```

The value assigned to variable `a` is:

A. static
B. dynamic
C. neither static nor dynamic
D. it depends

iClicker 1b.2

- Assume that a variable of type `int` is assigned a value that is read from the user through an input dialog.

This value is:

A. static

B. dynamic

C. neither static nor dynamic

D. it depends

The processor (CPU)

- Implements a set of instructions
- Each instruction is implemented using logic gates
  - Built from transistors: fundamental mechanism of computation
  - (Recall your CPSC 121 labs)

- Instruction design philosophies
  - RISC: fewer and simpler instructions makes processor design simpler
  - CISC: having more types of instructions (and more complex instructions) allows for shorter/simpler program code and simpler compilers

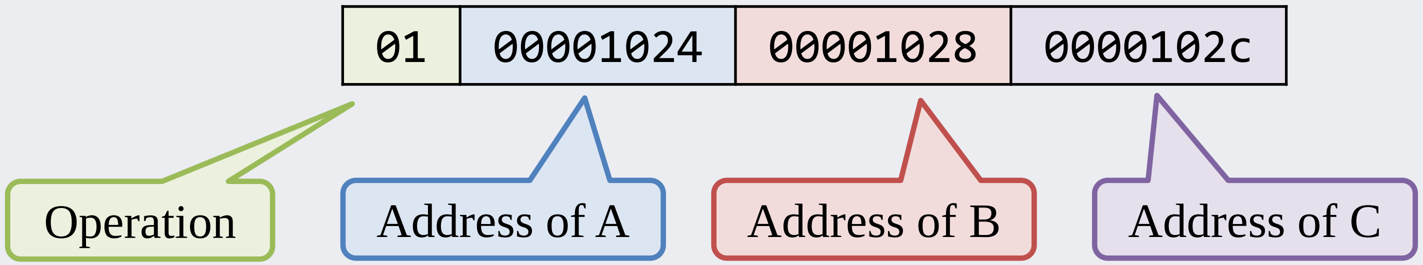
First proposed instruction: ADD

- Let's propose an instruction that does: $A \leftarrow B + C$
  - where A, B, and C are stored in memory

- Instruction parameters: addresses of A, B, C
  - each address is 32 bits (modern computers: 64 bits)

- Instruction is encoded as a sequence of bits (stored in memory)
  - Operation name (e.g. add)
  - Addresses for A, B, C

This will occupy at least 13 bytes in memory



Improving the ADD instruction

Problems with memory access

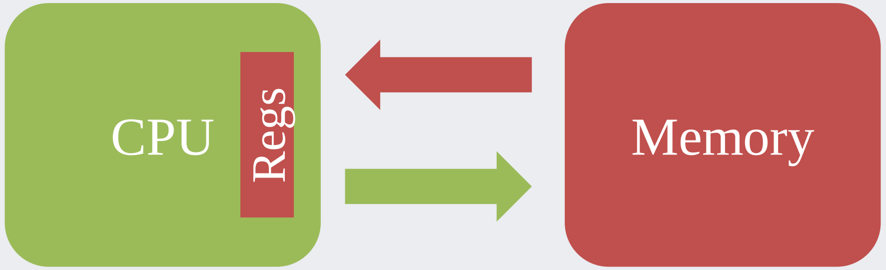
- Accessing memory is SLOW
  - ~100 CPU cycles for every memory access
  - goal: fast programs that avoid accessing memory when possible

- Big instructions are costly
  - Memory addresses are big (so instructions that use them are big)
  - Big instructions lead to big programs
  - Reading instructions from memory is slow (fewer caching options)
  - Large instructions use more CPU resources (transfer, storage)

General purpose registers

- Register file
  - Small, fast memory stored in CPU itself
  - roughly single-cycle access

- Registers
  - Each register named by a number (e.g., 0–7)
    - some of these may sometimes be used for other purposes
  - Size: architecture's common integer (32 or 64 bits)



Instructions using registers

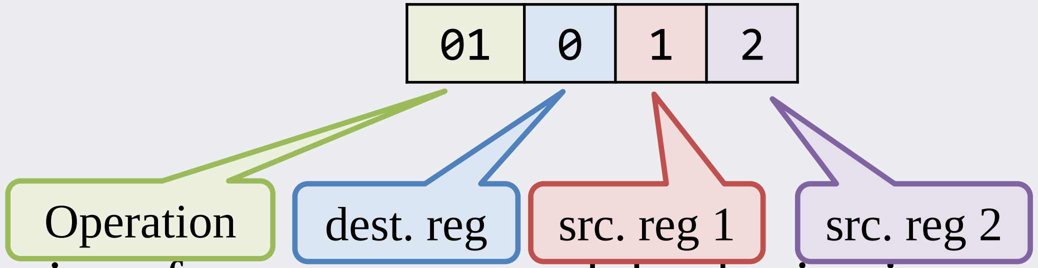
Improving our ADD instruction

Let's modify our instruction design

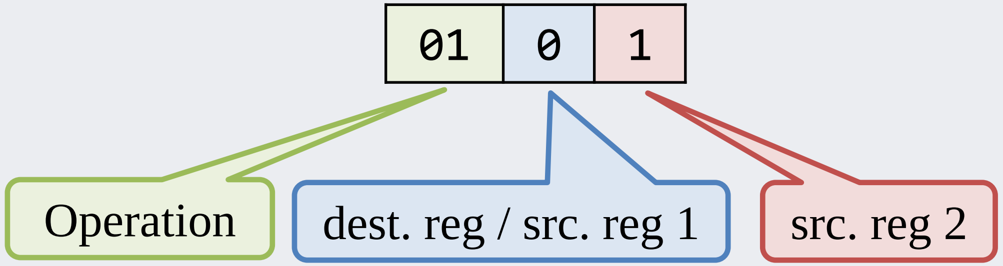
01	00001024	00001028	0000102c

- Memory instructions handle only memory
  - Load data from memory into a register (slow)
  - Store data from register into memory (slow)

- Other instructions access data in registers
  - small and fast



- To further improve instruction size: share register for one source and the destination



Special purpose registers

- A special-purpose register can only be used for certain purposes
  - physically separate from the general-purpose register file
  - May not be accessible by all instructions (e.g., cannot be used as an argument for an add instruction)
  - May have special meaning or be treated specially by CPU

- Examples:
  - PC (Program Counter): contains address of next instruction to execute
  - IR (Instruction Register): contains the machine instruction that has just been fetched from memory

Instruction Set Architecture (ISA)

- ISA is a formal interface to a processor implementation
  - defines the instructions the processor implements
  - defines the format of each instruction

- Types of instructions:
  - math and logic
  - memory access
  - control transfer: "goto" and conditional "goto"

ISA design

- Design alternatives:
  - CISC: simplify compiler design (e.g. Intel x86, x86-64)
  - RISC: simplify processor implementation

- Instruction format
  - sequence of bits: opcode and operand values
  - all represented as numbers (in binary / hex)

- Assembly language:
  - symbolic (textual) representation of machine code

Representing Instruction Semantics

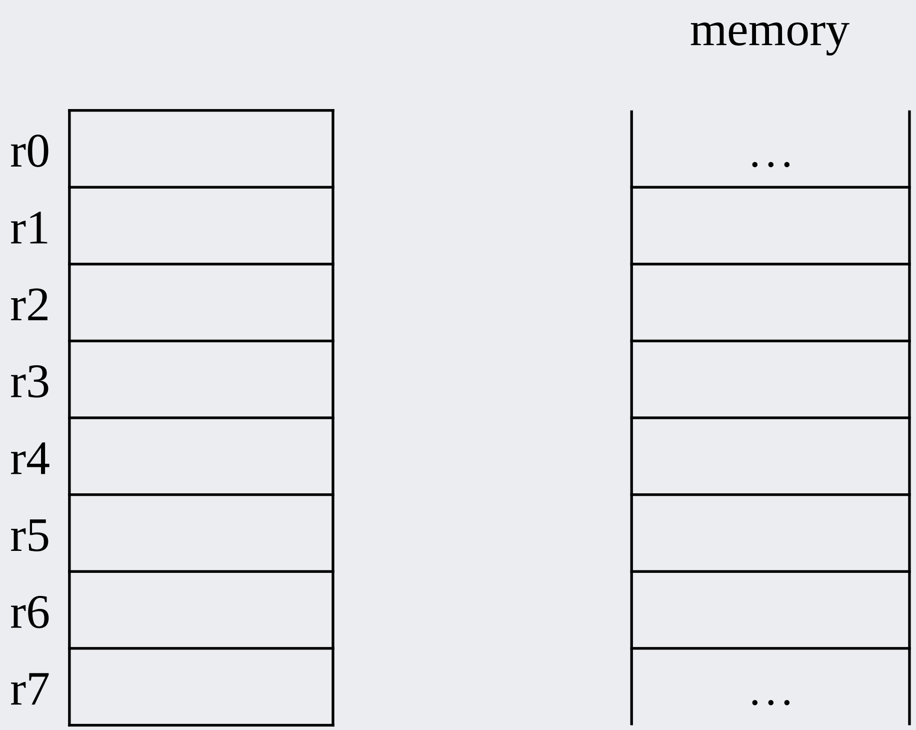
- RTL: simple, convenient pseudo language to describe semantics
  - easy to read/write, describes machine steps

- Syntax:
  - each line is of the form: `LHS ← RHS`
  - `LHS` is a memory or register that receives a value
  - `RHS` is constant, memory, register, or expression on two registers
  - `m[a]` is a memory in address `a`
  - `r[i]` is a register with number `i`

Register file and memory are treated as arrays

RTL examples

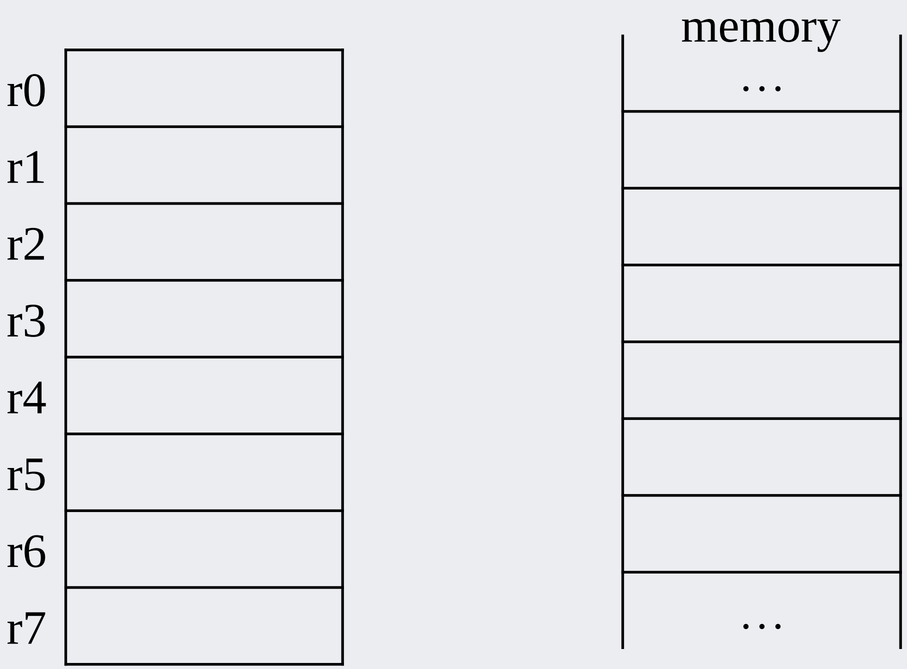
- Register 0 receives `0x2000`
  - `r[0] ← 0x2000`
- Register 1 receives memory whose address is in register 0
  - `r[1] ← m[r[0]]`
- Register 2 is increased by the value in register 1:
  - `r[2] ← r[2] + r[1]`



RTL example

- Assume the value 7 is stored at address 0x1234.
- What is the result of the following sequence of instructions?

```
r[0] ← 10
r[1] ← 0x1234
r[2] ← m[r[1]]
r[2] ← r[0] + r[2]
m[r[1]] ← r[2]
```



Variables

- Variables are named storage locations for values

- Features:
  - Name
  - Type/size
  - Scope
  - Lifetime
  - Memory location (address)
  - Value
- Which of these are static? Which are dynamic?
  - Which are determined or can change while the program is running?

Static variables, built-in types

- In 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 (later)

Java:

```java
public class Foo {
  static int a;
  static int[] b; // array not static, ignore for now
  public void foo() {
    a = 0;
    b[a] = a;
  }
}
```

C:

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

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

Static variable allocation

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

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

Static memory layout

0x1000: value of a
…
0x2000: value of b[0]
0x2004: value of b[1]
…
0x2024: value of b[9]

- Allocation is
  - assigning a memory location (i.e. an address) to a variable
  - When does this happen? How is the location found?

- Static vs dynamic computation
  - global/static variables can exist before program starts and live until after it finishes
  - compiler allocates variables, giving them a constant address
  - no dynamic computation is required to allocate; they just exist
  - compiler tracks free space during compilation – it is in complete control of program's memory

iClicker 1b.3

- Assume `a` is a global variable in C. When is space for `a` allocated? In other words, when is its address determined?

A. The compiler assigns the address when it compiles the program
B. The compiler calls the memory to allocate `a` when it compiles the program
C. The compiler generates code to allocate `a` before the program starts running
D. The program locates available space for `a` before it starts running
E. The program locates available space as soon as the variable is used for the first time

Static variable access

Scalars

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

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

Static memory layout

0x1000: value of a
…
0x2000: value of b[0]
0x2004: value of b[1]
…
0x2024: value of b[9]

- Key observation:
  - Addresses of `a`, `b[0]`, `b[1]`, … are constants known to the compiler

- Let's now describe:
  - How the statement **`a = 0;`** changes machine state
  - What the hardware instructions that implement it need to do

iClicker 1b.4

- What is a proper RTL instruction for
  `a = 0;`

A. `r[0x1000] ← 0`
B. `m[0x1000] ← 0`
C. `0x1000 ← r[0]`
D. `m[r[0x1000]] ← 0`
E. `0x1000 ← m[0]`

Static memory layout

0x1000: value of a
…
0x2000: value of b[0]
0x2004: value of b[1]
…
0x2024: value of b[9]

Static variable access

Static arrays

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

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

Static memory layout

```
0x1000: value of a
…
0x2000: value of b[0]
0x2004: value of b[1]
…
0x2024: value of b[9]
```

- Key observation:
  - Value of `a` is generally unknown at compilation time
  - Even though addresses of `b[0], b[1]` … are known statically, which one is used here is dynamic
  - compiler does not know address of `b[a]`
    - unless it knows the value of `a` statically, which it could by if there is a line of code like `a = 0;`, but not in general
- Array access is computed from a base address and index
  - address of element = base + offset; offset = index × element size
  - The base address (0x2000) and element size (4) are static; the index is dynamic

iClicker 1b.5

- What is a proper RTL instruction for

  `b[a] = a;`

  Assume that the compiler does not know the current value of `a`.

A. `m[0x2000+m[0x1000]] ← m[0x1000]`
B. `m[0x2000+4*m[0x1000]] ← m[0x1000]`
C. `m[0x2000+m[0x1000]] ← 4*m[0x1000]`
D. `m[0x2000]+4*m[0x1000] ← m[0x1000]`
E. `m[0x2000]+m[0x1000] ← m[0x1000]`

Static memory layout

0x1000: value of a
…
0x2000: value of b[0]
0x2004: value of b[1]
…
0x2024: value of b[9]

Alternatives for instructions sets

- What RTL instruction(s) can we use for `a = 0`?

- **Option 1:** static address and value

  ```
  m[0x1000] ← 0x0
  ```

  - 9 bytes (instruction code + two integers)

- **Option 2**: static address, dynamic value

  ```
  r[0] ← 0x0
  m[0x1000] ←r[0]
  ```

  - 5 bytes for immediate (constant) value instruction
    - instruction code + register + one integer
  - 5 bytes for memory instruction
    - instruction code + register + one integer

Alternatives for instruction sets

- What instructions can we use for `a = 0`?

- Option 3: dynamic address and value

```
r[0] ← 0x0
r[1] ← 0x1000
m[r[1]] ← r[0]
```

  - 2 bytes for memory instruction
    - instruction code, two registers
  - More flexibility (address and value can be dynamic)

Alternatives for instruction sets

- What RTL instruction(s) can we use for `b[a] = a`?

- **Option 1**: static base address, index (address), and value (address)

  `m[0x2000 + 4*m[0x1000]] ← m[0x1000]`

  - Inputs:
    - Base address (`0x2000`)
    - Index address(`0x1000`)
    - Input value address (`0x1000`)
  - 13 bytes to encode the instruction (instruction code + three integers)

Alternatives for instruction sets

- What RTL instruction(s) can we use for `b[a] = a`?

- **Option 2**: calculate address explicitly

```
r[0] ← 0x1000
r[1] ← m[r[0]]
r[3] ← r[1] * 4 (or r[3] ← r[1] << 2)
r[2] ← 0x2000
r[2] ← r[2] + r[3]
m[r[2]] ← r[1]
```

  - Uses same instructions described from earlier
  - Issue: array access are very common

Alternatives for instruction sets

- What RTL instruction(s) can we use for `b[a] = a`?

- **Option 3**: dynamic base address, index, and value

```
r[0] ← 0x1000
r[1] ← m[r[0]]
r[2] ← 0x2000
m[r[2] + r[1] * 4] ← r[1]
```

  - 2 bytes for indexed memory instruction
    - instruction, three registers

ISA design goals (RISC paradigm)

Choosing the instructions

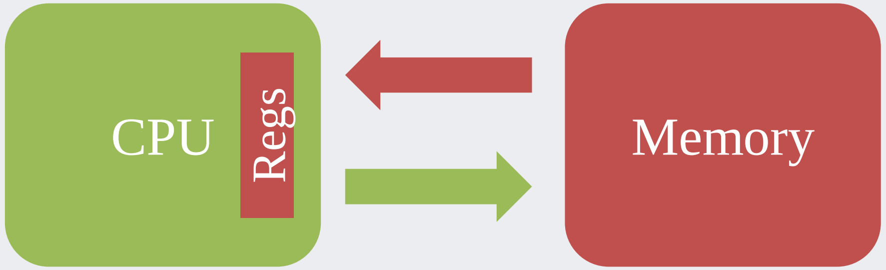
```
a = 0;
```

```
m[0x1000] ← 0x0
```

```
b[a] = a;
```

```
m[0x2000 + 4*m[0x1000]] ← m[0x1000]
```

- minimize the number of memory instructions in ISA
  - at most 1 memory access per instruction
  - No other operation in a memory instruction
- minimize the total number of instructions in ISA
- minimize the size of each instruction




```
r[0] ← 0x0
r[1] ← 0x1000        1
m[r[1]] ← r[0]       2
```

```
r[0] ← 0x1000
r[1] ← m[r[0]]       3
r[2] ← 0x2000
m[r[2] + 4*r[1]] ← r[1]   4
```

and we also have a "load" version of 4 5

What instructions do we need so far?

- Requirements for scalars (e.g. assign value to global variable):
  - Load a constant value into a register

    `r[x] ← v` (where `v` is a constant integer)

  - Store a value in a register into memory at some address

    `m[r[y]] ← r[x]` (assume `r[y]` has address of value)

  - Load a value in memory into a register

    `r[x] ← m[r[y]]` (assume `r[y]` has address of value)

- Additional requirements for arrays:
  - Store value in a register into memory at an indexed location (at address in register * 4 plus base address)

    `m[r[z] + r[x]*4] ← r[y]`

  - Load value in memory from an indexed location (address in register * 4 plus base address) into register

    `r[y] ← m[r[z] + r[x]*4]`

SM213 ISA

The first five instructions so far

	Name	Semantics	Assembly	Machine
1	load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
3	load base + offset	`r[d] ← m[r[s]+4*p]`	`ld o(rs), rd`	`1psd`
5	load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`	`2sid`
2	store base + offset	`m[r[d]+4*p] ← r[s]`	`st rs, o(rd)`	`3spd`
4	store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`	`4sdi`

Translating the code

RTL and assembly

- From high level code

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

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

- To RTL

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

r[2]     ← m[r[1]]
r[3]     ← 0x2000
m[r[3]+4*r[2]] ← r[2]
```

- To SM213 assembly

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

ld (r1), r2
ld $0x2000, r3
st r2, (r3, r2, 4)
```

Name	Semantics	Assembly	Machine
load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
load base + offset	`r[d] ← m[r[s]+4*p]`	`ld o(rs), rd`	`1psd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`	`2sid`
store base + offset	`m[r[d]+4*p] ← r[s]`	`st rs, o(rd)`	`3spd`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`	`4sdi`

Translating the code

…and machine code

...	
0x1000	
...	
0x2000	
0x2004	
0x2008	
0x200c	
0x2010	
0x2014	
0x2018	
0x201c	
0x2020	
0x2024	
...	

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

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

```
ld $0, r0               # r0 = 0
ld $0x1000, r1          # r1 = address of a
st r0, (r1)             # a = 0

ld (r1), r2             # r2 = a
ld $0x2000, r3          # r3 = address of b
st r2, (r3, r2, 4)      # b[a] = a

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

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

```
00-- 00000000
01-  00001000
3001
1102
03-  00002000
4232
```

iClicker 1b.6

```
int i;
int a[10];

a[2] = a[i];
```

Name	Semantics	Assembly
load immediate	`r[d] ← v`	`ld $v, rd`
load base + offset	`r[d] ← m[r[s]+4*p]`	`ld o(rs), rd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`
store base + offset	`m[r[d]+4*p] ← r[s]`	`st rs, o(rd)`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`

- Which correctly implements `a[2] = a[i];`?

A.
```
st ($a, $i, 4), ($a, 2, 4)
```

B.
```
ld ($a, $i, 4), r0
st r0, ($a, 2, 4)
```

C.
```
ld $a, r0
ld $i, r1
ld (r1), r1
ld (r0, r1, 4), r2
st r2, 2(r0)
```

D.
```
ld $a, r0
ld $i, r1
ld (r1), r1
ld (r0, r1, 4), r2
ld $2, r1
st r2, (r0, r1, 4)
```

E. None of these / I don't know

The Simple Machine (SM213) ISA

- Architecture
  - Register file — Eight 32-bit general purpose registers (numbered 0-7)
  - CPU — one cycle per instruction (fetch + execute)
  - Main Memory — byte addressed, Big endian integers

- Instruction format
  - 2- or 6-byte instructions (each character below is a hex digit)
    - `x-01`, `xxsd`, `x0vv`, or `x-sd vvvvvvvv`
  - where:
    - `x` is an opcode (unique identifier for this instruction)
    - `-` means unused
    - `s` and `d` are operand register numbers
    - `vv`, `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 an instruction with opcode x and operands 0 and 1
    - `vvvvvvvv` hex value of optional extended value

- Assembly code
  - `[label:] [instruction | directive] [# comment]`

```
directive    :: (.pos number) | (.long number)
instruction  :: opcode operand+
operand      :: $literal | reg | offset(reg) | (reg, reg, 4)
reg          :: r0..7
literal      :: number
offset       :: number
number       :: decimal | 0xhex
```

Memory access instructions

Load and Store with different addressing modes

Name	Semantics	Assembly	Machine
load immediate	r[d] ← v	ld $v, rd	0d-- vvvvvvvv
load base + offset	r[d] ← m[r[s]+(o=4*p)]	ld o(rs), rd	1psd
load indexed	r[d] ← m[r[s]+4*r[i]]	ld (rs, ri, 4), rd	2sid
store base + offset	m[r[d]+(o=4*p)] ← r[s]	st rs, o(rd)	3spd
store indexed	m[r[d]+4*r[i]] ← r[s]	st rs, (rd, ri, 4)	4sdi

- We have specified 4 addressing modes for operands
  - **immediate** constant value stored as part of instruction
  - **register** operand is register number; register stores value
  - **base+offset** operand is register number;
    register stores memory address of value (+ offset = p*4)
  - **indexed** two register-number operands;
    store base memory address and value of index

Basic arithmetic, shifting, NOP, and halt

- Arithmetic

Name	Semantics	Assembly	Machine
register move	`r[d] ← r[s]`	`mov rs, rd`	`60sd`
add	`r[d] ← r[d] + r[s]`	`add rs, rd`	`61sd`
and	`r[d] ← r[d] & r[s]`	`and rs, rd`	`62sd`
inc	`r[d] ← r[d] + 1`	`inc rd`	`63-d`
inc address	`r[d] ← r[d] + 4`	`inca rd`	`64-d`
dec	`r[d] ← r[d] - 1`	`dec rd`	`65-d`
dec address	`r[d] ← r[d] - 4`	`deca rd`	`66-d`
not	`r[d] ← ~r[s]`	`not rd`	`67-d`

- Logical (shifting), NOP, and halt

Name	Semantics	Assembly	Machine
shift left	`r[d] ← r[d] << v=s`	`shl $v, rd`	`7dss (ss>0)`
shift right	`r[d] ← r[d] >> v=s`	`shr $v, rd`	`7dss (ss<0)`
halt	*`halt machine`*	`halt`	`F0--`
nop	*`do nothing`*	`nop`	`FF--`

iClicker 1b.7

- What instruction is represented by the bytes: `0x46 0x24`?

A. `ld (r4, r4, 4), r6`
B. `st r4, (r6, r2, 4)`
C. `st r6, (r2, r4, 4)`
D. `st r6, 2(r4)`
E. `st r6, 8(r4)`

Name	Semantics	Assembly	Machine
load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
load base + offset	`r[d] ← m[r[s]+(o=4*p)]`	`ld o(rs), rd`	`1psd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`	`2sid`
store base + offset	`m[r[d]+(o=4*p)] ← r[s]`	`st rs, o(rd)`	`3spd`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`	`4sdi`

iClicker 1b.8

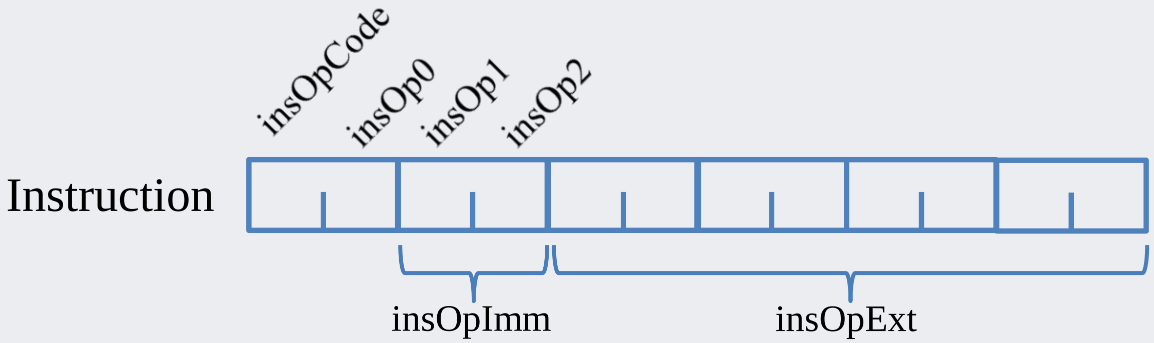
- What instruction is represented by the bytes: `0x75 0xF8`?

```
A. shl $5, r8
B. shl $8, r5
C. shr $5, r8
D. shr $8, r5
E. shl $0xf8, r5
```

Name	**Semantics**	**Assembly**	**Machine**	
shift left	`r[d] ← r[d] << v=s`	`shl $v, rd`	`7dss`	`(ss>0)`
shift right	`r[d] ← r[d] >> v=s`	`shr $v, rd`	`7dss`	`(ss<0)`

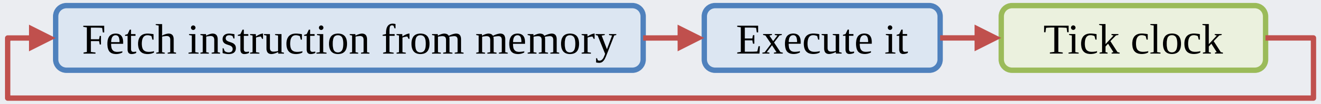
The SM213 CPU implementation

- Internal state
  - PC (program counter): address of ***next*** instruction to fetch
  - Instruction: the value of the current instruction



Reg	Value
PC:	0000010e
Instruction:	3001 00000000
Ins Op Code:	3
Ins Op 0:	0
Ins Op 1:	0
Ins Op 2:	1
Ins Op Imm:	01
Ins Op Ext:	00000000

- Cycle stages



  - Fetch
    - read instruction at PC, determine size, separate components, set next sequential PC
  - Execute
    - read internal state, perform specified computation (insOpCode), update internal state
    - read and update may be to memory as well

The SM213 CPU implementation

Java syntax

- Internal registers
  - `insOp0`, `insOp1`, `insOp2`, `insOpImm`, `insOpExt`, `pc`
  - Read using `get()`
  - Change using `set(value)`

```
int i = insOp1.get();
insOp1.set(i);
```

- General purpose registers
  - `reg.get(registerNumber)`
  - `reg.set(registerNumber, value)`

```
int i = reg.get(3);   // i <- r[3]
reg.set(3, i);        // r[3] <- i
```

- Main memory
  - `mem.readInteger(address)`
  - `mem.writeInteger(address, value)`

```
int i = mem.readInteger(0x1000);  // i <- m[0x1000]
mem.writeInteger(0x1000, i);      // m[0x1000] <- i
```

Global dynamic arrays

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

```
public class Foo {
  static int a;
  static int b[];

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

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

dynamic array

```
int a;
int* b;

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

static array

```
int a;
int b_data[10];
int* b = &b_data[0];
```

C pointers

Compared to Java

- Terminology
  - use the term ***pointer*** instead of ***reference*** (in Java); they mean the same thing
- Declaration
  - the type is a pointer to the type of its element(s), indicated with a *
  - Stored internally as an address, regardless of element type
- Type safety
  - any pointer can be typecast to any other pointer type
- Bounds checking
  - C performs no array bounds checking
  - out-of-bounds access manipulates memory that is not part of the array
  - Performance is faster, but this is a major source of vulnerability

Dynamic allocation in C

- Dynamic allocation is done using a call to `malloc`

- Returns the address of a block of bytes, with no associated type or initialization
  - Element type and dereferencing determines how the binary data are interpreted

- Dynamic allocation will be discussed in more details in future units

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

- 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 (will contain address of array)

Static vs dynamic arrays

- …then when the program runs
  - the dynamic array is allocated by a call to `malloc`, e.g. at address `0x3000`
  - the (static) variable `b` is set to the memory address of this array

0x2000: 0x3000 → 0x3000: value of b[0]
0x3004: value of b[1]
…
0x3024: value of b[9]

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

Static

```
r[0]      ← 0x1000
r[1]      ← m[r[0]]
r[2]      ← 0x2000

m[r[2]+4*r[1]] ← r[1]
```

Dynamic

```
r[0]      ← 0x1000
r[1]      ← m[r[0]]
r[2]      ← 0x2000
r[3]      ← m[r[2]]

m[r[3]+4*r[1]] ← r[1]
```

Array access in SM213 assembly

b[a] = a;

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

- Static allocation:

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

.pos 0x1000
a: .long 0 # the variable a
.pos 0x2000
b: .long 0 # the variable b[0]
   .long 0 # the variable b[1]
   ...
   .long 0 # the variable b[9]
```

```
int a;
int* b = malloc(10 * sizeof(int));
```

- Dynamic allocation:

```
ld $a, r0   # r0 = address of a
ld (r0), r1 # r1 = a
ld $b, r2   # r2 = address of b
ld (r2), r3 # r3 = b = &b[0]
st r1, (r3, r1, 4)   # b[a] = a
```

- For our simulator (since it has no malloc)

```
...
.pos 0x2000
b: .long b_data # malloc result
.pos 0x3000 # or wherever malloc put this
b_data: .long 0 # b[0]
        .long 0 # b[1]
        ...
        .long 0 # b[9]
```

iClicker 1b.9

- How many memory load operations are required to handle this instruction?

```
b[a[0]] = a[b[0]];
```

A. 2
B. 3
C. 4
D. 5
E. Not enough information to tell

Summary: arrays in Java and C

- In Java and C:
  - an array is a list of items of the same type
  - array elements are named by non-negative integers starting at 0
  - syntax for accessing element `i` of array `b` is `b[i]`

- In Java:
  - variable stores a reference to the array

- In C:
  - variable can store a pointer to the array, or the array itself
    - distinguished by variable type (i.e. pointer variable vs array variable)
  - array element access via pointer can be done by
    - `b[i]`, or
    - `*(b+i)` 🤔??

Declaring (and dereferencing) C pointers

- Pointer variables declared as:

  `TYPE* varname;`

  where `TYPE` is any type (even another pointer type), e.g. `int*` points to an `int`, `int**` points to an `int*`
  - multiple variables can be declared on a line, but beware
    - `int* x, y;   // this declares x as int*, and y as int`
    - `int *x, *y;    // this declares x and y both as int*`
  - Even more complicated with in-line initialization, so Geoff recommends keeping one declaration per line
- Can be assigned the address of a variable of type TYPE

```
int a = 5;
int* p = &a;
```

- Access the value being pointed to by dereferencing with *

```
int a = 5;
int* p = &a;
*p = 37;
```

Pointer arithmetic

`*(b+i)` 😮

- Purpose:
  - an alternative way to access array elements compared to a[i]
- Addresses are just numbers, so…
  - adding or subtracting an integer index to a pointer (address)
    - results in the address of something else (another pointer of the same type)
    - value of the pointer is offset by index × size of the pointer's referent
  - e.g. adding 3 to an `int*` yields a pointer value 12 bytes larger than the original

  - subtracting two pointers of the same type
    - results in an integer: the number of referent-type elements between the two pointers
    - e.g. `(&a[7] - &a[2]) == 5 == (a+7) – (a+2)`

Pointer arithmetic

Example

- The following C programs are identical

```
int* a;
a[7] = 5;
```

```
int* a;
*(a+7) = 5;
```

- For array access, the compiler would generate this code

```
r[0]     ← a
r[1]     ← m[r[0]]
r[2]     ← 7
r[3]     ← 5
m[r[1]+4*r[2]] ← r[3]
```

```
ld $a, r0
ld (r0), r1
ld $7, r2
ld $5, r3
st r3, (r1, r2, 4)
```

  - multiplying the index (7) by 4 (size of integer) to compute the array offset

iClicker 1b.10

- Which element of the original 10-element array is modified with the highlighted instructions?

A. Element with index 3
B. Element with index 6
C. Element with index 0
D. No element is modified
E. More than one element is modified

```
int* c;

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

0x1000	
...	
0x2000	
0x2004	
0x2008	
0x200c	
0x2010	
0x2014	
0x2018	
0x201c	
0x2020	
0x2024	

Previous code in SM213 assembly

```
ld  $0x2000, r0 # r0 = address of c
ld  (r0), r1    # r1 = c (malloc result)
ld  $12, r2     # r2 = 3*sizeof(int)
add r1, r2      # r2 = c+3 = &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 = c[3]

.pos 0x2000
c: .long 0 # or some data used in simulator to emulate malloc
```

```
int* c;

void foo() {
  c = malloc(10 * sizeof(int));
  c = &c[3];
  *c = *&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 of the array
- What C does that Java doesn't
  - static arrays
  - arrays can be accessed using pointer dereference operator
  - pointer arithmetic
- What Java does that C doesn't
  - type-safe dynamic allocation
  - automatic array bounds checking

Aside: Strings in C

- In C, strings are implemented as arrays of characters
  - Each element is a one-byte integer (type char), representing the character as an entry in the ASCII table
    - See www.asciitable.com for binary/hex codes
  - e.g. `'A'` is represented as 65_{10}, `'a'` as 97_{10}, `'0'` as 48_{10}, `' '` (space) as 32_{10}
  - The value 0_{10} (`'\0'`) indicates the end of the string (but not necessarily the end of the array)

- So the string `"CPSC 213"` is represented as:

'C'	'P'	'S'	'C'	' '	'2'	'1'	'3'	'\0'			
67	80	83	67	32	50	49	51	0			

- Some special characters:
  - `'\n'` : line break
  - `'\r'` : carriage return (back to start of line, but don't move to next line

Aside: Program arguments in C

- The main function in C typically receives two arguments:

  ```
  int main(int argc, char* argv[]) { ...
  ```

  - `argc` : the number of arguments (includes the name of the program itself)
  - `argv` : an array of strings, each representing an argument
  - `argv[0]` is the name of the program, so argc is always ≥ 1

- So, if you call your program:

  ```
  ./myprogram –x 7
  ```

  - Your program will receive:
    - `argc = 3`
    - `argv[0] = "./myprogram"`
    - `argv[1] = "-x"`
    - `argv[2] = "7"`

Aside: `printf` and `scanf` in C

To output text to the console, use `printf`

- First argument: format string in double quotes, exact characters plus placeholders
- Remaining arguments replace placeholders in format string, in same order
  - `%s` : string
  - `%d` or `%i` : signed integer in base 10
  - `%u` : unsigned integer in base 10
  - `%x` : hexadecimal integer
  - e.g. `printf("My grade in %s was %d.\n", "CPSC 213", gr);`
    - where `gr` is a variable of integer type

- To read text from the console, use `scanf`
  - arguments are similar to `printf` (input string plus destination variables)
  - arguments must be pointers to destination variables