CPSC 213

Introduction to Computer Systems

Unit 1b

Static Scalars and Arrays

Reading for Next 3 Lectures

- Companion
  - 2.4.1-2.4.3
- Textbook
  - Array Allocation and Access
  - 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 hexits)
    - `x-01`, `xx01`, `x0vv` or `x-01 vvvvvvvv`
  - where
    - `x` is *opcode* (unique identifier for this instruction)
    - `-` means unused
    - `0` and `1` are operands
    - `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 instruction with opcode x and operands 0 and 1
  - `vvvvvvvv` 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 of Built-In Types

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)

```
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)

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

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

Static Variable Allocation

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

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

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

Static Memory Layout

```
0x1000: value of a
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2020: value of b[9]
```

- 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 variable's 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)

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

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

```
a = 0;
```

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

Static Memory Layout

```
0x1000: value of a
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2020: value of b[9]
```

- 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 its a = a + 2? or a = b? or a = foo ()?
- * What about reading the value of a?

Question (scalars)

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

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

```
a = 0;
```

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

Static Memory Layout

```
0x1000: value of a
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2020: value of b[9]
```

- When is space for **a** allocated (when is its address determined)?
  - [A] The program locates available space for **a** when program starts
  - [B] The compiler assigns the address when it compiles the program
  - [C] The compiler calls the memory to allocate **a** when it compiles the program
  - [D] The compiler generates code to allocate **a** before the program starts running
  - [E] The program locates available space for **a** when the program starts running
  - [F] The program locates available space for **a** just before calling **`foo()`**

Static Variable Access (static arrays)

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

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

```
a = 0;
```

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

Static Memory Layout

```
0x1000: value of a
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2020: value of b[9]
```

- Key Observation
  - compiler does not know address of **`b[a]`**
    - unless it can knows the value of a statically, which it could here by looking at a=0, but not in general
- Array access is computed from base and index
  - address of element is *base* plus *offset*; *offset* is *index* times element size
  - the base address (0x2000) and element size (4) are static, the index is dynamic
- Use RTL to specify instructions for **`b[a] = a`**, not knowing **`a`**?

Designing ISA for Static Variables

- Requirements for scalars `a = 0;`
  - load constant into register
    - r[x] ← v
  - store value in register into memory at constant address
    - m[0x1000] ← r[x]
  - load value in memory at constant address into a register
    - r[x] ← m[0x1000]
- Additional requirements for arrays `b[a] = a;`
  - store value in register into memory at address in register*4 plus constant
    - m[0x2000+r[x]*4] ← r[y]
  - load value in memory at address in register*4 plus constant into register
    - r[y] ← m[0x2000+r[x]*4]
- Generalizing and simplifying we get
  - r[x] ← constant
  - m[r[x]] ← r[y] and r[y] ← m[r[x]]
  - m[r[x] + r[y]*4] ← r[z] and r[z] ← m[r[x] + r[y]*4]

The compiler's semantic translation

- it uses these instructions to compile the program snippet

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

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

→

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

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

ISA Specification for these 5 instructions

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

The compiler's assembly translation

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

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

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

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

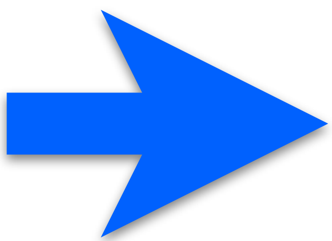
```
int a;
int b[10];

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

```
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

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

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



```
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

.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]
```

Addressing Modes

- In these instructions

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

Basic Arithmetic, Shifting NOP and Halt

- Arithmentic

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[d]	not rd	67-d

- Shifting NOP and Halt

Name	Semantics	Assembly	Machine
shift left	r[d] ← r[d] << S = s	shl rd, s	71SS
shift right	r[d] ← r[d] << S = -s	shr rd, s	
halt	*halt machine*	halt	f0--
nop	*do nothing*	nop	ff--

Global Dynamic Array

Global Dynamic Array

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

```
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

```
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

```
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;
}
```

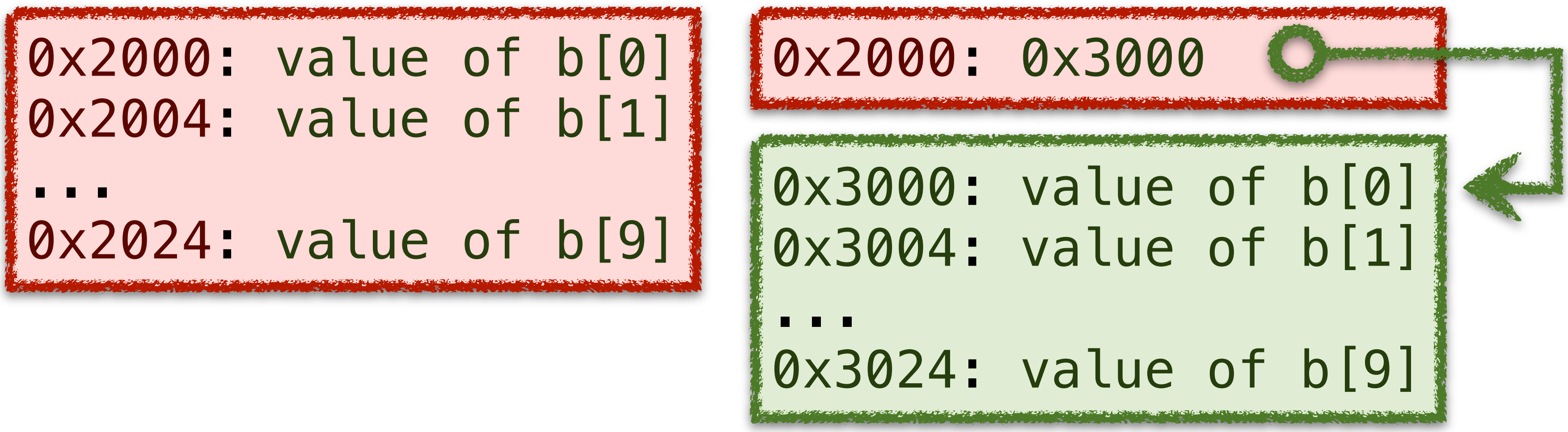
- Static allocation
  - for static arrays, the compiler allocates the array
  - for dynamic arrays, the compiler allocates a pointer

```
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2024: value of b[9]
```

```
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

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

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

load a

load b

b[a]=a

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?

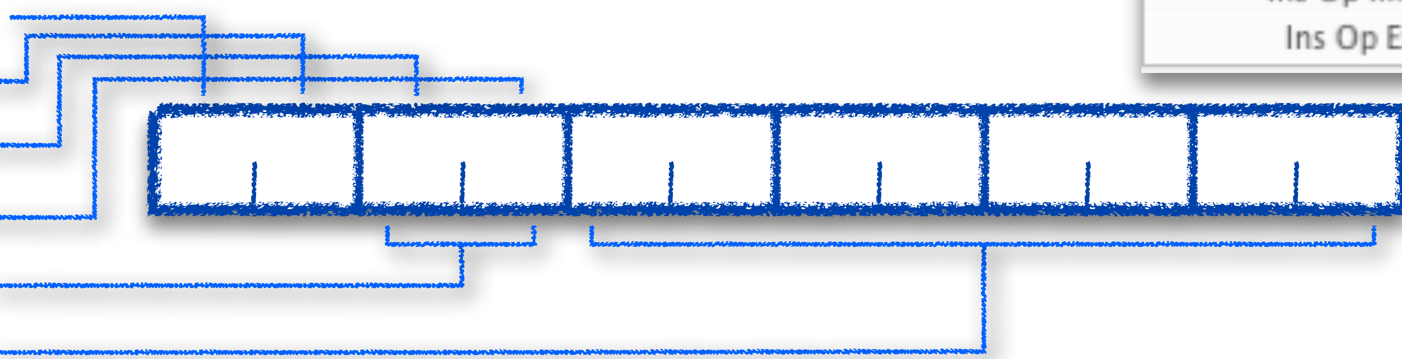
Implementing the ISA

The CPU Implementation

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

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

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 * 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*
or *m[**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)

Example

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

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

  - 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

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

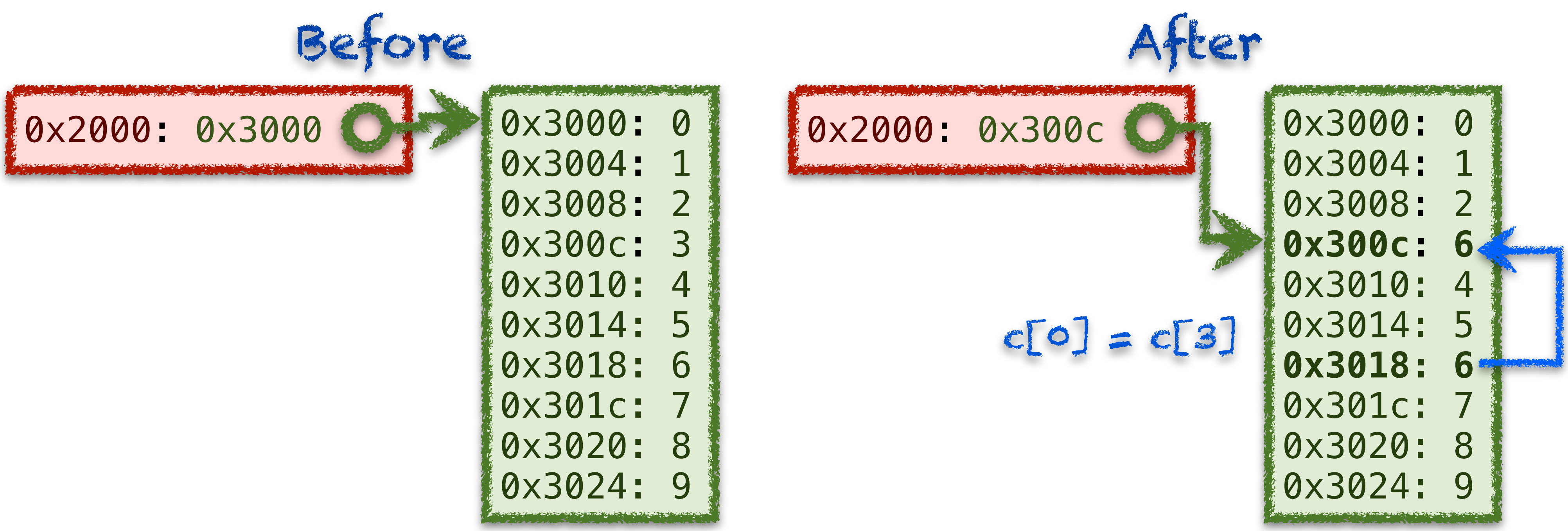
```
int *c;

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

- What is the equivalent Java statement to
  - [A] c[0] = c[3];
  - [B] c[3] = c[6];
  - [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];
```

```
r[0]      ← 0x2000           # r[0] = &c
r[1]      ← m[r[0]]          # r[1] = c
r[2]      ← 12               # r[2] = 3 * sizeof(int)
r[3]      ← r[2]+r[1]        # r[2] = c + 3
m[r[0]]   ← r[2]             # c    = c + 3

r[3]      ← 3                # r[3] = 3
r[4]      ← m[r[2]+4*r[3]]   # r[4] = c[3]
m[r[2]]   ← r[4]             # c[0] = c[3]
```



And in assembly language

```
r[0]     ← 0x2000          # r[0] = &c
r[1]     ← m[r[0]]         # r[1] = c
r[2]     ← 12              # r[2] = 3 * sizeof(int)
r[3]     ← r[2]+r[1]       # r[2] = c + 3
m[r[0]] ← r[2]             # c    = c + 3

r[3]     ← 3               # r[3] = 3
r[4]     ← m[r[2]+4*r[3]] # r[4] = 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