New Zealand

COMPSCI 210 Part 2

"There are 10 types of people in this world: those who understand binary, and those who don't." - *Malcolm Nielsen*

THE UNIVERSITY OF AUCKLAND

NEW ZEALAND

Te Whare Wānanga o Tāmaki Makaurau

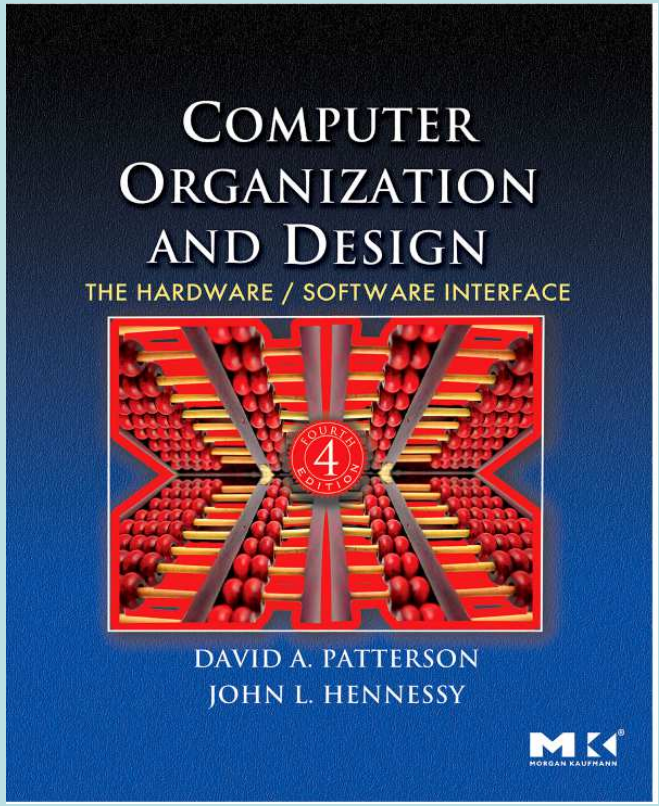
Mano Manoharan, Room 723.315, Tamaki Campus
mano@cs.auckland.ac.nz

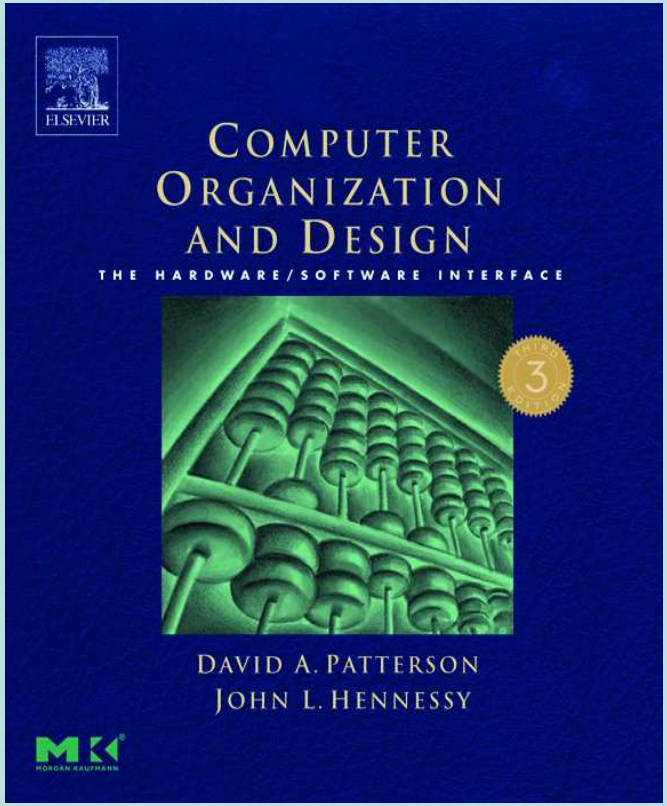
The University of Auckland

Where we are heading

- A bit of C/C++ programming
- How some simple programming constructs translate to assembly
- How programs are run on a computer – the hardware-software interface
- A programmer's view of computer memory
- A look at caches

Recommended Reading





Patterson & Hennessy, Computer Organization & Design: The Hardware/Software Interface, Morgan Kaufmann, 4th edition.

The 3rd or 2nd edition is OK too.

Recommended Reading

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

- For C/C++, the recommended book is “How to Think Like a Computer Scientist” by Allen Downey.
- This is free to download from http://www.greenteapress.com/thinkcpp/
- Google & Wiki

Free C/C++Compilers

- Linux/Mac: use gcc – the GNU C/C++/Objective C compiler
  - Use file extension .c for C source files, .cpp for C++ files, and .m for objective C files.
  - The command g++ is a convenient alias to compile C++ files.
    - *Find out how g++ relates to gcc.*
- Windows:
  - Microsoft Visual C++ Express Edition is a free compiler and IDE that is easy to install and use.

Hello World – C++ Version


```
#include <iostream>

int
main()
{
   // Send to the standard output stream the string "hello ..."
   std::cout << "hello world\n";

   return 0;
}
```

g++ helloWorld.cpp (to compile – on Mac/Linux)
./a.out (to run – on Mac/Linux)

Basic IO – C++ Version


```
#include <iostream>

int
main()
{
   // Send to the output stream the string "enter two ..."
   std::cout << "enter two numbers: ";
   int a, b;
   // Receive from the input stream the integer values 'a' and 'b'
   std::cin >> a >> b;
   // Send to the output stream a string and the integer sum
   std::cout << "sum: " << (a+b) << "\n";

   return 0;
}
```

Bitwise Operations

Operation	Operator	Examples
AND	&	a & b
OR	|	a | b
NOT	~	~a
XOR	^	a ^ b
LEFT SHIFT	<<	a << 2
RIGHT SHIFT	>>	a >> 2

Bitwise Operations

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0

Bitwise Operations: A & B

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0

Bitwise Operations: A & B

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0
	1										1	1				

Bitwise Operations: A | B

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0

Bitwise Operations: A | B

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0
					0			0	0							0

Bitwise Operations: ~A

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0

Bitwise Operations: ~A

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
	0	0	1	0	1	1	0	1	1	1	0	0	0	1	0	1

Bitwise Operations: A ^ B

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0

Bitwise Operations: A ^ B

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0
B	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0	0
		1	1	1		1	1			1			1		1	

Bitwise Operations: A << n

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0

Bitwise Operations: A >> n

#	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
A	1	1	0	1	0	0	1	0	0	0	1	1	1	0	1	0

Below the program

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

The University of Auckland
New Zealand

```
void
swap(int v[ ], int k)
{
      int temp;
      temp = v[ k];
      v[ k] = v[ k+1];
      v[ k+1] = temp;
}
```

C/C++ compiler

Assembler

```
SWAP    ADD R2, R3, R4
        LDR R5, R2, #0
        LDR R6, R2, #1
        STR R6, R2, #0
        STR R5, R2, #1
        RET
```

```
0000000010100001
1000110001100010
1000110011110010
1010110011110010
1010110001100010
0000001111100000
```

Below the program

- Pick your favourite C++ program and generate the assembly code for it.
  - ‘g++ -S helloWorld.cpp’ will create no executable but simply an assembly file ‘helloWorld.s’. See if you are able to understand the generated assembly.
- You can generate the machine code of this assembly code using ‘as’
  - ‘as -o helloWorld.o helloWorld.s’ will give you the machine code in ‘helloWorld.o’
- Can you execute the machine code (e.g. ‘helloWorld.o’)?
  - Find out how to get the executable code ‘a.out’ from ‘helloWorld.o’. You may find the ‘-v’ option of ‘g++’ handy here.

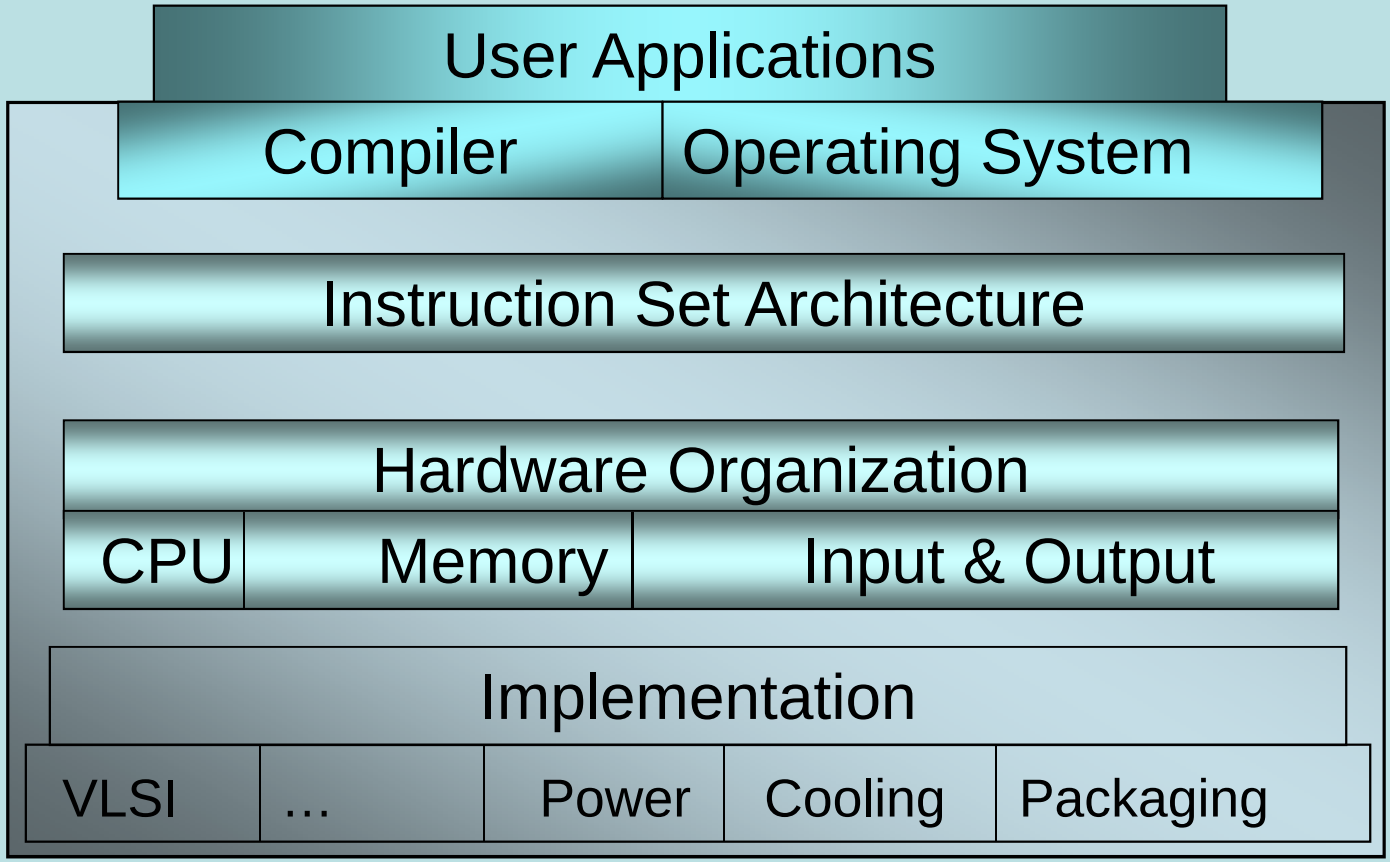
Exercises

- Find out what the command ‘nm’ does. Try out ‘nm helloWord.o’ and understand its output.
- Find out what the command ‘strip’ does.
- And try ‘make love’.

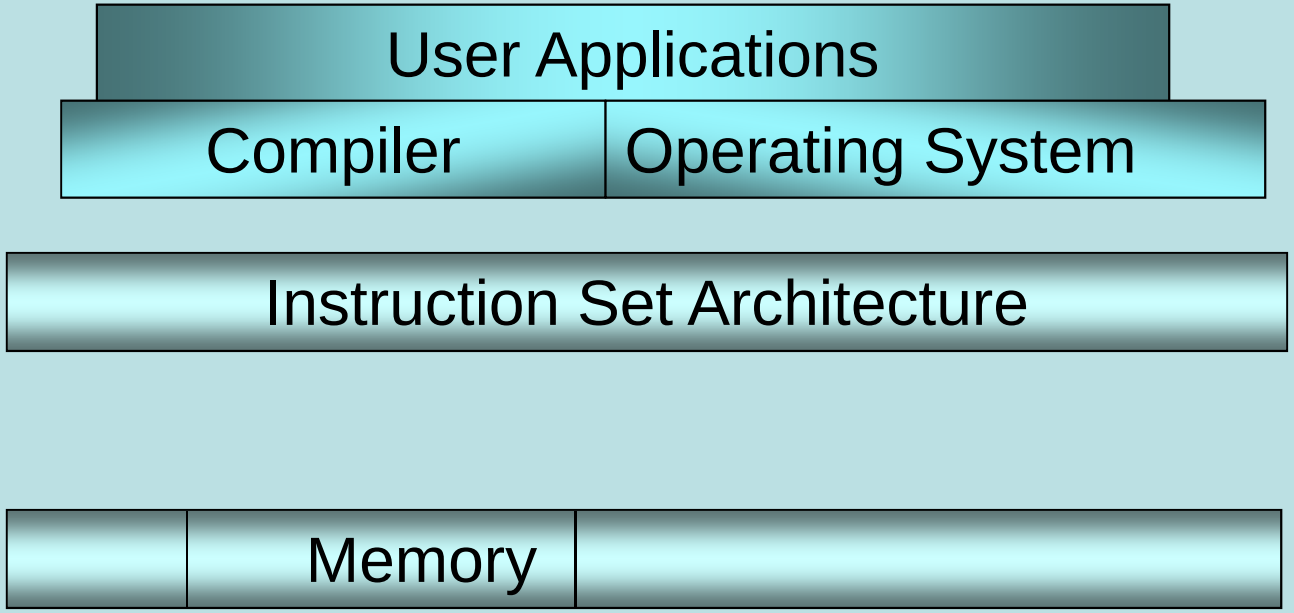
Quiz

- We have three machines – a Mac, a Linux PC, and a Windows PC. All have an Intel CPU in them.
- I compile my ‘helloWorld.cpp’ on the Mac. Can I run the resulting executable on Linux and/or Windows?

The Big Picture



The Big Picture



Memory Organization

- Viewed as a large, single-dimension array.
- A memory *address* is an index into the array
- *Byte addressing* means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
…	8 bits of data
…	8 bits of data

Memory Organization

- Bytes are nice, but most data items use larger "words“
  - Words are the smallest units of data we get out of/into memory
- A word is 32 bits or 4 bytes in a 32-bit CPU.
  - 2^{32} bytes with byte addresses from 0 to 2^{32}-1
  - 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Similarly, in a 16-bit CPU (e.g. LC-3), a word is 16 bits

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
…	8 bits of data

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
…	32 bits of data

0	16 bits of data
2	16 bits of data
4	16 bits of data
6	16 bits of data
…	16 bits of data

Byte Address vs. Word Address

- Byte Address is the fundamental address in a memory system, and is the address of a byte.
- Word Address is the address of a word.
  - Not all byte addresses are valid word addresses.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
…	8 bits of data

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
…	32 bits of data

0	16 bits of data
2	16 bits of data
4	16 bits of data
6	16 bits of data
…	16 bits of data

Pointers

- A pointer is the address of any object. If you have an integer called ‘n’ then, ‘&n’ will give you the address of ‘n’.
- A pointer type is declared with a ‘*’ suffix attached to the type. An ‘int’ is an integer, and an ‘int*’ is a pointer to an integer.
  - A floating-point type is a ‘float’; what is ‘float*’?

```
int n = 36;
int* an = &n;        // address of ‘n’ or “pointer” to ‘n’
*an = 32;            // set the contents of ‘an’ to 32 - this is equivalent
                     // to saying ‘n = 32’, since ‘an’ is pointing to ‘n’
```

Arrays

- Arrays are declared using the standard square bracket syntax: int myArray[128] declares an integer array of size 128.
- The first array element is myArray[0], the second is myArray[1] and so on.
- What is the address of myArray[4]?
  - Answer: &myArray[4] – this is no different from taking the address of a simple scalar object!
- In C/C++, 'myArray' is a short-hand notation for &myArray[0] – the start address of the array 'myArray'.

Arrays and Pointers


```
const int MYMAX = 16;
int array[MYMAX];
for ( int i = 0; i < MYMAX; ++i )
{
   array[i] = i*i;
}

std::cout << "index\tvalue\taddress\n";
for ( int i = 0; i < MYMAX; ++i )
{
   std::cout << i << "\t" << array[i]  << "\t"
             << (unsigned long) &array[i] << "\n";
}
```

Work out what is going on here. Download the sample code ‘arraysAndPointers.cpp’ and experiment with it.

Decimal to 32-bit Floating Point


```
float v = -4.375;
int* pi = (int*)(&v);
std::cout << std::hex << *pi << "\n";
```

&v gives the address of the floating point object *v*. The cast (int *) tells the compiler to treat the address as the address of an integer. Both *pi* and *&v* has the same address. However, the compiler now thinks pi is the address of an integer. Then we print the contents of *pi*. The floating point representation of the *v* is therefore printed out.

32-bit Floating Point to Decimal


```
int v = 0x44361000;
float* pf = (float*)(&v);
std::cout << *pf << "\n";
```

&v gives the address of the integer object *v*. The cast (float*) tells the compiler to treat the address as the address of a floating point object. Both *pf* and *&v* has the same address. However, the compiler now thinks *pf* is the address of a floating point object. Then we print the contents of *pf*. The floating point value of the of floating point representation in *v* is therefore printed out.

Floating Point Operations


```
#include <iostream>

int
main()
{
   for (double d = 0; d != 0.3; d += 0.1)
   {
       std::cout << "d: " << d << "\n";
   }
   return 0;
}
```

Try this example out, and see if the output meets your expectations.

Decimal to 32-bit Floating Point

- Consider the decimal value 0.1_{10}.
  - This in binary is $0.0001100110011001100110011001100..._2$.
  - Normalizing this gives us $1.100110011 \times 2^{-4}$.

		Integral part
0.1 x 2	0.2	0
0.2 x 2	0.4	0
0.4 x 2	0.8	0
0.8 x 2	0.6	1
0.6 x 2	0.2	1
0.2 x 2	0.4	0
0.4 x 2	0.8	0
0.8 x 2	0.6	1

- The sign is positive, so S = 0
- Exponent is -4 + bias (127) = $123_{10} = 01111011_2$.
- Mantissa is 100110 (leaving out the hidden 1 and the point).

- So we have 1 01111011 10011001100…1100

Comparing Floating Point Numbers

- Do not compare two floating point numbers for equality or inequality.
  - The results of equality or inequality aren't likely to meet your expectation.
- Always take the difference of the two numbers and see if this difference is less or greater than a chosen threshold.
  - E.g. rather than d == 0.3 do *fabs*(d – 0.3) < 0.00001.
  - Rather than d != 0.3 do *fabs*(d – 0.3) > 0.00001.

Floating Point Arithmetic


```
#include <iostream>

int
main(int argc, char*argv[])
{
   if ( argc < 2 ) return 1;

   int k = ::atoi(argv[1]);
   float kBy10 = (float)k/10;
   float kSum = 0;
   for (int i = 0; i < k; ++i)
   {
      kSum += 0.1;
   }
   float kDiff = kBy10 - kSum;

   std::cout << "kBy10: " << kBy10 << "\n";
   std::cout << "kSum: " << kSum << "\n";
   std::cout << "kDiff: " << kDiff << "\n";
   return 0;
}
```

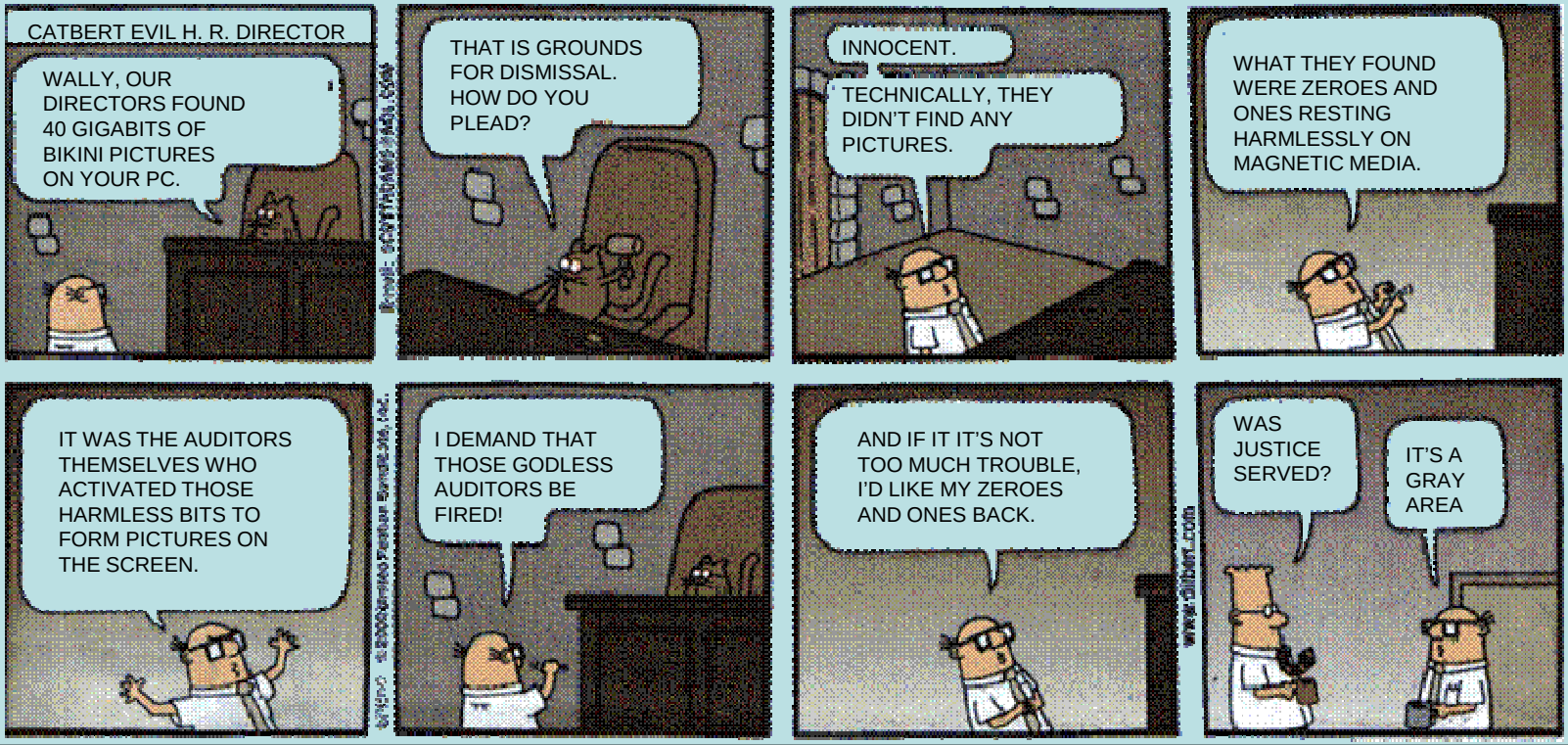
Try this example out with different input values (i.e. k values).

Do we expect kBy10 and kSum be equal?

Data & Code

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

- When you look at the memory, there is no difference between data and code.
- Both are stored in the memory using 0s and 1s.
  - The difference lies in how we interpret 0s and 1s.



Function Pointers

- Functions (or methods) are laid out in memory just like data. A function pointer is simply the address of the first instruction of the function.
  - If you have an integer called ‘n’ then, ‘&n’ will give you the address of ‘n’. If you have a function called ‘foo’ then, ‘&foo’ will give you the address of the function ‘foo’.

```
void
swap(int v[ ], int k)
{
      int temp;
      temp = v[ k];
      v[ k] = v[ k+1];
      v[ k+1] = temp;
}
```

```
SWAP    ADD R2, R3, R4
        LDR R5, R2, #0
        LDR R6, R2, #1
        STR R6, R2, #0
        STR R5, R2, #1
        RET
```

```
void (*swapFnPtr)(int v[], int k);
swapFnPtr = &swap;
```

```
(*swapFnPtr)(myArray, 4);
swap(myArray, 4); // same as above
```

Function Pointers

- Where are function pointers useful?

- Imagine you have two sort functions that sort an array of integers: void QuickSort(int a[]) and void ShellSort(int a[]).
  - void (*sortFnPtr)(int a[]); // Pointer to any sort function
  - Set sortFnPtr to the sort function of your choice somewhere in the code where you choose the sort algorithm to use.
  - The remainder of the code will simply use (*sortFnPtr) rather than a specific sort function.

Function Pointers

- A function pointer always points to a function with a specific signature.
  - Thus all functions you want to use with the same function pointer must have the same parameters and return-type!
- More on the use of function pointers when we look at polymorphism.

Passing Arguments to Functions

- The default way of passing arguments to functions (or methods) is' by value'. Passing by 'value' means passing a copy of the object.
- Changes to the object within the function won't be therefore reflected in the original object.

- You can pass arguments 'by reference'. Passing by 'reference' means the object itself gets passed.
- This is handy when the object needs to be changed within the function.
  - In which other circumstance, 'passing by reference' is useful?

Download the sample code 'swap.cpp' and experiment with it.

Passing Arguments to Functions


```
int a = 34;
foo(a);
```

void foo(int n) { … } – ‘n’ can be changed but has no effect on ‘a’

void foo(const int n) { … } – ‘n’ cannot be changed

void foo(int& n) { … } – ‘n’ is an alias to ‘a’ so any changes to ‘n’ will change ‘a’ – ‘n’ and ‘a’ represent the same object.

void foo(const int& n) { … } ‘n’ is an alias to ‘a’ but ‘n’ cannot be changed since it is marked a constant.

Passing Arguments to Functions

- Primitive types (such as int, float etc) are always passed as values.
- Arrays are passed by reference – passing an array means passing the address of its first element, i.e. address of the array, i.e. reference to the array
- Other types (e.g. any user-defined or library type) are
  - Passed by reference in Java & C#
  - Passed by value in C/C++ (consistent but inefficient) by default, but user can choose to pass by reference (preferred because of efficiency). If an object is to be changed, pass it by reference; if it is not to be changed, mark it a constant to prevent accidental changes to it.

THE UNIVERSITY OF AUCKLAND

NEW ZEALAND

Te Whare Wānanga o Tāmaki Makaurau

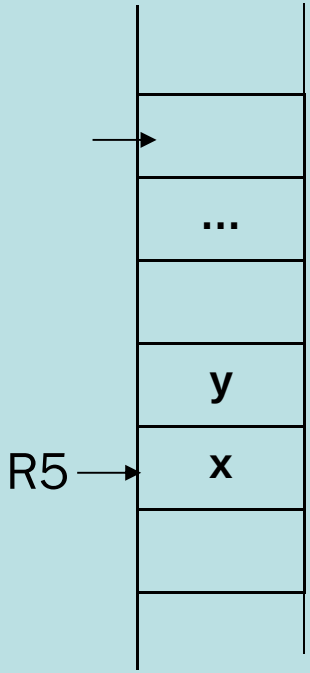
The University of Auckland

New Zealand

Compiling Programs

Conditionals

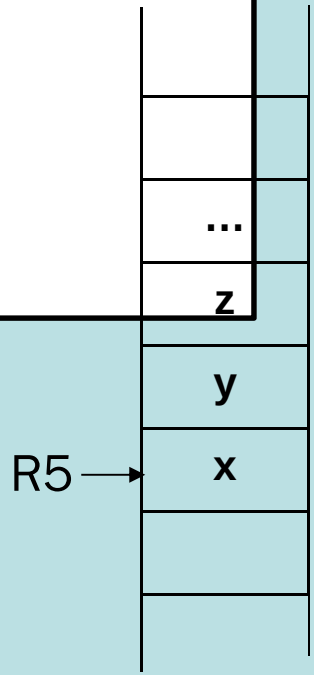

```
if (x == 2)
        y = 5;
```




```
LDR  R0, R5, #0          ; load x into R0
ADD  R0, R0, #-2         ; subtract 2
BRnp NOT_TRUE    ; if non-zero, x is not 2

AND  R1, R1, #0          ; store 5 to y
ADD  R1, R1, #5
STR  R1, R5, #-1

NOT_TRUE ...             ; next
statement
```

Conditionals


```
if (x == 2)
{
  y++;
  z--;
}
else
{
  y--;
  z++;
}
```




```
LDR  R0, R5, #0        ; load x into R0
ADD  R0, R0, #-2       ; subtract 2
BRnp ELSE              ; if non-zero, x is not 2

LDR  R1, R5, #-1  ; load y
ADD  R1, R1, #1   ; incr y
STR  R1, R5, #-1  ; store y
LDR R1, R5, #-2   ; load z
ADD  R1, R1, #-1  ; decr z
STR  R1, R5, #-2  ; store z
BR  DONE  ; skip else code

ELSE  LDR  R1, R5, #-1  ; decr y
      ADD  R1, R1, #-1
      STR  R1, R5, #-1
      LDR  R1, R5, #-2  ; incr z
      ADD  R1, R1, #1
      STR  R1, R5, #-2

DONE  ...  ; next statement
```

Loops


```
int x = 0;
while (x < 10)
{
  printf(...);
  x = x + 1;
}
```

```
        AND  R0, R0, #0
        STR  R0, R5, #0 ; x = 0

LOOP    LDR  R0, R5, #0 ; load x
        ADD  R0, R0, #-10 ; x = x - 10
        BRzp DONE
        ; loop body

        LDR  R0, R5, #0 ; load x (why?)
         ...
        <printf>
        ...
        ADD  R0, R0, #1 ; incr x
        STR  R0, R5, #0 ; store x
        BR  LOOP   ; go back to loop

DONE    ; next statement
```

Loops


```
for (int x = 0; x < 10; ++x)
{
  printf(…);
}
```

```
        AND  R0, R0, #0
        STR  R0, R5, #0 ; x = 0

LOOP    LDR  R0, R5, #0 ; load x
        ADD  R0, R0, #-10 ; x = x - 10
        BRzp DONE
        ; loop body

        LDR  R0, R5, #0 ; load x (why?)
         ...
        <printf>
        ...
        ADD  R0, R0, #1 ; incr x
        STR  R0, R5, #0 ; store x
        BR  LOOP   ; go back to loop

DONE    ; next statement
```

Do we need to do:
LDR R0, R5, #0 ; load x
after the call to printf as well?

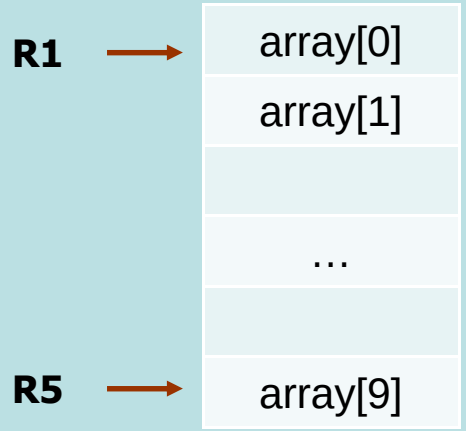
Think about which registers printf might modify.

Loops


```
int i = 0;
while ( i < 10 )  {
        sum += array[i];
        ++i;
}
```

```
for (i = 0;  i < 10; ++i ) {
        sum += array[i];
}
```

```
        ADD R5, R1, #10 ; set end addr

LOOP    LDR R2, R1, #0 ; load array[i]
        ADD R3, R3, R2 ; sum is in R3
        ADD R1, R1, #1 ; R1 points to next
                       ; array element
        SUB R4,R5,R1   ; check if R1 got
                       ; to R5; if so exit
        BRzp LOOP
EXIT    …
```



How to do ‘SUB’?

Assembler Macros

SUB Rd, Rs1, Rs2

NOT **Rt**, Rs2
ADD **Rt**, **Rt**, #1
ADD Rd, Rs1, **Rt**

```
        ADD R5, R1, #10 ; set end addr

LOOP    LDR R2, R1, #0 ; load array[i]
        ADD R3, R3, R2 ; sum is in R3
        ADD R1, R1, #1 ; R1 points to next
                       ; array element
        SUB R4,R5,R1   ; check if R1 got
                       ; to R5; if so exit
        BRzp LOOP
EXIT    …
```

Note the use of a temporary register Rt. What happens if we use Rt elsewhere in the assembly code? For example, what happens in our loop example if Rt happens to be R3?

Assembler Macros

SUB Rd, Rs1, Rs2

NOT Rd, Rs2
ADD Rd, Rd, #1
ADD Rd, Rs1, Rd

Can we re-write the macro to avoid using a temporary register? Will it work if Rs2, Rs1 and Rd are all the same registers?

To Loop or Not


```
int i = 0;
while ( i < 10 )  {
        sum += array[i];
        ++i;
}
```

```
sum += array[0];
sum += array[1];
sum += array[2];
sum += array[3];
sum += array[4];
sum += array[5];
sum += array[6];
sum += array[7];
sum += array[8];
sum += array[9];
```

Which of these is better code?
Why?

To Loop or Not


```
LDR R2, R1, #0 ; load array[0]
ADD R3, R3, R2 ; sum is in R3
LDR R2, R1, #1 ; load array[1]
ADD R3, R3, R2 ; sum is in R3

…

LDR R2, R1, #9 ; load array[9]
ADD R3, R3, R2 ; sum is in R3
```

```
sum += array[0];
sum += array[1];
sum += array[2];
sum += array[3];
sum += array[4];
sum += array[5];
sum += array[6];
sum += array[7];
sum += array[8];
sum += array[9];
```

To Loop or Not


```
LDR R2, R1, #0 ; load array[0]
ADD R3, R3, R2 ; sum is in R3
LDR R2, R1, #1 ; load array[1]
ADD R3, R3, R2 ; sum is in R3

…

LDR R2, R1, #9 ; load array[9]
ADD R3, R3, R2 ; sum is in R3
```

```
        ADD R5, R1, #10 ; set end addr

LOOP    LDR R2, R1, #0 ; load array[i]
        ADD R3, R3, R2 ; sum is in R3
        ADD R1, R1, #1 ; R2 points to next
                       ; array element
        SUB R4,R5,R1   ; check if R1 got
                       ; to R5; if so exit
        BRzp LOOP
EXIT    …
```

How many instructions are there in the two code fragments (static instruction count)?

How many instructions get executed in the two code fragments (dynamic instruction count)?

CPU & Memory: The Big Picture

Register File
Register #0
Register #1
Register #2
Register #N

Program Counter

Address	Instruction
30000401	AND R0, R0, #0
30000402	ADD R0, R3, #2
30000403	LDR R5, R2, #1
30000404	LDR R6, R2, #2
30000405	STR R6, R5, #1
30000406	STR R5, R6, #2
30000407	BRzp 30000401
…	
…	

Where does Java fit in?

A Simple Java Class


```
class Swap
{
   static void SwapElements(int a[], int k)
   {
      int temp = a[k];
      a[k] = a[k+1];
      a[k+1] = temp;
   } // SwapElements
} // class Swap
```

Java Bytecode

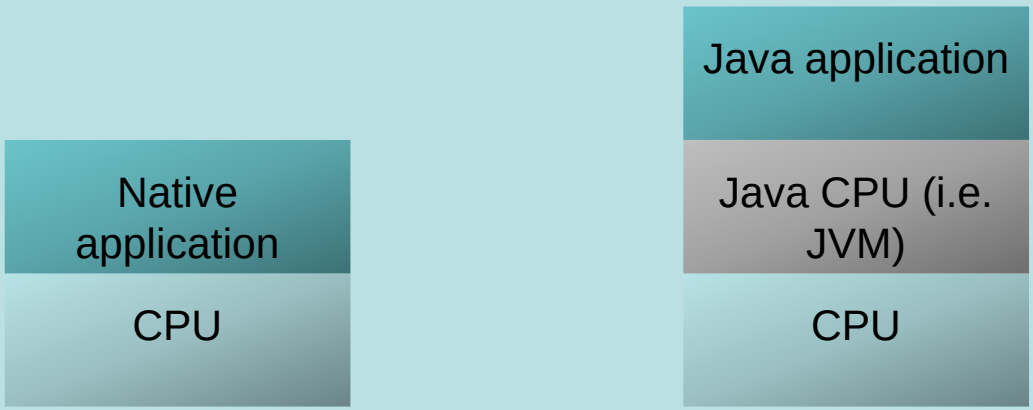

```
class Swap
{
   static void SwapElements(int a[], int k)
   {
      int temp = a[k];
      a[k] = a[k+1];
      a[k+1] = temp;
   } // SwapElements
} // class
```

Java Compiler

```
0:    aload_0
1:    iload_1
2:    iaload
3:    istore_2
4:    aload_0
5:    iload_1
6:    aload_0
7:    iload_1
8:    iconst_1
9:    iadd
10:   iaload
11:   iastore
12:   aload_0
13:   iload_1
14:   iconst_1
15:   iadd
16:   iload_2
17:   iastore
18:   return
```

Java Bytecode

- Java Bytecode (i.e. Java assembly) executes on the Java Virtual Machine (i.e. Java CPU)
  - JVM executes on a real CPU
- JVM is stack-based: uses a stack as the “operation space” rather than registers.



Java Bytecode

- The load and store instructions transfer values between the local variables and the operand stack
- Prefix denotes type: e.g. iload is integer load, aload is array load, iaload is integer array load.
  - Note that aload loads an array reference (pointer) while iaload loads an element of an integer array.
- Common operands can go with the mnemonic: e.g. iload_0 is the same as iload with operand 0.
- Operands can be in the standard form.
  - E.g., iinc *index const* (increment local variable #*index* by signed byte *const*).

Java Bytecode

- The arithmetic instructions (e.g. add, sub) compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack
- Control flow instructions transfer control
  - goto label: takes the PC to the instruction at the given label.
  - icmplt label: compare if less than, if so go to label.
  - icmple label: compare if less than or equal, if so go to label.

Executing Java Bytecode

- A new frame is created each time a method is invoked.
- Each frame has its own array of local variables and its own operand stack.
- The PC points to the currently-executing bytecode.

```
do {
    fetch an opcode;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Executing Java Bytecode

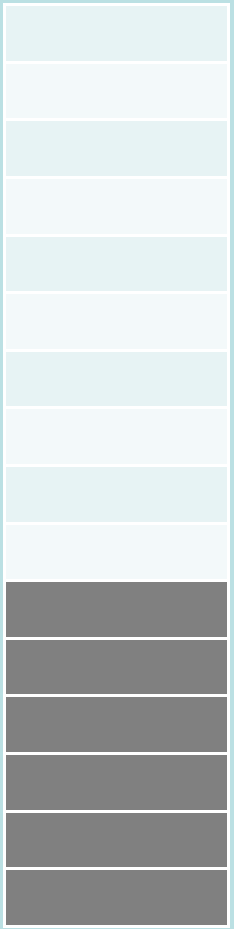
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return

Frame

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



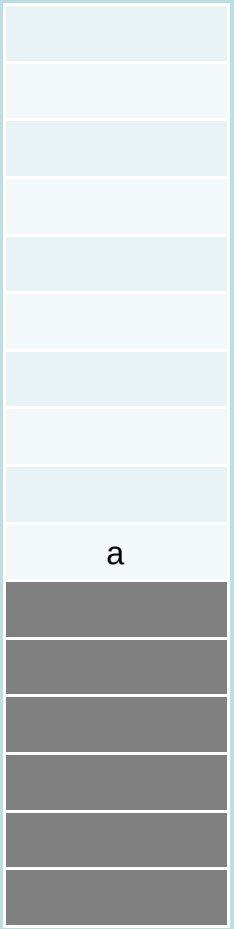
Executing Java Bytecode

aload_0 ← PC
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



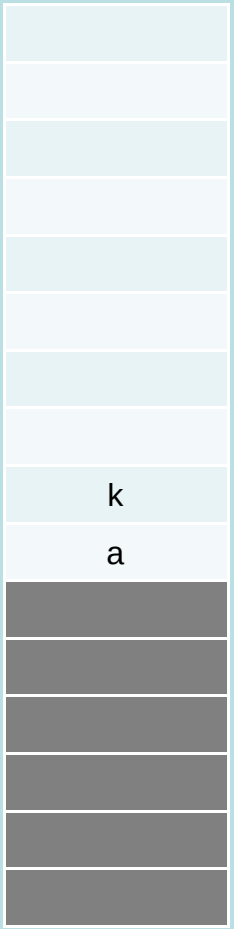
Executing Java Bytecode


```
aload_0
iload_1      <- PC
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



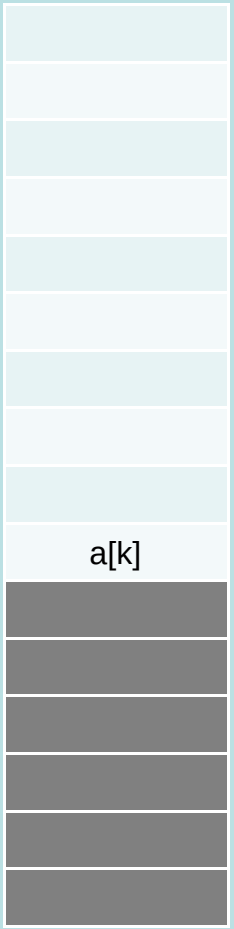
Executing Java Bytecode


```
aload_0
iload_1
iaload      <- PC
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



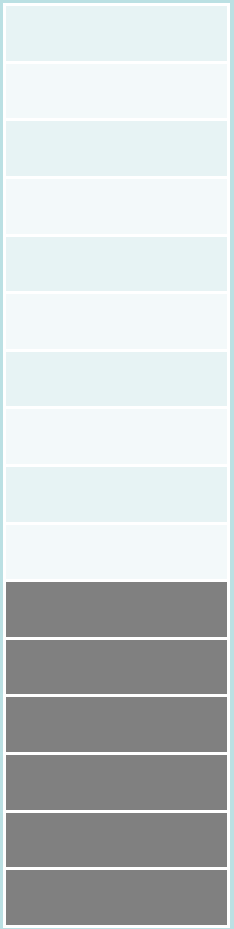
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2   <- PC
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



temp gets the value a[k].

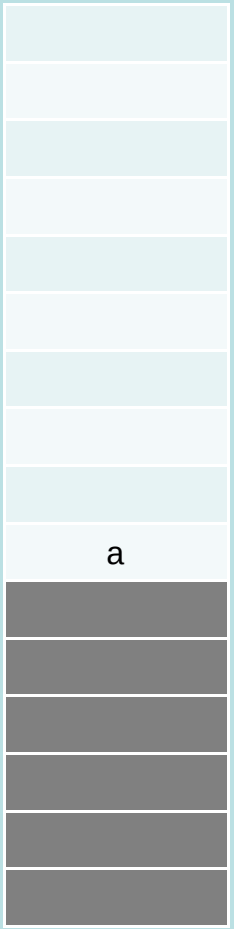
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0   <- PC
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



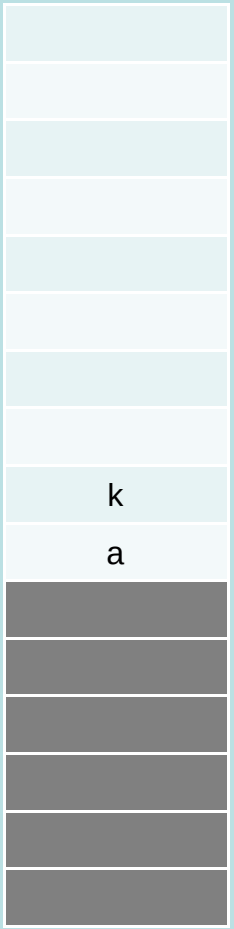
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1   <- PC
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



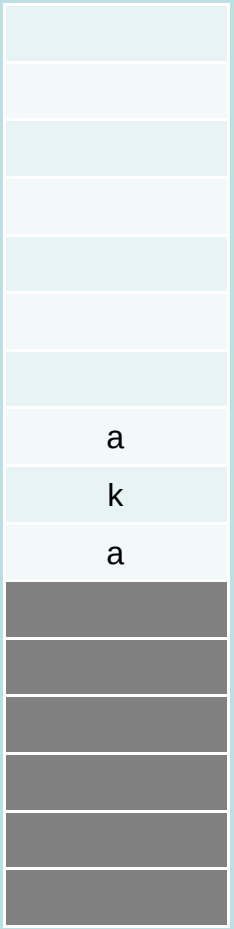
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0   <- PC
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



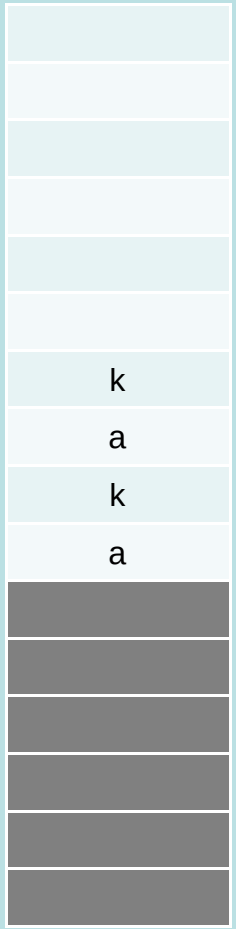
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1   <- PC
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



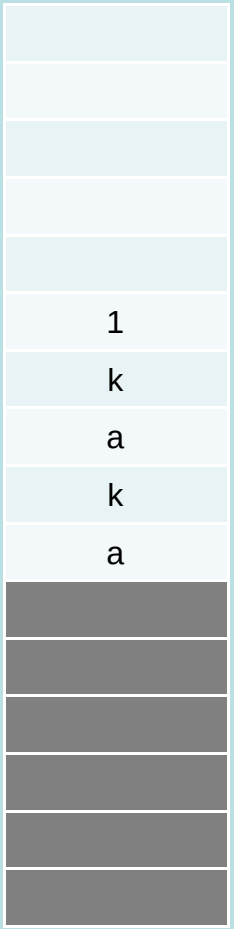
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1   <- PC
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



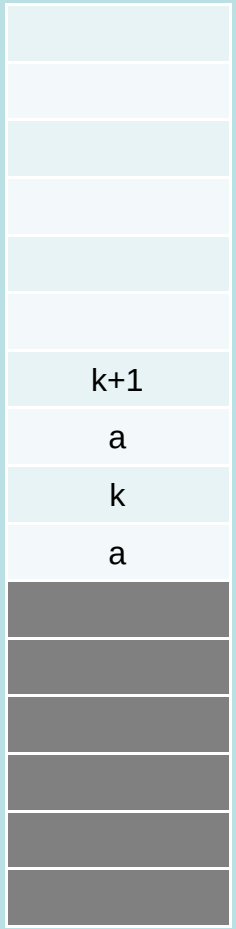
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd        <- PC
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



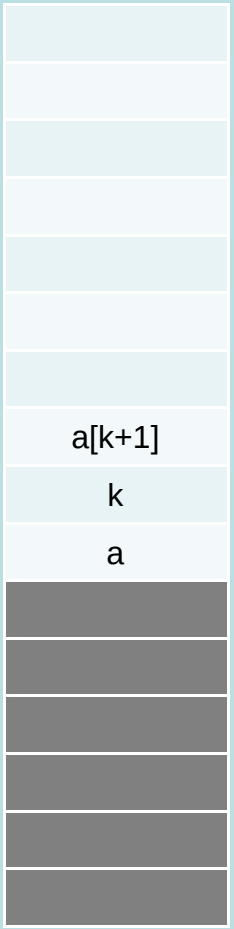
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload      <- PC
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



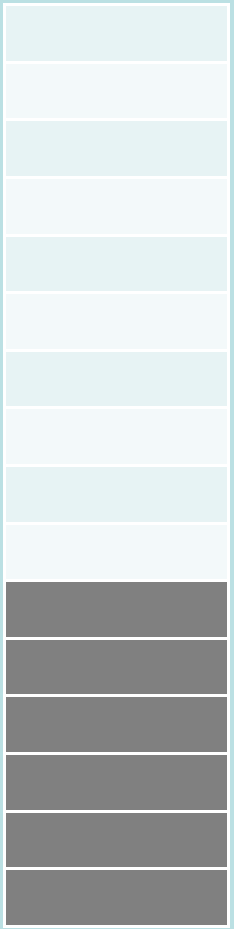
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore   <-- PC
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



a[k].gets the value a[k+1].

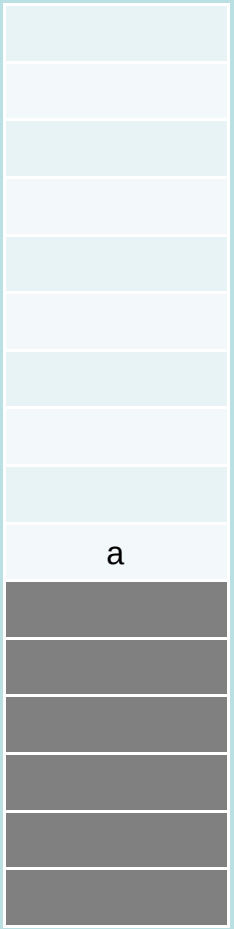
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0   <- PC
iload_1
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



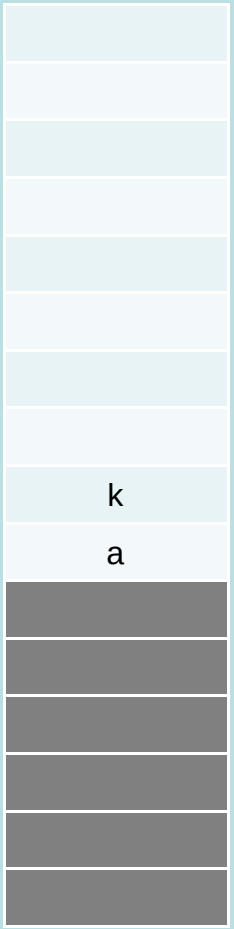
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1   <- PC
iconst_1
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



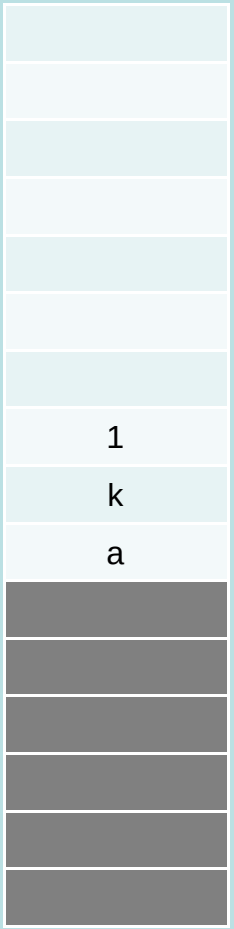
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1   <- PC
iadd
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



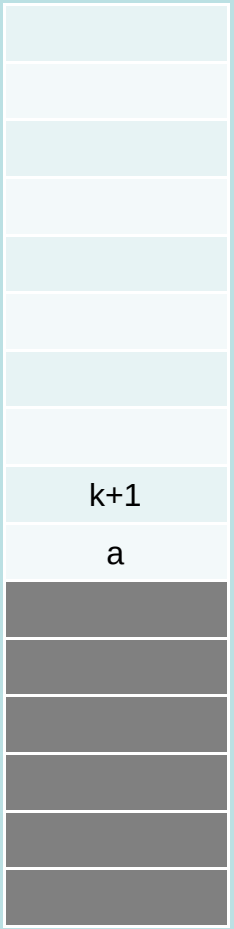
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd        <- PC
iload_2
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



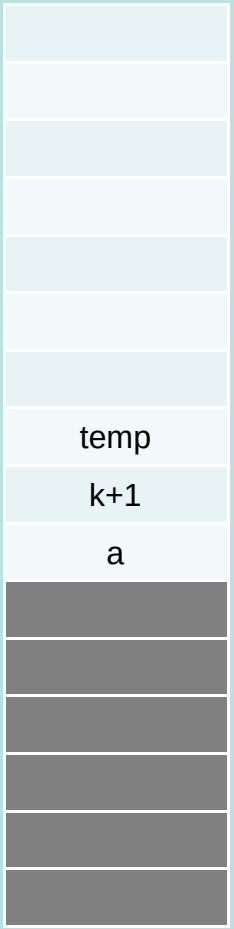
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2   <-- PC
iastore
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



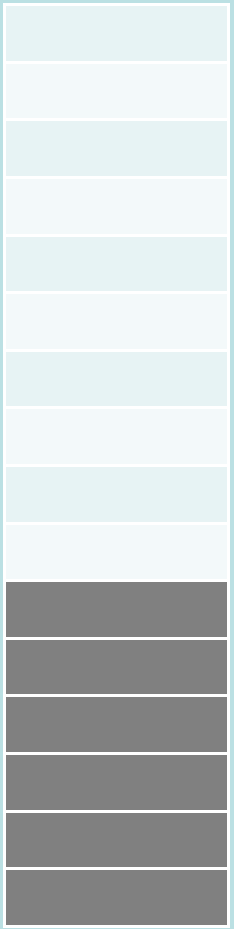
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore   <- PC
return
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



a[k+1].gets the value temp

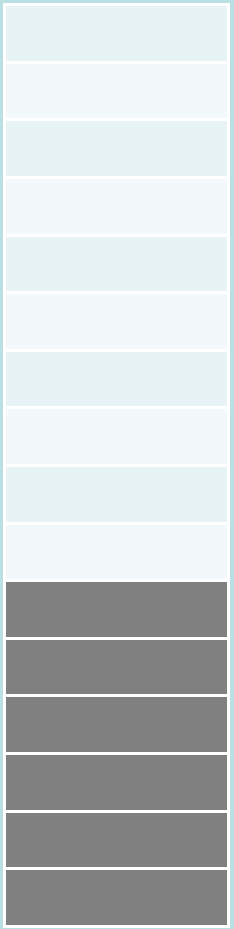
Executing Java Bytecode


```
aload_0
iload_1
iaload
istore_2
aload_0
iload_1
aload_0
iload_1
iconst_1
iadd
iaload
iastore
aload_0
iload_1
iconst_1
iadd
iload_2
iastore
return   <-- PC
```

Local variables

Index	Variable
0	a
1	k
2	temp

Operand stack



Executing Java Bytecode

- JVM is an application (just like Word or Excel) that runs on the host computer.
- JVM reads the class file, picks all the byte codes, and runs them one by one.
  - This is called interpretation, and is inherently slow (why?)
  - Current implementations are not interpreted, but rather compiled on the fly (Just In Time compilation)

```
do {
    fetch an opcode;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Polymorphism

Polymorphic Methods

- In Java, all methods are polymorphic. Meaning that a derived class can override the method to provide implementation specific to the derived class.

```
class Vehicle
{
    public void Move()  { /* Generic move */ }
    public int Wheels() { return 0; }
}

class Bicycle extends Vehicle
{
    public void Move()  { /* Ride */ }
    public int Wheels() { return 2; }
}

class Helicopter extends Vehicle
{
    public void Move()  { /* Fly */ }
}

...

Vehicle v = new Helicopter();
v.Move(); // This is Fly
int myWheels = v.Wheels(); // This is 0
```

Polymorphic Methods

If you have:

```
Vehicle v = new Bicycle();
v.Move();
```

then, the compiler can figure out that Bicycle::Move is the one to use here.

Now, if you have:

```
void ArrangeVehicle(Vehicle v)
{
    v.Move();
}
```

It becomes more difficult for the compiler to figure out which Move to use with *v* here. But with an extensive code analysis this may still be possible

Polymorphic Methods

Now consider this case:

```
Vehicle v;
std::cin >> myint;

switch (myint)
case 1 :
   v= new Helicopter();
   break;
case 2 :
   v= new Bicycle();
   break;
default :
   v= Car();
}

v.Move();
```

In general, the compiler cannot figure out the correct method to use (even with extensive code analysis).

The solution is vTables: this is a smart way for an object to carry with it information that identifies the methods that are applicable to the object.

Polymorphic Methods


```
class Vehicle
{
    *__vtable_pointer; // points to Vehicle VTable
    public void Move()  { /* Generic move */ }
    public int Wheels() { return 0; }
}

class Bicycle extends Vehicle
{
    *__vtable_pointer; // points to Bicycle VTable
    public void Move()  { /* Ride */ }
    public int Wheels() { return 2; }
}

class Helicopter extends Vehicle
{
    *__vtable_pointer; // points to Helicopter VTable
    public void Move()  { /* Fly */ }
}

...

Vehicle v = new Helicopter();
v.Move(); // This is Fly
int myWheels = v.Wheels(); // This is 0
```

Vehicle VTable
MoveFnPtr (points to Vehicle::Move)
WheelsFnPtr (points to Vehicle::Wheels)

Bicycle VTable
MoveFnPtr (points to Bicycle::Move)
WheelsFnPtr (points to Bicycle::Wheels)

Helicopter VTable
MoveFnPtr (points to Helicopter::Move)
WheelsFnPtr (points to Vehicle::Wheels)

v.Move() will be translated as
v.__vtable_pointer.MoveFnPtr;

v.Wheels() will be translated as
v.__vtable_pointer.WheelsFnPtr.

Polymorphic Methods

- Executing polymorphic method requires an extra vtable lookup.
- In Java, all methods are polymorphic*. So those that are not functionally polymorphic incur the overhead of vtable lookup.
- In C++ (and C#), a polymorphic method needs to be explicitly marked so using the keyword virtual.
  - virtual void Move() { /* Generic move */ }
  - Thus, non-polymorphic methods do not incur the overhead of polymorphic methods

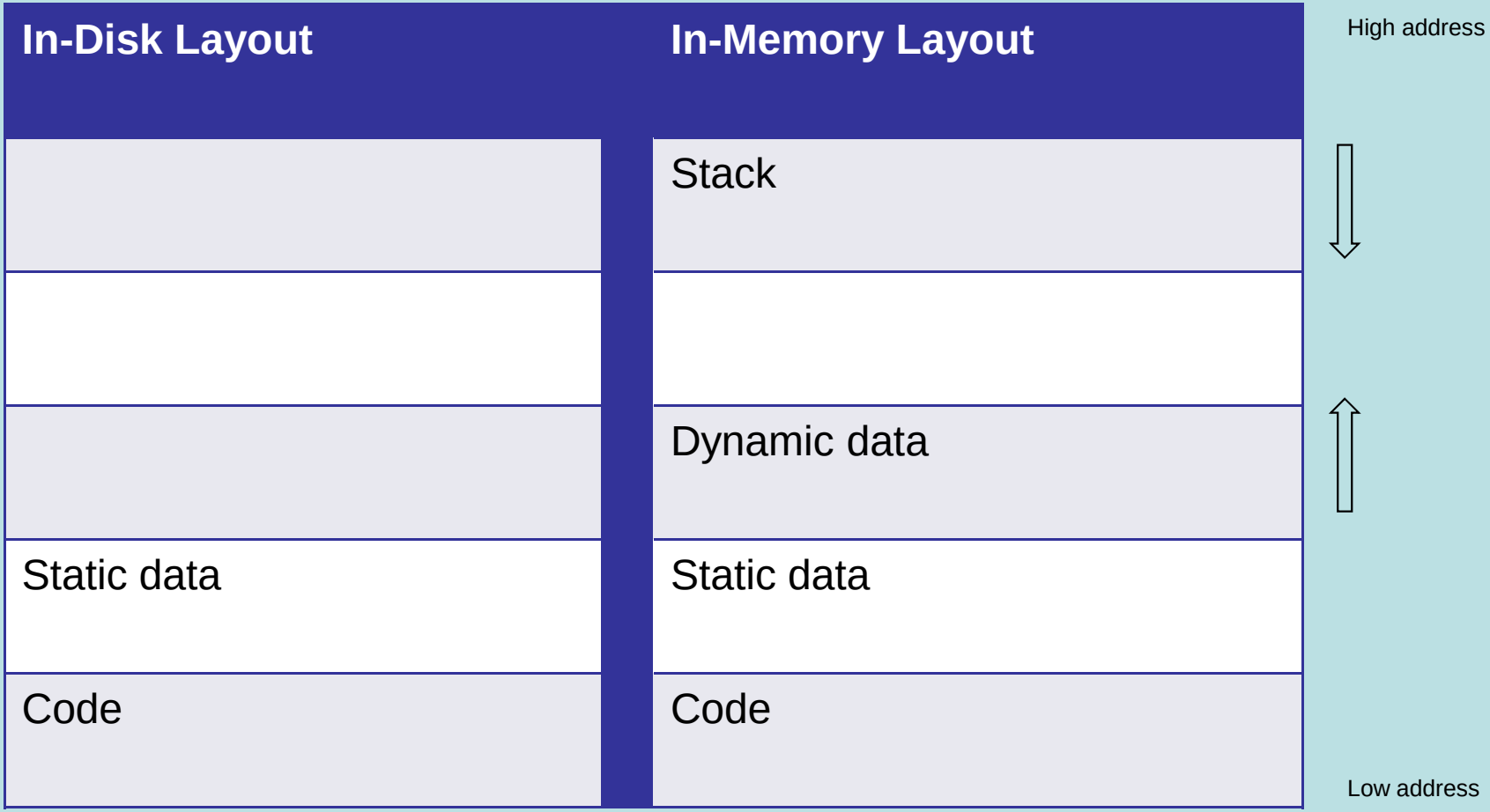
*Except those that cannot be inherited (e.g. those marked private or final).

Executing Programs

Program layout

- A program contains both instructions and data
  - The instructions that define the sequence of actions
  - Data that the compiler knows about and thus can pre-allocate space – this is static data
  - Data that gets set and used only at runtime; the compiler cannot determine the space required – this is dynamic data
    - *This is typically data acquired through the new operator*
- The program layout in disk (e.g. some EXE file stored on the disk) contains the code (i.e. instructions) and the static data only.
- When loaded into the memory for execution, the OS lays out the dynamic data as well.

Program Layout



Program Execution

- The OS initializes the registers required for program execution
  - It also pushes the command line arguments to the stack.
- It then passes control to the program by setting the PC (program counter) to the address of the entry point of the program.

CPU & Memory: The Big Picture

Register File
Register #0
Register #1
Register #2
Register #N

Program Counter

Address	Instruction
30000401	AND R0, R0, #0
30000402	ADD R0, R3, #2
30000403	LDR R5, R2, #1
30000404	LDR R6, R2, #2
30000405	STR R6, R5, #1
30000406	STR R5, R6, #2
30000407	BRzp 30000401
…	
…	

THE UNIVERSITY OF AUCKLAND

NEW ZEALAND

Te Whare Wānanga o Tāmaki Makaurau

The University of Auckland

New Zealand

Talking to the Kernel

System Calls

- System calls are calls to the OS functions.
  - These request special OS services such as printing, file access (opening, reading, writing or closing) , memory allocation/de-allocation, etc.
- System calls pass information to the OS – this includes the system function ID that we like to execute, and the parameters to the system function.

System Calls

- On LC-3 the instruction to talk to the OS is TRAP.
  - The function ID in this case is part of the instruction itself – the last 8 bits.
  - Register R0 is used to pass argument or return result.

```
            .ORIG x3000
            LDR2, TERM
            LDR3, ASCII
AGAIN       TRAP x23            ; IN(R0)
            ADD R1, R2, R0
            BRz
EXIT        ADDR0, R0, R3       ; Change to lowercase
            TRAP x21            ; OUT(R0) to monitor
            BRnzp AGAIN;
TERM        .FILL xFFC9         ; -‘7’
ASCII       .FILL x0020         ; lowercase bit
EXIT        TRAP x25            ; HALT
            .END
```

System Calls


```
#define   SYS_read      3
#define   SYS_write     4
#define   SYS_open      5
#define   SYS_close     6
#define   SYS_unlink    10
#define   SYS_chdir     12
#define   SYS_brk       17      // sets the break value
#define   SYS_stat      18      // get information about a file
#define   SYS_lseek     19
#define   SYS_getpid    20
#define   SYS_nice      34
#define   SYS_sbrk      69      // adds incr bytes to the break value
```

Selected system call constants from sys/syscall.h

System Calls

- At the point of executing TRAP
  - The current PC and the state (registers & stack) are saved
  - The entry point of the system function becomes the new PC
  - The system function is completed
  - The saved PC and state are restored
  - Register R0 contains the return value from the system function

Caches

Caches

- Cache is a small local memory holding items of interest
  - Items of interest include items we used in the recent past and items we may use in the future.
  - Example: a browser cache that holds the web content you looked at recently
- Why is such a cache useful?
  - Avoiding re-fetching items over a (slow) network
  - Processors are a lot faster than memory

Caches

- Assume 5 slots where you can keep numbered (i.e. addressable) items.
  - Items can be anything so long as they have a unique sequence number (i.e. address) with which they can be identified. Example: customer name that is identified through a customer id.
  - We pick items using their ids (e.g. pick a customer using the customer id).
- You want to keep in the 5 slots some items that have been used in the recent past.
  - These 5 slots make up our cache.
- What process should we follow to make our cache work?

Example

Id	Item

Cache

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

Example

Id	Item
10	Sue
3	Allan
7	Chris
6	Ben
8	Mitch

Cache

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

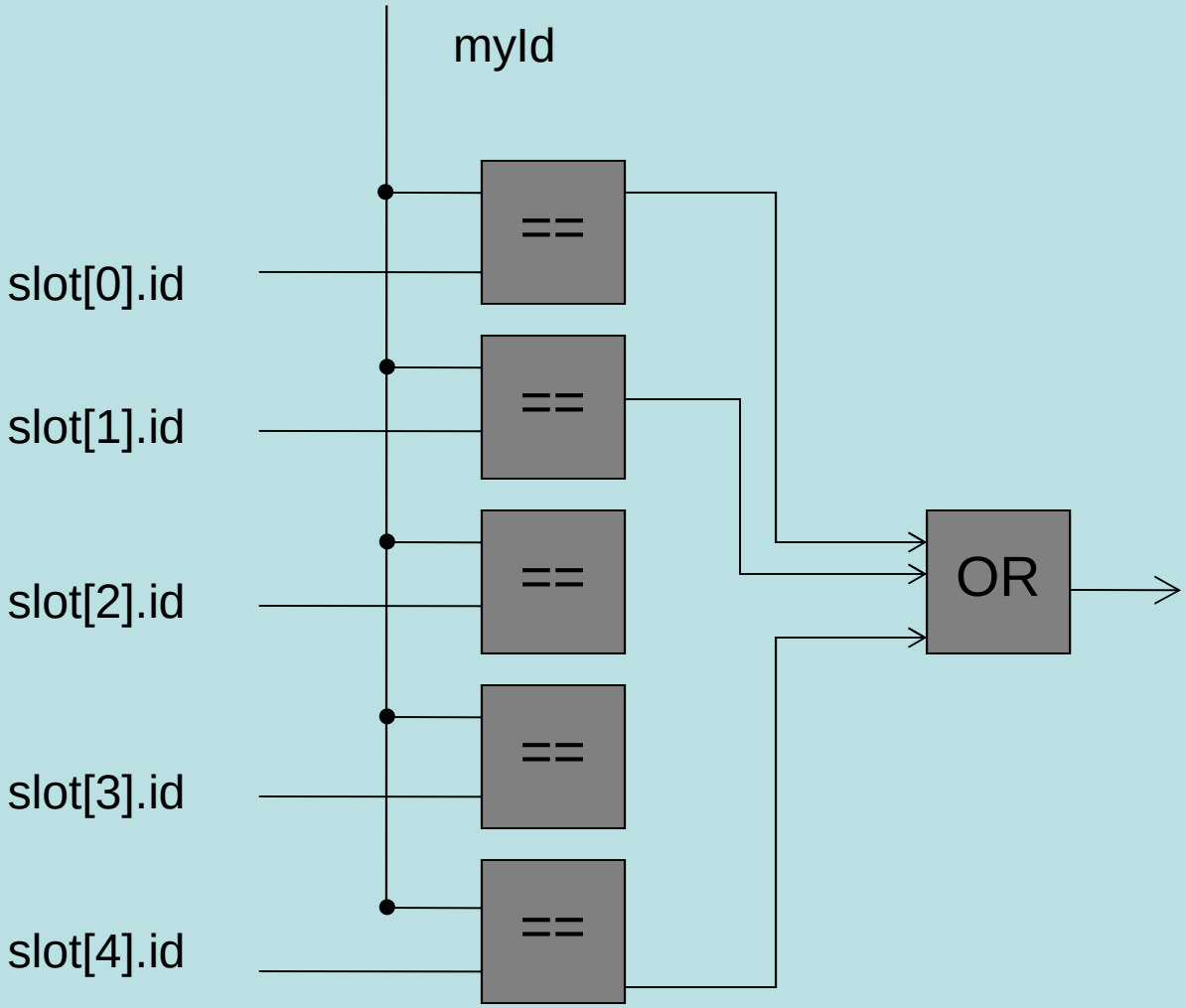
List of Items

C++ Model of the Cache


```
#include <string>
#include <vector>

class CacheEntry
{
   int id;
   std::string item;
};

const int NoOfCachedItems = 5;
static std::vector<CacheEntry> cacheSlot(NoOfCachedItems);

bool DoesCacheContain(const int myId) {
   bool rval = false;
   // fill in the detail: what is the time-complexity of this?

   return rval;
}
```

Issues with our Cache

- Searching through the cache for a given id is expensive.
  - What is the time complexity of the search, in terms of the size of the cache?
  - How do we do the search in hardware? What is the space complexity?


myId
slot[0].id
slot[1].id
slot[2].id
slot[3].id
slot[4].id
==
==
==
==
==
OR

Issues with our Cache

- Is there a way to reduce the complexity (i.e. speed up the search)?

Direct-Mapped Caches

- In our first attempt, any id from the list can go to any slot in the cache – thus there was a need to search *every* slot in the cache, when we looked for an id.
- Now, we have a variant of the cache where an id can go to *just one* predefined slot in the cache.
  - Id 0 will go to slot 0 (and only slot 0); id 1 will go to slot 1 (and only slot 1); etc;
  - Id 5 will go to slot 0 (and only slot 0); id 6 will go to slot 1 (and only slot 1); etc;
  - This is called a direct-mapped cache. In contrast, our first cache is called a fully-associative cache (where an id can go to any slot).

Direct-Mapped Caches

Id	Item

Cache

Which slot in the cache each id of the data will go to?

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

Direct-Mapped Cache: C++ Model

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

Do you see any improvement in the search in direct-mapped caches?

Direct-Mapped Cache: C++ Model

Do you see any improvement in the search in direct-mapped caches?

```
bool
DoesCacheContain(const int myId)
{
   bool rval = false;
   int where2look = myId % CacheSize; // % is the mod operation
   if ( myId == cacheSlot[where2look].id )
   {
      rval = true;
   }

   return rval;
}
```

Direct-Mapped Cache

- There is now only one slot where a particular id can be. We don't therefore need to search every slot.
- We improve the complexity.
  - Well, do we?

Direct-Mapped Cache

- There are two problems with our direct-mapped cache.
- What price is this:

  int where2look = myId % CacheSize
- How do we do this mod better?

Direct-mapped Caches

- In hardware, we can replace the mod operation by a mask operation provided that the denominator is a power of 2.
- 11011010 AND 00000011
- Direct-mapped caches in computers therefore have number of entries that is always a power of 2.
  - There are very many other things that are powers of 2 when it comes to computing hardware. Can you think of some others?
  - What is magical about 2?

Direct-mapped Caches

- Suppose we have a direct-mapped cache of size 4 (i.e. with 4 slots).
- Ids 0, 4, 8, 12, … map to slot 0; ids 1, 5, 9, 13, … map to slot 1; ids 2, 6, 10, 14, … map to slot 2; and ids 3, 7, 11, 15, … map to slot 3.

- Exercise: Imagine the cache has no valid entries to start with (i.e. cold start). Work out if the following accesses are hits or misses in the cache.
  - 1, 2, 3, 4, 2, 3, 1, 4, 5
  - 1, 4, 1, 4, 1, 4, 1, 4, 1
  - 1, 5, 1, 5, 1, 5, 1, 5, 1

Direct-Mapped Caches

	Id	Item
0		
1		
2		
3		

Cache

ids 0, 4, 8, 12, … map to slot 0;
ids 1, 5, 9, 13, … map to slot 1;
ids 2, 6, 10, 14, … map to slot 2;
ids 3, 7, 11, 15, … map to slot 3.

- Exercise: Imagine the cache has no valid entries to start with (i.e. cold start). Work out if the following accesses are hits or misses in the cache.
  - 1, 2, 3, 4, 2, 3, 1, 4, 5
  - 1, 4, 1, 4, 1, 4, 1, 4, 1
  - 1, 5, 1, 5, 1, 5, 1, 5, 1

Direct-mapped Caches

- Direct-mapped caches can lead to thrashing: two items go in and out all the time even though there is plenty of space available.
  - A cache miss resulting from thrashing is a conflict miss.
- Is there a way to reduce thrashing?
- Is there a way to get rid of thrashing?
  - Fully-associative cache – our first cache where an id can go to any slot
- Is there a way to reduce thrashing?

Direct-Mapped Caches

Id	Item

Cache

Which slot in the cache each id of the data will go to?

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

Set-Associative Caches

Id	Item	Id	Item

Cache

We have two half-size direct-mapped caches side-by-side. Which slot in the cache each id of the data will go to?

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

Set-Associative Caches

Id	Item	Id	Item

Cache

We have 2 slots where each data id can go to. This reduces thrashing (does it?) but does not prevent it.

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

Fully Associative Caches

Id	Item
10	Sue
3	Allan
7	Chris
6	Ben
8	Mitch

Cache

Id	Item
0	Geoff
1	Joe
2	Nora
3	Allan
4	Rick
5	Gary
6	Ben
7	Chris
8	Mitch
9	Dan
10	Sue
11	Fred

List of Items

Fully Associative Caches

- Which of the items to throw out when we need to make room for a new item?
- A replacement policy determines this.

- The most commonly used policy is LRU (least-recently-used): every slot has a counter; every time a slot is accessed, a global counter is incremented and the value stored in the slot counter; the slot with the lowest counter value is chosen for replacement.

Fully-Associative Caches: Exercise

- Suppose we have a fully-associative cache of size 4 (i.e. with 4 slots).

- Exercise: Imagine the cache has no valid entries to start with (i.e. cold start). Work out if the following accesses are hits or misses in the cache.
  - 1, 2, 3, 4, 2, 3, 1, 4, 5
  - 1, 4, 1, 4, 1, 4, 1, 4, 1
  - 1, 5, 1, 5, 1, 5, 1, 5, 1
  - 2, 5, 1, 3, 4, 2, 5, 1, 3

Cache Exercises

Direct-Mapped Caches

Consider a direct-mapped cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	2	1	0	1	4	0	2	1	3	7	3	7	3	7	3	7	2	1	0	4
Hit/Miss																				

Fully-Associative Caches

Consider a fully-associative cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

The cache uses an LRU replacement policy: every slot has a counter; every time a slot is accessed, a global counter is incremented and the value stored in the slot counter; the slot with the lowest counter value is chosen for replacement.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	2	1	0	1	4	0	2	1	3	7	3	7	2	1	0	3	7	2	1	0
Hit/Miss																				

Direct-Mapped Caches

Consider a direct-mapped cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss																				

Direct-Mapped Caches

Slot #	Address	Item
0		
1		
2		
3		

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss																				

Direct-Mapped Caches

Slot #	Address	Item
0		
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M																			

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M																		

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H																	

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H																

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H															

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M														

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M													

Direct-Mapped Caches

Slot #	Address	Item
0	8	
1		
2		
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M												

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1		
2		
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M											

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1		
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M										

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M									

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	7	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M								

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M							

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H						

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H	H					

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H	H	H				

Direct-Mapped Caches

Slot #	Address	Item
0	0	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H	H	H	M			

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H	H	H	M	M		

Direct-Mapped Caches

Slot #	Address	Item
0	0	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H	H	H	M	M	M	

Direct-Mapped Caches

Slot #	Address	Item
0	4	
1	1	
2	2	
3	3	

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	M	M	M	M	M	M	M	H	H	H	M	M	M	M

Fully-Associative Caches

Consider a fully-associative cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

The cache uses an LRU replacement policy: every slot has a counter; every time a slot is accessed, a global counter is incremented and the value stored in the slot counter; the slot with the lowest counter value is chosen for replacement.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss																				

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0			
1			
2			
3			

Global Counter
Value

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss																				

Fully-Associative Caches

Consider a fully-associative cache of size 4. Each slot in the cache can have just one item (i.e. the line size is 1 item). The cache is empty to start with.

The cache uses an LRU replacement policy: every slot has a counter; every time a slot is accessed, a global counter is incremented and the value stored in the slot counter; the slot with the lowest counter value is chosen for replacement.

Work out if the following accesses to the given addresses are hits or misses. Each access is numbered with a sequence id for your convenience.

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss																				

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0			
1			
2			
3			

Global Counter
Value

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss																				

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		0
1			
2			
3			

Global Counter	
Value	0

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M																			

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		0
1	8		1
2			
3			

Global Counter	
Value	1

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M																		

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		2
1	8		1
2			
3			

Global Counter	
Value	2

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H																	

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		2
1	8		3
2			
3			

Global Counter	
Value	3

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H																

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		4
1	8		3
2			
3			

Global Counter	
Value	4

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H															

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		4
1	8		3
2	3		5
3			

Global Counter	
Value	5

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M														

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		6
1	8		3
2	3		5
3			

Global Counter	
Value	6

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H													

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		6
1	8		3
2	3		7
3			

Global Counter	
Value	7

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H												

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		6
1	8		3
2	3		7
3	4		8

Global Counter	
Value	8

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M											

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	7		6
1	~~8~~ 2		9
2	3		7
3	4		8

Global Counter	
Value	9

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M										

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	~~7~~ 1		10
1	2		9
2	3		7
3	4		8

Global Counter	
Value	10

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M									

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	1		10
1	2		9
2	~~3~~ 7		11
3	4		8

Global Counter	
Value	11

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M								

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	1		10
1	2		9
2	7		11
3	4 3		12

Global Counter	
Value	12

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M							

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	1		10
1	~~2~~ 4		13
2	7		11
3	3		12

Global Counter	
Value	13

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M						

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	~~1~~ 2		14
1	4		13
2	7		11
3	3		12

Global Counter	
Value	14

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M	M					

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	2		14
1	4		13
2	~~7~~ 1		15
3	3		12

Global Counter	
Value	15

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M	M	M				

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	2		14
1	4		13
2	1		15
3	~~3~~ 0		16

Global Counter	
Value	16

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M	M	M	M			

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	2		14
1	4		17
2	1		15
3	0		16

Global Counter	
Value	17

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M	M	M	M	H		

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	2		14
1	4		17
2	1		15
3	0		18

Global Counter	
Value	18

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M	M	M	M	H	H	

Fully-Associative Caches

Slot #	Address	Item	Local Counter
0	2		14
1	4		19
2	1		15
3	0		18

Global Counter	
Value	19

Sequence Id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Address	7	8	7	8	7	3	7	3	4	2	1	7	3	4	2	1	0	4	0	4
Hit/Miss	M	M	H	H	H	M	H	H	M	M	M	M	M	M	M	M	M	H	H	H

THE UNIVERSITY OF AUCKLAND

NEW ZEALAND

Te Whare Wānanga o Tāmaki Makaurau

The University of Auckland

New Zealand

CPU Caches

CPU Caches

- Caches bridge the speed gap between the CPU and the main memory.
- Small fast memory close to the CPU holding data the CPU is likely to access in the near future.
- Caches rely on two properties of the access patterns of most programs:
  - **temporal locality** - if a memory location is accessed once, it is likely to be accessed again soon
  - **spatial locality** - if a memory location is accessed then nearby memory locations are also likely to be accessed.

CPU Caches

- In order to exploit spatial locality, caches operate on several words at a time, called a *cache line* or a cache block.
  - Main memory reads and writes are whole cache lines.
  - This takes advantage of the principle of locality of reference: if one location is read then nearby locations (particularly following locations) are likely to be read soon afterwards.

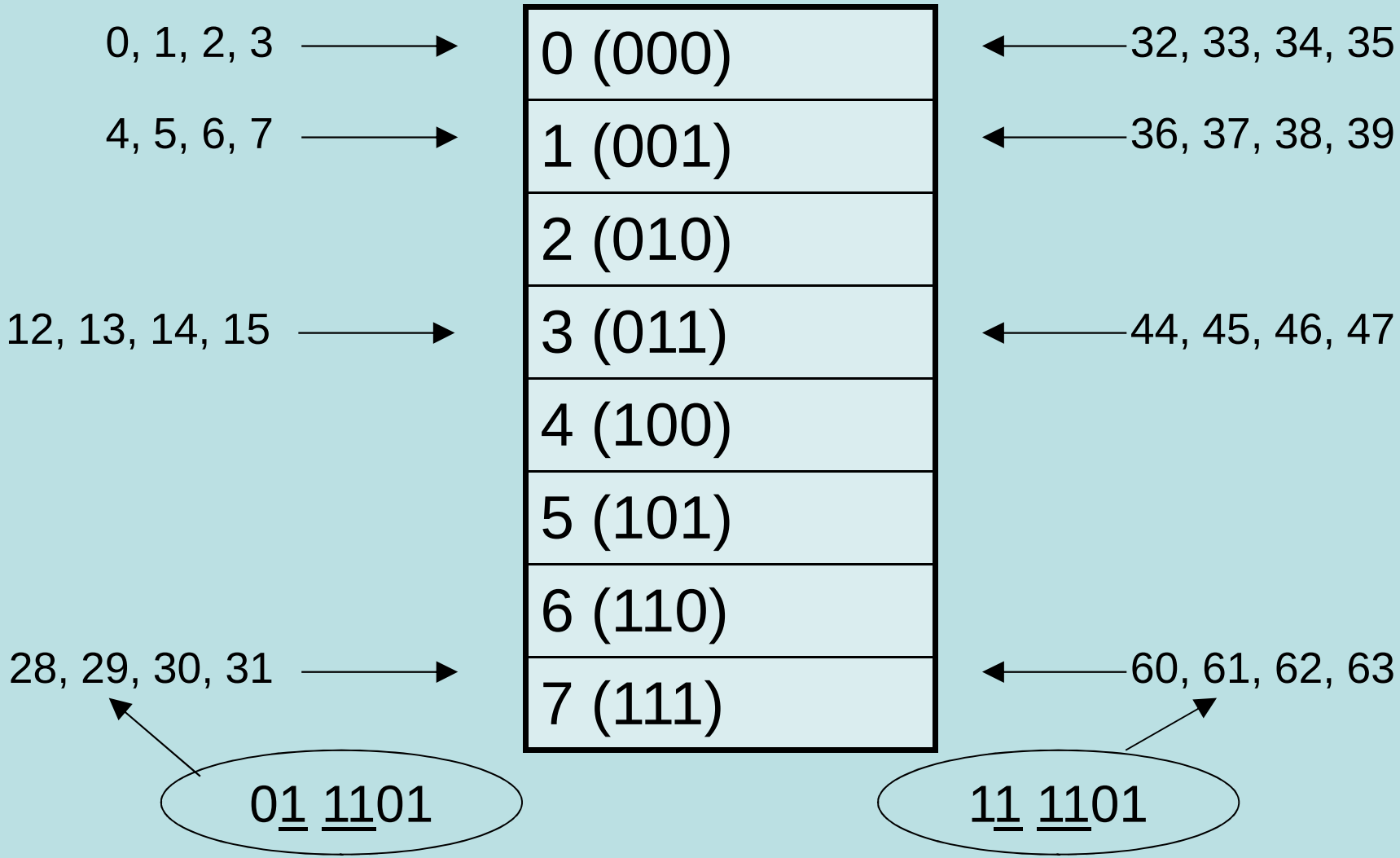
Cache Hits

- When data is read from, or written to, main memory, a copy is also saved in the cache, along with the associated main memory address.
- The cache monitors addresses of subsequent reads to see if the required data is already in the cache. If it is (a *cache hit*), then it is returned immediately and the main memory read is aborted (or not started).

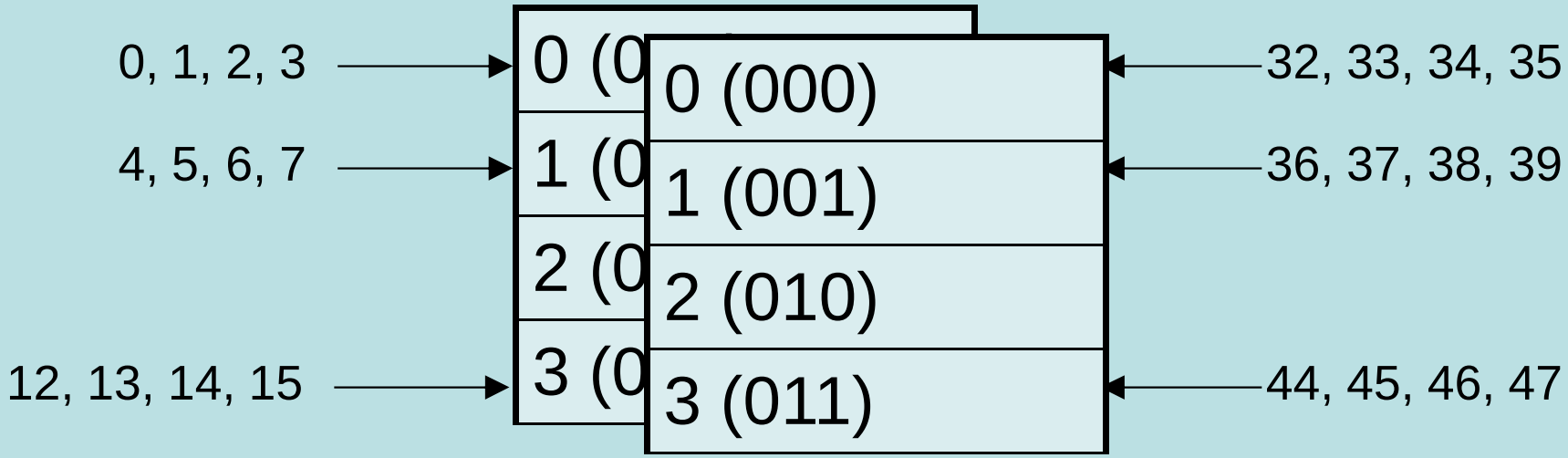
Cache Miss & Miss Penalty

- If the data is not cached (a *cache miss*) then it is fetched from main memory and also saved in the cache.
- The time it takes to service a memory request under a cache miss is called a *miss penalty*.

Direct-Mapped Caches



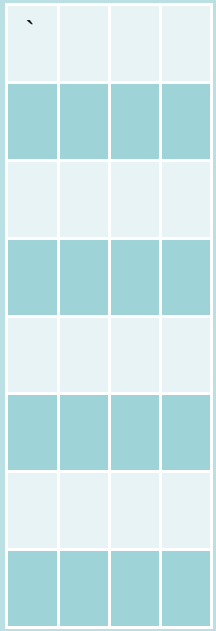
Set-Associative Caches



A memory line can map to any of the two possible cache lines. When both cache lines are being used and we need to bring a line from memory, which of the two lines do we throw out? A replacement policy determines this.

A fully-associative cache is where we have any memory line can map to any cache line (an *n*-to-1 mapping).

Array Layout in Memory



Layout is row by row.

Two ways to sum a 2-D array


```
const int MAXX = 5;
const int MAXY = 4;
int array[MAXX][MAXY]; // init
int sum = 0;
for ( int i = 0; i < MAXX; ++i )
{
   for ( int j = 0; j < MAXY; ++j )
   {
      sum += array[i][j];
   }
}
```

```
const int MAXX = 5;
const int MAXY = 4;
int array[MAXX][MAXY]; // init
int sum = 0;
for ( int j = 0; j < MAXY; ++j )
{
   for ( int i = 0; i < MAXX; ++i )
   {
      sum += array[i][j];
   }
}
```

Download the sample code ‘array2d.cpp’ and experiment with it.

Two ways to sum a 2-D array


```
const int MAXX = 5;
const int MAXY = 4;
int array[MAXX][MAXY]; // init
int sum = 0;
for ( int i = 0; i < MAXX; ++i )
{
   for ( int j = 0; j < MAXY; ++j )
   {
      sum += array[i][j];
   }
}
```

a(0,0), a(0,1), a(0,2), a(0,3),
a(1,0), a(1,1), a(1,2), a(1,3),,
a(2,0), a(2,1), ...
Access order = memory layout order

```
const int MAXX = 5;
const int MAXY = 4;
int array[MAXX][MAXY]; // init
int sum = 0;
for ( int j = 0; j < MAXY; ++j )
{
   for ( int i = 0; i < MAXX; ++i )
   {
      sum += array[i][j];
   }
}
```

a(0,0), a(1,0), a(2,0), a(3,0), a(4,0),
a(0,1), a(1,1), a(2,1), a(3,1), a(4,1),
a(0,2), a(1,2), ...
Access order != memory layout order

Access Sequence #1

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

New Zealand

The University of Auckland

Array			
a(0,0)	a(0,1)	a(0,2)	a(0,3)
a(1,0)	a(1,1)	a(1,2)	a1,3)
a(2,0)	a(2,1)	a2,2)	a(2,3)
a(3,0)	a(3,1)	a(3,2)	a(3,3)
a(4,0)	a(4,1)	a(4,2)	a(4,3)
a(9,0)	a(9,1)	a(9,2)	a(9,3)

Cache			
`			

Memory
a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)

a(0,0), a(0,1), a(0,2), a(0,3),
a(1,0), a(1,1), a(1,2), a(1,3),,
a(2,0), a(2,1), ...
Access order = memory layout order

Work out the cache hits/misses for the memory accesses.

Access Sequence #2

Array			
a(0,0)	a(0,1)	a(0,2)	a(0,3)
a(1,0)	a(1,1)	a(1,2)	a1,3)
a(2,0)	a(2,1)	a2,2)	a(2,3)
a(3,0)	a(3,1)	a(3,2)	a(3,3)
a(4,0)	a(4,1)	a(4,2)	a(4,3)
a(9,0)	a(9,1)	a(9,2)	a(9,3)

Cache			
`			

Memory
a(0,0)
a(0,1)
a(0,2)
a(0,3)
a(1,0)
a(1,1)
a(1,2)
a(1,3)
a(2,0)
a(2,1)
a(2,2)
a(2,3)
a(3,0)

a(0,0), a(1,0), a(2,0), a(3,0), a(4,0),
a(0,1), a(1,1), a(2,1), a(3,1), a(4,1),
a(0,2), a(1,2), ...
Access order != memory layout order

Work out the cache hits/misses for the memory accesses.

Cache Write

- When we write, should we write to cache or memory?
- Write-through cache – write to both cache and main memory. Cache and memory are always consistent
- Write-back cache – write only to cache and set a “dirty bit”. When the block gets replaced from the cache, write it out to memory.

Categorizing cache misses

- **Compulsory miss**: First ever reference to a cache line.
- **Capacity miss**: Miss due to the cache not being big enough to hold the current data set. If the number of active lines is more than the cache can contain, capacity misses take place.

Categorizing cache misses

- **Conflict miss**: The required line was previously there in the cache, but was replaced by another line which happened to map to the same cache location.
  - Conflict misses take place because of limited or zero associativity when lines must be discarded in order to accommodate new lines which are mapped to the same line in the cache.
  - A miss occurs when the replaced line needs to be accessed again.

Categorizing cache misses

- **Coherency miss**: The required line was previously there in the cache, but was invalidated by another processor core.
- There are two types of coherency misses:
  - True-sharing miss
  - False-sharing miss

False Sharing

- False sharing occurs when two items that happen to map to the same cache line are used by two different processor cores concurrently, thus resulting in a number of cache invalidation operations.
  - Example: Item A and item B map to the same cache line. Item A is written by processor core 1 while B is written by processor core 2. Even though processor cores 1 and 2 do not share A or B, there will still be invalidation operations just because A and B map to the same cache line.

Exercise

Consider the following code fragment:

```
01      double t [8]; double a[1024][8];    // t: total
02      int i, j;
03      ....
04
05      for ( int i = 0; i < 8; ++i ) {
06          for ( int j = 0; j < 1024; ++j ) {
07              t[i] += a[j][i];
08          }
09      }
```

This code is to be executed in parallel on an 8 core system by assigning each of the outer loop iteration to a separate core. Assuming that **sizeof**(**double**) is 8 and the cache line size is 64 bytes, what performance problems the parallel execution may face? How can you fix these problems? Re-write the code fragment with the problems fixed.

Exercise

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
...	...	...	...	...	...	...	...

Memory organized as cache lines.

Exercise

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
...	...	...	...	...	...	...	...

Memory organized as cache lines.

Reducing False Sharing

- Use local (e.g., register) variables. Write to the memory just at the end.
- Reduce the cache line size. In one extreme where a cache line is no bigger than the size of the object, there won't be any false sharing. The problem is that this throws away the principle of spatial locality. This leads to a drastic increase in compulsory misses.

Reducing False Sharing

- Use padding to separate out objects into different cache lines. This requires analysis and identification of the false sharing, and knowledge of the size of the cache line. If the code is to be run on a different system with a different cache line size, code needs re-tuning. There is wasted memory/cache space. Padding also leads to an increase in capacity misses.

Reducing False Sharing

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

- With some effort, the program could be re-structured to minimize false sharing. This approach is program-specific.

Further Work

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

- Google and Wiki are your mates. Dig into the following topics:
  - Assembly language programming
  - The Java Virtual Machine Specification
- Install the Java disassembler *jad* and try it out. Use option –*dis* to see the bytecodes
- Try the floating-point calculators at http://babbage.cs.qc.edu/IEEE-754/
- Read the relevant sections of the ‘Memory Hierarchy’ chapter of Patterson & Hennessy.