CPSC 213

Introduction to Computer Systems

Unit 1a

Numbers and Memory

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 Java 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

Languages and Tools

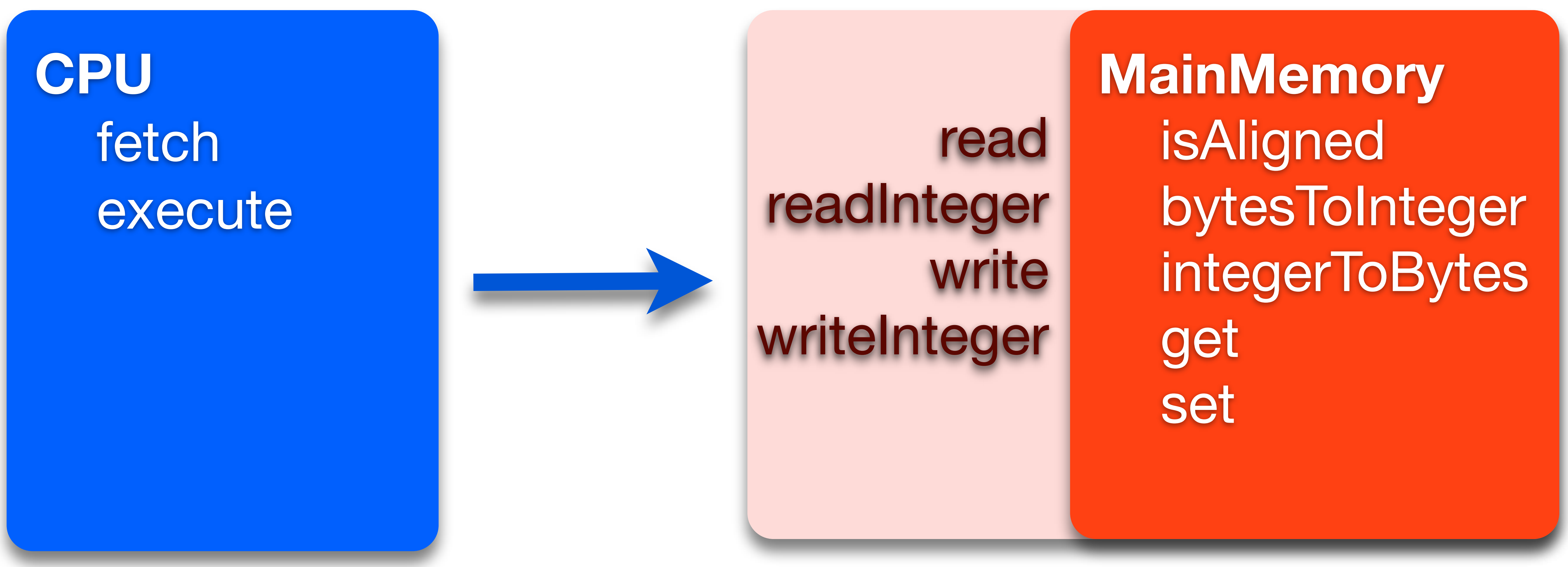
- SM213 Assembly
  - you will trace, write, read
  - use SM213 simulator to trace and execute
- Java
  - you will read, write
  - use Eclipse IDE to edit, compile, debug, run
  - SM213 simulator written in Java; you will implement specific pieces
- C
  - you will read, write
  - gcc to compile, gdb to debug, command line to run

Lab/Assignment 1

- SimpleMachine simulator
  - load code into Eclipse and get it to build/run
  - write and test `MainMemory.java`
    - `get`
    - `set`
    - `isAccessAligned`
    - `bytesToInteger`
    - `integerToBytes`

The Main Memory Class

- The SM213 simulator has two main classes
  - CPU implements the fetch-execute cycle
  - MainMemory implements memory
- The first step in building our processor
  - implement 6 main internal methods of MainMemory



The Code You Will Implement

```
/**
 * Determine whether an address is aligned to specified length.
 * @param address memory address
 * @param length byte length
 * @return true iff address is aligned to length
 */
protected boolean isAccessAligned (int address, int length) {
  return false;
}
```

```java
/**
 * Convert an sequence of four bytes into a Big Endian integer.
 * @param byteAtAddrPlus0 value of byte with lowest memory address
 * @param byteAtAddrPlus1 value of byte at base address plus 1
 * @param byteAtAddrPlus2 value of byte at base address plus 2
 * @param byteAtAddrPlus3 value of byte at base address plus 3
 * @return Big Endian integer formed by these four bytes
 */
public int bytesToInteger (UnsignedByte byteAtAddrPlus0,
                           UnsignedByte byteAtAddrPlus1,
                           UnsignedByte byteAtAddrPlus2,
                           UnsignedByte byteAtAddrPlus3) {
  return 0;
}

/**
 * Convert a Big Endian integer into an array of 4 bytes
 * @param  i an Big Endian integer
 * @return an array of UnsignedByte
 */
public UnsignedByte[] integerToBytes (int i) {
  return null;
}
```

```
**
 * Fetch a sequence of bytes from memory.
 * @param address address of the first byte to fetch
 * @param length  number of bytes to fetch
 * @return an array of UnsignedByte
 */
protected UnsignedByte[] get (int address, int length) throws ... {
  UnsignedByte[] ub = new UnsignedByte [length];
  ub[0] = new UnsignedByte (0); // with appropriate value
  // repeat to ub[length-1] ...
  return ub;
}

/**
 * Store a sequence of bytes into memory.
 * @param  address                address of the first memory byte
 * @param  value                  an array of UnsignedByte values
 * @throws InvalidAddressException if any address is invalid
 */
protected void set (int address, UnsignedByte[] value) throws ... {
  byte b[] = new byte [value.length];
  for (int i=0; i<value.length; i++)
    b[i] = (byte) value[i].value();
  // write b into memory ...
}
```

Reading

- Companion
  - previous module: 1, 2.1
  - new: 2.2 (focus on 2.2.2 for this week)
- Textbook
  - *A Historical Perspective, Machine-Level Code, Data Formats, Data Alignment.*
  - 2ed: 3.1-3.2.1, 3.3, 3.9.3
    - (skip 3.2.2 and 3.2.3)
  - 1ed: 3.1-3.2.1, 3.3, 3.10

Numbers and Bits

Binary, Hex, and Decimal Refresher

- Hexadecimal notation
  - number starts with “0x” , each digit is base 16 not base 10
  - e.g.: 0x2a3 = $2x16^2 + 10x16^1 + 3x16^0$
  - a convenient way to describe numbers when binary format is important
  - each hex digit (hexit) is stored by 4 bits: (0|1)x8 + (0|1)x4 + (0|1)x2 + (0|1)x1
- Examples
  - 0x10 in binary? in decimal?
  - 0x2e in binary? in decimal?
  - 1101 1000 1001 0110 in hex? in decimal?
  - 102 in binary? in hex?

B	H	D
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	a	10
1011	b	11
1100	c	12
1101	d	13
1110	e	14
1111	f	15

Bit Shifting

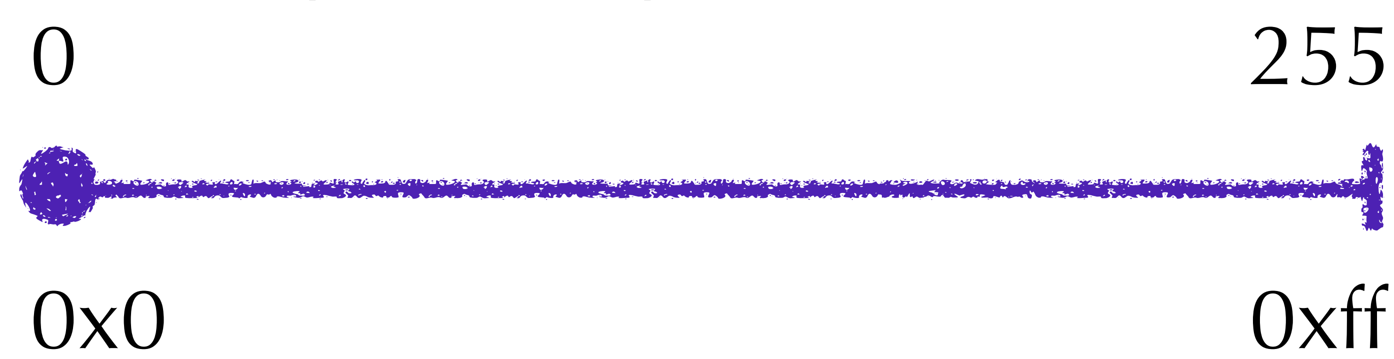
- bit shifting: multiply/divide by powers of 2
- left shift by k bits, "<< k": multiply by 2^k
  - old bits on left end drop off, new bits on right end set to 0
  - examples
    - 0000 1010 << 1 = 0001 0100; 0x0a << 1 = 0x14; 10 << 1 = 20; 10 * 2 = 20
    - 0000 1110 << 2 = 0011 1000; 0x0e << 2 = 0x38; 14 << 2 = 28; 14 * 4 = 56
  - << k, left shift by k bits, multiply by 2^k
    - old bits on left end drop off, new bits on right end set to 0
- right shift by k bits, ">> k": divide by 2^k
  - old bits on right end drop off, new bits on left end set to 0
    - (in C etc... stay tuned for Java!)
  - examples
    - 1010 >> 1 = 0101
    - 1110 >> 2 = 0011
- why do this? two good reasons:
  - much faster than multiply. much, much faster than division
  - good way to move bits around to where you need them

Masking

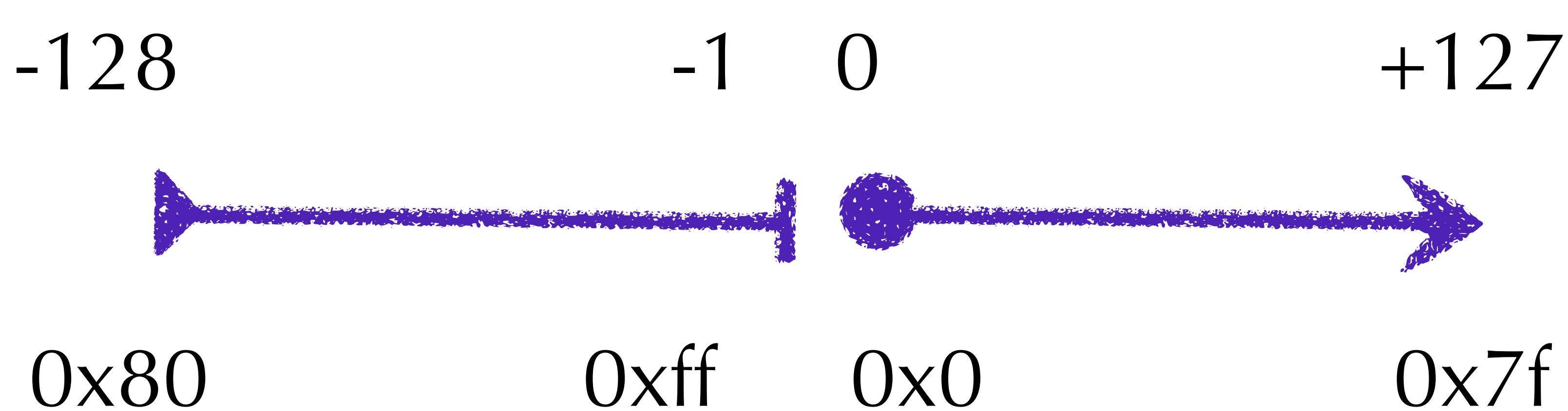
- bitmask: pattern of bits you construct with/for logical operations
  - mask with 0 to throw bits away
  - mask with 1 to let bit values pass through
- masking in binary: remember your binary truth tables!
  - &: AND, |: OR
  - 1&1=1, 1&0=0, 0&1=0, 0&0=0
  - 1|1=1, 1|0=1, 0|1=1, 0|0=0
  - example: 1111 & 0011 = 0011
- masking in hex:
  - mask with & 0 to turn bits off
  - mask with & 0xf (1111 in binary) to let bit values pass through
  - example: 0x00ff & 0x3a2b = 0x002b

Two's Complement: Reminder

- unsigned
  - all possible values interpreted as positive numbers
  - byte (8 bits)



- signed: two's complement
  - the first half of the numbers are positive, the second half are negative
  - start at 0, go to top positive value, "wrap around" to most negative value, end up at -1

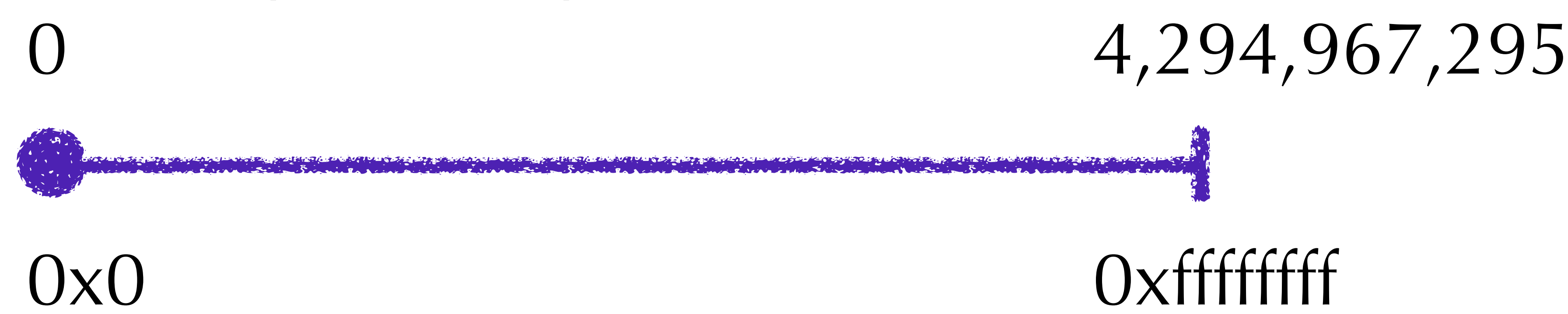


Two's Complement: Byte

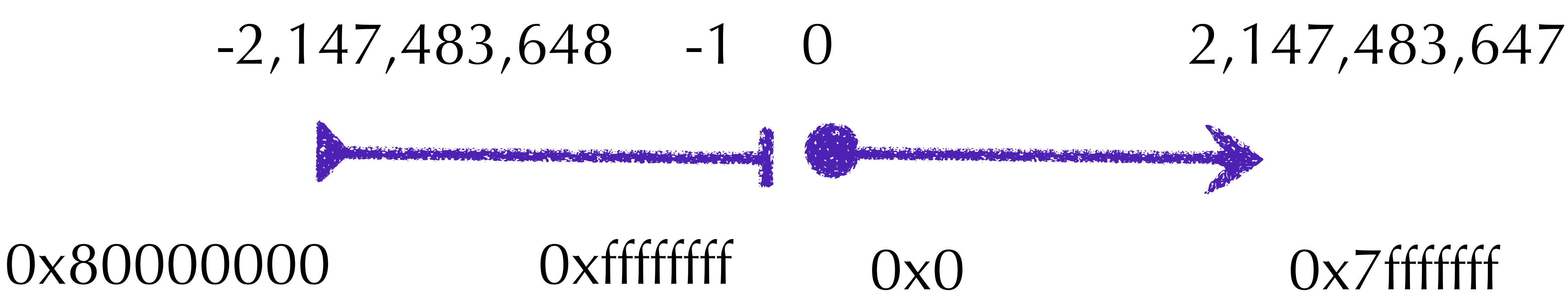
B	H	Signed Decimal	Unsigned
1111 1111	0xff	-1	255
1111 1110	0xfe	-2	254
1111 1101	0xfd	-3	253
1111 1100	0xfc	-4	252
1111 1011	0xfb	-5	251
1111 1010	0xfa	-6	250
1111 1001	0xf9	-7	249
1111 1000	0xf8	-8	248
1111 0111	0xf7	-9	247
1111 0110	0xf6	-10	246
1111 0101	0xf5	-11	245
1111 0100	0xf4	-12	244
1111 0011	0xf3	-13	243
1111 0010	0xf2	-14	242
1111 0001	0xf1	-15	241
1111 0000	0xf0	-16	240

Two's Complement: 32-Bit Integers

- unsigned
  - all possible values interpreted as positive numbers
  - int (32 bits)



- signed: two's complement
  - the first half of the numbers are positive, the second half are negative
  - start at 0, go to top positive value, "wrap around" to most negative value, end up at -1



Two's Complement and Sign Extension

- normally, pad with 0s when extending to larger size
  - 0x8b byte (139) becomes 0x0000008b int (139)
- but that would change value for negative 2's comp:
  - 0xff byte (-1) should not be 0x000000ff int (255)
- so: pad with Fs with negative numbers in 2's comp:
  - 0xff byte (-1) becomes 0xffffffff int (-1)
  - in binary: padding with 1, not 0
- reminder: why do all this?
  - add/subtract works without checking if number positive or negative

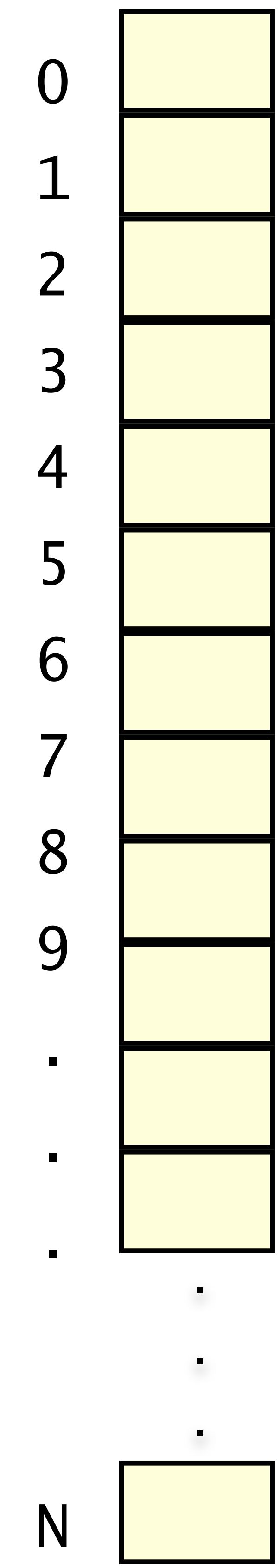
Bit Shifting in Java

- **signed/arithmetic** right shift by k bits, ">> k": divide by 2^k
  - old bits on right end drop off, new bits on left end set to top (sign) bit
  - examples
    - 1010 >> 1 = 1101
    - 1110 >> 2 = 1111
    - 0010 >> 1 = 0001
    - 0110 >> 2 = 0001
- **unsigned/logical** right shift by k bits, ">>>k":
  - old bits on right end drop off, new bits on left end set to 0
  - but.. be careful - requires int/long and automatically promotes up
    - so bytes automatically promoted, but with sign extension
    - safest to construct bitmasks with int/long, not bytes

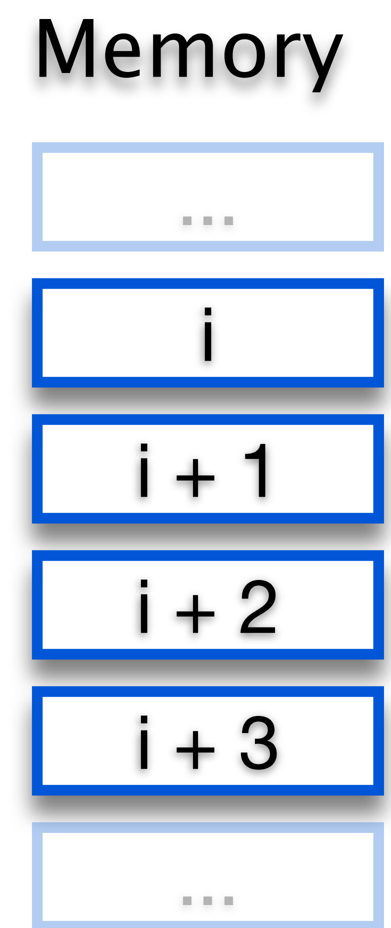
Numbers in Memory

Memory and Integers

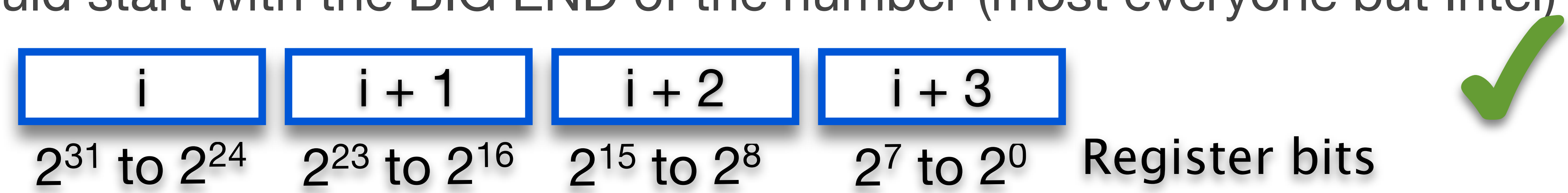
- Memory is byte addressed
  - every byte of memory has a unique address, numbered from 0 to N
  - N is huge: billions is common these days (2-16 GB)
- Integers can be declared at different sizes
  - ***byte*** is 1 byte, 8 bits, 2 hexits
  - ***short*** is 2 bytes, 16 bits, 4 hexits
  - ***int*** or ***word*** is 4 bytes, 32 bits, 8 hexits
  - ***long*** is 8 bytes, 64 bits, 16 hexits
- Integers in memory
  - reading or writing an integer requires specifying a range of byte addresses



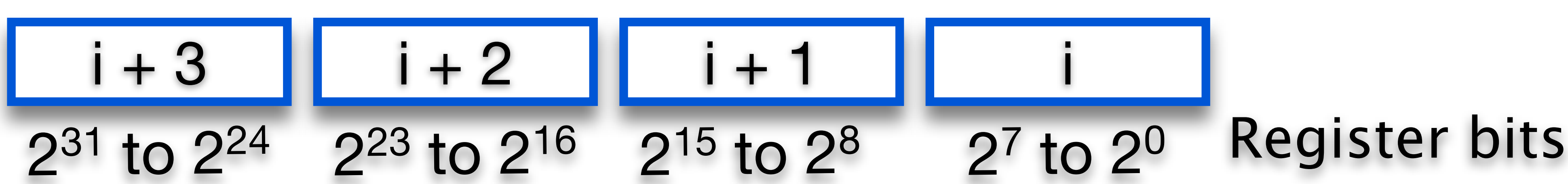
Making Integers from Bytes



- Our first architectural decisions
  - assembling memory bytes into integer registers
- Consider 4-byte memory word and 32-bit register
  - it has memory addresses i, i+1, i+2, and i+3
  - we'll just say it's "***at address i and is 4 bytes long***"
  - e.g., the word at address 4 is in bytes 4, 5, 6 and 7.
- Big or Little Endian (end means where start from, not finish)
  - we could start with the BIG END of the number (most everyone but Intel)



  - or we could start with the LITTLE END (Intel x86, some others)



Making Integers from Bytes

- Our first architectural decisions
  - assembling memory bytes into integer registers
- Consider 4-byte memory word and 32-bit register
  - it has memory addresses i, i+1, i+2, and i+3
  - we'll just say it's "***at address i and is 4 bytes long***"
  - e.g., the word at address 4 is in bytes 4, 5, 6 and 7.
- Big or Little Endian (end means where start from, not finish)
  - we could start with the BIG END of the number (most everyone but Intel)

i	i + 1	i + 2	i + 3	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Register bits

  - or we could start with the LITTLE END (Intel x86, some others)

i + 3	i + 2	i + 1	i	
2^{31} to 2^{24}	2^{23} to 2^{16}	2^{15} to 2^{8}	2^{7} to 2^{0}	Register bits

Memory

Memory
...
i
i + 1
i + 2
i + 3
...

Aligned or Unaligned Addresses

- we could allow any number to address a multi-byte integer

* disallowed on many architectures
* allowed on Intel, but slower

- or we could require that addresses be aligned to integer-size boundary

* SM213 alignment: 4-byte words

address modulo chunk-size is always zero

- Power-of-Two Aligned Addresses Simplify Hardware
  - smaller things always fit complete inside of bigger things

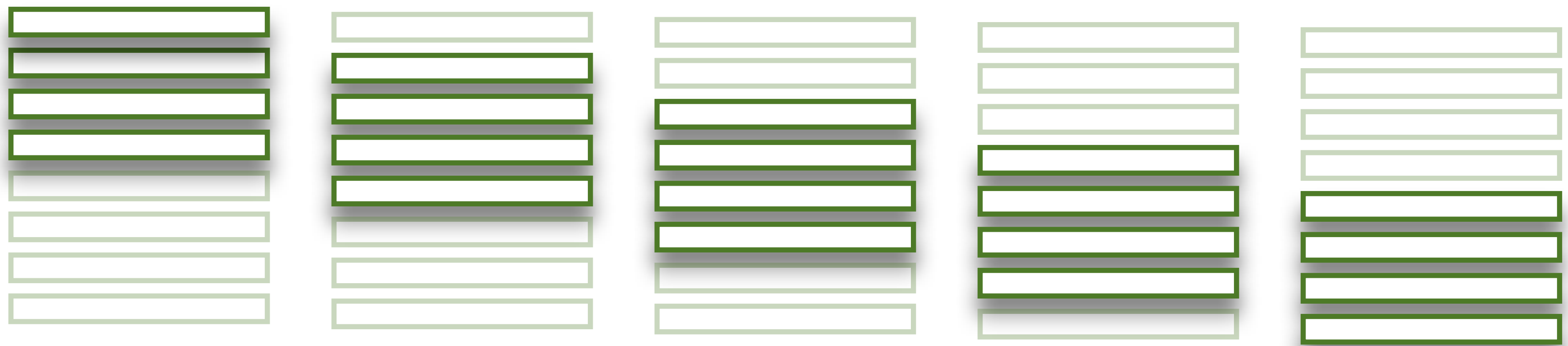
word contains exactly two complete shorts

  - byte address from integer address: divide by power to two, which is just shifting bits

$j \,/\, 2^k == j >> k$ (j shifted k bits to right)

Aligned or Unaligned Addresses

- we could allow any number to address a multi-byte integer



- or we could require that addresses be aligned to integer-size boundary

* SM213 alignment: 4-byte words

address modulo chunk-size is always zero

- Power-of-Two Aligned Addresses Simplify Hardware
  - smaller things always fit complete inside of bigger things

word contains exactly two complete shorts

  - byte address from integer address: divide by power to two, which is just shifting bits

$j \ / \ 2^k == j >> k$ (j shifted k bits to right)

Computing Alignment

- boolean align(number, size)
  - does a number fit nicely for a particular size (in bytes)?
- divide number n by size s (in bytes), aligned if no remainder
  - easy if number is decimal
  - otherwise convert from hex or binary to decimal
- check if n mod s = 0
  - mod notation usually '%'. same as division, of course...
- check if certain number of final bits are all 0
  - pattern?
    - last 1 digit for 2-byte short
    - last 2 digits for 4-byte world
    - last 3 digits for 8-byte longlong
  - last k digits, where $2^k = s$ (size in bytes)
  - easy if number is hex: convert to binary and check

B	H	D
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	a	10
1011	b	11
1100	c	12
1101	d	13
1110	e	14
1111	f	15

In the Lab ... Revisited

- SimpleMachine simulator
  - load code into Eclipse and get it to build/run
  - write and test `MainMemory.java`
    - `get/set` should check for out of bounds access but not alignment
    - `isAccessAligned` checks for alignment

Questions

- Which of the following statement (s) are true
  - [A] 6 == 110_2 is aligned for addressing a *short*
  - [B] 6 == 110_2 is aligned for addressing a *int*
  - [C] 20 == 10100_2 is aligned for addressing a *int*
  - [D] 20 == 10100_2 is aligned for addressing a *long*

- Which of the following statements are true
  - [A] memory stores Big Endian integers
  - [B] memory stores bytes interpreted by the CPU as Big Endian integers
  - [C] Neither
  - [D] I don’t know

- Which of these are true
  - [A] The Java constants 16 and 0x10 are exactly the same integer
  - [B] 16 and 0x10 are different integers
  - [C] Neither
  - [D] I don't know

- What is the Big-Endian integer value at address 4 below?
  - [A] `0x1c04b673`
  - [B] `0xc1406b37`
  - [C] `0x73b6041c`
  - [D] `0x376b40c1`
  - [E] none of these
  - [F] I don't know

Memory

0x0:	0xfe
0x1:	0x32
0x2:	0x87
0x3:	0x9a
0x4:	0x73
0x5:	0xb6
0x6:	0x04
0x7:	0x1c

- What is the value of i after this Java statement executes?

```
i = 0xff8b0000 & 0x00ff0000;
```

- [A] `0xffff0000`
- [B] `0xff8b0000`
- [C] `0x008b0000`
- [D] I don’t know

- What is the value of i after this Java statement executes?

```
int i = (0x0000008b) << 16;
```

- [A] 0x8b
- [B] 0x0000008b
- [C] 0x008b0000
- [D] 0xff8b0000
- [E] None of these
- [F] I don't know

- What is the value of i after this Java statement executes?

```java
int i = (byte)(0x8b) << 16;
```

- [A] 0x8b
- [B] 0x0000008b
- [C] 0x008b0000
- [D] 0xff8b0000
- [E] None of these
- [F] I don't know