Georgia Tech Computer Science

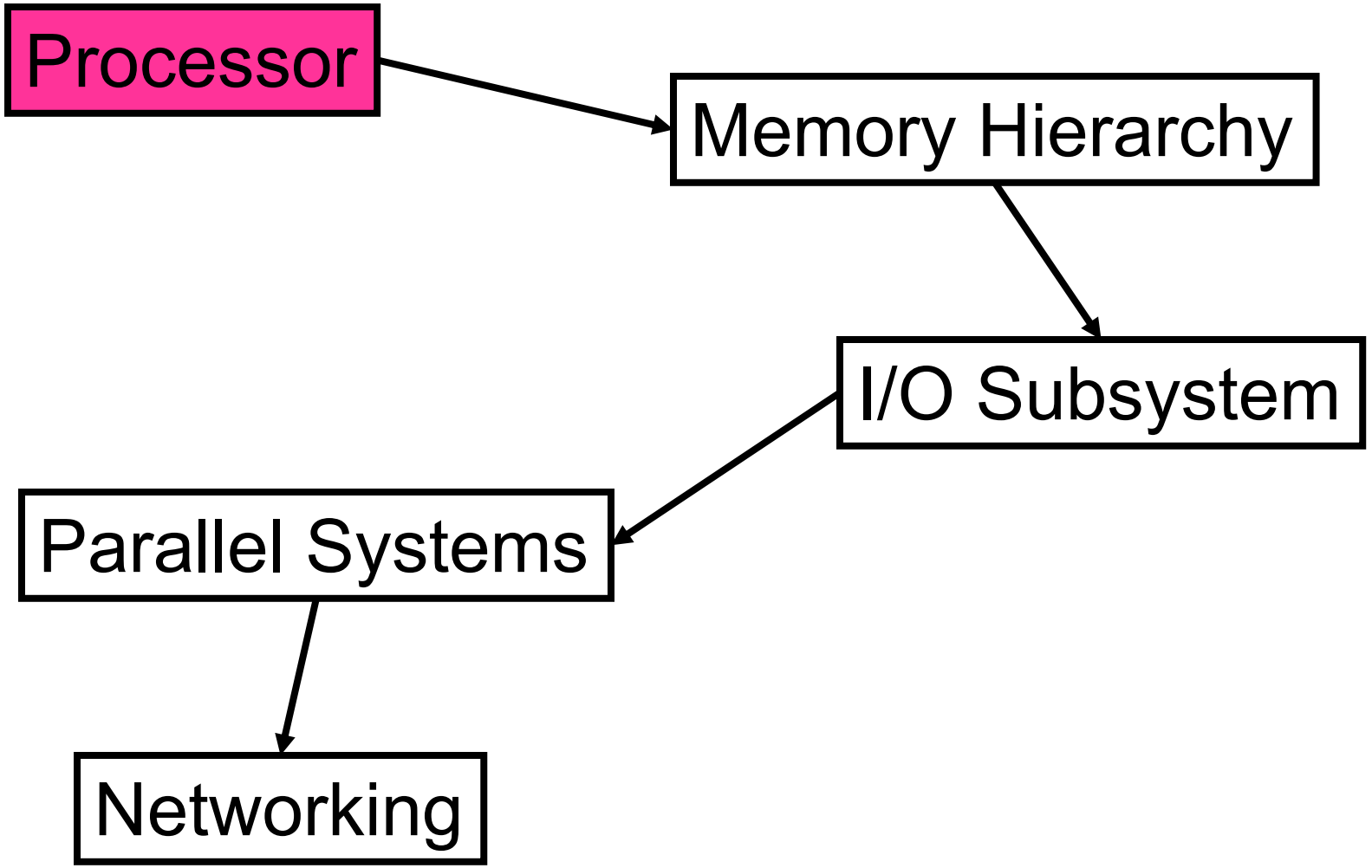
CS2200
Systems and Networks
Spring 2024

Lecture 2: Processors

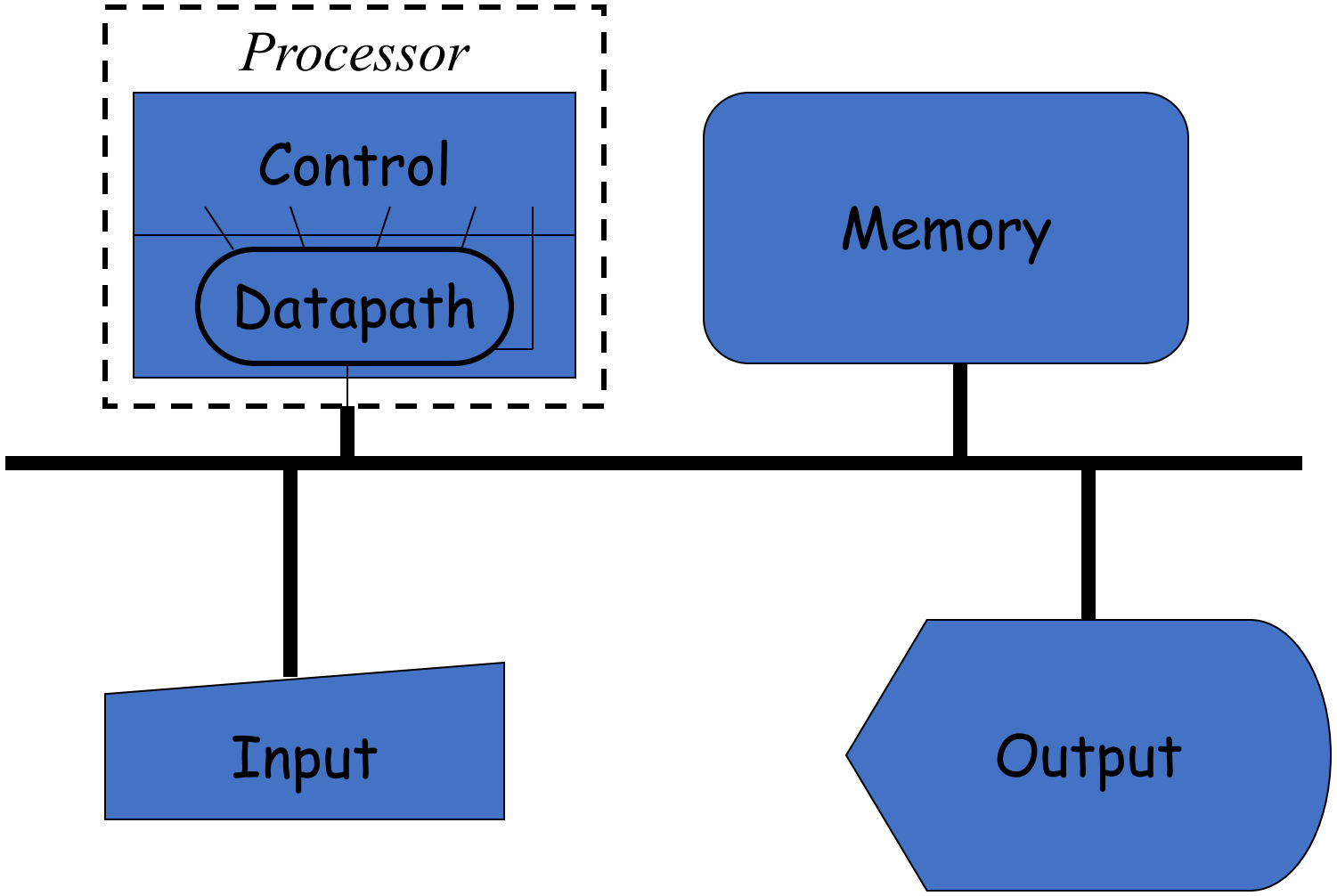
Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Leahy, Ramachandran of Georgia Tech

Our Road Map


Processor
Memory Hierarchy
I/O Subsystem
Parallel Systems
Networking

A Computer's Key Components

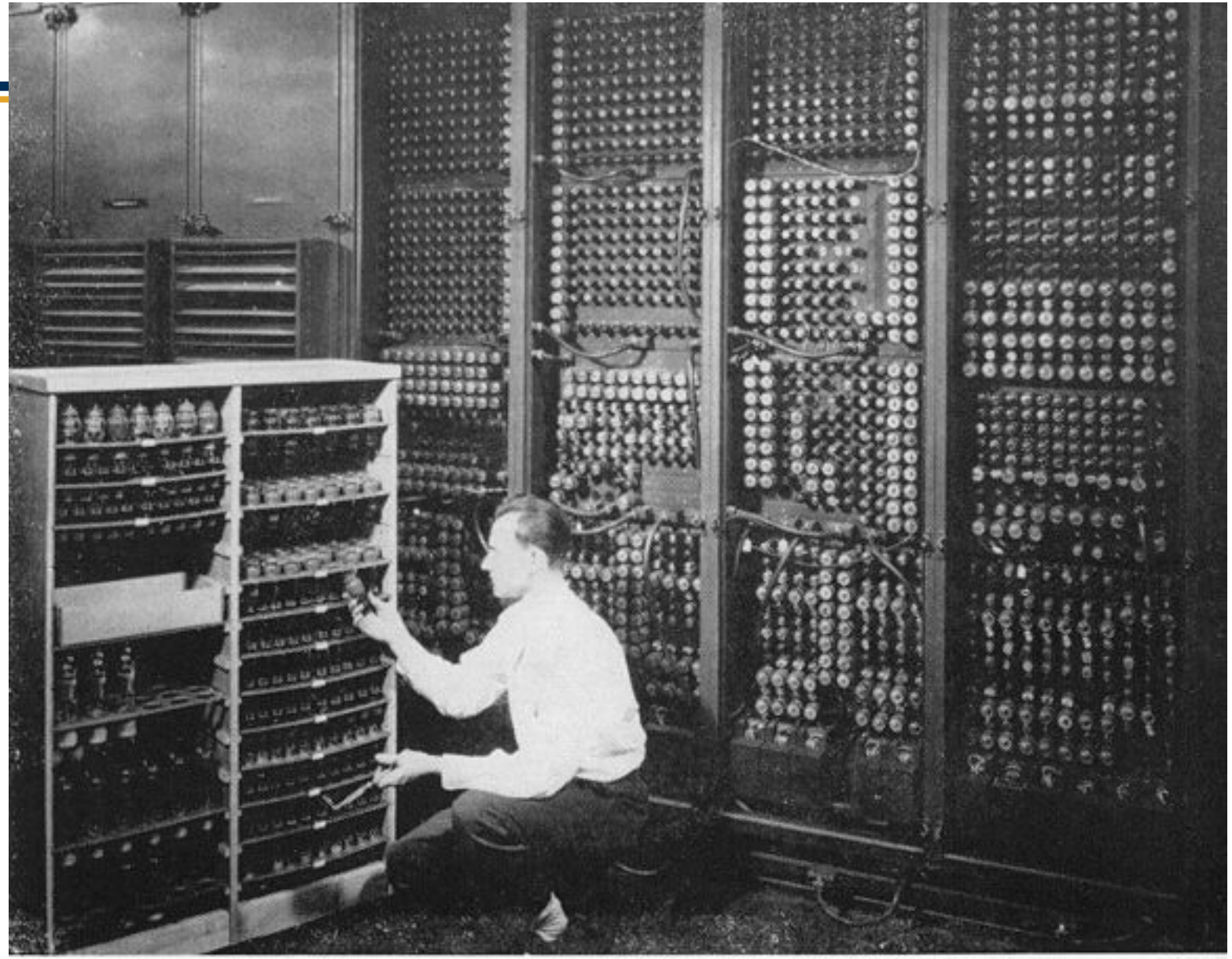


What does the processor do?

- Knows where it is in program
- Can get and put data into memory
- Can do some arithmetic
- Can make tests and take different paths depending on the results

- Do you need a language to make a computer run?

A Little History



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

A Little History

- First computers programmed by hand

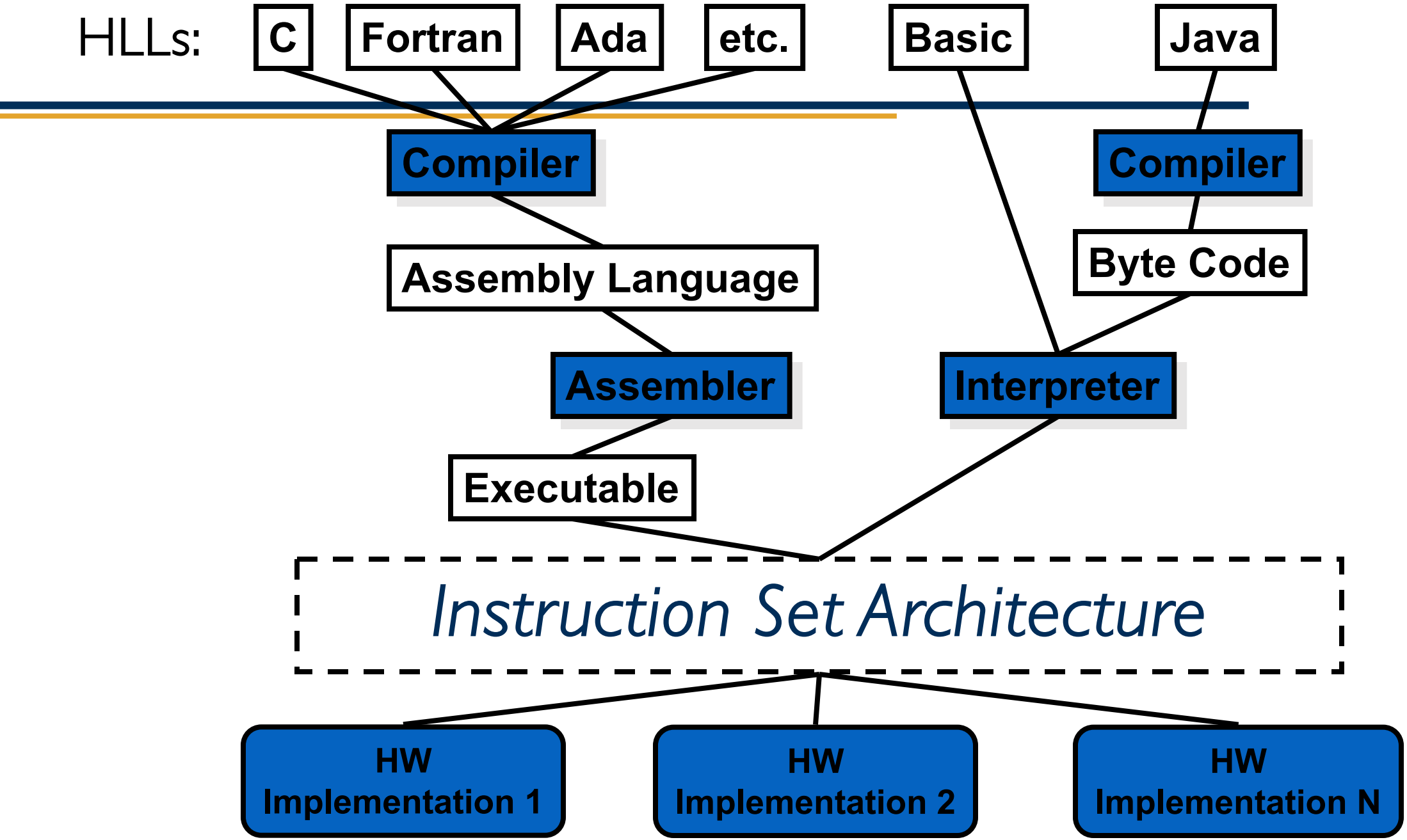
  1000110010100000

- Somewhat tedious, so invented:
- Assembler

  add A,B

- If we can convert from Assembly Language to machine code why not from some higher-level language to Assembler?

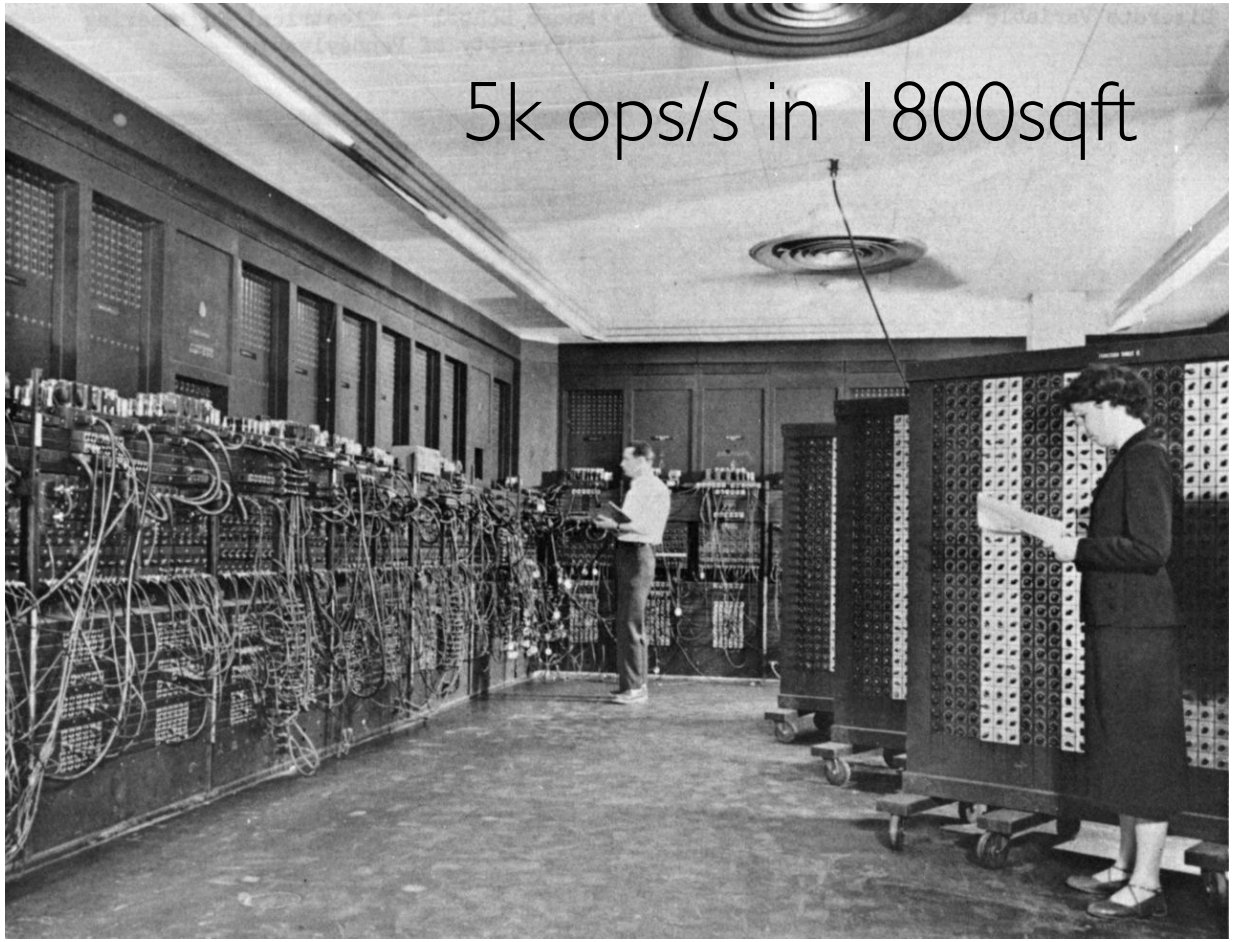
  A + B


HLLs:
C
Fortran
Ada
etc.
Basic
Java
Compiler
Compiler
Assembly Language
Byte Code
Assembler
Interpreter
Executable
Instruction Set Architecture
HW Implementation 1
HW Implementation 2
HW Implementation N

Instructions

- Language of the machine
- Vocabulary is the instruction set (ISA)
- Two levels
  - Human readable (assembly)
  - Machine readable (machine code)

Computing Evolution in ~70 Years



17T ops/s in 16sq in

3 billion ENIACs in your palm

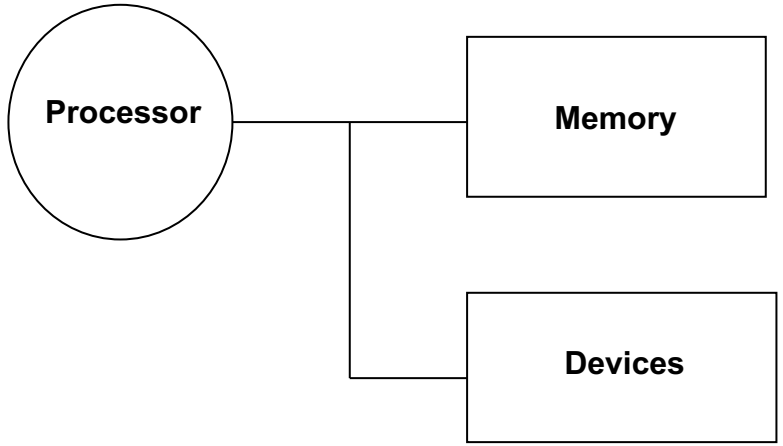
55 trillion times higher compute density

Moving forward: ISA Construction

- Instruction set design from HLL constructs
  - Expressions, assignments => ALU instructions
  - Data abstraction => Addressing modes
  - Conditional & loop statements => Branch instructions
  - Procedure calls/returns => stack management

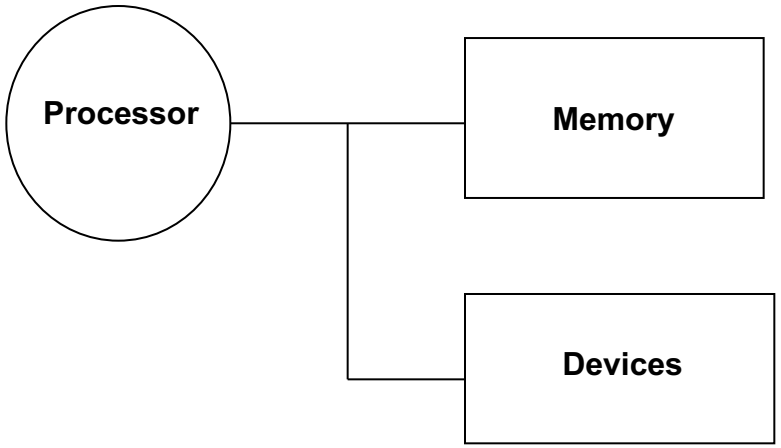
- Please note the reading assignments in the schedule: Start reading chapter 2

Simple Machine Model



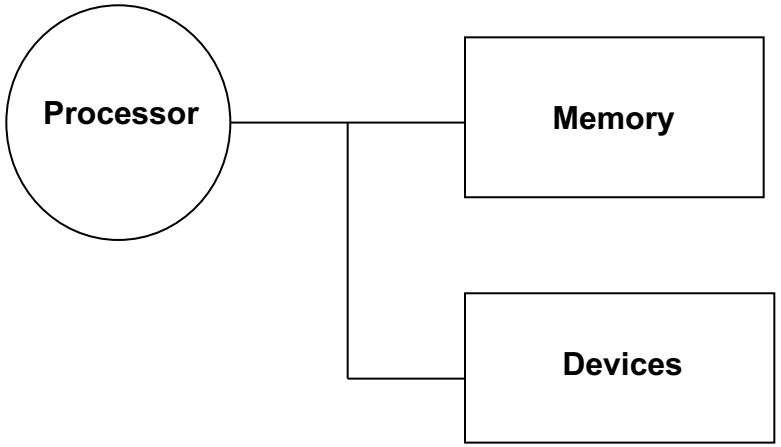
- Remember the LC-3? It used a greatly simplified ARM instruction set
- We'll be introducing the LC-2200, a greatly simplified MIPS instruction set
  - Architecture that's similar, but not the same as the LC-3.

Simple Machine Model



- Let's consider the execution of a HLL
- a = a + 1
  c = a + b
  if (c == d) {
  …
  }

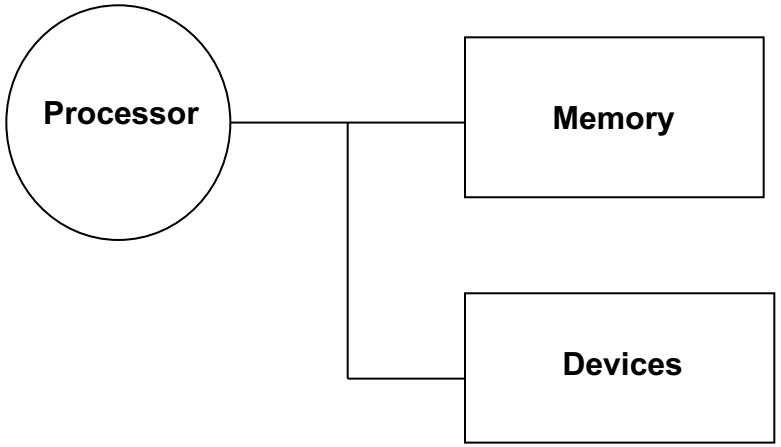
How to Design an Instruction Set?



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

Arithmetic/Logical Expressions



Start thinking like a compiler writer

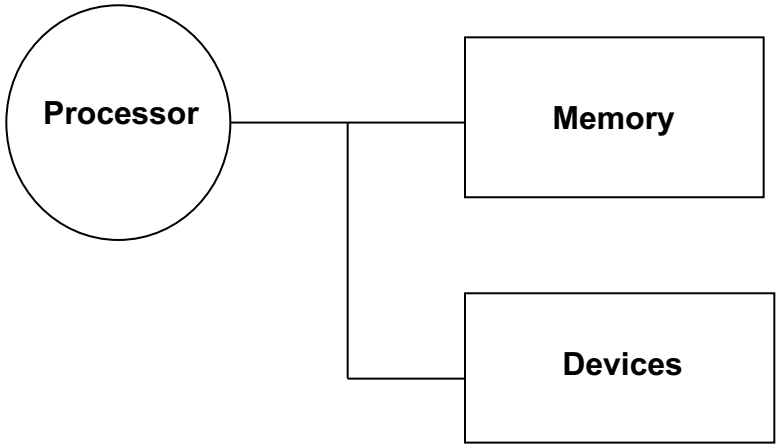
➔ What instructions are needed for each HLL construct?

c = a + b

becomes

add c, a, b

Arithmetic/Logical Expressions



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

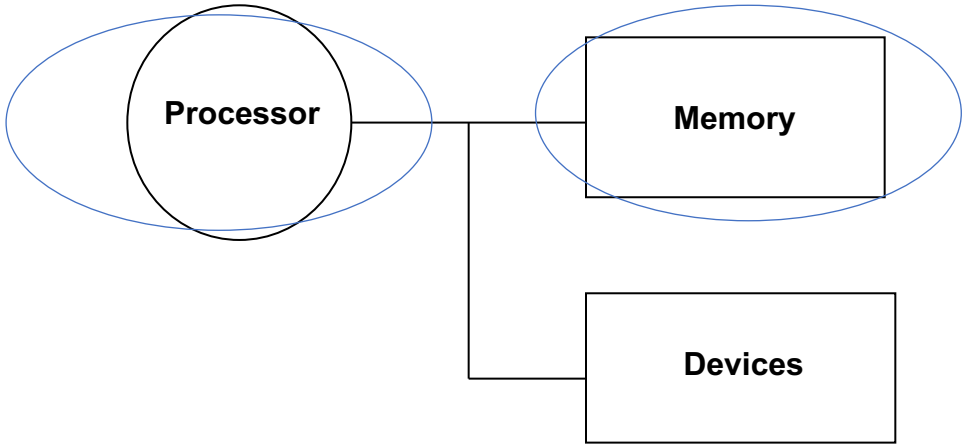
c = a + b ➔ add c, a, b

Keep adding to repertoire

c = a – b ➔ sub c, a, b

c = !(a & b) ➔ nand c, a, b

Arithmetic/Logical Expressions



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

c = a + b ➔ add c, a, b

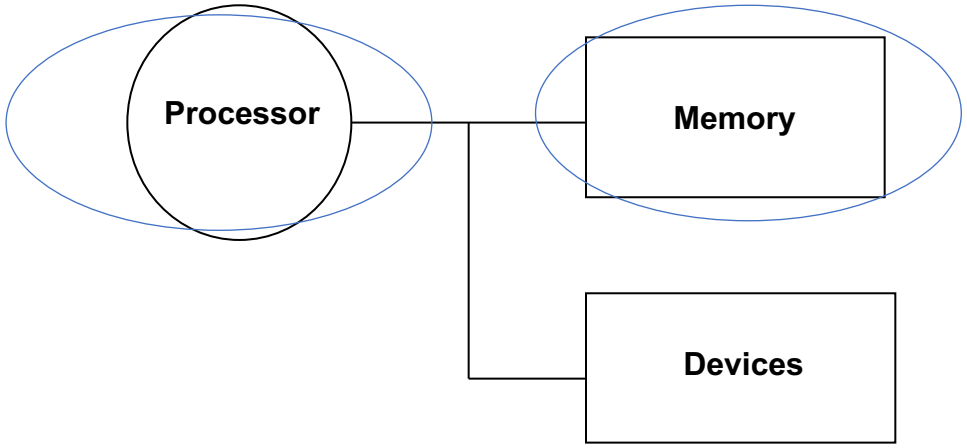
What do you call these?

➔memory operands

➔memory addressing mode

Where are they?

Arithmetic/Logical Expressions



Start thinking like a compiler writer

➔ What instructions are needed for each HLL construct?

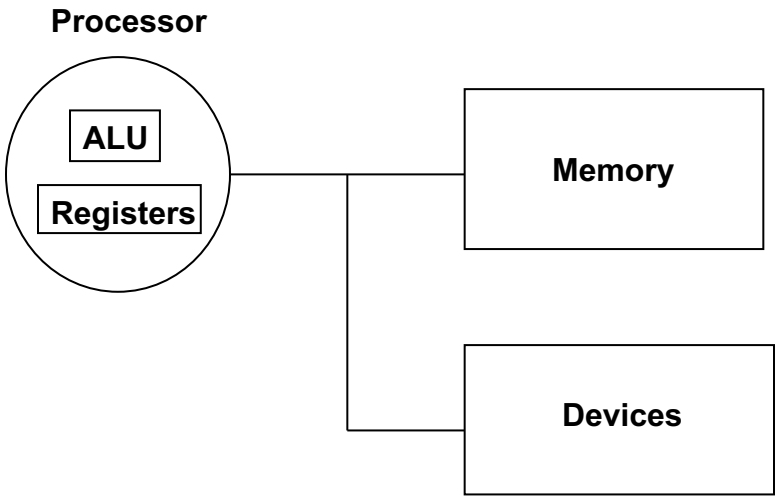
c = a + b ➔ add c, a, b

Is there a downside to operands in memory?

How can we address that?

A trip to memory is EXPENSIVE!

Operands?



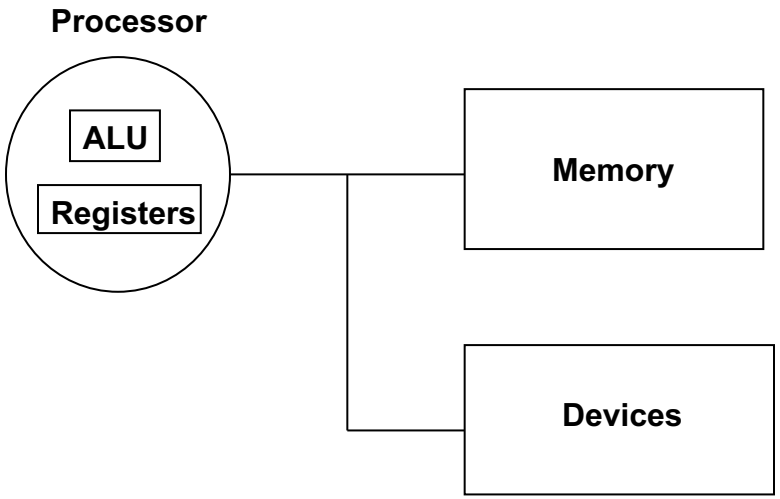
c = a + b ➔ add c, a, b

How about introducing Load/Store instructions?

ld r_1, a

st c, r_2

Load/Store Instructions



ld r_1, a

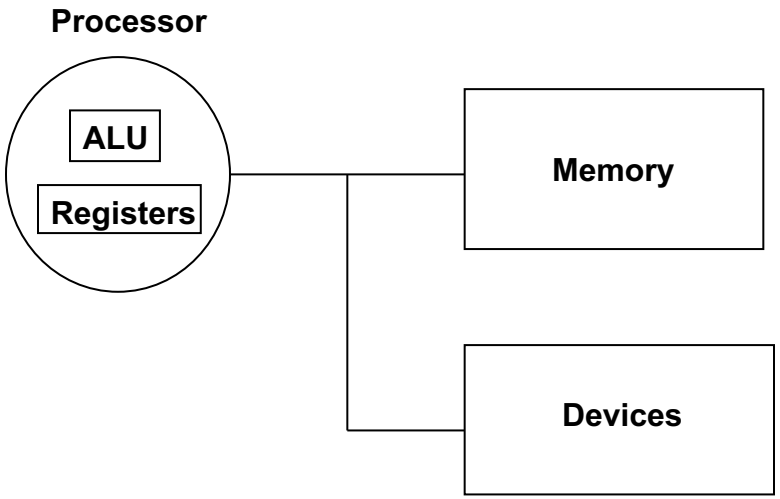
st c, r_2

➔ We've got operands in registers

➔ Register addressing mode!

So how do we compile c = a + b now?

Register Operands



Old way:

add c, a, b

New way:

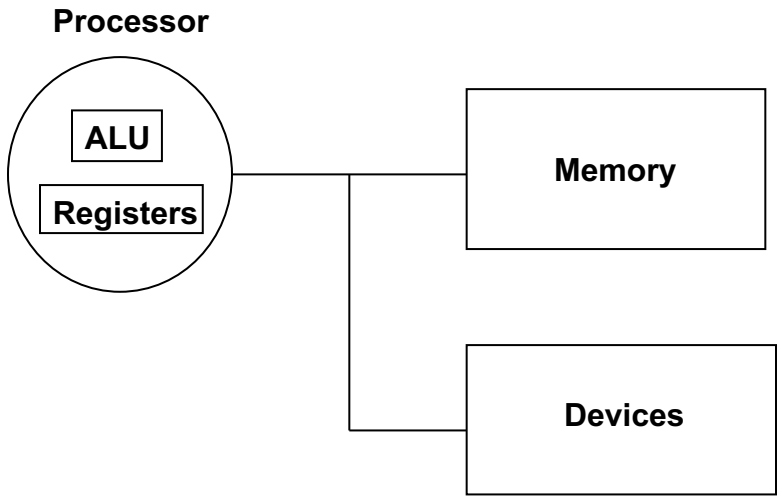
ld r_1, a

ld r_2, b

add r_3, r_1, r_2

st c, r_3

Compiling with Register Operands



Old way:

```
add  c, a, b
```

1 instruction

New way:

```
ld   r1, a
ld   r2, b
add  r3, r1, r2
st   c, r3
```

4 instructions

This looks dumb!

Not really. Why?

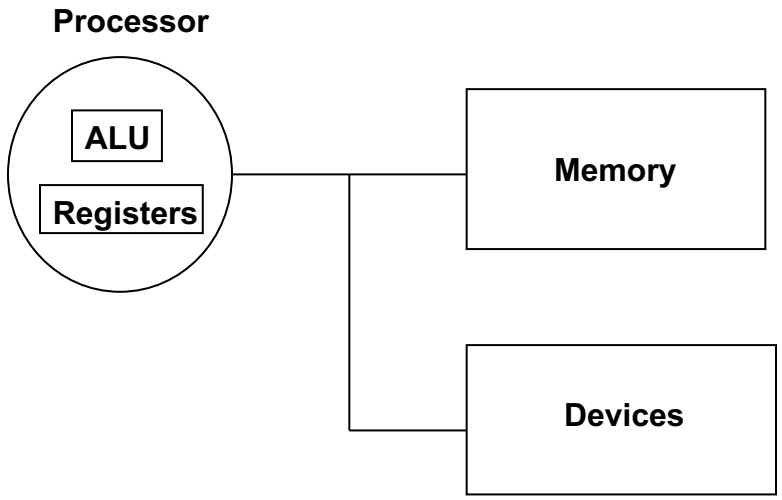
We can re-use the values in registers!

Keep Frequently Used Tools Nearby!

...or, the principle of **locality**

http://search.coolclips.com/m/vector/vc017870/Businesswoman-of-many-trades/

Reusing Values




```
c = a + b
d = a * b + c
if (c == d) {
    …
}
```

With operands left in registers, we can save three memory accesses here!

Session ID: dag2200

Why would we consider loading values into registers before computing with them?

Doesn't that use more instructions?

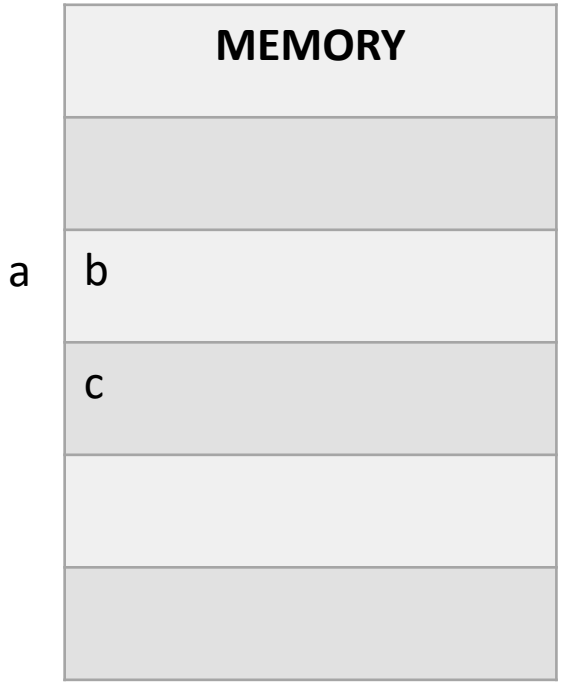
- 0% A. Yes it does, but it doesn't matter how many instructions it takes.
- 0% B. No it doesn't. You counted wrong.
- 0% C. Yes it does, but it saves memory accesses because we can re-use the values.
- 0% D. Yes it does and it's a terrible design choice.

Structs in HLL

```
struct {
        int b;
        int c;
} a;
```

Elements of a struct are contiguous in memory

How do we load b and c into registers?



Accessing Struct Members

```
struct {
        int b;
        int c;
} a;
```

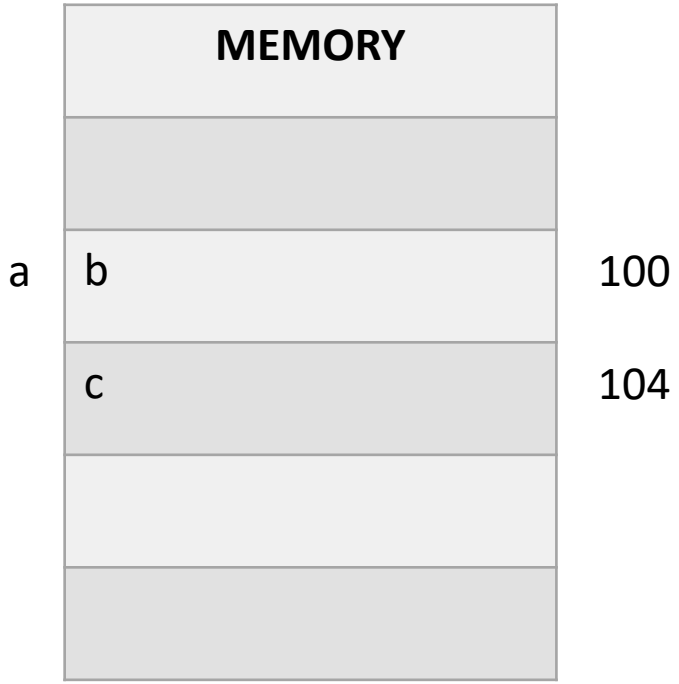
Let's say &a is already in register R_1

To load b:

$R_2 \leftarrow$ memory[R_1+0]

To load c:

$R_3 \leftarrow$ memory[R_1+4]



Base + Offset Addressing Mode

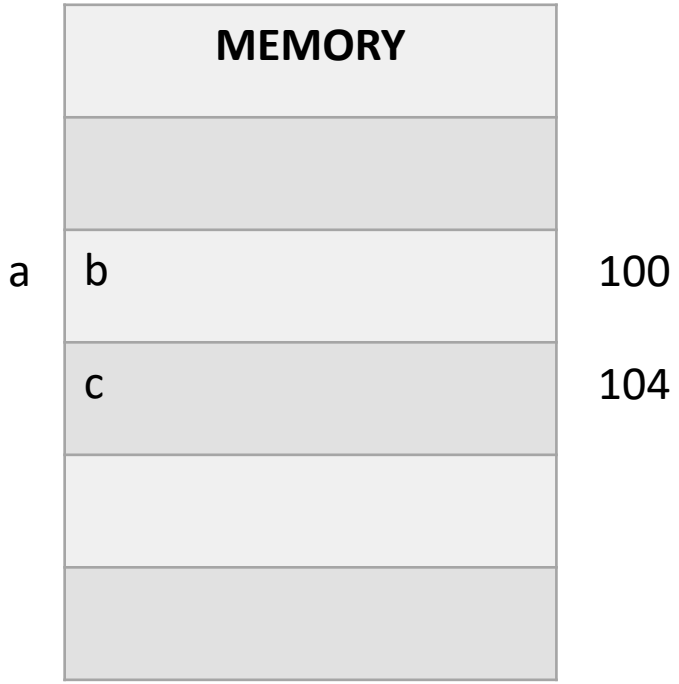
```
struct {
        int b;
        int c;
} a;
```

Let's say &a is already in register R_1

```
    ld  Ri, offset(Rbase)
```

To load b and c:

```
    ld  R2, 0(R1)
    ld  R3, 4(R1)
```



Operand Granularity

char → 8 bits → byte
short → 16 bits → half word
int → 32 bits* → word
long → 64 bits*

*depends on the word size of the architecture:
int=16, long=32 and others can happen

We need some instruction variants:

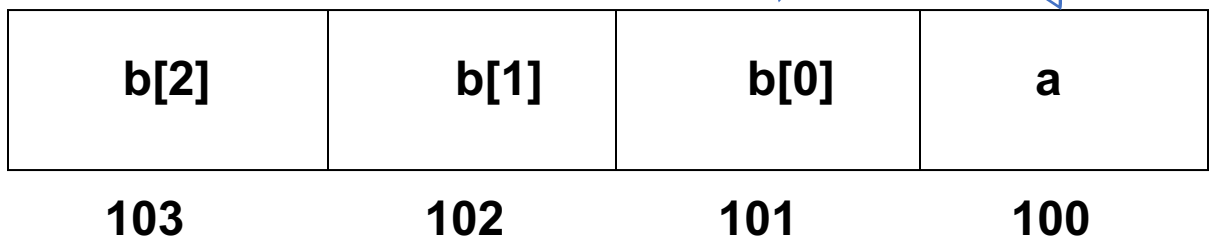
ldb, ldh, ldl, …
similar for store instructions

Operand Alignment

```
struct {
        char a;
        char b[3];
}
```

Dense Packing

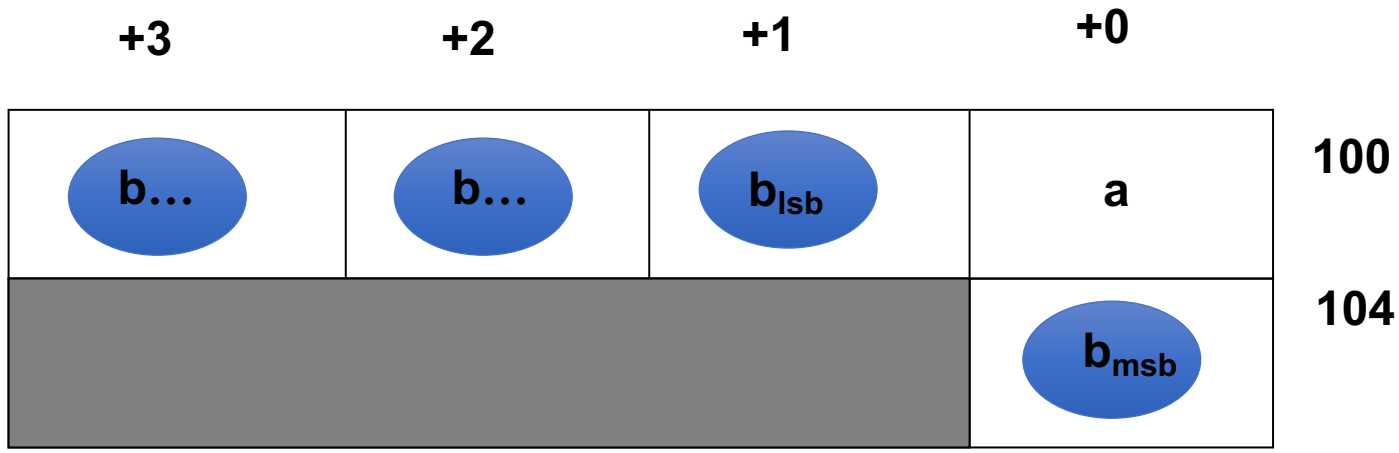
```
struct {
        char a;
        char b[3];
}
```



- We're packing operands to save space.

A Different Struct

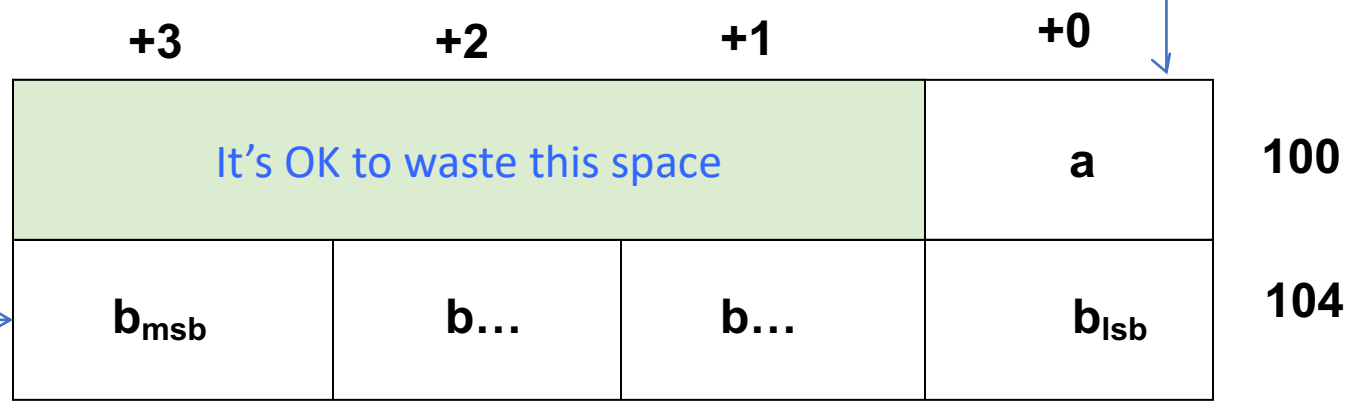
```
struct {
        char a;
        int b;
}
```



If we have a 32-bit-aligned path to memory, how are we going to load b?

Why Alignment Rules Matter

```
struct {
        char a;
        int b;
}
```

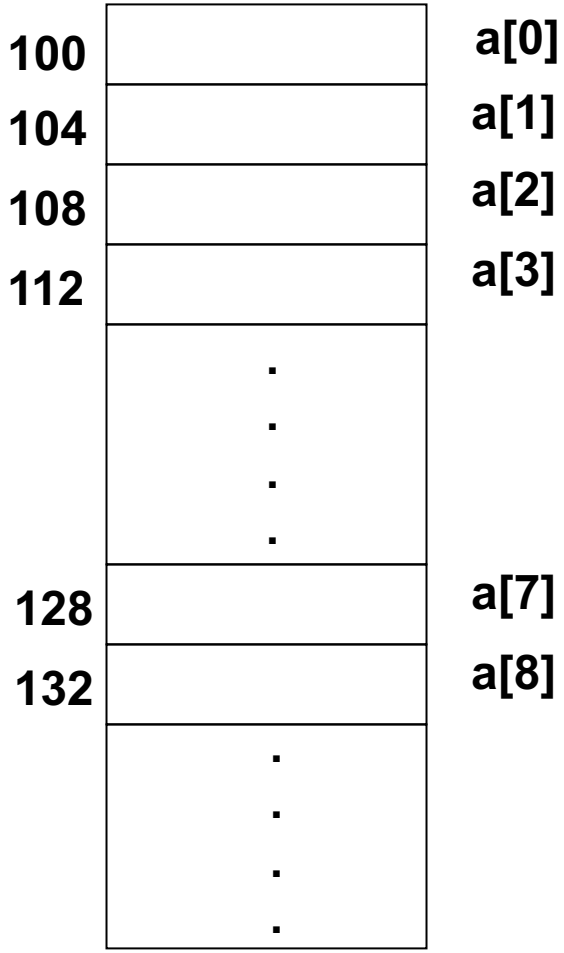


Now, how to load b is obvious

This is one of many space/time tradeoffs that ISA designers must address.

Accessing Array Operands

int a[100];



One approach, use base+offset

$r_1 = 100$

To load a[8]

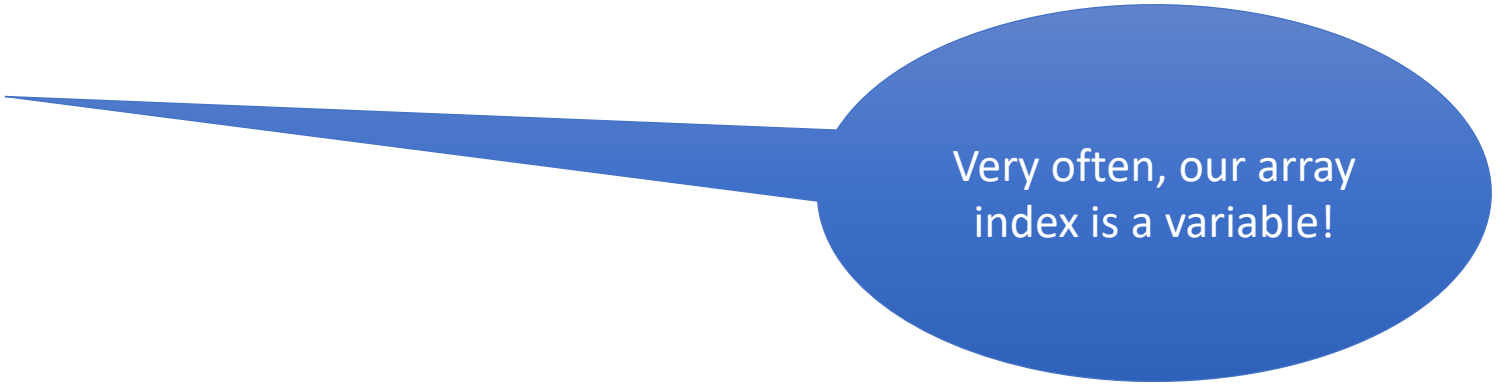
ld r_2, 32(r_1)

Is this the best we can do?
What does code that accesses an array usually look like?

Typical Array Use

- How do we typically use arrays?

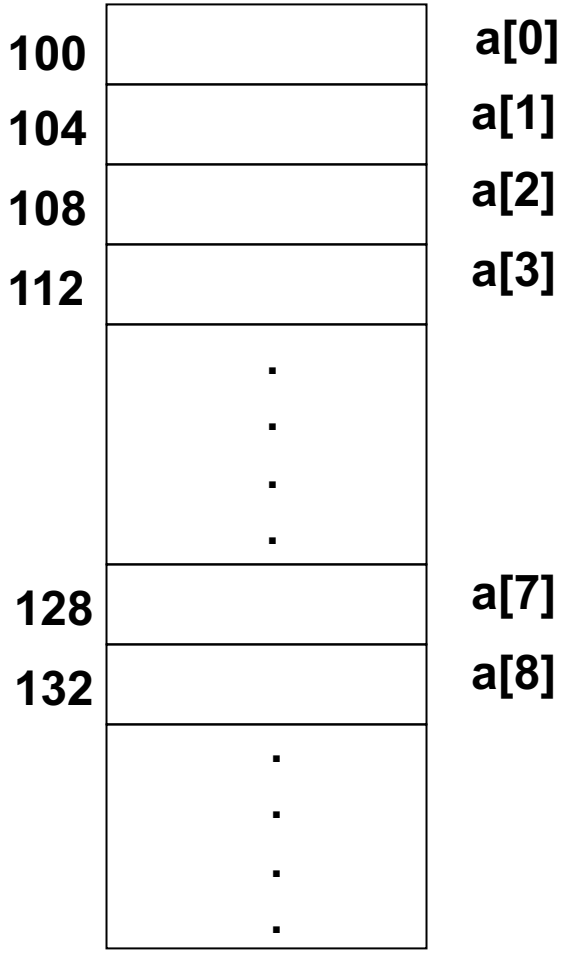
```
loop
  c[i] = a[i]
  …
  i = i + 1
end loop
```



- Looks like it's time for a new addressing mode
- Let's implement base+index addressing

Accessing Array Operands

int a[100];



With base+index mode

$r_1 = 100$

$r_2 = i * 4$

Why?

To load a[i]

ld r_8, $r_2(r_1)$

Using base + index

Now we can have this ld in a loop, only need to update r_2 value in each iteration

Endianness

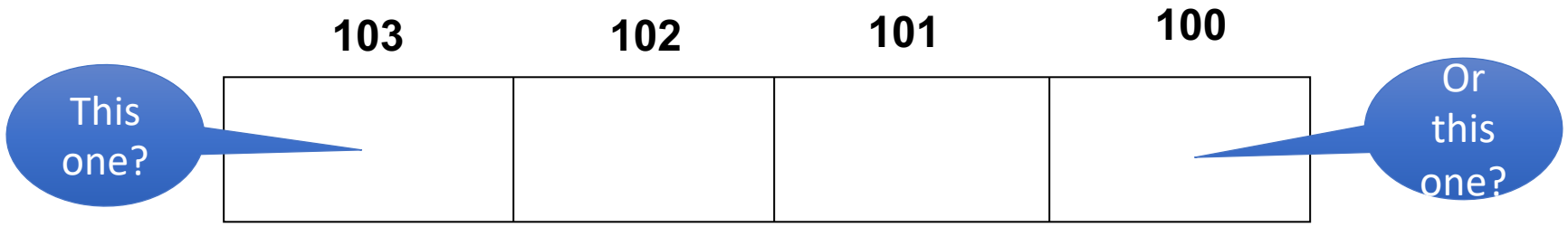
Say we have a 32-bit register, R1, that contains the value 1 in two's-complement.

We store that register into an (aligned) memory location

ST R1, Mem[100]

The value is stored in addresses 100-103 (of course).

Which byte contains the 1? (The other 3 will be zero, right?)

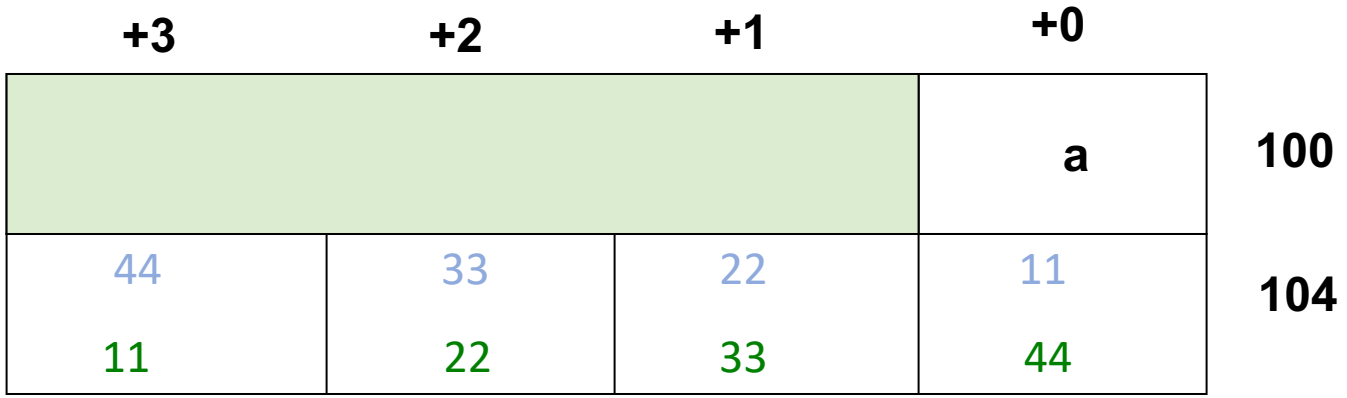


Endianness

Tip: Endianness often shows up in byte-addressable memories because we can access individual bytes out of longer data types

Little Endian ➔ addresses the Least Significant Byte (LSB) of the word

```
int b = 0x11223344;  //placed in address 104
```



Big Endian ➔ addresses the Most Significant Byte (MSB) of the word

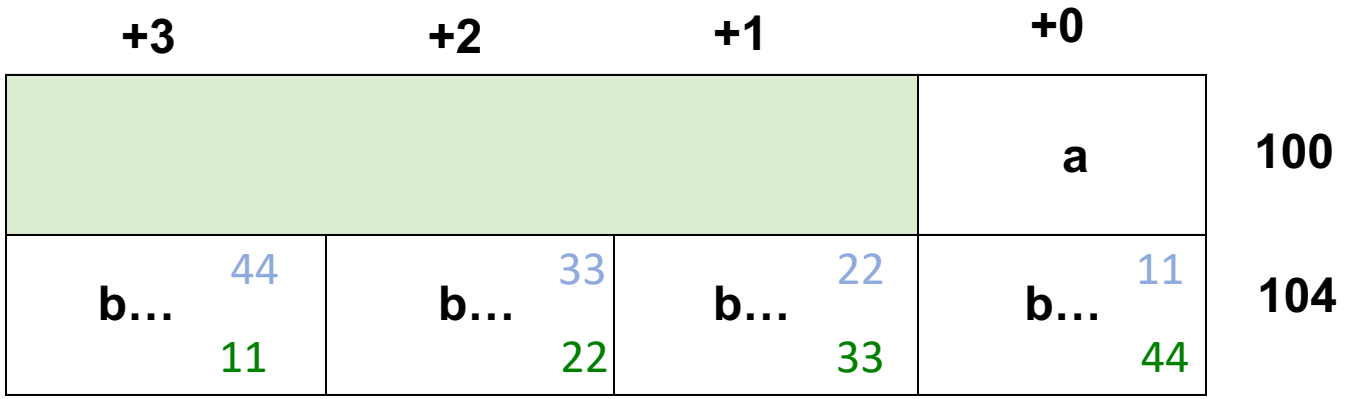
So What's the Difference?

The difference only shows up when taking a "word" apart (in this case, loading 8 bits from a 32-bit integer)

ldb r_1, Mem[104]

Big Endian ➔ loads 0x11 into r_1

Little Endian ➔ loads 0x44 into r_1



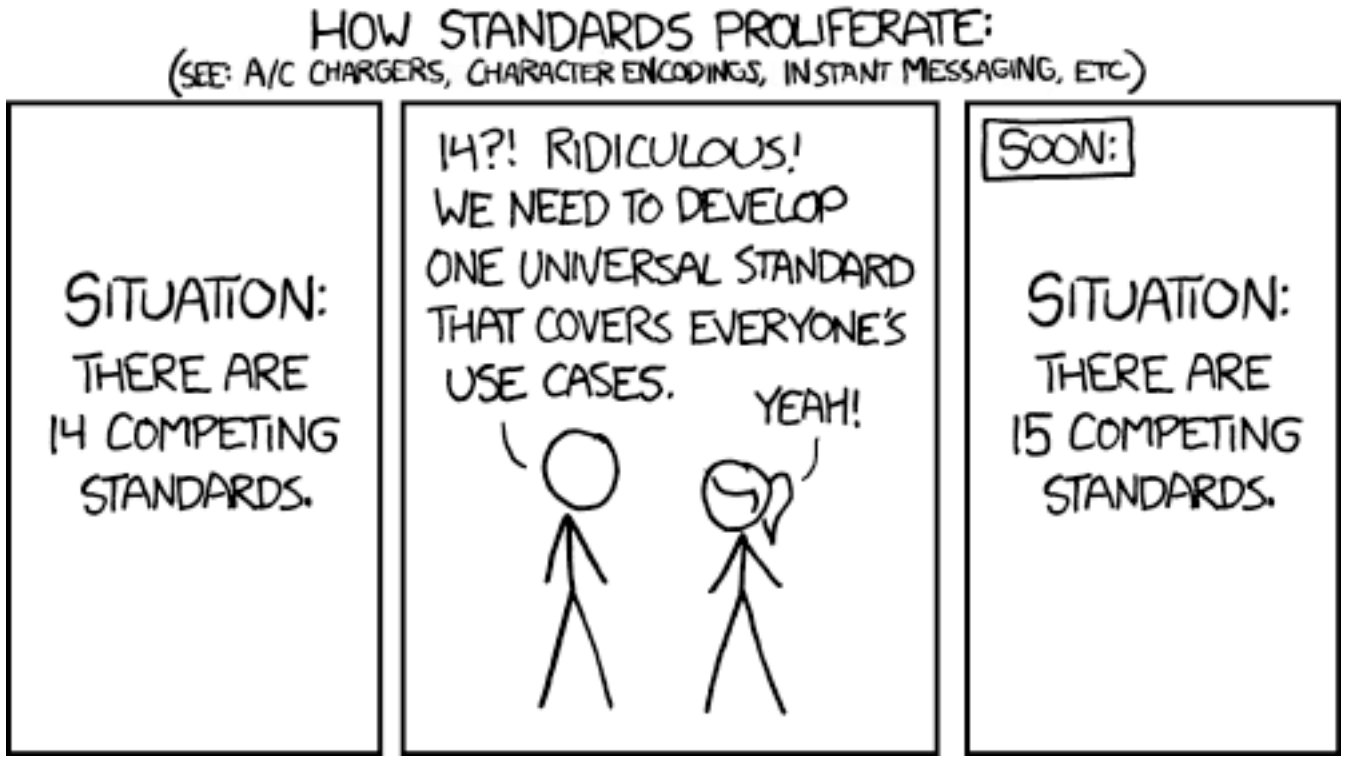
So What's the Difference

- The difference shows up when taking a word apart
- Let's store 0x11223344 (a 32-bit integer) at location 104

	Little Endian Result	Big Endian Result
ldb r_1, Mem[104]	0x44	0x11
ldh r_1, Mem[104]	0x3344	0x1122
ldw r_1, Mem[104]	0x11223344	0x11223344

	107	106	105	104
	b…	b…	b…	b…
Big Endian	44	33	22	11
Little Endian	11	22	33	44

... why do two different Endiannesses even exist?



So, about the LC-3

- Was it Big Endian or Little Endian?
- Think carefully…
- You can't tell!
- There are no instructions that manipulate more or fewer than 16 bits
- So there isn't any way to show this implementation detail!
- Was this accidental or on purpose?

Recap

Software	Hardware
Expressions & assignments	ALU instructions
Variable reuse	register addressing mode ld/st instructions
Data abstraction • struct • array	 base + offset addr mode base + index addr mode
Granularity of operands	ldb/ldh/ldw instructions addressability (byte, word)
Packing operands	Memory alignment (space/time tradeoff)
Endianness 0x11223344	Little (first byte is 0x44) / Big (first byte is 0x11)

Session ID: dag2200

Review Question 1

An instruction set...

- 0% A. Serves as a level of abstraction between software and hardware.
- 0% B. Provides the low-level details of the machine implementation.
- 0% C. Deals with the datapath and control implementation of the processor.
- 0% D. None of the above.

Session ID: dag2200

Review Question 2

Addressing mode...

- 0% A. Refers to the kinds of opcodes supported in an architecture.
- 0% B. Refers to the way the operands are specified in an instruction that accesses memory.
- 0% C. Refers to the granularity of the memory element that can be addressed in an instruction.
- 0% D. Is a critical tool for USPS operations.
- 0% E. None of the above.

Session ID: dag2200

Review Question 3

Endianness of an architecture...

0% A. Is a key determinant of processor performance.

0% B. Is a key determinant of how the compiler lays out data structures in memory.

0% C. Matters if one declares a datatype of a particular granularity and accesses it at a different granularity.

0% D. Is the official name of the party held after completing cs2200.

E. None of the above.