CPSC 213

Introduction to Computer Systems

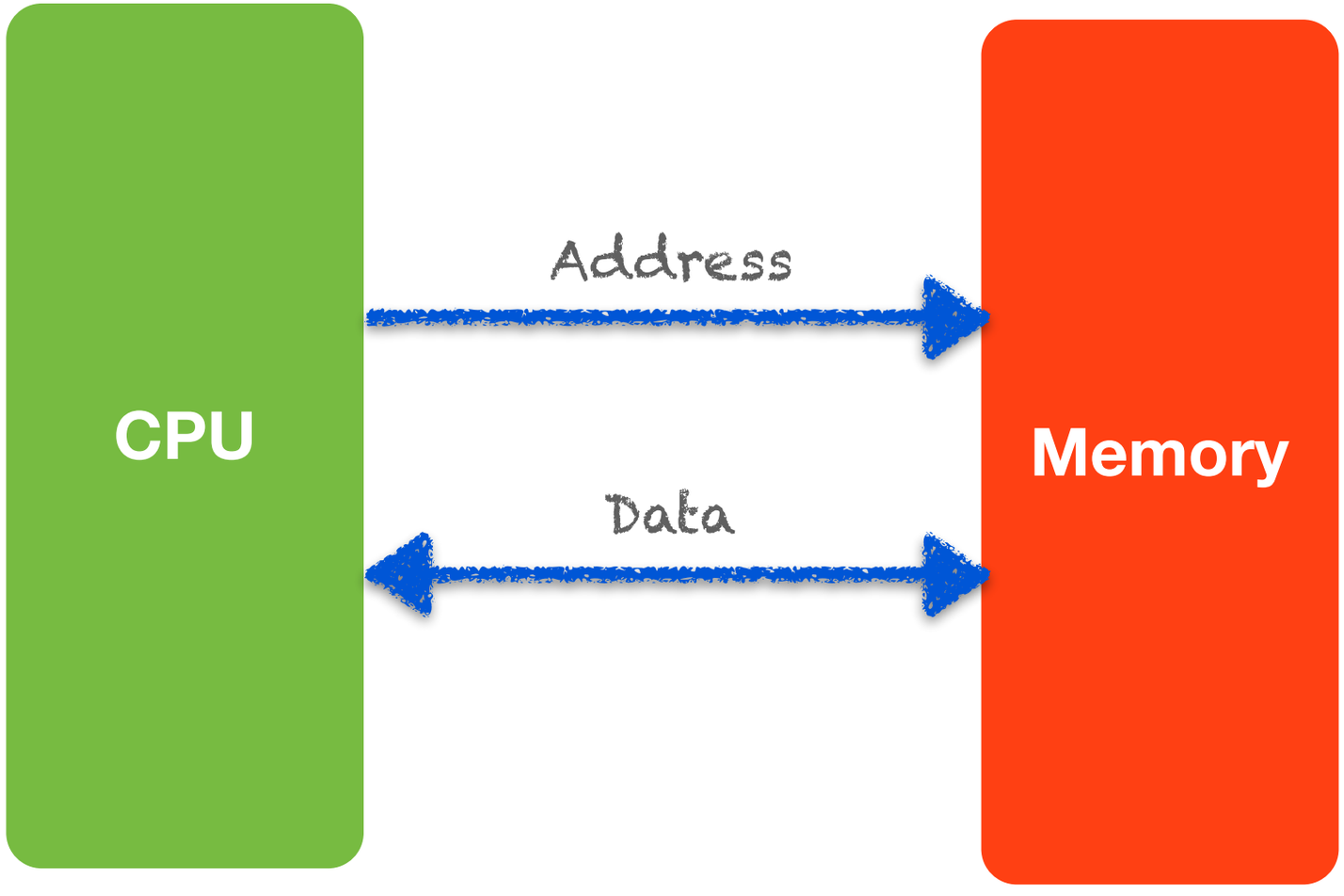
Summer Session 2019, Term 2

Unit 1b – July 4, 9

Static Scalars and Arrays

Recap

- Memory = big bag of bytes
- Program interprets those bytes!
- Multi-byte integers and endianness
- Bit operations



Overview of Unit 1b

- Reading
  - Companion: 1, 2-1-2.3, 2.4.1-2.4.3
  - Textbook: 3.1-3.2.1
- Reference (as needed)
  - Textbook: 3.1-3.5, 3.8, 3.9.3
- Learning Objectives
  - list the basic components of a simple computer and describe their function
  - describe ALU functionality in terms of its inputs and outputs
  - describe the exchange of data between ALU, registers, and memory
  - identify and describe the basic components of a machine instruction
  - outline the steps a RISC machine performs to do arithmetic on numbers stored in memory
  - translate between array-element offsets and indices.
  - distinguish static and dynamic computation for access to global scalars and arrays in C
  - describe the tradeoffs between static and dynamic arrays
  - describe why C does not perform array-bounds checking and what that means for C programs
  - translate between C and assembly language for access to global scalars and arrays
  - translate between C and assembly for code that performs simple arithmetic
  - explain the difference between arrays in C and Java
  - use C's dereference and address-of operators and pointer arithmetic to access elements of arrays

Our Approach

- Develop a model of computation
  - that is rooted in what the machine actually does
  - by examining C, bit-by-bit (comparing to Java as we go)
- The processor
  - we will design (and you will implement) a simple instruction set
  - based on what we need to compute C programs
  - similar to a real instruction set (MIPS)
- The language
  - we will act as compiler to translate C into machine language
  - bit by bit, then putting the bits together to do interesting things
  - edit, debug and run using simulated processor to *visualize* execution

We will start here …

Java:

```java
public class Foo {
  static int   a;
  static int[] b;   // array is not static, so skip for now

  public void foo () {
    a = 0;
  }
}
```

C:

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

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

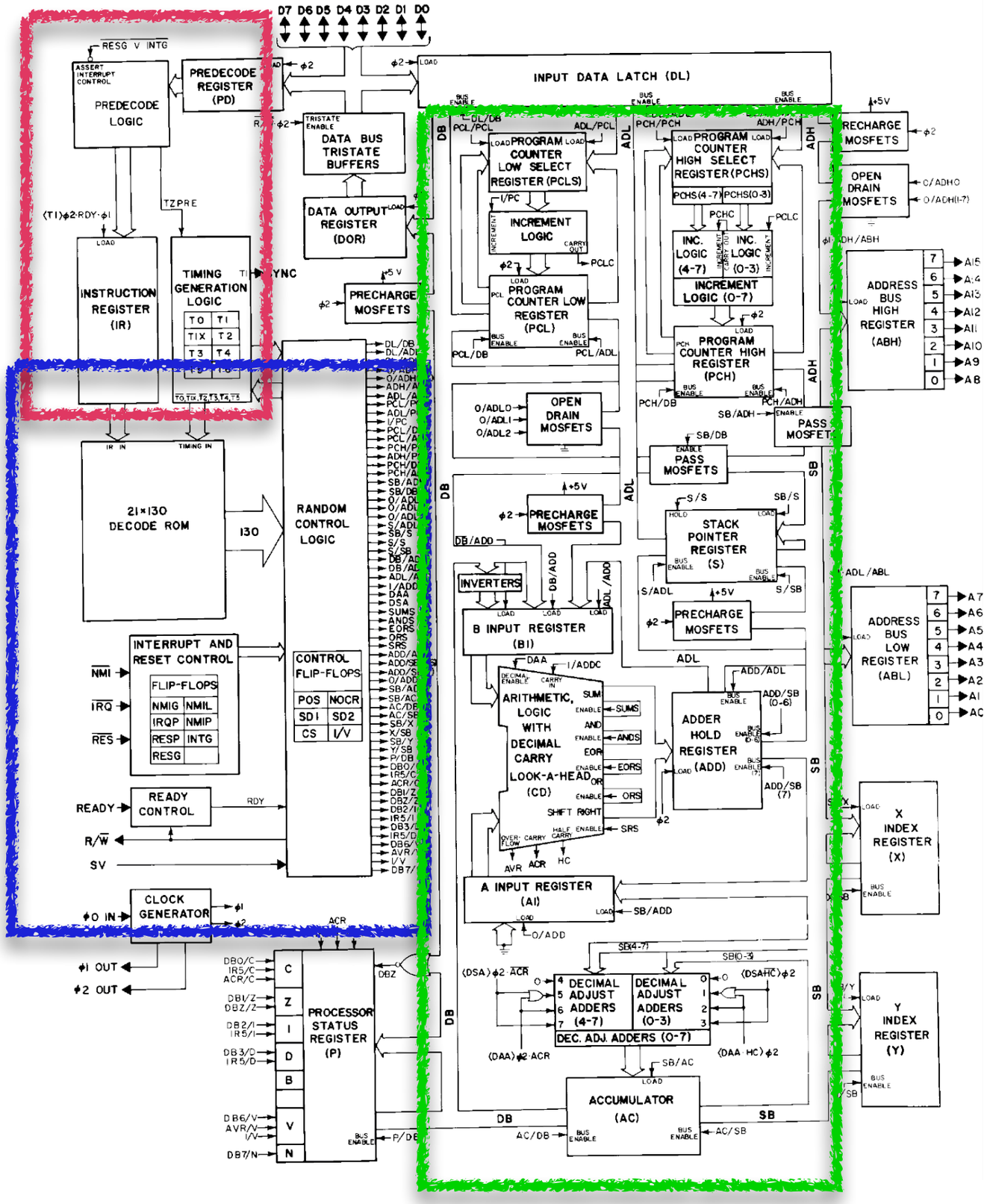
The CPU

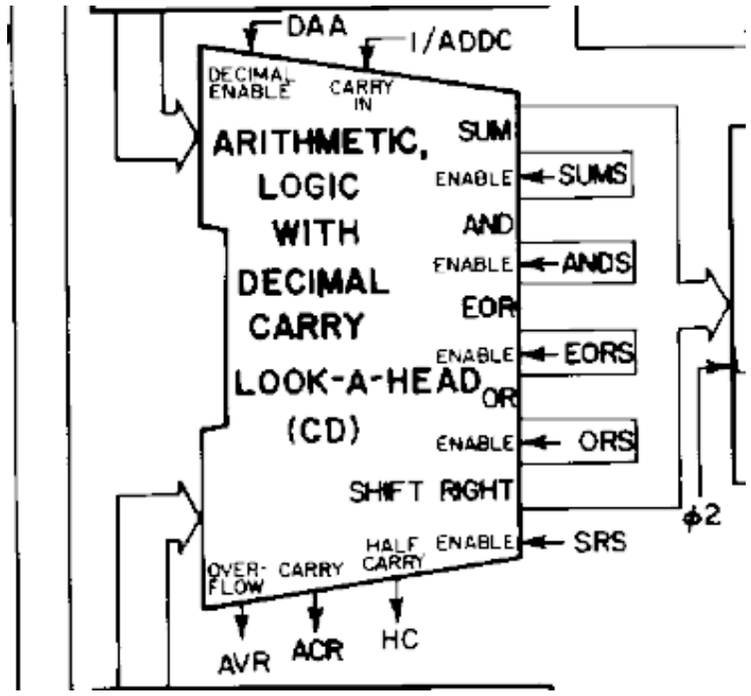
- CPUs execute *instructions*, not C or Java code
- Execution proceeds in three phases:
  - Fetch - load the next instruction from memory
  - Decode - figure out what the instruction will do
  - Execute - do what the instruction asks
- These phases are looped over and over again forever

The CPU

- Instructions are very simple
  - Add/Subtract two numbers
  - And/Or two numbers
  - Bit shift a number
  - Read/Write memory
  - Control flow (later…)
- Operations carried out by the ALU (Arithmetic & Logic Unit)
- ALU is programmed by the Execute step
  - ALU carries out one mathematical or logical operation based on inputs

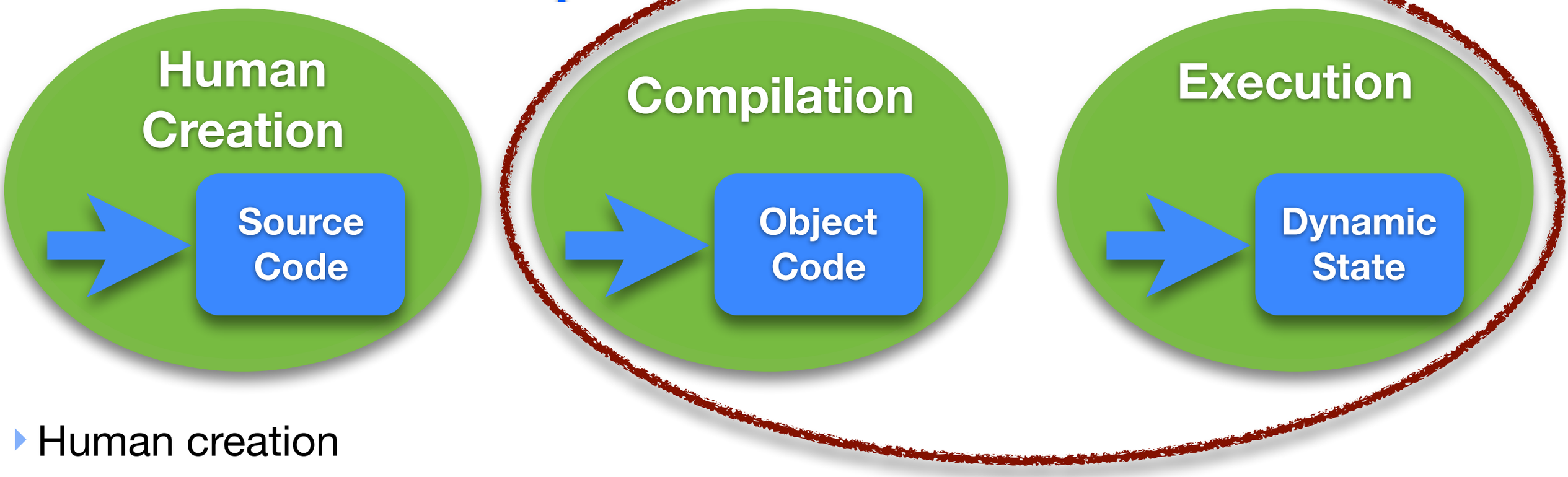
The CPU





Donald F. Hanson, WCAE 1995

Phases of Computation

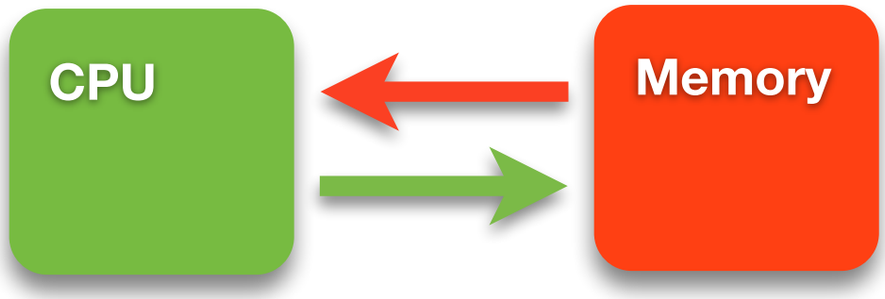


- Human creation
  - design program and describe it in high-level language
- Compilation
  - convert high-level, human description into machine-executable text
- Execution
  - a physical machine executes the text
  - parameterized by input values that are unknown at compilation
  - producing output values that are unknowable at compilation
- Two Crucial Phases of Computation
  - **STATIC** anything that the compiler can possibly compute is called *static*
  - **DYNAMIC** anything that can not possibly be know until the program runs is called *dynamic*

First

An Introduction to Computing with a Simple Machine

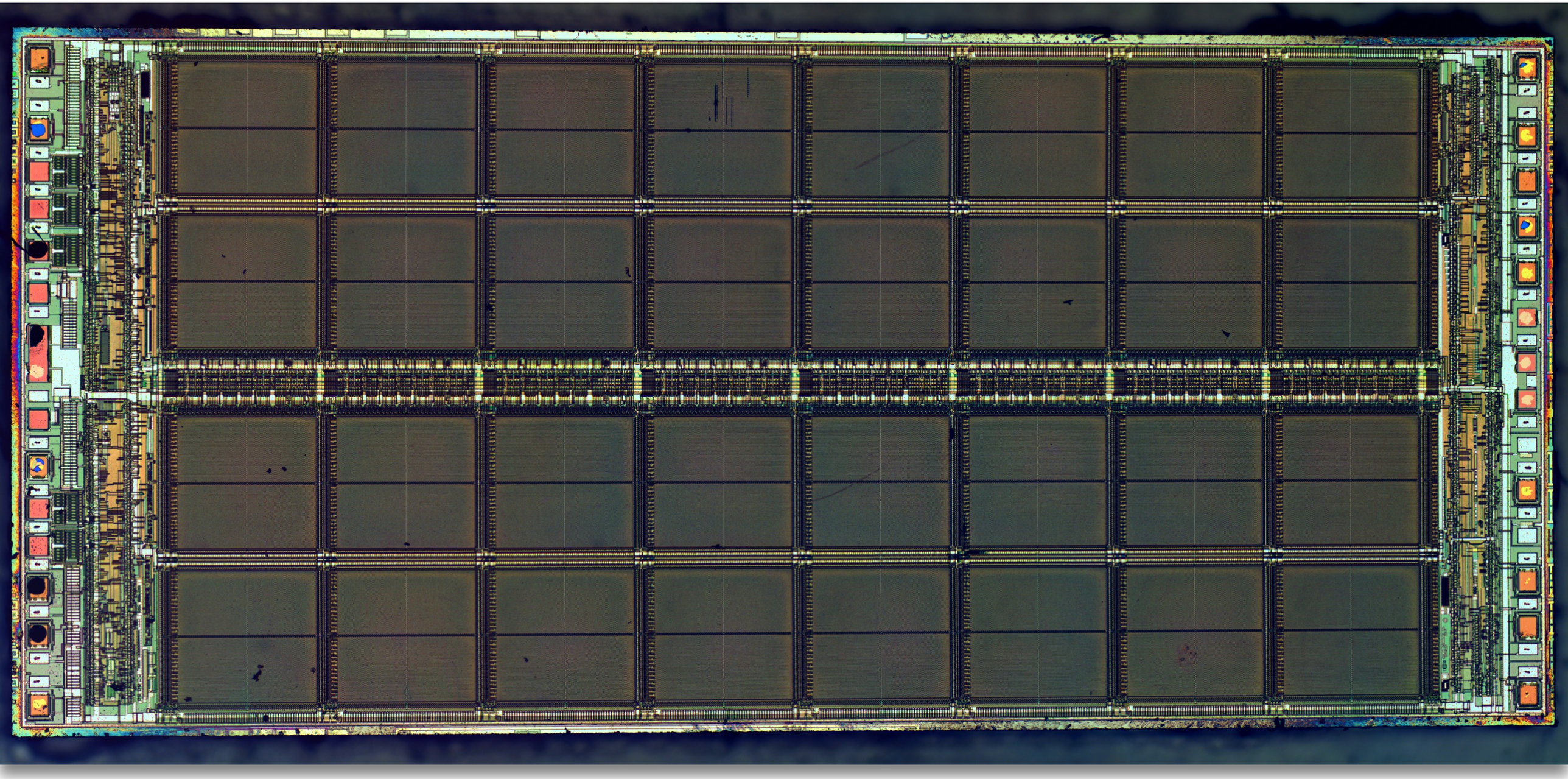
The Processor (CPU)



How many memory accesses?

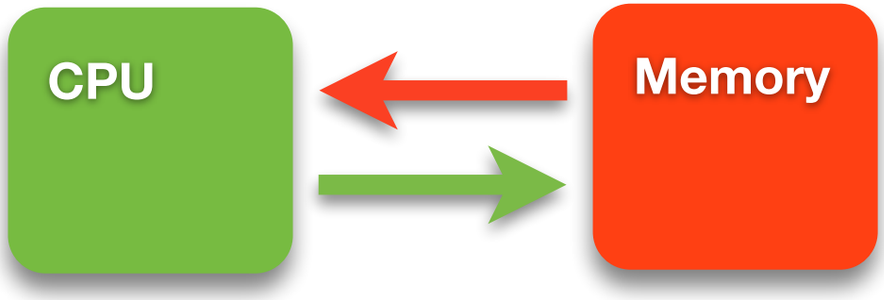
- Implements a set of simple instructions
  - each instruction is implement using logic gates, built from transistors
  - these transistors are the fundamental mechanism of computation
  - the fewer and simpler the better
- For example, we'll want to do arithmetic
  - we'll have an add instruction that does something like: C = A + B
    - A, B and C are values stored in memory — named by 32-bit addresses (in our case, 64-bits typically today)
  - every instruction is encoded as a set of bits stored in memory
    - the instruction will need to name the operation and the three operands (assume their addresses are constants
  - the instruction might look something like this (in hex)

The Memory (RAM)



Micron MT4C1024 128KB RAM, zeptobars.org

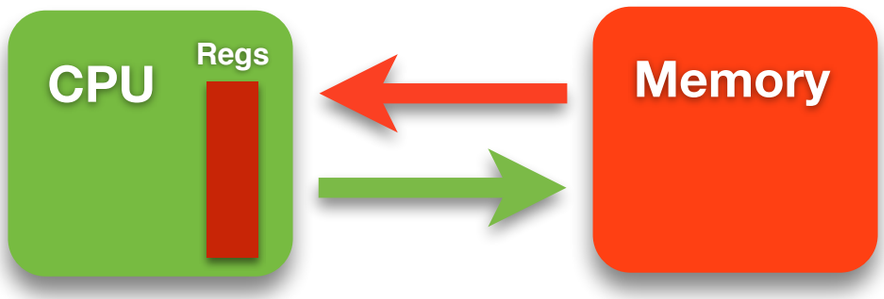
The Problems with Memory Access

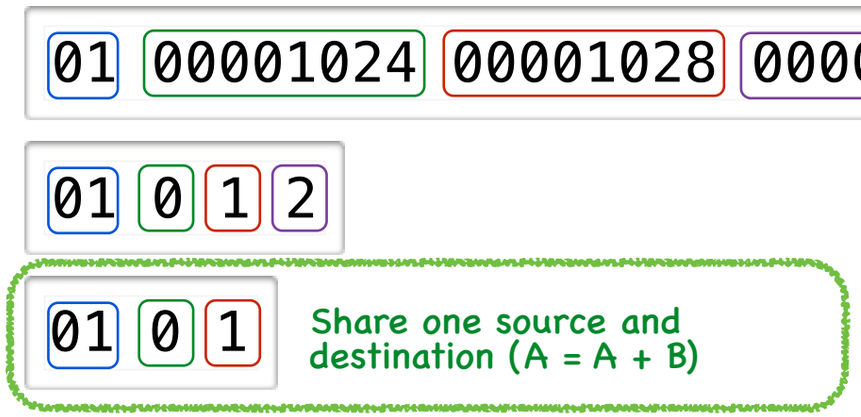



```
01 00001024 00001028 0000102c
```

- Accessing memory is slow
  - ~20–100 cycles for every memory access
    - there are tricks to help (more in CPSC 313), but its still slow
  - fast programs avoid accessing memory whenever possible
- Big instructions are costly
  - memory addresses are big, thus so are instructions that access memory
  - if most instructions are big, then programs will be big too
    - memory is a finite resource; we should use it efficiently
  - if most instructions are big, then programs will be slow
    - CPU reads an instruction every cycle, caching them locally in case they are accessed again
    - if enough instructions are cached, the CPU rarely waits to get instruction from memory
    - if instructions are big, fewer will fit in the cache and there will be more memory stalls
      - the CPU must stall whenever it needs an instruction that is not stored locally … to get it from memory
    - also, big instructions take longer to transfer to CPU and use more resources in the process

The General-Purpose Register File



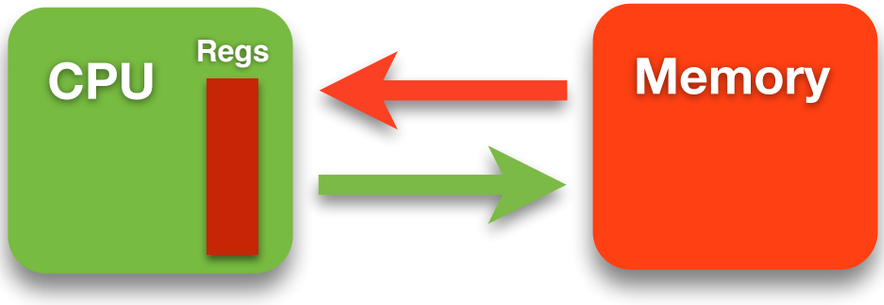


- Register File
  - small, fast, on-chip memory directly manipulated by instructions
  - each register is named by a number (e.g., 0 — 7)

8 times smaller than a 32-bit memory address

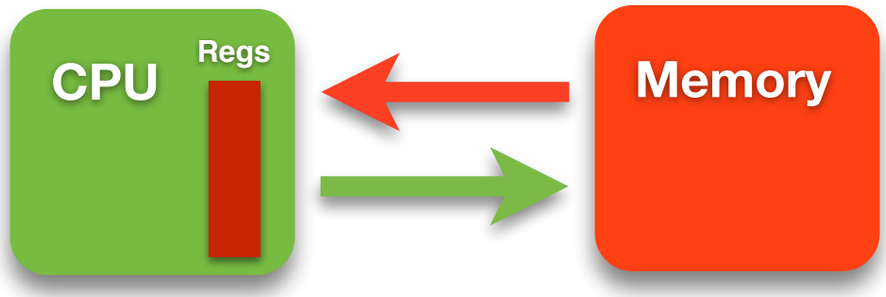
- Registers
  - size of largest integer (e.g., 32 bits or maybe 64)
  - built from fast memory (roughly single-cycle access)
- Instructions
  - load data from memory into register; store data from register into memory
    - these are large and slow
  - other instructions access data in registers
    - these are small and fast

Special-Purpose Registers



- A *special-purpose register* can only be used for certain purposes
  - May not be accessible by all instructions
  - May have special meaning or be treated specially by CPU
- Some CPUs have many
- **PC**, the Program Counter
  - Address of the next instruction to execute
- **IR**, the Instruction Register
  - Instruction that has just been loaded from RAM
  - Separate smaller registers for the parts

Simple Machine Hardware Architecture

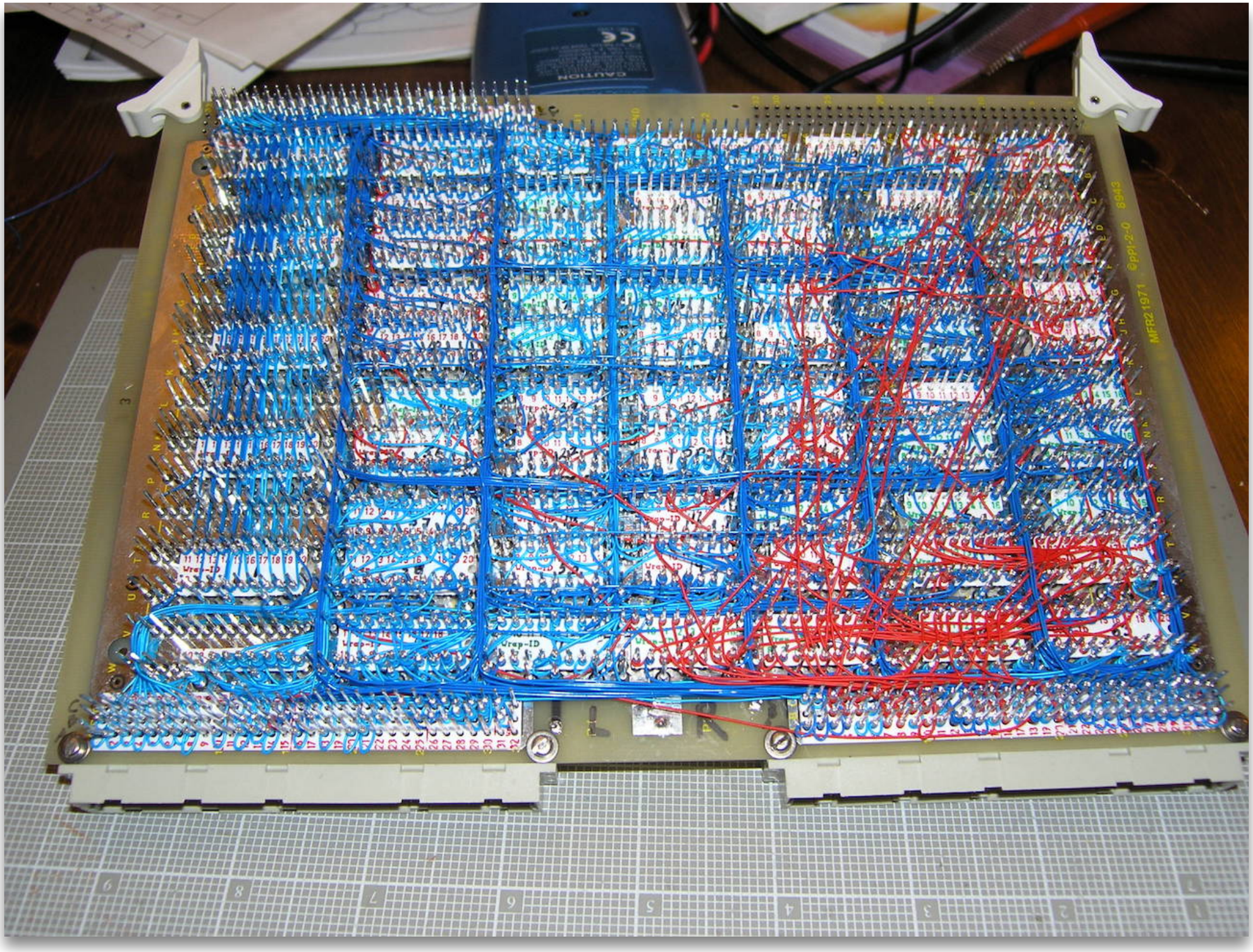


- CPU

The Program and the Machine

- Register File
- Main Memory

Let's build a CPU!



Home-made CPU control board, homebrewcpu.com

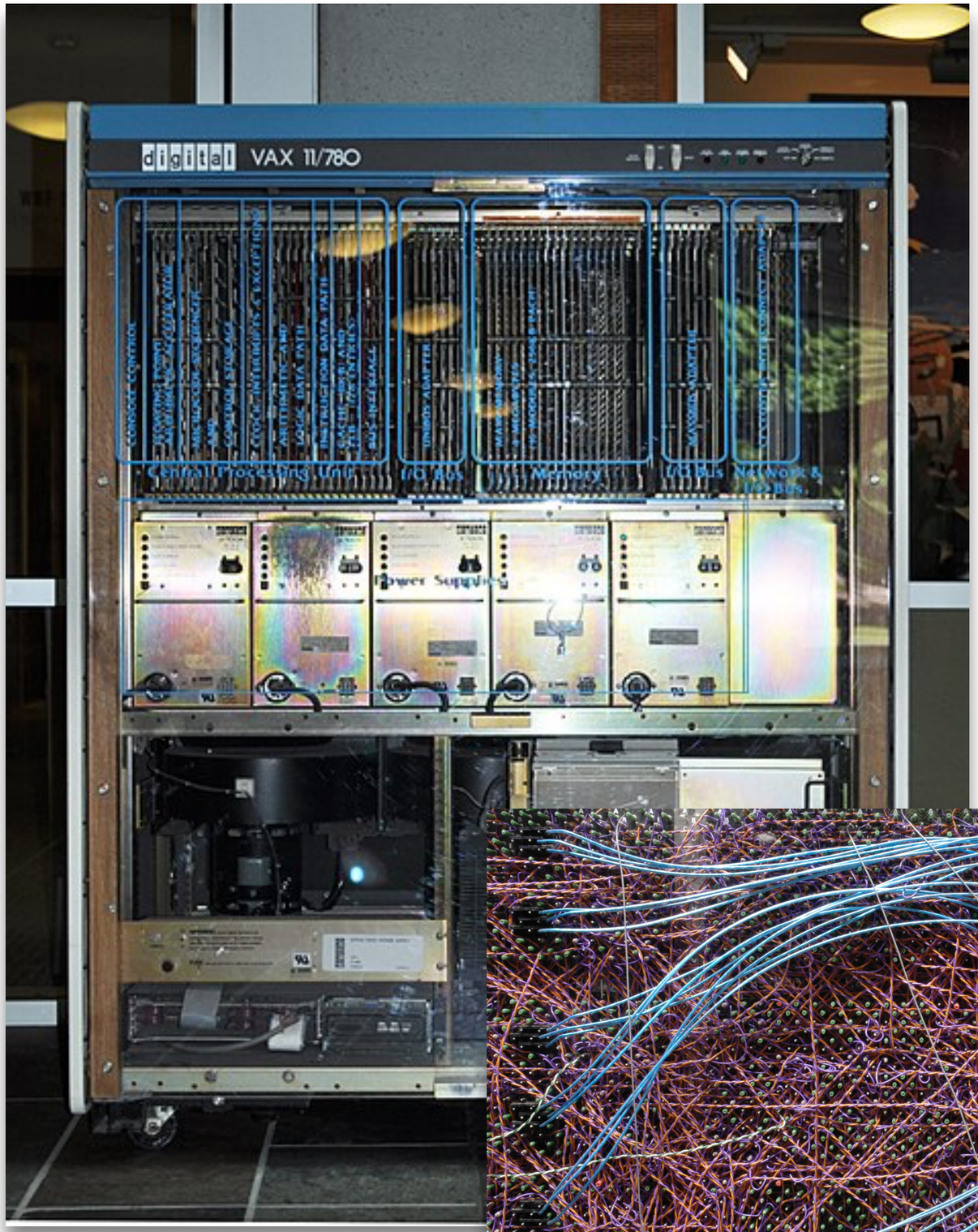
The Simulator …

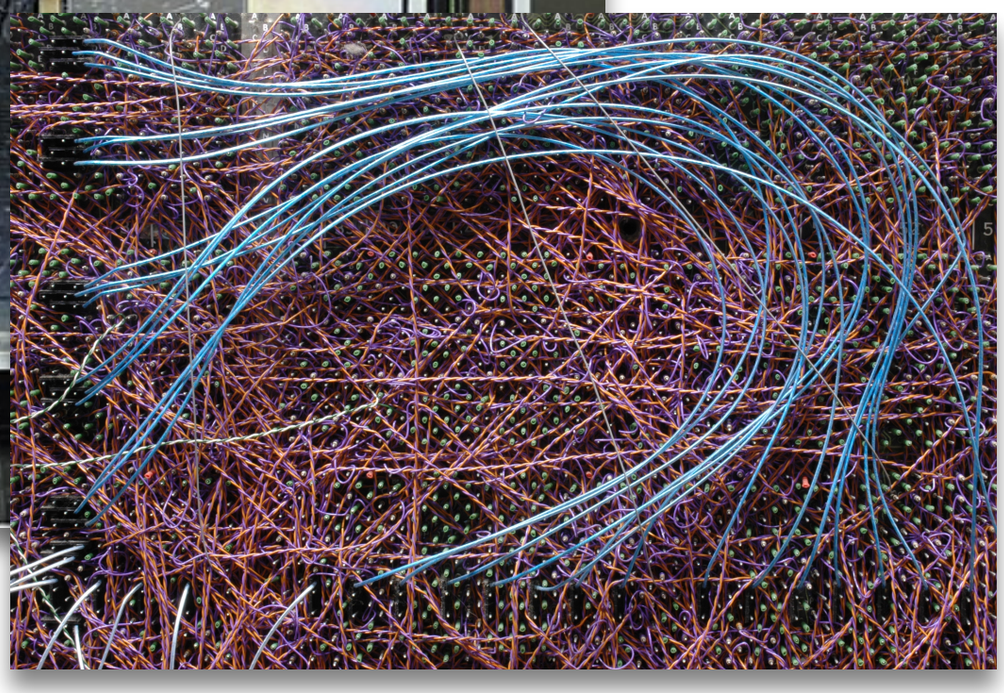
The Processor

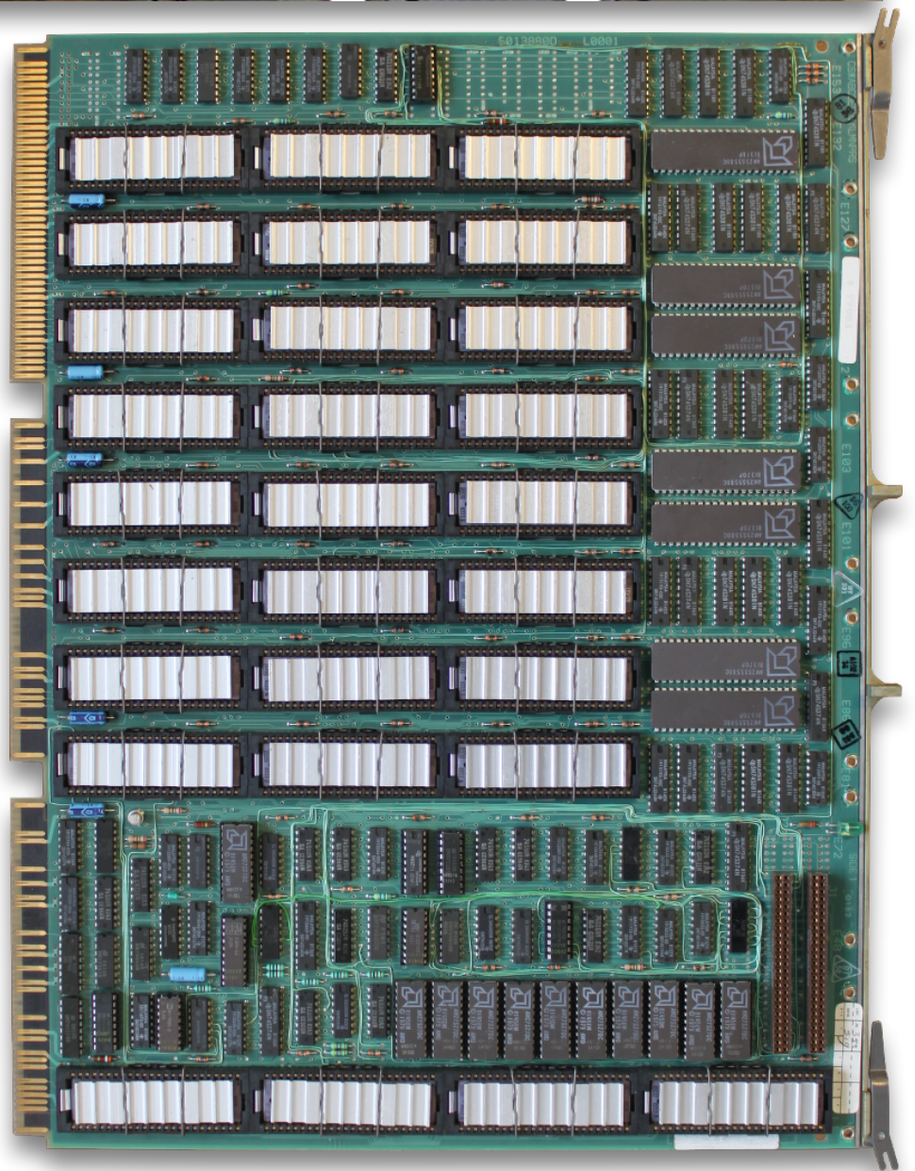
Instruction Set Architecture (ISA)

- The ISA is the interface to a processor implementation
  - defines the instructions the processor implements
  - defines the format of each instruction
- Types of instruction
  - math
  - memory access
  - control transfer (gotos and conditional gotos)
- Design alternatives
  - simplify compiler design (CISC such as Intel Architecture 32 (aka x86))
  - simplify processor implementation (RISC)
- Instruction format
  - is a sequence of bits (or, typically, in hex)
  - an opcode and set of operand values
- Assembly language
  - symbolic representation of machine code

DEC VAX 11/780

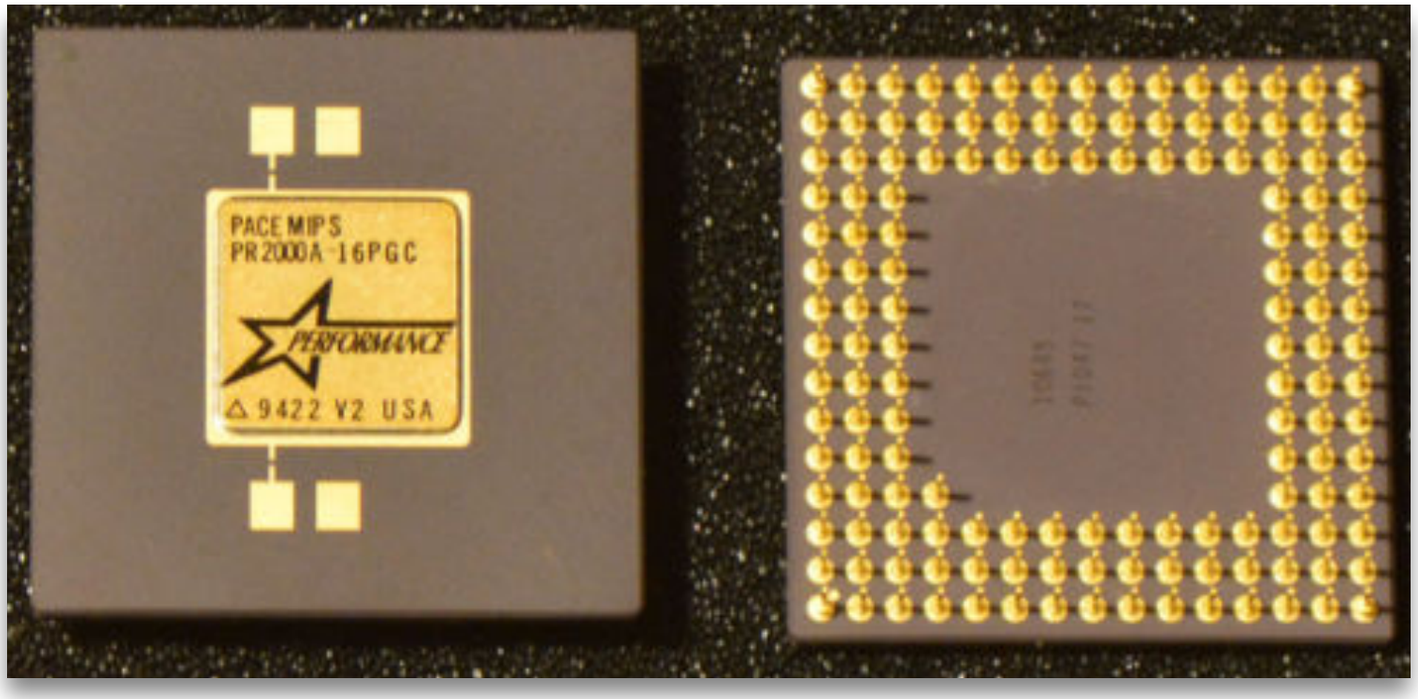




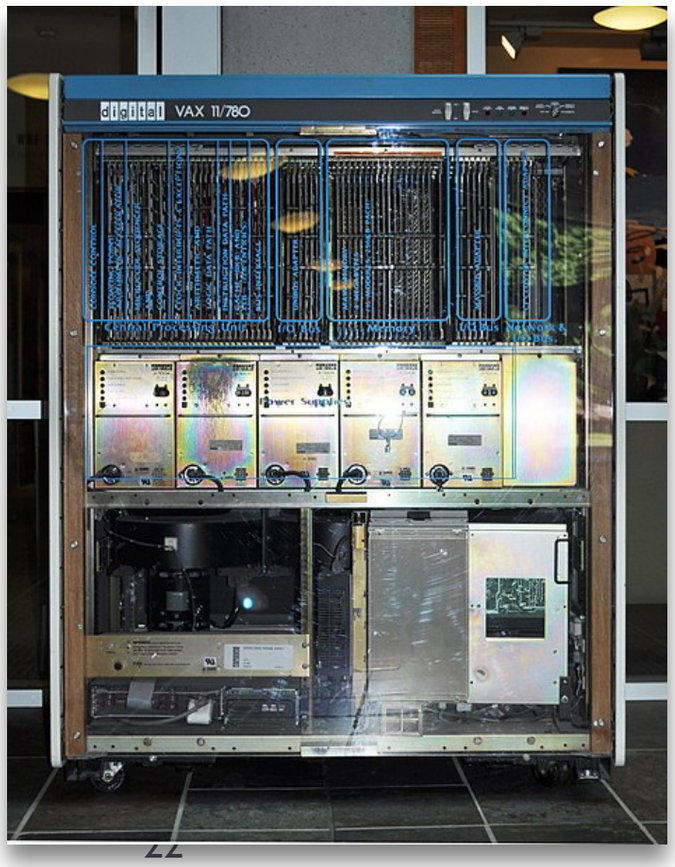


MIPS R2000

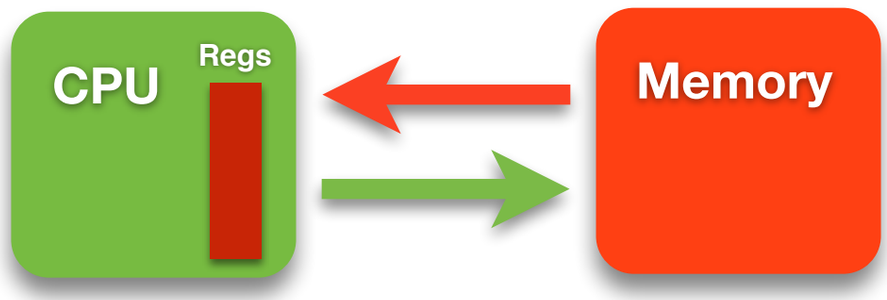
- R2000
  - 1986
  - 8.3 to 15 Mhz
  - 110,000 transistors
  - 80 mm² die
  - 2,000 nm feature size CMOS



- VAX 11/780
  - 1977
  - 5 Mhz
  - 2-MB – 8-MB main memory



Describing Instruction Semantics



- RTL is
  - a simple, convenient pseudo language to describe instruction semantics
  - easy to read and write, directly translated to machine steps
- Syntax
  - each line is of the form `LHS ← RHS`
  - `LHS` is memory or register specification
  - `RHS` is constant, memory, or arithmetic expression on two registers
- Register and Memory are treated as arrays
  - `m[a]` is value in memory at address `a`
  - `r[i]` is register number `i`
- For example
  - `r[0] ← 10`
  - `r[1] ← m[r[0]]`
  - `r[2] ← r[0] + r[1]`

Static Variables of Built-In Types

Variables

- What are they?
  - named storage location for values
- What features to they have?
  - name
  - size
  - type
  - scope
  - lifetime
  - memory location; i.e., an address
  - value
- Static or Dynamic in Java (or if you know it, C/C++)?
  - which of these are determined at compile time?
  - which are determined (and thus can change) while the program is running?
- Allocation
  - is assigning a memory location (i.e., an address) to a variable
  - WHEN does this happen? HOW is location found?

Static Variables, Built-In Types (S1-global-static)

- Java
  - static data members are allocated to a class, not an object
  - they can store built-in scalar types or references to arrays or objects (references later)

```
public class Foo {
  static int   a;
  static int[] b;   // array is not static, so skip for now

  public void foo () {
    a = 0;
  }}
```

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

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

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

Static Variable Allocation

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

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

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

Static Memory Layout

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

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

Static Variable Access (scalars)

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

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

```
a = 0;
```

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

Static Memory Layout

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

- Key Observation
  - address of **a**, **b[0]**, **b[1]**, **b[2]**, ... are constants known to the compiler
- Use RTL to specify instructions needed for **a = 0**

Now Generalize … What about, for example

a = x + y?

c = a?

Question 1b.1

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

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

```
a = 0;
```

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

Static Memory Layout

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

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

Static Variable Access (static arrays)

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

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

```
a = 0;
```

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

Static Memory Layout

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

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

Choosing the instructions

```
a = 0;
```

```
m[0x1000] ← 0x0
```

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

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

ISA Design Goals

- minimize # of memory instructions in ISA
- at most 1 memory access per instruction
- minimize # of total instructions in ISA
- minimize instruction size

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

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

and load version of #4 (5)

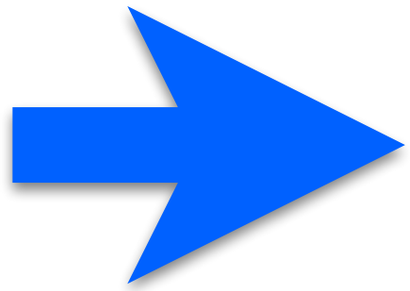
The First Five Instructions

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

Translating the Code

- To RTL

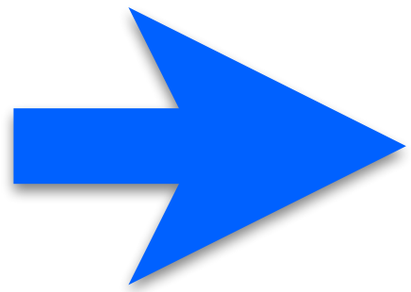
```
int a;
int b[10];

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



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

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

- To Assembly Code

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

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

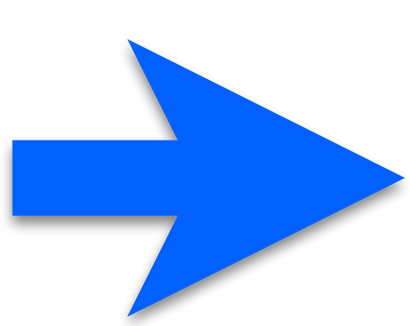


```
ld $0, r0
ld $0x1000, r1
st r0, (r1)

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

If a human wrote this assembly

- list static allocations, use labels for addresses, add comments

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

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



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

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

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

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

The SM213 ISA

(Part 1)

The Simple Machine (SM213) ISA

- Architecture
  - Register File — 8, 32-bit general purpose registers
  - CPU — one cycle per instruction (fetch + execute)
  - Main Memory — byte addressed, Big Endian integers
- Instruction Format
  - 2 or 6 byte instructions (each character is a hex digit)
    - **`x-01`**, **`xxsd`**, **`x0vv`** or **`x-sd vvvvvvvv`**
  - where
    - **`x`** is an *opcode* (unique identifier for this instruction)
    - **`-`** means unused
    - **`s`** and **`d`** are operand register numbers
    - **`vv vvvvvvvv`** are immediate / constant values

Machine and Assembly Syntax

- Machine code
  - `[ addr: ] x-01 [ vvvvvvvv ]`
    - `addr:` sets starting address for subsequent instructions
    - `x-01` hex value of instruction with opcode x and operands 0 and 1
    - `vvvvvvvv` hex value of optional extended value part instruction
- Assembly code
  - `( [label:] [instruction | directive] [# comment] | )*`
    - `directive :: (.pos number) | (.long number)`
    - `instruction :: opcode operand+`
    - `operand :: $literal | reg | offset (reg) | (reg,reg,4)`
    - `reg :: r 0..7`
    - `literal :: number`
    - `offset :: number`
    - `number :: decimal | 0x hex`

Memory-Access Instructions

- Load and Store with different *Addressing Modes*

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

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

Basic Arithmetic, Shifting NOP and Halt

- Arithmetic

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

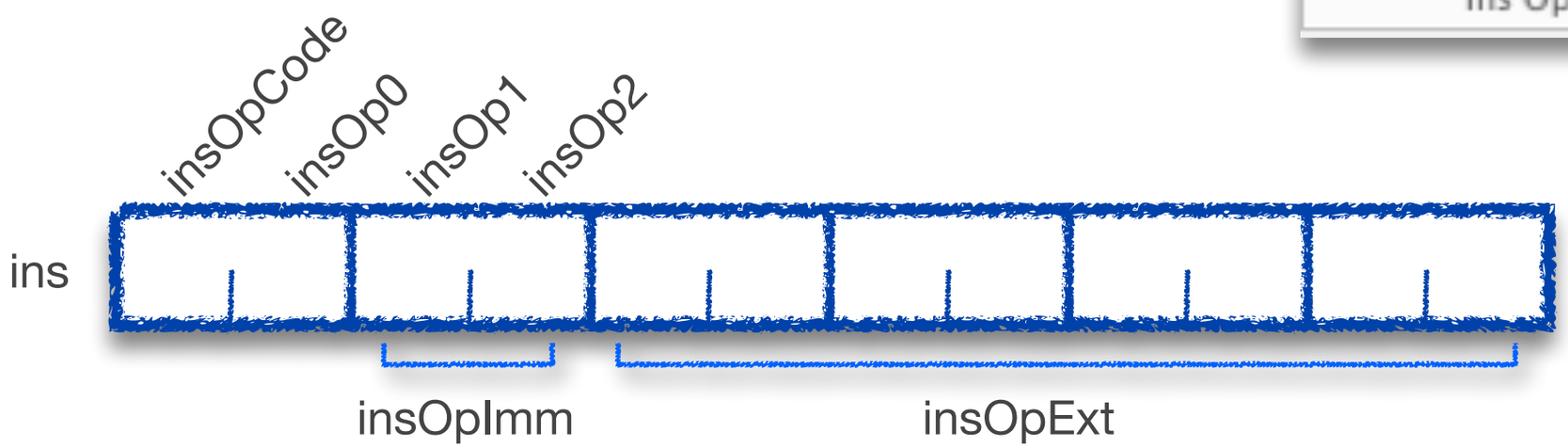
- Logical (Shifting), NOP and Halt

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

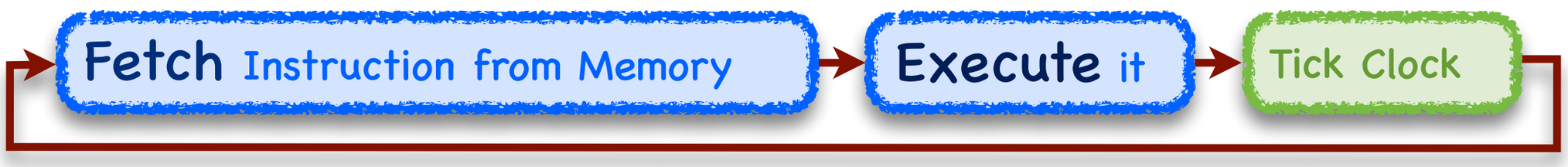
The CPU Implementation

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

- Internal state
  - pc address of ***next*** instruction to fetch
  - instruction the value of the current instruction



- Cycle stages



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

Java Syntax

- Internal Registers
  - insOp0, insOp1, insOp2, insOpImm, insOpExt, pc
  - .get()
  - .set (value)

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

- General Purpose Registers
  - reg.get (registerNumber)
  - reg.set (registerNumber, value)

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

- Main Memory
  - mem.readInteger (address)
  - mem.writeInteger (address, value)

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

CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 1b – July 4, 9

Pointers and Dynamic Arrays

Types

- In Java & C
  - Every **variable** has a **type**
  - Compiler enforces type safety: restricts what you can do with a variable
- In Java
  - Types are arranged in a hierarchy (except primitives)
  - Strong type safety: references can't be casted to incompatible types
  - Runtime type information: every object carries its type in memory
- In C
  - A type defines the *size* and *interpretation* of a bunch of bytes
  - `sizeof` operator can determine the size *at compile-time*
  - No runtime type information: all that matters is how code accesses bytes
- In Assembly
  - No types! Everything is a number! Instructions decide how to interpret them.

Dynamic Arrays

Global Dynamic Array

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

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

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

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

```
int  a;
int *b;

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

# of bytes to allocate

Static vs Dynamic Arrays

- Declared and allocated differently, but accessed the same

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

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

```
int  a;
int *b;

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

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

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

```
0x2000: value of b
```

Then when the program runs

- the dynamic array is allocated by a call to malloc, say at address 0x3000
- the value of variable b is set to the memory address of this array

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

```
0x2000: 0x3000
```

```
0x3000: value of b[0]
0x3004: value of b[1]
...
0x3024: value of b[9]
```

Generating code to access the array

- for the dynamic array, the compiler generates an additional load for b

Static Array

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

Dynamic Array

```
r[0]                ← 0x1000
r[1]                ← m[r[0]]   load a
r[2]                ← 0x2000
r[3]                ← m[r[2]]   load b
m[r[3]+r[1]*4]      ← r[1]      b[a]=a
```

malloc

- Dumb but functional, illustrative implementation

```
int next = 0;
int memory[100000000];

void *malloc(int size) {
  int *result = &memory[next];
  next += (size + 3) / 4;
  return result;
}
```

- The hard part is *free* - releasing memory you don't need

How C Arrays are Different from Java

- Terminology
  - use the term ***pointer*** instead of ***reference***; they mean the same thing
- Declaration
  - the type is a pointer to the type of its elements, indicated with a *
- Allocation
  - malloc allocates a block of bytes; no type; no constructor
- Type Safety
  - any pointer can be type cast to any pointer type
- Bounds checking
  - C performs no array bounds checking
  - out-of-bounds access manipulates memory that is not part of array
  - this is the major source of virus vulnerabilities in the world today

Question: Can array bounds checking be performed statically?

* what does this say about a tradeoff that Java and C take differently?

```
void izero(int *a, int n) {
   for(int i=0; i<n; i++)
      a[i] = 0;
}
```

Array Access in Assembly

Static Array

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

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

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

Dynamic Array

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

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

.pos 0x2000
b:
.long 0             # the variable b
```

b_data

For Simulator …

For the simulator, we give the allocation of `b_data` even though this would have been allocated dynamically, and give b a value even though this would have happened while the program was running. And so, while the compiler would have assigned `b` the value 0 as you see above, in the snippet you'll see "`b: .long b_data`" instead.

```
.pos 0x3000
b_data:
.long 0             # the variable b[0]
.long 0             # the variable b[1]
...
.long 0             # the variable b[9]
```

Pointers

- In both languages
  - A pointer/reference variable is **just a number** - a memory address
  - A pointer/reference type refers to another type
- In Java
  - References can be casted up and down the hierarchy (Object is the root)
  - Runtime checks type during casting
  - References can’t be used like numbers, compiler forbids it
- In C
  - Pointers can be casted to other pointers or even numbers
  - Compiler lets you treat pointers like numbers
  - No checks in casting at all (none from compiler or runtime)

Pointers

- Pointers are just numbers (addresses)!
- Pointer type determines size & meaning

C and Java Arrays and Pointers

- In both languages
  - an array is a list of items of the same type
  - array elements are named by non-negative integers starting with 0
  - syntax for accessing element `i` of array `b` is `b[i]`
- In Java
  - variable a stores a pointer to the array
  - b[x] = 0 means *m[m[b] + x * sizeof(array-element)] ← 0*
- In C
  - variable a can store a pointer to the array or the array itself
  - b[x] = 0 means *m[b + x * sizeof(array-element)] ← 0*
or *m[**m[b]** + x * sizeof(array-element)] ← 0*
  - dynamic arrays are just like all other pointers
    - stored in TYPE*
    - **access with either a[x] or *(a+x)**

Example

- The following C programs are identical

```
int *a;
a[4] = 5;
```

```
int *a;
*(a+4) = 5;
```

- For array access, the compiler would generate this code

```
r[0]              ← a
r[1]              ← m[r[0]]
r[2]              ← 4
r[3]              ← 5
m[r[1]+r[2]x4]    ← r[3]
```

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

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

- So, what does this tell you about pointer arithmetic in C?

Adding X to a pointer of type Y*, adds X * sizeof(Y)
to the pointer's memory-address value.

Pointer Arithmetic in C

- Its purpose
  - an alternative way to access dynamic arrays compared to a[i]
- Adding or subtracting an integer index to a pointer
  - results in a new pointer of the same type
  - value of the pointer is offset by index times size of pointer's referent
  - for example
    - adding 3 to an int* yields a pointer value 12 larger than the original
- Subtracting two pointers of the same type
  - results in an integer
  - gives number of referent-type elements between the two pointers
  - for example
    - `(& a[7]) - (& a[2]) == 5 == (a+7) - (a+2)`
- other operators
  - `& X` the address of X
    - `&a[0]` (or just `a` with no `&` and no `[]`) equals the address of the first element of the array
  - `* X` the value X points to

Question 1b.2

```
int *c;

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

- What is the equivalent Java statement to
  - *that is, which one makes the same change to the array C*

A. c[0] = c[3];

B. c[3] = c[6];

C. there is no type-safe equivalent

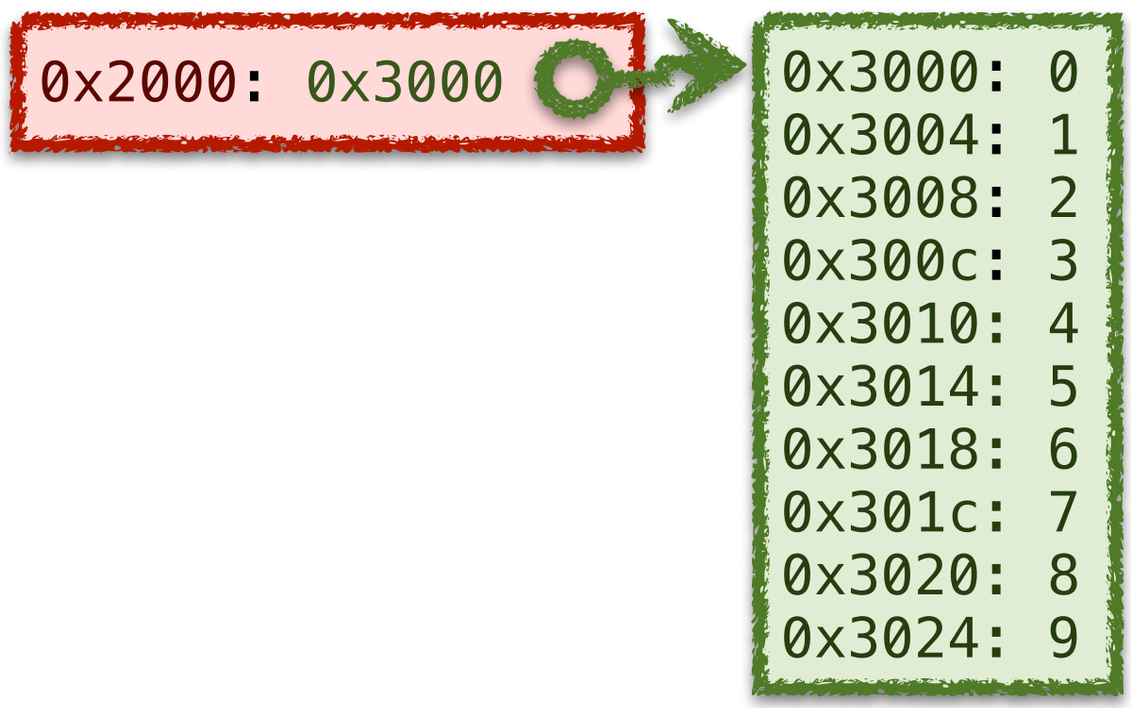
D. not valid, because you can't take the address of a static in Java

Looking more closely

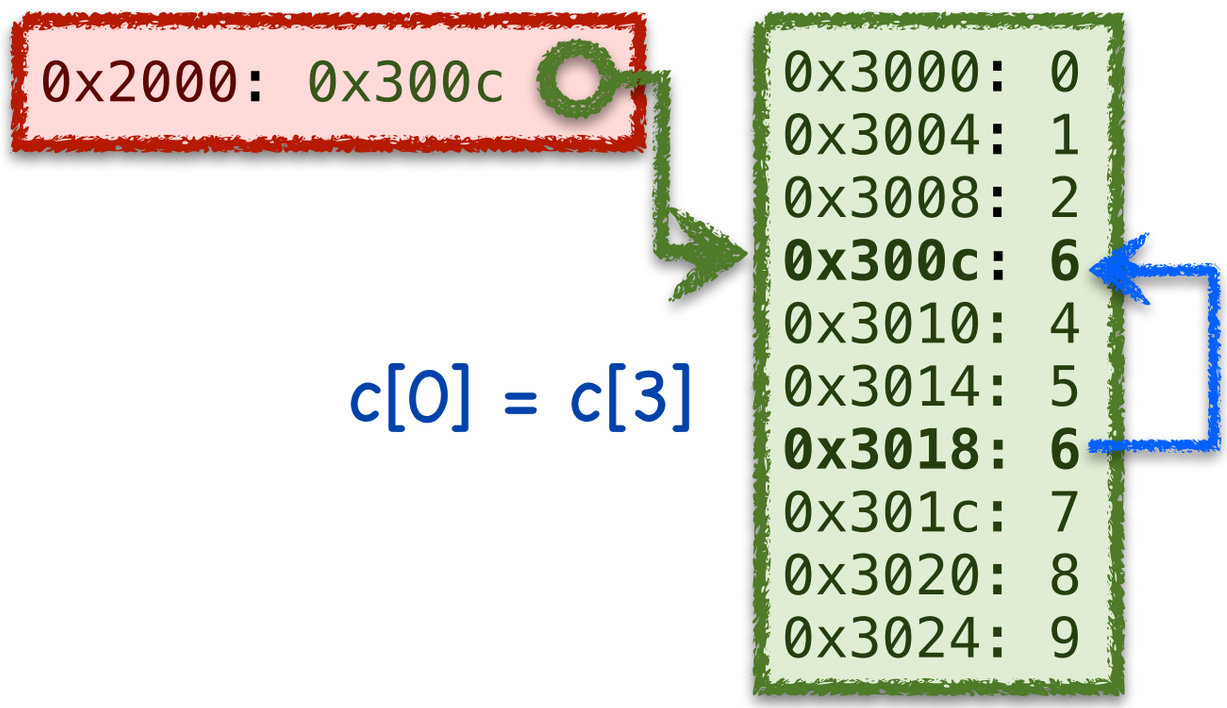
```
c = &c[3];
*c = *&c[3];
```

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

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

Before



After



- And in assembly language

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

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

```
ld $0x2000, r0                 # r0 = &c
ld (r0), r1                    # r1 = c
ld $12, r2                     # r2 = 3*sizeof(int)
add r1, r2                     # r2 = c+3
st r2, (r0)                    # c  = c+3

ld $3, r3                      # r3   = 3
ld (r2,r3,4), r4               # r4   = c[3]
st r4, (r2)                    # c[0] = c[3]
```

Aside: Strings in C

- A3 shows real C programs: `int main(int argc, char **argv)`
- `argc` = *count* of arguments
- `argv` = *vector* (array) of arguments. `argv[0]` = program name
- `char *` = array of characters (1 byte each) = *string*
  - Terminated by a NUL character
  - *e.g.* `"Hello World!"` =

H	e	l	l	o		W	o	r	l	d	!	\0

  - Characters translated to numbers using ASCII
- `char **` = array of strings
- Use `atoi` or `strtol` to convert string to int

Aside: I/O in C

- *printf:* Print Formatted - print some data using format
  - %s = string
  - %d = decimal (signed) integer
  - %u = unsigned integer
  - %x = hexadecimal integer
  - For each format specifier, add one function argument
  - *fprintf:* print to file (*e.g.* stderr = standard error, like System.err)
  - Example: `printf("My name is %s and I'm %d years old", "Robert", 30);`
- *scanf:* Scan Formatted - read some data using format
  - All arguments must be pointers!

Aside: Declaring Pointers in C

- What happens when you want to declare multiple pointers?
  - To declare multiple ints, just `int a, b;`
  - To declare multiple pointers, what about `int *a, b;`?
- No! The * belongs to *each* variable in the line!
- Correct way to declare multiple pointers
  - **`int *a, *b;`**
- Can combine multiple levels of pointers and initialization…
  - `int a=42, *b=&a, **c=&b;`
  - Don’t do this at home
- Same thing applies to arrays
  - `int a, b[5], c[10];`

Summary: Static Scalar and Array Variables

- Static variables
  - the compiler knows the address (memory location) of variable
- Static scalars and arrays
  - the compiler knows the address of the scalar value or array
- Dynamic arrays
  - the compiler does not know the address the array
- What C does that Java doesn't
  - static arrays
  - arrays can be accessed using pointer dereferencing operator
  - arithmetic on pointers
- What Java does that C doesn't
  - type-safe dynamic allocation
  - automatic array-bounds checking