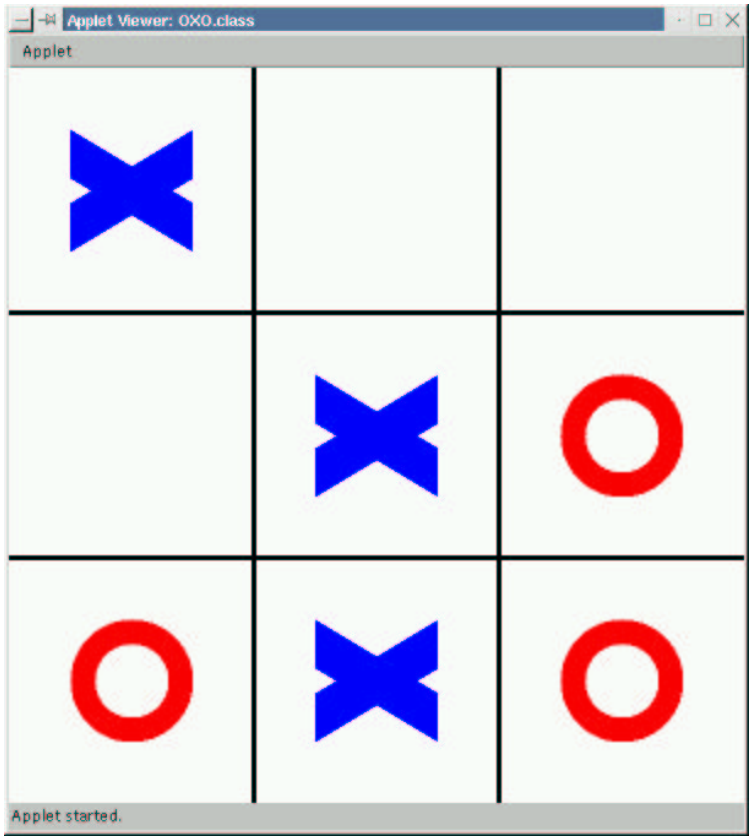
Machines

Murray Cole


Applet Viewer: OXO.class
Applet
Applet started.

Implementing Systems

Monitor, mouse, keyboard etc are electrical devices which produce simple effects in response to simple physical actions and electrical signals.

How do we arrange for the overall behaviour of some application to be implemented in terms of these simple actions and signals?

We need some kind of controller which notices input signals and generates appropriate output control.

We could design this as a Finite State Machine and implement it as a single purpose electronic circuit.

This is how almost all machines (whether electrical or mechanical) were implemented until 50-60 years ago.

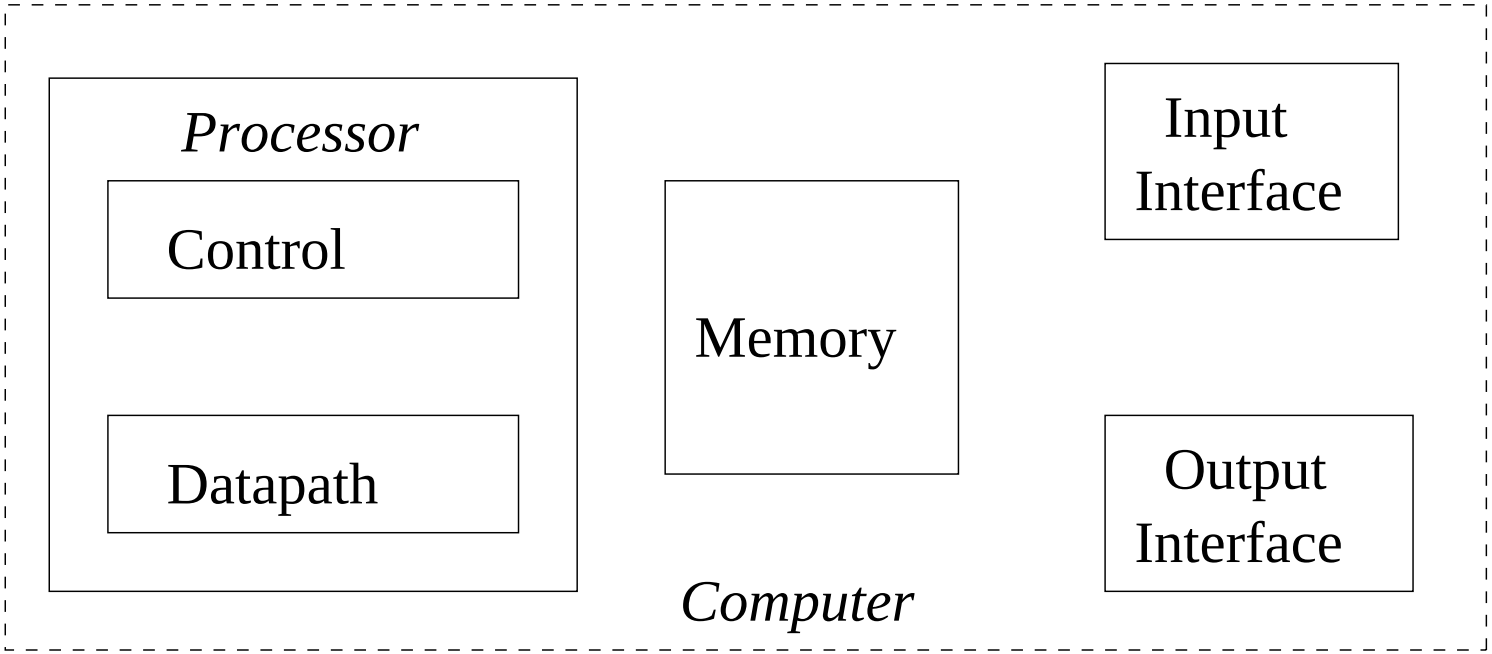
The von Neumann Machine

Rather than building a new machine for every task, why not build a single machine which can be given a description of a new task in some more convenient way?

Such as description is a program and the machine is said to be programmable and universal.

von Neumann sketched one of the earliest designs for such a machine and the concept of the stored program computer which it embodied.

Its structure remains at the core of computers to this day.

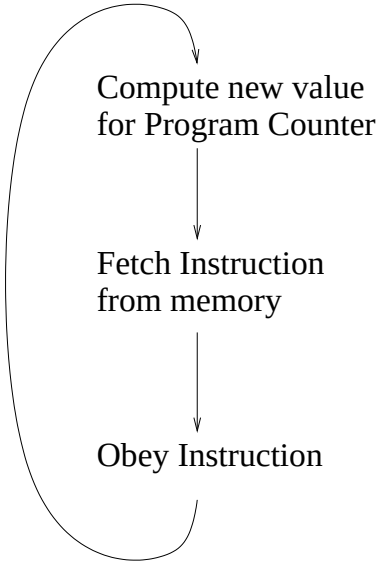


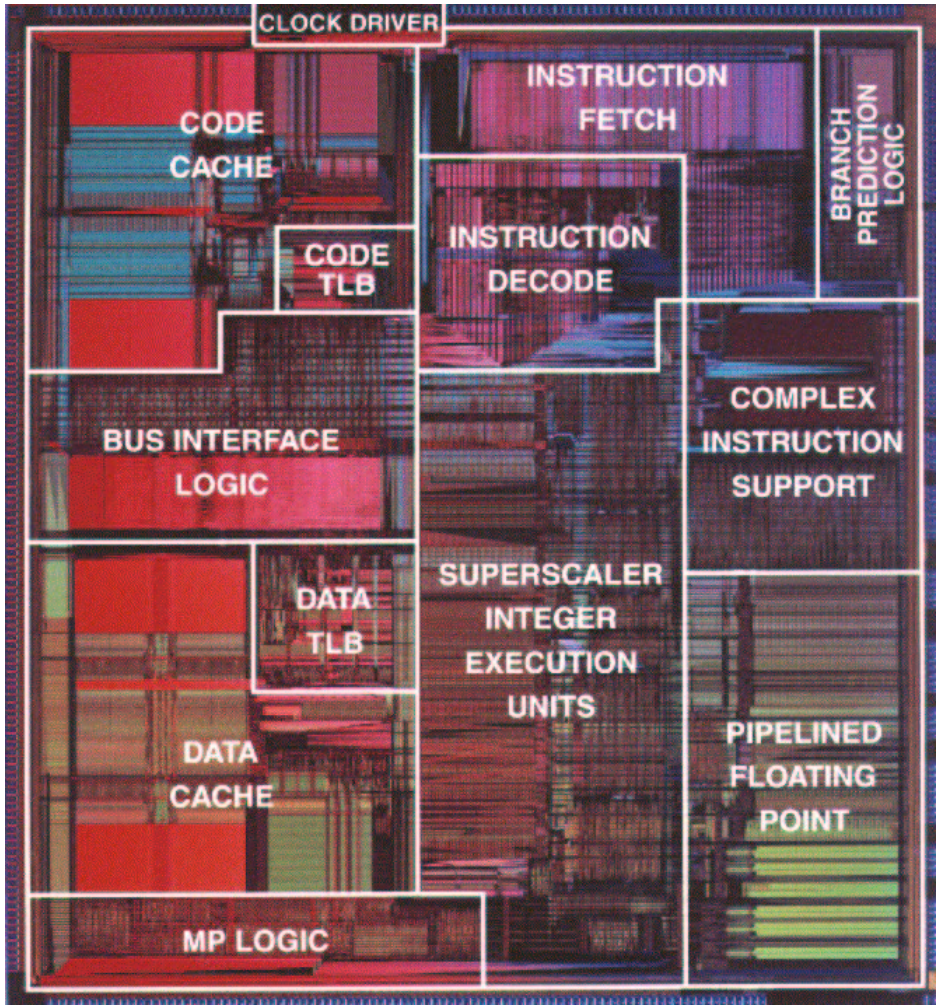
Memory stores data and program.

The Fetch-Execute Cycle

How does such a machine carry out the work of the program?

One step at a time!




CLOCK DRIVER
CODE CACHE
INSTRUCTION FETCH
BRANCH PREDICTION LOGIC
CODE TLB
INSTRUCTION DECODE
BUS INTERFACE LOGIC
COMPLEX INSTRUCTION SUPPORT
DATA TLB
SUPERSCALER INTEGER EXECUTION UNITS
DATA CACHE
PIPELINED FLOATING POINT
MP LOGIC

OK, Let's Make One!

We'll need

- a physical device which can store and retrieve programs and data (i.e. a memory)
- a physical device which can perform the fetch-execute cycle guided by a program in memory (i.e. a processor)

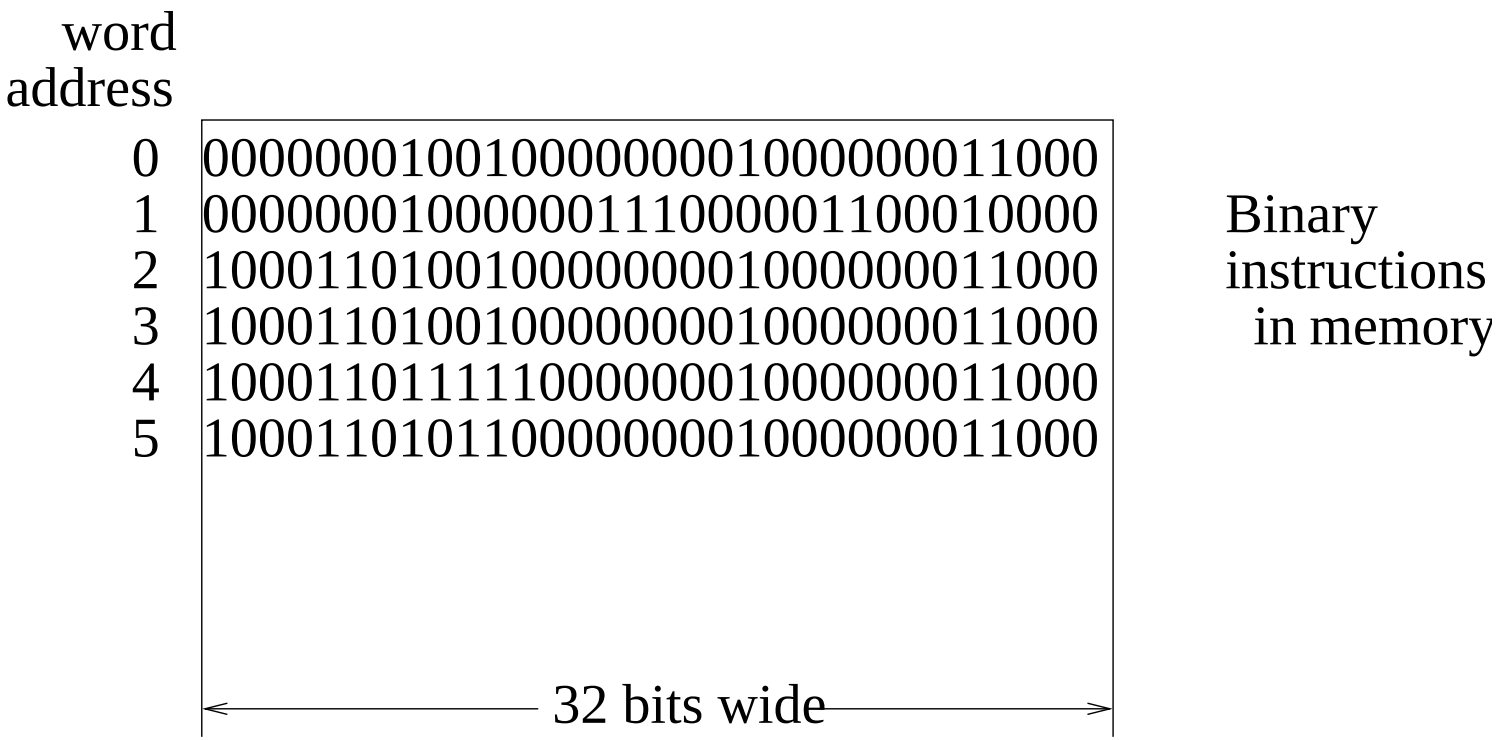
We'll use the best suitable (currently) available technology, digital electronics, which operates with binary (two value) data in very simple ways (but fortunately, very quickly and in huge quantities).

Memory

Digital electronic technology lets us build memory chips which can store, update and retrieve millions of bits (binary digits) cheaply and quickly.

It is useful (and cost effective) to collect bits together into larger groups (8, 16, 32, 64, 128), called words (or bytes for 8 bits).

A memory is effectively a huge array of words, which the machine can access (very quickly) by an index called an address.


word
address
0
1
2
3
4
5
00000001001000000010000000011000
00000001000000111000001100010000
10001101001000000010000000011000
10001101001000000010000000011000
10001101111100000010000000011000
10001101011000000010000000011000
32 bits wide
Binary
instructions
in memory

Designing a Processor

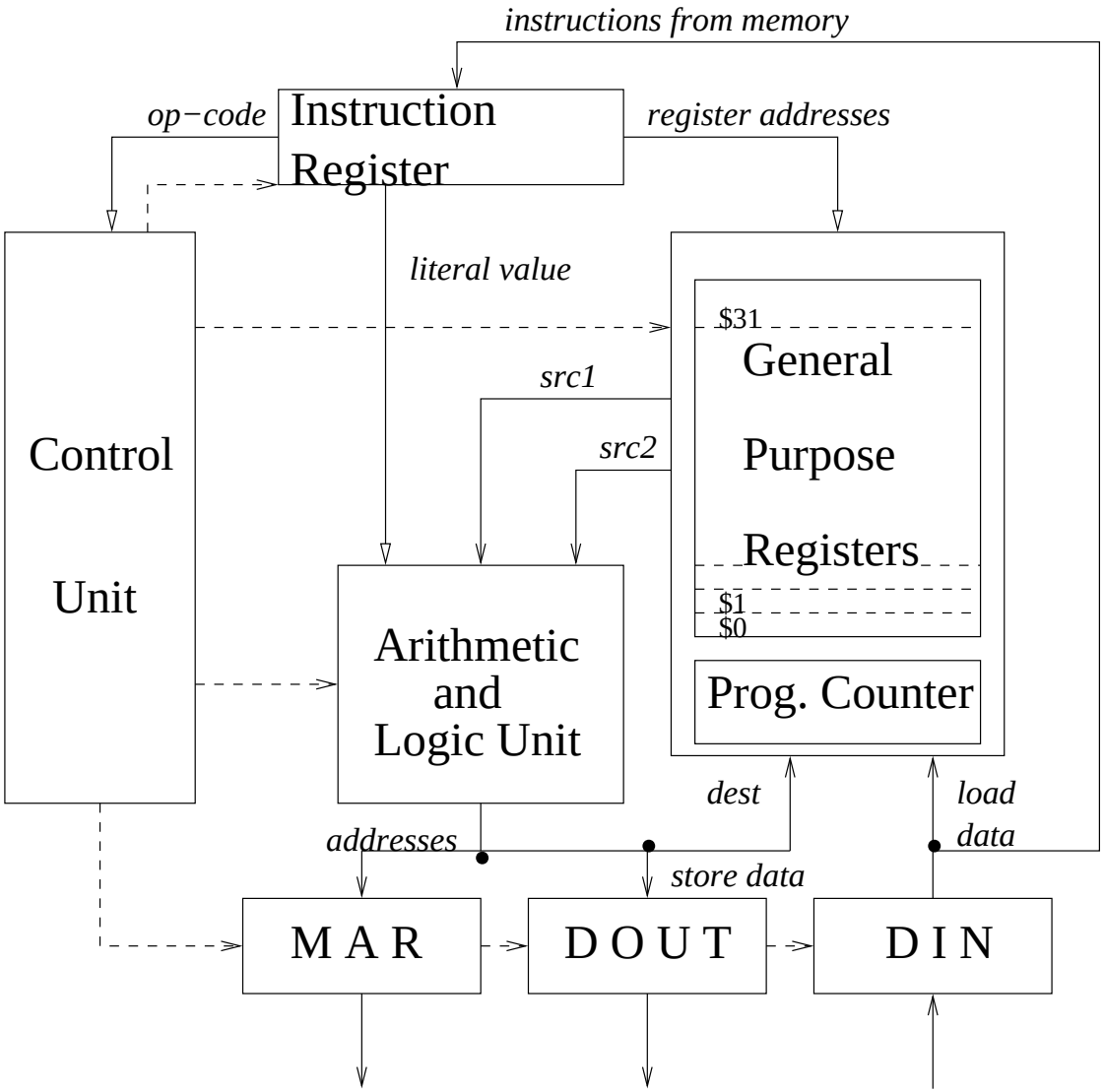
Need some way of keeping track of our progress through the program.

Need some circuits for manipulation of word values.

Need a mechanism for getting values to and from the memory and the other peripheral devices we want to control.

We will also find it useful to have some more memory in the processor itself, for the temporary (but even faster) storage of frequently accessed values from memory.

Crucially, we need a mechanism which can control all of the above, following the instructions in the program.


instructions from memory
op−code
Instruction Register
register addresses
literal value
$31
General Purpose Registers
$1
$0
src1
src2
Control Unit
Arithmetic and Logic Unit
Prog. Counter
dest
load data
addresses
store data
M A R
D O U T
D I N

Instruction Sets

The operations executable by a processor are defined by its instruction set. For example, for MIPS R2000

- `lw $6, 18808`, loads a word from memory into a register
- `sw $8, 16704`, stores a word to memory from a register
- `add $4, $6, $3` adds two registers, storing the result in a third
- `bgt $4, $6, 128732` conditionally jumps to another instruction

These are actually assembly language instructions, which are converted one-for-one to binary words for real execution.

```
for (i=9; i>=0; i--)
  x += a[i];
```

maps (roughly) to

```
      lw   $1, x
      li   $2, 9
      la   $3, a
test: bltz $2, end
      lw   $4, ($3)
      add  $1, $1, $4
      subi $2, 1
      addi $3, 4
      j    test
end:  sw   $1, x
```

which maps to a sequence of 10 32-bit words in memory

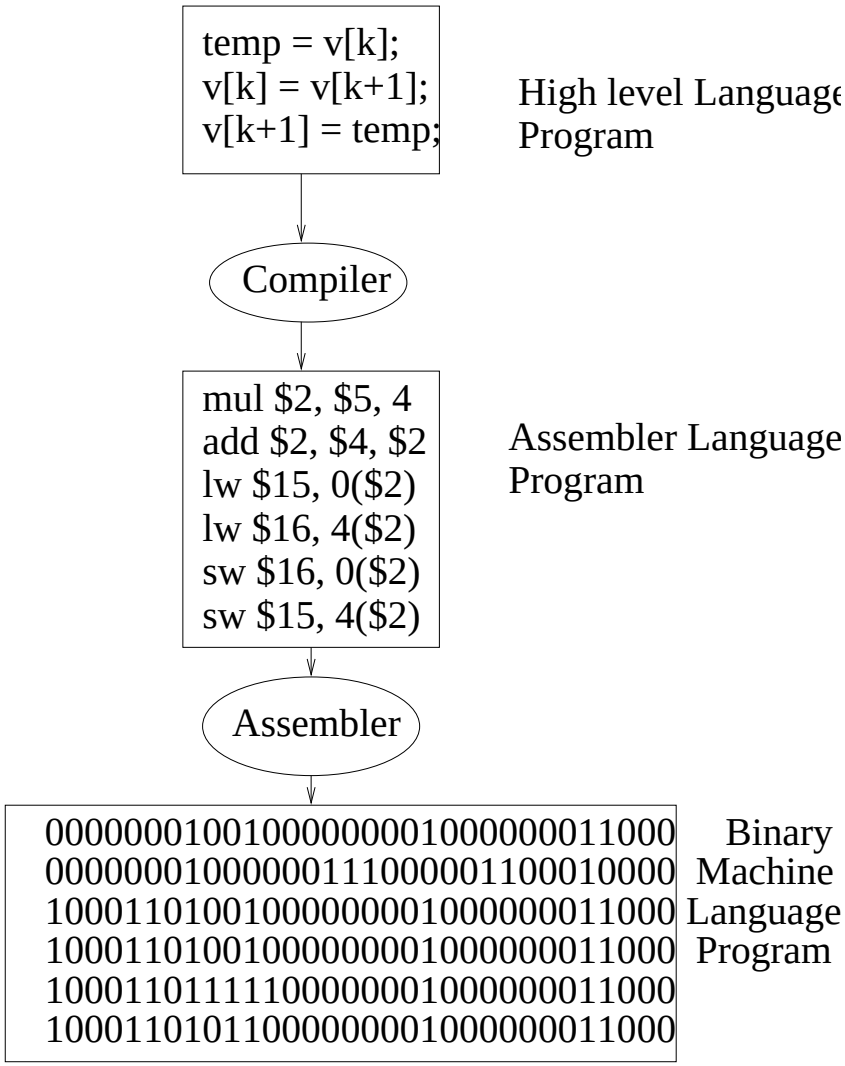
Language: Dreams and Reality

We would like our language to be understandable (to us) so that we can use it to describe tasks in the way we understand them. Natural language is too ambiguous and imprecise - our programming language should be as high level as possible.

BUT

as we have seen, the available technology works best with a very simple language (all 0's and 1's).

We will need to have several complex layers of translation from high to low level programs, but we can program our machine to help us with this task!


temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
High level Language Program
Compiler
mul $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
Assembler Language Program
Assembler
00000001001000000010000000011000
00000001000000111000001100010000
10001101001000000010000000011000
10001101001000000010000000011000
10001101111100000010000000011000
10001101011000000010000000011000
Binary Machine Language Program

Handling Other Devices

How do we handle our interactions with other devices?

They are mapped into the memory address space.

The hardware which interfaces the processor chip to the memory chips and device controller chips “knows” that certain memory addresses correspond to monitor, keyboard, hard drive etc, and routes accesses to them accordingly.

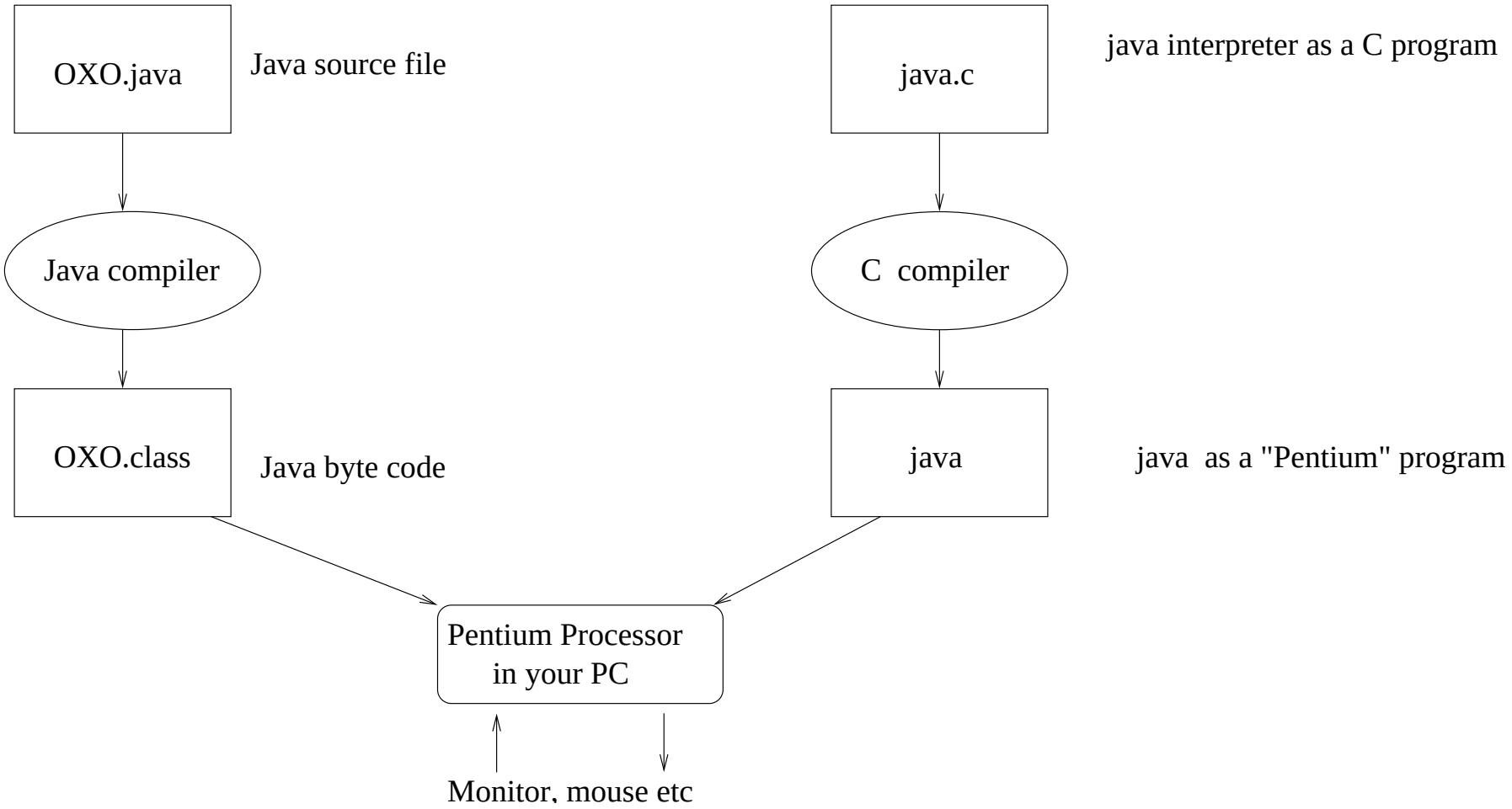
The processor reads from and writes to these addresses with ordinary loads and stores to achieve the required control.

Java and the JVM

How do Java, byte code and the JVM fit into this?

The program which runs when you type “java” is an interpreter, a program which mimics the execution of a byte code program on an imaginary machine, the Java Virtual Machine (JVM). The JVM interpreter itself is written in another language (often “C”) and is compiled to machine code for a real machine, in the way we have just seen.

To run your byte code, you actually run the interpreter program on the real machine, and this simulates the effect of running your byte code program on the JVM.


OXO.java
Java source file
java.c
java interpreter as a C program
Java compiler
C compiler
OXO.class
Java byte code
java
java as a "Pentium" program
Pentium Processor
in your PC
Monitor, mouse etc