Technical Computing for the Earth Sciences

EARS 80.03

Running Julia:

Locally, on your own computer:
(recommended)
There are a few steps. First you need the language binary, and then you need an editor.
(1) Download a Julia binary for your OS from https://julialang.org/downloads/

(2) Pick an editor. Options include:
a) VSCode, via Julia-vscode. Probably the default if you want a IDE-like experience, now that GitHub got bought by MS and stopped actively developing Atom. Follow instructions at: https://github.com/julia-vscode/julia-vscode#installing-juliavs-codevs-code-julia-extension
b) Atom, via "Juno". Follow directions at: http://docs.junolab.org/latest/man/installation/
c) A command-line editor, like Emacs or Vim. Potentially very efficient, but with steep learning curves. Directions at: https://github.com/JuliaEditorSupport/julia-emacs or https://github.com/JuliaEditorSupport/julia-vim
d) Editors that cost money, like Sublime Text or IntelliJ. Get plugins from https://github.com/JuliaEditorSupport/Julia-sublime or https://github.com/JuliaEditorSupport/julia-intellij

Remotely, on other people's computers:
(i.e., "the cloud" — don't have to install anything, but can be harder to get files in and out)

a) repl.it — Try starting at https://julialang.org/learning/tryjulia/ or https://repl.it/@logankilpatrick/TryJulia#main.jl. Will need to make an account if you want to save changes, etc.
b) Binder notebook: e.g. https://mybinder.org/v2/gh/binder-examples/demo-julia/master?filepath=demo.ipynb (from https://github.com/binder-examples/demo-julia). Works pretty well if you like Jupyter notebooks, but times out if left inactive for more than 15 or 20 minutes. Great if you want to demonstrate your Julia code to other people.
c) Google Colab notebook: e.g. https://colab.research.google.com/github/ageron/julia_notebooks/blob/master/Julia_Colab_Notebook_Template.ipynb. Slightly inelegant (have to install Julia each time), but gives you access to free GPUs and TPUs (for now).

Julia basics

REPL* MODES:

Normal: julia> type ? → help mode] → pkg mode ; → shell mode (e.g. BASH, etc.)

COMMON OPERATORS

exponent, modulus, OR, pipe

`+ , - , * , / , ÷ , ^ , % , & , | , > , < , >= , <= , == , |>`

Integer division (\div), AND, isequal

`x |> sin |> cos` is same as `cos(sin(x))`

SOME IMPORTANT TYPES

Numeric Types

Float64, Int64, UInt64

Int32, 16, (8) etc.*

*These are "primitive types". Can make more, but probably shouldn't

Arrays

```
foo = [1, 2, 3, 4, 5]
```

Array{Float64}(undef, 100, 10)

Text

single quotes

'a', 'b' - Characters

"hello!" - String (like an array of characters)

Double quotes

Tuples

```
a = (1,2,3,4,5)
```

Like arrays, but immutable

Composite Types

```
struct Foo
    a::Float64
    b::Int32
    c::Array{Float64,1}
end
```

Step ranges

```
a = 1:5
b = 1:0.01:10
c = 5:-1:1
```

- Turn into an array with `collect(a)`
- Actually just composite types w/ methods for iteration

FUNCTIONS

```
function foo(a,b)
    c = 3*a + 7*b + 9
    return sin(c)
end
```

- Julia functions are built around multiple dispatch
- Dispatch favors function w/ most specific type signature
- All the operators above are just functions that can be extended like any other
- Can also write one-liners
  `foo(x) = sin(x)`
  and anonymous functions
  `x -> 2x`

OTHER SYNTAX

Type annotations

```
a::Float64
```

Use in function signatures (for dispatch)

& occasionally elsewhere (to avoid type-instability)

"Dot-broadcasting"

```
A .+ 5 .* sin.(C)
```

Applies the function/operator element-wise

Can also write as

```
@. A + 5 * sin(C)
```

*"Read-Eval-Print Loop"

Julia basics

SOME USEFUL FUNCTIONS & MACROS*

- `typeof()`, `sizeof()`, `size()`, `bitstring()` - May help figure out what something is
- `@time`, `@allocated`, `@code_native`, `@code_warntype`, `@btime` - For measuring performance
- `print()`, `println()`, `display()` - Show output in REPL

CONTROL FLOW

```
for i=1:100
    println(i)
end
```

```
i=0
while i<100
    i=i+2
    println(i)
end
```

```
if i<0
    b=sqrt(-i)
else
    b=sqrt(i)
end
```

- Watch out for global scope

GENERAL LANGUAGE STUFF

- Julia is "Just-in-time" compiled. This puts it somewhere in between

Compiled languages like Fortran, C, C++, Rust & Interpreted languages like Matlab, Python, Perl

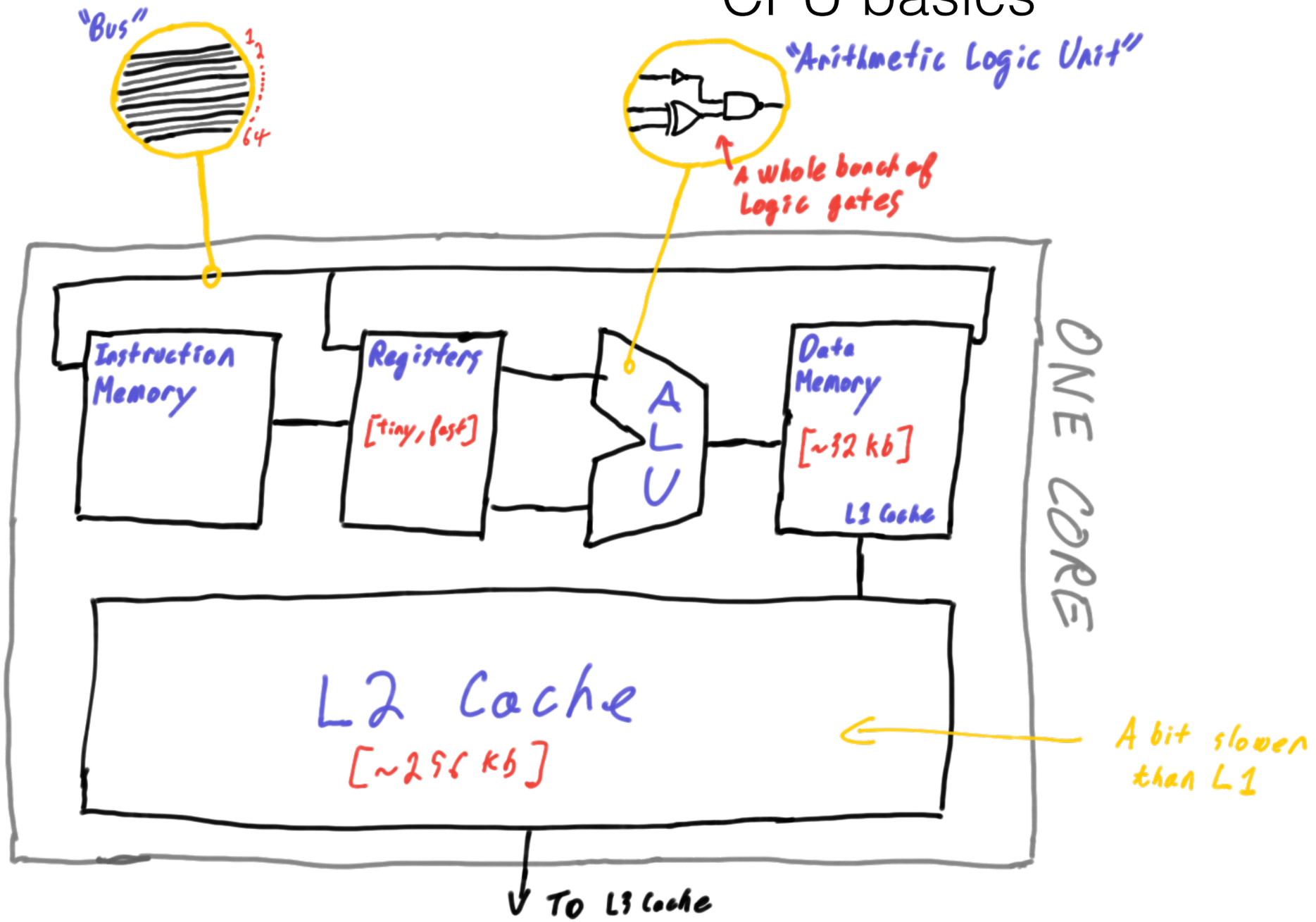
It's not the only JIT-compiled language†, but is the only one where JIT compilation is used to allow efficient multiple dispatch

} Probably actually missing a bunch of steps
Only need to know if you want to do something really unconventional

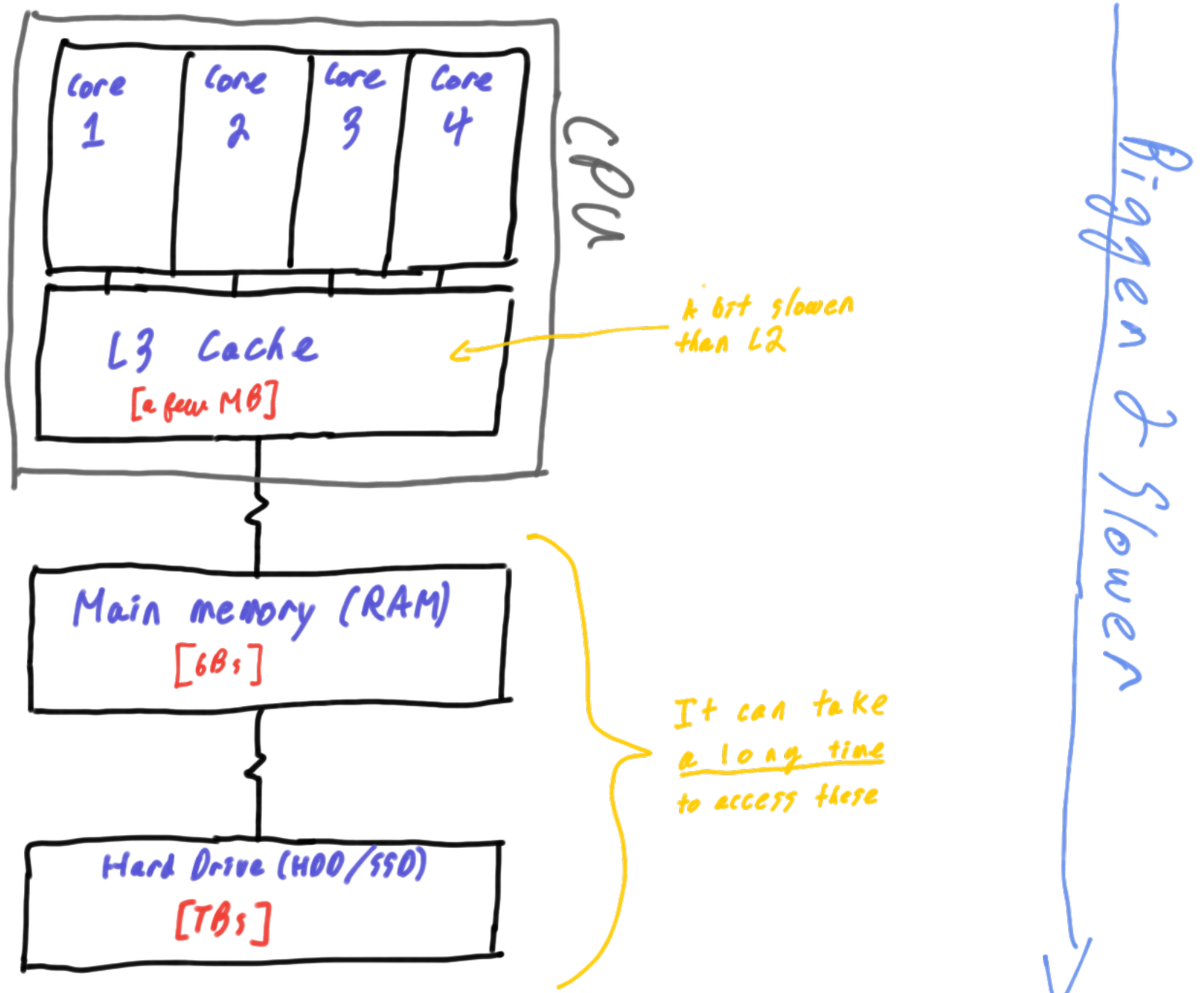
† JavaScript is often JIT compiled, because so much of the web depends on it. People have tried to JIT python, but the design of the language makes it hard. Other languages are JITted to "bytecode" that runs on a VM (ex. Java) but not properly compiled to machine (assembly) code.

*Anything beginning with `@` is a macro, & that's all we're going to say about that for now

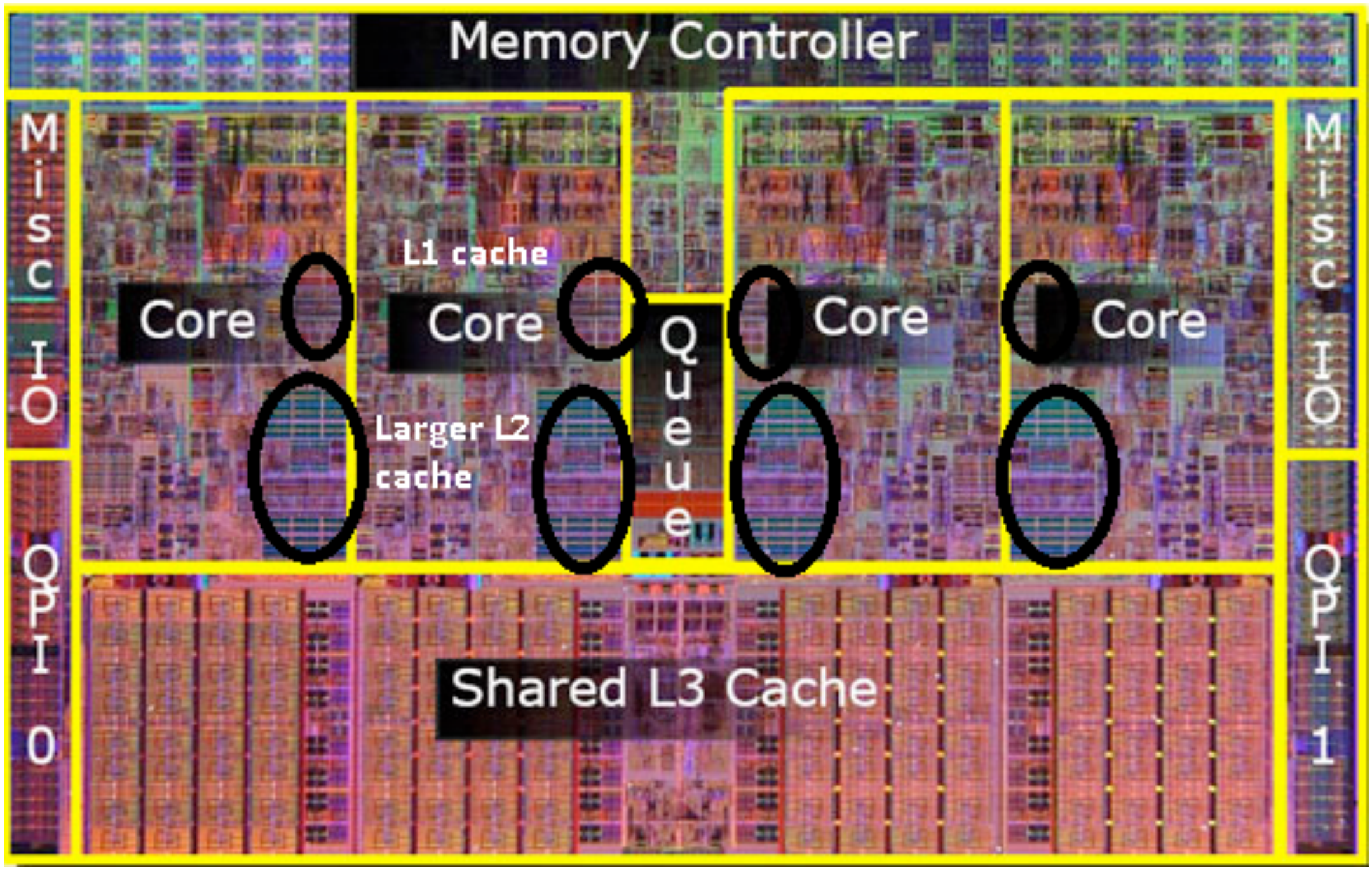
CPU basics


"Bus"
1
2
64
"Arithmetic Logic Unit"
A whole bunch of Logic gates
Instruction Memory
Registers
[tiny, fast]
ALU
Data Memory
[~32 kb]
L1 Cache
ONE CORE
L2 Cache
[~256 kb]
A bit slower than L1
TO L3 Cache

CPU basics


Core 1
Core 2
Core 3
Core 4
CPU
L3 Cache
[a few MB]
A bit slower than L2
Main memory (RAM)
[GBs]
Hard Drive (HDD/SSD)
[TBs]
It can take a long time to access these
Bigger & Slower

CPU basics


Memory Controller
Misc IO
Misc IO
L1 cache
Core
Core
Core
Core
Larger L2 cache
Queue
QPI 0
QPI 1
Shared L3 Cache

CPU basics

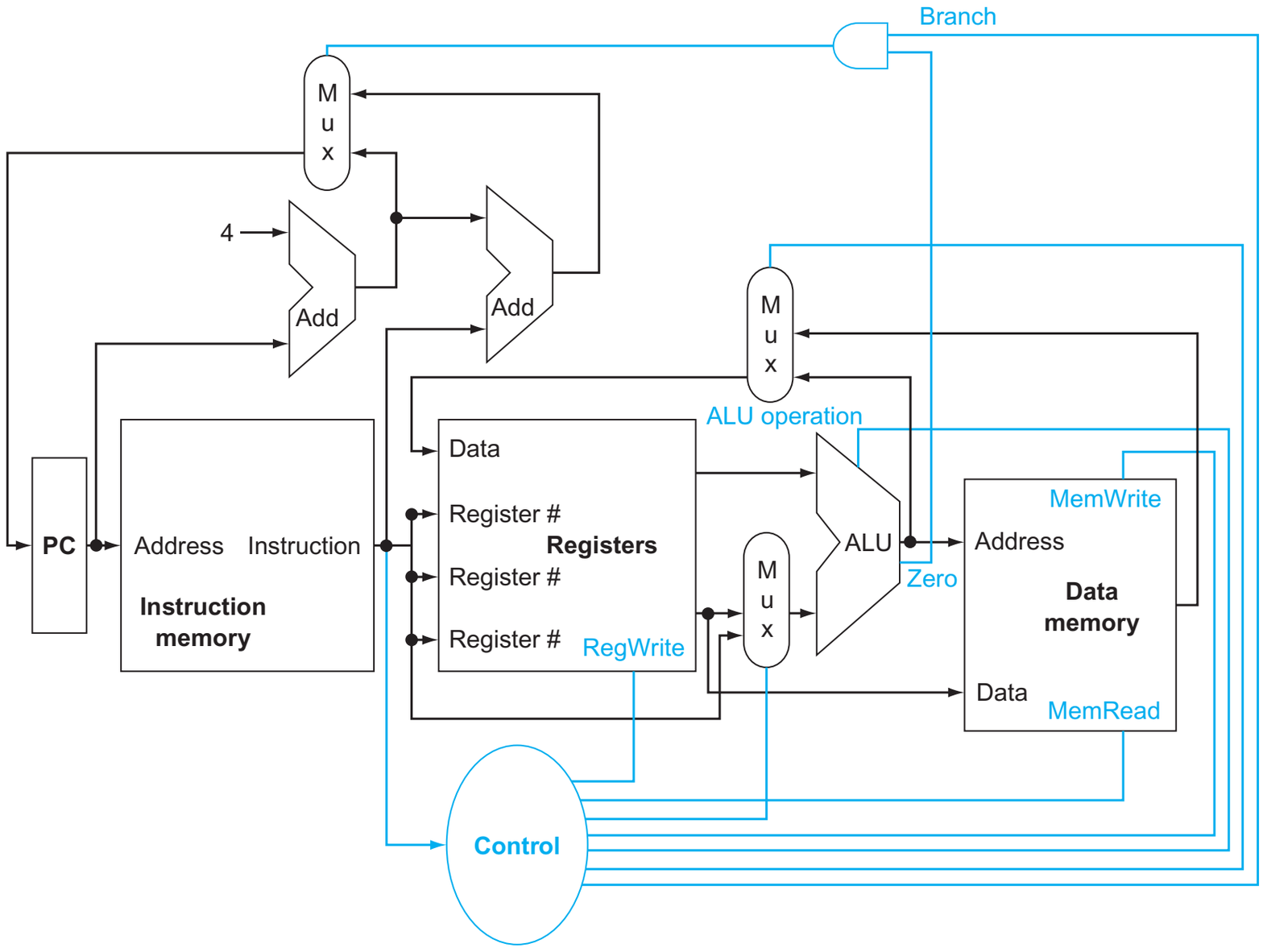
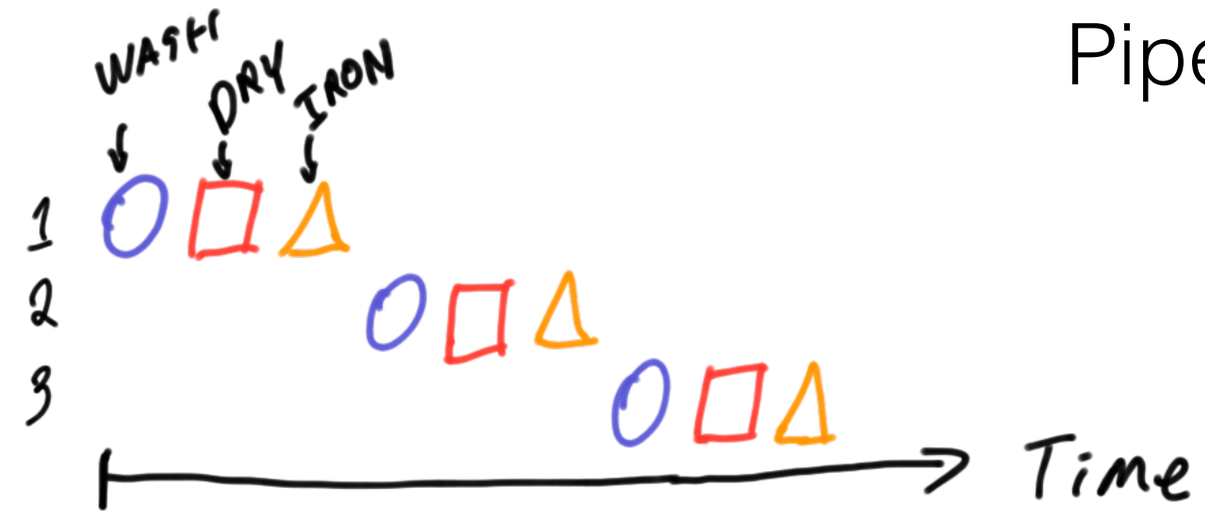


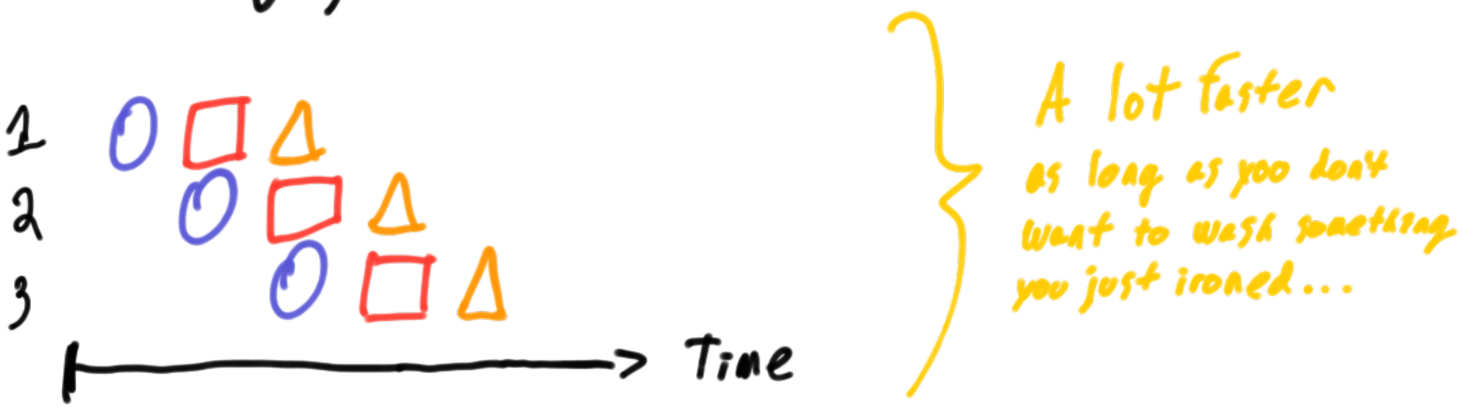
FIGURE 4.2 The basic implementation of the MIPS subset, including the necessary multiplexors and control
The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is con
by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The r
multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instructi
the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to dete
whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instr
(for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data m
should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them ea
see.

Patterson & Hennessy, *Computer Organization and Design*, 5th ed.

Pipelining



VS



Pipelining

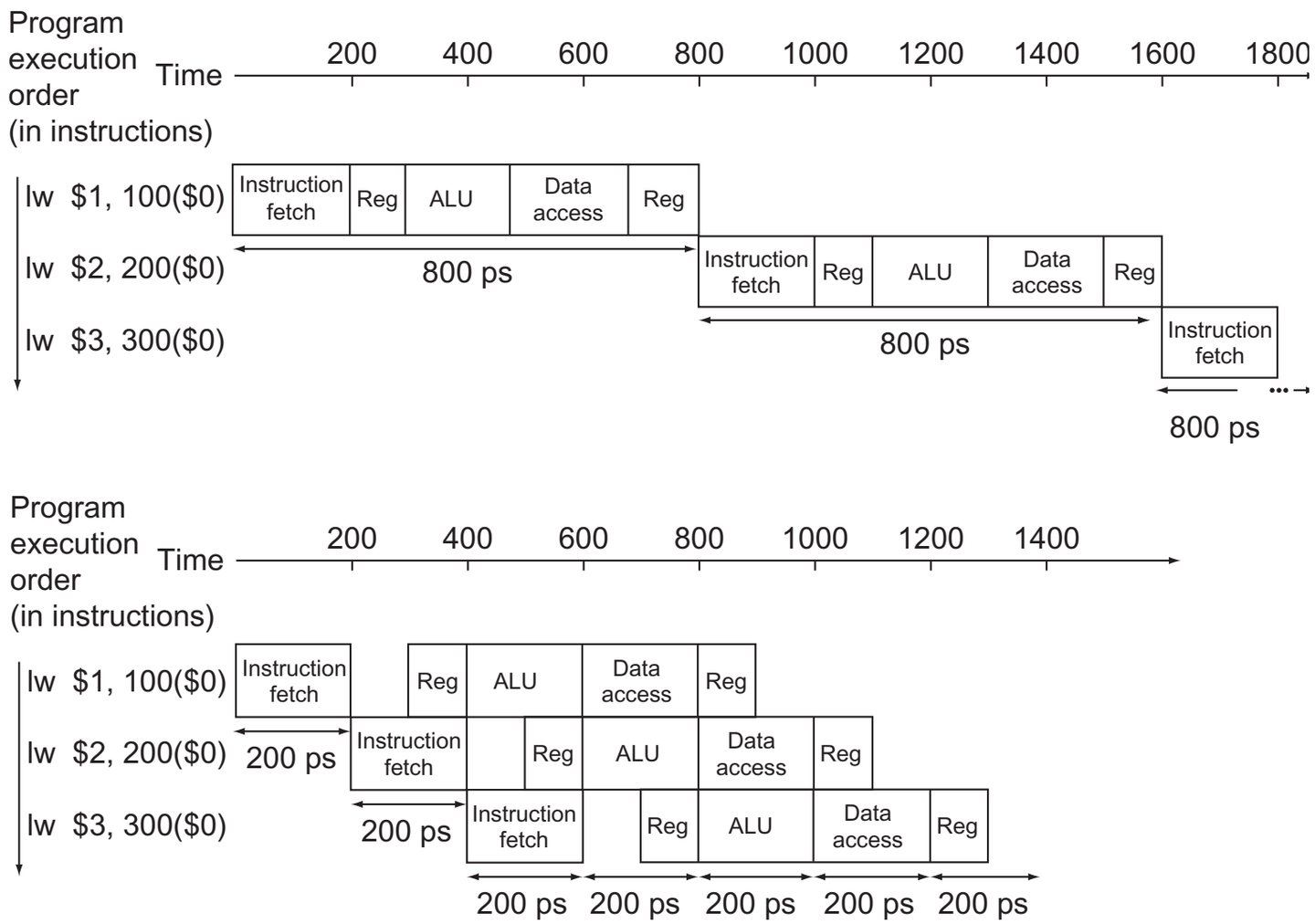


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource: either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Latency

"Latency numbers every programmer should know"

one CPU clock cycle
(so must be for a 3GHz CPU)

L1 cache reference	0.5 ns		
Branch mispredict	5 ns		
L2 cache reference	7 ns		14x L1 cache
Main memory reference	100 ns		200x L1 cache
Read 1 MB sequentially from memory	250,000 ns	250 us	
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us	1 ms
Disk seek	10,000,000 ns	10,000 us	10 ms
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us	20 ms

So tl;dr: RAM is 200 clock cycles away
HD is 20,000,000 clock cycles away