Concurrency and Parallelism

Roadmap

- In the following sections,
- we'll first turn our attention to implicit parallelism, and how best to optimize our software to take advantage of it.
- Next, we'll review the most common forms of explicit parallelism. Then we'll explore the various concurrent programming techniques used to harness explicitly parallel computing platforms.
- Finally, we'll round out the discussion of parallel programming by discussing SIMD vector processing, and how it applies to GPU design and general-purpose GPU programming (GPGPU) techniques.

The Advent of Parallelism and Concurrency

- 1970s
  - Intel 8087 FP coprocessor: 50 kFLOPS (5.0×10^4)
  - Cray-1 supercomputer: 160 MFLOPS (1.6×10^8)
- Today
  - AMD Jaguar x86-64 8-core: 100 GFLOPS (1.0×10^{11})
  - China's Sunway TaihuLight: 93 PFLOPS (9.3×10^{16})
  - 7 to 8 orders of magnitude improvement!
  - Currently, **Summit is the fastest supercomputer**
    - **Speed:** 148.6 petaFLOPS
      Cores: 2,414,592
    - **Vendor:** IBM
      Location: Oak Ridge National Laboratory, United States

Overview

- Today's computer hardware is increasingly **parallel**
- Need to understand **concurrency** to take advantage of parallelism
- Need to understand the **hardware** and the **kernel** to understand concurrent programming!
- In this talk we'll cover:
  - modern **parallel** computing **hardware**,
  - **concurrent programming** techniques, and
  - how to apply this to building a **high-performance game engine**

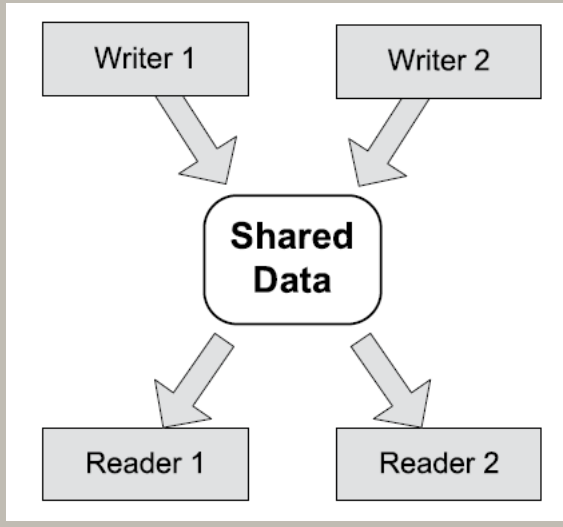
What is Concurrency?

- What is concurrency?
- What is parallelism?
- How are they related? How do they differ?

Discussion: 5 mins

What is Concurrency?

- Concurrency can be defined as the **transformation** of a ***shared* data file**
  - Shared by multiple **flows of control**
    - processes,
    - threads,
    - fibers…
  - Multiple **readers** and/or **writers**
  - … of ***shared*** data!
- Did I mention the data has to be **shared**?
- *race condition (data race)*
- two or more flows of control "compete" to be the first to read, modify and write a chunk of shared data

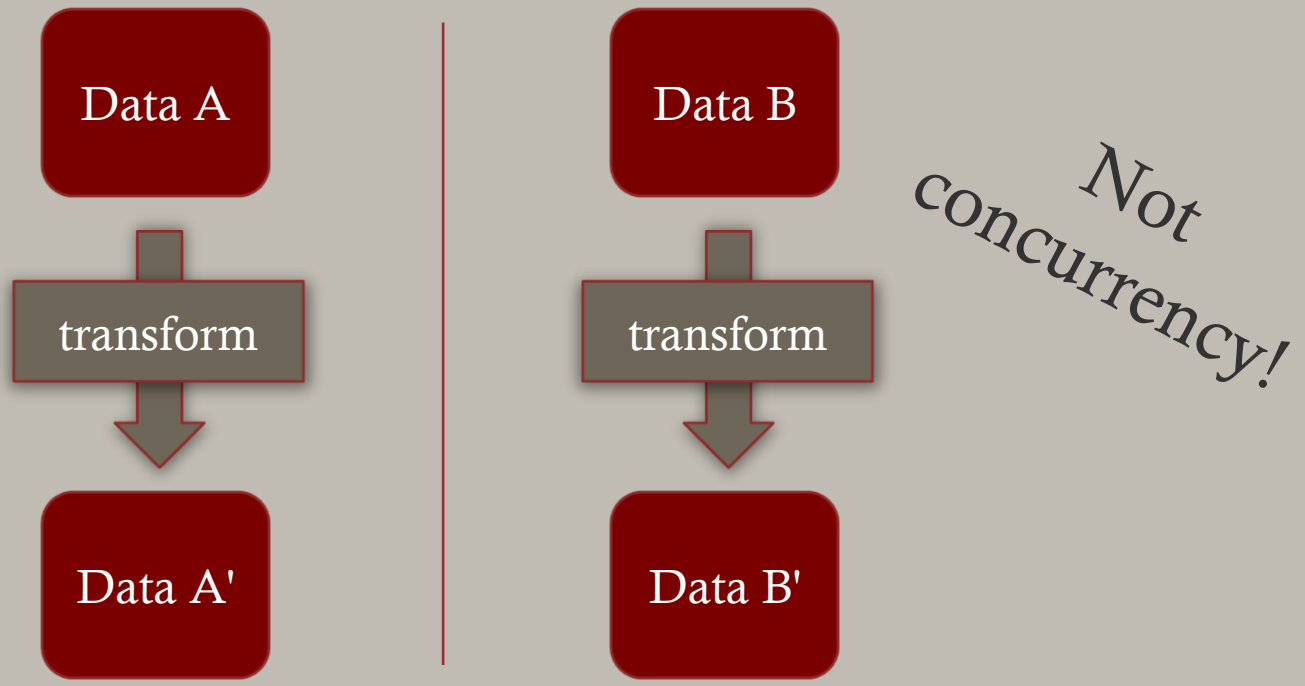


What is Concurrency?

- Data "file" can **be** anything
  - A **global Boolean variable** shared between threads
  - A shared **queue**
- Data file can **live** anywhere
  - Virtual memory space within a **multithreaded process**
  - Multiple processes **sharing virtual memory pages** within a single computer
  - GPU and CPU(s) **sharing physical RAM**
    - e.g., PS4, Xbox One
  - A **pipe** between two processes (`cat foo.txt | grep "bar"`)
  - A file living on a **network drive** that's accessible by multiple computers

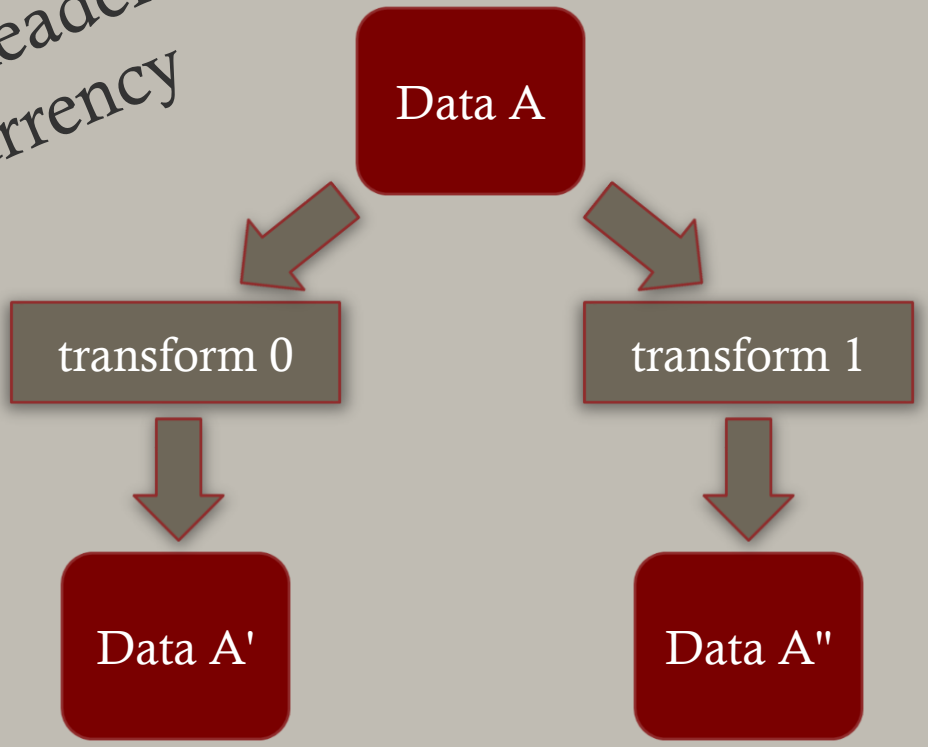
What is Concurrency?

- If data **isn't being shared**, it **isn't concurrency**
  - It's just computing "at the same time"

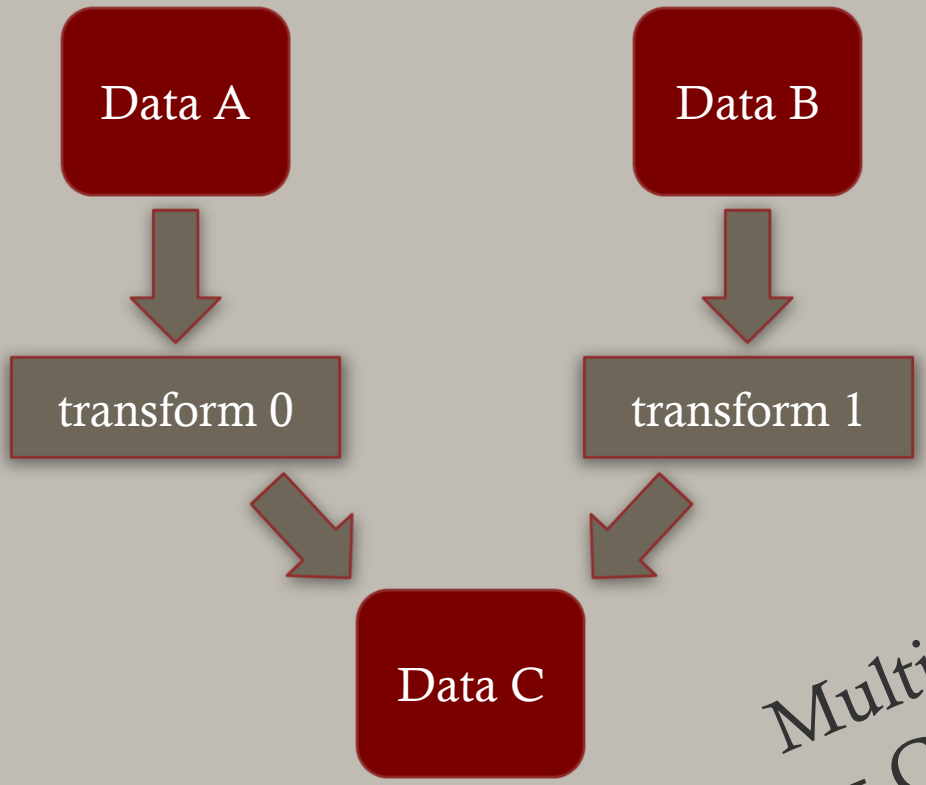


What is Concurrency?

Multiple Readers
= Concurrency

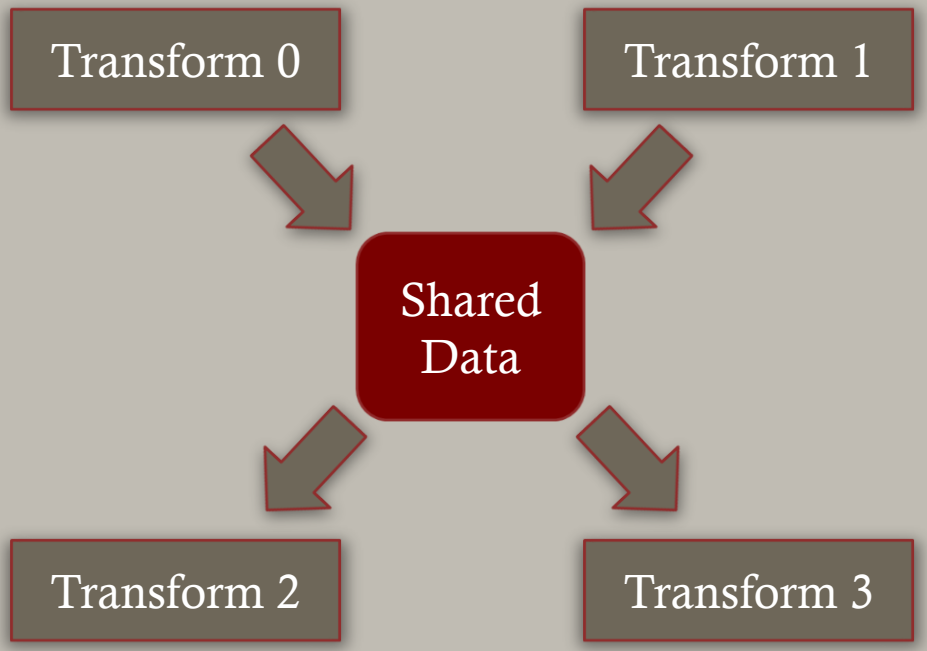


What is Concurrency?



Multiple Writers
= Concurrency

Multiple Writers
= Concurrency



Multiple Readers
= Concurrency

Concurrency versus Parallelism

- **Concurrency** means multiple flows of control operating on shared data
- **Parallelism** is multiple hardware components operating simultaneously
  - *Concurrency* is possible even on a **single-core** CPU
    - e.g., **preemptive** multitasking
  - Likewise, *parallel hardware* can improve performance of even **single-threaded** code
    - implict parallelism (**pipelined** or **superscalar** CPU architectures)

The Advent of Parallelism and Concurrency

- In 1965, Intel co-founder Gordon Moore noticed:
  - computing power, measured in **transistors per unit area** of silicon, had been roughly **doubling** every year since the invention of the digital computer
- Moore's "Law" states that:
  - At a given price point…
  - amount of computing power **doubles** roughly every **18 months**
- Reality is a bit more complex…

History of Processor Performance

Source: https://booksite.elsevier.com/9780123838728/figures/PDF/Chapter_01.pdf

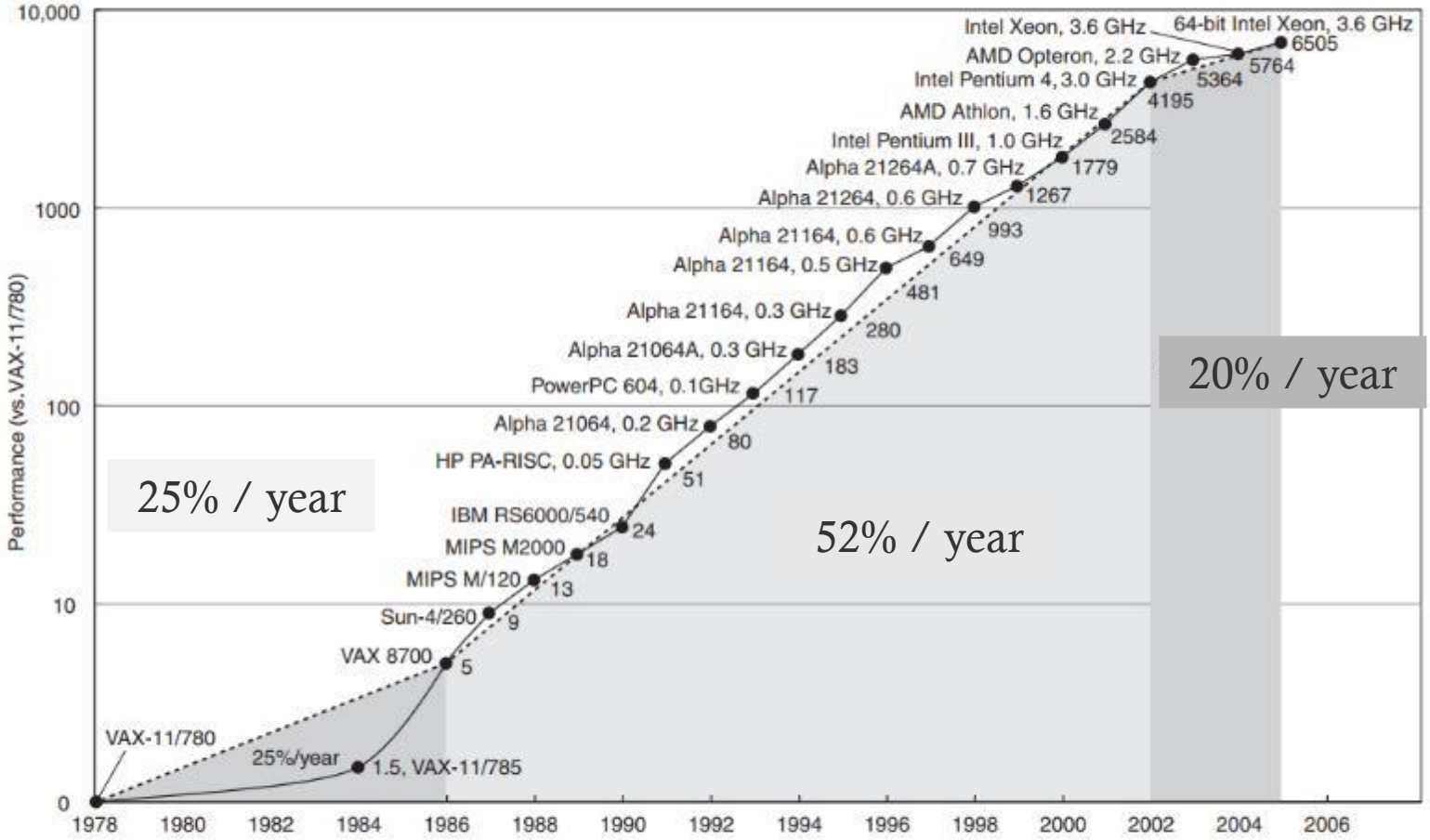


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

History of Processor Performance

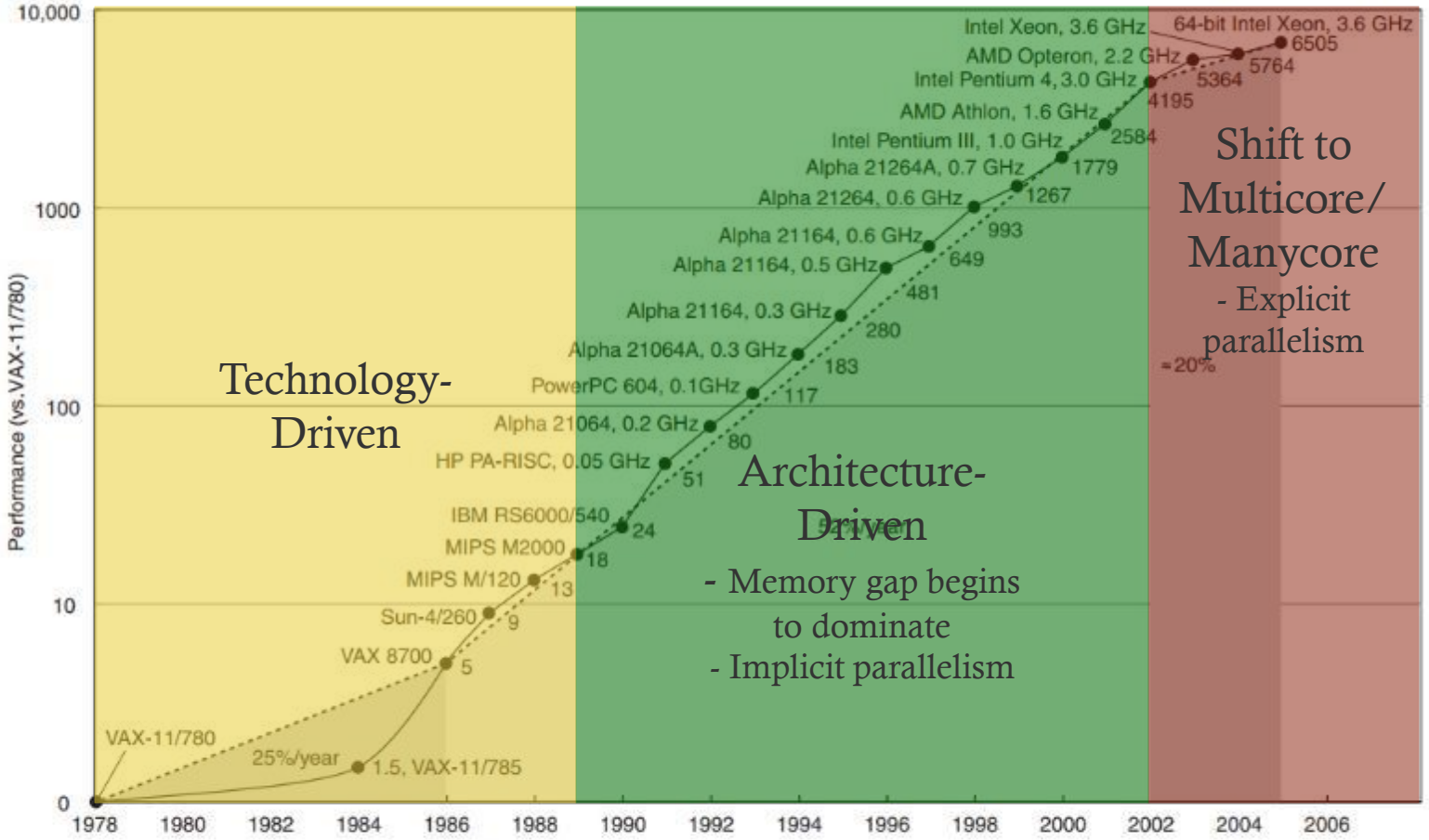
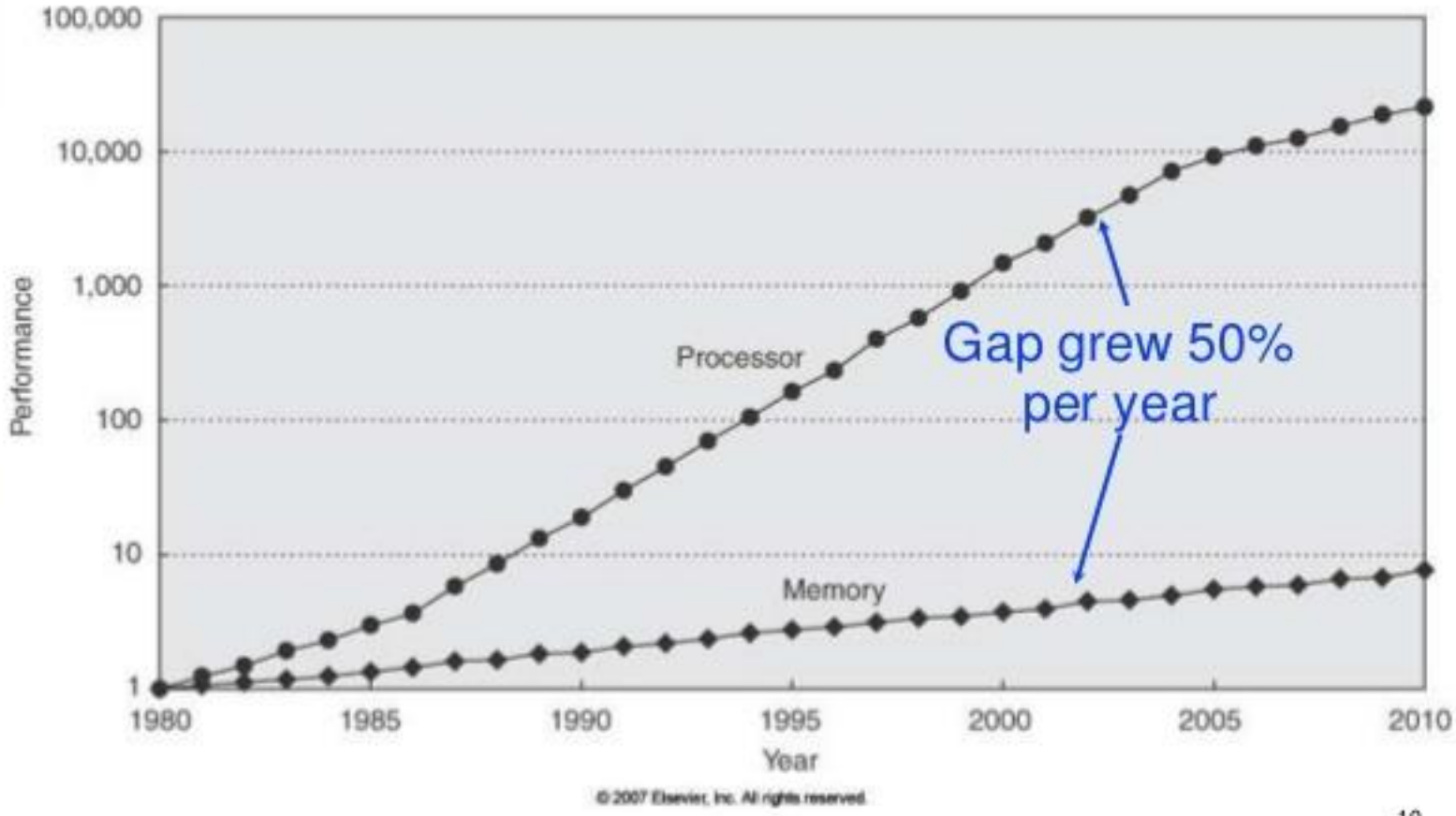


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

Processor Memory Gap



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

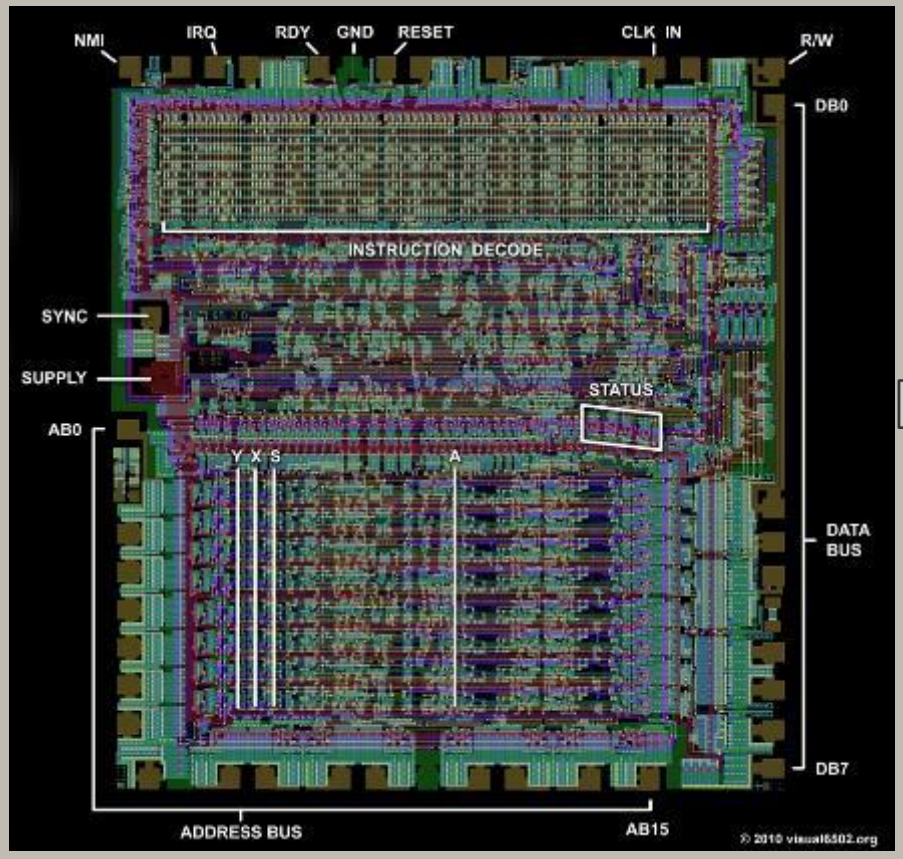
The Advent of Parallelism and Concurrency

- Increases in **clock speed** led to:
  - increased heat generation
  - increased power consumption
  - decreased signal-to-noise ratio
- Increase in **memory gap** led to:
  - Less bang for buck when improving single CPU core
- Increases in **transistor count** led to question:
  - How best to use all those transistors?

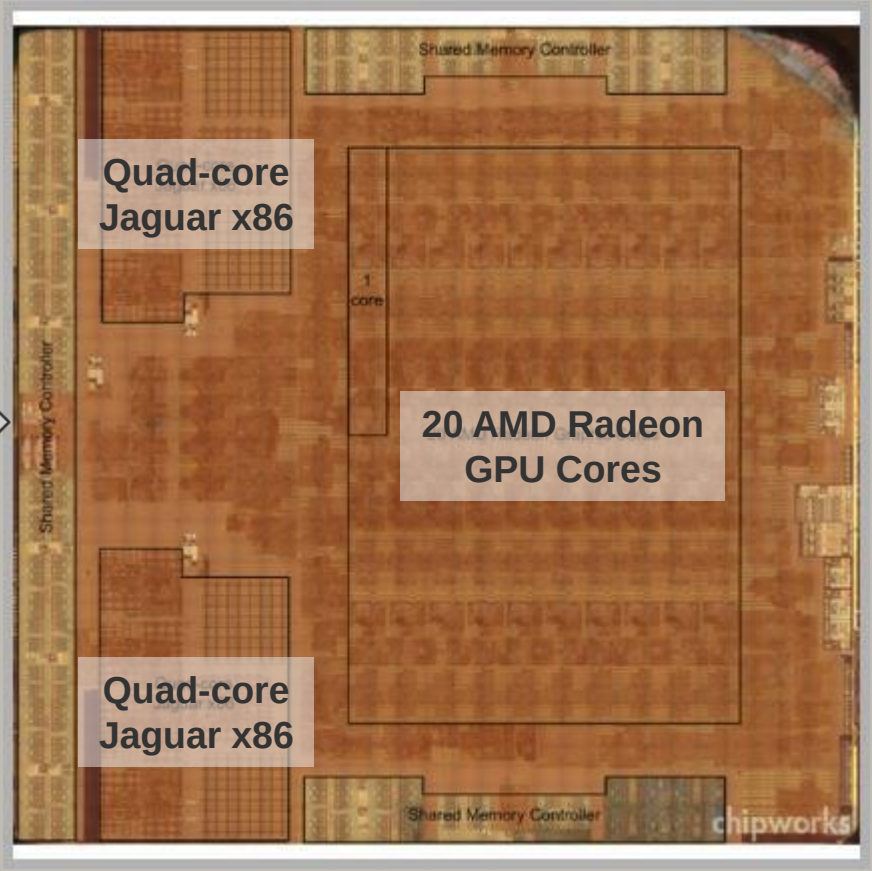
The Advent of Parallelism and Concurrency

- Result: Increased use of **parallelism** in consumer electronics

Single-core CMOS Tech 6502 (Apple II)



8-core AMD Jaguar + Radeon HD 7870 GPU (PS4)



The Advent of Parallelism and Concurrency

- Principles of parallelism known and in use since the **1960s**
- But back then, parallelism was reserved for high-end **supercomputers**
  - 1964: Seymour Cray pioneered **pipelining** with CDC 6600
  - 1972: CDC 8600 pioneered **multiprocessor** design by strapping four CDC 7600s together, executing in lock step
  - 1975: Cray-1 pioneered **vector processing** (SIMD)

Implicit Parallelism

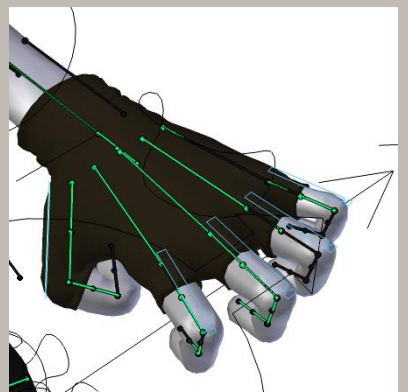
- **Implicit parallelism** is the use of parallelism to improve performance of **preexisting (sequential)** code
  - Little or no programmer/compiler intervention required
    - *although **knowledge** of implicit parallelism **aids optimization**!*
- Used in consumer electronics since ~1989
  - **Pipelining**
  - **Superscalar**
  - **VLIW**
  - e.g., pipelined CPU can run existing code faster than a non-pipelined CPU that supports the same ISA

Explicit Parallelism

- **Explicit parallelism** exposes the presence of parallel computing hardware to the:
  - programmer and/or
  - compiler
- Used in consumer electronics since ~2002
  - **Multiprocessor** computers
  - **Multicore** CPUs
  - SIMD **vector processing unit** in x86 (SSE)
  - **Manycore** GPUs (SIMT)
  - computer clusters
  - grid computing
  - cloud computing
- Explicit parallelism supports **concurrency**

Task versus Data Parallelism

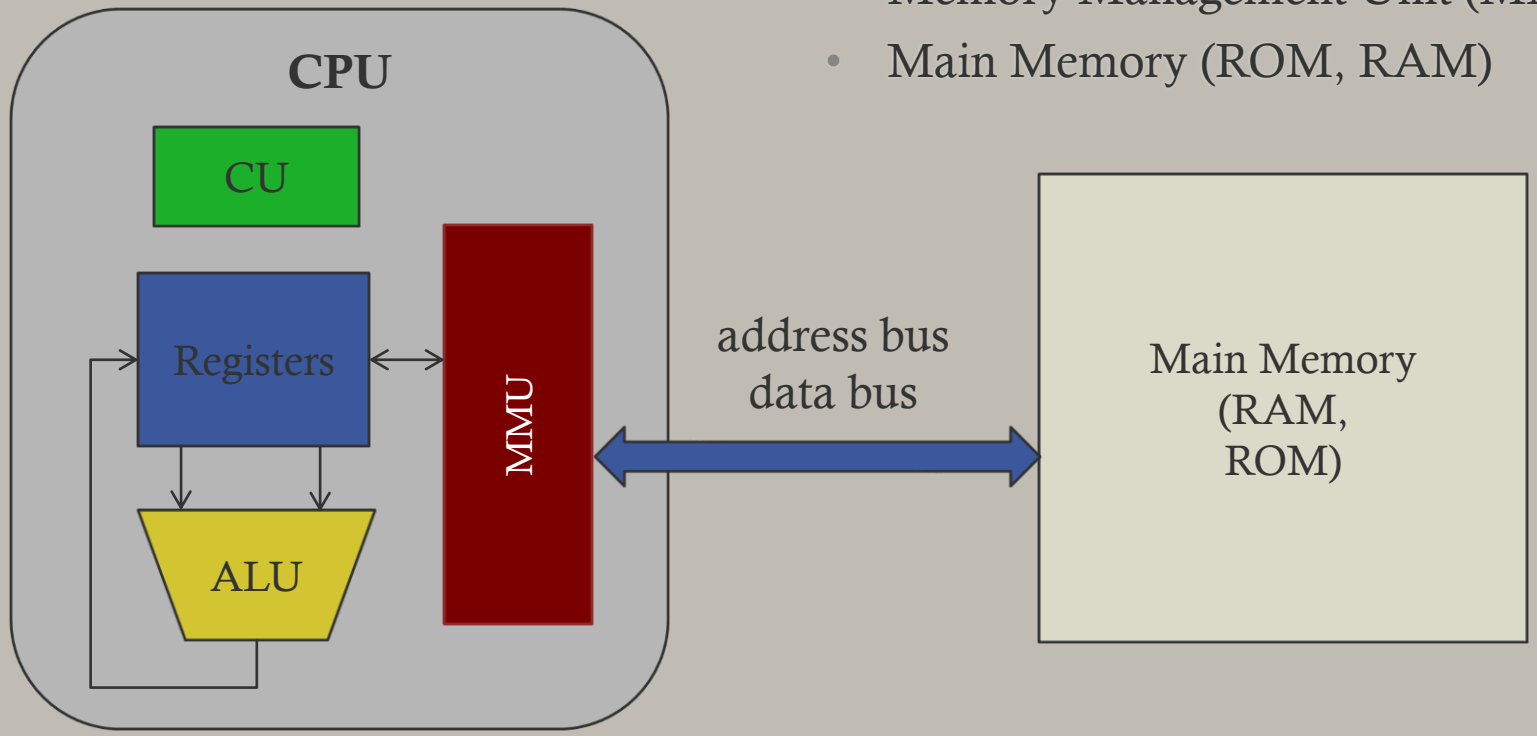
- *Task parallelism:* When multiple heterogeneous operations are performed in parallel.
  - Example: Performing animation calculations on one core while performing collision checks on another.
- *Data parallelism:* When a *single* operation is performed on multiple data items in parallel.
  - Example: Calculating 1000 skinning matrices by running 250 matrix calculations on each of four cores.



A Simple Computer

- Simplest von Neumann architecture

- Control Unit (CU)
- Arithmetic/Logic Unit (ALU)
- Registers
- Memory Management Unit (MMU)
- Main Memory (ROM, RAM)

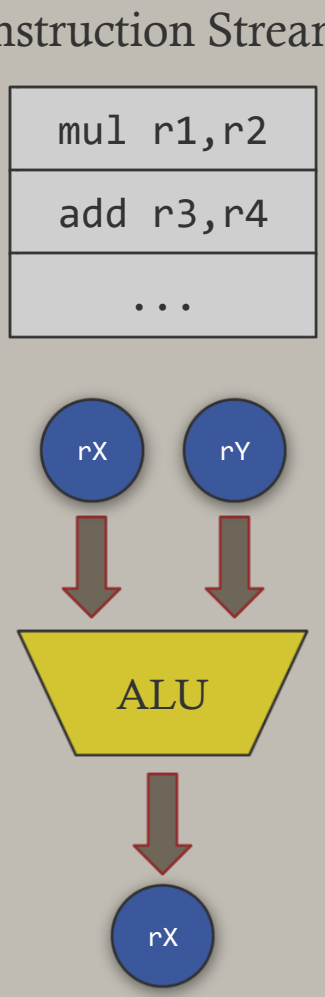


Flynn's Taxonomy

- Proposed by Michael J. Flynn of Stanford University in 1966
- Breaks parallelism into **instruction-level parallelism** (ILP) and **data-level parallelism** (DLP)
  - Single instruction, single data (SISD)
  - Multiple instruction, multiple data (MIMD)
  - Single instruction, multiple data (SIMD)
  - Multiple instruction, single data (MISD)
  - Single instruction, multiple thread (SIMT): GPUs

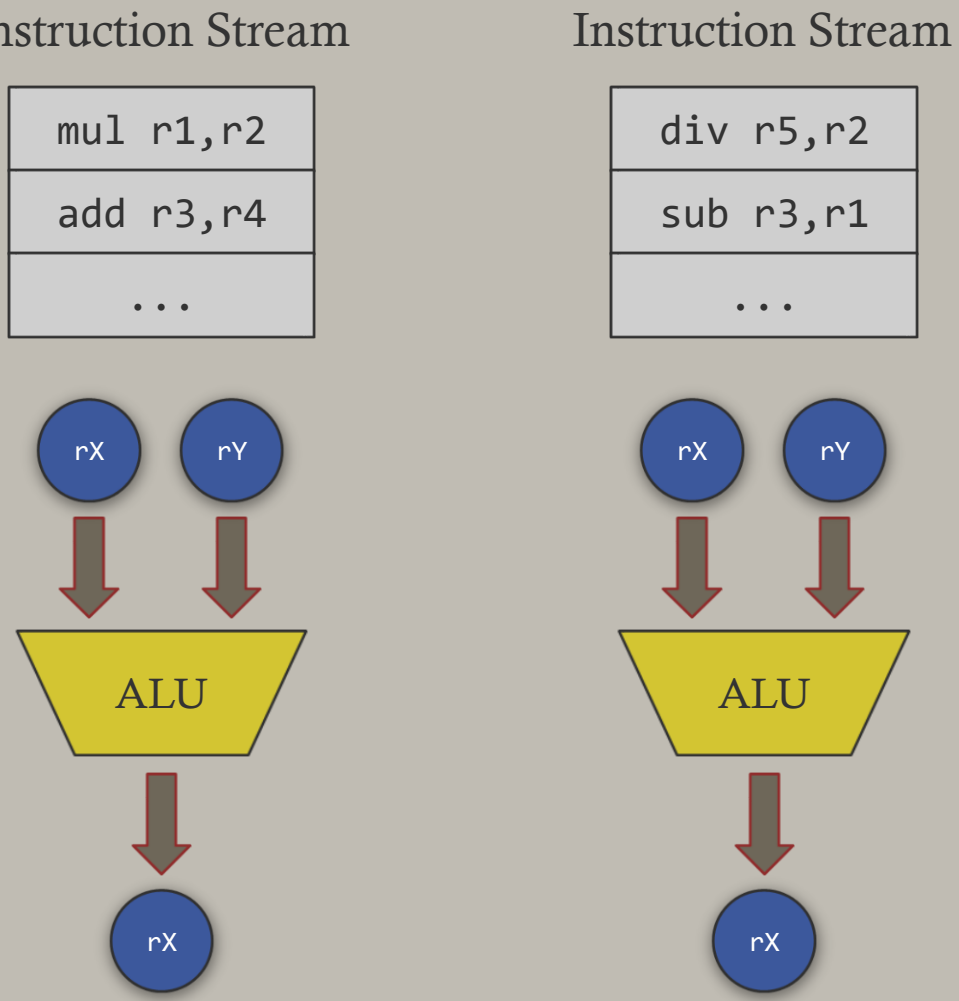
Flynn's Taxonomy

SISD



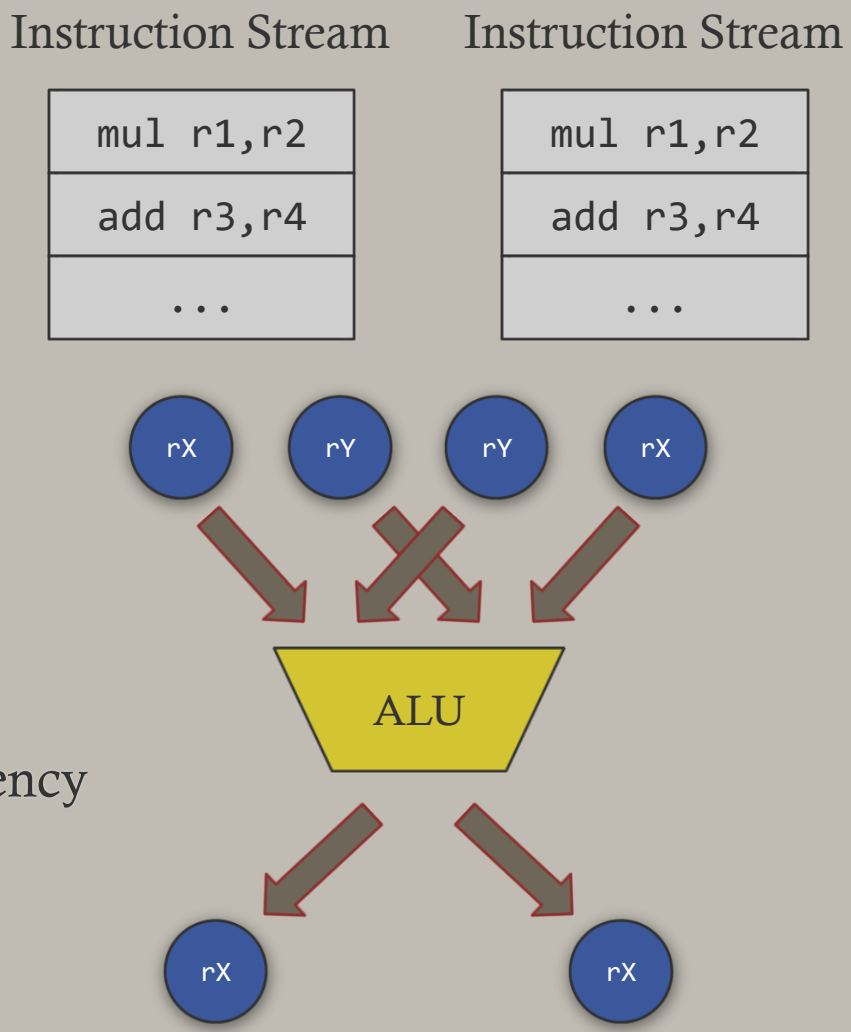
Flynn's Taxonomy

MIMD



Flynn's Taxonomy

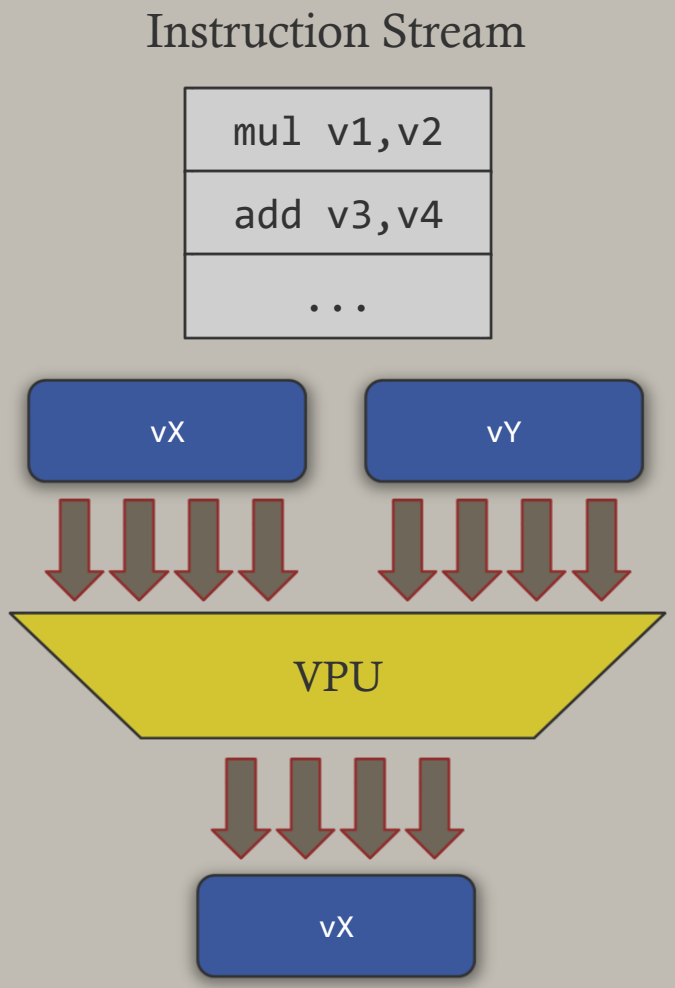
MIMD
(time-sliced)



- Not parallelism…
- but this ***is*** concurrency

Flynn's Taxonomy

SIMD

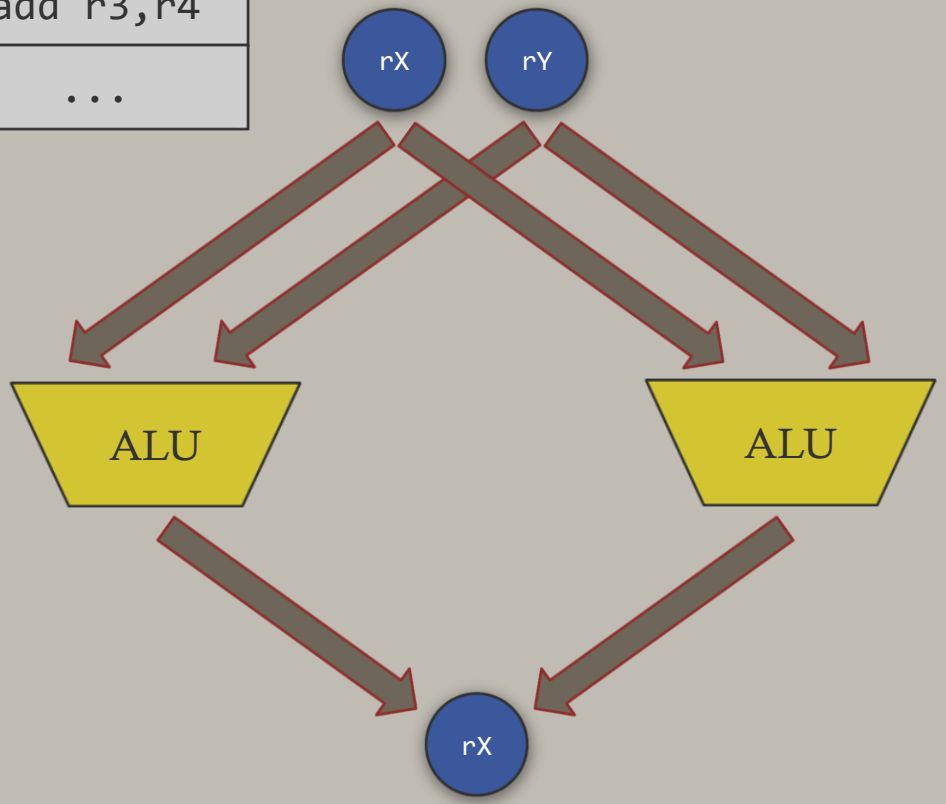


Flynn's Taxonomy

MISD

Instruction Stream

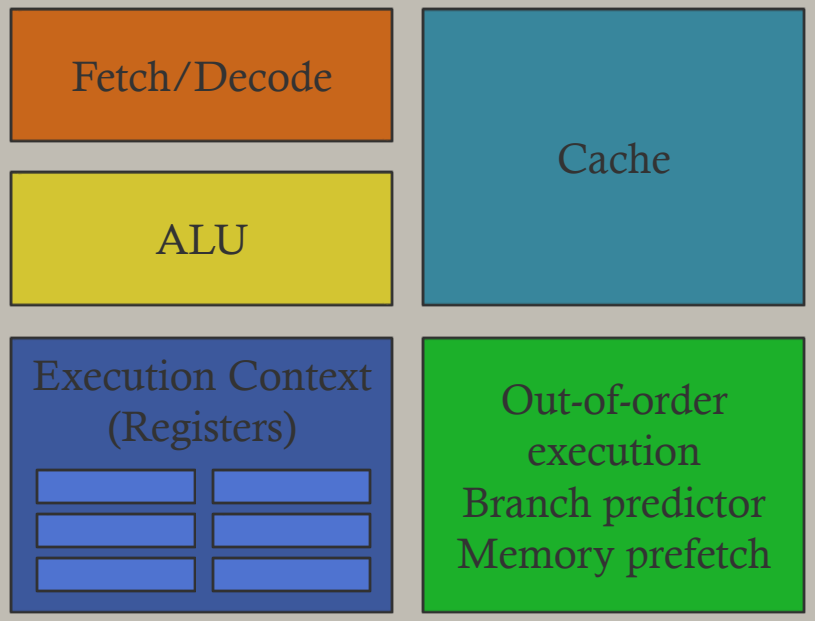
```
mul r1,r2
add r3,r4
...
```



- Primarily used for **fault tolerance** via redundancy

Manycore GPU (SIMT)

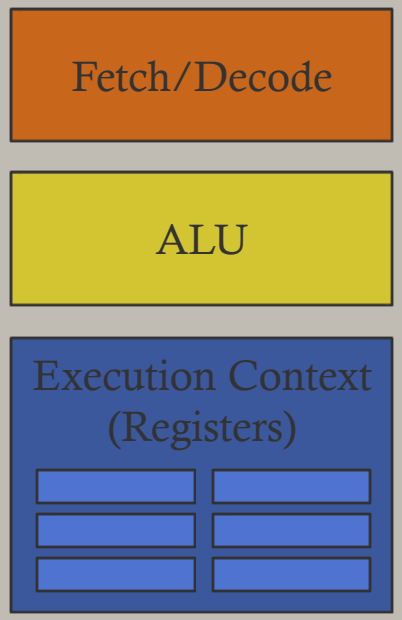
- Typical CPU core



http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

Manycore GPU (SIMT)

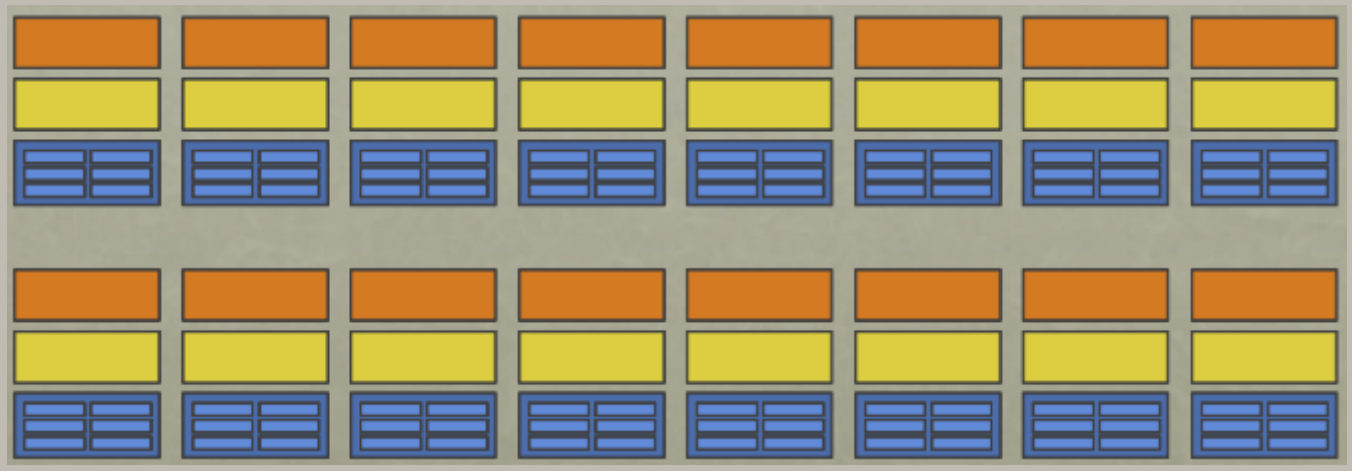
- Remove fancy OOO logic, branch predictor, cache



http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

Manycore GPU (SIMT)

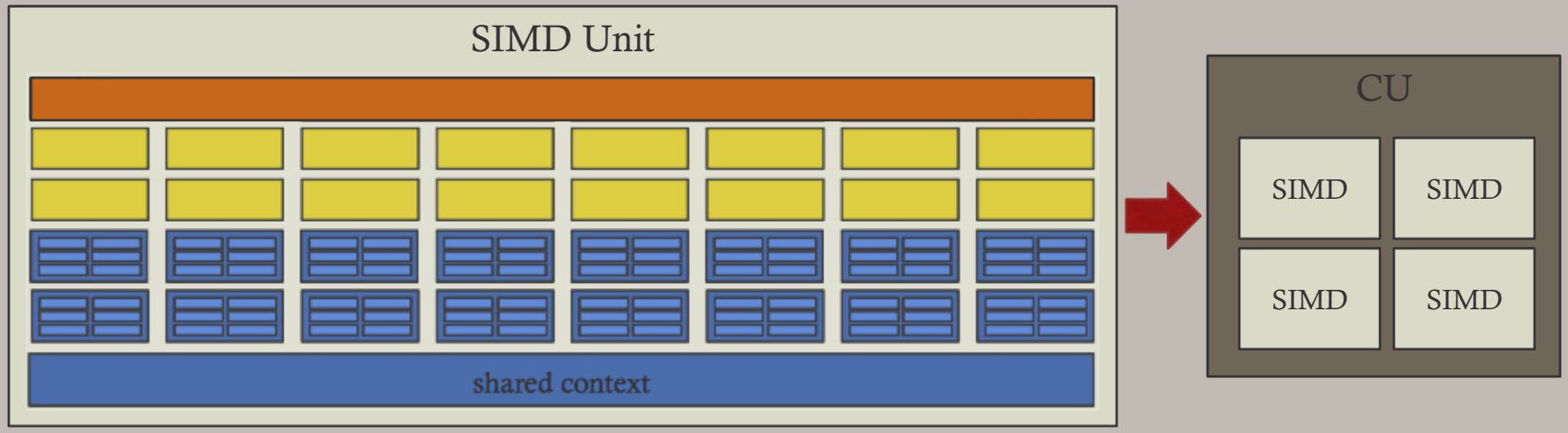
- Duplicate 16 times



http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

Manycore GPU (SIMT)

- Group **16 ALUs** and **16 execution contexts**, managed by **one fetch/decode unit** to make a **SIMD unit**
- Four SIMDs per **compute unit** (CU) = 64 "lanes"



http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

IMPLICIT PARALLELISM

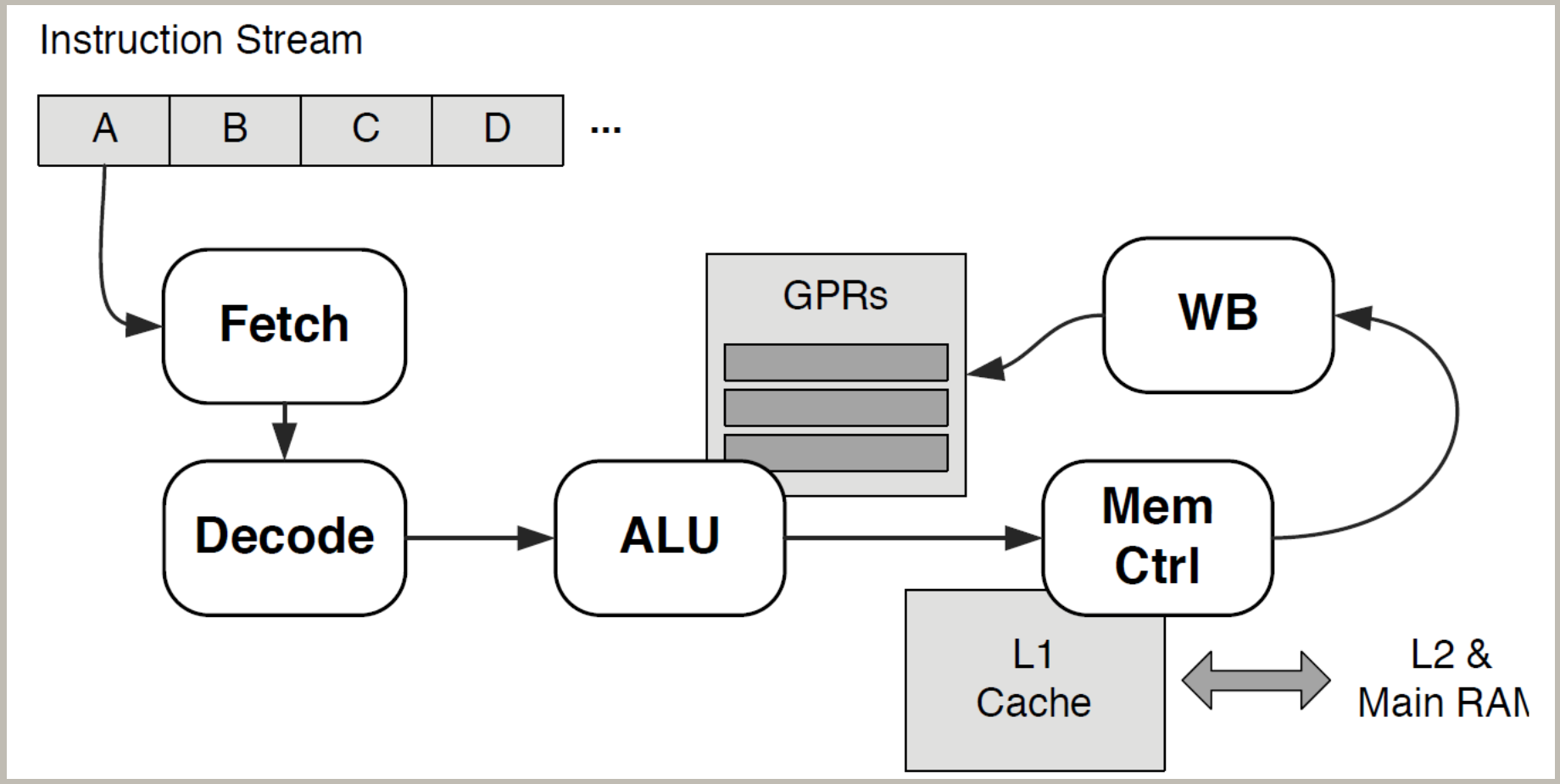
Pipelining

- Early CPUs executed **serially**
  - One instruction at a time
  - Components within CPU not well separated
  - Many components **idle** while an instruction executed
- Modern CPUs are **pipelined**
  - Each instruction passes through clearly-defined **stages**, each corresponding to a component within CPU core
  - Clear separation between components
  - Can keep all components busy by allowing multiple instructions to be "in flight" simultaneously
    - Each using a different component within the core

Pipelining

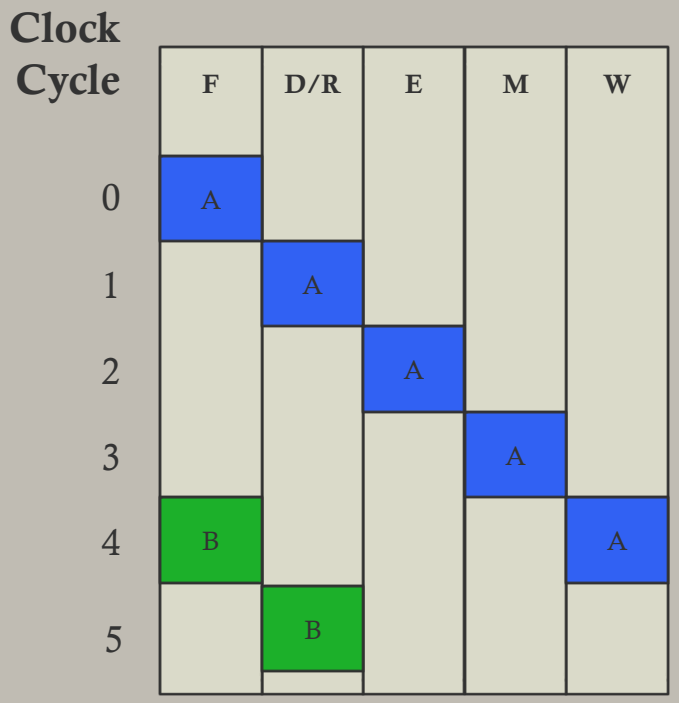
- Different CPUs define their stages differently
  - From four to five stages…
  - … to up to 30+ stages in deeply pipelined CPUs
- Basic stages:
  1. **Fetch** an instruction word from memory (**F**)
  2. **Decode** instruction into opcode and operands (**D**)
  3. **Register** access (**R**)
     - We'll assume D and R are *combined* (**D/R**)
  4. **Execute** instruction on ALU (**E**)
  5. **Memory** read/write (**M**)
  6. **Register writeback**, i.e., store results in registers (**W**)

Pipelining


Instruction Stream
A
B
C
D
...
Fetch
Decode
ALU
GPRs
WB
Mem
Ctrl
L1
Cache
L2 &
Main RAM

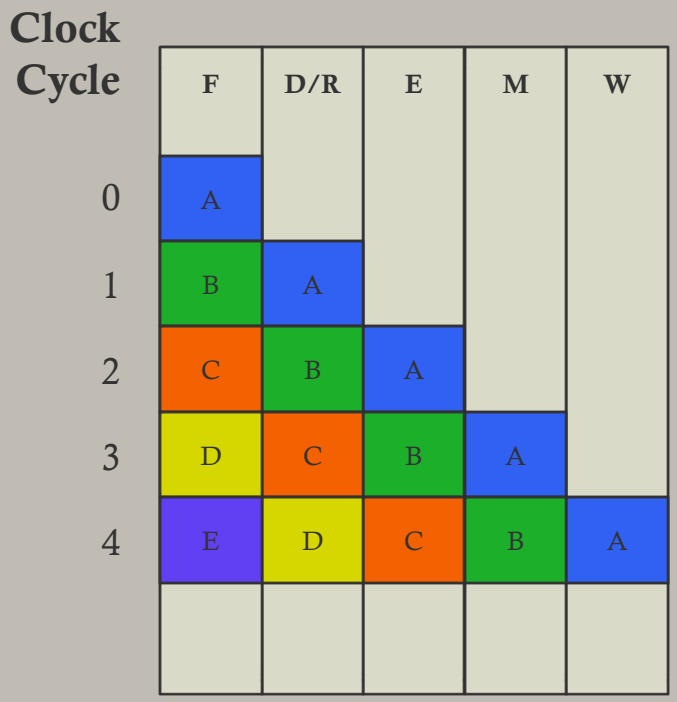
Pipelining

- In a non-pipelined CPU, instruction stages are idle much of the time.



Pipelining

- To keep each stage busy, **issue** a new instruction every clock



Pipelining

- Pipelining is a form of **instruction-level parallelism** (ILP)
- Measuring performance
  - **Throughput**: Number of instructions retired per unit time
  - **Latency**: Length of time required to retire a single instruction

Pipelining

Throughput:
$R = 1 / T_{stage}$ (instr/s)

Pipelining

- A **stall** is defined as a situation in which not all stages are kept busy
  - What kinds of situations might cause stalls?

Discussion: 5 mins

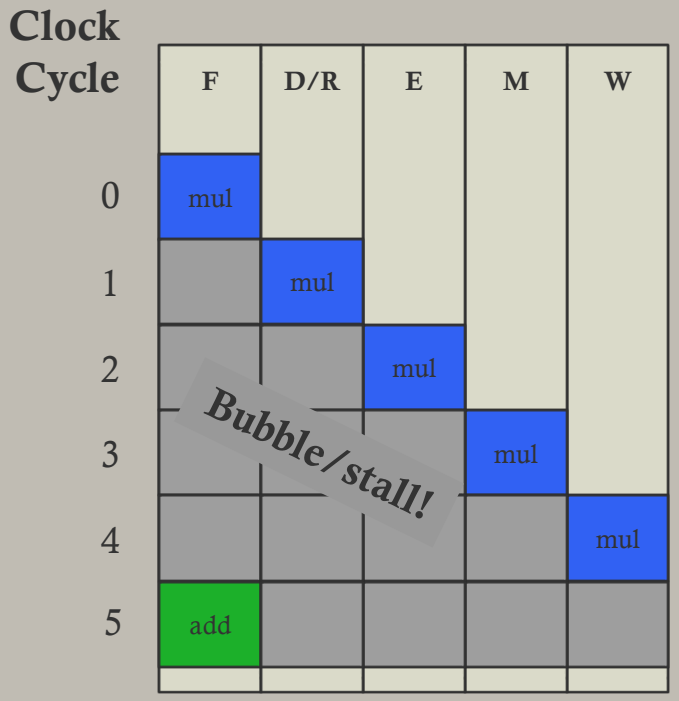
Pipelining

- A **stall** is defined as a situation in which not all stages are kept busy
  - In other words, the **next instruction** cannot be **issued** due to some kind of **dependency**
- Three kinds of dependencies in CPU pipeline:
  1. **Data** dependency: A register used in one instruction is used again in the next instruction
  2. **Branch** dependency: An instruction's operation is dependent on the results of the previous instruction (e.g. BZ)
  3. **Resource** dependency: Certain types of instructions can only execute on certain hardware (e.g. integer ALU)

Pipelining

- Dependencies are a property of an instruction stream
- Dependencies can result in **hazards**: Impacts on the CPU pipeline (stalls)
  1. **Data** dependency → **data hazard**: Results of a ***previous instruction*** are needed in order to issue this instruction
  2. **Branch** dependency → **control hazard**: Need to ***predict branches*** before result of conditional expression is known
  3. **Resource** dependency → **structural hazard**: Would like to issue, e.g., an integer add, but the ALU is ***busy***

Data Hazards



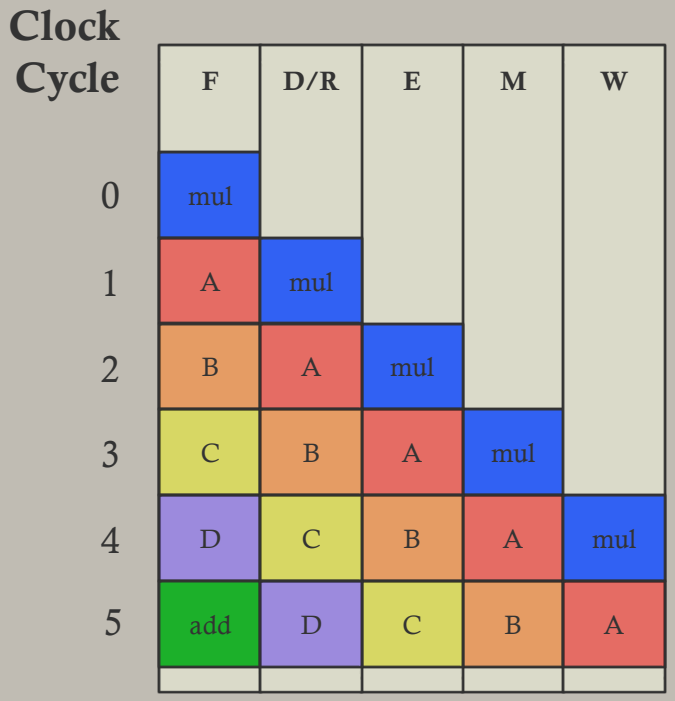

```
int madd(int a, int b, int c)
{
    int temp = a * b;
    int result = temp + c;
    return result;
}
```

```
mul   r1, r2, r3  ; r1 = r2 * r3
add   r5, r1, r4  ; r5 = r1 + r4
```

Data Hazards

- Solution to data hazard problem:
  - Find something for the **CPU to do** during the bubble!
  - **Delay slots** are the instruction slots during which the CPU is idle waiting for the current instruction to complete
- Options:
  - **Programmer** can **manually reorder** statements in code
  - **Compiler** (optimizer) can reorder instructions automatically, based on analysis of program logic
  - **CPU** can reorder instructions on the fly
    - Called **out-of-order execution** (OOO execution)

Data Hazards



OOO execution allows the CPU to **reorder** instructions A, B, C and D (taken from *before* the **add**) so that they come between the **add** and the **mul** instructions

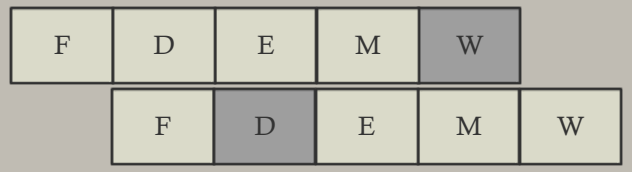
```
mul    r1, r2, r3  ; r1 = r2 * r3
instrA r5
instrB r6, r7
instrC r0, r8
instrD r9
add    r5, r1, r4  ; r5 = r1 + r4
```

Data Hazards

http://cseweb.ucsd.edu/classes/wi05/cse240a/pipe2.pdf

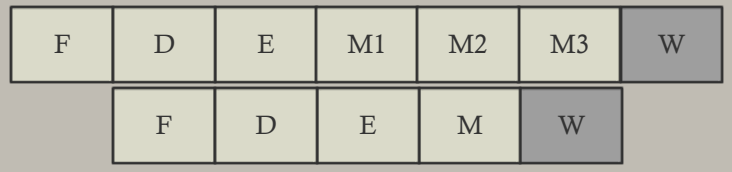
- Data dependencies come in three flavors
  - **Read after write** (RAW)

```
mul   r1,r2,r3
add   r5,r1,r4
```



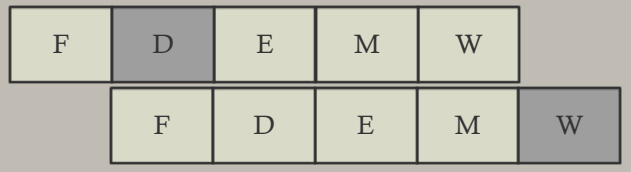
  - **Write after write** (WAW)

```
add   r1,r2,r3
sub   r1,r2,r4
```



  - **Write after read** (WAR)
    - Can't happen in vanilla pipeline

```
mul   r2,r1,r3
sub   r1,r5,r4
```



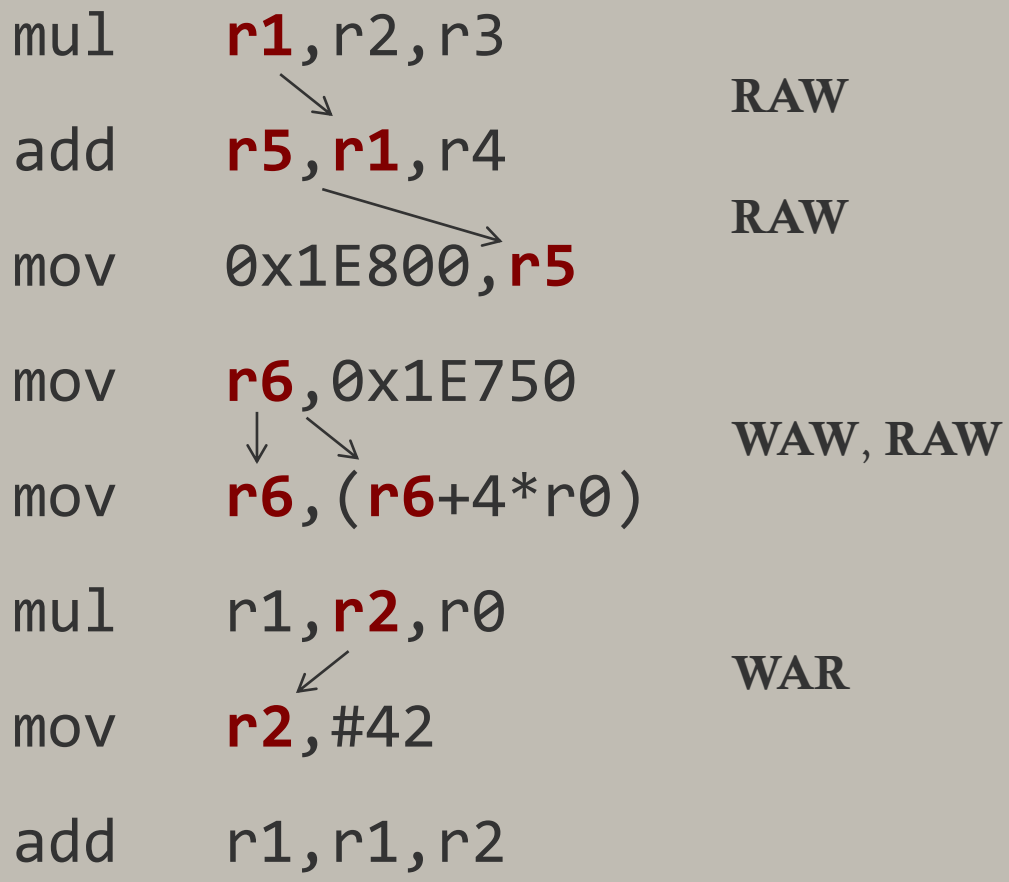
Data Hazards

- Exercise: Identify the RAW, WAW and WAR dependencies in the following instruction sequence

```
mul   r1,r2,r3
add   r5,r1,r4
mov   0x1E800,r5
mov   r6,0x1E750
mov   r6,(r6+4*r0)
mul   r1,r2,r0
mov   r2,#42
add   r1,r1,r2
```

Data Hazards

- Solution:



Data Hazards

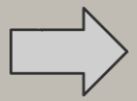
- Exercise: Reorder the instruction sequence to **minimize** stalls

```
mul   r1,r2,r3
add   r5,r1,r4
mov   0x1E800,r5
mov   r6,0x1E750
mov   r6,(r6+4*r0)
mul   r1,r2,r0
mov   r2,#42
add   r1,r1,r2
```

Data Hazards

- One possible solution…
  - noticing that there are really two independent sequences
  - so interleave them

```
mul   r1,r2,r3
add   r5,r1,r4
mov   0x1E800,r5
mov   r6,0x1E750
mov   r6,(r6+4*r0)
mul   r1,r2,r0
mov   r2,#42
add   r1,r1,r2
```



```
mul   r1,r2,r3
mov   r6,0x1E750
add   r5,r1,r4
mul   r1,r2,r0
mov   0x1E800,r5
mov   r2,#42
mov   r6,(r6+4*r0)
add   r1,r1,r2
```

Register Renaming

- Register reuse within a sequence of instructions can introduce false dependencies (WAW, WAR)
- One solution: **register renaming**
  - Register names used by program ("virtual" registers) are **mapped** to real hardware registers by CPU or compiler

before register renaming…

```
add   r3,r4,r5
ld    r7,(r3)

sub   r3,r12,r11
st    (r15),r3
```

false dependency! (WAR)

after register renaming…

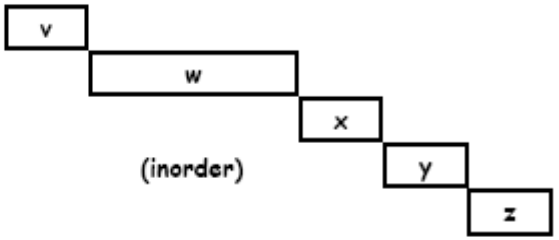
```
add   hw3,hw4,hw5
ld    hw7,(hw3)

sub   hw20,hw12,hw11
st    (hw15),hw20
```

Register Renaming

http://www.math.nsysu.edu.tw/~lam/MPI/lecture/SUPERSCALAR_ARCHITECTURE.ppt

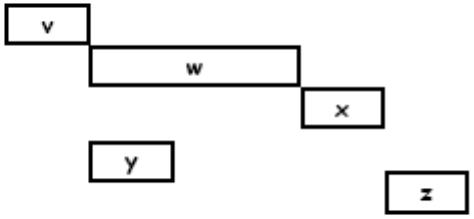
(Single adder, data dependence)

```
v:   addt  $f2, $f4, $f10
w:   mult $f10, $f6, $f10
x:   addt $f10, $f8, $f12
y:   addt  $f4, $f6,  $f4
z:   addt  $f4, $f8, $f10
```

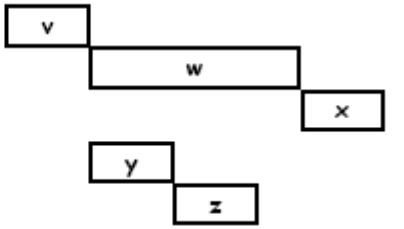


- Can start y as soon as adder available
- Must hold back z until `$f10` not busy & adder available

```
v:   addt  $f2, $f4, $f10
w:   mult $f10, $f6, $f10
x:   addt $f10, $f8, $f12
y:   addt  $f4, $f6,  $f4
z:   addt  $f4, $f8, $f10
```




```
v:   addt  $f2, $f4, $f10a
w:   mult $f10a, $f6, $f10a
x:   addt $f10a, $f8, $f12
y:   addt  $f4, $f6,  $f4
z:   addt  $f4, $f8, $f14
```



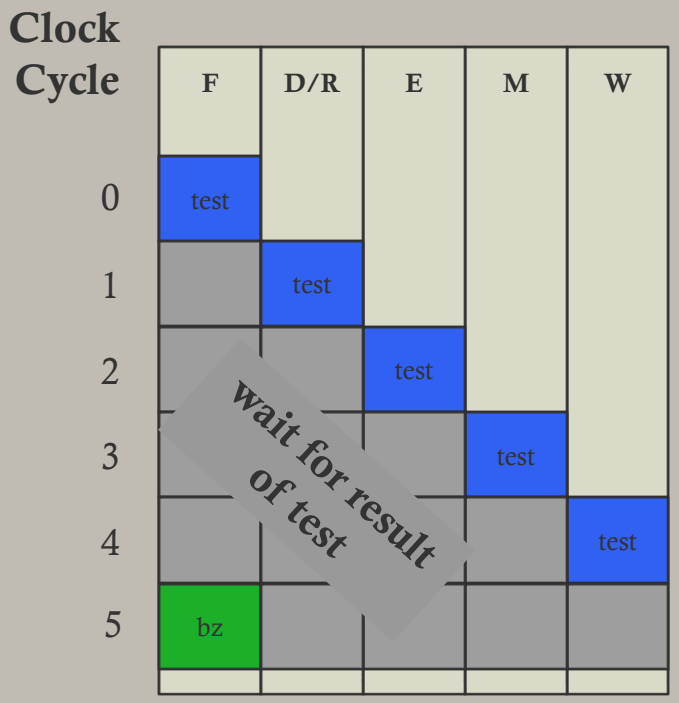
Memory Latency Hiding

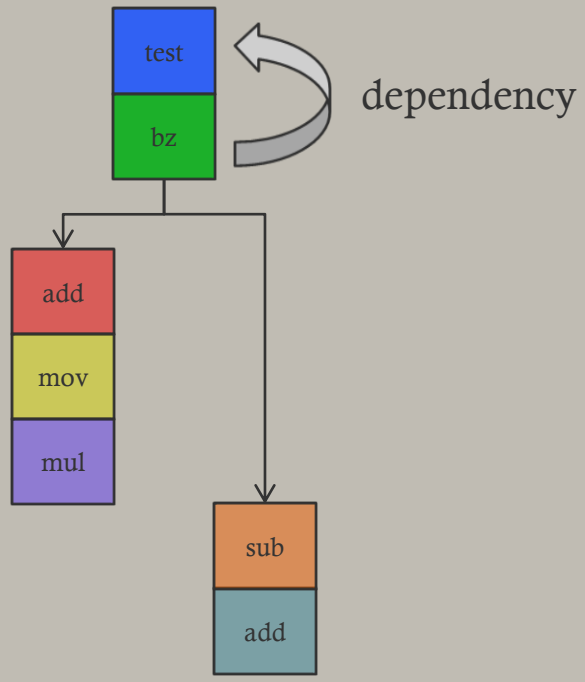
- Note that stalls can also occur due to **memory latency**
  - mov r3,[r1+8]
  - Might take 4, 40 or 400 cycles! (L1, L2, main RAM)
- **OOO execution** can also help to **hide** memory latency
  - While waiting for memory controller to respond, execute other independent instructions in delay slots
- **Multithreading** is another way to **hide** memory latency
  - While one thread is waiting for memory, other thread(s) can be scheduled to use the CPU core

Control Hazards

- What should the CPU do when a **conditional branch** instruction is encountered?
  - Result of the condition ***not yet known*** at time of issue
  - Pipeline can be 10+ stages deep
  - Conditional branch therefore introduces 10+ cycle stall
- This is called a **branch penalty**
  - Control hazard
  - It's a special case of a RAW data hazard

Control Hazards





Control Hazards

- Solutions to branch penalty problem:
- Software:
  - **Loop unrolling** to increase run length (fewer if tests)
  - **Rearrange code** to minimize "branchiness"
- Hardware:
  - **Delay slots**
    - Choose instructions to run during the stall (**OOO execution**)
  - **Choose one** of the two branches and execute **speculatively**
    - If the guess is **correct**, no stall! ☺
    - If the guess is **wrong**, **flush** the pipeline, **back out** intermediate results, and **start again** on the correct branch

Speculative Execution

- How should the CPU guess?
  - Simple idea: Just **assume**…
    - ***backwards*** branches are ***always*** taken,
    - ***forward*** branches are ***never*** taken
  - Very common pattern in for loops

```
for (i = 0; i < count; ++i)
{
    // do work...

    if (special_case)
       break; // forward branch (rare)

} // backward branch (common)
```

Speculative Execution

- Another idea: Rely on manual tagging of **`if`** statements

```
for (i = 0; i < count; ++i)
{
    // do work...

    if (UNLIKELY(special_case))
       break;

}
```

```
// gcc definitions:
#define LIKELY(x)       __builtin_expect((x),1)
#define UNLIKELY(x)     __builtin_expect((x),0)
```

Speculative Execution

- Another idea: Include **branch prediction logic** on the CPU die
  - Keeps a history of which branches were taken
  - Logic can get quite complex
  - Some CPUs can detect **patterns** like Y, N, Y, N, Y, N, …
- Increases transistor count and circuit complexity significantly

- Speculative execution requires results of calculations to be stored temporarily in

Predication

- Yet another idea:
  - Don't branch **at all**!
  - Instead, always execute **both** sides of the branch
  - But **mark** each instruction with the condition (predicate) with which it is associated
  - CPU only **actually commits** results of predicated instructions when the condition (predicate) is true
- This approach is called **predication**
- Simpler form: partial predication
  - Conditional **move**, conditional **select** instructions

Predication

```
int SafeFloatDivide_pred(float a, float b, float d)

{

// convert Boolean (b != 0.0f) into either 1U or 0U

const unsigned condition = (unsigned)(b != 0.0f);

// convert 1U -> 0xFFFFFFFFU

// convert 0U -> 0x00000000U

const unsigned mask = 0U - condition;

// calculate quotient (will be QNaN if b == 0.0f)

const float q = a / b;

// select quotient when mask is all ones, or default value d when mask is all zeros (NOTE: this won't work as written -- you'd need //to use a union to interpret the floats as unsigned for masking)

const float result = (q & mask) | (d & ~mask);

return result;

}
```

Partial Predication

- Conditional select
  - e.g., PowerPC **fsel** and **isel** instructions

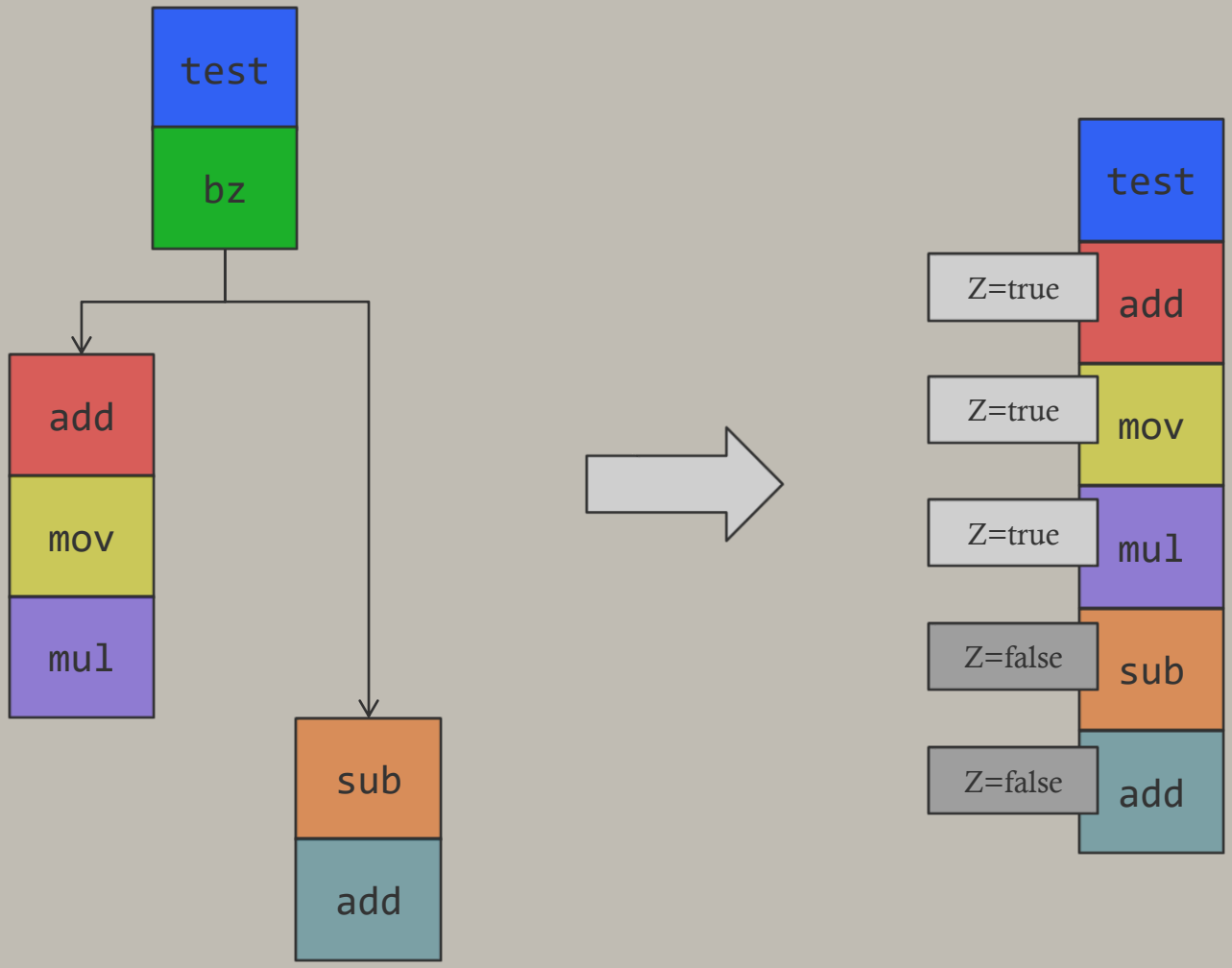
    isel RR, RA, RB, *predicate_code*
    - If predicate bit is 1, move RA→RR
    - If predicate bit is 0, move RB→RR

```
  cmpwi %cr7,%r11,42
  beq cr7,Label1    ; if r11 == 42
  li %r0,0          ; else r0 = 0
  b Label2
Label1:
  li %r0,1          ; set r0 = 1
Label2:
```

```
li    %r1,0
li    %r2,1
cmpwi %cr7,%r11,42
isel  %r0,%r2,%r1,30
```

30 means "check the EQ status bit of %cr7"

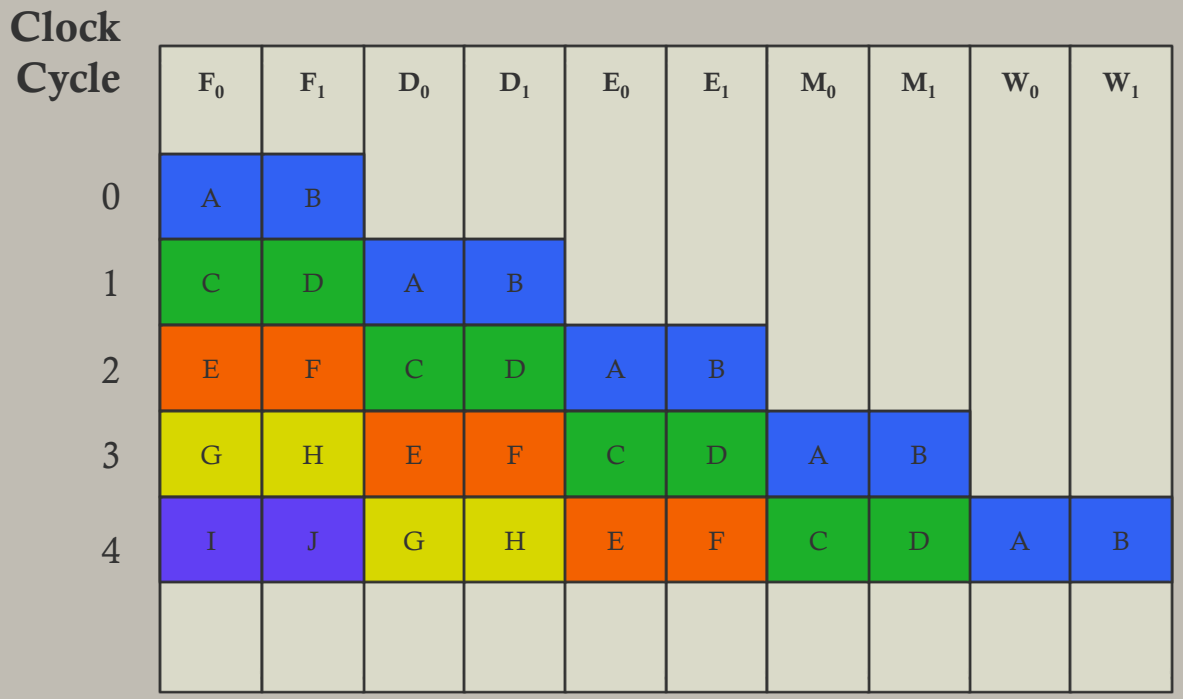
Full Predication



Superscalar

- If pipelining improves throughput, can we improve it further by adding **more parallelism**?
  - Yes: If CPU has **two of each** functional unit, it can issue **two instructions per clock**
- This is called **superscalar** CPU architecture

Superscalar



Superscalar

- Resource dependencies (structural hazards) can occur in a superscalar CPU
  - e.g., we'd like to issue 3 **integer arithmetic operations** consecutively, but there are only 2 integer ALUs
    - The third integer op must stall, or…
    - we must defer it and find a **non-integer op** to put in its place
      - floating-point arithmetic op
      - memory op
      - comparison op
      - SIMD vector op
      - etc.

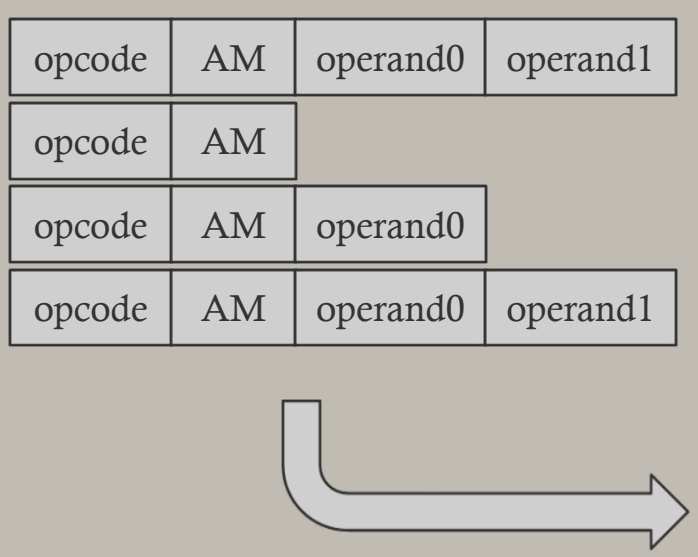
Very Long Instruction Word (VLIW)

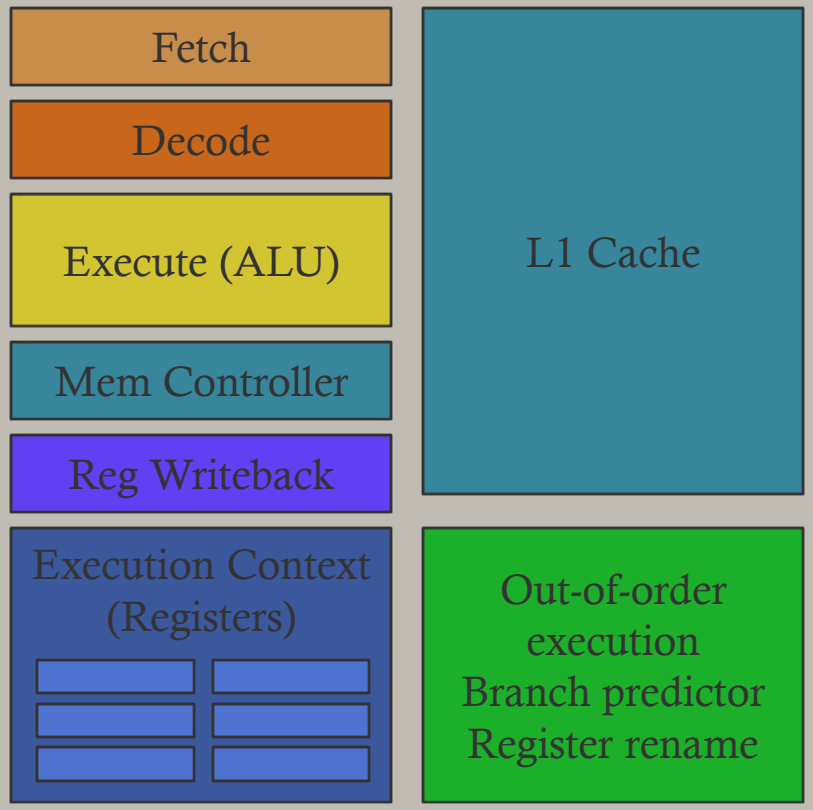
- Lots of silicon/transistors are required to implement:
  - OOO execution logic
  - Branch predictors
  - Register renaming
- Simpler design:
  - Include all the **parallel functional units**
  - … but **omit** the complex **scheduling logic**
  - Leave it up to the **programmer/compiler** to find all opportunities for instruction-level parallelism…

Very Long Instruction Word (VLIW)

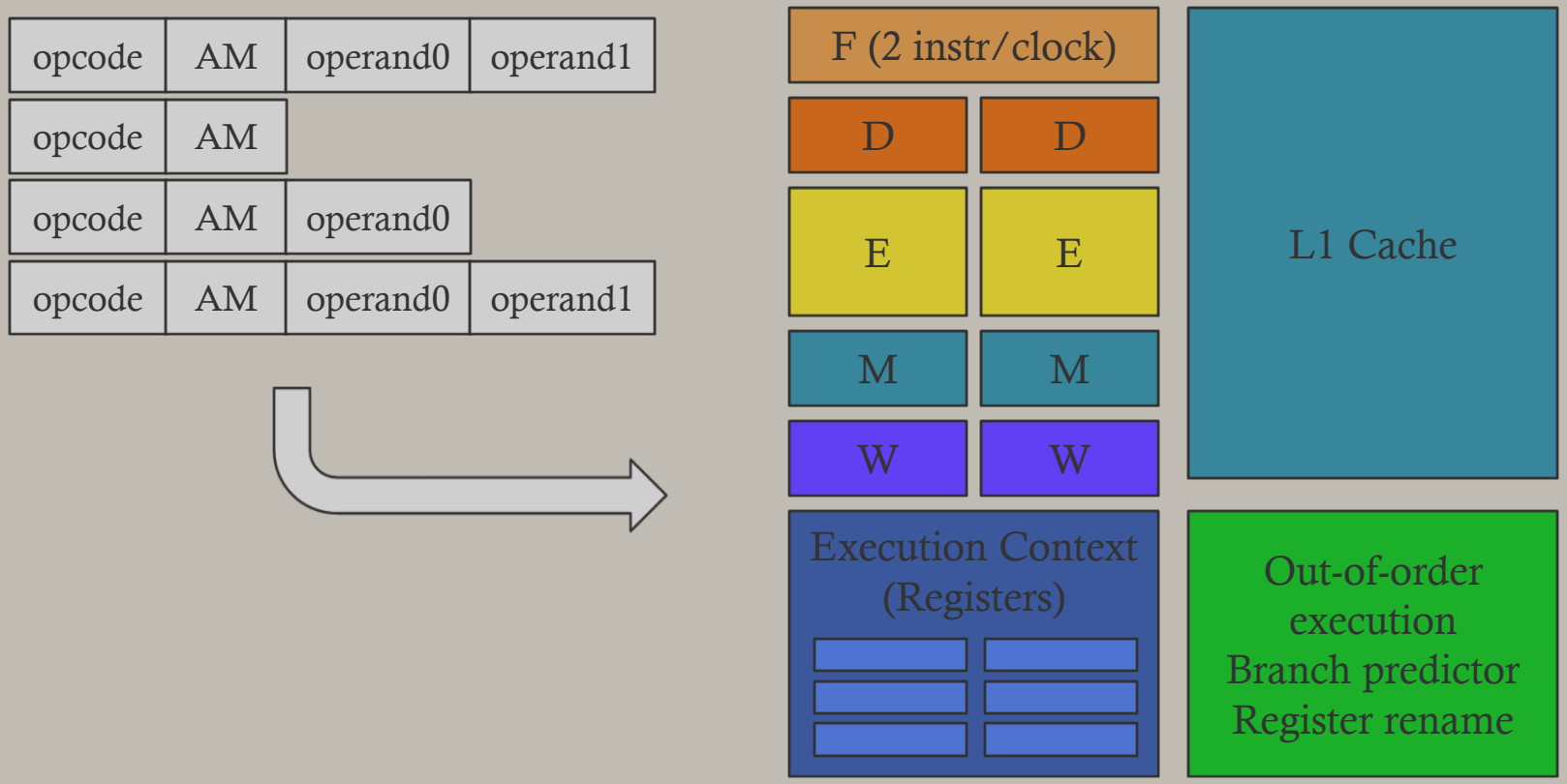
- But how to expose the parallel functional units to the programmer/compiler?
  - Each **instruction word** can encode **two or more instructions**
  - The program requests ILP **explicitly**
  - Gives rise to very long (wide) instruction words
  - **Very long instruction word** (VLIW) architecture
- e.g., VU0 and VU1 on the PlayStation2
  - Each VU had **two "slots"** per instruction
  - Up to the programmer/compiler to **reorder** assembly code in order to **fill both slots** of each instruction word

Pipelined CPU





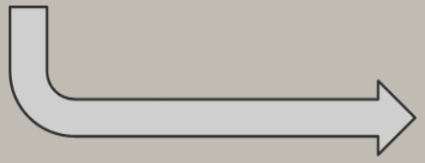
Superscalar CPU

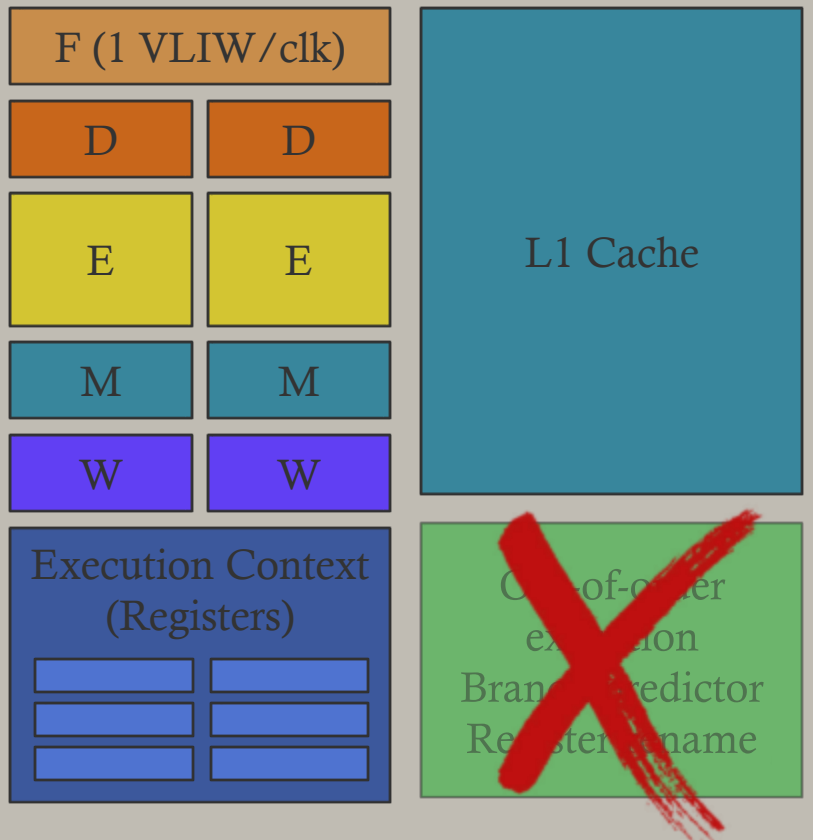


VLIW CPU

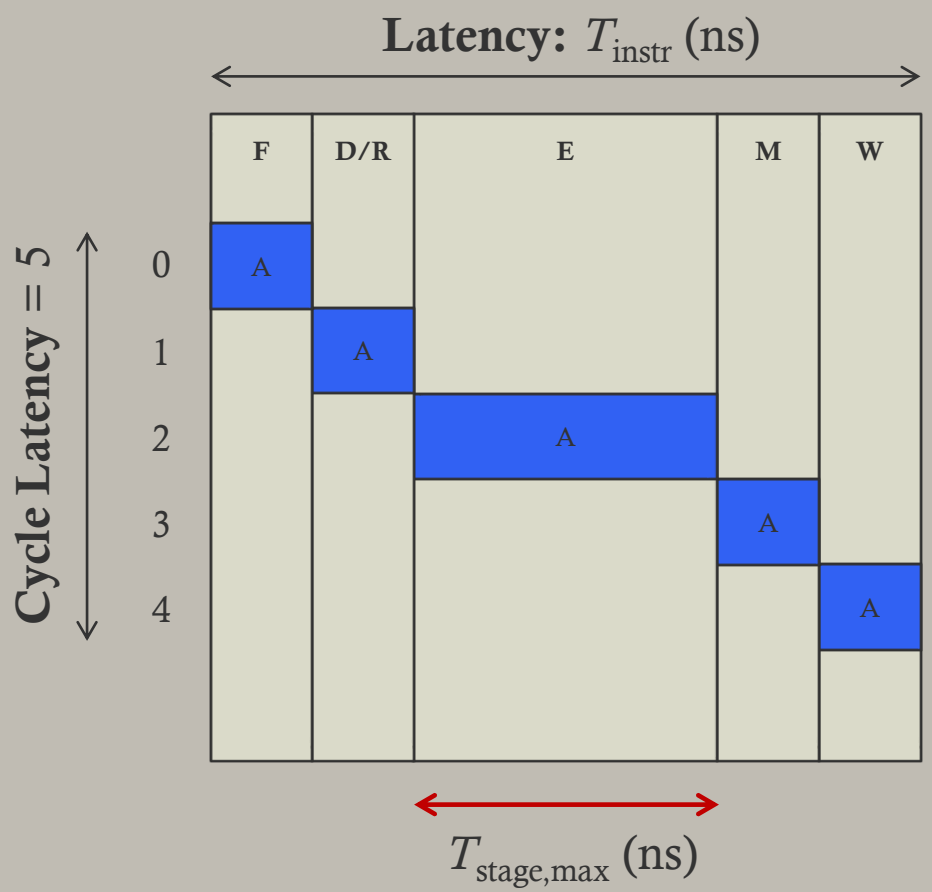
opcode	AM	operand0	operand1	opcode	AM	operand0	operand1
opcode	AM	operand0	operand1	opcode	AM	operand0	operand1

...



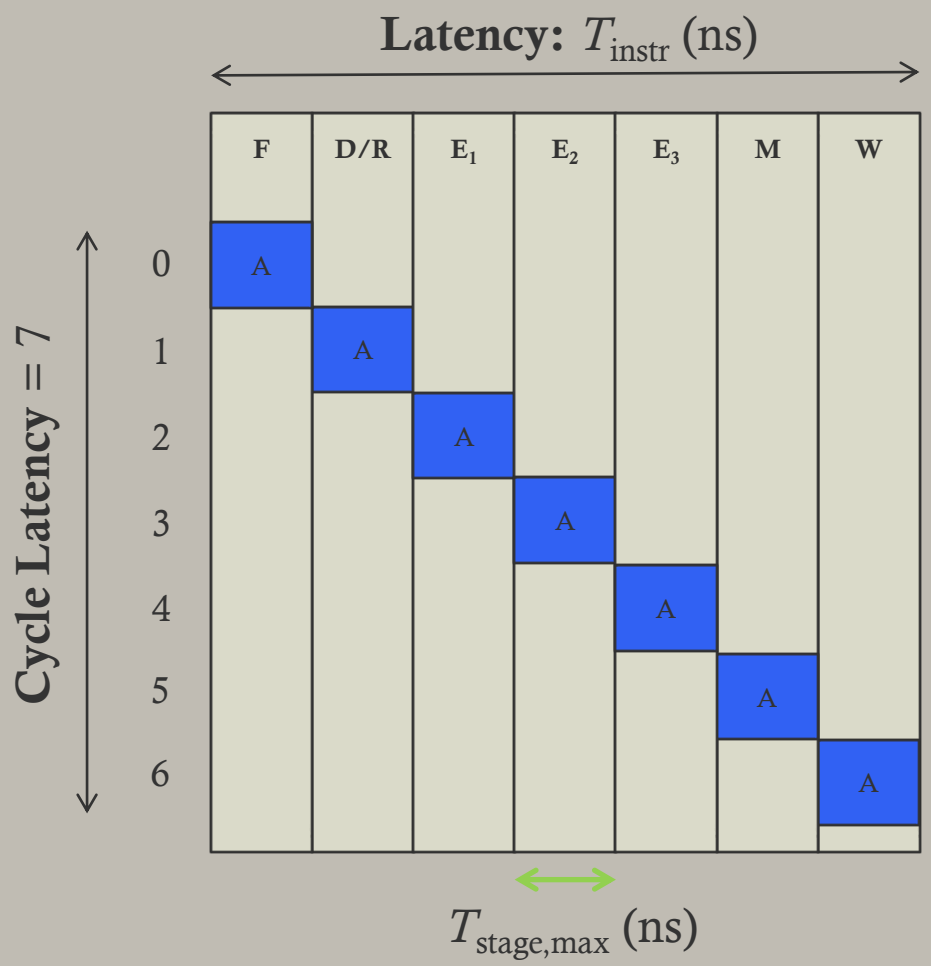


Toward Deeper Pipelines



Throughput:
$R = 1 / T_{stage,max}$ (instr/s)

Toward Deeper Pipelines



Throughput:

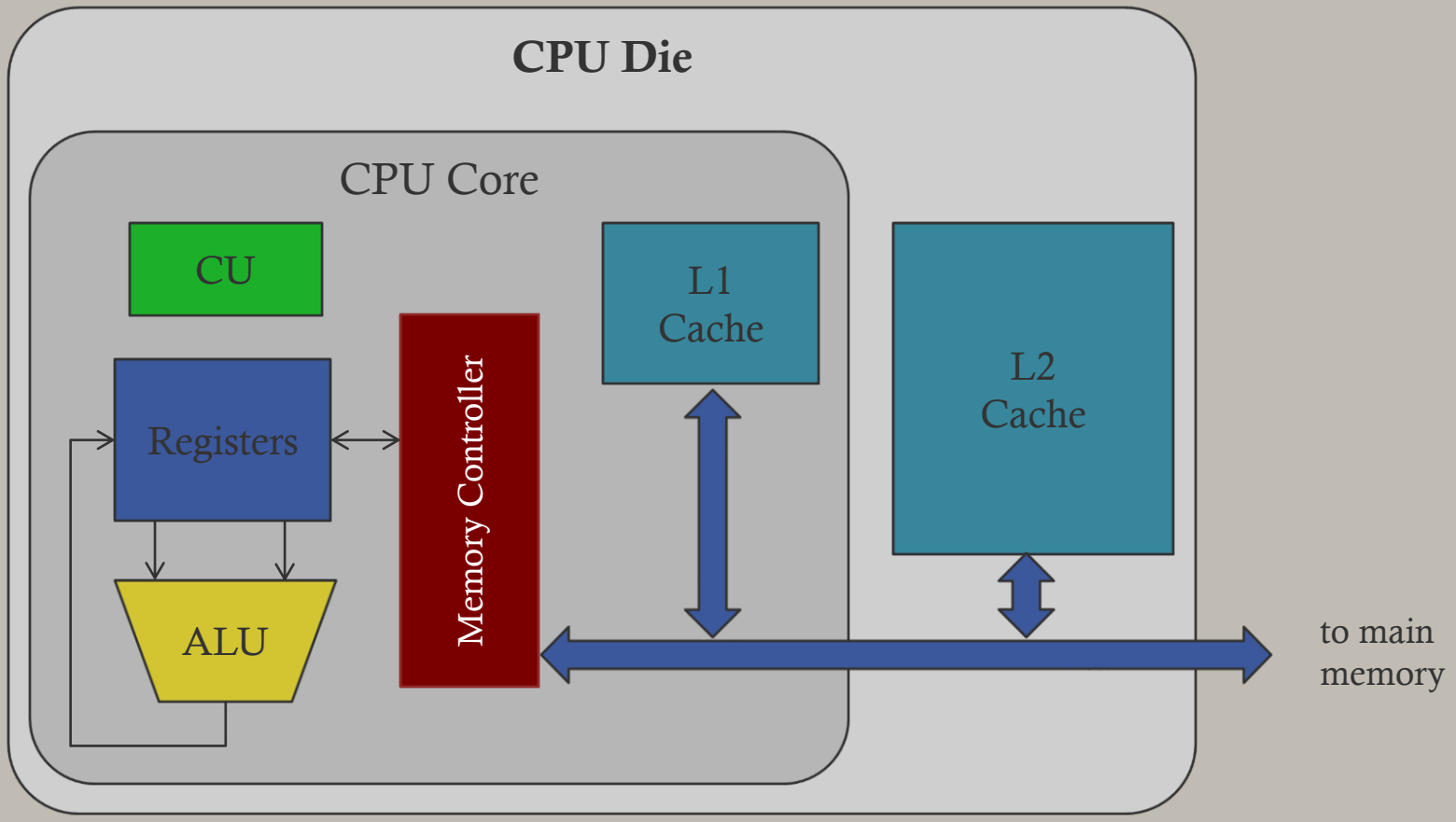
$R = 1 / T_{stage,max}$ (instr/s)

Toward Deeper Pipelines

- There are limits to how many stages a pipeline can be broken into
  - Increased clock speeds → increased heat and power consumption
  - More stages → more-complex forwarding logic
  - More-complex logic → more transistors, larger die area
  - More stages → bigger cost on branch mispredict
  - Minimum time to complete a stage is limited by technology considerations such as register access latency
- As such, pipeline depths *typically* vary between 5 stages at the low end, and 31 stages at the very high end

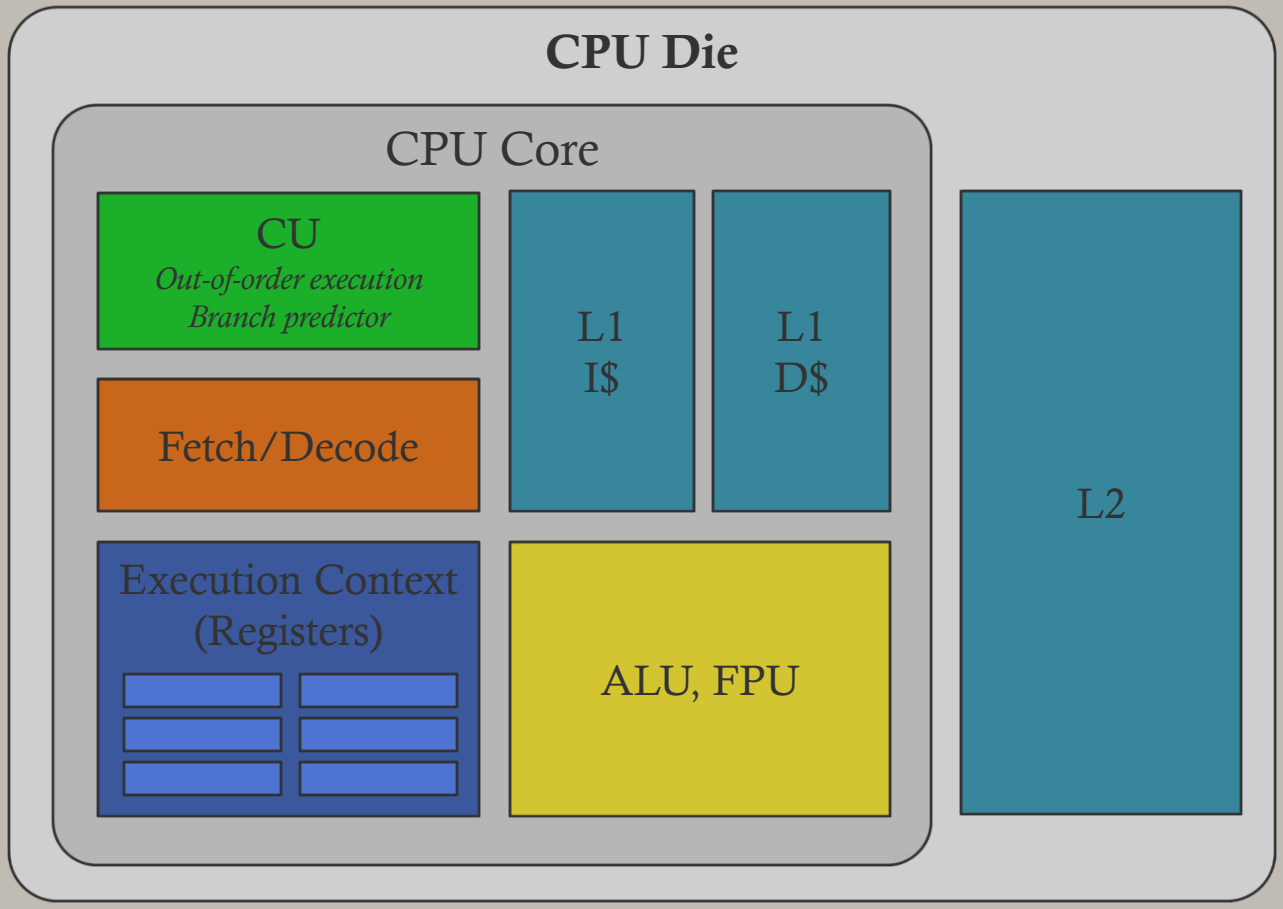
EXPLICIT PARALLELISM

Review: Simple Serial CPU

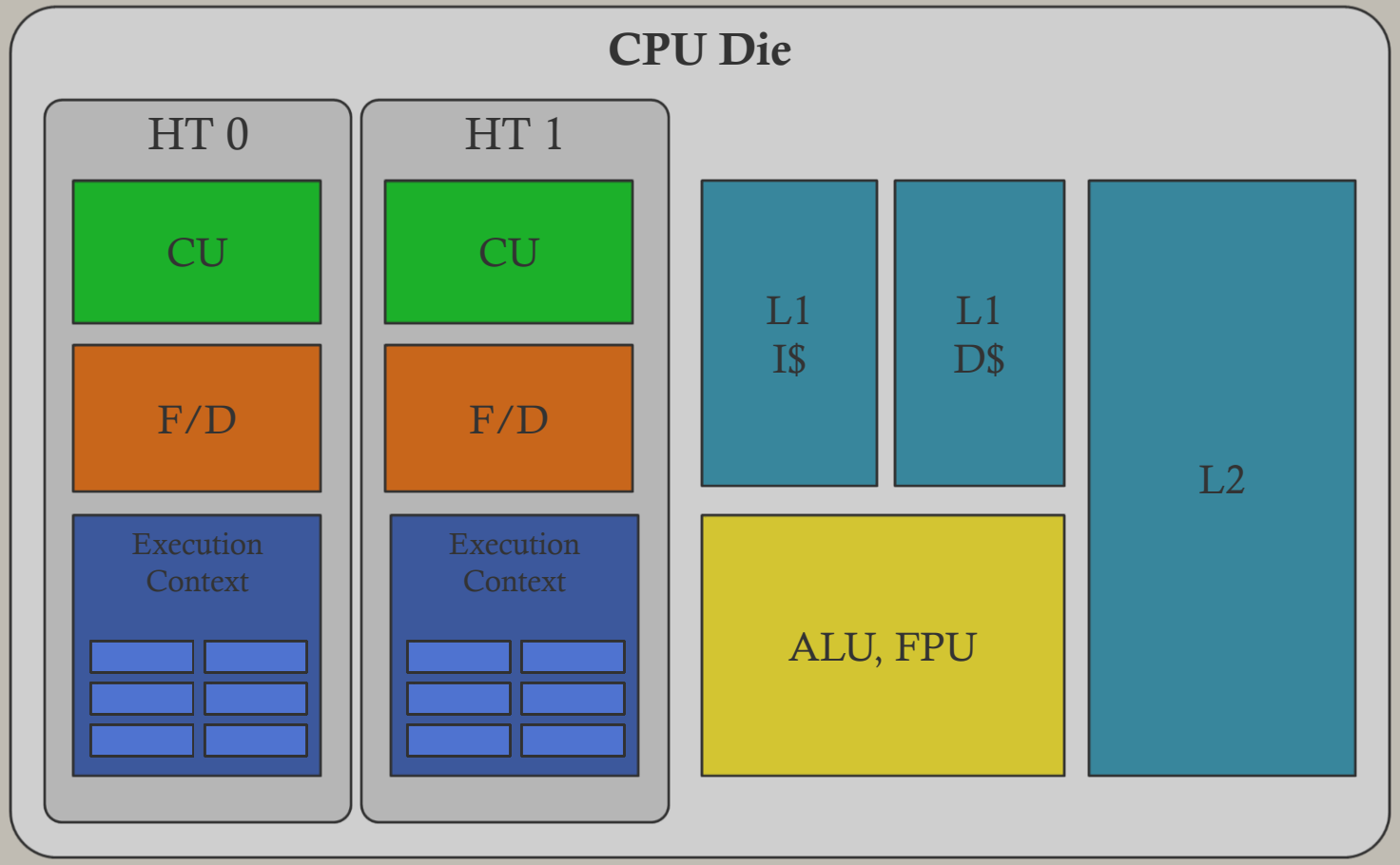


Review: Simple Serial CPU

- Rearrange the diagram a bit…



Hyperthreaded CPU



HyperThreading

- If you have a windows OS, open task manager and go to CPU section, in that screen if you see the number of logical processors say 4 but your CPU is dual core, it is hyper threading.
- If you have Linux, open system monitor or whichever task manager you have on your Linux, and go to resources tab. Check how many CPUx do you see. If you see CPU1, CPU2 CPU3, CPU4 for a dual core CPU, it's hyperthreading.
- Run wmic (Windows Management Instrumentation) in command line:
  - CPU Get NumberOfCores,NumberOfLogicalProcessors /Format:List

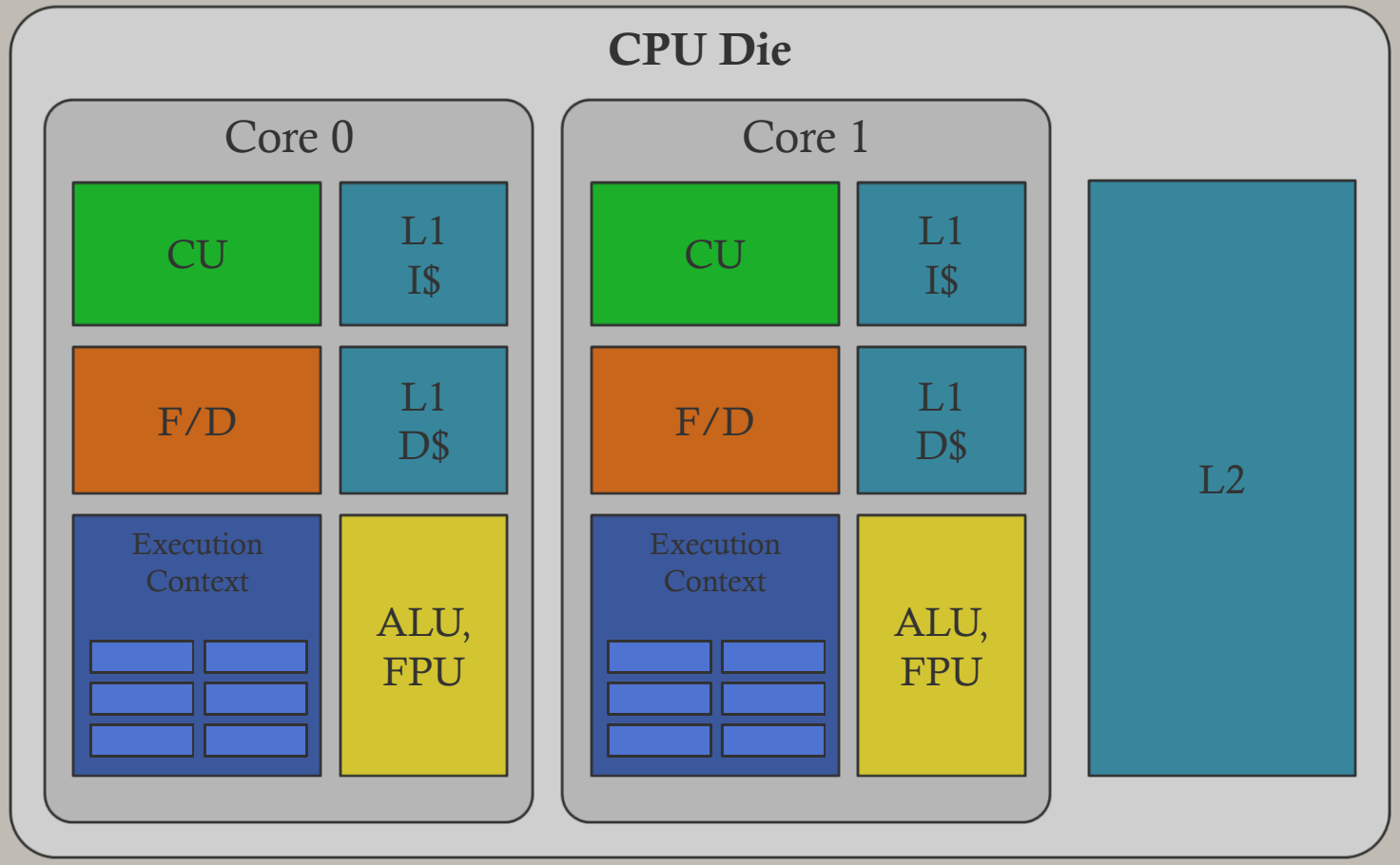
Check number of cores

- Step 1. Press **Win + R** to open the Run window.
- Step 2. Type **cmd** into the dialog box and press **Shift + Ctrl + Enter** to run Command Prompt as administrator.
- Step 3. Type **wmic** into the window and hit **Enter**.
- Step 4. Type **CPU Get NumberOfCores ,NumberOfLogicalProcessors /Format:List** and hit **Enter** to execute this command line.

How to Enable Hyper-Threading

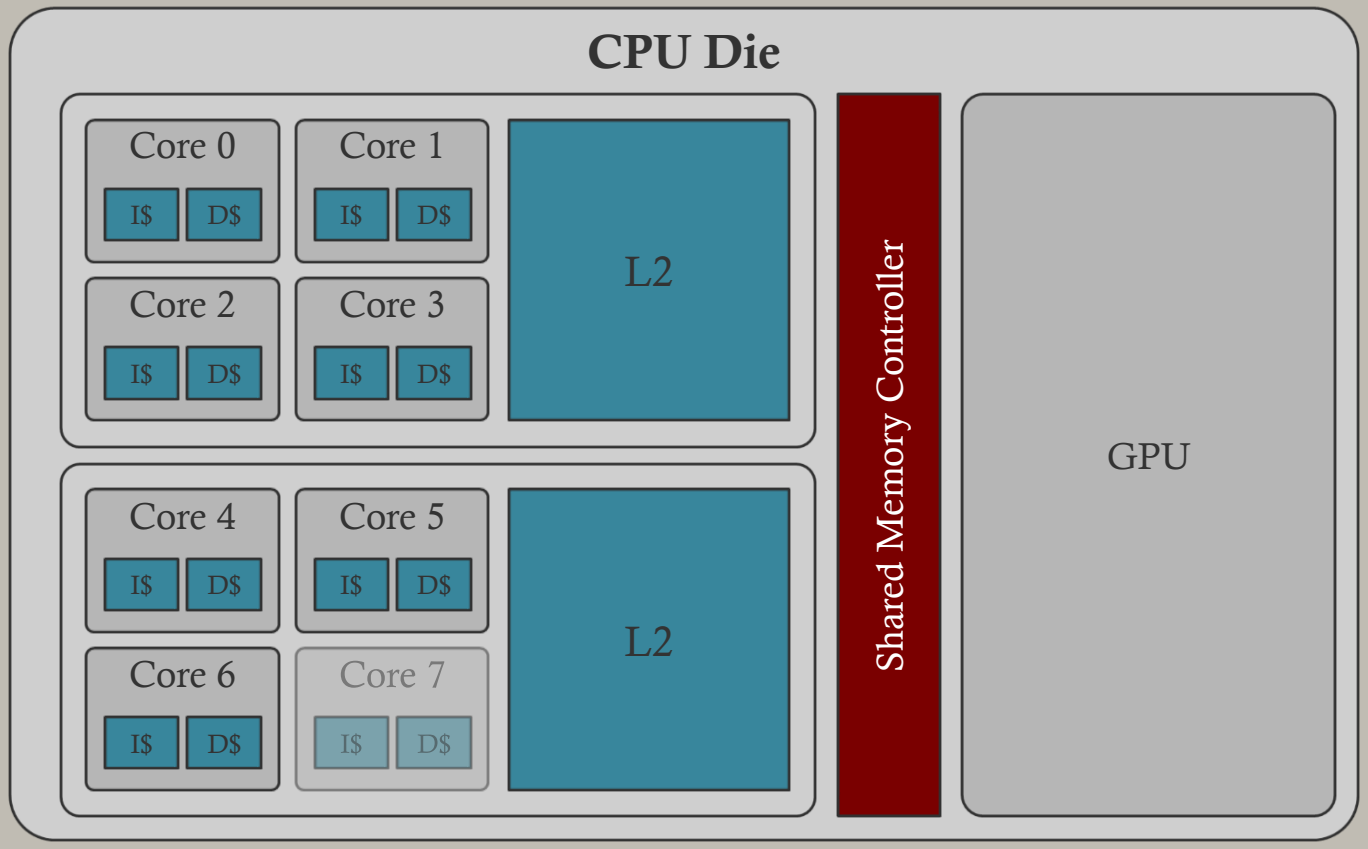
- Step 1. Press **Win + I** to open Windows Settings.
- Step 2. Head to **Update & Security > Recovery**, then click **Restart** now under the **Advanced startup** at the right pane.
- Step 3. Wait for your computer to restart, then you need to choose **Troubleshoot > Advanced options > UEFI Firmware Settings** to boot your computer into the BIOS settings.
- Step 4. In the following window, select the **Processor** option and head to **Intel ® Hyperthreading Options**. You can select the option you need. To disable hyper-threading, you should choose **Disabled** here.
- Step 5. Save your changes and exit the BIOS settings.

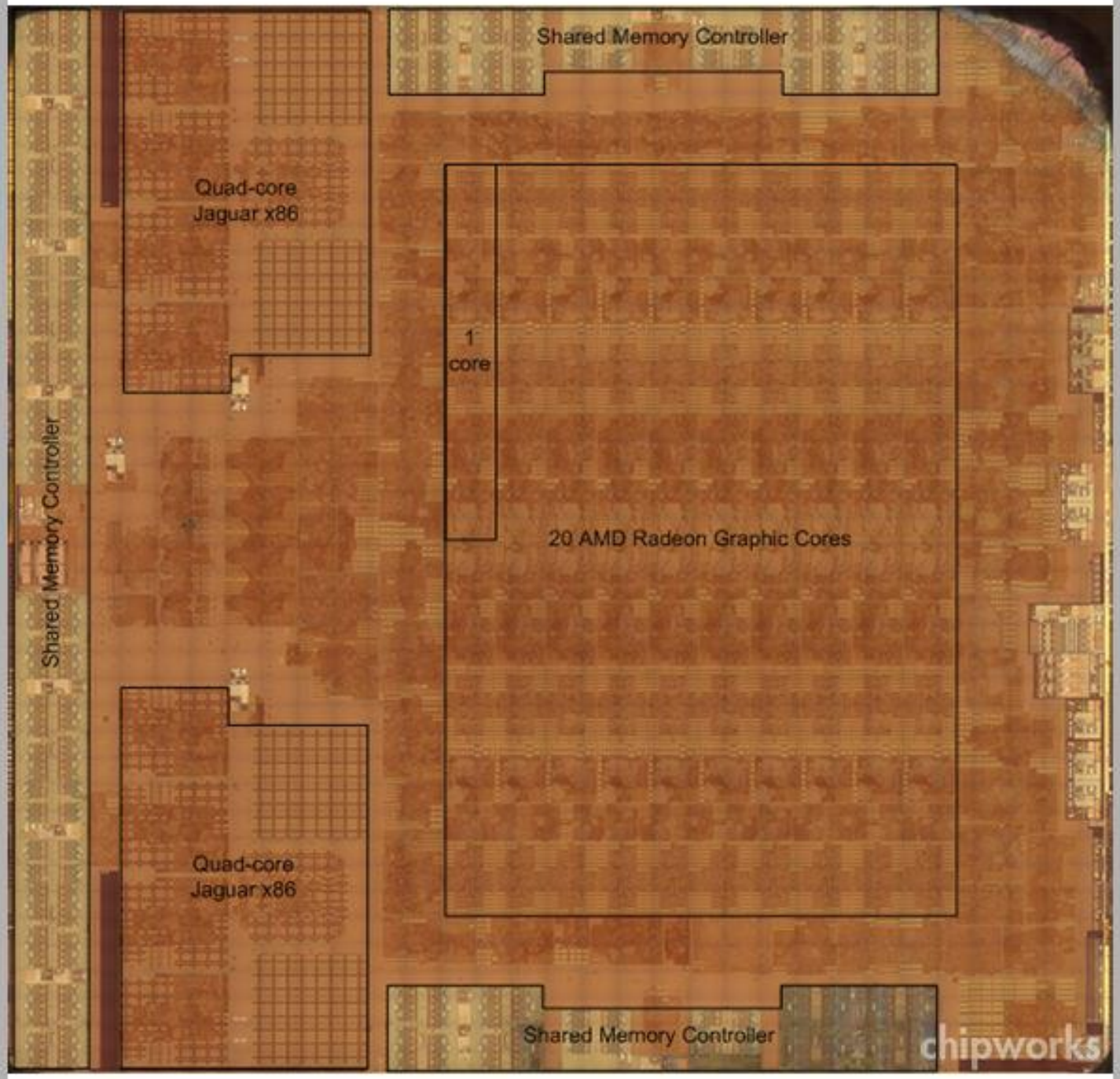
Multicore CPU



Multicore CPU

- e.g., PS4 architecture
  - 2 quad-core Jaguar CPUs
  - Radeon HD 7870(ish) GPU




Shared Memory Controller
Quad-core
Jaguar x86
1
core
Shared Memory Controller
20 AMD Radeon Graphic Cores
Quad-core
Jaguar x86
Shared Memory Controller
chipworks

Manycore GPU (SIMT)

- GPU architecture is a hybrid between SIMD and MIMD
- **SIMD cores** apply a single instruction stream (thread) to multiple data "lanes" in parallel
- To hide latency, many **threads** share a SIMD core via **time-slicing**
  - Nvidia calls this single instruction multiple **thread** (SIM**T**)
- We'll dive into GPU architecture later (time permitting)…

Computer Clusters, Grids and the Cloud

- **Computer cluster**: Multiple homogenous computers, rack-mounted in close proximity, connected via fast local network
- **Grid computing**: Loosely coupled heterogeneous computers (e.g., SETI@home)
- **Cloud computing**: On-demand computing and storage
  - Software as a Service (SaaS)
  - *Insert-flavor-here* as a Service (*X*aaS)
  - Flexible, scalable, cost-effective
    - use only what you need, pay as you go