CPU Microarchitecture

Instruction Set Architecture

The instruction set architecture (ISA) is the contract between the software and the hardware, which defines the rules of communication.

Two main categories:

1) register-based, load-store architectures memory (is accessed only using load and store instructions): ARM/RISC-V
2) register-memory architecture, where operations can be performed on registers and memory operands: x86

Instruction Set Architecture

Basic functions:

load, store, control, integer arithmetic operations

Extensions:

- Floating points (half, single, double precision)
- Number representations for AI (INT8,FP8,BF16)
- vector processing (e.g., Intel AVX2, AVX512, ARM SVE, RISC-V “V” vector extension)
- matrix/tensor instructions (Intel AMX, ARM SME).

Pipeline

Pipelining is a foundational technique used to make CPUs fast, wherein multiple instructions overlap during their execution.

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction x	IF	ID	EXE	MEM	WB				
Instruction x+1		IF	ID	EXE	MEM	WB			
Instruction x+2			IF	ID	EXE	MEM	WB		
Instruction x+3				IF	ID	EXE	MEM	WB	
Instruction x+4					IF	ID	EXE	MEM	WB

Pipeline

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction x	IF	ID	EXE	MEM	WB				
Instruction x+1		IF	ID	EXE	MEM	WB			
Instruction x+2			IF	ID	EXE	MEM	WB		
Instruction x+3				IF	ID	EXE	MEM	WB	
Instruction x+4					IF	ID	EXE	MEM	WB

This is the well-known DLX pipeline. High-end processors have 10-20 pipeline stages.

Pipeline

The **throughput** of a pipelined CPU is defined as the number of instructions that complete and exit the pipeline per unit of time.

The **latency** for any given instruction is the total time through all the stages of the pipeline.

The time required to move an instruction from one stage to the next defines the basic machine **clock cycle** for the CPU.

Pipeline hazards

Structural hazards: are caused by resource conflicts. Can be eliminated by replicating hardware resources, such as execution units, instruction decoders, multi-ported register files.

Control hazards: are caused due to changes in the program flow. They arise from pipelining branches and other instructions that change the program flow.

Pipeline hazards

Data hazards: are caused by data dependencies in the program and are classified into three types.

Read-after-write (RAW)

R1 = R0 ADD 1
R2 = **R1** ADD 2

Can be (partially) solved by using data forwarding/bypassing. The longer the pipeline, the more effective bypassing becomes.

Pipeline hazards

Write-after-read (WAR)

```
R1 = R0 ADD 1
R0 = R2 ADD 2
```

WAR hazards occur only in superscalar/OoO processors. The first instruction reads the **new** value of **R0 i**nstead of the old one.

A WAR hazard is not a true dependency and can be eliminated by a technique called *register renaming*.

Register renaming maps the logical registers defined by the ISA to a large set of physical registers.

Pipeline hazards

Write-after-read (WAR)

WAR hazards occur only in superscalar/OoO processors.

A WAR hazard is not a true dependency and can be eliminated by a technique called *register renaming*.

With register renaming we can reorder/parallelize instructions.

Pipeline hazards

Write-after-Write (WAW)

```
R1 = R0 ADD 1  ;
R2 = R1 SUB R3 ; RAW
R1 = R0 MUL  3 ; WAW + WAR
```

Pipeline hazards

Write-after-Write (WAW)

R1 = R0 ADD 1 ;
R2 = **R1** SUB R3 ; **RAW**
R1 = R0 MUL 3 ; **WAW + WAR**

R101 = R100 ADD 1
R101 SUB R103 ; **RAW**
R104 = R100 MUL 3

WAW hazards are also eliminated by register renaming

Out-Of-Order Execution

Most modern CPUs support **out-of-order (OOO)** execution: sequential instructions can enter the execution stage in any arbitrary order only limited by their dependencies and resource availability.

An instruction is called **retired** when its results are **visible in the architectural state**. To ensure correctness, CPUs must retire all instructions in the program order. This constraint is called *in-order retire*.

Out-Of-Order Execution

Example: Instruction x+2 started executing before instruction x+1.

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Out-Of-Order Execution

Fetch/decode are in order

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Out-Of-Order Execution

Also WB is in order

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Out-Of-Order Execution

Execution is out of order

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Superscalar Engines

Most modern CPUs are superscalar, i.e., they can issue more than one instruction in any given cycle. **Issue width** is the maximum number of instructions that can be issued during the same cycle

Instruction	Clock cycle					
	1	2	3	4	5	6
Instruction x	IF	ID	EXE	MEM	WB	
Instruction x+1	IF	ID	EXE	MEM	WB	
Instruction x+2		IF	ID	EXE	MEM	WB
Instruction x+3		IF	ID	EXE	MEM	WB

2-way superscalar CPU.

Static scheduling

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required. For example
    - One integer instruction (or branch)
    - Two independent floating-point operations
    - Two independent memory references

- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (**VLIW**)

VLIW shortcomings

- Stalls due to branch can significantly affect throughput, in particular for control-heavy code
- VLIW compilers are significantly more complex because they must carefully analyze code to **handle data dependencies** and arrange instructions optimally
- Memory Access Parallelism: VLIW architectures often have limited ability to handle **memory access conflicts**
- VLIW is more susceptible to stalls if an instruction requires an unpredictable amount of time to execute (e.g., due to a cache miss or **memory latency**)

VLIW application domains

- Audio/video processing (e.g encoding/decoding)
- DSP applications (radar, radio communications, 4G/5G)
- Machine learning: Neural network inference, Matrix and tensor operations

Dynamic scheduling

To overcome the problem with static scheduling, modern processors use dynamic scheduling. CPUs use

- the **Tomasulo algorithm** (register renaming) to eliminate false dependencies
- A **Reorder Buffer (ROB)** in which instructions are inserted in order, can execute out of order, and retire in program order. From the ROB, instructions are inserted in the RS.
- In the **Reservation Station (RS)** instructions wait for their input operands and execution unit to become available.

Dependency chains

the Tomasulo algorithm does not resolve RAW dependencies, (also known as a dependency chain)

Dependency chains are often found in loops (*loop carried dependency*) where the current loop iteration depends on the results produced on the previous iteration.

Dependency chains, together with control hazard are the main factor that limits the actual IPC.

Reorder Buffer

- The ROB is a circular buffer that keeps track of the state of each instruction
- It has several hundred entries. The size of the ROB determines how far ahead the hardware can look for scheduling instructions independently.
- Instructions are inserted in the ROB in program order, can execute out of order, and retire in program order.
- Register renaming is done when instructions are placed in the ROB.

Reservation Station

- Instructions are inserted in the RS from the ROB.
- RS entries: Typically, per execution unit type (ALU, Load/Store Unit, FP unit).
- Once instructions are in the RS, they wait for their input operands to become available.
- Instructions from the RS can be executed in any order.
- RS supports speculative execution

Speculative Execution

- Control hazards can cause significant performance loss in a pipeline.
- One technique to avoid this performance loss is hardware branch prediction. A CPU predicts the likely target of branches and starts executing instructions from the predicted path (known as speculative execution).
- If the prediction turns out to be correct, it saves a lot of cycles.
- Otherwise, the results from the speculative execution must be squashed and thrown away. This is called the **branch misprediction penalty**.

Branch Prediction

Consider the example in the box. The CPU should know whether the condition a < b is false or true.

```
if (a < b)
  foo();
else
  bar();
```

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
BRANCH (a<b)	IF	ID	EXE	MEM	WB				
CALL foo				IF	ID	EXE	MEM	WB	
// Instr from foo					IF	ID	EXE	MEM	WB

Branch Prediction

With speculative execution, the CPU guesses an outcome of the branch and initiates processing instructions from the chosen path.

```
if (a < b)
  foo();
else
  bar();
```

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
BRANCH (a<b)	IF	ID	EXE	MEM	WB				
CALL foo		IF*	ID*	EXE	MEM	WB			
// Instr from foo			IF*	ID	EXE	MEM	WB		

Branch Prediction

- If a branch instruction depends on a value loaded from memory, we can save hundreds of cycles.
- An instruction that is executed speculatively is marked as such in the ROB.
- Once it is not speculative any longer, it can retire in program order. Here is where the architectural state is committed, and architectural registers are updated.
- The use of ROB allows an easy roll back when a misprediction happens.

Branch Prediction

Modern CPUs employ sophisticated dynamic branch prediction mechanisms that provide very high accuracy. There are three types of branches:

Unconditional jumps and direct calls: they are always taken

Conditional branches: they have two potential outcomes: taken or not taken.

- *Forward conditional* branches are usually generated for *if-else* statements, which have a high chance of not being taken, as they frequently represent error-checking code.
- *Backward conditional* jumps are frequently seen in loops and are usually taken.

Indirect calls and jumps: they have many targets. An indirect jump or indirect call can be generated for a switch statement, a function pointer, or a virtual function call.

Branch Prediction

All prediction mechanisms try to exploit two important principles:

1. **Temporal correlation**: the way a branch resolves may be a good predictor of the way it will resolve at the next execution. This is also known as local correlation.

2. **Spatial correlation:** several adjacent branches may resolve in a highly correlated manner (a preferred path of execution). This is also known as global correlation.

The best accuracy is often achieved by leveraging local and global correlations together.

Branch Prediction

Most prediction algorithms are based on previous outcomes of the branch.

The branch prediction unit (BPU) is composed of:

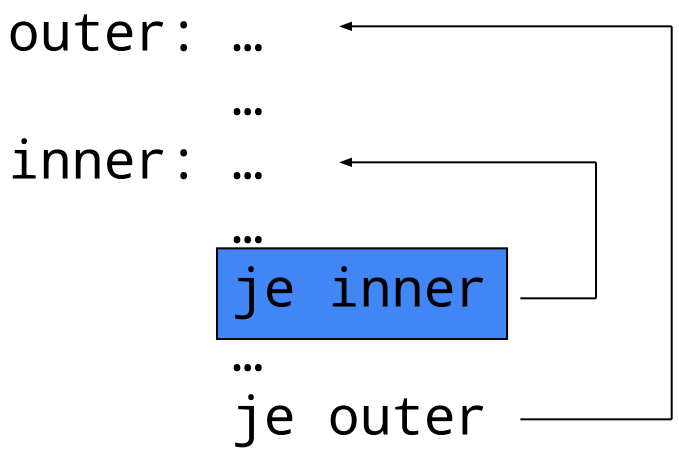
- A branch target buffer (BTB), which caches the target addresses for every taken branch

- Pattern History Table (PHT): Stores **past branch behavior** to predict future outcomes. Often implemented with **2-bit counters**

- **Return Address Stack (RAS): u**sed for predicting the **return address** of function calls.

1-Bit Predictor: Shortcoming

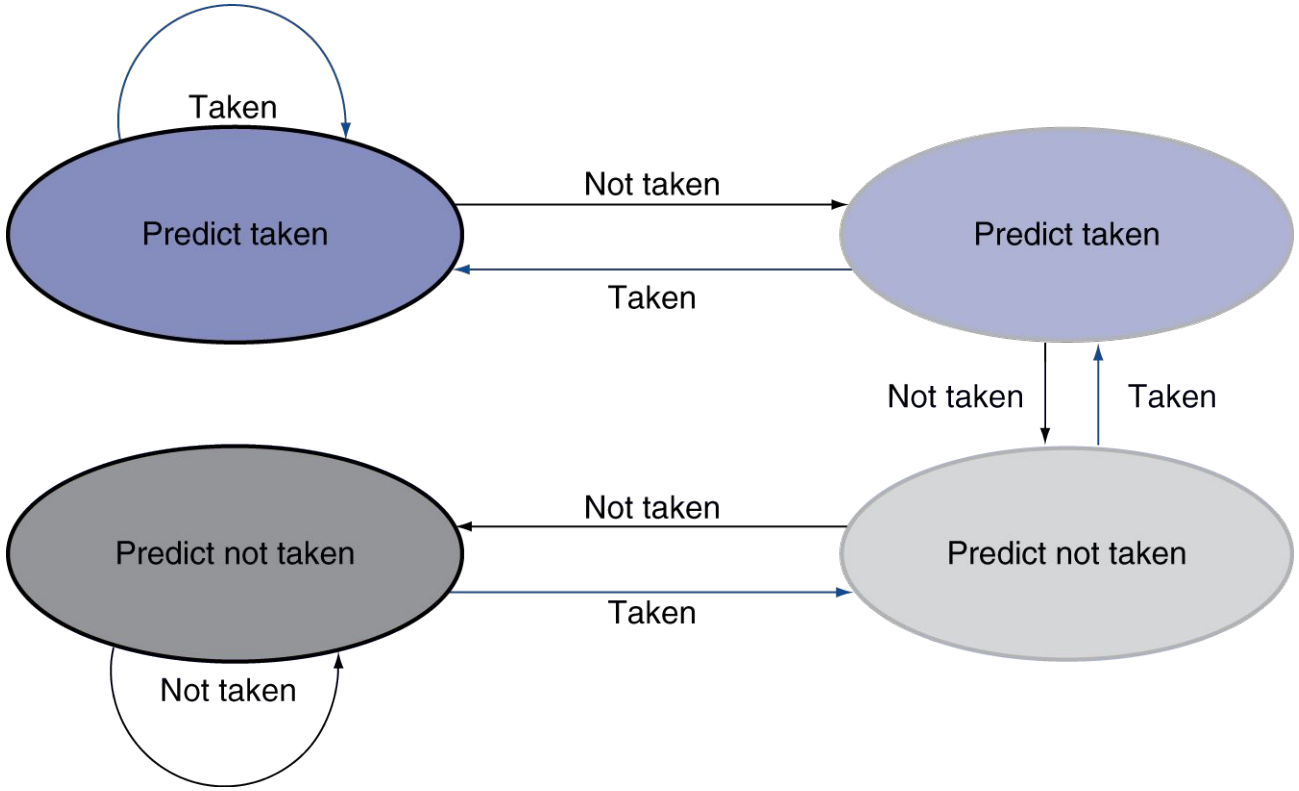
Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

Only change prediction on two successive mispredictions



Break

Thread-Level Parallelism

CPUs support techniques to exploit parallelism across processes and/or threads executing on a CPU. There are three main techniques to exploit Thread-Level Parallelism:

- multicore systems
- simultaneous multithreading
- hybrid architectures

Multicore Systems

- After 2000 the GHz race slowed down and designers had to focus on other innovations to improve CPU performance.
- The idea of Multicore design is to replicate multiple processor cores on a single chip and let them serve different programs at the same time.
- The first consumer-focused dual-core processor was the Intel Core 2 Duo, released in 2005, which was followed by the AMD Athlon X2 architecture released later that same year.
- Nowadays, high-end laptops contain more than ten physical cores and server processors contain more than 100 cores on a single socket.

Multicore Systems

- Since each core generates heat when it’s working and safely dissipating that heat remains a challenge.
- Multicore processors usually reduce clock speeds due to this heat dissipation wall.
- This is one of the reasons you can see server chips with a large number of cores having much **lower frequencies** than processors that go into laptops and desktops.

Multicore Systems

- Cores in a multicore system are connected to each other and to shared resources, such as last-level cache and memory controllers.

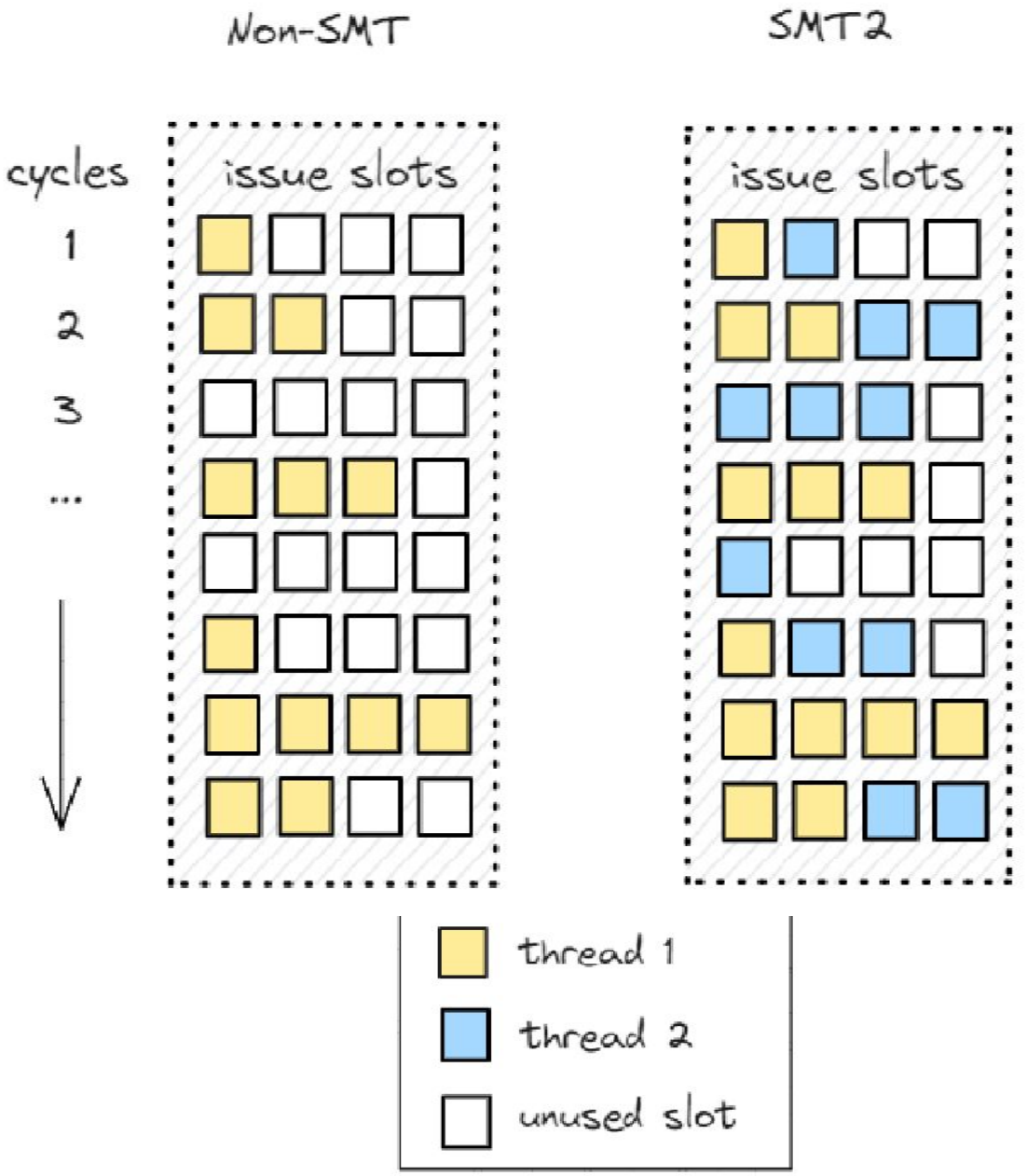
- This results in diminishing returns to performance as cores are added, unless you also address the throughput of other shared resources, e.g., interconnect bandwidth, last-level cache size and bandwidth, and memory bandwidth.

- Shared resources frequently become the source of **performance issues** in a multicore system.

Simultaneous Multithreading

SMT allows multiple software threads to run simultaneously (in the same clock cycle) on the same physical core using shared resources.

- To support SMT, a CPU must replicate the architectural state (program counter, registers) to maintain thread context. Other CPU resources can be shared.



Simultaneous Multithreading

- The most likely SMT penalty is caused by competition for L1 and L2 caches. Since they are shared between two logical cores, they could lack space in caches and force eviction of the data that will be used by another thread in the future.

- SMT brings a considerable burden on software developers as it makes it harder to **predict and measure the performance** of an application that runs on an SMT core.

- There is also a security concern. Researchers showed that some earlier implementations had a vulnerability through which one application could steal critical information (like cryptographic keys) from another application that runs on the sibling logical core

Hybrid Architectures

- Computer architects also developed a hybrid CPU design in which two (or more) types of cores are put in the same processor. Typically, more powerful cores are coupled with relatively slower cores to address different goals.

- The first mainstream hybrid architecture was Arm's big.LITTLE, which was introduced in October 2011. Other examples are the Apple M1 with four high-performance "Firestorm" and four energy-efficient "Icestorm" cores. Intel introduced its Alder Lake hybrid architecture in 2021 with eight P- and eight E-cores in the top configuration.

Hybrid Architectures

Optimal scheduling for hybrid architectures becomes challenging. Here are a few considerations for optimal scheduling:

- Leverage small cores to conserve power. Do not wake up big cores for background work.
- Recognize candidates (low importance, low IPC) for offloading to smaller cores. Similarly, promote high importance, high IPC tasks to big cores.
- When assigning a new task, use an idle big core first. In the case of SMT, use big cores with both logical threads idle. After that, use idle small cores. After that, use sibling logical threads of big cores.

Performance comparison

Feature	P-core (Performance)	E-core (Efficiency)
Target workload	High single-thread performance, latency-sensitive tasks, high IPC	Background tasks, low-priority work, throughput-oriented tasks
Base frequency	~2.5–3.0 GHz	~1.8–2.0 GHz
Turbo frequency	Up to ~4.8–5.0 GHz	Up to ~3.6–3.8 GHz
Instructions per cycle (IPC)	High (~2× E-core for typical workloads)	Lower (~50–60% of P-core)
Power consumption (per core)	Higher, ~10–15 W under load	Lower, ~2–5 W under load
Thermal output	High; contributes more to package power	Low; more thermally efficient
SMT support	Yes (2 threads per core)	No SMT
Area per core	Larger (more transistors, out-of-order execution)	Smaller (simpler pipeline)
Use case	Performance-critical applications (gaming, scientific computing, responsiveness)	Background tasks, lightweight workloads, always-on services

Memory Hierarchy

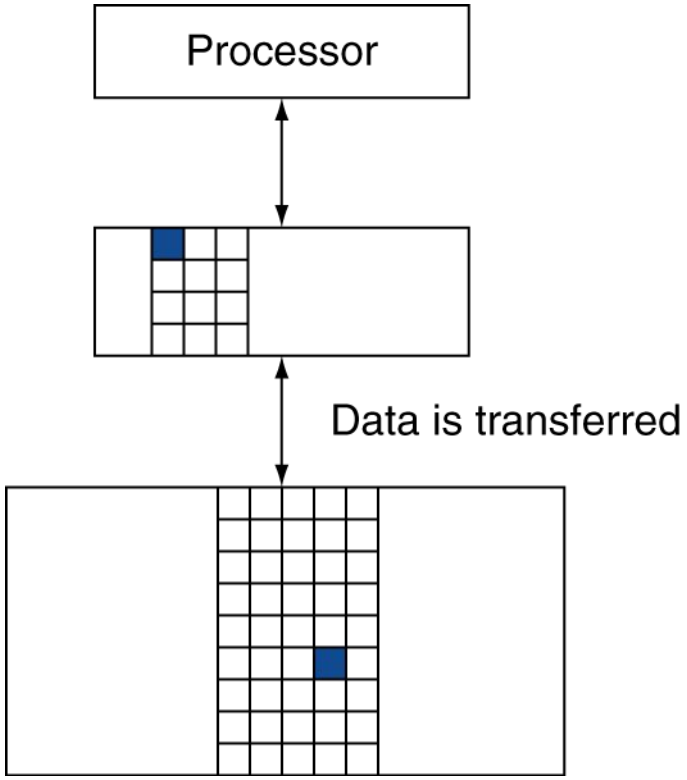
Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - e.g., sequential instruction access, array data

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
    - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
    - Cache memory attached to CPU

Memory Hierarchy Levels



- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

Memory Technology

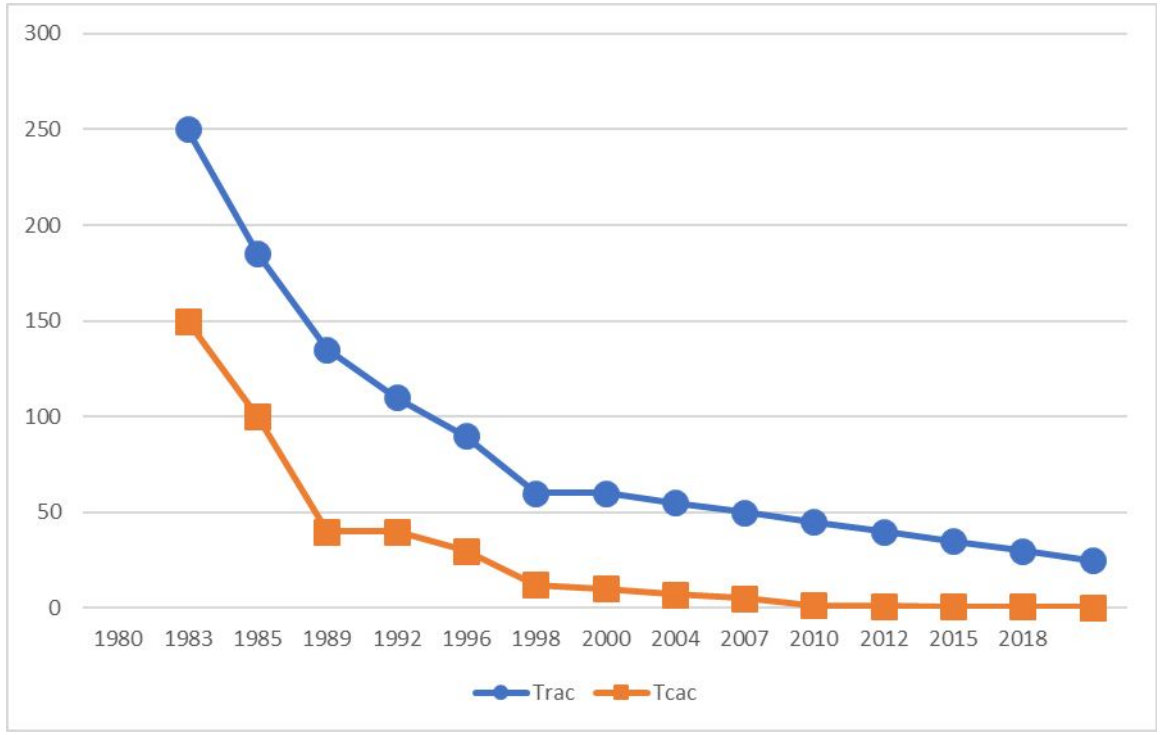
- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $500 – $1000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $3 – $6 per GB
- Magnetic disk
  - 5ms – 20ms, $0.01 – $0.02 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

DRAM Technology

- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM “row”
- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges

DRAM Generations

Year	Capacity	$/GB
1980	64 Kibibit	$6,480,000
1983	256 Kibibit	$1,980,000
1985	1 Mebibit	$720,000
1989	4 Mebibit	$128,000
1992	16 Mebibit	$30,000
1996	64 Mebibit	$9,000
1998	128 Mebibit	$900
2000	256 Mebibit	$840
2004	512 Mebibit	$150
2007	1 Gibibit	$40
2010	2 Gibibit	$13
2012	4 Gibibit	$5
2015	8 Gibibit	$7
2018	16 Gibibit	$6



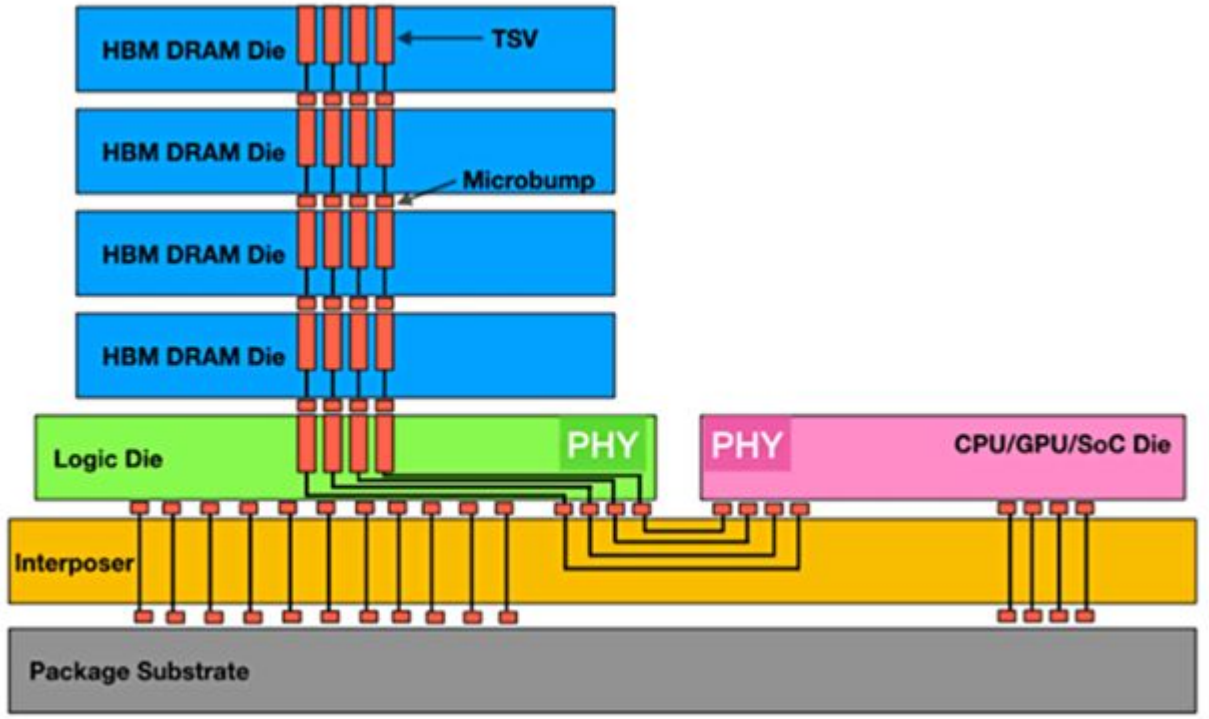
DRAM Performance Factors

- Row buffer
  - Allows several words to be read and refreshed in parallel
- Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth
- DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth

High Bandwidth Memory

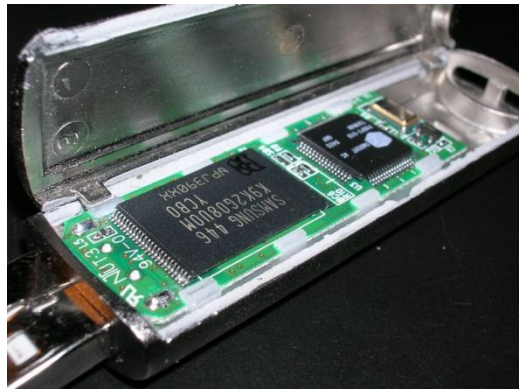
- *HBM (High Bandwidth Memory)* is a new type of CPU/GPU memory that vertically stacks memory chips. HBM drastically shortens the distance data needs to travel to reach a processor.

- HBM memory bus is very wide: **1024** bits for each HBM stack. This enables HBM to achieve ultra-high bandwidth. The latest HBM3 standard supports up to **665** GB/s bandwidth per package. It also operates at a low frequency of **500** MHz and has a memory density of up to **48GB** per package



Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)



Disk Storage

- Nonvolatile, rotating magnetic storage