Module 1

Introduction and overview

- Review of basic computer architecture, Technology trend and design goals, Performance metrics and performance enhancement techniques, Power dissipation in processors, power metrics, and low-power design techniques.

Design goals for high performance

- Function requirements: process data according to instructions
    1. Tests and verifications are required at different stages since it is not possible to modify the processor once fabricated.
- Reliability: should continue to perform correctly
    1. Important for all modern computers and more important for mission-critical application
    2. Reliable and fault-tolerant computing is an important area of study, which considers various approaches to improve fault tolerance and reliability.
- Cost: a very important factor
    1. Embedded consumer products are particularly highly sensitive to cost
- Performance: is a basic requirement
    1. A computing system needs to provide the desired performance.
- Power consumption: is a very important requirement nowadays
    1. A system needs to consume less power for many reasons: battery life, cost of electricity, thermal problem, and cooling cost.
- **Performance** and **power consumption**: The evolution of computer architecture is driven to improve these two goals.

Iron law

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ time = IC \times CPI \times T$$

- A decrease in one may lead to an increase in the other two

Speed up of a processor

$$\text{Speed up} = \frac{Time\ original}{T\ enhanced} = \frac{1}{[(1-E)+E/S]}$$

Amdahl's law

Speedup via parallelism is limited by that component of the application that cannot be enhanced; in a program with parallel processing, a relatively few instructions that have to be performed in sequence will have a limiting factor on program speedup such that adding more processors may not make the program run faster.

Gustafson’s law

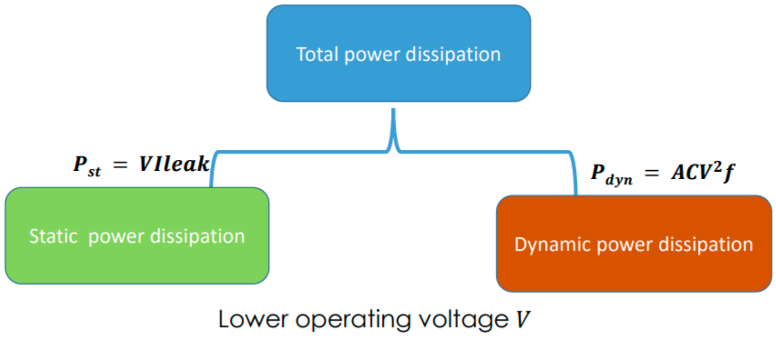
Completing more tasks in a given amount of time using more processors. Run each task on one processor, but do these task together❤️

MIPS as a performance metric

A million instructions per second (MIPS) is an approximate measure of a computer's raw processing power.

Power dissipation in processor

Power metrics (Dynamic and static power)



A: switch factor, C: capacitance, V=voltage, f=clock frequency

Low power design techniques

- Efficient cache and memory hierarchy design
- Power gating: shutting down the unused components(enable signals)
- Clock gating: to reduce unnecessary switching
- Reducing the data movement, number of memory access, and register transfer

Module 2: Instruction set architecture design

Introduction to ISA

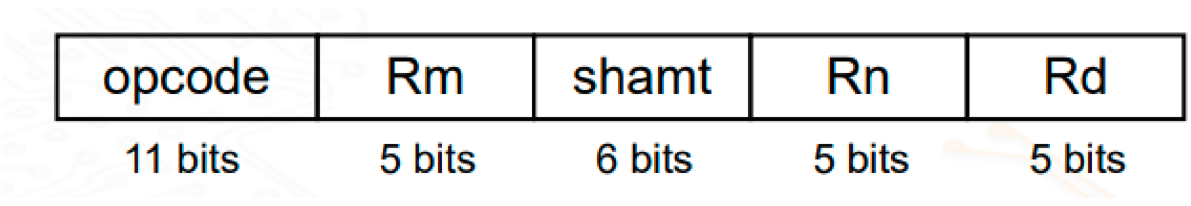
A set of following specifications that a programmer must know to write a correct and efficient program for a specific machine.

- Instruction format
- Length of instruction and size of the field
- Word size: 16-bit, 32-bit or 64-bit
- Set of all operations: opcodes/machine language
- Register file specification: size, width, and its usage of registers in CPU
- Memory address space and addressability: no of addressable locations & bits per location
- Addressing modes: ways of specifying and accessing operands(s): indicate how and address is determined
- Operand locations: all in registers, register and memory or all in memory

ARMv8 ISA: A design example named LEGv8

Instructions classified based on functionality

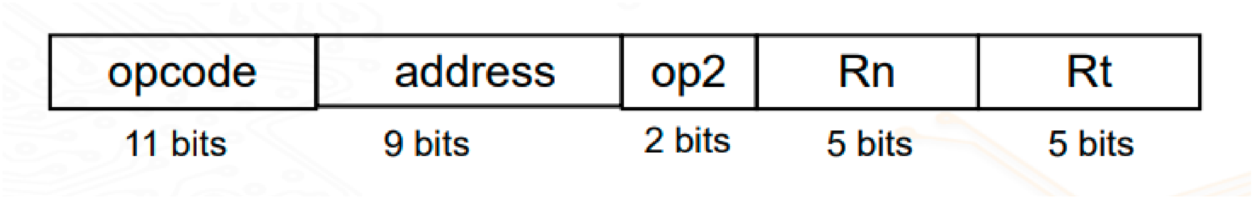
- **Register(R) Type:** fetching the data from two register and perform an operation, store the result to the new register



opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

  - ➢ All data values are located in the register
  - ➢ **Rn & Rm:** specify the first and the second source register
  - ➢ **Rd:** specifies the destination register
  - ➢ **Shamt** stands for shift amount: specifies the number of bit positions to be shifted
  - ➢ **Opcode:** type of instruction

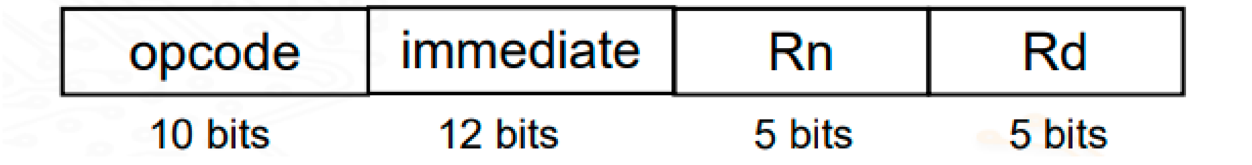
- **Data Transfer(D) Type(Load/Store):** load: data will be fetched from one location to register/ store: data be fetched then store in another location with the help of registers.



opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- ➢ **Rn:** specify the base registers
- ➢ **Address:** constant offset from contents of base register
- ➢ **Rt:** specifies the destination register (load) or as the **source**(store) register number(contain data to be stored into **Rn +address**)
  - ★ ALU will calculate the address(address +[Rn])
  - ★ Read the Data at memory location(address + [Rn])

- **Immediate (I) Type:** add an immediate number with **Rn** and save it into **Rd**

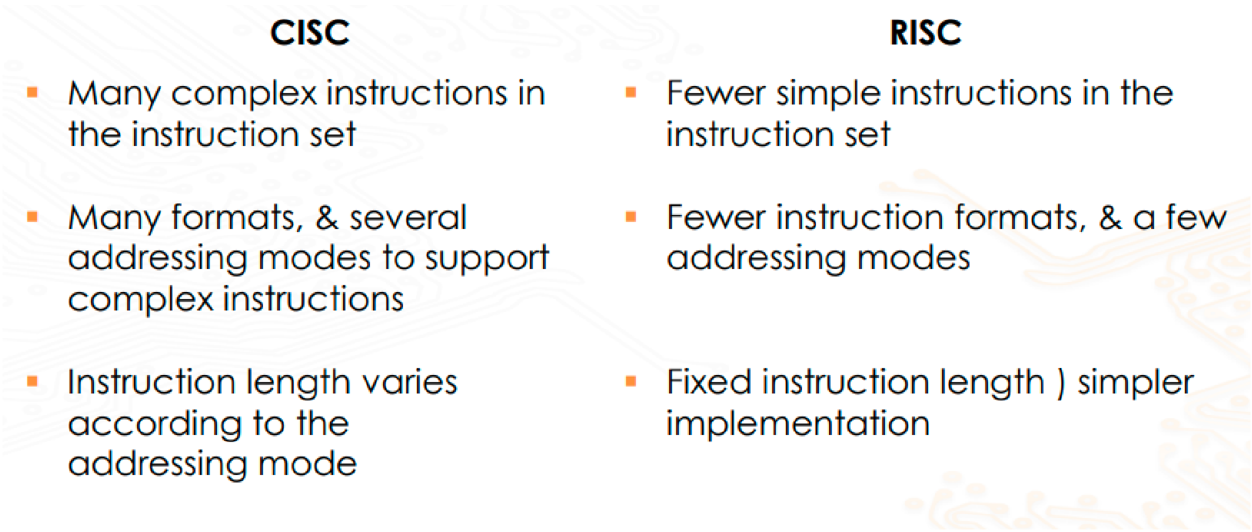


opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

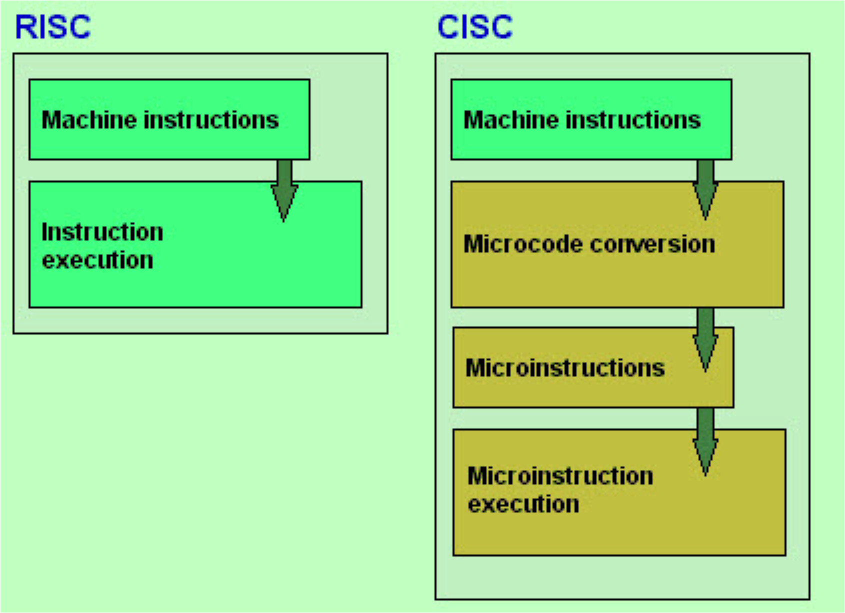
- ➢ **Rn**: Source register
- ➢ **Rd**: destination register

- **Conditional instructions:** branch to a labeled instruction if a condition is true, otherwise just continue
  1. **CBZ register, L1 :** if(register ==0) branch to instruction labelled L1
  2. **CBNZ register, L1 :** if(register !=0) branch to instruction labelled L1
  3. **B L1:** branch unconditionally to instruction labelled L1

Relative advantages of RISC and CISC instruction set



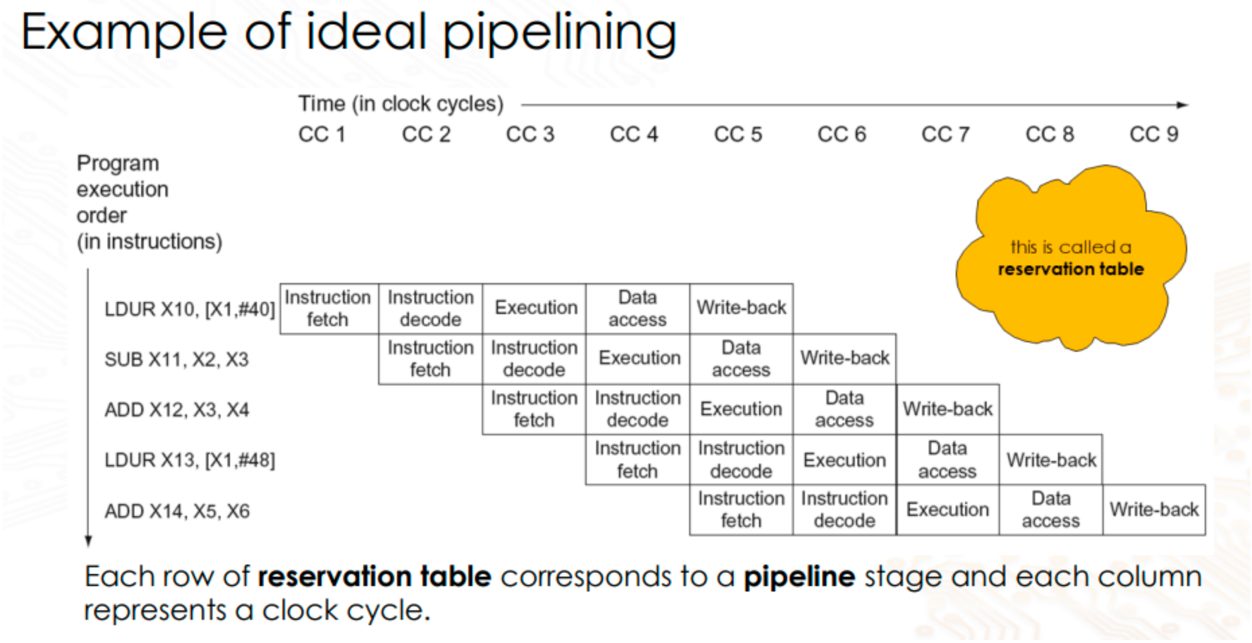
CISC	RISC
▪ Many complex instructions in the instruction set	▪ Fewer simple instructions in the instruction set
▪ Many formats, & several addressing modes to support complex instructions	▪ Fewer instruction formats, & a few addressing modes
▪ Instruction length varies according to the addressing mode	▪ Fixed instruction length) simpler implementation



Module 4: Pipelining and Instruction Level Parallelism

Visualization of pipeline architecture

Fetch Decode Execute Memory access Write-back



Example of ideal pipelining

Time (in clock cycles)

Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
LDUR X10, [X1,#40]	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
SUB X11, X2, X3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
ADD X12, X3, X4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
LDUR X13, [X1,#48]				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
ADD X14, X5, X6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

this is called a **reservation table**

Each row of **reservation table** corresponds to a **pipeline** stage and each column represents a clock cycle.

Challenges in the realization of pipeline architecture

- **Ideal pipeline**
  1. Identical task of all instructions
  2. Uniform decomposition of task
  3. Independent Computations

- **Real pipeline**
    1. Except for load instruction, all other instructions do not need all stages
    2. Not all pipeline stages involve, using the same time to complete their respective sub-task
    3. The execution of one instruction may depend on one of the preceding instructions
- **Complications**
    1. Datapath
        a. Many instructions in flight
    2. Control
        a. Must correspond to multiple instructions
    3. Instructions may have
        a. Data and control flow dependencies
        b. One may have to stall and wait for another

Pipeline Hazards

- **Stalling the pipeline will result in a drop in efficiency**
- **Structural hazard:** if two instruction needs the same piece of hardware resource(eg memory) at the same time, a stall will happen, one instruction needs to wait for at least one clock cycle.
    - **Solution:** get additional hardware elements

- **Data hazard:** either the source or destination register of instruction is not available at the time expected by the pipeline, causing a stall
    - **Solution**: know the dependencies between data in the program and eliminate them

- **Control/Instruction hazard:** conditional jumps or jumps or subroutine calls can stall a pipeline because of the delay in the availability of instruction.
    - **Solution:** Need to predict the location of branches to avoid stalling

Data-dependence and Handling data-dependence

- Types of data dependence
    - **Flow dependence >** RAW: read after write
        - Detect and wait until the value is available in register file
        - Detect and bypass data to dependent instruction
        - Detect and eliminate the dependence at the software level
        - Do something else
    - **Output dependence >** WAW: write after write
    - **Anti dependence >** WAR: write after read

Out of order execution and dynamic scheduling

- In-order instruction execution
  - Instructions are fetched, executed & completed in compiler-generated order
  - One stall, they all stall
  - Instructions are statically scheduled
- Out-of-order instruction execution
  - Instructions are fetched in compiler-generated order, but they may be executed in some other order
  - Independent instructions behind a stalled instruction may pass it
  - Instructions are dynamically scheduled during run time
- Dynamic scheduling
  - If there are no structural or data hazards(hardware or register conflicts), the instruction can bypass and execute its instruction first.

Instruction reordering and Register renaming

- Reorder instructions to maximise efficiency
- Register rename: need more registers

Loop unrolling

Change loops into a longer code sequence

- Greater demand for registers
- Minimize structural hazards and data hazards

Control hazards handling

When branching, the pipeline will have to stall as it changes the sequence of instructions to be executed

- Instruction fetched previously by the instruction fetch unit needs to be discarded and replaced with the new instruction resulting in stall cycles.
- Time lost due to branching is referred as **branch penalty**
  - **Solution:** stall the pipeline immediately after we fetch a branch instruction; **3 stalls** when Branch Target Address is updated in the Memory stage.

Static and dynamic branch prediction

- **Branch prediction**: assume the branch does not happen and continue executing the instructions in the pipeline. If the branch does occur, the pipeline must be flushed of the speculative instructions.
  - Static branch prediction: when behavior is highly predictable

        - Branch always taken: make sense for if pipeline location is known before the branch outcome > the pipeline will execute the instructions in the branch while waiting for the outcome
        - Branch always not taken: speculation
    - Dynamic branch prediction: prediction decision may change depending on the execution history

<u>Instruction level parallelism</u>

Parallel execution of instructions requires 3 major task

- Check for data dependence between instructions to see if the instructions can be grouped together for parallel execution
- Assigning instructions to different functional units in the processor
- Determine when the instruction is to be executed
- Methods to extract more parallelism
    - Instruction reordering and out of order execution: change the order of execution of instruction if it does not violate the data dependence
    - Speculative execution with dynamic scheduling: to execute an instruction without exactly knowing if that needs to be executed: ahead of branch outcome (do first then check)
    - Loop unrolling

<u>Concepts of superscalar processing</u>

(down to each core, maximize the use of every functional unit in the core)

- Multiple Instruction fetch in parallel
- Multiple instructions decode in parallel
- Requirements:
    - Multiple functional units in **a CPU** are used
    - More than 1 independent instruction needs to be available to be executed
    - Use specialised hardware to issue independent instructions that can be executed simultaneously

<u>VLIW architecture (advantages and disadvantages)</u>

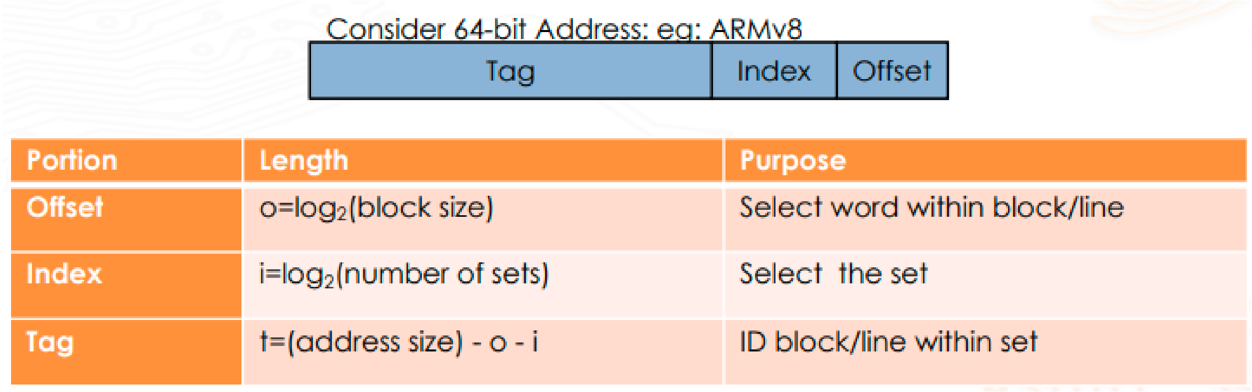
- Packs multiple independent operations into one instruction.
- VLIW processor needs a compiler to break the program instructions down into basic operations that can be performed by the processor in parallel
- These operations are put into a very long instruction word which the processor gets executed in appropriate FUs.
- There must be enough parallelism in the program code to fill the available slots, to reduce the CPI significantly
- **Advantage:**

    - The compiler will take care of all the scheduling and plans for execution
- **Disadvantage:**
    - Compatibility across implementations is a major problem.
        - Cannot run properly with different number of FU or FUs with different latencies
        - Unscheduled events may stall the entire processor

Module 5: Memory Systems

Cache design

- Level 1 cache: small, resides in the processor
- Level 2 cache: bigger than L1, resides between CPU and main memory, is slower than L1 cache.
- Needs to convert the main memory address into a cache location.



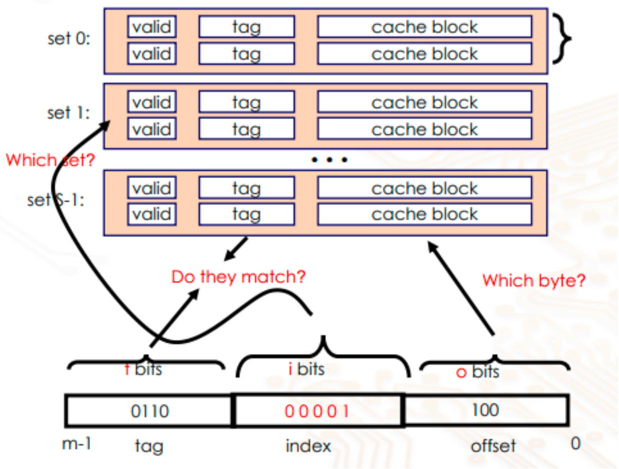
Consider 64-bit Address: eg: ARMv8

Portion	Length	Purpose
Offset	o=log$_2$(block size)	Select word within block/line
Index	i=log$_2$(number of sets)	Select the set
Tag	t=(address size) - o - i	ID block/line within set

- Placement policy: **cache organization schemes**
    - Direct-mapped cache
        - Simplest and fastest address mapping
        - A memory block is mapped to a specific cache line only

- Cache consists of an array of fixed size frames called cache line
- Each frame holds a block of main memory
- Direct mapping: each memory block is mapped to a specific cache line
- A set of memory blocks with the same cache index are mapped to the same cache line.
- Cache size (roughly): C - BS x S data bytes
  - Index in the address will be used to locate the **set** in the cache
  - Tag in the address will be used to check if the cache set is valid(cache hit) (if tag are equal)
  - Offset in the address be used to find the block in that cache line

- Fully associative cache
  - A memory block can be placed in any of the cache lines.
  - Flexible but expensive
    - Need to search all the cache
    - Search the requested block tag with every tag in the cache to find the desired block
    - Main memory only need to be partitioned as tag and offset
    - Expensive in terms of speed and complexity
- Set-associative
  - Combines the strategy of fully associative and direct-mapped caches
  - Similar to direct mapping but more lines per set



Cache replacement and write policies

Replacement: Choosing which cache line to replace with new memory data

- Several policies:
    - FIFO
    - LRU(least recently used): takes care of temporal locality
        - Needs to keep a history of access and hence slows the system down
    - NMRU(Not most recently used)
    - Pseudo-random

Write policies

- Memory hierarchy
    - Two or more copies on the same block
    - Main memory or disk
    - Cache
- What to do on a write
    - All copies need to be changed
    - The cache must propagate to all levels of the memory hierarchy
- **Write Through:**
    - Write propagates directly through the hierarchy (L1, L2, memory, disk)
    - High bandwidth requirement
    - Memory becomes slow when its size increases
    - Popular in the system only for writing to the L2 (beyond L2, use write back)
    - Defeats the purpose of having a cache which is to reduce access to the main memory
- **Write Back:**
    - Maintain state of each line in a cache
        - Invalid = not present in the cache
        - Clean = present, but not written
        - Dirty = present and written
    - Store state in tag array, next to address tag
        - Mark dirty bit on a write
    - On eviction/when the cache line is going to be cast out
        - If the dirty bit is set, write back the dirty bit to the next level
- Complications in write policies
    - More stale copies in the lower level of the hierarchy
    - Must always check higher for dirty copies before accessing copy in a lower level, because dirty copies are being updated in the higher level before lower-level cast out.

    - Cache coherence problem in multiprocessors
    - I/O devices that use direct memory access have to check the cache first before reading the main memory
- Instruction cache(better to have it apart from data cache)
    - Stores instructions only.
    - Tends to have high locality and thus a high hit rate

<u>Cache performance analysis</u>
Cache

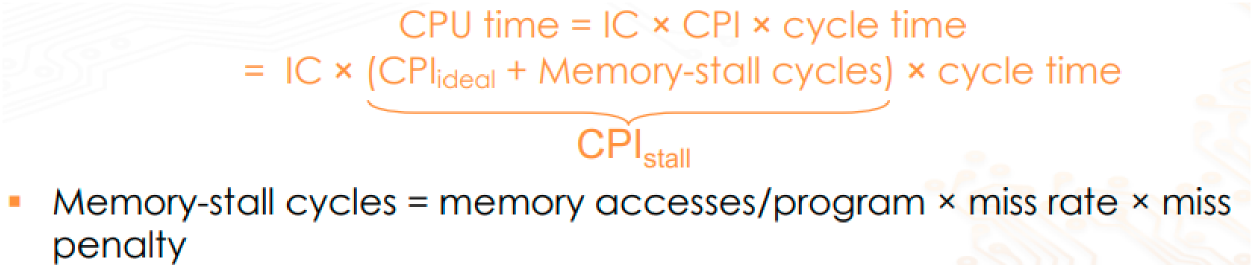
- Cache hit: life is made easy and improves performance
- Cache miss: fetch from next level

What is a performance impact?

- Miss penalty leads to a reduction in performance
    - Detect miss: 1 or more cycles
    - Find victim(line to be replaced)
        - Write back if the victim is dirty
    - Request line from next level: several cycles
    - Transfer line from next level
    - Fill line into data array, update tag array
    - Resume execeution
- A cache miss is determined by
    - Temporal locality
    - Spatial locality
    - Cache organization

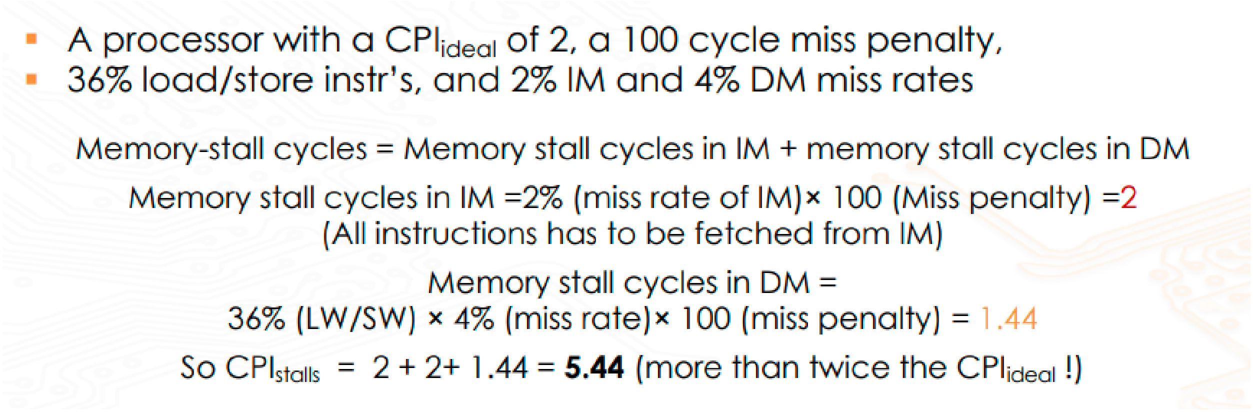
- Reasons for a cache miss
    - Compulsory miss
        - First-ever reference to a given block of memory
    - Capacity miss
        - Working set exceeds cache capacity
        - Useful blocks (with future references) displaced
    - Conflict miss
        - Placement restrictions (not fully assoc). Cause useful blocks to be displaced
- Cache miss effects
    - More block is better, since greater capacity, fewer conflicts
    - Associativity
        - Higher associativity reduces conflicts
        - Very little benefits beyond 8 way set associative

- Block size
  - Larger blocks exploit spatial locality
  - Usually, the miss rate improves until 64B-256B
  - 512B or more miss rates get worse
    - Fewer placement choices: more conflict misses
    - High miss penalty



$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{cycle time}$$
$$= \text{IC} \times \underbrace{(\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles})}_{\text{CPI}_{\text{stall}}} \times \text{cycle time}$$

- Memory-stall cycles = memory accesses/program × miss rate × miss penalty



- A processor with a $\text{CPI}_{\text{ideal}}$ of 2, a 100 cycle miss penalty,
- 36% load/store instr's, and 2% IM and 4% DM miss rates

Memory-stall cycles = Memory stall cycles in IM + memory stall cycles in DM

Memory stall cycles in IM =2% (miss rate of IM)× 100 (Miss penalty) =2
(All instructions has to be fetched from IM)

Memory stall cycles in DM =
36% (LW/SW) × 4% (miss rate)× 100 (miss penalty) = 1.44

So $\text{CPI}_{\text{stalls}}$ = 2 + 2+ 1.44 = **5.44** (more than twice the $\text{CPI}_{\text{ideal}}$!)

Module 6:GPU Architecture and CUDA Programming

Motivation for GPU programming

GPUs are designed to accelerate 3D graphics renderings on PC
GPUs are also used extensively for many mathematical and scientific purposes

Evolution of GPU architectures

Before GPUs, graphics display is based on a simple framebuffer subsystem standard known as the VGA
The original design of the graphics processing unit

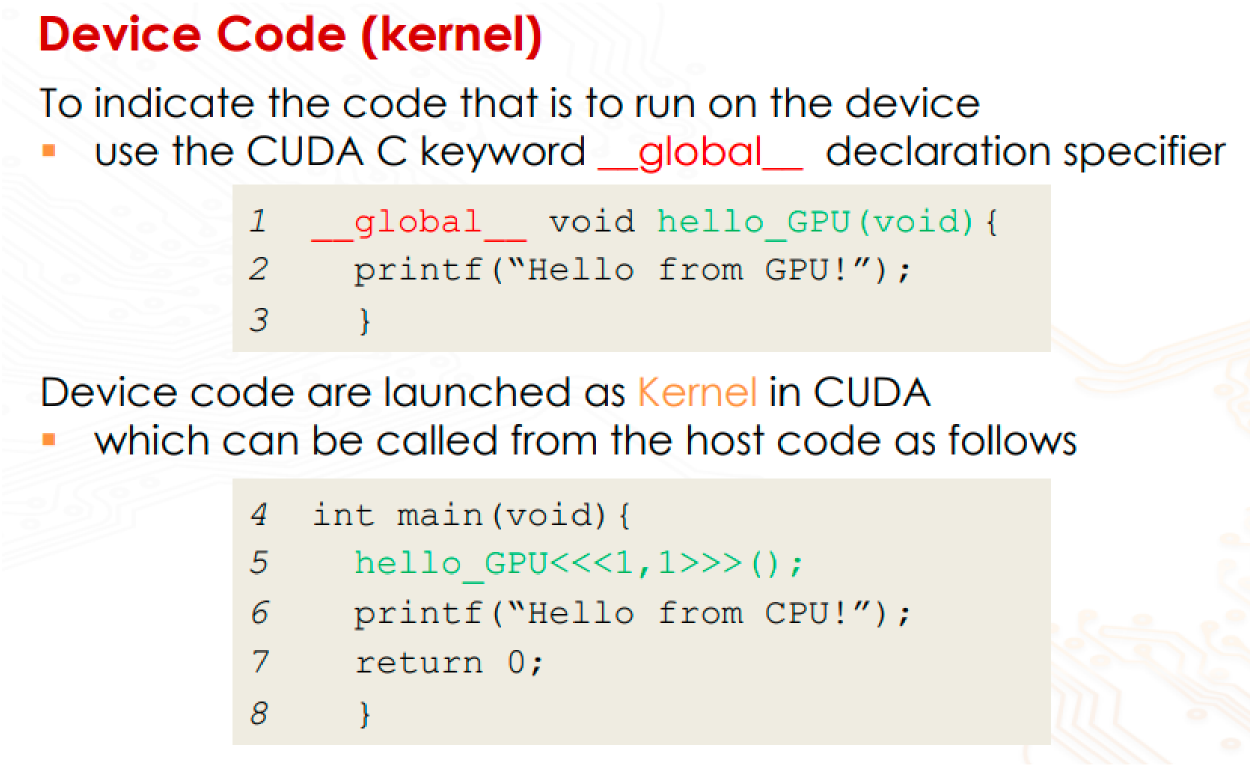
- Based on fixed-function pipelines
- Configurable but not programmable
- Inflexible

Programming on GPU (CUDA)

Nvidia released CUDA(compute Unified Device Architecture)

- A software platform that gives direct access to GPU’s internal processing units to support parallel programming
- General-purpose parallel computing programming model where a problem is first expressed in the form of multiple threads
    - And execute in parallel using CUDA cores in the CPU
    - Enable many complex computational problems to be solved in a very efficient way
        - First decomposed into subproblems each of which is allocated to a “thread block”
        - Each subproblem is then separated into finer pieces that can be solved cooperatively in parallel by using multiple threads within the thread block
- CUDA Programming Model
    - CUDA operates on a heterogeneous programming model that consists of a host and a device
        - Typically a CPU and its memory(host)
        - The device is GPU and its memory(device)
    - A program will start and be run by the host
        - Which then launches the parallel threads that are executed on the device

<u>Basic internal operations of GPUs</u>



Device Code (kernel)

To indicate the code that is to run on the device

- use the CUDA C keyword __global__ declaration specifier

```
__global__ void hello_GPU(void){
  printf(“Hello from GPU!”);
  }
```

Device code are launched as Kernel in CUDA

- which can be called from the host code as follows

```
int main(void){
  hello_GPU<<<1,1>>>();
  printf(“Hello from CPU!”);
  return 0;
  }
```

- During compilation, the source file will be split into host and device components
- <<<x,y>>>: x means the number of thread blocks, y means the number of threads in every block, so xy amount of parallel threads currently concurrently
- **Synchronization between host:**
  - We need the host to finish the kernel execution using CUDA function
    - **cudadeviceSynchronize()**
    - **cudaMalloc() #allocate memory**
    - **cudamemcpy() #sending data to device**
    - **cudaMemcpyHostToDevice**
    - **cudaMemcpyDeviceToHost**
- **__syncthreads() : synchronize between Threads**

- Block vs Threads
  - Not ideal to have 1 thread per block or only one block with multiple threads

Module 7: Data and Thread Level Parallelism

Three levels of parallelism

- Instruction-Level parallelism
    - Multiple independent instructions are identified and grouped to be executed concurrently in different functional units in a single processor
    - Can reduce CPI to values less than 1
    - Eg. Superscalar and VLIW processor
- Data-level parallelism
    - The same operation is performed on multiple data values concurrently in multiple processing units. Can reduce the Instruction Count to enhance performance. Examples: vector processors and array processors
- Thread/Task Level parallelism
    - More than one independent thread/task is executed simultaneously.
    - Can reduce the total execution time of multiple tasks

Data-level parallelism and SIMD processors

Data level parallelism

- Divide the parts of data between different tasks and perform the tasks in parallel
- **Key:** no dependencies between tasks that cause their results to be ordered
- Eg. Fig A
- Eg. Vector addition

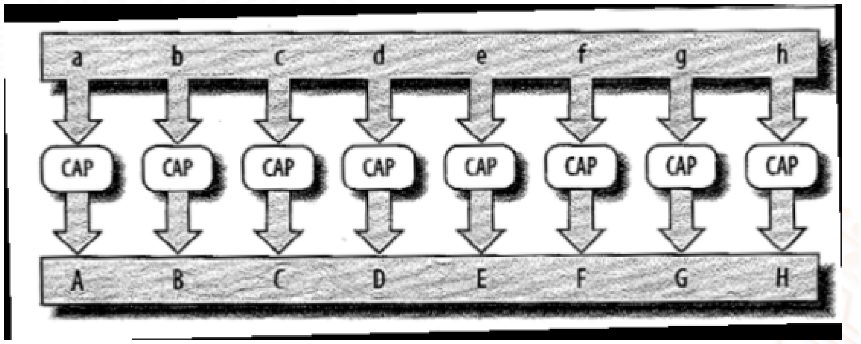
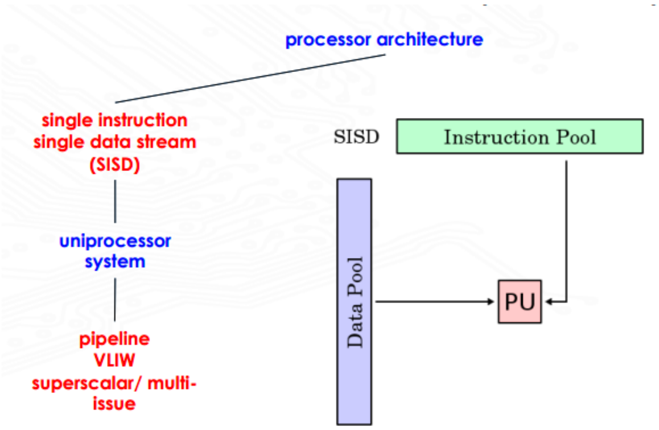
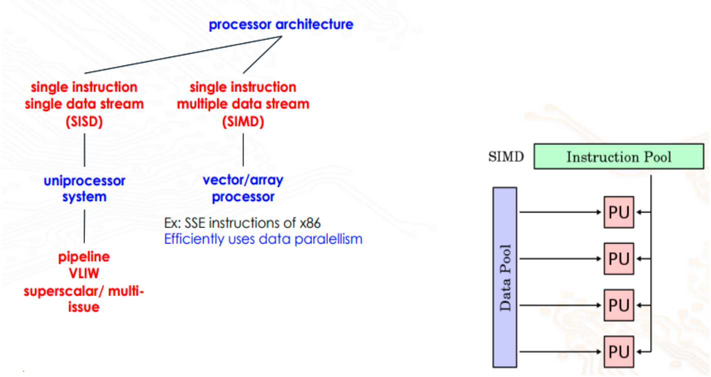


Fig A

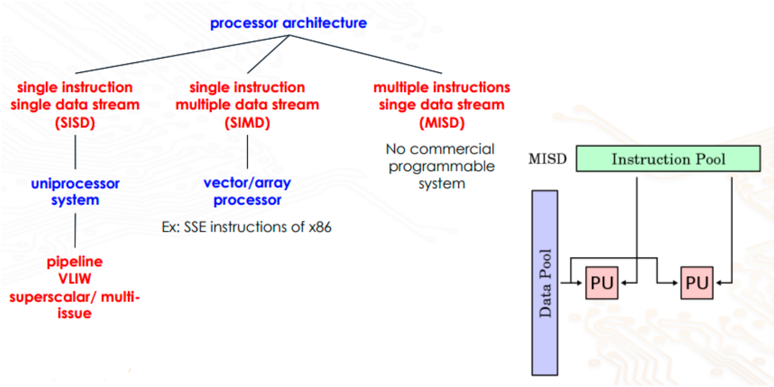
- Computing models-Flynn’s classification
    - Single instruction, single data stream(SISD)
        - Conventional sequential processor
        - Not suitable to realize data-level parallelism


processor architecture
single instruction single data stream (SISD)
uniprocessor system
pipeline VLIW superscalar/ multi-issue
SISD
Instruction Pool
Data Pool
PU

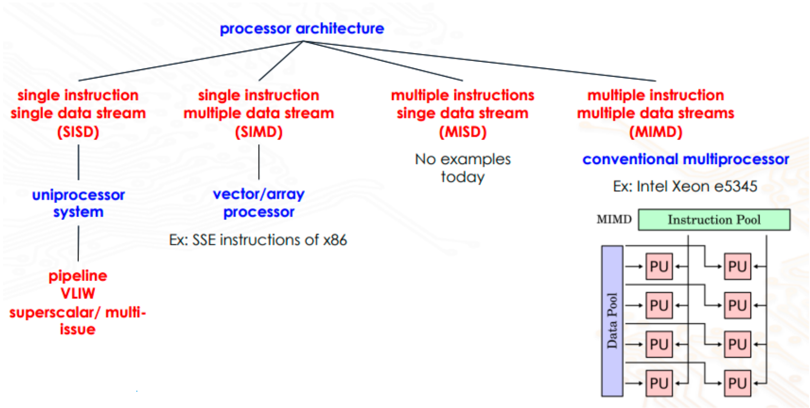
- Single instruction, multiple data stream(SIMD)
  - Efficiently utilize the data-level parallelism


processor architecture
single instruction single data stream (SISD)
single instruction multiple data stream (SIMD)
uniprocessor system
vector/array processor
Ex: SSE instructions of x86
Efficiently uses data paralellism
pipeline VLIW superscalar/ multi-issue
SIMD
Instruction Pool
Data Pool
PU
PU
PU
PU

- Multiple instructions, single data stream(MISD)
  - No commercial programmable system *MISD structure is only of theoretical interest since no practical system has been constructed using this organization.


processor architecture
single instruction single data stream (SISD)
single instruction multiple data stream (SIMD)
multiple instructions singe data stream (MISD)
uniprocessor system
vector/array processor
No commercial programmable system
Ex: SSE instructions of x86
pipeline VLIW superscalar/ multi-issue
MISD
Instruction Pool
Data Pool
PU
PU

- Multiple instructions, multiple data streams (MIMD)
  - True multiprocessor system that issues multiple instructions
  - Can support data-level parallelism


processor architecture
single instruction single data stream (SISD)
single instruction multiple data stream (SIMD)
multiple instructions singe data stream (MISD)
multiple instruction multiple data streams (MIMD)
uniprocessor system
vector/array processor
Ex: SSE instructions of x86
No examples today
conventional multiprocessor
Ex: Intel Xeon e5345
pipeline
VLIW
superscalar/ multi-issue
MIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
PU
PU
PU
PU

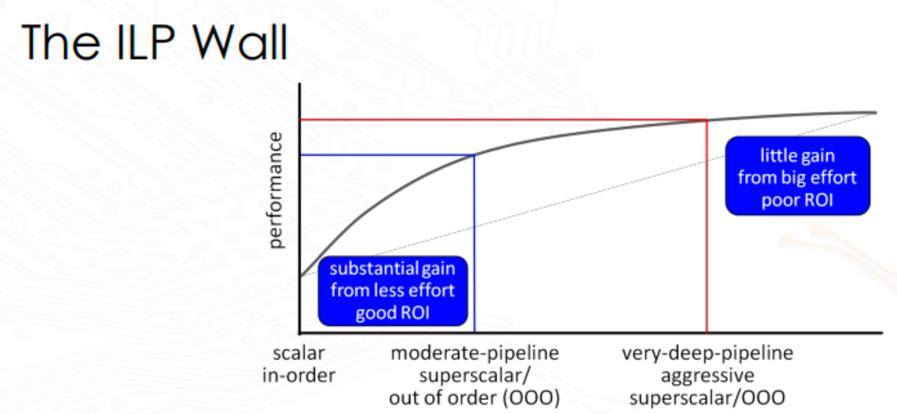
- **Single instruction multiple data stream (SIMD)**
    - One instruction is issued every cycle, the same instruction is used to execute for different data elements by different processors
    - The central control unit issues instructions and controls the simultaneous execution of instructions.
    - Each processing unit has its own data memory and also could interact via shared memory.
    - All processing units work synchronously
    - Application: signal processing, matrix/scientific computation.
    - Examples: vectors and array processors

- **Advantages of SIMD over MIMD**
    - SIMD make use of data-level parallelism efficiently.
    - If there are multiple processing units, N- operations can be performed in one clock cycle =>high performance
    - Loop-overhead instructions(address increment and branch condition check) will also be reduced by a factor of N
        - **overhead** is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task
    - Energy efficiency: energy used in instruction fetch will be reduced since there is only one instruction
    - The biggest advantage is that the programmer continues to think sequentially yet achieves parallel speedup by having parallel data operation
- SIMD Processors
    - Vector processor
        - Same operation at the same processing element (this PE will do + only)
        - Different operations at the same time(at time t, one PE can only handle one operation)
    - Array processor

        - Same operation at the same time(a few PE will do the same operation at the same time)
        - Different operations at the same processing element(every PE can do different operations, but cannot do them all at the same time)

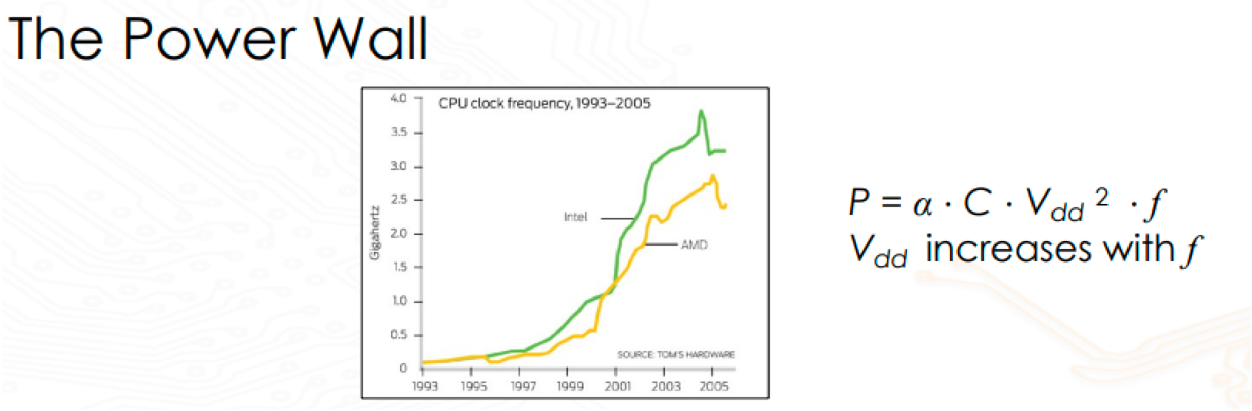
Thread-level parallelism and multi-core processors
Doing many tasks at the same time: running 2programs concurrently

- Motivation for multicore
    - Need for performance enhancement
        - Need a faster and more capable system to support increasing transmission speed and communication bandwidth
        - Increase resolution of image/video data
        - Increase the use of video conferencing and surveillance.
        - New applications: Drug discovery with a high volume of DNA data processing, scientific computing, and weather forecasting
- Moore's law
    - The number of transistors in an IC chip doubles approximately every 1.5 years
- Traditional approaches for performance enhancement
    - Increase clock speed
        - Not really a good idea: might overheat the CPU, damage Processors, or Excessive Cooling that would take more time
    - Decrease clocks per instruction(CPI)
        - Instruction level parallelism
- The ILP wall



        - High design and verification time and cost
        - Diminishing returns in more ILP
        - Instruction-level parallelism is near its limit

- The Power Wall



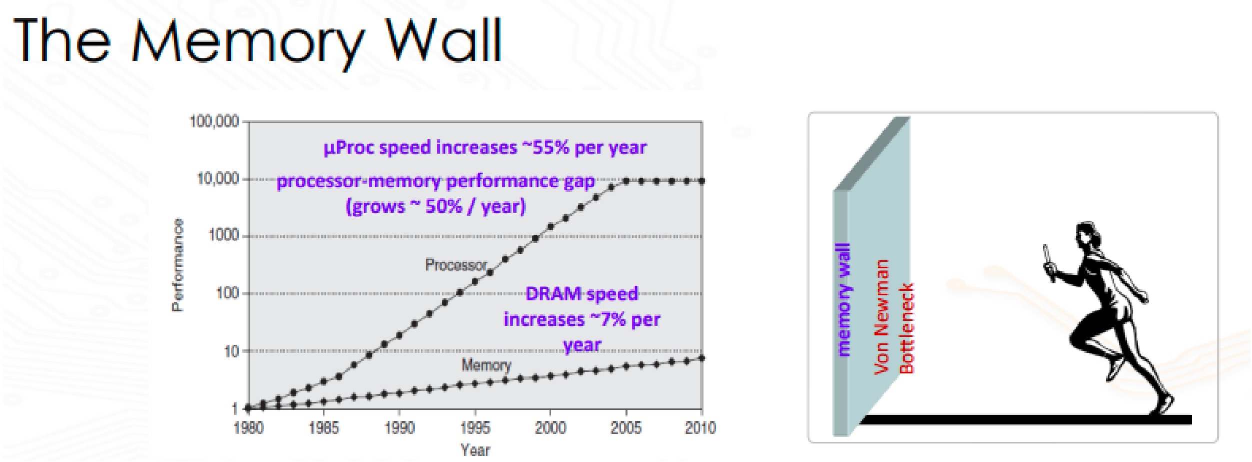
The Power Wall

$P = \alpha \cdot C \cdot V_{dd}{}^{2} \cdot f$

V_{dd} increases with f

  - Power consumption increases more rapidly with the increasing operating frequency
  - When frequency increases, the power dissipated dynamically increases as well\

- The Memory Wall



The Memory Wall

  - The rate of improvement in microprocessor performance far exceeds the rate of improvement in DRAM memory speed. Such a trend rendered the memory subsystem to become one of the most crucial system-level performance bottlenecks.

- The Brick Wall
  - Can be overcome by having more cores/ multicore processor
    - Able to do more task simultaneously
    - Consumes lesser power due to lower clock frequency

- Advantage of multicore

    - Overcome power wall
    - Derive parallelism by thread-level parallelism
    - **Homogenous multicore systems:** consist of identical cores, **less design and verification cost**
    - **Heterogeneous multicore systems:** consist of different types of cores each optimized for different roles

<u>Challenges:</u>

- **A New Power Wall**
    - Moore’s law, more transistor density, more heat dissipated per unit area
- **A New Memory Wall**
    - Due to a lack of ***memory bandwidth**
        - *the rate at which data can be read from or stored into a semiconductor memory by a processor.*
    - Contention between cores over the memory bus