Rechnerarchitektur

Abdullah Alzein - WS 2024/2025

Contents

RISC vs CISC, Microprogramming

	RISC	CISC
Number of Instructions	Small	Large
Length of instructions	fixed	variable (1-15 bytes)
Architecture	load/store, otherwise access to registers	memory-to-memory direct access
Registers	many (RISC-V has 32)	Few (x86 has 8)
Control	Hardwired	Microprogrammed
Examples	ARM, MIPS, RISC-V	x86, IBM, VAX

Microprogramming

Control Signals (**Micro Instructions**) build a Microprogram $\overset{\text{together}}{\Longrightarrow}$ Code Line (**Macro Instruction**)

Pros:

- Flexible for adding new instructions instead of hardwiring new hardware
- Update-able by simply adding new programs or fixing bugs
- Backwards compatible
- Simpler hardware than hardwiring everything (see control unit)

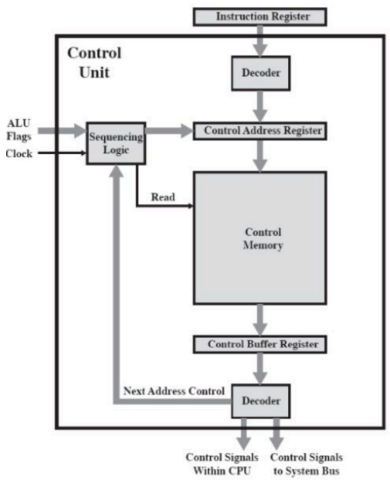
Cons

- Performance: Slower than hardwired direct mapped controls
- overhead: additional memory (control unit)
- limited parallelism: Microprograms are sequential while hardwired control can be more easily pipelined

Control unit

How, in MP, instructions are mapped to signals

- Instruction Register: runtime macro instructions
- Decoder: obtain micropgoram address
- Control memory: contains microprograms
- Control Buffer Register: micro instruction is loaded into it from memory
- Decoder: determines what signals to generate based on micro instructions from memory
- Sequence: Determines the next micro instruction to execute



Pipelining and Superscalarity

Pipeline Stages:

Instruction Fetch IF ⇒ Instruction Decode ID ⇒ Operand Fetch OF ⇒ Execution EX ⇒ Memory Access MEM ⇒ Write Back WB

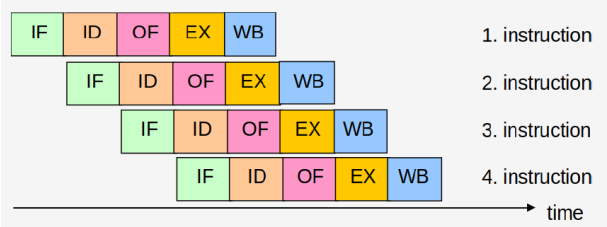
Clock Cycle:
1. Single Cycle CPU: time it takes to complete all stages for a single instruction
2. Pipelined CPU: time it takes to complete one stage

Cycle Time: Duration of one clock cycle
Clock Frequency: $\frac{1}{\text{Cycle Time}}$ = number of cycles per second
Speed-Up:
- Single Cycle CPU: $T_S = n \cdot (\tau_1 + \tau_2 + \tau_3 + \tau_4 + \tau_5 + \tau_6)$
- Pipelined CPU: $T_P = k \max(\tau) + (n-1) \cdot \max(\tau)$
  (Add register delays when they are given)
  ($k = \text{number of stages}, n = \text{number of instructions}, \max(\tau) = \text{critical path} \equiv \text{longest delay}$)
- Speed-Up: $\frac{T_S}{T_P} \underset{n \to \infty}{\longrightarrow} k$

Pipelining with RISC vs CISC:
Easier because RISC has uniform and simple instructions and typically need only one cycle each, (hardwired) $\Rightarrow$ stages well separable

Superscalarity:
- Multiple instructions within the same pipeline and the same stage by multiple hardware units
- Best to have multiple units for stages that take longest to perform, thereby addressing the bottle-neck of the pipeline by

3 Branch Prediction:
Given a pipeline in the following shape,



A Branch (if/else) at Instruction 2 leading to either 3 or 12 will force us to flush (stall or basically reset) the pipeline. Since one cannot know the outcome (target instruction) of a branch, this stalling creates **gaps** wasting time
Therefore: **Branch Prediction**

Static Branch Prediction

Stall = No Prediction
- Flushing the pipeline upon branches, see example
- keeps pipeline empty a long time (branches are ~25% of assembly code)
- very inefficient but easiest to implement

Always not Taken
- CPU fetches next sequential instruction as if branch wasn't taken
- if branch is taken, pipeline is flushed
- slightly better than stalling statistically

Always taken

- CPU fetches target instruction of the branch, usually the label: `beq a0, a1, label`
- Requires overhead to tell what instructions are branches and their targets
- Is better than Always not Taken since most branches are taken (function calls are branches)

By OPCode

- applying Always (Not) Taken on certain branches based on common behavior of programs
- Example: `beq` is always not taken, `blt` is always taken (loops)
- better results while harder to implement

Delayed Branch

- reorder n instructions that are independent from the branch outcome. Place them after the branch such that the are executed before it is done resolving
- If no such instructions exist due to dependencies, add `NOP`s

(+) Static/before runtime optimization
(-) not portable, compiler must know architecture and number of delay slots

Speedup Delayed Branches:

$$S = \frac{n \cdot k \cdot \tau}{\tau(k + n - 1 + b \cdot c)}$$

- $n =$ number of instructions **executed** in the code
- $k =$ number of stages in the Pipeline
- $b =$ number of branch instructions **executed** in the code
- $c =$ cost of control hazards

Additionally with taken/not taken costs:

$$S = \frac{n \cdot k \cdot \tau}{\tau(k + n - 1 + b_{\text{taken}} \cdot c_{\text{taken}} + b_{\text{notTaken}} \cdot c_{\text{notTaken}})}$$

Dynamic Branch prediction

- Branch history table: stores recent history of branches, entry contains a few bits of the address of a branch instruction and the instruction outcome

1-Bit Branch history table

- A 1 Bit state machine
- Future decision is whatever the last outcome was
- 1 if taken, 0 if not taken

2-Bit Branch history table

- A 2 Bit state machine (4 states)
- Mispredictions would not be forgiven in a 1-Bit table, here they get one extra chance
- If e.g. weakly taken, then set to strongly taken

Saturating Counter

- keeping track of history mathematically based on previous outcomes:

```
case taken: counter = min(counter + 1, MAX_VAL)
case not_taken: counter = max(counter -1 , 0)
```

- Decision is made based on whether the counter is below or above `MAX_VAL/2` or not

(+) easy to implement (-) does not adapt well to radical changes

Other Branch Prediction information:

- **Local, Global or hybrid Predictors**: keep track of prediction history for respectively a single branch, all branches or both
- **Branch Target Buffer**: Store target address of a branch
- **Branch Loop Buffer**: Store last `BUFFER_SIZE` instructions to speed up **`IF`**.
  (+) Speeds up loops (-) but bad for large loops

Data Hazards

- **Structural Hazard**: Instructions require same hardware simultaneously (2 Instr. want 1. MUL)
- **RAW**: Read After Write: Reading an old value because an earlier instruction has not yet WB
- **WAW**: Two instructions write to the same destination. Later writes before earlier ⇒ result lost
- **WAR**: Earlier instruction wants to read value but newer one may have written to it ⇒ old val lost

Out-of-Order

- Issue: **`ID/OF`** to **`EX`** in a different order than program order
- Execute: **`EX`** in a different order than program order - Introduces WAW and WAR Hazards
- Commit: **`WB`** in a different order than program order

Scoreboard

- Centralized, dynamically resolves RAW and prevents WAW and WAR hazards
- Used in Out of Order Execution or Commit pipelines
- Determines when an instruction can be issued, executed or committed

(+) out-of-order execution to avoid stalls, simple and avoids RAW
(-) centralized (bottle-neck), complex with too many functional units, issue remains in order, inefficient WAW and WAR compared to Tomasulo

Tomasulo

- decentralized, each functional unit is given a reservation station to keep track of its status
- WAW and WAR are avoided using register renaming
- Resolves ... by:
  - RAW: Keeping track which registers are currently (busy)
  - WAR: saving a reference to the responsible functional unit instead of reading from register
  - WAW: not writing the value to the register, but only to the common data bus, only newest instruction will write to the register

(+) OoO-Commit is now possible; structural hazards dont cause stalls; Decentral ⇒ better scalability
(-) More complex than scoreboard, latency due to overhead, common databus is a bottleneck

Reorder Buffer: Rollback requires enforcing in-order commit to restore register values ⇒ buffering write backs in a Reorder buffer (FIFO)

Caches

Locality

Temporal

- Access same memory address multiple times within a short time (`i` in a `for`-loop)

Spatial

- Accessing entire memory chunks (cache lines), to get access to data close to each other, e.g. arrays

Instruction

- Temporal and Spatial localities apply on instruction addresses (*Programs are on memory too*)

⇒ Caches for instructions too

- If Instruction cache is separate from data cache ⇒ **Harvard Architecture**

Cache Lines ≡ **Cache Block**: Cache Miss ⇒ entire cache line (often 64 Bytes) is fetched from memory

Byte Offset: a few bits used to determine the byte within the cache line **Cache Tag**: The remaining bits of the memory

Placement and Identification:

- Placement problem: Which cache line contains memory address x
  ⇒ Mapping Strategies (direct, n-way-associative, fully associative)
- Identification problem: Where in that line is the address x
  ⇒

Fully Associative Cache:

- Any memory block can be placed in any cache line. There is no fixed mapping between memory blocks and cache lines

(+) low probability of conflict misses, flexible, maximum utilization
(-) many checks (all cache lines in worst case), expensive, low performance

Direct-Mapped Cache

- Each memory block maps to exactly one cache line. Index bits of address to determine cache line

(+) simple and fastest lookup since each block has a line (1 comparator and multiplexer)
(-) conflict misses (multiple addresses map to the same cache line)

N-way associative Cache

- The cache is divided into sets each containing N lines. A block maps to a specific set determined by the index buts but can occupy any line within that set

(+) Balance both other mappings, reduces conflict misses
(-) More complex than direct-mapped due to more checks, gets more complex with increasing N

Cache Calculation

Cache Size = $\#\text{lines} \cdot \text{size(line)}$ **Number of Sets** = $\frac{\#\text{lines}}{\text{associativity}}$ **Associativity**= $\frac{\#\text{lines}}{\#\text{sets}}$

- Associativity: Direct-mapped:1 n-way: n fully-associative: $\#\text{lines}$ $(\#\text{sets} = 1)$
- tag - index(#sets) - byteoffset (line size) (= how many bits needed to represent those values)

Cache misses:

- Compulsory miss: first access miss
- Capacity miss: in fully ass. cache, happens when an address needed was evicted for lack of space
- Conflict misses: dont happen in fully ass. cache; eviction due to duplicate.

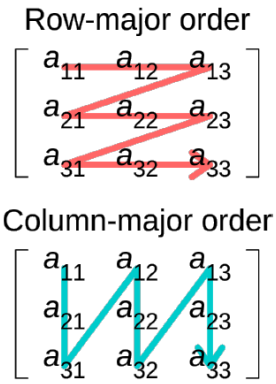
Replacement strategies:

- Least Recently Used LRU
- Pseudo-LRU (instead of time counting overhead; binary tree, pointing to cache line and inverting all pointers upon change on that line. Line pointed to by root is chosen as LRU)
- Least Frequently Used LFU
- FIFO

- Random

Cache in Code:

Spatial locality:
You should use **Row-Major** to make use of locality



Temporal locality:
- use **loop fusion** to access same data in one loop:

Before:

```
for(int i = 0; i < N; i++){
b [i] = a [i] + 1;
}
for (int i = 0; i < N; i++){
c [i] = a [i] * 2;
}
```

After:

```
for(int i = 0; i < N; i++){
b [i] = a [i] + 1;
c [i] = a [i] * 2;
}
```

Conflict reduction by Array Merging

Before:

```
int a [SIZE] , b [SIZE];
. . .
for(int i = 0; i < SIZE; i++){
  a [i] = b [i] * 2;
}
```

After:

```
struct {
int a ;
int b ;
} data [SIZE] ;
. . .
for(int i = 0; i < SIZE; i++){
  data [i].a = data [i].b * 2;
}
```

Cache Blocking
- large array exceeds length of cache line
- access in blocks of array size to be cache-friendly

Caches II

Write back/Write Through
- **WB:** write to the next higher level when data gets evicted, reducing writes to memory
- **WT:** write to the next higher level immediately. Robust and always up to date but too many writes

Write-Allocate/ No-Write-Allocate
- **WA**: fetch cache line to write changes
- **N-WA**: write changes to next-higher level without bringing down the line to this level

Victim Cache: last level cache that stores evicted lines from main cache and is checked before resorting to the main memory

Cache Latency/Throughput

- **Latency:** time to access the memory [ns]
- **Throughput:** number of memory accesses than can be handled in a given time [GB/s]

Memory Latency on average T_a

Cache Latency T_c

Main memory latency T_m

hit rate h

Single-level-cache System: $T_a = T_c + (1-h)T_m$
Multi-level-cache System: $T_a = T_{c1} + (1-h_1)(T_{c2} + (1-h_2)(...))$
(Dont add register delays if those are given - I did that and lost a point -_-)

- Increasing Cache Size, associativity or line size will increase h but also T_c
- Speculative pre-fetching and out of order and memory-level parallelism reduce or hide T_m

Cache Coherence/Consistency

- Consistency: data in cache line is identical to data in main memory (achieved using write-through)
- Coherence: all caches have the same view of the memory, no different copies of the same data. Protocols are used to ensure coherence on multi-core
- Snooping based protocols: Cache sends message to other caches upon modification of a line to invalidate their copies of that line

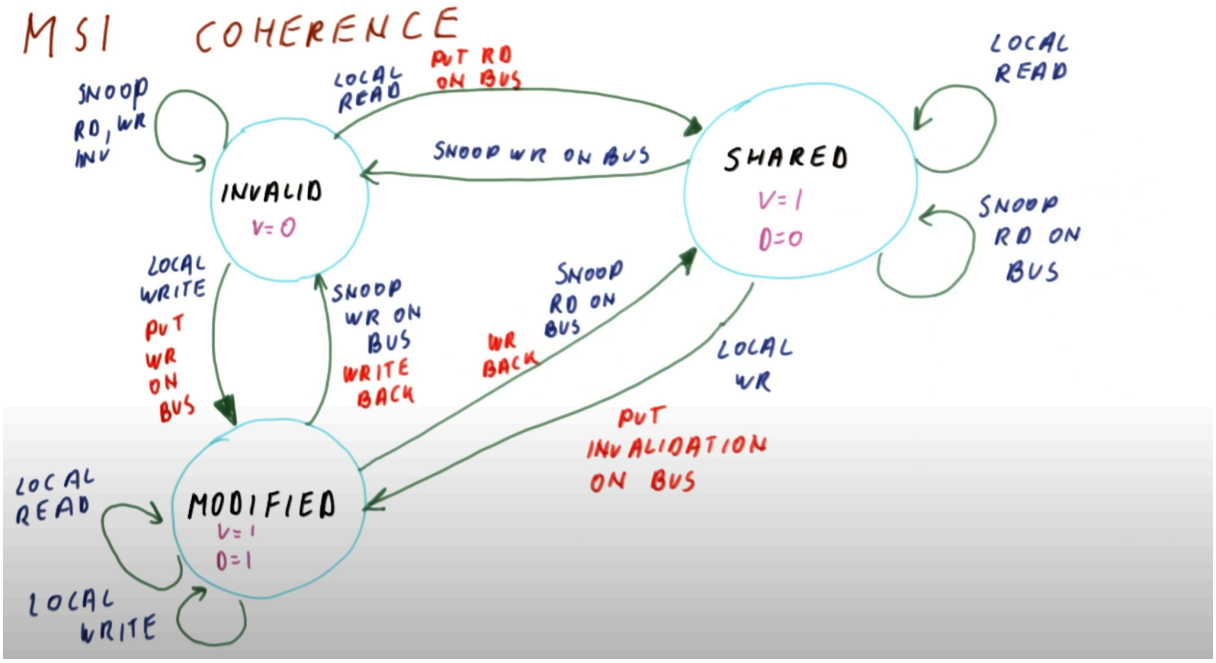
MSI Protocol

- Modified: Cache line is modified and different from main memory
- Shared: Cache line is unmodified and potentially shared with other caches
- Invalid: cache line is invalid (not present in the cache)

MESIF

- Exclusive: Cache line isn't shared with other caches (exclusive to one cache)
- Forward: cache line is shared and the cache has the most up to date copy (cache sends data to other caches upon request, those other caches then assume the role of forwarding the copy)

False sharing: accessing different data on the same line in multicore: too much sharing overhead
$\Rightarrow$ align parallelization to cache line lengths to avoid that



Multicore/Parallelism

- Moore's law (#transistors double every year) (1975)
- Kraznich (every 2.5 years) (2010)
- Gelsinger (every 3 years) (2023)

Power consumption: $P = C \cdot V^2 \cdot f \cdot N$

- C: Capacitance $:= \varepsilon \cdot \frac{A}{d}$
- V: Voltage
- f: Frequency
- N: transistor density (transistors per area)
- ε: Material Permittivity (given)
- A: Area, d: Distance between plates/wires/transistors

Dennard's Scaling

- Assume that A is reduced to $\frac{1}{\alpha}$ of original size

$\Rightarrow \alpha^2$ more transistors for same area
$\Rightarrow$ Voltage is reduced by $\frac{1}{\alpha}$ too since less distance required $\quad \Longrightarrow P_{\text{new}} = \frac{1}{\alpha} P_{\text{old}}$ (now increase f)
$\Longrightarrow$ Reducing size by $\frac{1}{\alpha}$ allows us to increase frequency by α and number of transistors by α^2 while keeping power consumption

Multicore:

- Multicore: multiple cores on a single chip
- Manycore: high number of cores up to 64
- Hetero/Homo: Different/Same types of cores on the MC-chip
- **Amdahl's Law for parallelization**: $S = \frac{1}{ser\% + \frac{1 - ser\%}{N}}$ for **N** cores and ser% + par% = 100%
- **Pollack's Rule**: Performance increase is square the increase factor n of #transistors: $S \approx \sqrt{n}$

$\Rightarrow$ installing 2 cores with half transistors (thus **same total amount**) yields $2 \cdot \sqrt{0.5} = 1.41 > 1$

- **Koomey's Law**: Energy efficiency = amount of computations per Joule - doubles every 1.57 years
- **Multi-Core is more** energy **efficient** due to usage, clocking and work distribution optimizations
  - P-Cores/E-Cores: Power vs Efficiency Cores (difference is in transistors count)

SMT, Taxonomy, Roofline

Simultaneous Multithreading:

Running multiple threads on a single core

- Register files, program counter and control logic are duplicated since unique to each thread
- Functional Units (ALU) and the Cache are shared by threads

(+) Better utilization of resources, performance for high memory latency or low parallelism, cheaper
(-) increasd cache contention / works only if threads require different resources

Temporal Multithreading

- basically the work of a scheduler. Switching between multiple threads on the same core

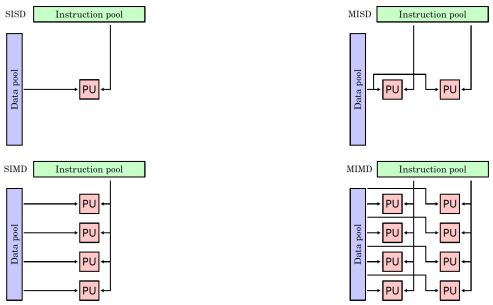
(+) Better resource usage, bigger gaps in the pipeline that hide other latencies or hazard stalls, simpler than SMT

(-) lower performance than SMT, cache has to remain shared, where in pure multicore it isnt

Flynn's Taxonomy:

- S = Single, M = Multiple, I = Instruction(s), D = Data
  - SISD
  - SIMD
  - MISD
  - MIMD

(PU = Processing Unit)



Roofline

Algorithms are either:

- **Memory bound** : Bottleneck (or usage) in memory
- **computer bound**: bottle neck is the computation (algorithm requires more CPU than memory)
- **Operational Intensity** $= \frac{\text{Number of Operations}}{\text{Amount of Data transfered from memory}} \quad \left(\frac{\text{FLOP}}{\text{Byte}}\right)$

Peak Floating Point Performance
maximum number of FLOPs per second that can be executed on a processor ($\frac{FLOP}{s}$)

- Parameters
  - #Cores
  - SMT
  - Clock Rate
  - Latency/pipelining
  - SIMD unit width
  - number of floating point units

Peak memory bandwidth
maximum amount of data that can be transferred between CPU and memory ($\frac{Byte}{s}$)

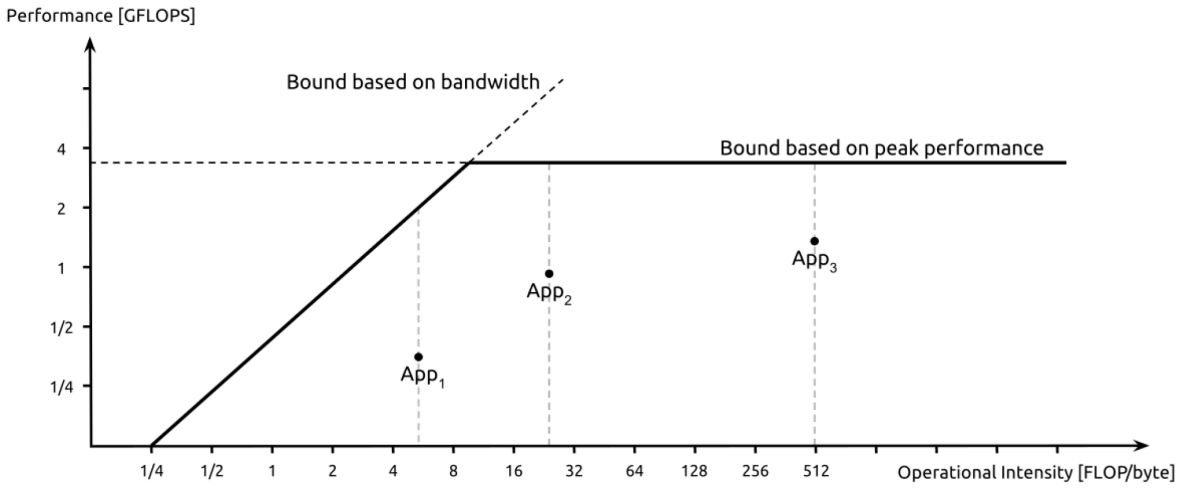
- Parameters
  - memory type (DDR)
  - memory clock rate
  - memory bus width
  - number of memory channels
  - memory technology

- **Maximum attainable performance**:
  min(peak floating point performance, peak memory bandwidth · operational intensity)

Roofline model

- visualizes maximum performance possible by an algorithm on a certain hardware

- Roof Top: PFPP
- Roof Slope: PMB × OpIntens.



GPU

- GPU-Thread: a single thread in a group of threads that work on the exact same task, the data is different (Single Instruction Multiple Threads SIMT)
- Branches: Possible but due to SIMT cause performance loss as all threads will execute both branches - the correct branch is then written back by one of them

What can be parallelized on a GPU

- SIMD (same operation on different date)
- SIMT (same instruction at the same time)
- no dependencies between threads `Dont: c [thread] = c [ thread-1] + a [thread];`
- no system calls `print("result:%d", c[threadIndex]);`
- simple arithmetic operations
- simple arithmetic operations (GPU best for add, sub, mul not sin,cos or others)
- rare use of branches
- **Kernel**: A function that is executed on the GPU - device code (GPU), host code (CPU)

Grid ⇒ Blocks ⇒ Threads

- **Grid**: the entire dataset that is to be processed by the GPU
- **Block**: group of data processed by a GPU multiprocessor
- **Threads**: single unit of execution, handles either one or a small number of data elements

GPU ⇒ Streaming Multiprocessors ⇒ Shaders (CUDA cores) and Special Function Units

- **SM**: group of shaders working together, they work on the same block (**1:1 — SM:Block**)
- **Shader**: a processing unit working with a single thread at a time, has a simple ALU and registers. All shaders in one SM execute the same instruction at the same time
- **SFU**: special shaders that execute complex functions, limited in number so parallelism is limited

Grid	GPU

Block	Streaming Multiprocessor
Thread	Shader (CUDA Core)

- Programmer: defines division of datasets into blocks and threads (elements multiple of unites)
- GPU: schedules blocks to SMs, assigns threads to shaders and runes instructions all at runtime

GPU Memory

- Register: fastest — inside the shader
- Shared Memory: Inside the SM, shared across shaders within that SM
- L1-Cache: Pro SM, shared by Shaders in an SM
- L2-Cache: Shared between all SMs in a GPU
- Global Memory: Largest within GPU, accessible by the whole GPU (across multiple Grids)
- RAM: External: Slowest due to latency on PCIe, low bandwidth

The L1 and Shared memory are the same physically, L1 is used by hardware and shared by software. Modern GPUs allow you to configure ratios

- **Memory Coalescing** Combining thread requests into a single larger memory transaction. Possible if:
  - accesses are contiguous: spatial locality
  - accesses are aligned: multiples of bus width

Positive Example of Good Memory Access:

```
int i = threadIdx;
c[i] = a[i] + b[i];
```

Negative Example of Bad Memory Access:

```
int i = threadIdx;
c[i] = a[i] + b[i + 1]; // Unaligned access
```

Negative Example of Bad Memory Access:

```
int i = threadIdx;
c[i] = a[i * 2] + b[i * 2]; // Non-contiguous access
```

What To Do Next:

- Redo Exercise Sheets: Await Lecturer's notes on what parts to leave out, this is helpful
- Mock Exam: Best to solve this to know exactly how the exam is structured

Do Not rely entirely on this summary
Make sure the contents have not changed between WS2024 and now.
If you find mistakes or have any thoughts to share, write me:

abdulalh.alzein@fau.de

FIN