

# 60001

Advanced Computer Architecture  
Imperial College London

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Course Structure and Logistics . . . . .	3
<b>2</b>	<b>Pipelining</b>	<b>4</b>
2.1	Instruction Layout . . . . .	4
2.2	Pipeline Structure . . . . .	5
2.3	Pipeline Hazards . . . . .	5
2.3.1	Structural Hazard . . . . .	5
2.3.2	Data Hazard . . . . .	6
2.3.3	Control Hazard . . . . .	8
2.4	Simultaneous Multithreading . . . . .	9
2.5	Pipelining Roundup . . . . .	9
<b>3</b>	<b>Caches</b>	<b>10</b>
3.1	Why Caches . . . . .	10
3.2	Locality . . . . .	10
3.3	Cache Types . . . . .	11
3.3.1	Directly Mapped Cache . . . . .	11
3.3.2	Two Way Associative . . . . .	12
3.3.3	N Way Associative & Block Placement . . . . .	13
3.4	Block Identification . . . . .	13
3.5	Block Replacement . . . . .	13
3.6	Write Strategy . . . . .	13
3.7	Miss Rate Reduction Using Hardware . . . . .	14
3.7.1	Reducing Misses . . . . .	14
3.7.2	Increase Block Size . . . . .	15
3.7.3	Increase Associativity . . . . .	15
3.7.4	Victim Cache . . . . .	15
3.7.5	Skewed-Associative Caches . . . . .	15
3.7.6	Hardware Prefetching . . . . .	16
3.8	Miss Rate Reduction Using Software . . . . .	17
3.8.1	Software Prefetching . . . . .	17
3.8.2	Reducing Instruction Cache Misses . . . . .	17
3.8.3	Storage Layout & Iteration Space Transformations . . . . .	17
3.9	Miss Penalty Reduction . . . . .	18
3.9.1	Write Buffers . . . . .	18
3.9.2	Early Restart . . . . .	19
3.9.3	Non-Blocking Cache . . . . .	19
3.9.4	Multiple Cache Levels . . . . .	20
3.10	Hit Time Reduction . . . . .	20
3.10.1	Parallel Cache Access . . . . .	20
3.10.2	Address Translation . . . . .	21
<b>4</b>	<b>Dynamic Scheduling</b>	<b>23</b>
4.1	Bypassing Stalls . . . . .	23
4.2	Tomasulo's Algorithm . . . . .	23
4.3	Precise Interrupts . . . . .	25
4.4	Store Buffering . . . . .	25
4.5	Register Update Unit . . . . .	25

4.6 Register Alias Tables . . . . .	26
<b>5 DRAM</b>	<b>27</b>
<b>6 Side Channels</b>	<b>28</b>
6.1 Exfiltration . . . . .	28
6.2 Shared State . . . . .	29
6.3 Triggering Victim Execution . . . . .	29
6.4 Side Channels in Speculative Execution . . . . .	29
6.5 Mitigation . . . . .	30
6.6 Spectre v2 . . . . .	30
<b>7 Static Scheduling</b>	<b>31</b>
<b>8 Credit</b>	<b>32</b>

# Chapter 1

## Introduction

### 1.1 Course Structure and Logistics



Teaching the entire course.

- Microprocessor design.
- Optimising software for hardware, and compiler design.
- Optimising hardware for specific software tasks.
- Challenges past, present & future.

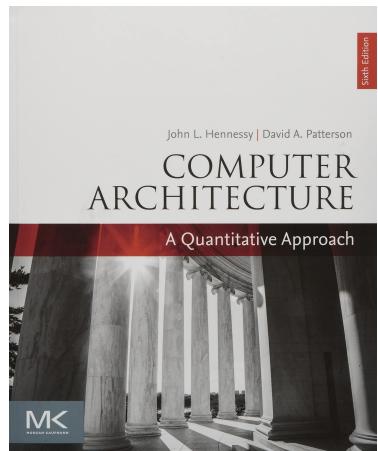
Taught through pre-recorded lectures and live tutorial sessions.

Prof Paul Kelly

This course is largely textbook based.

- 936 pages covering the course content and more.
- Useful appendices covering both introductory and advanced material.

The book is written by John Hennessy and David Patterson.



Computer Architecture:  
A Quantitative Approach (6<sup>th</sup> Edition)



Chapter 1 - Part 1: Introduction

# Chapter 2

## Pipelining



### Chapter 1 - Part 2: Pipelines

#### MIPS/Microprocessor without Interlocked Pipelined Stages

#### Definition 2.0.1

MIPS is a reduced instruction set (RISC) architecture originally developed for the R2000 microprocessor.

- 3 types of instruction layouts
- Load-store architecture
- Support for floating point arithmetic

## 2.1 Instruction Layout

The instructions set architecture (ISA) determines the layout of instructions. Here we consider the mips architecture.

### Register Type

31	26 25	21 20	16 15	11 10	6 5	0
opcode	Register Source 2 (Rs)	Register Source 1 (Rt)	Register Destination (Rd)	Shift code (shamt)	Function Code (funct)	
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

5 bit register specifier  
limits MIPS to 32 registers

### Immediate Type

31	26 25	21 20	16 15	0
opcode	Register Source 2 (Rs)	Register Destination (Rd)	immediate operand	
6 bits	5 bits	5 bits	16 bits	

Opcode specifies how the fields will be interpreted by specifying the instruction type

Opcode specifies how the fields will be interpreted by specifying the instruction type

### Jump Type

31	26 25	target
opcode		

6 bits

26 bits

Can use word-alignment (4 bytes - 2 bits) of instructions to address with 28 bits

### Branch (Immediate Type)

31	26 25	21 20	16 15	0
opcode	Register Source 2 (Rs)	Branch operation	pc relative address	
6 bits	5 bits	5 bits	16 bits	

address is offset from the current PC, with word alignment of instructions considered

The size of fields in the instruction layouts determines characteristics such as:

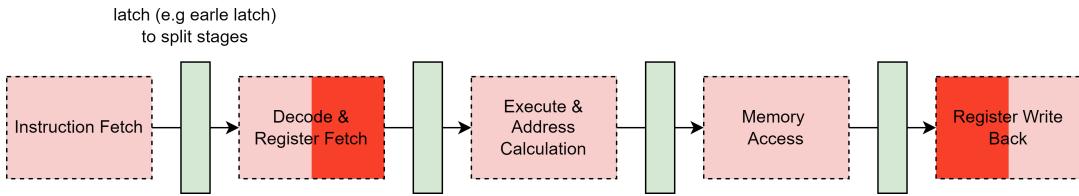
- Maximum number of registers
- Maximum distance for a conditional jump
- Size of immediate operands
- Range of addresses that can be used.

### MIPS Assembly

### Extra Fun! 2.1.1

A basic guide listing of mips instructions can be found here [Basic MIPS instructions](#).

## 2.2 Pipeline Structure



- Execution of an instruction is split into stages
- Throughput is potentially increased by factor  $1/\text{number of stages}$  (ideally)
- All stages work on an instruction simultaneously/in parallel (very little extra hardware required for the speedup advantage)

The speedup is reduced by

- Latency increased due to latches
- Pipeline rate limited by slowest stage (unbalanced stages / fragmentation)
- Time required to fill and drain the pipeline.
- Pipeline hazards which result in stalls (unable to dispatch another instruction in a given cycle).

## 2.3 Pipeline Hazards

### 2.3.1 Structural Hazard

#### Structural Hazard

#### Definition 2.3.1

Where hardware is unable to support a combination of instructions.

Multiple pipeline stages may need to access the same hardware resources:

- Register file (register operand fetch and register write back)
- Access to memory (RAM port in older machines, cache (SRAM) now)

#### Not enough ports!

#### Example Question 2.3.1

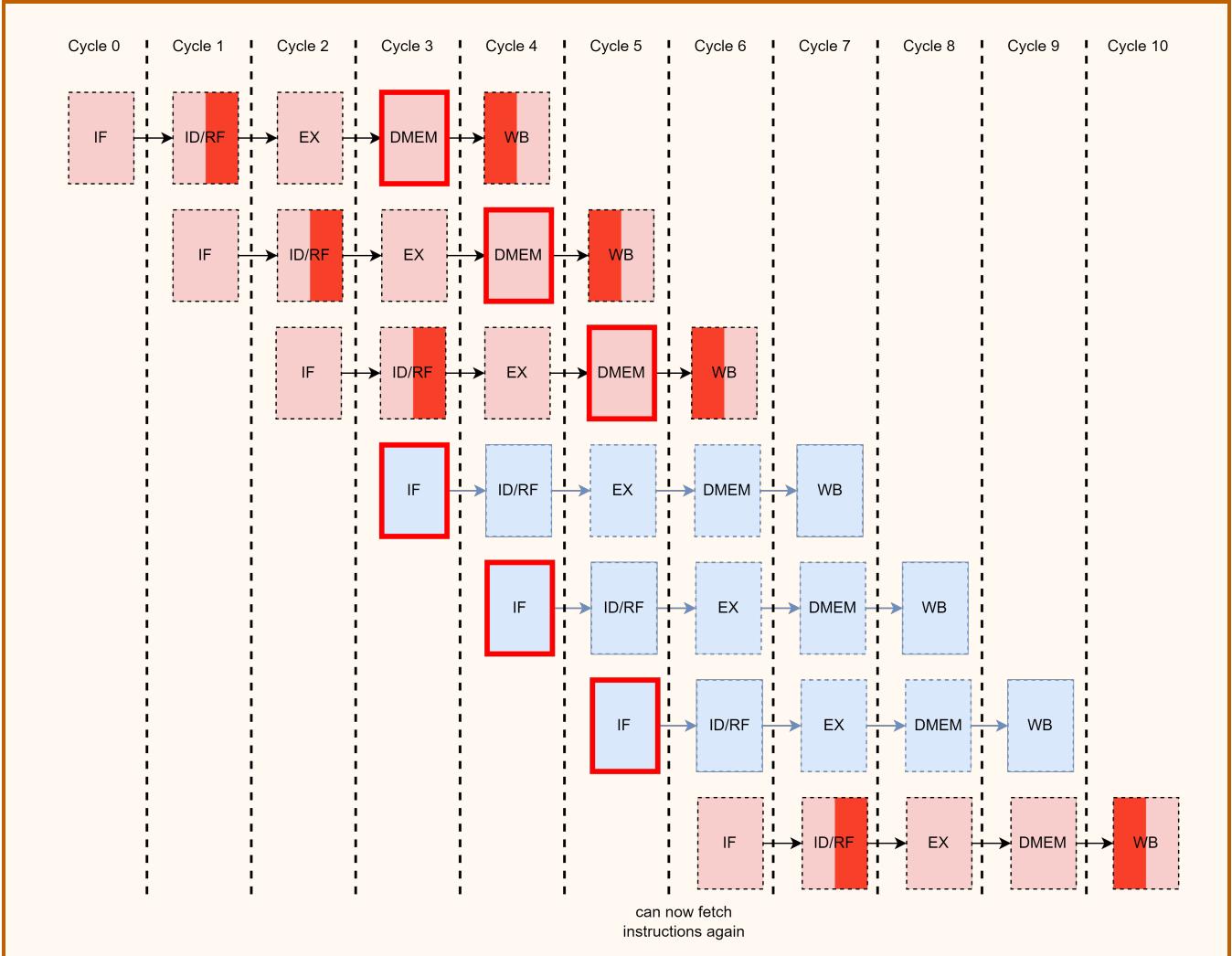
Given basic pipeline structure above, what structural hazard could occur between *Memory Access* and *instruction fetch* if there is only one RAM port?

No instruction can be fetched when the *Memory Access* stage is filled, this results in a stall.

The maximum potential speedup for a 5 stage pipeline is  $5\times$ , however due to the stalls we can see a recurring pattern:

Cycle:	$6n$	$6n + 1$	$6n + 2$	$6n + 3$	$6n + 4$	$6n + 5$
Instructions:	2	2	3	3	3	2

We would expect a  $5\times$  speedup from this pipeline. However we are only getting a  $2.5\times$  speedup due to the stalls.



### 2.3.2 Data Hazard

#### Data Hazard

#### Definition 2.3.2

Instruction is dependent on the result of a prior instruction still in the pipeline.

Most often caused by a dependency between instructions.

#### Forwarding Paths

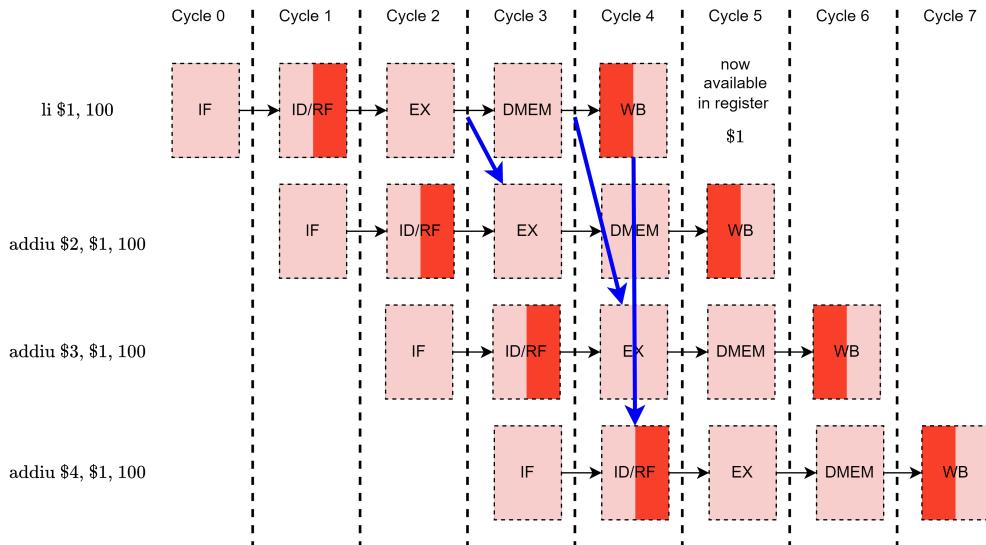
#### Definition 2.3.3

Paths between pipeline stages to allow results from previous instructions (not yet written back) to be sent to instructions afterwards that are in the pipeline.

### Result Used By Many Subsequent Instructions

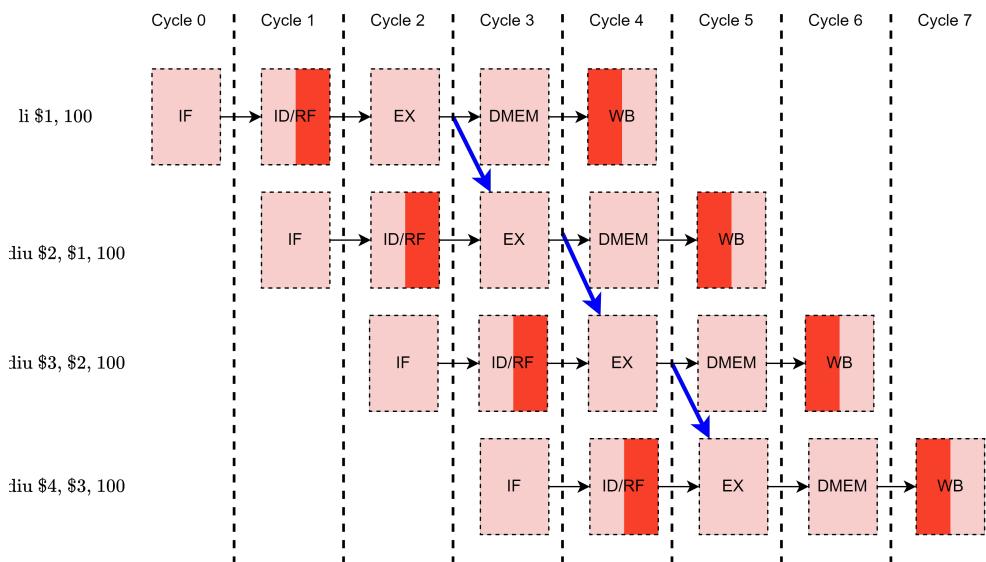
```

li    $1, 100      # $1 = 100
addiu $2, $1, 100 # $1 += 100 # here onwards depends on $1
addiu $3, $1, 100 # $1 += 100
addiu $4, $1, 100 # $1 += 100
  
```



### Chain of Results

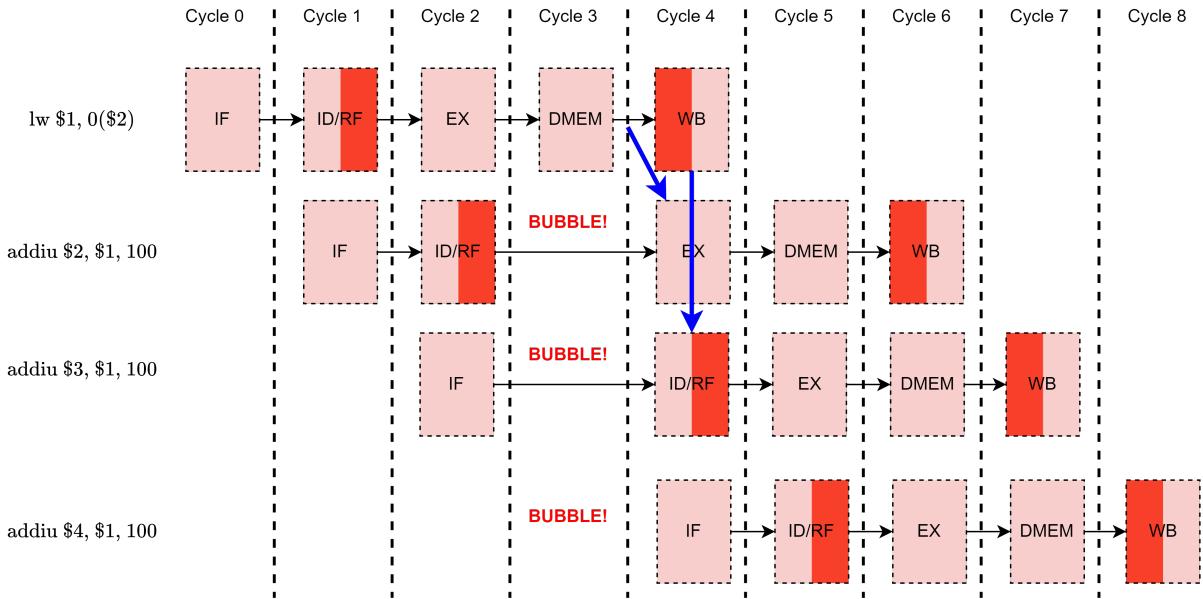
```
li    $1, 100      # $1 = 100
addiu $2, $1, 100 # $1 += 100
addiu $3, $2, 100 # $1 += 100
addiu $4, $3, 100 # $1 += 100
```



### Data Hazard Despite Forwarding

Here have a *load to use stall/delay*, forwarding paths will not work here as the memory access stage is 2 stages later than execute (where the instruction is required).

```
lw    $1, 0($2)      # $1 = *($2)
addiu $2, $1, 100 # $1 += 100
addiu $3, $1, 100 # $1 += 100
addiu $4, $1, 100 # $1 += 100
```



We can attempt to solve this issue using the compiler (e.g reorder instructions to put at least one non-dependent instruction between the load and the use).

## Forwarding Paths

### Software Scheduling

#### 2.3.3 Control Hazard

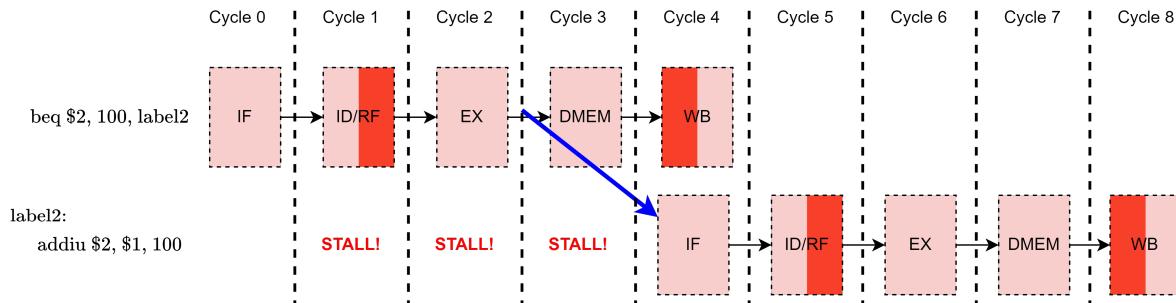
##### Control Hazard

##### Definition 2.3.4

A stall created by the delay between getting the result of some branch/jump and fetching the next instruction using that data.

Instruction fetch (without branch prediction) requires the conditional branch result to be known. Hence the number of stages between instruction fetch and when the branch condition is determined is the size of the stall resulting from a conditional branch.

- This is also true for jumps/unconditional branches where the address is provided by some register and arithmetic (e.g jump with offset)
- Branch prediction can be done dynamically (in hardware) or statically (specific branch likely, branch unlikely instructions used by compiler).

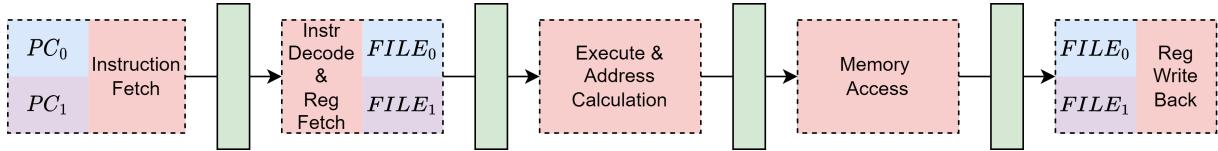


### Early Branch Determination

To decrease the number of cycles stalled in a branch, we can move the branch determination to earlier in the pipeline.

- Instruction decode determines branch.
- Still a one cycle delay in the MIPS example pipeline above.

## 2.4 Simultaneous Multithreading



We can eliminate stalls by interleaving the instructions of independent programs.

- Maintain two program counters for two programs, two sets of registers. Alternate between instructions from each program.
- No dependencies between adjacent stages, less forwarding, less complex instruction decode and control required.
- Each program sees half the clock frequency.

## 2.5 Pipelining Roundup

Pipelining offers increased throughput without much added hardware complexity by allowing execution stages to run in parallel as a pipeline.

- Simple 5 stage pipeline can run at  $5 \rightarrow 9\text{GHz}$
- Limited by critical path through slowest pipeline stage
- Clock period is  $330\text{ps} \approx 10$  gate delays at  $3\text{GHz}$  (3  $\rightarrow$  5 FO4 for latches, 5  $\rightarrow$  8 FO4 for work).
- Memory access needs to be done in 5  $\rightarrow$  8 FO4 delays (large constraint).

### FO4 Delays

### *Extra Fun! 2.5.1*

The gate delay of a component with a fan-out (gate inputs driven by a gate's output) of 4.

# Chapter 3

## Caches

### 3.1 Why Caches

The difference between cycle time (time of a stage in a pipeline) and memory access time has continually increased.

Size	Access Time	Storage/Memory	Managed By	Transfer Unit
100Bs	< 1ns	Registers	programmer/compiler	1 – 16Bs
10Kbs	1ns	Cache (SRAM)	L1 L2 L3	cache controller 8 – 128Bs
100Kbs	10ns			
GBs	100ns 300ns	Main Memory (DRAM)	Operating System	4 – 8KBs
TBs	10ms	Secondary Storage (Disk, SSD, Flash)	user/operator	MBs
<i>unbounded seconds minutes</i>		Backup Storage (Tape)		

### 3.2 Locality

Programs typically access only a small part of their address space during a short time period.

Temporal Locality	Definition 3.2.1
Locality in time. The same location referenced is often referenced multiple times.	

Temporal Locality	Definition 3.2.2
	Locality in space. Locations near an accessed location tend to be referenced soon.

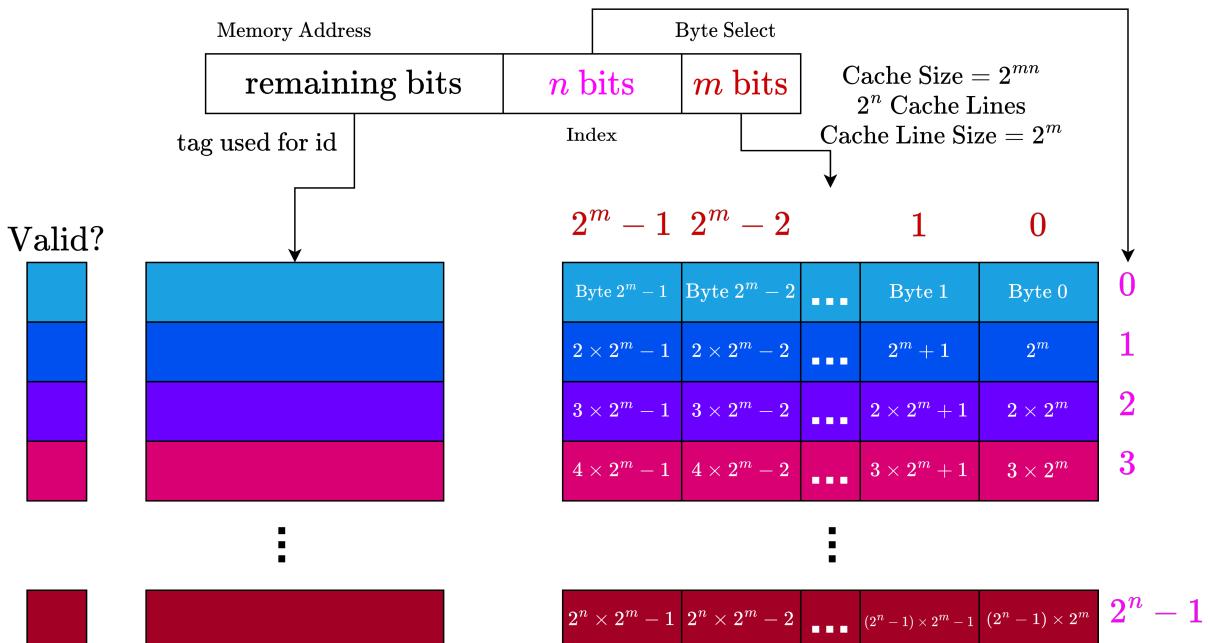
Most modern architectures are reliant on locality to determine when and what locations should be cached.

- Cache is a scarce resource.
- Cache misses are expensive.

### 3.3 Cache Types

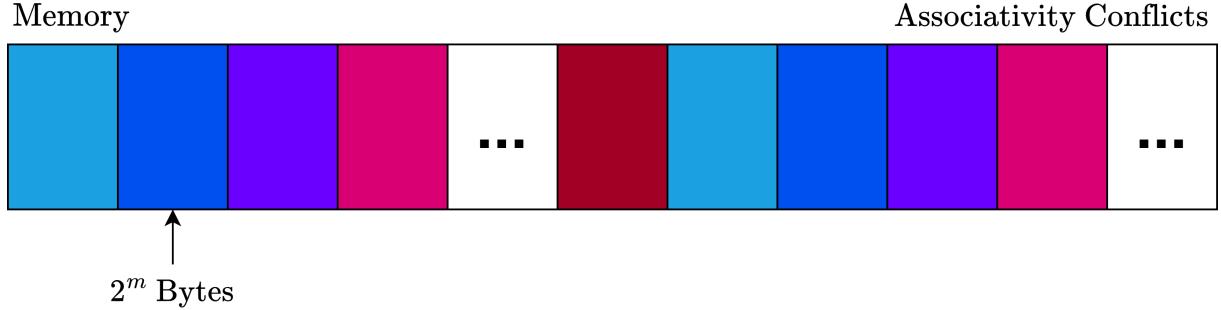
#### 3.3.1 Directly Mapped Cache

Associativity Conflicts	Definition 3.3.1
Where two or more locations are mapped to the same cache line/set of cache lines, and repeatedly replace each other.	<pre>/* Example with arrays, assume cache line is 256 bytes  * and both arrays start at same cache index  */  int array_a[64]; int array_b[64];  int some_function() {     int sum = 0;     for (int i = 0; i &lt; 64; i++) {         r += array_a[i] /* array_a moved into cache line */         + array_b[i]; /* array_b evicts array_a and replaces */     }     return sum; }</pre>



- Index and byte select used to find entry. Then tag compared to determine hit/miss.
- We can see a pattern in memory of where locations can be cached based on the index.

- Block/line received before the hit/miss is known (recover later if miss).

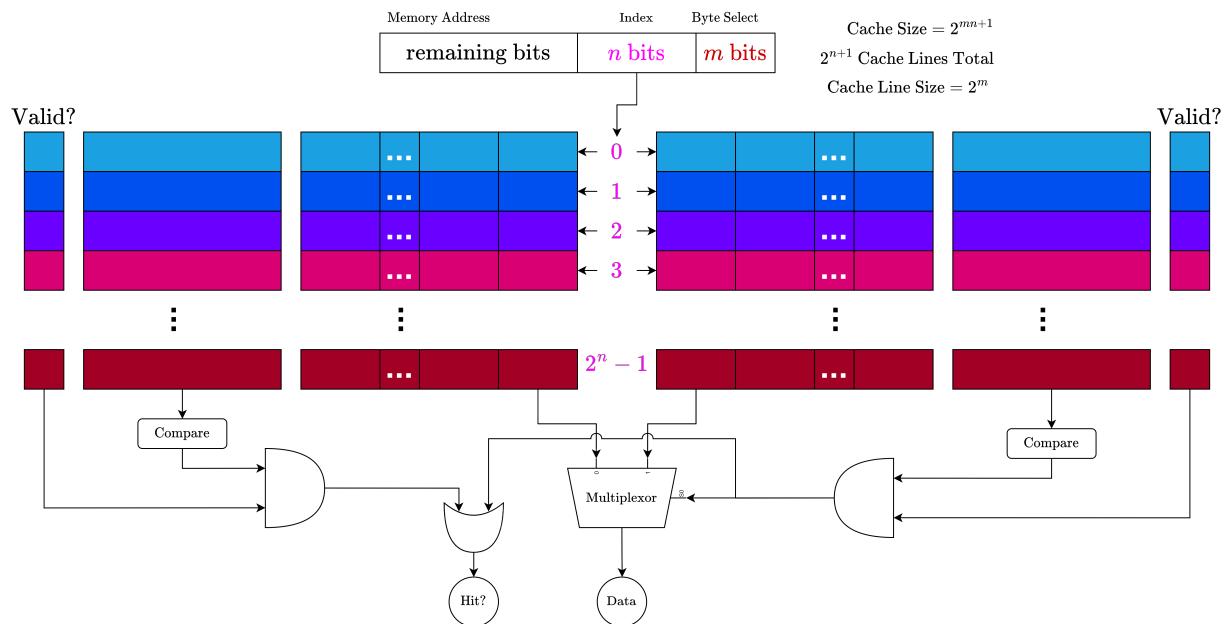


**Simplicity** Simple indexing of cache & compare to determine hit/miss.  
**Fast Lookup** Only one location where a cached value may be.

**Associativity Conflicts** As location can only be cached in one place, associativity conflicts are common.

### 3.3.2 Two Way Associative

Combine two directly mapped caches, and only cache a given location in one.



- Both caches searched in parallel.
- Only one hit possible, this is selected from result of both caches (selection is in the critical path)
- Cache block/line is available after the hit/miss is determined.

**Fewer Assoc Conflicts** Any location can now select two different locations in the cache, hence two addresses with the same index can both be cached.

**Multiplexer Delay Complexity** A multiplexer is added in the critical path  
 Requires more comparators, and more complexity in placement & replacement.

### 3.3.3 N Way Associative & Block Placement

A generalisation of the directly mapped and two way associative caches. Block placement is restricted by the cache's associativity.

- Increasing associativity reduces associativity conflicts  $\Rightarrow$  better hit rate (with diminishing returns)
- Greater overhead in terms of multiplexers in the critical path and the hardware complexity
- Fully associative cache can place any location in any cache location, and uses parallel search of tag (index is 0 bits) to find entry
- More associative  $\Rightarrow$  less sensitivity to storage layout

Intel Pentium 4 Level 1 Cache (pre-prescott)

*Extra Fun! 3.3.1*

Capacity:	8KB	Block/Line Size:	64B so $8K/64 = 128$ blocks
Ways/Associativity:	4	Sets:	32 (128 blocks, but 4 way $\Rightarrow 128/4$ )
Index:	5 bits	Tag:	21 bits

Resulting access time is 2 cycles (6ns at 3GHz), with cache/memory being dual ported (load and store).

## 3.4 Block Identification

Index and tag identify a block.

- Increasing associativity decreases index size, increases tag size.
- Increasing block size decreases index size.

## 3.5 Block Replacement

When introducing a new location to the cache & possible locations are full.

- No choice in directly mapped.
- $n$  choices for  $n$ -way associative.

The least recently used (LRU) evicts the oldest cache entry

- In practice only a marginal advantage over random eviction.
- Can be pathologically bad (e.g a loop accessing many locations may evict the first just before restarting the loop & accessing again).

## 3.6 Write Strategy

Write Through

**Definition 3.6.1**

On a cache hit, write to cache, and to the block in lower-level memory.

- Combined with write buffers to prevent a wait on memory
- Can always discard cached data, the most up to date is always present in memory
- Only requires a valid bit (cache control metadata)

**Simpler** Cache management is simpler as the most up to date data is always in memory also.

**Sharing** Next level of cache/potentially memory has the most up to date data.

## Write Back

## Definition 3.6.2

On a cache hit, only write back when evicting from cache.

- Track write backs with a dirty bit
- Absorb cost of repeated writes
- Cannot discard cache, when evicting it must be written back to memory
- Cache entries require both valid and dirty bits (cache control metadata)

<b>Bandwidth</b>	Memory is often overwritten several times, with write-back this will only require a memory write back when the cache entry is evicted.
<b>Tolerance</b>	Fewer memory accesses can result in a better tolerance to longer-latency memory (cheaper).

## Write Allocate

## Definition 3.6.3

When a cache miss occurs on write, allocate a new cache line and write to it.

- A read miss is required to fill in the rest of the cache line.
- As only *part* of the line is valid, a valid bit is required per word.

## Write Non-Allocate / Write Around

## Definition 3.6.4

When a cache miss occurs on write, send the data to memory / lower cache level (do not allocate a cache line).

Neither avoid the cache-coherence problem (inconsistent values for locations cached on multiple cores/processors).

## 3.7 Miss Rate Reduction Using Hardware

$$\text{Average Memory Access Time (AMAT)} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

In order to reduce AMAT:

- Reduce Miss Rate.
- Reduce Miss Penalty.
- Reduce time to hit cache.

### 3.7.1 Reducing Misses

**Compulsory** First access so not in cache, also called a *cold start miss* or *first reference miss*.

**Capacity** Cache cannot contain all blocks needed during the execution of a program. A capacity miss occurs when a block discarded due to capacity is later retrieved.

**Conflict** Where the block placement strategy results in blocks being discarded as too many are mapped to a set (associativity conflicts). Also called *collision misses* or *interference misses*.

**Compulsory**

Infinite Cache

**Capacity**

Fully associative, finite cache

**Conflict**

*n*-way associative, finite cache

## Coherence Miss

## Extra Fun! 3.7.1

A miss caused by cache coherence protocols. For example another core or an I/O device may invalidate a cache entry.

### 3.7.2 Increase Block Size

<b>Spatial Locality</b>	Larger block means more locations are speculatively cached.
<b>Cold Misses</b>	As more speculatively cached, fewer cold misses (cached speculatively before first access).

<b>Wasted Space</b>	More of the cache is wasted on speculatively cached but unused data.
<b>Conflicts</b>	larger Blocks $\Rightarrow$ Fewer lines $\Rightarrow$ increase capacity conflicts as there is potentially more contention over lines.
<b>Loading</b>	Larger blocks may take longer to load (increased miss penalty) or require a wider bus (expensive hardware).

### 3.7.3 Increase Associativity

<b>Fewer Associativity Conflicts</b>	More ways $\Rightarrow$ more ways to not conflict. This reduces the miss rate.
--------------------------------------	--

<b>Cycle Time</b>	Comparators in the critical path
-------------------	----------------------------------

### 3.7.4 Victim Cache

The main cache is a large directly mapped cache. A victim cache is fully associative and smaller, and contains data discarded from the main cache.

- Checked in parallel.
- Rarely used for L1 cache, but often used for last-level caches.

This is an example of combining two strategies to avoid both's worst case behaviour.

#### Competitive Algorithms

#### Extra Fun! 3.7.2

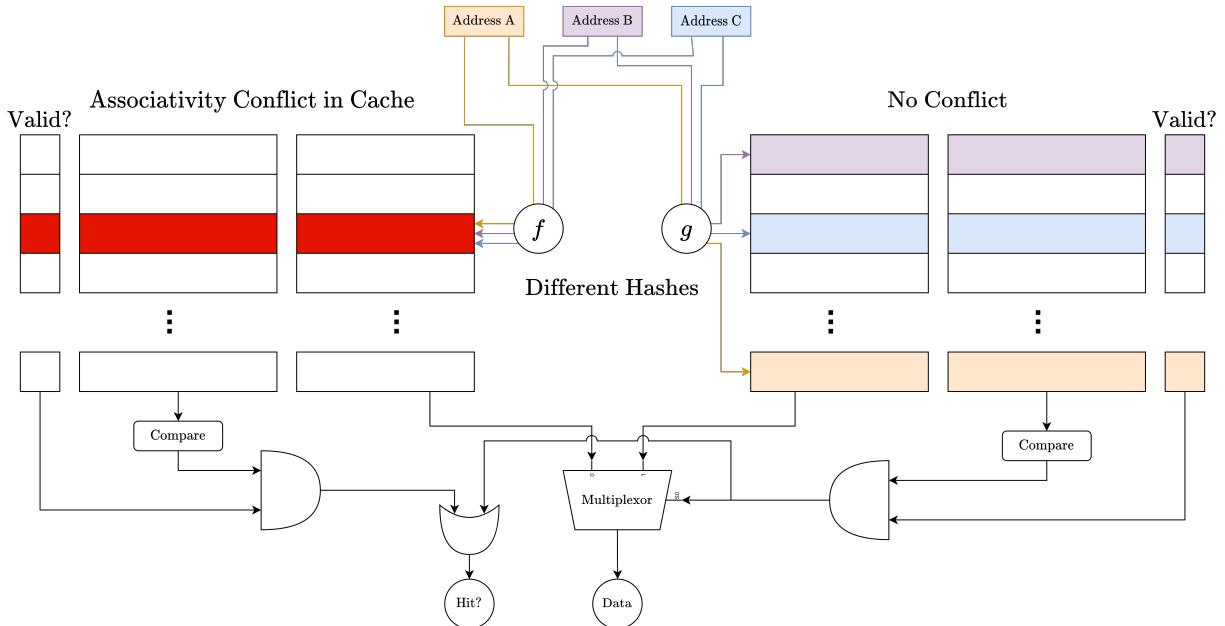
Given two strategies, combining to create a good composite strategy.

- Ski Rental Problem (Combining the renting & buying strategies)
- Spinlocks vs context-switching (e.g spin before blocking?)
- Paging (replacement & eviction)
- 

### 3.7.5 Skewed-Associative Caches

Given a two way set-associative cache.

- Use a different *hash function* for each cache (e.g one using regular (tag and index), other xors bits of index & tag and reorders)
- Associativity conflicts may be present in one cache, but not the other.



### Associativity Conflicts Reduce Associativity

Can reduce associativity conflicts.

Fewer conflicts for a  $w$ -way set associativity means the same conflict rate can be achieved with lower associativity (less hardware complexity).

### More Predictable Average

e.g. with traversing two arrays and a non-skewed cache if we are “unlucky” and get an associativity conflict on one element, we will get it on all subsequent. With skewed the next element may not.

### Some Conflicts

Its very difficult to write a program free of all conflicts.

Need an address decoder per way/cache rather than a single for all.

Complex hash increases latency (it is in the critical path).

Difficult to implement LRU eviction policy (though this is not necessarily a good policy).

### 3.7.6 Hardware Prefetching

When a cache miss occurs, fetch the data (as required), but also pre-buffer the next block in a *stream buffer*.

- Prefetched blocks are placed in the cache (would pollute and potentially evict blocks to be used).
- Stream buffer checked in parallel with cache.
- Can add several prefetch buffers (multi-way stream buffer) to prefetch up  $w$ -way, fetch to  $w$  blocks ahead.
- Used in most high performance processors.

**Sequential Access** Can avoid misses when traversing arrays.

**Cache Untouched** Can use any type of cache design with this - an addition.

**Memory Bandwidth** Need extra bandwidth to transfer block selected (cache miss) and block for pre-fetch.

### Decoupled Access-Execute

### Extra Fun! 3.7.3

Decouple the processor into an access and execution sides.

- Access side fetches data to provide to the execute side.

- Execute side takes data from access and runs arithmetic instructions on it.
- Access side can be far ahead of execute, streaming the required data to it at close to memory bandwidth.

## 3.8 Miss Rate Reduction Using Software

### 3.8.1 Software Prefetching

Many modern processors provide prefetching instructions.

- Rarely needed - hardware prefetching is good!
- Useful on simpler processors with less or no hardware prefetching.
- Care required to prevent unwanted side effects.
- Prefetch instructions may target addresses that result in a page fault/protection violation (here they silently fail).

### 3.8.2 Reducing Instruction Cache Misses

Associativity conflicts can occur in the instruction cache.

- We want to avoid hot loops calling functions who's code have an associativity conflict with each other.
- By using the caller graph, with each loop labelled, we can determine how to pack subroutines into the program binary to avoid associativity conflicts.
- Needs to consider the entire program, and the layout of all subroutines so must be done at link-time.

### 3.8.3 Storage Layout & Iteration Space Transformations

#### Merging Arrays

Improve spatial locality by merging two arrays into a single array of compound elements (i.e a zip). (Struct of Arrays vs Array of Structs)

#### Multidimensional Array Permutation

Match array layout to traversal order.

#### Loop Interchange

Change nesting of loops to access data in order stored in memory.

#### Loop Fusion

Combine independent loops that have the looping behaviour (e.g bounds) and overlapping variables. Sometimes this can then enable *Array Contraction*, where some array can be replaced by a scalar value.

#### Blocking

Improve temporal locality by accessing cache line sized blocks of data repeatedly instead of accessing columns or rows.

A traversal order for blocks.

- Split blocks into four.
- Traverse four blocks in  $Z$  shape, recursively.
- A texture caching layout used in some GPUs.

```
data QuadTree a = Single a |
    Quad {
        topLeft :: QuadTree a, topRight :: QuadTree a,
        bottomLeft :: QuadTree a, bottomRight :: QuadTree a
    }

{-
A----->B
  /
-----
/
C----->D
-}

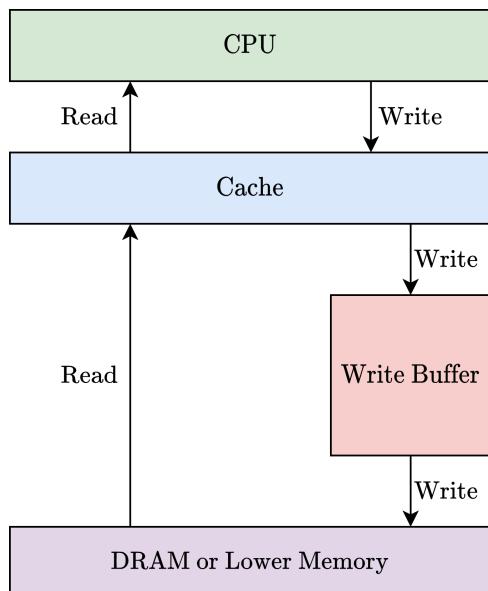
morton :: (a -> b) -> (b -> b -> b -> b -> b) -> QuadTree a -> b
morton fun collect (Quad {tL, tR, bL, bR})
= collect (morton' tL) (morton' tR) (morton' bL) (morton' bR)
where
    morton' :: QuadTree a -> b
    morton' = morton fun collect
morton fun _ (Single s) = fun s
```

## 3.9 Miss Penalty Reduction



### Chapter 3 - Part 3: Reducing Miss Penalty

#### 3.9.1 Write Buffers



- RAW conflict on main memory reads when the location has a write in the write buffer.
- One solution is to wait until the write buffer is empty before reading, however this increases the read cache miss penalty
- A better solution is to check the write buffer on every memory read, if present in the buffer, take the value from there, else go to memory.

With write back we can reduce the stall for a read-miss that evicts a dirty cache line by:

1. Read miss on cache, evict dirty block.
2. Write dirty block to write buffer (fast).
3. Start read, CPU can resume/end stall when read is complete.
4. After read, write from write buffer to memory.

A cache is structured in terms of lines, hence the eviction of a cache entry means an entire line must be written

back to memory.

- Larger memory writes require more time, or more expensive/wider buses.
- The write buffer needs to be large enough to store multiple lines being evicted. Small write buffer will lead to stalls when full.

<b>Coalescing Write Buffers</b>	Adjacent writes are merged into a single entry in the write buffer. This is especially important in write-through caches.
<b>Dependency Checks</b>	Use comparators to check load addresses against pending stores. On a match a dependency is present, so the load must be stalled (other instructions can run).
<b>Load Forwarding</b>	If a store and load match address, forward the data to the load.

### 3.9.2 Early Restart

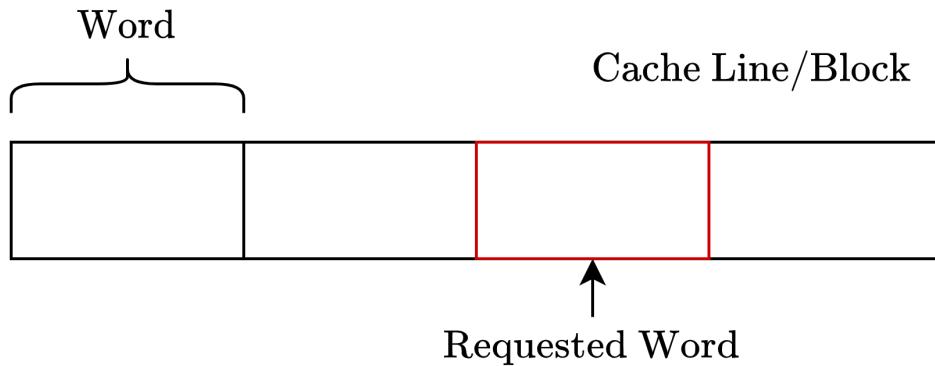
#### Sected Cache Lines

#### Definition 3.9.1

A cache line can be divided into sectors.

- Each has its own validity bit, potentially dirty bit also.
- Cache allocated in units of cache lines
- Data delivered to cache in units of sectors.
- Sectors can be fetched in any order, potentially even remaining invalid until requested.

During a read-miss induced stall, the processor can restart as soon as the requested word arrives.



A cache line consists of many words, when loading a line into cache following a read-miss, we can restart the CPU as soon as the requested word is present.

#### Early Restart

#### Critical Word First

As soon as the requested word arrives, send to the CPU and end stall/restart.  
Request the word on which the read-miss occurred from memory first.

**Complexity** Sectoring and ordering reads increase hardware complexity. Must be careful for edge cases (e.g read-miss on cache entry that is currently in the process of being loaded).

### 3.9.3 Non-Blocking Cache

#### Non-Blocking / Lockup-free Cache

#### Definition 3.9.2

Allows data cache to continue to supply hits for other locations, during a cache miss.

- Requires full/empty bits on registers, and out-of-order execution.
- Requires multi-bank memories

<b>Hit Under Miss miss under miss</b>	Effective miss penalty reduced as useful work is completed during a miss. Misses are overlapped to reduce effective penalty.
---------------------------------------	---

<b>Cache Controller Complexity</b>	Needs to support multiple outstanding memory accesses to support <i>miss-under-miss</i> . Requires extra hardware (e.g multiple memory banks), and complexities of out of order execution.
<b>Fences</b>	Hit under miss allows for load to be serviced out of order, hence a fence/barrier instruction must be available to prevent this when required.

With In-Order pipeline processors, it is possible to implement some of this functionality by effectively making memory accesses out of order only.

- Freeze pipeline in Mem stage, but continue the rest.
- Use full/empty bits on registers, and a MSHR (Miss Status/Handle Registers) queue where each entry tracks the status of an outstanding memory request, a register may be marked as *empty* from a load, it will only stall if still empty when another instruction uses the register, at the decode stage.
- This is a popular approach with in-order superscalar processors.

### 3.9.4 Multiple Cache Levels

$$AMAT = \text{Hit Time}_{L1} + \text{Local Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Local Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

Can continue recursively for  $L3, L4$  etc...

Local Miss Rate	Definition 3.9.3	Global Miss Rate	Definition 3.9.4
$\frac{\text{Misses for this Cache}}{\text{Accesses to this Cache}}$ <p>Misses in a given cache, divided by the total number of memory accesses to the cache.</p> <ul style="list-style-type: none"> <li>• Relevant as a cache may not be accessed often, if the cache a level higher has a high hit rate.</li> </ul>		$\frac{\text{Misses for this Cache}}{\text{Total Memory Accesses}}$ <p>Misses in a given cache, divided by the total number of memory accesses.</p>	

Multilevel Inclusion	Definition 3.9.5
Where lower caches contain all entries in the higher caches.	$d \in L1 \Rightarrow d \in L2 \Rightarrow d \in L3$

- Inclusion strategy affects placement of data, and hence cache controls on coherence, search (should lower caches be checked?) etc.
- Can use the L2 cache to filter coherence protocol invalidations. If not in L2, no need to check L1, and hence no need to impact bandwidth to L1.
- L3 / LLC (Last Level Cache) often managed as a victim cache, only allocated when data is displaced from L2.

## 3.10 Hit Time Reduction



### Chapter 4 - Part 4: Hit Time Reduction and Address Translation

#### 3.10.1 Parallel Cache Access

When attempting to issue multiple instructions per cycle, parallel accesses to the cache are required.

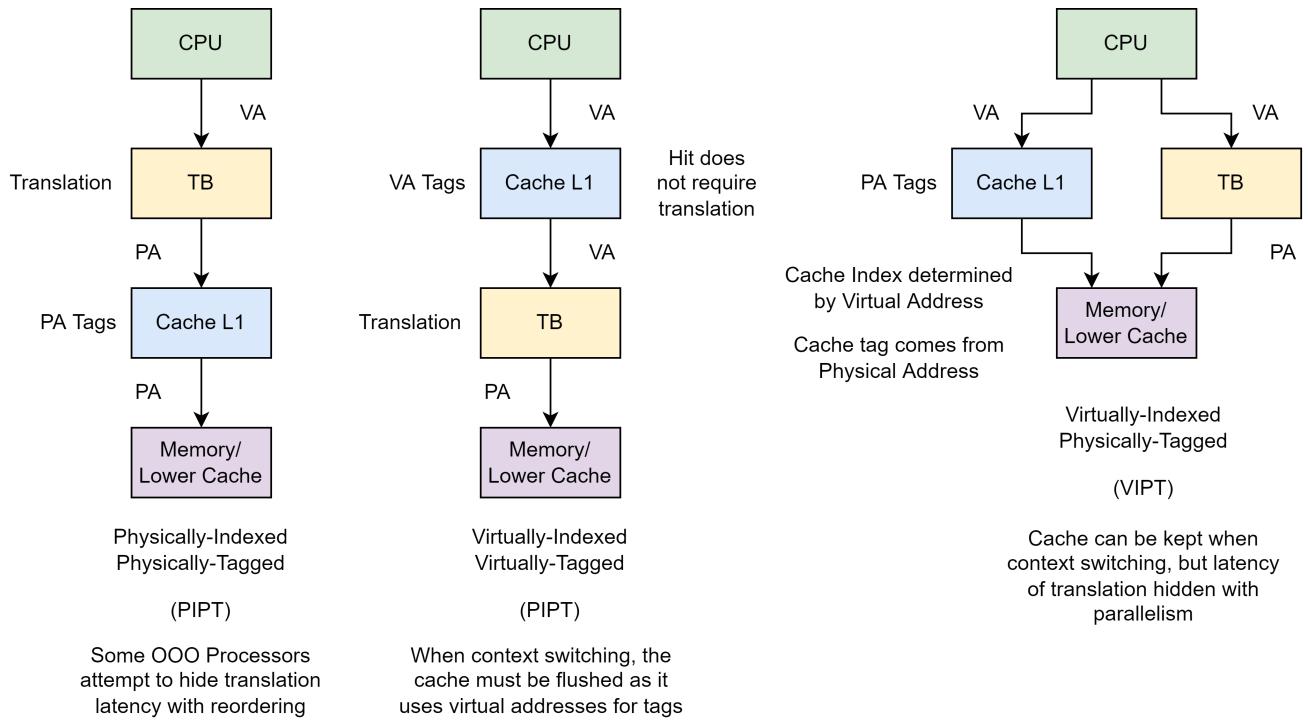
### Multiple Banks

Divide the cache into several banks, with addresses mapped to banks (e.g using low order bits, or a hash function). Accesses to different banks can occur in parallel.

### Duplicate the Cache Multi-ported RAM

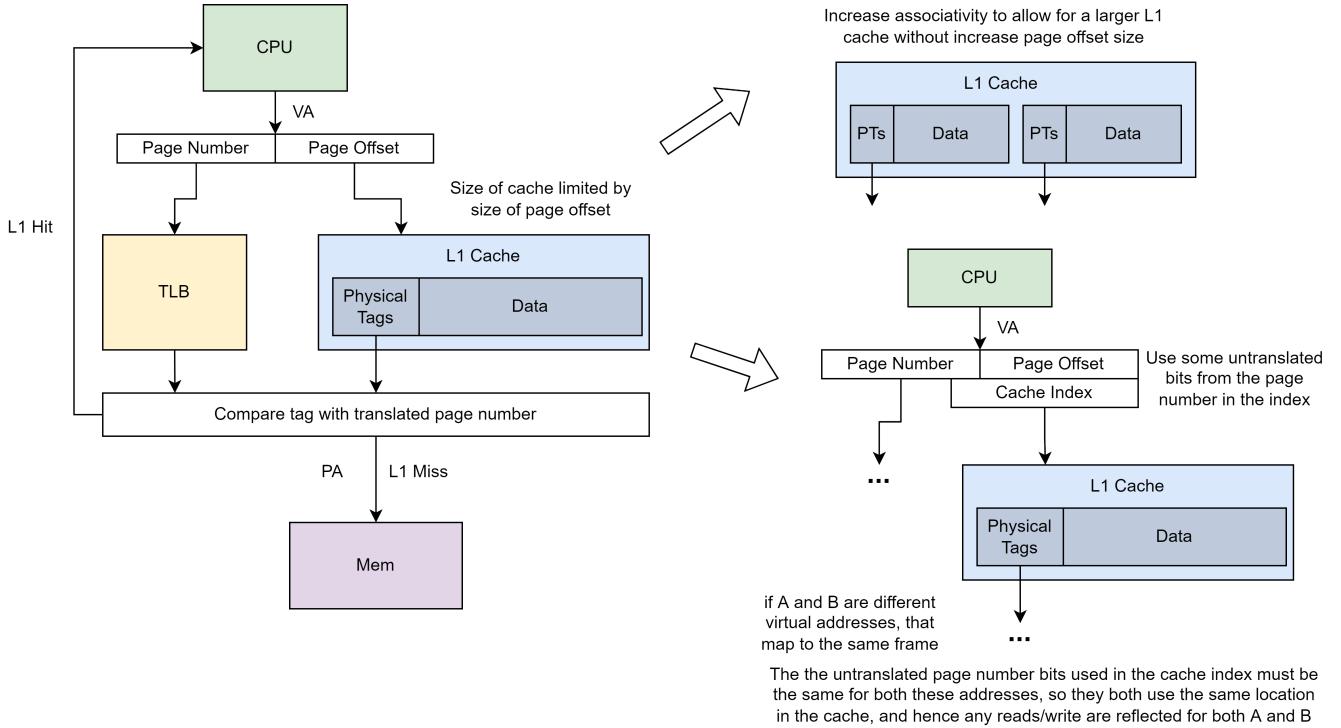
By using multiple copies of the same cache, each can be accessed separately. Add another wordline per row, and another bitline per column to allow multiple accesses to the RAM at the same time, cache uses this multiported RAM. This is effectively duplicating the cache, but sharing the flipflops between caches.

### 3.10.2 Address Translation



Homonyms	Definition 3.10.1	Synonyms	Definition 3.10.2
<p><i>Same sound, different meaning</i></p> <p>The same virtual address can point to different physical locations, in the context of different processes.</p> <ul style="list-style-type: none"> <li>• A virtually indexed cache, as the tags are compared to determine hit/miss, we must flush the cache.</li> <li>• In a TLB, where the cache is necessarily virtually tagged &amp; indexed, a flush must occur, unless some process identifier is included in the tag (e.g ASID - Address Space ID).</li> </ul>	<p><i>Same meaning, different sound</i></p> <ul style="list-style-type: none"> <li>• Multiple virtual addresses point to the same physical.</li> <li>• In a virtually indexed cache multiple addresses may map to the same page.</li> <li>• Updates to one cached copy must be reflected in the others (shared pages)</li> </ul>		

## Faster Cache Hits by Avoiding Translation



## Linux MMap

### Extra Fun! 3.10.1

`mmap` in linux does not specify the address in order to allow the operating system to determine this. This allows the OS to perform system specific tricks such as ensuring the page number bits in the cache index are identical for virtual addresses mapped to the same frame.

The L2 cache makes use of physical addresses, and hence relies on the virtual-to-physical mapping provided by the OS.

- The OS may choose mappings that result in associativity conflicts in the L2 cache.
- This means that different instances of the same program, with the same computation can have different performance, as the OS may assign frames such that an associativity conflict occurs.

Operating systems can also use different page sizes.

- large pages require smaller page tables, and use fewer TLB entries, which increases the hit rate for the TLB.
- Requires support in hardware for multiple page sizes.

# Chapter 4

## Dynamic Scheduling



### Chapter 2 - Part 1: Tomasulo

### 4.1 Bypassing Stalls

The basic concept behind out of order scheduling is that instructions behind a stall can be allowed to continue provided data dependence/hazards allow.

- When an instruction stalls (e.g cache miss or forwarding not possible) save the state of that instruction.
- Instructions are issued in order, have dependencies analysed and can then be executed out of order.
- When operands are available allow execution of the stalled instruction to continue.

#### Read After Write / True Dependence

#### Definition 4.1.1

```
add $3, $2, $1 # $3 = $2 + $1 (Write $3)
sub $4, $3, $6 # $4 = $3 - $6 (Read $3) (needs previous instruction's value)
```

The output of one instruction is required as the input to another.

#### Write After Read / Anti Dependence

#### Definition 4.1.2

```
sub $4, $3, $6 # $4 = $3 - $6 (Read $3) (use $3 before the next instruction overwrites)
add $3, $2, $1 # $3 = $2 + $1 (Write $3)
```

Some instruction will overwrite an input to a preceding instruction.

#### Write after Write / Output Dependence

#### Definition 4.1.3

```
add $3, $2, $1 # $3 = $2 + $1 (Write $3)
sub $3, $4, $6 # $3 = $4 - $6 (Write $3)
addiu $7, $3, 100 # $7 = $3 + 100 (Read $3)
```

The writes have a dependency as they write to the same location, the correct value must be present in the location for subsequent reads.

### 4.2 Tomasulo's Algorithm

An out of order execution algorithm used to dynamically rename registers to bypass the limited number of floating-point registers in the IBM architecture specification, and allow faster computation on the IBM 360/91.

- Each register contains a tag. (null means the value is present, otherwise it is the identifier of the unit the result will come from)
- By adding tags register renaming (simple) is achieved

- A common data bus is used to broadcast the result of an operation, with its tag (unit it came from)

```
"""Super abbreviated pseudocode for the IBM360/91 Out of Order Execution """
class IBM36091:
    def issue_instruction(instruction: Instr):
        # for each argument, check if register value is present or waiting.

        unit: FunUnitId = get_unit_from_opcode(get_opcode(instruction))
        dest: Register = get_dest_register(instruction)
        operands: List[Register] = get_operands(instruction)

        # overwrite destination with new unit to take result from (all subsequent instructions use result
        dest.set_tag(unit)

        # Get arguments (some from registers, some )
        args: List[ArgType] = []
        for op in operands:
            if (tag := op.get_tag()) is not None:
                args.append(WaitFor(tag))
            else:
                args.append(Value(op.get_register_value()))

    class Unit:
        def broadcast(data):
            # Broadcast data to registers and other functional units via the common data bus
            common_data_bus.broadcast(self.unit_id, data)

        def receive(unit_id, data):
            # Given some data broadcast determine if it is needed, and if any instructions can be execute
            for instr in self.reservation_station:
                # Check if an instruction is waiting for the tag
                instr.take_args(unit_id, data)
                if instr.is_ready():
                    instr.queue_execute()

    class Register:
        def receive(unit_id, data):
            # Check if the data broadcast is for the register.
            if self.tag == unit_id:
                self.tag = None
                self.value = data
```

**Complexity** Led to delays in design, hardware overhead to overcome an ISA issue.

**Limited by CDB** CBD must go through all functional units, and only one instruction can write to bus per cycle.

**No Precise Interrupts** As instructions are executed out of order, we cannot clearly define a point in the *in-order* program text where the processor is at at any given time.

It is possible to overlap loop iterations:

- (Effectively) Register renaming allows for different physical destinations (e.g ignore register and straight to functional unit).
- Reservation stations can buffer old values to avoid write after read / anti dependence stalls.



## 4.3 Precise Interrupts

In order to use precise interrupts we need a consistent state.

- All instructions up to some point have committed changes to machine state (registers & memory).
- No instructions past have committed.
- Hence on an interrupt (e.g page fault, syscall) we can easily save state, and restart where the interrupt suspended execution of a program.
- This is also important for branches (need to undo prevent committing work executed speculatively)

Hence we want to make a *speculative tomasulo algorithm*

1. Issue/Dispatch (Get instruction from buffer of fetched instructions, send operands & reorder buffer number to destination)
2. Execution (Out of order execution of issued instructions)
3. Write Back (in order to common data bus and waiting functional units)
4. Commit (Update register with reorder result, reorder buffer takes completed instructions, puts in issue order and updates state)

This requires several additions

- Commit unit to manage reorder buffer
- Issue side registers for execution
- Commit side registers for the committed results
- Ability to flush the reorder buffer on a branch mispredict

## 4.4 Store Buffering

Stores are an issue as they cannot be completed until committed, but succeeding loads can be executed straight away.

- We could stall all preceding loads until the store is complete
- We can buffer uncommitted stores, associated with addresses, and check these for any load (to get the nearest hit, or on miss go to memory). Loads must be stalled until all possibly aliasing store addresses are resolved

Loads and stores use computed addresses (not always known at issue time)

- Can speculate, and forward a store's result to a load
- Must recover when the computed address is not the speculated

Hence we can add a *forwarding predictor* to determine if a store should be forwarded to some load behind it in the pipeline.

### Dependence Prediction

### Extra Fun! 4.4.1

More can be read about predicting the dependence of a load on another store instruction  
<https://jilp.org/vol2/v2paper13.pdf>.

## 4.5 Register Update Unit

An alternative to reservation stations and the reorder buffer.

- A single table of instructions after fetch, acting as a reservation station.
- Once the operands are found, the instruction can be issued (hence functional unit determined after operands, unlike in Tomasulo's)
- RUU entries are committed to update the commit side registers.

- |                 |   |
|-----------------|---|
| <b>Monitors</b> | In Tomasulo's every reservation station and reorder buffer entry needs to have a comparator and monitor the common data bus. With the RUU strategy, fewer comparators are required. |
| <b>Tags</b>     | With RUU the tags are ROB entries. Furthermore the RUU is indexed by the tag.   |

## 4.6 Register Alias Tables

**UNFINISHED!!!**

# Chapter 5

## DRAM



Chapter 4 - Part 5: Main Memory

**UNFINISHED!!!**

# Chapter 6

## Side Channels



### Chapter 5 - Part 1: Sidechannel Vulnerabilities

#### Side Channel Attack

#### Definition 6.0.1

An exploit that attacks the implementation of an algorithm by observing the state of system it runs within. For example:

- Detecting what is cached through memory access times
- Power consumption, electromagnetic leaks and other physical effects

## 6.1 Exfiltration

#### Prime and Probe

#### Definition 6.1.1

Detect eviction of the attacker's working set, that is caused by the victim.

1. Attacker primes the cache by filling some sets with its own lines.
2. Victim executes, once finished the attacker probes (timing its memory accesses) to see which of its lines were evicted.

If a line has been evicted, then the victim accessed an address mapping to the same set.

- Requires the attacker to be able to force the start of victim execution.

#### Evict and Time

#### Definition 6.1.2

Detecting cache usage by the victim, by monitoring its performance after altering the cache.

1. Attacker causes victim to execute, and to preload the cache with its working set.
2. Attacker evicts a specific line from the cache.
3. Victim executes, attacker monitors execution time.

By repeating this process for many lines, the attacker can see where the time to execute is lower, and hence which lines the victim was using.

- Requires the attacker to be able to force the start of victim execution.

## Flush and Reload

## Definition 6.1.3

Given the victim runs in the same address space. Flush the cache, let the victim run and study which lines it accessed.

1. Flush the shared line of interest (using dedicated instructions, or through contention by caching dummy data)
2. Allow the victim to execute, the victim will then load any data it uses.
3. Attacker then reloads the evicted line through an access, and measures time taken.

A fast reload indicates the victim used that line.

## 6.2 Shared State

On a modern CPU a large amount of state is shared between cores.

- L2 and lower Cache
- core interconnects/buses

If two threads run on the same core (as with SMT, or HyperThreading):

- Instruction and data caches, as well as TLB
- Branch predictor
- Prefetcher
- Rename Registers
- Dispatch Ports

## 6.3 Triggering Victim Execution

<b>System Call</b>	If the operating system can invoke the victim, or the operating system itself is the victim.
<b>Lock Release</b>	A victim may be waiting on a held lock (e.g a lock on a file in the filesystem)
<b>Call it</b>	If the victim is contained in the same address space, call it as a function.

The latter point ("call it") applies as the attacker's code may be running in the same process as the victim, under some runtime system (e.g the JVM, or a browser's javascript engine).

## Language Based Security

## Definition 6.3.1

The runtime system bundled with and provided by the language enforces separation between threads.

- i.e prevents pointer arithmetic, checks array index bounds

## 6.4 Side Channels in Speculative Execution

Speculative execution of instructions can impact the state of the cache.

- Can include instructions in code, that will be speculatively executed and as a result impact cache.
- e.g Bypassing array bounds checks using the CPU's speculative execution of the invalid instruction to modify cache.

This vulnerability is commonly known as *Meltdown & spectre* and is present in most modern dynamically scheduled processors.



## 6.5 Mitigation

<b>ASLR</b>	Address Space Layout Randomisation is where the operating system distributes user pages across memory. It can also randomly distribute the kernel pages locations within a process' address space.
<b>KPTI</b>	Kernel Address Space Isolation is where the virtual address mapping of kernel pages is changed every time the kernel is entered. This requires a reload of the TLB and a substantial performance penalty.

## 6.6 Spectre v2

Even with these mitigations, it is still possible by training the branch predictor to mispredict a call within the victim's code.

1. Train the branch predictor to predict jumps to your subroutine. (Need to consider design of the BTB)
2. Make the syscall to the kernel.
3. Kernel code starts running, hits a call that is mispredicted to the attacker's code.
4. Attacker's code speculatively executes before being flushed, but effect on cache, size channels is still present.

This is called *Spectre Variant 2*

There are several possible mitigations:

<b>Block Microarchitecture and Cache Side Channels</b>	Very difficult to impossible.
<b>Reduce Accuracy of Probing</b>	Need to add noise to timers, but this will now affect other applications that need precision timing.
<b>Prevent Branch Predictor Poisoning</b>	Can add an instruction to prevent branch prediction, but there is a performance cost.
<b>Block Branch Predictor Contention</b>	Keep separate predictions for each thread protection domain (ring/protection level).

Another solution is *retpoline* - an indirect branch using a return instruction.

```
RP0:  call RP2          ; Push address of
RP1:  int 3             ; A breakpoint (will be speculatively executed, but is a mispredict)
RP2:  mov [rsp], <jump target> ; overwrite the return address, now return will go to <jump target>
RP3:  ret               ; jump to jump target
```

# Chapter 7

## Static Scheduling

Dynamic scheduling (out of order scheduling) requires significant hardware complexity.

- Register Update Units, reorder buffers, registers backed by commit registers and associated with tags, instruction dependency checks.
- All of these take space on the die (not only does a larger chip necessitate fewer chips per wafer, but the yield is also decreased)
- Also requires more energy, results in more heat and hence lower thermal limits.

With static scheduling this complexity is removed from hardware, and moved to the compiler, with the ISA providing necessary mechanisms to express how instructions should be scheduled (e.g in parallel).

# Chapter 8

## Credit

### Image Credit

**Front Cover** Intel i386 die shot by Pauli Rautakorpi on wikimedia here.

### Content

Based on the architecture course taught by Prof Paul Kelly.

These notes were written by Oliver Killane.