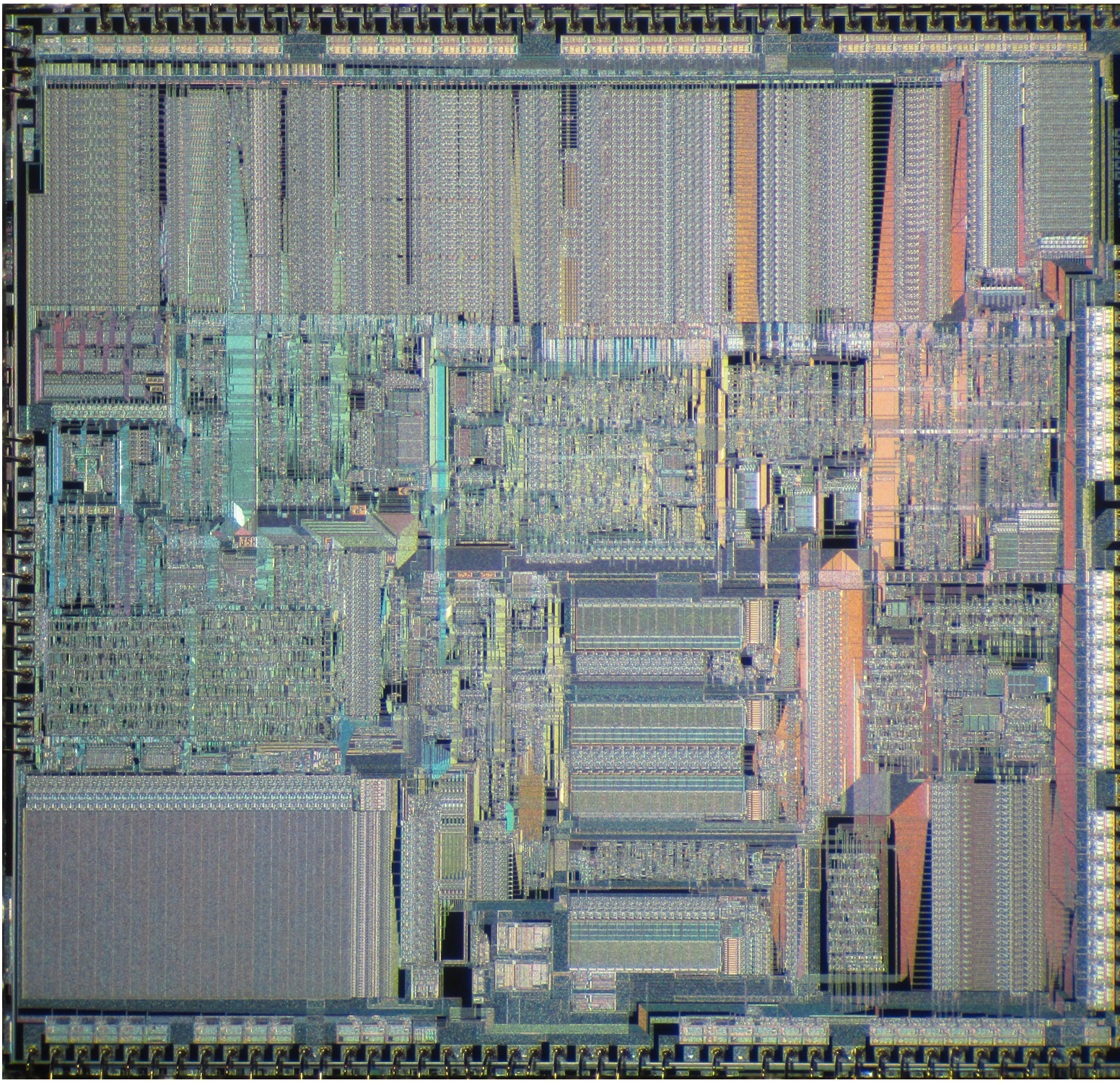


60001

Advanced Computer Architecture

Imperial College London

Contents

Chapter 1

Introduction

1.1 Course Structure and Logistics

Prof Paul Kelly

Teaching the entire course.

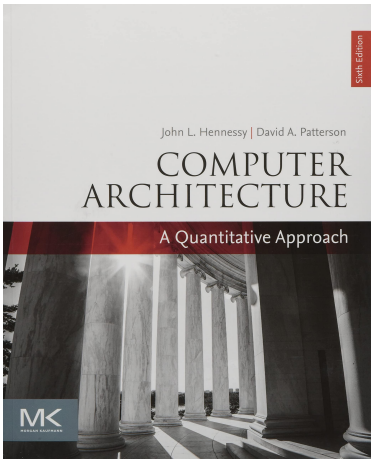
- Microprocessor design.
- Optimising software for hardware, and compiler design.
- Optimising hardware for specific software tasks.
- Challenges past, present & future.

Taught through pre-recorded lectures and live tutorial sessions.

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^{th} Edition)

Chapter 1 - Part 1: Introduction

Chapter 2

Pipelining

Chapter 1 - Part 2: Pipelines

MIPS/Microprocessor without Interlocked Pipelined Stages **Definition 2.0.1**

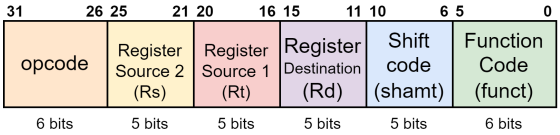
MIPS is a reduced instruction set (RISC) architecture originally developed for the R2000 microprocessor.

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

2.1 Instruction Layout

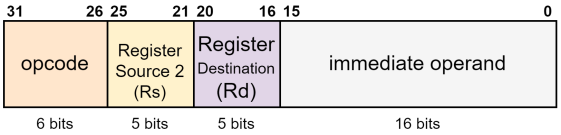
The instructions set architecture (ISA) determines the layout of instructions. Here we consider the mips architecture.

Register Type



5 bit register specifier limits MIPS to 32 registers

Immediate Type



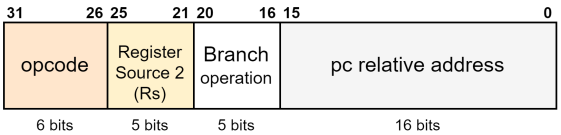
Opcode specifies how the fileds will be interpreted by specifying the instruction type

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

Jump Type

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

Branch (Immediate Type)



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

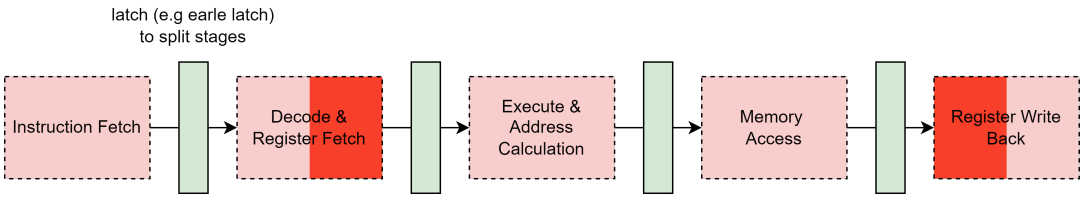
The size of fields in the instruction layouts determines characteristics such as:

- 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 $1/\text{number of stages}$ (idealistically)
- 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**

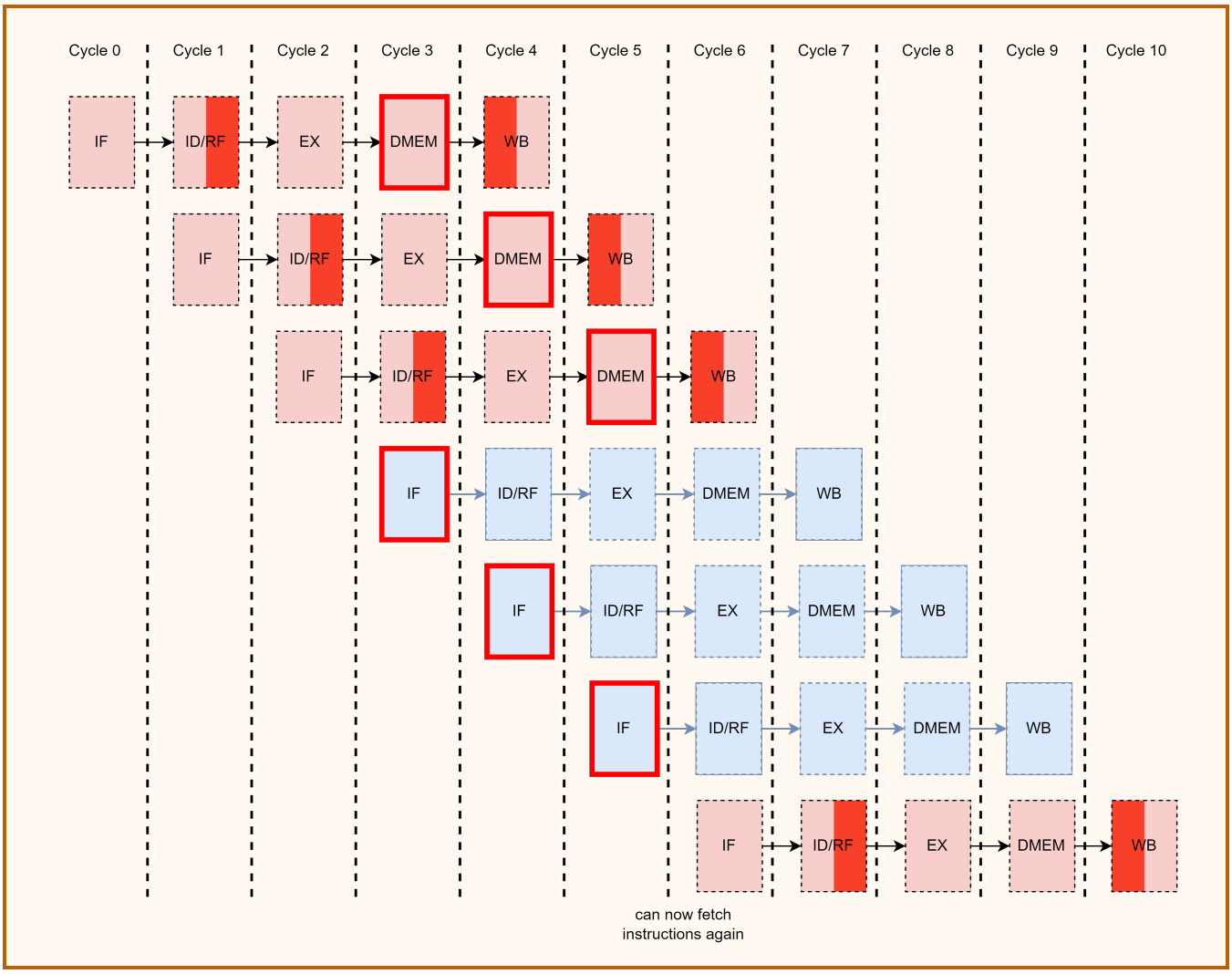
Given basic pipeline structure above, what structural hazard could occur between *Memory Access* and *instruction fetch* if there is only one RAM port?

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×, 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× speedup from this pipeline. However we are only getting a 2.5× 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.

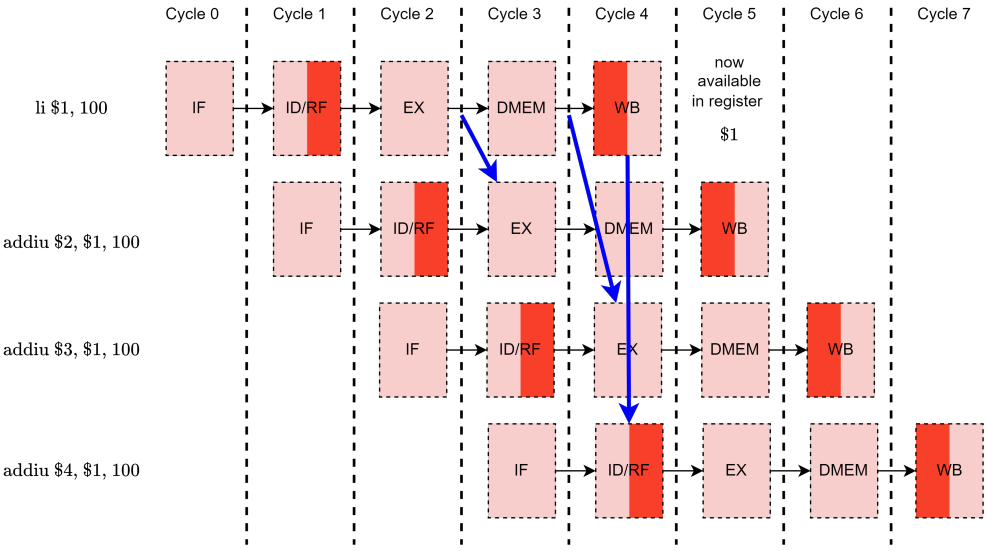
Most often caused by a dependency between instructions.

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.

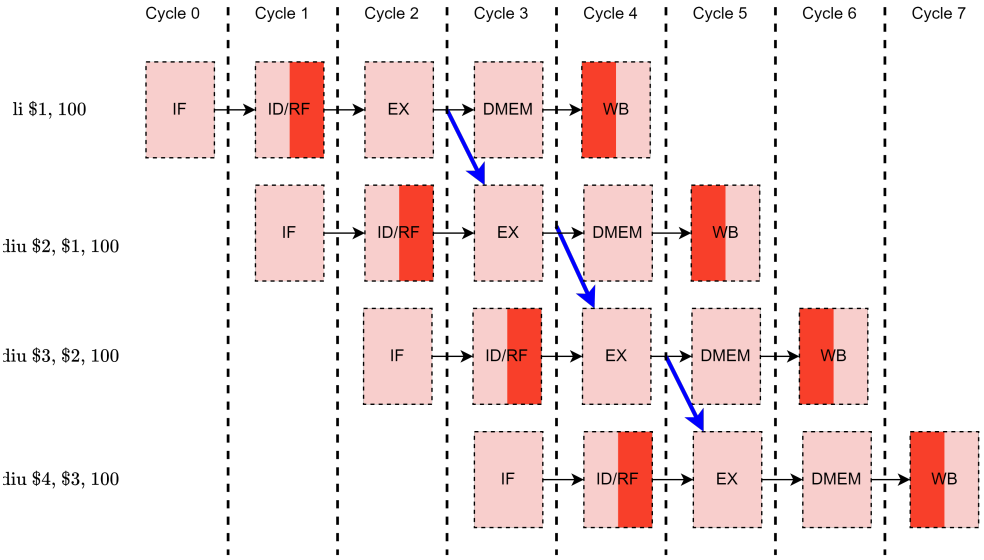
Result Used By Many Subsequent Instructions

```
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

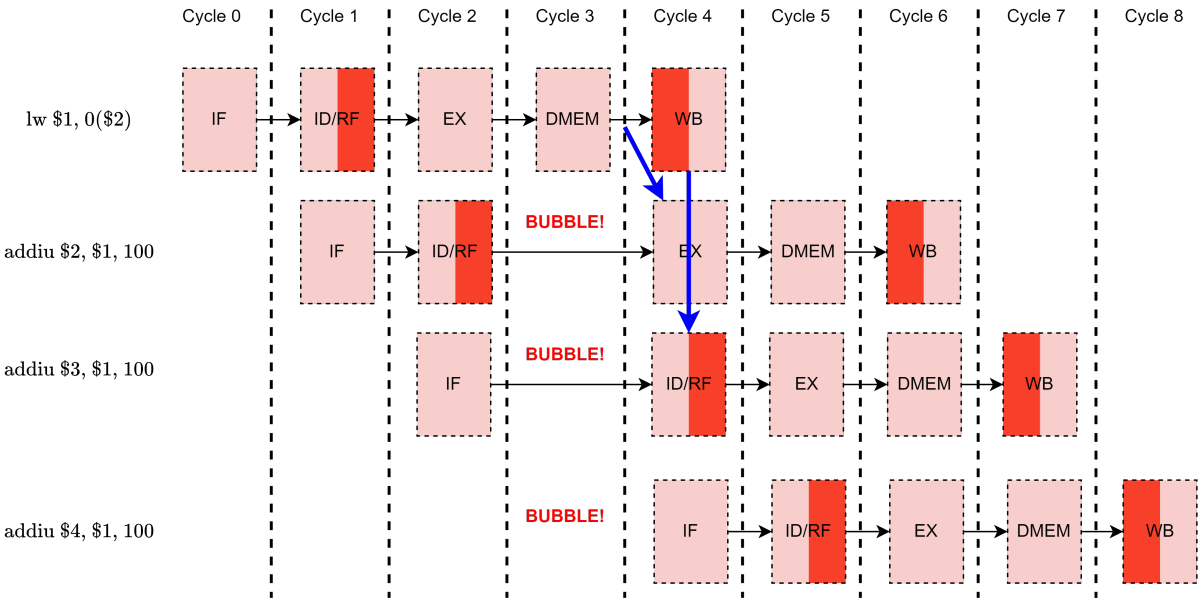
```
li    $1, 100      # $1 = 100
addiu $2, $1, 100  # $1 += 100
addiu $3, $2, 100  # $1 += 100
addiu $4, $3, 100  # $1 += 100
```



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

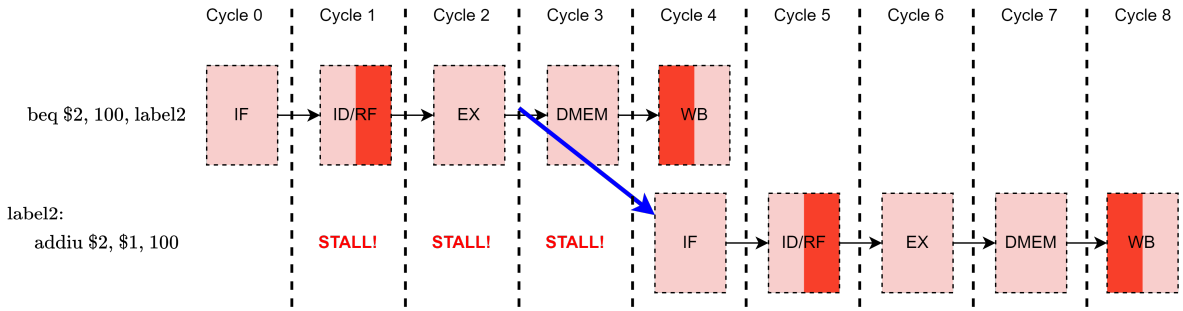
2.3.3 Control Hazard

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).

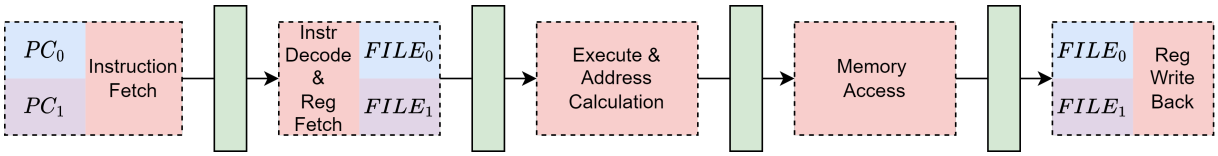


Early Branch Determination

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 $5 \rightarrow$ 9GHz
- Limited by critical path through slowest pipeline stage
- Clock period is $330ps \approx 10$ gate delays at $3GHz$ ($3 \rightarrow 5$ 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**

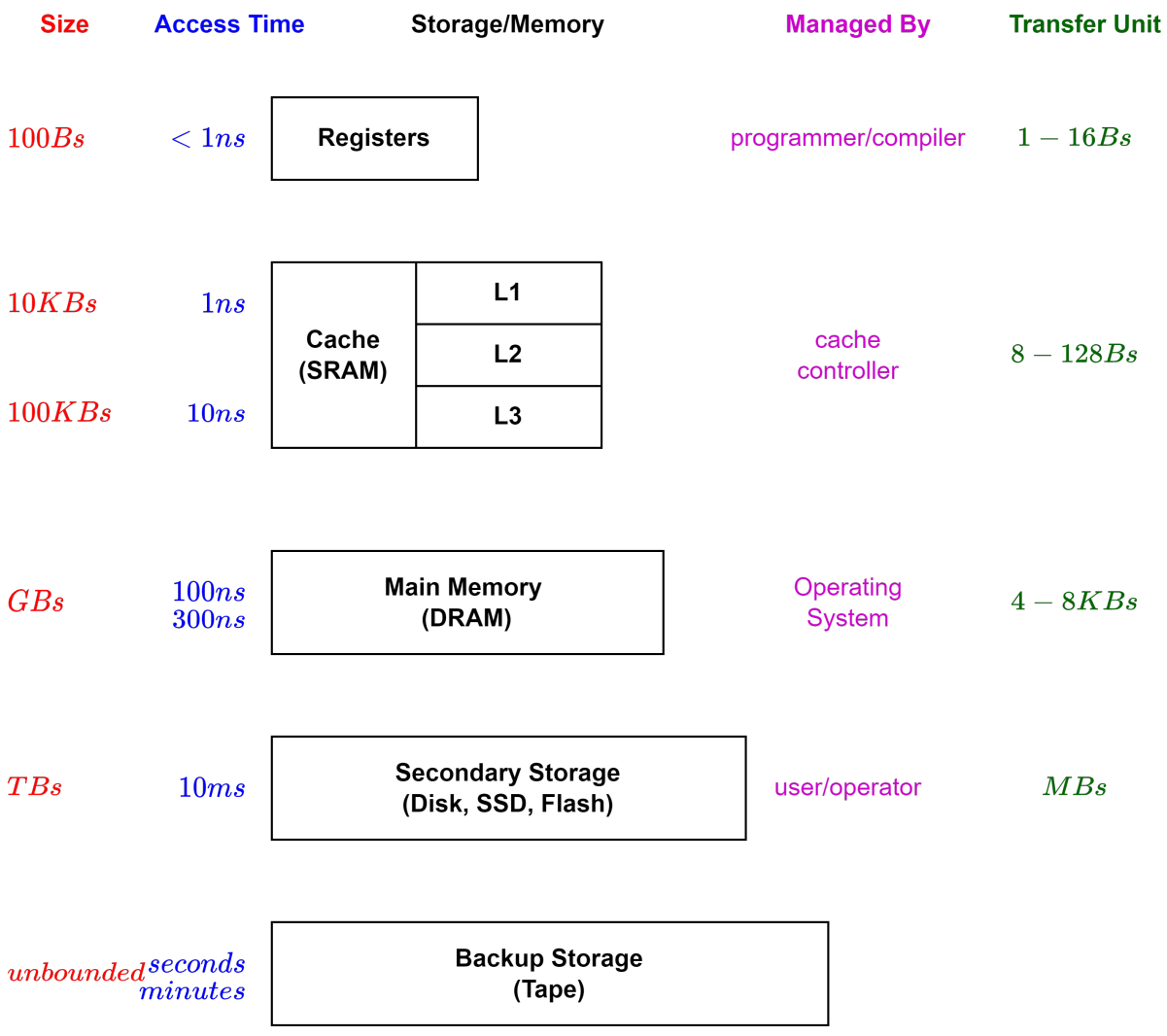
The gate delay of a component with a fan-out (gate inputs driven by a gate's output) of 4.

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.



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.

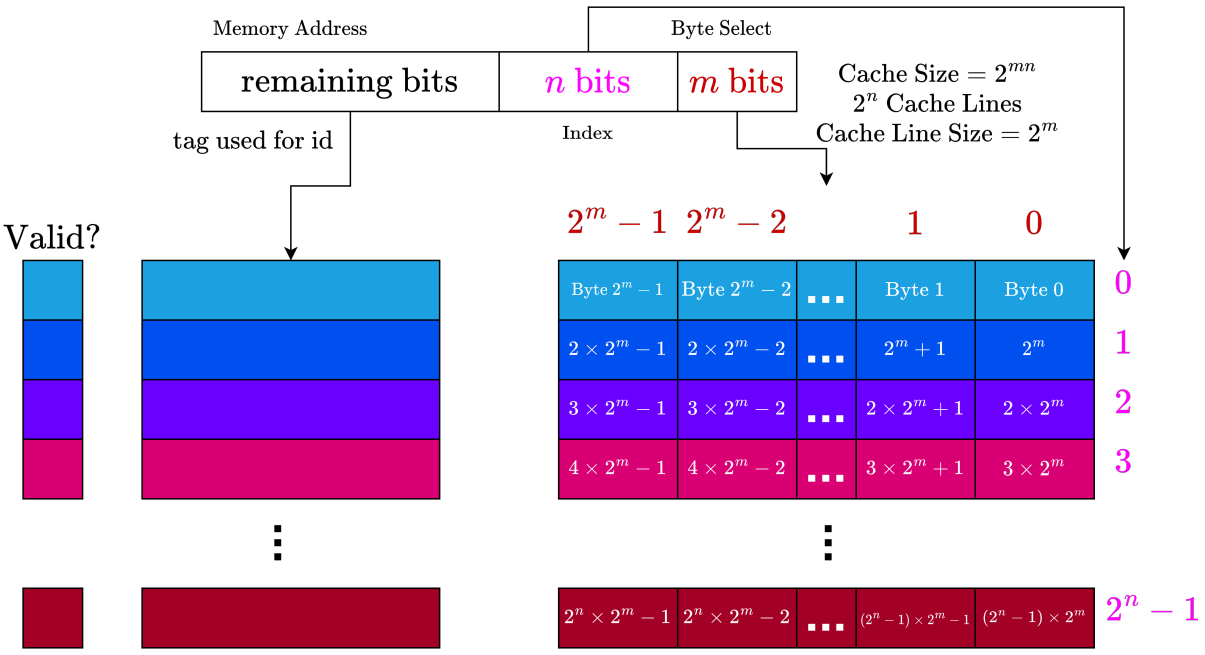
3.3 Cache Types

3.3.1 Directly Mapped Cache

Associativity Conflicts — **Definition 3.3.1**

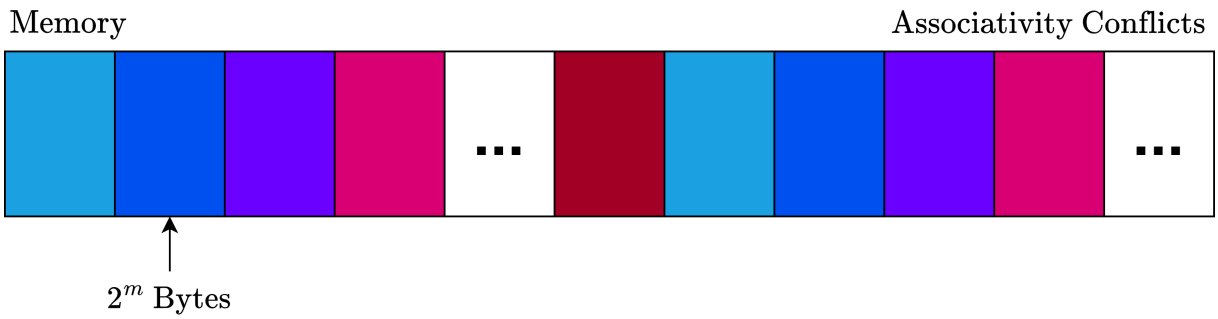
Where two or more locations are mapped to the same cache line/set of cache lines, and repeatedly replace each other.

```
/* 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 < 64; i++) {
        r +=
        array_a[i]    /* array_a moved into cache line */
        + array_b[i]; /* array_b evicts array_a and replaces */
    }
    return sum;
}
```



- 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).

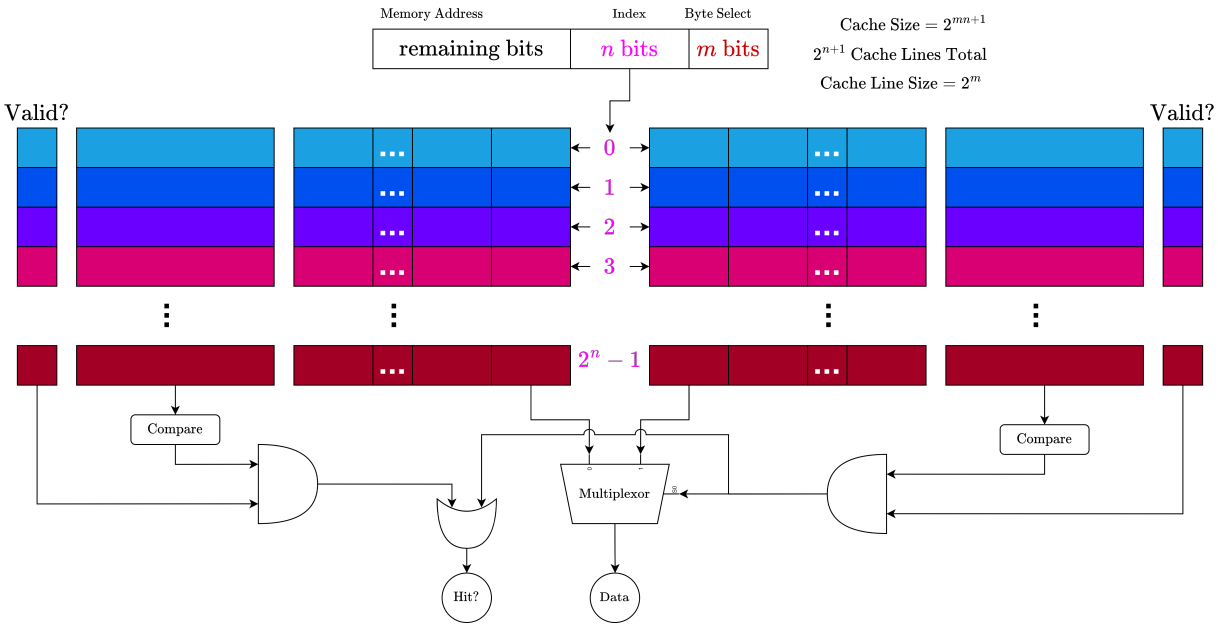


Simplicity	Simple indexing of cache & compare to determine hit/miss.
Fast Lookup	Only one location where a cached value may be.

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	A multiplexer is added in the critical path
Complexity	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 ⇒ 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 ⇒ 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 ⇒ ${}^{128}/_{4}$)
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**

Write to cache, and to the block in lower-level memory.

- Combined with write buffers to prevent a wait on memory

Write Back **Definition 3.6.2**

Only write back when evicting from cache.

- Track write backs with a dirty bit
- Absorb cost of repeated writes

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

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 pof 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 registers 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 recieve(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 recieve(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.

Chapter 2 - Part 2: Speculation

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

UNFINISHED!!!

Chapter 5

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.