week 6 - executing MIPS instructions and hardware parallelism in a single core processor

Overview

This week we look at how a MIPS processor actually executes instructions. We then overview how instruction throughput has been increased between 1980s and 2000s, by relying on Instruction Level Parallelism (ILP) and memory hierarchy (in particular caches). In doing so, we analyse the hardware concurrency hazards raised by those innovations, and how such hazards are addressed within processors.

References

Essential readings**: P&H 4.1, 4.5, 5.1, 5.2**
Further readings: P&H 4.2, 4.3, 4.4, 4.6, 4.7, 4.10, 5.3

P&H - How to develop a **pipelined** MIPS implementation

Chapter 4.1

For every MIPS instruction, the first two steps are identical:

1. send PC to memory that contains the code and fetch the instruction from that memory
2. read one or two registers, using tfields of the instruction to select the registers to read. For the load word instruction, we need to read obnly one register, but most other instructions would require reading 2 registers.
   The next action required to complete the instruction depends on the instruction class (e.g. memory-reference, arithmetic-logical, branches, jump)

All instruction classes apart from jump instructions uses the ALU

- memory reference uses ALU for address calculation
- arithmetic-logical for operation erxecution
- branches for comparison

Chapter 4.5

Pipelining

What is pipelining?

https://www.youtube.com/watch?v=**zPmfprtdzCE - this video explains it well using the textbook (watch 1 3 1 - 1 3 6 which covers chapter 4.5)

- **implementation technique in which multiple instructions are overlapped in execution in parallel**, just like an assembly line
- takes less time as instructions are run concurrently

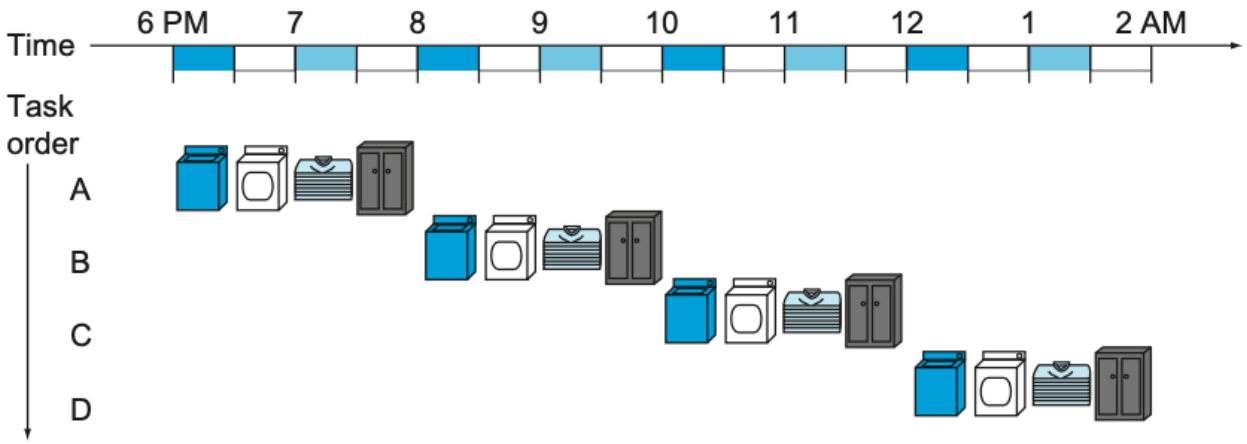
Tasks can be pipelined if there exist separate resources for each stage (in the pipelining process)

Example - using laundry

Task 1: place 1 load to clothes in washer
2: place wet load to dryer once washer is done
3: once dryer finished, place dry load on table and fold
4: when folding is finished, ask roommate to put the clothes away

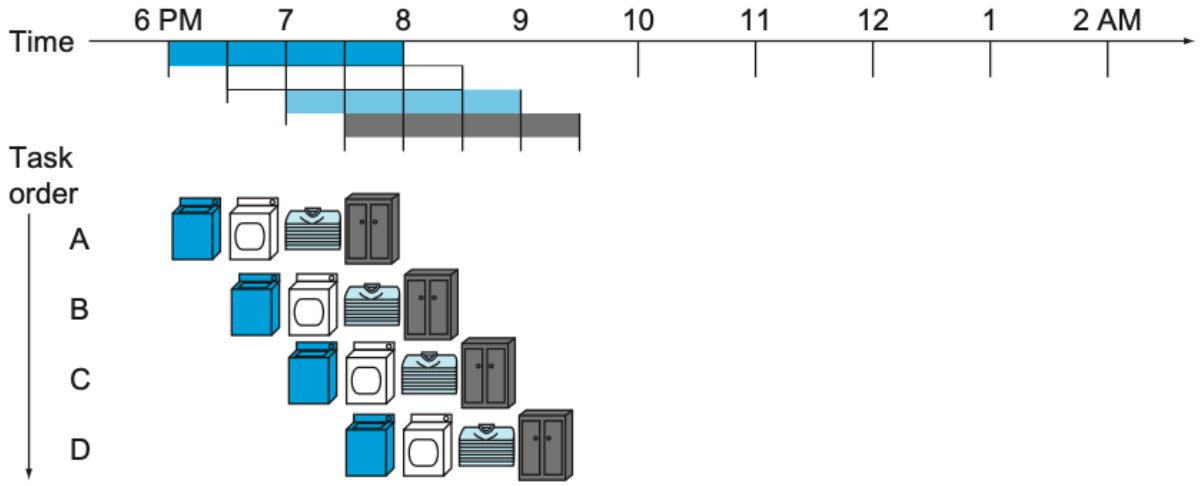
Pipelining vs No Pipelining

No pipelining approach:



- Time taken: 8 hours for 4 loads of laundry

Approach with pipelining:



- Tiime taken: 3.5 hours to do 4 loads of laundry (total of 4.5 hours saved)
- In this example, all steps (called stages in pipelining) are operating concurrently - we can pipeline tasks provided we have separate resources for each stage
- IF all the stages take same amount of time and there is enough work to do, the speedup due to pipelining would be equal to number of stages in the pipeline

Why pipelining?

- More efficient as everything is working in parallel
- Improves throughput (number of instructions per time) and decrease total time to complete work (like the laundry for example) even if all stages take about the same amount of time.

The same principles apply to processors where we pipeline instruction-execution.
MIPS instructions classically take five steps:

MIPS pipelining
5 stages in MIPS pipeline:

1. IF - Fetch instruction from memory.
2. ID - Read registers while decoding the instruction. Th e regular format of MIPS instructions allows reading and decoding to occur simultaneously.
3. EX - Execute the operation or calculate an address.
4. MEM - Access an operand in data memory. (Load from memory to register/Store register value to memory)
5. WB - Write the result into a register.

Note that pipelining means these stages won't get executed at same time but an instruction is broken into 5 stages and are overlapped - these can happen even in a single core processor

(ILP - instruction level parallelism but not true parallelism as it only can happen in multicore processor). once the first stage is done, next instruction starts first stage while the current instruction proceeds to next stage, and so on (good to understand the datapath)

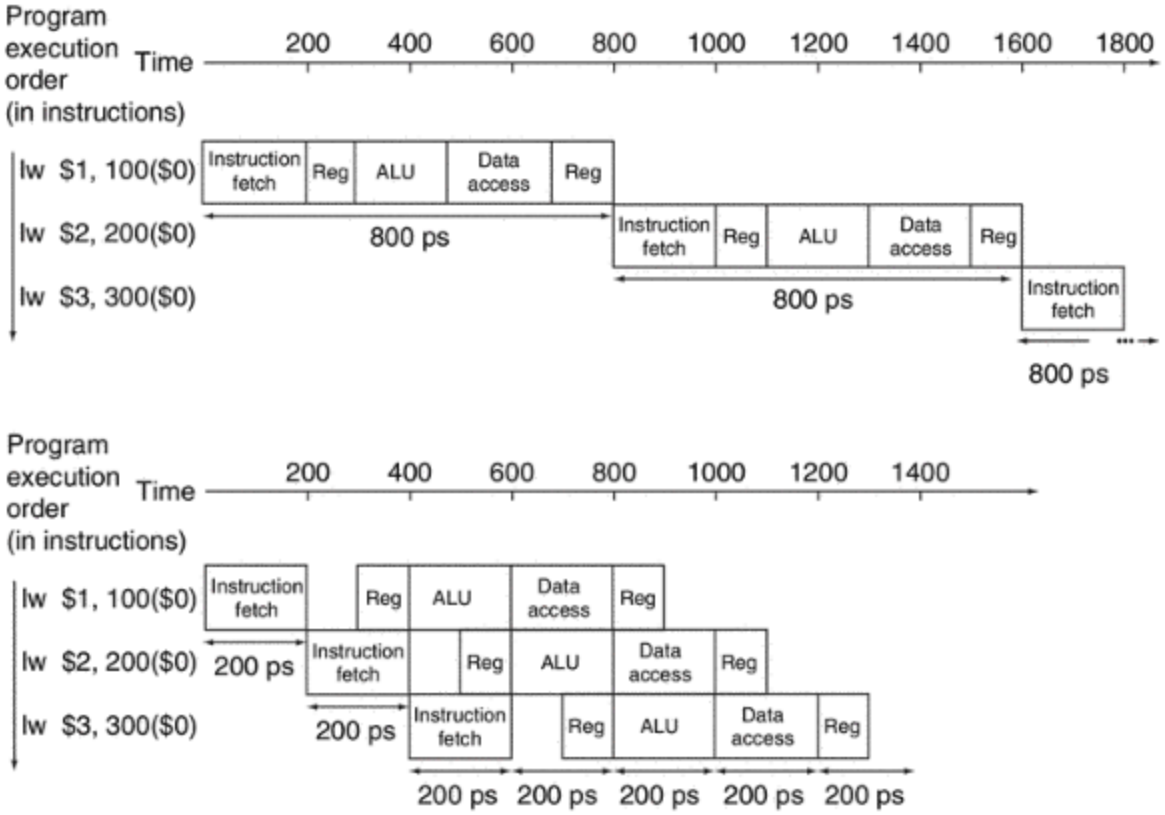
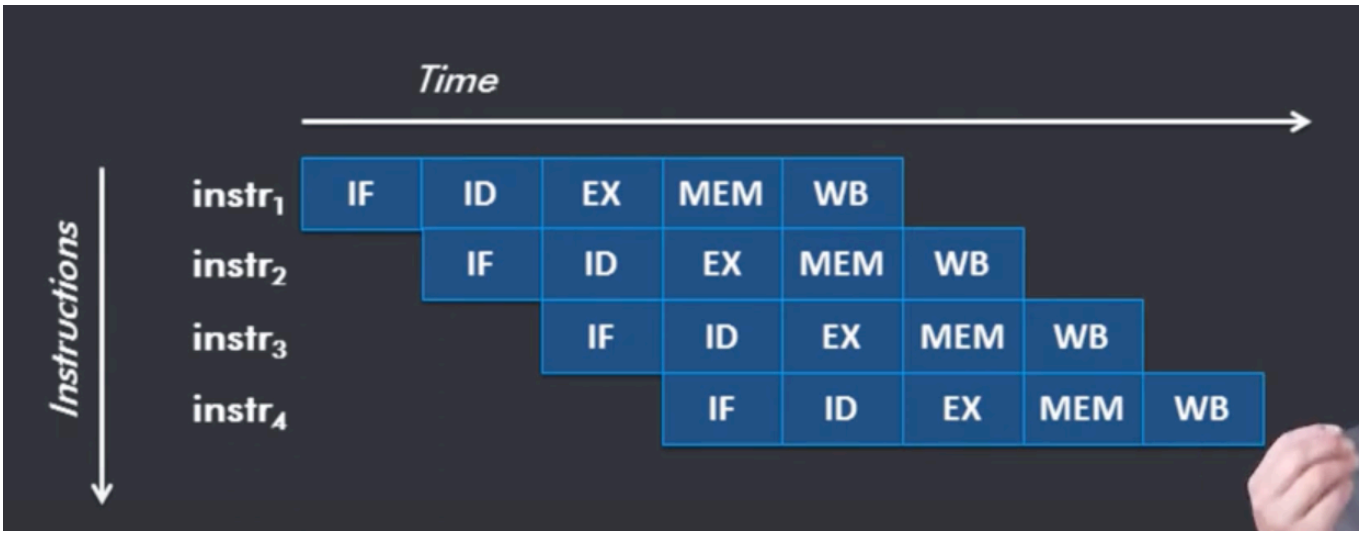
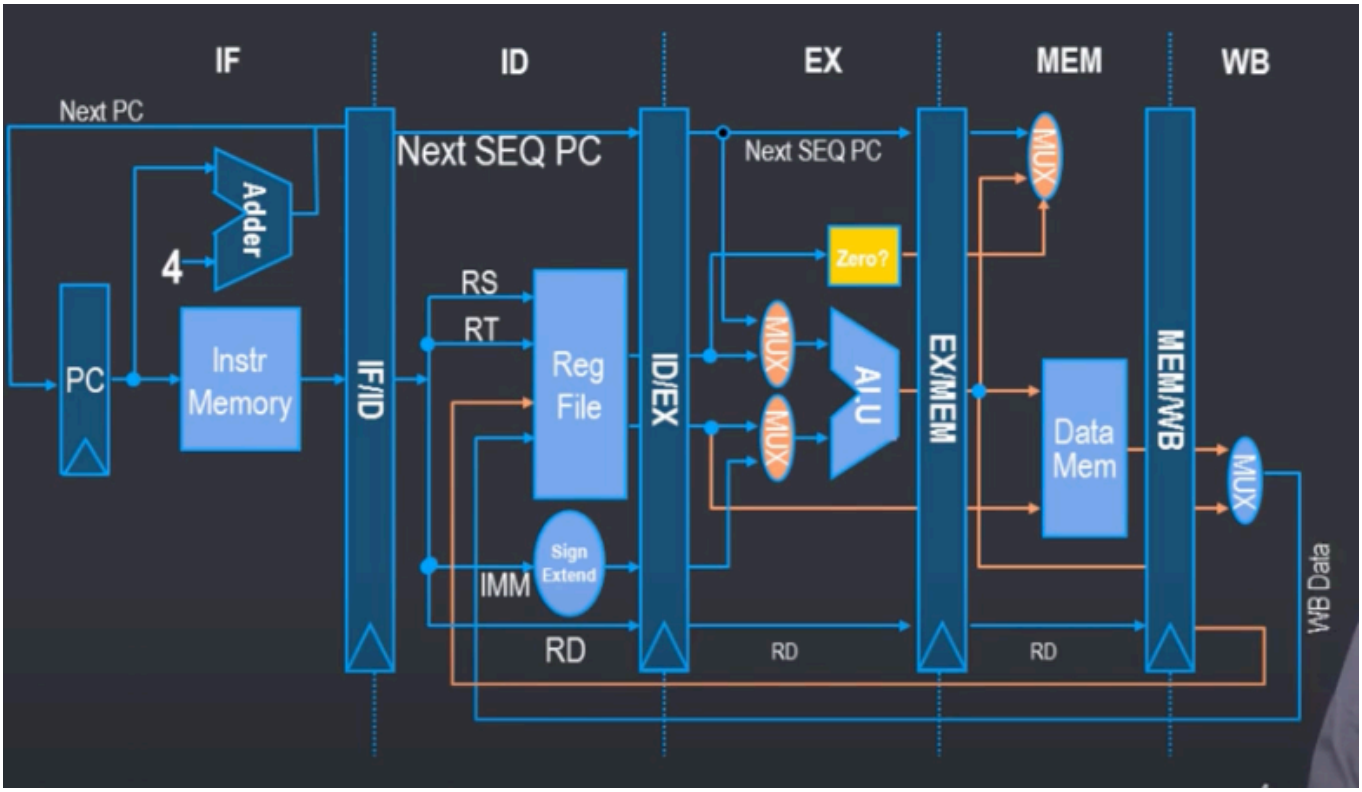


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Shows that pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction. Pipelining works best when all stages take same amount of time as time can increase if stages are not same time as wait time is needed



In first clock cycle, instruction 1 is fetched
In 2nd clock cycle, instruction 1 gets decoded, and instruction 2 is fetched at same time
in 3rd clock cycle, instruction 1 gets executed, 2 gets decoded, 3 gets fetched
in 4th clock cycle, instruction 1 access memory, 2 gets executed, 3 gets decoded, 4 gets fetched
in 5th clock cycle, instruction 1 writes back to register file, 2 accesses memory, 3 gets executed, 4 gets decoded, 5 gets fetched, so on



Datapath of pipelining - helps to understand the ILP (how you can have paralleism even in a single core processor)

Designing Instruction Sets for Pipelining

It is easier to pipeline MIPS ISA compared to other assembly languages because:

- All MIPS instructions are the same length (32 bits)
  - easier to fetch instructions in first pipeline stage and decode them in second stage
- MIPS only has few instruction formats - source register fields located in same place in each instruction
  - symmetry allows the second stage can begin reading register file at same time that the hardware is determining what type of instruction was fetch
- Memory operands only appear in loads or stores in MIPS
  - can use execute stage to calculate memory address and thjen access memory in following stage
- Operands are aligned in memory
  - single data transfer instructions can be transferred between processor and memory in a single pipeline stage

Pipeline Hazards

When next instruction in pipeline cannot execute in following clock cycle - they are known as **hazards**. There are 3 types of pipeline hazards: structural, data and control hazards. THese hazards can reduce performance of pipelined processors.

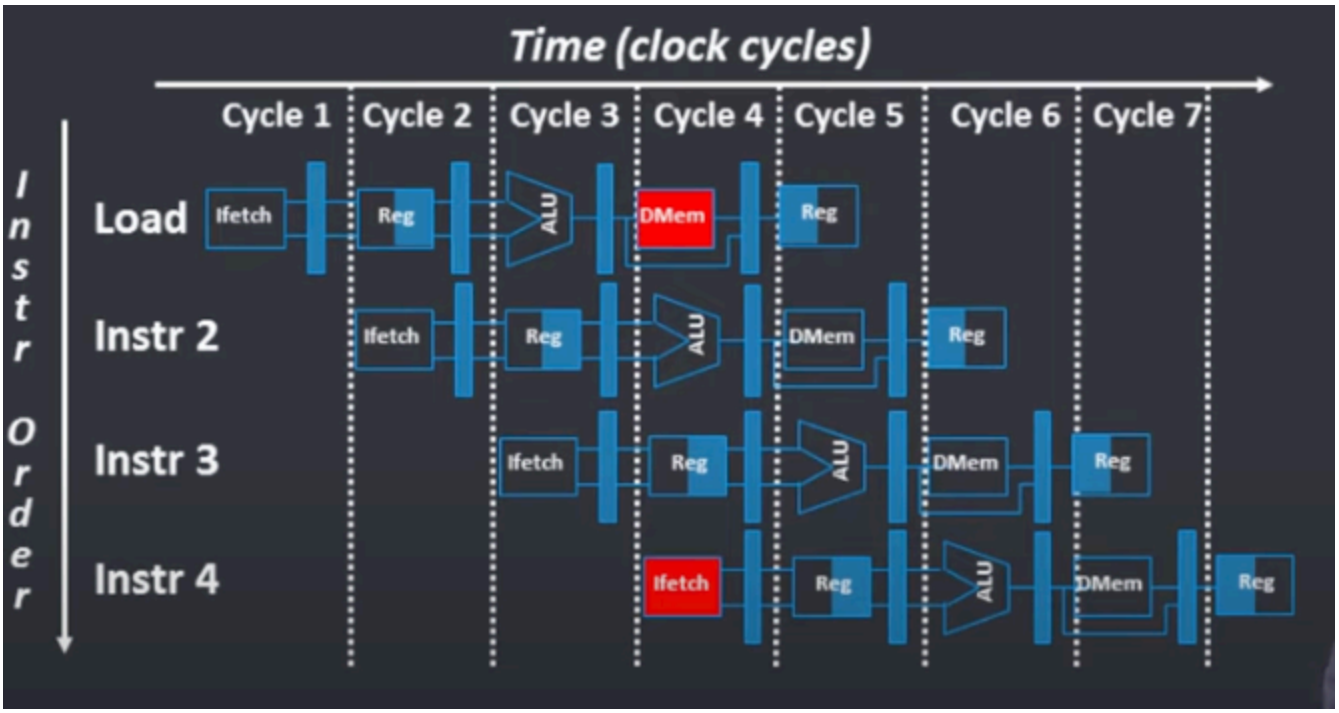
Example book: (https://www.youtube.com/watch?v=8yxrT1isnpE)

Structural Hazards

When a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that we want to execute in the same clock cycle

MIPS ISA designed to handle pipelines so is easy for designers to avoid structural hazards when designing pipelines

An example of structural hazards is if there is only a single memory, first instruction is accesing data from that memory and fiourth instruction is fetching an instruction from that memory. This raises a structual hazard as **both instructions are using the same hardware resource at the same time** (if there is only a single memory, both operations can't happen at the same time - stalling the pipeline as processor must delay one operation)



in this example, instruction 4 has to be stalled by 1 clock cycle as instruction 1/4 are accessing same memory at the same time

Data Hazards

Occurs when the pipeline must be stalled as **one step has to wait for another to complete**

They arise from the dependence of one instruction on an earlier instruction still in the pipeline, where instruction needs result of other instruction still in pipeline (in the example, laundry does not have any dependent relationships)

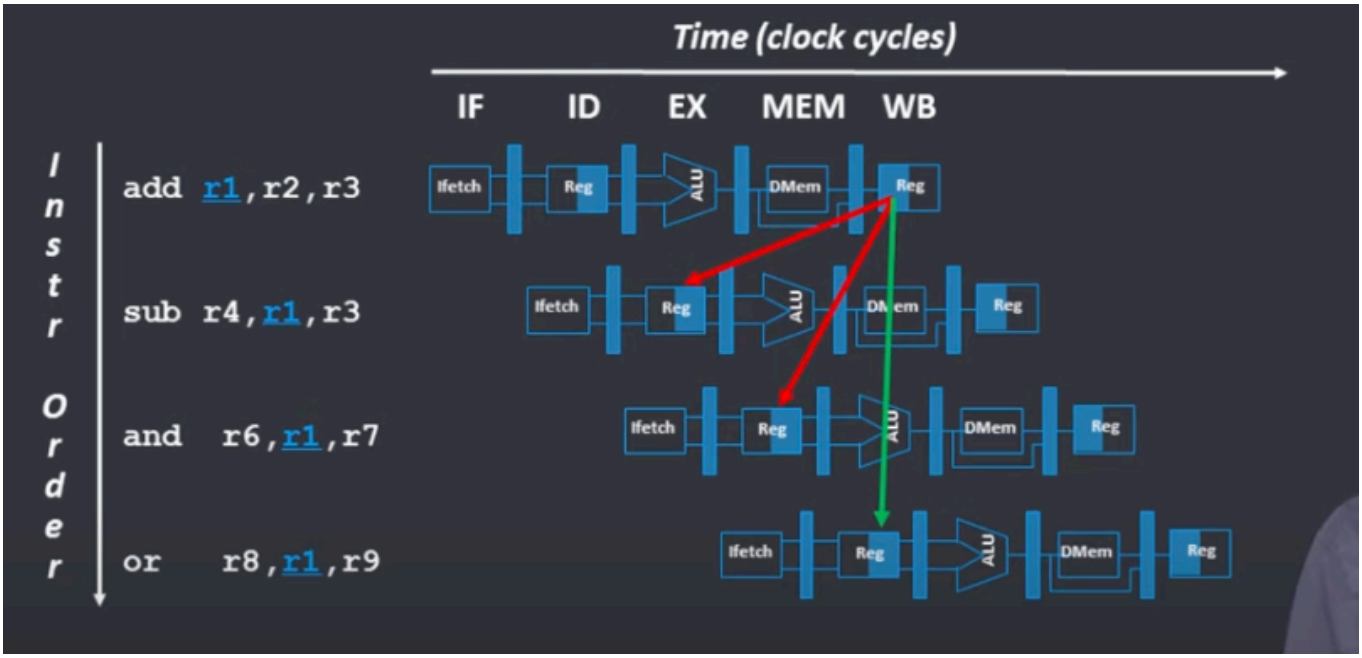
They can stall the pipeline (causing waste of clock cycles)

example:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

- add instruction doesn't write the result until the fifth stage
- sub instruction uses the result from the add instruction, i.e. $s0

diagram:



For every instruction, it goes through the process

- IF - instruction fetch
- ID - instruction decode
- EX - execution stage
- MEM - memory access (access data memory)
- WB - write back (register file gets written)

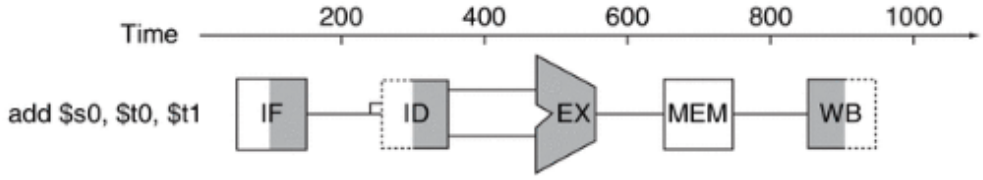


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

We don't need to wait for instruction to complete before trying to resolve the data hazard. Can just use the result after ALU computes the sum for the `add` instruction, for the input of the `sub` instruction.

Solution: **forwarding (bypassing)** - when you add extra hardware to retrieve missing item early from internal sources. you dont wait until result is written back to register file, but forward it to next stage immediately

Forwarding:
Diagram shows connection to forward the value in $s0 after executing the `add` instruction as input to execution stage of the `sub` instruction

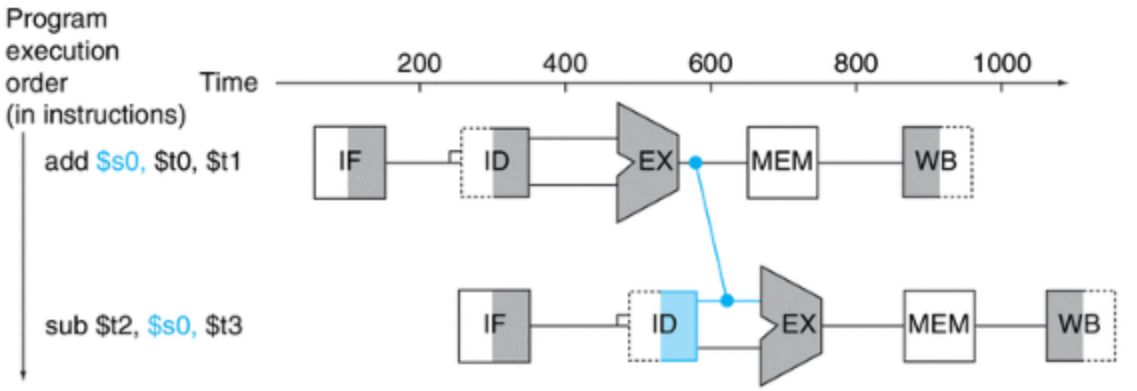


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

In this example, the value of $s0 is needed by the ALU for EX stage of the `sub` instruction

Forwarding paths are valid only if the destination stage is later in time than source stage - e.g. there cannot be a forwarding path from output of memory access stage in first instruction to input of execution stage of the following (go back in time)

Forwarding is effective but doesn't prevent all pipeline stalls e.g.

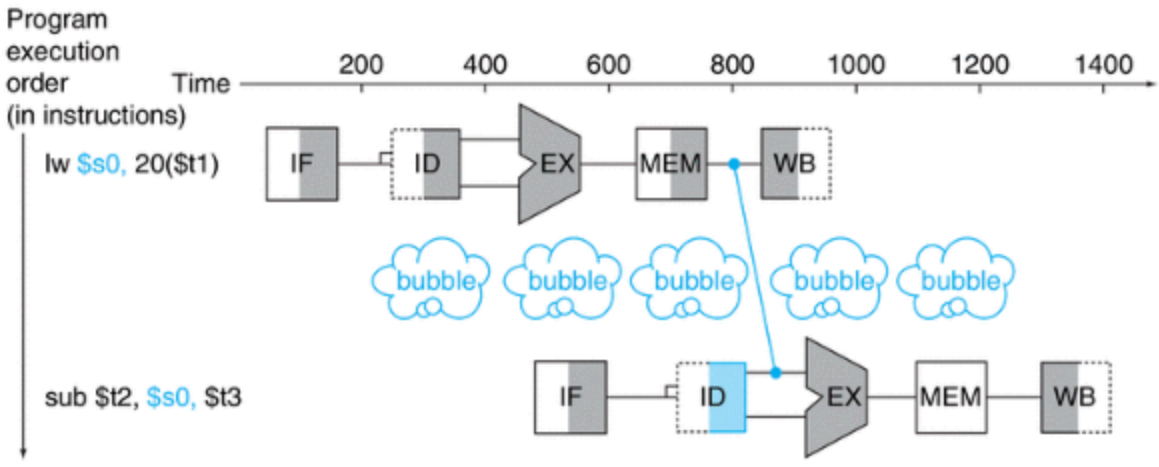
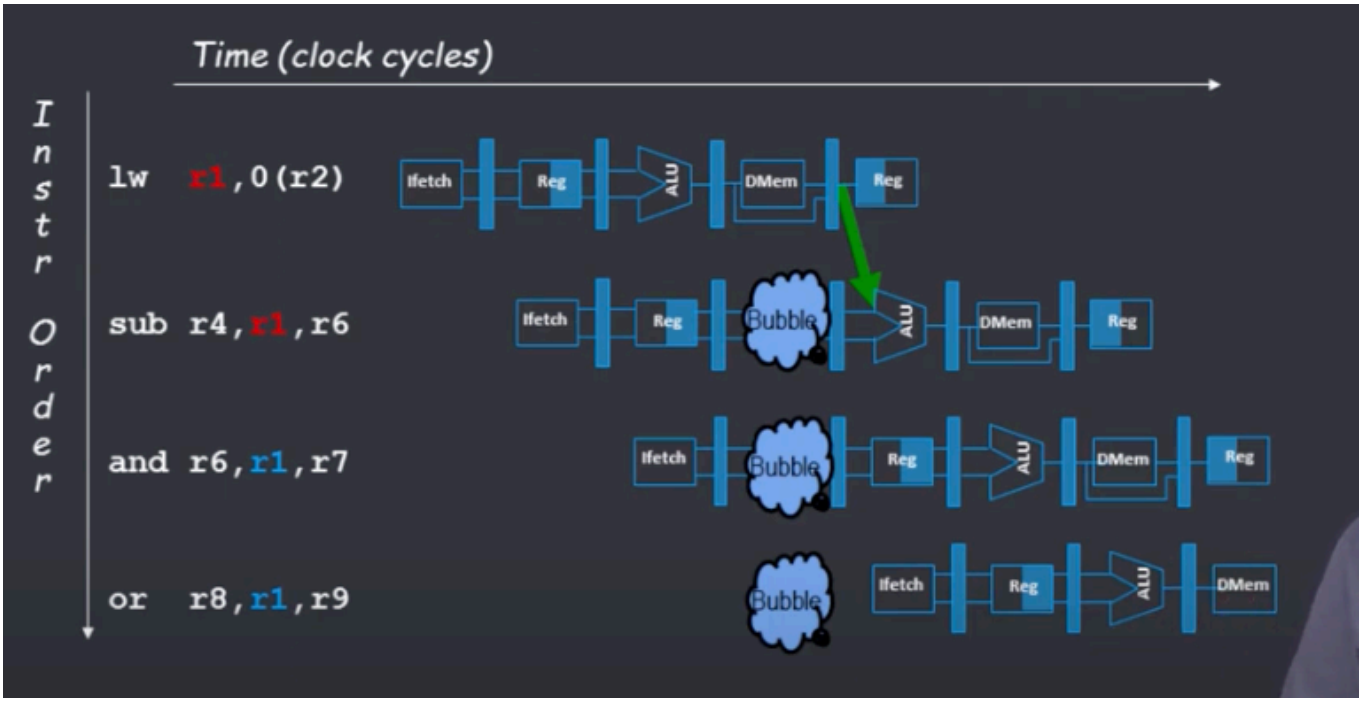


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard.

In this example, the word stored uin $s0 is only avaailable after the fourth stage (as loading to memory happens in 4th stage) of the first instruction in the depedencne, which is too late for input of third stage of sub. Hence stalling for one stage is needed for **load-use data hazard** (data hazard in which data being loaded by load instruction has not yet become available when it is needed by another instruction).

After stalling, forwarding can occur from MeM of `lw` instruction to the `EX` of the `sub` instruction



stalling occurs usually when there is a load instruction before an arithmetic instruction where the instruction depends on the load instruction as it uses the register loaded by that load instruction (because load instructions doesn't have loaded value ready until after memory access (MEM) stage)

In the diagram, the **bubble** is known as a **pipeline stall** (initiated to resolve a hazard)

Code can be reordered to avoid load-use pipeline stalls
for example:

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from `$t0`:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

hazards:

- both add instructions have a hazard as both are dependent on the immediate preceding `lw` instruciton
- load instructions do not have data ready until after MEM stage as add instruction requires both $t1, $t2 values during EX stage to perform the addition
- stall still happens even with forwarding
- storing word will not cause a hazard as it needs the value of $t5 after performing add operation (the value of $t5 can be obtained after forwarding from EX or MEM stage of add to MEM stage of store instruction)
- move the third `lw` instruction to be the first instruction would eliminate both hazards as bypassing eliminates several other potential hazards

resullt:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1,$t2
sw   $t3, 12($t0)
add  $t5, $t1,$t4
sw   $t5, 16($t0)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

However, forwarding is harder if there are multiple results to forward per instruction, or they need to write a result early on in instruction execution

Control Hazards

Also known as **branch hazard**

Happens when the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, thje flow of instruction addresses is not what the pipeline expected.

It arises from the need to make a decision based on the results of one instruction while others are executing - i.e. **branch instructions** in the computer (`beq` / `bne`)

Stall on branch:
Computer would not know what is the instruction after just receiving a branch instruction from memory - hence an immediate stall would be needed after fetching a branch, waiting until pipeline determines the outcome of the branch and knows what instruction address to fetch from

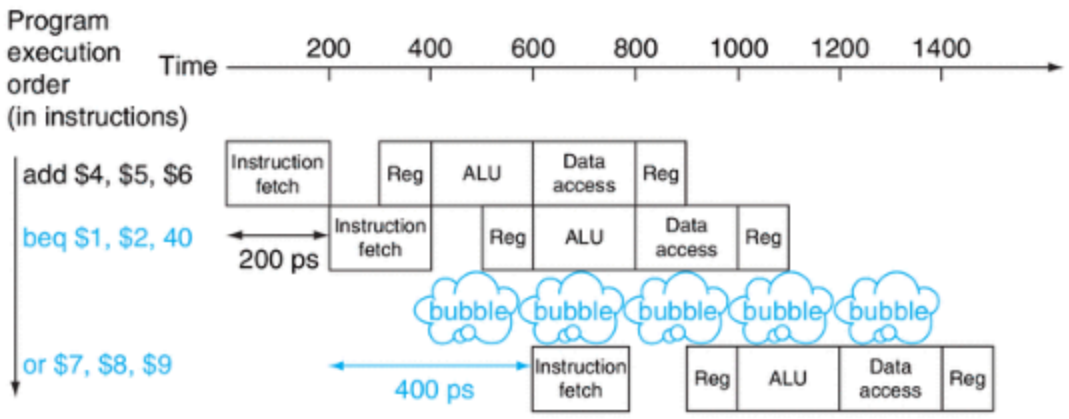


FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.8. The effect on performance, however, is the same as would occur if a bubble were inserted.

Computers use prediction to handle branches - method of resolving branch hazard that assumes for a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

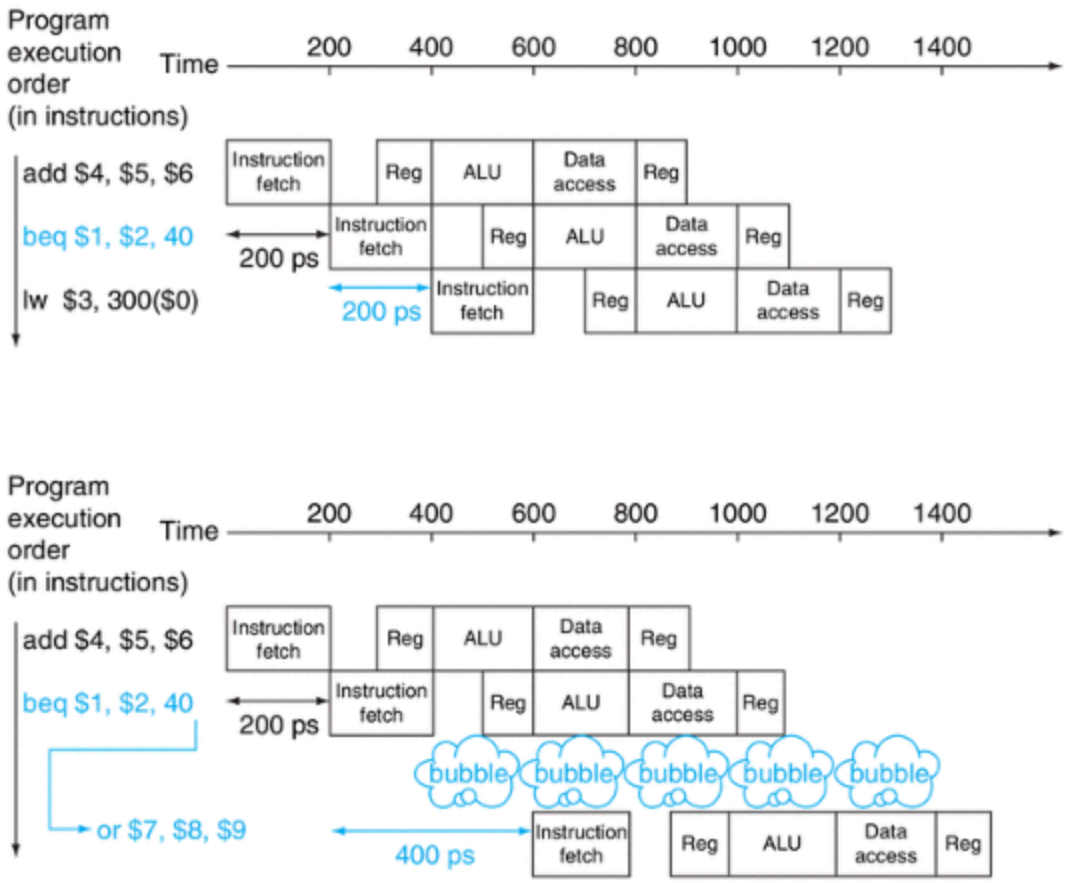


FIGURE 4.32 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.8 will reveal the details.

Chapter 5.1

Memory hierarchy - a structure describing computer memory that uses multiple levels of memory; as the distance from the processor increases, the size of the memories and access time both increase

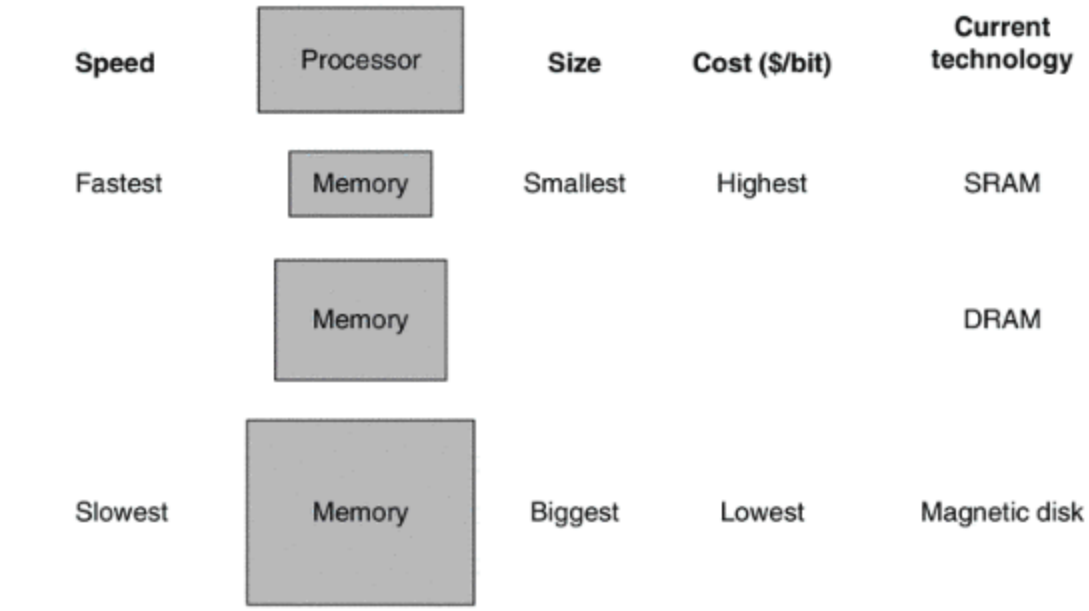


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many embedded devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 6.4.

AIm - present illusion to the user with as much memory as avaialble in cheapest technoplogy while prvoide memory access at speed offerred by fast memory

However data is copied between only 2 adjacent levels at a time

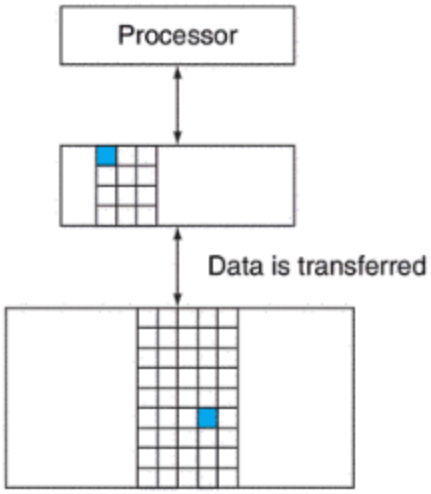


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

Big picture:

Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

Figure 5.3 shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. Thus, accesses that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

In most systems, the memory is a true hierarchy, meaning that data cannot be present in level i unless it is also present in level $i+1$.

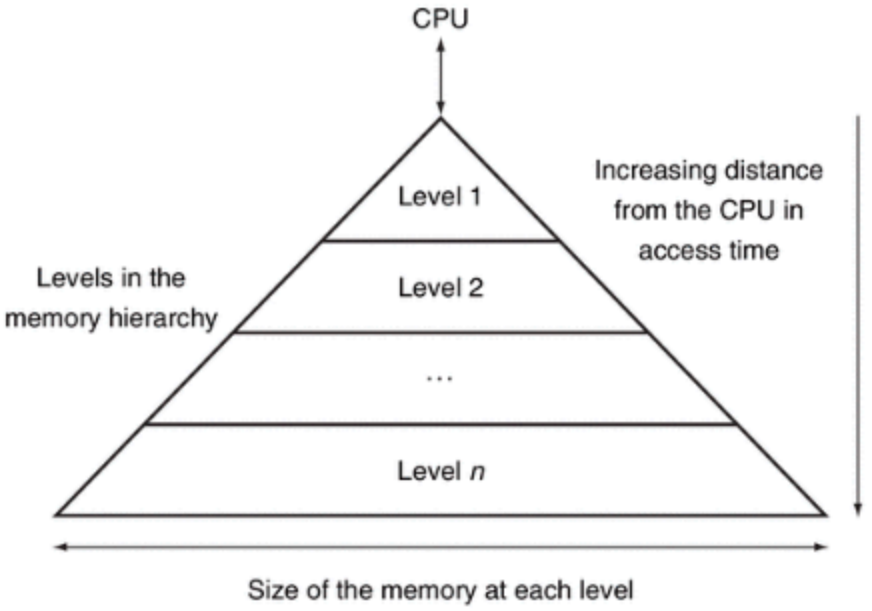


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level n. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

Chapter 5.2

Cache - memory storage in memory hierarchy that is between processor and main memory (locality of access)

- dictionary definition: a safe place for hiding / storing things

A cache is used to bridge the speed gap between the fast CPU and slower main memory (RAM). It stores frequently accessed data and instructions in a small, high-speed memory located closer to the CPU.

The main purposes of a cache include:

1. **Reducing memory access time**: By keeping copies of frequently used data closer to the processor, the CPU can access information much faster than retrieving it from main memory.
2. **Exploiting locality**: Caches take advantage of temporal locality (recently accessed data is likely to be accessed again soon) and spatial locality (data near recently accessed locations is likely to be accessed soon).
3. **Improving system performance**: By reducing the average memory access time, caches significantly improve overall system performance and throughput.
4. **Reducing memory bandwidth requirements**: Since many memory accesses are satisfied by the cache, the demand on the main memory system is reduced.

Modern computers typically use a hierarchy of caches (L1, L2, L3) with different sizes and speeds to maximize both speed and capacity while minimizing cost.

Formula to find blocks in cache:
`(Block address) % (number of blocks in the cache)`

Caches map different memory locations to cache locations (**direct-mapped cache**) where each memory location (word) is mapped to exactly one location in the cache and there is a separate tag for each entry

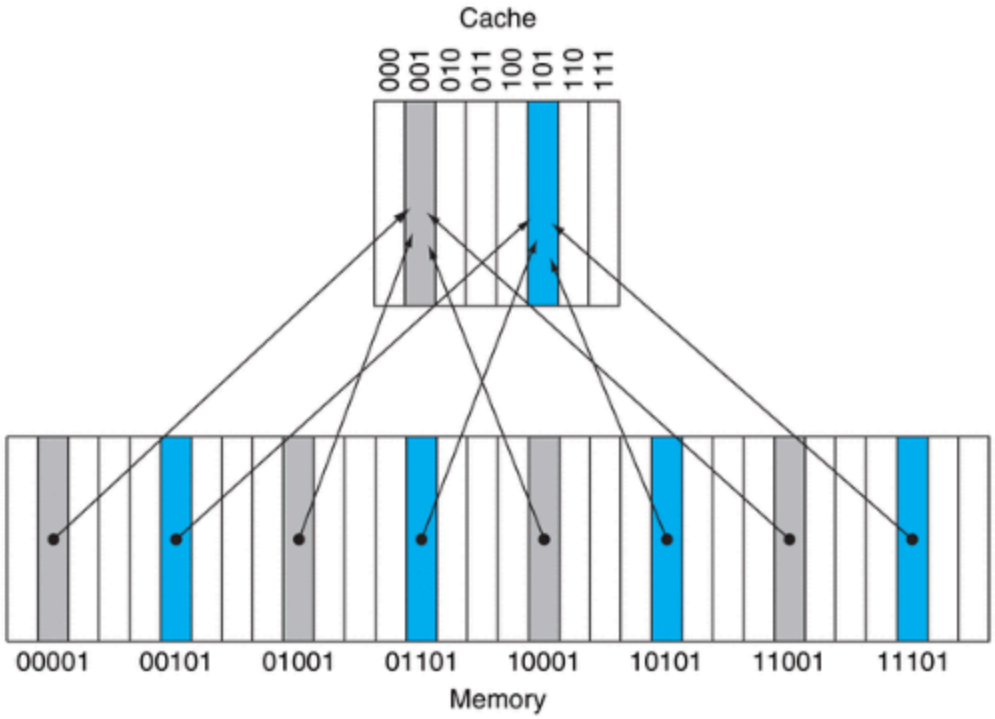


FIGURE 5.5 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

Accessing a Cache

As there are 8 blocks in the acche, the low-order 3 bits of an address give the block number

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (7.6b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (7.6c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (7.6d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (7.6e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (7.6f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

hits or **miss** refer to if the address reference is found in the cache. if address is deemed a miss (not found in cache), it will write the tag/data of the address in cache, otherwise it will direclty use the data in cache (hit) as data is already in cache

reference address is divided to:

- tag field - used to compare with value of tag field of cache

- cache index - used to select the block
  both are used to specify the memory address of the word contained in cache block - least 2 significant bits isgnored when selecting word in block as words are aligned to multiples of 4 in MIPS

Handling miss of address in cache:

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.6 The cache contents are shown after each reference request that *misses*, with the index and tag fields shown in binary for the sequence of addresses on page 461. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field j and the index i. For example, in cache f above, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}.

Total cache size (number of bits in direct-mapped cache):

$$2^n \times (\text{block size} + \text{tag size} + \text{valid field size}).$$

Handling cache miss

Happens when request for data from cache that cannot be filled because the **data is not**

present in the cache. Control unit must detect a miss and process the miss by fetching the requested data from memory. If cache reports a hit, iit will continue to use the data as if nothing happened.

Steps:

1. Send original PC value (current PC - 4) to memory
2. Instruct main memory to performa. read and wait for memory to complete its acces
3. Write the cache entry, putting the data from memory in data portion of entry, writtinfg upper bits of address (from ALU) into tag field, and turn valid bit on
4. Restart instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.
   On a miss we stall the processor until memory responds with data when controlling cache on a data acesss. (identical to handle cache miss)

Handling writes to cache
You write to cache when there is a cache miss (data requested not in cache)

Methods to handle writes to both cache and memory, ensuring consistency between memory/cache:

- **Write-through** - always write data into both next lower-level memory and cache to keep both consistent. Howeover performance would be slow as every write causes data to be written to main memory (solution would be to use a write buffer)
- **Write buffer** - Queue that stores data while it is waiting to be written to memory
- **Write back** - alternative to write-through scheme - it handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced

Designing memory system to handle caches

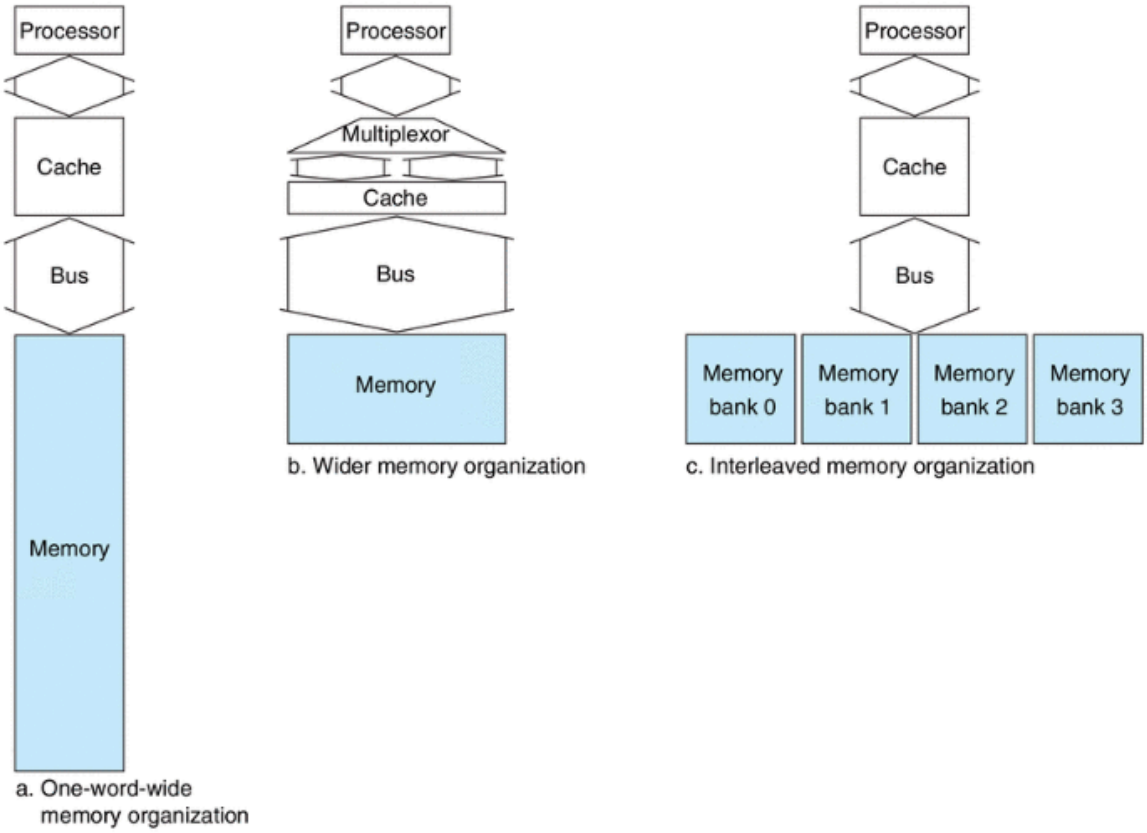


FIGURE 5.11 The primary method of achieving higher memory bandwidth is to increase the physical or logical width of the memory system. In this figure, memory bandwidth is improved two ways. The simplest design, (a), uses a memory where all components are one word wide; (b) shows a wider memory, bus, and cache; while (c) shows a narrow bus and cache with an interleaved memory. In (b), the logic between the cache and processor consists of a multiplexor used on reads and control logic to update the appropriate words of the cache on writes.

Interleaving - retains advantage of incurring the full memory latency only once (memory banks all read simultaneously).

Summary

We began the previous section by examining the simplest of caches: a direct-mapped cache with a one-word block. In such a cache, both hits and misses are simple, since a word can go in exactly one location and there is a separate tag for every word. To keep the cache and memory consistent, a write-through scheme can be used, so that every write into the cache also causes memory to be updated. The alternative to write-through is a write-back scheme that copies a block back to memory when it is replaced; we'll discuss this scheme further in upcoming sections.

To take advantage of spatial locality, a cache must have a block size larger than one word. The use of a larger block decreases the miss rate and improves the efficiency of the cache by reducing the amount of tag storage relative to the amount of data storage in the cache. Although a larger block size decreases the miss rate, it can also increase the miss penalty. If the miss penalty increased linearly with the block size, larger blocks could easily lead to lower performance.

To avoid performance loss, the bandwidth of main memory is increased to transfer cache blocks more efficiently. Common methods for increasing bandwidth external to the DRAM are making the memory wider and interleaving. DRAM designers have steadily improved the interface between the processor and memory to increase the bandwidth of burst mode transfers to reduce the cost of larger cache block sizes.

Chapter 5.3

Cache misses primarily contribute to memory-stall clock cycles which then contributes to CPU time, where memory-stall clock cycles is the sum of read-stall cycles and write-stall cycles. Number of memory-stall cycles depend on both miss rate and miss penatlty

Methods on improving cache performance

- **Associativity**
  - used to **reduce miss rates** (fraction of memory accesses not found in a level of memory hierarchy)
  - REduces the probability that two different memory blocks will contend for the same cache location
  - Allow more flexible placement of blocks within the cache
  - Fully associative cache - cache structure in which a block can be placed in any location in the cache, but every block in the cache has to be searched to satisfy a request
    - higher costs -> impractical to implement
  - Set-associative cache - cache that has fixed number of locations (at least 2) where each block can be placed
    - higher miss rates but faster access times

- all the tags of all the elements of the set must be searched as the block may be placed in any element of the set

FIGURE 5.13 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

- Increasing degree of associativity => Decreases miss rate, but could increase the hit time

- **Use of multilevel cache hierarchies**
  - used to **reduce miss penalties** (time required to fetch a block into a level of memory hierarchy from the lower level,, including the time to access the block, transmit it from one level to the other and inserting it in the level that experienced the miss, and then pass the block to the requestor)
  - Done by adding an additional level to the hierarchy, resulting in having multiple levels of caches rather than just the cache and main memory
  - Works by allowing a larger secondary cache to handle misses to the primary cache where the secondary cache can be 10 or more times larger than primary cache.
  - Miss penatly for first-level cache is same as access time of second-level cache (which is much lesser than the access time of main memory)
  - Tradeoff between size of secondary cache and access time depends on number of aspects of implementation